THE UNIVERSITY OF MICHIGAN

COMPUTING RESEARCH LABORATORY[1]

DEVELOPMENT AND ANALYSIS OF A GLOBAL COMPACTION TECHNIQUE FOR MICROPROGRAMS

Jehkwan Lah

CRL-TR-40-84

September 1984

Room 1079, East Engineering Building
Ann Arbor, Michigan 48109
USA
Tel: (313) 763-8000

[1]Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the funding agency.

DEVELOPMENT AND ANALYSIS OF A GLOBAL COMPACTION TECHNIQUE FOR MICROPROGRAMS

by
Jehkwan Lah

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Electrical Engineering)
in The University of Michigan
1984

Doctoral Committee:

Professor Daniel E. Atkins, Chairman
Professor Gideon Frieder
Associate Professor Trevor Mudge
Professor Norman Scott
Associate Professor Toby Teorey

ABSTRACT

DEVELOPMENT AND ANALYSIS OF A GLOBAL COMPACTION TECHNIQUE FOR MICROPROGRAMS

by

Jehkwan Lah

Chairman: Daniel E. Atkins

The need for a better microprogramming tool has increased considerably as increased demand and support of computer technology has brought about wide use of microprograms. The eventual goal of microprogramming tool development would be to make a high level microprogram language and a compiler to generate minimal-execution-time microcode for a variety of machines. In generating minimal-execution-time microcode, one aspect that differentiates microprogramming languages from macroprogramming languages is the need for compaction in highly horizontal microarchitecture.

Among the proposed microprogram compaction methods, the trace scheduling is the most general and appears to give the fastest execution of compacted microcode. However, the growth of memory size by extensive copying of blocks can be enormous, exponential in the worst case, and the complicated bookkeeping stage of the trace scheduling has been an obstacle to implementation.

A technique called beta compaction, based on trace scheduling, is proposed to mitigate the drawbacks of trace scheduling. Basically, it identifies the junction blocks (the blocks beginning with a join and ending with a conditional branch) as the major source of complication, and cut traces at those junction blocks. It achieves almost all the compaction of the trace scheduling except that which causes copying of blocks. Memory size after the beta compaction is usually smaller than the original. Even when the memory size grows in rare instances, it

is bounded by $O(n^2)$ in the worst case. And the bookkeeping stage is very much simplified. The compacted microcode size variation as the source microcode changes is also very small.

A loop-free version of both beta compaction and trace scheduling has been implemented. Comparison between the two was done using artificially synthesized microcodes and the above properties of the beta compaction was confirmed.

A simple microprogrammable machine based on AM2900 components was designed and simulated with an interactive user-friendly interface. A realistic application program was written and hand-compiled into microcode. The microcode was executed on the simulator both before and after compaction, which demonstrated the applicability of the compaction technique and the correctness of the implementation.

TABLE OF CONTENTS

LIST OF FIGURES

Figure

LIST OF APPENDICES

Appendix

LIST OF ABBREVIATIONS

MO	Micro operation
MI	Micro instruction
DDG	Data dependency graph
SLM	Straight line microcode
CI	Complete instruction
DRS	Date ready set
EP	Early partition
LP	Late partition
CP	Critical partition

CHAPTER I

INTRODUCTION

Since Maurice Wilkes first introduced the concept of microprogramming in 1951 [WIL51], the use of the microprogram has increased continuously as the demand for and support of computer technology have increased. Increasing complexity of digital systems requires more complex microprograms. Decreasing high speed memory cost and emerging development tools help meet the requirement.

However, until recently microcode was produced with little help of development tools. The only practical tool available to many microprogrammers was an assembly-like bit stuffer using mnemonics. The microcode was written by someone who had a thorough knowledge of the machine to be programmed. The microprogramming was considered a part of the hardware design and so it was often done by the hardware designer. And once developed, the microcode was not touched except to fix bugs. Those microcode developments were manageable by a few specialists with little help from development tools, mainly because the microcode was relatively small in size and less complex.

However, things have changed. With the introduction of writable control store and user microprogramming, the need for better microprogramming tools is higher. And increasingly complex digital systems and the advent of VLSI are increasing the use of microprograms. Design time for VLSI systems depends on sophisticated design aids for hardware and microprogramming. VLSI hardware and microcode design problems are large and technology-dependent; humans alone cannot handle the detail and explore all the alternatives involved in correct

optimal design [PAR81]. VLSI technology has also reduced the cost of high-speed memory available, so the size of micromemory has become less of a constraint and the microprogramming of some special purpose processors involves larger microcodes than conventional macro instruction emulation. All these changes are making it more critical to have better tools in microprogramming.

The eventual goal of microprogramming tool development would be to make a high level language and a compiler to generate minimal-execution-time microcode for a wide variety of machines. There have been various attempts to design and implement higher level languages for microprogramming but none of these has resulted in the production of a generally available compiler. Several proposed high level microprogramming languages are reviewed in [SIN80]. Even though the same pressure that has led to the widespread acceptance of conventional high level languages now applies to microprogramming [DAV78], the development of microprogramming languages lags far behind that of macroprogramming languages. There are a couple of factors which complicate compilation to microcode.

First, the structure of horizontal[1] microcode is much more complicated than that of conventional machine code. In horizontal microarchitecture, a microinstruction contains several microoperations which directly control parallel hardware resources. The microoperations contained in one microinstruction are executed at the same time. Accordingly, the detection of available parallelism and scheduling microoperations are very difficult. Timing can be very complicated. Certain microoperations require different clock phases than others or might take more than one clock cycle. If some degree of (if not total) machine independence is sought, generating equally fast-executing microcode for different machines is an extremely different task.

Second, in most microprogramming environments, minimal-execution-time microcode is very important. In particular, when the microprogram is used to

[1] See the definition in chapter II.

emulate macroarchitecture, the efficiency of the whole systems ultimately depends on the efficiency of the microprograms implementing the instruction set of the macromachine. The user microprogram (of non-emulation application) may allow less strict efficiency requirements. But the only reason that a user would want to write microcode is to gain speed. If the compiler cannot generate sufficiently fast-executing microcode, it is of no use.

One of the critical issues in developing a high level microprogram language is how to generate minimal-execution-time microcode. As machines have more concurrency available in datapath at the register transfer level, it is particularly important to efficiently control all the available parallel resources. It is a common belief that microcode generated by a machine (or a compiler) cannot be executed faster than one written by a highly skilled microprogrammer, but as the size of microprograms grows, it becomes beyond human intelligence to handle all opportunities to optimize a microprogram [PAT76].

In generating minimal-execution-time microcode, one aspect that differentiates *micro*programming languages from *macro*programming languages is the need for compaction in highly horizontal microarchitecture. Microprogram compaction is the process of exploiting parallelism to combine microoperations (MO's) into microinstructions (MI's) to reduce the time and/or space needed for the execution of a microprogram. In solving microprogram compaction problems, optimality is pursued but not necessarily arrived at, since the microprogram optimization (optimal compaction) problem has been proved to be NP-complete both in microword dimension [DEW76] and in bit dimension [ROB79].

The microprogram compaction problem has been approached in two ways: local compaction and global compaction. Local compaction deals only with a straight line microcode (SLM) section also known as a basic block. A SLM is a sequence of MO's with no jump into the microcode except at the beginning, and no jump out except possibly at the end. Global compaction deals with microprograms which have more than one basic block with conditional jumps, joins, and

loops.

Several methods have been proposed to solve the local compaction problem, and they are reviewed in an article by Landskov et al [LAN80]. The same group of people ran some experiments with the methods and reported the results in [DAV81] and claimed that the local compaction problem is considered to be essentially solved. However, Vegdahl presented an algorithm in his thesis [VEG83] which showed that the classical compaction problem, which does not consider reordering of source MO's, can be solved in $O(n^v)$, where v is a bounded number of registers in the machine. And so, Vegdahl conjectures, the problem of finding a semantics-preserving partial ordering which yields optimum compaction is a more difficult problem, because the general compaction problem - that is, the problem that considers all semantics-preserving partial orderings - is still NP-complete even with the bounded-register assumption. Here we are dealing with the classical compaction problem and do not consider reordering of source MO's. Even though the problem can be solved in polynomial time, the order which is equal to the number of registers in the hardware is, in most cases, so high that one is justified in using heuristics, which has low order (usually $O(n^2)$) polynomial time complexity. Several methods to solve the local compaction problem, including Vegdahl's thesis results, are reviewed in section 3.1.

Four methods have been proposed to solve the global compaction problem [DAS79, WOO79a, TOK78, TOK81, FIS79, FIS81a] as summarized in [FIS81b]. Also, a recent paper by Isoda [ISO83] has proposed another approach, which uses a generalized data dependency graph.

The trace scheduling by Fisher [FIS79, FIS81a] is the most general method and appears to give the fastest execution of compacted microcode. Although Fisher's trace scheduling procedure for global compaction may produce significant reduction in execution time of compacted microcode, the growth of memory size by extensive copying of blocks can be enormous. In the worst case, the memory

size can grow exponentially[2] [FIS81a], and the complex bookkeeping stage of the trace scheduling is an obstacle to implementation.

In [FIS81a], Fisher suggests modifications to mitigate the memory growth effect. These modifications are based upon selection of a probability threshold below which (with some exceptions) MO's are not allowed to move across block boundaries. Both memory size and execution time appear to be very sensitive to changes in the source microcode. See section 3.2.3 for further discussion of the modifications. A few methods to solve the global compaction problem are reviewed in section 3.2.

In this dissertation a technique called beta compaction, based on trace scheduling, is proposed to mitigate the drawbacks of trace scheduling. Basically, it identifies the junction blocks (the blocks beginning with a join and ending with a conditional branch) as the major source of complication, divides traces at those junction blocks, and compacts each divided trace separately. It achieves almost all the compaction of the Fisher's trace scheduling except that which causes copying of blocks. An earlier effort to correct the drawbacks of trace scheduling called tree compaction is reported in [LAH83].

A loop-free version of both beta compaction and trace scheduling has been implemented. Comparison between the two was done using some synthesized microcodes and it was confirmed that the beta compaction generates almost as fast-executing microcode as the trace scheduling but with much less memory.

A microprogrammable machine based on AMD2900 components was designed and simulated with an interactive interface. A realistic application programs was written and hand-compiled into microcode. The microcode was executed on the simulator both before and after compaction, to demonstrate the applicability of the compaction technique and the informal correctness of the implementation.

---

[2] One such example is shown in section 5.6.1.

Chapter II presents a simple microprogram model and definitions of the terms which are used throughout the thesis. Chapter III reviews some work done previously in the area of microprogram compaction including local compaction and global compaction. Chapter IV and Chapter V are the core of this thesis. Chapter IV presents a detailed description of the beta compaction which are developed to solve the global compaction problem. It also gives one illustrative example to show the actual working of the heuristic. Chapter V reports two experimental results. First, the comparison of beta compaction against the trace scheduling using an artificial microcode model is presented. Second, compaction of an application program and its execution on a simulated machine is described. At the end of the chapter, exponential memory growth of the trace scheduling and the worst-case space complexity analysis of the beta compaction are shown. Chapter VI summarizes the dissertation and suggests some future research.

Throughout the dissertation a few abbreviations of terms are used to make sentences short and clear. The abbreviations are listed at the beginning of this document.

CHAPTER II

A MODEL OF A MICROPROGRAM

The purpose of this chapter is to present a microprogram model which will be used in later chapters of the thesis. A part of the model is based on D. Landskov et al [LAN80]. The model is to satisfy the minimum requirement of allowing presentation of the essential concepts of the microprogram compaction heuristics with clarity. Since this is the basic model, many extensions are possible. Some of them will be discussed along with the basic model.

Definition: A *microinstruction* (MI) is an ordered set of all control signals in a machine at a given (quantized) time.

Definition: A *microprogram* is a time-ordered sequence of MI.

Definition: Each separate machine activity specified in an MI is called a *microoperation* (MO).

Thus an MI can be characterized as a set of MO's and the MO is the basic unit of operation that we are dealing with. In some literature [MAL78,LAN80], microbundle (MB) is defined as a set of MO's, all of which are coupled to one another. Thus every MO in an MB must go into the same MI. The concept of bundling can be useful in dealing with MO's which operate on transitory data storage.

The most common classification applied to microinstruction formats is to describe them as horizontal or vertical [SAL76]. The horizontal and the vertical are relative terms used to indicate the degree of encoding in microinstruction

formats. There are two extremes, totally horizontal and totally vertical, and many variations in between. The highly horizontal microinstruction format would have a sufficient number of separate control fields to exercise simultaneous control over all independent hardware facilities, with encoding limited to mutually exclusive control signals. The highly vertical microinstruction format, on the other hand, would have a high degree of encoding and, in many ways, resembles conventional machine instruction. We will use the terms horizontal and vertical to indicate highly horizontal and highly vertical, which include not only extremes but some neighborhood of the extremes.

Definition: Microprogram *compaction* is the process of exploiting parallelism to combine MO's into MI's to reduce the time and/or space needed for the execution of a microprogram.

The vertical MI's, in a strict sense, specify only a single MO to be performed in each MI. Therefore vertical microprogramming allows no room for microprogram compaction. Horizontal microprogramming is implied throughout the discussion of microprogram compaction.

Definition: A *straight line microcode* (SLM) is an ordered set of MO's with no entry point, except at the beginning, and no branches, except possibly at the end.

Microprogram compaction which deals only with a single SLM at a time is called local compaction. Microprogram compaction which deals with more than one SLM, where SLM's are connected with each other through conditional branches, joins or loops, is called global compaction. In local compaction, the execution time of compacted microcode is reduced by reducing the number of compacted MI's. However, in global compaction, simply reducing the total number of compacted MI's does not necessarily reduce the execution time of the compacted microcode because the probabilities of SLM's being executed are different. It is possible, in global compaction, that a microcode with larger size is executed faster than a microcode with smaller size for some data set.

2.1. Data Dependency Analysis

In the process of microcode compaction, a given sequence of MO's are placed into a sequence of MI's where an MI may contain one or more MO's. Although the original order of MO's is invariably changed, we want the original semantics of the microcode retained and the data integrity not violated after the compaction. To accomplish these we need to analyze data dependency between MO's.

The following definitions about data dependency are made only with respect to a SLM or a trace. A trace is a sequence of blocks and MO's in the blocks are treated as if they form a SLM for the purpose of data dependency analysis. The trace will be defined in sec. 2.3. The definitions of data dependency can be extended to include any program structure [ISO83].

Definition: In a given sequence of MO's mo_1, mo_2, ..., mo_i, ..., mo_j, ..., mo_t, we say mo_i and mo_j have a *data interaction* if they satisfy any of the following conditions.

(1) An output resource of mo_i is also an input resource of mo_j.

(2) An input resource of mo_i is also an output resource of mo_j.

(3) An output resource of mo_i is also an output resource of mo_j.

To maintain data integrity, the order of any two MO's cannot be changed if they have a data interaction. Using the definition of data interaction, we can define a partial ordering over the MO's.

Definition: Given two MO's mo_i and mo_j, where mo_i precedes mo_j in the original SLM, mo_j is *directly data dependent* on mo_i (written $mo_i \rightarrow mo_j$) if the two MO's have a data interaction and if there is no sequence of MO's, mo_{k1}, mo_{k2}, ..., mo_{kn}, $n \geq 1$, such that $mo_i \rightarrow mo_{k1}$, $mo_{k1} \rightarrow mo_{k2}$, ..., $mo_{k(n-1)} \rightarrow mo_{kn}$, $mo_{kn} \rightarrow mo_j$.

The second part of the definition ensures that two directly data dependent MO's will have no other chain of directly data dependent MO's between them.

Data dependency is the transitive closure of the direct data dependency relation.

Definition: The partial ordering over MO's defined by direct data dependency can be represented by a directed acyclic graph. We call this the *data dependency graph* (DDG). Each node on a DDG corresponds to a MO. An edge on a DDG from a node i corresponding to mo_i to a node j corresponding to mo_j, indicates that mo_j is directly data dependent on mo_i.

Vegdahl adopted the definition of data antidependency from Banerjee et al. [BAN79] and used that as a basis for his analysis.

Definition: If MO may destroy data that is required by another MO, the former is said to be *data antidependent* on the latter.

This is the case (2) of our definition of data interaction. For further discussion of data antidependency and Vegdahl's analysis, see section 3.1.5.

2.2. Host Machine Description

Besides data dependency among MO's, there is another restriction that has to be considered in microprogram compaction which is imposed by the host machine itself. If two MO's require exclusive use of a single resource, they cannot be placed into the same MI. The situation is called a resource conflict. To handle the resource conflict, we need a model to describe the host machine resources seen by the microprogram. Even though two MO's require separate resources, they may not necessarily be placed in the same MI because, as a result of partial encoding, the two MO's use the same MI field. The situation is called a MI field conflict. In the case of totally horizontal microinstruction, the MI field conflict does not exist because each resource in the hardware has its own control field in the MI. Therefore, the resource conflict and the MI field conflict are equivalent in totally horizontal microcodes. To simplify the discussion, we will assume a

totally horizontal MI format without losing generality. It would then be straightforward to extend our discussion to handle partially encoded microcodes. We now present a model of a machine control word which includes a simple description of the host machine resources.

Definition: An MO is represented by a six-tuple (label, instruction, next-address, destination-registers, source-registers, resource vector). The *label* is a number to identify each MO, the *instruction* is a command to the microsequencer (microaddress controller), for example, CONT, CJMP, GOTO, etc., and the *next-address* is a target address for a branch. The *destination-registers* and the *source-registers* are the registers written and read respectively in a given MO. A *resource vector* is a vector in which each component corresponds to an individual resource available in the hardware. The value of each component is either 1 or 0 depending on whether the corresponding resource is used in a given MO.

The resource vector gives a necessary description of the host machine resources to detect and resolve the resource conflicts.

Other models used in the literature [LAN80, DAV81, TOK81], which were intended for local compaction, do not have sequencer instructions and branch addresses. On the other hand, they have a set of clock phase required for MO execution and a set of all MI fields required by the MO. It would be straightforward to add these attributes to our model as necessary.

Definition: A register is *alive* at some point in a program if its contents will be read in the future prior to being overwritten. Otherwise, the register is *dead.*

2.3. Microprogram Description

A loop-free program can be represented as a connected directed graph with one node of in-degree zero (entrance) and one node of out-degree zero (exit). For programs with multiple entry points, a dummy entry block from which

execution is branched to the actual entry points, is added to comply to the above rule. Similarly, a dummy exit block is added for programs with multiple exit points.

Definition: A *trace* is a set of blocks which forms a path of execution and has higher probability of execution than any other path in a given microprogram or a part of it.

Definition: A *weighted execution time* of a microprogram is the sum of execution time of each block multiplied by the probability of that block executing.

CHAPTER III

PREVIOUS RESEARCH

Previous research attempts to solve the microprogram compaction problem has been approached in two different ways: local compaction and global compaction. Earlier attempts concentrated on what is now known as local compaction, which schedules MO's within block boundaries. First, it was attempted to find an *optimal* solution[3] [YAU74] only to find out that the algorithm was exponential in complexity. So efforts were directed to finding reasonable heuristics. It was later proved that the microprogram optimization (optimal compaction) problem is indeed NP-complete both in microword dimension [DEW76] and in bit dimension [ROB79]. Four different heuristics have been proposed to solve the local compaction problem and are reviewed in [LAN80].

More recently, in an effort to get more efficient microcode, the compaction was tried not only among MO's within each block boundary but also among MO's beyond block boundaries. This is now known as global compaction. The global compaction of microprograms has been an active research area for the last five years or so and several heuristics have been proposed.

[3]Note that in a straight line microcode, getting the *minimum* microcode size is equivalent to getting the *minimum* execution speed of the microcode.

3.1. Local Compaction

In [DAV81], the same group of people who reviewed four local compaction heuristics in [LAN80] reported the experimental result on these heuristics and claimed that the local compaction problem was essentially solved. Based on the experimentation, they recommended first-come first-serve (see Section 3.1.1) with some modification and list scheduling (see Section 3.1.4).

However, Vegdahl revisited the local compaction problem again in his thesis [VEG83] and proved that the *classical* local compaction problem, which does not consider reordering of source MO's, is not NP-complete, but polynomial in complexity. He presented such an algorithm.

In this section, four local compaction heuristics are briefly reviewed and Vegdahl's thesis results are discussed. In describing the heuristics, we will use one simple example to enhance understanding. The DDG of the example microcode is shown in Figure 3.1. The circled numbers represent microoperations.

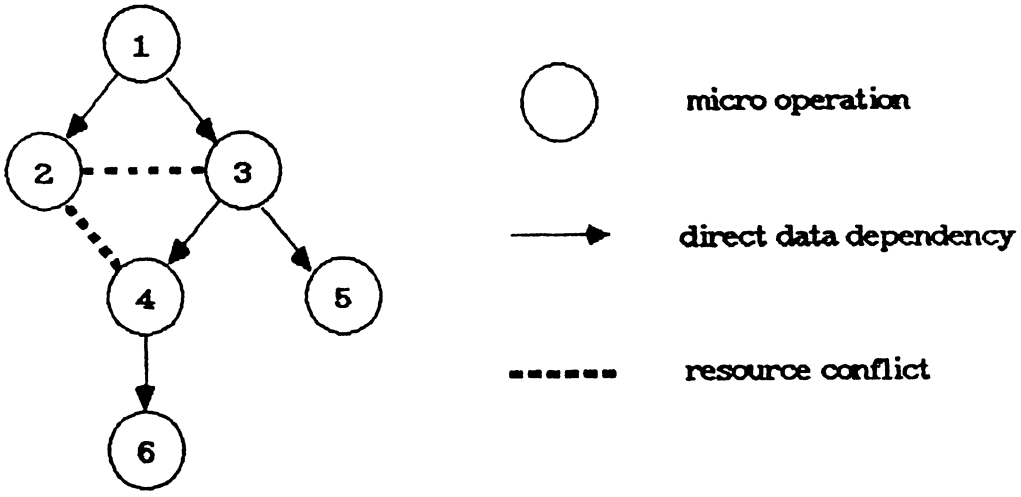


Figure 3.1 Data dependency graph of example microcode

The arrows indicate the direct data dependency relation between MO's. The dotted lines are used for convenience to indicate resource conflict between MO's. We will show the working of each heuristic on the same sample microcode along with the description of the heuristic itself.

3.1.1. First-Come, First-Serve

The first-come first-serve heuristic described by Dasgupta and Tartar [DAS76] operates on a straight line microcode (SLM) which is in the form of a list of MO's. MO's from the SLM are added, in the order in which they appear on the list, to an initially empty list of MI's. Here is the outline of the heuristic.

```
for each MO do
{
        Determine the rise limit of the MO by searching MI's
        from bottom to top using data dependency.

        Place the MO in the first MI which has no resource
        conflict with the MO, starting from the rise limit
        to the last MI

        if the MO is not placed, create a MI at the bottom
        to hold it.
}
```

The detail procedure follows.

The search for an existing MI to which the current MO can be added begins with a data dependency analysis. Starting at the bottom of the MI list and proceeding upward, examine each MI if the MO under current consideration is not data dependent on any MO in that MI. If the current MO is data dependent

on an MO in the i th MI, the current MO cannot be placed in any MI earlier than i or i itself. The object of this search is to find the earliest (highest) MI in which the new MO can be placed without violating the ordering imposed by the data dependencies in the SLM. This MI is called the *rise limit.*

The next step in the search for an existing MI is the examination of resource conflicts. To be placed in a MI, the current MO must not conflict with any MO already in that MI. Assuming that a rise limit i was found, search downward in the list, starting with MI i, for some MI in which the new MO can be placed. When such a MI is found, add the new MO. If no such MI is found, add the new MO to the end of the MI list, thus forming a new MI containing only one MO.

If no rise limit was found, then the current MO was not data dependent on any MO in the MI list, and the MO can be added to any MI with which it has no conflicts. In this case begin the downward search at the top of the MI list. If

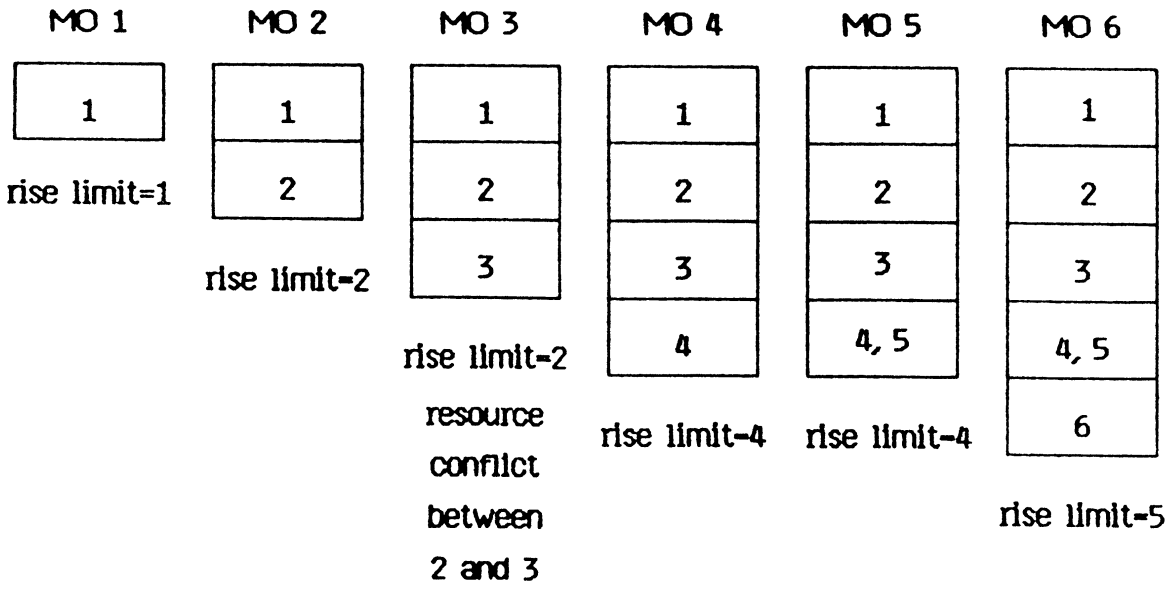


Figure 3.2 The working of first-come first serve

there are no MI's to which the MO can be added without conflict, use the MO to form a new MI at the top of all the other MI's. Placing this MO at the top will keep it from blocking any new MO's that depend on it.

Figure 3.2 shows the working of the first-come first-serve heuristic on the sample microcode. Each column shows the placement of one MO as indicated on top. The boxes represent MI's. The rise limit is shown for each MO. After MO 1 is placed, the rise limit for MO 2 is 2 because of data dependency. Even though the rise limit of MO 3 is also 2, it can not be placed in the same MI with MO 2 because of resource conflict between MO 2 and MO 3. The remaining MO's are placed similarly and the result is five MI's.

3.1.2. Critical Path

The critical path heuristic for microcode compaction was introduced by Ramamoorthy and Tsuchiya [RAM74]. This critical path heuristic attempts to identify MO's that must be executed at a certain time in order for the list of MI's to be optimal. The MO's chosen are those which are on a longest path (the critical path) through the DDG. As noted, the minimum possible number of MI's is just the length of the longest path. Here is the outline of the heuristic.

Determine early partition
Determine late partition
Determine critical partition
Revise the critical partition
Place non-critical MO's

The detailed description of each step follows.

The first step is to create an *early partition* (EP). Each time frame of the EP contains those MO's that can be executed in that time, at the earliest. So

starting from the first time frame, fill in each time frame with all executable MO's and if there is no more executable MO's, then go on to the next time frame. A MO must be placed in the time frame of the EP after the frame of its latest ancestor. Data dependencies alone determine the placement of MO's in the EP. Conflicts will be resolved later. The early partition of the sample microcode is shown in Figure 3.3(a). Each different shading indicates a separate time frame.

The next step is the creation of a *late partition* (LP). In the LP the latest possible timings of the MO's are considered. Each time frame of the LP contains those MO's that can be executed in that time, at the latest. It starts from the MO for which no other MO is data dependent and goes on to the one earlier time frame and fills it with all MO's for which no other MO is data dependent but those which are in the later time frame. If the data dependency graph is represented by a matrix, the LP can be created by applying the EP algorithm to the transpose of the graph matrix. The construction of the LP resembles the

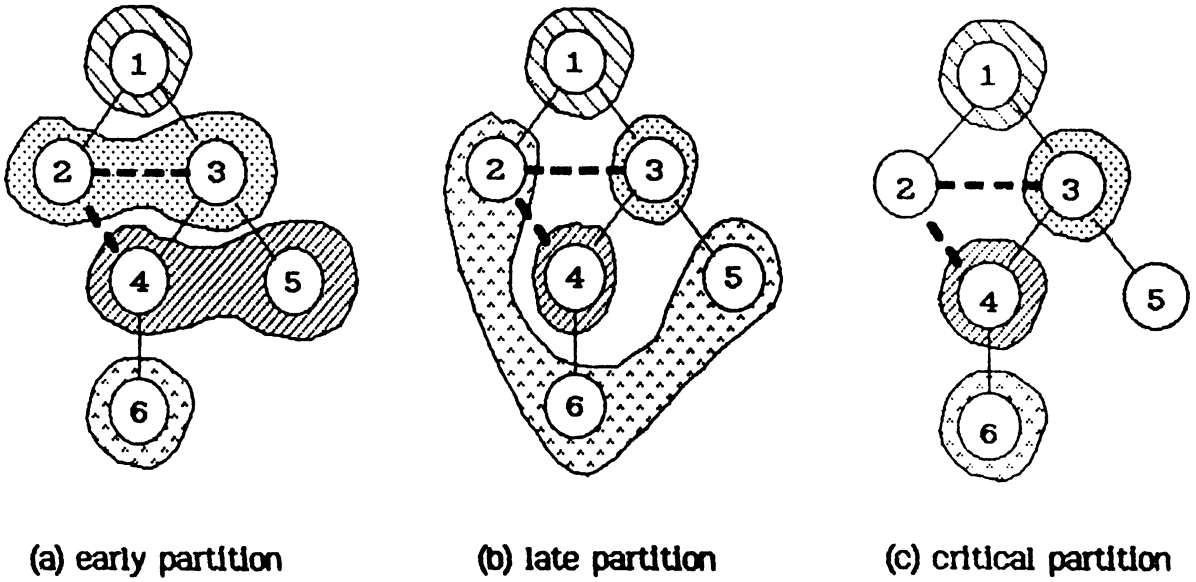


Figure 3.3 The working of critical path

construction of the EP in reverse. The late partition of the sample microcode is shown in Figure 3.3(b).

The construction of these partitions facilitates the identification of MO's on the critical path of the DDG. These MO's, called critical MO's, are just those with the same timing in both early partition and late partition. It is called a *critical partition* (CP). The critical partition of the sample microcode is shown in Figure 3.3(c) by the shaded nodes.

Since the CP was constructed by considering only data dependencies between MO's, two MO's in the same frame of the CP may conflict. And since the frames of the CP will serve as a basis for MI's in the list of MI's, conflicting MO's in a frame must be separated. The result of separating all the conflicting MO's in each frame is called the ***revised critical partition*** (RCP). The separated frame forms subframes. The revised critical partition of the sample microcode happens to be the same as the critical partition shown in Figure 3.3(c). In other words, in this particular example, the critical partition contains no time frame which contains two conflicting MO's.

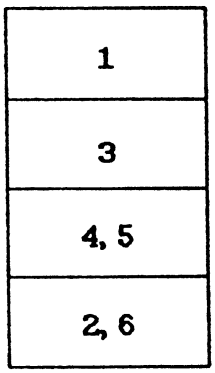


Figure 3.4 Result of critical path

The next and final step of the heuristic is to add the noncritical MO's to the RCP, forming the final list of MI's. For each noncritical MO, we search the RCP from the frame containing the MO in the EP to the frame containing the MO in the LP for a MI to which the noncritical MO can be added. If the MO cannot be added to any of the frames within this range, a new subframe is created. The final result of critical path heuristic on the sample microcode is shown in Figure 3.4 after placing non-critical MO's 2 and 5.

3.1.3. Branch and Bound

The third method of solving the local compaction problem is branch and bound (BAB), a general class of tree-searching scheduling algorithms. Yau, Schowe and Tsuchiya [YAU74] were the first to describe the application of this technique to microcode compaction.

In BAB, a tree is built, the nodes of which correspond to microinstructions. A path from the root of the tree to a leaf is an ordering of MI's, and thus a list of MI's. The tree will branch whenever there is more than one MI that can be placed at a point in the list of MI's. A complete tree represents every possible MI ordering.

There are two variants of this algorithm. The first is BAB exhaustive, in which every branch of the tree that can possibly lead to an optimal MO ordering is explored. The other is BAB heuristic, in which pruning is done to the tree. BAB exhaustive is an optimal algorithm, and running in exponential time, while BAB heuristic is not guaranteed optimal and can be made to run in polynomial time.

The growth of the tree can be bounded even in BAB exhaustive. As in the other algorithms, calculate the lower bound on the number of MI's in the best possible ordering. (Remember that this is the longest path through the DDG.)

A path through the BAB tree of this length represents an optimal ordering, and the algorithm can stop once such a path is obtained. The growth of the tree can be further bounded by remembering the length of the best (shortest) path found so far. If the length of an incomplete list of MI's is greater than or equal to this length, the current path needs no further consideration.

Definition: *data ready set* (DRS) is the set of all unplaced MO's that are not data dependent on any unplaced MO.

Definition: *complete instructions* (CI's) is defined as an instruction to which no other MO's in the DRS can be added. Thus a CI is not a subset of any other legal MI given a DRS.

The outline of the BAB exhaustive algorithm is presented here using a recursive function.

```
form initial data ready set;
call FormCI;

procedure FormCI
{
        form all possible Complete Instructions;
        for each Complete Instruction do
        {
                form new data ready set;
                if data ready set is not empty
                        call FormCI;
        }
}
```

A more detailed description of the algorithm follows.

Like the critical path heuristic, the BAB algorithm gets its information on the data dependencies of MO's of an SLM through a DDG. The first step of the BAB exhaustive algorithm is the construction of a DRS. The contents of the DRS change as execution of the algorithm progresses. The initial DRS contains just those MO's not data dependent on any other MO.

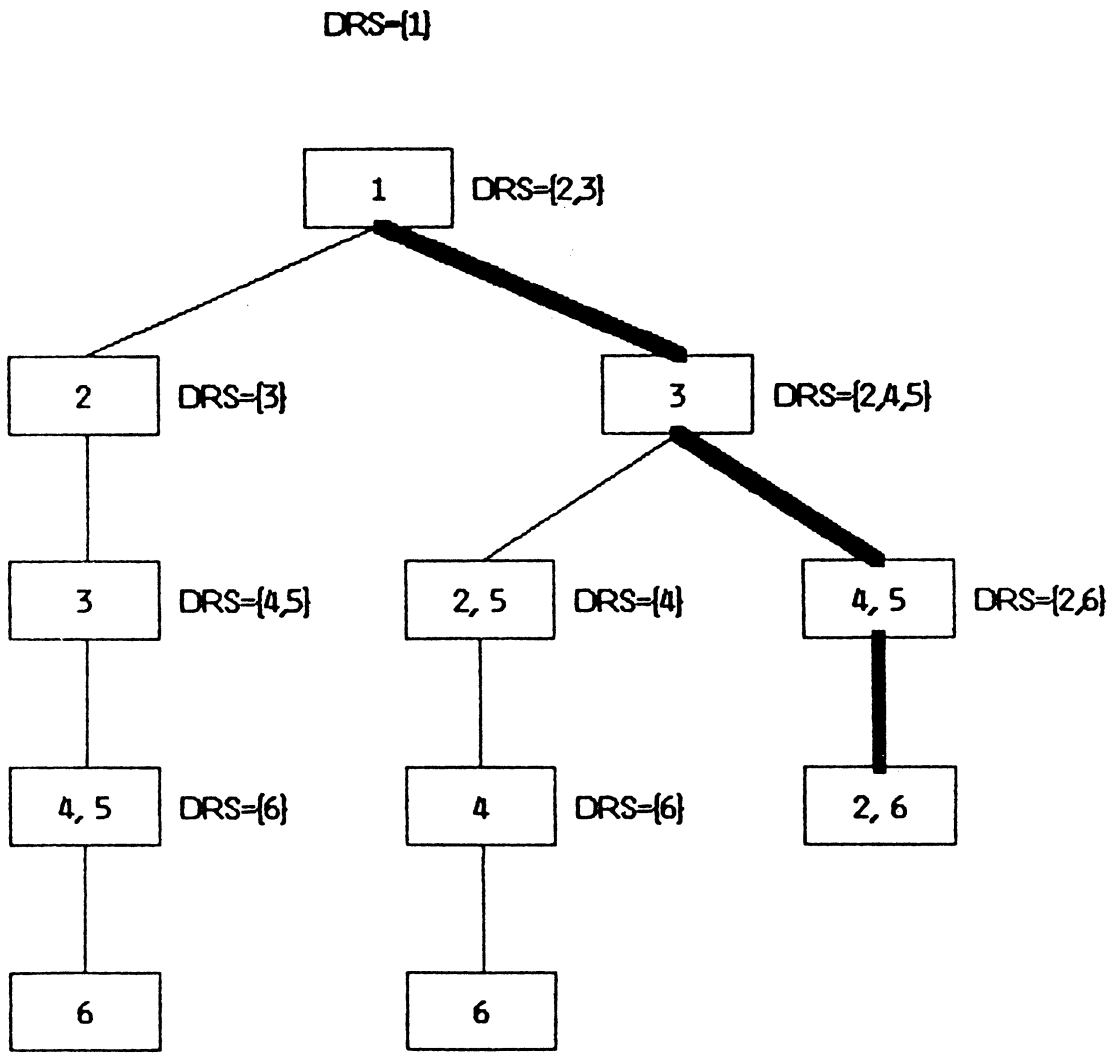


Figure 3.5 Working of branch and bound exhaustive

The next step is to form MI's from the MO's in the DRS. We wish to form only the largest possible MI's. These are called CI's. CI's make up the nodes of the BAB tree so that a path through the tree is a list of CI's or a list of MI's. For the detailed algorithm of forming CI's, see [LAN80] Sec. 3.

Now, for each CI, compute a new DRS and form the next CI's which will make branches from the previous node (or CI). Repeat this for all the branches until there is no more MO's left.

Figure 3.5 shows working of the BAB exhaustive algorithm on the sample microcode in Figure 3.1. It starts with the initial DRS which contains just MO 1. So there is only one CI with MO 1, which is shown in the box. The next DRS after the CI is shown on the right side of the CI box. From the DRS of { 2, 3 }, two CI's are formed; namely one with MO 2 and the other with MO 3. The two CI's make two branches in the graph. The process continues similarly until all possible branches are considered. The shortest path from the node to a leaf, which is indicated with thick lines, represents the optimal solution.

3.1.4. List Scheduling

The list scheduling can be considered a special case of the BAB heuristic but it is important in its own right. The heuristic used in the list scheduling is as follows.

Instead of examining every complete instruction generated from a DRS, examine only the *best* CI, where the best CI is determined by some metric. The outline of the list scheduling, shown below, is very similar to that of the BAB exhaustive shown in Section 3.1.3.

```
form initial data ready set;
call FormCI;

procedure FormCI
{
        form best complete instruction;
        form new data ready set;
        if data ready set is not empty
                call FormCI;
}
```

The only difference is that inside the procedure FormCI, one best CI is formed and considered instead of examining all possible CI's. This heuristic requires only an amount of time that grows polynomially with the number of MO's in the SLM. The metric used in forming the CI affects the optimality of the compaction. Extensive test of different metrics were reported by Fisher [FIS79]. In the test, the metric used by Wood [WOO78] was shown to be one of the best and is presented here.

Assign a weight to each MO in the DDG. The weight of an MO is the number of levels of descendants of that MO in the DDG. The execution of the heuristic proceeds as follows. Compute a DRS from the DDG. Find the MO in the DRS with the highest weight. Add it to the MI, which is initially empty. Then find the MO with the second highest weight in the DRS. If this MO does not conflict with any MO already in the MI, add it. Otherwise, consider the MO with the next highest weight. Repeat until all the MO's in the DRS have been examined. The resulting MI is the CI with the highest weight. Add this MI to the list of MI's and compute a new DRS. Repeat until all of the MO's in the SLM have been placed.

Figure 3.6 shows the DDG of the sample microcode with the weight of each MO indicated outside of circles. The working of the list scheduling is the same as

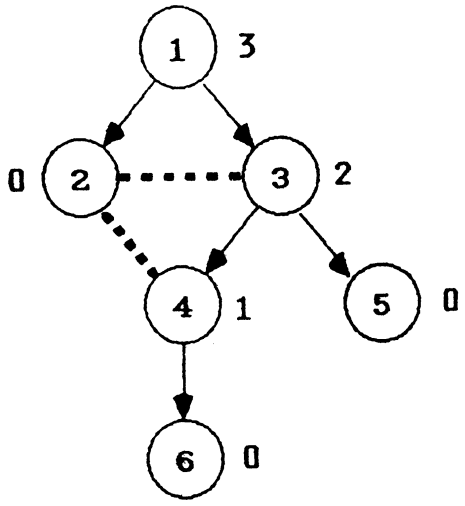


Figure 3.6 Weight of MO's for list scheduling

the optimal path of the BAB exhaustive shown in Figure 3.5 with dark lines. From the DRS of { 2, 3 }, MO 3 is selected since its weight of 2 is higher than the weight of 0 of MO 2. From the DRS of { 2, 4, 5 }, MO 4 is selected for the same reason. Then MO 5 is placed in the same MI while MO 2 is not because of resource conflict with MO 4. The forming of the last MI is obvious.

One last note on this example microcode. Keep in mind that this is just one very specific example. The fact that the first-come first-serve heuristic did not get the optimal solution while the others did does not necessarily indicate inferior performance of first-come first-serve or superior performance of the others. As noted at the beginning, the test result by Landskov et al. [LAN80] recommended first-come first-serve with some modification and list scheduling.

3.1.5. Vegdahl's Thesis

One of the contributions of Vegdahl's thesis is that he has shown that the classical microprogram compaction problem, which does not consider semantics-preserving reordering of source MO's, can be solved optimally in time $O(n^v)$, where v is the number of registers in the hardware, instead of in exponential time, although the order v of the polynomial may be very high.

Definition: A MO may require data that is produced by another MO. If this is the case, then the former is said to be *data dependent* on the latter.

Definition: A MO may destroy data that is required by another MO. If this is the case, then the former is said to be *data antidependent*[4] on the latter. [BAN79]

Vegdahl recognized that data antidependency should not be treated as data dependency in solving microprogram compaction problems, as demonstrated by the following example [VEG82]. Figure 3.7(a) shows an example, where register x is twice used as a temporary. Figure 3.7(b) shows one valid partial ordering of the MO's with data dependency and data antidependency among them. However, Figure 3.7(c) shows another valid partial ordering of the MO's with the same data dependency but different data antidependency. Clearly, the partial ordering shown in Figure 3.7(c) can yield less number of MI's when compacted.

Vegdahl contended that the problem of optimally ordering the MO's - and thereby determining the data antidependencies - is a more difficult problem than compaction. He then presented an algorithm, called a chain-matrix compaction algorithm, to optimally solve the compaction problem, once data antidependencies are specified. He concluded that the determination of data antidependencies is likely the more difficult problem, because the general compaction problem is NP-complete.

---

[4] Note that data antidependency was defined as a part of data dependency in chapter II.

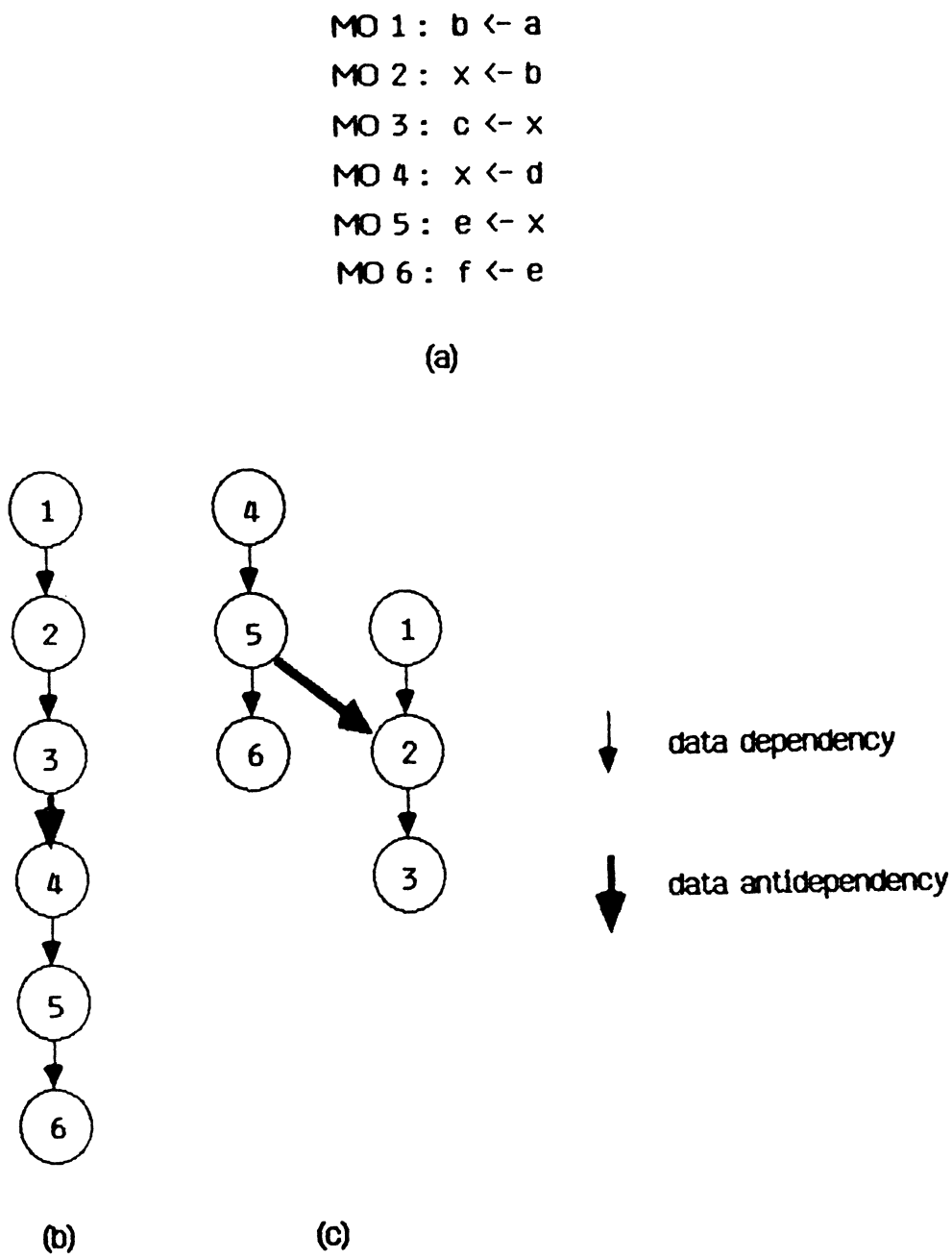


Figure 3.7 MO's with different data antidependencies

We now present a summary of the chain-matrix compaction algorithm from [VEG83]. For a complete description with an example, refer to [VEG82].

The MO's are first divided into v chains, each being placed in the chain corresponding to the register it writes. There is a strict ordering on the MO's in any chain - namely the order in which they appear in the source program. Each chain defines one dimension of a directed acyclic graph that is shaped as a v-dimensional matrix. The edges of the graph correspond to MI's; missing edges denote conflicting MI's. Each node in the graph represents a set of MO's; the node corresponding to matrix element <1,2, ... > represents the set of MO's containing the first MO of chain 1, the first two MO's of chain 2, etc. Any node whose MO's violate a data dependency is removed. At this point, the problem is reduced to a single-source shortest path problem that can be solved using dynamic programming in time that is linear in the number of nodes.

3.2. Global Compaction

While the local compaction problem is concerned only with possible MO movements within each block boundary, methods for solving the global compaction problem attempt to achieve even more compaction by moving MO's over the block boundaries. This widened scope opens up a whole new set of issues. To name a few: When is it legal to move a MO to another block? What kind of provisions have to be made? How far can a MO move: just to adjacent blocks only, or further to over several block boundaries? How do we optimize overall performance, given that some blocks are more frequently executed than others? How much extra space is needed?

The global compaction problem is still considered an active research area. We present some of the previous works here and propose some improvements in the next chapter.

3.2.1. Wood [WOO79a, WOO79b]

This is one of the earlier attempts to compact MO's beyond block boundaries. It is an elegant and clean method even if somewhat restrictive. The basic idea is to allow MO movements among the same block level in a block structured microprogram.

The big precondition of this work is that the source microcode has to be written in a block structured fashion, i.e., only if-then-(else) and (nested) loop structures are allowed. Given a block structured source microcode, the goal is to allow MO's to *migrate* over a conditional block, but not to let them *land* inside the block nor to allow any MO to migrate beyond the confines of its own block.

To achieve the goal, a concept of a hierarchy of levels of MO's within the microprogram is introduced. MO's may be of two basic types: a primitive MO which is an atomic entity, or a block-type MO which may be expanded at one level lower to a set of component MO's. These component MO's may, in turn, be either primitive or block type. A block-type MO represents a whole block of MO's which may be an if-then-else block or a loop block.

Using the MO's of these types, data dependency is defined between MO's at the same level as follows.

> Multi-level dependency rule: If A and B are primitive MO's 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.

The basic structure of the heuristic can be best described by Wood's own diagram [WOO79a], which is reproduced in Figure 3.8. In the figure, the term DAS (data available set) is used which has the same meaning with the DRS (data ready set). In selecting MO from DRS, block type MO's are selected first before primitive MO's are selected. So, the compaction starts at the lowest level block and works its way up. A separate DRS and waiting list is maintained for

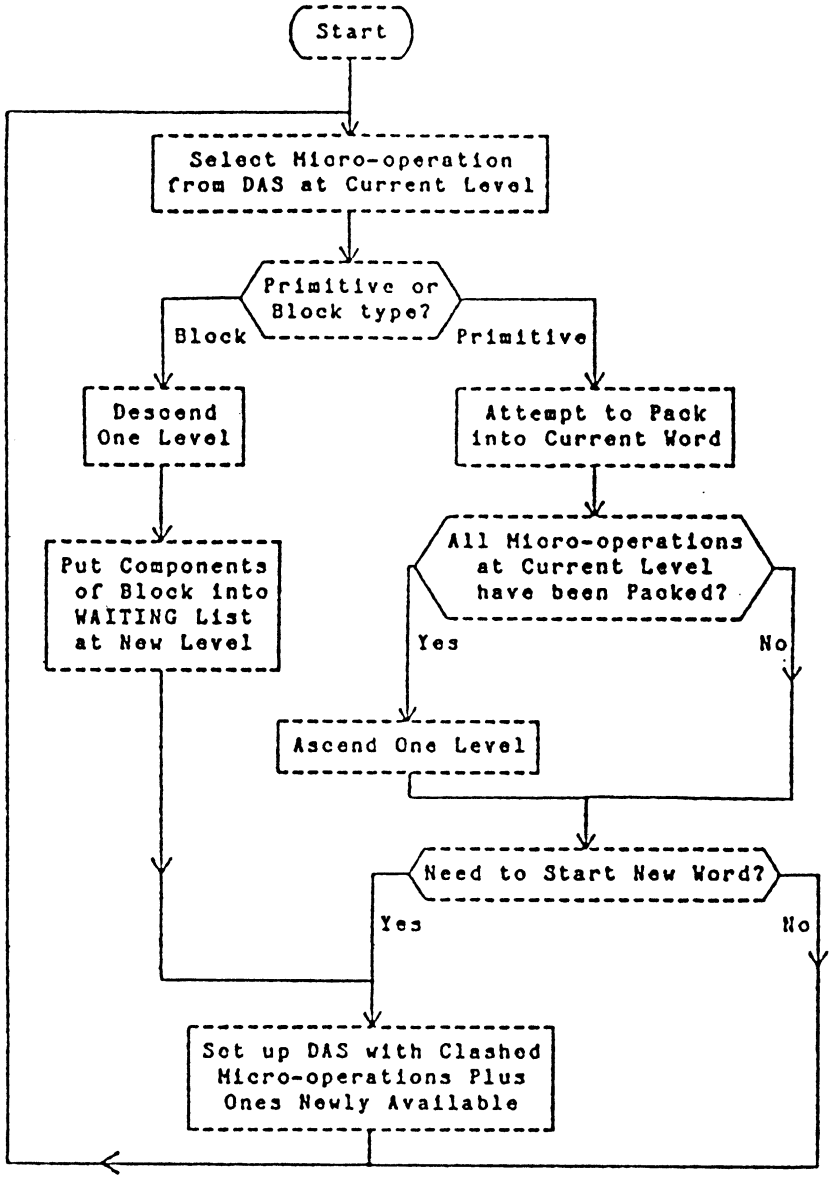


Figure 3.8 Wood's compaction heuristic [WOO79a]

each level of MO's. The waiting list is a list of the MO's at that level which have not yet been made available for packing.

Even though this is a nice clean way of solving the global compaction problem, it is too restrictive and may not perform as well as the other methods. It does not allow many of the legal MO motions which will be discussed in the following sections.

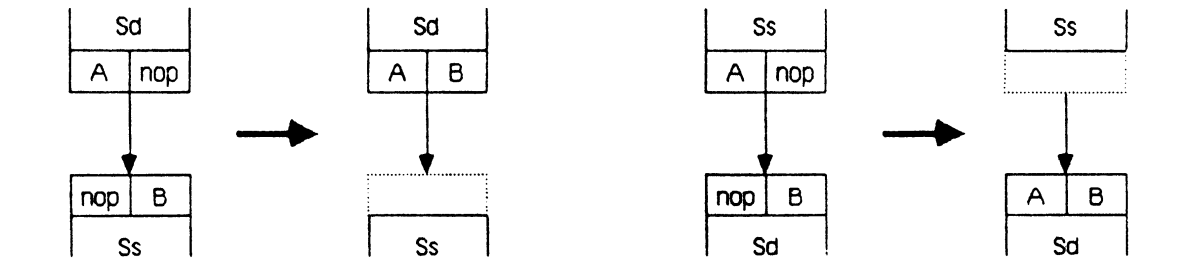
3.2.2. Tokoro [TOK81]

Tokoro's approach to solving the global compaction problem is based on his identification of a set of rules of legal MO motions over the block boundaries. It is sometimes referred to as *automated menu* method. Even though the set of rules is very comprehensive and almost complete,[5] the proposed heuristic procedure to utilize these rules is rather ineffective.

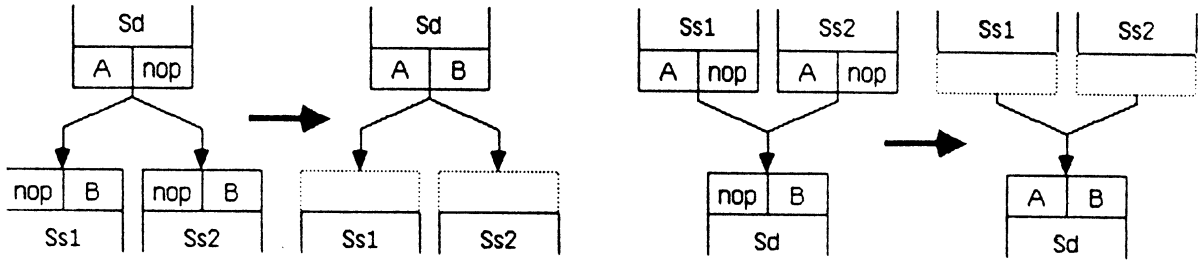
The pictorial representations of Tokoro's MO movement rules are reproduced from his paper [TOK81] in Figure 3.9. Note that the MO's in these figures are drawn at the edges of the blocks just to make the explanation of the concept simpler. In fact, MO's can be transferred from the inside of blocks and/or into MI's within blocks as long as data dependency relations are maintained. In Figure 3.9, A, B and nop (no operation) are MO's and S_s represents a source block and S_d represents a destination block. MO's are transferred from source blocks to destination blocks.

The simple transfer type optimization shown in Figure 3.9(a) and the common MO type optimization shown in Figure 3.9(b) are obvious and self-explanatory. A redundant MO is one in which all resources written by the MO are never referenced afterwards. The redundancy reduction type optimization shown in Figure 3.9(c) simply eliminates a redundant MO to save a cycle. The redundancy insertion type optimization shown in Figure 3.9(d) actually adds a redundant MO to one path without adding a cycle to make it possible to save a

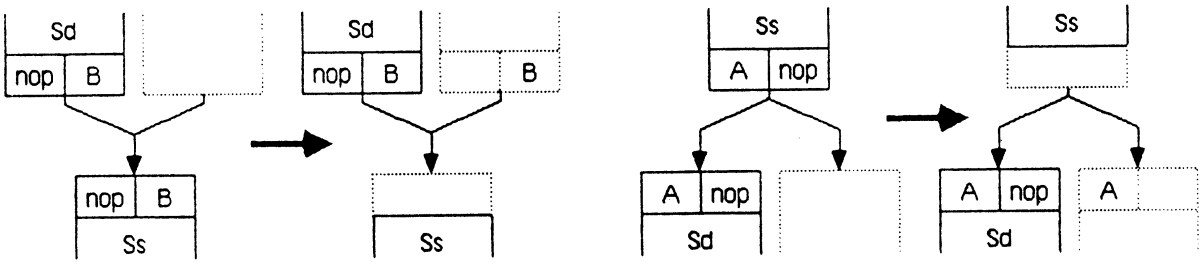
[5]Since Tokoro's rules deal only with MO motions between adjacent blocks (he calls it contiguous segments), MO movements over several blocks, the kind of MO migration described by Wood [WOO79b] (see sec. 3.2.1) for example, are not allowed.



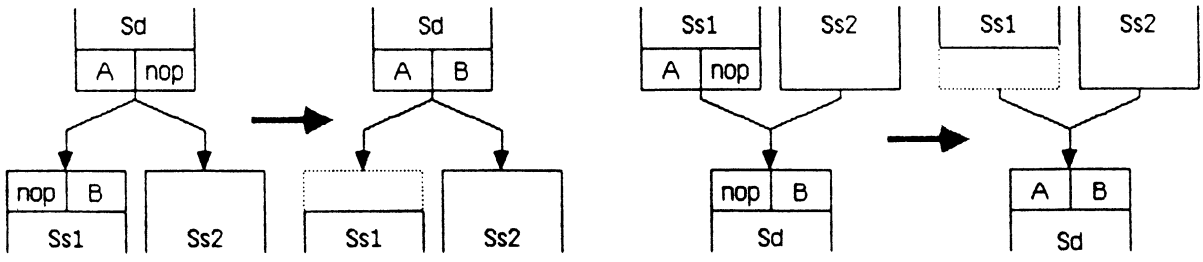
(a) Simple transfer type optimization



(b) Common micro-operation type optimization



(c) Redundancy reduction type optimization



(d) Redundancy insertion type optimization

Figure 3.9 Tokoro's MO movement rules

cycle at the other path.

The detailed procedure to apply these rules, which will not be repeated here, is basically as follows. For each MO in each block, search for a possible move which can save a cycle by checking against all the rules. If such a move can be found, move the MO, otherwise select the next MO and repeat the entire search.

This automated menu method appears to suffer from the following shortcomings [FIS81a]. Each time a MO is moved, it opens up more possible motions. Thus, the automated menu method implies a massive and expensive tree search with many possibilities at each step. Evaluating each move means recompacting up to three blocks, an expensive operation which would be repeated quite often. To find a sequence of very profitable moves, one often has to go through an initial sequence of moves which are either not profitable, or, worse still, actually make the code longer. Locating such a sequence involves abandoning attempts to prune this expensive search tree.

3.2.3. Fisher [FIS79, FIS81a]

The trace scheduling appears to be the best proposed method to solve the global compaction problem. Its basic concept is nice and elegant and the available indications show that it gives good speed-up of execution time of compacted microcodes. However, in actual realization of the elegant concept, it gives some undesirable side effects which need to be corrected. And it is this correction process, called bookkeeping, that is nontrivial and complicated as will be described later.

In brief, trace scheduling proceeds as follows.

To schedule a given MO, we repeatedly pick the *most likely* trace from among the uncompacted MO's, build the trace DDG, and compact it. After each trace is compacted, some duplication of MO's into locations off

the trace is done to keep the semantics of the original microprogram unchanged. When no MO's remain, compaction has been completed.

Figure 3.10 shows the basic steps of the trace scheduling. The figures are redrawn from [FIS82] with additional labeling to make it easier to explain. In Figure 3.10(a), each box represents a block of MO's and the lines between blocks indicate possible execution path. So the blocks with more than one line at the bottom are the blocks which end with a conditional branch. In Figure 3.10(b), the blocks covered by the shaded area are assumed to form a path with the highest probability of execution. In other words, the blocks form a trace. In Figure 3.10(c), after some preprocessing,[6] the MO's in the trace are treated as if they belong to a single block and compacted using list scheduling. After the list scheduling is done on the trace, bookkeeping is necessary, which possibly includes massive copying of blocks.

First, let's consider repairing rejoins. Rejoin is a join in a trace to which an execution flow from a non-trace block is reached. When there were joins *to* the trace, we now must find a location in the new sequence of MI's to join to. MO's may have moved above and below the old join. We may only rejoin to places that have no MO's at or below them which had been above the old join, since we don't want to execute such MO's. When the highest point for rejoin is found, all MO's which had been below the old join but are now above the new rejoin have to be copied into the joining block. In Figure 3.10(d), blocks denoted by "R" are the blocks which contain the copied MO's because of repairing rejoins, and N_a and N_b are two newly found rejoining points.

Second, consider conditional jumps. Some MO's which were originally above a conditional jump on the trace may have been scheduled in an MI below the conditional jump. We must copy these MO's to the blocks that the conditional jump jumps to. If all registers written by the MO are dead at the beginning of

[6]See conditional source registers

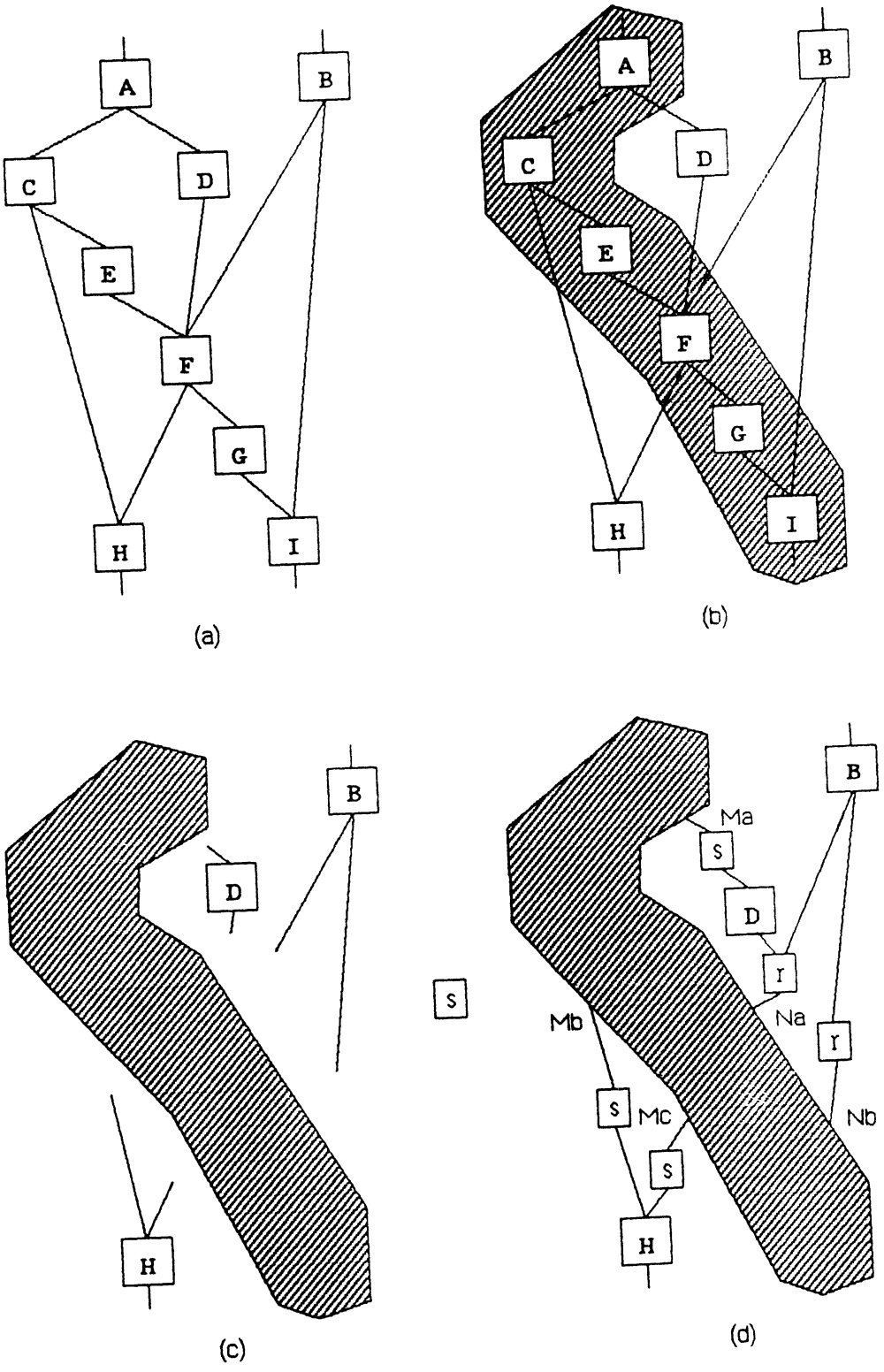


Figure 3.10 Fisher's trace scheduling

the block, then the MO does not need to be copied. In Figure 3.10(d), blocks denoted by "S" are the blocks which contain the copied MO's because of conditional jumps.

Up to this point, the bookkeeping stage is not overly-complicated. But here is the big question.

> What if the rejoin N_a happens to be below the conditional jump M_c in Figure 3.10(d)?

In the original source microcode, the conditional jump was *below* the join but if the conditional jump is *above* the join, for the execution flow coming from the block B, the conditional jump does not exist any more. So the execution path of block B, block F and block H or the semantics of that path of the microprogram cannot be realized after the scheduling. To solve this problem, instead of rejoining block B to the trace, the blocks F, G, H and I are duplicated after the block B, as shown in Figure 3.11. This kind of block copying is the major source of complication and possible memory explosion.

To prevent possibly exponential growth of memory size, Fisher suggested modifications of trace scheduling [FIS81a].

(1) If the block ends in a conditional jump, we draw a DDG edge to the jump from each MO which is above the jump on the trace and write a register alive in the branch.

(2) If the start of the block is a point at which a rejoin to the trace is made, we draw DDG edges to each MO free at the top of the block from each MO which is in an earlier block on the trace and has no successors from earlier blocks.

Fisher recommended the above be used if the expected probability of a block's being reached is below some threshold. However, we suspect that there are some difficulties. First, the major source of extensive copying of blocks is

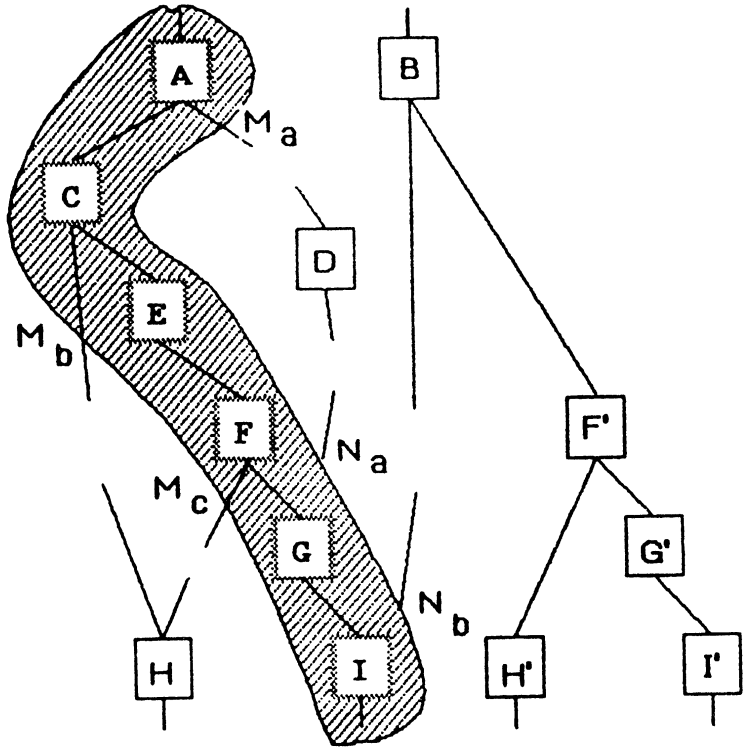


Figure 3.11 Copied blocks in the trace scheduling

from initial long traces which therefore have higher probabilities of being executed. Thus, fixing blocks with lower probability of being executed, after possibly extensive copying has already been done, helps relatively little. If, however, the threshold is raised to include blocks of high probability of execution, then it becomes close to compacting each block separately. Second, in long traces, the memory size growth is so sensitive to minor changes in source code that it is possible for one small change in source code to almost double the size of compacted code. The example microcode in section 5.6.1 has such sensitivity. Third, the bookkeeping stage which is the major source of complication remains to be the same.

CHAPTER IV

BETA COMPACTION ALGORITHM

Chapter III has reviewed several local compaction and global compaction techniques for microprograms. It is clear that global compaction gives better performance than local compaction, since some extra compaction over the block boundaries is done. Among the heuristics to solve the global compaction problem, trace scheduling appears to be the best proposed method so far. However, the trace scheduling has some serious drawbacks, as described in section 3.2.3. In particular, the kind of block copying shown in Figure 3.11, which can possibly cause memory explosion, may be unacceptable in many applications. Also, the bookkeeping stage of the trace scheduling is so complex that Fisher avoided a formal description of it in his paper [FIS81a] and later found a major bug during verification [NIC84].

In Figure 3.10 of previous chapter, it can be seen that the reason for all the block copying is the fact that the rejoin N_a happens to be below the conditional jump M_c after the compaction of the trace. This can happen only around the block which has a join at the beginning and a conditional jump at the end, as shown in Figure 4.1. We call such block a junction block. Having recognized that junction blocks can cause the serious complication, we propose a modification on the trace scheduling to avoid the complication. The modification is to cut traces at each junction block. By making this modification, we not only prevent the kind of block copying shown in Figure 3.11, but also simplify the bookkeeping stage tremendously. Also, the worst case memory growth is reduced from exponential to $O(n^2)$. We call this improved trace scheduling a beta compaction.

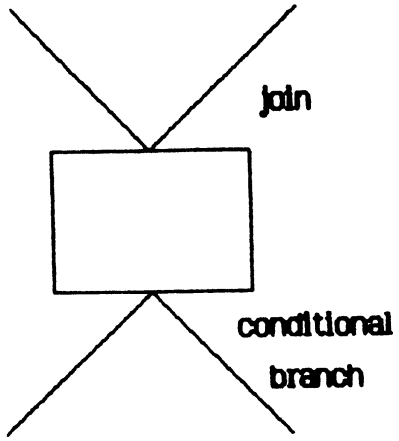


Figure 4.1 Junction block

The beta compaction has all the above nice properties with very little trade off in compaction. It performs all the compaction that the trace scheduling does except that it does not allow MO movement across a junction block. MO's may still move into and out of junction blocks but MO's may not migrate across junction blocks. We conjecture that this kind of MO movement, which is allowed by trace scheduling, occurs very rarely, as will be shown experimentally in the next chapter.

4.1. Algorithm Description

Informally, the beta compaction algorithm proceeds as follows.

An entry block with highest probability is selected. Starting from the block, a trace is selected by following its descendant block of highest probability until it reaches an exit block or a *junction block* or an already compacted block. The blocks in the trace are treated as if they are a sin-

gle block, and compacted using list scheduling with copying of MO's as necessary. This process is repeated until all blocks are compacted.

A more detailed description follows. In Section 4.2, a working example of beta compaction is given.

Input:

Sequence of MO's (label, instruction, next address, destination registers, source registers, resource vector).

Output:

Sequence of MI's where each MI contains one or more MO's.

Procedure:

(1) Initialization and preprocessing

(2) Repeat until all blocks are compacted

```
{
        Pick trace
        List scheduling
        Bookkeeping
}
```

During the initialization and preprocessing, dummy blocks may be inserted wherever a conditional branch and a join directly meet as shown in Figure 4.2. These points are the only places where extra blocks may have to be created to hold copied MO's in the bookkeeping stage. Insertion of dummy blocks at the beginning totally eliminates the need to create blocks later. Empty dummy blocks after the compaction will simply be deleted.

Loops are handled the same way as they are in trace scheduling. The inner most loop is compacted first and the compacted loop is represented as a pseudo MO which contains all the data dependency information and serves as a MO in

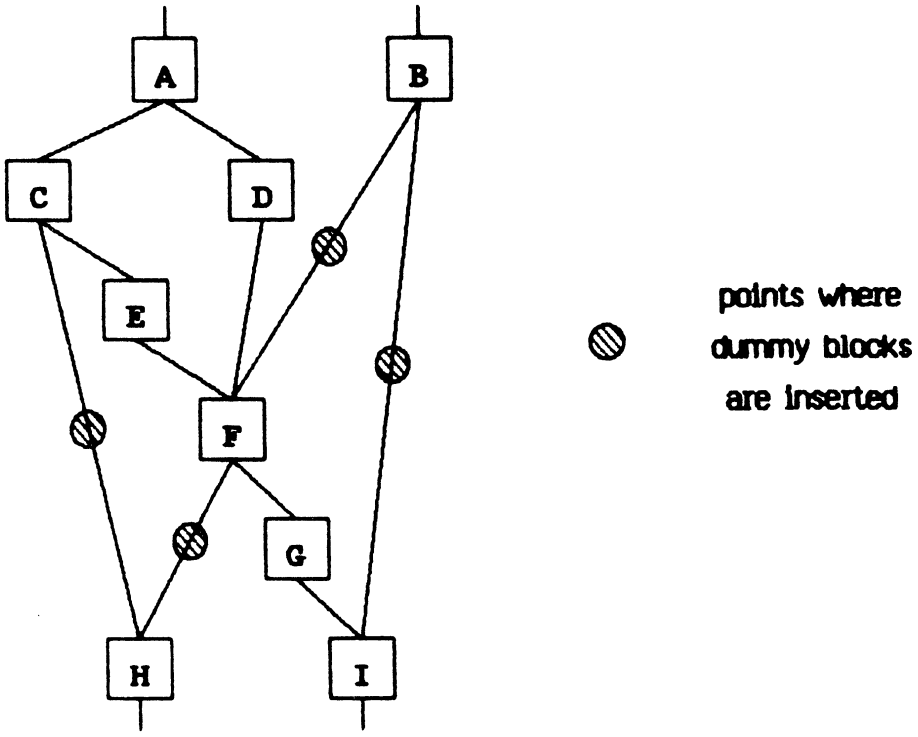


Figure 4.2 Insertion of dummy blocks

the compaction of outer loop. Fisher proposed unwinding of loops in [FIS82] as a way to compact loops. In the beta compaction, similar loop unwinding is possible but, except for small inner loops, is probably too costly in space.

The next few sections will describe in detail the major steps of the beta compaction, including live register analysis. Here are a few definitions to help the description.

Definition: A *tracehead set* is a set of blocks in which each element of the set will make the beginning block of a trace.

Definition: *Off-the-trace blocks* are non-trace blocks which are the targets of conditional branches.

4.1.1. Live register analysis

The live register analysis is the process of determining which registers are alive at some point in a program. This information is necessary to allow as much compaction as possible without allowing illegal MO movement. Here we are interested in the live register information at the beginning and at the end of each block. The live register analysis is done once at the beginning as a part of the preprocessing and after new block boundaries are identified in the bookkeeping stage. It is also done every time a MO is copied from one block to another in the bookkeeping stage. The following summary is taken from [AHO77].

Definition: The *depth-first-ordering* of the nodes of a graph is the reverse of the order in which we last visit the nodes in the preorder traversal. *Depth-first-numbers DFN(n)* assigned to each node indicate the depth-first-ordering of the nodes.

Definition: *IN[n]* is the set of registers alive at the point immediately before block n and *OUT[n]* is the same immediately after the block.

Definition: *DEF[n]* is the set of registers which are assigned values in block n prior to any use of that register in the block n, and *USE[n]* is the set of registers used in block n prior to any definition thereof.

Input:

A set of blocks each containing a sequence of MO's.

Output:

A list of live registers at the beginning and end of each block.

Procedure:

(1) Compute a depth-first ordering of the nodes. Let the nodes be $n_1, n_2, \ldots, n_N$, such that $DFN[n_i] = i$.

(2) begin

```
        for i := 1 to N do
        begin
                IN[n] := USE[n];
                OUT[n] := empty;
        end
        while changes occur do
                for i := n to 1 by -1 do
                /* in reverse depth-first order */
                begin
                        OUT[n_i] := ⋃ IN[s];
                                    s a successor of n_i
                        IN[n_i] := ( OUT[n_i] - DEF[n_i] ) ⋃ USE[n_i];
                end
end
```

4.1.2. Pick Trace

The pick trace routine determines the next set of trace blocks to be compacted. It selects the highest probability block among non-compacted blocks and iteratively selects the highest probability descendant block until it reaches an exit block, or a junction block, or an already compacted block. After selecting the trace blocks, all the MO's from those blocks are converted to a form acceptable to the list scheduling routine which performs local compaction regardless of block boundaries.

Input:

A set of blocks where each block is a 2-tuple. The first element is the probability of execution and the second is a flag indicating if the block has been compacted.

Output:

The set of blocks which forms an execution path.

Procedure:

(1) Delete empty blocks from the tracehead set.

(2) Initialize the conditional source registers of each MO to empty.

(3) Select a block with the highest probability of execution in the tracehead set. Call the selected block the current block and add it to the trace.

(4) Repeat until the current block is an exit block, a junction block or an already compacted block

{

Update the current block with a block of the highest execution probability among its descendant blocks.

If the new current block is not a compacted block, mark it as compacted and add it to the trace.

}

(5) Update the tracehead set.

(6) Convert the MO's in the trace to the appropriate format for the list scheduling.

The first step of the pick trace procedure is to delete empty blocks from the tracehead set. It is possible that some of the blocks in the tracehead set became empty in the previous bookkeeping procedure. So it is necessary to delete them from the tracehead set and instead add their descendent blocks to the tracehead set.

Search through the blocks in the tracehead set to pick up a block with the highest probability of execution among them. The selected block (call it start block) will be the first block in the trace selected.

Starting from the start block, repeatedly select the next block in the trace by picking the block with the highest probability among the descendant blocks of the current trace block. Mark all the blocks selected as trace blocks. The selection is stopped when a program exit block, a junction block or a marked block (already compacted) block is encountered. A program exit block or a junction block is included as a part of the trace but not a marked block. In the process, the descendant blocks of any trace block except the next trace block is collected as the off-the-trace blocks for that trace block. These off-the-trace blocks are necessary to update the tracehead set and to make some necessary copies of MO's in the bookkeeping stage.

Delete the start block from the tracehead set and add all the off-the-trace blocks of each trace block to the tracehead set.

Convert the MO's in the trace to the appropriate format for the list scheduling. Note that the list scheduling does not know about the blocks and our implementation of the list scheduling requires a slightly different MO format, e.g. sequential numbering of MO's. Also, add the live registers at the beginning of the off-the-trace blocks as the conditional source registers of the conditional branch MO in the trace for those off-the-trace blocks. Add a dummy register to all the conditional branch MO's as a conditional source register and destination register to establish a DDG edge between the conditional branch MO's, and thus keep the order of those MO's.

Figure 4.3 shows an example of selecting traces throughout the course of the beta compaction. Each box in the figure represents a block of MO's. Each subfigure shows the result of a call to the pick trace routine, Figure 4.3(a) being the first and Figure 4.3(e) being the last. It is assumed that the execution probability of blocks A, F, C, G, B, D and H are from high to low in the order of listing. Trace blocks are indicated with dark lined boxes and compacted blocks are indicated with shaded boxes.

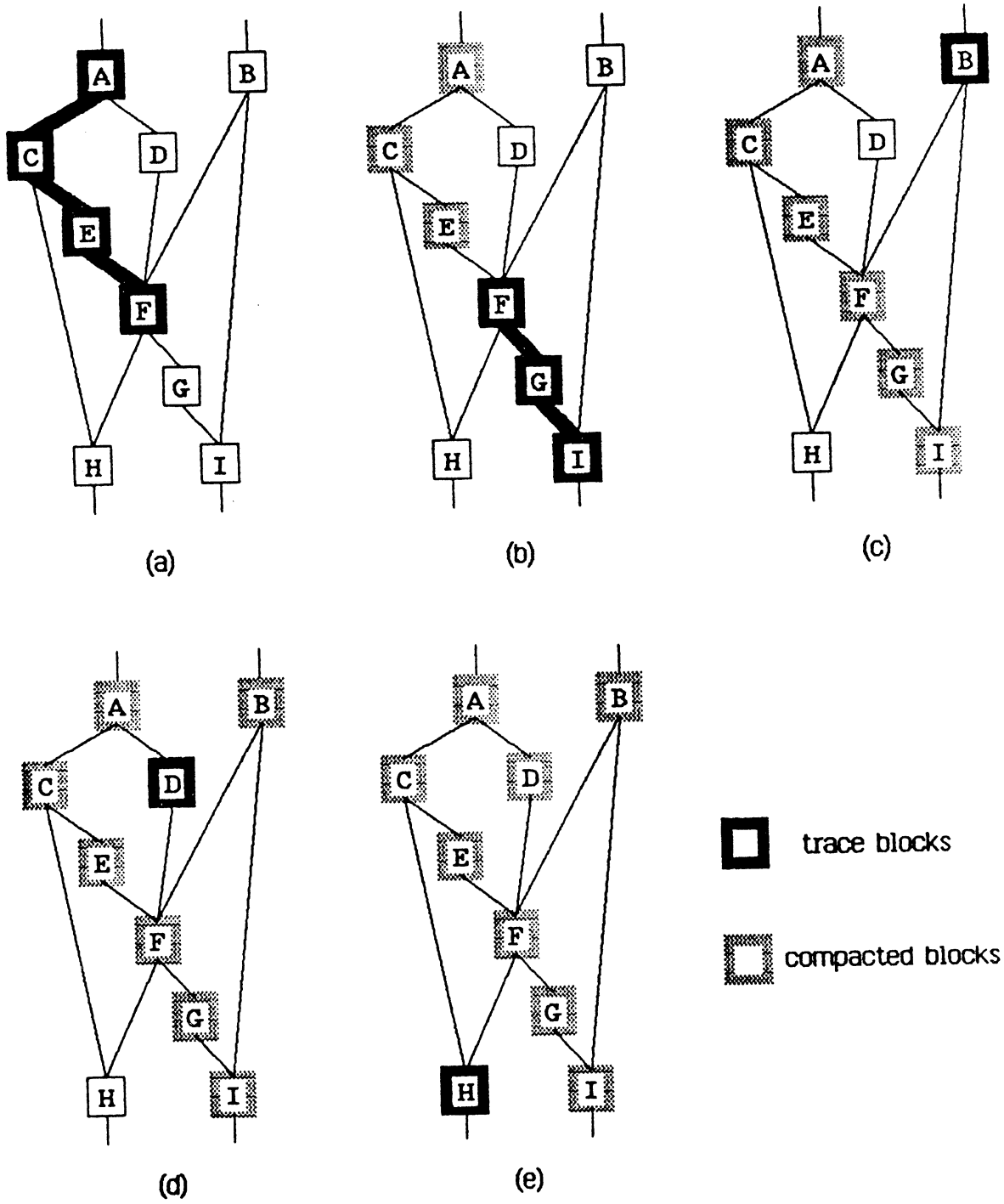


Figure 4.3 Selecting traces

4.1.3. List Scheduling

The list scheduling routine does just local compaction on the list of MO's without any knowledge of block boundaries and generates a list of compacted MI's.

Input:

The sequence of MO's in a basic block or in a trace

Output:

The sequence of compacted MI's

Procedure:

(1) Build DDG on MO's starting from the first MO.

(2) Assign priority value (number of levels of successors in DDG) to each MO.

(3) CYCLE = 0

(4) Data Ready Set (DRS) is formed from all MO's with no predecessors on the DDG.

(5) While DRS is not empty, DO

{

CYCLE = CYCLE + 1

The MO's in DRS are placed in iteration CYCLE
in order of their priority until DRS is exhausted
or no more resources are available for CYCLE.

All MO's so placed are removed from DRS.

All unscheduled MO's not in DRS whose
predecessors have all been scheduled are added
to DRS.

}

The above procedure may be seen as three stages, i.e., build DDG (step 1), assign priority (step 2) and schedule MO's (steps 3, 4 and 5). A more detailed description of each step follows.

(A) Build DDG

Given a sequence of MO's, a DDG is built based on source and destination registers of the MO's.

Starting from the second MO (the first MO forms a single node DDG by itself), add one more node for the MO on the partially built DDG. The DDG is a directed acyclic graph with a set of source nodes and a set of sink nodes as shown in Figure 4.4. The source nodes are the nodes with in-degree zero and the

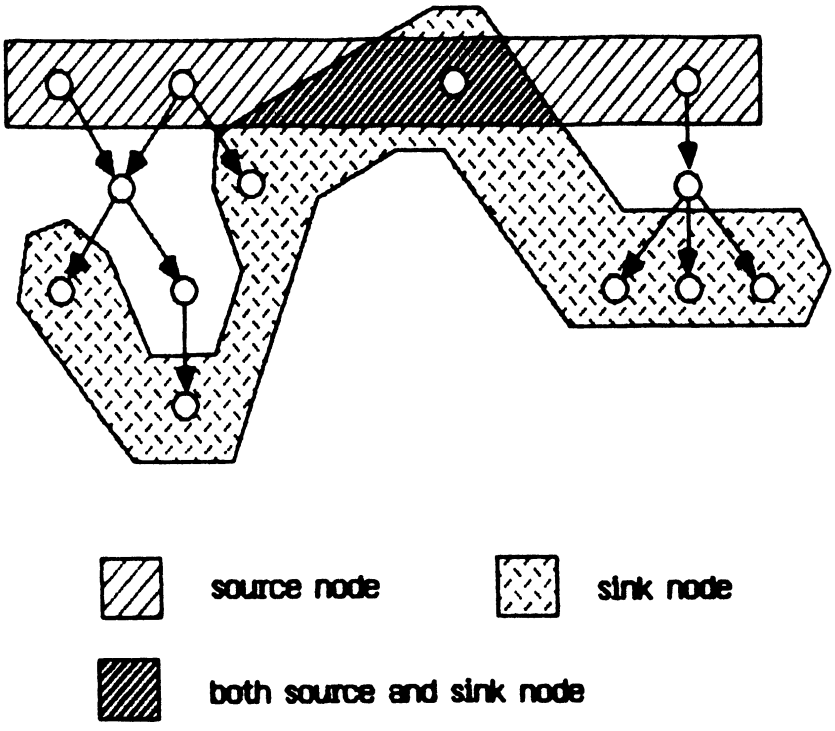


Figure 4.4 Sample data dependency graph

sink nodes are the nodes with out-degree zero. A single node not connected to the rest of the graph is considered to be both a source node and a sink node. Let us call the MO to be added the *current* MO, and the set of nodes being considered in the partially built DDG the *current node set.* At first the elements of the current node set are the sink nodes of the partially built DDG and are updated as the algorithm progresses.

Now take the current MO and see if there is any data interaction between the current MO and the MO's represented by the current node set. If there is any data interaction, i.e., the current MO is directly data dependent on some MO's represented by the current node set, draw edges from the corresponding nodes in the current node set to the new node representing current MO. Note that since we handle MO's in sequence, the current MO may be data dependent on the MO's of the current node set but the MO's of the current node set can not be data dependent on the current MO. Conditional source registers are used when considering data interaction between a conditional jump MO and the MO's that follow but not the MO's that precede. This constraint is to prevent movement of MO's above a conditional jump which may overwrite live registers at off-the-trace blocks but still allow movement of MO's below a conditional jump. In the case of a join, list scheduling is done on a trace with no constraint and new joining points are identified at the bookkeeping stage.

For all the MO's of the current node set on which the current MO is data dependent, tag all their ancestor nodes so that those ancestor nodes may not be included in the updated current nodes. When all the elements of the current node set are considered, update the current node set and repeat the process until the updated current node set is empty.

To update the current node set, consider each node of the set at a time. First, delete the node from the current node set. If the deleted node is not the source node, add the untagged ancestors of the node to the current node set. And delete any element of the current node set if it is an ancestor of another

node.

An example of building DDG is shown in Figure 4.5. The figure shows the change of the current node set and how edges are drawn to the newly added node. The black circle is the newly added node and it is assumed that one data dependency has been found between the node and current node set in Figure 4.5(a). The node labeled P is tagged because it is the ancestor node of the node

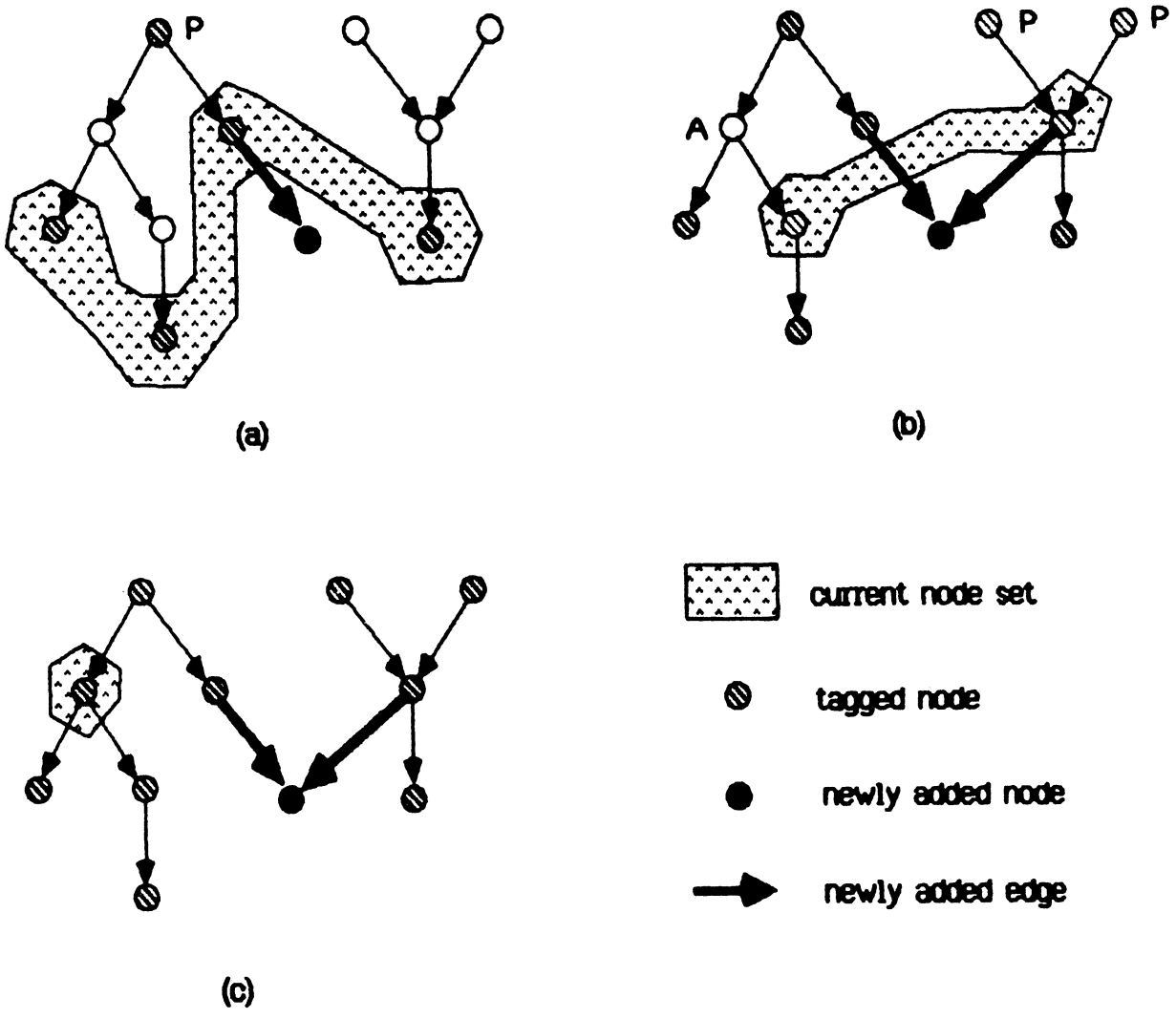


Figure 4.5 Example of building DDG

that the newly added node is data dependent upon. Another data dependency is assumed to be found in Figure 4.5(b) and there are two tagged nodes labeled P for the similar reason. In Figure 4.5(b), the node labeled A is not included in the current node set because it is an ancestor node of another element of the current node set.

(B) Assign priority

There are many ways to define priorities on nodes in DDG. Here we adapt the best known scheme [FIS79] which is the level of descendants in DDG. The algorithm to assign priorities to nodes has a flavor similar to building the DDG algorithm. Basically, assign priority 0 to the sink nodes and priority 1 to their immediate ancestors and the like, up until all the nodes are assigned priorities.

Let us call the set of nodes which have the same priority the *unipriority node set*. The algorithm starts by assigning the set of sink nodes to the unipriority node set. Then assign priority 0 to all the nodes in the unipriority node set. Repeat updating the unipriority node set and assign the priority one higher than the previous one until the unipriority node set becomes empty. Note that some node may be a member of several consecutive unipriority node sets and be assigned the priority associated with the last unipriority node set.

After assigning the priority to each of its elements, the unipriority node set is updated. The node which has no descendant is tagged and deleted. The node whose descendant nodes are all tagged is also tagged and deleted. All the ancestor nodes of the deleted nodes are added to the unipriority node set. Note that if a node in the unipriority node set has a descendant node which is not tagged, the node is neither tagged nor deleted nor its ancestor nodes added to the unipriority node set. Such a node will be assigned a higher priority next time.

An example of assigning priority is shown in Figure 4.6. The figure shows the change of the unipriority node set. Note that some nodes belong to more than one unipriority node set and get the priority of the last unipriority node set.

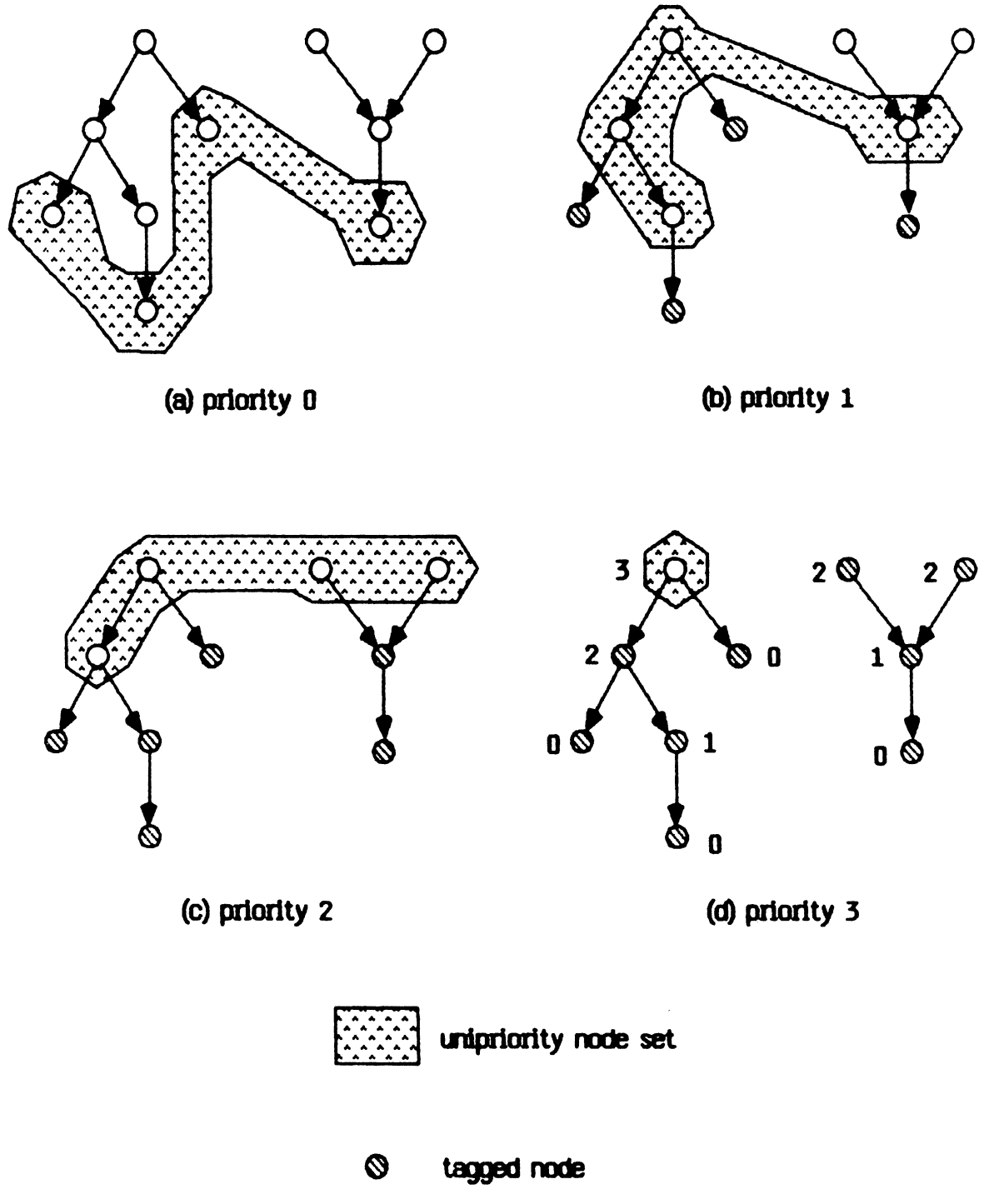


Figure 4.6 Example of assign priority

In Figure 4.6(d), the final priorities are shown on each node.

(C) Schedule MO's

This step actually places MO's into MI's using data dependency, resource usage and priority of MO's. Let us call the MI where MO's are being placed as the current MI. New MI's are created as necessary.

Let us start from the first MI which is the current MI. The initial data ready set (DRS) contains all the MO's of source nodes of DDG. DRS always contains those MO's which can be placed into the current MI without data dependency. But some of the MO's in the DRS may require the same resource, in which case we have to choose one MO over the others, using their priority.

Pick MO's from the DRS in the order of their priority and place them in the current MI until any of the required resources of an MO to be placed is used by an already placed MO in the current MI. When this process of placing MO's into the current MI is stopped because either there is a resource conflict or all the MO's in DRS are placed, DRS is updated.

To update DRS, all the MO's which have been placed in the current MI are first deleted. Then all unplaced MO's not in DRS whose ancestors in DDG have all been placed are added to DRS.

There is one exception in updating DRS. When the trace ends with a conditional branch MO, we want to keep the conditional branch MO in the last MI after the scheduling, which is not guaranteed by the above procedure. To ensure that such conditional branch MO stays in the last MI, we put all the other MO's in DRS before we put the conditional branch MO when we update DRS.

There is another way of handling the placement of such a conditional branch MO. Just schedule MO's without any exception and if the conditional branch MO happens not to be in the last MI, then the MI's below the MI which contains the conditional branch MO will form new blocks in each branch. Using this

method, we may be able to save a cycle in some rare cases at the expense of some space. However, we chose not to use it because it changes the structure of the program, which we are against.

When DRS is updated, another MI is created to hold some MO's in DRS and the process is repeated until all the MO's are placed.

An example of straightforward list scheduling was shown in section 3.1.4. For an example of list scheduling of a trace, which involves conditional source registers, see the beta compaction example in section 4.2.

4.1.4. Bookkeeping

The result of list scheduling is a list of compacted MI's. And we know the list of blocks to which those compacted MI's belong; the blocks are the trace blocks selected at the pick trace stage. The task of this bookkeeping stage is to put the MI's into appropriate blocks and, if necessary, make some copies of MO's to off-the-trace blocks to keep the original semantics of the microprogram unchanged.

Input:

A program in which a trace has been compacted into a sequence of MI's.

Output:

A program in which the given trace is partitioned into blocks and the rest of the program modified to maintain the original semantics.

Procedure:

(1) Identify the MI's which contain conditional branch MO's as block boundaries for the first part of the trace.

(2) For the last part of the trace, find rejoining points such that below the new rejoin there is no MO's which were above the old join.

(3) Do necessary copying of MO's for the first part of the trace.

(4) Do necessary copying of MO's for the last part of the trace.

The bookkeeping stage is divided into two major parts. The first part maps MI's into blocks; the second part makes appropriate copies of MO's. Each part handles the first half and the second half of the trace blocks separately.

Because of the way that the traces are picked, there is no junction block in any trace and all the traces in the beta compaction has certain forms. The traces are composed of three components, as shown in Figure 4.7. First, there are zero or more blocks which have conditional branch MO's at the end. Then there is one[7] block with one entry point and one exit point. Last, there are zero or more blocks which have a join at the beginning of them. Depending on a trace, some

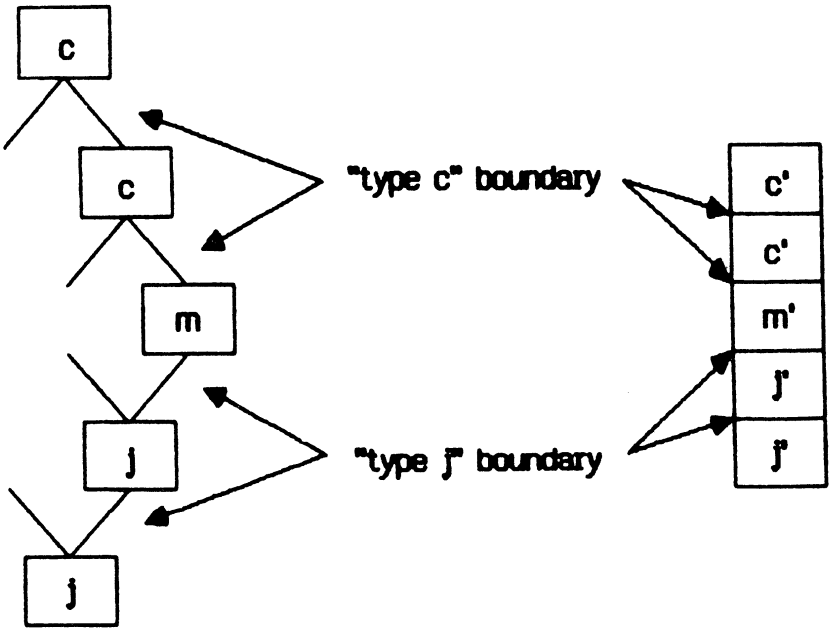


Figure 4.7 Components of trace

[7] If the dummy blocks are *not* inserted where a conditional branch and a join directly meet, this block may not exist sometimes.

of the above three components may be empty. However, the order of the components, when they exist, does not change.

The first part of the bookkeeping stage is to map compacted MI's into blocks. In other words, since the MO's may have been moved across the block boundaries during the list scheduling, we need to find out the new block boundaries. For the c-type boundaries shown in Figure 4.7, the job is very simple. The MI's which contain conditional branch MO's form natural block boundaries. For the j-type boundaries shown in Figure 4.7, the job is not as easy. We need to find new rejoining points as follows. Below the new rejoin, we want to have *no* MO's which were above the old join before the list scheduling. We want the earliest possible point which satisfies the above condition to be a new rejoining point.

The same order of the original joins are kept among the new joins except possibly two or more adjacent old joins become the same point when new rejoining points are found. This order-preserving property does not apply to c-type boundaries in general. It is possible that the order of two conditional branch MO's may change after the local compaction. We do not allow this to happen by drawing artificial DDG edges between conditional branch MO's before local compaction, because allowing a change of order of conditional branch MO's, we believe, would complicate the heuristic unnecessarily with very little or no gain. There is a recent study of some possibilities by allowing such change of order [LIN83].

The second part of the bookkeeping stage is to copy some of the MO's to off-the-trace blocks to keep the original semantics of the microprogram unchanged. The copying MO's is done again in two parts; first for c-type boundaries, then for j-type boundaries.

Let us first consider c-type boundaries. Recall that MO's are not allowed to move above conditional branches if they write any conditional source registers (live registers at the beginning of off-the-trace blocks). And MO's which are

moved above conditional branches either write no register or dead registers for the off-the-trace blocks. So such MO's are not candidates for copying. The MO's which need to be copied are the ones which were above conditional branches but are below them after the local compaction. These MO's need to be copied to the beginning of the off-the-trace blocks for each conditional branch which the MO's have moved below. However there is an exception, as shown in Figure 4.8. Suppose a MO has been moved from block A to block H as a result of the local compaction. The MO would have been copied to both block C and block E because block C and block E are off-the-trace blocks. But since the block E is in the same execution path with the block H, the MO needs not be copied to block E. So the MO is copied only to block C. To do this systematically, for each MO which needs to be copied, form a target block set (block C

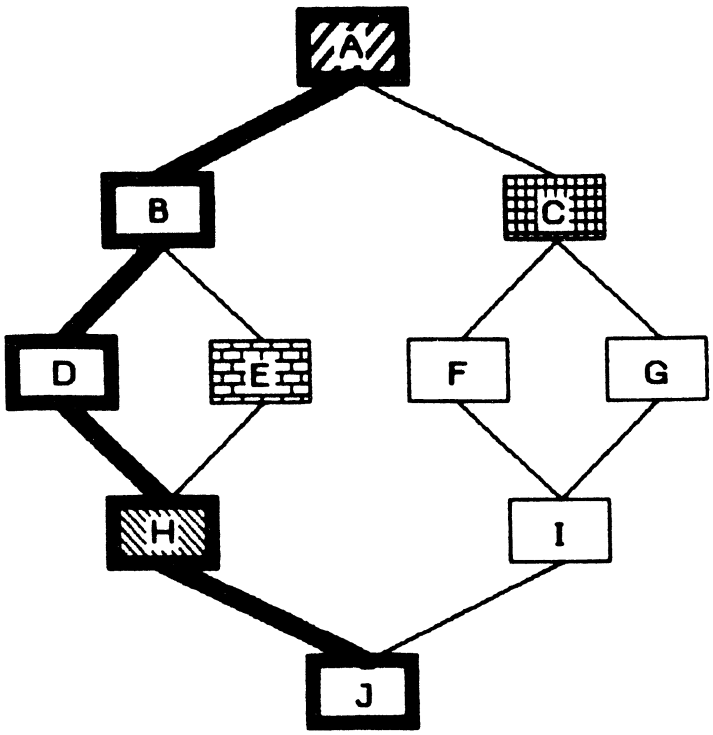


Figure 4.8 Copying MO's

and block E in the above example). Then subtract[8] all the blocks that can be backtraced starting from the block that the MO has been moved to (blocks H, D, E, B and A in the above example). Then copy the MO only to the resultant target block set, which is just the block C in the above example.

The copying of MO's for the j-type boundaries works much the same way as for the c-type boundaries.

4.2. Simple Illustrative Example

Here is an example of how the beta compaction algorithm works. Figure 4.9 is a microprogram with 21 MO's in 7 blocks. The letters A through G are assigned to each of the 7 blocks and the numbers 1 to 21 indicate MO's. Figure 4.10 shows a list of the MO's represented by the six-tuple model of Chapter II. MO 3 and MO 12 are conditional jumps. It is assumed that the probability of block B executing is 0.7 and that of block E is 0.6. Accordingly, the probability of block C executing is 0.3 and that of block F is 0.4. The probability of blocks A, D and G executing is all 1's, naturally.

The first step of the beta compaction algorithm is to perform live register analysis and the result is shown in Figure 4.11. The first trace blocks are A, B and D, and they are compacted together as if a single block. The compaction is done first by building a DDG with MO's in the blocks considered. When building DDG, the live registers at the beginning of block C or IN[C] are assumed to be read at MO 3, which is a conditional jump. In other words, registers in the set IN[C] are conditional source registers of MO 3. However, those registers are used only when considering data interaction between MO 3 and the MO's in block B, but not between MO 3 and other MO's of block A.

---

[8] set subtraction

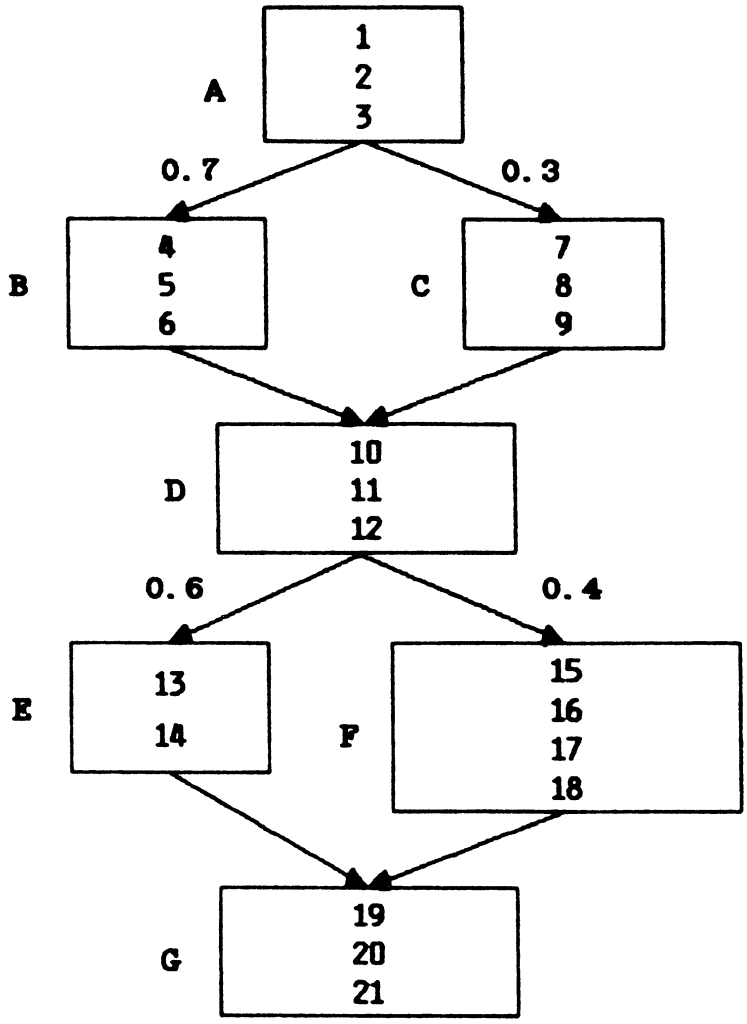


Figure 4.9 Sample microprogram

The DDG is shown in Figure 4.12(a). Note that, even though conditional source register 10 of MO 3 is also a destination register of MO 2, no edge is drawn between MO 2 and MO 3 to allow possible movement of MO 2 below MO 3. Also note that conditional source register 8 of MO 3 and destination register 8 of MO 4 makes an edge between MO 3 and MO 4 which prevents the movement of MO 4 above MO 3. Next, list scheduling is done on the MO's in the trace and the result is shown in Figure 4.13(a). Now, necessary bookkeeping is done on the

block	name	inst	next_addr	dest_reg	src_reg	resource_vec
	1	CONT		1	2, 3	0 1 1 0
A	2	CONT		10	1, 5	0 0 1 0
	3	CJMP	7	2	3, 4	1 0 1 0
	4	CONT		8	9, 5	0 1 0 0
B	5	CONT		2	2, 5	0 0 0 1
	6	GOTO	10	15	4, 8	0 1 0 0
	7	CONT		9	8, 8	0 1 0 1
C	8	CONT		1	1, 5	0 1 0 0
	9	CONT		8	11, 12	0 0 0 1
	10	CONT		14	2, 13	0 1 0 1
D	11	CONT		7	3, 5	0 1 0 0
	12	CJMP	15	6	4, 10	1 0 0 0
E	13	CONT		9	3, 4	1 0 1 0
	14	GOTO	19	6	3, 6	0 1 0 0
	15	CONT		2	6, 8	0 0 0 1
F	16	CONT		7	2, 14	0 1 0 1
	17	CONT		4	2, 5	0 1 0 0
	18	CONT		14	11, 12	0 0 1 0
	19	CONT		9	6, 10	0 0 0 1
G	20	CONT		13	7, 3	0 0 0 1
	21	STOP		3	4, 11	0 1 1 0

Figure 4.10 List of MO's

sequence of MI's. The point right after conditional branch MO 3 is the boundary between block A and block B. The rejoin from block C is made right before the last MI which contains no MO which were above the old join. Since MO 2 is scheduled below conditional jump MO 3, MO 2 is copied to block C. Note that MO 11 has been moved from block D to block A and is not copied since block A and block D are on the same execution path.

```
 inreg[A] = 2 3 4 5 8 9 11 12 13
outreg[A] = 1 2 3 4 5 8 9 10 11 12 13
 inreg[B] = 2 3 4 5 9 10 11 12 13
outreg[B] = 2 3 4 5 8 10 11 12 13
 inreg[C] = 1 2 3 4 5 8 10 11 12 13
outreg[C] = 2 3 4 5 8 10 11 12 13
 inreg[D] = 2 3 4 5 8 10 11 12 13
outreg[D] = 3 4 5 6 7 8 10 11 12 14
 inreg[E] = 3 4 6 7 10 11
outreg[E] = 3 4 6 7 10 11
 inreg[F] = 3 5 6 8 10 11 12 14
outreg[F] = 3 4 6 7 10 11
 inreg[G] = 3 4 6  7 10 11
outreg[G] =
```

Figure 4.11 Live registers

Similar compaction is done for the next trace of blocks D, E and G. The DDG is shown in Figure 4.12(b) and the result of list scheduling is shown in Figure 4.13(b). Since MO 20 is above the new rejoin from block F, the MO 20 is copied to the block F.

The third trace contains a single block C. It is compacted using straightforward list scheduling. The DDG is shown in Figure 4.12(c) and the result of list scheduling is shown in Figure 4.13(c). The last trace is also a single block which contains block F. The DDG is shown in Figure 4.12(d) and the result of list

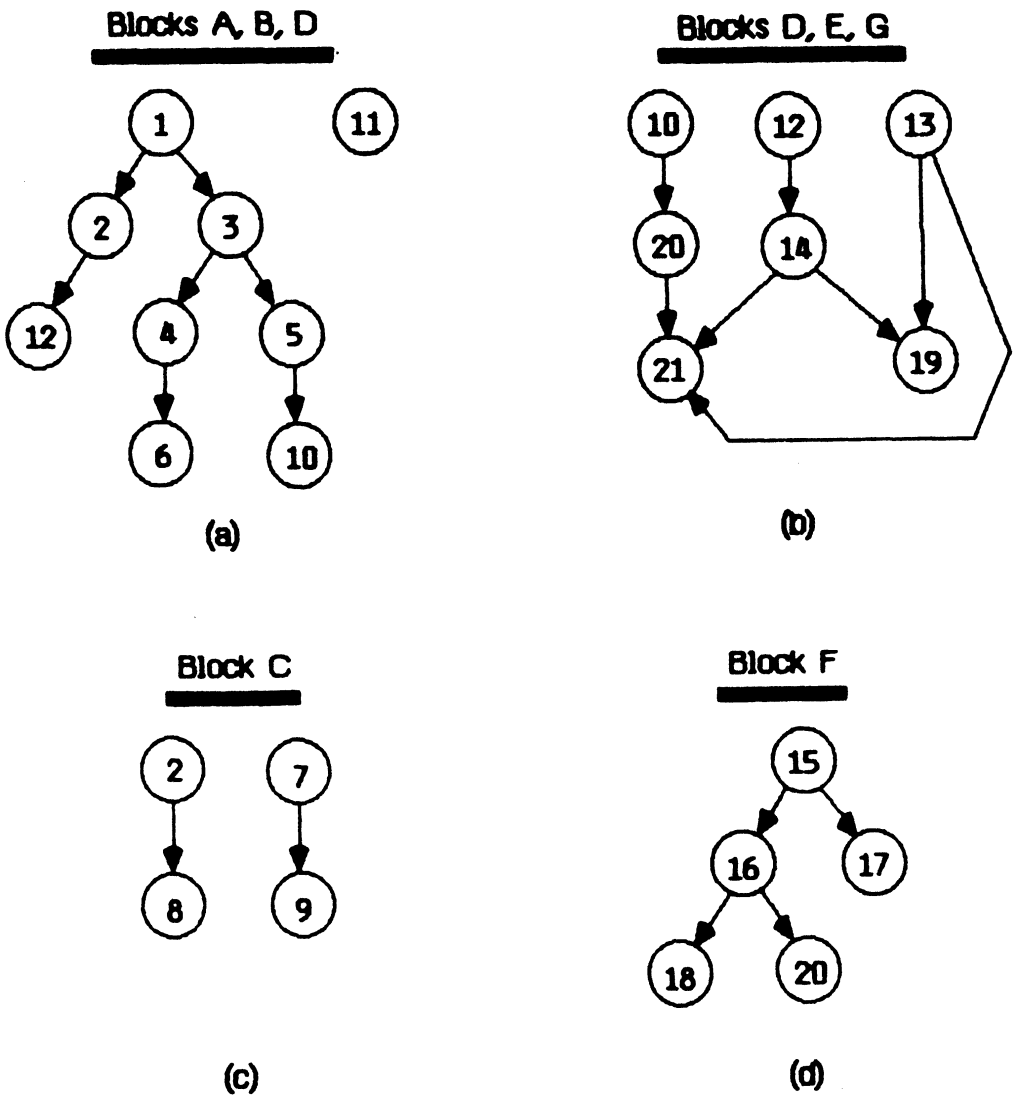


Figure 4.12 Data dependency graphs

scheduling is shown in Figure 4.13(d).

The final result of the beta compaction done on a given example microprogram is shown in Figure 4.14. Execution time of the trace blocks A, B, D, E, G is 7, and weighted execution time is 7.8. 12 MI's are used to hold the compacted microcode.

```
MO's in cycle 1 =  1                    block A'
MO's in cycle 2 =  3  11
MO's in cycle 3 =  2   4   5            block B'
MO's in cycle 4 =  6
MO's in cycle 5 = 10  12                block D'
```

(a)

```
MO's in cycle 1 =  10  12               block D'
MO's in cycle 2 =  13  14  20           block E'
MO's in cycle 3 =  19  21               block G'
```

(b)

```
MO's in cycle 1 =  2   7                block C'
MO's in cycle 2 =  8   9
```

(c)

```
MO's in cycle 1 =  15
MO's in cycle 2 =  16                   block F'
MO's in cycle 3 =  17  18  20
```

(d)

Figure 4.13 Intermediate results

The trace scheduling done on the same example is briefly described in Figure 4.15. Figure 4.15(a) shows the state after the list scheduling and bookkeeping on the first trace (blocks A, B, D, E, G) is done. Since MO 6 has been scheduled below conditional branch MO 12, the join from block C to block D had to be moved down with blocks D, E and F copied, as shown. The second trace of blocks C', D' and E' is compacted and some more copying is done, as shown in Figure 4.15(b). Next the remaining single blocks F' and F" are compacted. The final result is shown in Figure 4.15(c) and it is redrawn in Figure 4.15(d) in a more conventional form. Compare the result of trace scheduling in Figure 4.15(d)

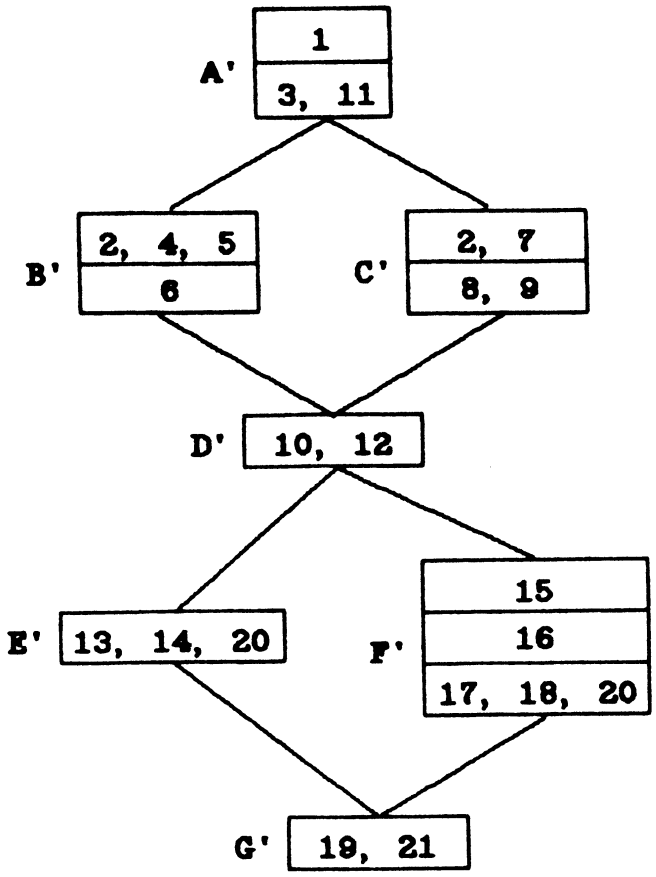
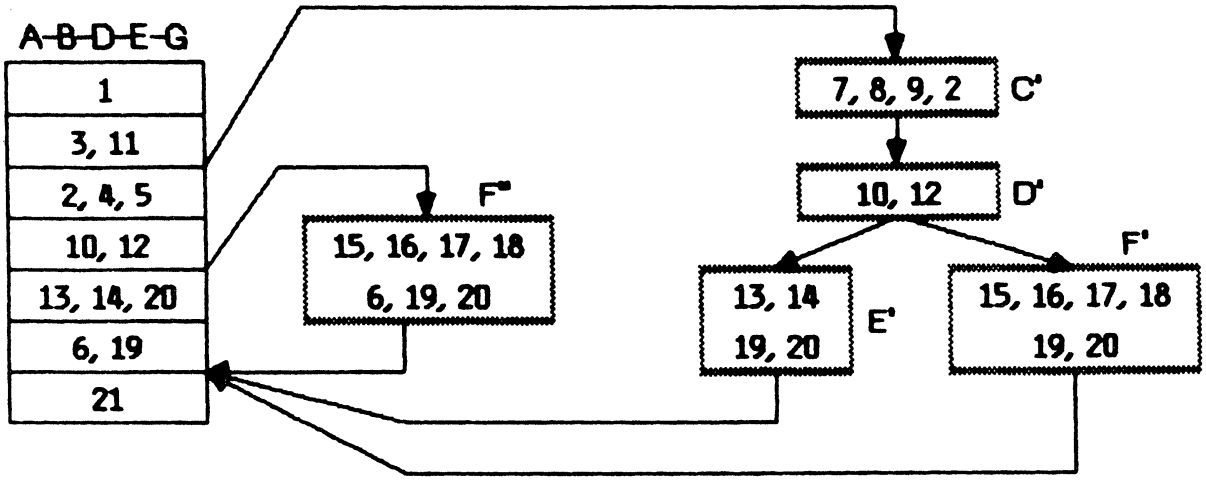


Figure 4.14 Final result

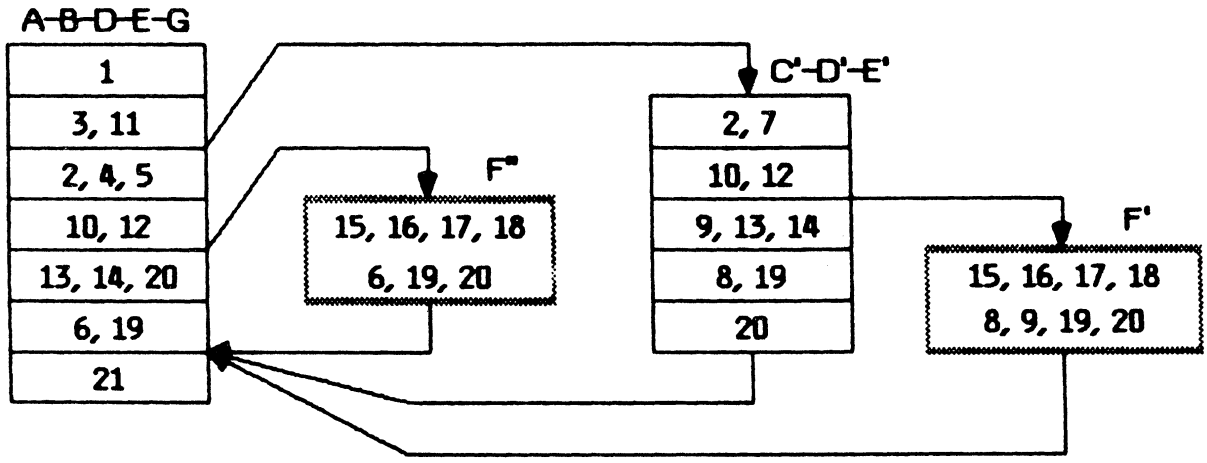
with the result of beta compaction in Figure 4.14 and note the change in program structure.

After the trace scheduling, the execution time of the trace blocks A, B, D, E, G is 7 and weighted execution time is 8.1, which is about the same[9] as the weighted execution time from the beta compaction. However, space used is 21 MI's, a 75% increase. The space time product of the trace scheduling result is

[9] In this particular example, it just happened that the weighted execution time from the beta compaction is slightly faster than the weighted execution time from the trace scheduling. Generally, we expect the trace scheduling gives slightly shorter weighted execution time than the beta compaction as will be shown in the next chapter.

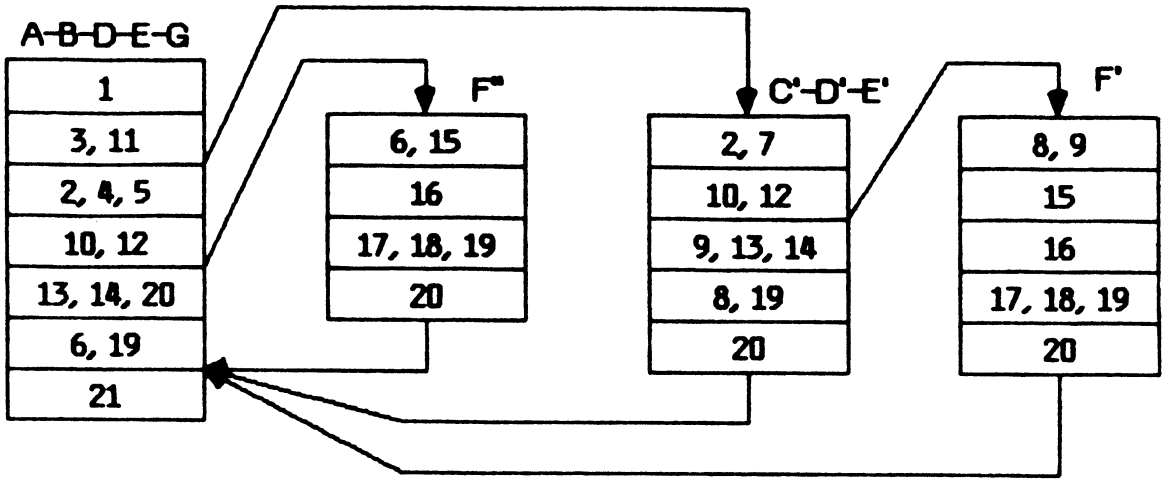


(a)

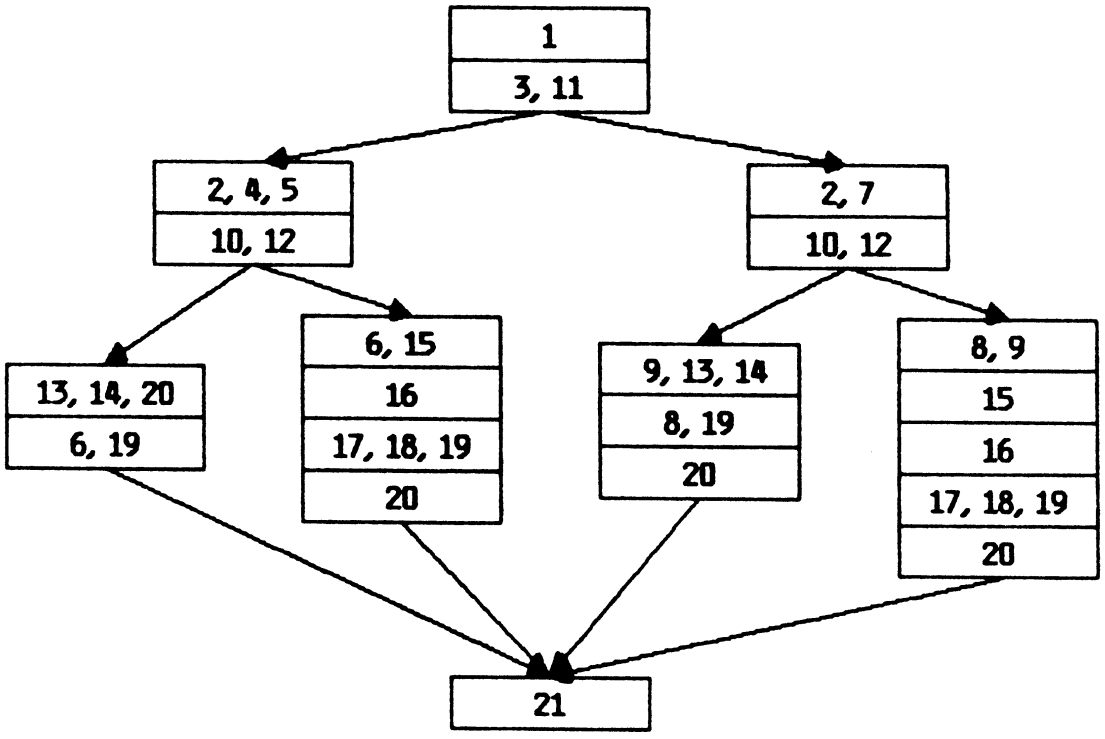


(b)

Figure 4.15 Trace scheduling on the example



(c)



(d)

Figure 4.15 Trace scheduling on the example (cont'd)

about 80% bigger than that of the tree compaction result. These are summarized in Figure 4.16.

This specific example shows the potential advantages of the beta compaction algorithm, i.e., it saves memory space with similar speedup in execution time. More analysis and experimentation are described in the next chapter.

	time in trace	weighted time	space needed	space time product
beta compaction	7	7.8	12	93.6
trace scheduling	7	8.1	21	170.1

Figure 4.16 Comparison summary

CHAPTER V

EXPERIMENTATION AND ANALYSIS

5.1. Purpose

A new technique to solve the global compaction problem has been developed. It is called beta compaction and is based on Fisher's trace scheduling. The purpose of this experimentation is to demonstrate its feasibility and practicality as well as to empirically verify its effectiveness.

For the experimentation, two separate approaches have been taken. First, some artificial microcodes are synthesized in a controlled environment by a microcode model, and are compacted using list scheduling, trace scheduling, and beta compaction. The purpose is to compare the performance of the beta compaction against that of the trace scheduling and the list scheduling. Second, an application program written in Pascal are hand-compiled into microcode and compacted using the same three methods. Both uncompacted and compacted microcodes are run on the software simulator for a two ALU AMD2900-based system. The purpose is to demonstrate that the beta compaction performs well in compacting real application programs as well as to informally show that the implementation of the three compaction methods is correct.

5.2. Hypothesis

Fisher's trace scheduling seems to be the best proposed heuristic so far for solving the global compaction problem in reducing the execution time of the compacted microcode. However, it has some serious drawbacks. First, the resultant microcode size can grow rapidly because of the extensive copying of blocks necessary, exponentially in the worst case. Second, the bookkeeping phase of the heuristic is very complicated and thus has been an obstacle to implementation. Third, the resultant microcode size can change significantly by a small change in source microcode.

The beta compaction is basically a careful limitation of trace scheduling. With this new heuristic, we claim to achieve almost the same level of execution time reduction, with much less memory required than the trace scheduling. The bookkeeping phase of the heuristic is very much simplified. Also, the size variation of the compacted microcode is much less sensitive to changes of the uncompacted source microcode.

Comparing against the local compaction technique (list scheduling), the beta compaction should perform better in both execution time and memory size.

5.3. Implementation

A loop-free version of both beta compaction and trace scheduling has been implemented. Implementation of loop handling has been omitted because it is not critical in comparing the two methods, since both methods handle loops the same way.

Standard Pascal was used to implement these global compaction heuristics on a VAX-11/780 running Unix 4.1bsd. Pascal was chosen over C to implement the compaction heuristics mainly because it is generally safer and it has built-in

set operations. However, it turned out that the set operations available in standard Pascal are so minimal that they were not as useful as expected. The most needed construct in dealing with sets was something like:

```
for all the elements of this set, do
        statement;
```

Since such a construct is not provided in standard Pascal, what we had to do was:

```
for all the elements of universal set, do
        if it is an element of this set
        then
                statement;
```

And it was not very efficient.

Implementation of list scheduling and beta compaction is very close to what was described in Chapter IV. The list scheduling is implemented as a part of the beta compaction. The source code size of the beta compaction is about 80k bytes, including some comments. Parts of the beta compaction has been rewritten to implement trace scheduling. Major differences are pick trace and bookkeeping routines.

5.4. Comparison against trace scheduling

Performance of any compaction method depends on so many variables that it is virtually impossible to accurately model them all. For example, the performance depends on the architecture that the microcode is compacted for, i.e., how much hardware resources are available, what kind of restrictions (timing, encoding) are imposed on using the resources, and what kind of parallelism is available

in the uncompacted source microcode, and how much of such parallelism is available over block boundaries, etc. Hence, we have developed an artificial microcode model where parameters are kept constant whenever possible or are otherwise random. This artificial microcode model is described in Section 5.4.1.

Since we are dealing with the global compaction problem, i.e., programs may contain several conditional jumps, joins, and/or loops, we need a way to represent program structure. In the absence of any available model to represent program structure, we have developed a simple program structure model to be used for our purpose of comparing two heuristics. We chose to restrict our program structure model to represent if-then-else structure only. There is no loop or multiway branching. This is sufficient for our purposes and at the same time easy to implement. The program structure model is described in Section 5.4.2.

5.4.1. Artificial Microcode Model

The purpose of this model of a microprogram is to empirically compare the performance of the beta compaction against the trace scheduling. It has been developed to make this comparison as fair as possible.

Each MO is represented by a six-tuple as described in chapter II.

(label, sequencer command, jump address, destination registers,
source registers, resource vector)

The purpose of the label is just to distinguish one MO from the others. The sequencer command and the jump address determine the block structure of microprograms. It is not a simple task to generate all possible configurations of microprograms. A somewhat simplified but still systematic way of generating block configurations is presented in the next section. The number of destination registers and the number of source registers are kept constant respectively and so

are the number of resources. Each block contains a constant number of MO's described above and the number of blocks, which is a constant, and the relationship among them is determined by the program structure model described in the next section.

The selection of register usage and resource usage is done randomly as follows. One register is selected randomly out of a available register set and assigned as a source or destination register. It is repeated as many times as there are source and destination registers. For resource selection, the probability of each resource being used in a MO is preset. Whether or not each resource is to be used is determined independently using the assigned probability.

It is presumed that by using the above random selection of registers and resources, the hard-to-grasp parameters like parallelism in microcodes and resource conflict are kept random. In the experiment, many number of microcodes are synthesized for each set of constant parameters and the results are averaged.

5.4.2. Program Structure Model

In this model, we have made several assumptions:

(1) only two way branching (if then else) is allowed
(2) only two way join is allowed
(3) no loop
(4) no goto

Multi-way join can be achieved by allowing some blocks to be empty but we are not concerned about the size of blocks here. We are only interested in the topology of block structure, which is represented as a graph where a node is a block and an edge is a possible path of program execution.

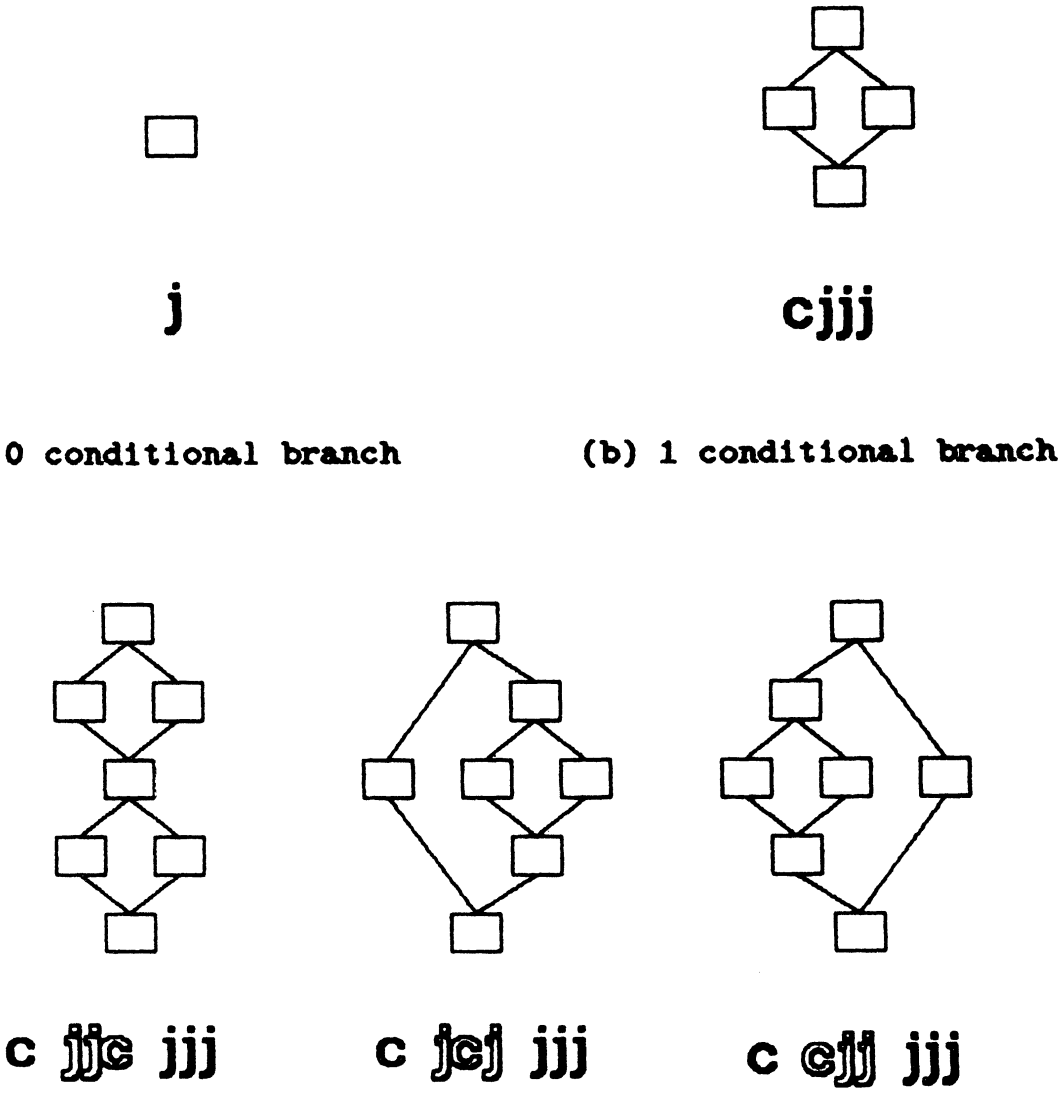


Figure 5.1 Program structure model

First, consider a case where there is no conditional branch. There is only one possible configuration in this case, which is just a single block or a node in graph representation. It is shown in Figure 5.1(a). If there is one conditional branch, the possible configuration is still unique, with 4 blocks, and is shown in

Figure 5.1(b). If there are two conditional branches, the number of possible configurations becomes three, as shown in Figure 5.1(c).

There is another way of representing these configurations by using strings. In the string representation, each letter corresponds to a block. The blocks which end with conditional branches are represented as letter **c** and the other blocks are represented as letter **j**. The order of these letters is similar to that of the preorder traversal of a tree. String representations are shown below each graph representation in Figure 5.1.

If we see the string representation in Figure 5.1(c), we can see a pattern. The first letter is always **c** and the last three letters are always **j**'s. The location of the **c** in the middle three letters determines three different configurations. Note that for each **c** in a string, there cannot be more than two **j**'s until the next **c** comes except the last **c** in the string. This rule is accumulative, i.e., if there

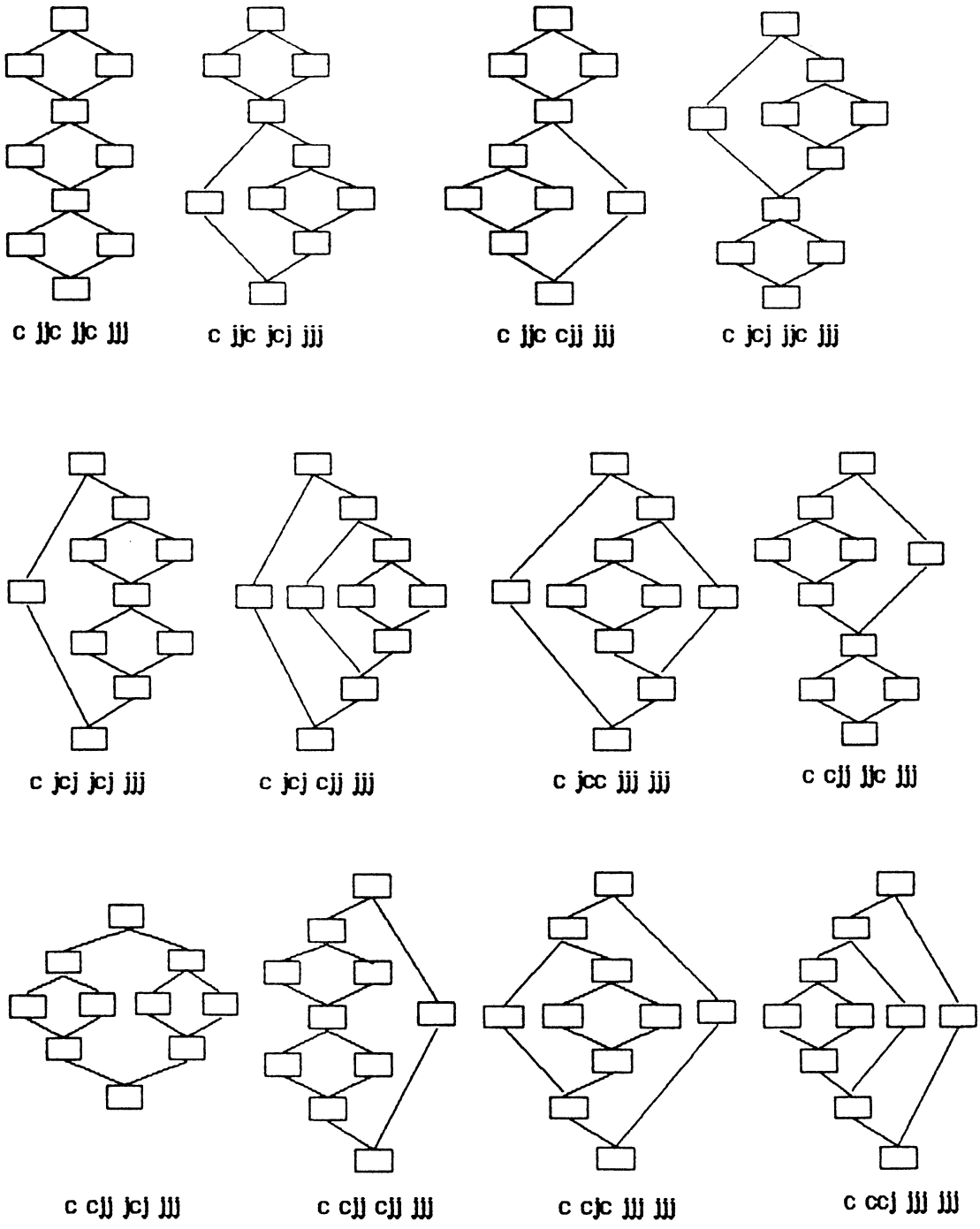
---

c jjc jjc jjj	c jcj jjc jjj	c cjj jjc jjj
c jjc jcj jjj	c jcj jcj jjj	c cjj jcj jjj
c jjc cjj jjj	c jcj cjj jjj	c cjj cjj jjj
	c jcc jjj jjj	c cjc jjj jjj
		c ccj jjj jjj

(a) string representations of 12 possible configurations

Figure 5.2 Program structure model for 3 conditional branches

---



(b) graph representation

Figure 5.2 Program structure model for 3 conditional branches (cont'd)

are two consecutive **c**'s, then four **j**'s may follow before another **c**. Described differently, for any letter in a string except the last **j**, if its prefix contains less than twice as many **j**'s as **c**'s, the letter can be a either **c** or **j**. If its prefix contains exactly twice as many **j**'s as **c**'s, the letter has to be a **c**. If its prefix contains more than twice as many **j**'s as **c**'s, then the prefix contains a letter **j** which represents a premature exit block.

Now we can represent all the configurations for the case of three conditional branches using the above rules, which is shown in Figure 5.2(a). For each string representation, corresponding graph representation is shown in Figure 5.2(b).

5.4.3. Procedure

The experiment was done using the following parameters:

The number of registers: 16
The number of destination registers: 1
The number of source registers: 2
The number of resources: 4
The probability of each resource used: .25
The number of blocks: 10
The number of MO's in each block: 3

The above parameters are chosen based on the system simulated in Section 5.5.1 and the program structure model in Figure 5.2. For each MO, one register among 16 is selected randomly and assigned as a destination register and two (possibly the same) registers are selected and assigned as source registers. For each MO, it is assumed that the probability of each resource being used is 1/4. So it is possible that all four resources are used in a particular MO with the probability of 1/256, or no resource is used again with the probability of 1/256. With 10 blocks, there exist 12 possible configurations as described in previous

```
c     1   1   2  15     1  0  0  0  1.000000
c     2   2   4  12     0  1  0  0  1.000000
j     7  12  11  10     0  0  1  0  1.000000
c     4   7  13   5     0  0  1  0  0.800000
c     5   7   6   4     0  0  1  0  0.800000
g    10   8   3   8     0  0  0  1  0.800000
c     7   9  10  10     0  1  0  0  0.200000
c     8  14   5   7     1  0  0  0  0.200000
c     9  12  10   5     0  1  0  0  0.200000
c    10  11   2   4     0  0  0  1  1.000000
c    11  10   3   0     0  0  1  0  1.000000
j    16   5   1  13     0  0  0  1  1.000000
c    13   5  13   5     0  0  1  0  0.800000
c    14   0   8  14     1  0  0  0  0.800000
g    28   6  11   4     1  0  0  0  0.800000
c    16   3   9  11     0  0  0  1  0.200000
c    17   5   0   0     0  0  1  0  0.200000
j    22  15  14   4     1  0  0  0  0.200000
c    19  13   2   3     0  1  0  0  0.160000
c    20   4   2  12     0  1  0  0  0.160000
g    25   5   2  11     0  0  1  0  0.160000
c    22  15   3  14     0  0  1  0  0.040000
c    23   7   7  13     0  1  0  0  0.040000
c    24  11  12  12     0  0  0  1  0.040000
c    25   8   3  10     0  0  0  1  0.200000
c    26   2   8   9     0  0  1  0  0.200000
c    27  13   8   7     0  0  0  1  0.200000
c    28  10   7   8     1  0  0  0  1.000000
c    29  13  10   9     0  0  0  1  1.000000
c    30   4   5  13     0  0  1  0  1.000000
```

Figure 5.3 Example of an artificially synthesized microcode

section. For each configuration, 20 microprograms are synthesized and compacted using list scheduling, beta compaction, and trace scheduling.

An example of microcode synthesized using the models described in the previous two sections is shown in Figure 5.3.

5.4.4. Result and Discussion

The result of the experiment described above is summarized in Figure 5.4 in terms of number of junction blocks. We are particularly interested in the performance of the beta compaction compared to trace scheduling as the number of junction blocks varies, since handling of the junction block is the key in beta compaction. Figure 5.4(a) shows weighted execution time of compacted microcode and Figure 5.4(b) shows control memory size of compacted microcode. Since there are 5 configurations with no junction block, 6 configurations with one junction block, and 1 configuration with two junction blocks, as shown in Figure 5.2(b), the numbers in Figure 5.4 for the zero junction block case represent the average of 100 microprograms, the numbers for the one junction block case represent the average of 120 microprograms, and the number for the two junction block case represents the average of 20 microprograms.

As shown in Figure 5.4(a), the weighted execution times of the beta compaction result and the trace scheduling result are almost the same regardless of the number of junction blocks. The weighted execution times of the global compaction result (beta compaction and trace scheduling) are about half of original weighted execution time and about 25% less than the local compaction result (list scheduling).

The memory sizes of the beta compaction result and the trace scheduling result are almost the same for the zero junction block case. However, the memory sizes of the beta compaction result are about 15% less than the memory sizes of the trace scheduling result for the one and two junction block cases. The memory size difference between local and global compaction results is not as large as the weighted execution time difference between local and global compaction results. Note that the memory sizes of the trace scheduling result for one and two junction block cases are actually larger than the memory size of the list scheduling result.

	number of junction blocks		
	0	1	2
original	22.6		
list scheduling	15.6		
beta compaction	10.2	11.9	13.0
trace scheduling	10.1	11.7	12.8

(a) weighted execution time

	number of junction blocks		
	0	1	2
original	30		
list scheduling	21.3		
beta compaction	20.4	19.8	20.4
trace scheduling	20.5	22.8	23.9

(b) control memory size

Figure 5.4 Summary of experimental result

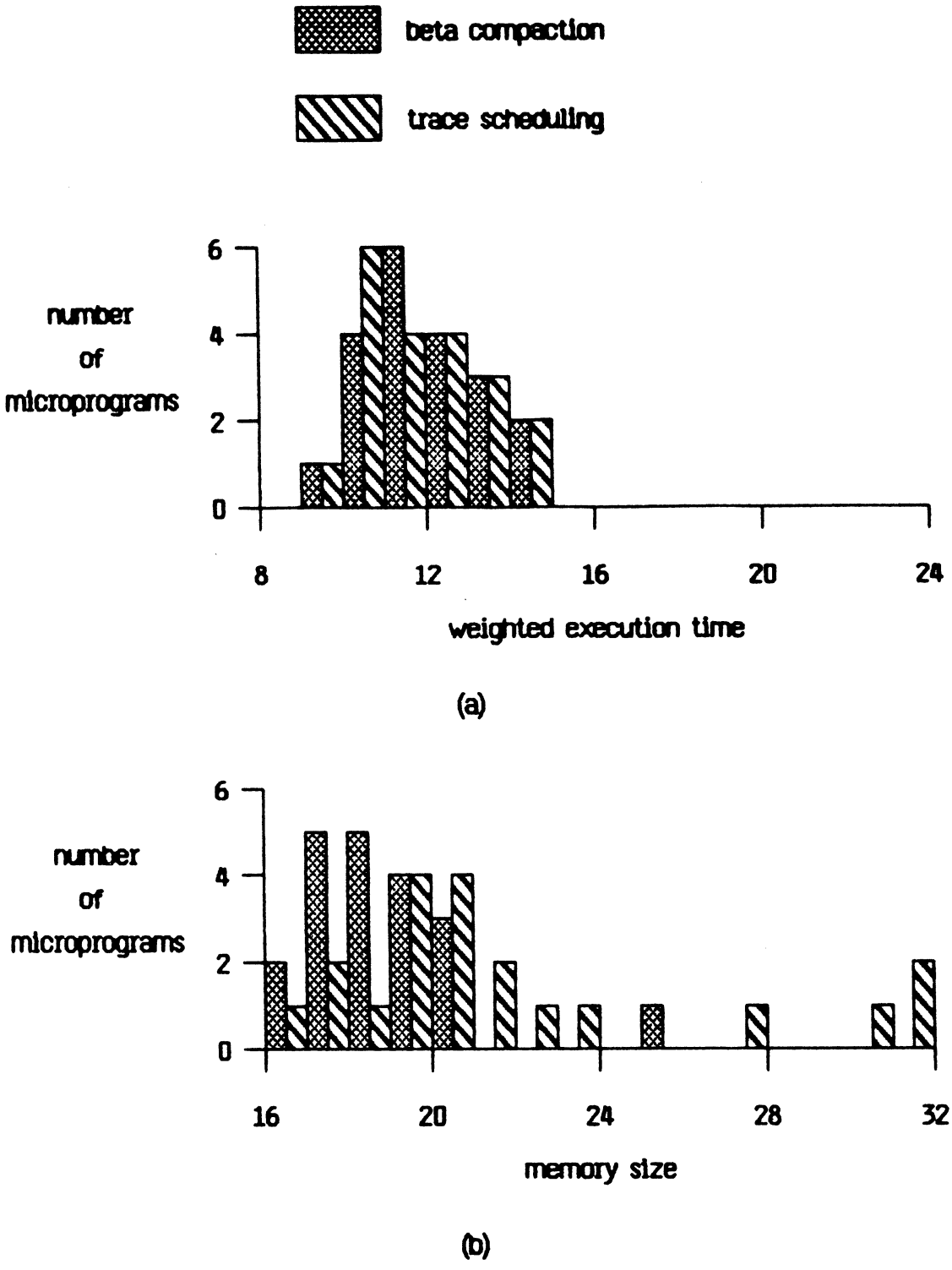


Figure 5.5 Distribution of the two junction block case

The distribution of 20 microprograms for the two junction block case is shown in Figure 5.5. Figure 5.5(a) shows the distribution for weighted execution time and Figure 5.5(b) shows the distribution for memory size. In the distribution for execution time, a bar of height N above execution time T is interpreted to mean that N microprograms out of the 20 tested had weighted execution time T. The interpretation is similar for the memory size distribution. It is clear that the distribution of weighted execution time is almost the same for the beta compaction and the trace scheduling, as shown in Figure 5.5(a). However, the distribution of memory size is quite different between the two. While most microprograms yielded the memory size of between 16 and 20 in the beta compaction, the trace scheduling result showed a wide distribution, ranging from 16 to 31. This implies the high sensitivity of memory size in the trace scheduling for the changes of source microcode.

5.5. Simulator of an AMD2900-based system

The previous section described the comparison between the trace scheduling and the beta compaction using synthesized microcodes and confirmed the claimed properties of the beta compaction. However, we are interested in the performance of the beta compaction not only on the synthesized microcodes but also on some real application programs. Since we do not have access to a microprogrammable machine with parallel resources, we decided to simulate one. Microprogrammable hardware based on AMD 2900 components is designed and simulated to test both uncompacted and compacted microcodes. The simulator is based on Glenford Myers' paper[MYE81] and has most of the nice menu driven human interface described in the paper and more. The simulation is done at the register transfer level which is seen by a microprogrammer. The simulator is written using C and the *curses* screen drawing package on VAX-11/780 running UNIX 4.1bsd.

5.5.1. System description

The block diagram of the simulated system is shown in Figure 5.6. It is a simple system using the AM2903 ALU and the AM2910 microprogram controller.

The system has two ALU's (AM2903) to provide parallel resources for compacted microcodes. In the experment, an application program is hand-compiled for the target system assuming there is just one ALU, then compacted for the target system with two ALU's.

To make the register file accessible from both ALU's, the internal registers of AMD2903 are ignored and external registers are used. Each ALU has its own register file and they always have the same value in order to allow parallel access to the registers. So up to four registers can be read by two ALU's at the same time, addressed by a and b addresses of each ALU. Any register write operation will write both register files the same value. Since b address is used for register write operation the proper b address among two is selected in the hardware and fed to the register files. All the non-ALU registers including pipeline registers and status registers are latched at the rising edge of each clock or at the beginning of each clock cycle. Therefore, micro memory fetch, main memory fetch and ALU operation are done at the same cycle.

Since CPU and main memory are the only things that are on the system bus, no bus arbitration is necessary, except the DOUT register should not be writing onto the data bus in memory read cycle. Main memory is assumed to be fast enough to allow access in a single microcycle. However, the wait states can be inserted if necessary.

When two ALU's are utilized, some values to be loaded in non-ALU registers may be lost by other operation before they are actually loaded. For example, an address calculated by ALU 0 on its y bus may be changed by the next ALU 1 operation before it is loaded into memory address register (MAR), since MAR load operation is not guaranteed to follow the address calculation immediately.

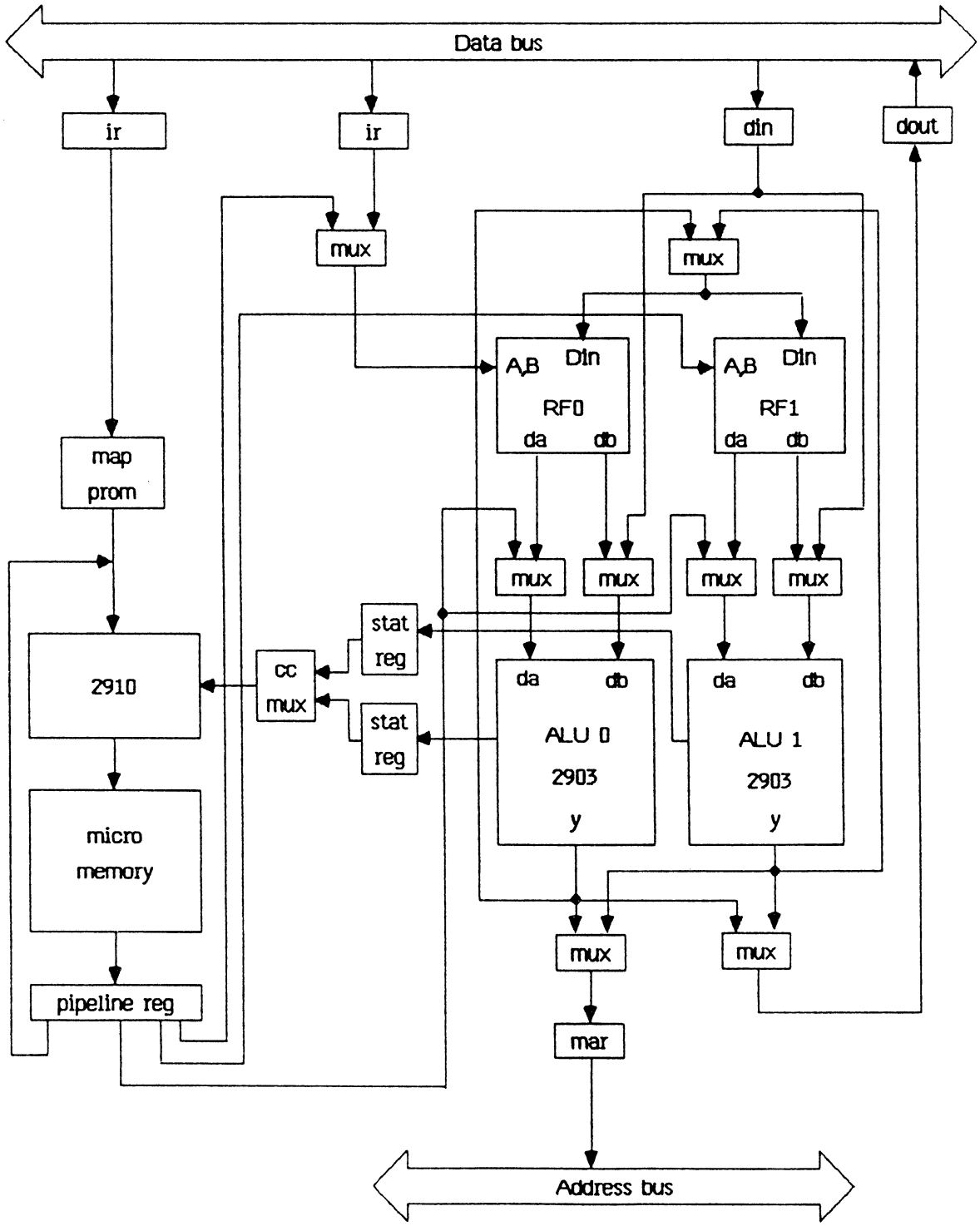


Figure 5.6 Block diagram of the simulated system

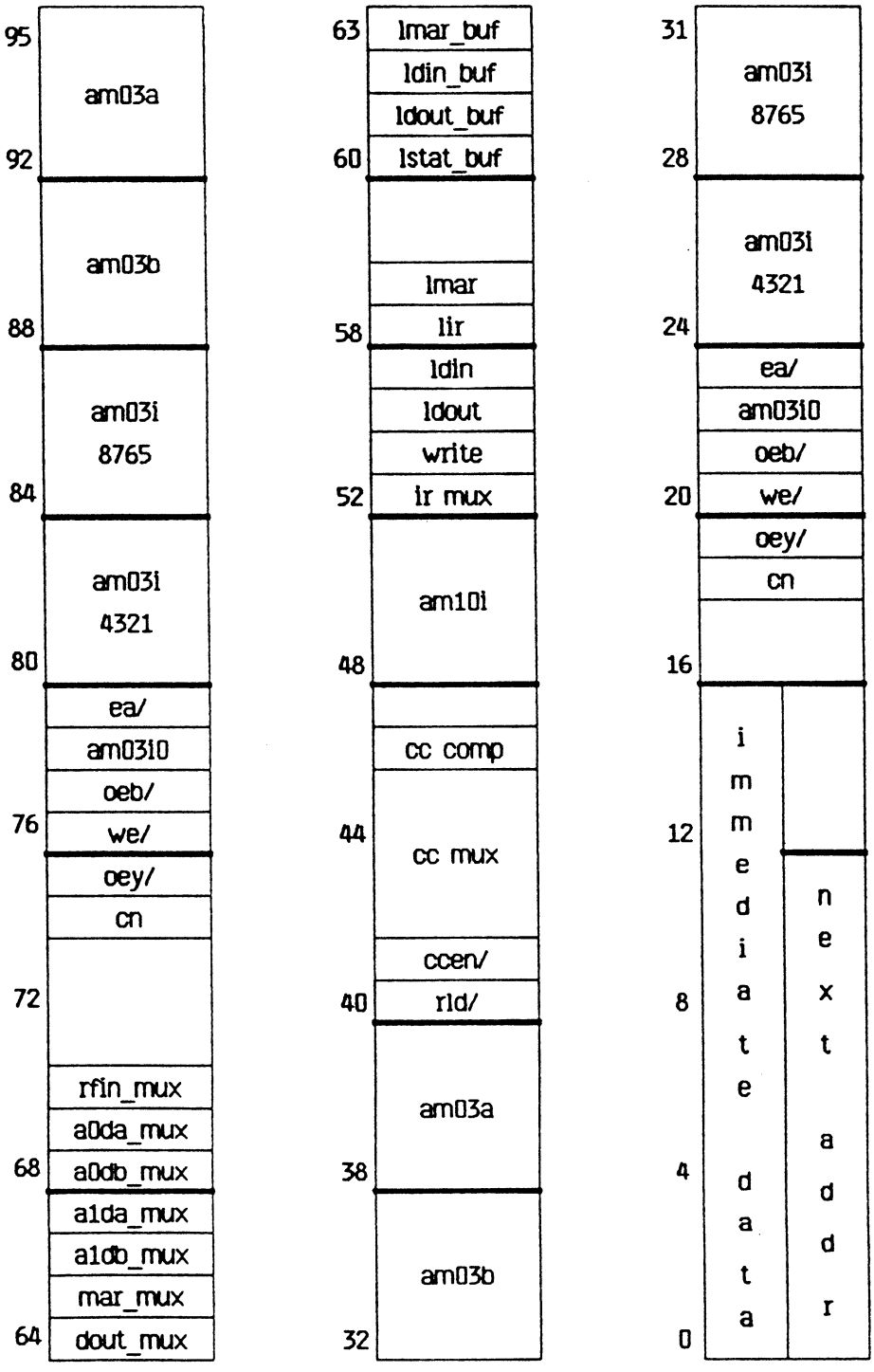


Figure 5.7 Microinstruction format

Special buffers are built in to mar, din, dout and status registers. These buffers are loaded at the end of micro cycle controlled by pipeline register signals synchronized with a sub-micro phase clock for the purpose. These buffers will hold correct values until they are loaded into corresponding registers.

The micro instruction format is shown in Figure 5.7. The width of the MI is 96 bits (3 double words). For one ALU application, only 2 double words (64 bits) are necessary. Note that the selection between register and external data for ALU input is controlled ordinarily using ea/ and oeb/ signals when internal registers are used, but for this simulator the selection is controlled externally using da_mux and db_mux when external registers are used.

5.5.2. Functions and human interface

The simulator is an interactive menu driven software package and all its functions are controlled by selecting appropriate entries in its menu window. All the internal states are displayed and can be modified. Control store can be loaded from a file and saved to a file. The user program can be executed either in a free run mode of in a single step mode. Break points can be set and the trace is collected. There are many more useful functions and human interface features and Appendix A contains a detailed description.

5.5.3. An Application Program

The insert operation to 2-3 tree was chosen to be used as an example program to be run on the simulator because it contains many decision makings, i.e. conditional branches. It is hypothesized that programs with many conditional branches, i.e. many blocks have more parallelism over the block boundaries which can be exploited in global compaction. The more conditional branches, the more chances are there for global compaction.

The program was written in Pascal and hand compiled into microcode. The Pascal program source listing is attached in Appendix B and a part of AMDASM[1] listing of hand compiled microcode is in Appendix C. During the hand compilation, all the variables were preassigned to registers, since the number of variables was small. The Pascal program of the insertion algorithm is written using a recursive procedure. So the recursive procedure is implemented on the microcode using a simple stack in main memory to save and restore local variables. Argument passing on procedure calls is done using call by reference, so the same registers are used to represent argument variables inside procedures. The program builds its 2-3 tree data structure in main memory. A simple get-space routine is written in microcode to allocate the main memory space. The program was first compiled into microcode for the simulated system assuming there is one ALU and then compacted for two ALU system using list scheduling and beta compaction. Since the compaction program requires data format different from the microcode generated by AMDASM, a conversion program was written which finds out all the register and resource usage of each MI. After compaction, the result was converted to a format acceptable to the simulator by another conversion program which also calculates new branch addresses and determines multiplexer controls.

To demonstrate that the compaction procedure has generated correct microcode, both uncompacted and compacted versions of the program has been run on the simulator and give the same result as the compiled object code of the original Pascal program on VAX-11/780.

For a data set which builds a 2-3 tree with 5 leaf nodes, weighted execution time and space needed are summarized in Figure 5.8 for both before and after compaction. The probabilities for calculating weighted execution time were measured from a pascal program execution profile on a large data set. Even though the result confirmed the general properties of three compaction techniques, the difference between beta compaction and the trace scheduling is small. The main reason is that the program has little parallelism over the block

[1] AMDASM is a trademark of Advanced Micro Device, Inc.

	weighted time	space (# of words)	space (# of bits)
original	75.15	238	14042
list scheduling	43.86	159	13356
trace scheduling	40.77	155	13020
beta compaction	40.83	150	12600

Figure 5.8 Summary of 2-3 tree insertion program execution

boundaries because main memory read and write operations impose very heavy local data interactions between MO's and therefore very small number of MO's can be moved to different blocks without violating data integrity. It would be possible to relieve some of these data interactions due to main memory access by having dual port memory with separate address and data bus.

5.6. Space Complexity Analysis

Space complexity of compacted microcode size is analyzed for both trace scheduling and beta compaction. For the trace scheduling, one microprogram is analyzed to show that the memory size can grow exponentially after the compaction. For the beta compaction, the worst case microprogram is presented and

informally proved. The worst case space complexity of the beta compaction is $O(n^2)$.

5.6.1. Trace Scheduling

One microprogram shown in Figure 5.9 is analyzed to demonstrate that the memory requirement of trace scheduling can grow exponentially.

For the purpose of the analysis, it is assumed that each block contains a large unspecified but equal number of MO's and the blocks $A_1, B_1, A_2, B_2, \cdots A_n, B_n, A_0$ form a trace. Furthermore, it is assumed that, after the trace blocks are compacted, one MO from each blocks $B_1, B_2, \cdots, B_n$ has been combined with MO's in block A_0 so that it is necessary to make copies of blocks as shown in Figure 5.9(b). Since it is assumed that the number of MO's in each block is equal and larger than one, and the only over-the-block-boundary movements of MO's are the ones from blocks $B_1, B_2, \cdots, B_n$ to A_0, it is reasonable to assume that the size of each block does not change. Let the size of each block be unity for convenience.

Now the rest of the microprogram except the first trace has to be compacted and it is assumed again that similar situations arise as the first trace and copies are made.

Let f(n) be the size of the microprogram shown in Figure 5.9(a) after compaction in terms of number of blocks. n is the number of conditional branches.

$$\begin{aligned} f(n) &= 2n + 1 + f(n-1) + 1 + f(n-2) + 1 + \cdots + f(0) + 1 \\ &= 3n + 1 + f(n-1) + f(n-2) + \cdots + f(0) \end{aligned}$$

It is obvious that $f(0) = 1$.

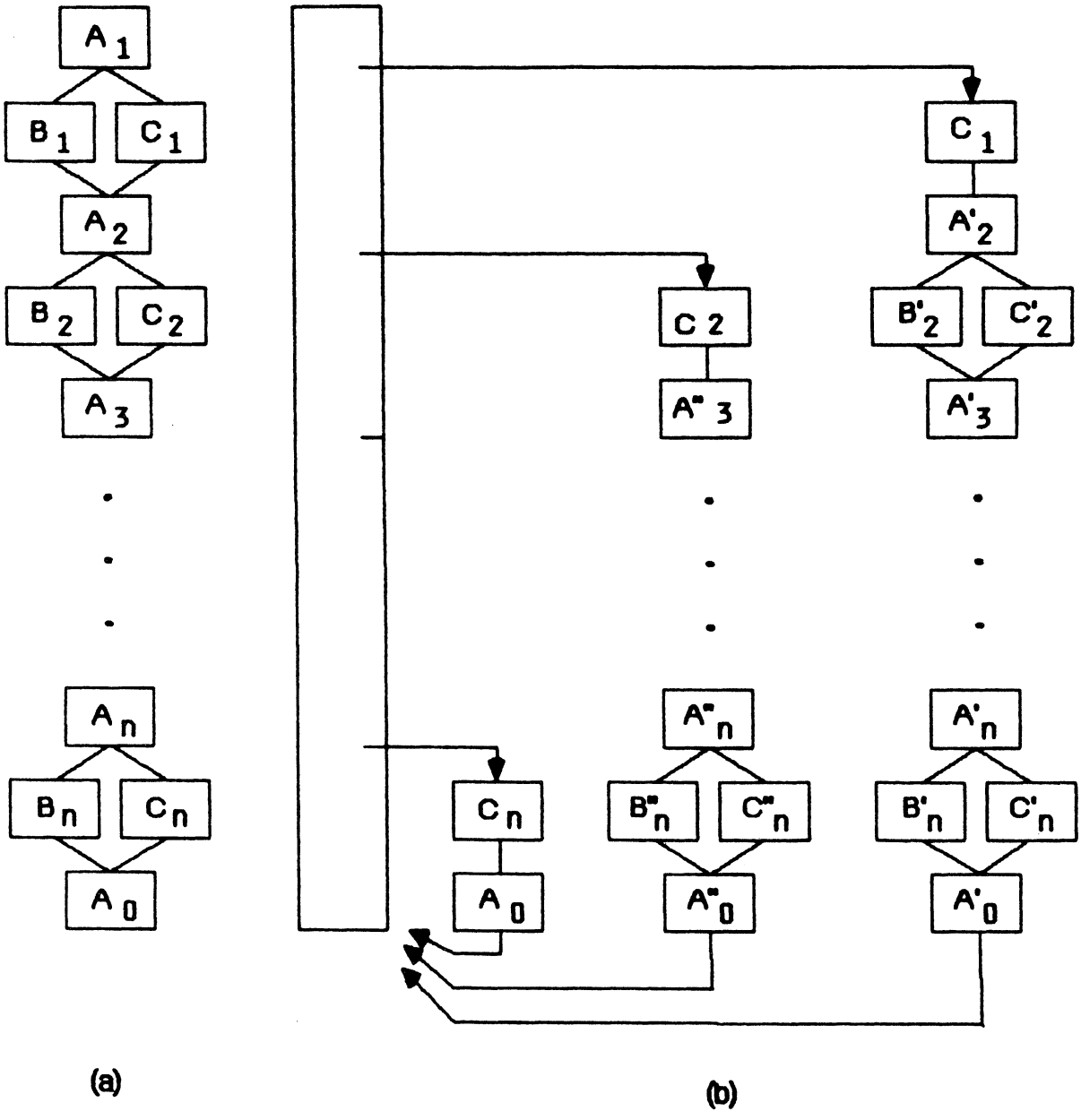


Figure 5.9 Exponential memory growth in trace scheduling

$$f(n) = 3n + 1 + f(n-1) + f(n-2) + \cdots + f(0)$$

$$-) \; f(n-1) = 3(n-1) + 1 + \quad + f(n-2) + \cdots + f(0)$$

$$f(n) - f(n-1) = f(n-1) + 3$$

Therefore,

$$f(n) = 2f(n-1) + 3$$

This and $f(0) = 1$ give

$$f(n) = 2^{n+2} - 3$$

i.e., f(n) is the exponential function of n, or the number of conditional branches in the microprogram shown in Figure 5.9(a).

This is of course a pathological case which is not likely to happen in real microcode. However, the implication that movements of MO's below joins cause extensive copying of blocks is serious. In real microcode, it is possible that movements of MO's below joins happen frequently enough to increase the memory size forbiddingly. Note that a single movement of an MO from block B_1 to A_0 would approximately double the memory size.

5.6.2. Beta Compaction

In beta compaction, the worst case in memory space growth happens when a given microprogram has the structure shown in Figure 5.10(a). It is assumed that in any conditional branch, the left branch has higher probability of execution than the right branch for convenience. An informal proof by induction that this microprogram gives the worst space complexity follows.

Starting from a single block or a SLM, adding one conditional branch to it yields 4 blocks A_0, A_1, B_1 and A'_0 - the only case, thus the worst case. Since we are concerned only about the topology of the microprogram, the size of each

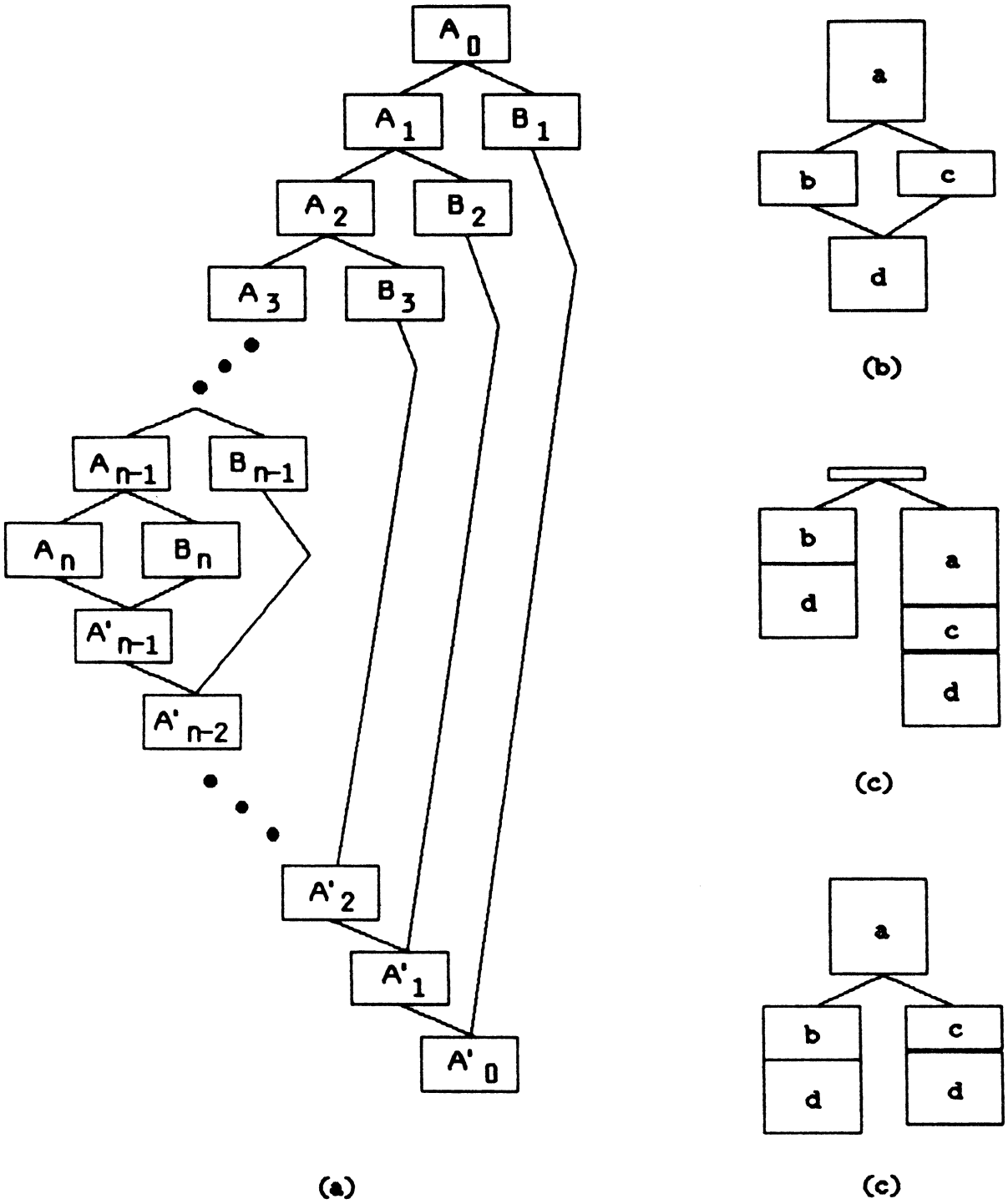


Figure 5.10 Worst case in beta compaction

block is assumed to be unity and blocks newly created by adding a conditional branch are still assumed to be of size unity. Let us assume that the set of blocks shown in Figure 5.10(a) except the 3 blocks A_n, B_n, A'_{n-1}, with block A_{n-1} connected to block A'_{n-2}, gives the worst space complexity with respect to n-1 conditional branches.

At this point, let us consider a microprogram shown in Figure 5.10(b). Given such a program, the beta compaction result which would give the largest memory is shown in Figure 5.10(c). First, the trace of blocks a, b and d was compacted. Since we are trying to maximize the resultant memory size, we want all MO's in block a to be copied to block c, which may be caused by the movement of MO's in block a into block b. The MO's moved to block b do not increase the size of block b, however, because they are moved only when they are combined with the MO's in block b to form new MI's. Therefore, the attempt to make MO's in block a copied to block c did not increase the overall size. It is assumed that a MO from block b has been moved to the bottom of block d as a result of compaction of the trace. So MO's in block d is copied to the end of block c, as shown in Figure 5.10(c). Figure 5.10(d) shows a compacted microcode without copying of MO's in block a and its size is the same as the size of the microcode in Figure 5.10(c). So we will use the kind of transformation from the microcode of Figure 5.10(a) to that of Figure 5.10(d) in our discussion.

Now we need to find out to which block we should add the n th conditional branch in order to maximize the resultant microcode size. We claim that adding one more conditional branch to either block A_{n-1} or B_{n-1} increases the space the most. If the conditional branch is added to one of the blocks A_0, A_1, ..., A_{n-2} or A'_{n-2}, A'_{n-3}, ..., A'_0, then a junction block is introduced and so the traces become shorter, therefore reducing space increase. If the conditional branch is added to one of the blocks B_k, $1 \leq k \leq n-1$, then the increased size of the microprogram is $4+2k-k$ which becomes a maximum of $n+3$ when $k = n-1$. Or, if a conditional branch is added to block A_{n-1}, then the increased size of the microprogram is

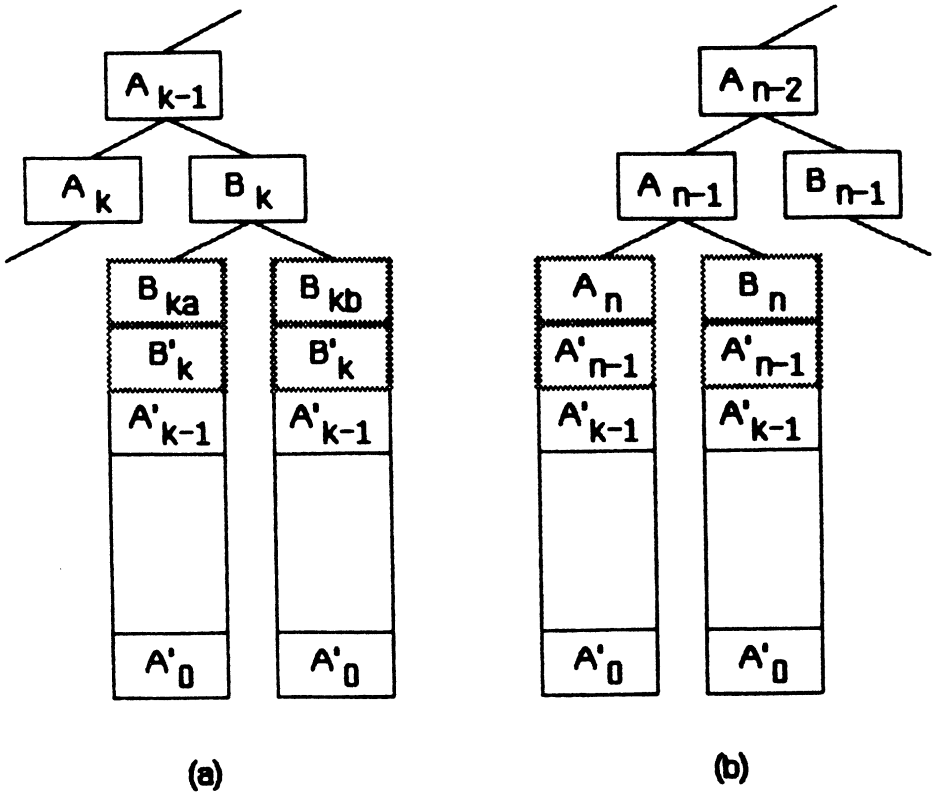


Figure 5.11 Calculation of memory size increase

n+3, which is the same as for block B_{n-1}.

To calculate these size increases, refer to Figure 5.11. It shows the case where the n th conditional branch is added to one of the blocks B_k, $1 \leq k \leq n-1$. Similar to what is shown in Figure 5.10(d), four blocks are added, and the blocks which follow block B_k are also attached at the end of each branch. So the total size increase is 4+2k minus k, because k blocks were there before the conditional branch is added. Similarly, the total size increase when the n th conditional branch is added to block A_{n-1} is 4+2(n-1)-(n-1), as shown in Figure 5.11(b). So

the size increase is maximum when the conditional branch is added to either block A_{n-1} or block B_{n-1}. Therefore, a microprogram with the topology shown in Figure 5.10(a), where the n th conditional branch is added to block A_{n-1}, requires the largest possible memory space after the tree compaction is performed. If the conditional branch is added to block B_{n-1}, the topology of the microprogram is different, but the size increase is identical.

The size of the whole microcode after the compaction is

$$S = 1 + 2n + n + \sum_{k=1}^{n} k$$

$$= 3n + 1 + \frac{n(n+1)}{2}$$

$$= O(n^2)$$

CHAPTER VI

CONCLUSION

6.1. Summary

The need for a better microprogramming tool has increased considerably as increased demand and support of computer technology has brought about wide use of microprograms. The eventual goal of microprogramming tool development would be to make a high level microprogram language and a compiler to generate minimal-execution-time microcode for a variety of machines. In generating minimal-execution-time microcode, one aspect that differentiates microprogramming languages from macroprogramming languages is the need for compaction in highly horizontal microarchitecture.

Among the proposed microprogram compaction methods, the trace scheduling is the most general and appears to give the fastest execution of compacted microcode. However, the growth of memory size by extensive copying of blocks can be enormous, exponential in the worst case, and the complicated bookkeeping stage of the trace scheduling has been an obstacle to implementation.

A technique called beta compaction, based on trace scheduling, is proposed to mitigate the drawbacks of trace scheduling. Basically, it identifies the junction blocks (the blocks beginning with a join and ending with a conditional branch) as the major source of complication, and cut traces at those junction blocks. It achieves almost all the compaction of the trace scheduling except that which

causes copying of blocks. Memory size after the beta compaction is usually smaller than the original. Even when the memory size grows in rare instances, it is bounded by $O(n^2)$ in the worst case. And the bookkeeping stage is very much simplified. The compacted microcode size variation as the source microcode changes is also very small.

A loop-free version of both beta compaction and trace scheduling has been implemented. Comparison between the two was done using artificially generated microcodes and the above properties of the beta compaction was confirmed.

A simple microprogrammable machine based on AM2900 components was designed and simulated with an interactive user-friendly interface. A realistic application program was written and hand-compiled into microcode. The microcode was executed on the simulator both before and after compaction, which demonstrated the applicability of the compaction technique and the correctness of the implementation.

6.2. Future Research

The first immediate tasks are to implement loop handling in beta compaction and to test it with more programs of various kinds.

To improve the beta compaction, it might be possible to incorporate Isoda's extended data dependency graph [ISO83] into the basic scheme of the beta compaction. This may serve as a more elegant solution to the global compaction problem.

In this broad area of research toward developing better microprogramming tools, more work is called for in the area of microprogramming language design, the code generation problem including resource allocation, microcode verification, and the integration of available knowledge.

APPENDICES

APPENDIX A

THE SIMULATOR FUNCTIONS AND HUMAN INTERFACE

The simulator is an interactive menu driven software package and all its functions are controlled by selecting appropriate entries in its menu window. So the best way to describe the simulator function is by describing entries in the menu windows.

The menu system has 5 major frames where you can move from one frame to the other freely and one of the 5 frames (Alter and Display frame) has 5 sub-frames of its own.

Processor initialization frame (Figure A.1)
trap/trace frame (Figure A.2)
Alter/display frame
    Datapath subframe (Figure A.3)
    Pipeline register subframe (Figure A.4)
    Control store subframe (Figure A.5)
    Main store subframe (Figure A.6)
    Trace table subframe (Figure A.7)
Recording frame (Figure A.8)
Performance frame (Figure A.9)

Each frame has a one-line title on top of the screen and a menu and data display area in the middle. At the lower part of the screen, there are the one-line error message display field, the function field and the frame select field. And in the same line with the function field, there may be either the file name field or the address field depending on the frame selected. The last line is used to display the status of clock, stop, cycle, lcsar (last control store address register) and csar (control store address register). The clock field displays whether internal

clock is running or has stopped. The stop field displays the reason of a stopped clock, i.e. if the clock is stopped because of error, break point encountered, single step, microprogram exit, or time out. The cycle field indicates the time limit in terms of number of cycles which can be easily changeable. The lcsar displays the address of control store which contains the micro instruction which has just been executed. The csar displays the address of control store which contains the micro instruction which is to be executed at the next cycle.

Whenever a new frame is displayed, the cursor is located at the function field. The cursor can be moved around using either four arrow keys (up, down, left and right) or h(left), j(down), k(up), l(right) keys which are UNIX vi convention.

The cursor can be moved only to those fields where a user is allowed to change its value. Trying to move toward the wrong direction will yield a 'beep'. The fields where the user is allowed to change its value are indicated by '=>' and those which are mere display of something are indicated by '->'.

The following keys are not displayed on the menu but are available throughout the simulator and could be very useful.

h or left arrow	Move cursor to next allowable field to the left
j or down arrow	Move cursor to next allowable field to downward
k or up arrow	Move cursor to next allowable field to upward
l or right arrow	Move cursor to next allowable field to the right
^L (control-L)	Redraw the screen
^Z (control-Z)	Suspend the program and go back to UNIX (Execution can be resumed by shell command 'fg' (foreground))
break key	Stop the program and go back to UNIX (equivalent to function 'q')
: (colon)	Move cursor to function field
+ (plus)	In numeric field, increment the value by 1

- (minus)	In numeric field, decrement the value by 1
. (period)	In numeric field, reset the value to zero
? (question mark)	Help command. Displays the above info.

A.1 Processor Init Frame

The processor initialization frame is shown in Figure A.1. This is the first frame a user will see when he starts the simulator. Here he can load the control store and main store from appropriate files. To do this, first move the cursor to file name field and type the file name followed by <carriage return> which makes the cursor go back to the function field. Then type function selection character. The files are required to be in a certain format. One way to make the

```
 i ========================= Processor Initialization ========================

Fct  Description                          Hardware Timings

 c   Load Control Store                    Machine Cycle Time      =>  200 nS
 m   Load Main Store                       Memory Read Cycle Time  =>  350 nS
 q   Quit and Return to UNIX               Memory Write Cycle Time =>  350 nS
 x   Exit to Selected Frame
     i   Processor Initialization
     d   Alter/Display                    Note: Above numbers are in decimal
     t   Traps/Traces
     r   Recording Functions
     p   Performance Data



---> Control Store has been loaded from amdasm/cs.tree
Function =>                    Filename => amdasm/cs.tree

Frame Select =>
Clock -> STOPPED     Stop -> NONE     Cycles => fff     LCSAR ->   0   CSAR =>   0
```

Figure A.1 Processor initialization frame

files with a right format is to store default values (all zeros) of the memories in files using store functions in recording frame (Section A.4) and edit the files.

Basic hardware timings can be set up. Currently settable parameters are machine cycle time (micro-cycle time), main store read cycle time and main store write cycle time. If either read or write cycle time is greater than machine cycle time, proper wait states are generated and added to total elapsed time. See the Section A.5 for performance data frame.

A.2 Trap/Trace Frame

The trap/trace frame is shown in Figure A.2. Here traps (break points) and trace can be set up or cleared. Trace can be turned on or off and the trace

```
 t ============================== Traps/Traces ==============================

Fct  Description                          Control Store Traps ::::::::::::

 n   Csar Trace ON                        Trap => a  From =>   6e  To =>   6e
 f   Csar Trace OFF                       Trap => x  From =>    0  To =>    0
 e   Empty Trace Table                    Trap => x  From =>    0  To =>    0
 q   Quit and Return to UNIX              Trap => x  From =>    0  To =>    0
 x   Exit to Selected Frame               Trap => x  From =>    0  To =>    0
     i   Processor Initialization
     d   Alter/Display                    a-Active   x-Off
     t   Traps/Traces
     r   Recording Functions              Main Store Traps ::::::::::::::
     p   Performance Data
                                          Trap => r  From =>    4  To =>    7
                                          Trap => s  From =>   1a  To =>   1a
                                          Trap => x  From =>    0  To =>    0
CSAR Trace -> ON
                                          f-Fetch   s-Store   r-Reference   x-Off

Function =>

Frame Select =>
Clock -> STOPPED    Stop -> NONE    Cycles => fff    LCSAR ->   0   CSAR =>   0
```

Figure A.2 Trap/trace frame

table can be emptied using appropriate functions. There are two different kinds of traps. One for the control store and the other for the main store. For the control store traps, a user can activate of deactivate the traps. For the main store traps, he can activate the traps for three different ways: fetch, store and reference (fetch and store). All traps have their ranges. So, for example, any attempt to execute the micro instruction of an address between the range will make the execution of the user microprogram stop. Assigning the same values for both lower and upper limits is equivalent to setting up just one trap (break point) in the conventional way.

A.3 Alter/Display Frame

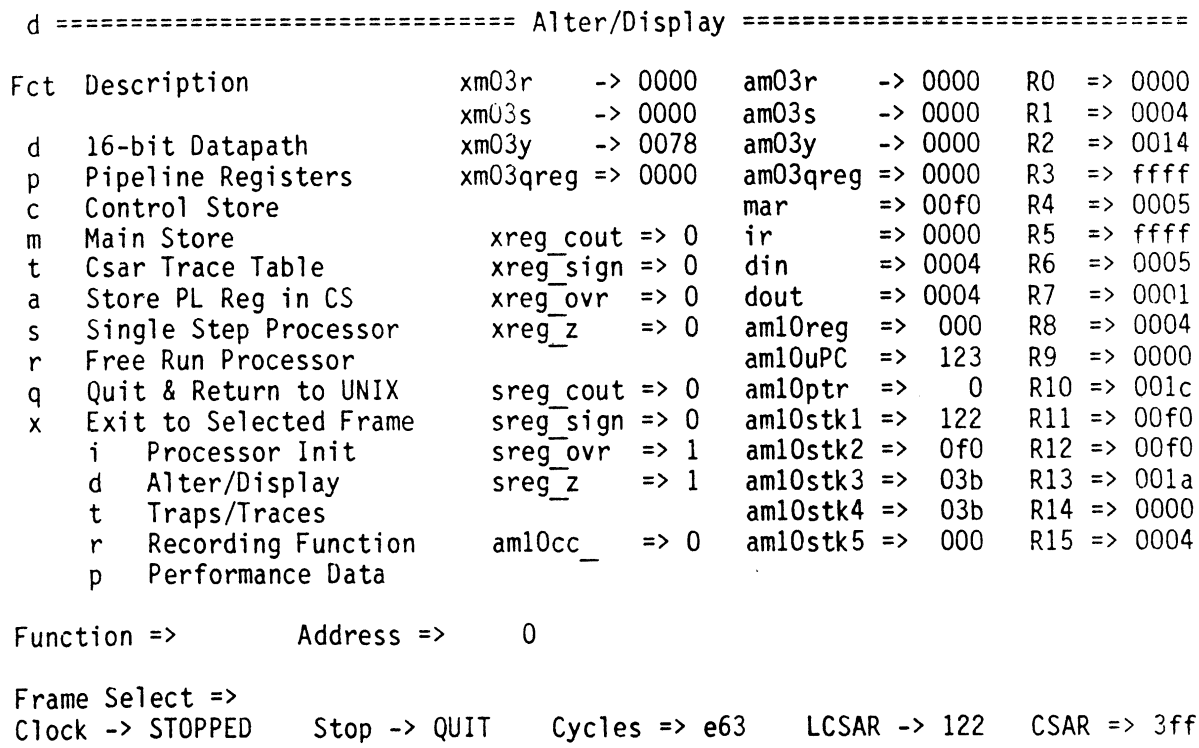


Figure A.3 Alter/display frame (datapath subframe)

This is considered the major frame of the simulator. It is the only frame where a user can single step or free run the microprogram. It displays and/or allows to change values of various parts of the simulated machine. It has five subframes which are shown in Figure A.3 through Figure A.7. The default subframe is the datapath subframe, which is displayed whenever this alter/display frame is entered.

For displaying control store, main store or trace table, this frame has an additional field called address. The address is used to indicate the first entry to be displayed. The address field is also used to indicate the index of trace table to be displayed on top. The address (index) zero of trace table subframe has a special meaning (see the description of trace table subframe).

```
 d ================================ Alter/Display ================================

Fct  Description              xm03a      => 0   am03a      => 0   lmar   => 0
                              xm03b      => 0   am03b      => 0   lir    => 0
 d   16-bit Datapath          xm03i8765  => 0   am03i8765  => 0   ldin   => 0
 p   Pipeline Registers       xm03i4321  => 0   am03i4321  => 0   ldout  => 0
 c   Control Store            xm03ea     => 0   am03ea     => 0
 m   Main Store               xm03i0_    => 0   am03i0_    => 0   write  => 0
 t   Csar Trace Table         xm03oeb_   => 0   am03oeb_   => 0   ir_mux => 0
 a   Store PL Reg in CS       xm03we_    => 0   am03we_    => 0   am10i  => 0
 s   Single Step Processor    xm03oey_   => 0   am03oey_   => 0
 r   Free Run Processor       xm03cn     => 0   am03cn     => 0   cc_comp   => 0
 q   Quit & Return to UNIX                                        cc_mux    => 0
 x   Exit to Selected Frame   xrfin_mux  => 0                     am10ccen_ => 0
     i   Processor Init       x1da_mux   => 0   x0da_mux  => 0   am10rld_  => 0
     d   Alter/Display        x1db_mux   => 0   x0db_mux  => 0
     t   Traps/Traces         xmar_mux   => 0                     ddata  => 0000
     r   Recording Function   xdout_mux  => 0
     p   Performance Data

Function =>        Address =>     0

Frame Select =>
Clock -> STOPPED    Stop -> QUIT    Cycles => e63    LCSAR -> 122    CSAR => 3ff
```

Figure A.4 Alter/display frame (pipeline register subframe)

When running (or single stepping) through the microprogram, once a runtime error occurs, the remaining simulation of the cycle is meaningless. So the simulator returns immediately if a runtime error is detected without going through the cycle completely.

(1) Datapath Subframe

The datapath subframe is shown in Figure A.3. This displays all 16 registers in AM2903 and other registers and allows a user to change them. It also displays input and output of the ALU which are not registers and thus not allowed to be changed by a user (indicated by '->').

(2) Pipeline Register Subframe

```
 d ================================ Alter/Display ================================

Fct  Description                          Control Store

 d   16-bit Datapath                      addr   msb                      lsb
 p   Pipeline Registers                     0    00030300 0010010b
 c   Control Store                          1    000e0102 c680ffff
 m   Main Store                            2    000e0101 c3900000
 t   Csar Trace Table                      3    020e0100 00100000
 a   Store PL Reg in Control Store         4    008e0100 c1b4000f
 s   Single Step Processor                 5    00034500 00100027
 r   Free Run Processor                    6    000e0101 c3900001
 q   Quit and Return to UNIX               7    020e0100 00100000
 x   Exit to Selected Frame                8    008e0100 c1340000
     i   Processor Initialization         9    000a0500 00100000
     d   Alter/Display                     a    000e010e c680000f
     t   Traps/Traces                      b    00010300 00100122
     r   Recording Functions
     p   Performance Data

Function =>        Address =>     0

Frame Select =>
Clock -> STOPPED    Stop -> SSTEP   Cycles => f43    LCSAR ->  71   CSAR =>  72
```

Figure A.5 Alter/display frame (control store subframe)

```
d ============================== Alter/Display ==============================

Fct  Description                          Main Store

 d   16-bit Datapath                     addr   data
 p   Pipeline Registers                    0    000f 0003 000f 0007
 c   Control Store                         4    00f0 0000 0002 ffff
 m   Main Store                           8    0007 0000 000f 0005
 t   Csar Trace Table                     c    0000 0000 0000 0000
 a   Store PL Reg in Control Store       10    0000 0000 0000 0000
 s   Single Step Processor               14    0000 0000 0000 0000
 r   Free Run Processor                  18    0000 0000 0000 0000
 q   Quit and Return to UNIX             1c    0000 0000 0000 0000
 x   Exit to Selected Frame              20    0000 0000 0000 0000
     i   Processor Initialization        24    0000 0000 0000 0000
     d   Alter/Display                   28    0000 0000 0000 0000
     t   Traps/Traces                    2c    0000 0000 0000 0000
     r   Recording Functions
     p   Performance Data

Function =>        Address =>     0

Frame Select =>
Clock -> STOPPED    Stop -> SSTEP   Cycles => f43    LCSAR ->  71   CSAR =>  72
```

Figure A.6 Alter/display frame (main store subframe)

The pipeline register subframe is shown in Figure A.4. This displays all the fields in the pipeline register and allows the user to change them. It is also possible to load the pipeline register from arbitrary location of control store or to store the pipeline register to arbitrary location of control store. Loading the pipeline register can be done by changing CSAR values at the bottom right of the screen. Storing the pipeline register is done by function 'a' which stores the current value of pipeline register in the control store addressed by the address field on the screen.

(3) Control Store Subframe

The control store subframe is shown in Figure A.5. This simply displays part of the control store on the screen starting from the address specified by the

```
 d ============================== Alter/Display ==============================

Fct  Description                          Trace Table

 d   16-bit Datapath                      index   CSAR
 p   Pipeline Registers                     b6     6c
 c   Control Store                          b7     6d
 m   Main Store                            b8     6e
 t   Csar Trace Table                       b9     6f
 a   Store PL Reg in Control Store          ba     70
 s   Single Step Processor                  bb     71
 r   Free Run Processor                     bc     72
 q   Quit and Return to UNIX                bd     73
 x   Exit to Selected Frame                 be     74
     i   Processor Initialization          bf     75
     d   Alter/Display                      c0     76
     t   Traps/Traces                       c1     77  <-- last trace
     r   Recording Functions
     p   Performance Data

Function =>        Address =>     0

Frame Select =>
Clock -> STOPPED    Stop -> SSTEP   Cycles => f3d    LCSAR ->  77   CSAR =>  78
```

Figure A.7 Alter/display frame (trace table subframe)

address field on the screen. A user is not allowed to change them on the screen. But there are two ways to change the contents of control store. Either set up the pipeline register and store it in the control store as explained above (this is a lot more sensible way than dealing with hexadecimal numbers), or save the contents of the control store in a file from recording frame, suspend the execution of this simulator, edit the file, resume execution by 'fg', and load the file into the control store.

(4) Main Store Subframe

The main store subframe is shown in Figure A.6. This displays part of the main store on the screen starting from the address specified by the address field on the screen. The user is not allowed to change them on the screen. To change

```
 r ============================ Recording Functions ==========================

Fct  Description

 c   Save Control Store
 m   Save Main Store
 f   Save Machine Facilities
 q   Quit and Return to UNIX
 x   Exit to Selected Frame
     i   Processor Initialization
     d   Alter/Display
     t   Traps/Traces
     r   Recording Functions
     p   Performance Data




---> Main Store has been saved in ms.tree
Function =>                    Filename => ms.tree

Frame Select =>
Clock -> STOPPED    Stop -> SSTEP   Cycles => f3d    LCSAR ->  77   CSAR =>  78
```

Figure A.8 Recording frame

the contents of the main store, the latter procedure described in Control Store Subframe has to be followed.

(5) Trace Table Subframe

The trace table subframe is shown in Figure A.7. This displays part of the main store on the screen starting from the index specified by the address field on the screen. When the address field is zero, and the trace table is not empty, then this subframe will display the last entry (most recently updated entry) at the bottom so that most recent 12 entries are displayed. This feature eliminates the need for a search of the last entry in the buffer of 256 entries. The last entry of the trace is clearly marked.

A.4 Recording Frame

```
p ============================ Performance Data ============================

Fct  Description                              Measurement

 r   Reset all Performance Data                Elapsed Micro Cycle   ->  621
 q   Quit and Return to UNIX                   Number of Wait States ->  119
 x   Exit to Selected Frame                    Total Elapsed Time    ->  148.0 usec
     i   Processor Initialization
     d   Alter/Display                         Note: Above numbers are in decimal
     t   Traps/Traces
     r   Recording Functions
     p   Performance Data

Function =>

Frame Select =>
Clock -> STOPPED    Stop -> QUIT    Cycles => d92    LCSAR -> 121    CSAR => 3ff
```

Figure A.9 Performance data frame

The recording frame is shown in Figure A.8. This frame allows a user to save the contents of the main store and the control store in files. To do this, he has to type file name first and then type the appropriate function.

A.5 Performance Data Frame

The performance data frame is shown in Figure A.9. This frame displays statistics collected during the simulation. Currently, elapsed micro cycle, number of wait states and total elapsed time are displayed. All of these performance data can be reset to zero.

APPENDIX B

2-3 TREE INSERTION ALGORITHM

```
program tree( input, output, data );

type   elementtype = record
                       key : integer;
                       (* other fields as warranted *)
                     end;

       nodetypes = ( leaf, interior );

       nodeptr = ^twothreenode;
       twothreenode = record
                       case kind : nodetypes of
                             leaf : ( element : elementtype );
                             interior : ( firstchild  : nodeptr;
                                          secondchild : nodeptr;
                                          thirdchild  : nodeptr;
                                          lowofsecond : integer;
                                          lowofthird  : integer )
                       end;


var    root : nodeptr;
       newp : elementtype;
       numofdata, i : integer;
       data : file of char;




procedure Printtree( pnode : nodeptr );

begin
       writeln;
       if pnode^.kind = leaf
       then
             writeln( 'element = ':10, pnode^.element.key:3 )
       else
       begin
```

```
            writeln( 'lowofsecond = ':14, pnode^.lowofsecond:3 );
            writeln( 'lowofthird  = ':14, pnode^.lowofthird:3 );

            if pnode^.firstchild <> nil
            then
            begin
                  write( 'First child ' );
                  Printtree( pnode^.firstchild );
            end;

            if pnode^.secondchild <> nil
            then
            begin
                  write( 'Second child ' );
                  Printtree( pnode^.secondchild );
            end;

            if pnode^.thirdchild <> nil
            then
            begin
                  write( 'Third child ' );
                  Printtree( pnode^.thirdchild );
            end;
      end;
end;




procedure insert1( node : nodeptr;
                  x : elementtype; (* x is to be inserted into the subtree
                                      of node *)
                  var pnew : nodeptr; (* pointer to new node created
                                         to right of node *)
                  var low : integer ); (* smallest element in the subtree
                                          pointed to by pnew *)

var   pback : nodeptr;
      lowback : integer;
      child : 1..3; (* indicates which child of node is followed
                       in recursive call *)
      w : nodeptr; (* pointer to the child *)
```

```
begin
        pnew := nil;
        if node^.kind = leaf
        then
        begin
                if node^.element.key <> x.key
                then
                begin
                        (* create new leaf holding x.key and return this node *)

                        new( pnew, leaf );
                        pnew^.kind := leaf;
                        if node^.element.key < x.key
                        then
                        begin
                                (* place x in new node to right of current node *)

                                pnew^.element := x;
                                low := x.key;
                        end
                        else
                        begin
                                (* x belongs to left of element at current node *)

                                pnew^.element := node^.element;
                                node^.element := x;
                                low := pnew^.element.key;
                        end;
                end;
        end

        else
        begin
                (* node is an interior node *)
                (* select the child of node that we must follow *)

                if x.key < node^.lowofsecond
                then
                begin
                        child := 1;
                        w := node^.firstchild;
                end
                else if (node^.thirdchild = nil ) or
                        (x.key < node^.lowofthird)
                then
```

```
begin
        (* x is in second subtree *)

        child := 2;
        w := node^.secondchild;
end
else
begin
        (* x is in third subtree *)

        child := 3;
        w := node^.thirdchild;
end;


insert1( w, x, pback, lowback );


if pback <> nil
then
        (* a new child of node must be inserted *)

        if node^.thirdchild = nil
        then
                (* node had only two children, so insert
                   new node in proper place *)

                if child = 2
                then
                begin
                        node^.thirdchild := pback;
                        node^.lowofthird := lowback;
                end
                else (* child = 1 *)
                begin
                        node^.thirdchild := node^.secondchild;
                        node^.lowofthird := node^.lowofsecond;
                        node^.secondchild := pback;
                        node^.lowofsecond := lowback;
                end
        else
        begin
                (* node already had three children *)

                new( pnew, interior );
```

```
pnew  kind := interior;

if child = 3
then
begin
        (* pback and third child become
          children of new node *)

        pnew^.firstchild := node^.thirdchild;
        pnew^.secondchild := pback;
        pnew^.thirdchild := nil;
        pnew^.lowofsecond := lowback;

        (* lowofthird is undefined for pnew *)

        low := node^.lowofthird;
        node^.thirdchild := nil;
end
else
begin
        (* child <= 2; move third child of
          node to pnew *)

        pnew^.secondchild := node^.thirdchild;
        pnew^.lowofsecond := node^.lowofthird;
        pnew^.thirdchild := nil;
        node^.thirdchild := nil;
end;
if child = 2
then
begin
        (* pback becomes first child of pnew *)

        pnew^.firstchild := pback;
        low := lowback;
end;
if child = 1
then
begin
        (* second child of node is moved to
          pnew; pback becomes second child
          of node *)

        pnew^.firstchild := node^.secondchild;
        low := node^.lowofsecond;
```

```
                        node^.secondchild := pback;
                        node^.lowofsecond := lowback;
                    end;

                end;

        end;

end; (* insert1 *)



procedure INSERT( x : elementtype; var S : nodeptr );

var     pback : nodeptr; (* pointer to new node returned by insert1 *)
        lowback : integer; (* low value in subtree of pback *)
        saveS : nodeptr; (* place to store temporary copy of the pointer S *)

begin
        (* checks for S being empty or a single node should occur here,
           and an appropriate insertion procedure should be included *)

        insert1( S, x, pback, lowback );

        if pback <> nil
        then
        begin
                (* create new root; its children are now pointed to by S
                   and pback *)

                saveS := S;
                new( S, interior );
                S^.kind := interior;
                S^.firstchild := saveS;
                S^.secondchild := pback;
                S^.lowofsecond := lowback;
                S^.thirdchild := nil;
        end;

end; (* INSERT *)
```

```
begin

        new( root, leaf );
        root^.kind := leaf;
        root^.element.key := 5;
        Printtree( root );

        newp.key := 7;
        INSERT( newp, root );
        writeln( 'INSERT ', newp.key );
        Printtree( root );

        newp.key := 3;
        INSERT( newp, root );
        writeln( 'INSERT ', newp.key );
        Printtree( root );

        newp.key := 2;
        INSERT( newp, root );
        writeln( 'INSERT ', newp.key );
        Printtree( root );

        newp.key := 4;
        INSERT( newp, root );
        writeln( 'INSERT ', newp.key );
        Printtree( root );

end.
```

APPENDIX C

AMDASM ASSEMBLY LISTING OF 2-3 TREE INSERTION ALGORITHM

The following is a AMDASM definition source listing of 2-3 tree insertion algorithm.

```
TITLE  Test Program 1 for Microprogram Simulator
WORD 96
;
WD:   EQU  B#1                          ; Write disable
WDDEF: DEF  19X, WD, 12X, 43X, WD, 20X       ; Write disable definition
;
; Datapath Control Signals
;
LMAR:  DEF  19X, WD, 12X, 6X,  B#1, 36X, WD, 20X      ; Load MAR
LIR:   DEF  19X, WD, 12X, 7X,  B#1, 35X, WD, 20X      ; Load IR
LDIN:  DEF  19X, WD, 12X, 8X,  B#1, 34X, WD, 20X      ; Load Data In
LDOUT: DEF  19X, WD, 12X, 9X,  B#1, 33X, WD, 20X      ; Load Data Out
WRITE: DEF  32X, 10X, B#1, 53X                        ; Memory Write
IRAM:  DEF  19X, WD, 12X, 11X, B#1, 31X, WD, 20X      ; IR RAM Select
;
; Definitions for AM2910 Sequencer
;
JZ:    DEF  19X, WD, 12X, 12X, H#0, 6X,       B#1, 1VB#1, 8X, 11X,
/      WD, 20X  ; Jump Zero
CJS:   DEF  19X, WD, 12X, 12X, H#1, 1X, 1VB#0, 4VH#0, B#0, 1VB#1, 8X, 11X,
/      WD, 8X, 12V%H#000  ; Cond JSB PL
JSB:   DEF  19X, WD, 12X, 12X, H#1, 6X,       B#1, 1VB#1, 8X, 11X,
/      WD, 8X, 12V%H#000  ; Unconditional JSB PL
JMAP:  DEF  19X, WD, 12X, 12X, H#2, 6X,       B#1, 1VB#1, 8X, 11X,
/      WD, 20X  ; Jump Map
CJP:   DEF  19X, WD, 12X, 12X, H#3, 1X, 1VB#0, 4VH#0, B#0, 1VB#1, 8X, 11X,
/      WD, 8X, 12V%H#000  ; Cond Jump PL
JMP:   DEF  32X, 12X, H#3, 6X,       B#1, 1VB#1, 8X, 20X, 12V%H#000
/      ; Unconditional Jump PL
PUSH:  DEF  19X, WD, 12X, 12X, H#4, 2X, 4VH#0, B#0, 1VB#1, 8X, 11X,
/      WD, 20X  ; Push/Cond Load CNTR
PHLC:  DEF  19X, WD, 12X, 12X, H#4, 6X,       B#1, 1VB#1, 8X, 11X,
/      WD, 20X  ; Push and Load CNTR
JSRP:  DEF  19X, WD, 12X, 12X, H#5, 6X,       B#1, 1VB#1, 8X, 11X,
/      WD, 20X  ; Cond JSB R/PL
CJV:   DEF  19X, WD, 12X, 12X, H#6, 2X, 4VH#0, B#0, 1VB#1, 8X, 11X,
/      WD, 20X  ; Cond Jump Vector
JMPV:  DEF  19X, WD, 12X, 12X, H#6, 6X,       B#1, 1VB#1, 8X, 11X,
/      WD, 20X  ; Unconditional Jmp Vector
JRP:   DEF  19X, WD, 12X, 12X, H#7, 6X,       B#1, 1VB#1, 8X, 11X,
/      WD, 20X  ; Cond Jump R/PL
RFCT:  DEF  19X, WD, 12X, 12X, H#8, 6X,       B#1, 1VB#1, 8X, 11X,
/      WD, 20X  ; Repeat Loop, CNTR <> 0
RPCT:  DEF  19X, WD, 12X, 12X, H#9, 6X,       B#1, 1VB#1, 8X, 11X,
/      WD, 20X  ; Repeat PL, CNTR<> 0
CRTN:  DEF  19X, WD, 12X, 12X, H#A, 1X, 1VB#0, 4VH#0, B#0, 1VB#1, 8X, 11X,
```

```
/       WD, 20X  ; Cond RTN
RTN:  DEF  32X, 12X, H#A, 6X,       B#1, 1VB#1, 8X, 32X          ; Unconditional RTN
CJPP:  DEF  19X, WD, 12X, 12X, H#B, 6X,       B#1, 1VB#1, 8X, 11X,
/       WD, 20X  ; Cond Jump Pl & Pop
LDCT:  DEF  19X, WD, 12X, 12X, H#C, 6X,       B#1, 1VB#1, 8X, 11X,
/       WD, 20X  ; Load CNTR & Cont
LOOP:  DEF  19X, WD, 12X, 12X, H#D, 6X,       B#1, 1VB#1, 8X, 11X,
/       WD, 20X  ; Test End Loop
CONT:  DEF  32X, 12X, H#E, 6X,       B#1, 1VB#1, 8X, 32X                ; Continue
TWB:  DEF  19X, WD, 12X, 12X, H#F, 6X,       B#1, 1VB#1, 8X, 11X,
/       WD, 20X  ; Three-way Branch
;
; Register Load ( AM2910 )
;
RLD:  EQU  B#0        ; Register Load
;
; Datapath Definition
;
DATAPATH:  DEF  19X, WD, 12X, 24X, 4VH#0, 4VH#0, 4VH#C, 4VH#0,
/       3VQ#0, 1VB#1, 1VB#0, 19X
;
; Register Definition
;
R0:  EQU  H#0
R1:  EQU  H#1
R2:  EQU  H#2
R3:  EQU  H#3
R4:  EQU  H#4
R5:  EQU  H#5
R6:  EQU  H#6
R7:  EQU  H#7
R8:  EQU  H#8
R9:  EQU  H#9
R10:  EQU  H#A
R11:  EQU  H#B
R12:  EQU  H#C
R13:  EQU  H#D
R14:  EQU  H#E
R15:  EQU  H#F
;
RX:       EQU  H#1
RNODE:    EQU  H#2
RPNEW:    EQU  H#3
RLOW:     EQU  H#4
RPBACK:   EQU  H#5
RLOWBACK:  EQU  H#6
RCHILD:   EQU  H#7
RW:       EQU  H#8
RNODEBEG:  EQU  H#9
RFREENOD:  EQU  H#A
RSTKBEG:  EQU  H#B
RTOPSTK:  EQU  H#C
RNEWPTR:  EQU  H#D
RNODETYP:  EQU  H#E
RSAVE:    EQU  H#F
;
; Equates for ALU Destination Control
;
ADR:  EQU  H#0        ; Arithmetic Shift Down, Results into RAM
LDR:  EQU  H#1        ; Logical Shift Down, Results into RAM
ADRQ:  EQU  H#2       ; Arithmetic Shift Down, Results into RAM and Q
LDRQ:  EQU  H#3       ; Logical Shift Down, Results into RAM and Q
RPT:  EQU  H#4        ; Results into RAM, Generate Parity
```

```
LDQP:  EQU  H#5        ; Logical Shift Down Q, Generate Parity
QPT:   EQU  H#6        ; Results into Q, Generate Parity
RQPT:  EQU  H#7        ; Results into RAM and Q, Generate Parity
AUR:   EQU  H#8        ; Arithmetic Shift Up, Results into RAM
LUR:   EQU  H#9        ; Logical Shift Up, Results into RAM
AURQ:  EQU  H#A        ; Arithmetic Shift Up, Results into RAM and Q
LURQ:  EQU  H#B        ; Logical Shift Up, Results into RAM and Q
YBUS:  EQU  H#C        ; Results to Y Bus only
LUQ:   EQU  H#D        ; Logical Shift Up Q
SINEX: EQU  H#E        ; Sign Extend
REG:   EQU  H#F        ; Results to RAM, Sign Extend
;
; Equates for ALU Functions
;
HIGH:  EQU  H#0        ; Fi = 1
SUBR:  EQU  H#1        ; Subtract R from S
SUBS:  EQU  H#2        ; Subtract S from R
ADD:   EQU  H#3        ; Add R and S
PASS:  EQU  H#4        ; Pass S
COMPLS: EQU H#5        ; 2's Complement of S
PASSR: EQU  H#6        ; Pass R
COMPLP: EQU H#7        ; 2's Complement of R
LOW:   EQU  H#8        ; Fi = 0
NOTRS: EQU  H#9        ; Complement R AND with S
EXNOR: EQU  H#A        ; Exclusive NOR R with S
EXOR:  EQU  H#B        ; Exclusive OR R with S
AND:   EQU  H#C        ; AND R with S
NOR:   EQU  H#D        ; NOR R with S
NAND:  EQU  H#E        ; NAND R with S
OR:    EQU  H#F        ; OR R with S
;
; ALU Operand Sources
;
NOTSQ: EQU  Q#5        ; S = not Q
SQ:    EQU  Q#7        ; S = Q
;
XAB:   EQU  B#00       ; R = RAM A, S = RAM B
XADB:  EQU  B#01       ; R = RAM A, S = DIN
XDAB:  EQU  B#10       ; R = PL reg, S = RAM B
XDADB: EQU  B#11       ; R = PL reg, S = DIN
;
; AM2903 Control Signals
;
WE:    EQU  B#0        ; Write Enable
DISY:  EQU  B#1        ; Disable Y
;
; Immediate Data
;
IMMD:  DEF  19X, WD, 12X, 43X, WD, 4X, 16VH#        ; Immediate Data
;
; Load buffers
;
LMARBUF:  DEF  32X, B#1, 63X
LDINBUF:  DEF  33X, B#1, 62X
LDOUTBUF: DEF  34X, B#1, 61X
LSTATBUF: DEF  35X, B#1, 60X
;
; Various Instructions
;
IMMDLOAD: DEF  19X, WD, 6X, XDADB, 4X, 24X, 4X, 4VH#0, YBUS, PASSR, SQ,
/         WE, B#0, 3X, 16VH#
;
CALADREG: DEF  19X, WD, 6X, XAB, 4X, 24X, 4X, 4VH#0, YBUS, PASS, NOTSQ,
```

```
/       WD, B#0, 3X, 16X
;
CALADIMD: DEF  19X, WD, 6X, XDAB, 4X, 24X, 4X, 4VH#0, YBUS, ADD, NOTSQ,
/       WD, B#0, 3X, 16VH#0000
;
LDNTESTI: DEF  19X, WD, 6X, XDADB, 4X, 8X, B#1, 15X, 4X, 4X, YBUS, SUBR, NOTSQ,
/       WD, B#0, B#1, 2X, 16VH#
;
LDNSUBR:  DEF  19X, WD, 6X, XADB, 4X, 8X, B#1, 15X, 4VH#0, 4X, YBUS, SUBR, NOTSQ,
/       WD, B#0, B#1, 2X, 16X
;
REGLOAD:  DEF  19X, WD, 6X, XADB, 4X, 24X, 4VH#0, 4VH#0, YBUS, PASSR, SQ,
/       WE, B#0, 3X, 16X
;
ADDNLOAD: DEF  19X, WD, 6X, XDAB, 4X, 24X, 4X, 4VH#0, YBUS, ADD, NOTSQ,
/       WE, B#0, 3X, 16VH#0000
;
PASSIMD:  DEF  19X, WD, 6X, XDADB, 4X, 24X, 4X, 4X, YBUS, PASSR, SQ,
/       WD, B#0, 3X, 16VH#
;
PASSREG:  DEF  19X, WD, 6X, XADB, 4X, 24X, 4VH#0, 4X, YBUS, PASSR, SQ,
/       WD, B#0, 3X, 16X
;
PASSDIN:  DEF  19X, WD, 6X, XADB, 4X, 8X, B#1, 15X, 4X, 4X, YBUS, PASS, NOTSQ,
/       WD, B#0, 3X, 16X
;
DINLOAD:  DEF  19X, WD, 6X, XADB, 4X, 8X, B#1, 15X, 4X, 4VH#0, YBUS, PASS, NOTSQ,
/       WE, B#0, 3X, 16X
;
REGTESTI: DEF  19X, WD, 6X, XDAB, 4X, 24X, 4X, 4VH#0, YBUS, SUBR, NOTSQ,
/       WD, B#0, B#1, 2X, 16VH#0000
;
;
;  Constants
;
;
NOT:      EQU  B#1
IFZERO:   EQU  H#1
IFNEG:    EQU  H#2
ONE16:    EQU  16H#0001
NEGONE16: EQU  16H#FFFF
ZERO16:   EQU  16H#0000
HIGH16:   EQU  16H#FFFF
LOW16:    EQU  16H#0000
NIL:      EQU  16H#FFFF
NODETYPE: EQU  16H#0000
ELEMENT:  EQU  16H#0001
LEAF:     EQU  16H#000F
INTERIOR: EQU  16H#00F0
FSTCHILD: EQU  16H#0001
SNDCHILD: EQU  16H#0002
TRDCHILD: EQU  16H#0003
LOWOFSND: EQU  16H#0004
LOWOFTRD: EQU  16H#0005
QUIT:     EQU  12H#3FF          ; highest legal cs address
;
END
```

The following is a partial AMDASM assembly listing of 2-3 tree insertion algorithm. AMDASM assembly source code written by hand for each line of Pascal program. For each line of AMDASM source code, the bit pattern is generated where X indicates a "don't care".

```
 989  000EF  ;procedure INSERT( x : elementtype; var S : nodeptr );
 990  000EF  ;
 991  000EF  ;var   pback : nodeptr; (* pointer to new node returned by insert1 *)
 992  000EF  ;      lowback : integer; (* low value in subtree of pback *)
 993  000EF  ;      saveS : nodeptr; (* place to store a temporary copy of the pointer S *)
 994  000EF  ;
 995  000EF  ;begin
 996  000EF  ;      (* checks for S being empty or a single node should occur here,
 997  000EF  ;        and an appropriate insertion procedure should be included *)
 998  000EF  ;      (* for our purposes, we assume that S has a 'legal' 2-3 tree *)
 999  000EF  ;
1000  000EF  ;      insert1( S, x, pback, lowback );
1001  000EF  ;
1002  000EF  INSERT: JSB  , INSERT1
          XXXXXXXX XXXXXXXX XXX1XXXX XXXXXXXX XXXXXXXX XXXX0001
          XXXXXX11 XXXXXXXX XXXXXXXX XXX1XXXX XXXX0000 00000001
1003  000F0  ;
1004  000F0  ;      Link return argument
1005  000F0  ;
1006  000F0        REGLOAD  RPNEW, RPBACK  &
1007  000F0  /     CONT
          XXXXXXXX XXXXXXXX XXX1XXXX XX01XXXX XXXXXXXX XXXX1110
          XXXXXX11 00110101 11000110 11100XXX XXXXXXXX XXXXXXXX
1008  000F1  ;
1009  000F1        REGLOAD  RLOW, RLOWBACK  &
1010  000F1  /     CONT
          XXXXXXXX XXXXXXXX XXX1XXXX XX01XXXX XXXXXXXX XXXX1110
          XXXXXX11 01000110 11000110 11100XXX XXXXXXXX XXXXXXXX
1011  000F2  ;
1012  000F2  ;      if pback <> nil
1013  000F2  ;
1014  000F2        REGTESTI  RPBACK, NIL  &  LSTATBUF  &
1015  000F2  /     CONT
          XXXXXXXX XXXXXXXX XXX1XXXX XX10XXXX XXX1XXXX XXXX1110
          XXXXXX11 XXXX0101 11000001 101101XX 11111111 11111111
1016  000F3  ;
1017  000F3        CRTN  , IFZERO
          XXXXXXXX XXXXXXXX XXX1XXXX XXXXXXXX XXXXXXXX XXXX1010
          X0000101 XXXXXXXX XXXXXXXX XXX1XXXX XXXXXXXX XXXXXXXX
1018  000F4  ;
1019  000F4  ;      then
1020  000F4  ;      begin
1021  000F4  ;            (* create new root; its children are now pointed to by S
1022  000F4  ;              and pback *)
1023  000F4  ;
1024  000F4  ;            saveS := S;
1025  000F4  ;
1026  000F4        REGLOAD  RNODE, RSAVE  &
1027  000F4  /     CONT
          XXXXXXXX XXXXXXXX XXX1XXXX XX01XXXX XXXXXXXX XXXX1110
          XXXXXX11 00101111 11000110 11100XXX XXXXXXXX XXXXXXXX
1028  000F5  ;
1029  000F5  ;            new( S, interior );
```

```
1030  000F5 ;
1LINE   ADDR          META ASSEMBLER ASSEMBLY PROG              PAGE  29

1031  000F5       REGLOAD  RFREENOD, RNEWPTR  &
1032  000F5 /     CONT
          XXXXXXXX XXXXXXXX XXX1XXXX XX01XXXX XXXXXXXX XXXX1110
          XXXXXX11 10101101 11000110 11100XXX XXXXXXXX XXXXXXXX
1033  000F6 ;
1034  000F6       ADDNLOAD  RFREENOD, H#0006  &  ; increment node storage offset
1035  000F6 /     CONT
          XXXXXXXX XXXXXXXX XXX1XXXX XX10XXXX XXXXXXXX XXXX1110
          XXXXXX11 XXXX1010 11000011 10100XXX 00000000 00000110
1036  000F7 ;
1037  000F7       REGLOAD  RNEWPTR, RNODE  &  ; RNODE <- pointer from 'new'
1038  000F7 /     CONT
          XXXXXXXX XXXXXXXX XXX1XXXX XX01XXXX XXXXXXXX XXXX1110
          XXXXXX11 11010010 11000110 11100XXX XXXXXXXX XXXXXXXX
1039  000F8 ;
1040  000F8 ;           S^.kind := interior;
1041  000F8 ;
1042  000F8       PASSIMD  INTERIOR  &  LDOUTBUF  &  ; DOUT <- interior
1043  000F8 /     CONT
          XXXXXXXX XXXXXXXX XXX1XXXX XX11XXXX XX1XXXXX XXXX1110
          XXXXXX11 XXXXXXXX 11000110 11110XXX 00000000 11110000
1044  000F9 ;
1045  000F9       LDOUT  &
1046  000F9 /     CONT
          XXXXXXXX XXXXXXXX XXX1XXXX XXXXXXXX XXXXXXXX X1XX1110
          XXXXXX11 XXXXXXXX XXXXXXXX XXX1XXXX XXXXXXXX XXXXXXXX
1047  000FA ;
1048  000FA       CALADREG  RNODE  &  LMARBUF  &        ; calculate address
1049  000FA /     CONT
          XXXXXXXX XXXXXXXX XXX1XXXX XX00XXXX 1XXXXXXX XXXX1110
          XXXXXX11 XXXX0010 11000100 10110XXX XXXXXXXX XXXXXXXX
1050  000FB ;
1051  000FB       LMAR  &  WRITE  &                  ; write to &RNODE
1052  000FB /     CONT
          XXXXXXXX XXXXXXXX XXX1XXXX XXXXXXXX XXXXXX1X XX1X1110
          XXXXXX11 XXXXXXXX XXXXXXXX XXX1XXXX XXXXXXXX XXXXXXXX
1053  000FC ;
1054  000FC ;           S^.firstchild := saveS;
1055  000FC ;
1056  000FC       PASSREG  RSAVE  &  LDOUTBUF  &        ; DOUT <- saveS
1057  000FC /     CONT
          XXXXXXXX XXXXXXXX XXX1XXXX XX01XXXX XX1XXXXX XXXX1110
          XXXXXX11 1111XXXX 11000110 11110XXX XXXXXXXX XXXXXXXX
1058  000FD ;
1059  000FD       LDOUT  &
1060  000FD /     CONT
          XXXXXXXX XXXXXXXX XXX1XXXX XXXXXXXX XXXXXXXX X1XX1110
          XXXXXX11 XXXXXXXX XXXXXXXX XXX1XXXX XXXXXXXX XXXXXXXX
1061  000FE ;
1062  000FE       CALADIMD  RNODE, FSTCHILD  &  LMARBUF  &  ; write to &RNODE
1063  000FE /     CONT
          XXXXXXXX XXXXXXXX XXX1XXXX XX10XXXX 1XXXXXXX XXXX1110
          XXXXXX11 XXXX0010 11000011 10110XXX 00000000 00000001
1064  000FF ;
1LINE   ADDR          META ASSEMBLER ASSEMBLY PROG              PAGE  30

1065  000FF       LMAR  &  WRITE  &                  ; write to &RNODE
1066  000FF /     CONT
          XXXXXXXX XXXXXXXX XXX1XXXX XXXXXXXX XXXXXX1X XX1X1110
          XXXXXX11 XXXXXXXX XXXXXXXX XXX1XXXX XXXXXXXX XXXXXXXX
```

```
1067  00100 ;
1068  00100 ;          S^.secondchild := pback;
1069  00100 ;
1070  00100       PASSREG RPBACK & LDOUTBUF &          ; DOUT <- RPBACK
1071  00100 /     CONT
          XXXXXXXX XXXXXXXX XXX1XXXX XX01XXXX XX1XXXXX XXXX1110
          XXXXXX11 0101XXXX 11000110 11110XXX XXXXXXXX XXXXXXXX
1072  00101 ;
1073  00101       LDOUT &
1074  00101 /     CONT
          XXXXXXXX XXXXXXXX XXX1XXXX XXXXXXXX XXXXXXXX X1XX1110
          XXXXXX11 XXXXXXXX XXXXXXXX XXX1XXXX XXXXXXXX XXXXXXXX
1075  00102 ;
1076  00102       CALADIMD RNODE, SNDCHILD & LMARBUF & ; calculate address
1077  00102 /     CONT
          XXXXXXXX XXXXXXXX XXX1XXXX XX10XXXX 1XXXXXXX XXXX1110
          XXXXXX11 XXXX0010 11000011 10110XXX 00000000 00000010
1078  00103 ;
1079  00103       LMAR & WRITE &                   ; write to &RNODE
1080  00103 /     CONT
          XXXXXXXX XXXXXXXX XXX1XXXX XXXXXXXX XXXXXX1X XX1X1110
          XXXXXX11 XXXXXXXX XXXXXXXX XXX1XXXX XXXXXXXX XXXXXXXX
1081  00104 ;
1082  00104 ;          S^.thirdchild := nil;
1083  00104 ;
1084  00104       PASSIMD NIL & LDOUTBUF &          ; DOUT <- NIL
1085  00104 /     CONT
          XXXXXXXX XXXXXXXX XXX1XXXX XX11XXXX XX1XXXXX XXXX1110
          XXXXXX11 XXXXXXXX 11000110 11110XXX 11111111 11111111
1086  00105 ;
1087  00105       LDOUT &
1088  00105 /     CONT
          XXXXXXXX XXXXXXXX XXX1XXXX XXXXXXXX XXXXXXXX X1XX1110
          XXXXXX11 XXXXXXXX XXXXXXXX XXX1XXXX XXXXXXXX XXXXXXXX
1089  00106 ;
1090  00106       CALADIMD RNODE, TRDCHILD & LMARBUF & ; calculate address
1091  00106 /     CONT
          XXXXXXXX XXXXXXXX XXX1XXXX XX10XXXX 1XXXXXXX XXXX1110
          XXXXXX11 XXXX0010 11000011 10110XXX 00000000 00000011
1092  00107 ;
1093  00107       LMAR & WRITE &                   ; write to &RNODE
1094  00107 /     CONT
          XXXXXXXX XXXXXXXX XXX1XXXX XXXXXXXX XXXXXX1X XX1X1110
          XXXXXX11 XXXXXXXX XXXXXXXX XXX1XXXX XXXXXXXX XXXXXXXX
1095  00108 ;
1096  00108 ;          S^.lowofsecond := lowback;
1097  00108 ;
1098  00108       PASSREG RLOWBACK & LDOUTBUF & ; DOUT <- RLOWBACK
1099  00108 /     CONT
          XXXXXXXX XXXXXXXX XXX1XXXX XX01XXXX XX1XXXXX XXXX1110
          XXXXXX11 0110XXXX 11000110 11110XXX XXXXXXXX XXXXXXXX
1LINE   ADDR          META ASSEMBLER ASSEMBLY PROG              PAGE  31

1100  00109 ;
1101  00109       LDOUT &
1102  00109 /     CONT
          XXXXXXXX XXXXXXXX XXX1XXXX XXXXXXXX XXXXXXXX X1XX1110
          XXXXXX11 XXXXXXXX XXXXXXXX XXX1XXXX XXXXXXXX XXXXXXXX
1103  0010A ;
1104  0010A       CALADIMD RNODE, LOWOFSND & LMARBUF & ; calculate address
1105  0010A /     CONT
          XXXXXXXX XXXXXXXX XXX1XXXX XX10XXXX 1XXXXXXX XXXX1110
          XXXXXX11 XXXX0010 11000011 10110XXX 00000000 00000100
```

```
1106  0010B ;
1107  0010B        LMAR  &  WRITE  &                    ; write to &RNODE
1108  0010B /      RTN
         XXXXXXXX XXXXXXXX XXX1XXXX XXXXXXXX XXXXXX1X XX1X1010
         XXXXXX11 XXXXXXXX XXXXXXXX XXX1XXXX XXXXXXXX XXXXXXXX
1109  0010C ;
1110  0010C ;      end;
1111  0010C ;
1112  0010C ;end; (* INSERT *)
1113  0010C ;
1114  0010C ;
1115  0010C ;
1116  0010C ;
1117  0010C ;
1118  0010C ;
1119  0010C ;begin
1120  0010C ;
1121  0010C ;      Initialize 'start node' address and 'start stack' address
1122  0010C ;
1123  0010C START: IMMDLOAD  RNODEBEG, H#0000  &
1124  0010C /      CONT
         XXXXXXXX XXXXXXXX XXX1XXXX XX11XXXX XXXXXXXX XXXX1110
         XXXXXX11 XXXX1001 11000110 11100XXX 00000000 00000000
1125  0010D ;
1126  0010D        IMMDLOAD  RSTKBEG, H#00F0  &
1127  0010D /      CONT
         XXXXXXXX XXXXXXXX XXX1XXXX XX11XXXX XXXXXXXX XXXX1110
         XXXXXX11 XXXX1011 11000110 11100XXX 00000000 11110000
1128  0010E ;
1129  0010E        IMMDLOAD  RTOPSTK, H#00F0  &
1130  0010E /      CONT
         XXXXXXXX XXXXXXXX XXX1XXXX XX11XXXX XXXXXXXX XXXX1110
         XXXXXX11 XXXX1100 11000110 11100XXX 00000000 11110000
1131  0010F ;
1132  0010F ;      reset( data );
1133  0010F ;
1134  0010F ;      new( root, leaf );
1135  0010F ;
1136  0010F        REGLOAD  RFREENOD, RNEWPTR  &
1137  0010F /      CONT
         XXXXXXXX XXXXXXXX XXX1XXXX XX01XXXX XXXXXXXX XXXX1110
         XXXXXX11 10101101 11000110 11100XXX XXXXXXXX XXXXXXXX
1138  00110 ;
1139  00110        ADDNLOAD  RFREENOD, H#0002  & ; increment node storage offset
1LINE   ADDR          META ASSEMBLER ASSEMBLY PROG              PAGE  32

1140  00110 /      CONT
         XXXXXXXX XXXXXXXX XXX1XXXX XX10XXXX XXXXXXXX XXXX1110
         XXXXXX11 XXXX1010 11000011 10100XXX 00000000 00000010
1141  00111 ;
1142  00111        REGLOAD  RNEWPTR, RNODE  & ; RPNEW <- pointer from 'new'
1143  00111 /      CONT
         XXXXXXXX XXXXXXXX XXX1XXXX XX01XXXX XXXXXXXX XXXX1110
         XXXXXX11 11010010 11000110 11100XXX XXXXXXXX XXXXXXXX
1144  00112 ;
1145  00112 ;      root^.kind := leaf;
1146  00112 ;
1147  00112        PASSIMD  LEAF  &  LDOUTBUF  &        ; DOUT <- leaf
1148  00112 /      CONT
         XXXXXXXX XXXXXXXX XXX1XXXX XX11XXXX XX1XXXXX XXXX1110
         XXXXXX11 XXXXXXXX 11000110 11110XXX 00000000 00001111
1149  00113 ;
1150  00113        LDOUT  &
```

```
1151  00113 /     CONT
          XXXXXXXX XXXXXXXX XXX1XXXX XXXXXXXX XXXXXXXX X1XX1110
          XXXXXX11 XXXXXXXX XXXXXXXX XXX1XXXX XXXXXXXX XXXXXXXX
1152  00114 ;
1153  00114       CALADREG  RNODE & LMARBUF &          ; calculate address
1154  00114 /     CONT
          XXXXXXXX XXXXXXXX XXX1XXXX XX00XXXX 1XXXXXXX XXXX1110
          XXXXXX11 XXXX0010 11000100 10110XXX XXXXXXXX XXXXXXXX
1155  00115 ;
1156  00115       LMAR  &  WRITE  &                    ; write to &RNODE
1157  00115 /     CONT
          XXXXXXXX XXXXXXXX XXX1XXXX XXXXXXXX XXXXXX1X XX1X1110
          XXXXXX11 XXXXXXXX XXXXXXXX XXX1XXXX XXXXXXXX XXXXXXXX
1158  00116 ;
1159  00116 ;     root^.element.key := 5;
1160  00116 ;
1161  00116       PASSIMD  H#0005 & LDOUTBUF &         ; DOUT <- 5
1162  00116 /     CONT
          XXXXXXXX XXXXXXXX XXX1XXXX XX11XXXX XX1XXXXX XXXX1110
          XXXXXX11 XXXXXXXX 11000110 11110XXX 00000000 00000101
1163  00117 ;
1164  00117       LDOUT  &
1165  00117 /     CONT
          XXXXXXXX XXXXXXXX XXX1XXXX XXXXXXXX XXXXXXXX X1XX1110
          XXXXXX11 XXXXXXXX XXXXXXXX XXX1XXXX XXXXXXXX XXXXXXXX
1166  00118 ;
1167  00118       CALADIMD  RNODE, ELEMENT  & LMARBUF  &
1168  00118 /     CONT
          XXXXXXXX XXXXXXXX XXX1XXXX XX10XXXX 1XXXXXXX XXXX1110
          XXXXXX11 XXXX0010 11000011 10110XXX 00000000 00000001
1169  00119 ;
1170  00119       LMAR  &  WRITE  &     ; write to element of &RNODE
1171  00119 /     CONT
          XXXXXXXX XXXXXXXX XXX1XXXX XXXXXXXX XXXXXX1X XX1X1110
          XXXXXX11 XXXXXXXX XXXXXXXX XXX1XXXX XXXXXXXX XXXXXXXX
1172  0011A ;
1173  0011A ;     Printtree( root );
LINE   ADDR          META ASSEMBLER ASSEMBLY PROG              PAGE  33

1174  0011A ;
1175  0011A ;(*
1176  0011A ;     read( data, numofdata );
1177  0011A ;     for i := 1 to numofdata do
1178  0011A ;     begin
1179  0011A ;          readln( data, newp.key );
1180  0011A ;*)
1181  0011A ;          INSERT( 7, root );
1182  0011A ;
1183  0011A       IMMDLOAD  RX, H#0007  &
1184  0011A /     CONT
          XXXXXXXX XXXXXXXX XXX1XXXX XX11XXXX XXXXXXXX XXXX1110
          XXXXXX11 XXXX0001 11000110 11100XXX 00000000 00000111
1185  0011B ;
1186  0011B       JSB  , INSERT
          XXXXXXXX XXXXXXXX XXX1XXXX XXXXXXXX XXXXXXXX XXXX0001
          XXXXXX11 XXXXXXXX XXXXXXXX XXX1XXXX XXXX0000 11101111
1187  0011C ;
1188  0011C ;          INSERT( 3, root );
1189  0011C ;
1190  0011C       IMMDLOAD  RX, H#0003  &
1191  0011C /     CONT
          XXXXXXXX XXXXXXXX XXX1XXXX XX11XXXX XXXXXXXX XXXX1110
          XXXXXX11 XXXX0001 11000110 11100XXX 00000000 00000011
```

```
1192  0011D  ;
1193  0011D        JSB  , INSERT
          XXXXXXXX XXXXXXXX XXX1XXXX XXXXXXXX XXXXXXXX XXXX0001
          XXXXXX11 XXXXXXXX XXXXXXXX XXX1XXXX XXXX0000 11101111
1194  0011E  ;
1195  0011E  ;             INSERT( 2, root );
1196  0011E  ;
1197  0011E        IMMDLOAD  RX, H#0002  &
1198  0011E  /     CONT
          XXXXXXXX XXXXXXXX XXX1XXXX XX11XXXX XXXXXXXX XXXX1110
          XXXXXX11 XXXX0001 11000110 11100XXX 00000000 00000010
1199  0011F  ;
1200  0011F        JSB  , INSERT
          XXXXXXXX XXXXXXXX XXX1XXXX XXXXXXXX XXXXXXXX XXXX0001
          XXXXXX11 XXXXXXXX XXXXXXXX XXX1XXXX XXXX0000 11101111
1201  00120  ;
1202  00120  ;             INSERT( 4, root );
1203  00120  ;
1204  00120        IMMDLOAD  RX, H#0004  &
1205  00120  /     CONT
          XXXXXXXX XXXXXXXX XXX1XXXX XX11XXXX XXXXXXXX XXXX1110
          XXXXXX11 XXXX0001 11000110 11100XXX 00000000 00000100
1206  00121  ;
1207  00121        JSB  , INSERT
          XXXXXXXX XXXXXXXX XXX1XXXX XXXXXXXX XXXXXXXX XXXX0001
          XXXXXX11 XXXXXXXX XXXXXXXX XXX1XXXX XXXX0000 11101111
1208  00122  ;
1209  00122  ;(*
1210  00122  ;             writeln;       writeln( 'INSERT ', newp );
1211  00122  ;             Printtree( root );
1LINE   ADDR          META ASSEMBLER ASSEMBLY PROG                PAGE  34

1212  00122  ;       end;
1213  00122  ;*)
1214  00122  ;
1215  00122  ;end.
1216  00122  ;
1217  00122        JMP  , QUIT  &  WDDEF   ; Stop the program
          XXXXXXXX XXXXXXXX XXX1XXXX XXXXXXXX XXXXXXXX XXXX0011
          XXXXXX11 XXXXXXXX XXXXXXXX XXX1XXXX XXXX0011 11111111
1218  00123  ;             by jumping to the highest address.....
1219  00123  ;
1220  00123  ;
```

BIBLIOGRAPHY

BIBLIOGRAPHY

[AGE76]
T. Agerwala, "Microprogram Optimization: A Survey," *IEEE Transactions on Computers*, Vol. C-25, No. 10, October 1976, pp. 962-973.

[AHO77]
A. V. Aho, and J. D. Ullman, "Principles of Compiler Disign," Reading, MA: Addison-Wesley, 1974.

[BAN79]
U. Banerjee, D. J. Kuck, and R. A. Towle, "Time and Parallel Processor Bounds for Fortran-Like Loops," *IEEE Transactions on Computers*, Vol. C-28, No. 9, September 1979, pp. 660-670.

[DAS76]
S. Dasgupta, and J. Tartar, "The Identification of Maximal Parallelism in Straight Line Microprograms," *IEEE Transactions on Computers*, Vol. C-25, No. 10, October, 1976, pp. 986-991.

[DAS79]
S. Dasgupta, "The Organization of Microprogram Stores," *ACM Computing Surveys*, Vol. 11, March 1979, pp. 39-65.

[DAS80]
S. Dasgupta, "Some Aspects of High Level Microprogramming," *ACM Computing Surveys*, Vol. 12, No. 3, September 1980, pp. 295-323.

[DAV78]
S. Davidson, and B. D. Shriver, "An Overview of Firmware Engineering," *Computer*, Vol. 11, No. 5, May 1978, pp. 21-33.

[DAV81]
S. Davidson, D. Landskov, B. D. Shriver, and P. W. Mallett, "Some Experiments in Local Microcode Compaction for Horizontal Machine," *IEEE Transactions on Computers*, Vol. C-30, No. 7, July 1981, 460-477.

[DEW76]
D. J. DeWitt, "A Machine Independent Approach to the Production of Optimal Horizontal microcode," Ph. D. Dessertation, The University of Michigan, August 1976.

[FIS79]
J. A. Fisher, "The Optimization of Horizontal Microcode within and beyond Basic Blocks: Application of Processor Scheduling with Resources," Ph. D. Dissertation, New York University, New York City, October 1979.

[FIS81a]
J. A. Fisher, "Trace Scheduling: A Technique for Global Microcode Compaction," *IEEE Transactions on Computers*, Vol. C-30, No. 7, July 1981, 478-490.

[FIS81b]
J. A. Fisher, D. Landskov, and B. D. Shriver, "Microcode Compaction: Looking backward and Looking forward," *Proceedings National Computer Conference*, 1981, pp. 95-102.

[FIS82]
J. A. Fisher, "Very Long Instruction Word Architectures and the ELI-512," Research Report #253, Yale University, Department of Computer Science, December 1982.

[ISO83]
S. Isoda, Y. Kobayashi, and T. Ishida, "Global Compaction of Horizontal Microprograms Based on the Generalized Data Dependency Graph," *IEEE Transactions on Computers*, Vol. C-32, No. 10, October 1983, pp. 922-933.

[LAH83]
J. Lah and D. E. Atkins, "Tree Compaction of Microprograms," *Proceedings the 16th Annual Microprogramming Workshop*, October 1983, pp. 11-22.

[LAN80]
D. Landskov, S. Davidson, B. Shriver, and P. W. Mallett, "Local Microcode Compaction Techniques," *ACM Computing Surveys*, Vol. 12, No. 3, September 1980, pp. 261-294.

[LIN83]
J. L. Linn, "SRDAG Compaction - A Generalization of Trace Scheduling to Increase the Use of Global Context Information," *Proceedings the 16th Annual Microprogramming Workshop*, October 1983, pp. 11-22.

[MAL78]
P. W. Mallett, "Methods of Compacting Microprograms," Ph. D. Thesis, University of Southwestern Louisiana, December 1978.

[MYE81]
G. J. Myers, and D. G. Hocker, "The Use of Software Simulators in the Testing and Debugging of Microprogram Logic," *IEEE Transactions on Computers*, Vol. C-30, No. 7, July 1981, pp. 519-523.

[NIC84]
A. Nicolau, Personal communication.

[PAR81]
A. C. Parker, and W. T. Wilner, "Microprogramming - The Challenge of VLSI," *Proceedings National Computer Conference*, 1981, pp. 63-68.

[PAT76]
D. A. Patterson, "STRUM: Structured Programming System for Correct Firmware," *IEEE Transactions on Computers*, Vol. C-25, No. 10, October 1976, pp. 974-985.

[RAM74]
C. V. Ramamoorthy, and M. Tsuchiya, "A High-Level Language for Horizontal Microprogramming," *IEEE Transactions on Computers*, Vol. C-23, No. 8, August 1974, pp. 791-801.

[ROB79]
E. L. Robertson, "Microcode Bit Optimization is NP-complete," *IEEE Transactions on Computers*, Vol. C-28. No. 4, April 1979, pp. 316-319.

[SAL76]
A. B. Salisbury, "Microprogrammable Computer Architecture," American Elsevier Publishing Co., Inc., 1976.

[SIN80]
M. Sint, "A Survey of High Level Microprogramming Language," *Proceedings 13th Annual Microprogramming Workshop*, November 1980, pp. 141-153.

[TOK78]
M. Tokoro, T. Takizuka, E. Tamura, and I. Yamaura, "A Technique of Global Optimization of Microprograms," *Proceedings 11th Annual Microprogramming Workshop*, 1978,

[TOK81]
M. Tokoro, E. Tamura, and T. Takizuka, "Optimization of Microprograms,"

IEEE Transactions on Computers, Vol. C-30, No. 7, July 1981, pp. 491-504.

[VEG82]
S. R. Vegdahl, "Local Code Generation and Compaction in Optimizing Microcode Compilers," Ph. D. Thesis, Carnegie-Mellon University, December 1982.

[VEG83]
S. R. Vegdahl, "A New Perspective on the Classical Microcode Compaction Problem," *ACM SIGMICRO Newsletter*, Vol. 14, No. 1, March 1983, pp. 11-14.

[WIL51]
M. V. Wilkes, "The Best Way to Design an Automatic Calculating Machine," In Manchester University Computer Inaugural Conference, Ferrante, London, July 1951.

[WOO78]
G. Wood, "On the packing of Micro-operations into Micro-instruction Words," *Proceedings the 11th Annual Microprogramming Workshop*, 1978,

[WOO79a]
W. G. Wood, "The Computer Aided Design of Microprograms," Ph. D. Thesis, University of Edinburgh, Scotland, 1979.

[WOO79b]
W. G. Wood, "Global Optimization of Microprograms Through Modular Control Constructs," *Proceedings the 12th Annual Microprogramming Workshop*, 1979, pp. 1-6.

[YAU74]
S. S. Yau, A. C. Schowe, and M. Tsuchiya, "On Stroage Optimization of Horizontal Microprograms," *Proceedings 7th Annual Workshop on Microprogramming*, October 1974, pp. 98-106.