PLACEMENT OF MICROINSTRUCTIONS IN A TWO-DIMENSIONAL ADDRESS SPACE

John F. Wakerly, Clifford R. Hollander*, and Daniel Davies
Digital Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, California

1. INTRODUCTION

The problem of addressing a large address space with a limited number of address bits is an old one [Bell and Newell, 1971]. The problem occurs in minicomputers which, with their short instruction word lengths of 12 to 16 bits, allocate only 8 to 12 bits of each instruction for specifying operand addresses. Hence each instruction can directly access only 2^8 to 2^{12} words of memory. In order to access the remainder of a large (say 2^{16} word) memory, mechanisms such as bank switching, page registers, base registers, zero/current page addressing, and indirection have been used. Each of these mechanisms allows a subset of the entire address space to be accessed easily while requiring some additional overhead to access the remainder.

A similar problem has arisen in the development of integrated circuit microprogram sequencers. A microprogram sequencer is a circuit which, along with microprogram memory, comprises a microprogram control unit. As a minimum, a sequencer must contain a microprogram address register and logic to update it. The output of a sequencer is a microprogram address which is used as the input of a microprogram memory. Each microinstruction (contained in the memory) contains various fields used by the controlled machine (e.g., computer processor) and also a specification for how the address of the next microinstruction is to be computed. This specification, along with status information from the controlled machine, is used by the sequencer to compute the next microprogram address.

A typical microprogram control unit has between 2^8 and 2^{10} microinstructions, requiring a microprogram address of 8 to 10 bits. Most microprogram control units have an 8 to 10 bit microprogram counter that is normally incremented to get the next address. Conditional action can be implemented with "skip" instructions that add either one or two to the counter depending on a machine status bit. Thus a microprogram sequencer would require only 4 bits to specify whether the microprogram address was to be incremented or whether one of 15 different conditional skips was to take place. However, when unconditional jumps such as the jump back to the beginning of a loop are to be executed, the entire next address must be loaded into the microprogram address register from a field in the microprogram. Hence at least 9 to 11 bits are normally used to specify the next address: 1 bit specifies an unconditional jump or not, and 8 to 10 bits give the target address of the unconditional jump or further specify an increment or conditional skip.

This work was sponsored by the Joint Services Electronics Program under contract N00014-75-C-0601.

*The author is now with the IBM Scientific Center, Palo Alto, Calif. 94304.

It is desirable to minimize the number of next-address bits for two reasons. First, this reduces the total number of microprogram bits since a next address must be specified in every microinstruction. Second, it reduces the number of inputs that must be provided for an integrated circuit microprogram sequencer; this is very important because of the rapid growth of cost and unreliability with the number of input/output pins of integrated circuits.

To reduce the number of next-address bits a two-dimensional addressing scheme is used in the Intel 3001 microprogram sequencer [Rattner et. al., 1975]. This scheme, described in the next section, allows the next address and the conditional or unconditional jump type in a 2^9-word address space to be specified using only 7 bits. Extensions of this scheme to higher dimensions could allow the next address in a 2^{16}-word space to be specified with as few as 5 bits. However, the reduction in the number of next-address bits is paid for by a corresponding decrease in addressing flexibility. The reduction in the number of next-address bits is obtained by limiting the number of addresses that can be reached in one step from any particular address. It thus becomes possible for a programmer to write symbolic (assembly language) programs for which there is no valid assignment in the two-dimensional address space. Although there usually is a valid assignment for a "well-behaved" program, it takes many hours for a programmer to make the assignment by hand.

The work described in this paper is an investigation of procedures for the automatic placement of symbolic programs in a two-dimensional address space such as that of the Intel 3001 microprogram sequencer. By obtaining results for the specific case of the Intel 3001 we are able to make recommendations for the use of multidimensional address spaces in future designs.

2. A TWO-DIMENSIONAL ADDRESSING SCHEME

The address space of the Intel 3001 microprogram sequencer is organized as a two-dimensional array with 32 rows and 16 columns. Hence there are a total of $512 = 2^9$ cells, which are accessed by a 9-bit address. The high-order 5 bits of the address give the row number and the low-order 4 bits give the column. There is no microprogram counter; each microinstruction specifies a conditional or unconditional jump. At the end of each microinstruction execution cycle the address register is updated as a function of a 7-bit jump code $\langle d_6 d_5 d_4 d_3 d_2 d_1 d_0 \rangle$ in the microinstruction, the current address register contents $\langle m_8 m_7 m_6 m_5 m_4 m_3 m_2 m_1 m_0 \rangle$ in the sequencer, and certain condition bits ($x_7, x_6, x_5, x_4, p_3, p_2, p_1, p_0, f, c, z$). The high-order bits of the jump code specify a jump type and the low-order bits specify new values for a subset of the address register bits, as detailed in Table 1.

The first four entries in Table 1 are unconditional jumps. They allow the next instruction to be located anywhere in the same row or column as the current instruction, or in the zero row. (Note that JCE is

Table 1 Jump types of Intel 3001.

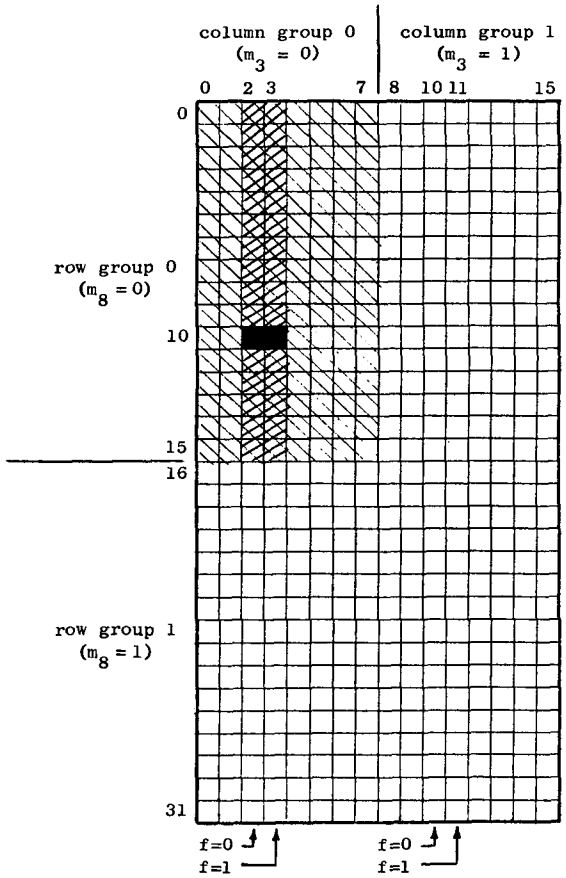
MNEMONIC	DESCRIPTION	JUMP CODE							NEXT ROW					NEXT COLUMN				
		d_6	d_5	d_4	d_3	d_2	d_1	d_0	M_8	M_7	M_6	M_5	M_4	M_3	M_2	M_1	M_0
JCC	Jump in current column	0	0	-	-	-	-	-	d_4	d_3	d_2	d_1	d_0	m_3	m_2	m_1	m_0
JZR	Jump to zero row	0	1	0	-	-	-	-	0	0	0	0	0	d_3	d_2	d_1	d_0
JCR	Jump in current row	0	1	1	-	-	-	-	m_8	m_7	m_6	m_5	m_4	d_3	d_2	d_1	d_0
JCE	Jump in column/enable	1	1	1	0	-	-	-	m_8	m_7	d_2	d_1	d_0	m_3	m_2	m_1	m_0
JFL	Jump/test F-latch	1	0	0	-	-	-	-	m_8	d_3	d_2	d_1	d_0	m_3	0	1	f
JCF	Jump/test C-flag	1	0	1	0	-	-	-	m_8	m_7	d_2	d_1	d_0	m_3	0	1	c
JZF	Jump/test Z-flag	1	0	1	1	-	-	-	m_8	m_7	d_2	d_1	d_0	m_3	0	1	z
JPR	Jump/test PR-latches	1	1	0	0	-	-	-	m_8	m_7	d_2	d_1	d_0	p_3	p_2	p_1	p_0
JLL	Jump/test left PR bits	1	1	0	1	-	-	-	m_8	m_7	d_2	d_1	d_0	0	1	p_3	p_2
JRL	Jump/test right PR bits	1	1	1	1	1	-	-	m_8	m_7	1	d_1	d_0	1	1	p_1	p_0
JPX	Jump/test PX bus	1	1	1	1	0	-	-	m_8	m_7	m_6	d_1	d_0	x_7	x_6	x_5	x_4

similar to JCC except for a side effect that is of no interest here.) Since these are the only unconditional jumps, the successor of any instruction must be located in the same row, the same column, or row zero, unless the instruction is a conditional jump.

The last seven entries in Table 1 are conditional jumps. JFL, JCF, and JZF are two-way jumps based on a single status bit, f, c, or z. JLL and JRL are four-way jumps based on two status bits, and JPR and JPX are 16-way jumps based on four status bits.

A JFL instruction placed in a particular row group of the memory as defined by bit 8 of the current address will jump to targets in the same row group; that is, bit 8 of the next address will be the same as bit 8 of the current address. The row within the group is determined by $\langle d_3 d_2 d_1 d_0 \rangle$. The target column will be in the same column group as the jump instruction, as defined by $\langle m_3 \rangle$. The column within the group is determined by the status bit f. If the current instruction is in column group 0, the jump goes to column 2 if f=0 and to column 3 if f=1; if the current instruction is in column group 1, the target is in column 10 if f=0 and in column 11 if f=1. These constraints are illustrated in Fig. 1.

The word descriptions of the other conditional jump types are equally verbose, but each jump type is succinctly described by its entry in Table 1. Further explainations of the jump types of the Intel 3001 may be found in [Intel, 1974]. For our purposes, it is sufficient to observe that (1) the targets of unconditional jumps must be in the same row or column as the jump, or in row zero, and (2) the targets of conditional jumps are constrained to lie within the same row and column groups as the jumps themselves, and the target columns within the column groups are fixed. These constraints influenced the basic approach of the placement procedure.



-- Possible locations for JFL in row group 0, column group 0.

-- Possible locations of targets of JFL in row group 0, column group 0.

-- Locations of targets of JFL in row group 0, column group 0 if $\langle d_3 d_2 d_1 d_0 \rangle = \langle 1010 \rangle$.

Fig. 1 Placement of JFL and targets.

3. THE PLACEMENT PROBLEM

An assembly language programmer writes symbolic programs, assigning labels to instructions that are referenced by other instructions, without regard to the addresses in memory to which the instructions will ultimately be assigned. The assignment of memory addresses to instructions of the symbolic program is a straightforward process for most computers (for example, see [Stone, 1972]). For microprograms written for the Intel 3001 sequencer, however, the restrictions imposed on the jump structure by the two-dimensional addressing scheme make the assignment process non-trivial.

The symbolic code written by the programmer may be in a conventional assembly language, as illustrated by the program fragment of Fig. 2. From the viewpoint of placement, a program can be completely characterized by its jump structure. Hence, in the figure (and in our procedures) we suppress all operator/operand details not relevant to explicit jumps. As is the usual practice, program control is assumed to flow sequentially through the list of instructions, except when an explicit jump is indicated. The programmer indicates explicit unconditional jumps by "JMP <LABEL>" and conditional jumps by "<JUMP TYPE> <LABEL LIST>". Since all Intel 3001 instructions must specify a jump (there is no program counter), the placement procedure must specify jumps wherever the programmer has left the program flow implicit. Also, the procedure must determine the type of unconditional jump (in current column, current row, or zero row) to be used when the programmer has specified "JMP," and it must assign memory addresses to all instructions such that the desired program flow can take place within the constraints of the two-dimensional addressing scheme.

4. OVERVIEW OF THE PLACEMENT PROCEDURE

Algorithmic approaches that guarantee to find a valid program placement if one exists tend to be combinatorial in either their cell selection or backtracking mechanisms. Therefore we have devised a heuristic placement procedure that operates in two phases, each with several distinct stages. The procedure works on a "best effort" basis and is capable of taking advice in the form of an initial partial placement. Both phases must terminate successfully in order to achieve a placement; certain types of stage failures, however, can be tolerated. A limited amount of backtracking is performed. The procedure attempts to maintain maximum flexibility by making the most difficult placements first and by leaving uncommitted the coordinates for as many non-jump/non-target instructions as possible.

The two phases of the placement procedured are:

Phase I) Place explicit jump instructions and their targets.

Phase II) Place instructions not involved in explicit jumps.

The instructions to be placed in each phase are determined by an analysis of the jump structure of the program which partitions the program into basic blocks. Each basic block is a maximal length instruction sequence with a single entry point and a single exit point [Allen, 1970]. For example, Fig. 2 indicates the basic blocks of the example program. With respect to control flow each basic block can be treated as a single atomic unit, and therefore the control flow of the example program can be completely described by the flow graph of Fig. 3. After analysis of the block structure of the program, the placement procedure can be described as:

Phase I) Place the first and last instruction of each basic block.

Phase II) Place the internal instructions of each basic block.

5. PHASE I

Phase I places the first and last instructions of each basic block. Since the run time of a program that tried every possible placement for every instruction before giving up could best be measured in years, we instead have a procedure with many heuristics that will take care of almost all cases, leaving the microprogrammer to help in the really difficult spots. He does this by placing selected instructions by hand, enabling the procedure to make the remaining placements correctly.

```
          :
LOOP:     ---
          :                          1
          JFL C1,C2
C1:       ---
          :                          2
          JPX L1,L2,...,L16
L1:       ---
          :                          3
          JMP JOIN
L2:       ---
          :                          4
          JMP JOIN
L3:       ---
          :                          5
          :                          :
          :                          17
          JMP JOIN
L16:      ---
          :                          18
JOIN:     ---
          :                          19
          JMP TEST
C2:       ---
          :                          20
          JCF C2,TEST
TEST:     JZF LOOP,NEXT              21
NEXT:     ---
          :                          22
          JMP LOOP
          :
```

Fig. 2 A program fragment.

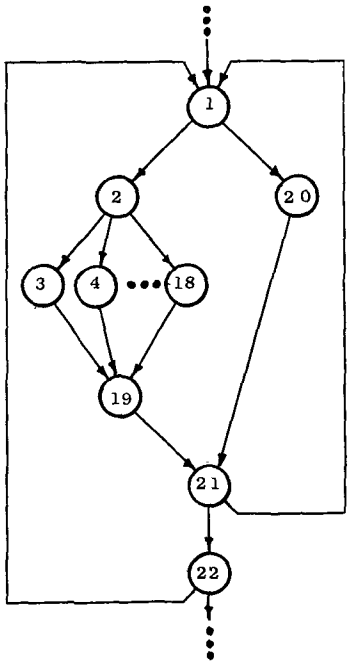


Fig. 3 Control flow graph.

Definitions:

Target set (t-set) -- The set of all the targets of a jump. An unconditional jump has one target, a conditional jump has more. The targets of a conditional jump must all be placed in a single row, the columns being set by the type of conditional jump and the conditions.

Reference -- A jump "references" its targets.

The operation of phase I is divided into several stages.

Stage 1

The microprogram is scanned and its jump structure is extracted. The locations of any instructions that have been hand-placed are noted.

Stage 2

"Families" of instructions are found; two instructions are in the same family if they are connected in the control flow graph by a set of basic blocks each containing only a conditional jump instruction. The targets of conditional jumps in the Intel 3001 are restricted to the same four- or eight-row group as the jump itself. Since in a family no unconditional jumps intervene to allow spanning the groups, all instructions of a family must be in the same group. Once found, the families are assigned to row groups as follows:

a) First, families containing hand-placed instructions are assigned to the proper row-group.

b) Second, families containing instructions explicitly referenced by JZR's are assigned to the zero row group.

c) Third, the remaining families are ordered according to the number of instructions contained in each. Beginning with the largest, each family is then assigned to the row group which is least occupied.

Stage 3

The t-sets are ordered according to the number of targets in each, and beginning with the largest they are placed as follows:

a) If the jump to the t-set has already been placed, the area in which the t-set may be placed is restricted appropriately.

b) If any element of the t-set is referenced by a JZR, the t-set bounds are restricted to row zero.

c) If any element of the t-set has been placed, either the t-set is placed over the placed element(s), or an error condition is signaled. No instructions are duplicated.

d) If still necessary, the t-set will be placed at the first available place within its bounds; if no place is found an error condition is signaled.

e) If the t-set was placed and its jump was not, the bounds for the jump are set.

No attempt is ever made to "un-place" a target set on an error. Again, the complexity of a procedure to determine which t-set to remove, how to remove it and all the instructions that depend upon it, and what intelligent action to take at that point, is prohibitive.

Stage 4

A check is now made to verify that all unconditional jumps between placed instructions may be satisfied.

Stage 5

Finally, all of the remaining instructions that start or end a block are placed. These may be conditional and unconditional jumps and unconditional jump targets. The order in which these are placed is determined by the number of references for each, with the most heavily constrained placed first.

6. PHASE 2

Phase 2 places the internal cells of each basic block. Phase 1 passes to phase 2 a list of basic blocks which specifies for each block the starting and the ending instructions (which have already been placed) and the number of internal instructions which must be placed. The task of phase 2 is to link up the starting and ending cells of all the basic blocks with the appropriate number of unconditional jumps to the current column, current row, or zero row.

When phase 2 takes over, the memory array is partially filled, and at the end of phase 2 the memory array will be completely filled if the microprogram is large. Therefore placements become increasingly difficult as the execution of this phase progresses. Our initial solution of the phase 2 placement was as follows:

a) If there are n internal instructions to be placed in a block, place the first n-2 of them one-at-a-time using the following heuristic: At each step choose a reachable target cell that best balances the current filling of the array according to a scoring function.

b) Place the last 2 internal instructions of the block by choosing the best (according to the scoring function) of all possible placements from the last cell placed in step (a) to the end cell of the block.

Although this technique was successful in balancing the assignment of instructions to the memory array, it still left insufficient flexibility for placing the last blocks in a program and many internal cells could not be placed in programs that used most of the memory space. So we elected a different approach that was much more successful. This approach relies on the notion of pivot cells, illustrated in Fig. 4. In Fig. 4, it is desired to link instruction a with instruction b using n steps. A pivot cell is an instruction whose predecessor and successor are in neither the same row nor the same column. Fig. 4 shows four different ways of linking up cells a and b using either one or two pivot cells (p_1 and p_2). The important thing to notice here is that once the pivot cells have been fixed, the remaining instructions may be placed anywhere in a particular column or columns. For example, in Fig. 4 (a) and (b), n-1 instructions must be placed in addition to the instruction placed at the pivot cell p_1, but these instructions may be placed anywhere in p_1's column and reference each other using JCC's. Hence once the pivot instruction has been placed, placement of the remaining instructions can be deferred until later, as long as the free cells of the pivot column are not overcommitted.

Hence the operation of Phase 2 can be described as follows:

Stage 1

Order the blocks according to length and place the longest blocks first. For each block examine each of the possible pivoting schemes of Fig. 4 in the order shown. Choose the first pivoting scheme that does not overcommit the number of free cells remaining in the pivot column(s). Update the number of committed (but yet unplaced) free cells remaining in the pivot column(s). If there is no pivoting scheme that will not overcommit the free cells, take another pass without regard to the number of free cells available. Update the number of committed free cells to show the overcommitment - we will make room later. If no pivots can be found mark the block not placed for stage 2.

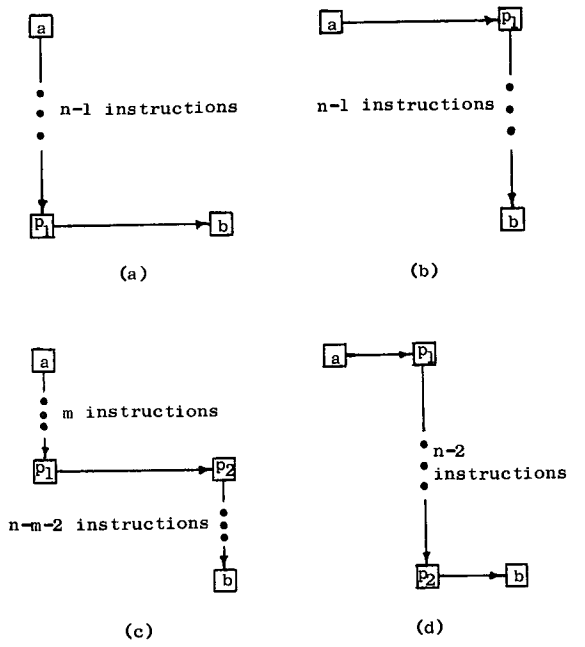


Fig. 4 Placement of n instructions between a and b.

Stage 2

At this point there may be some blocks for which no pivot cells were found. For each such block find a block placed in stage 1 that can be unplaced to make placement of the problem block possible. Place the problem block and mark it so that it will not be unplaced later. Now look for different placements of the blocks that were just unplaced. If any blocks cannot be replaced, iterate stage 2 a few more times and signal an error if unplaceable blocks still remain.

Stage 3

Now pivot cells for all blocks have been found, but the free cells of some columns may have been overcommitted. If the number of instructions in the program does not exceed the size of the memory array, there will be an undercommitment of free cells in other columns to balance the overcommitment. Try to balance the free cell committment by changing the values of m in blocks that were placed according to Fig. 4(c). An optimum balance can be found in a short time using a simple recursive algorithm.

Stage 4

There may still be overcommitted columns. Find all of the pivot cells in each overcommitted column. Each pivot cell references or is referenced by one cell in its column and one cell in its row. If there are any uncommitted free cells in the pivot row, try to "slide" overcommitted cells from the pivot column into the pivot row.

Stage 5

Assign instructions to the committed free cells in each column. If stage 4 was successful, there will be sufficient free cells to meet the commitments in each column.

7. PERFORMANCE

Phase 2 was programmed and tested first, using a program that generated random blocks to be placed. The inputs to the test program were a list of block lengths and the number of blocks of each length to be generated. The blocks were generated with starting and ending cells at random locations in the memory array, and the phase 2 program attempted to make a complete placement of the internal cells of the blocks. Eighteen different placements were tested, with varying block sizes and numbers of blocks which filled almost the entire 512-word memory array. The results of these tests are given in Table 2. The table gives for each test the number of blocks of each length used, where the length is the number of internal instructions of the block to be placed. It also gives the number of blocks for which no pivots had been found after stage 2, and the free cell overcommitment remaining after stage 4. It can be seen that the procedure worked best when the block sizes were large. This is an expected result since large blocks leave many cells uncommitted in stage 1 for greatest flexibility. In the cases where unplaced blocks and cell overcommitments occurred, it was possible to adjust the placement by hand in about 5 to 10 minutes. The procedures for doing this could be automated as a set of special-purpose routines to follow stages 2 and 4 if desired.

Phase I was programmed and tested second, mainly because it was the most difficult part of the procedure. This phase was tested using an example program given by Intel [Intel, 1975]. The example is a microprogram for a 16-bit minicomputer instruction set. The program requires 254 words of memory, and is composed mainly of jumps and jump targets since a lot of decoding and testing takes place; there are 176 jumps and jump targets and only 78 instructions internal to basic blocks. Two tests were run with this program, the first placing it in a 512-word array, and the second placing it in a 256-word array (16 rows by 16 columns, an Intel 3001 option). In both cases the programmer running the phase 1 placement routine required a number of iterations, hand-placing difficult instruction sequences after each try, and introducing both errors and new placement difficulties at each iteration until the last. Placement of the program in a 512-word array required two iterations and about 10 instructions to be placed

Table 2 Phase 2 performance.

TEST#	BLOCK LENGTH - NUMBER	TOTAL # OF INSTRUCTIONS	UNPLACED BLOCKS	CELL OVER-COMMITMENT
1	2-30,3-36,4-35	510	0	0
2	"	"	0	0
3	"	"	0	2
4	2-30,5-30,20-7	510	0	0
5	"	"	0	3
6	"	"	0	0
7	5-30,7-20,8-12	510	0	0
8	"	"	0	0
9	"	"	0	0
10	3-102	510	0	0
11	"	"	0	0
12	"	"	0	0
13	2-128	512	1	1
14	"	"	0	0
15	"	"	3	0
16	"	"	3	0
17	"	"	2	0
18	"	"	0	2

by hand, and placement in a 256-word array required five iterations and about 16 instructions to be placed by hand. It was apparent that an on-line iteractive program would be the best solution for phase 1, although phase 1 could be expected to perform much better for microprograms with a less complex jump structure.

Both phases were programmed in ALGOLW and executed on an IBM 360/67. Phase 1 typically required 7 seconds of execution time and phase 2 required 1 second. At the time of this writing, the Intel program was the only real source program we had access to for testing - we would welcome receiving programs from others to complete the testing of the procedures.

8. CONCLUSION

We have developed a procedure for the placement of microprograms in a two-dimensional address space. The procedure has two phases -- placement of the starting and ending instructions of basic blocks and placement of instructions internal to basic blocks. The performance of both phases is sensitive to the number and size of the basic blocks of the program. A large number of basic blocks indicates a complex jump structure, which hampers phase 1. Small basic blocks have few internal cells that can be left uncommitted to improve flexibility in phase 2, and hence degrade its performance.

Thus we conclude that the placement of microprograms, such as instruction set emulators, that have a complex jump structure are difficult to perform either by hand or by automatic procedure. On the other hand, our procedures work quite well for programs having a ratio of jump/target to other instructions of 1 to 2 or better. For such programs automatic placement in a two-dimensional address space is not difficult, and in fact one can propose higher-dimension spaces for saving even more address bits. For example, a 64K (2^{16}) word address space could be addressed as an 8-dimensional array with length 4 in each dimension. Jumps could be specified with five bits -- three to give the coordinate to change and two to give the new value of the coordinate.

REFERENCES

Allen, F. E., 1970, "Control flow analysis," Proc. Symp. Compiler Optim., ACM SIGPLAN (1970), pp. 1-19.

Intel, 1974, "3001 microprogram control unit," data sheets, Intel Corp., Santa Clara, Calif.

Intel, 1975, "Central processor design using the Intel series 3000 computing elements," Application Note AP-16, Intel Corp., Santa Clara, Calif.

Rattner, J., J.-C. Cornet, and M. E. Hoff, 1974, "Bipolar LSI computing elements usher in new era of digital design," Electronics, vol. 47, no. 18, Sept. 5, 1974.

Stone, H. S., 1972, Introduction to Computer Organization and Data Structures, New York: McGraw-Hill.