Edgar H. Sibley
Panel Editor

Data flow analysis is used to optimize variable definitions in a program that translates microprocessor object code to a higher order language.

A DEFINITION OPTIMIZATION TECHNIQUE USED IN A CODE TRANSLATION ALGORITHM

DAVID M. DEJEAN and GEORGE W. ZOBRIST

One of the goals of code optimization in a translator is to reduce the code space needed for the translated code that represents the original code. In developing methods for code optimization, the subject can be divided into three interrelated areas:

(1) local optimizations, those performed within a straight–line block of code,
(2) loop optimizations, those performed on software loops, and
(3) data flow analysis, the transmission of useful relationships from parts of the program to places where the information can be used [1].

For the purpose of this article, data flow analysis is used in an attempt to optimize variable definitions in a translation program. An application of the analysis is presented in a program that translates microprocessor object code to a higher order language, with flags as the optimizing target typically inherent in any microprocessor instruction set.

DATA FLOW ANALYSIS DEFINITIONS

Certain basic concepts and constructs must first be defined. Data flow analysis begins by breaking the code into basic blocks—sequences of consecutive statements that may be entered only at the beginning—and when entered, are executed in sequence without halt or possibility of branch (except at the end of the basic block). Each basic block is terminated by a jump statement (conditional or non–conditional), an exit statement, or an input/output operation. Other blocks may be terminated due to the next sequential statement being a merge point of other basic blocks or the object of one or more jump statements. Grouping these basic blocks and connecting them result in a data flow graph which shows the possible data flow of the executing code. A cluster of basic blocks, typically a procedure or small program, can be called a control section, abbreviated CSECT, the first basic block called the initial block. Each basic block in a flow graph is considered a node, i.e., $N(1)$, $N(2)$, . . . , $N(\max)$. Figure 1 shows an example of a data flow graph containing several loops and exit points.

The terms *node* and *basic block* are interchangeable. The term node is typically used when referring to a data flow graph, and basic block is used when referencing the sequence of statements in the node. The immediate predecessor of a node $N(j)$ is any node directly preceding node $N(j)$ (i.e., there exists a path in the flow graph from the predecessor $N(p)$ directly to the node $N(j)$). Predecessors of node $N(j)$ are all nodes for which there exists any path, whether through other nodes or directly, to node $N(j)$. An immediate successor of node $N(j)$ is any node directly following $N(j)$ in the flow graph, that is, there exists a direct path from $N(j)$ to the immediate successor $N(s)$. Successors of $N(j)$ are all nodes for which there exists a path from $N(j)$, through other nodes or directly, to the successor $N(s)$ [2]. An exit node is any node that may conditionally or non-conditionally exit from the flow graph or CSECT.

At this point, data flow relationships can be more precisely defined. Every node consists of sequential statements that either use or modify data items. A data item is a variable that can be referenced or redefined in an expression. If an expression modifies a data item, it is called a data definition. Conversely, if an expression references a data item, it is called a data use. A local definition of a block is a definition within that block. If a data definition is made available to a data use, then the data definition is said to potentially affect the data

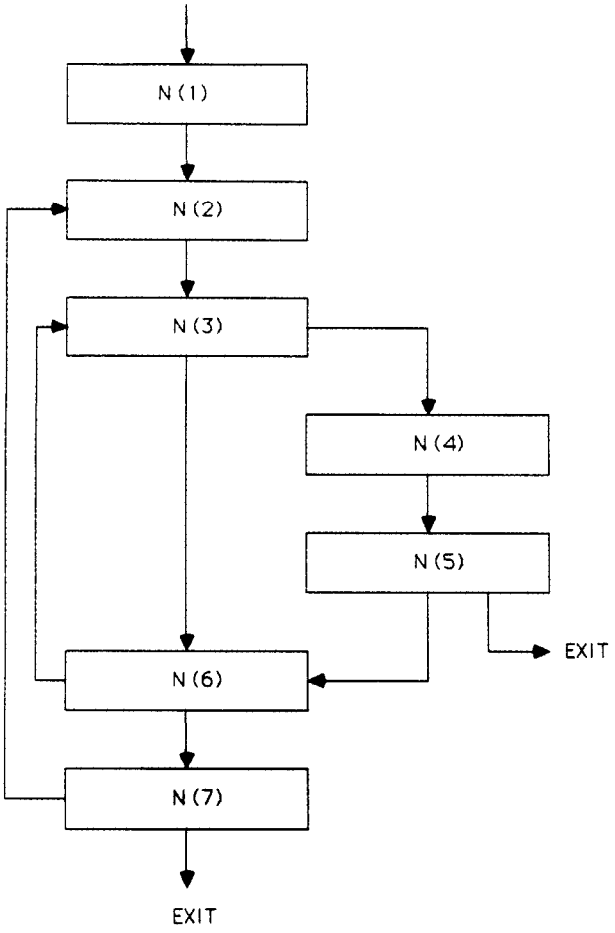

FIGURE 1. Data Flow Graph Example

use. The obvious example below shows data definitions and a data use:

$$X = Y$$

$$Z = X * 2$$

$$A = C/6$$

The data definition of X obviously affects the next statement, but does not affect the last statement. The term *potentially* is used since a data definition and a data use may be in separate basic blocks where flow of control is not yet known until the actual execution of the code. Also, a data definition reaches a point if there exists a path in the flow graph from the definition to that point, such that no other definitions of the data item appear on the path. A data definition is *active* or *live* at a point if it reaches that point.

GLOBAL DATA FLOW ANALYSIS AND ALGORITHM

The common objective of this optimization is to eliminate excessive or unnecessary definitions of variables in a code segment. An obvious elimination of a data definition can be performed where two data definitions exist in a group of sequential statements without any correlating use of the data between them. The first data definition can be eliminated from the code regardless of how the statement defines that data item.

When analyzing data definitions on a global basis for an entire CSECT, the analysis becomes much more complex. Manual analysis would require searching through every possible path from each data use back to every data definition. Algorithms exist that can be used to optimize data definitions.

In order to determine the definitions that are to remain in the code, it would be extremely useful to know which data definitions were active at the output of each node and which definitions were active at the input of each node. Given this information, a data definition can be labeled as required by data use provided the data definitions are referencing the same data item as the data use. For example, data definitions d3 and d5 are labeled as required for the data use in block 3 of the flow graph in Figure 2. The other definitions obviously can be discarded. The set of active definitions to the output of block 1 is d2, d3; d4, d5, d6 for block 2, and d7, d8 for block 3. The active set of definitions to the input of block 3 is the union of all the predecessors' active output data definitions. The set of active definitions at the output of block 3 is formed by gathering all the new data definitions in block 3 plus any active input data definitions not redefined by any statement in the block, plus the data definitions that redefine an already existing data definition. The redefining of a data item has killed any other previous active data definitions entering the block. For instance, the definition

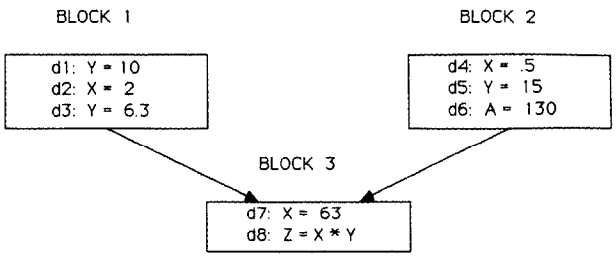

FIGURE 2. Sample Data Definitions and Data Uses

d7 in block 3 has killed definitions d2, d4. The active definitions at the output of block 3 include d3, d5, d6 simply because they appear at the input of block 3 and are not redefined within the block. Since d7 is the last definition for X before exiting block 3, it is active. Finally, d8 is active because it is a new definition within the block.

At this point, mathematical equations are needed to express the relationship between active input and active output definitions and to specify which definitions are killed and which are generated within each block. Bit arrays *OUT*, *IN*, *GEN*, and *KILL* shall be defined, and equations shall be developed leading to an algorithm to

optimize the number of data definitions. The *OUT* array can be defined as a group of *N* bit vectors, one bit for every data definition in the CSECT, one vector per basic block. Each bit set indicates that the data definition is active at the output of that block. Grouping the vectors together forms a two-dimensional array *N* bits wide by *M* blocks long comprising the *OUT* array. The *OUT* array for the previous example is defined as:

BLOCK	d1	d2	d3	d4	d5	d6	d7	d8
OUT(*BLK* 1) =	0	1	1	0	0	0	0	0
OUT(*BLK* 2) =	0	0	0	1	1	1	0	0
OUT(*BLK* 3) =	0	0	1	0	1	1	1	1

The *IN* array is of the same structure but for each bit set in the vector of a given block, it indicates an active definition to the input of that indicated block. For example:

BLOCK	d1	d2	d3	d4	d5	d6	d7	d8
IN(*BLK* 1) =	0	0	0	0	0	0	0	0
IN(*BLK* 2) =	0	0	0	0	0	0	0	0
IN(*BLK* 3) =	0	1	1	1	1	1	0	0

The *GEN* array, again the same structure, indicates which definitions have been generated within the block. The *GEN* array is:

BLOCK	d1	d2	d3	d4	d5	d6	d7	d8
GEN(*BLK* 1) =	1	1	1	0	0	0	0	0
GEN(*BLK* 2) =	0	0	0	1	1	1	0	0
GEN(*BLK* 3) =	0	0	0	0	0	0	1	1

Finally, each block also has a vector of *N* bits associated with it, one bit for every definition, indicating which definitions outside of the block can be killed by definitions of the same type within the block. Grouping these from each block forms the *KILL* array. For example, *KILL* is defined as:

BLOCK	d1	d2	d3	d4	d5	d6	d7	d8
KILL(*BLK* 1) =	0	0	0	1	1	0	1	0
KILL(*BLK* 2) =	1	1	1	0	0	0	1	0
KILL(*BLK* 3) =	0	1	0	1	0	0	0	0

Now an equation can be written describing all active inputs to each block as being the union of all predecessors' active output list. Let *IN*(*b*) = *UNION OUT*(*p*), where *b* is the block number and *p* is every predecessor of block *b* [2]. Furthermore, the *OUT* array can be found to be: *OUT*(*b*) = [*IN*(*b*) **AND** *NOT KILL*(*b*)] **OR** *GEN*(*b*), the **AND** *NOT* and **OR** statements being logical bit operations [2]. The first equation shows that a definition reaches the beginning of a block *b* only if it reaches the end of one of its predecessors [1]. If a block has no predecessors, then the *IN* vector is the empty set.

The second equation states that a definition reaches the end of block *b* only if either 1) the definition is in *IN*(*b*) (i.e., the definition reaches the beginning of *b* and is not killed by *b*), or 2) the definition is generated within *b* (i.e., it appears in *b* and the data item is not subsequently redefined within *b*) [1]. These equations are known as the data flow equations relating *IN*s and *OUT*s and are the basis for what is commonly called the Basic Reach Algorithm [2]. The solution to these equations is not unique, but there is a minimal solution that will result in the minimum number of active data definitions [1]. The method of reaching that solution is by iterating through the equations block by block until there is no change in the arrays *OUT* and *IN*.

By estimating the *IN* array to be the empty set and *OUT* to be equal to the *GEN* array, the following algorithm can be realized:

```
BEGIN
  FOR EACH BLOCK B DO BEGIN
                                  /* initialize IN, OUT */
    IN(B) = 0;
    OUT(B) = GEN(B);
  END;
  CHANGE = TRUE;
  WHILE CHANGE = TRUE DO BEGIN
                                  /* loop until no change */
    CHANGE = FALSE;
    FOR EACH BLOCK B DO
                                  /* one iteration all blocks */
      NEWIN = UNION OUT(B);
                                  /* hold IN for now */
      IF NEWIN ⟨ ⟩ IN(B) THEN CHANGE = TRUE;
      IN(B) = NEWIN;
      OUT(B) = IN(B) AND NOT(KILL(B)) OR GEN(B);
    END;
  END;
END;  /* algorithm */
```

It can be proven that the algorithm does indeed converge, and it converges in less than *j* iterations, where *j* is the number of basic blocks. This is shown in a paper by Kildall [3].

For the purpose of illustrating the algorithm, the fol-

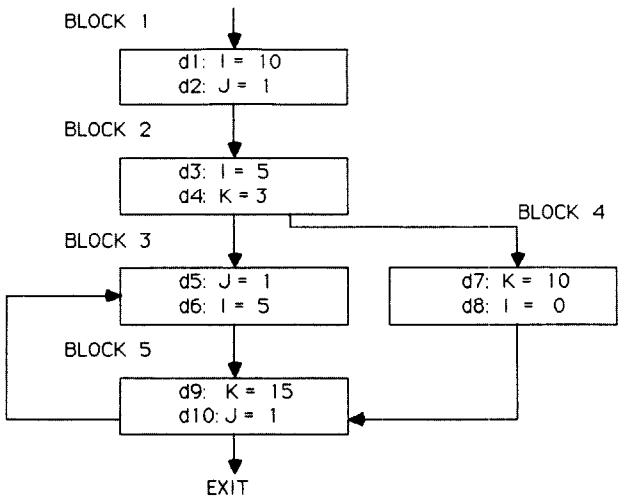

FIGURE 3. Flow Graph for Example

lowing example is presented. Consider the flow graph of Figure 3.

Initially, the *GEN* and *KILL* arrays must be formed. All the arrays will contain ten bits associated with the ten data definitions of the flow graph. *GEN* is easily formed by observing which definitions are defined in each block. Block 1 has data definitions d1, d2, and therefore, will have a vector of *GEN*(1) = 1100000000. The remaining *GEN* array is shown in Figure 4. Since block 1 has data definitions of variables *I* and *J*, these definitions may kill any other definitions in the CSECT of *I* and *J*. These include definitions d3, d5, d6, d8, and d10, and therefore:

$$KILL(1) = 0010110101$$

as shown in Figure 4.

After initializing *IN*(*b*) = empty set and *OUT*(*b*) = *GEN*(*b*), the algorithm proceeds to pass 1. Block 1 has

	GEN(B)	KILL(B)
BLOCK 1	1100000000	0010110101
BLOCK 2	0011000000	1000011110
BLOCK 3	0000110000	1110000101
BLOCK 4	0000001100	1011010010
BLOCK 5	0000000011	0101101000
	initial IN(B)	initial OUT(B)
BLOCK 1	0000000000	1100000000
BLOCK 2	0000000000	0011000000
BLOCK 3	0000000000	0000110000
BLOCK 4	0000000000	0000001100
BLOCK 5	0000000000	0000000011

FIGURE 4. Gen, Kill, Initial In, and Initial Out Arrays

no predecessors, and therefore, *IN*(1) = 0000000000. *OUT*(1) is formed by [*IN*(1) **AND** *NOT KILL*(1)] **OR** *GEN*(1) = 1100000000. Block 3 has two predecessors, and *IN*(3) = *OUT*(2) **OR** *OUT*(5) = (0111000011 **AND** *NOT*(1110000101)) **OR** 0000110000 = 0001110000. As shown in Figure 5, only three passes were required before convergence of the *IN* and *OUT* arrays.

Given this information, a data use now has means of determining directly which definitions potentially affect the data use. Obviously, if a definition potentially affects the data use, the data definition must not be eliminated.

If a data use exists at the end of a block, the OUT array can be used to determine which definitions are required for the data use. Similarly, the *IN* array is used when a data use appears at the beginning of a block. For the previous example, if the variable *K* was referenced at the end of block 3, definitions d4 and d9 must remain as shown by the *K* definitions in the vector *OUT*(3). Likewise, if the variable *K* was referenced at the beginning of block 3, again definitions d4 and d9 must exist as represented by the definitions of *K* in the vector *IN*(3).

A data use, however, may exist anywhere within a group of sequential statements, not always at the beginning or end of a basic block. This situation can be rectified by flagging local definitions. In executing any algorithm, a search through the code is required for all data definitions and data uses. While searching through each block, it would be a trivial detail to keep a record of the last data definition within a block. Therefore, when a data use is found, the last data definition of that type can be flagged. If no such definition exists within the block, then the *IN* vector, generated by the algorithm, must be used to find active definitions.

APPLICATION OF ALGORITHM TO FLAG DEFINITIONS

Marshall and Zobrist presented a method of translating microprocessor object code into a behaviorally-equivalent, high-level language of PL/I for the purpose of electronic system simulation [4]. For the actual translation, groups of PL/I statements are substituted for each assembly instruction. It was noticed, however, that a large portion of the PL/I translation consisted of statements relating to flags associated with the particular microprocessor involved. Flags, such as the parity, zero, sign, carry, and auxiliary-carry in the INTEL 8085 microprocessor, are altered by many of the processors' op-codes [5]. In fact, over fifty percent of the INTEL 8085 instruction set opcodes alter at least one flag [5]. Throughout a code translation, very few instances occur where a flag is referenced. An example of a flag

PASS 1	IN(B)	OUT(B)
BLOCK 1	0000000000	1100000000
BLOCK 2	1100000000	0111000000
BLOCK 3	0111000011	0001110010
BLOCK 4	0111000000	0100001100
BLOCK 5	0101111110	0000010111
PASS 2	IN(B)	OUT(B)
BLOCK 1	0000000000	1100000000
BLOCK 2	1100000000	0111000000
BLOCK 3	0111010111	0001110010
BLOCK 4	0111000000	0100001100
BLOCK 5	0101111110	0000010111
PASS 3	IN(B)	OUT(B)
BLOCK 1	0000000000	1100000000
BLOCK 2	1100000000	0111000000
BLOCK 3	0111010111	0001110010
BLOCK 4	0111000000	0100001100
BLOCK 5	0101111110	0000010111

FIGURE 5. Three Passes of the Algorithm

reference, or flag use, would be a jump on zero condition instruction. The op-code preceding the conditional jump might be a subtract instruction defining all five flags in the INTEL 8085 microprocessor. The PL/I statements defining the other four flags, in this instance, are excessive and unnecessary and could be eliminated from the PL/I translation. The optimization and determination of which flag definitions to discard or retain is the function of this article.

The Basic Reach Algorithm has many applications for optimizing any type of variable. Consequently, this algorithm can easily be adapted to work well in optimizing the flag definitions in the code translation program. All the essential elements are already determined by the translation program to execute the algorithm. At one point in the translation program, a CSECT has been divided into basic blocks, and a PL/I structured information block, as shown in Figure 6, called the *critical array*, has been formed containing valuable information for the optimizing program. The *NODE* and *JUMP_NODE* fields are the successor nodes in the data flow graph. *NODE* is the unconditional jump or exit to the next successor node, and the *JUMP_NODE* is the successor node that would follow if a jump condition is true.

The other vital information field is the *EXIT_NODE* bit indicating a possible exit point from the entire CSECT. Because the code translation program operates on one CSECT at a time, a problem exists in determining whether or not a flag definition in one CSECT potentially affects data uses in other CSECTs. This problem is alleviated by translating all active flag definitions at the output of all exit nodes. Now all flag definitions that can possibly reach a data use in another CSECT are translated.

Another information block associated with each assembly code instruction is the assembly list, as shown in Figure 7. This list contains labels, the op-codes, operands, and other information. For implementation of the Basic Reach Algorithm, the assembly list must be appended with fields that describe whether an instruction requires a flag definition, such as a jump carry instruction, or defines a flag, such as a subtract instruction which defines most common flags. Some instructions, depending on the microprocessor, both define and use a flag.

The flag optimization program was written as a procedure whereby the code translation program could call the optimization procedure as an option to the user. The code translation program was also modified to review the flag defined field of the assembly list in order to determine whether a flag definition is to be translated. Initially, before the optimization procedure is called, all flag definitions will have the flag defined field set on, indicating that a flag definition must be translated.

The flag optimization procedure, when called, will process the given information and set off any flag definitions not required by the translation code. If the user chooses not to optimize, the procedure is not called and all flag definitions are translated. Conversely, if the procedure is called, only those flag definitions not set off by the optimization procedure will be translated.

THE OPTIMIZATION PROGRAM

The optimization program is subdivided into five separate procedures. The main control is illustrated below:

FOR THIS CSECT **DO**
1) *CALL PROCESS_DATA*
2) *CALL ALGORITHM*
3) *CALL FORM_RESULTS*
4) *CALL UPDATE_DATA*
5) *CALL FREE_BLOCKS*

END *OPTIMIZATION*;

```
DCL 1 CRIT_ARRAY(2000)   EXTERNAL,          /* CRITICAL PT. ARRAY   */
      2 LBEL             CHAR(8),           /* LABEL FIELD          */
      2 ADR              CHAR(8),           /* ADDRESS FIELD        */
      2 NODE_INFO,
        3 JUMP_LBEL      CHAR(8),           /* TARG. LABEL OF JUMP  */
        3 JUMP_ADR       CHAR(8),           /* TARG. ADR. OF JUMP   */
        3 JUMP_NODE      FIXED BIN(15),     /* TARGET NODE OF JUMP  */
        3 NODE           FIXED BIN(15),     /* NON-JUMP TARGET      */
        3 COND_DELAY     FIXED BIN(15),     /* COND. DELAY OF CRIT. */
        3 DELAY          FIXED BIN(15),     /* DELAY OF CRIT. BLOCK */
        3 EXE_LINES      FIXED BIN(15),     /* NO. OF EXE. LINES    */
      2 NODE_TYPE,
        3 SQ             BIT(1),            /* SEQ. BLOCK PRIMITIVE */
        3 EX             BIT(1),            /* EXIT BLOCK PRIMITIVE */
        3 EC             BIT(1),            /* COND. EXIT BLOCK     */
        3 JU             BIT(1),            /* UNCOND. JUMP BLOCK   */
        3 JC             BIT(1),            /* COND. JUMP BLOCK     */
      2 EXIT_NODE        BIT(1),            /* CSECT EXIT BOOLEAN   */
      2 COND_TYPE_NODE   BIT(1),            /* COND. CRIT. BOOLEAN  */
      2 IO_CRITPT        BIT(1),            /* I/O CRIT. POINT      */
      2 LAST_NODE        BIT(1),            /* LAST CRIT. NODE      */
      2 BLOCK_LIST_HEAD  BIT(1),            /* PTR. TO ASSEM_LIST   */
```

FIGURE 6. Critical Array Information Block

```
DCL 1 ASSM_LIST          EXTERNAL,          /* LIST OF ASM.LINES    */
      2 ASSEM_TYPE,                         /* CATEGORIZE ASM.LINE  */
      3 EQU              BIT(1),            /* EQU DIRECTIVE        */
      3 CSECT            BIT(1),            /* CSECT DIRECTIVE      */
      3 COMMENT          BIT(1),            /* COMMENT LINE         */
      3 EXE              BIT(1),            /* EXECUTABLE LINE      */
      3 CRITPT           BIT(1),            /* USER CRIT POINT      */
      2 ASSEM_FIELD,                        /* ASM. LINE FIELDS     */
        3 ADR            CHAR(8),           /* ADDRESS FIELD        */
        3 OPCODE         CHAR(2),           /* OPCODE FIELD         */
        3 BYTE2          CHAR(2),           /* 2ND BYTE OF OBJ CODE*/
        3 BYTE3          CHAR(2),           /* 3RD BYTE OF OBJ CODE*/
        3 LBEL           CHAR(8),           /* LABEL FIELD          */
        3 MNEMONIC       CHAR(4),           /* MNEMONIC FIELD       */
        3 OP(9)          CHAR(20),          /* CHAR. OPERANDS       */
        3 JMPADR         CHAR(8),           /* JUMP DEST. ADDRESS   */
        3 MOP(9)         CHAR(8),           /* MODIFIED OPERAND 1   */
      2 PSW_FIELD(8),                       /* PSW FLAGS(8 MAX.)    */
        3 REQUIRED       FIXED BIN(15),     /* PSW FLAG REQUIRED    */
        3 DEFINED        FIXED BIN(15),     /* PSW FLAG DEFINED     */
      2 NEXT_LINE_PT     FIXED BIN(15),     /* LINK TO NEXT LINE    */
      2 JUMP_EXIT_PT     BIT(1);            /* EXTERNAL EXIT        */
```

FIGURE 7. Assembly List Information Block

The following is a brief summary for each procedure:

PROCESS_DATA

Information from *ASSEM_LIST* and *CRIT_ARRAY* in the translation program is collected, forming the arrays required for the Basic Reach Algorithm. In a scan of *ASSEM_LIST*, for each flag definition found, a data block is allocated in memory storing the type of flag, its location in the assembly listing, *GEN*, *KILL*, *OUT*, and *IN* bit vectors, and a pointer to the next flag definition block creating a linked list.

ALGORITHM

The actual algorithm is executed utilizing the arrays generated in *PROCESS_DATA*, resulting in minimized *OUT* and *IN* arrays.

FORM_RESULTS

The assembly list is again scanned, looking for flag uses that require a flag definition. The location of the flag use in a block determines if either *OUT* or *IN* is needed to find all flag definitions reaching the flag use. In some cases, neither is used when a definition resides in the same block as the flag use and the flag use is not the last statement of the block. Another special case occurs when the last statement of a block contains both a flag definition and a flag use of the same flag. The *OUT* array cannot be used in this case since the flag definition in that statement cannot be used to satisfy its own flag use.

UPDATE_DATA

The *ASSEM_LIST* file now is updated with the new list of flag definitions to be translated. *ASSEM_LIST* is first cleared of all the flag definitions and then replaced with the new list created in *FORM_RESULTS*.

FREE_BLOCKS

All data blocks allocated for each flag definition are freed to the operating system. This allows memory to be freed for the next time the optimization procedure is called.

SAMPLE OUTPUT

A sample CSECT for an INTEL 8085 microprocessor was run with the flag optimization activated, and the following figures show the results. Figure 8 displays the flag defined/required status prior to executing the optimization procedure. Figure 9 contains a data flow graph showing exit nodes and flow information drawn from *CRIT_ARRAY* for the CSECT. Finally, the results are shown in Figure 10, a display of the flag defined/required status. An asterisk is placed by those flag definitions that were set off by the optimization so one can verify that the flag definitions were not needed in the translation.

SUMMARY

A procedure for optimizing flag definitions has been presented with application to an object code translation program. The overall result of calling the procedure is that several flag definitions are eliminated from the

BLOCK 1	ASSEM	INST	ZERO	SIGN	PARITY	CARRY	AUX-CARRY
IAOEAOFF:	JMP	*+16	N	N	N	N	N

BLOCK 2	ASSEM	INST	ZERO	SIGN	PARITY	CARRY	AUX-CARRY
	PUSH	PSW	R	R	R	R	R
	PUSH	B	N	N	N	N	N
	PUSH	D	N	N	N	N	N
	PUSH	H	N	N	N	N	N
	XRA	A	N	N	N	N	N
	STA	PAOOWORK	N	N	N	N	N
	LXI	H,IKBTKBD	N	N	N	N	N
	SHLD	PAOAWORK	N	N	N	N	N
	LXI	H,ICOTICO	N	N	N	N	N
	SHLD	PAOACALV	N	N	N	N	N

BLOCK 3	ASSEM	INST	ZERO	SIGN	PARITY	CARRY	AUX-CARRY
LBTBWHL1:	LHLD	PAOACALV	N	N	N	N	N
	MOV	A,M	N	N	N	N	N
	CPI	X'FF'	D	D	D	D	D
	JZ	LBTEWHL1	R	N	N	N	N

BLOCK 4	ASSEM	INST	ZERO	SIGN	PARITY	CARRY	AUX-CARRY
	LHLD	PAOACALV	N	N	N	N	N
	MOV	B,M	N	N	N	N	N
	INX	H	N	N	N	N	N
	MOV	C,M	N	N	N	N	N
	INX	H	N	N	N	N	N
	MOV	D,M	N	N	N	N	N
	LHLD	PAOAWORK	N	N	N	N	N
	MOV	A,B	N	N	N	N	N
	CMP	M	D	D	D	D	D
	JNZ	LBTENIF9	R	N	N	N	N

BLOCK 5	ASSEM	INST	ZERO	SIGN	PARITY	CARRY	AUX-CARRY
	INX	H	N	N	N	N	N
	MOV	A,C	N	N	N	N	N
	CMP	M	D	D	D	D	D
	JNZ	LBTENIF9	R	N	N	N	N

BLOCK 6	ASSEM	INST	ZERO	SIGN	PARITY	CARRY	AUX-CARRY
	INX	H	N	N	N	N	N
	MOV	A,D	N	N	N	N	N
	CMP	M	D	D	D	D	D
	JNZ	LBTENIF9	R	N	N	N	N

BLOCK 7	ASSEM	INST	ZERO	SIGN	PARITY	CARRY	AUX-CARRY
	INX	H	N	N	N	N	N
	INX	H	N	N	N	N	N
	MOV	A,M	N	N	N	N	N
	RLC		N	N	N	D	N
	JC	LSTELSE5	N	N	N	R	N

BLOCK 8	ASSEM	INST	ZERO	SIGN	PARITY	CARRY	AUX-CARRY
	MVI	A,X'09'	N	N	N	N	N
	STA	PAOOWORK	N	N	N	N	N
	MVI	C,X'02'	N	N	N	N	N
	JMP	LSTENDF8	N	N	N	N	N

FIGURE 8. Assembly List for CSECT Before Optimization

BLOCK 9	ASSEM	INST	ZERO	SIGN	PARITY	CARRY	AUX-CARRY
LSTELSE5:	RLC		N	N	N	D	N
	JC	LSTELSE7	N	N	N	R	N

BLOCK 10	ASSEM	INST	ZERO	SIGN	PARITY	CARRY	AUX-CARRY
	MVI	A,X'OB'	N	N	N	N	N
	STA	PAOOWORK	N	N	N	N	N
	MVI	C,X'03'	N	N	N	N	N
	JMP	LSTENDF8	N	N	N	N	N

BLOCK 11	ASSEM	INST	ZERO	SIGN	PARITY	CARRY	AUX-CARRY
LSTELSE7:	MVI	A,X'OF'	N	N	N	N	N
	STA	PAOOWORK	N	N	N	N	N
	MVI	C,X'05'	N	N	N	N	N

BLOCK 12	ASSEM	INST	ZERO	SIGN	PARITY	CARRY	AUX-CARRY
LSTENDF8:	INX	H	N	N	N	N	N
	SHLD	PAOAWRKA	N	N	N	N	N
	LHLD	PAOACALV	N	N	N	N	N
	INX	H	N	N	N	N	N
	INX	H	N	N	N	N	N
	INX	H	N	N	N	N	N
	SHLD	PAOACALA	N	N	N	N	N
	MOV	A,C	N	N	N	N	N
	MOV	B,M	N	N	N	N	N
	INX	H	N	N	N	N	N
	MOV	C,M	N	N	N	N	N
	LHLD	PAOAWRKA	N	N	N	N	N

BLOCK 13	ASSEM	INST	ZERO	SIGN	PARITY	CARRY	AUX-CARRY
LBTADDUM:	MOV	D,M	N	N	N	N	N
	INX	H	N	N	N	N	N
	MOV	E,M	N	N	N	N	N
	MOV	H,B	N	N	N	N	N
	MOV	L,C	N	N	N	N	N
	DAD	D	N	N	N	D	N
	XCHG		N	N	N	N	N
	LHLD	FUDGEFAC	N	N	N	N	N
	DAD	D	N	N	N	D	N
	XCHG		N	N	N	N	N
	LHLD	PAOAWRKA	N	N	N	N	N
	MOV	M,D	N	N	N	N	N
	INX	H	N	N	N	N	N
	MOV	M,E	N	N	N	N	N
	INX	H	N	N	N	N	N
	SHLD	PAOAWRKA	N	N	N	N	N
	DCR	A	D	D	D	N	D
	JNZ	LBTADDUM	R	N	N	N	N

BLOCK 14	ASSEM	INST	ZERO	SIGN	PARITY	CARRY	AUX-CARRY
	LHLD	PAOOWORK	N	N	N	N	N
	XCHG		N	N	N	N	N
	LHLD	PAOAWORK	N	N	N	N	N
	DAD	D	N	N	N	D	N
	SHLD	PAOAWORK	N	N	N	N	N

FIGURE 8. *Continued*

```
BLOCK 15    ASSEM  INST          ZERO  SIGN  PARITY  CARRY  AUX-CARRY
   LBTENIF9: LHLD   PAOACALV      N     N     N       N      N
             LXI    D,5           N     N     N       N      N
             DAD    D             N     N     N       D      N
             SHLD   PAOACALV      N     N     N       N      N
             JMP    LBTBWHL1      N     N     N       N      N

BLOCK 16    ASSEM  INST          ZERO  SIGN  PARITY  CARRY  AUX-CARRY
   LBTEWHL1: POP    H             N     N     N       N      N
             POP    D             N     N     N       N      N
             POP    B             N     N     N       N      N
             POP    PSW           D     D     D       D      D
             RET                  N     N     N       N      N
```

FIGURE 8. *Continued*

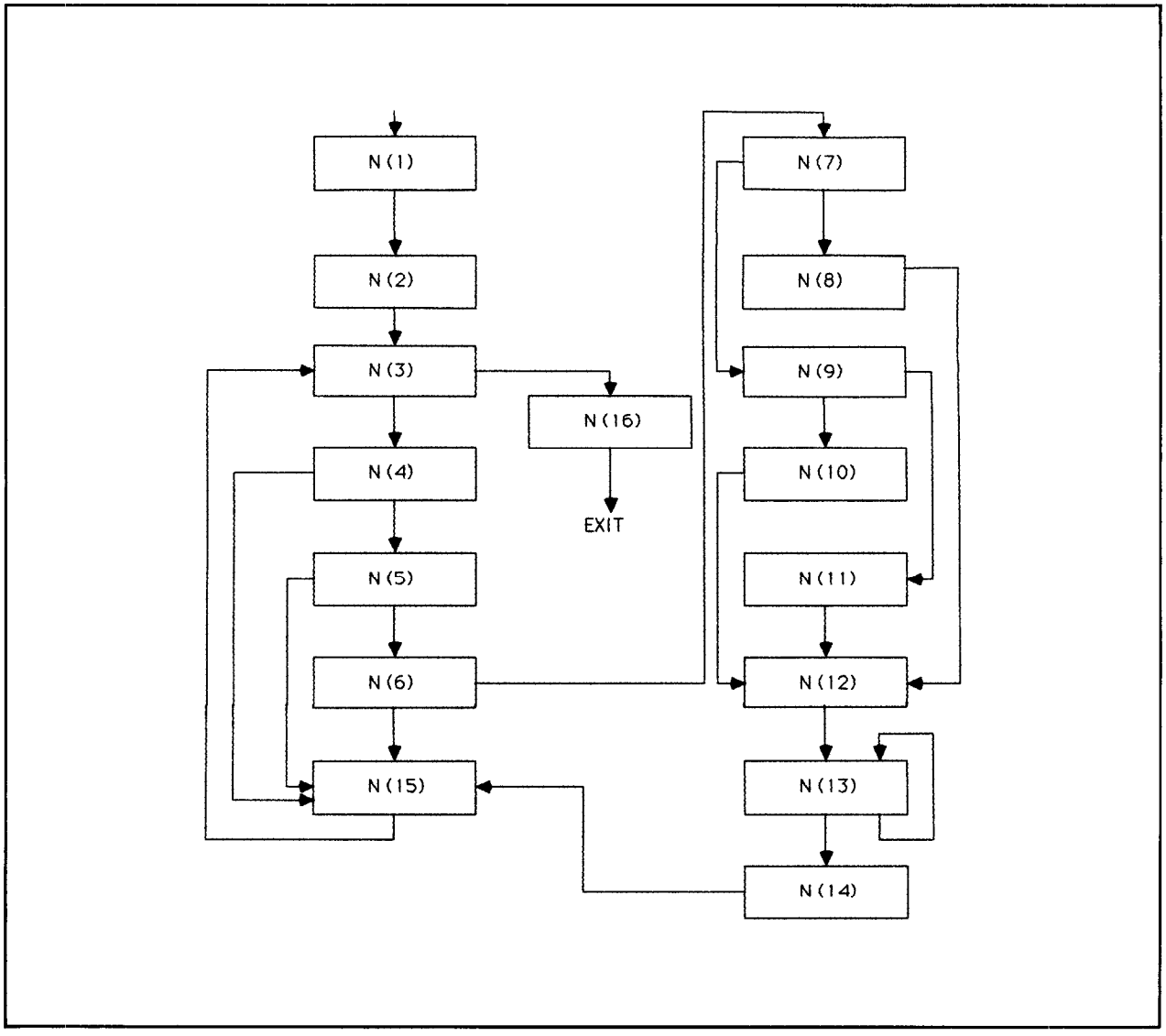

FIGURE 9. Data Flow Graph for Application

BLOCK 1	ASSEM	INST	ZERO	SIGN	PARITY	CARRY	AUX-CARRY
IAOEAOFF:	JMP	*+16	N	N	N	N	N

BLOCK 2	ASSEM	INST	ZERO	SIGN	PARITY	CARRY	AUX-CARRY
	PUSH	PSW	R	R	R	R	R
	PUSH	B	N	N	N	N	N
	PUSH	D	N	N	N	N	N
	PUSH	H	N	N	N	N	N
	XRA	A	N	N	N	N	N
	STA	PAOOWORK	N	N	N	N	N
	LXI	H,IKBTKBD	N	N	N	N	N
	SHLD	PAOAWORK	N	N	N	N	N
	LXI	H,ICOTICO	N	N	N	N	N
	SHLD	PAOACALV	N	N	N	N	N

BLOCK 3	ASSEM	INST	ZERO	SIGN	PARITY	CARRY	AUX-CARRY
LBTBWHL1:	LHLD	PAOACALV	N	N	N	N	N
	MOV	A,M	N	N	N	N	N
	CPI	X'FF'	D	N*	N*	N*	N*
	JZ	LBTEWHL1	R	N	N	N	N

BLOCK 4	ASSEM	INST	ZERO	SIGN	PARITY	CARRY	AUX-CARRY
	LHLD	PAOACALV	N	N	N	N	N
	MOV	B,M	N	N	N	N	N
	INX	H	N	N	N	N	N
	MOV	C,M	N	N	N	N	N
	INX	H	N	N	N	N	N
	MOV	D,M	N	N	N	N	N
	LHLD	PAOAWORK	N	N	N	N	N
	MOV	A,B	N	N	N	N	N
	CMP	M	D	N*	N*	N*	N*
	JNZ	LBTENIF9	R	N	N	N	N

BLOCK 5	ASSEM	INST	ZERO	SIGN	PARITY	CARRY	AUX-CARRY
	INX	H	N	N	N	N	N
	MOV	A,C	N	N	N	N	N
	CMP	M	D	N*	N*	N*	N*
	JNZ	LBTENIF9	R	N	N	N	N

BLOCK 6	ASSEM	INST	ZERO	SIGN	PARITY	CARRY	AUX-CARRY
	INX	H	N	N	N	N	N
	MOV	A,D	N	N	N	N	N
	CMP	M	D	N*	N*	N*	N*
	JNZ	LBTENIF9	R	N	N	N	N

BLOCK 7	ASSEM	INST	ZERO	SIGN	PARITY	CARRY	AUX-CARRY
	INX	H	N	N	N	N	N
	INX	H	N	N	N	N	N
	MOV	A,M	N	N	N	N	N
	RLC		N	N	N	D	N
	JC	LSTELSE5	N	N	N	R	N

BLOCK 8	ASSEM	INST	ZERO	SIGN	PARITY	CARRY	AUX-CARRY
	MVI	A,X'09'	N	N	N	N	N
	STA	PAOOWORK	N	N	N	N	N
	MVI	C,X'02'	N	N	N	N	N
	JMP	LSTENDF8	N	N	N	N	N

FIGURE 10. Assembly List for CSECT After Optimization

BLOCK 9	ASSEM	INST	ZERO	SIGN	PARITY	CARRY	AUX-CARRY
LSTELSE5:	RLC		N	N	N	D	N
	JC	LSTELSE7	N	N	N	R	N

BLOCK 10	ASSEM	INST	ZERO	SIGN	PARITY	CARRY	AUX-CARRY
	MVI	A,X'OB'	N	N	N	N	N
	STA	PAOOWORK	N	N	N	N	N
	MVI	C,X'03'	N	N	N	N	N
	JMP	LSTENDF8	N	N	N	N	N

BLOCK 11	ASSEM	INST	ZERO	SIGN	PARITY	CARRY	AUX-CARRY
LSTELSE7:	MVI	A,X'OF'	N	N	N	N	N
	STA	PAOOWORK	N	N	N	N	N
	MVI	C,X'05'	N	N	N	N	N

BLOCK 12	ASSEM	INST	ZERO	SIGN	PARITY	CARRY	AUX-CARRY
LSTENDF8:	INX	H	N	N	N	N	N
	SHLD	PAOAWRKA	N	N	N	N	N
	LHLD	PAOACALV	N	N	N	N	N
	INX	H	N	N	N	N	N
	INX	H	N	N	N	N	N
	INX	H	N	N	N	N	N
	SHLD	PAOACALA	N	N	N	N	N
	MOV	A,C	N	N	N	N	N
	MOV	B,M	N	N	N	N	N
	INX	H	N	N	N	N	N
	MOV	C,M	N	N	N	N	N
	LHLD	PAOAWRKA	N	N	N	N	N

BLOCK 13	ASSEM	INST	ZERO	SIGN	PARITY	CARRY	AUX-CARRY
LBTADDUM:	MOV	D,M	N	N	N	N	N
	INX	H	N	N	N	N	N
	MOV	E,M	N	N	N	N	N
	MOV	H,B	N	N	N	N	N
	MOV	L,C	N	N	N	N	N
	DAD	D	N	N	N	N*	N
	XCHG		N	N	N	N	N
	LHLD	FUDGEFAC	N	N	N	N	N
	DAD	D	N	N	N	N*	N
	XCHG		N	N	N	N	N
	LHLD	PAOAWRKA	N	N	N	N	N
	MOV	M,D	N	N	N	N	N
	INX	H	N	N	N	N	N
	MOV	M,E	N	N	N	N	N
	INX	H	N	N	N	N	N
	SHLD	PAOAWRKA	N	N	N	N	N
	DCR	A	D	N*	N*	N	N*
	JNZ	LBTADDUM	R	N	N	N	N

BLOCK 14	ASSEM	INST	ZERO	SIGN	PARITY	CARRY	AUX-CARRY
	LHLD	PAOOWORK	N	N	N	N	N
	XCHG		N	N	N	N	N
	LHLD	PAOAWORK	N	N	N	N	N
	DAD	D	N	N	N	N*	N
	SHLD	PAOAWORK	N	N	N	N	N

FIGURE 10. *Continued*

BLOCK 15	ASSEM	INST	ZERO	SIGN	PARITY	CARRY	AUX-CARRY
LBTENIF9:	LHLD	PAOACALV	N	N	N	N	N
	LXI	D,5	N	N	N	N	N
	DAD	D	N	N	N	N*	N
	SHLD	PAOACALV	N	N	N	N	N
	JMP	LBTBWHL1	N	N	N	N	N

BLOCK 16	ASSEM	INST	ZERO	SIGN	PARITY	CARRY	AUX-CARRY
LBTEWHL1:	POP	H	N	N	N	N	N
	POP	D	N	N	N	N	N
	POP	B	N	N	N	N	N
	POP	PSW	D	D	D	D	D
	RET		N	N	N	N	N

FIGURE 10. *Continued*

translation code. In the application presented, over fifty percent, twenty-three of forty INTEL 8085 flag definitions were eliminated from the translation. The optimization procedure is versatile in that it can be used for optimizing flag definitions of any microprocessor and has been utilized in the INTEL 8051 and INTEL 8086. Since the procedure only modifies already existing data fields, its execution is used only if the optimization is desired.

A possible topic for future study in the area of optimization of data definitions is an optimization on a more global basis, where data definitions that are active upon exit of a CSECT are optimized. Several CSECTs may be grouped together to form nodes whereby the Basic Reach Algorithm can be executed again.

This technique can be used for general code optimization. Through the use of the Basic Reach Algorithm, one can determine if a variable is defined at a point in a program, but never used. One can then remove these variables from the code, thereby resulting in a more compact code package. It can also be used to determine whether a variable that has been used at a particular point has been defined somewhere in a path arriving at that point.

This technique has also been used to achieve optimization beyond local transformations in automated logic design [7, 8]. Global data flow analysis has achieved the screening of logic for possible redundancies, correction of timing problems by moving late inputs forward, and remedial effects of poor specification that may result in a test specification too early in the functions description.

A generalization of this algorithm would be: to any system that can be represented by a directed flow graph with the notion of determining whether some property that is to be used at a node has been defined previously; or determining whether a property has been defined and if it reaches any use of that property. This would allow for both determination of system properties that are undefined, and system optimization.

Acknowledgment. Partial support for this paper was from IBM, Lexington, KY grant number 200-614.

REFERENCES

1. Aho, A.V., and Ullman, J.D. *Principles of Compiler Design.* Addison-Wesley, Reading, Mass., 1979.
2. Allen, F.E., and Cocke, J. A program data flow analysis procedure. *Commun. ACM 19*, 3 (Mar. 1976), 137–145.
3. Intel Corp. *Intel 8080/8085 Assembly Language Programming User's Manual.* Man. 9800940. Intel Corp., Santa Clara, Calif., 1979.
4. Kildall, G.A. A unified approach to global program optimization. In *Conference Record of ACM Symposium on Principles of Programming Languages* (Boston, Mass., Oct. 1–3). ACM, New York, 1973, pp. 194–206.
5. Marshall, W.K., Zobrist, G.W., Bach, W., and Richardson, A. A functional mapping for a microprocessor system simulation. In *Proceedings of the 1985 IEEE Microprocessor Forum* (Atlantic City, N.J., Apr. 2–4). IEEE, Piscataway, N.J., 1985, pp. 37–39.
6. Trevillyan, L. An overview of logic synthesis systems. In *Proceedings of the 24th ACM/IEEE Design Automation Conference* (Miami Beach, Fla., June 29–July 1). IEEE, Piscataway, N.J., 1987, pp. 166–172.
7. Trevillyan, L., Joyner, W., and Berman, L. Global flow analysis in automated logic design. *IEEE Trans. Comput. c-35*, 1 (Jan. 1986), 77–81.
8. Zobrist, G.W., and Smith, B. A functional mapping for a microcontroller simulation. In *Proceedings of the IEEE Work Station Technology And Systems Conference* (Atlantic City, N.J., Mar. 17–20). IEEE, Piscataway, N.J., 1986, pp. 37–39.

CR Categories and Subject Descriptors: B.5.2 [**Register-Transfer-Level Implementation**]: Design Aids—*optimization, simulation*; D.3.4 [**Programming Languages**]: Processors—*compilers, optimization, translator writing systems and compiler generators*; I.6.1 [**Simulation and Modeling**]: Simulation Theory—*model classification*
General Terms: Algorithms, Design, Languages
Additional Key Words and Phrases: Basic reach algorithm, global data flow analysis

ABOUT THE AUTHORS:

DAVID M. DeJEAN studied at the University of Evansville, Indiana, and did graduate work at the University of Missouri at Rolla-St. Louis extension. He is an automatic test equipment designer at McDonnell Douglas Aircraft. Author's present address: 12423 Bernie, Maryland Heights, MO 63043.

GEORGE W. ZOBRIST is a professor of computer science at the University of Missouri-Rolla, Rolla, Missouri, where his current interests are in computer aided engineering and design, software engineering tools, and simulation techniques. Author's present address: Department of Computer Science, University of Missouri-Rolla, Rolla, MO 65401.