Evaluation of Boolean Expressions on One's Complement Machines

Peter L. Montgomery

System Development Corporation
4810 Bradford Boulevard, N. W.
Huntsville, Alabama 35805

Abstract

An algorithm is presented which evaluates an arbitrary boolean expression in terms of a small instruction set and which often generates fewer instructions than other methods the author has seen. The algorithm tells what code to generate whenever a boolean operator, a branch test, a boolean assignment statement, or a conditional expression is encountered. One's complement arithmetic is assumed throughout.

1. Background and Assumptions

My installation has a Control Data (CDC) 7600 for which numerous FORTRAN programs are written. Almost everyone here uses the FTN compiler, an optimizing compiler written by CDC which generates very good code. One area in which its object code can be further improved involves the evaluation of boolean expressions (called logical in FORTRAN) containing more than one relational, especially in IFs. This paper formalizes a procedure for generating the improved code.

Because of the background, the algorithm is oriented towards the CDC CYBER series and uses many of the conventions of the FTN compiler. In particular, for variables stored in memory, a value of TRUE is represented by an arbitrary negative quantity and a value of FALSE by an arbitrary positive quantity. (Alternatively, TRUE can be represented by -0 and FALSE by +0.) Each boolean expression is evaluated as one quantity; the algorithm does not issue a separate branch or skip instruction for each relational in an expression. The required hardware instructions (all of which operate on full word operands) are listed in Table 1; except possibly for ANDNOT and NXOR, these are standard on one's complement machines. Note the add and subtract instructions are assumed not to interrupt on overflow. Also, a sum of two complementary operands or a difference of two identical operands is assumed to be +0, not -0.

2. Description

The algorithm specifies the code to be generated for each boolean operator in an expression; it is intended to be used in conjunction with another procedure which evaluates arbitrary expressions. Before one can apply the algorithm, he must decide what type of code should be generated for each boolean primitive, if the primitive were to appear alone in a branch on true. In FORTRAN, this requires deciding upon the best code for each of the six relational operators (.EQ., .GT., .GE., .LE., .LT., and .NE.), for each permissible type of operand (Real, Integer, Double Precision, Complex, Character). In each case, either the relational is recognized as identically true or identically false (if its

Table 1. Basic Instruction Set Assumed by Algorithm

Instruction Description	Notation
Branch if operand is +0 or -0	
Branch if operand is nonzero	
Branch if operand is positive (or +0)	
Branch if operand is negative (or -0)	
Branch unconditionally	
Get copy of operand	
AND two or more operands*	AND
AND two operands, negating second	ANDNOT
Get one's complement of operand	COMPL
Negative exclusive OR of two operands	NXOR
OR two or more operands*	OR
Exclusive OR two operands	XOR
Get word in which all bits are same as sign bit of operand	XSIGN
One's complement sum of two operands, overflow ignored	+
One's complement difference of two operands, overflow ignored	-
Load +0 into a register	+0,0
Load -0 into a register	-0
Load operand from memory	
Store operand in memory	

*If more than two operands are present, more than one instruction will be required, of course.

arguments can be evaluated at compile time), or it is replaced by a series of computations, the last being a branch on positive, a branch on negative, a branch on zero, or a branch on nonzero. For example, to branch when J.GT.K has value true, the quantity K-J+0 might be computed and a branch on negative generated. The purpose of the algorithm is to decide what code to generate if the relational occurs in a more complicated context, while always preserving the optimal code if the relational stands alone.

The algorithm uses ten tags, listed in Table 2, to identify what is known about each intermediate expression. The tag identifies which possible values correspond to true, which possible values correspond to false, and which values are known to be impossible. Of the ten tags, four (NEG, NONZERO, POS, ZERO) correspond directly to assumed hardware branch instructions. Four others (ZEROM, NPLZERO, ZEROP, PLZERO) are special cases of the first four, in which certain operand values are known to be impossible; these are essential if full optimization is desired but can be omitted if only moderate optimization is necessary. The other two (FALSE, TRUE) are desirable for processing boolean constants and for identifying dead code, although they, too, could be omitted with little loss. Note the table is closed under negation; if a tag appears in the table, its logically opposite tag also appears. Consequently, the operations of branching and of negation are trivial.

Besides branching and negation, the other standard operations one must be able to perform are the ANDing or ORing of two boolean expressions, the storing of a boolean expression in a boolean variable, and the evaluation of a conditional expression. The OR of two boolean expressions B1 and B2 may be expressed in terms of one AND and three NOTs, i.e.,

```
          B1.OR.B2 = .NOT.((.NOT.B1).AND.(.NOT.B2)).
```

Tables for the other operations appear in Appendix I. The AND table is summarized in Appendix II; the latter is more suitable for direct implementation. Appendix II also considers the optimal way to AND three or more expressions of differing tags.

Certain sequences of instructions occur repeatedly in the tables and are given special names (EQ0, EQP0, NE0, NEP0); these return a mask of ones if the argument is, respectively, zero, plus zero, nonzero, not plus zero. Each returns a mask of zeros otherwise. Formal definitions are:

```
                    EQ0(I) = NXOR(0+I,0-I)
                    EQP0(I) = NXOR(I,0-I)
                    NE0(I) = XOR(0+I,0-I)
                    NEP0(I) = XOR(I,0-I)
```

3. Examples

As a sample application, let L1 be a logical variable. Consider the FORTRAN statement

```
          IF(.NOT.L1) I1=0
```

Table 2. Tags Used by Algorithm

Tag	Meaning
FALSE	Expression always false.
NEG	Expression true if value is negative, false if positive.
NONZERO	Expression true if value is nonzero, false if +0 or -0.
NPLZERO	Expression true if value is nonzero, false if +0. Expression known not to be -0.
PLZERO	Expression true if value is +0, false if nonzero. Expression known not to be -0.
POS	Expression true if value is positive, false if negative.
TRUE	Expression always true.
ZERO	Expression true if value is +0 or -0, false if nonzero.
ZEROM	Expression true if value is -0, false if +0. Expression known to be +0 or -0.
ZEROP	Expression true if value is +0, false if -0. Expression known to be +0 or -0.

The only boolean primitive is L1. Since logical variables are assumed true if negative and false if positive, a branch on true would require a load of L1 and a branch on negative. Therefore the code which loads L1 from memory is generated and tagged NEG. The statement looks like

```
          IF(.NOT.NEG(L1)) I1=0
```

where the argument to NEG is assumed to have already been converted to machine code. The next operator is NOT. To negate an expression tagged NEG, change its tag to POS (no additional code is generated). The statement has been reduced to

```
          IF(POS(L1)) I1=0
```

The next operator is IF. To branch when POS fails, issue a branch on negative instruction. The right hand side of the IF is evaluated in the standard way. The entire statement requires one load, one branch on negative, one load of 0, and one store - no complement instruction is required.

The last example involves only a negation and a branch, so it is very simple. A more complicated example is

```
          IF(I1.GT.5 .OR. I2.EQ.0) GO TO 20
```

To branch when I1.GT.5 is true, one would compute 5-I1 and branch on negative (note the result cannot be -0). To branch when I2.EQ.0 is true, one would load I2 and branch on zero (both +0 and -0 are considered equal to zero). The statement is therefore transformed into

```
          IF(NEG(5-I1) .OR. ZERO(I2)) GO TO 20
```

To evaluate the OR, negate both operands

```
          IF(.NOT.(POS(5-I1) .AND. NONZERO(I2))) GO TO 20
```

Evaluation of the AND reduces the code to (see rule for ANDing a POS tag and a NONZERO tag in Appendix I)

```
          IF(.NOT.NONZERO(ANDNOT(I2, XSIGN(5-I1)))) GO TO 20
```

as a shift and an ANDNOT instruction are generated. Evaluation of the NOT changes the NONZERO tag to ZERO; no further transformation is required before the IF. The statement is evaluated as though it were

```
          IF(ZERO(ANDNOT(I2, XSIGN(5-I1)))) GO TO 20
```

The statement requires 7 instructions (2 loads, 1 load immediate, 1 subtract, 1 shift, 1 and with complement, 1 branch on zero). This compares to 10 instructions (2 loads, 1 load immediate, 1 load of +0, 2 subtracts, 1 add, 1 negative exclusive or, 1 or, 1 branch on negative) generated by version 4.6 of FTN.

4. Further Optimizations

The ANDing of two NONZERO or NPLZERO tags, or the ORing of two ZERO or PLZERO tags, deserves special attention, as hidden opportunities for optimization will often occur. The statement

```
          IF(I.EQ.3 .OR. I.EQ.-7) GO TO 60
```

is translated into (note I-3 and I+7 are never -0)

```
          IF (PLZERO(I-3) .OR. PLZERO(I+7) GO TO 60
```

and then into

```
          IT1 = I-3
          IT2 = I+7
          IF (ZERO(XOR(IT1+IT2, IT1-IT2))) GO TO 60
```

Here IT1 and IT2 are compiler-generated variables. Note IT1 and IT2 are defined in terms of addition and subtraction, but only their sum and their difference are used. Furthermore, it is unimportant here whether IT1+IT2 and IT1-IT2 evaluate to +0 or -0 if they are indeed zero; hence the standard mathematical identities about addition and subtraction may be used freely. A better expansion would be

```
          IF (ZERO(XOR(I+I+4,-10))) GO TO 60
```

This opportunity arises whenever an integer variable is being compared against two different constants. Incidentally, since both constants have the same parity (i.e., 3 and -7 are both odd), a divide by 2 can be used to further simplify the code to

```
          IF (ZERO(XOR(I+2,-5))) GO TO 60
```

5. Conditional Expressions

The FORTRAN language does not allow conditional expressions (except that some of its intrinsic functions like ABS may be thought of as conditional) Nonetheless, a sophisticated compiler should be able to recognize when a user is thinking in terms of conditional expressions, as in

```
          REAL SCX, SCAX
          IF(SCX.LE.0) THEN
          SCAX = 1.0
          ELSE
          SCAX = SCX
          END IF
```

On some machines optimal code is gotten by loading both SCX and 1.0 into registers, evaluating SCX.LE.0.0 as a mask of zeros or a mask of ones, and using hardware boolean instructions to complete the evaluation - no branch instruction is generated. The relational SCX.LE.0.0 is equivalent to POS(0-SCX) (integer subtract). To convert this to a mask of zeros or a mask of ones (i.e., to change the tag from POS to ZEROM or ZEROP), apply the XSIGN operator. The expansion is

```
        IT1 = XSIGN (0-SCX)
        SCAX = OR(AND(SCX,IT1), ANDNOT(1.0,IT1))
```

6. Other Uses of Boolean Expressions

Some operations have been left out of the tables. The use of a boolean expression (other than a boolean variable) as a procedure argument is a special case of an assignment statement. Similarly, the output of a boolean procedure is treated as a special case of a boolean variable. The equivalence or the exclusive or of two boolean expressions, neither operand having tag TRUE or FALSE, may be processed by first converting each tag to NEG, POS, ZEROM, or ZEROP, then generating an exclusive OR instruction and tagging the result appropriately. Also omitted is the evaluation of a conditional expression which is itself type boolean and in which the operands may be evaluated without danger of interrupt; a sequence such as

```
        LOGICAL L
        L = B2
        IF (B1) L = B3
```

may be replaced by

```
        L = (B3.AND.B1) .OR. (B2.AND..NOT.B1).
```

One optimization is to check if B2 and B3 have the same tag; if so the tag can be applied to L and the statements can be treated like a regular conditional expression. Also, if any of B1, B2, B3 are tagged TRUE or FALSE, the expression can be simplified by standard identities.

APPENDIX I

Changing the Tag of an Operand

Tag of I	To Convert to NEG	To Convert to ZEROM	To Convert to ZEROP
FALSE	+0	+0	-0
NEG	I	XSIGN(I)	COMPL(XSIGN(I))
NONZERO	NEO(I)	NEO(I)	EQO(I)
NPLZERO	NEPO(I)*	NEPO(I)	EQPO(I)
PLZERO	EQPO(I)*	EQPO(I)	NEPO(I)
POS	COMPL(I)	COMPL(XSIGN(I))	XSIGN(I)
TRUE	-0	-0	+0
ZERO	EQO(I)**	EQO(I)	NEO(I)
ZEROM	I	I	COMPL(I)
ZEROP	COMPL(I)	COMPL(I)	I

Choice of Branch Instruction, and Negation of Operand

Tag of Operand	To branch on Success	To branch on Failure	Tag of Negation
FALSE	never	always	TRUE
NEG	negative	positive	POS
NONZERO	nonzero	zero	ZERO
NPLZERO	nonzero	zero	PLZERO
PLZERO	zero	nonzero	NPLZERO
POS	positive	negative	NEG
TRUE	always	never	FALSE
ZERO	zero	nonzero	NONZERO
ZEROM	negative	positive	ZEROP
ZEROP	positive	negative	ZEROM

* On the 7600, alternative expansions are COUNT(I)+377777B for NPLZERO and COUNT(I)-1 for PLZERO. Here COUNT denotes population count, and 18-bit arithmetic is used.

** An alternative 7600 expansion is SHIFT(MASK(1), COUNT(I)), where SHIFT, MASK, and COUNT have the obvious meanings.

ANDing Two Expressions

Tag of I*	Tag of J*	Output Tag	Output Expression
NEG	NEG	NEG	AND(I,J)
NEG	NONZERO	NONZERO	AND(J,XSIGN(I))
NEG	NPLZERO	NPLZERO	AND(J,XSIGN(I))
NEG	PLZERO	NEG	AND(I,EQPO(J))
NEG	POS	NEG	ANDNOT(I,J)
NEG	ZERO	NEG	AND(I,EQO(J))
NEG	ZEROM	NEG	AND(I,J)
NEG	ZEROP	NEG	ANDNOT(I,J)
NONZERO	NEG	NONZERO	AND(I,XSIGN(J))
NONZERO	NONZERO	NONZERO	XOR(I+J,I-J)
NONZERO	NPLZERO	NONZERO	XOR(I+J,I-J)
NONZERO	PLZERO	NONZERO	AND(I,EQPO(J))
NONZERO	POS	NONZERO	ANDNOT(I,XSIGN(J))
NONZERO	ZERO	NONZERO	AND(I,EQO(J))
NONZERO	ZEROM	NONZERO	AND(I,J)
NONZERO	ZEROP	NONZERO	ANDNOT(I,J)
NPLZERO	NEG	NPLZERO	AND(I,XSIGN(J))
NPLZERO	NONZERO	NONZERO	XOR(I+J,I-J)
NPLZERO	NPLZERO	NONZERO	XOR(I+J,I-J)
NPLZERO	PLZERO	NPLZERO	AND(I,EQPO(J))
NPLZERO	POS	NPLZERO	ANDNOT(I,XSIGN(J))
NPLZERO	ZERO	NPLZERO	AND(I,EQO(J))
NPLZERO	ZEROM	NPLZERO	AND(I,J)
NPLZERO	ZEROP	NPLZERO	ANDNOT(I,J)
PLZERO	NEG	NEG	AND(J,EQPO(I))
PLZERO	NONZERO	NONZERO	AND(J,EQPO(I))
PLZERO	NPLZERO	NPLZERO	AND(J,EQPO(I))
PLZERO	PLZERO	PLZERO	I+XOR(I,J)
PLZERO	POS	ZEROP	XOR(I,XSIGN(J)-I)
PLZERO	ZERO	ZERO	J+XOR(I,J)
PLZERO	ZEROM	ZEROP	XOR(I,COMPL(J)-I)
PLZERO	ZEROP	ZEROP	XOR(I,J-I)
POS	NEG	NEG	ANDNOT(J,I)
POS	NONZERO	NONZERO	ANDNOT(J,XSIGN(I))
POS	NPLZERO	NPLZERO	ANDNOT(J,XSIGN(I))
POS	PLZERO	ZEROP	XOR(J,XSIGN(I)-J)
POS	POS	POS	OR(I,J)
POS	ZERO	ZEROP	XOR(XSIGN(I)+J,XSIGN(I -J)
POS	ZEROM	NEG	ANDNOT(J,I)
POS	ZEROP	POS	OR(I,J)
ZERO	NEG	NEG	AND(J,EQO(I))

*To save space, the tags TRUE and FALSE have been omitted from this table, but are considered in Appendix II.

Tag of I	Tag of J	Output Tag	Output Expression
ZERO	NONZERO	NONZERO	AND(J,EQO(I))
ZERO	NPLZERO	NPLZERO	AND(J,EQO(I))
ZERO	PLZERO	ZERO	I+XOR(I,J)
ZERO	POS	ZEROP	XOR(XSIGN(J)+I, XSIGN(J)-I)
ZERO	ZERO	ZERO	I+XOR(I,I+J)
ZERO	ZEROM	ZEROP	XOR(COMPL(J)+I, COMPL(J)-I)
ZERO	ZEROP	ZEROP	XOR(J+I,J-I)
ZEROM	NEG	NEG	AND(I,J)
ZEROM	NONZERO	NONZERO	AND(I,J)
ZEROM	NPLZERO	NPLZERO	AND(I,J)
ZEROM	PLZERO	ZEROP	XOR(J,COMPL(I)-J)
ZEROM	POS	NEG	ANDNOT(I,J)
ZEROM	ZERO	ZEROP	XOR(COMPL(I)+J, COMPL(I)-J)
ZEROM	ZEROM	ZEROM	AND(I,J) or I+J
ZEROM	ZEROP	ZEROM	ANDNOT(I,J) or I-J
ZEROP	NEG	NEG	ANDNOT(J,I)
ZEROP	NONZERO	NONZERO	ANDNOT(J,I)
ZEROP	NPLZERO	NPLZERO	ANDNOT(J,I)
ZEROP	PLZERO	ZEROP	XOR(J,I-J)
ZEROP	POS	POS	OR(I,J)
ZEROP	ZERO	ZEROP	XOR(I+J,I-J)
ZEROP	ZEROM	ZEROM	ANDNOT(J,I) or J-I
ZEROP	ZEROP	ZEROP	OR(I,J)

APPENDIX II

Evaluation of $T_1(I_1)$ AND ... AND $T_n(I_n)$

i) Partition the $T_i(I_i)$ into six classes depending on T_i:

Class I:	T_i = FALSE
Class II:	T_i = TRUE
Class III:	T_i = POS or ZEROP
Class IV:	T_i = NEG or ZEROM
Class V:	T_i = PLZERO or ZERO
Class VI:	T_i = NONZERO or NPLZERO

ii) If Class I is nonempty, the AND is identically FALSE. Terminate.

iii) If Class III has more than one element, replace by a single member of Class III. This requires the issuance of several OR instructions, with the result tagged POS or ZEROP.

iv) If Class IV has more than one element, replace by a single member of Class IV. This requires the issuance of several AND instructions, with the result tagged NEG or ZEROM.

v) If Class V has more than one element, replace by a single member of Class V. This requires the repeated replacement of

```
ZERO(I).AND.ZERO(J) by ZERO(I+XOR(I,I+J))
ZERO(I).AND.PLZERO(J) by ZERO(I+XOR(I,J))
PLZERO(I).AND.PLZERO(J) by PLZERO(I+XOR(I,J))
```

Alternatively, if Class V is large, replace

$$\text{PLZERO}(P_1)\text{.AND.} \ \ldots \ \text{.AND.PLZERO}(P_k)\text{.AND.}$$
$$\text{ZERO}(Z_1)\text{.AND.} \ \ldots \ \text{.AND.ZERO}(Z_m)$$

by

```
ZEROP(XOR(TEMP,0-TEMP))
```

where

$$\text{TEMP} = \text{OR}(P_1,\ldots,P_k,Z_1+0,\ldots,Z_m+0)$$

A third possibility, on the CDC 7600, is to use PLZERO(COUNT(TEMP)). Here COUNT returns the number of 1 bits in its argument and TEMP is defined as above. Still another possibility, if the machine has a load magnitude instruction, is to test the OR of the absolute values for +0.

vi) If Class VI has more than one element, replace by a single member of Class VI. This requires the repeated replacement of

```
NONZERO(I).AND.NONZERO(J) by NONZERO(XOR(I+J,I-J))
NONZERO(I).AND.NPLZERO(J) by NONZERO(XOR(I+J,I-J))
NPLZERO(I).AND.NPLZERO(J) by NONZERO(XOR(I+J,I-J))
```

As mentioned in Section 4, it will often be possible to optimize during this process.

vii) Merge Class IV into Class III. If both classes are nonempty, this requires issuing an ANDNOT instruction and tagging the result NEG or ZEROM. Alternatively, if the machine has an ORNOT instruction and Class V is nonempty, use the ORNOT and tag the result POS or ZEROP.

viii) Merge Class V into Class III. If both classes are nonempty, this requires changing the tag of the Class III member to ZEROP, then replacing

```
PLZERO(I).AND.ZEROP(J) by ZEROP(XOR(I,J-I))
ZERO(I).AND.ZEROP(J) by ZEROP(XOR(J+I,J-I))
```

Alternatively, if the Class III member has tag NEG, then change the tag of the Class V member to NEG, issue an AND, and tag the result NEG.

ix) Merge Class VI into Class III. If both classes are nonempty, this requires changing the tag of the Class III member to ZEROM or to ZEROP, then issuing an AND or an ANDNOT, and tagging the result the same as the Class VI member.

x) If Class III is empty, the AND is identically TRUE. Otherwise Class III contains precisely one member; this is the output of the AND.

REFERENCES

Control Data CYBER 170 Series, CYBER 70 Series, 6000 Series, 7000 Series Computer Systems - Fortran Extended Version 4 Reference Manual (Pub. No. 60305600). Control Data Corporation, Software Documentation, Sunnyvale, California, 1971.

Control Data 7600/CYBER 70 Model 76 Computer Systems - Hardware Reference Manual (Pub. No. 60367200). Control Data Corporation, Technical Publications Department, Arden Hills, Minnesota, 1972.

Nichols, J. E., The Structure and Design of Programming Languages, Reading, Massachusetts. Addison-Wesley, 1975, pp. 295-417.

"Draft proposed ANS FORTRAN, BSR X39, X3J3/1976," SIGPLAN NOTICES, 11,3 (March, 1976).