INTRODUCTION TO BRANCHING

UNCONDITIONAL BRANCHING

There are two forms of unconditional branching in the MC68000.

BRA instruction

BRA	<label>	Program control passes directly to the instruction located at label. The size of the jump is restricted to -32768 to +32767.

Example:

```
LOOP:       <instruction>
                  •
                  •
                  •
            BRA LOOP   ;program control passes to the instruction at LOOP
```

FORMAT

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	0	0	0	8-bit displacement (BRA.S)							
16-bit displacement (BRA.L) if 8-bit displacement = 0000000															

JMP Instruction

JMP	<ea>	Program controls jumps to the specified address. There is no restriction on the size of the jump.

FORMAT

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
										<effective address>					
0	1	0	0	1	1	1	0	1	1	mode			register		

Examples:

```
        JMP   AGAIN         ;absolute long addressing mode
        JMP   (A2)          ;address register indirect addressing mode
```

CONDITIONAL BRANCHING

The Bcc instructions

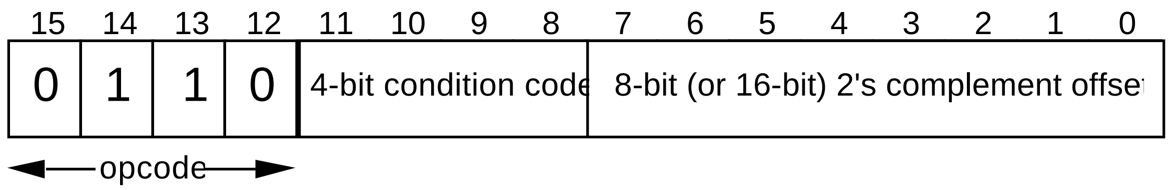
dependent upon the value of a bit in the Status Register

bit	instruction	action
Z	BEQ <label>	branch if SR indicates zero, i.e. Z=1
Z	BNE <label>	branch if SR indicates a non-zero number, i.e. Z=0
N	BMI <label>	branch if SR indicates a negative number, i.e. N=1
N	BPL <label>	branch if SR indicates a positive (this includes zero) number, i.e. N=0
V	BVS <label>	branch if SR indicates that overflow occurred, i.e. V=1
V	BVC <label>	branch if SR indicates that no overflow occurred, i.e. V=0
C	BCS <label>	branch if SR indicates that carry/borrow occurred, i.e. C=1
C	BCC <label>	branch if SR indicates that carry/borrow did not occur, i.e. C=0

NOTE: You don't test the X bit.

The general form of a Bcc

branch instruction



where bits 11-8 indicate the branch condition code, i.e. BHI=0010, BNE=0110, etc.

The offset is relative to the current value of the PC. Recall that the PC is incremented in the read cycle of the instruction. Note that most assemblers automatically use a 16-bit offset using an extension word to automatically handle forward branching.

BIT MANIPULATION INSTRUCTIONS

Can be used to change the value of and test individual bits of a binary word?

BTST	#N,<ea>	value of the tested bit is placed into Z bit of status register
BTST	Dn,<ea>	
BSET	#N,<ea>	sets the value of the specified bit to 1
BSET	Dn,<ea>	
BCLR	#N,<ea>	sets the value of the specified bit to 0
BCLR	Dn,<ea>	
BCHG	#N,<ea>	changes the value of the specified bit, $0 \rightarrow 1$ or $1 \rightarrow 0$
BCHG	Dn,<ea>	

The number of the bit to be tested can be specified as an immediate constant, i.e. #N, or it can be contained in a data register. The allowed range of bits to be tested is 0-7 for a memory location, i.e. it only tests bytes of memory, or 0-31 for a data register.

The BTST instruction is a good way to set a bit prior to a conditional branch.

INSTRUCTIONS WHICH TEST NUMBERS

TEST INSTRUCTION

Can be used to set Status Register bits before a branch instruction. SInce it has only one argument it is called a unary operation.

```
TST.<size>   <ea>
```

size	can be B, W or L
<ea>	cannot be an address register
Action	Sets N and Z according to what is found in <ea>. Clears C and V.

COMPARE INSTRUCTION

Can be used to set Status Register bits before a branch instruction

```
CMP.<size>          <ea>,Dn
CMPI.<size>         #N,<ea>
```

size	can be B, W or L
Action	Computes the difference (destination-source). It DOES NOT change the value of anything contained in <ea> or Dn but does change the Status Register's N,C,Z,V codes.

Computes

Dn - <ea>
<ea> - #N

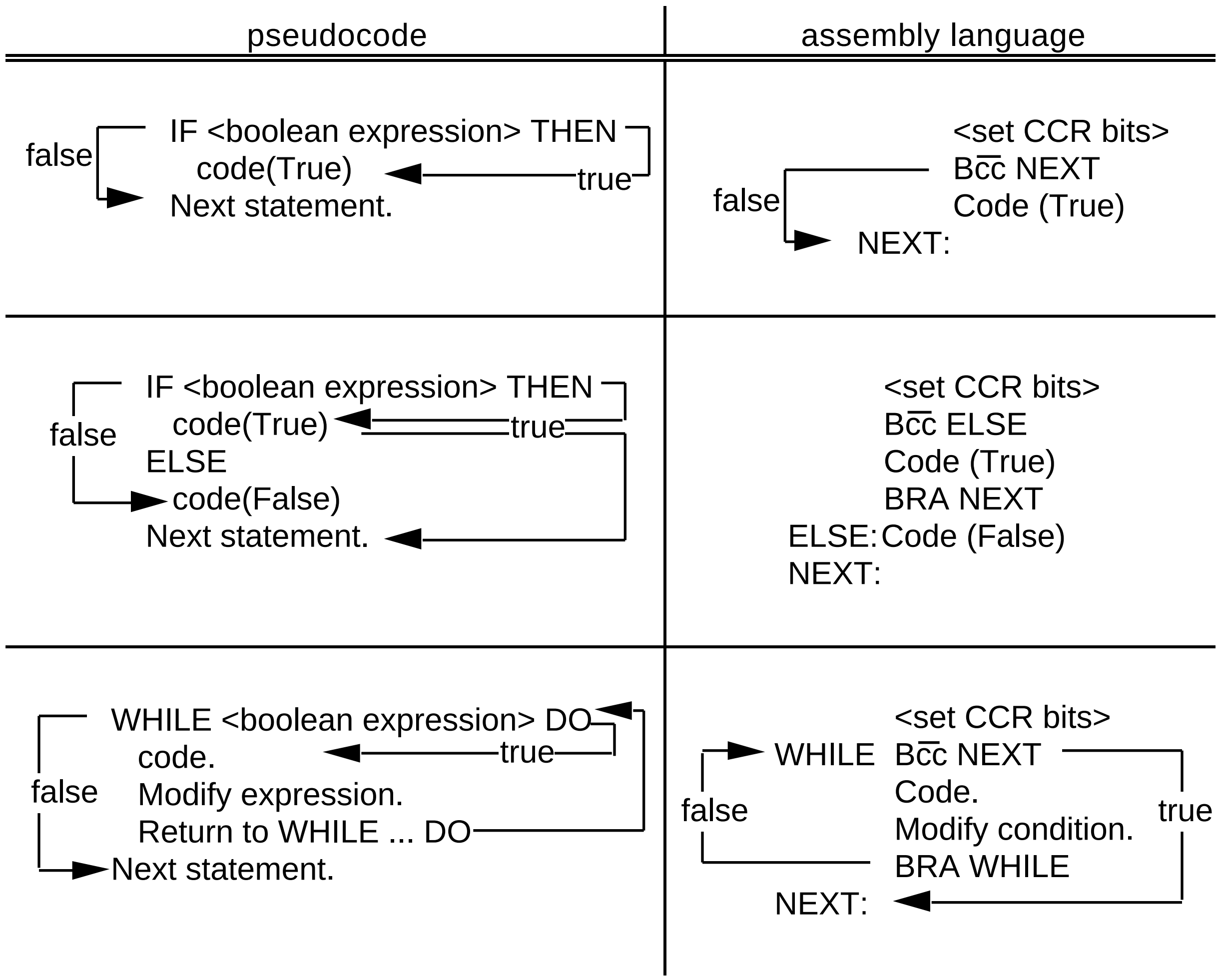
```
CMPA.<size>         <ea>,An
```

size	can be W or L
Action	Subtracts contents of <ea> from 32-bit contents of An, i.e. it computes An-(<ea>). If <ea> is a word it will be sign extended for the subtraction. It DOES NOT change the value of anything contained in <ea> or Dn but does change the Status Register's N,C,Z,V codes.

Computes

An - <ea>

structured programming:



pseudocode	assembly language

IF <boolean expression> THEN
   code(True)
Next statement.
	<set CCR bits>
	B$\overline{cc}$ NEXT
	Code (True)
	NEXT:

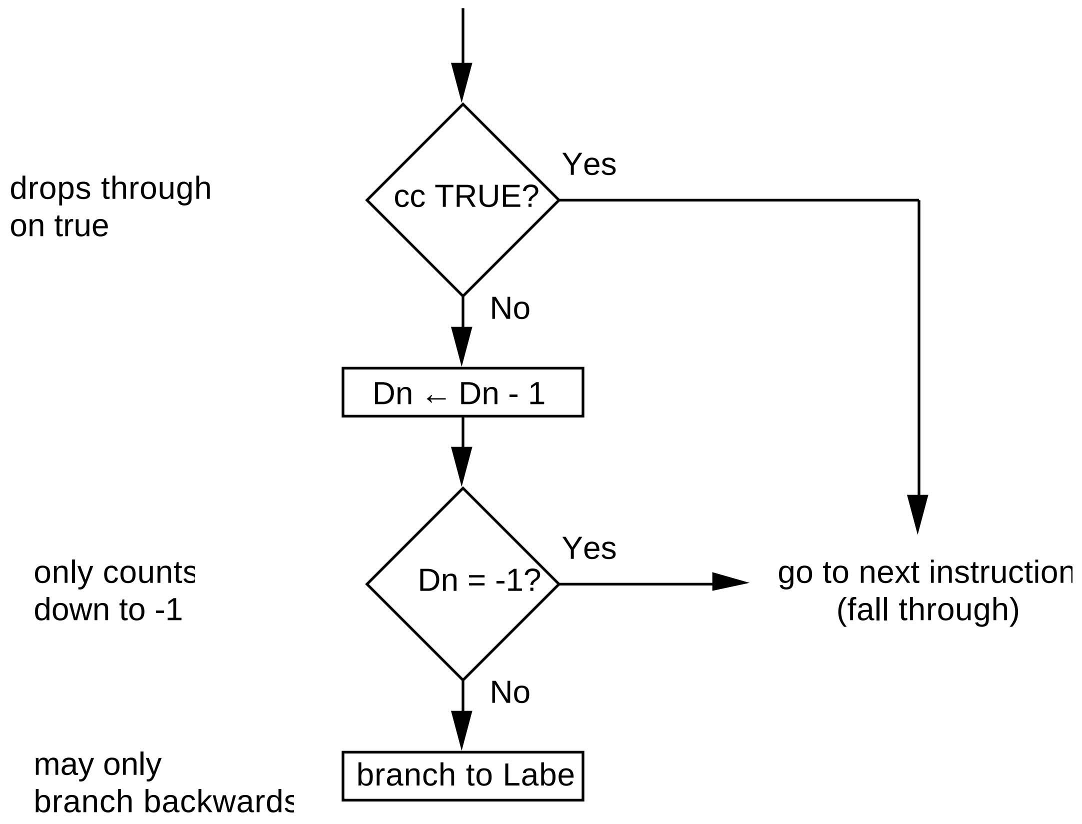
IF <boolean expression> THEN
   code(True)
ELSE
   code(False)
Next statement.
	<set CCR bits>
	B$\overline{cc}$ ELSE
	Code (True)
	BRA NEXT
	ELSE: Code (False)
	NEXT:

WHILE <boolean expression> DO
   code.
   Modify expression.
   Return to WHILE ... DO
Next statement.
	WHILE B$\overline{cc}$ NEXT
	Code.
	Modify condition.
	BRA WHILE
	NEXT:

DBcc instruction

DBcc Dn,<label> Program control passes directly to the instruction located at label if cc is false. This is to be compared with the Bcc instruction which passed control to <label> if cc was true. The logic of this instruction is shown below.

Example: DBcc D0,LOOP



Example:

using the DBcc instruction		using a conventional branch instruction
LOOP ...	⇔	LOOP ....
DBNE D0,LOOP		BNE.S NEXT
		SUBQ #1,D0
		BPL LOOP ;see Note
		NEXT ...

Note: BPL is used in the equivalent code because the form of D0 is to count down to -1. However, the actual DBcc actually checks only for -1.

The DBT instruction does nothing; it simply falls through to the next instruction.

The DBF instruction is used in loops to decrement a loop counter to -1.

Example DBcc instructions:

What is the value of D0 after executing the following instructions?

```
                MOVE.L      #15,D0
LABEL           ADD         D1,D2
                DBF         D0,LABEL
```

Answer: The DBF never satisfies the condition code so it only decrements Do and goes to label. Since it never "falls through" to the next instruction until D0=-1, we know that the result of this loop must be D0=-1. This is the most common form of the DBcc instruction.

What is the value of D0 after executing the following instructions?

```
                MOVE.L      #15,D0
LABEL           SUBQ        #1,D0
                DBT         D0,LABEL
```

Answer: In this case, the condition code is always true and the program flow automatically "falls through" to the next instruction. As a result, the only action of this code is to put 15 into D0, subract 1 from it to get 14, and then "fall through" to the next instruction with D0=14.

Given that (D0)=$ 0012 3456, what is the contents of D0 after the following program segment is executed?

```
        MOVEQ     #1, D0      ;put 1 into counter
LOOP    ADD.W     #1, D0      ;add 1 to counter
        DBF       D0, LOOP    ;if D0≤0 goto loop
        ADD.W     #2, D0      ;add 2 to counter
```

MOVEQ:	D0:	1
ADD.W	D0:	2
DBF	D0:	1
ADD.W	D0:	2

<loop never finishes - infinite loop>

The thing to look for in a problem of this type is that the loop variable is being manipulated inside the loop.

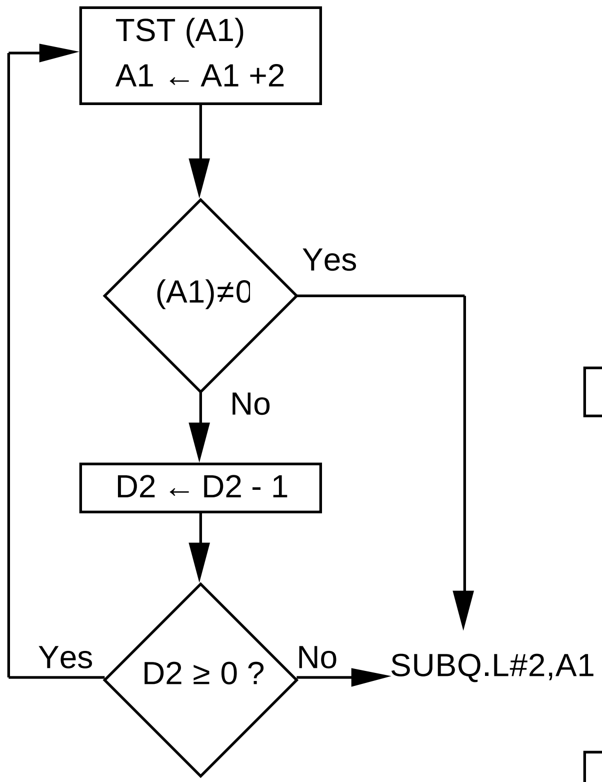
The instruction DBRA is equivalent to DBF.

Rewrite the sequence to use a DBcc instruction:

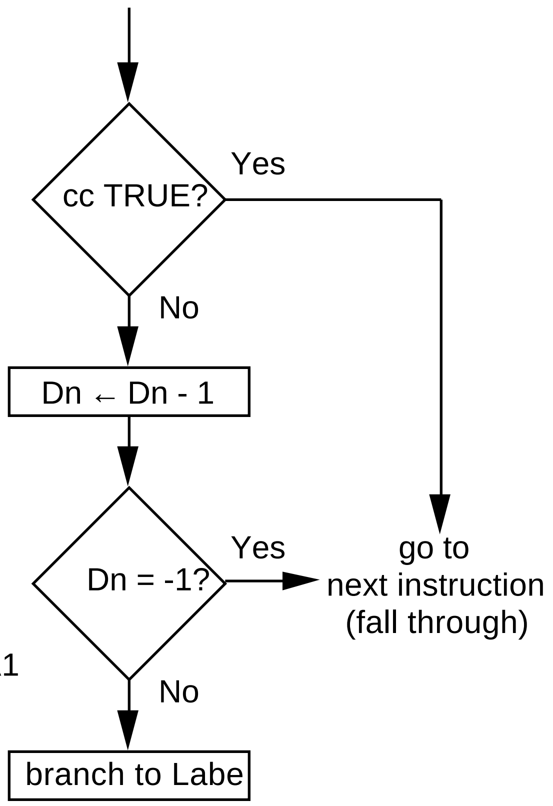
```
LOOP1       TST.W     (A1)+
            BNE       DONE1
            SUBQ.W    #1,D2
            BPL       LOOP1
DONE1       SUBQ.L    #2,A1
```

To answer this problem you need to consider the logic of the loop.

The logic of the program segment



The logic of the DBcc instruction



As you can see the logic of the two loops is almost identical. Dn≥0 is the same as testing Dn=-1. Then, all you need to do is identify the label as being the beinning of the loop, and Dn as being D2 and you have the following code using a DBNE instruction.

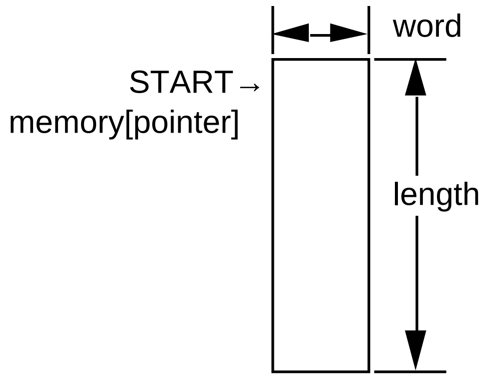
```
LOOP1       TST.W     (A1)+
            DBNE      D2,LOOP1
DONE1       SUBQ.L    #2,A1
```

EXAMPLE: COUNT NEGATIVE NUMBERS

The correct way to design a program is by starting with your inputs, outputs and functional requirements.

Functional specification (pseudocode)

```
                                                  ;define inputs
        START=location of words in memory
        LENGTH=# of words to examine
        TOTAL                                     ;where to put answer

        count=0                                   ;# of negative words
        pointer=START                             ;pointer variable

        if (LENGTH=0) then quit                   ;if length=0 do nothing

loop:                                             ;basic loop for advancing to
                                                  ;next word
        if (memory[pointer] ≥ 0) then             ;if word is not negative
            goto looptest                         ;then don't count it
        count=count+1                             ;advance negative word
                                                  counter
looptest:
        pointer=pointer+1                         ;increment the word pointer
        LENGTH=LENGTH-1                           ;decrement the word counter
        if LENGTH>=0 then goto loop               ;if more words then repeat

quit:
```



Structure of DBF	Negative counting program

IF (A0) > 0 then
   code (TRUE)
else
   code (FALSE)
Next statement.
	Loop: IF (A0) > 0 then
	   count=count-1
	   if count = -1 then goto Done
	   goto loop
	else
	   count=count+1
	Done: output

NOTE: This illustrates one of the most useful modes of the
DBcc Dn,<label>
instruction where cc=F. The F means that the conditional code is ALWAYS false and the conditional test to “drop through” to the next instruction will never occur. In this mode the DBF instruction is very similar to a simple DO loop where Dn is the loop variable.

PROGRAM

```
DATA      EQU       $6000         ;data placed at $6000
PROGRAM   EQU       $4000         ;program begins at $4000

          ORG       DATA
LENGTH    DC.W      $1000         ;$1000 numbers to check
START     DC.L      $10000        ;data begins at $10000
TOTAL     DS.W      1             ;put answer here

          ORG       PROGRAM
main:     MOVEA.L   START,A0      ;load starting address, could also
                                  use LEA instruction
          MOVEQ     #0,D0         ;set count to zero
          MOVE.W    LENGTH,D1     ;load length of memory area
                                  ;into D1
          BEQ.S     DONE          ;if size of memory is zero
                                  ;then quit
LOOP:     TST.W     (A0)+         ;compares (A0) with 0
                                  ;sets Z bit if (A0)<0
          BPL.S     LPTEST        ;if (A0) ≥ 0 goto looptest,
                                  branches if N=0
          ADDQ.W    #1,D0         ;if (A0)<0 increment neg counter
LPTEST:   DBF       D1,LOOP       ;decrement and branch, could
                                  also use DBRA instruction
                                  ;decrement  memory counter D1
                                  ;if counter ≥ then repeat
                                  ;end of program
          MOVE.W    D0,TOTAL      ;put answer somewhere
DONE:     TRAP      #0
```

MORE BRANCH INSTRUCTIONS

The previous branch instructions only tested a single bit of the CCR. Many times you want to test things, like whether a number is greater than or equal to another number, which require testing more than one bit. These operations are designed for signed number comparisons and usually follow a CMP instruction.

Bcc instructions appropriate for signed numbers
The logic assumes a CMP <source>,<destination> command immediately precedes the instruction. Remember that the CMP instruction computes (destination-source) without changing either source or destination. These branches are appropriate for signed numbers since they use the N bit.

instruction	action	logic
BGT <label>	branch if destination > source	branch if NV~Z+~N~V~Z
BGE <label>	branch if destination≥source	branch if NV+~N~V
BLE <label>	branch if destination≤source	branch if Z+(N~V+~NV)
BLT <label>	branch if destination<source	branch if N~V+~NV

where "~" indicates a logical NOT (i.e., an inversion)

Bcc instructions appropriate for unsigned numbers
The logic assumes a CMP <source>,<destination> command immediately precedes the instruction. Remember that the CMP instruction computes (destination-source) without changing either source or destination. These branches are appropriate for unsigned numbers since they do NOT use the N bit.

instruction	action	logic
BHI <label>	branch if destination > source	branch if ~C~Z
BCC <label>	branch if destination≥source	branch if ~C
BLS <label>	branch if destination≤source	branch if C+Z
BCS <label>	branch if destination<source	branch if C

CMP instruction:
Computes (Destination) - (Source)

X	N	Z	V	C
-	*	*	*	*
	set if result is negative	set if result is zero	set if an overflow is generated	set if borrow is generated

Example:
For the following program segment:

```
              CLR.L     D1                ;clear the register D1 for sum
              MOVE.L    #10,D0            ;counter (D0) = 10 decimal
LOOP:         ADD.L     D0,D1             ;add counter 10 to 0 (first time)
              SUBQ      #1,D0             ;subtract 1
              BGE       LOOP              ;if counter≥0 goto loop
              TRAP      #0                ;end of program
              END
```

How many times does the SUBQ gets executed and what is (D1) after the program stops?

at	after ADD.L instruction	after SUBQ instruction
D0: 10	(D1)=10	(D0)=9
D0: 9	(D1)=10+9	(D0)=8
D0: 8	(D1)=10+9+8	(D0)=7
D0: 7	(D1)=10+9+8+7	(D0)=6
D0: 6	(D1)=10+9+8+7+6	(D0)=5
D0: 5	(D1)=10+9+8+7+6+5	(D0)=4
D0: 4	(D1)=10+9+8+7+6+5+4	(D0)=3
D0: 3	(D1)=10+9+8+7+6+5+4+3	(D0)=2
D0: 2	(D1)=10+9+8+7+6+5+4+3+2	(D0)=1
D0: 1	(D1)=10+9+8+7+6+5+4+3+2+1	(D0)=0
D0: 0	(D1)=10+9+8+7+6+5+4+3+2+1+0	(D0)=-1

BGE will branch if NV+~N~V (destination ≥ source)

There is no overflow until D0=-1

D0-1		D0	N	V	NV+~N~V
1 - 1	→	0	0	0	0•0+1•1=1 so branch
0 - 1	→	-1	1	0	1•0+0•1=0 so drop through

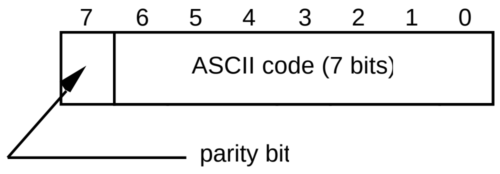
Note that on the last calculation we have

$$\begin{array}{l} \texttt{0000} \\ \underline{\texttt{FFFF}} \\ \texttt{FFFF} \end{array}$$

which sets N=1 (the result is negative) but there is no signed overflow so V=0.

The SUBQ gets executed 11 times.

Review of ASCII character representation:



ASCII uses 8 bits to represent characters. Actually, only 7 bits are used to uniquely define the character and the 8-th bit (called the parity bit) is used for error detection. When used, the value of the parity bit depends upon the numbers of 1's in bits 0-7. For odd parity, bit 8 is set to make the total number of 1's in the byte an odd number such as 1 or 7. For even parity, bit 8 is set to make the total number of 1's in the byte an even number such as 0, 2 or 8.

Some useful ASCII character codes:

character	ASCII code (in hex)
/	2F
0	30
1	31
2	32
8	38
9	39
:	3A
;	3B
@	40
A	41
B	42
Z	5A
[	5B
\	60
a	61
z	7A
{	7B

etc.

EXAMPLE: PARITY PROGRAM

The correct way to design a program is by starting with your inputs, outputs and functional requirements.

Functional specification (pseudocode)

```
get ASCII byte
sum bits 0 thru 6
put bit(0) of sum in bit(7) of ASCII byte
put ASCII byte somewhere
```

Now define how to sum bits 0 thru 6

```
set counter to 0                ;bit pointer
set sum to 0                    ;sum of bits

loop:
sum=sum+byte[counter]           ;sum up bits 0...6
                                ;byte is ASCII character being
                                ;processed
counter=counter+1
if counter<7 goto loop
byte[7]=sum[bit0]               ;if sum[bit0] is 1 the sum is odd
                                ;if sum[bit1] is 0 the sum is even
                                ;this program generates even
                                ; parity
```

For even parity, if bits 0 thru 6 sum to an odd number then set bit #7 to 1 to make the parity even. If you wanted to change the program to odd parity, you simply need to change the last line of the pseudocode.

Examples:

If the sum of the character's bits is an odd number then the parity bit must be set to 1.

1	0	0	0	0	1	1	1

If the sum of the character's bits is an even number then the parity bit must be set to 0.

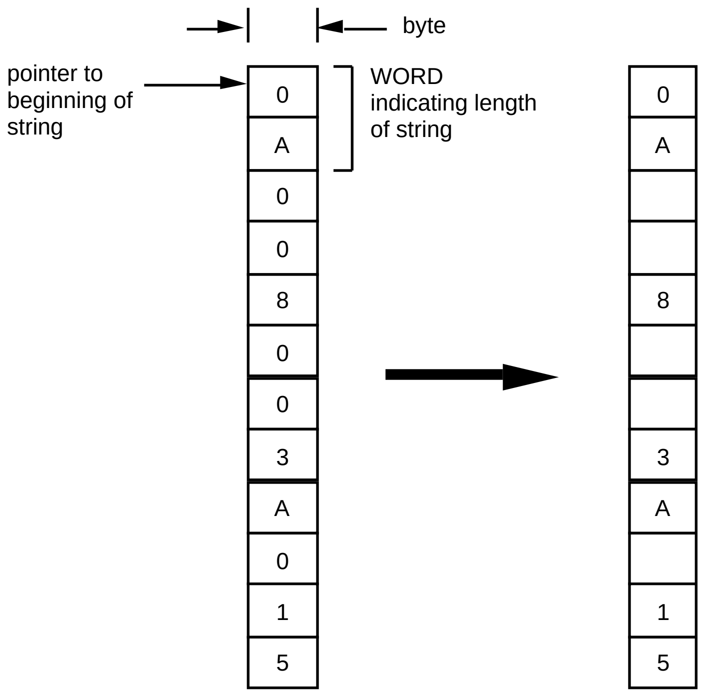
0	0	0	0	0	1	1	0

MC68000 assembly code for parity program:

```
main_loop   EQU      *
* could have also used i/o to get data from keyboard
            MOVE.B   $1000,D1        ;get ASCII byte from $1000
* used quick instructions but not necessary
            MOVEQ    #0,D0           ;clear counter
            MOVEQ    #0,D2           ;clear sum

SUM         BTST.B   D0,D1           ;test D0-th bit of D1, sets Z-bit
            BEQ      SKIP_INCRE      ;if Z-bit=0 don't increment sum
            ADDQ     #1,D2           ;sum=sum+1

SKIP_INCRE
            ADDQ     #1,D0           ;increment counter

            MOVE     D0,D3           ;temp storage in D3
* subtract seven and compare to zero
            SUBQ     #7,D3           ;counter=7?
* could have used a compare instruction here
            BNE      SUM             ;No, sum more bits
            BCLR     #7,D1           ;Yes, clear parity bit
            BTST     #0,D2           ;get parity bit from sum[0]
            BEQ      PAR_SET         ;if parity bit=0 goto PAR_SET
            BSET     #7,D1           ;set parity bit to 1
PAR_SET     MOVE.B   D1,????         ;put ASCII byte somewhere
```

EXAMPLE: REPLACING 0's BY BLANKS PROGRAM

The correct way to design a program is by starting with your inputs, outputs and functional requirements.


byte
pointer to beginning of string
WORD indicating length of string
0
A
0
0
8
0
0
3
A
0
1
5
0
A
8
3
A
1
5

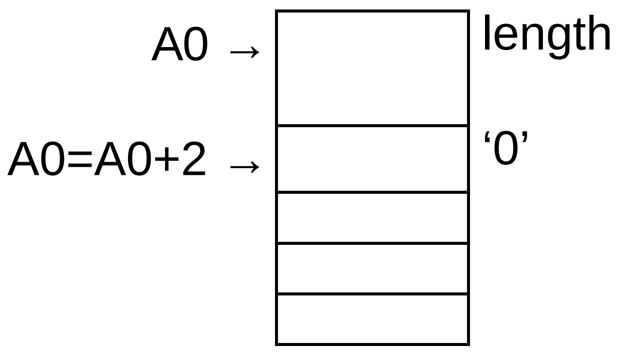
Functional specification (pseudocode)

```
                                    ;define inputs
pointer=location of character string in memory
length=length of string (bytes)     ;this will be contained in first
                                    ;word of string input
blank=' '                           ;define a blank character
if (length=0) then quit             ;if string length=0 do nothing

nextchar:                           ;basic loop for advancing to
                                    ;next character
if (char[pointer]≠'0') then         ;if character is NOT a zero
   goto notzero                     ;then goto nonzero
char[pointer]=blank                 ;replace ASCII zero by blank
notzero:
length=length-1                     ;decrement the char counter
if (length≥0) goto nextchar         ;if more characters then repeat
```

What the program does is search for all the ASCII zeros in the string and replace them with blanks. This might be useful for eliminating leading zeros in a print routine.

SAMPLE PROGRAM

```
            ORG        $6000
START       DS.L       1               ;START is the address of the string
CHAR_0      EQU.B      '0'             ;define CHAR_0 as ASCII 0
BLANK       EQU.B      ' '             ;define BLANK as ASCII space

            ORG        $4000
begin       MOVEA.L    START,A0        ; set pointer to start of string,
                                       cannot use LEA START
            MOVEQ      #BLANK,D1       ; put a blank in D1
            MOVE.W     (A0)+,D2        ; get length of string
            BEQ        DONE            ; if the string is of length zero
                                       ;then goto DONE
NEXT_CHAR:
            MOVEQ      #CHAR_0,D0      ;put ASCII 0 into D0
            SUB.B      (A0)+,D0        ;compute '0'-current character
            BNE        NOT_ZERO        ;goto next char if non-zero
            MOVE.B     D1,-1(A0)       ;go back, get last byte and
                                       ;replace it by ASCII zero
N OT_ZERO:
            SUBQ       #1,D2           ;decrement the character counter
            BPL        NEXT_CHAR       ;if count >=0 go to next character
                                       ;otherwise quit
DONE        END        begin
```



EXAMPLE: LONG DIVISION USING REPEATED SUBTRACTION

Input, using HexIn, nonnegative numbers M and N where N>0. Using repreated subtraction, find the quotient M/N and remainder.

Algorithm

Repeatly subtract the divisor N from M (M:=M-N). Count the number of iterations Q until M<0. This is one too many iterations and the quotient is then Q-1. The remainder is M+N, the previous value of M.

<u>Pseudocode:</u>

```
QUOTIENT:=0;
READLN(M);      {No error checking.  Assume M≥0}
READLN(N);      {No error checking.  Assume N≥0}
REPEAT
        QUOTIENT:=QUOTIENT+1;
        M:=M-N;
UNTIL M<0;
QUOTIENT:=QUOTIENT-1;
REMAINDER:=M+N;
```

<u>Sample calculations:</u>

Suppose Q=$0000, R=$0000

Start with M=$0015, N=$0004 {corresponds to 15/4 = 4 w /remainder=3}

```
Q=1:  M=M-N=$0015-$0004=$0011
Q=2:  M=M-N=$0011-$0004=$000D
Q=3:  M=M-N=$000D-$0004=$0009
Q=4:  M=M-N=$0009-$0004=$0005
Q=5:  M=M-N=$0005-$0004=$0001
Q=6:  M=M-N=$0001-$0004=$FFFD
```

Since quotient is negative stop algorithm and back up one.

```
Q=Q-1=6-1=5                  ;correct quotient
R=M+N=$FFFD+$0004=$0001      ;correct remainder
```

SAMPLE PROGRAM

```
            INCLUDE     io.s            ;contains the i/o routines
            ORG         $6000
START       MOVE.W      #0,D2           ;quotient in D2, set to zero
GETM        JSR         HexIn           ;get M, put in D0
            TST.W       D0              ;test for M≥0
            BMI         GETM            ;if M<0 get another M
            MOVE.W      D0,D1           ;put M in D1
GETN        JSR         HexIn           ;get N, put in D0
            TST.W       D0              ;test for N>0
            BPL         LOOP            ;if N>0, start calculations
            BRA         GETN            ;if N≤0 get another N
LOOP        ADDI.W      #1,D2           ;increment the quotient
            SUB.W       D0,D1           ;compute M-N
            BPL         LOOP            ;branch back if M not negative,
                                        corresponds to doing another
                                        division
RESULT      SUBI.W      #1,D2           ;decrement the quotient
            ADD.W       D0,D1           ;set remainder
            MOVE.W      D2,D0           ;move quotient to D0
            JSR         HexOut          ;display quotient
            MOVE.W      D1,D0           ;move remainder to D0
            JSR         HexOut          ;display remainder
            JSR         NewLine         ;advance to next line
            TRAP        #0              ;trick to end program
            END         START
```

EXAMPLE: Tests for Signed and UnSigned Overflow

Description:
Enter two 16-bit numbers and compute their sum. The addition operation sets the CCR bits. These bits are then read from the SR into the least significant word of D0 using the MOVE SR,Dn instruction. After isolating the C and V bits in D0, a message indicating if overflow has occurred is printed.

Pseudocode:

```
READLN(M);              /*No error checking.  Assume M≥0*/
READLN(N);              /*No error checking.  Assume N≥0*/
M:=M+N;
D0:=SR;                 /*put the value of the SR into D0*/
D0:=D0&&0x0003;         /*Clear bits 2-15 by ANDing with $0003*/
WRITELN(D0);            /*Write out D0*/
SWITCH (D0) {
        CASE 1:   WRITELN('NO OVERFLOW'); BREAK;
        CASE 2:   WRITELN('ONLY UNSIGNED OVERFLOW');
                  BREAK;
        CASE 3:   WRITELN('ONLY SIGNED OVERFLOW'); BREAK;
        CASE 4:   WRITELN('SIGNED AND UNSIGNED
                  OVERFLOW'); BREAK;
        DEFAULT;
        }
```

MASKing:
ANDI.W #$3,D0 masks bits 0-1

0003_{16} = $0000\ 0000\ 0000\ 0011_2$
$\underline{(D0) = \quad xxxx\ xxxx\ xxxx\ xxxx_2}$
$(D0) = \quad 0000\ 0000\ 0000\ 00xx_2$

SInce the AND operates according to 0•x=0 and 1•x=x the result contains only whatever was is bits 0 and 1 — all other bits were set to zero. Basically we masked out bits 0 and 1; hence the name, masking.

SAMPLE PROGRAM

```
            INCLUDE   io.s           ;contains the i/o routines
            ORG       $6000
START       JSR       HexIn          ;get M, put in D0
            MOVE.W    D0,D1          ;put M in D1
            JSR       HexIn          ;get N, put in D0
            ADD.W     D0,D1          ;D0:=M+N
            MOVE      SR,D0          ;get contents of SR
            ANDI.W    #$0003,D0      ;clears bits 2-15
            JSR       HexOut         ;display C and V bits
            LEA       OVRFLSTR,A1    ;base address of output messages
            ADD.W     D0,D0          ;compute 4*D0 by adding D0 to
                                     itself twice
            ADD.W     D0,D0          ;faster than a multiply
            ADDA.L    D0,A1          ;add message offset to base
                                     address
            MOVEA.L   (A1),A0        ;set (A1) to start address of
                                     message
            MOVE.W    #28,D0         ;each string has 28 characters
                                     (bytes)
            JSR       StrOut         ;string output routine
            JSR       NewLine        ;advance line
            TRAP      #0             ;exit to debugger

OVRFLSTR    DC.L      NO_OVR,USGNOVR,SGNOVR,DUALOVR
NO_OVR      DC.B      ‘NO OVERFLOW                 ’
USGNOVR     DC.B      ‘ONLY UNSIGNED OVERFLOW      ’
SGNOVR      DC.B      ‘ONLY SIGNED OVERFLOW        ’
DUALOVR     DC.B      ‘UNSIGNED AND SIGNED OVERFLOW’
            END       START
```

HOW DOES PROGRAM IMPLEMEMENT SWITCH:

```
LEA          OVRFLSTR,A1
```
loads the base address of the table of messages

D0 can only have the values

D0	V	C
0	0	0
1	0	1
2	1	0
3	1	1

Multiply D0 by 4 to make these values in D0 correspond to the message since

```
OVRFLSTR          DC.L          NO_OVR,USGNOVR,SGNOVR,DUALOVR
```
places the beginning addresses of the messages in consecutive long words beginning at OVRFLSTR.

Use
```
MOVEA.L   (A1),A0
```
to get the starting address of the correct message into A0

NOTE:
```
MOVEA.L   A1,A0
```
will simply place the address of the address of the message into A0 which is NOT what was wanted.

The instruction
```
LEA          0(A1,D0.W),A0
```
would have also worked by directly adding the offset

ROTATE AND SHIFT INSTRUCTIONS

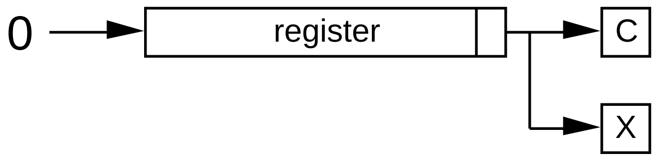
logical shift for **unsigned** numbers

Provide a means for shifting blocks of bits within a register or memory.

Logical shift right

LSR.<size>	#N,Dn
LSR.<size>	Dm,Dn
LSR.W	<ea>

Action: The contents of the data register Dn are shifted right by the number of bits specified in the source operand. The vacated bits are filled with zeros. The shifted bits are stored in the X and C bits of the Status Register.



Notes:

1. A shift in the range 1-8 may be written as immediate data; anything larger than 8 will be replaced by Nmod8. A shift in the range 0-63 may be contained in a data register Dm.
2. Use of the <ea> operand will result in a shift of exactly one bit. The size for this operand can only be word.
3. The result of the operation is specified by the N and the Z bits. The overflow (V) bit is always cleared.

Example:

```
LSR    #4,D3
```

BEFORE

32	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
▨	0	1	0	0	1	0	1	0	0	1	1	1	1	0	1	0

STATUS REGISTER	X	C
	0	0

AFTER

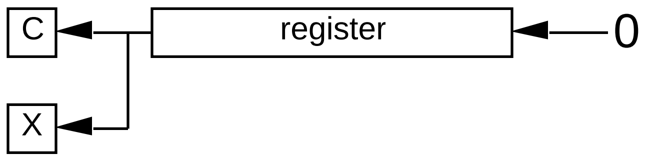
32	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
▨	0	0	0	0	0	1	0	0	1	0	1	0	0	1	1	1

STATUS REGISTER	X	C
	1	1

<u>Logical shift left</u>

LSL.<size>	#N,Dn
LSL.<size>	Dm,Dn
LSL.W	<ea>

Action The contents of the data register Dn are shifted left by the number of bits specified in the source operand. The vacated bits are filled with zeros. The shifted bits are stored in the X and C bits of the Status Register.



Notes:

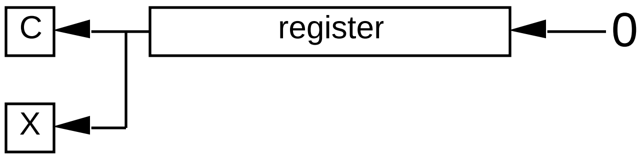
1. A shift in the range 1-8 may be written as immediate data; anything larger than 8 will be replaced by Nmod8. A shift in the range 0-63 may be contained in a data register Dm.
2. Use of the <ea> operand will result in a shift of exactly one bit. The size for this operand can only be word.
3. The result of the operation is specified by the N and the Z bits. The overflow (V) bit is always cleared.

arithmetic shift for **signed** numbers

Arithmetic shift left

ASL.<size>	#N,Dn	;shifts Dn by #N, #N must satisfy $1 \le \#N \le 8$
ASL.<size>	Dm,Dn	;shifts Dn by Dm
ASL.W	<ea>	;shifts word in memory by ONLY 1 bit

Action The contents of the data register are shifted preserving the sign of the original number. A shift count in the range 1-8 can be written as immediate data (#N). A shift count in the range 0-63 may be contained in data register Dm.

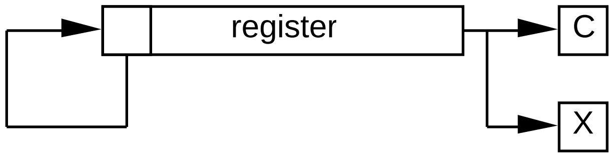


NOTES:
1. The size parameter can be byte, word or long word. If the shift is greater than 8 bits it MUST be stored in a data register Dm.
2. ASL <ea> can only operate on words and can only shift 1 bit.
3. The shift count can be loaded into Dm during program execution allowing variable shift counts in loops.
4. It is faster to move data to a register and shift it than using multiple ASL <ea> commands if the shift is greater than or equal to three bits.
5. An overflow is set if the sign bit changes. Consider the binary number $0110_2=6_{10}$. An ASL of 1 bit produces the number $1100_2=-4_{10}$. More formally, the V bit indicates if a sign change occurred. The Z and N bits are set according to the result of the operation. With ASL bits shifted out of the high-order bit go to both the X and C bits.

Arithmetic shift right

ASR.<size>	#N,Dn	;shifts Dn by #N, #N must satisfy 1≤#N≤8
ASR.<size>	Dm,Dn	;shifts Dn by the value in Dm
ASR.W	<ea>	;shifts word in memory by ONLY 1 bit

Action The contents of the data register are shifted preserving the sign of the original number. A shift count in the range 1-8 can be written as immediate data (#N). A shift count in the range 0-63 may be contained in data register Dm.



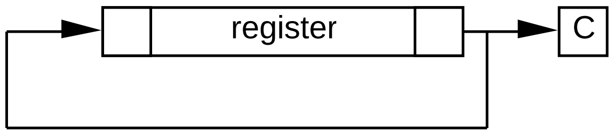
NOTES:
1. Consider the binary number 1010_2=-6_{10}. An ASR of 2 bits produces the number 1110_2=-2_{10}. The circular nature of the MSB in this instruction is, in effect, a sign extension to preserve the sign of the signed number.
2. The size parameter can be byte, word or long word. If the shift is greater than 8 bits it MUST be stored in a data register Dm.
3. ASL <ea> can only operate on words and can only shift 1 bit.
4. The shift count can be loaded into Dm during program execution allowing variable shift counts in loops.
5. It is faster to move data to a register and shift it than using multiple ASR <ea> commands if the shift is greater than or equal to three bits.
6. The overflow bit (V) is set if the sign bit changes. As the sign is preserved in the shift this should never occur. The Z and N bits are set according to the result of the operation. Bits shifted out of the least significant bit go to both the X and C bits.

Rotate instructions are similar to shift instructions; however, rotate instructions do an end around, shifts do NOT

rotate right

ROR.<size>	#N,Dn	;rotates Dn by #N, #N must satisfy 1≤#N≤8
ROR.<size>	Dm,Dn	;rotates Dn by Dm
ROR.W	<ea>	;shifts word in memory by ONLY 1 bit

Action	The bits of the destination are rotated right The extend bit is NOT included in the rotation. The number of bits rotated is determined by the source operand.

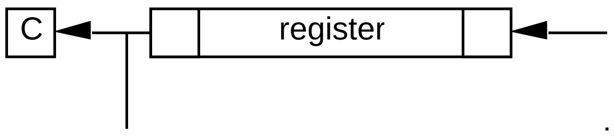


NOTES:
1. The bits rotated out of the least significant bit of the operand go to both the carry bit and the most significant bit of the operand.
2. The size parameter can be byte, word or long word. If the rotation is greater than 8 bits it MUST be stored in a data register Dm.
3. ROR <ea> can only operate on words and the rotation is always 1 bit.

rotate left

ROL.<size>	#N,Dn	;rotates Dn by #N, #N must satisfy 1≤#N≤8
ROL.<size>	Dm,Dn	;rotates Dn by Dm
ROL.W	<ea>	;shifts word in memory by ONLY 1 bit

Action	The bits of the destination are rotated left. The extend bit is NOT included in the rotation. The number of bits rotated is determined by the source operand.



NOTES:
1. The bits rotated out of the most significant bit of the operand go to both the carry bit and the least significant bit of the operand.
2. The size parameter can be byte, word or long word. If the rotation is greater than 8 bits it MUST be stored in a data register Dm.

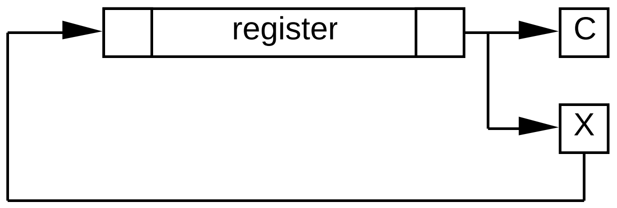
3. ROL <ea> can only operate on words and the rotation is always 1 bit.

rotate with extend instructions

<u>rotate right with extend</u>

```
ROXR.<size>   #N,Dn         ;rotates Dn by #N, #N must satisfy 1≤#N≤8
ROXR.<size>   Dm,Dn         ;rotates Dn by Dm
ROXR.W        <ea>          ;rotates word in memory by ONLY 1 bit
```

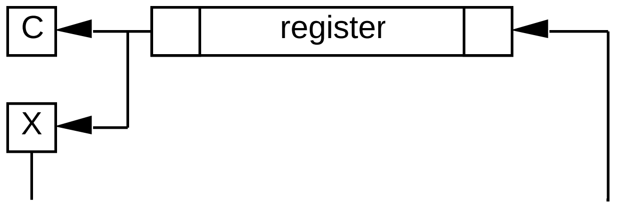
Action The bits of the destination are rotated right with the X bit included in the rotation. The number of bits rotated is determined by the source operand. The least significant bit of the operand is shifted into the C and X bit. The X bit is shifted into the most significant bit of the operand. This process continues for each succeeding shift.



<u>rotate left with extend</u>

```
ROXL.<size>   #N,Dn         ;rotates Dn by #N, #N must satisfy 1≤#N≤8
ROXL.<size>   Dm,Dn         ;rotates Dn by Dm
ROXL.W        <ea>          ;rotates word in memory by ONLY 1 bit
```

Action The bits of the destination are rotated left with the X bit included in the rotation. The number of bits rotated is determined by the source operand. The most significant bit of the operand is shifted into the C and X bit. The X bit is shifted into the most significant bit of the operand. This process continues for each succeeding shift.



EXAMPLE: SIMPLE MATH PROGRAM

The correct way to design a program is by starting with your inputs, outputs and functional requirements.

This program accepts as input a 16-bit signed number N and outputs the following values:

N	2*N	16*N	N DIV 2	N DIV 16

Functional specification (pseudocode)

```
        get signed number N
        multiply by 2 using left shift by 1
        multiply by 16 using left shift by 4
        divide by 2 using right shift by 1
        divide by 16 using right shift by 1
```

MC68000 assembly code for simple math program:

```
         ORG      $5000        ;put data here
NEWLINE  DC.B     $0A,$0       ;ascii code for carriage return
                               followed by end of string
                               character “0”
         include  io.s         ;insert appropriate code for io
                               routines
start    JSR      HexIn        ;get N and put in D0
         MOVE.W   D0,D1        ;copy N to D1 for safekeeping
         JSR      HexOut       ;output N
         ASL.W    #1,D0        ;multiply N by 2 by shifting left by
                               1
         JSR      HexOut       ;output 2*N
         MOVE.W   D1,D0        ;get new copy of N
         ASL.W    #4,D0        ;multiply N by 2^4 by shifting left
                               by 4
         JSR      HexOut       ;output 2^4*N
         MOVE.W   D1,D0        ;get new copy of N
         ASR.W    #1,D0        ;divide N by 2 by shifting right by
                               1
         JSR      HexOut       ;output N DIV 2
         MOVE.W   D1,D0        ;get new copy of N
         ASR.W    #4,D0        ;divide N by 2^4 by shifting right
                               by 4
         JSR      HexOut       ;output N DIV 2^4
         LEA      NEWLINE,A0   ;load starting address of new line
                               control characters into A0
         JSR      PrintString
         END      start
```

EXAMPLE: BLANK SEARCH PROGRAM

This program will search a string of ASCII characters for the first non-blank character and return the address of this character.

STRING	sequence of ASCII characters
START	starting address of STRING in memory
POINTER	address of first non-blank character in STRING

Functional specification (pseudocode)

```
point = START;

LOOP:
IF character(point) = blank THEN
    point = point + 1;
    goto LOOP;
    END
POINTER = point;
```

MC68000 assembly code for blank search program:

```
        ORG      $2000
START   DS.L     1              ;contains starting address of
                                string
POINTER DS.L     1              ;answer, will contain address of
                                first non-blank character
BLANK   equ      $32            ;ASCII code for blank space
        include  io.s           ;insert appropriate code for io
                                routines
start   MOVEA.L  START,A0       ;set A0 to start of string
        MOVEA.L  POINTER,A1     ;set A1 to answer
        MOVE     #BLANK,D1      ;put ASCII blank into D1
LOOP    CMP.B    (A0)+,D1       ;is current character a blank?
        BEQ      LOOP           ;if YES, then continue looping
        SUBA     #1,A0          ;if NO, then point = point -1 to
                                correct for previous (A0)+
        MOVE.L   A0,(A1)        ;save address of first non-blank
                                character in POINTER
        END      start
```

EXAMPLE: ASCII SEARCH PROGRAM (2)

This program will search a block of memory containing ASCII characters for a specified character and return the address of the first occurrance of the specified character.

CHAR	character to search for
BLOCK	memory block containing ASCII characters
START	starting address of BLOCK in memory
STOPA	ending address of BLOCK in memory
POINTER	address of specified character in BLOCK

Functional specification (pseudocode)

```
point = START;

LOOP:
IF character(point) ≠ CHAR THEN
        BEGIN
        point = point + 1;
        IF point ≤ STOP THEN goto LOOP;
        END
POINTER = point - 1;
```

MC68000 assembly code for ascii search program:

```
          ORG       $70000
BSTART    DC.L      $2000           ;start of BLOCK to search
BSTOP     DC.L      $4000           ;end of BLOCK to search

CHAR      equ       $40             ;ASCII character to search for
prog      MOVEA.L   BSTART,A0       ;set A0 to start of BLOCK
          MOVEA.L   BSTOP,A1        ;set A1 to end of BLOCK
          MOVE      #CHAR,D1        ;put ASCII character into D0
LOOP      CMP.B     (A0)+,D0        ;is current character what we are
                                    searching for?
          BEQ       DONE            ;if YES, then get out of here
          CMPA.L    A0,A1           ;if NO, then have we searched
                                    entire block?
          BCC       LOOP            ; this is a CARRY CLEAR
                                    instruction and is equivalent to ≤
                                    comparison since there will be
                                    no carry (actually borrow in this
                                    case) if A0 ≤ A1
DONE      SUBA      #1,A0           ;adjust A0 to correct for the post
                                    increment in the CMP instruction
          END       prog
```

EXAMPLE: WORD SEARCH PROGRAM

This program will search for a given word in memory.

WORD	word to search for
BLOCK	block of memory containing ASCII characters
START	starting address of BLOCK in memory
STOP	ending address of BLOCK in memory
POINTER	address of specified character in BLOCK

Functional specification (pseudocode)

```
point = START;

LOOP:
IF word(point) = WORD THEN
        BEGIN
        point = point + 2;
        IF point ≤ STOP THEN goto LOOP;
        END
POINTER = point - 2;
```

MC68000 assembly code for word search program:

```
        ORG     $3000
START   DC.L    $2000       ;start of memory to search
STOPA   DC.L    $4000       ;end of memory to search
WORD    DC.W    $4E40       ;word to search for

prog    MOVEA.L START,A0    ;set A0 to starting address of
                            search
        MOVEA.L STOPA,A1    ;set A1 to ending address of
                            search
        MOVE    WORD,D0     ;put search word into D0
LOOP    CMP.W   (A0)+,D0    ;is current word what we are
                            searching for?
        BEQ     DONE        ;if YES, then get out of here
        CMPA.L  A0,A1       ;if NO, then have we searched all
                            required memory?
        BCC     LOOP        ; this is a CARRY CLEAR
                            instruction and is equivalent to ≤
                            comparison since there will be
                            no carry (actually borrow in this
                            case) if A0 ≤ A1
DONE    SUBA.L  #2,A0       ;adjust A0 to correct for the post
                            increment in the CMP instruction.
                            Note that since it was
                            incremented by a word we must
                            subtract 1 word (2 bytes).
        END     prog
```

EXAMPLE: SEQUENTIAL SEARCH PROGRAM

This program implements a sequential search program defined as:

> Given an N-element list of 16-bit numbers and a KEY, store the KEY in the N+1-st element of the list. Execute a sequential search of the list for KEY. KEY will always be found. If the address of the matching location is NOT the N+1-st element's address, the KEY was in the list. Otherwise, it is not present.

The program uses:

N	the number of elements in the list to search
KEY	the 16-bit number to search for
LIST	set of 16-bit numbers to search

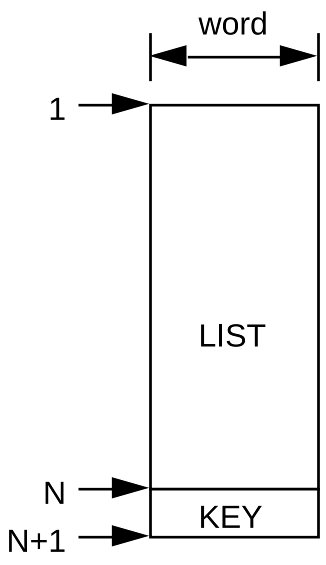
The program outputs one of the following:

> <value of KEY> is in the list.
> <value of KEY> is NOT in the list.

The program uses the DBEQ instruction to implement the search loop.

Functional specification (pseudocode)

```
input (N);
input (KEY);
LIST(N+1)=KEY;

FOR j=0 to N+1
    IF LIST(j) = KEY THEN KEYADDR=j;
IF KEYADDR≠N+1 THEN
    output(KEY in list.")
    ELSE
    output(KEY NOT in list.");
```


word
1
LIST
N
KEY
N+1

MC68000 assembly code for key search program:

```
            ORG       $5000
LIST:       DS.W      20                 ;reserve space for 20 words
FNDMSG      DC.B      ‘IS IN THE LIST’,0
NOTMSG      DC.B      ‘IS NOT IN THE LIST’,0
NEWLINE     DC.B      $0A,0              ;new line command message
            include   io.s               ;enter i/o declarations
START       JSR       HexIn              ;enter N into D0
            MOVE.W    D0,D1              ;store N in D1 for DB instructions
            SUBQ.W    #1,D1              ;correct N for DB instruction
            MOVE.W    D0,D2              ;save original N in D2
            LEA       LIST,A0            ;put starting LIST address into A0
LOAD        JSR       HexIn              ;enter entire LIST from keyboard
            MOVE.W    D0,(A0)            ;put in LIST
            ADDA      #2,A0              ;increment LIST address
            DBRA      D1,LOAD            ;decrement and repeat until done
            JSR       HexIn              ;get KEY
            LEA       LIST,A0            ;reset starting address
            LEA       LIST,A1            ;set working address
            ASL.W     #1,D2              ;double D2 for byte count since
                                         words
            ADDA      D2,A1              ;set A1 to end of LIST
            MOVE.W    D0,(A1)            ;put KEY at end of LIST
COMPARE     CMP.W     (A0),D0            ;LIST(j) = KEY?
            BEQ.S     OUTPUT             ;if yes then stop
            ADDA      #2,A0              ;if no then increment by one word
            BRA       COMPARE            ;and repeat
OUTPUT      MOVE.L    A0,D0
            JSR       HexOut             ;print address of where key was
                                         found
            CMPA.L    A0,A1              ;was KEY found in LIST? Is A0
                                         equal to end of LIST?
            BEQ.S     NOTFND             ;if not equal then KEY was not in
                                         LIST
```

```
          LEA      FNDMSG,A0    ;load starting address of
                                message for KEY found
          BRA      PRINTIT
NOTFND    LEA      NOTMSG,A0    ;load starting address of
                                message for KEY not found
PRINTIT   JSR      PrintString
          LEA      NEWLINE,A0   ;load starting address of new line
                                command
          JSR      PrintString
          END      START
```

Comments on use of DBcc instruction in this program:

```
          MOVE.W   D2,D1        put N-1 into D1 for loop count
          SUBQ.W   #1,D1

COMPARE   CMP.W    (A0)+,D0     compare (A0) with KEY
          DBEQ     D1,COMPARE   if they are equal then fall through
                                else goto compare.
```

Multiply unsigned

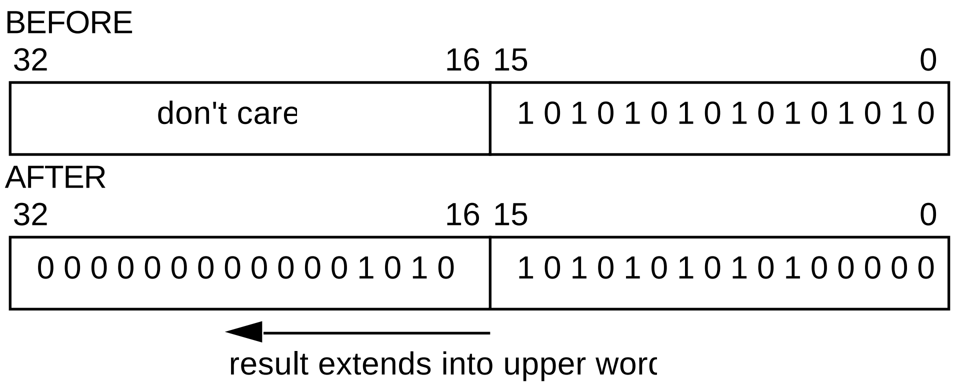
MULU<ea>,Dn

Action Multiplies the word length <ea> times the least significant word in Dn. The result is a long word.

Notes:
1. The lowest word of Dn contains the multiplier.
2. The result is a 32-bit long word.
3. The negative (N) and zero (Z) flags are set according to the result. The overflow (V) and carry (C) bits are always cleared since there can never be an overflow or carry with this instruction.

Example:

```
MULU#$10,D4
```



Multiply signed

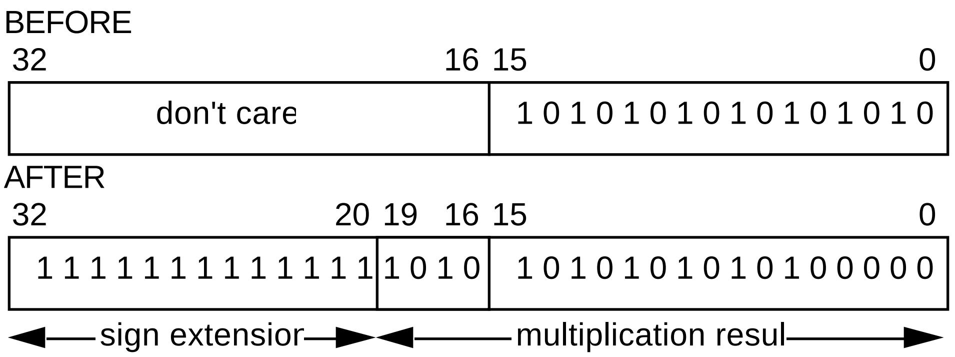
MULS<ea>,Dn

Action Multiplies the word length <ea> times the least significant word in Dn. The result is a sign extended long word.

Notes:
1. The lowest word of Dn contains the multiplier.
2. The result is a 32-bit long word which takes account of the multiplier and multiplicand's signs.
3. The negative (N) and zero (Z) flags are set according to the result. The overflow (V) and carry (C) bits are always cleared since there can never be an overflow or carry with this instruction.

Example:

MULS#$10,D4



Divide unsigned
DIVU <ea>,Dn

Action Divides the 32-bit integer in Dn by the 16-bit integer in <ea>. The most significant word of the 32-bit result is the remainder; the least significant word is the quotient.

32 16	15 0
remainder	quotient

Notes:
1. There is also a DIVS but you will need to sign extend what's in Dn before you can divide with sign. This can be done using the instruction EXT.L, which extends the lowest word to a long word, for signed numbers.
2. You may use the instruction ANDI.L #$0000FFFF,Dn to clear bits 16-32 for unsigned number division.

Example:

DIVU #10,D4

BEFORE (D4 contains 32_{10}) Note that 32_{10} = \$20 = %100000

32 16	15 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0

AFTER (the result is a quotient of 3_{10} with a remainder of 2_{10})

32 16	15 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1
remainder = 2	quotient = 3

Example:
SUppose you want to do a signed divide of -32 in D4 by 10, i.e.

```
        DIVS  #10,D4
```

Consider what happens if you put -32 in D4 using a MOVE immediate

```
        MOVE.W      #-32,D4
```

32 16	15 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0

The -32 is sign-extended to a word.

You must extend this to a long word before you can do a DIVS

```
        EXT.L D4
```

32 16	15 0
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1	1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0

Now you can correctly use the DIVS

```
        DIVS  #10,D4
```

to get the resulting quotient of -3_{10} with a remainder of -2_{10},
i.e. (D4) = $FFFE FFFD

32 16	15 0
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0	1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1
remainder = -2	quotient = -3

MATH INSTRUCTIONS

Instruction	Comments
ADDI	Add a constant, cannot be used with An as a destination.
ADDQ	Adds an immediate constant between 1 and 8.
SUB SUBI SUBQ	flags in normal way
SUBX	Clears Z only if the result is non-zero, i.e. it sets Z to 1 if the result is zero else Z remains unchanged. This instruction subtracts the source and the X bit from the destination.
ADDX	basically the same as SUBX but adds.
SUBA	Doesn’t effect status register.
NEG	Negates (subtracts from zero). WARNING: NEG is NOT a COMPLEMENT. It computes 0 - (Destination) - (X-bit)
NEGX	Adds X bit to destination, then subtracts from zero.
MULS MULU	Multiply two words.
DIVS DIVU	Divides a long word by a word. WARNING: Division by zero generates a TRAP.
EXT	Sign extend

EXAMPLE: DOUBLE PRECISION ADDITION

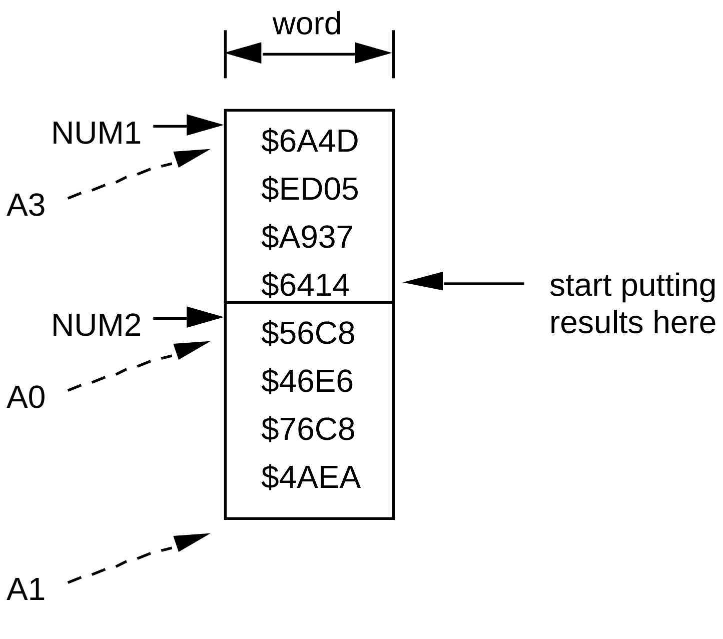
This program adds two 64-bit (8-byte) numbers together.

The program uses:

NUM1	64-bit number, 8 consecutive bytes
NUM2	64-bit number, 8 consecutive bytes stored immediately higher in memory than NUM1

Functional specification (pseudocode)

```
A3 = starting address of NUM1
A0 = A3 + 8                  ;starting address of NUM2
A1 = A0 + 8                  ;ending address of NUM2 plus 1
                             byte
X = 0                        ;clear X-bit

;loop starting with least significant bytes

FOR j = 1,4 DO
NUM1(j) = NUM1(j) + NUM2(j) + X;
```



MC68000 assembly code for double precision add program:

```
            ORG         $5000
NUM1        DC.W        $6A4D, $ED05,$A937,$6414
NUM2        DC.W        $56C8, $46E6,$76C8,$4AEA

BYTECNT     EQU         8                   ;number of bytes to add together
MAIN        LEA         NUM1,A3             ;use A3 for address first number
            LEA         BYTECNT(A3),A0      ;the second number begins 8
                                            bytes beyond the beginning of
                                            the first number - use address
                                            register indirect with
                                            displacement
            LEA         BYTECNT(A0),A1      ;address beyond end of second
                                            number
            MOVEQ       #0,CCR              ;clear the X bit of the SR
            MOVEQ       #BYTECNT-1,D2       ;set up loop counter, adjusted for
                                            DBRA.  MOVEQ is ok since
                                            counter is 7
LOOP        MOVE.B      -(A0),D0
            MOVE.B      -(A1),D1
            ADDX.B      D1,D0               ;D0=D0+D1+X-bit
            MOVE.B      D0,(A0)             ;save result in NUM1
            DBF         D2,LOOP             ;repeat 7 times
            END         MAIN
```

EXAMPLE: BINARY MULTIPLICATION

This program multiplies two 8-bit unsigned numbers using a shift and add algorithm to generate a 16-bit product.

The multiplier is in D2 and the multiplicand in D1.
The product is returned in D1.

algorithm:

1. Starting with most significant bit of multiplier, i.e. bit=8
2. Shift product to line up properly (product = 2*product)
3. If multiplier[bit] = 1 then product=product+multiplier
4. Decrement bit. If bit≥0 then goto 2.

The program uses:

MULTIPLICAND	8-bit number to be multiplied
MULTIPLIER	8-bit number that MULTIPLICAND is multiplied by
PRODUCT	32-bit result

Functional specification (pseudocode)

```
PRODUCT = 0;                          /*clear PRODUCT*/
BIT=8 /* starting at MSB */


FOR j = 1,8 DO                        /*do for each bit of MULTIPLIER*/
    BEGIN
    PRODUCT = PRODUCT*2;              /*shift PRODUCT left by one bit*/
    IF MULTIPLIER[9-bit] = 1 THEN
        PRODUCT = PRODUCT + MULTIPLICAND;
/* do calculations from most significant bit to least significant bit */

    BIT=BIT-1;                        /* decrement bit */
    END
```

DETAILED EXAMPLE:

multiplier = 61_{16} (97_{10})
multiplicant = $6F_{16}$ (111_{10})

multiplier:	(D2)	=	00000000 01100001	($00 61)
multiplicand	(D1)	=	00000000 01101111	($00 6F)
initial product:	(D0)	=	00000000 00000000	($00 00)
shift product:	(D0)	=	00000000 00000000	($00 00)
MUL[8] = 0 don’t add				
new product	(D0)	=	00000000 00000000	($00 00)
shift product:	(D0)	=	00000000 00000000	($00 00)
MUL[7] = 1 so add	(D1)	=	$\underline{00000000\ 01101111}$	($00 6F)
new product	(D0)	=	00000000 01101111	($00 6F)
shift product:	(D0)	=	00000000 11011110	($00 DE)
MUL[6] = 1 so add	(D1)	=	$\underline{00000000\ 01101111}$	($00 6F)
new product	(D0)	=	00000001 01001101	($01 4D)
shift product:	(D0)	=	00000010 10011010	($02 9A)
MUL[5] = 0 don’t add				
new product	(D0)	=	00000010 10011010	($02 9A)
shift product:	(D0)	=	00000101 00110100	($05 34)
MUL[4] = 0 don’t add				
new product	(D0)	=	00000101 00110100	($05 34)
shift product:	(D0)	=	00001010 01101000	($0A 68)
MUL[3] = 0 don’t add				
new product	(D0)	=	00001010 01101000	($0A 68)
shift product:	(D0)	=	00010100 11010000	($14 D0)
MUL[2] = 0 don’t add				
new product	(D0)	=	00010100 11010000	($14 D0)
shift product:	(D0)	=	00101001 10100000	($29 A0)
MUL[1] = 1 so add	(D1)	=	$\underline{00000000\ 01101111}$	($00 6F)
new product	(D0)	=	00101010 00001111	($2A 0F)

final answer: (D0) = 00101010 00001111 ($2A 0F)

where \$2A0F = $10767_{10} = 97_{10} \times 111_{10}$

MC68000 assembly code for binary multiply program:

```
            ORG        $5000
A           DC.W       $61
B           DC.W       $62
RESULT      DS.L       1

MAIN        CLR.L      D0               ;clear 32-bit product register
            MOVE.L     D0,D1            ;clear upper word for ADD.L
            MOVE.W     A,D1             ;copy multiplicand into D1
            MOVE.W     B,D2             ;copy multiplier into D2
            MOVE.W     #16-1,D3         ;loop count = 16-1 for DBRA
                                        instruction
LOOP        ADD.L      D0,D0            ;shift product left one bit
            ADD.W      D2,D2            ;shift multiplier left one bit
            BCC.S      STEP             ; Use carry to check whether to
                                        add.  If carry=0 goto next step.
            ADD.L      D1,D0            ;if multiplier [15] was one then
                                        add multiplicand.
STEP        DBRA       D3,LOOP          ;else continue with loop
            LEA        RESULT,A1        ;get where to put answer
            MOVE.L     D0,(A1)          ;store result
            END        MAIN
```

NOTES:

1. Program uses shift and add algorithm.
2. DBRA is equivalent to DBF and works in most assemblers.

REVIEW for Integer arithmetic functions

```
ADD.<size>  <source>,<destination>
```

One operand MUST be a data register; affects all five status codes in CCR

Overflow (V)

Set if two like-signed numbers (both positive or both negative) are added and the has a different sign. This is causes by the result exceeding the 2's complement range of numbers, causing the sign bit to change. Mathematically, $V = C_s \quad C_p$

The V and N flags are pertinent only for signed numbers but are set for all additions.

```
ADDA        <ea>,An
```

If the destination is an address, the condition codes are not changed.

For adding multiple words, the extend can be used.

ADDX

Adds two (data) registers or memory locations. However, zero is only cleared if the result is non-zero; otherwise, zero is unchanged.

```
ADD.L       D0,D2
ADDX.L      D1,D3
```

The above code adds the double precision numbers together:

X	
D1	D0
D3	D2

Memory to memory adds do not change X, Z. You must set them. For example:

```
MOVE        #4,CCR      ;sets Z bit, clears all others
```