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 = 00000000															

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
0	1	0	0	1	1	1	0	1	1	<effective address>					
										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

15	14	13	12	11 10 9 8	7 6 5 4 3 2 1 0
0	1	1	0	4-bit condition code	8-bit (or 16-bit) 2's complement offset

◄—opcode—►

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

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. (false / true)	<set CCR bits> B$\overline{cc}$ NEXT Code (True) NEXT: (false)

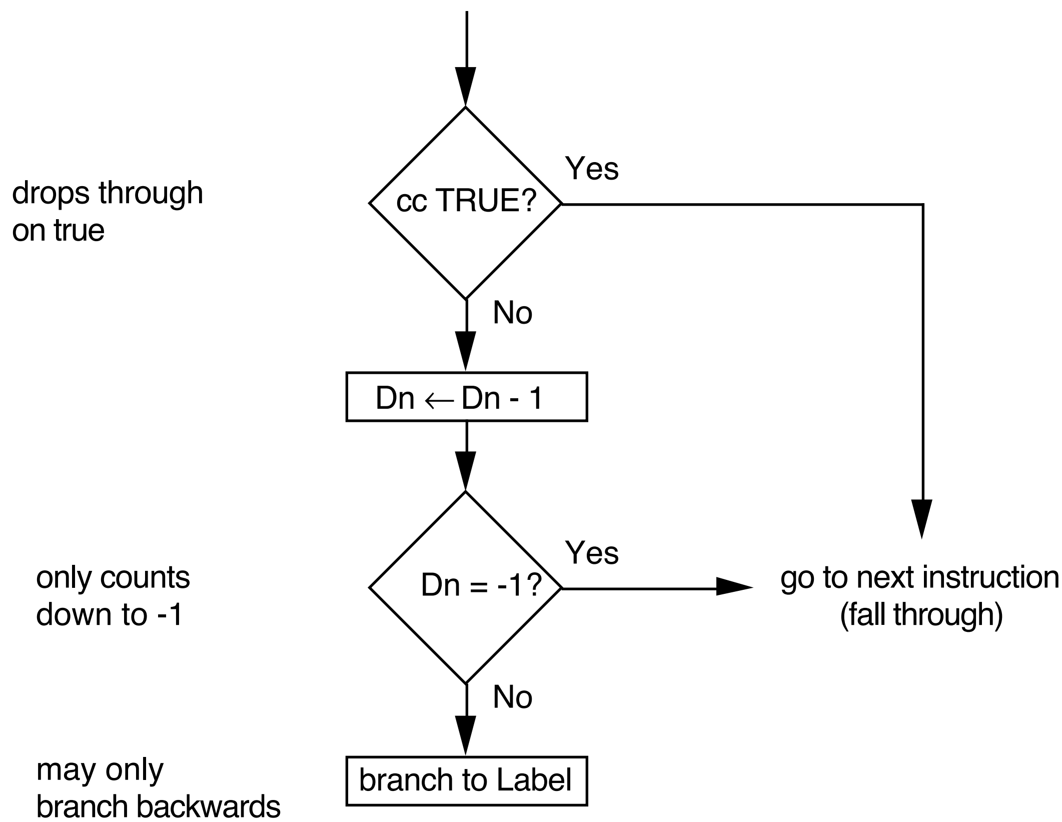
IF <boolean expression> THEN
code(True)
ELSE
code(False)
Next statement.
(false / true)
	<set CCR bits>
	B$\overline{cc}$ ELSE
	Code (True)
	BRA NEXT
	ELSE: Code (False)
	NEXT:

WHILE <boolean expression> DO code. Modify expression. Return to WHILE ... DO Next statement. (false / true)	<set CCR bits> WHILE B$\overline{cc}$ NEXT Code. Modify condition. BRA WHILE NEXT: (false / true)

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 ....
⇔	BNE.S NEXT
DBNE D0,LOOP	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.

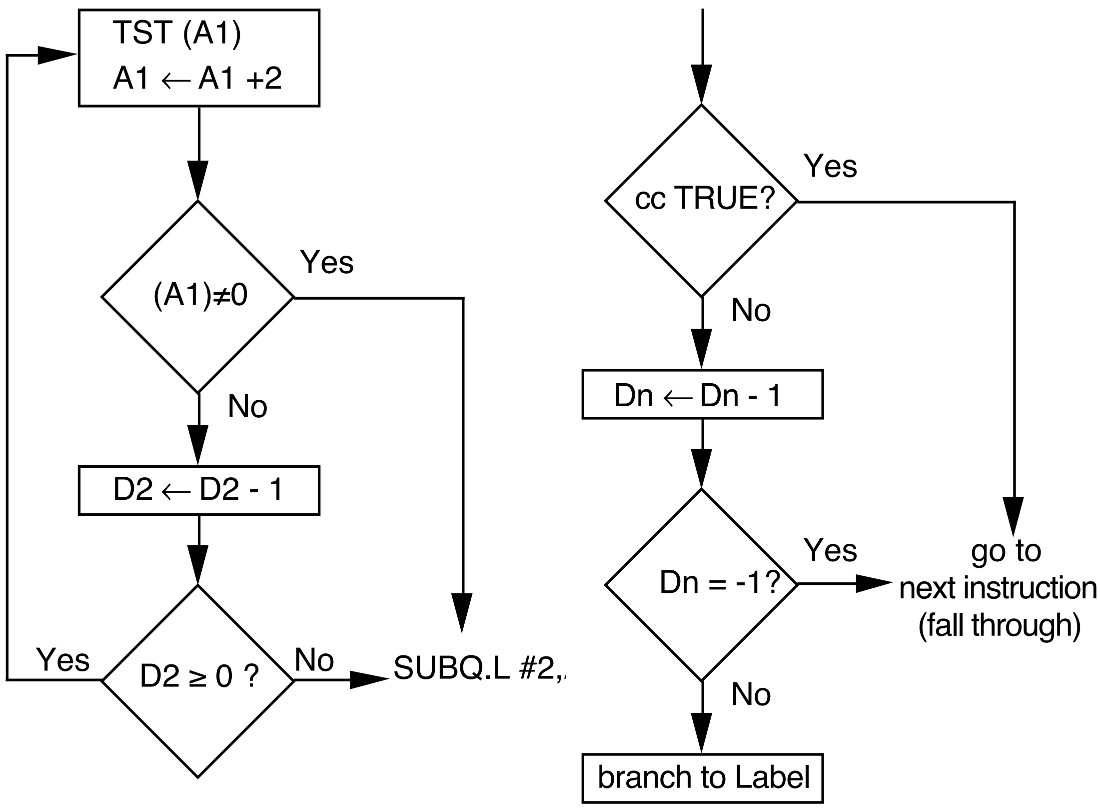
The instruction DBRA is equivalent to DBF.

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)

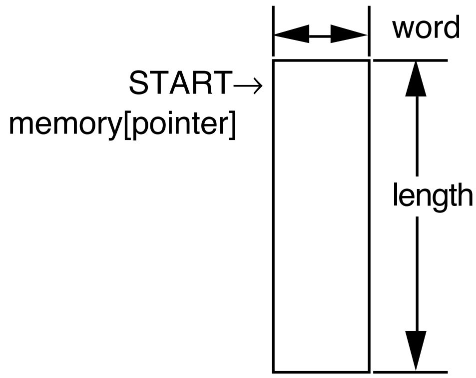
```
                                        ;define inputs
    START=location of words in memory
    LENGTH=# of words to examine
    TOTAL                               ;where to put answer

    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:
```


word
START→
memory[pointer]
length

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.

<u>Bcc instructions appropriate for signed numbers</u>
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)

<u>Bcc instructions appropriate for unsigned numbers</u>
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

```
0000
FFFF
----
FFFF
```

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

The SUBQ gets executed 11 times.