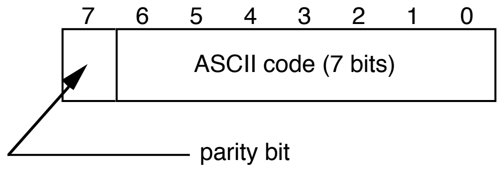
Review of ASCII character representation:



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.

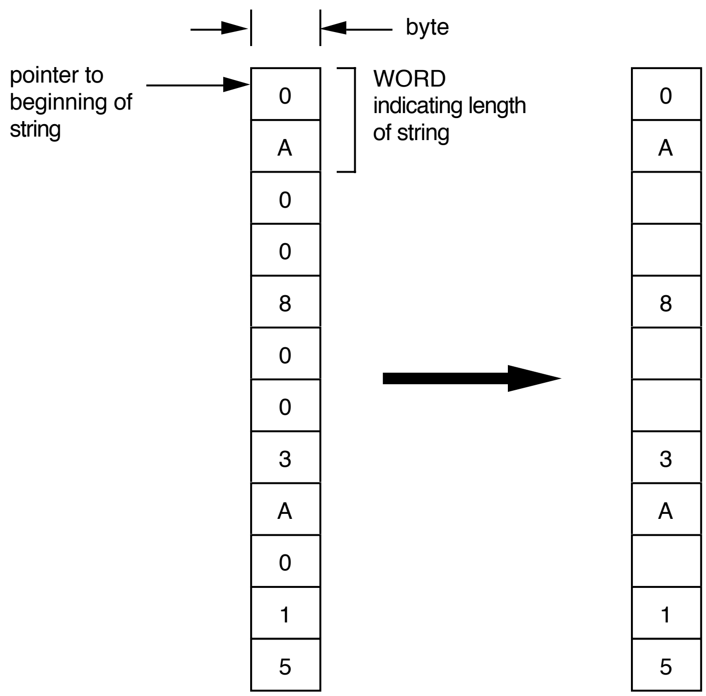
0	0	0	0	0	1	1	0

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.



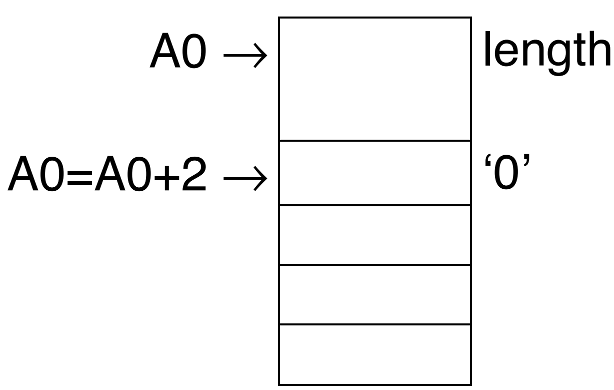
Functional specification (pseudocode)

```
                                    ;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} = $0000\ 0000\ 0000\ 0011_2$

(D0) = $xxxx\ xxxx\ xxxx\ xxxx_2$

(D0) = $0000\ 0000\ 0000\ 00xx_2$

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