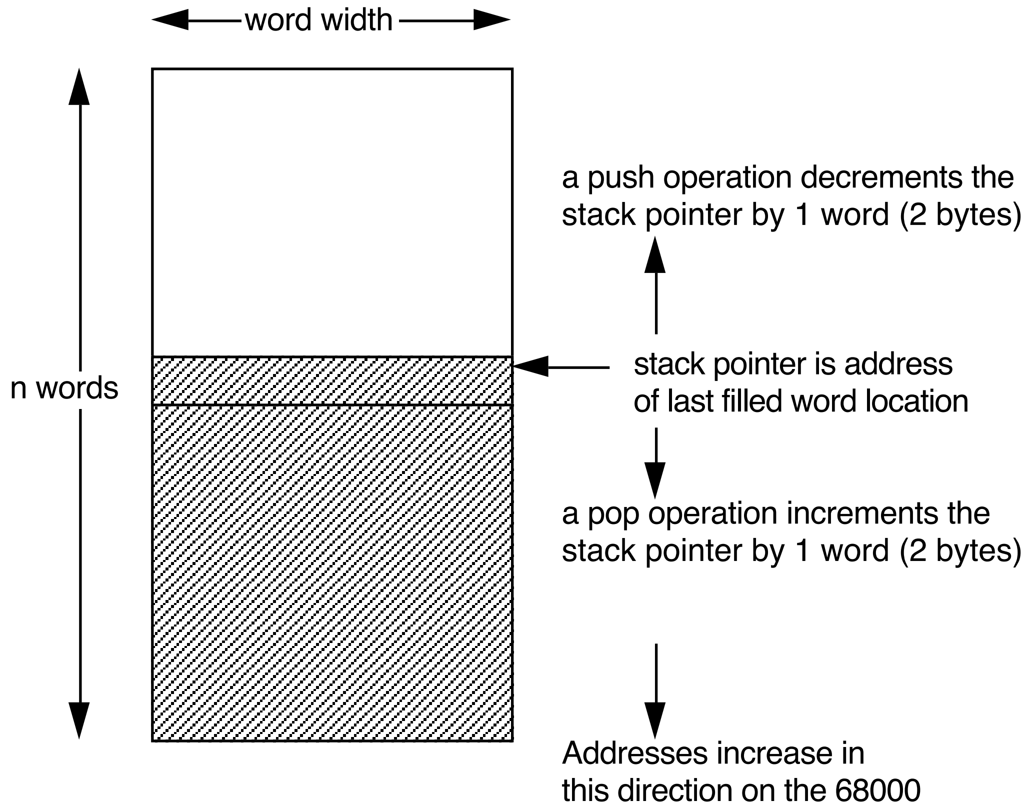
STACK

A stack is a first in, last out buffer usually implemented as a block of n consecutive bytes (it doesn't have to be bytes—it could be words or long words). In the example below, the stack is composed of words.



NOTES ABOUT 68000 STACKS

On the 68000 stack addresses begin in high memory ($60000 for example) and are pushed toward low memory ($50000 for example). Other machines might do this in the reverse order.

A stack can be implemented as bytes or longwords. The normal 68000 stack pointer is in A7 (Don't use this register for anything else!!!). If you want to use a special stack which is byte or long word in width you will need to use another register; A7 is only for word width stacks.

USES FOR STACKS

- data storage — This application is similar to an array, but is more useful for handling input/output information.

- program tracking & control — The stack is usually used to pass variables to and from subroutines and for storage of local variables.

ALLOCATING THE STACK IS THE PROGRAMMER'S RESPONSIBILITY!

This means that the programmer is responsible for reserving memory for stack operations and for properly initializing the value of the stack pointer at the top of the stack memory area.

For example, the following code will allocate memory for a stack of 200 words

```
                DS.W      $200
BOTTOM          EQU       *
```

To initialize the stack pointer, put the high memory address of the stack into A7

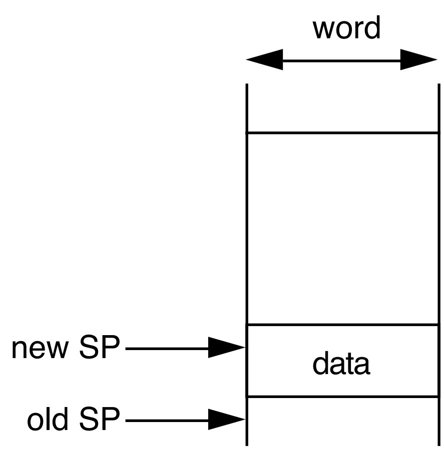
```
                MOVE      #BOTTOM,A7
```

To “push” something onto the stack, the stack pointer must be decremented by one word and then <source> can be put on the stack.

```
MOVE      <source>,-(SP)
```

To “pop” something off the stack, the information must be fetched from the stack, the stack pointer incremented by 1 word, and the information put into <destination>.

```
MOVE   (SP)+,<destination>
```



The stack is usually put just ahead of the program in embedded microprocessor systems. This is not true for personal computers such as the Macintosh. They put the stack in very high memory (just under the heap) and put program information in low memory. For example, the program would begin just after the memory reserved for the stack in an embedded system.

```
                DS.W      $200
BOTTOM          EQU       *
                <program code begins here>
```

A major problem with stacks is that the programmer makes them too small. The word size of a stack is a measure of the greatest number of data items that might be put into it.

stack overflow	attempt to push below the bottom end of the stack
stack underflow	attempt to pop an item from an empty stack

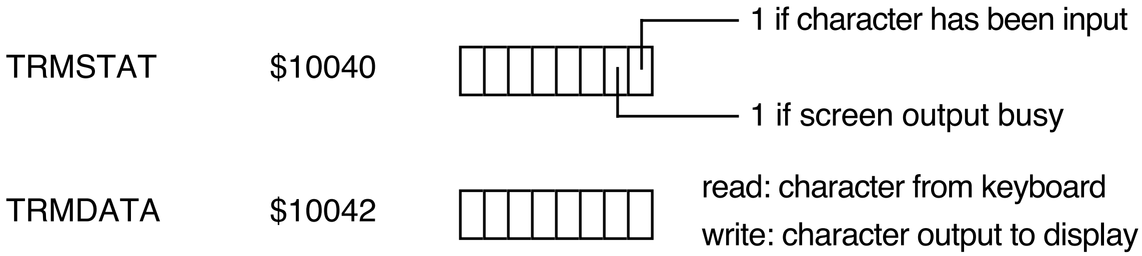
EXAMPLE: BACKWARD ECHO PROGRAM

This program will accept a character string terminated by a carriage return-line feed (CR-LF), place it into a stack buffer (temporary storage area), and output the string in reverse order to a computer terminal.

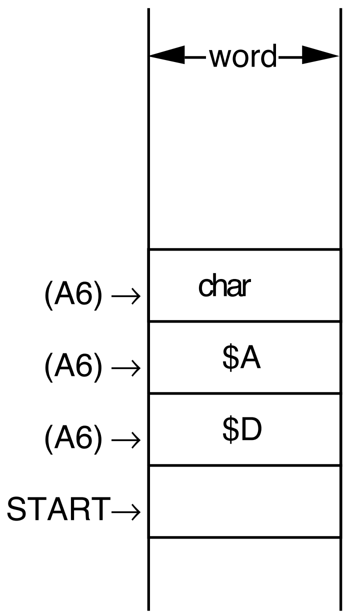
Functional specification (pseudocode)

```
initialize stack
push CR onto stack; push LF onto stack

inloop
if (TRMSTAT[0] ≠ 1) then goto inloop            ;wait for input from
                                                ;keyboard - this is polled
                                                i/o
get next char
if (char = CR) goto outloop                     ;CR denotes end of input
push char onto stack
goto inloop

outloop
if (TRMSTAT[1] = 1) then goto outloop           ;wait for busy display
pop char from stack
output char                                     ;ideal application for
                                                CharOut
if (SP less than initial SP) then goto outloop  ;anything left in stack?
```

TRMSTAT and TRMDATA are special memory locations which are connected to the hardware of a computer terminal. Bit 0 of TRMSTAT whether a character has been input from the keyboard: 1 indicates a character has been input and can be found in TRMDATA, 0 indicates that nothing has been input since the last read of TRMDATA. Bit 1 of TRMSTAT indicates whether the terminal display is busy outputting the character last placed into TRMDATA. A 1 indicates that the terminal is still busy and is not ready for the next character to be output. TRMDATA is used for input and output of ASCII data. When read, TRMDATA indicates input from the keyboard whereas a write to TRMDATA will send the character to the display.

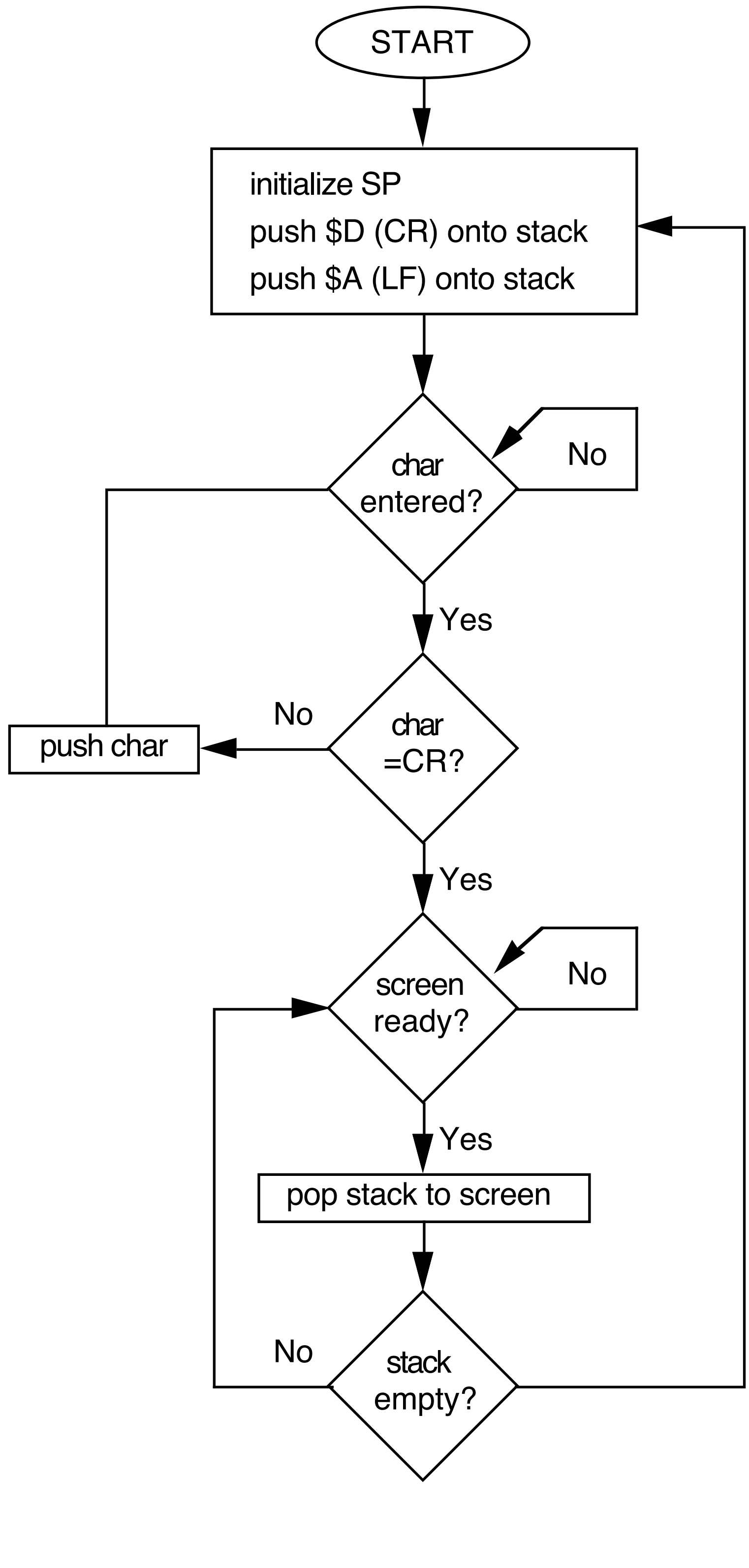

TRMSTAT
$10040
1 if character has been input
1 if screen output busy
TRMDATA
$10042
read: character from keyboard
write: character output to display

This is a stack for my data so I will use A6 NOT A7 for the stack pointer.



Note that the stack builds down in memory.

Program accepts input:
AB...YZ<cr>
then outputs
ZY...BA<lf><cr>



MC68000 CODE

```
        INCLUDE  io.s          ;include io definitions
TRMSTAT EQU      $10040        ;terminal status register
TRMDATA EQU      $10042        ;terminal data register
        ORG      $4000         ;start program here
        DS.W     200           ;save 200 words for a stack
START   EQU      *             ;assign an address to START
        LEA      START,A6      ;initialize SP to START address
        CLR.L    D0
        MOVE     #$D,-(A6)     ;push CR onto stack
        MOVE     #$A,-(A6)     ;push LF onto stack
LOOP    EQU      *
        BTST     #0,TRMSTAT    ;character entered?
                               ;bit[0]=1 when character waiting
        BEQ      LOOP          ;no input, keep waiting
        MOVE.B   TRMDATA,D0    ;have input, get char entered
        CMP      #$D,D0        ;is char entered a CR?
        BEQ      OUT           ;YES, goto to output routine
        MOVE     D0,-(A6)      ;NO, push char onto stack
        BRA      LOOP          ;and repeat input loop

OUT     EQU      *
        MOVE     (A6)+,D0      ;pop char from stack
        JSR      CharOut       ;output character
        CMPA     START,A6      ;is stack empty?
        BNE      OUT           ;NO, keep outputting chars
        BRA      START         ;YES, get new line
        END      START
```

NOTE: CMPA is a new instruction.

EXAMPLE: RPN CALCULATOR (problem 6.3)

This program implements a reverse Polisn (RPN) calculator using a stack.

Examples of input:
11* equals 1 AND 1
10+ equals 1 OR 0

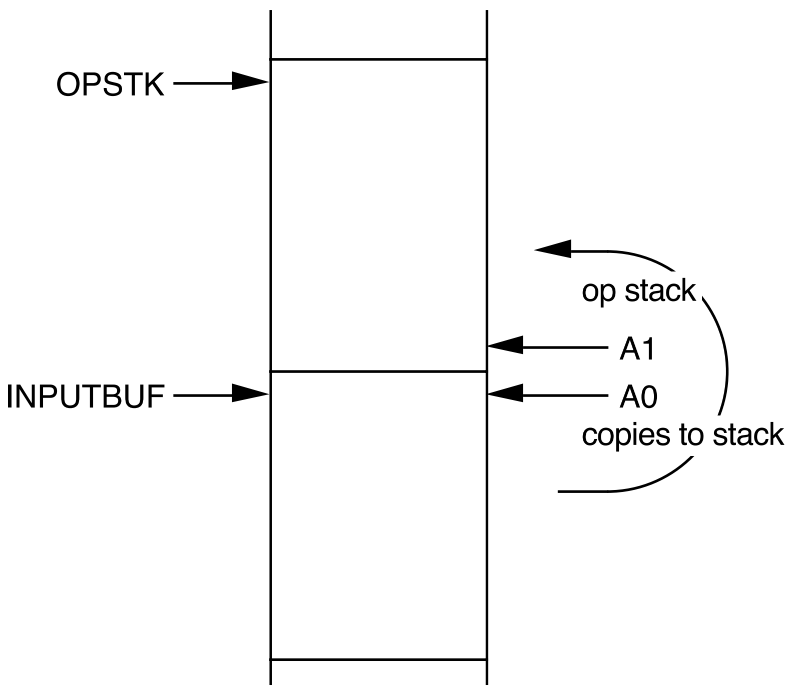
The operands ‘0’ and ‘1’ have ASCII values $30 and $31 respectively. Convert ASCII to binary by subtracting ‘0’, i.e. ASCII $30 from the ASCII value. Reverse the process for input.

The program uses:

MULTIPLICAND 8-bit number to be multiplied

Functional specification (pseudocode)

PRODUCT = 0; /*clear PRODUCT*/



MC68000 assembly code for RPN calculator program:

```
            ORG      $5000
BUFSIZ      EQU      80              ;input buffer size
OPSTK       DS.B     20              ;size of operations stack
INPUTBUF    DS.B     BUFSIZ
START       LEA      INPUTBUF,A0     ;load address of input buffer into
                                     A0
            MOVE.W   #BUFSIZ,D0      ;set D0 to size of input buffer
; (A0) = address of input, (D0.W) = max number of characters to read
; on input (D0.W) is # of characters to input
            JSR      STRIN           ;get input
            JSR      STROUT          ;echo input
            SUBQ     #2,D0           ;adjust character count for DB
                                     instruction
            LEA      INPUTBUF,A1     ;set A1 to top of stack
SCANNEXT    CMPI.B   #'0',(A0)       ;input='0'?
            BLT.S    EVALUATE        ;if input<0 then input is operator
            MOVE.B   (A0)+,-(A1)     ;push input onto stack
            SUBI.B   #'0',(A1)       ;convert stack entry to binary
            BRA.S    CHKCNT          ;test for more input
EVALUATE    MOVE.B   (A1)+,D2        ;pop the operand stack
            MOVE.B   (A1)+,D1        ;
            CMPI.B   #'*',(A0)+      ;is operand an '*'?
            BEQ      ANDOP           ;Yes it is - goto AND operand
            OR.B     D1,D2           ;otherwise OR arguements
            BRA.S    PUSHOP
ANDOP       AND.B    D1,D2           ;AND arguements
PUSHOP      MOVE.B   D2,-(A1)        ;push result onto stack
CHKCNT      DBF      D0,SCANNEXT
PUTANS      ADDI.B   #'0',(A1)       ;convert stack to ASCII
            MOVEA.L  A1,A0           ;set up pointer to output, i.e. A0
            MOVE.W   #1,D0           ;set up # of characters to output,
                                     i.e. D0.W
            JSR      STROUT
```

```
JSR        NEWLINE
```

PC RELATIVE ADDRESSING MODES

Bcc DBcc	Both of these branches use relative addressing allowing a program to work anywhere in memory independent of absolute addresses.

program counter with displacement

d(PC) d is a 16-bit 2's complement displacement (-32K to + 32K bytes) which is sign extended

program counter with index and displacement

d(PC, Ri.W) d(PC, Ri.L)	Ri can be wither an address or data register. The register is sign extended if <size> is .W. Note that the displacement is -128 to +127 bytes.

Consider the instruction

MOVE.W $500(PC),D4

This is a two word instruction. Assume that (PC) = $1000 at start of instruction.

1. fetch first instruction word
2. increment PC, PC=PC+2
3. decode instruction
4. then add $500 to $1502
5. (PC)=$1004 at end of instruction

PEA implements call by reference parameter passing

PEA <ea> pushes an address onto stack

Equivalent to the instruction

MOVE.L <ea>,-(SP)

CMPM compare memory

CMPM.<size> (Ay)+,(Ax)+
Both source and destination MUST be in post increment mode.

RTR returen and restore instruction
Word is popped from the stack and the least significant byte (LSB) of this word is put into the CCR. Long word is popped from the stack and placed into the PC.

Should execute
MOVE.W CCR,-(SP)
at beginning of program

Problem: How to save registers (subroutine needs to use registers also)

Solution: Push all registers onto stack after JSR
Pop all registers off stack before RTS

MOVEM.<size> <register list>,<ea>
MOVEM.<size> <ea>,<register list>

Push registers onto stack.
MOVEM.<size> <register list>,-(SP)
Pop registers off stack.
MOVEM.<size> (SP)+,<register list>

Register list (no commas)
D0,D2,D3,D4,A0,A1,A6
is equivalent to
D0/D2-D4/A0-A1/A6

where you use the ‘/’ instead of a comman to seperate registers and ‘-’ indicates a range of registers, i.e. D2-D4 indicates all data registers from D2 to D4.

<size> = .W or .L
When <size>=.W all registers are sign extended first.