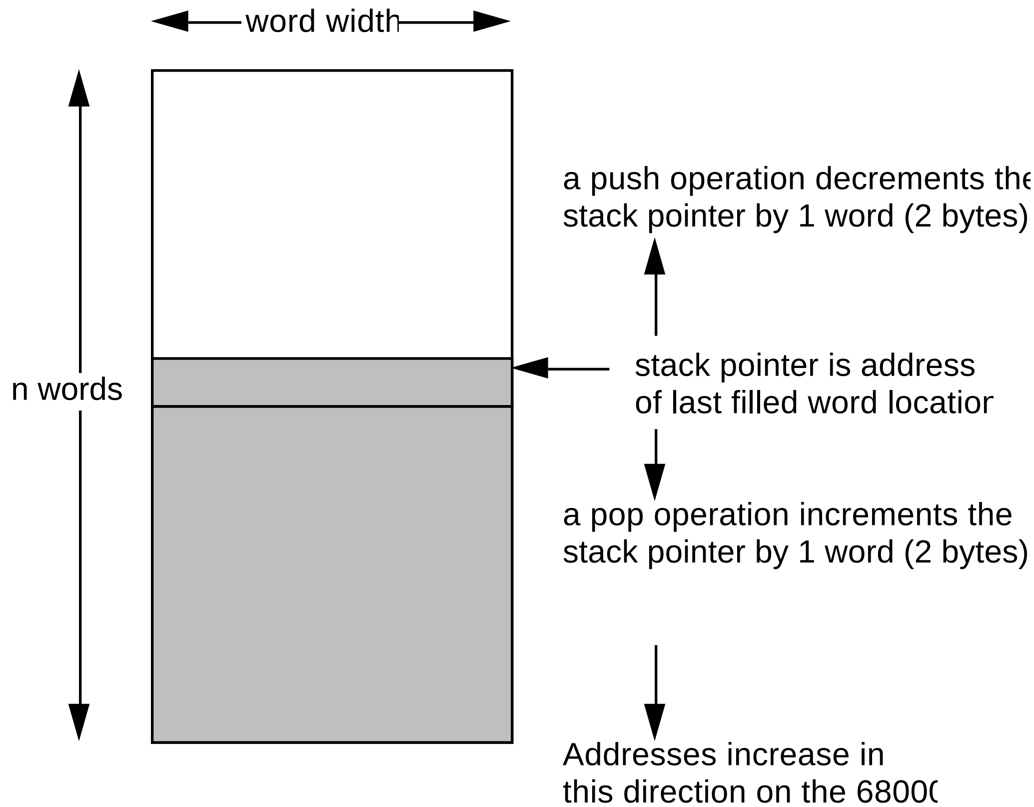
STACK

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

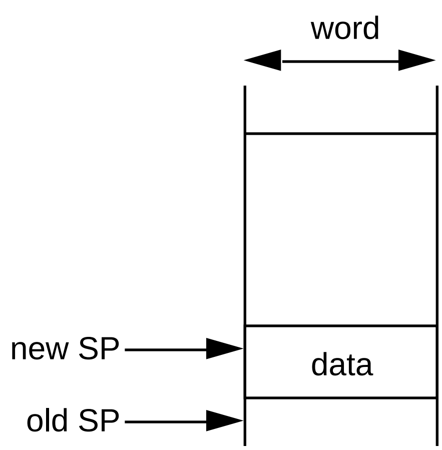
```
            MOVE      #BOTTOM,A7
```

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

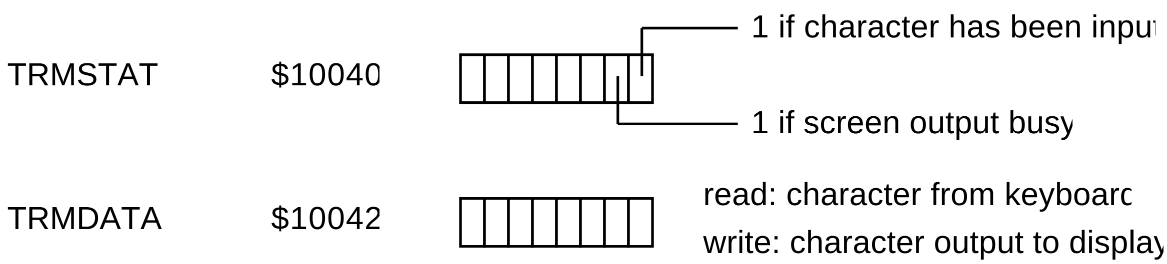
EXAMPLE: BACKWARD ECHO PROGRAM

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.

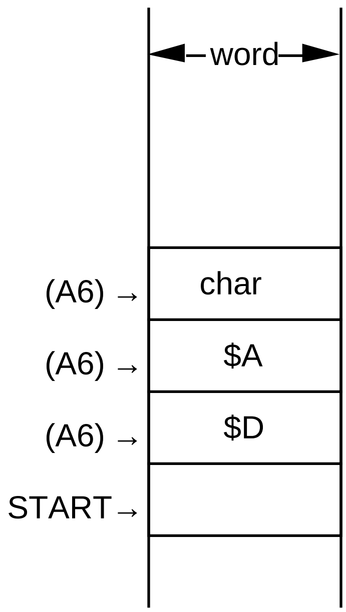
Functional specification (pseudocode)

```
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.

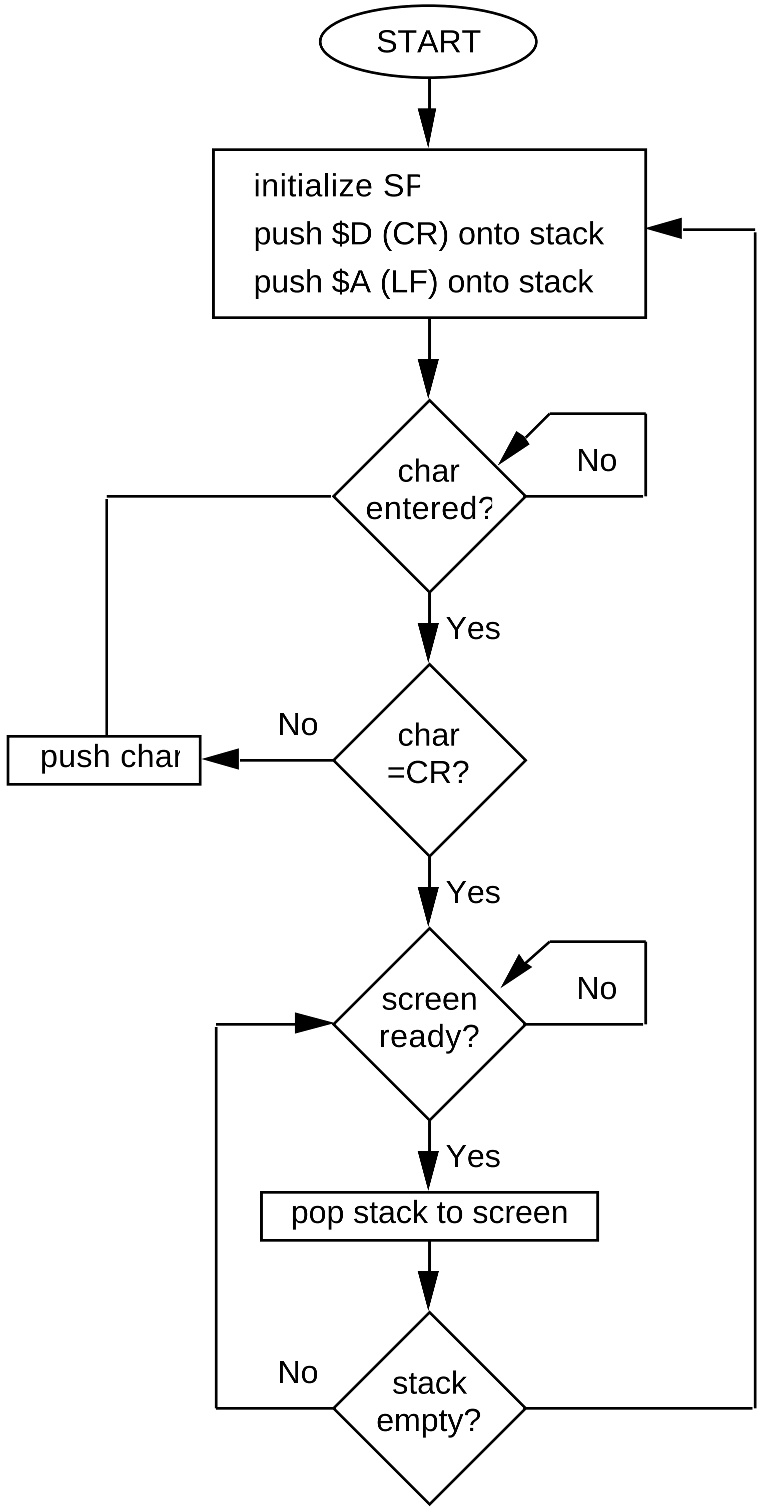


This is a stack for my data so I will use A6 NOT A7 for the stack pointer.



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.

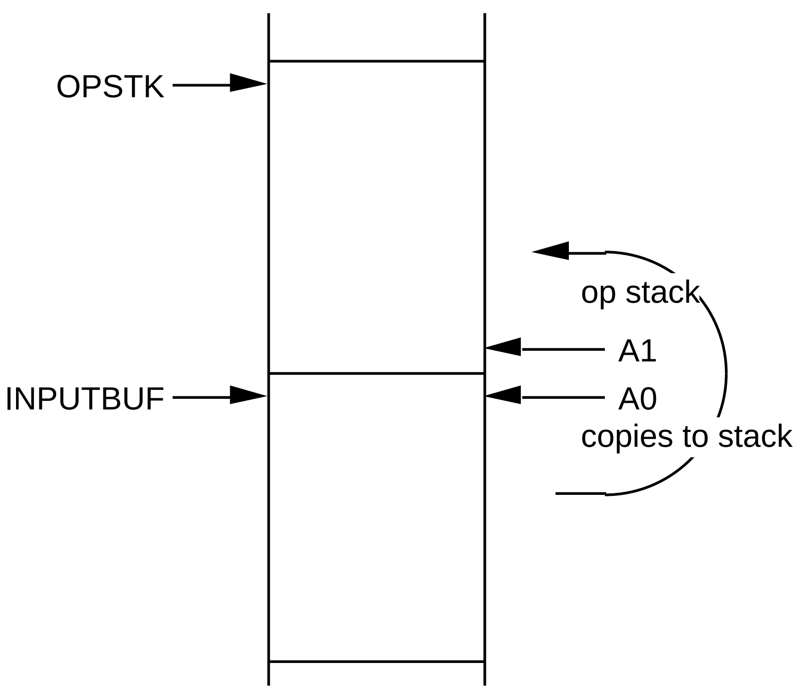
Examples of input:
11*   equals 1 AND 1
10+   equals 1 OR 0

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 return 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>

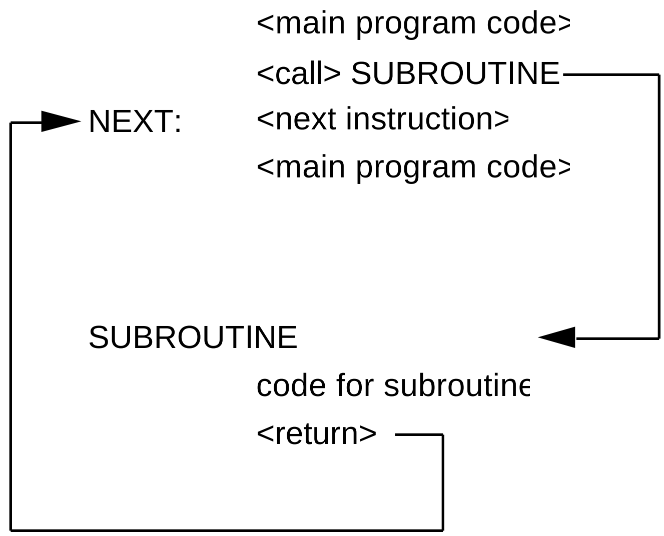
Push registers onto stack.
MOVEM.<size> <register list>,-(SP)
Pop registers off stack.
MOVEM.<size> (SP)+,<register list>

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.

SUBROUTINES

General format of calling and returning from a subroutine



Problem: How do we know where to return to when the subroutine is completed?
Solution: store the address of the next instruction after the call (as well as the current value of the registers and any local variables) on a stack

PROGRAMMER IS RESPONSIBLE FOR SETTING THE STACK POINTER AND ALLOCATING MEMORY FOR THE STACK. THIS IS NORMALLY A7.

Examples of calling a subroutine:

BSR	<label>	where label MUST be a label with no more than a 16-bit signed offset, i.e. within ±64K of the BSR instruction
JSR	<ea>	where <ea> must be a memory addressing mode, i.e. <ea> cannot be a data or address register. This is the most common form of calling a subroutine.

Both forms put the address of the next instruction on the 68000 stack into A7, i.e. they push the long word address of the next instruction after the call onto the stack.

Examples of returning from a subroutine:

RTS	pops a long word, an address, off the stack (in A7) and and loads the PC with that address.

<u>WARNING</u> If the stack pointer is not pointing to the correct return address you will not return to the next instruction after the subroutine call.

WHY USE A SUBROUTINE

- If you use the same code at different points in your program, the use of a subroutine will result in a savings of program memory.
- Use of subroutines results in modular program design which is easier to comprehend, debug, etc.

ISSUES IN WRITING SUBROUTINES

linkage	this is the address at which the program resumes after executing the subroutine
argument transmission	how do you supply the subroutine with values for its arguments
coding	subroutines should always be written as pure procedures with no self-modifying code

Linkage:

Both of the following instructions

```
JSR     SUB     ;jumps to a subroutine anywhere in memory
BSR     SUB     ;jumps to a subroutine within a limited addressing range
```

are equivalent to the instruction sequence

```
MOVE.L  address of next instruction,-(SP)
JMP     SUB
```

which pushes the return address onto the stack and jumps to the subroutine code. SP is a mnemonic for the stack pointer and means the same as A7 on the 68000.

The following instruction

```
RTS             ;return from subroutine
```

is equivalent to the instruction

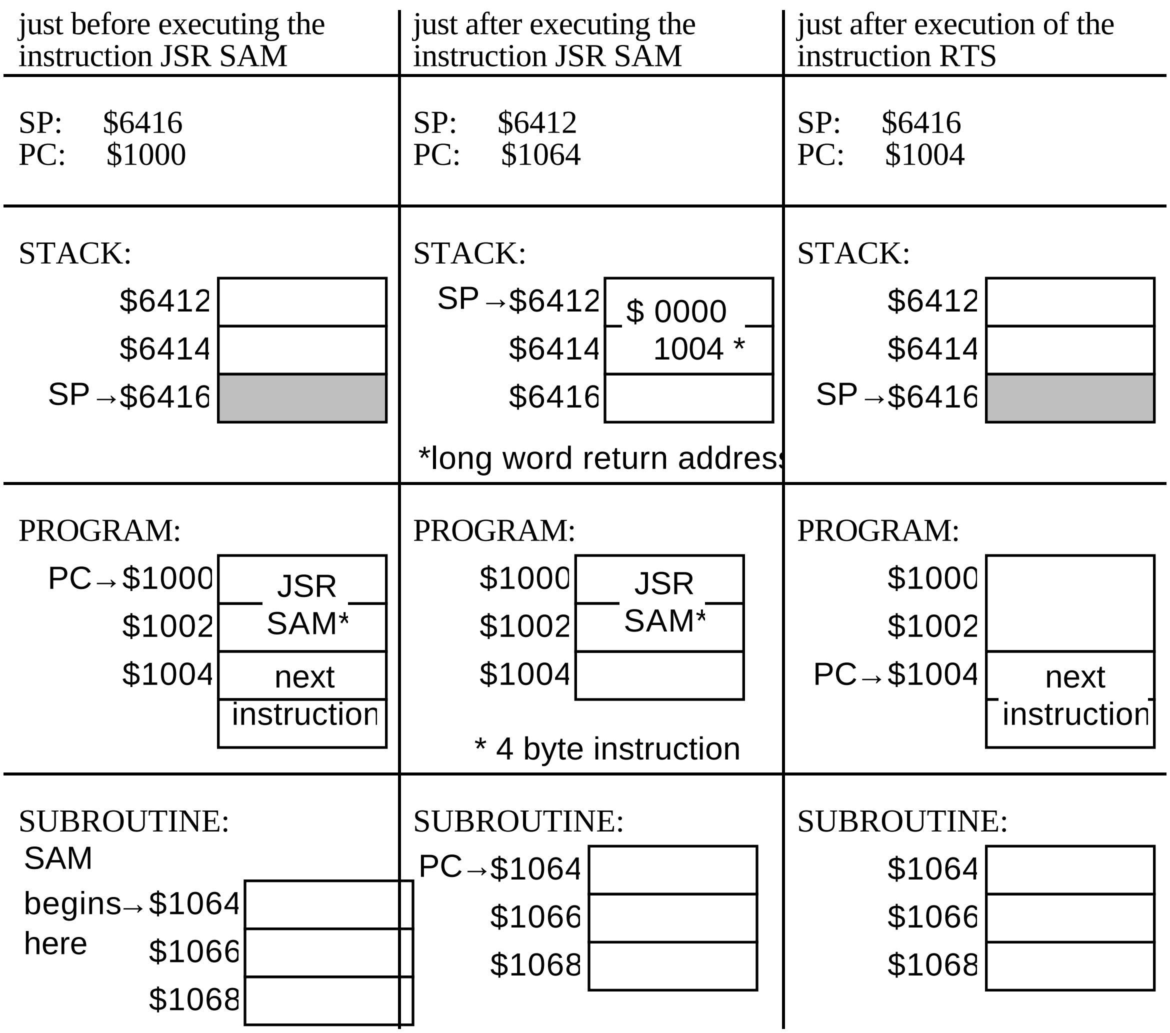
```
JMP     (SP)+   ;does not affect condition codes of SR
```

which jumps to the next instruction after the JSR (assuming the SP is correctly placed) and pops the return address off the stack.

EXAMPLE:

```
            ORG         $1000           ;beginning of CODE section
            JSR         SAM             ;jump to subroutine SAM
            <next instruction>
            <rest of program>

SAM         <subroutine code>           ;keep for comparison
            RTS
```

Example of the above subroutine call sequence:
NOTE: There is NO saving of any register contents, the SR, or any local variables.



HOW TO PASS PARAMETERS TO SUBROUTINES

• using registers	data registers—call by value (uses actual data values) put arguments in data registers before JSR
• using registers	address registers—call by reference (uses actual data values) put the addresses of the arguments in address registers before JSR
• in-line coding	• put arguments immediately after JSR, address of arguments passed via return address on stack • put addresses of arguments immediately after JSR, address of arguments passed via return address on stack • arguments listed in a table or array, pase base address of table to subroutine via an address register
• using the stack	(this is the preferred method) Optionally use LINK and UNLK instruction to create and destroy temporary storage on stack.

The MOVEM instruction
This instruction saves or restores multiple registers. If you have a small assembly language program this instruction allows you to save to values of registers NOT used to pass parameters.

MOVEM has two forms:

```
MOVEM      register_list,<ea>
MOVEM      <ea>,register_list
```

Example:

```
SUBRTN     EQU        *
           MOVEM      D0-D7/A0-A6,SAVEBLOCK
           • • •
           MOVEM      SAVEBLOCK,D0-D7/A0-A6
           RTS
```

where SAVEBLOCK is local memory. This is bad practice in general since SAVEBLOCK could be overwritten.

Example:

```
SUBRTN     EQU        *
           MOVEM      D0-D7/A0-A6,-(SP)
           • • •
           MOVEM      (SP)+,D0-D7/A0-A6
           RTS
```

This is the most common method of using the MOVEM instruction to save registers on the stack and restore them when the subroutine is done. This is especially useful for re-entrant and/or recursive subroutines. A recursive procedure is one that may call or use itself. A re-entrant procedure is one that is usable by interrupt and non-interrupt driven programs without loss of data.

The MOVEM instruction always transfers contents to and from memory in a predetermined sequence, regardless of the order in which they are listed in the instruction.

address register indirect with pre-decrement

transferred in order $A7 \rightarrow A0, D7 \rightarrow D0$

for all control modes and address register indirect with post-increment

transferred in order $D0 \rightarrow D7, A0 \rightarrow A7$

This allows you to easily build stacks and lists.

POWR subroutine

This subroutine accepts two input parameters, a base and an exponent, and calculates the function $base^{exponent}$.

Functional specification (pseudocode)

```
POWR (base, exponent)

D1=base                              ;input arguments
D2=exponent                          ;exponent must be an integer

initialize D3 to 1                   ;
exponent=exponent-1
while exponent≥0 D3=base*D3          ;compute using continued
                                     ;product of base
end POWR.
```

Basic documentation of POWR (see p.3 of lab manual)

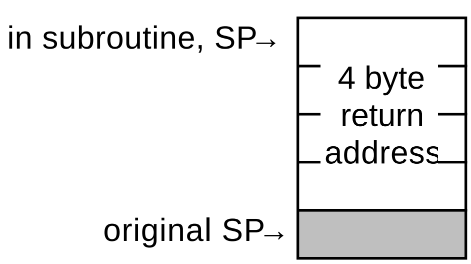
Subroutine documentation:

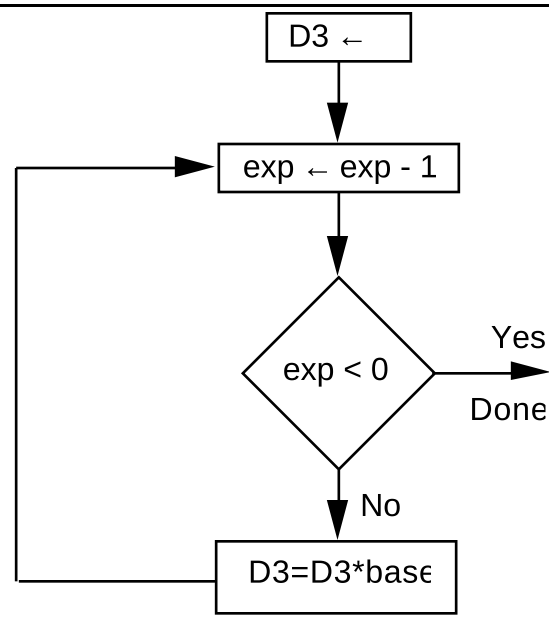
name:	POWR
function:	computers $base^{exponent}$ where exponent is an interger using continued product
input/output:	input: D1=base, D2=exponent output: D3=result
registers destructively addressed:	D2,D3
memory requirements:	none
subroutines called:	none
length of subroutine (bytes):	40

POWR (parameter passing using data registers)

```
            ;Program to compute the power of a number using subroutine.
            ;Power MUST be an integer. A and B are signed numbers.
            ;Parameter passing via data registers.

            MOVE      A,D1        ;put base into D1
            MOVE      B,D2        ;put exponent into D2
            JSR       POWR        ;call subroutine POWR
            LEA       C,A5        ;put address of where to put answer
                                  into A5
            MOVE      D3,(A5)     ;save answer

DATA        EQU       *
A           DC.W      4
B           DC.W      2
C           DS.W      1

POWR        MOVE.L    #1,D3       ;put starting 1 into D3
LOOP        EQU       *
            SUBQ      #1,D2       ;decrement power
            BMI       EXIT        ;if D2<0 then quit subroutine
            MULS      D1,D3       ;multiply out
            BRA       LOOP        ;and repeat as necessary
EXIT        EQU       *
            RTS
```

Behavior of the stack	Basic flow chart of POWR
Note that the initial value of the stack pointer must be set by the programmer even if we don't explicitly use it for anything. The 68000 MUST have a value for A7 when subroutines are used. In this case, RTS uses it to return to the LEA instruction.	





POWR (parameter passing using address registers)

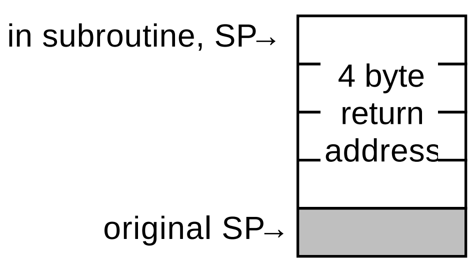
```
            ;Program to compute the power of a number using subroutine.
```

```
        ;Power MUST be an integer. A and B are signed numbers.
        ;Parameter passing via address registers.

            LEA      A,A1        ;put address of base into A1
            LEA      B,A2        ;put address of exponent into A2
            JSR      POWR        ;call subroutine POWR
            LEA      C,A5        ;put address of where to put answer
                                 into A5
            MOVE     D3,(A5)     ;save answer

DATA        EQU      *
A           DC.W     4
B           DC.W     2
C           DS.W     1

POWR        EQU      *
* only difference is that following instructions are address register indirect
            MOVE     (A1),D1     ;get base
            MOVE     (A2),D2     ;get exponent
            MOVE.L   #1,D3       ;put starting 1 into D3
LOOP        EQU      *
            SUBQ     #1,D2       ;decrement power
            BMI      EXIT        ;if D2<0 then quit subroutine
            MULS     D1,D3       ;multiply out
            BRA      LOOP        ;and repeat as necessary
EXIT        EQU      *
            RTS
```

POWR (parameter passing using inline coding of data)

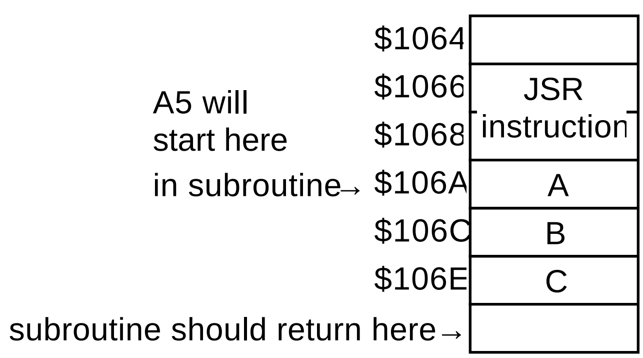
```
        ;Program to compute the power of a number using subroutine.
        ;Power MUST be an integer. A and B are signed numbers.
        ;Parameter passing via inline coding of data.

* no longer load parameters into registers BEFORE subroutine call
        JSR         POWR        ;call subroutine POWR
* parameters are inline AFTER subroutine call
DATA    EQU         *
A       DC.W        4           ;base
B       DC.W        2           ;exponent
C       DS.W        1           ;result

* the rest of the program would go here

POWR    EQU         *
        MOVE.L      (SP),A5     ;put return address into A5
        MOVE        (A5)+,D1    ;get A,  increment  A5 to point to B
        MOVE        (A5)+,D2    ;get B,  increment  A5 to point to
                                where to put result
        MOVE.L      #1,D3       ;put starting 1 into D3
LOOP    EQU         *
        SUBQ        #1,D2       ;decrement power
        BMI         EXIT        ;if D2-1<0 then quit subroutine
        MULS        D1,D3       ;multiply out
        BRA         LOOP        ;and repeat as necessary
EXIT    EQU         *
        MOVE        D3,(A5)+    ;(C)=answer,
                                ;(A5)=return address
        MOVE.L      A5,(SP)     ;put correct return address on stack
        RTS
```

Behavior of the stack



Return address on stack is address of A, NOT the next program instruction which would be several bytes beyond this.

How program memory is arranged

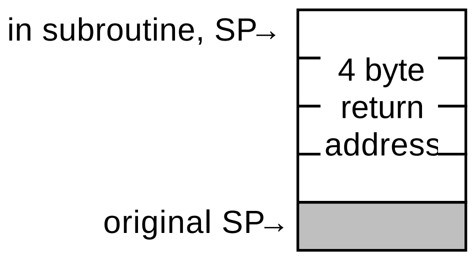


POWR (parameter passing using inline coding of addresses)

```
        ;Program to compute the power of a number using subroutine.
        ;Power MUST be an integer. A and B are signed numbers.
        ;Parameter passing via inline coding of addresses.
```

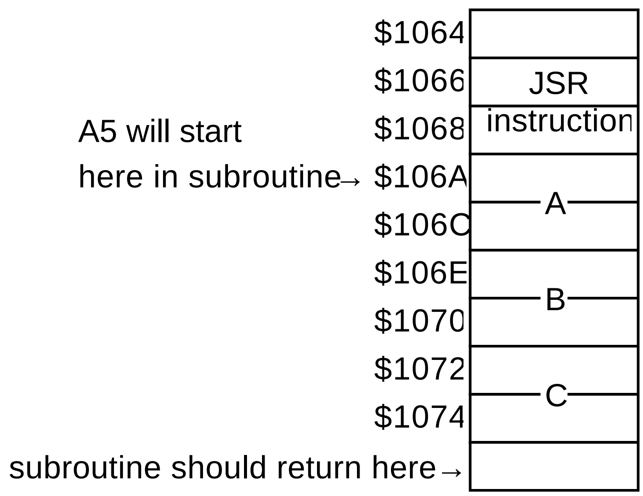
```
        JSR     POWR    ;call subroutine POWR
* addresses of parameters are put inline AFTER subroutine call
        DC.L    A,B,C   ;address of A,B and C are inline
* the rest of the program would go here

DATA    EQU     *
A       DC.W    4       ;base
B       DC.W    2       ;exponent
C       DS.W    1       ;result


POWR    EQU     *
        MOVE.L  (SP),A5     ;put return address into A5
        MOVE    (A5)+,A1    ;get address of A, increment A5 so
                            (A5)=address of B
        MOVE    (A5)+,A2    ;get address of B, increment A5 so
                            (A5)=address of C
        MOVE    (A1),D1     ;put A into D1
        MOVE    (A2),D2     ;put B into D2

        MOVE.L  #1,D3       ;put starting 1 into D3
LOOP    EQU     *
        SUBQ    #1,D2       ;decrement power
        BMI     EXIT        ;if D2<0 then quit subroutine
        MULS    D1,D3       ;multiply out
        BRA     LOOP        ;and repeat as necessary
EXIT    EQU     *
        MOVE.L  (A5)+,A3    ;increment A5 to point to correct
                            return address, put address of C into
                            A3
        MOVE    D3,(A3)     ;put answer into C
        MOVE.L  A5,(SP)     ;restore correct return address onto
                            stack
        RTS
```

Behavior of the stack



Return address on stack is address of A, NOT the next program instruction which would be several bytes beyond this.

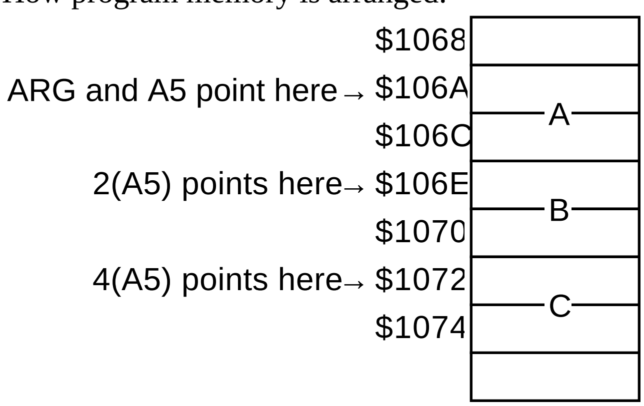
How program memory is arranged



POWR (parameter passing using the address of a parameter array in an address register)

```
        ;Program to compute the power of a number using subroutine.
        ;Power MUST be an integer. A and B are signed numbers.
        ;Parameter passing via  the address of a parameter array in an address register.

            LEA       ARG,A5          ;put address of argument array in A5
            JSR       POWR            ;call subroutine POWR
* the rest of the program would go here

ARG         EQU       *
A           DC.W      4               ;base
B           DC.W      2               ;exponent
C           DS.W      1               ;result


POWR        EQU       *
            MOVE      (A5),D1         ;put A into D1
            MOVE      2(A5),D2        ;put B into D2
* table means use address register indirect with displacement and/or offset

            MOVE.L    #1,D3           ;put starting 1 into D3
LOOP        EQU       *
            SUBQ      #1,D2           ;decrement power
            BMI       EXIT            ;if D2-1<0 then quit subroutine
            MULS      D1,D3           ;multiply out
            BRA       LOOP            ;and repeat as necessary
EXIT        EQU       *
            MOVE      D3,4(A5)        ;put answer from D3 into C
            RTS
```

How program memory is arranged:

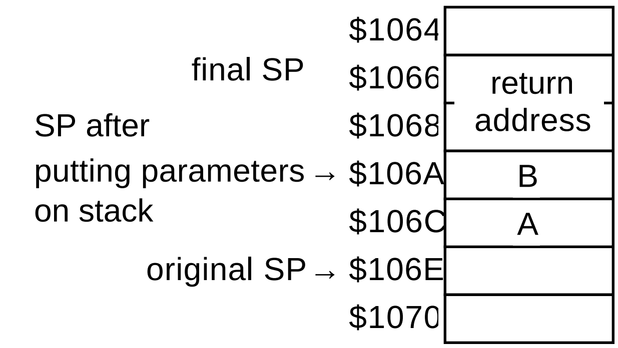


POWR (parameter passing by placing parameters on stack)

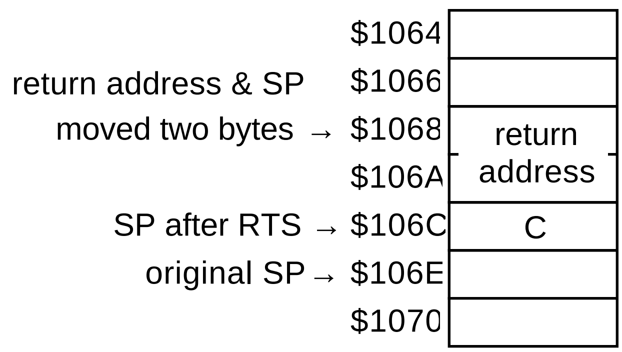
```
        ;Program to compute the power of a number using subroutine.
        ;Power MUST be an integer. A and B are signed numbers.
        ;Parameter are passed on the stack.

        MOVE.W  A,-(SP)         ;push A onto stack
        MOVE.W  B,-(SP)         ;push B onto stack
        JSR     POWR            ;call subroutine POWR
        MOVE.W  (SP)+,C         ;pop answer from stack resetting SP
                                to original value
* the rest of the program would go here

ARG     EQU     *
A       DC.W    4               ;base
B       DC.W    2               ;exponent
C       DS.W    1               ;result


POWR    EQU     *
        MOVE.W  6(SP),D1        ;put A into D1
        MOVE.W  4(SP),D2        ;put B into D2

        MOVE.L  #1,D3           ;put starting 1 into D3
LOOP    EQU     *
        SUBQ    #1,D2           ;decrement power
        BMI     EXIT            ;if D2-1<0 then quit subroutine
        MULS    D1,D3           ;multiply out
        BRA     LOOP            ;and repeat as necessary
EXIT    EQU     *
        MOVE.W  D3,6(SP)        ;put answer on stack on top of A
        MOVE.L  (SP),2(SP)      ;move return address two bytes up in
                                stack
        ADDQ.L  #2,SP           ;increment SP by 2 bytes
        RTS
```

How the stack is manipulated by this program:

The stack just after JSR has been executed



The stack just before the RTS is executed. Notice how the stack had to be corrected by two bytes to account for the fact that two parameters were passed to POWR but only one parameter was returned



Recursive subroutine

This subroutine accepts one input and computes the factorial of that number using recursive procedure calls on the stack.

Functional specification (pseudocode)

```
FACTOR(input)
factorial=input                     ;number input
push factorial on stack             ;save the current number on
                                    ;stack
factorial=factorial-1               ;decrement the number
if number≠1 call FACTOR             ;continue putting on stack?
    else {end FACTOR}               ;this ends up with factorial=1
temp=pop stack                      ;pop number from stack
factorial=factorial*temp            ;compute factorial
end FACTOR.
```

Basic documentation of FACTOR (see p.3 of lab manual)

Subroutine documentation:

name:	FACTOR
function:	computes the factorial of a given number
input/output:	input: D0.W output: D0.W
registers destructively addressed:	D0
memory requirements:	none
subroutines called:	none
length of subroutine (bytes):	40 (estimated)

FACTOR (parameter passing using data register D0)

```
        ;Program to compute the factorial of a number using subroutine.
        ;Parameter passing via data registers.

DATA     EQU      $6000          ;data segment
PROGRAM  EQU      $4000          ;program segment
         ORG      DATA
NUMB     DS.W     1              ;number to be factorialized
F_NUMB   DS.W     1              ;factorial of number
         ORG      PROGRAM
MAIN     MOVE.W   NUMB,D0        ;get number
         JSR      FACTOR         ;goto factorial routine
         MOVE.W   D0,F_NUMB      ;store result

* subroutine FACTOR (parameter passing using data register D0)
* Computes the factorial of a number.
*        Initial conditions:   D0.W=number to compute factorial of.
*                              0<D0.W<9
*        Final conditions:     D0.W=factorial of input number
```

```
*       Register usage:         D0.W destructively used
*       Sample case:    Input   D0.W=5
*                               Output D0.W=120

FACTOR      MOVE.W      D0,-(SP)    ;push input number onto stack
            SUBQ.W      #1,D0               ;decrement number
            BNE.S       F_CONT              ;reached 1 yet?
            MOVE.W      (SP)+,D0            ;yes, factorial=1
            RTS                             ;return
F_CONT      JSR         FACTOR              ;no, call FACTOR
            MULU        (SP)+,D0            ;multiply only after stack contains all
                                            numbers
RETURN      RTS
```

Stack usage by subroutine FACTOR