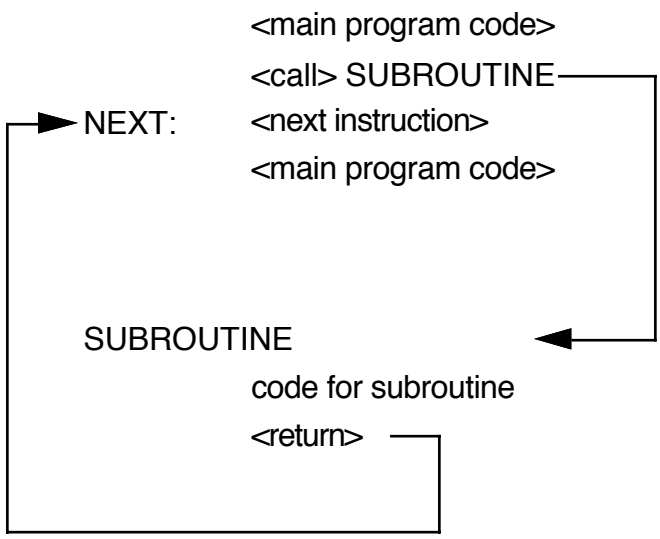
SUBROUTINES

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.

WARNING 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.

just before executing the instruction JSR SAM	just after executing the instruction JSR SAM	just after execution of the instruction RTS
SP: $6416 PC: $1000	SP: $6412 PC: $1064	SP: $6416 PC: $1004
STACK: $6412 $6414 SP→$6416	STACK: SP→$6412 $ 0000 $6414 1004 * $6416 *long word return address	STACK: $6412 $6414 SP→$6416
PROGRAM: PC→$1000 JSR $1002 SAM* $1004 next instruction	PROGRAM: $1000 JSR $1002 SAM* $1004 * 4 byte instruction	PROGRAM: $1000 $1002 PC→$1004 next instruction
SUBROUTINE: SAM begins here→$1064 $1066 $1068	SUBROUTINE: PC→$1064 $1066 $1068	SUBROUTINE: $1064 $1066 $1068

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→A0,D7→D0

for all control modes and address register indirect with post-increment
transferred in order D0→D7,A0→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:

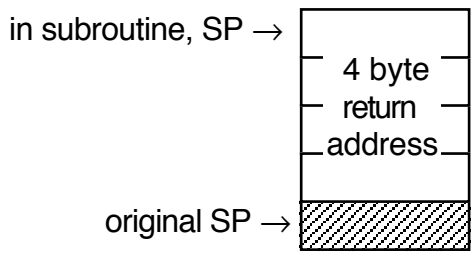
name:	POWR
function:	computers $base^{exponent}$ where exponent is an interger using continued product
input/output:	input: D1=base, D2=exponent output: D3=result
registers destructively addressed:	D2,D3
memory requirements:	none
subroutines called:	none
length of subroutine (bytes):	40

POWR (parameter passing using data registers)

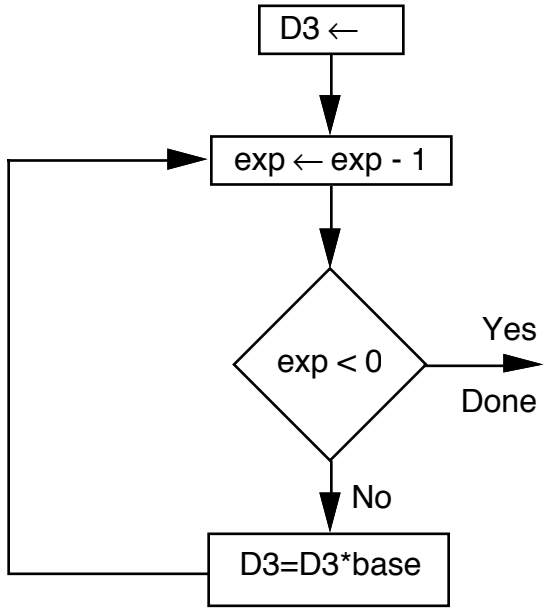
```
;Program to compute the power of a number using subroutine.
;Power MUST be an integer. A and B are signed numbers.
;Parameter passing via data registers.

            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

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.



Basic flow chart of POWR



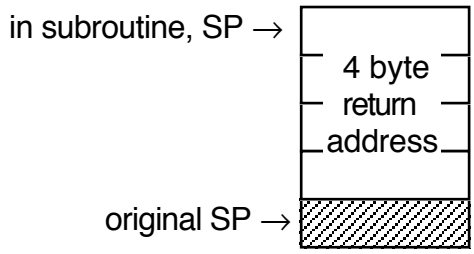
POWR (parameter passing using address registers)

```
        ;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
```

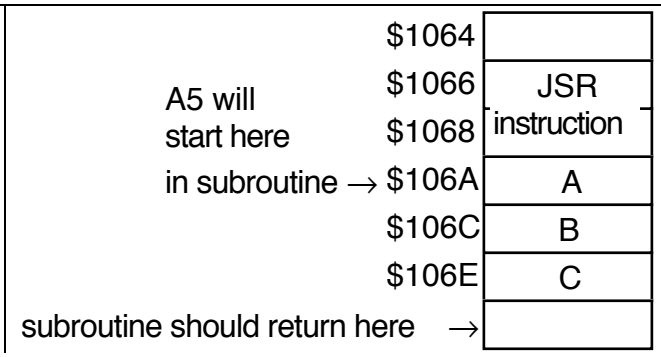
POWR (parameter passing using inline coding of data)

```
        ;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	How program memory is arranged



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

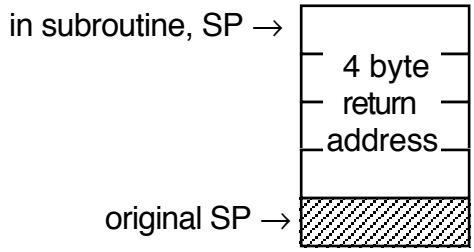


POWR (parameter passing using inline coding of addresses)

```
        ;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     *
```

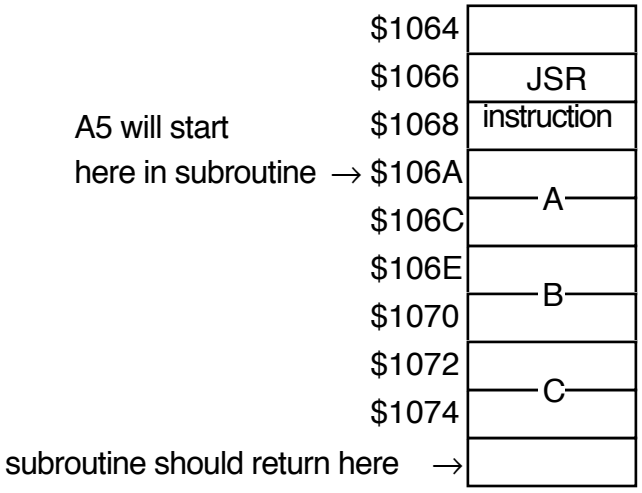
```
MOVE.L    (A5)+,A3      ;increment A5 to point to correct
                        return address, put address of C
                        into A3
MOVE      D3,(A3)       ;put answer into C
MOVE.L    A5,(SP)       ;restore correct return address
                        onto stack
RTS
```

Behavior of the stack



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

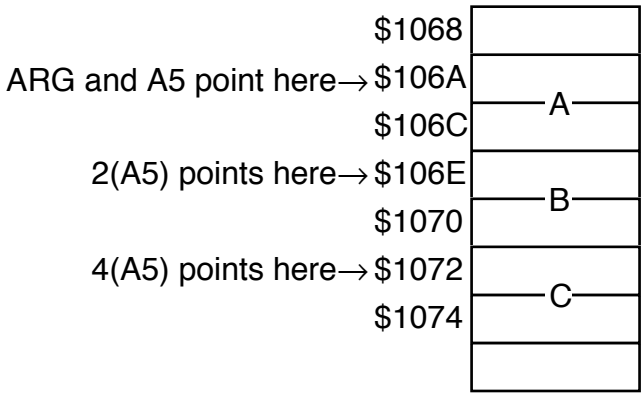
How program memory is arranged



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:

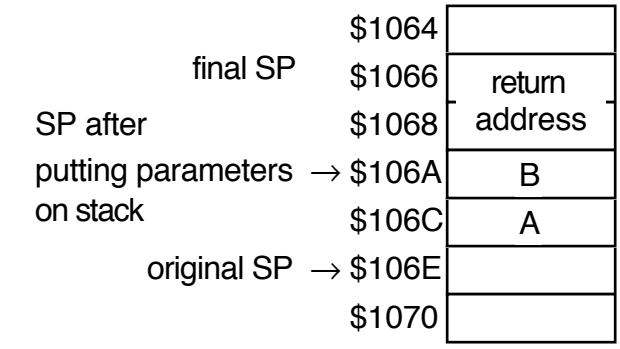


POWR (parameter passing by placing parameters on stack)

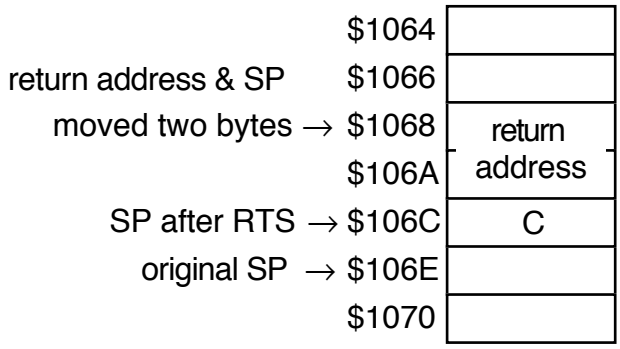
```
;Program to compute the power of a number using subroutine.
;Power MUST be an integer. A and B are signed numbers.
;Parameter are passed on the stack.

        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