SEE 3223 Microprocessors

8: Stacks & Subroutines

Muhammad Mun'im Ahmad Zabidi (munim@utm.my)

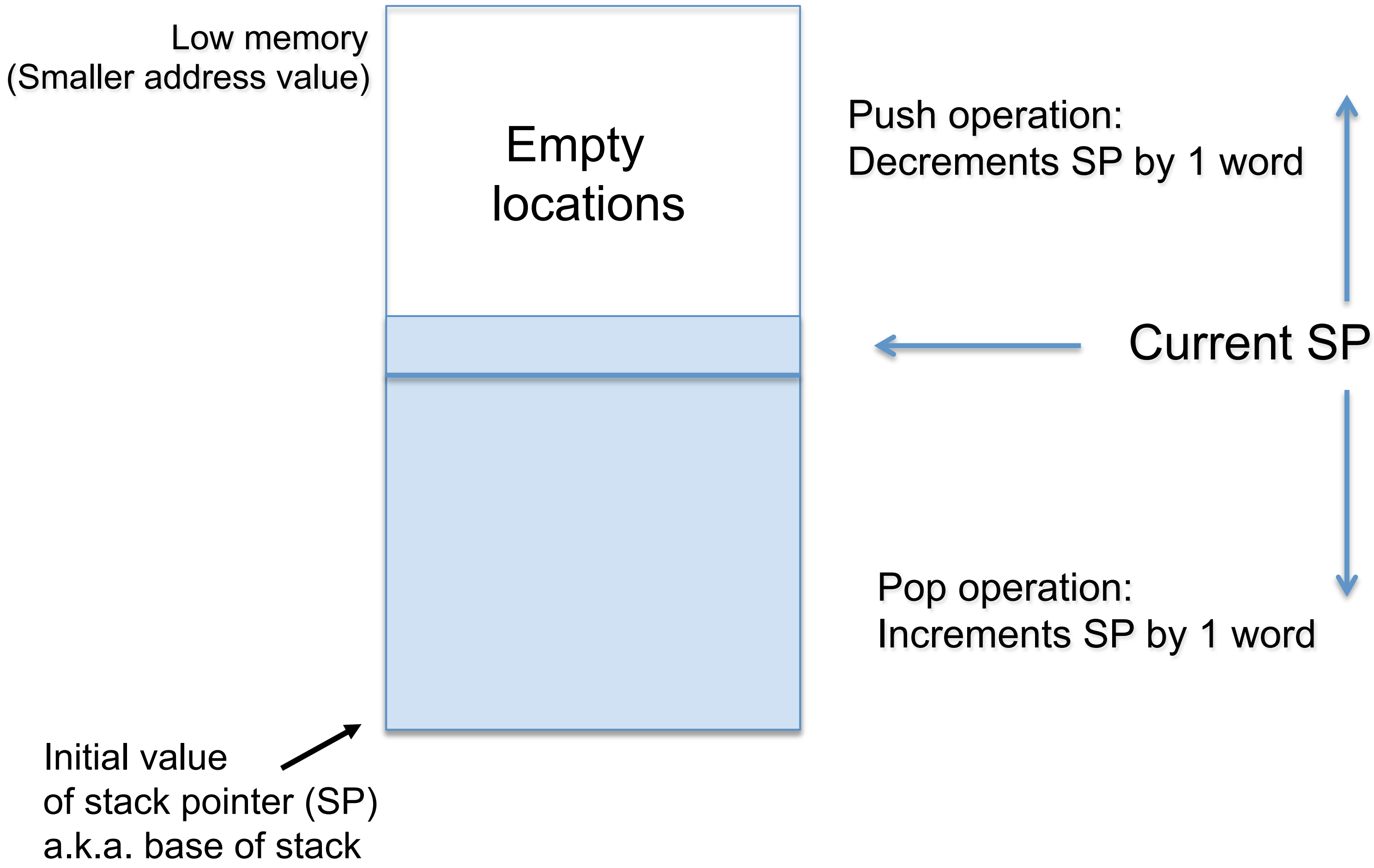
Module 8: Stack & Subroutines

- Concepts of Stack
- Using the 68000 Stack Pointer
- Subroutine Concepts
- Call & Return Instructions
- Parameter Passing

Stacks

- A stack is a Last In First Out (LIFO) buffer containing a number of data items usually implemented as a block of n consecutive bytes, words or long words in memory .
- The address of the last data item placed into the stack is pointed to by the Stack Pointer (SP).
- Application of stacks:
  - Temporary storage of variables
  - Temporary storage of program addresses
  - Communication with subroutines

Stacks


Low memory
(Smaller address value)
Empty
locations
Push operation:
Decrements SP by 1 word
Current SP
Pop operation:
Increments SP by 1 word
Initial value
of stack pointer (SP)
a.k.a. base of stack

68000 Stacks

- Stack addresses begin in high memory ($07FFE for example) and are pushed toward low memory ($07F00 for example). i.e. 68000 stacks grow into low memory.
- Other CPUs might do this in the reverse order (grow in high memory).
- Normally, address register A7 is used as a main stack pointer (SP) in the 68000. Using this register for other addressing purposes may lead to incorrect execution.
- 68000 stack item size:
  - One word for data.
  - One long word for addresses.
- User-defined stacks that use other item sizes (byte, long word), may be created by using address registers other than A7.

The Stack Pointer

- A7 is a special address register, called the *stack pointer*.
- When programming assembly, we can use SP as an alias for A7.

```
MOVEA.L  #$3000,SP
```

- It is also called USP (*user stack* pointer)
- There is also a supervisor stack pointer, but we won't worry about it yet.

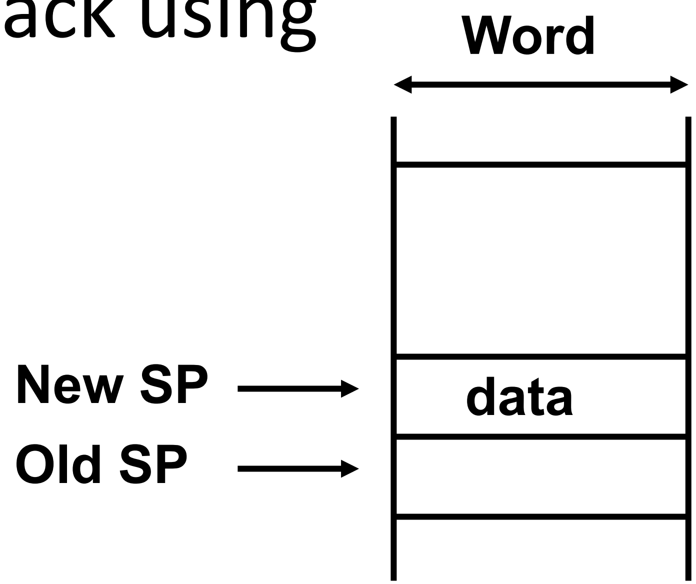
Push & Pop

- The stack grows upward toward the low address when items are pushed to the top of the stack.
- The stack pointer always points to the top item on the stack.
- When an item is pushed,
  - the stack pointer is decreased to point to the consecutive memory above
  - then the new item is added onto the stack
- When an item is popped,
  - the item on the top is copied to destination
  - then the stack pointer is increased to point to the consecutive memory below

Stack Push Operations

- To push an item onto the stack:
  - The stack pointer must be decremented by one word (i.e decremented by 2)
- We push values onto the stack using predecrement mode

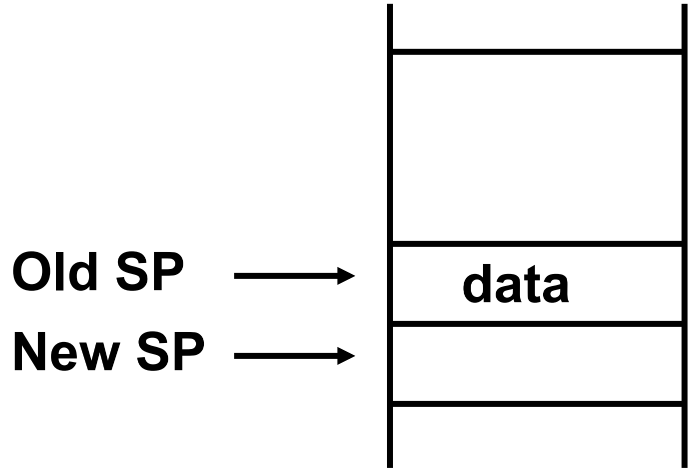
```
MOVE.W   D2,-(SP)
MOVE.W   D3,-(SP)
```



Stack Pop Operation

- To pop an item off the stack:
  - The information or data is read from the stack.
  - The stack pointer incremented by one word
- We pop values from the stack using postincrement mode

```
MOVE.W   (SP)+,  D3
MOVE.W   (SP)+,  D2
```



Other Instructions Affecting the Stack

- Special instruction MOVEM pushes multiple registers

```
MOVEM D0-D4/A0-A2,-(A7) for a push
MOVEM (A7)+,D0-D4/A0-A2 for a pop
```

  - Most commonly used during procedure calls
- Another way to put things on the stack is with the PEA instruction. It pushes an effective address on the stack, used when pushing pointers. This will decrease A7 with 4 (the size of a pointer).

Initializing The Stack Pointer

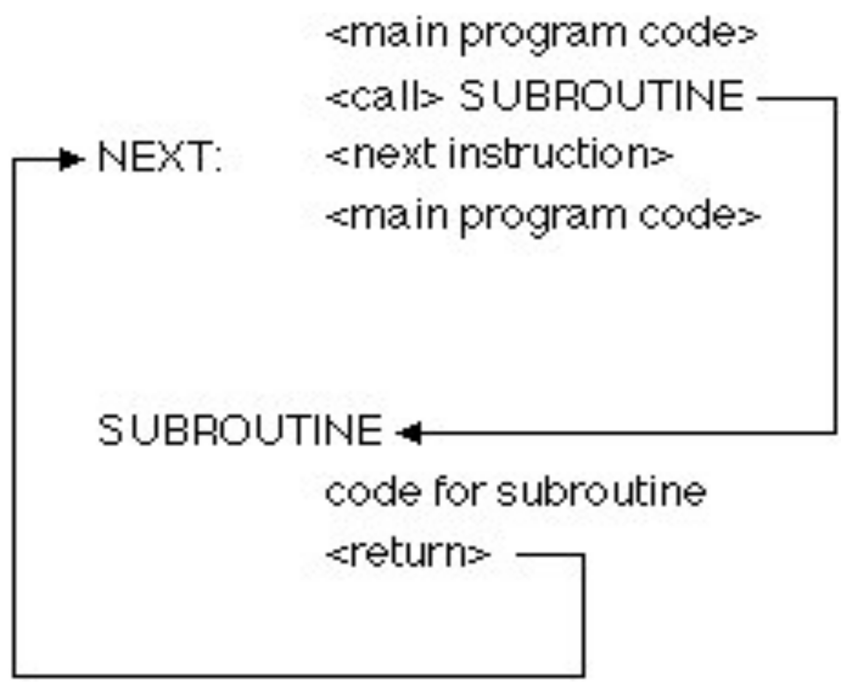
- It's the programmer's responsibility to initialize the stack. This involves two steps:
  - Initialize the stack pointer: The initial starting address or bottom of the stack.
  - Allocate sufficient memory for items to be pushed onto the stack. This could be done by locating the initial stack pointer at a very high memory address.

- Example:

```
INITSP  EQU     $07FFE          Value of INITSP
        MOVEA.L #INITSP,A7       Initialize  SP, A7
Or …
        LEA     INITSP,SP       Initialize  SP
```

Subroutines Basics

- A subroutine is a sequence of, usually, consecutive instructions that carries out a single specific function or a number of related functions needed by calling programs.
- A subroutine can be called from one or more locations in a program.
- Subroutines may be used where the same set of instructions sequence would otherwise be repeated in several places in the program.



Programming Subroutines

- Why use subroutines?
  - Code re-use
  - Easier to understand code (readability)
  - Divide and conquer
    - Complex tasks are easier when broken down into smaller tasks
  - Simplify the code debugging process.
- How do we call a subroutine in assembly?
  - Place the parameters somewhere known
  - JSR or BSR to jump to the subroutine
  - RTS to return
- Examples of subroutines:
  - Convert binary to ASCII
  - Convert Fahrenheit to Celcius
  - Perform output to 7-segment display

C Assembly

```
main() {
   int a, b;
   a = 5;
   b = sqr(a);
   printf("%d\n" b);
}
/* subrtn sqr */
int sqr(int val) {
   int sqval;
   sqval = val * val;
   return sqval;
}
```

```
main        MOVE.W      A,D1
            JSR         sqr
            MOVE.W      D0,B
            STOP        #$2700
;*** subroutine sqr ***
sqr         MUL.W       D1,D1
            MOVE.W      D1,D0
            RTS
;*** data area ***
            ORG         $2000
A           DC.W        5
B           DS.W        1
            end
```

68000 Subroutine Calling Instructions

BSR <subroutine_label>

- BSR = branch to subroutine
- **subroutine_label** is the address label of the first instruction of the subroutine.
- **subroutine_label** must be within no more than a 16-bit signed offset, i.e. within plus or minus 32K of the BSR instruction.
- Does not affect CCR

Example:

```
Main                . . .

                    BSR       Subroutine1
                    . . .
                    . . .
                    STOP

Subroutine1         MOVE ...
                    . . .
                    . . .
```

68000 Subroutine Calling Instructions

JSR <ea>

- JSR = jump to subroutine
- Similar in functionality to BSR, addressing mode <ea> must be a memory addressing mode.
  - i.e. <EA> cannot be a data or address register.
- The advantages of this instruction:
  - A number of different addressing modes are supported.
  - The address of the subroutine can be determined dynamically at execution time
    - Allows the selection of the subroutine to call at runtime
  - JSR does not affect CCR
- JSR is the most common form used for calling a subroutine.

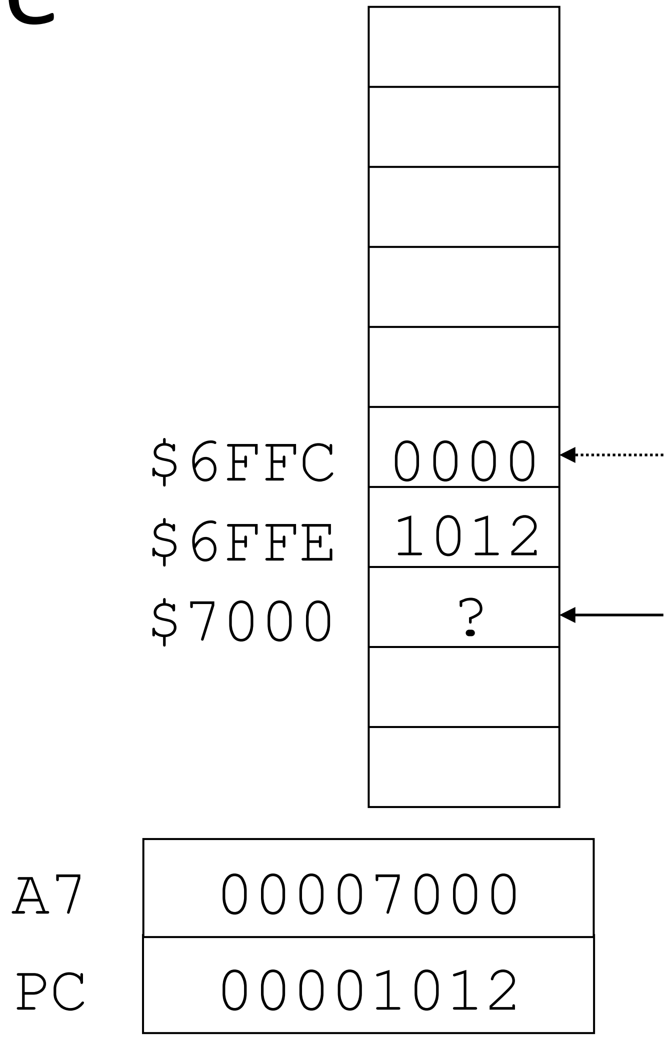
JSR vs BSR

- JSR *label* does:
  1. Decrement SP by 4
  2. Save current PC on top of stack
  3. Jump to subroutine. New PC can be derived using absolute mode and several address register indirect mode.
- In other words:
  1. SP ← [SP] – 4
  2. [SP] ← [PC]
  3. PC ← *<ea>*

- BSR *label* does:
  1. Decrement SP by 4
  2. Save current PC on top of stack
  3. Branch to subroutine. New PC is computing using current PC and offset provided by instruction.
- In other words:
  1. SP ← [SP] – 4
  2. [SP] ← [PC]
  3. PC ← *PC + offset*

JSR Example

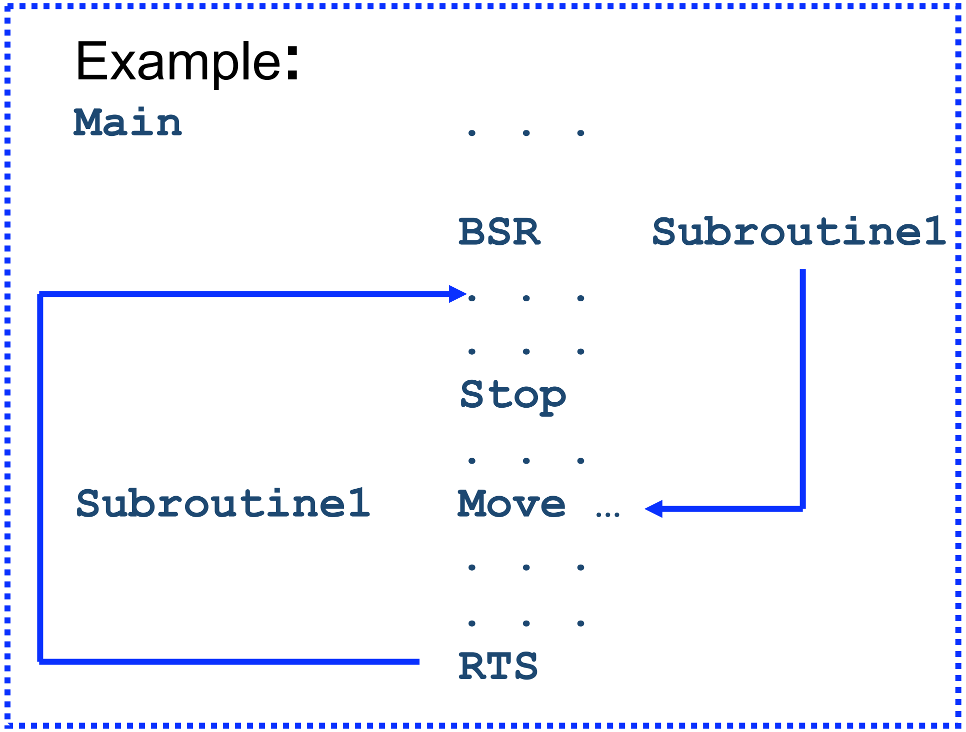
```
1000            MOVE.L    #5,D1
1006            LEA       ARRAY,A0
100C            JSR       SUMARR
1012            MOVE.L    D0,SUM
1018            STOP      #$2700
101E            NOP
1020   SUMARR   CLR.L     D0
                …
                RTS
       ARRAY    DC.L      12,15,31
       SUM      DS.L      1
                END
```



68000 Subroutine Return Instruction

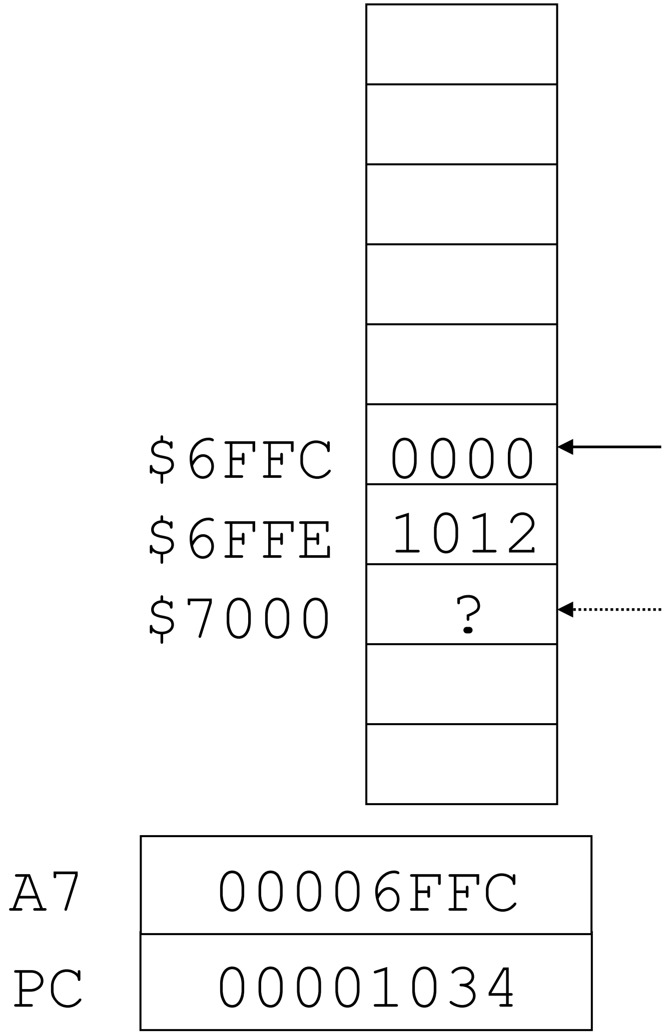
RTS

- RTS = ReTurn from Subroutine
- Pops the long word (return address) off of the top of the stack and puts it in the program counter in order to start executing after the point of the subroutine call.
- Post increments the stack pointer A7 by 4
- In other words, RTS does:
  - PC ← [SP]
  - [SP] ← [SP] + 4
- Does not affect CCR



RTS Example

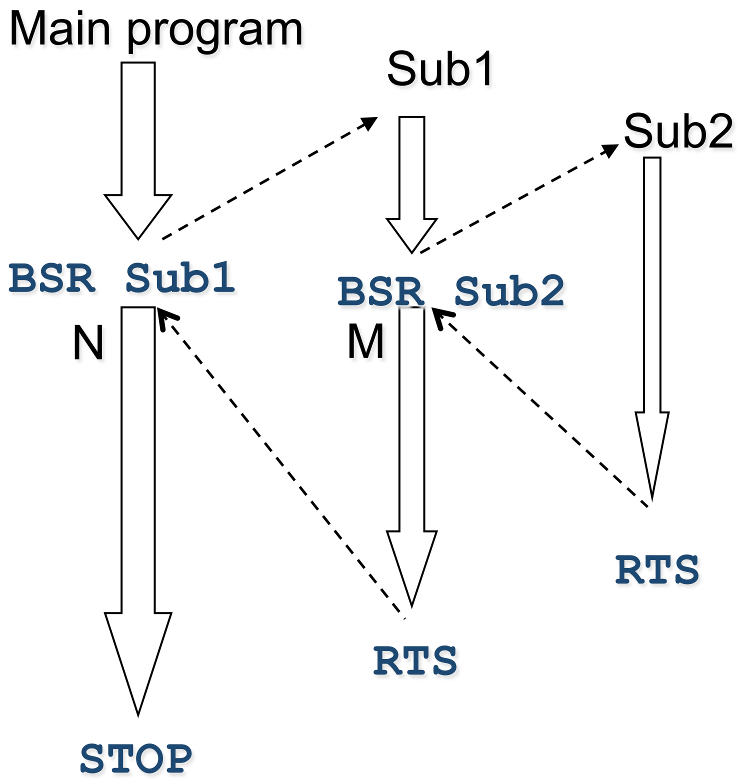
```
1000                MOVE.L    #5,D1
1006                LEA       ARRAY,A0
100C                JSR       SUMARR
1012                MOVE.L    D0,SUM
1018                STOP      #$2700
101E                NOP
1020      SUMARR    CLR.L     D0
                    …
1032                RTS
          ARRAY     DC.L      12,15,31
          SUM       DS.L      1
                    END
```



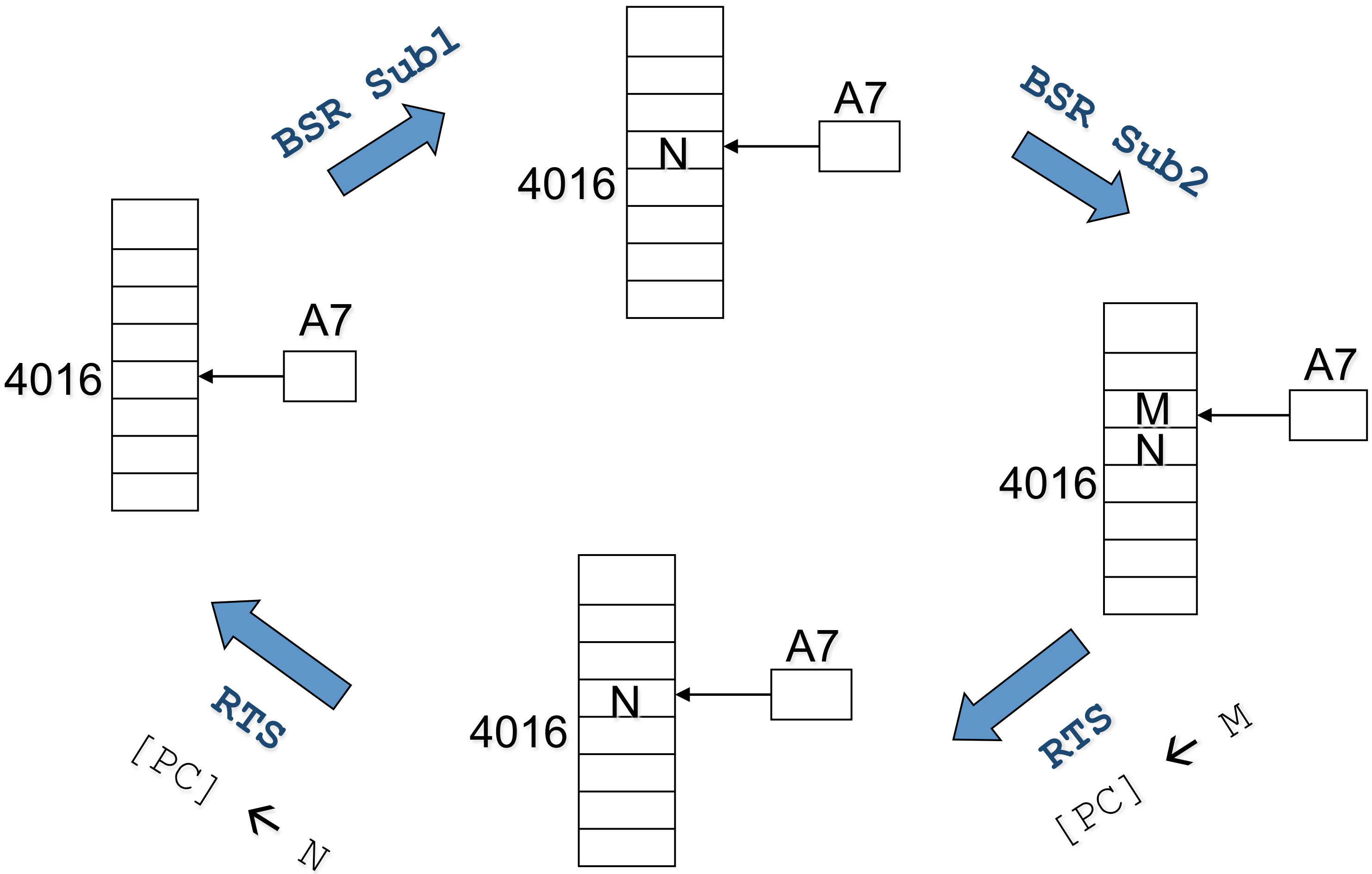
Nested Subroutines

```
* Main Program
          …
          BSR Sub1
N:        …
          . . .
          STOP #$2700
Sub1:     …
          BSR Sub2
M:        …
          . . .
          RTS

Sub2      …
          RTS
```



Nested Subroutines



Passing Parameters to Subroutines

- Parameters may be passed to a subroutine by using:
  - Data and Address Registers:
    - Efficient, position-independent.
    - It reduces the number of registers available for use by the programmer.
  - Memory locations:
    - This is similar to using static or global data in high level languages.
    - Does not produce position independent code and may produce unexpected side effects.
  - Stacks:
    - This is the standard, general-purpose approach for parameter passing. The LINK and UNLK instructions may be used to create and destroy temporary storage on the stack.
    - Similar to the approach used by several high-level languages including C.

Passing Parameters in Registers

```
; caller
main        MOVE.W      A,D1
            JSR         sqr
            MOVE.W      D0,B
            STOP        #$2700
; callee
sqr         MOVE.W      D1,D0
            MULS.W      D0,D0
            RTS
; data area
            ORG         $2000
A           DC.W        5
B           DS.W        1
            end
```

- The number to be squared is in D1.
- The result is returned in D0, D1 is unchanged.

Passing Parameters in Memory

```
; caller
main        MOVE.W      A,TEMP
            JSR         sqr
            MOVE.W      TEMP,B
; callee
sqr         MOVE.W      TEMP,D0
            MULS.W      D0,D0
            MOVE.W      D0,TEMP
            RTS
; data area
            ORG         $2000
A           DC.W        5
B           DS.W        1
TEMP        DS.W        1
            end
```

- The number to be squared is in stored in TEMP first.
- The result is returned in TEMP.

Parameter Passing on the Stack

- If we use registers to pass our parameters:
  - Limit of parameters to/from any subroutine.
  - We use up registers so they are not available to our program.
- So, instead we push the parameters onto the stack.
- Our conventions:
  - Parameters are passed on the stack
  - One return value can be provided in D0.
  - D0, D1, A0, A1 can be used by a subroutine. Other registers must first be saved.
- Both the subroutine and the main program must know how many parameters are being passed!
  - In C we would use a prototype:

```
int power (int number, int exponent);
```

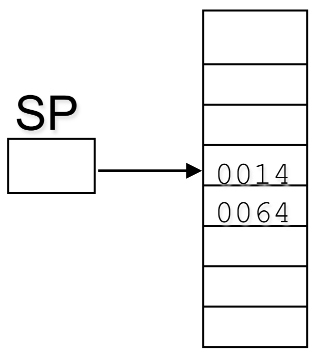
  - In assembly, you must take care of this yourself.

Steps in Using Stacks

CALLER

1. Push parameters on stack

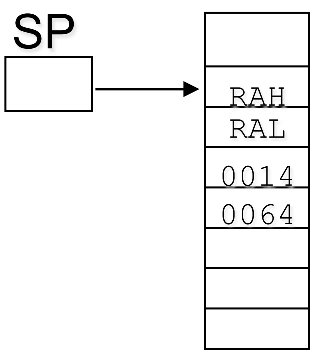
```
        MOVE.W   #100,-(SP)
        MOVE.W   #20,-(SP)
```



CALLER

2. Call the subroutine

```
        JSR      SQR
```



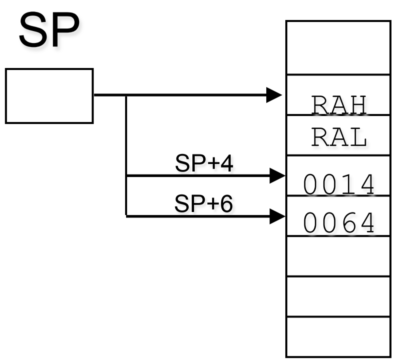
RAH = Return Address High
RAL = Return Address Low

Steps in Using Stacks

CALLEE

3. Extract parameters from stack

```
MOVE.W   4(SP),D0
MOVE.W   6(SP),D1
```



CALLEE

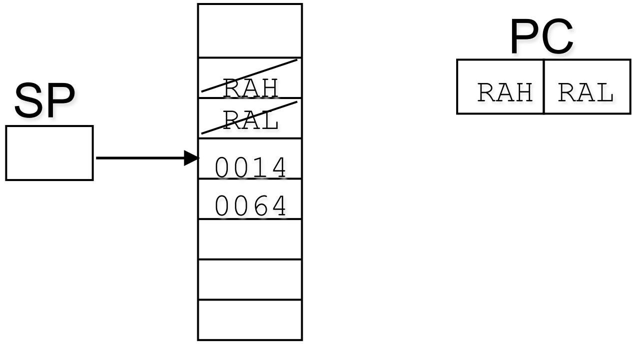
4. Use the parameters & store calculation result in D0.

```
MULU    D1,D0
```

CALLEE

5. Return to caller

```
RTS
```



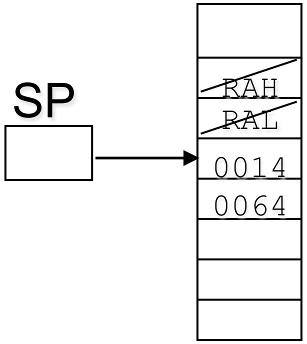
RAH = Return Address High
RAL = Return Address Low

Steps in Using Stacks

CALLER

6. Use returned data

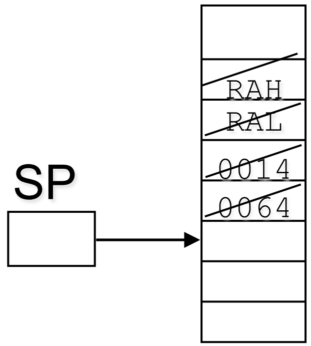
```
MOVE.W   D0,SAVE
```



CALLER

7. Clean up the stack

```
ADDA   #4,SP
```



Passing Parameters On The Stack

Mul3 – multiply three numbers and place the result in D0.

```
; ******** Main Program *********************************
1000      START      MOVE.W      NUM1,-(SP)       ;Push first param
1006                 MOVE.W      NUM2,-(SP)       ;Push 2nd param
100C                 MOVE.W      NUM3,-(SP)       ;Push 3rd param
1012                 JSR         MUL3
1018                 ADDA.L      #6,SP            ;Clean the stack!
101E                 STOP        #$2700
1020                 NOP
; ******* Subroutine Mul3 ******************************
1022      MUL3       MOVE.W      4(SP),D0         ;D0 = NUM3
1026                 MULS.W      6(SP),D0         ;D0 *= NUM2
102A                 MULS.W      8(SP),D0         ;D0 *= NUM1
102E                 RTS                          ;SP --> rtrn addr!
                     ORG         $2000
2000      NUM1       DC.W        5
2002      NUM2       DC.W        8
2004      NUM3       DC.W        2
                     END
```

Writing *Transparent* Subroutines

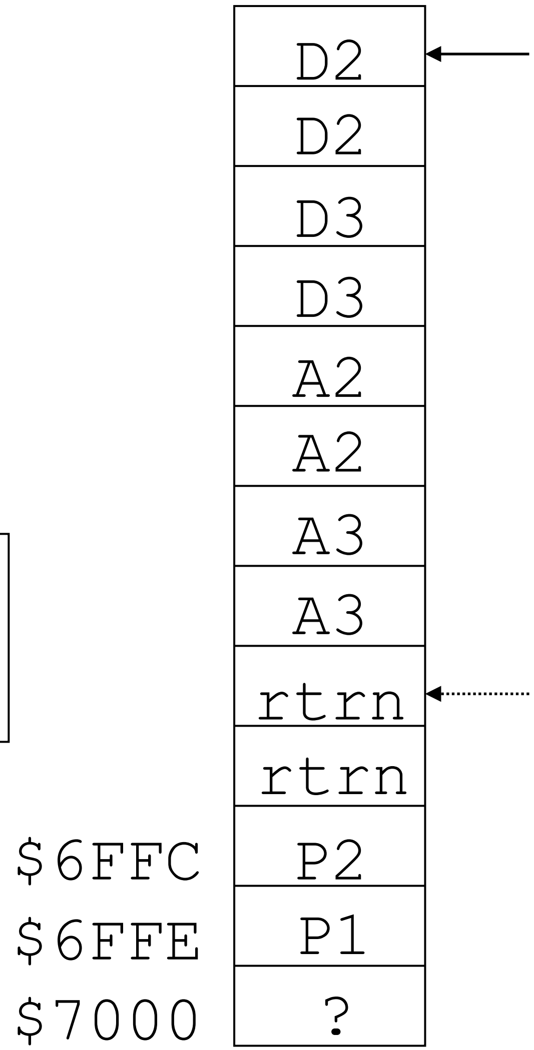
- A *transparent* subroutine doesn't change any registers except D0, D1, A0 and A1.
- If we need more registers than this, we must save the register values when we enter the subroutine and restore them later.
- Where do we store them? The stack, of course.
- The 68000 provides a convenient instruction, MOVEM, to push the contents of several registers to the stack at one time.

A Transparent Subroutine

```
subr1   MOVEM.L   D2-D3/A2-A3,-(SP)
        MOVE.L    #32,D2
        MOVE.L    20(SP),D3
        …
        MOVEM.L (SP)+,D2-D3/A2-A3
        RTS
```

We can now safely modify D2, D3, A2 and A3, knowing that we will restore their original contents later.

We saved 4 registers, so the last parameter lives at SP + (4×4) + 4. (4 bytes/reg + 4 for the return addr.)



Review Problem

```
; ******** Main Program *********
1000    START    MOVE.L     NUM1,-(SP)
1006             MOVE.L     NUM2,-(SP)
100C             MOVE.L     NUM3,-(SP)
1012             JSR        SUB1
1018             Next instr…
…                …
; ******* Subroutine SUB1 *******
1022    SUB1     MOVEM.L    D2-D3/A4,-(SP)
1026             … ; show stack
102E             MOVEM.L    (SP)+,D2-D3/A4
1032             RTS
                 ORG        $2000
2000    NUM1     DC.L       $150
2004    NUM2     DC.L       $180
2008    NUM3     DC.L       $12
                 END
```

Two Mechanisms For Passing Parameters

- By Value:
  - Actual value of the parameter is transferred to the subroutine .
  - This is the safest approach unless the parameter needs to be updated.
  - Not suitable for large amounts of data.
  - To pass a parameter by value through the stack, use the instruction:

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

- By Reference:
  - The address of the parameter is transferred.
  - This is necessary if the parameter is to be changed.
  - Recommended in the case of large data volume.
  - To pass a parameter by reference through the stack, use the instruction:

```
PEA    <EA>
```

The PEA Instruction

- PEA – *Push effective address*

  PEA label

  is the same as…

```
LEA       label,A0
MOVEA.L   A0,-(SP)
```

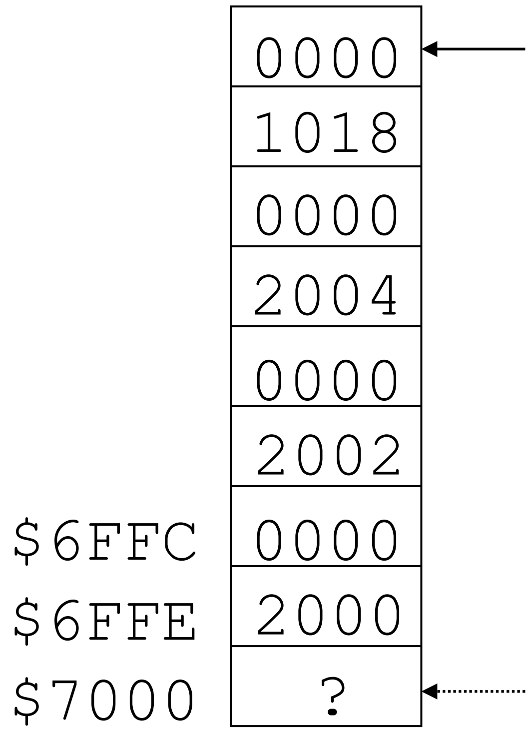
  but without using A0.

- *You can "abuse" this instruction to push a constant or any value on the stack.*

Passing Parameters By Reference

dbl3 – double the values of three parameters

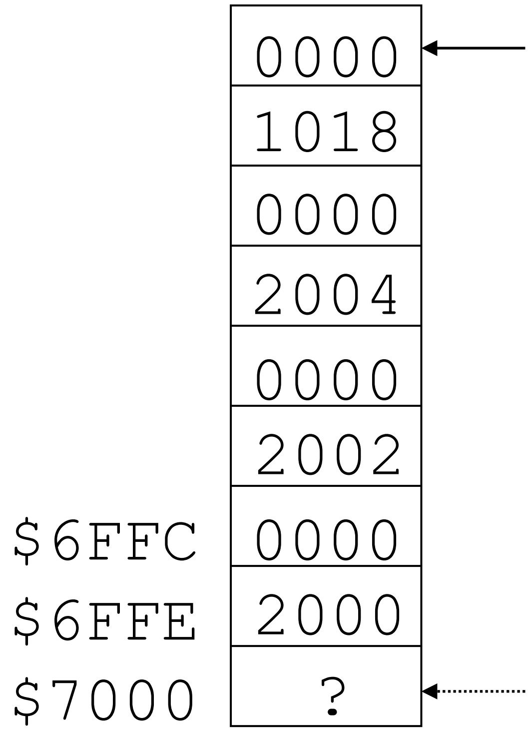
```
; ******** Main Program *********
1000    START   PEA      NUM1
1006            PEA      NUM2
100C            PEA      NUM3
1012            JSR      dbl3
1018            ADDA.L   #12,SP
101E            STOP     #$2700
1020            NOP
                ORG      $2000
2000    NUM1    DC.W     5
2002    NUM2    DC.W     8
2004    NUM3    DC.W     2
                END
```



Using Parameters Passed By Reference

dbl3 – double the values of three parameters

```
; ******* Subroutine dbl3 *********
        DBL3    MOVEA.L   4(SP),A0
                MOVE.W    (A0),D0
                MULS.W    #2,D0
                MOVE.W    D0,(A0)
                MOVEA.L   8(SP),A0
                … ; repeat for each
                RTS
                ORG       $2000
2000    NUM1    DC.W      5
2002    NUM2    DC.W      8
2004    NUM3    DC.W      2
                END
```



Characteristics Of Good Subroutines

- **Generality** – can be called with *any* arguments
  - Passing arguments on the stack does this.
- **Transparency** – you have to leave the registers like you found them, except for D0, D1, A0, and A1.
  - We use the MOVEM instruction for this purpose.
- **Readability** – well documented.
- **Re-entrant** – subroutine can call itself if necessary
  - This is done using *stack frames*…

ASCII-Encoded Decimal To Binary Conversion

- A useful subroutine

```
* Subroutine DECBIN
* A0 points to the highest character of  a valid five character
*  ASCII-encoded decimal number with a maximum value 65535
* The decimal number is converted to a one word binary value
*  stored in the low word of D0

DECBIN   CLR.L    D0         Clear result register
         MOVEQ    #5,D6      Initialize loop counter to get 5 digits
NEXTD    CLR.L    D1         Clear new digit holding register
         MOVE.B   (A0)+,D1   Get one ASCII digit from memory
         SUB.B    #$30,D1    Subtract ASCII bias $30
         MULU     #10,D0     Multiply D0 by 10
         ADD.W    D1,D0      Add new digit to binary value in D0
         SUB.B    #1,D6      Decrement counter
         BNE      NEXTD      If not done get next digit
         RTS
```

ASCII-Encoded Decimal To Binary Conversion

- A better version

```
* Subroutine DECBIN
* 4(SP) points to the highest character of a valid five character
*  ASCII-encoded decimal number with a maximum value 65535
* The decimal number is converted to a one word binary value
*  stored in the low word of D0

DECBIN     MOVE.L     4(SP),A0      Get the pointer from the stack
           MOVEM.L    D1/D6,-(SP)   Save the registers we're borrowing
           MOVEQ      #0,D0         MOVEQ faster than CLR.L
           MOVEQ      #5,D6         Initialize loop counter to get 5 digits
NEXTD      MOVEQ      #0,D1         Clear new digit holding register
           MOVE.B     (A0)+,D1      Get one ASCII digit from memory
           SUB.B      #$30,D1       Subtract ASCII bias $30
           MULU       #10,D0        Multiply D0 by 10
           ADD.W      D1,D0         Add new digit to binary value in D0
           SUB.B      #1,D6         Decrement counter
           BNE        NEXTD         If not done get next digit
           MOVEM.L    (SP)+,D1/D6   Restore registers
           RTS
```