Programming in Assembly

Data Structures, Addressing Modes, and Flow Control Basics

Working with ASCII

DC.B ‘Hello world‘

2000	H	e
2002	l	l
2004	o	-
2006	w	o
2008	r	l
200A	d	

2000	48	65
2002	6C	6C
2004	6F	20
2006	77	6F
2008	72	6C
200A	64	

Programming with subroutines

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
- How do we call a subroutine in assembly?
  - Place the parameters *somewhere known*
  - JSR or BSR to jump to the subroutine
  - RTS to return

C

```
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;
}
```

Assembly

```
main     MOVE.W   A,D1
         JSR      sqr
         MOVE.W   D0,B
         move.w   #228,D7
         TRAP     #14
;*** subroutine sqr ***
sqr      MUL.W    D1,D1
         MOVE.W   D1,D0
         RTS
         ORIGIN   $400800
A        DC.W     5
B        DS.W     1
         end
```


JSR and BSR

- **`JSR`** *label*

```
MOVE.L   PC, -(SP)
LEA      address-of-label, PC
```

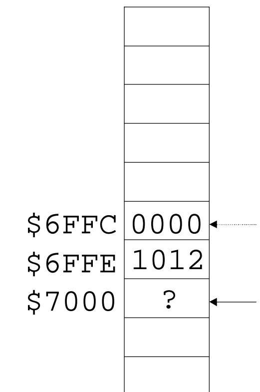
In other words:

```
SP ¬  [SP] - 4
[SP] ¬  [PC]
PC ¬  address-of-label
```

- **`BSR`** *label*
  - Same, but offset from the current PC is stored instead of the absolute address

JSR example

```
1000            MOVE.L   #5,D1
1006            LEA      ARRAY,A0
100C            JSR      SUMARR
1012            MOVE.L   D0,SUM
1018            MOVE.L   #228, D7
101E            TRAP     #14
1020   SUMARR   CLR.L    D0
                …
                RTS
       ARRAY    DC.L     12,15,31
       SUM      DS.L     1
                END
```



A7	00007000
PC	00001012

RTS

- **`RTS`**

  Pop the address from the stack back into the PC

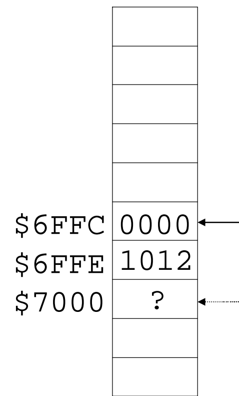
```
MOVE.L   (SP)+, PC
```

In other words:

```
PC ¬  [SP]
SP ¬  [SP] + 4
```

RTS example

```
1000            MOVE.L    #5,D1
1006            LEA       ARRAY,A0
100C            JSR       SUMARR
1012            MOVE.L    D0,SUM
1018            MOVE.L    #228, D7
101E            TRAP      #14
1020    SUMARR  CLR.L     D0
                …
1032            RTS
        ARRAY   DC.L      12,15,31
        SUM     DS.L      1
                END
```



A7 00006FFC

PC 00001034

Passing parameters in registers

```
main    MOVE.W    A,D1
        JSR       sqr
        MOVE.W    D0,B
        MOVE.W    #228,D7
        ...
;
sqr     MOVE.W    D1,D0
        MULS.W    D0,D0
        RTS
        ORG       $400800
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.

Parameter passing on the stack

- If we use registers to pass our parameters:
  - Limit of 7 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.*

First things first…

- 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.
- After you return from a subroutine, you must also *clear the stack*. You have to clean up your mess.

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            MOVE.W    #228,D7
1020            TRAP      #14
; ******* 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
```


Know how to…

- Push parameters onto the stack
- Access parameters on the stack using indexed addressing mode
- Draw the stack to keep track of subroutine execution
  - Parameters
  - Return address
- Clean the stack after a subroutine call

Review problem

```
; ******** Main Program *********
1000   START    MOVE.L    NUM1,-(SP)
1006            MOVE.L    NUM2,-(SP)
100C            MOVE.L    NUM3,-(SP)
1012            JSR       SUB1
1018            ...
                ...
; ******* Subroutine SUB1 *******
1022   SUB1     …
1026            …
102E            RTS
                ORG       $2000
2000   NUM1     DC.L      $150
2004   NUM2     DC.L      $180
2008   NUM3     DC.L      $12
                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? You guessed it: the stack.
- 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.W   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.)

Address	Stack
	A3
	A3
	A2
	A2
	D3
	D3
	D2
	D2
	rtrn
	rtrn
$6FFC	P2
$6FFE	P1
$7000	?

Review problem #2

```
; ******** Main Program *********
1000   START   MOVE.L   NUM1,-(SP)
1006           MOVE.L   NUM2,-(SP)
100C           MOVE.L   NUM3,-(SP)
1012           JSR      SUB1
1018           ......
               ......
; ******* Subroutine SUB1 *******
1022   SUB1    MOVEM.L  D2-D3/A4,-(SP)
1026           ......
102E           MOVEM.L  (SP)+,D2-D3/A4
1032           RTS
               ORG      $400800
2000   NUM1    DC.L     $150
2004   NUM2    DC.L     $180
2008   NUM3    DC.L     $12
               END
```


Know how to…

- Write transparent subroutines
- Draw the stack and access parameters on the stack, taking into account the saved register values and the subroutine return address.

Passing by value & reference

- We pushed the value of NUM1, NUM2, and NUM3 on the stack.
- *What if we want to change the input parameter values?*
- For example, what if we want to write a subroutine that will multiply all of its arguments' values by 2, *actually changing the values in memory?*
- We must pass the parameters by reference…

The PEA instruction

- PEA – *Push effective address*

```
PEA label
```

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

Both instructions are the same. **PEA** allows us to avoid using **A0** in a more compact way.

Passing parameters by reference

dbl3 – double the values of three parameters

```
; ******** Main Program *********
1000   START   PEA.L    NUM1
1006           PEA.L    NUM2
100C           PEA.L    NUM3
1012           JSR      dbl3
1018           ADDA.L   #12,SP
101E           MOVE.W   #228,D7
1020           TRAP     #14
               ORG      $2000
2000   NUM1    DC.W     5
2002   NUM2    DC.W     8
2004   NUM3    DC.W     2
               END
```

Address	Stack
	0000 ←
	1018
	0000
	2004
	0000
	2002
$6FFC	0000
$6FFE	2000
$7000	? ←

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      $400800
2000   NUM1    DC.W     5
2002   NUM2    DC.W     8
2004   NUM3    DC.W     2
               END
```

Address	Stack
	0000 ←
	1018
	0000
	2004
	0000
	2002
$6FFC	0000
$6FFE	2000
$7000	? ←

Putting it all together

```
MAIN     MOVE.W     COUNT,-(SP)
         PEA        ARRAY
         JSR        SUB2
         …
SUB2     MOVEM.L    D2-D3,-(SP)
         ;HERE
         ORG        $400900
COUNT    DC.W       105
ARRAY    DC.W       10,12,3,7,9,18
```

- What does the stack look like at the line **;HERE** ?

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.
  - See the example handout.
- **Re-entrant** – subroutine can call itself if necessary
  - This is done using *stack frames*, something we will look at in a few days…

Know how to…

- Pass parameters by value and by reference
- Use address registers to access parameters passed by reference
- Write general, transparent, readable subroutines
  - We'll work on re-entrant subroutines next time.

Go over example subroutine

- Grading:
  - Correctness: *Does it assemble, and run correctly. Does it produce correct output for various input values?*
  - Readability
    - High-level algorithms
    - Register usage lists
    - Inline and header comments
    - Input/output parameter tables
  - Efficiency
    - ADDQ, CLR, registers in loops, (A0)+, etc.

Stack Frames: Using LINK

Assembling source code

- One source file:
  - Use ORG statements for data and code
  - Assemble and link in one step using: **`asm68k file.asm -lx`**
- Multiple source files:
  - ORG statement is **not allowed.**
  - New assembler directives:
    - SECTION *name*,code — Needed for linking.
    - XDEF *label* — Make a label available to the world
    - XREF *label* — Import a label from another file
  - See handout for examples…

Using multiple source files

- First assemble into relocatable object files:

```
asm68k file1.asm -lx -r
asm68k file2.asm -lx -r
```

  - Both of the .o files will have internal addresses starting at $0000 0000
- Then link into an S-record file:

```
link 1000 -m prog.s file1.o file2.o
```

  - Use link68k on UNIX machines
  - The 1000 is the base address for the program
  - Use the .map file and the .lst file to find addresses for your code.

Local Variables in Subroutines

- *What if we need more than 8 local variables?*
- If we use DC and DS, then all local variables *for all subroutines*, take up memory all the time.
  - For large programs, this can be a problem.
- Using DC and DS also causes problems with recursion – only one copy of the variables.

Recursion

- A recursive function to compute Fibonacci #'s:

```
int fib(int N)
{
  int prev, pprev;

  if (N == 1) {
      return 1;
  }
  else if (N == 2) {
      return 1;
  }
  else {
    prev = fib(N-1);
    pprev = fib(N-2);
    return prev + pprev;
  }
}
```

Local Variables and Recursion

- We could use named memory locations, but then what happens if a subroutine calls itself (recursion)?

```
SUB1  MOVE.W  4(SP),VAR1
      MOVE.W  8(SP),VAR2
      …
      JSR     SUB1
      MOVE.W  VAR2,D0
      RTS
VAR1  DS.W    1
VAR2  DS.W    1
```

- Anything that can be done in a high level language must be do-able in assembly!

Using the stack for local variables

- To allow for the possibility of recursion, we keep local variables on the stack.
- The LINK statement allocates space for local variables in subroutines.

LINK A6,#-8

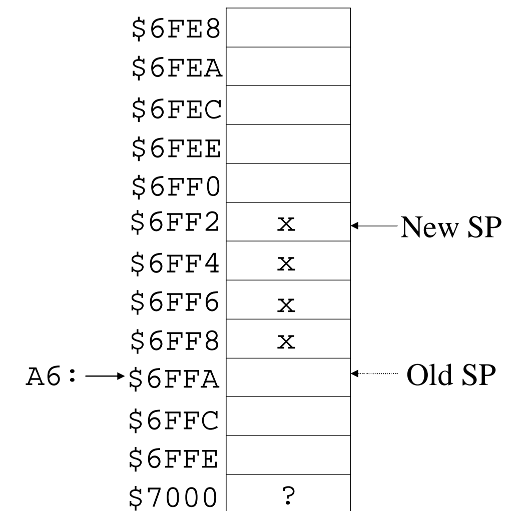
- Allocates 8 bytes (4 words) for local variables. Called a *stack frame*.
- Saves the value of **A6** and establishes **A6** as a *frame pointer*.

What LINK does. (Conceptually)

- Conceptually, link makes some room on the stack for local variables.

LINK A6,#-8

- UNLK puts things back "like they were".
  - Much like RTS

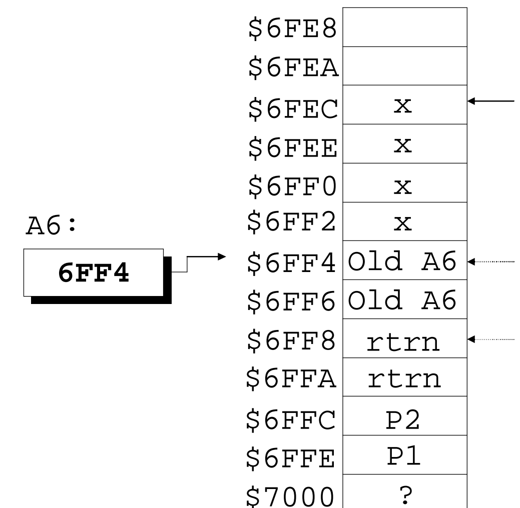


What LINK does. (Exactly)

LINK A6,#-8

MOVEM.L …

- SP <- [SP] – 4
- [SP] <- [An]
- An <- [SP]
- SP <- [SP] + disp
- Parameters:
  - 8(A6) = P2
  - 10(A6) = P1
- Local variables:
  - -2(A6) through -8(A6)
- Now MOVEM doesn't affect these values!

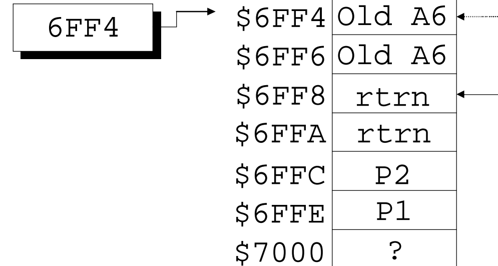


UNLK

UNLK A6

- SP <- [An]
- An <- [[SP]]
- SP <- [SP] + 4

- Back to where we started, ready to execute RTS.



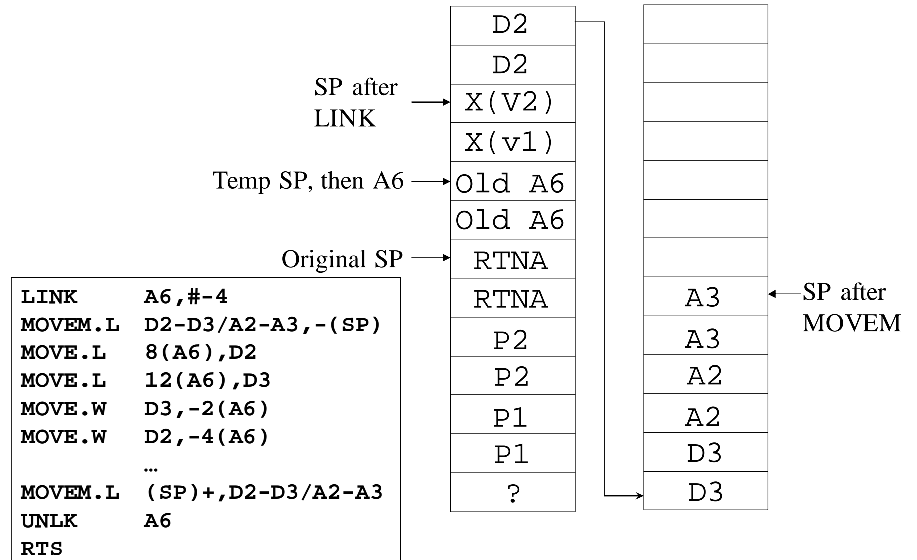
A subroutine using LINK

Assume we are expecting two parameters, and both are long words:

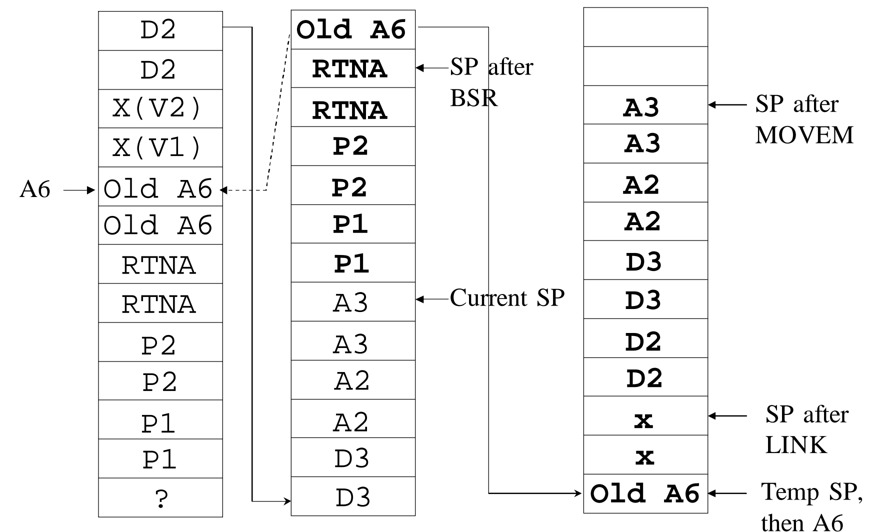
```
SUB1   LINK      A6,#-4
       MOVEM.L   D2-D3/A2-A3,-(SP)
       MOVE.L    8(A6),D2        ; Param2 = 8(A6)
       MOVE.L    12(A6),D3       ; Param1 = 12(A6)
       MOVE.W    D3,VAR1(A6)     ; Local Var1 = -2(A6)
       MOVE.W    D2,VAR2(A6)     ; Local Var2 = -4(A6)
       …
       MOVEM.L   (SP)+,D2-D3/A2-A3
       UNLK      A6
       RTS

VAR1   EQU       -2
VAR2   EQU       -4
```


The stack for that subroutine:



What if the subroutine calls itself?

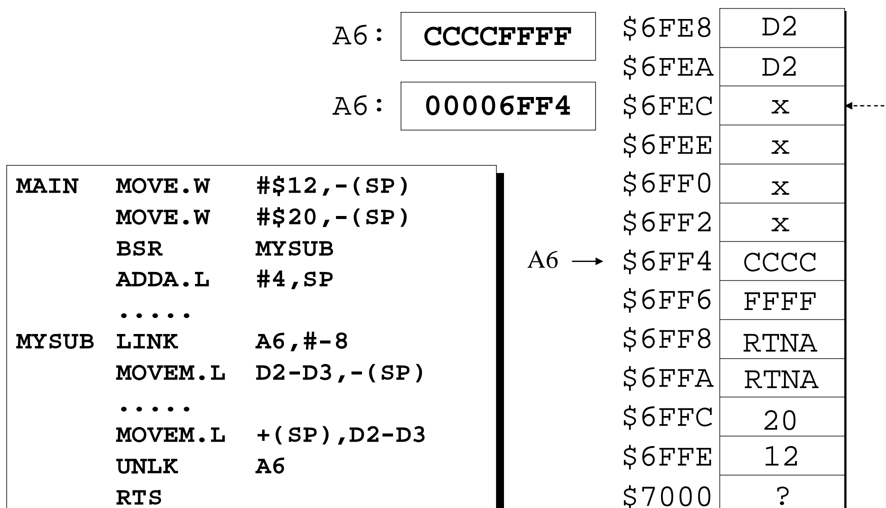


Example

```
MAIN  MOVE.W  #$12,-(SP)
      MOVE.W  #$20,-(SP)
      BSR     MYSUB
      ADDA.L  #4,SP
      ......
MYSUB LINK    A6,#-8
      MOVEM.L D2-D3,-(SP)
      ;DRAW THE STACK
      ......
      MOVEM.L +(SP),D2-D3
      UNLK    A6
      ;IS SP pointing at the return address?
      RTS
```

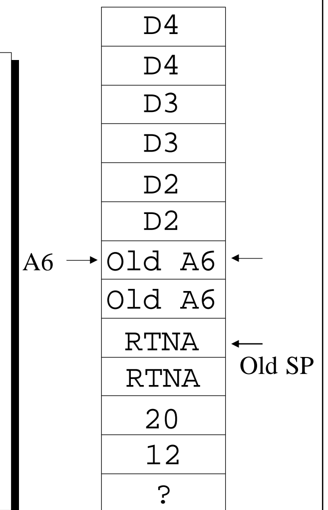

Example Stack

```
MAIN  MOVE.W   #$12,-(SP)
      MOVE.W   #$20,-(SP)
      BSR      MYSUB
      ADDA.L   #4,SP
      .....
MYSUB LINK     A6,#-8
      MOVEM.L  D2-D3,-(SP)
      .....
      MOVEM.L  +(SP),D2-D3
      UNLK     A6
      RTS
```



LINK An,#0 – Why would I do that??

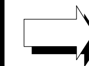
```
MAIN  MOVE.W  #$12,-(SP)
      MOVE.W  #$20,-(SP)
      BSR     MYSUB
      ADDA.L  #4,SP
      .......
MYSUB LINK    A6,#0
      MOVEM.L D2-D7/A2-A5,-(SP)
      .......
; Params are easy to find!
      MOVEM.L +(SP), D2-D7/A2-A5
      UNLK    A6
      RTS
```



The stack and the heap

- C and C++ allocate *all local variables in a stack frame*, using registers only temporarily.
- Malloc'd memory comes from a different place: *the heap*.

```
int main() {
    int a, b;
    a = 3;
    b = 7;
}
```



```
MAIN  LINK    A6,#-8
      MOVE.L  #3,-4(A6)
      MOVE.L  #7,-8(A6)
      …
```

A Recursive Subroutine

```
main     MOVE.W    NUM,-(SP)       ; Push NUM by value
         JSR       fact            ; Puts result in D0
         MOVE.W    #228,D7         ; Magic exit code
         .......
fact     LINK      A6,#0           ; Prepare for recursion
         MOVE.L    D1,-(SP)        ; Save D1 to the stack
         MOVE.W    8(A6),D0        ; D0 = num
         CMP.W     #1,D0           ; if (num <= 1)
         BLE       done            ;  return(num)
         MOVE.W    D0,D1           ; else { D1 = num;
         SUBQ.W    #1,D0           ;
         MOVE.W    D0,-(SP)        ; D0 = fact(num-1);
         JSR       fact
         ADDA.L    #2,SP           ; clean the stack
         MULU.W    D1,D0           ; D0 *= num;
done     MOVE.L (SP)+,D1           ; restore D1
         UNLK      A6              ; clean up
         RTS                       ; Return the result in D0
NUM      DC.W      7               ; # to factorialize
         end
```


How C uses LINK…

```
int sub1(int a, int b) {
  int var1, var2, var3;
  int i, j, k, l;
  …
  var1 = 7;
  var2 = a;
  var2 += 15;
  …
  return 32;
}
```

```
_sub1   LINK     A6,#-28
        ......
        MOVE.L   #7,-4(A6)
        MOVE.L   12(A6),-8(A6)
        MOVE.L   8(A6),D0
        ADDI.L   #15,D0
        MOVE.L   D0,8(A6)
        ......
        MOVE.L   #32, D0
        UNLK     A6
        RTS
```


You should know…

- Why LINK is important for recursion.
- How to use LNK to set up local variables
- How to access parameters and local variables using a frame pointer.
- How to draw the stack to show the effects of BSR/JSR, MOVEM, LINK, UNLK, and RTS operations.
- Write a subroutine that uses a stack frame for local variables.