Passing parameters on the stack.

- Short coming of passing parameters in registers:
  - one-level subroutine call only
  - More than one level subroutine call requires subroutine to save parameters to memory locations (very clumsy).

- A better way to pass parameters:

  Consider function activation sequence:

A's parameter is only needed while A is active

(not needed when A exits - returns to main.

same holds for B and C

- <u>Conclusion</u>: using the stack to pass parameters will be very effective, because the parameters are only on the stack as long as the function is active!

- <u>How to pass parameters on the stack.</u>

Example:

```
main: int a,b,c;

      c = sum(a,b)
```

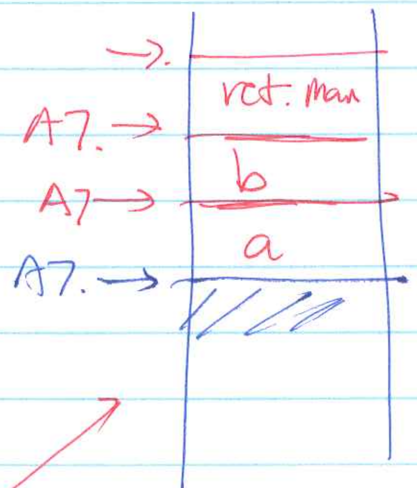
```
int sum(int x, int y)
{
   return(x²+y²);
}
```

<u>Before</u>:

```
move.l a, d0
move.l b, d1
bsr sum.
```

(you must pass parameter first; then call subroutine.)

↓ pass a, b on stack.




```
subq.l #4, A7
move.l a, (A7)
subq.l #4, A7
move.l b, (A7)
```

Stack looks like this.

Special push addressing mode

~~Because~~ The operation to push a value on the system stack is:

① suba.l #4, A7 — increase stack top
② move.l value, (A7) — save on stack.

This happens so often, M68000 has a special addressing mode for this:

```
move.l  value, -(A7)
```

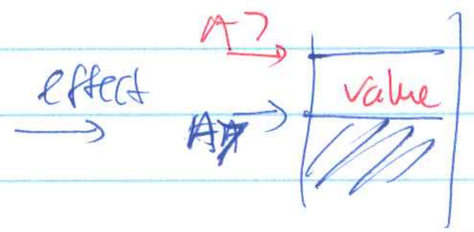
① decrement A7 by an amount equal to long (4) (suba.l #4, A7)

② move.l value, (A7).

You can also push words:

```
move.w  value, -(A7)
```

effect →

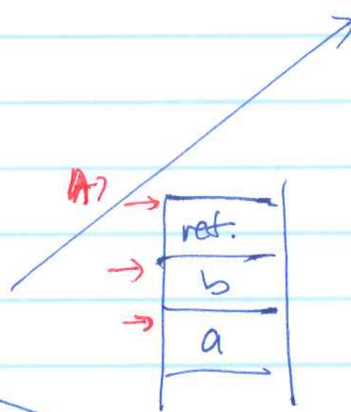


≡ ① suba.l #2, A7

② move.w value, (A7)

Caller:

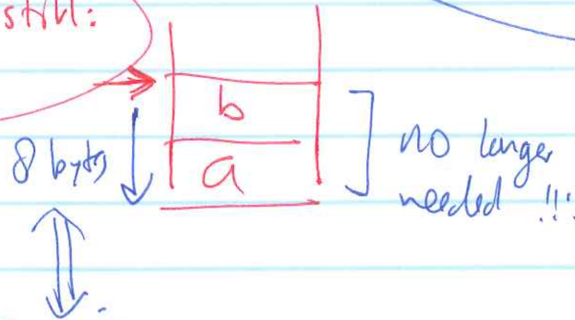
```
move.l a, -(A7)
move.l b, -(A7)

call sum
```



sum:

```
move.l 8(A7),d0
muls d0,d0

move.l 4(A7),d1
muls d1,d1

add.l d1,d0
RTS
```

only pops return address!




```
adda.l #8, A7
```

clean up stack

```
move.l d0, C
```

Demo:

sub-stack-param1.s

Passing Parameters on Stack: Local variables on stack

- For the same reason why we want to pass parameters on the stack, we should put local variables on the stack also:
  - the local variables are only "active" (needed) when the function is running.
  - They must be clean up when the function returns.
  - It is very effective to use the stack to store local variables.

- Example program:

main:

```
int A[10]
int sum

sum = ArraySum(A, 10);
```

```
int ArraySum(int A[],
             int n)
{   int i, s;

    s = 0;
    for (i=0; i<n; i++)
        s = s + A[i];
} return(s);
```

main:

```
move.l #A,-(A7)
move.l #10,-(A7)

bsr ArraySum

adda.l #8,A7
move.l d0,sum
```

DEMO

sub-stack-param2.s

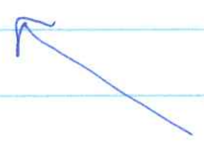
ArraySum:

```
suba.l #8, A7
```

stack now:

0(A7)	(S)	← a7
4(A7)	(i)	
8(A7)	return addr	
12(A7)	#10 (n)	
16(A7)	#A (A)	

```
        move.l #0, (A7)        [S=0]

        move.l #0, 4(A7)       [i=0]

while:  move.l 4(A7), D0       [i]
        move.l 12(A7), D1      [n]
        cmp.l  D1, D0          i ≥ n

        BGE    WhileEnd

        movea.l 16(A7), A0
        move.l  4(A7), D0
        muls    #4, D0
        move.l  (A0, D0.w), D0
        add.l   D0, (A7)

        addq.l  #1, 4(A7)
        BRA     WHILE

WhileEnd: move.l (A7), D0
        adda.l #8,A7
        RTS
```



- This solution works, but it is very clumpsy:

  if at a future time, the algorithm/program need to be expanded with:

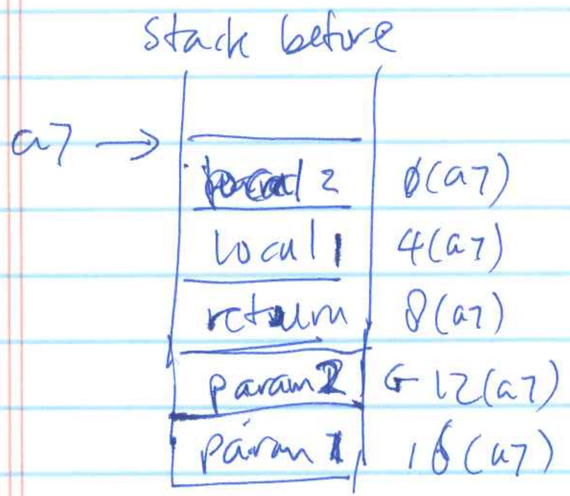
  - addition parameters

  or
  - addition local variables (to write program)

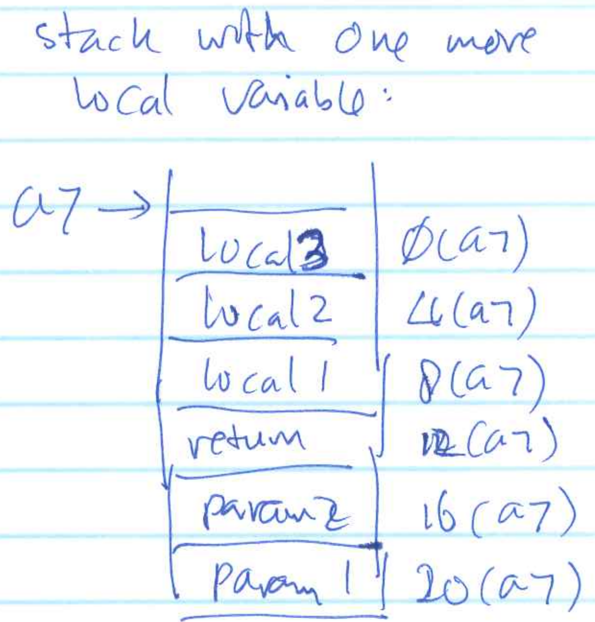
  then the WHOLE program changes:

Eg: adding one more local variable:

stack before



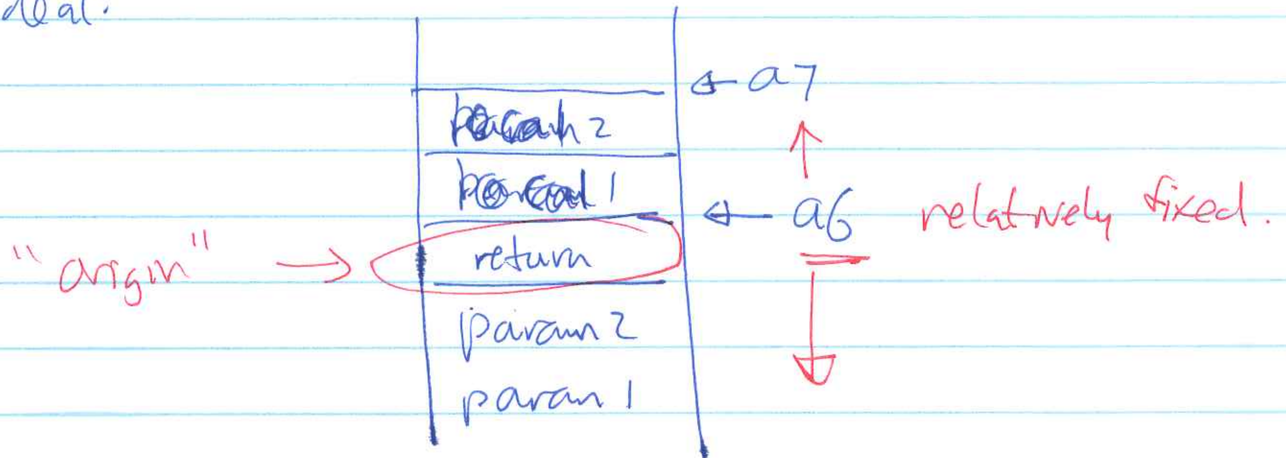
stack with one more local variable:



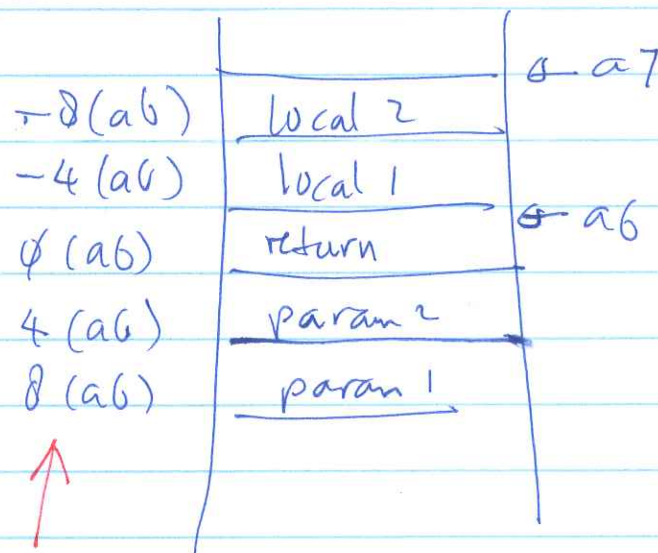
All offsets in the program must be recalculated !!!

Improved solution: use fixed "frame pointer" to access parameters and local variables.

Ideal:



Scheme for accessing local variables and parameters:



neg. offsets to access local variables

positive offsets to access parameters.

main: (unchanged)

```
move.l  #A, -(a7)
move.l  #10, -(a7)

bsr  ArraySum

adda.l  #8, a7
move.l  dø, sum
```

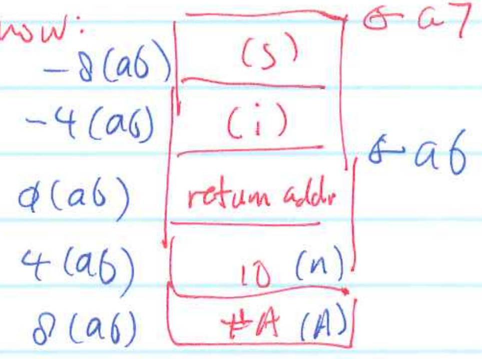
Demo

sub-stack-param3.s

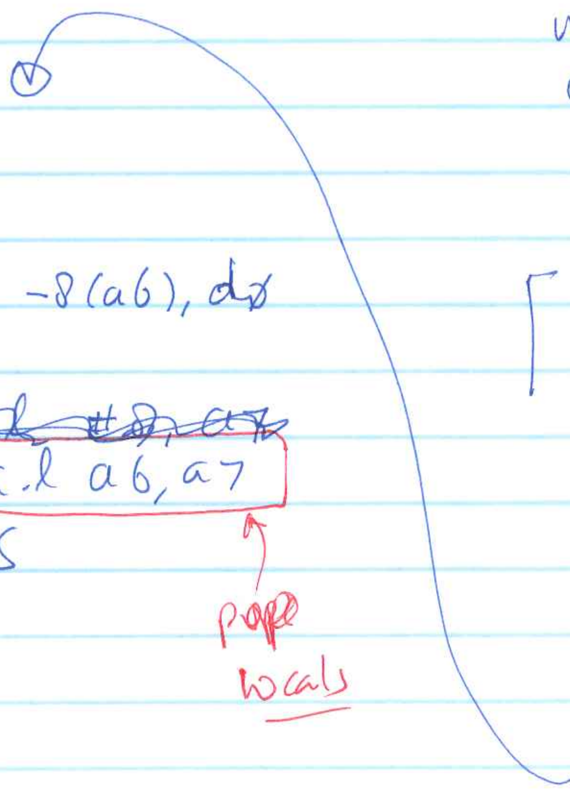
ArraySum:

```
movea.l  a7, a6        make a6 point to base

suba.l  #8, a7         (reserve 2 local vars.
```

Stack is now:




```
        move.l  #ø,  -8(a6)      [s=0]
        move.l  #ø,  -4(a6)      [i=0]

while:  move.l  -4(a6), Dø       [i]
        move.l  4(a6), d1        [n]
        cmp.l   D1, dø

        BGE  whileEnd

        moveal  8(a6), Aø
        move.l  -4(a6), dø
        muls    #4, dø
        move.l  (aø,dø.w), dø
        add.l   dø, -8(a6)
        addq.l  #1, -4(a6)
        BRA  while
```



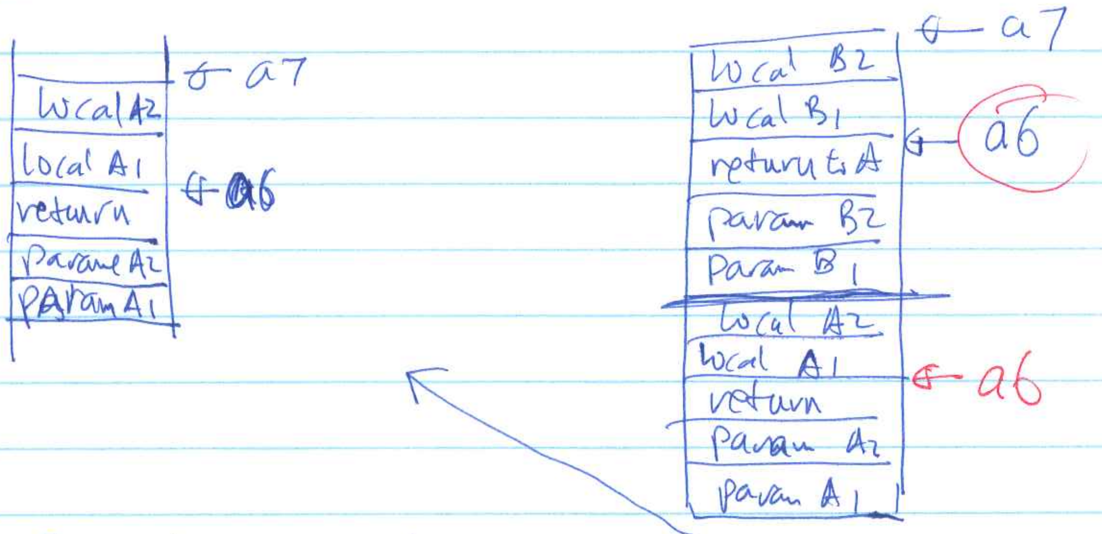
```
whileEnd:

        move.l  -8(a6), dø
        .
Better  adda.l #8, a7   (crossed out)
        movea.l a6, a7      ← pops locals
        RTS
```

- **There is ONE problem with the solution:**

  It does not work with multiple level subroutine call.

- Reason:

main → A → B.



when B returns to A.
(if successful)
we lost the value of a6 !!!

There is no way to recover the value of a6!

Solution: save the value of a6

restore value of a6 when we return !!!

Question: Where ~~How~~ should we save a6 of caller?

Answer: Stack!!!

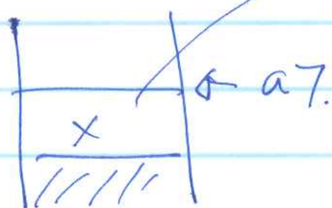
Because the saved value is only needed as long as the subroutine is active.

When subroutine returns, we restore the callers a6 value, and the saved value is no longer need!

This require popping from stack.

How to pop off a stack:

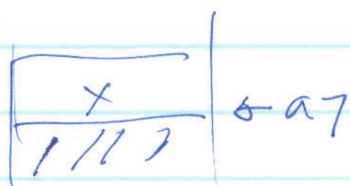
Before:



① move.l (a7), d∅

② adda.l #4, a7

After:



New instruction mode:

move.l (a7)+, d∅.

(1) move.l (a7)+ , dn

≡ { move.l (a7) , dn
  adda.l #4 , a7

(2) move.w (a7)+ , dn

≡ { move.w (a7) , dn
  adda.l #2 , a7

main: (unchanged)

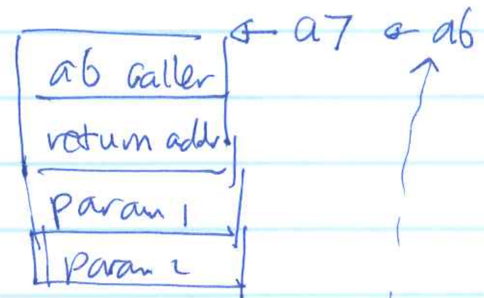
```
move.l #A, -(a7)
move.l #10, -(a7)

bsr ArraySum

adda.l #8, a7
move.l d0, sum
```

ArraySum:

```
movea.l a6, -(a7)
```

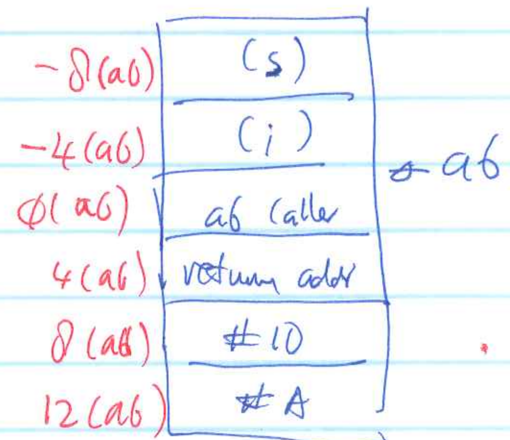
save a6 first

stack now:




```
movea.l a7, a6

suba.l #8, a7.
```

stack now:



rest of program same.

make sure restore a6!

Continued

```
        move.l   #0, -8(a6)        [s=0]
        move.l   #0, -4(a6)        [i=0]

While:
        move.l   -4(a6), d0        [i]
        move.l   -8(a6), d1        [n]

        cmp.l    d1, d0
        BGE      WhileEnd

        movea.l  12(a6), a0
        move.l   -4(a6), d0
        muls     #4, d0
        move.l   (a0, d0.w), d0
        add.l    d0, -8(a6)

        addq.l   #1, -4(a6)

        BRA      While

WhileEnd:
        move.l   -8(a6), d0        return(s)

        movea.l  a6, a7

        movea.l  (a7)+, a6         recover a6 for caller !!!

        RTS.
```

Demo
sub-stack-param4.s