Using the stack to pass parameters and store local variables --- using a *frame pointer* to access variables !!!

- **Solving the "offset problem" by using a *frame pointer* (a6) to access local variables and parameters**
  - **Recall** the **problem** when we use the ***stack* pointer** as **offset** to access the **variables** on the **stack**:

    - When we ***add*** a **new variable** to the **function**, the ***offsets*** of ***all* existing variables *changes***
    - ***all* offsets** used in the **assembler program** will **have to be *updated*** !!!

  - **Solution:**

    We use a **fixed pointer** that points to a relatively fixed location in the stack and use this **fixed pointer** to access the parameters and local variables

  - **Where** must this **fixed pointer** be pointing at ?

    Answer: at the **separation** point between the parameters and local variables

```
        +---------------+ <----- a7 (stack pointer)
-8(a6)  |       s       |  Local variable
        +---------------+
-4(a6)  |       i       |  Local variable
        +---------------+ <----- a6 (frame pointer)
 0(a6)  | return address|
        +---------------+
 4(a6)  |       n  (10) |  Parameter
        +---------------+
 8(a6)  |       A  (#A) |  Parameter
        +---------------+
```

  - **Example:**

Offsets used to access variables **BEFORE** adding extra local variable	Offsets used to access variables **AFTER** adding extra local variable

<pre>+---------------+ <----- a7
-8(a6) | s |
+---------------+
-4(a6) | i |
+---------------+ <----- a6
0(a6) | return address|
+---------------+
4(a6) | n (10) |
+---------------+
8(a6) | A (#A) |
+---------------+</pre>
	<pre>+---------------+ <----- a7
	-12(a6) | x |
	+---------------+
	-8(a6) | s |
	+---------------+
	-4(a6) | i |
	+---------------+ <----- a6
	0(a6) | return address|
	+---------------+
	4(a6) | n (10) |
	+---------------+
	8(a6) | A (#A) |
	+---------------+</pre>

    - The offset of the **existing variables** from **A6** are **UNCHANGED**
    - So the **existing program code** is accessing the existing variables **CORRECTLY** !!!

- **Example:**

```
main:

  int A[10];
  int sum;

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

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

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

   return (s);
}
```

- **"main" in assembler language:**

```
main:

    move.l #A, -(a7)         * Pass address of array A
    move.l #10, -(a7)        * Pass # elements in array

    bsr     ArraySum

    adda.l #8, a7            * remove (pop) #A and #10 from stack

    move.l d0, sum           * put return value in variable "sum"
```

- **Stack content when subroutine "sum" begins execution:**

```
        +---------------+
        | free space    |
        +---------------+ <----- a7 (stack pointer)
 0(a7)  | return address|
        +---------------+
 4(a7)  |      n  (#10) |
        +---------------+
 8(a7)  |      A  (#A)  |
        +---------------+
        |    .....      |
```

- **Subroutine "ArraySum" using a *frame pointer* to access parameter variables and local variables:**

  - Pay special attention to how **A6** is used to access the parameters on the stack !!!
  - **Note:**
    - ***This* version** of the **implementation** is ***not* completely correct**
      - It will ***only*** work if the **main** function **does *not* store** any **important information** in **register A6** !!!
    - I show this version ***only*** to **illustrate *how to*** use **register a6 (called the frame pointer)** to **access** parameter variables and local variables (without you having to worry how to save the

value in A6).

- **BTW**, it's **easy** to fix the problem:
  - **If** the **main** function has some **important information** stored in **A6**, then:
    - the **ArraySum** function **must** ***save*** the register **A6** at the **start**
    - the **ArraySum** function **must** ***restore*** the register **A6** **before** the **ArraySum** function **returns**
- I will **fix** the **problem** in another **implementation** below....

The subroutine returns the value in **register D0**.

```
*********************************************************
* This version does NOT save A6 - we will fix it later  *
*********************************************************

ArraySum:
        movea.l a7, a6          * setup a6 !!!

* The stack is:
*
* Offsets
*          +---------------+ <----- a6 and a7 (stack pointer)
*   0(a6) | return address|
*          +---------------+
*   4(a6) |      n  (10)  |
*          +---------------+
*   8(a6) |      A  (#A)  |
*          +---------------+

        suba.l #8, a7           * Create 2 local variables on stack !!!

* NOW the stack is:
*
* Offsets
*          +---------------+ <----- a7 (stack pointer)
*  -8(a6) |      s        |     (you decide which location is s and i)
*          +---------------+     (This program uses s and i in the given manner)
*  -4(a6) |      i        |
*          +---------------+ <----- a6 (frame pointer)
*   0(a6) | return address|
*          +---------------+
*   4(a6) |      n  (10)  |
*          +---------------+
*   8(a6) |      A  (#A)  |
*          +---------------+

        move.l #0, -8(a6)       * s = 0
        move.l #0, -4(a6)       * i = 0
While:
        move.l -4(a6), d0       * puts local variable i in d0
        move.l 4(a6), d1        * puts parameter n in d1
        cmp.l  d1, d0

        BGE     WhileEnd        * Exit while loop if i >= n

* ---- body of while loop

        movea.l 8(a6), a0       * put base address of array in A0
*                                 (prepare to access A[i])

        move.l  -4(a6), d0      * now d0 = i
```

```
        muls    #4, d0          * offset is now in d0
        move.l  (a0, d0.w), d0  * put A[i] in d0

        add.l   d0, -8(a6)      * add A[i] to local variable s

        move.l  -4(a6), d0
        add.l   #1, d0
        move.l  d0, -4(a6)      * i = i + 1

        BRA While

WhileEnd:

        move.l  -8(a6), d0      * Return s in the agreed location (d0)

* The stack is STILL:
*
* Offsets
*         +---------------+ <----- a7 (stack pointer)
*  -8(a6) |      s        |
*         +---------------+
*  -4(a6) |      i        |
*         +---------------+ <----- a6 (frame pointer)
*   0(a6) | return address|
*         +---------------+
*   4(a6) |      n  (10)  |
*         +---------------+
*   8(a6) |      A  (#A)  |
*         +---------------+
*
* If you return NOW, your program will NOT pop the return address
* into the Program counter and it will CRASH !!!

        movea.l a6, a7      * NEW (better) way to remove local variables !!!
                            * Better you don't need to know how many local
                            * variables to remove !!!

* NOW the stack is:
*
*         +---------------+ <----- a7 (stack pointer)
*         | return address|
*         +---------------+
*         |      n  (10)  |
*         +---------------+
*         |      A  (#A)  |
*         +---------------+
*
* NOW you can rexecute the return instruction !!!

        rts
```

- **Example Program:** (Demo above code) **Example**
  - Prog file: click here

- **Fixing the intentional error: saving the *frame pointer* (a6) of the *caller* subroutine**
  - The subroutine **SumArray** will **use a6 (frame pointer)** to **access *parameter variables* and *local variables***

    Therefore:

    - If the **caller** (in our example: main()) function ***also*** uses **a6** as **frame pointer**, then the **value** stored in **ab** belonging to the **caller function** will be ***destroyed*** by the subroutine **SumArray** !!!!!

- **Fixing** the **problem**:

  1. **Save** the frame pointer **a6 on the stack before** you change its value
  2. **Restore** the **saved a6 value from the stack before** the subroutine returns to the caller !

- **The FINAL form of Subroutine "ArraySum" in assembler language:**

  Pay special attention to how **A6** is SAVED on the stack and RECOVERED **before** subroutine exits

  The subroutine returns the value in **register D0**.

```
ArraySum:
        movea.l a6, -(a7)       ********** Save a6 from the caller subroutine !!!!!!!!!


* The stack is now:
*
*         +---------------+ <----- a7 (stack pointer)
*         | a6 from caller|
*         +---------------+
*         | return address|
*         +---------------+
*         |      n  (10)  |
*         +---------------+
*         |      A  (#A)  |
*         +---------------+

        movea.l a7, a6          * (setup a6 to access local vars and parameters)

* The parameters can now be accessed through a6 as follows:
*
* Offsets
*         +---------------+ <----- a6 and a7 (stack pointer)
*   0(a6) | a6 from caller|
*         +---------------+
*   4(a6) | return address|
*         +---------------+
*   8(a7) |      n  (10)  |
*         +---------------+
*  12(a7) |      A  (#A)  |
*         +---------------+

        suba.l #8, a7           * Create 2 local variables on stack !!!

* NOW the stack is (along with offsets on how to access local variables):
*
* Offsets
*         +---------------+ <----- a7 (stack pointer)
*  -8(a6) |      s        |     (you decide which location is s and i)
*         +---------------+     (This program uses s and i in the given manner)
*  -4(a6) |      i        |
*         +---------------+ <----- a6 (frame pointer)
*   0(a6) | a6 from caller|
*         +---------------+
*   4(a6) | return address|
*         +---------------+
*   8(a6) |      n  (10)  |
*         +---------------+
*  12(a6) |      A  (#A)  |
*         +---------------+

        move.l #0, -8(a6)       * s = 0
        move.l #0, -4(a6)       * i = 0
While:
        move.l -4(a6), d0       * puts local variable i in d0
        move.l 8(a6), d1        * puts parameter n in d1
        cmp.l  d1, d0
```

```
        BGE     WhileEnd        * Exit while loop if i >= n

* ---- body of while loop

        movea.l 12(a6), a0      * put base address of array in A0
*                                 (prepare to access A[i])

        move.l  -4(a6), d0      * now d0 = i
        muls    #4, d0          * offset is now in d0
        move.l  (a0, d0.w), d0  * put A[i] in d0

        add.l   d0, -8(a6)      * add A[i] to local variable s

        move.l  -4(a6), d0
        add.l   #1, d0
        move.l  d0, -4(a6)      * i = i + 1

        BRA While

WhileEnd:

        move.l  -8(a6), d0      * Return s in the agreed location (d0)

* The stack is STILL:
*
* Offsets
*         +---------------+ <----- a7 (stack pointer)
*  -8(a6) |      s        |
*         +---------------+
*  -4(a6) |      i        |
*         +---------------+ <----- a6 (frame pointer)
*   0(a6) | a6 from caller|
*         +---------------+
*   4(a6) | return address|
*         +---------------+
*   8(a6) |      n  (10)  |
*         +---------------+
*  12(a6) |      A  (#A)  |
*         +---------------+
*
* If you return NOW, your program will NOT pop the return address
* into the Program counter and it will CRASH !!!

        movea.l a6, a7      * Remove the local variables.


* NOW the stack is:
*
*         +---------------+ <----- a7 (stack pointer)
*         | a6 from caller|
*         +---------------+
*         | return address|
*         +---------------+
*         |      n  (10)  |
*         +---------------+
*         |      A  (#A)  |
*         +---------------+
*
* NOW is the time to recover the a6 value for the caller subroutine !!!


        movea.l (a7)+, a6   ***************** restore a6   !!!!!!!!!!!!
                            * (The (a7)+ address mode is explained below)

* NOW the stack is:
*
*         +---------------+ <----- a7 (stack pointer)
*         | return address|
*         +---------------+
*         |      n  (10)  |
*         +---------------+
*         |      A  (#A)  |
*         +---------------+
*
* NOW you can rexecute the return instruction !!!
```

```
        rts
```

- Because **popping** values from the **top** of the system stack (to some register or memory variable) is a frequently used operation, M68000 has provided a special **addressing mode** to perform the **pop** operation:

```
move.l   (a7)+, <ea>      is same as:     move.l   (a7), <ea>
                                          adda.l   #4, a7

move.w   (a7)+, <ea>      is same as:     move.w   (a7), <ea>
                                          adda.l   #2, a7
```

  - So when you pop a **long (4 bytes)**, the stack pointer **a7** is incremented by 4.
  - But when you pop a **word (2 bytes)**, the stack pointer **a7** is incremented by 2 !!!
  - This address mode is called "indirect with **post-increment**"

- **Example Program:** (Demo above code) *Example*
  - Prog file: click here
