Tower of Hanoi

- **Introduction:**
  - Hands on "Tower of Hanoi": click here or click here

- Again, I want to show you more than just implementing the recursive solution for Hanoi in assembler.

  Here is my CS170 webpage that explain the important of **pre-conditioning** in the Tower of Hanoi problem: click here

- **The TowerOfHanoi function**
  - The TowerOfHanoi function can be written explicitly as follows:

```
void hanoi(int ndisks, int fromPeg, int toPeg)
{
   int helpPeg;
   if (ndisks == 1) then
      WriteLn "move disk from peg " + fromPeg + " to " + " + toPeg
   else
   {
      helpPeg := 6 - fromPeg - toPeg;
      hanoi(ndisks-1, fromPeg, helpPeg);
      WriteLn "move disk from peg " + fromPeg + " to " + " + toPeg
      hanoi(ndisks-1, helpPeg, toPeg);
   }
}
```

  - The **hanoi** function is called with the following statement:

```
hanoi(n, 1, 3);
```

    to move **n** disks from peg 1 to peg 3.

- **The stack frame structure for the *hanoi* function**
  - The stack frame structure created will looks as follows:

```
+---------------------+ <------------ Stack pointer (A7)
|   use for helpPeg   |
+---------------------+ <------------ Frame Pointer (A6)
| Saved Frame Pointer |
+---------------------+
|  Return Address     |
+---------------------+
| use for ndisks      |
+---------------------+
| use for fromPeg     |
+---------------------+
| use for toPeg       |
+---------------------+
|     .......         |
```

```
|  rest of the stack  |
|     .......         |
```

- **The *hanoi* program:**
  - The complete example can be found in the following assembler program file: click here
  - You may want to get the following Debug file for EGTAPI to use with it: click here
  - I will highlight certain steps in the program in the remainder of the webpage....

- **Passing parameters from main program to hanoi**

The main program passes the parameter n to **hanoi** by pushing N, 1, and 3 in the reverse order onto the system stack with the following instructions:

```
move.l #3,-(a7)          ; Push toPeg
move.l #1,-(a7)          ; Push fromPeg
move.l N,-(a7)           ; Push ndisks
```

This will create the following stack structure:

```
+--------------------+ <------------ Stack pointer (A7)
|     parameter N    |
+--------------------+
|     parameter 1    |
+--------------------+
|     parameter 3    |
+--------------------+
|     .......        |
|  rest of the stack |
|     .......        |
```

- **How the main program calls the *hanoi* function**

The main program calls the **hanoi** function with a **bsr** instruction:

```
pass parameters (see above)
bsr  hanoi
```

This will create the following stack structure:

```
+--------------------+ <------------ Stack pointer (A7)
|     return address |
+--------------------+
|     parameter N    |
+--------------------+
|     parameter 1    |
+--------------------+
|     parameter 3    |
+--------------------+
|     .......        |
|  rest of the stack |
|     .......        |
```

- **Prelude of the *hanoi* function:**

The prelude of the **hanoi** function consists of these 3 instructions:

```
****************************** PRELUDE
        move.l a6, -(a7)    ; Save caller's frame pointer
        move.l a7, a6       ; Setup my own frame pointer
        suba.l #4, a7       ; Allocate space for the local valiable 'helpPeg'
******************************
```

I will explain what each one does below. Make sure that you realise that the structure of the stack frame is like this when the prelude is **always** executed:

```
+--------------------+ <------------ Stack pointer (A7)
|     return address  |
+--------------------+
|     parameter N     |
+--------------------+
|     parameter 1     |
+--------------------+
|     parameter 3     |
+--------------------+
|     .......         |
|  rest of the stack  |
|     .......         |
```

- move.l a6, -(a7)

  This will save the frame pointer on the stack, creating this partial stack frame structure:

```
+--------------------+ <------------ Stack pointer (A7)
|     saved a6        |
+--------------------+
|     return address  |
+--------------------+
|     parameter N     |
+--------------------+
|     parameter 1     |
+--------------------+
|     parameter 3     |
+--------------------+
|     .......         |
|  rest of the stack  |
|     .......         |
```

- move.l a7, a6

  This will make the frame pointer A6 points to the stack frame that is now being built:

```
+--------------------+ <---- Frame pointer A6 & Stack pointer (A7)
|     saved a6        |       point to the same location....
+--------------------+
|     return address  |
+--------------------+
|     parameter N     |
+--------------------+
|     parameter 1     |
+--------------------+
|     parameter 3     |
+--------------------+
|     .......         |
|  rest of the stack  |
|     .......         |
```

- suba.l #4, a7

  This will push the stack pointer A7 8 bytes up, allocating 1 integer variable. This variable will be used is for helpPeg.

```
+---------------------+ <---- Stack pointer (A7)
|     helpPeg         |
+---------------------+ <---- Frame pointer (A6)
|     saved a6        |
+---------------------+
|     return address  |
+---------------------+
|     parameter N     |
+---------------------+
|     parameter 1     |
+---------------------+
|     parameter 3     |
+---------------------+
|     .......         |
|  rest of the stack  |
|     .......         |
```

- When the prelude is finish, the stack frame is complete and the actual function can begin.

- **How to access the parameter and the local variables in *Hanoi*:**

  - From the **stack** strcuture:

```
        +---------------------+ <---- Stack pointer (A7)
-4(a6)  |     helpPeg         |
        +---------------------+ <---- Frame pointer (A6)
 0(a6)  |     saved a6        |
        +---------------------+
        |     return address  |
        +---------------------+
 8(a6)  |     param NDisks    |
        +---------------------+
12(a6)  |     param fromPeg   |
        +---------------------+
16(a6)  |     param toPeg     |
        +---------------------+
        |     .......         |
        |  rest of the stack  |
        |     .......         |
```

  - Parameter ndisks is located 8 bytes **below** starting from the address contained in the frame pointer A6.

    So the address mode that will let you get to this variable is 8(A6)

  - Parameter fromPeg is located 12 bytes **below** starting from the address contained in the frame pointer A6.

So the address mode that will let you get to this variable is 12(A6)

- Parameter toPeg is located 16 bytes **below** starting from the address contained in the frame pointer A6.

  So the address mode that will let you get to this variable is 16(A6)

- Local variable helpPeg is located 4 bytes **above** starting from the address contained in the frame pointer A6.

  So the address mode that will let you get to this variable is -4(A6)

- Use the above way to gain access to the **private copy of parameters and local variables** of **each function invocation**....

- **How *Hanoi* calls itself:**

It is no different from how the main program calls the Hanoi function. Simply push the parameter **in the proper order** on the stack, and call Hanoi.

But **make sure** you **pop the parameter** from the stack after Hanoi returns - because the parameter has not been cleaned up.

The following is the program fragment where Hanoi calls hanoi(ndisks-1, fromPeg, thirdPeg):

```
move.l -4(a6), -(a7)      ; Push and pass toPeg
move.l 12(a6), -(a7)      ; Push and pass fromPeg
move.l 8(a6), d0          ; d0 = ndisks
sub.l #1, d0              ; d0 = ndisks-1
move.l d0, -(a7)          ; Push and pass ndisks-1
bsr hanoi                 ;
adda.l #12,a7             ; Pop parameters (3 ints) off stack
```

Hanoi will call itself a second time with hanoi(ndisks-1, thirdPeg, toPeg); The following is the program fragment where Hanoi calls hanoi(ndisks-1, thirdPeg, toPeg):

```
move.l 16(a6), -(a7)      ; Push and pass toPeg
move.l -4(a6), -(a7)      ; Push and pass thirdPeg
move.l 8(a6), d0          ; Get ndisks
sub.l #1, d0              ; d0 = ndisks-1
move.l d0, -(a7)          ; Push and pass ndisks-1
bsr hanoi                 ;
adda.l #12,a7              ; Pop parameters (3 int) off stack
```

- **Help Subroutine: WriteLn**

  - In the Hanoi subroutine, we need to print out a string, e.g.:

    ```
    WriteLn "move disk from peg " + fromPeg + " to " + " + toPeg
    ```

  - I have provided a number of helpful subroutines in a library that is linked into the assembler

program compiled with the command **as255**.

One of these helpful subroutine is **WriteLn** that prints a string stored in memory (where else ?) to an output file.

- How to use a library subroutine (like **WriteLn**) in assembler:

  1. You must define the subroutine name as "external" using the "xref" assembler directive:

     ```
     xref WriteLn
     ```

     It tells the assembler that the name "WriteLn" will be supplied by another source file (the library file linked with the program)

  2. Use **jsr WriteLn** when you call a library subroutine.

     JSR is similar to **BSR**, except it jumps farther away. (BSR is limited to a location that is < 32 Kbytes from the current program location)

- Beside this, you need to know **WHAT parameters TO PASS** to **WriteLn** and WHAT IT RETURNS. The parameters to **WriteLn** are:

  - A0 = starting address of the string in memory
  - D0 = the length of the string (number of bytes)
  - WriteLn does not return any value

- **Sample program:**

```
* Demo the use of WriteLn
*
        xdef Start, Stop, End
        xref WriteLn
*
Start:
*                                   ; Print the text message
        move.l #Text, a0            ; Location of text
        move.l #(EndText-Text),d0   ; Length of text
        jsr    WriteLn

Stop:
            nop

Text:    dc.b   'Hello World !'
EndText: dc.b   ' '

End:
        end
```

- **Example Program:** (Demo above code)

  - Prog file: click here

How to run the program:

- **Right click** on link and **save** in a scratch directory
- To compile: `as255 WriteLn`
- To run: use **m68000**