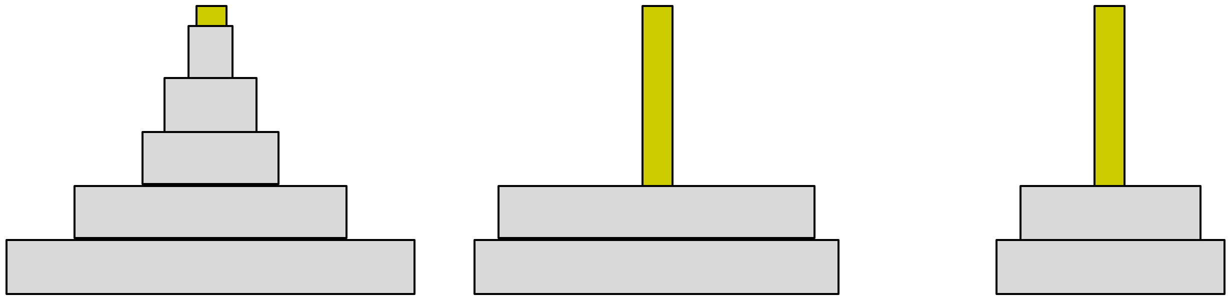
Lab 07 – Tower of Hanoi

R. Ferrero, A. C. Marceddu,
M. Russo

Politecnico di Torino
Dipartimento di Automatica e Informatica (DAUIN)
Torino - Italy

Tower of Hanoi

- The tower of Hanoi consists of 3 poles.
- Each pole contains a certain number of discs (possibly zero), arranged in descending order of size (the largest disc is the lowest).



Data structures

- Each pole can be represented across a stack.
- Each value in the stack indicates the size of a disc contained in the corresponding pole.
- The 3 stacks are defined in a READWRITE area: the 3 stack pointers point to different memory locations within the data area.
- Freedom of choice is left to manage the 3 stacks as full descending or empty ascending*

Data structures (cont.)

- The 3 stacks are not the main stack, just 3 different areas of memory that we will handle like a stack. Therefore, their respective stack pointers are not SP!
- 5 words for each stack is enough for our purpose
- *Remember that the main stack is full descending, so working with FD operations with the other 3 stacks is suggested to avoid confusion in your reasoning

Exercise 1 – Poles initialization

- Write a `fillStack` subroutine to initialize the stack associated to a pole.
- The subroutine receives two parameters:
  - the stack pointer (not SP) associated to the pole
  - the address to a READONLY area (for instance code area) containing a sequence of constants.
- The subroutine updates the two parameters:
  - the stack pointer points to the last entered data
  - the address in the READONLY area is subsequent to the last constant entered.

Exercise 1 (cont.)

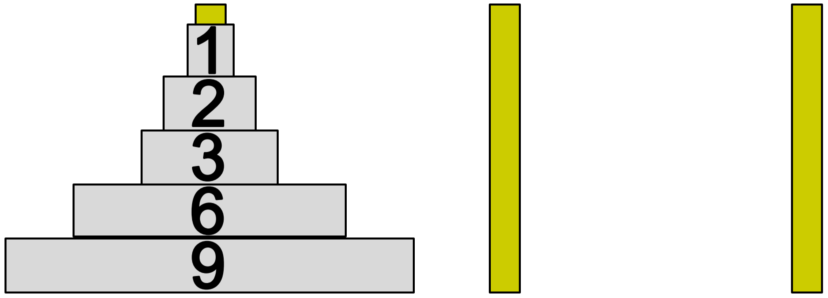
- The parameters are passed to the subroutine through the main stack (`SP`).
- The insertion ends when:
  - either the constant to be entered is 0
  - or the constant to be inserted is greater than the last element on the stack.

Exercise 1: example

- `r1` is the stack pointer associated with the first pole
- The sequence of constants is

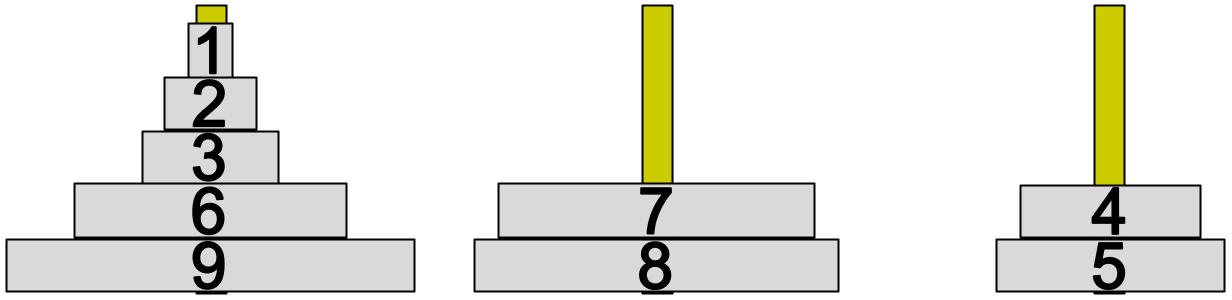
```
DCD 9, 6, 3, 2, 1, 8, 7, 0, 5, 4, 0
```

- `r0` contains the address of the first constant (9).
- After calling `fillStack` once, the situation is
  - $\texttt{r1}_{\texttt{new}} = \texttt{r1}_{\texttt{old}} \pm 20$
  - $\texttt{r0}_{\texttt{new}} = \texttt{r0}_{\texttt{old}} + 20$



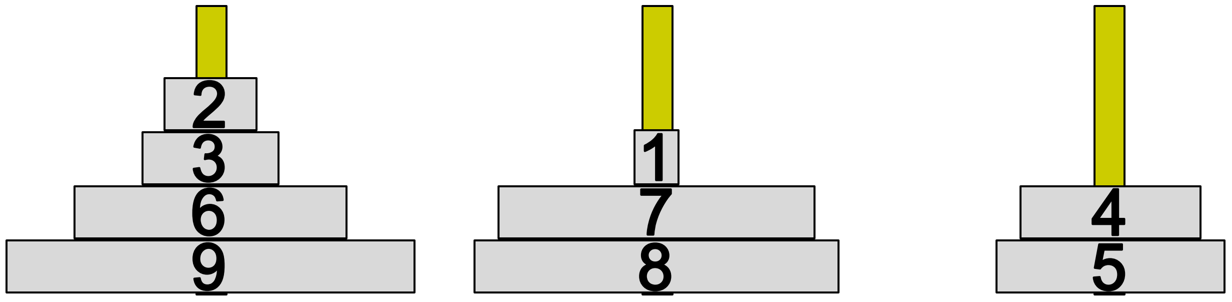
Exercise 1: example

- With three consecutive calls to `fillStack` (one for each pole) the tower of Hanoi becomes:



Exercise 2 – Single movement

- It is possible to move a disc from one pole to another if:
  - either the new pole is empty
  - or the upper disc in the new pole has a larger size than the disc to be moved.
- Example: a disc is moved from pole 1 to pole 2



Exercise 2 (cont.)

- Write a `move1` subroutine that handles moving a disk.
- `move1` receives 3 parameters across the stack
  - stack pointer of the starting pole
  - stack pointer of the destination pole
  - space for the return value
- `move1` returns 1 if it was possible to move the disk, 0 otherwise. Stack pointers are updated if the disk has been moved.

Exercise 2: warning

- To pass parameters to `move1`, it is preferable to use 3 PUSH instead of just one.
- Example: let `r1` be the stack pointer of the starting pole, `r2` the stack pointer of the destination pole, `r0` contain the return value.

-
  ```
  PUSH {r1, r2, r0}
  BL move1
  POP {r1, r2, r0}
  ```

  moves from pole `r2` to pole `r1`, because the registers are reordered as `{r2, r1, r0}`

Exercise 2: warning

- To ensure the movement from pole `r1` to pole `r2`, the following code can be used:

```
    PUSH {r1}
    PUSH {r2}
    PUSH {r0}
    BL move1
    POP {r0}
    POP {r2}
    POP {r1}
```

Exercise 3 – Multiple movements

- It is possible to move N discs from pole X to pole Y using a recursive procedure:
  - Move the first N-1 discs from X to Z
  - Move disk N from X to Y
  - Move N-1 discs from Z to Y

Exercise 3 (cont.)

- Write a `moveN` subroutine that manages the movement of N disks.
- `moveN` receives across the stack:
  - the stack pointer of the starting pole X
  - the stack pointer of the destination pole Y
  - the stack pointer of the auxiliary pole Z
  - the number of discs to move
- `moveN` updates the stack pointers and replaces the fourth parameter with the number of movements performed.

Pseudocode of moveN(X, Y, Z, N)

```
/* M is the number of mov. made*/
M = 0;
if (N == 1)
{
    move1(X, Y, a);
/*a is the return value: 0-1 */
    M = M + a;
}
```

Pseudocode of moveN(X, Y, Z, N)

```
else {
    moveN(X, Z, Y, N-1);
    M = M + b;
    move1(X, Y, a);
    if (a == 0) return;
    else M = M + 1;
    moveN(Z, Y, X, N-1);
    M = M + c;
}
```

Exercise 3: suggestions

- Since each recursive call adds its own parameters to the stack, it is suggested to increase the stack size by changing the value of `Stack_Size`.
- As a final check, considering a tower of Hanoi with N discs in one pole and 0 in the other two, note that the number of movements required to move the N discs into another pole is equal to $2^N - 1$.

Possible strategy for procedure calling

- In the caller of **f**, you can push before (and pop after) calling **f** just the registers containing the arguments and the return value*
- In the called function **f**, you can furtherly push at the beginning all the registers used by the function + LR, and, at the end of **f**, you can pop all those registers + PC to restore them and go back to the caller

*The corresponding values should not be popped by **f**, but only read and written using [sp,#…]

Possible strategy for procedure calling (cont.)

- This helps you conceptually not to get lost: the only job of **f** becomes to read the arguments and write into the space that the caller set up in the stack, and also to backup all the registers that it uses and to restore them after
- At the same time, the only job of the caller becomes to set up (PUSH) the arguments and some space for the return value, and then to read the results (POP). Note: the symmetry of PUSH/POP implies some redundancy (the caller will read back the arguments)

Procedure calling: warning

- Remember: the stack pointer must be the same before and after calling **f**!