Architetture dei Sistemi di Elaborazione **Laboratory 7**	Delivery date: Friday 3/12 Expected delivery of lab_07.zip must include: - zipped project folder of the exercises 1 and 2 - this document compiled possibly in pdf format.

Eurovision 2022 in Turin!

Exercise 1)
The **Eurovision Song Contest** 2022 will be held at the PalaOlimpico in **Turin**, Italy, following the country's victory at the 2021 contest with the song "Zitti e buoni" by *Måneskin*.
Write a program in **ARM assembly** language to manage the sale of contest tickets. PalaOlimpico is organized in different sectors with different prices. For example, you have the following lists to be initialized in a pool:

```
Sector_prices DCD 0x01, 25, 0x02, 40, 0x03, 55, 0x04, 65, 0x05, 80
              DCD 0x06, 110

Sector_quantity DCD 0x02, 250, 0x05, 250, 0x03, 550, 0x01, 150, 0x04,
                DCD 100, 0x06, 200

Num_sectors DCB 6
```

Sector_prices is a table where each entry consists of two integer values: the ID of the sector (4 bytes) and the price of each ticket in that sector (4 bytes).
Sector_quantity is a table where each entry consists of two integer values: the ID of the sector (4 bytes) and the number of places available in that sector (4 bytes).
Num_sectors is a 1 byte constant and indicates the number of sectors available in PalaOlimpico.
Write a program to respond to a purchase request.
The request is stored in the following pool, where you have a set of 2 items: the sector ID (hexadecimal) and the whished quantity. The variable `Tickets_requests` stores the amount of ticket requests.
For instance, in the following example the user wishes to buy tickets from three different sectors:

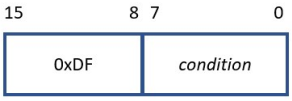
```
Tickets DCD 0x05, 2, 0x03, 10, 0x01, 120
Ticket_requests DCD 3
```

If the tickets are available, update the `Sector_quantity` and store the total cost of the purchase in a 4-byte variable stored in RAM, named *total_tickets*; otherwise, if the sector is sold out (or the desired quantity is not available), store zero in the same variable, and 0x01 in R11 to underline that the procedure has not been completed.
Moreover, if the desired number of tickets is greater than 10, apply a 50% group discount for **Black Friday.**

Exercise 2) Experiment the SVC instruction.

Write an ARM assembly program that invokes an `SVC` instruction when running a user routine with unprivileged access level.
By means of invoking a SuperVisor Call, we want to implement the squared power ($\mathbf{x^2}$) or the **integer approximation of the square root** of a number ($\lfloor\sqrt{x}\rfloor$). The approximation of the square root is calculated by iterating on all natural numbers *n* in ascending order until the following condition is satisfied: $\boldsymbol{n^2 \leq x}$. The value of **x** is stored in R0.

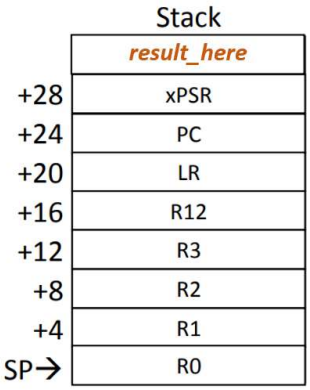
The SVC instruction is encoded as follows:



- Bits [15:8]: Opcode of the SVC Assembly ARM Instruction
- Bits [7:0]: This field indicates the operation that must be performed, according to its content.
    - If the content is equal to 0, the squared power must be done,
    - Else if the content is equal to 1, the integer approximation of the square root must be done.
    - Else, NOP operations.

> ***Example:*** *SVC 1 and R0=0x11*
> *Your algorithm must return 4.*

The result of your code must be saved in the PSP and returned as specified in the figure below. Then, outside the *SVC_handler*, save the result in a 4-byte variable *SQResult*.



Q1: Describe how the stack structure is used by your project and which stack you are using and why.

The stack structure is Full Descending (like the above picture).
I'm using the PSP stack because the request tells us to use user routine with unprivileged access.

Q2: What needs to be changed in the SVC handler if the access level of the caller is privileged? Please report the code chunk that satisfies this request.

If the caller is privileged and using PSP, nothing must be changed. (CONTROL = 0x10)

If the caller is privileged and using only MSP, the code will look like this:

```
SVC_Handler     PROC
                EXPORT  SVC_Handler

1               STMFD   SP, {R0-R12, LR}
2               MRS     R1, msp
                LDR     R2, [R1, #24]    ;get
                LDR     R2, [R2,#-4]     ;get
                BIC     R2, #0xFF000000 ;get
                LSR     R2, #16          ;shi

                ; your code here
                CMP     R2, #0
                MULEQ   R3, R0, R0
                STREQ   R3, [R1, #32]
                BEQ     uscita
                NOPLT
                BLT     uscita

                MOV     R3, #0
loop            CMP     R2, #1
                NOPGT
                BGT     uscita
                MOVEQ   R4, R3
                ADDEQ   R3, R3, #1
                MULEQ   R5, R3, R3
                CMPEQ   R5, R0
                STREQ   R3, [R1, #32]
                BEQ     uscita
                BLT     loop
                STRGT   R4, [R1, #32]

uscita      3   SUB     R1, R1, #56
                MOV     SP, R1
                LDMFD   SP!, {R0-R12, LR}
                BX      LR
```

Changes:
1 – when pushing registers at the beginning (STMFD instruction) don't update the SP using the ! operator.
2 – We need a reference to the MSP, not the PSP like before (MRS instruction). This can also be omitted and just use SP instead of R1 in the first LDR instruction.
3 – before the pop (LDMFD instruction) we need to move the SP to the right point, so we subtract from it 4*13=56 (13 is the number of registers we stored at the beginning).