Computer Architectures 02LSEOV **Laboratory 6**	Delivery date: November 22th 2025, 11.59 PM Expected delivery of lab_06.zip must include: - Solutions of exercises 1, 3, and 4. - This file, filled with information and possibly compiled in a pdf format.

This lab will continue to delve into ARM programming.

1) Write a program using the ARM assembly that performs the following operations:
   1. Initialize registers *R3*, *R4*, and *R5* with arbitrary (chosen by you) signed values.
   2. Subtract *R3* to *R4* (*R4 – R3*) and store the result in *R6*.
   3. Sum *R4* to *R5* (*R4 + R5*) and store the result in *R7*.
2) Using the debug log window, change the values of the written program in order to set the following flags to 1, one at a time and when possible:
   - carry
   - overflow
   - negative
   - zero

   Report the selected values in the table below:

Updated flag	Hexadecimal representation of the obtained values			
	R4 – R3		R4 + R5	
	R4	R3	R4	R5
Carry = 1	0xFFFFFFFF	0x80000000	0xFFFFFFFF	0x00000002
Carry = 0	0x00000000	0x80000001	0x00000000	0x00000001
Overflow	0x7FFFFFFF	0x80000000	0x7FFFFFFF	0x00000001
Negative	0x00000003	0x00000005	0x00000003	0xFFFFFFFC
Zero	0x00000000	0x00000000	0x00000000	0x00000000

Please explain the cases where it is **not** possible to force a **single** FLAG condition:

Nel caso del flag Zero, per la sottrazione R4 – R3 la condizione Z = 1 implica necessariamente R4 = R3 e quindi assenza del prestito, cioè C = 1. Di conseguenza non è posibile avere Z = 1 da solo dopo una sottrazione.
Nel caso del flag Overflow, in aritmetica two's complement l'overflow si verifica solo quando gli operandi hanno lo stesso segno e il risultato cambia segno. Ciò comporta che, se V = 1, almeno uno tra N e C (e talvolta anche Z) risulta anch'esso impostato a 1. Non esiste quindi una combinazione di operandi che attivi soltanto il flag V mantenendo tutti gli altri

3) Write two versions of a program that performs the following operations:
   1. Initialize registers *R1* and *R2* with arbitrary (chosen by you) signed values.
   2. Compare the two registers:
      - If they differ, store in register *R8* the maximum among *R1* and *R2*.
      - Otherwise, perform a logical right shift of 1 on *R1* (is it equivalent to what?), then subtract this value from *R2* and store the result in *R9* (i.e., *R4 = R2 – (R1 >> 1)*).

   The first version should be based on a traditional assembly programming approach using conditional branches, while the second version should be based on a conditional statement execution approach.
   Considering a CPU clock frequency (clk) of *16 MHz*, report the number of clock cycles (cc) and the simulation time in milliseconds (ms) in the following table:

	R2 == R3 [cc]	R2 == R3 [ms]	R2 != R3 [cc]	R2 != R3 [ms]
1) Traditional	11	0.000733	13	0.000867
2) Conditional Execution	12	0.000750	12	0.000750

Note: you can change the CPU clock frequency by following the brief guide at the end of the document.

4) Write a program that calculates the parity bit of a variable. The parity bit is used for error detection and indicates whether the number of 1s in the binary data is odd (parity = 1) or even (parity = 0), calculated by performing an XOR operation on all bits of the variable: for example, the parity of 0b00000101 is 0 (even parity) because it has two 1s. The variable to be checked is in R10 that you have to initialize to 0b00110101. After calculating the parity, store the result (0 or 1) in R13.

   To recap, implement ASM code that does the following:
   - Calculate the parity bit of R10 by XORing all its bits together.
   - Store the parity bit result in R13 (0 for even parity, 1 for odd parity).

   a) Assuming a *15 MHz* clk, report the code size and execution time in the following table:

Code size [Bytes]	Execution time
28	15.200 us

   b) Add some comments regarding the solution you wrote:

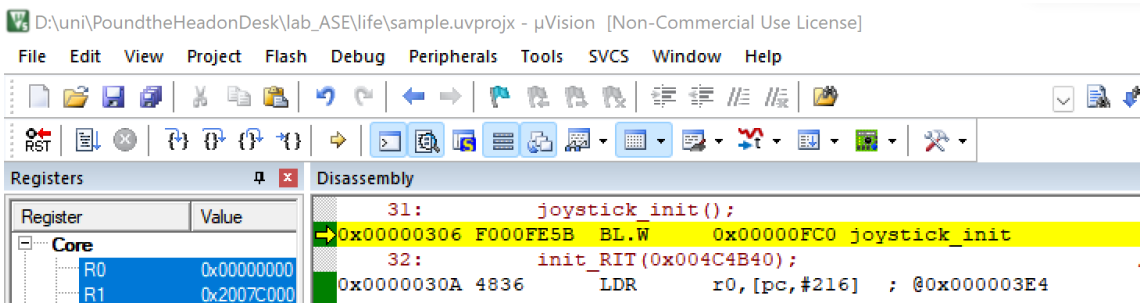
Il programma implementa il calcolo del bit di parità con un approccio iterativo: viene analizzato il bit meno significativo di R10, accumulando la parità tramite XOR in un registro di lavoro e poi shiftato a destra il valore fino a esaminare tutti i bit. La logica è semplice, lineare e facilmente estensibile ad altri valori di ingresso senza modificare la struttura del codice; alla fine il risultato viene copiato in R13 come richiesto dalla traccia.

La code size totale del programma è pari a 28 byte (indirizzo finale 0x000000E6, indirizzo iniziale 0x000000CC, distanza 0x1A = 26 byte, a cui si aggiungono 2 byte dell'ultima istruzione). Questo è un valore contenuto, ottenuto grazie all'uso di un unico loop che riutilizza le stesse istruzioni per tutti i bit. In altre parole, si privilegia la compattezza del codice rispetto a soluzioni "unrolled" o basate su sequenze di istruzioni senza branch, che potrebbero ridurre il tempo di esecuzione ma a costo di una code size maggiore.

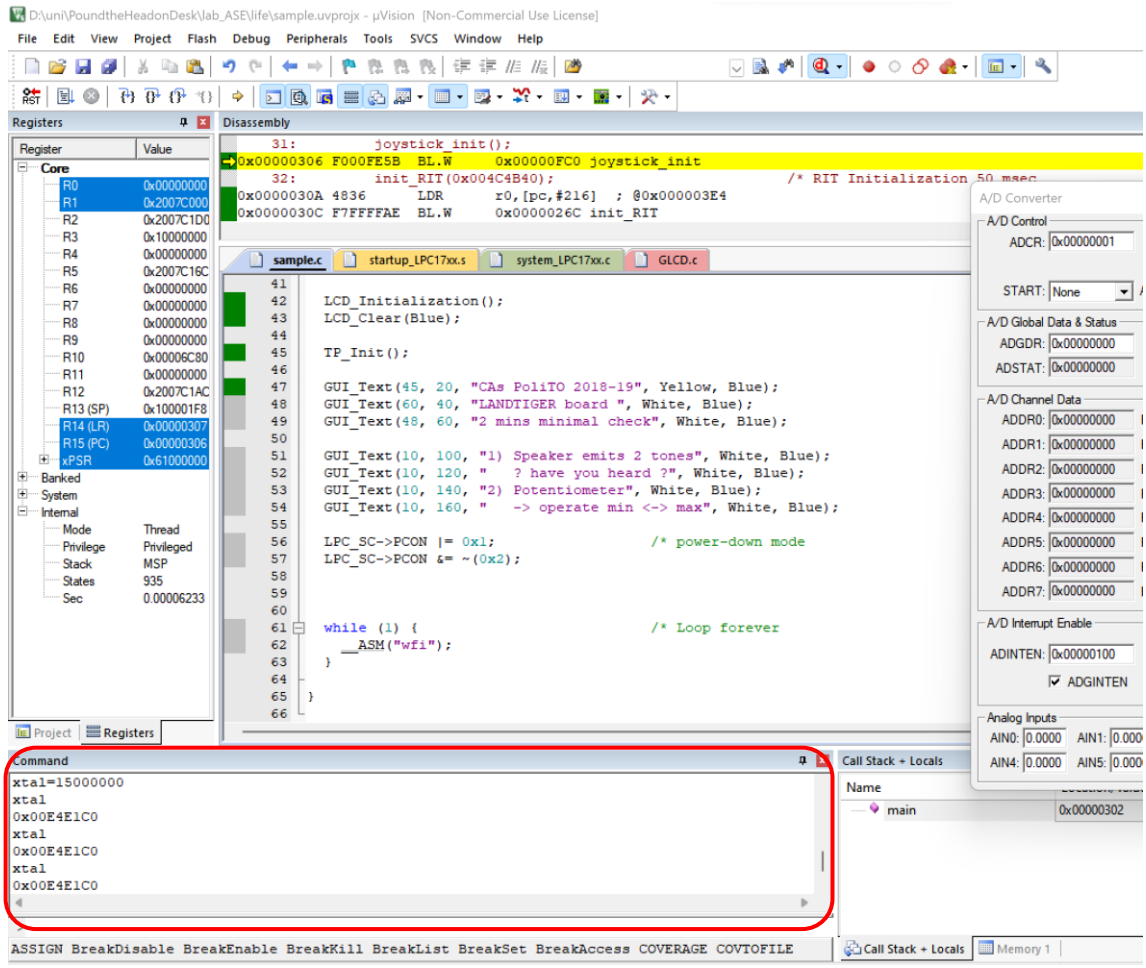
Con una frequenza di clock di 15 MHz, il tempo di esecuzione misurato è di 15.200 µs, che corrispondono a circa 228 cicli di clock. Considerando che il loop attraversa 32 iterazioni, il costo medio per bit è di pochi cicli, ma la presenza del branch nel loop e il fatto di processare tutti i 32 bit (anche quelli alti che in questo caso sono a zero) introduce un overhead non trascurabile rispetto a soluzioni più ottimizzate. Tuttavia, per un singolo calcolo di parità in un contesto didattico o non critico in termini di performance real-time, il trade-off è accettabile: il programma rimane facile da leggere, da spiegare e da mantenere, pur rispettando i vincoli richiesti sull'uso dei registri e fornendo un tempo di esecuzione comunque molto ridotto in assoluto.

How to set the CPU clock frequency in Keil

1) Launch the debug mode and activate the command console.



2) A window will appear:



You can type *xtal* to check its value. To change its value, make a routine assignment, i.e., *xtal=frequency*, keeping in mind that frequency is in Hz must be entered. To set a frequency of *15 MHz*, you must write as follows: *xtal=15000000*.