z1. Implementazione

- ➪ **ambiente sw/hw di riferimento**
  - insieme software e hardware che costituisce l'ambiente di sviluppo del progetto

File di Progetto

1. ***cfs32.nasm e cfs64.nasm*:** Questi sono file di codice assembly scritti rispettivamente per le architetture x86-32 e x86-64. Conterranno l'implementazione dell'algoritmo di selezione delle caratteristiche ottimizzato in linguaggio assembly per le due diverse architetture.
2. ***cfs32c.c e cfs64c.c*:** Sono file di codice sorgente in linguaggio C scritti per le architetture x86-32 e x86-64. Questi file includono parti dell'algoritmo implementate in C che richiederanno ottimizzazioni.
3. ***sseutils32.nasm e sseutils64.nasm:*** Questi file contengono funzioni di utilità o macro specifiche per ottimizzazioni SSE per le architetture x86-32 e x86-64. Spesso vengono utilizzati per ottimizzare operazioni specifiche che sfruttano le istruzioni SIMD.

- ➪ per generare il file eseguibile viene specificata la seguente istruzione:

```
nasm -f elf32 att32.nasm && gcc -m32 -msse -O0 -no-pie sseutils32.o att32.o att32c.c -o att32c -lm && ./att32c $pars
```

Questa sequenza di comandi genera un file eseguibile a partire da codice sorgente Assembly e C.

1. `nasm -f elf32 att32.nasm`: Questo comando assembla un file sorgente Assembly (att32.nasm) e produce un file oggetto in formato ELF a 32 bit (att32.o). `-f elf32` specifica il formato di output del file oggetto.
2. `gcc -m32 -msse -O0 -no-pie sseutils32.o att32.o att32c.c -o att32c -lm`: Questo comando compila il codice C (att32c.c) insieme a file oggetto preesistenti (sseutils32.o, att32.o) per generare un eseguibile a 32 bit chiamato "att32c".
   - `-m32`: Indica di compilare per un'architettura a 32 bit.
   - `-msse`: Abilita le estensioni SSE (Streaming SIMD Extensions).
   - `-O0`: Disabilita l'ottimizzazione del compilatore.
   - `-no-pie`: Disabilita l'eseguibile a posizione indipendente.
   - `-lm`: Linker con la libreria matematica.
3. `/att32c $pars`: Esegue l'eseguibile appena compilato (att32c) passando eventuali parametri dalla riga di comando (indicati con `$$pars`).

- adattando in base ai nomi dei file che abbiamo:

`nasm -f elf32 sseutils32.nasm` → genera file `sseutils32.o`

`nasm -f elf32 cfs32.nasm && gcc -m32 -msse -O0 -no-pie sseutils32.o cfs32.o cfs32c.c -o cfs32c -lm`

- per eseguire
  `./cfs32c $pars`

  > verrà richiesto di specificare i parametri

`gcc -m32 -msse -O0 -no-pie cfs32c_senza_commenti.c -o cfs32c_senza_commenti`

`./cfs32c_senza_commenti -ds esempio_dataset.ds2 -labels labels.ds2 -k 3`

- ➪ Per fare funzionare il programma dobbiamo scrivere un dataset.ds ed un vettore label.ds in binario
  - altrimenti avremo errore di segmentazione (ostacola anche il debug)
- *scriviMatrice*
  - per avere il dataset di esempio in *esempio_dataset.ds2*
- *scriviMatrice2*
  - per avere il dataset di esempio in *labels.ds2*

- ➪ è possibile iniziare a sviluppare la soluzione ad alto livello interamente in linguaggio C
  - modificheremo la funzione *cfs*
- idea su come strutturare il lavoro:
  - *cfs32c.c* → file principale → contenente la funzione cfs principale
  - *correlation.h* → header in cui definiamo le signatures delle nostre funzioni
  - *correlation.c* → file in cui includeremo l'implementazione estesa delle nostre funzioni dichiarate in *correlation.h*
- esempio su come effettuare questo collegamento

correlation.h

```
#ifndef CORRELATION_H
#define CORRELATION_H
void stampaValore();
#endif /* CORRELATION_H */
```

correlation.c

```
#include correlation.h
#include <stdio.h>
void stampaValore(){
    printf("Prova da altro file");
}
```

cfs32c_senza_commenti.c

```
#include "correlation.h"
void cfs(params* input){
    //proviamo ad invocare la funzione da un altro file
    stampaValore();
}
```

per compilare

```
gcc cfs32c_senza_commenti.c correlation.c -o cfs32c_senza_commenti
```
