UNIVERSITÀ DI PISA

Dipartimento di Ingegneria dell'Informazione

Laurea Magistrale in Ingegneria Elettronica

Led Sandbox

Professori:

Prof: Federico Baronti

Studenti:

Leonardo Bove

Alessandro Palla

Filippo Passarella

Abstract

Il Led Sandbox è un'architettura progettata per replicare il comportamento di granelli di sabbia, visualizzati tramite una matrice di LED RGB. Il sistema sfrutta un accelerometro integrato per rilevare l'inclinazione, mentre l'elaborazione dei dati è affidata al soft-core Nios II, istanziato sulla FPGA DE10-Lite. L'interazione tra sensore e display consente di simulare dinamicamente il movimento dei granelli in risposta ai cambiamenti di orientamento della board.

Una possibile integrazione futura sarà inserire una seconda modalità di funzionamento, che elaborando con l'IA i dati forniti da una webcam, permetterà di muovere la sabbia attraverso i gesti della mano.

Indice

Capitolo 1

Introduzione

I file sorgente di questo progetto possono essere trovati su GitHub

1.1 Obiettivi di progetto

In questo progetto intendiamo realizzare un simulatore di una **sabbiera**, i cui granelli di sabbia si muovono concordi alla direzione della forza di gravità. L'inclinazione della teca verrà rilevata tramite un **accelerometro**, i cui dati guideranno la simulazione del comportamento dei granelli, riprodotti visivamente su una **matrice LED**.
Il cuore del progetto risiede nella programmazione in linguaggio C, che gestisce l'elaborazione dei dati provenienti dall'accelerometro e controlla la visualizzazione sulla matrice LED, il tutto gestito da uno **scheduler**. Il codice C sarà eseguito su un processore Nios II fast, integrato all'interno della FPGA grazie al supporto del tool **Platform Designer**.

1.2 Requirements

Per lo sviluppo del progetto sono state necessarie le seguenti componenti hardware e software:

1.2.1 Hardware

Le componenti utilizzate sono:

- FPGA DE10-Lite
- Adafruit 64x32 RGB LED MI-T21P4RGBE-AD
- Alimentatore da ATX per alimentare la matrice.

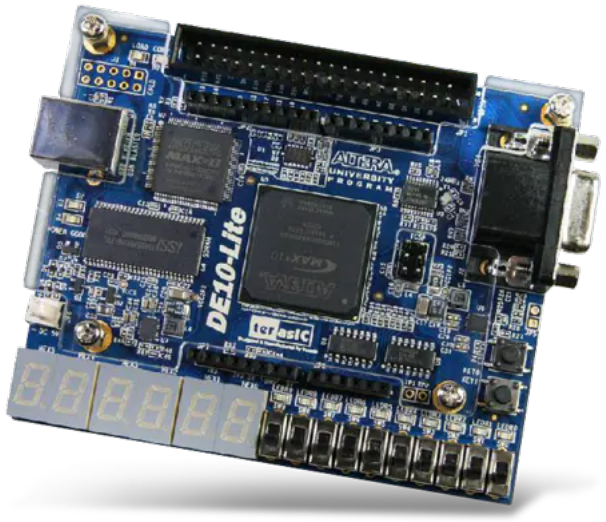


Figura 1.1: FPGA DE10 Lite

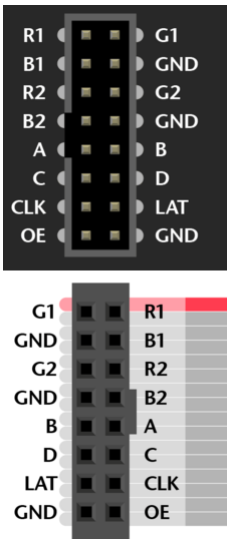
Figura 1.2: LED Matrix

1.2.2 Software

La configurazione della FPGA è stata effettuata utilizzando il software **Quartus Prime**, con il supporto del tool **Platform Designer** per l'implementazione del computer e la generazione del file `.sopcinfo`, necessario per la corretta importazione dei driver delle IP hardware.
La programmazione della scheda in linguaggio C è stata invece realizzata tramite **Eclipse**, anch'esso incluso nel pacchetto di sviluppo fornito da **Intel**.

1.3 Cablaggio

Il **pin assignment** della matrice è il seguente:



name	Matrix PIN	Board PIN	Description
R1	1	ARDUINO_IO[0]	RED led upper Sub-matrix
G1	2	ARDUINO_IO[1]	GREEN led upper Sub-matrix
B1	3	ARDUINO_IO[2]	BLUE led upper Sub-matrix
GND	4	GND	ground
R2	5	ARDUINO_IO[3]	RED led lower Sub-matrix
G2	6	ARDUINO_IO[4]	GREEN led lower Sub-matrix
B2	7	ARDUINO_IO[5]	BLUE led lower Sub-matrix
GND	8	GND	ground
A	9	ARDUINO_IO[6]	Select which column you are driving
B	10	ARDUINO_IO[7]	Select which column you are driving
C	11	ARDUINO_IO[8]	Select which column you are driving
D	12	ARDUINO_IO[9]	Select which column you are driving
CLK	13	ARDUINO_IO[10]	Clock for led visualization
LAT	14	ARDUINO_IO[11]	Latches the value until put low
OE	15	ARDUINO_IO[12]	Output enable
GND	16	GND	Ground

Capitolo 2

Descrizione del Sistema

2.1 Architettura generale

Per il nostro utilizzo abbiamo scelto di usare il processore Nios II in versione fast ad una frequenza di 100 MHz. L'unica memoria utilizzata in una SDRAM da 64 MB sempre alla frequenza di 100 MHz ma con un anticipo di 3 ns per bilanciare lo skew. L'accelerometro è invece settato come da datasheet a 2 MHz mentre il Video DMA controller è alla stessa frequenza della memoria e manda un dato valido ogni 3 cicli di clock. L'IP per la gestione della matrice è a 400 kHz e ci permette di avere un frame rate di $f_{driver}/\left(3 * RIGHE * COLONNE\right) = 130FPS$.

2.2 RGB LED Matrix

Il sistema per la gestione della matrice è stato realizzato con una IP hardware. Ciò, anche grazie all'uso di un *DMA*, ha permesso di alleggerire il processore dal compito di pilotarla.
L'architettura è composta da:

- **LED matrix driver**: Il driver per la gestione della matrice
- **Video DMA Controller**: DMA per applicazioni video con una interfaccia stream con *back-presure* utile per ridurre la complessità dell'IP e migliorare le prestazioni.
- **Dual clock ST FIFO**: Permette di gestire i clock diversi di DMA e LED driver.

2.2.1 Matrice LED Adafruit 64 x 32

La matrice è composta da 32 righe da 64 led RGB divisa in due sotto-matrici da 16 righe che condividono tutti i segnali di controllo eccetto *R, G, B* così da

permetterci di pilotare le due metà della matrice contemporaneamente. I led sono pilotati da dei driver a corrente costante a 16 canali (ICN2037).

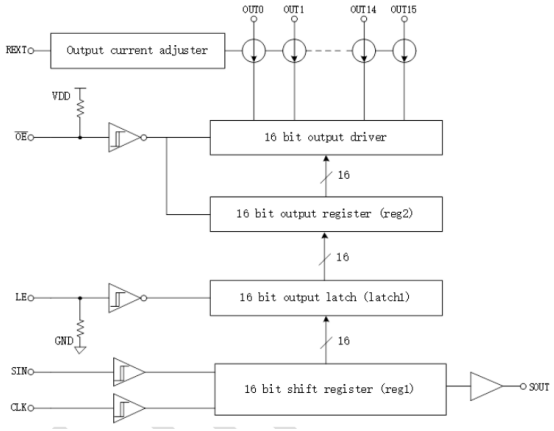


Figura 2.1: Blocco funzionale

I driver sono composti da degli shift register e da driver a corrente continua. Tra i due sono posizionati due buffer per il latch dei dati in ingresso e l'abilitazione dell'uscita. Come si può vedere dall'immagine 2.2, una volta che negli shift register ci sono i dati desiderati, il clock e l'invio di nuovi dati vengono sospesi. Successivamente si manda un impulso di *Latch Enable* che permetterà la visualizzazione dei colori nella matrice fino a quando *Output Enable* rimane attivo basso. Ognuno di questi driver pilota mezza riga. Una volta finito di stampare una riga è necessario passare alla riga successiva attraverso gli ingressi di selezione della riga `A,B,C,D`.

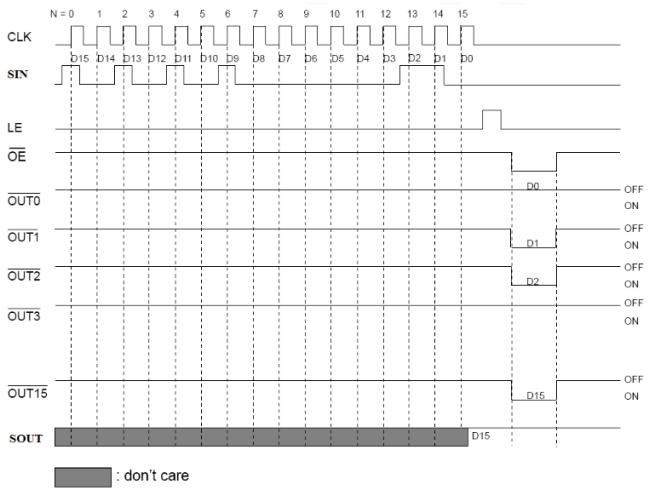


Figura 2.2: Timing

2.2.2 IP LED matrix driver

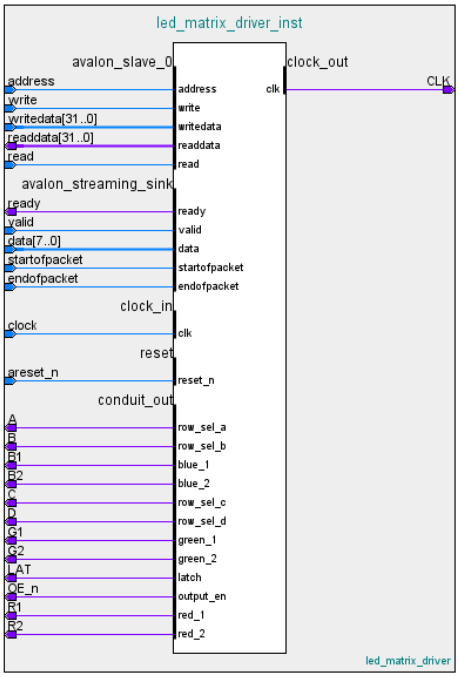


Figura 2.3: Simbolo IP

Come si può notare dall'immagine 2.3. La periferica comunica verso l'esterno con 3 interfacce di comunicazione:

- **Avalon Streaming con back-pressure**: usata per ricevere dal DMA i colori dei pixel solo quando l'interfaccia è pronta a stampare.
- **Avalon Memory Mapped Slave**: usata per resettare o disabilitare la periferica.
- **Conduit**: Usata per inviare i segnali alla matrice.

L'Ip si basa su una macchina a stati con 4 stati:

```
1 localparam RESET      = 3'd0,
2           IDLE        = 3'd1,
3           PUSH_ROW    = 3'd2,
4           CHANGE_ROW  = 3'd3;
```

Tale macchina a stati comanda i contatori di riga e colonna, i segnali di controllo e di colore. Nello stato di `IDLE` il driver aspetta che il DMA inizi a mandare un nuovo frame per poi andare nello stato `PUSH_ROW`. Nello stato `PUSH_ROW` il contatore di colonna ed il clock della matrice sono abilitati e vengono caricati i colori di una riga sulla matrice. Una volta finita la riga si va in `CHANGE_ROW` dove viene incrementato il contatore di riga, latchata la matrice con la nuova riga e disabilitato l'output enable. Poi, se giunti all'ultima riga, si torna in `IDLE` e si aspetta un nuovo `startofpacket` e `valid` dal DMA altrimenti si torna in `PUSH_ROW`.

Simulazione

Si è simulato il funzionamento del driver creando degli ingressi che simulassero il comportamento del DMA, inoltre si è simulato anche il reset sw per verificarne il corretto funzionamento. Come è possibile notare dall'immagine 2.4 il funzionamento previsto è rispettato.

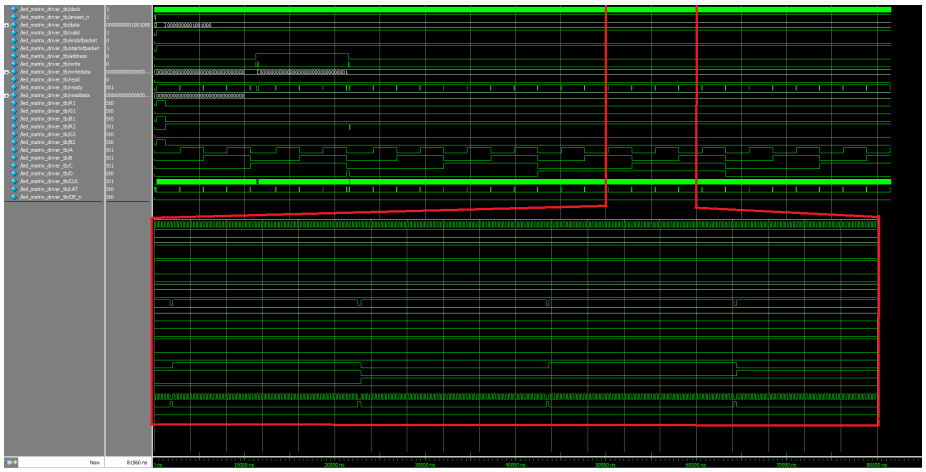


Figura 2.4: Simulazione del driver

2.2.3 Video DMA

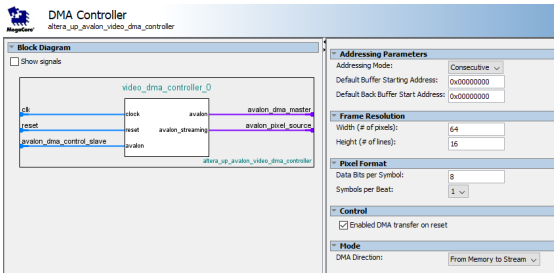


Figura 2.5: DMA

Il *Video DMA* prende i dati in maniera consecutiva dalla SDRAM mediante un'interfaccia memory mapped e la trasferisce all'IP tramite un'interfaccia stream. Essendo il back-buffer ed il buffer con lo stesso indirizzo di default, il back-buffer non viene implementato. Inoltre, selezionando la grandezza del frame il DMA una volta arrivato a trasferire tutto il frame torna all'inizio del buffer di pixel e si sincronizza nuovamente con la IP attraverso i segnali di `valid` e `startofpacket`. Ogni ciclo di clock il DMA preleva 8 bit dalla SDRAM di cui solo i primi 6 sono significativi con i valori R,G,B di un determinato LED nella metà bassa della matrice e del suo corrispettivo nella metà alta così da poter settare nella matrice due led per ciclo di clock.

2.2.4 ST Dual Clock FIFO

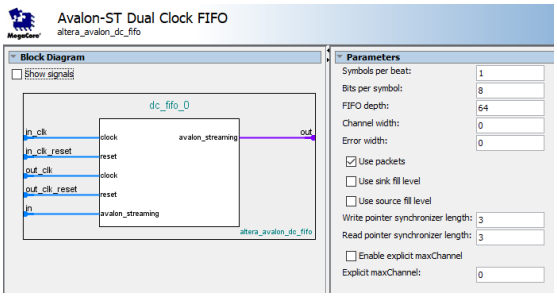
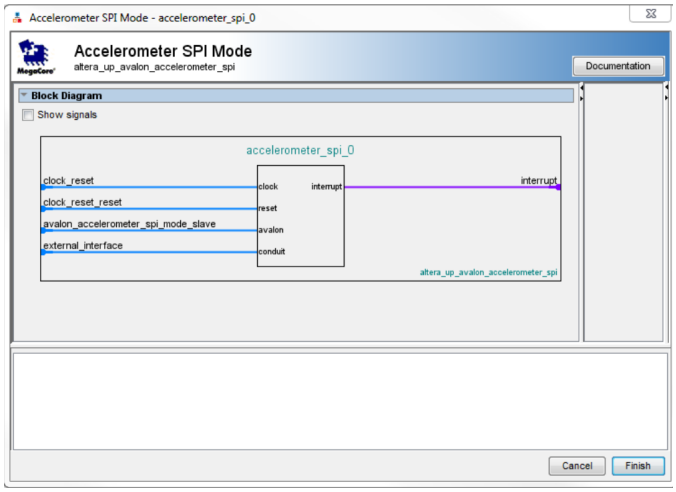


Figura 2.6: FIFO

La dual clock FIFO serve per temporizzare l'interfaccia Stream tra il DMA ed il driver della matrice in quanto hanno frequenze di clock differenti (100 MHz vs 400 kHz).

2.3 Accelerometro

L'FPGA DE10-Lite possiede il **chip ADXL345**, un accelerometro integrato in grado di riportare le misure dell'accelerazione lungo i tre assi x, y, z, con una risoluzione selezionabile compresa tra $\pm 2g$ e $\pm 16g$, mediante una comunicazione tramite SPI o I2C.
Intel mette a disposizione un core dell'accelerometro da istanziare nel Platform Desgin. Questo, tramite protocollo SPI permette di configurare l'accelerometro mediante lettura o scrittura dei registri nel memory-mapped.



È opportuno affermare che, come scritto nel datasheet del ADXL345, il clock richiesto è di 2 MHz. La condizione è stata soddisfatta aggiungendo un segnale di clock dal PLL di sistema, si guardi il capitolo 3.

L'obiettivo successivo è stato quello di realizzare una libreria in C `api_accelerometer` da poter includere nel programma principale. Nel fare ciò sono state utilizzate le HAL dell'accelerometro già fornite da Altera per poter accedere facilmente ai registri dell'accelerometro.

Capitolo 3

Implementazione del Sistema Embedded

3.1 Struttura globale del sistema

Il computer è stato strutturato come segue:

- Clock da 50 MHz
- CPU Nios II fast
- SDRAM
- IP per accelerometro
- JTAG-UART per Debug
- PLL
- Video DMA controller
- LED Matrix driver
- Buffer FIFO
- PIO per sliders
- PIO per push-buttons
- PIO per LEDs

3.2 Implementazione e Report

Dal report dell'implementazione si possono notare i seguenti risultati.

3.2.1 Utilizzo Risorse FPGA

Flow Status	Successful - Mon May 19 20:45:17 2025
Quartus Prime Version	18.1.0 Build 625 09/12/2018 SJ Lite Edition
Revision Name	led_sandbox
Top-level Entity Name	led_sandbox_top
Family	MAX 10
Device	10M50DAF484C7G
Timing Models	Final
Total logic elements	5,542 / 49,760 (11 %)
Total registers	3636
Total pins	80 / 360 (22 %)
Total virtual pins	0
Total memory bits	66,816 / 1,677,312 (4 %)
Embedded Multiplier 9-bit elements	6 / 288 (2 %)
Total PLLs	1 / 4 (25 %)
UFM blocks	0 / 1 (0 %)
ADC blocks	0 / 2 (0 %)

Figura 3.1: Risorse utilizzate

3.2.2 Timing Analyzer

	Clock	Setup	Hold	Recovery	Removal	Minimum Pulse Width
1	Worst-case Slack	7.910	0.143	12.881	0.408	9.378
1	MAX10_CLK1_50	7.910	0.143	12.881	0.408	9.378
2	altera_reserved_tck	44.398	0.147	48.019	0.601	49.412
2	Design-wide TNS	0.0	0.0	0.0	0.0	0.0
1	MAX10_CLK1_50	0.000	0.000	0.000	0.000	0.000
2	altera_reserved_tck	0.000	0.000	0.000	0.000	0.000

Figura 3.2: Timer Analyzer

	Fmax	Restricted Fmax	Clock Name	Note
1	82.71 MHz	82.71 MHz	MAX10_CLK1_50	
2	89.25 MHz	89.25 MHz	altera_reserved_tck	

Figura 3.3: frequenza massima del clock

Capitolo 4

Software

4.1 Architettura

Il software è suddiviso in tre cartelle principali:

- `led_sandbox_common`: questa contiene il software che implementa la logica del programma principale, descritta in maniera **hardware agnostic**. Questo significa che non sono presenti chiamate a funzioni specifiche di un certo environment, ma vengono utilizzate le funzioni offerte dall'**hardware abstraction layer**. Qui - in particolare nell'header file `hal.h` - sono dichiarati i prototipi di funzioni utili alla logica (ad esempio `hal_read_accelerometer`), ma non la loro implementazione. Questa verrà definita in un file sorgente specifico per l'ambiente in cui si desidera eseguire il programma - ad esempio il Nios II - nel quale verranno usate funzioni di basso livello dell'ambiente selezionato. Questa architettura risulterà utile per alcuni scopi specifici, come verrà spiegato in seguito (si veda 4.2).

- `led_sandbox_nios2`: contiene l'implementazione delle funzioni di alto livello specifiche per l'esecuzione su Nios II.

- `led_sanbox_posix`: contiene l'implementazione delle funzioni di alto livello specifiche per l'esecuzione in un ambiente POSIX. Ciò consente di poter emulare la logica del programma su una macchina con sistema operativo della famiglia UNIX.

Le ultime due cartelle contengono entrambe un file sorgente `main.c` (in base all'ambiente che si desidera eseguire, solo uno di questi file verrà effettivamente incluso nella compilazione). Questo file contiene l'entry-point del programma, in cui è presente una chiamata iniziale ad una funzione di inizializzazione - la quale viene quindi eseguita una sola volta - `led_sandbox_init` e un loop infinito nel quale viene eseguita la funzione `led_sandbox_loop` e, ad ogni tick, la funzione `scheduler_tick` (vedi 4.1.1). Le funzioni `led_sandbox_init` e

`led_sandbox_loop` contengono rispettivamente procedure di inizializzazione (ad esempio il set-up delle periferiche o la registrazione di task periodici) e istruzioni che devono essere eseguite continuamente alla massima velocità permessa dal loop, senza il bisogno di essere eseguite con un certo periodo. Queste funzioni sono definite in led_sandbox_common in maniera hardware agnostic.

```
int main() {
    led_sandbox_init();

    while (1) {
        // If tick has elapsed, call scheduler
        if (tick_flag) {
            scheduler_tick();

            // Reset tick flag
            tick_flag = 0;
        }

        // Call loop subroutine
        led_sandbox_loop();
    }

    return 0;
}
```

4.1.1 Scheduler

Il sistema è dotato di un kernel elementare, lo **scheduler**. Questo componente si occupa di eseguire periodicamente i task registrati dall'utente, nascondendo la gestione delle temporizzazioni, le quali avvengono per mezzo di un timer con un periodo di 1 ms, che definisce il *system tick*. Tutti i task periodici che possono essere registrati devono avere un periodo che sia multiplo intero del system tick.

Allo scadere del periodo del tick, una *interrupt service routine* setta un flag (`tick_flag`), il quale viene controllato tramite polling all'intero del loop principale, discusso in 4.1, per poter far eseguire uno step di 1 tick allo scheduler tramite la funzione `scheduler_tick`. Questa routine decrementa dei contatori, ciascuno inizializzato al periodo di un task precedentemente registrato espresso in numero di tick; se uno di questi contatori raggiunge lo 0, allora viene chiamata la callback associata al task periodico corrispondente e che deve essere fornita al momento della registrazione di un task periodico, tramite la funzione `scheduler_add_periodic_task`. Per conformità, Il tipo della callback deve essere obbligatoriamente l'indirizzo in memoria di una funzione che accetta in ingresso `void` e restituisce `void`.

```
bool scheduler_tick() {
    for (uint16_t i = 0; i < num_tasks; i++) {
        periodic_tasks[i].tick_counter -= 1;
```

```
4
5           if (periodic_tasks[i].tick_counter <= 0) {
6               periodic_tasks[i].cb();
7               periodic_tasks[i].tick_counter += periodic_tasks
                    [i].counter_reset;
8           }
9       }
10
11      for (size_t i = 0u; i < MAX_ONESHOT_EVENTS; ++i)
12      {
13          if (oneshot_events[i].cb != NULL)
14          {
15              oneshot_events[i].tick_counter -= 1;
16
17              if (oneshot_events[i].tick_counter <= 0)
18              {
19                  oneshot_events[i].cb();
20
21                  oneshot_events[i].cb = NULL;
22                  oneshot_events[i].tick_counter = 0;
23              }
24          }
25      }
26
27      return true;
28  }
```

4.1.2 Accelerometro

Di seguito vengono riportate le funzioni usate per la realizzazione della libreria dell'accelerometro, contenute in `altera_up_avalon_accelerometer_spi.h`. È possibile consultare le altre funzioni della libreria Altera qui.

```
1  alt_up_accelerometer_spi_dev*
       alt_up_accelerometer_spi_open_dev(const char *name)
```

Inizializza l'accelerometro, con risoluzione di default di $\pm 2g$ e ritorna un valore di tipo `alt_up_accelerometer_spi_dev`.

```
1  int alt_up_accelerometer_spi_read_x_axis(
       alt_up_accelerometer_spi_dev *accel_spi, alt_32 *x_axis)
```

Fornendo in ingresso il tipo accelerometro si ottiene in uscita il valore lungo l'asse X misurato, all'indirizzo del puntatore di tipo `int32_t`. Se la funzione viene eseguita con successo, il valore di ritorno è 0.
Le altre 2 funzioni usate sono quelle relative all'asse Y e Z, del tutto analoghe alla precedente. I dati forniti nei registri dall'accelerometro sono RAW e possono andare da -255 a 255.

Per gestire al meglio i dati estratti, è stata realizzata una Application Programming Interface (API). Una volta stabilito il range di funzionamento $\pm 2g$, ottimo per poter rilevare la forza di gravità e con una sensibilità maggiore, le 3 funzioni implementate sono le seguenti:

```
1 alt_up_accelerometer_spi_dev* accelerometer_init(void);
        // initialize value
2 int32_t accelerometer_get_data_raw(accelerometer_data_t*
    data);      // get data in raw value
3 int32_t accelerometer_get_data_g(accelerometer_data_t*
    data_g);      // get data in g vale
```

L'obiettivo è quello di riportare i valori della misura all'interno di una struttura chiamata `accelerometer_data_t` definita come segue:

```
1 typedef struct {
2     int32_t x;
3     int32_t y;
4     int32_t z;
5 } accelerometer_data_t;
```

Inoltre, per soli fini di test, è stata inserita una terza funzione che riporta il valore dell'accelerazione in m/s^2 approssimata al primo intero inferiore.

Essendo le misure di strumenti inerziali molto rumorose, è opportuno mediare nel tempo queste misure. Per questo motivo, è stato creato il task periodico `accelerometer_average_task`, registrato all'inizializzazione sullo scheduler con un periodo di 1 tick (equivalente a 1 ms). Questo task accederà alla API sopra descritta tramite l'interfaccia HAL. Ad ogni tick viene campionato il valore letto dall'accelerometro e, ogni 10 campioni, questi valori vengono mediati e resi accessibili agli altri task tramite una struttura del tipo `accelerometer_data_t`, nominata `acceleration_average`.

4.1.3 Macchina A Stati

Un ulteriore task registrato sullo scheduler con un periodo di 1 tick è quello che si occupa della gestione della macchina a stati. Gli stati permessi sono:

- `RESET`: stato di reset.
- `IDLE`: attesa di ulteriori comandi per l'attivazione della sandbox.
- `GSENSOR_SANDBOX`: stato di abilitazione della sandbox.

Il passaggio tra i vari stati è regolato dalla lettura dei push-buttons e degli sliders presenti sulla board. Il push-button 0 coincide con il pulsante di reset della macchina a stati; la pressione del push-button 1 permette di passare allo stato di `IDLE`; l'ulteriore pressione del push-button 1, insieme alla attivazione dello slider 0, abilita lo stato `GSENSOR_SANDBOX`.

4.1.4 Pixel Dust Task

L'ultimo task periodico presente è il `pixel_dust_task`, il cui periodo in tick dipende dal numero di FPS desiderati, definito dalla macro `MATRIX_FPS`.

Questo task, abilitato solo quando lo stato è `GSENSOR_SANDBOX`, ha il compito di inizializzare e aggiornare i buffer di memoria che contengono la posizione dei granelli di sabbia frame dopo frame. I buffer istanziati sono 3: un **front buffer**, al quale il DMA accederà direttamente in maniera consecutiva e ciclica, un **back buffer**, necessario per svolgere le operazioni di aggiornamento del frame evitando che queste vengano proiettate sulla matrice, e un buffer temporaneo, necessario per compensare - tramite una funzione opportuna - l'offset del frame di una riga, che risulterebbe altrimenti in uno scorrimento dell'immagine verso l'alto. Questi buffer sono costituiti da 3 array di `WIDTH * HEIGHT / 2` interi di tipo `uint8_t`, dove `WIDTH = 64` è il numero di colonne della matrice e `HEIGHT = 32` è il numero di righe. La dimensione degli array è dimezzata rispetto alla dimensione della matrice poiché ciascun byte contiene la codifica RGB di due pixel, uno alla riga `i` e l'altro alla riga `i + 16`. Il DMA è impostato per accedere alla memoria con allineamento al byte.

L'aggiornamento del frame avviene per mezzo di un engine fisico, importato da una libreria in C++ creata da Adafruit, il quale calcola la nuova posizione dei granelli in base alla accelerazione misurata e mediata dal task precedentemente descritto. Inoltre esso tiene in considerazione gli urti tra i granelli, la cui posizione dipenderà quindi anche dal coefficiente di elasticità impostato e dal numero di urti subiti.

```
// Erase old grain positions in pixel_buf[]
    dimension_t x, y;
    for(uint32_t i = 0; i < N_GRAINS; i++) {
        sand.getPosition(i, &x, &y);
        if (y >= (HEIGHT / 2)) {  // If the grain is in the
            upper half, erase the upper pixel
          pixel_back_buf[(y % (HEIGHT / 2)) * WIDTH + x] &=
              CLEAR_UPPER_PIXEL;
        } else {  // Otherwise erase the lower pixel
          pixel_back_buf[(y % (HEIGHT / 2)) * WIDTH + x] &=
              CLEAR_LOWER_PIXEL;
        }
    }

    // Run one frame of the simulation
    // X & Y axes are flipped around here to match physical
        mounting
    sand.iterate(acceleration_average.a_x,
        acceleration_average.a_y, acceleration_average.a_z);
    printf("Iteration\n");

    // Draw new grain positions in pixel_buf[]
    grain_color_t color = BLACK;
    for(uint32_t i = 0; i < N_GRAINS; i++) {
```

```
20          sand.getPosition(i, &x, &y);
21          sand.getColor(i, &color);
22          if (y >= (HEIGHT / 2)) { // If the grain is in the
                upper half, set the upper pixel
23            pixel_back_buf[(y % (HEIGHT / 2)) * WIDTH + x] |=
                  SET_UPPER_PIXEL((uint8_t)color);
24          } else { // Otherwise set the lower pixel
25            pixel_back_buf[(y % (HEIGHT / 2)) * WIDTH + x] |=
                  SET_LOWER_PIXEL((uint8_t)color);
26          }
27      }
```

Listing 4.1: Passaggio di aggiornamento del back buffer di un frame

4.2 Emulatore

Grazie al livello di astrazione introdotto, è possibile definire un'implementazione alternativa delle funzioni dichiarate in `hal.h` e utilizzare dalla in `led_sandbox_common`. L'implementazione alternativa realizzata è compatibile con gli ambienti operativi UNIX-like, poiché sfrutta l'interfaccia POSIX. Maggiori informazioni su come compilare il progetto, eseguirlo e manovrare gli input sono disponibili sul README della repository GitHub. Ciò ci ha consentito di emulare la logica del progetto prima ancora di avere l'architettura hardware funzionante, verificando così il corretto funzionamento dei vari task dichiarati, in particolar modo quello contenente la libreria dell'engine fisico.

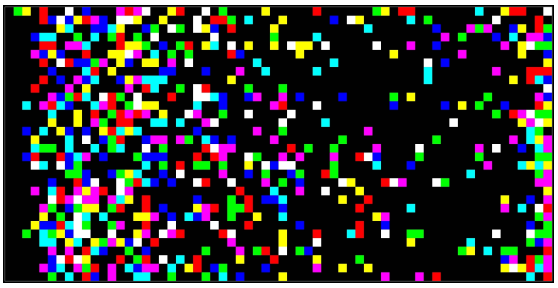


Figura 4.1: Emulatore in Python della matrice LED che aggiorna il frame letto da un file di testo, il quale viene modificato dalla implementazione POSIX del progetto.

4.3 Memory Footprint

La memory footprint espressa in byte del software è riportata in Tabella 4.1.

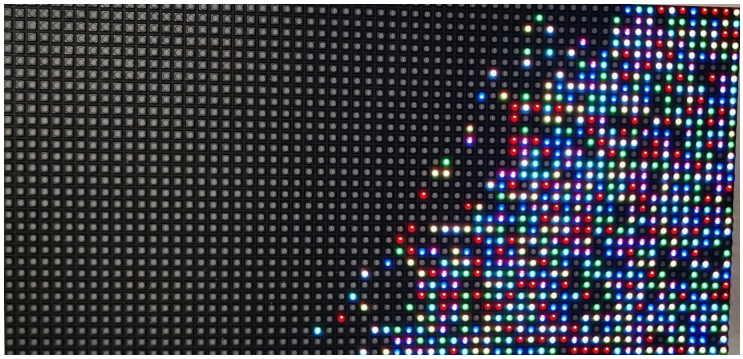
Tabella 4.1: Memory footprint

text	84120
data	7036
bss	4300
totale	95456

Capitolo 5

Conclusioni

Con questo progetto, sono state approfondite tematiche quali la programmazione embendded, la gestione delle risorse di sistema e l'interfacciamento con dispositivi periferici. Tutti i file C sono stati testati con debug, mentre per quanto riguarda l'analisi temporale del sistema, l'utilizzo del Timing Analyzer ha permesso di verificare che tutti i vincoli temporali fossero rispettati con un margine di sicurezza adeguato.



5.1 Miglioramenti futuri

Possibili miglioramenti futuri sono l'incremento del numero di colori visualizzabili mediante l'uso di PWM per pilotare gli ingressi *R,G,B*. Altro possibile miglioramento è l'aggiunta della possibilità di regolare la luminosità mediante l'interfaccia memory mapped regolando opportunamente l'*Output Enable* della matrice.

Un'altra idea è quella di inserire una seconda modalità di funzionamento, che elaborando con l'IA i dati forniti da una webcam, permetterà di muovere la sabbia attraverso i gesti della mano.