CHIP-8 STM32

Federico Bruzzone, Lorenzo Ferrante, Andrea Longoni
{federico.bruzzone, lorenzo.ferrante1, andrea.longoni3}@studenti.unimi.it

5 novembre 2023

1 Introduzione

CHIP-8 è un linguaggio di programmazione creato a metà degli anni '70 da Joseph Weisbecker per semplificare lo sviluppo di videogiochi per microcomputer a 8 bit. I programmi CHIP-8 vengono interpretati da una macchina virtuale che è stata estesa parecchie volte nel corso degli anni, tra le versioni più adottate citiamo S-CHIP e la più recente XO-CHIP.

La semplicità dell'interprete in aggiunta alla sua lunga storia e popolarità hanno fatto sì che emulatori e programmi CHIP-8 vengano realizzati ancora oggi. Nel corso degli anni molti videogiochi storici sono stati riscritti in CHIP-8 tra cui Pong, Space Invaders e Tetris.

Lo scopo del progetto è quello di costruire un emulatore CHIP-8 e S-CHIP in grado di funzionare su un microcontrollore STM32.

In questo documento ci riferiremo alla macchina virtuale che interpreta programmi CHIP-8 con "interprete". Mentre utilizzeremo "emulatore" per indicare l'interprete assieme ad una sua implementazione (o "port"), ovvero un programma che gestisce l'audio, il video, l'input da tastiera e interagisce con l'API della macchina virtuale.

2 Hardware

2.1 Schema di collegamento

2.2 Materiali e costi

3 Software

Il nostro software si divide in due componenti principali: l'interprete CHIP-8 e l'infrastruttura necessaria per "portarlo" su un microcontrollore STM32, ovvero l'interfaccia con lo schermo e i gestori per la scheda microSD, per il keypad e per il beeper.

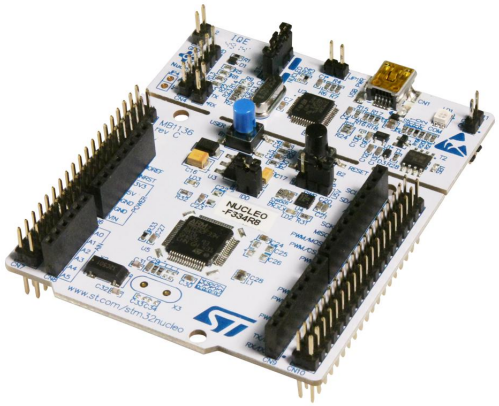
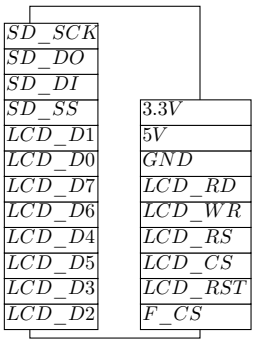


Figura 1: Il microcontrollore `STM32F334R8T6`.

(a) Lo schermo ILI9341.



(b) Pinout dello schermo `ILI9341`.

3.1 Interprete CHIP-8

Abbiamo deciso di scrivere l'interprete da zero e per farlo è stato necessario consultare le specifiche (de facto standard) che definiscono il comportamento di un interprete CHIP-8 [2] e S-CHIP [5].

L'interprete è scritto in C99, non ha I/O ed è freestanding [3], ovvero non dipende dalla libreria standard del C. Tutto questo è mirato a rendere l'interprete altamente portabile.

Per rimuovere la dipendenza dalla libreria standard del C è stato necessario includere alcune funzioni direttamente da libgcc, trovare un modo alternativo per implementare le asserzioni e includere una funzione ad hoc per la generazione di numeri casuali.

Per testare più comodamente l'interprete abbiamo sviluppato un semplice emulatore su desktop utilizzando SDL2 [4], una libreria scritta in C che consente di gestire audio, video e input da tastiera. In seguito l'interprete è stato sottoposto ad un'apposita test suite [1] che mira a verificare il comportamento corretto di ciascun opcode.

Descrizione	Modello	Costo unitario	Unità	Costo
Microcontrollore	STM32 F334R8T6	14.99	1	14.99
Schermo	ILI9341 2.4"	6.50	1	6.50
Tastierino	Matrix keypad 4×4	3.99	1	3.99
Beeper		0.99	1	0.99
Breadboard e cablaggio		4.99	1	4.99
			Totale	31.50€

Tabella 1: Materiali utilizzati per la costruzione del progetto. I costi indicati provengono da negozi online come Amazon e eBay.

3.1.1 Timing

Uno dei problemi principali durante lo sviluppo di un emulatore è la gestione del timing.

In particolare abbiamo voluto disaccoppiare la frequenza dell'emulatore (variabile e regolabile dal giocatore) dalla frequenza del delay timer e del sound timer (costante a 60 Hz).

La frequenza dell'emulatore equivale al numero di istruzioni che esegue ogni secondo.

Abbiamo sperimentato con due approcci diversi, nel primo il ritardo del game loop è variabile e dipende dalla frequenza dell'emulatore selezionata dal giocatore e ogni ciclo di emulazione gestisce esattamente un'istruzione. Mentre i timer vengono decrementati ogni n-esima iterazione del game loop, dove n = EMU_FREQ / 60.

Ad esempio, se EMU_FREQ = 540, i timer vengono decrementati ogni 9º ciclo (540 / 60 = 9).

Nel secondo approccio il ritardo del game loop è costante a 16.666 ms per ottenere un frame rate di 60fps. In questo approccio, ogni ciclo di emulazione gestisce un numero variabile di istruzioni. Ad esempio, se EMU_FREQ = 540, ogni ciclo di emulazione gestisce 9 istruzioni.

Dopo un attenta riflessione abbiamo deciso di impiegare il secondo approccio. Il principale svantaggio del primo approccio è che chiama una funzione simil-sleep per un periodo molto breve dopo ogni istruzione, se EMU_FREQ = 540, il ritardo di una sleep sarebbe di circa 1.85 ms. Purtroppo questo tipo di funzione non offre questo genere di precisione.

Per questo motivo abbiamo deciso di disaccoppiare la frequenza dei timer (costante a 60 Hz) dalla frequenza dell'emulatore (variabile e regolabile dal giocatore) utilizzando il secondo approccio.

Il ritardo del game loop è costante a 16.666 ms (per ottenere un frame rate di 60 fps), mentre il numero di istruzioni eseguite in ogni ciclo di emulazione è variabile.

Ciò consente di decrementare i timer dopo ogni ciclo di emulazione a un tasso costante di 60 Hz.

Abbiamo considerato anche eventuali problematiche che sarebbero potute sorgere con il secondo approccio. In particolare abbiamo considerato il fatto che anche se non tutte le istruzioni richiedono lo stesso tempo per essere eseguite, anche la più lenta richiede una quantità trascurabile di tempo. Ciò significa che possiamo comportarci come se tutte le istruzioni richiedessero il medesimo tempo.

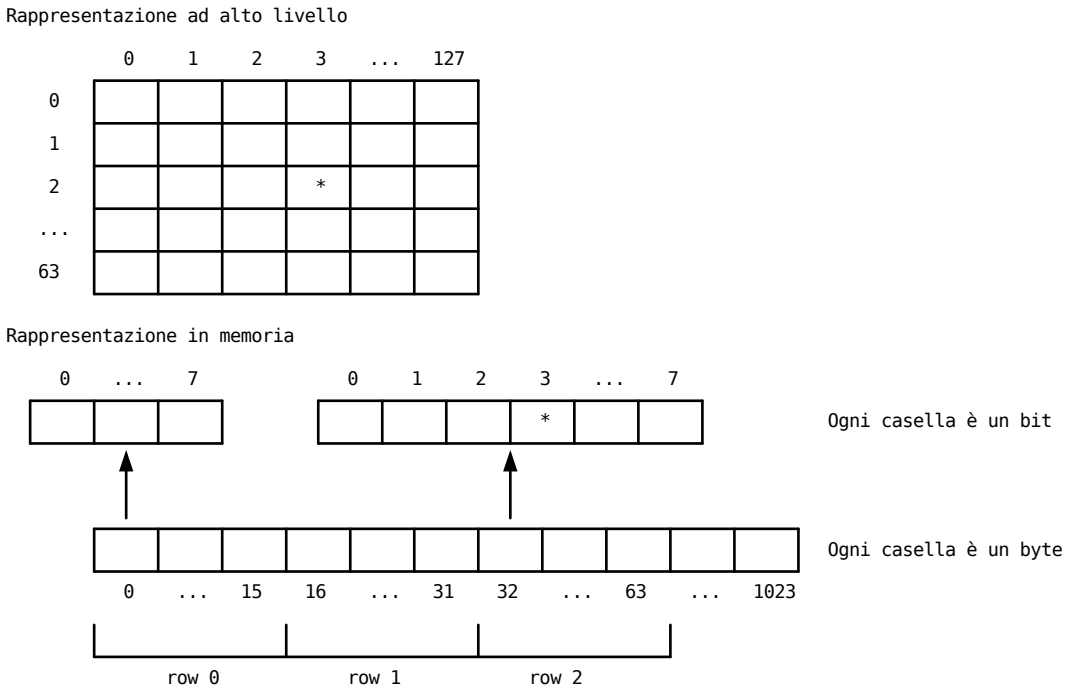


Figura 3: Esempio della mappatura di un pixel.

3.1.2 Ottimizzazioni

È stato necessario introdurre delle ottimizzazioni all'interno dell'interprete per poterlo far girare su un microcontrollore.

L'ottimizzazione principale è legata alla rappresentazione dello schermo in memoria. Ad alto livello lo schermo può essere visto come una matrice di 128x64 pixel monocromi. Purtroppo però una rappresentazione simile, dove ciascun pixel viene rappresentato da un byte occuperebbe 8192 byte, e dobbiamo considerare il fatto che il nostro microcontrollore ha a disposizione solamente di 16 Kb di SRAM.

Per questo motivo abbiamo deciso di rappresentare lo schermo come un array unidimensionale di 1024 byte, dove ciascun pixel viene rappresentato con un singolo bit. Questa decisione ha aggiunto però un livello di indirezione dato che una coordinata ad alto livello sulla matrice 128x64 deve essere mappata ad una coordinata "in memoria".

3.1.3 Quirks

3.2 Architettura

4 Analisi del consumo energetico

5 Considerazioni finali

Riferimenti bibliografici

[1] CHIP-8 test suite. `https://github.com/Timendus/chip8-test-suite`.

[2] Cowgod's Chip-8 Technical Reference v1.0. `http://devernay.free.fr/hacks/chip8/C8TECH10.HTM`.

[3] ISO/IEC 9899:1999 - 4. Conformance. `https://port70.net/~nsz/c/c99/n1256.html#4p6`.

[4] Simple DirectMedia Layer. `https://www.libsdl.org`.

[5] SUPER-CHIP v1.1. `http://devernay.free.fr/hacks/chip8/schip.txt`.