MSc in Computer Science
at University of Milan

CHIP-8 STM32

Proposta per il Progetto di PROS,
corso tenuto da **Danilo Bruschi**

Email:
federico.bruzzone@studenti.unimi.it
lorenzo.ferrante1@studenti.unimi.it
andrea.longoni3@studenti.unimi.it

Creato da:
Federico Bruzzone
Lorenzo Ferrante
Andrea Longoni

Anno accademico 2022/2023

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.

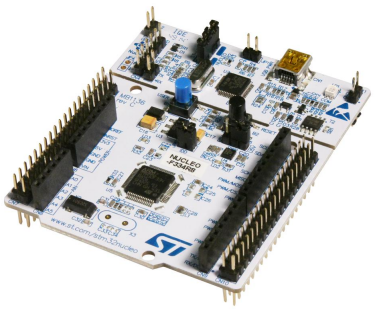
Lo scopo del progetto è quello di costruire un emulatore CHIP-8 e S-CHIP in grado di funzionare su un microcontrollore STM32.

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

Le componenti principali utilizzate per la relizzazione del progetto sono 5: una scheda ST `STM32F334R8T6` (Fig. **??**), uno schermo TFT LCD ILI9341 (Fig. 2a), e un lettore di schede microSD integrato nello schermo, un tastierino matriciale 4×4 e un beeper.

Il componente principale é il microcontrollore `STM32F334R8T6` basato su architettura ARM con processore Cortex-M4 da 72 `MHz`, 64 `Kb` di memoria flash e 16 `Kb` di SRAM. Abbiamo deciso di utilizzare questa scheda perché le ROM dei giochi CHIP-8 e S-CHIP hanno dimensione massima di 4 `Kb`. Considerando questo e il fatto che la macchina virtuale necessita 4 `Kb` per poter funzionare non è stato potuto utilizzare la scheda `STM32L053R8T6` fornitaci durante il corso a causa della sua quantità limita di SRAM, precisamente 8 `Kb`.



(a) LIl microcontrollore `STM32F334R8T6`.

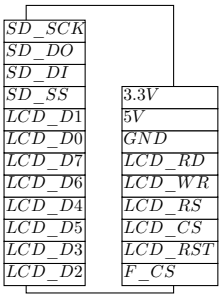
3V3	3V3	VB	RST
G	G	C13	C13
5V	5V	C14	C14
B9	B9	C15	C15
B8	B8	R	R
B7	B7	A0	A0
B6	B6	A1	A1
B5	B5	A2	A2
B4	B4	A3	A3
B3	B3	A4	A4
A15	A15	A5	A5
A12	A12	A6	A6
A11	A11	A7	A7
A10	A10	B0	B0
A9	A9	B1	B1
A8	A8	B2	B2
B15	B15	B10	B10
B14	B14	3V3	3V3
B13	B13	G	G
B12	B12	5V	5V

(b) Pinout del microcontrollore `STM32F334R8T6`.

Necessiteremo inoltre di un display TFT LCD (thin-film-transistor liquid-crystal display) a colori retroilluminato, per poterci interfacciare col gioco. Questo display, da 2.4 pollici, é basato sul controller `ILI9341` e ha una risoluzione di 320×240 `px`.

Il display dispone di un lettore di schede microSD, che verrà utilizzato per caricare una delle ROM presenti sulla SD in SRAM.

(a) Lo schermo ILI9341.



(b) Pinout dello schermo `ILI9341`.

Per interagire con l'emulatore utilizzeremo una tastiera matriciale 4×4 corrispondente alla tastiera esadecimale originale.

Per riprodurre gli effetti sonori generati dal gioco utilizzeremo un beeper a frequenza variabile. In CHIP-8 e S-CHIP vi é un solo tipo di suono che può essere riprodotto durante tutta l'esecuzione del gioco. Infine sarà possibile aggiungere uno slot per l'alimentazione tramite due batterie `AA`.

La piattaforma scelta per lo sviluppo del progetto é il microcontrollore STM32F334R8T6, un microcontrollore a 32 bit con 16 KB di SRAM e 64 KB di Flash. Il microcontrollore é dotato di un clock a 72 MHz, 51 pin GPIO,

2.1 Schema di collegamento

2.2 Materiali e costi

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€

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

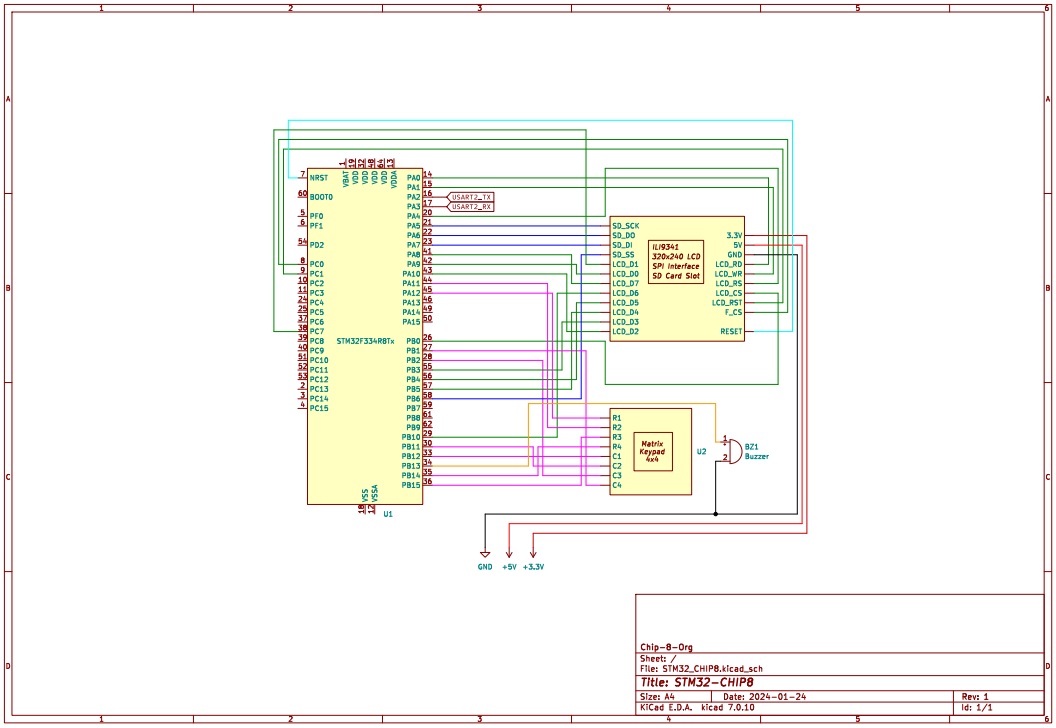


Figure 3: Schematic del progetto.

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.

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 ha un'architettura basata su registri e possiede 4 KB di memoria, 16 registri general purpose, un registro per gli indirizzi di memoria, un registro per il delay timer, un registro per il sound timer, uno stack per gestire le chiamate a subroutine, uno stack pointer e un program counter, come si puo' vedere nel listato [1].

Il delay timer viene utilizzato come cronometro mentre il sound timer è utilizzato per gestire gli effetti sonori, quando il suo valore è diverso da zero, l'emulatore attiva il beeper.

Ad ogni ciclo di esecuzione l'interprete effettua il fetch dell'istruzione puntata dal program counter in memoria, la decodifica e la esegue.

Sono supportate 45 istruzioni diverse, ciascuna delle quali è rappresentata da uno specifico opcode in cui al suo interno sono passati anche eventuali parametri.

```c
#define RAM_SIZE 4096
#define SCREEN_SIZE 1024         // 128x64 pixels = 8192 bits = 1024 bytes
#define KEYPAD_SIZE 16

typedef struct {
    uint8_t RAM[RAM_SIZE];

    uint16_t I;                  // Index register
    uint16_t PC;                 // Program counter

    uint16_t stack[16];
    uint8_t SP;                  // Stack pointer

    uint8_t V[16];               // Variable registers
    uint8_t hp48_flags[8];       // HP-48's "RPL user flag" registers (S-CHIP)

    uint8_t screen[SCREEN_SIZE];
    uint8_t keypad[KEYPAD_SIZE];
    uint8_t wait_for_key;

    uint8_t DT;                  // Delay timer
    uint8_t ST;                  // Sound timer

    uint16_t opcode;             // Current opcode
    uint64_t rng;                // PRNG state
    int IPF;                     // No. instructions executed each frame

    bool hi_res;                 // Enable 128x64 hi-res mode (S-CHIP)
    bool screen_updated;         // Was the screen updated?

    Platform platform;           // CHIP-8, CHIP-48/S-CHIP 1.0 or S-CHIP 1.1 behavior?
} Chip8;
```

Listing 1: Struttura dell'emulatore Chip8

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

Per rimuovere la dipendenza da libc è stato necessario includere alcune funzioni direttamente da libgcc, in particolare abbiamo re-implementato le funzioni `memset` e `memcpy` (listato [2]). Inoltre, abbiamo trovato un modo alternativo per implementare le asserzioni e includere una funzione ad hoc per la generazione di numeri casuali.

```c
static void
*memset_(void *dest, int val, uint64_t len)
{
    unsigned char *ptr = dest;
    while (len-- > 0)
        *ptr++ = val;
    return dest;
}
```

```c
static void
*memcpy_(void *dest, const void *src, uint64_t len)
{
    char *d = dest; const char *s = src;
    while (len--)
        *d++ = *s++;
    return dest;
}
```

Listing 2: Implementazioni di `memset` e `memcpy`

Infine 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.

3.1.1 Gestione del timing

Uno dei problemi principali durante lo sviluppo di un emulatore è la gestione del timing, in particolare è necessario limitare la "velocità" dell'emulatore bloccando temporaneamente la sua esecuzione.

Inoltre abbiamo dovuto disaccoppiare la frequenza dell'interprete (regolabile dal giocatore) dalla frequenza del delay timer e del sound timer (costante a 60 Hz). Dove con frequenza dell'interprete ci riferiamo al numero di istruzioni che esegue ogni frame.

Inizialmente abbiamo optato per la gestione di una singola istruzione per ciclo di esecuzione, di conseguenza il ritardo del game loop risultava variabile e dipendeva dalla frequenza selezionata dal giocatore. Per assicurare una frequenza di 60 Hz i timer venivano decrementati ogni n-esima iterazione del game loop, dove n = FREQ / 60. Ad esempio se FREQ = 540, i timer venivano decrementati ogni 9º ciclo.

Purtroppo però questo approccio presenta un problema non trascurabile, ovvero effettua una chiamata ad una funzione simil-sleep per un periodo molto breve dopo ogni istruzione. Ad esempio se FREQ = 540, il ritardo di una sleep sarebbe solo di 1.85 ms, e questo genere di funzione non offre una precisione simile. Per questo motivo abbiamo optato per una soluzione differente.

Abbiamo fissato il ritardo del game loop a 16.666 ms, un valore sufficientemente alto da non avere problemi di granularità. Inoltre in questo modo otteniamo un frame rate di 60 fps esatti. Avendo reso il ritardo costante abbiamo dovuto rendere variabile il numero di istruzioni gestite durante un ciclo di esecuzione. In particolare vengono gestite n istruzioni per ciclo, dove n = FREQ / 60. Ad esempio se FREQ = 540, vengono gestite 9 istruzioni per ciclo. A questo punto dato che il game loop viene ripetuto con una frequenza di 60 Hz risulta banale gestire la frequenza dei timer.

Sono state considerate anche eventuali problematiche che sarebbero potute sorgere con questo approccio. In particolare non tutte le istruzioni impiegano lo stesso tempo per essere eseguite, ma fortunatamente anche l'istruzione più lenta richiede una quantità trascurabile di tempo. Ciò significa che possiamo comportarci come se tutte le istruzioni richiedessero il medesimo tempo.

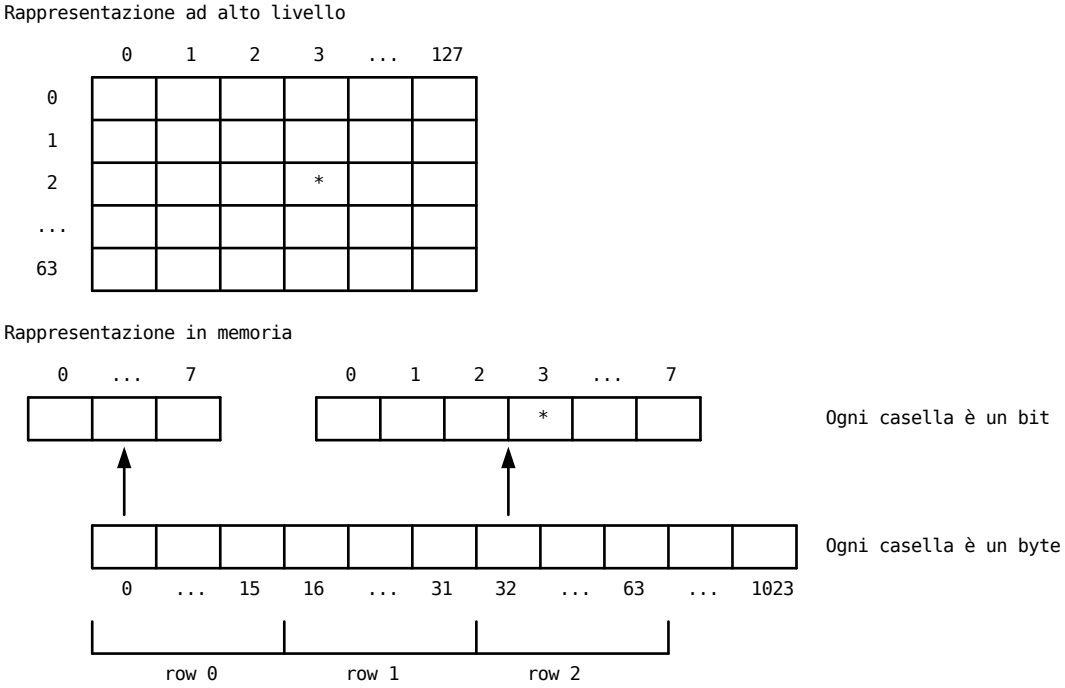


Figure 4: Esempio della mappatura di un pixel.

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. Una rappresentazione simile occuperebbe 8192 byte, dato che ciascun pixel verrebbe rappresentato da un byte.

Purtroppo il nostro microcontrollore ha a disposizione solamente 16 KB di SRAM, di conseguenza una soluzione simile non è praticabile.

Per questo motivo abbiamo deciso di rappresentare lo schermo come un array unidimensionale di 1024 byte, dove ciascun pixel viene rappresentato da un singolo bit. In questo modo otteniamo un risparmio di spazio pari a ben l'87.5%.

Questa decisione ha aggiunto però un livello di indirezione dato che una coordinata ad alto livello sulla matrice 128x64 é mappata ad una coordinata "in memoria", dove la prima componente é l'indice del byte nell'array unidimensionale e la seconda componenete é il bit all'interno del byte. La figura [4] mostra un esempio di mappatura di un pixel. Piú precisamente la funzione F é definita come segue:

$$F : \mathbb{N} \times \mathbb{N} \to \mathbb{N} \times \mathbb{N}$$
$$F(x, y) = \left(\left\lfloor \frac{128y + x}{8} \right\rfloor, 7 - (x \bmod 8) \right)$$
$$where \quad x \in [0, 127] \quad y \in [0, 63]$$

Un'ulteriore ottimizzazione viene resa disponibile attraverso l'API dell'interprete sotto forma di una funzione che consente al chiamante di controllare se l'array che rappresenta lo schermo è stato modificato

nell'ultimo ciclo di esecuzione. Avendo notato che il numero di opcode che modificano lo schermo è molto limitato, precisamente sono solo 7 su 45, questa funzione consente di ridurre di 6.42 volte il numero di volte in cui lo schermo viene aggiornato.

3.1.3 Comportamenti ambigui

Gli interpreti CHIP-8 e S-CHIP hanno sviluppato molteplici comportamenti ambigui nel corso degli anni. Questi cosiddetti "quirk" variano in base alle piattaforme per cui è stato sviluppato l'interprete. Ad esempio gli interpreti per calcolatrici HP48 presentano un comportamento leggermente diverso durante l'esecuzione delle istruzioni di SHIFT.

Questi comportamenti ambigui si propagano fino ai programmatori CHIP-8 che si appoggiano a quest'ultimi e scrivono videogiochi che non sono del tutto compatibili con interpreti più vecchi. Per evitare questa frammentazione è necessario supportare le piattaforme principali e i loro quirk.

Il nostro interprete supporta CHIP-8, CHIP-48, S-CHIP 1.0 e S-CHIP 1.1, in questo modo è in grado di eseguire la stragrande maggioranza dei videogiochi reperibili in rete.

3.2 Porting su STM32

3.2.1 Interfaccia con lo schermo

3.2.2 Gestori per le ulteriori periferiche

3.2.3 Menù di selezione

3.3 Architettura

4 Analisi del consumo energetico

5 Considerazioni finali

6 Sviluppi futuri

Aggiornare solo la sprite che è stata modificata.

References

[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.