Progetto Dino Game: Implementazione su FPGA DE1-SoC

Marco Losso

16 aprile 2025

Sommario

Il presente documento descrive l'implementazione di un gioco ispirato al "Chrome Dino Game" su una piattaforma FPGA DE1-SoC. Il progetto integra componenti hardware configurati tramite Platform Designer e un'applicazione software sviluppata in linguaggio C. Il sistema utilizza un display VGA per la visualizzazione grafica e un controller PS/2 per l'acquisizione degli input da tastiera, dimostrando l'efficace interazione tra componenti hardware e software in un sistema embedded. Vengono discusse le scelte progettuali, le ottimizzazioni implementate e le sfide affrontate durante lo sviluppo, evidenziando come l'architettura modulare adottata abbia permesso di realizzare un'applicazione interattiva con prestazioni ottimali.

1 Introduzione

Il progetto consiste nell'implementazione di un gioco ispirato al famoso "Chrome Dino Game" di Google su una piattaforma FPGA DE1-SoC. L'obiettivo è realizzare un sistema embedded completo che integri componenti hardware e software per creare un'esperienza di gioco interattiva e reattiva.

Il gioco presenta un dinosauro che deve saltare per evitare ostacoli (cactus) che si muovono verso di lui. Il giocatore controlla il dinosauro tramite tastiera PS/2, premendo la barra spaziatrice per saltare. Il punteggio aumenta man mano che il giocatore supera gli ostacoli, e la difficoltà aumenta progressivamente con l'incremento della velocità degli ostacoli.

La realizzazione di questo progetto ha richiesto competenze multidisciplinari:

- Progettazione hardware mediante l'utilizzo di Platform Designer
- Programmazione in linguaggio C per lo sviluppo del software di gioco
- Conoscenza dei protocolli di comunicazione (VGA, PS/2)
- Ottimizzazione delle prestazioni in un sistema con risorse limitate

Il documento è strutturato come segue: nella Sezione 2 viene descritta l'architettura hardware del sistema, nella Sezione 3 viene presentata l'implementazione software, nella Sezione 4 sono discusse le ottimizzazioni e le sfide affrontate, mentre le Sezioni 5 e 6 contengono rispettivamente le conclusioni e i possibili sviluppi futuri.

2 Architettura Hardware

2.1 Panoramica del Sistema

Il sistema hardware è basato sulla scheda FPGA DE1-SoC, che integra un processore ARM Cortex-A9 (HPS - Hard Processor System) e una parte FPGA programmabile. L'architettura è stata configurata utilizzando Platform Designer (precedentemente noto come Qsys) di Intel/Altera.

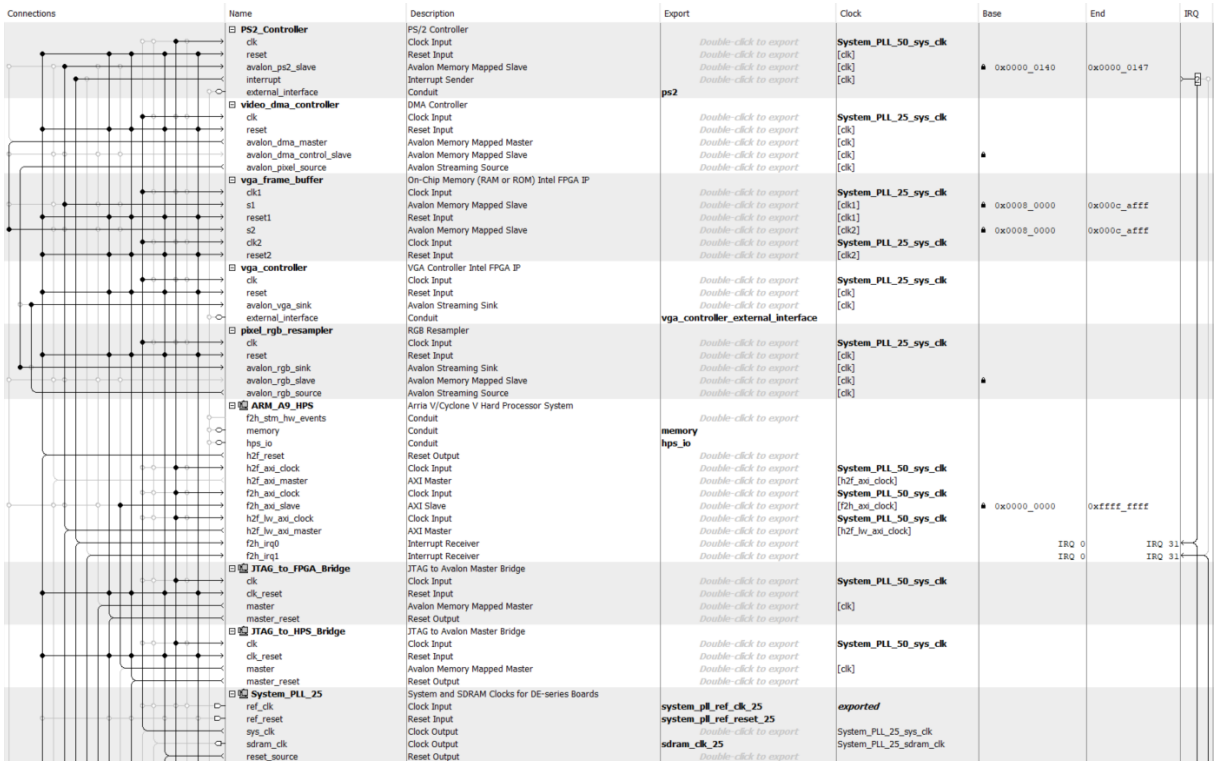

Figura 1: Schema del sistema in Platform Designer

2.2 Componenti Hardware

Il sistema include i seguenti componenti hardware principali:

2.2.1 Hard Processor System (HPS)

Il processore ARM Cortex-A9 dual-core integrato nella DE1-SoC costituisce il cuore del sistema. Operante a una frequenza di 800 MHz, esegue il software del gioco e coordina tutte le operazioni. L'HPS è connesso alla parte FPGA tramite un'interfaccia AXI (Advanced eXtensible Interface) che garantisce una comunicazione ad alta velocità tra i due domini.

L'HPS dispone di:

- 1 GB di memoria DDR3 SDRAM
- Controller per periferiche standard (USB, Ethernet, SD/MMC)
- Interfacce di comunicazione seriale (UART, SPI, I2C)

- Timer e contatori hardware

Nel contesto di questo progetto, l'HPS è responsabile dell'esecuzione del codice C che implementa la logica di gioco, il rendering grafico e la gestione degli input.

2.2.2 Controller VGA

Il controller VGA è implementato nella parte FPGA e gestisce la visualizzazione del gioco su un monitor esterno. È configurato per una risoluzione di 640x480 pixel con una profondità di colore di 8 bit, che permette di rappresentare fino a 256 colori simultaneamente. Il controller genera i segnali di sincronizzazione orizzontale e verticale necessari per il protocollo VGA, con una frequenza di refresh di 60 Hz.

Il controller VGA è stato implementato utilizzando un IP core disponibile nella libreria di Platform Designer, opportunamente configurato per le esigenze del progetto. La scelta di una profondità di colore di 8 bit rappresenta un compromesso ottimale tra qualità visiva e utilizzo delle risorse di memoria.

2.2.3 Frame Buffer VGA

Un buffer di memoria dedicata memorizza i dati dei pixel che vengono visualizzati sul display VGA. Il buffer è mappato nello spazio di indirizzamento del processore ARM, consentendo al software di scrivere direttamente i dati dei pixel.

2.2.4 Controller PS/2

Il controller PS/2 gestisce la comunicazione con la tastiera esterna, permettendo al giocatore di interagire con il gioco. Il controller è configurato per rilevare i codici di scansione della tastiera e renderli disponibili al software.

2.2.5 PLL (Phase-Locked Loop)

Due PLL sono utilizzati per generare i clock necessari al sistema:

- System_PLL_25: genera un clock a 25 MHz per il controller VGA
- System_PLL_50: genera un clock a 50 MHz per altri componenti del sistema

2.3 Interconnessioni

I componenti hardware sono interconnessi tramite bus Avalon-MM (Memory-Mapped). Questo permette al processore ARM di accedere ai vari periferici come se fossero locazioni di memoria.

2.4 Configurazione in Platform Designer

La configurazione del sistema in Platform Designer definisce:

- Gli indirizzi base di ciascun componente nello spazio di indirizzamento del processore
- Le interconnessioni tra i vari componenti
- I parametri di configurazione di ciascun componente
- I clock e i reset per ciascun componente

3 Implementazione Software

3.1 Struttura del Codice

Il software è stato progettato seguendo i principi della programmazione modulare, con una chiara separazione delle responsabilità tra i diversi componenti. Questa architettura facilita la manutenzione, il debug e l'estensione del codice. Il software è organizzato nei seguenti moduli funzionali:

- **Dino_Game_v0.8.c**: File principale che contiene la funzione main e coordina l'esecuzione del gioco. Gestisce l'inizializzazione del sistema, la selezione della modalità di gioco e il ciclo principale dell'applicazione.
- **core_utils.c**: Implementa funzioni di utilità generale come la gestione degli eventi, i ritardi temporizzati e la generazione di numeri casuali. Questo modulo fornisce anche le definizioni delle costanti di sistema e le strutture dati fondamentali utilizzate in tutto il progetto.
- **graphics.c**: Contiene le funzioni per il rendering grafico di base, come il disegno di pixel, linee, rettangoli e lo sfondo del gioco. Implementa l'interfaccia di basso livello per l'accesso al buffer VGA.
- **text_ui.c**: Fornisce funzioni specializzate per il rendering del testo e delle interfacce utente, come menu, punteggi e messaggi di stato. Implementa un sistema di font bitmap per la visualizzazione dei caratteri.
- **game_entities.c**: Definisce e gestisce le entità di gioco, come il dinosauro e gli ostacoli. Implementa le funzioni per il disegno, l'aggiornamento e la gestione delle collisioni tra le entità.
- **game_logic.c**: Contiene la logica principale del gioco, inclusi il loop di gioco, la gestione delle collisioni, l'aggiornamento del punteggio e la modalità autoplay. Coordina l'interazione tra le diverse entità di gioco.

Questa organizzazione modulare ha permesso di sviluppare e testare ciascun componente in modo indipendente, facilitando l'individuazione e la correzione di errori. Inoltre, ha reso possibile l'implementazione incrementale delle funzionalità, partendo da un prototipo minimo fino alla versione completa del gioco.

3.2 Accesso all'Hardware

Il software accede ai componenti hardware tramite puntatori a indirizzi specifici:

```
// Puntatori ai registri hardware
volatile uint8_t *vga_buffer = (volatile uint8_t *)VGA_BUFFER_BASE;
volatile uint32_t *ps2_base = (volatile uint32_t *)PS2_BASE;
```

Listing 1: Accesso all'hardware in Dino_Game_v0.8.c

Dove `VGA_BUFFER_BASE` e `PS2_BASE` sono definiti in `core_utils.h` e corrispondono agli indirizzi base dei rispettivi componenti hardware.

3.3 Rendering Grafico

Il rendering grafico è implementato scrivendo direttamente nel buffer VGA. La funzione principale per questo scopo è `set_pixel`:

```
void set_pixel(volatile uint8_t *vga_buffer, int x, int y, uint8_t color
    ) {
    // Calcola l'offset usando lo stride di 1024
    int offset = (y << 10) + x;  // equivalente a y * 1024 + x

    // Verifica che le coordinate siano valide
    if (x >= 0 && x < VGA_WIDTH && y >= 0 && y < VGA_HEIGHT) {
        // Scrivi il colore nel buffer
        vga_buffer[offset] = color;
    }
}
```

Listing 2: Funzione set_pixel in graphics.c

3.4 Interfaccia Utente e Stati di Gioco

Il gioco presenta tre stati principali, ciascuno con una specifica interfaccia utente:

Figura 2: Schermata iniziale del gioco con modalità autoplay. Il dinosauro salta automaticamente gli ostacoli mentre il messaggio "PREMERE SPAZIO PER INIZIARE" invita il giocatore ad avviare la partita.

La gestione di questi diversi stati di gioco è implementata attraverso una macchina a stati finiti nel file principale:

```
// Definizione degli stati di gioco
typedef enum {
    STATE_TITLE,     // Schermata iniziale con autoplay
    STATE_PLAYING,   // Gioco in corso
    STATE_GAME_OVER  // Schermata di Game Over
} GameState;

// Nel main loop
GameState current_state = STATE_TITLE;

while (1) {
    switch (current_state) {
```

Figura 3: Screenshot del gioco in esecuzione. Il dinosauro salta per evitare gli ostacoli mentre il punteggio (15) e le vite rimanenti (2 cuori) sono visualizzati nella parte superiore dello schermo.

Figura 4: Schermata di "GAME OVER" che appare quando il giocatore perde tutte le vite. Il punteggio finale (42) viene visualizzato nella parte superiore dello schermo.

```
13          case STATE_TITLE:
14              // Mostra schermata iniziale con autoplay
15              show_title_screen(vga_buffer, ps2_base);
16              // Passa allo stato di gioco quando viene premuto spazio
17              if (check_space_pressed(ps2_base)) {
18                  current_state = STATE_PLAYING;
19                  init_game(&score, &lives);
20              }
21              break;
22
23          case STATE_PLAYING:
24              // Esegui il loop di gioco principale
25              game_result = game_loop(vga_buffer, ps2_base, &score, &lives
    );
26              // Se il gioco è terminato, passa allo stato di Game Over
27              if (game_result == GAME_OVER) {
28                  current_state = STATE_GAME_OVER;
29              }
30              break;
```

```
31
32          case STATE_GAME_OVER:
33              // Mostra la schermata di Game Over
34              show_game_over(vga_buffer, ps2_base, score);
35              // Torna alla schermata iniziale quando viene premuto spazio
36              if (check_space_pressed(ps2_base)) {
37                  current_state = STATE_TITLE;
38              }
39              break;
40      }
41  }
```

Listing 3: Gestione degli stati di gioco in Dino_Game_v0.8.c

Nella Figura 3 è possibile osservare il gioco in esecuzione, con il dinosauro che salta per evitare gli ostacoli. L'interfaccia utente mostra il punteggio nella parte superiore dello schermo e le vite rimanenti rappresentate da cuori.

3.5 Gestione degli Input

Gli input dalla tastiera PS/2 sono gestiti leggendo i codici di scansione dal registro del controller PS/2:

```
1  void check_input(volatile uint32_t *ps2_base) {
2      // Controlla se ci sono dati disponibili dal PS2
3      if (*ps2_base & 0x8000) {  // Bit 15 indica dati disponibili
4          uint8_t scan_code = *ps2_base & 0xFF;  // Prendi i primi 8 bit
5
6          // Consuma il dato
7          volatile uint32_t dummy = *ps2_base;
8          (void)dummy; // Evita "unused variable warning"
9
10         // Svuota completamente il buffer PS/2 prima di procedere
11         while (*ps2_base & 0x8000) {
12             dummy = *ps2_base;
13         }
14
15         // Gestione semplificata
16         if (scan_code == 0x29) {  // PS2_SPACE
17             // Aggiungi un evento di salto alla coda
18             add_event(EVENT_JUMP, 0);
19         }
20     }
21 }
```

Listing 4: Funzione check_input in game_logic.c

3.6 Logica di Gioco

La logica principale del gioco è implementata nella funzione `game_loop` in `game_logic.c`. Questa funzione gestisce:

- L'inizializzazione delle entità di gioco
- Il loop principale che aggiorna lo stato del gioco
- La gestione delle collisioni

- L'aggiornamento del punteggio
- La gestione delle vite del giocatore

3.7 Gestione degli Ostacoli

Gli ostacoli sono generati casualmente e si muovono da destra a sinistra sullo schermo. La loro velocità aumenta con l'aumentare del punteggio:

```
// Aggiorna la posizione degli ostacoli
// Aumenta la velocità in base al punteggio
int current_speed = obstacle_speed;
if (score > 10) current_speed = 4;
if (score > 20) current_speed = 5;
if (score > 30) current_speed = 6;

update_obstacles(vga_buffer, obstacles, MAX_OBSTACLES, colors[5],
    current_speed, &score);
```

Listing 5: Aggiornamento degli ostacoli in game_logic.c

3.8 Meccanica del Salto

Il salto del dinosauro è implementato come una parabola, con una fase di salita e una di discesa:

```
// Calcola la nuova posizione usando una parabola
// y = h * (1 - 4 * (x - 0.5)^2) dove x va da 0 a 1
float x_param = progress - 0.5f;
float y_offset = jump_height * (1.0f - 4.0f * x_param * x_param);

// Aggiorna la posizione y
*y = original_y - (int)y_offset;
```

Listing 6: Aggiornamento del salto in game_entities.c

3.9 Modalità Autoplay

Il gioco include anche una modalità di autoplay, in cui il dinosauro salta automaticamente gli ostacoli. Questa modalità è utilizzata nella schermata iniziale:

```
void auto_jump(Obstacle obstacles[], int max_obstacles, int dino_x, int
    dino_size, int *is_jumping, int *jump_phase, int *jump_start_time) {
    // Se il dinosauro sta già saltando, non fare nulla
    if (*is_jumping) return;

    // Cerca l'ostacolo più vicino davanti al dinosauro
    for (int i = 0; i < max_obstacles; i++) {
        if (obstacles[i].active) {
            // Calcola la distanza tra il dinosauro e l'ostacolo
            int distance = obstacles[i].x - (dino_x + dino_size/2);

            // Se l'ostacolo è abbastanza vicino, inizia un salto
            if (distance > 0 && distance < 30) {
                start_jump(is_jumping, jump_phase, jump_start_time);
                break;
            }
```

```
16            }
17        }
18    }
```

Listing 7: Funzione auto_jump in game_logic.c

4 Ottimizzazioni e Sfide

4.1 Ottimizzazioni di Prestazioni

Per garantire un'esecuzione fluida del gioco su un sistema con risorse limitate, sono state implementate diverse strategie di ottimizzazione:

- **Ottimizzazioni aritmetiche**: L'utilizzo di operazioni bit-shift invece di moltiplicazioni dove possibile ha ridotto significativamente il carico computazionale. Ad esempio, la moltiplicazione per 1024 nel calcolo dell'offset del buffer VGA è stata sostituita con uno shift a sinistra di 10 bit (y « 10).
- **Rendering selettivo**: Invece di ridisegnare l'intero schermo ad ogni frame, vengono aggiornate solo le aree che cambiano. Questo approccio riduce drasticamente il numero di scritture nel buffer VGA, migliorando le prestazioni complessive.
- **Gestione efficiente della memoria**: Le strutture dati sono state progettate per minimizzare l'utilizzo della memoria e ottimizzare l'accesso ai dati. Ad esempio, gli ostacoli inattivi vengono riutilizzati invece di allocare nuove istanze.
- **Coda degli eventi**: L'implementazione di una coda degli eventi permette di gestire gli input in modo asincrono, evitando che il gioco si blocchi in attesa di un input dell'utente.
- **Calibrazione dei ritardi**: I ritardi nel loop di gioco sono stati attentamente calibrati per garantire un framerate stabile senza sovraccaricare la CPU.

Queste ottimizzazioni hanno permesso di ottenere un'esperienza di gioco fluida con un framerate costante, anche durante le fasi più intense del gioco.

4.2 Sfide Incontrate

Durante lo sviluppo del progetto, sono state affrontate e superate diverse sfide tecniche:

- **Sincronizzazione**: La sincronizzazione tra il rendering grafico e la logica di gioco ha rappresentato una sfida significativa. È stato necessario implementare un sistema che garantisse l'aggiornamento coerente dello stato di gioco e della sua rappresentazione visiva, evitando artefatti grafici.
- **Gestione delle collisioni**: L'implementazione di un algoritmo di rilevamento delle collisioni efficiente ha richiesto un attento bilanciamento tra precisione e prestazioni. La soluzione adottata utilizza bounding box semplificate per ridurre il carico computazionale mantenendo una buona accuratezza.

- **Sistema di salto non bloccante**: La realizzazione di un sistema di salto che non bloccasse l'esecuzione del gioco ha richiesto l'implementazione di una macchina a stati che gestisse le diverse fasi del salto in modo incrementale, permettendo l'aggiornamento simultaneo di altre entità di gioco.
- **Latenza degli input**: La gestione della latenza degli input dalla tastiera PS/2 ha richiesto l'implementazione di tecniche di buffering e polling ottimizzate per garantire una risposta reattiva ai comandi del giocatore.
- **Limitazioni hardware**: L'adattamento del gioco alle limitazioni hardware della piattaforma FPGA ha richiesto compromessi in termini di complessità grafica e numero di entità simultanee, mantenendo comunque un'esperienza di gioco soddisfacente.

Il superamento di queste sfide ha contribuito significativamente alla qualità finale del progetto, dimostrando l'efficacia dell'approccio di progettazione adottato.

5 Conclusioni

Il progetto Dino Game dimostra l'efficace integrazione di componenti hardware e software in un sistema embedded completo. L'utilizzo di Platform Designer ha permesso di configurare in modo flessibile i componenti hardware necessari, mentre la struttura modulare del codice C ha facilitato lo sviluppo incrementale e il debug del software.

Il gioco implementa tutte le funzionalità principali del Chrome Dino Game originale, con l'aggiunta di alcune caratteristiche innovative come il sistema di vite multiple e la modalità autoplay. Le ottimizzazioni implementate garantiscono un'esperienza di gioco fluida e reattiva, nonostante le limitazioni intrinseche della piattaforma hardware utilizzata.

Dal punto di vista didattico, il progetto ha permesso di applicare e integrare conoscenze multidisciplinari:

- Progettazione di sistemi digitali su FPGA
- Programmazione embedded in C
- Interfacciamento con periferiche esterne (VGA, PS/2)
- Ottimizzazione delle prestazioni in sistemi con risorse limitate
- Sviluppo di applicazioni interattive in tempo reale

I risultati ottenuti dimostrano che è possibile realizzare applicazioni interattive complesse su piattaforme FPGA, sfruttando l'integrazione tra la flessibilità della logica programmabile e la potenza di elaborazione di un processore embedded. Questo approccio ibrido offre vantaggi significativi in termini di prestazioni, flessibilità e consumo energetico rispetto a soluzioni puramente software o hardware.

In conclusione, il progetto Dino Game rappresenta un esempio concreto di come le moderne piattaforme FPGA possano essere utilizzate per implementare sistemi embedded completi, combinando efficacemente componenti hardware dedicati e software applicativo.

6 Possibili Miglioramenti Futuri

Ci sono diverse possibilità di miglioramento per il progetto:

- Aggiunta di effetti sonori tramite un DAC audio
- Implementazione di diversi tipi di ostacoli (uccelli volanti, cactus di diverse dimensioni)
- Aggiunta di una modalità a due giocatori
- Implementazione di un sistema di salvataggio dei punteggi più alti
- Miglioramento della grafica con sprite più dettagliati