Tetris su FPGA
Progetto per l'esame di Architetture Digitali

Erik Calligari, Luca Guerra

25 luglio 2013

Indice

1 Introduzione **2**
- 1.1 Il gioco del Tetris . . . 3
- 1.2 Hardware FPGA . . . 4

2 Struttura del progetto **4**
- 2.1 Componenti logici . . . 6
- 2.2 Convenzioni VHDL utilizzate . . . 7
  - 2.2.1 Macchine a stati finiti e registri . . . 7
  - 2.2.2 Assegnamenti multipli a segnali . . . 8

3 Memoria **10**

4 Grafica **11**
- 4.1 Segnali VGA e loro temporizzazione . . . 11
- 4.2 Generazione della grafica . . . 13
- 4.3 Griglia di gioco . . . 15
- 4.4 Font . . . 17
- 4.5 Indicatore Next Piece . . . 19

5 Gestione dell'input **20**

6 Logica di gioco **22**
- 6.1 Tetris Logic . . . 22
- 6.2 Movimento del tetramino . . . 27
- 6.3 Eliminazione delle linee piene . . . 31
- 6.4 Controllo delle linee . . . 34
- 6.5 Spostamento delle linee . . . 35
- 6.6 Punteggio . . . 37
- 6.7 Convertitore da binario a BCD . . . 39
- 6.8 Reset della griglia . . . 40
- 6.9 Generazione di numeri casuali . . . 41

7 Musica **43**

8 Conclusioni e sviluppi futuri **46**

A Note tecniche/operative sulla scheda e sull'ambiente Xilinx ISE **47**

1 Introduzione

Il progetto che proponiamo consiste nella realizzazione di un dispositivo hardware basato su Field Programmable Gate Array (FPGA) che implementi il noto videogioco *Tetris* in linguaggio VHDL. Il progetto, mostrato in Figura 1, prevede lo sviluppo del gioco su una board Xilinx Spartan-3, il collegamento della stessa a un monitor VGA per l'output video, a un gamepad per l'acquisizione dell'input del giocatore e a un altoparlante per l'output audio.

Figura 1: Tetris su FPGA

1.1 Il gioco del Tetris

Tetris è un videogioco inventato nel 1984 dal programmatore russo Aleksej Pažitnov. Il gioco fu in origine programmato in Pascal su hardware Electronika 60 e successivamente portato da Vadim Gerasimov su IBM PC con piattaforma MS-DOS. Nei successivi anni il gioco acquisì incredibile popolarità e fu sviluppato per una grande varietà di piattaforme, tra cui *Commodore 64*, *Amiga*, *Atari ST*, *Apple II* e *SEGA System 16* arcade. Una delle versioni che contribuì maggiormente a diffondere il gioco fu però sicuramente quella per *Nintendo GameBoy*, che si trovava inclusa nella confezione della prima edizione della nota console portatile. A oggi il gioco, che vanta oltre 70 milioni di copie vendute, è considerato uno dei videogame più giocati di sempre e si trova spesso inserito nei pacchetti giochi preinstallati in diversi sistemi operativi, console da gioco, dispositivi mobili o addirittura come *easter egg* in vari prodotti software e hardware tra cui editor di testo, calcolatrici programmabili e oscilloscopi.

Regole

Il gioco si svolge su una griglia rettangolare divisa in celle quadrate. La dimensione effettiva della griglia varia da versione a versione e si aggira normalmente attorno alle 10 celle di lunghezza e 20 celle di altezza. All'inizio del gioco la griglia è completamente vuota e viene introdotto un *tetramino* al centro della griglia che occupa quattro celle comprese nelle prime due righe dall'alto. Un tetramino è una figura geometrica composta da quattro quadrati connessi tra di loro lungo i lati. Di seguito si indicherà come tetramino uno dei sette del gioco del *Tetris* rappresentati in Figura 2, ognuno contraddistinto da una lettera tra I, Z, T, L, S, J, O.

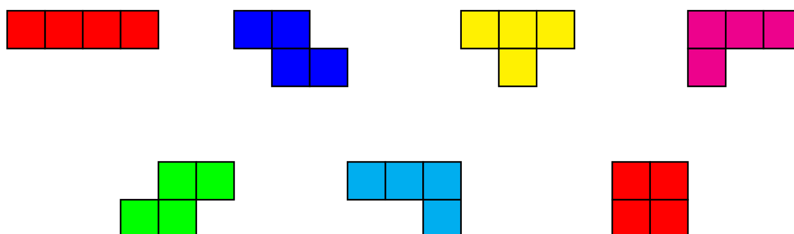


Figura 2: I differenti tetramini nel gioco del Tetris

Il giocatore, agendo su appositi tasti, può muovere il tetramino a destra o a sinistra sulla griglia, muoverlo rapidamente verso il basso oppure ruotarlo. Ogni tipo di tetramino ruota in modo diverso, eccezion fatta per O che non può ruotare. Il gioco fa inoltre cadere automaticamente il tetramino verso il basso con una certa velocità prefissata, indipendentemente dai comandi del giocatore.

Quando non è più possibile muovere il tetramino verso il basso, ossia quando un ulteriore movimento verso il basso farebbe passare il tetramino oltre il bordo inferiore della griglia o lo porterebbe a sovrapporsi con almeno una cella occupata, il tetramino si ferma. Da quel momento non lo si potrà più muovere in nessuna direzione e le corrispondenti celle sulla griglia rimarranno occupate. Il gioco introduce quindi un altro tetramino controllabile dal giocatore in cima alla griglia e il gioco continua.

Ogni volta che un tetramino si ferma il gioco cancella ogni linea piena, cioè ogni riga della griglia le cui celle sono tutte occupate, e fa scorrere tutte le righe soprastanti di una cella verso il basso.

Ogni volta che vengono eliminate una o più linee dalla griglia il gioco aumenta il punteggio del giocatore a seconda del numero di linee eliminate in un colpo solo e del livello di gioco. Il livello di gioco è impostato inizialmente a 0 e viene incrementato di una unità ogni 10 linee eliminate. Maggiore è il livello, più saranno i punti che il giocatore guadagna per ogni linea eliminata e maggiore sarà la velocità di caduta automatica del tetramino.

Se, una volta che un tetramino si ferma e tutte le righe piene sono state cancellate, non è possibile introdurre altri tetramini in cima alla griglia in quanto le corrispondenti celle risultano occupate, il gioco

termina con *game over*. Lo scopo del gioco risulta quindi accumulare più punti possibile prima del *game over*.

1.2 Hardware FPGA

Per lo sviluppo del progetto abbiamo utilizzato una scheda FPGA. FPGA sta per Field Programmable Gate Array e rappresenta una categoria particolare di Programmable Logic Device (PLD), ossia circuito logico combinatorio *programmabile*. A differenza di un microprocessore, che è in grado di eseguire un programma su hardware fissato, un PLD è programmabile a livello hardware, cioè è in grado di modificare il suo funzionamento a livello di circuito logico a seconda delle specifiche che gli vengono fornite dal programmatore. Un chip FPGA è normalmente costituito da una matrice di CLB (Configurable Logic Block) che a loro volta contengono una serie di componenti elettronici digitali di base, tra cui flip-flop e LUT (LookUp Table). In fase di programmazione, la FPGA collegherà tra loro i vari CLB e i componenti al loro interno in modo da sintetizzare il circuito specificato. I chip e le schede su cui si programmano FPGA sono inoltre comunemente dotati di altri componenti elettronici analogici e digitali, tra cui moltiplicatori di clock (PLL), memoria SRAM, connettori, DAC (Digital to Analog Converter), uno o più oscillatori al quarzo che hanno funzione di clock, circuiti e interfacce per la programmazione del chip eccetera.

La programmazione del chip FPGA è normalmente memorizzata su memoria volatile (RAM), ciò significa che la scheda si reimposterà da sola ogni volta che viene disconnessa dall'alimentazione elettrica. Alcune schede sono però dotate di memoria non-volatile (EEPROM, Flash, ecc.) che consente di salvare la programmazione della FPGA in modo che all'accensione carichi automaticamente in RAM la configurazione scelta senza che sia necessario riprogrammare la scheda via software ad ogni uso.

La scheda FPGA che abbiamo utilizzato è una Xilinx Spartan-3 XC3S200, fornita di 200000 porte logiche, 480 CLB per un totale di 4320 celle logiche (ognuna equivalente a un flip-flop e LookUp Table a 4 bit), 30Kbit di DRAM (Distributed RAM) e 216Kbit di block RAM. Per ulteriori dettagli tecnici sulla scheda invitiamo a consultare [1].

Il dispositivo da sintetizzare sulla scheda è stato scritto in linguaggio VHDL (VHSIC Hardware Definition Language) con l'uso delle librerie standard IEEE e alcuni blocchi logici precostituiti Xilinx CORE su ambiente di sviluppo Xilinx ISE.

2 Struttura del progetto

Il progetto concluso si presenta come una scheda FPGA connessa a un monitor tramite presa VGA e a un gamepad per *SEGA Mega Drive*. Inizialmente il gioco mostra la griglia, di dimensioni 10×18, sullo schermo e attende che il giocatore prema il tasto *START* sul controller per avviare una nuova partita. Il contenuto della griglia in questo stato di gioco non è rilevante e serve esclusivamente come *splash screen* prima del gioco. Quando il giocatore preme *START*, la griglia si svuota, un tetramino viene posizionato in cima ad essa e la partita ha inizio. Il giocatore può muovere il tetramino con i tasti *giù*, *destra* e *sinistra* del controller e lo può ruotare con i tasti *su*, *B* o *C*. Ogni volta che un nuovo tetramino viene introdotto nella griglia un riquadro sullo schermo (marcato *NEXT*) visualizza il tetramino successivo. A conclusione della partita il gioco mostra una scritta *GAME OVER* a coprire il centro della griglia e attende che il giocatore prema nuovamente *START* se desidera giocare un'altra partita.

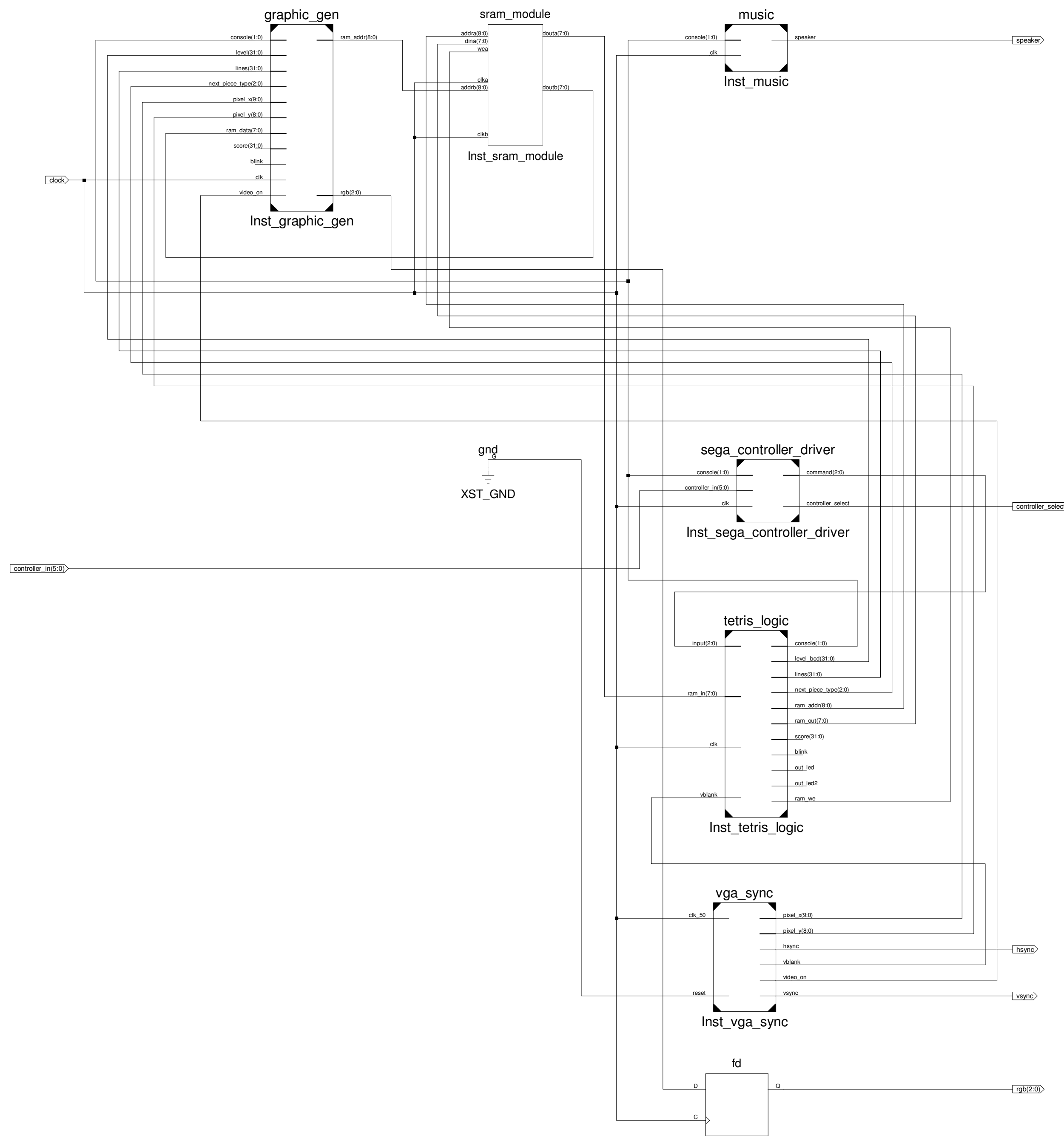


Figura 3: Struttura dei moduli

2.1 Componenti logici

Passiamo ora a osservare i componenti che, una volta sintetizzati su FPGA, realizzano il progetto appena presentato. Ogni componente è descritto da uno o più file VHDL presenti nell'archivio dei sorgenti, a cui faremo riferimento durante la presente discussione citando il nome di ogni modulo, reperibile nel file `<nome modulo>.vhd`. Ogni modulo contiene una singola *entity* VHDL e una singola *architecture*, dotate del medesimo nome del modulo che le contiene. Ogni volta che ci riferiamo a un determinato modulo ci riferiamo contemporaneamente al file, all'istanza dell'*entity* corrispondente e alle eventuali *entity* istanziate all'interno del modulo stesso.

Dando uno sguardo d'insieme al progetto (modulo `tetris` illustrato in Figura 3) si può operare una suddivisione dei moduli in cinque componenti principali, che descriviamo di seguito.

Memoria
: Consiste nel modulo `sram_module` e si interfaccia con i chip SRAM presenti sulla scheda, gestendone opportunamente la temporizzazione. Abbiamo deciso di utilizzare la memoria RAM esclusivamente per memorizzare lo stato della griglia di gioco. Vista questa decisione progettuale, il compito del modulo di memoria sarà quello di fornire bus indirizzo e dati da e verso il modulo che gestisce la logica del gioco (`tetris_logic`) per consentirgli di leggere lo stato del gioco e aggiornarlo quando opportuno e verso uno dei moduli che gestisce l'output grafico (`graphic_gen`) per consentire al gioco di visualizzare lo stato della griglia su schermo.

Grafica
: Consiste nei moduli `graphic_gen`, `vga_sync` e nel registro `rgb_reg`. Si occupa di gestire i segnali in uscita sulla VGA, garantendo il rispetto delle specifiche di temporizzazione (compito svolto sostanzialmente da `vga_sync` e motivo dell'esistenza del buffer `rgb_reg`) e il disegno delle varie aree della schermata di gioco che includono la griglia, il testo, il totale dei punti, delle linee eliminate, il livello corrente, il pezzo successivo e varie etichette testuali (`graphic_gen`). Quest'ultimo modulo, oltre a necessitare di essere collegato a `sram_module` per leggere lo stato della griglia riceve anche alcuni segnali di stato dalla logica del gioco, che rappresentano tutte le informazioni necessarie alla corretta visualizzazione grafica del gioco che non sono memorizzate in RAM.

Input
: Consiste nel modulo `sega_controller_driver` che si occupa di gestire il collegamento con il gamepad. È collegato direttamente ai pin di dati e segnali di controllo presenti sul connettore del gamepad e fornisce in uscita un segnale che rappresenta il comando che il giocatore desidera impartire al gioco (avvio partita, destra, sinistra o rotazione), necessario al modulo che gestisce la logica per modificare lo stato di gioco in risposta ai comandi del giocatore.

Logica di gioco
: Si tratta sicuramente del componente più complesso del sistema ed è composto da diversi moduli che fanno capo a `tetris_logic`. Si occupa di implementare effettivamente le regole di Tetris, reagendo correttamente all'input del giocatore e gestendo tutte le modifiche di stato automatiche, nonché il punteggio, la velocità di gioco, i livelli, lo stato di *game over* e altre caratteristiche. È dotato di accesso in lettura e scrittura alla SRAM, riceve in ingresso il comando letto dal modulo di gestione dell'input, segnali di controllo del modulo di generazione della grafica (`graphic_gen`) e fornisce in output vari segnali di stato che riguardano lo stato della partita, il numero di linee eliminate, il punteggio e il tipo del prossimo tetramino.

Musica
: Composto dal modulo `music` che si occupa di gestire il segnale audio in uscita sullo speaker per consentire la riproduzione della musica durante la partita.

Tutti i moduli sopra citati sono inoltre sincronizzati con lo stesso clock che lavora alla frequenza di 50 MHz presente sulla Spartan-3.

Nelle prossime sezioni passiamo a descrivere in dettaglio ognuno dei componenti appena citati, motivando le scelte architetturali e implementative operate per ciascuno di essi.

2.2 Convenzioni VHDL utilizzate

2.2.1 Macchine a stati finiti e registri

Come si può immaginare, un progetto del genere fa largo uso di macchine a stati finiti e registri. Durante le fasi iniziali di progettazione abbiamo deciso di adottare una convenzione specifica in tutto il codice per la gestione di questi due aspetti.

Per quanto riguarda le macchine a stati abbiamo preferito il primo approccio descritto a pagine 160 e seguenti di [2]. Per ogni macchina, definiamo una *upper section*, costituita da un *process* combinatorio che si occupa di assegnare il segnale di stato prossimo a seconda dello stato attuale, dell'input e di tutti gli altri segnali che concorrono ad effettuare la decisione, e una *lower section*, *process* sequenziale che si occupa esclusivamente di assegnare al valore di stato attuale il valore in uscita dalla upper section sul fronte di salita del clock. Secondo lo standard VHDL, questa operazione specifica la generazione di un registro (composto da uno o più flip-flop), ottenendo effettivamente una struttura simile a quella mostrata in Figura 4: una upper section puramente combinatoria e un insieme di flip-flop che formano la memoria della macchina. Di seguito si mostra un esempio di questa struttura in linguaggio VHDL.

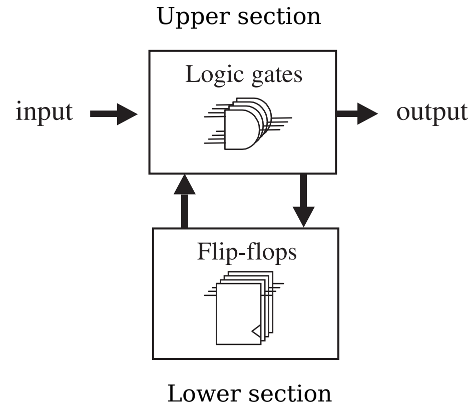


Figura 4: Schema implementativo delle macchine a stati

Come convenzione di *naming* abbiamo deciso di contraddistinguere con il suffisso `_reg` il segnale che corrisponde allo stato attuale del registro, mentre `_next` rappresenta il segnale che trasporta il valore del registro che verrà memorizzato al successivo fronte di salita del clock. Ad ogni macchina a stati inoltre corrisponde un tipo enumerativo che elenca tutti i possibili stati della macchina. Di seguito presentiamo un esempio di macchina con tre stati la cui unica funzione è quella di passare da uno all'altro ad ogni colpo di clock scritta seguendo le convenzioni scelte.

```
...
-- definizione di tipi e segnali nell'architecture

type state is (idle, working, done);

signal state_reg, state_next : state := idle;
...

-- upper section: logica combinatoria che definisce la funzione stato prossimo
process (state_reg)
begin
  case state_reg is
    when idle =>
      state_next <= working;
    when working =>
      state_next <= done;
    when done =>
      state_next <= idle;
  end case;
end process;

-- lower section: generazione flip-flop
process (clock)
begin
  if rising_edge(clock) then
    state_reg <= state_next;
  end if;
end process;
```

Abbiamo deciso di utilizzare la stessa convenzione per tutti i registri che utilizziamo, anche se non facenti parte dello stato di una macchina a stati. I nomi seguiranno quindi la stessa regola dell'utilizzo del suffisso `_reg` e `_next` e, in ogni modulo che fa uso di registri, vi sarà sempre un *process* analogo alla lower section appena esposta. Questo crea sicuramente la necessità di introdurre più codice, ma aiuta a esplicitare la presenza di flip-flop nei vari moduli distinguendoli a "colpo d'occhio" da puri segnali combinatori, ci evita di scrivere *process* in cui lo stesso segnale viene contemporaneamente assegnato e utilizzato in ingresso e fa sì che Xilinx ISE emetta *warning* quando, per esempio, non si specifica il valore del segnale `next` in tutti i rami alternativi di un costrutto `case` o `if`. In quest'ultimo caso, se si utilizzasse un segnale solo per descrivere il registro, verrebbe mantenuto con lo stesso valore nel caso in cui non si specifichi completamente il costrutto. Chiaramente però a volte si richiede esplicitamente questo comportamento. Nel prossimo paragrafo descriviamo come è possibile ottenere in maniera semplice anche questo effetto.

2.2.2 Assegnamenti multipli a segnali

Di norma, all'interno di un *process* VHDL non si assegna più volte lo stesso segnale. Come entità logica infatti il *process* rappresenta un circuito combinatorio che dunque può emettere un solo valore per una singola uscita.

In VHDL, però, se si assegna più volte lo stesso segnale all'interno dello stesso *process*, *solo l'ultimo assegnamento avrà effettivamente valore*. Se, per esempio, si assegna un segnale prima di un costrutto `if` e poi lo si assegna all'interno del ramo `then` e non del ramo `else`, il valore in quest'ultimo caso sarà quello specificato prima del costrutto. Il prossimo esempio presenta un caso di due process equivalenti in cui l'uso di questa tecnica rende più compatta la rappresentazione VHDL ma non modifica il circuito.

```
-- assegnazione esplicita
-- di tutti i segnali
process (sig_a, sig_d)
begin
  case sig_a is
    when 0 =>
      sig_b <= 3;
      sig_c <= 4;
    when 1 =>
      sig_b <= 3;
      sig_c <= 5;
    when 2 =>
      sig_b <= 4;
      sig_c <= 4;
    when others=>
      if sig_d > 3 then
        sig_b <= 3;
        sig_c <= 4;
      elsif sig_d < 2 then
        sig_b <= 3;
        sig_c <= 6;
      else
        sig_b <= 3;
        sig_c <= 5;
      end if;
    end case;
end process;
```

```
-- scrittura equivalente
process (sig_a, sig_d)
begin
  sig_b <= 3;
  sig_c <= 4;
  case sig_a is
    when 0 =>
      null;
    when 1 =>
      sig_c <= 5;
    when 2 =>
      sig_b <= 4;
    when others=>
      if sig_d < 2 then
        sig_c <= 6;
      else
        sig_c <= 5;
      end if;
    end case;
end process;
```

3 Memoria

La scheda Spartan-3 è dotata di un megabyte di SRAM asincrona, montata sul retro della scheda, divisa su due chip IS61LV25616AL-10T ciascuno in grado di memorizzare 256K word da 16 bit.

Come già accennato, abbiamo deciso di utilizzare la memoria SRAM esclusivamente per memorizzare la griglia di gioco, visto che si tratta senza dubbio del dato che richiede più spazio di memorizzazione in termini di bit in tutto il progetto e, come si vedrà in seguito, si presta bene a essere gestita in modo sequenziale come richiesto in un'implementazione basata sull'uso della RAM per salvare e caricare dati da e verso registri. La griglia è composta da 10×18 celle, più due righe non visibili sopra il bordo superiore della griglia. Ogni cella può essere colorata in rosso, verde, blu, ciano, magenta o giallo oppure essere vuota. Una cella colorata può inoltre venire marcata dalla logica del gioco come "ferma" oppure "in movimento". Ogni cella richiederà quindi 3 bit per il colore più un bit per il flag di movimento, raggiungendo un totale di 4 bit. La griglia, nel complesso, occupa quindi 800 bit di memoria. Vista la quantità di memoria a nostra disposizione abbiamo deciso di utilizzare come indirizzo di ogni cella in memoria la concatenazione della rappresentazione binaria delle sue coordinate y e x. Per rappresentare coordinate da $(0, 0)$ a $(9, 19)$ occorreranno quindi 9 bit (4 per la x e 5 per la y). Si deve dunque allocare uno spazio di $16 \times 32 = 512$ word in RAM per il salvataggio della griglia. Per ogni word si usano solo i 4 bit meno significativi e si usano solo $20 \times 10 = 200$ word su 512. Risulta evidente che la maggior parte della memoria allocata non viene poi utilizzata, ma un layout di questo genere ha l'indubbio vantaggio di semplificare molto il circuito di selezione dell'indirizzo RAM e di caricamento della cella. Non avendo altre informazioni da memorizzare sulla SRAM questa scelta non ha alcuno svantaggio dal punto di vista del progetto. In fase di testing abbiamo potuto verificare che il tempo richiesto ad effettuare tutti gli accessi di lettura e scrittura rientra abbondantemente nel margine temporale previsto per la gestione della logica di gioco, che tratteremo più avanti, e quindi non introduce ritardi su altri componenti.

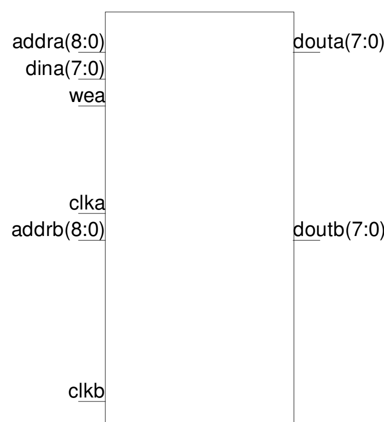


Figura 5: `sram_module`

In fase di analisi abbiamo inoltre valutato una possibile implementazione basata su registri. Utilizzare registri anzichè RAM per memorizzare la griglia avrebbe portato sicuramente un forte, per quanto non apprezzabile ai fini del gioco, incremento di velocità per quanto riguarda le operazioni svolte sulla griglia (in particolare l'identificazione delle linee piene, cancellazione e spostamento verso il basso delle linee) e una semplificazione della descrizione del circuito che si occupa di eseguire tali operazioni al costo di un elevato uso di risorse hardware, in particolare in termini di registri e lookup table. Nello specifico, la sola griglia necessiterebbe di 800 flip-flop, quando l'intero progetto, così come è concluso, ne usa approssimativamente lo stesso numero.

Per quanto riguarda l'implementazione del modulo di gestione della memoria `sram_module`, mostrato in Figura 5, abbiamo utilizzato il tool Xilinx CORE IP Generator per creare un modulo *dual-port SRAM* sincrono che fornisce una porta in lettura e scrittura (composta dai bus dati e indirizzo `douta`, `dina` e `addra`) verso il modulo di gestione della logica del gioco più una porta in sola lettura (composta dai bus dati e indirizzo `doutb`, e `addrb`) verso il modulo di generazione della grafica di gioco. Internamente, il modulo SRAM gestisce la temporizzazione dei chip di memoria asincroni e consente di effettuare ogni operazione di lettura e scrittura su entrambe le porte in un ciclo di clock (a 50 MHz).

4 Grafica

Il dispositivo, una volta avviato, presenta vari elementi a schermo. In dettaglio:

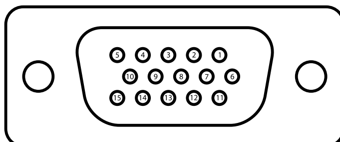
1. la griglia di gioco;
2. la casella che contiene il tetramino successivo che verrà posizionato sulla griglia;
3. il contatore dei punti;
4. il contatore delle linee eliminate;
5. l'indicatore del livello;
6. alcune etichette testuali statiche per le istruzioni e i *credit*.

Non appena si avvia la scheda e dopo un *game over* si vede anche una scritta *PRESS START* lampeggiante che invita il giocatore ad avviare una nuova partita. Dopo il *game over* è visibile una scritta *GAME OVER* che copre la griglia.

Come già accennato, il componente grafico consiste nel modulo `vga_sync` per gestire la corretta temporizzazione dei segnali verso la porta VGA, mentre `graphic_gen` effettua il *rendering* vero e proprio del gioco.

4.1 Segnali VGA e loro temporizzazione

La porta VGA montata sulla scheda è un connettore femmina DB15. Solo 5 pin sono però controllabili direttamente dalla FPGA e forniscono l'output e la sincronizzazione video, mentre gli altri sono scollegati o collegati a massa quando necessario. Di seguito elenchiamo i pin di interesse, tutti da fornire in output verso il monitor.



- pin 1: red video;
- pin 2: green video;
- pin 3: blue video;
- pin 13: horizontal sync (hsync);
- pin 14: vertical sync (vsync).

Secondo lo standard VGA, i pin per i colori rosso, verde e blu trasportano segnali analogici, cioè possono essere usati per mostrare diverse tonalità di colore a seconda della tensione introdotta sul filo. Sulla Spartan-3 però i tre pin sono connessi, tramite opportune resistenze, direttamente a segnali digitali senza uso di DAC, per cui per ogni pin abbiamo a disposizione solo due tensioni, corrispondenti ai valori digitali 0 e 1. Questo ci consente di ottenere una profondità di colore di soli 3 bit (8 colori), esposta di seguito.

Blu	**Verde**	**Rosso**	**Colore**
0	0	0	Nero
0	0	1	Rosso
0	1	0	Verde
0	1	1	Giallo
1	0	0	Blu
1	0	1	Magenta
1	1	0	Ciano
1	1	1	Bianco

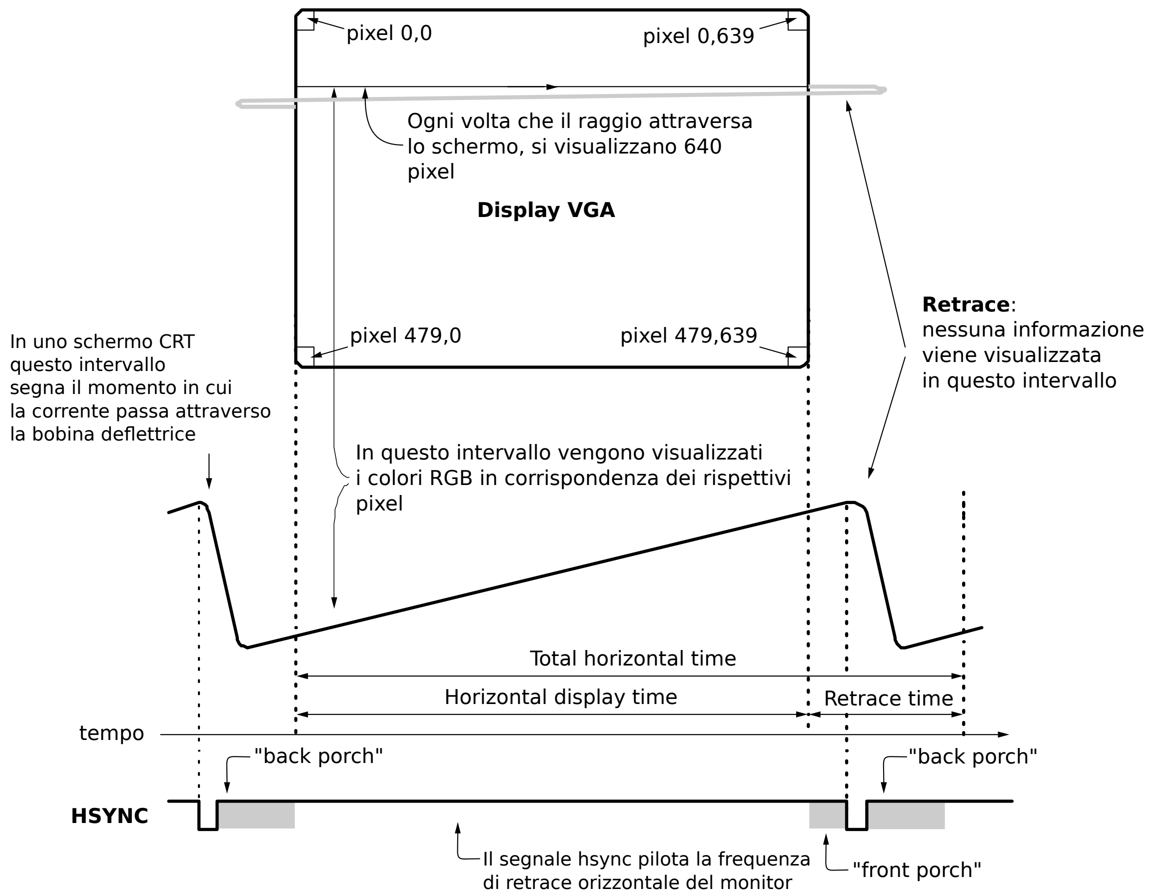


Figura 6: Schema della sincronizzazione orizzontale VGA

La temporizzazione dei segnali è gestita dal modulo `vga_sync`. Questo modulo è una versione leggermente modificata dell'omonimo presente a pagina 264 di [3].

Senza entrare nei dettagli dell'implementazione del modulo e nelle precise specifiche di temporizzazione dei segnali, per cui rimandiamo a [3] e [4], sintetizziamo di seguito il comportamento richiesto dallo standard video in termini di sincronizzazione.

La Figura 6 mostra uno schema del comportamento del segnale `hsync` durante il disegno dei pixel su uno schermo VGA con risoluzione di 640×480 pixel. Si nota che c'è un tempo fissato chiamato *retrace*, composto dagli intervalli di *front porch*, *back porch* (evidenziati in grigio in figura) e dalla durata dell'impulso di `hsync`, in cui non viene visualizzata alcuna informazione di colore sullo schermo. In particolare, i pin VGA R, G e B devono rimanere a 0 (tensione bassa) per tutto il periodo. Questo rappresenta il tempo richiesto dal raggio catodico dei monitor CRT per posizionarsi dal termine di una riga di pixel all'inizio della riga successiva dello schermo e si verifica quindi dopo che si disegna il pixel 639 di ogni riga. Durante l'intervallo marcato in figura come *horizontal display time*, invece, la tensione sui pin R, G e B viene convertita in colore dal display. Tutta l'unità video va sincronizzata con un clock a 25 MHz, tramite il quale si esprime, come si può verificare in [4], la durata dei tempi di attesa e impulsi appena descritti e la scansione del tempo di visualizzazione di ogni pixel.

Al termine del disegno di una schermata (d'ora in poi *frame*) è necessario attendere un altro intervallo di tempo, sensibilmente più lungo del tempo di retrace, che consente al raggio catodico di uno schermo CRT di posizionarsi dall'angolo inferiore destro dello schermo all'angolo superiore sinistro per poter disegnare il frame successivo. Questo periodo di tempo è spesso chiamato *vertical blanking interval* e la sua gestione è

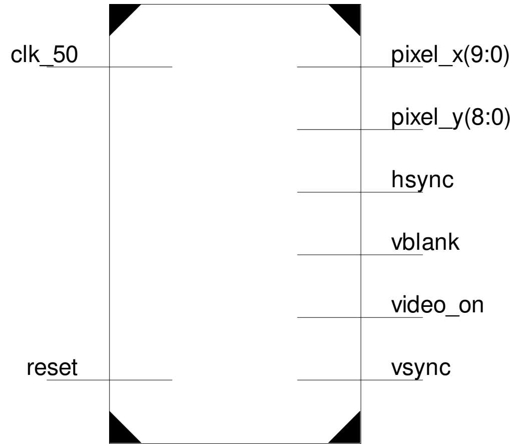


Figura 7: vga_sync

completamente analoga a quella del tempo di retrace appena illustrata. La principale differenza sta nella durata dei tempi di front porch, back porch e impulso sul pin vsync. Si nota inoltre che, per quanto la descrizione dei tempi si riferisca ai monitor CRT, lo standard video VGA impone le stesse specifiche quale che sia la tecnologia del monitor. Anche i monitor VGA LCD, pur necessitando di tempi tecnici di redraw e vertical blanking interval molto più brevi, rispettano esattamente le stesse tempistiche dei CRT, consentendo dunque a tutti i dispositivi già compatibili con i CRT VGA di essere compatibili anche con gli LCD VGA e viceversa.

Il modulo vga_sync implementa esattamente questa specifica e fornisce in output, oltre ai segnali sui pin vsync e hsync, due bus pixel_x e pixel_y più segnali vblank e video_on, illustrati di seguito:

- vblank presenta un impulso all'inizio del *vertical blanking interval*;
- I segnali corrispondenti ai pixel codificano le coordinate orizzontale e verticale del pixel da visualizzare a schermo. Si nota che, per come è implementato vga_sync, questi ultimi due segnali non cambiano nel preciso istante in cui lo schermo disegna il pixel corrispondente ma cambiano con un ciclo di clock da 50 MHz di anticipo. Questo dà il tempo necessario al modulo che genera i colori (graphic_gen) di fornire output a fronte del cambiamento del pixel che verrà catturato dal registro rgb_reg, sincronizzato con il clock da 50 MHz, e dunque mantenuto stabile sulla porta VGA per tutta la durata del tempo di disegno del pixel;
- video_on è alto se il monitor sta effettivamente visualizzando pixel, basso altrimenti.

4.2 Generazione della grafica

Una volta introdotto il modulo che gestisce la sincronizzazione dei segnali video, abbiamo predisposto una serie di moduli che si occupano della generazione dei segnali di colore veri e propri. Questi fanno capo al modulo graphic_gen e di seguito ne illustriamo l'organizzazione.

La funzione di graphic_gen, mostrato in Figura 8, è quella di comporre l'immagine completa a schermo. Il modulo riceve in ingresso i segnali relativi alle coordinate dei pixel e video_on in uscita da vga_sync, lo stato console, tipo enumerativo che può assumere i valori game, splash e game_over, necessario al modulo per stabilire se visualizzare la scritta *PRESS START*, *GAME OVER* o entrambe più i segnali necessari a leggere dalla RAM (utilizzando la *port* A) e una serie di segnali di stato del gioco relativi a punteggio, linee, livello eccetera.

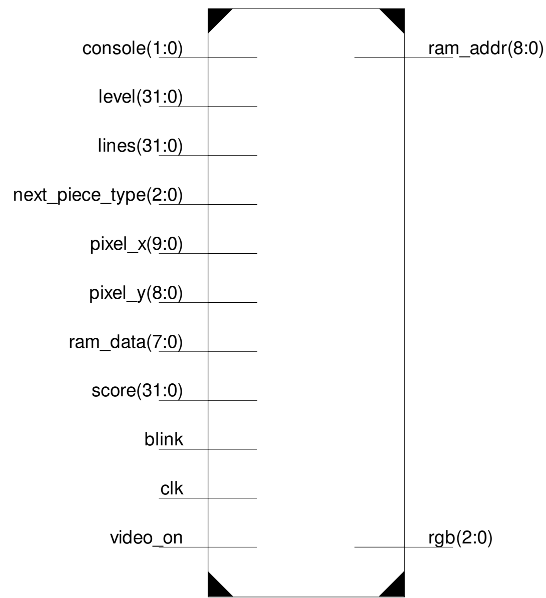


Figura 8: `graphic_gen`

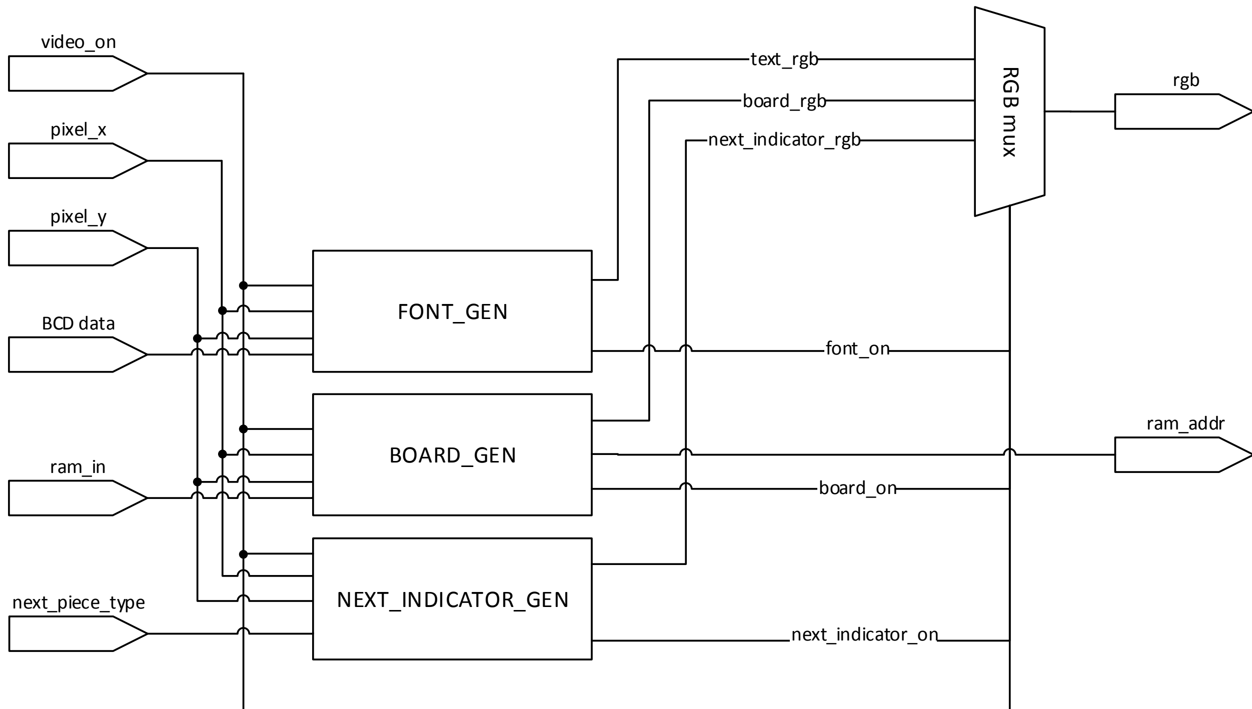


Figura 9: Schema di funzionamento del modulo `graphic_gen`

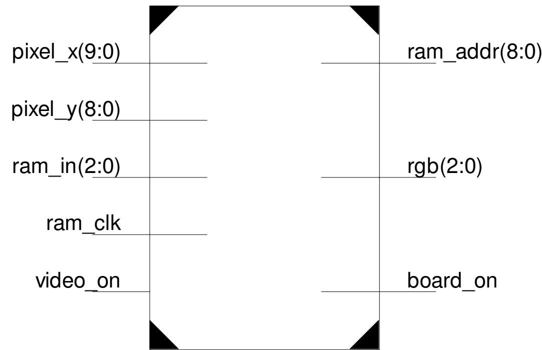


Figura 10: `board_gen`

Al fine di evitare complicazione eccessiva di `graphic_gen` questo modulo, il cui funzionamento è schematizzato in Figura 9, non disegna direttamente alcun elemento sullo schermo ma delega il compito ai seguenti:

- `board_gen` che si occupa di disegnare la griglia di gioco;
- `font_gen` che disegna tutte le scritte a schermo;
- `next_indicator` che disegna l'indicatore del pezzo successivo.

Ognuno di essi riceve in input le coordinate del pixel disegnato sullo schermo, il segnale di `video_on` e deve fornire in output un proprio segnale di attivazione (`on`) che indica se quella particolare sezione della grafica è attiva e il segnale `rgb` che indica il colore del pixel. Il compito di `graphic_gen` è quindi quello di selezionare il colore del pixel da mostrare a schermo a seconda di quale modulo abbia alzato il proprio segnale di `on`. Se più di un modulo dovesse risultare attivo in un determinato momento, il circuito di selezione (marcato come *RGB mux* nell'immagine) manderà in output il colore in uscita dal modulo con maggiore priorità (prima `font_gen`, poi `board_gen` e infine `next_indicator_gen`). Se nessun segnale di attivazione dovesse essere alto, l'uscita sarà semplicemente il colore di sfondo. Il circuito di selezione si occuperà inoltre di mantenere l'uscita a 0 nel caso in cui non ci siano pixel da visualizzare sullo schermo, cioè se `video_on` è a 0, come da standard VGA. Lo scopo degli altri segnali in ingresso ai vari moduli non è rilevante ai fini dell'illustrazione di `graphic_gen`, dunque ne rimandiamo la discussione al momento della descrizione di ciascun modulo.

4.3 Griglia di gioco

Come già illustrato, la griglia di gioco è memorizzata sulla RAM, dunque il compito di questa parte del componente di rendering è quello di leggere lo stato della griglia dalla memoria e mostrarlo a schermo. Per farlo, utilizziamo un'area rettangolare da 10×18 celle in cui ogni cella occupa un quadrato di dimensione 16×16 pixel. Nel caso in cui la cella non sia occupata viene mostrata interamente bianca, altrimenti viene mostrata del colore corrispondente memorizzato in RAM con un piccolo bordo interno di colore nero della dimensione di 1 pixel. Questo è lo scopo del modulo `board_gen`, rappresentato in Figura 10. Il modulo prende in ingresso le coordinate x e y del pixel, fornisce in output il colore rgb da inviare allo schermo e il proprio segnale di attivazione `board_on`. È inoltre fornito dei segnali necessari per la lettura dalla RAM.

Il funzionamento interno del modulo è schematizzato in Figura 11.

Il blocco rappresentato nello schema come *translate coordinate* rappresenta la logica combinatoria necessaria a traslare le coordinate `pixel_x` e `pixel_y` nel punto di origine della griglia (l'angolo in alto a sinistra) convertendole quindi in `local_x` e `local_y`. Eccezion fatta per il piccolo bordo che circonda la griglia, ogni cella assumerà il colore stabilito dal dato in ingresso in RAM. Come si deduce dal diagramma, il funzionamento si divide in due parti, la parte (inferiore) di selezione dell'indirizzo da leggere dalla RAM e la parte (superiore) di generazione del colore vero e proprio.

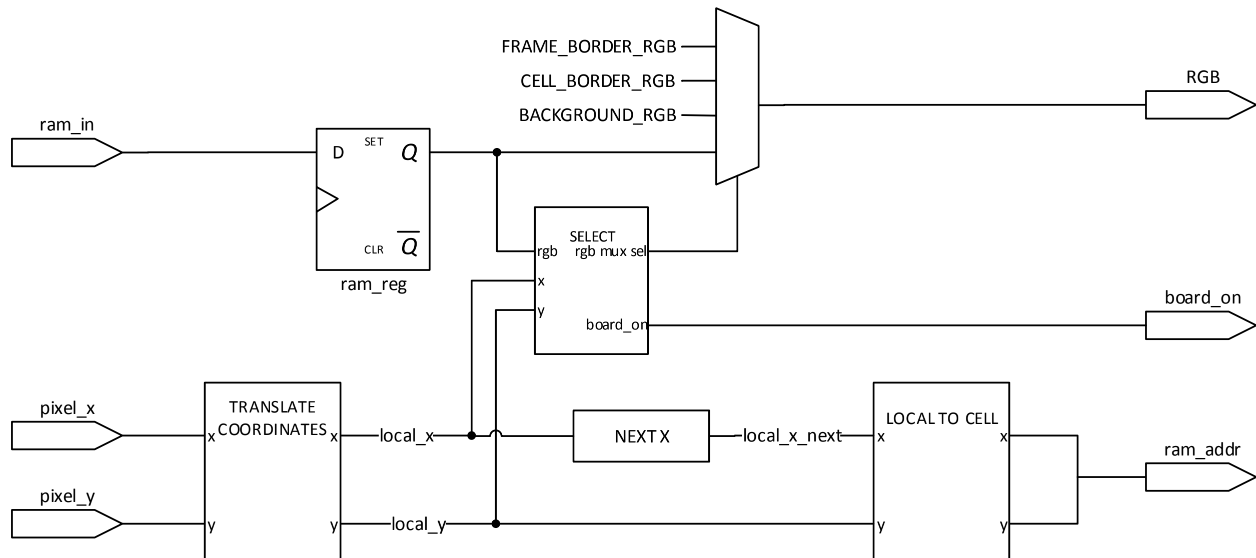


Figura 11: Schema del funzionamento di `board_gen`

Ricordiamo che la lettura da RAM impiega un ciclo di clock, e, per via del meccanismo di funzionamento di `vga_sync` e del registro `rgb_reg`, al momento in cui il segnale `pixel_x` cambia, la logica combinatoria di generazione del colore deve far sì che il dato sia già disponibile in uscita prima del successivo fronte di salita del clock. Il bus dati della RAM cambia invece dopo il successivo fronte di salita del clock, troppo tardi perché il registro `rgb_reg` lo salvi. Per ovviare a questo problema il circuito che seleziona l'indirizzo della RAM deve calcolare l'indirizzo corrispondente alla cella del pixel successivo a quello asserito in `local_x`. Per fare questo un sommatore, rappresentato da *next x* nel grafico, calcola `local_x_next` che viene poi tradotto in indirizzo di cella tramite un'opportuna traslazione (per compensare il bordo esterno della griglia) e divisione per 16 (implementata come uno shift). Per quanto riguarda invece la coordinata y, dopo aver traslato e diviso in maniera analoga si traduce la coordinata in modo che la cella 0 sia l'ultima in basso e il valore della cella progredisca verso l'alto. Queste operazioni vengono svolte nel blocco indicato come *local to cell*. Il risultato viene poi opportunamente concatenato e portato sul bus indirizzo della RAM.

Il circuito di generazione del colore, invece, stabilisce sulla base delle coordinate `local_x` e `local_y` e del dato letto dalla RAM il colore effettivo da visualizzare sullo schermo. Il compito del blocco marcato come *select* e del relativo mux è quello di identificare in quale area della griglia ci si trovi e di generare il relativo colore. In dettaglio:

- se ci si trova nel bordo esterno, il colore in output sarà `FRAME_BORDER_RGB`;
- se la cella corrente è vuota (il dato letto è 0), il colore sarà quello di sfondo della griglia `BACKGROUND_RGB`;
- se la cella corrente è colorata e ci si trova nel pixel di bordino interno, il colore sarà `CELL_BORDER_RGB`;
- se ci si trova all'interno di una cella colorata (escluso il bordo) il colore sarà quello del dato proveniente dalla RAM;
- altrimenti, se ci si trova fuori dalla griglia, il segnale `board_on` sarà abbassato e il colore in uscita sarà irrilevante, mentre `board_on` viene mantenuto alto negli altri casi.

Rimane un solo problema di temporizzazione. La RAM è collegata al clock da 50 MHz, mentre i pixel vengono disegnati sullo schermo alla frequenza di 25 MHz. Questo significa che, introducendo il segnale relativo al pixel successivo sul bus dati ogni volta che il segnale `pixel_x` cambia, il dato relativo sarà già pronto dopo un ciclo di clock, mentre dovrà essere effettivamente visualizzato dopo due cicli. Per compensare questo anticipo è sufficiente introdurre un ritardo di un ciclo di clock che si ottiene aggiungendo un registro

(`ram_reg`) sincronizzato con il clock da 50 MHz collegato in cascata al bus dati della memoria RAM, prima del circuito di selezione del colore.

4.4 Font

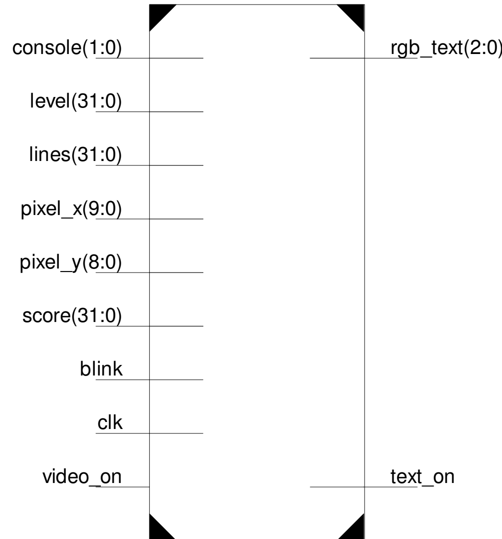


Figura 12: `font_gen`

Il testo visualizzato durante il gioco viene implementato sfruttando uno schema *tile-mapped*, raggruppando blocchi di 8×16 bit in diverse *tile*, ciascuna delle quali verrà trattata come *display unit*.

In particolare, ogni carattere del codice ASCII a 7-bit viene associato ad una tile, composta da un pattern di 0 e 1. Il pattern delle tile costituisce il *font* del set di caratteri. Nell'ampia scelta di font disponibili, per il progetto viene utilizzato il font *Atari* visibile in Figura 13. In questo font, ciascun carattere viene rappresentato da un pattern di 8×16 pixel. In Figura 14a viene mostrato il pattern per la lettera “A”.

Tutti i pattern vengono memorizzati in una *font ROM* e ogni pattern necessita di $2^4 \times 8$ bit. La font ROM viene organizzata in 2^{11} parole da 8 bit per contenere i pattern di tutti i 128 caratteri ASCII. Utilizzando questo tipo di tile, la risoluzione dello schermo da 640×480 pixel potrebbe essere riorganizzata in 80×25 tiles.

Figura 13: Atari font

Nella font ROM, i 7 bit più significativi dell'indirizzo identificano il carattere (`char_addr`) mentre i 4 bit meno significativi identificano la riga all'interno del pattern del carattere (`row_addr`), come in Figura 14b.

Il modulo `font_gen`, mostrato in Figura 12, si occupa di visualizzare sullo schermo tutte le etichette testuali e le cifre che compongono il punteggio, il contatore delle linee cancellate e l'indicatore del livello. Ogni etichetta viene memorizzata in una *tile memory* contenente il valore ASCII dei caratteri che la compongono.

Ogni etichetta avrà associati due segnali `local_x` e `local_y` ottenuti traslando nel punto di origine dell'etichetta stessa i segnali `pixel_x` e `pixel_y` in ingresso al modulo. Questa operazione viene effettuata per semplificare l'estrazione dei caratteri dalla *tile memory*.

La generazione dei pixel dei caratteri all'interno di un'etichetta avviene in due passaggi. Inizialmente vengono estratti i bit più significativi dei segnali `local_x` e `local_y` per ottenere l'indirizzo della *tile memory* da cui prelevare la tile desiderata (`char_addr`) che corrisponde al codice ASCII del carattere. I bit del segnale `local_y` sono necessari per visualizzare etichette multilinea. Nel passaggio successivo, tale codice costituisce

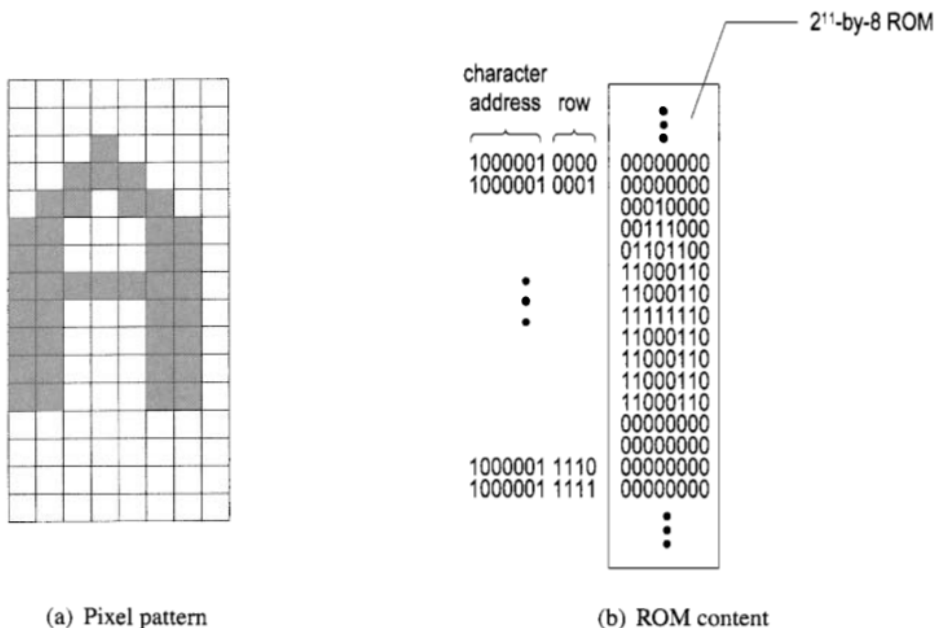


Figura 14: Font pattern della lettera A

i 7 MSB dell'indirizzo della font ROM e specificano la locazione in cui è memorizzato il pattern relativo al carattere. L'indirizzo completo viene ottenuto concatenando i bit meno significativi di `local_y` (`row_addr`) ottenendo in output una riga da 8 bit del pattern (`font_word`). Utilizzando un multiplexer pilotato dai bit meno significativi di `local_x` (`bit_addr`) vengono estratti i singoli bit che compongono ogni riga della tile, generando così il valore del pixel in output. La Figura 15 illustra come vengono estratti i bit per ottenere i segnali utili alla font ROM.

Tutte le etichette testuali vengono generate utilizzando il metodo sopra descritto. Per quanto riguarda l'indicatore dei punti, delle linee e del livello, vengono generati in maniera analoga con l'unica differenza che i caratteri da stampare non sono statici, ma sono dipendenti dal valore assunto da particolari segnali, che sono rispettivamente `score`, `lines` e `level`. Questi segnali sono i valori del punteggio, del numero di linee cancellate e del livello raggiunto, opportunamente convertiti in decimale per essere visualizzati. Rimandiamo alla Sezione 6.6 per la descrizione dettagliata del meccanismo di conteggio e di conversione.

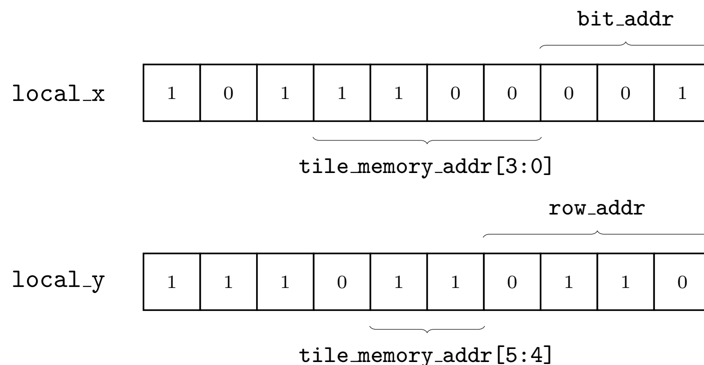


Figura 15: Estrazione dei segnali utili alla Font ROM

La scritta *GAME OVER* viene ottenuta utilizzando la tecnica di *font scaling*. Il font originario 8×16 bit viene raddoppiato a 16×32 bit espandendo il pixel originario a quattro pixel. Tale operazione è molto semplice da effettuare poiché è sufficiente applicare ai segnali `pixel_x` e `pixel_y` uno shift verso destra di un bit.

L'effetto lampeggiante della scritta *PRESS START* viene pilotato dal segnale periodico `blink`.

Il modulo `font_gen` prende in ingresso anche un segnale `console` che rappresenta lo stato globale del gioco. La presenza di questo segnale è necessaria poiché alcune scritte devono comparire solo in determinate occasioni. Si tratta di *PRESS START* che compare sia nella schermata iniziale di gioco (*splash screen*) sia a gioco terminato, e della scritta *GAME OVER* che appare quando il gioco termina.

4.5 Indicatore Next Piece

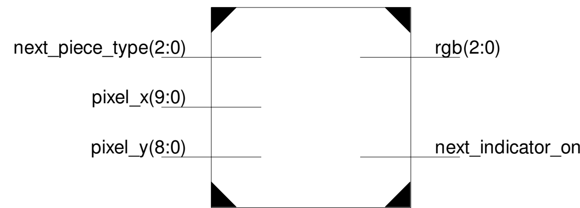


Figura 16: `next_indicator_gen`

L'ultimo elemento grafico rimasto da descrivere è l'indicatore del pezzo successivo, gestito dal modulo `next_indicator_gen`, mostrato in Figura 16. L'elemento appare a schermo come un quadrato di dimensione 4×4 celle ciascuna da 16×16 pixel. Il funzionamento interno del modulo risulta molto semplice: a partire dalle coordinate in ingresso, dopo aver effettuato un'opportuna traslazione per riportarle nell'origine dell'indicatore, si opera una divisione per 16 sia sulla componente x sia sulla componente y. Questo fornisce segnali di coordinata della cella che vanno da 0 a 3 sia in verticale che in orizzontale. Un circuito di selezione, collegato all'input `next_piece_type` e alle coordinate calcola il contenuto della cella e il colore da mostrare in output. In particolare, le celle del quadrato sono riempite esattamente come illustrato in Figura 19 a pagina 23, mostrando ogni tetramino a rotazione 0 e introducendo un piccolo bordo nero di un pixel quando necessario esattamente come descritto per `vga_sync`.

5 Gestione dell'input

Per quanto riguarda l'acquisizione dell'input dal gioco abbiamo potuto scegliere tra varie possibilità: si possono infatti utilizzare i pushbutton presenti direttamente sull'FPGA, una tastiera PS/2 collegata all'apposito connettore sulla scheda o un gamepad o un joystick di una vecchia console da gioco.

Abbiamo deciso di utilizzare un gamepad per il semplice motivo che rende il controllo più naturale per un'applicazione di questo genere e dà al progetto un indubbio *feeling* da console da gioco retrò. Tra i vari gamepad a nostra disposizione abbiamo scelto di collegare un controller per *SEGA Mega Drive* per via della sua eccezionale semplicità d'uso dal punto di vista del collegamento elettrico e per via del fatto che utilizza un'interfaccia standard comune a gamepad per numerose console. Questo consente, qualora fosse necessario, di sostituire il controller utilizzato correntemente con uno compatibile senza bisogno di modificare il progetto.

Il gamepad è dotato delle quattro frecce direzionali, tre bottoni *A*, *B* e *C* e un bottone *START*.

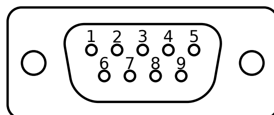


Figura 17: Connettore DB-9 del gamepad.

Il gamepad è dotato di un connettore femmina DB-9, mostrato nello schema in Figura 17 che abbiamo collegato alla scheda attraverso un cavo con connettore maschio costruito per l'occasione. Vista la diffusione di questo tipo di gamepad, in passato è stato già svolto abbondante lavoro di *reverse engineering* e le specifiche tecniche sono facilmente reperibili online (si consulti [5]). Per l'uso del controller è necessario collegare in modo appropriato i seguenti tre pin:

- pin 5: power
- pin 7: select
- pin 8: ground

Il *Mega Drive* utilizzava una tensione di +5V sul pin power, mentre la nostra FPGA fornisce una tensione di +3.3V. Questo non crea problemi d'uso visto che il chip 74HC157 utilizzato all'interno del controller può operare a diverse tensioni, tra cui quella che utilizziamo noi.

Il pin di select invece rappresenta un segnale binario da inviare verso il controller. A seconda del valore di questo segnale il controller invierà i dati relativi alla pressione di bottoni diversi sugli altri pin di uscita. In particolare, la Tabella 1 elenca la funzione dei rimanenti pin del connettore rispetto al segnale di select. Tutti i segnali sono da intendersi *active low*.

pin	**select basso**	**select alto**
1	gamepad up	gamepad up
2	gamepad down	gamepad down
3	0 (ground)	gamepad left
4	0 (ground)	gamepad right
6	button A	button B
9	button START	button C

Tabella 1: Pinout del controller *SEGA Mega Drive*

All'interno del progetto i pin del gamepad sono connessi direttamente al modulo `sega_controller_driver`, raffigurato in Figura 18. Il modulo prende in ingresso i pin 1, 2, 3, 4, 6 e 9 del connettore, lo stato della console di gioco, che ricordiamo poter essere uno tra `game_over`, `game` e `splash`, e fornisce in uscita il segnale di select, collegato anch'esso al connettore. Fornisce in uscita verso la logica di gioco un segnale `command` che

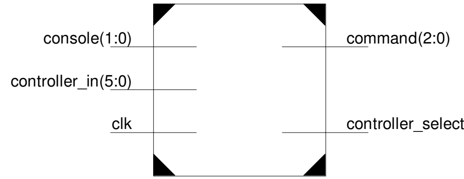


Figura 18: `sega_controller_driver`

rappresenta il comando impartito dal giocatore, uno tra `down`, `left`, `right`, `rotate`, `start` e `no_command`. Si nota che questo modulo nasconde al suo interno il valore del tasto effettivamente premuto. Per esempio, in questo caso abbiamo deciso che il modulo può emettere `rotate` se si preme un bottone tra *up*, *B* e *C*. Questo genere di configurazione consente di, qualora si decidesse di cambiare tipo di periferica di input, sostituire esclusivamente il modulo di gestione del gamepad mantenendo i moduli di gestione della logica completamente inalterati. Se, al contrario, avessimo deciso di mettere in output un segnale che rappresenta il bottone effettivamente premuto, dovremmo modificare di conseguenza anche la logica di gioco a seconda del dispositivo (un altro controller potrebbe non avere i bottoni *B* e *C* o il tasto *START*, per esempio).

Visti i tasti che abbiamo deciso di usare e la funzione del segnale di select esposto in Tabella 1, si nota che tutti i bottoni necessari sono leggibili sui pin con select alto, eccezion fatta per *START*. Considerato però che il tasto *START* è utile soltanto durante lo *splash screen* o quando il gioco è in *game over* e rappresenta l'unico comando che il giocatore impartisce in questo stato, mentre tutti gli altri vanno usati durante il gioco, ci è sufficiente porre select basso quando `console` è in `splash` o `game_over`, alto altrimenti. La gestione del segnale `command` si riduce quindi a un semplice circuito combinatorio di selezione sulla base di `console` e dei pin del controller. Per ridurre il rumore sulle linee in uscita, inoltre, abbiamo trovato opportuno introdurre un circuito di *debouncing* collegato a ogni pin di ingresso.

Per completare l'integrazione della gestione dell'input nel progetto occorre gestire un altro problema: il circuito di gestione del controller non emette un impulso alla pressione del tasto, ma mantiene il segnale in uscita stabile finchè il giocatore preme il pulsante, cambiandolo solo al momento in cui lo rilascia. Occorre quindi che il circuito di gestione dell'input possa distinguere il caso in cui la pressione di un bottone rappresenti il tentativo da parte dell'utente di effettuare una singola mossa (per esempio una singola rotazione del pezzo) da quello in cui la pressione prolungata abbia il significato di impartire più comandi in rapida sequenza, come è tipico quando si vuole far cadere il tetramino verso il basso.

Per risolvere questo problema abbiamo optato per un sistema semplice che si è rivelato efficace. L'effettiva implementazione si trova in `tetris_logic.vhd`, ma viene descritto ugualmente in questa sezione.

I comandi del gamepad vengono controllati una volta per ogni frame, quando la macchina a stati di `tetris_logic` è nello stato `check_input`. Contemporaneamente agisce un `process` che lavora con un registro counter `lock_count_reg` impostato inizialmente a 0 e regola il segnale di input in ingresso alla logica di transizione della macchina a stati. Se il segnale `game_tick` è alto o se `lock_count_reg` è diverso da 0 l'input viene semplicemente scartato. La macchina leggerà quindi sempre `no_command` in questi casi, indipendentemente dai bottoni premuti sul controller. Altrimenti, se il modulo di gestione dell'input presenta un segnale diverso da `no_command`, `lock_count_reg` viene impostato a un valore fissato e il comando viene lasciato passare verso la logica di scatto della macchina a stati. A ogni frame, inoltre, se il registro è diverso da 0 viene decrementato. Questo sistema fa sì che effettivamente l'input del giocatore venga bloccato per un certo numero di frame. In particolare, viene bloccato per 3 frame nel caso in cui il comando sia `down`, 12 frame nel caso in cui il comando sia `rotate`, altrimenti un numero di frame variabile a seconda della velocità di caduta del tetramino.

6 Logica di gioco

In questa sezione verrà descritta l'implementazione della logica di gioco del Tetris. Per quanto riguarda l'implementazione dei tetramini, abbiamo deciso di rappresentarli in un record `tetris_piece` che contiene:

- le coordinate (`coords`) dei quattro quadretti che compongono il tetramino;
- il valore `rgb` del colore del tetramino;
- il tipo (`piece_type`);
- l'orientamento del tetramino (`rotation`).

Ad ogni tipo di tetramino viene associato un colore differente come già illustrato in Figura 2, fatta eccezione per i pezzi di tipo I e O che condividono lo stesso colore. Questa scelta è dovuta al fatto che i colori disponibili sono solo 8, di cui due (il nero e il bianco) non possono essere utilizzati poiché denotano rispettivamente il bordino interno di ogni quadretto e lo sfondo della griglia di gioco. In caso contrario si sarebbe ottenuta una minor chiarezza durante il gioco.

Per quanto riguarda la rotazione dei tetramini, ci siamo ispirati alla logica *"Left-Handed version"* utilizzata da Nintendo nella versione del Tetris per Game Boy, nella quale i pezzi vengono ruotati intorno ad un quadretto specifico che funge da perno. Ogni tetramino può assumere al più quattro diverse configurazioni, che abbiamo numerato da 0 a 3. La rotazione dei pezzi viene implementata traslando opportunamente i quadretti che compongono il tetramino mediante l'uso di sommatori e sottrattori. La Figura 19 illustra la posizione dei singoli quadretti nelle diverse fasi di rotazione del tetramino. Il quadretto etichettato con 0 rappresenta il perno del tetramino.

La logica del gioco è implementata dal modulo `tetris_logic` il quale a sua volta istanzia diversi moduli elencati di seguito:

- `random_gen` per rendere casuale la scelta del pezzo successivo da far cadere;
- `piece_mover` che gestisce la caduta del pezzo corrente, esegue la mossa effettuata dal giocatore e verifica l'occorrenza di collisioni;
- `line_clearer` il cui compito è quello di controllare l'esistenza di linee complete, di cancellarle e spostare verso il basso eventuali linee soprastanti;
- `board_reset` che pulisce la griglia di gioco ogni volta che viene iniziata una nuova partita;
- `score_counter` per gestire il punteggio totalizzato durante la partita.

6.1 Tetris Logic

Vediamo ora nel dettaglio il modulo `tetris_logic`, in Figura 20 che, come già descritto in precedenza, implementa la logica del gioco vero e proprio.

Questo modulo prende in ingresso, oltre ai segnali di clock e di lettura dalla RAM, anche un segnale di `input` che trasporta il comando dato dal giocatore al controller, e un segnale `vblank`, fornito dal modulo `vga_sync`, per pilotare tutti gli aggiornamenti innescati dalla FSM del modulo. In particolare, tali aggiornamenti vengono effettuati durante il tempo di inattività che intercorre tra il disegno di due frame consecutivi. In uscita vengono forniti i segnali contenenti i valori del punteggio, linee e livello (`score, lines, level_bcd`) che vengono collegati al modulo `graphic_gen` che si occuperà di stamparli, lo stato globale del gioco `console`, la forma del tetramino successivo (`next_piece_type`), i segnali di gestione della RAM e un segnale di `blink` anch'esso collegato a `graphic_gen` che consente alla scritta *PRESS START* di lampeggiare.

Il tetramino scende di una riga verso il basso con una velocità dipendente dal livello raggiunto. La velocità di caduta viene scandita dal segnale `game_tick` che genera un impulso dopo un determinato numero di frame che varia in base al livello, come mostrato in Tabella 2.

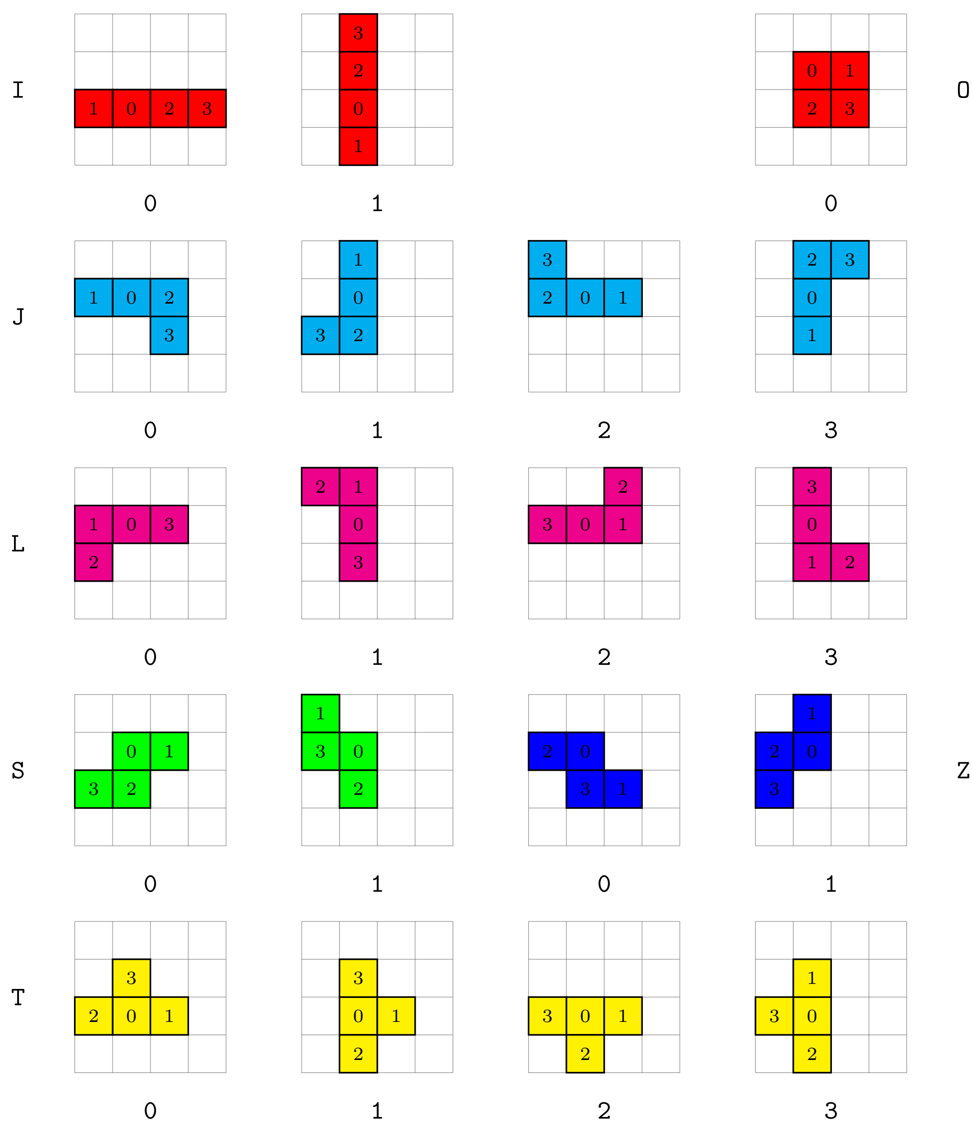


Figura 19: Rotazione dei tetramini

Livello	Numero di frame
0	53
1	49
2	45
3	41
4	37
5	33
6	28
7	22
8	17
9	11
10	10
11	9
12	8
13	7
14	6
15	5

Tabella 2: Velocità di caduta del pezzo in base al livello

La macchina a stati finiti realizzata nel modulo `tetris_logic` viene descritta dal grafo delle transizioni in Figura 21.

Gli stati che compongono la FSM vengono di seguito descritti più nel dettaglio.

splash (stato iniziale)

Quando la FPGA viene accesa, il gioco si trova in questo stato il cui compito è quello di visualizzare lo splash screen in Figura 1 letto dalla RAM, finché il giocatore non preme il tasto `start` del controller. Quando ciò avviene la macchina si sposta nello stato `reset_game`.

reset_game

Il compito di questo stato è quello di alzare il segnale `score_reset` che, letto dal modulo `score_counter`, ha l'effetto di reimpostare i contatori di punteggio, linee cancellate e livello raggiunto. Terminato il reset, la FSM si muove nello stato `reset_board`.

reset_board

Per iniziare una nuova partita è necessario preparare la griglia di gioco, cancellandone il contenuto precedente. Tale operazione viene svolta dal modulo `board_reset` che si attiva quando la FSM entra in questo stato. La FSM rimane in attesa che `board_reset` termini il suo compito, dopo di che si porta nello stato `first_piece`.

first_piece

In questo stato viene impostato il tipo del primo tetramino della partita che viene scelto in modo casuale. Quando la scelta è stata effettuata la FSM si pone in `new_piece`.

new_piece

Questo è lo stato che si occupa di generare un nuovo tetramino del tipo specificato nel registro `new_type_reg`. La FSM entra in questo stato sia per generare il primo tetramino della partita, sia per generare il pezzo successivo dopo che il tetramino corrente è stato posizionato. Generato il nuovo tetramino, `tetris_logic` genera automaticamente la mossa `place_new_piece` e la va ad effettuare posizionandosi in `attempt_move`.

attempt_move

In questo stato viene attivato il modulo `piece_mover` che si occupa di realizzare la mossa specificata dal

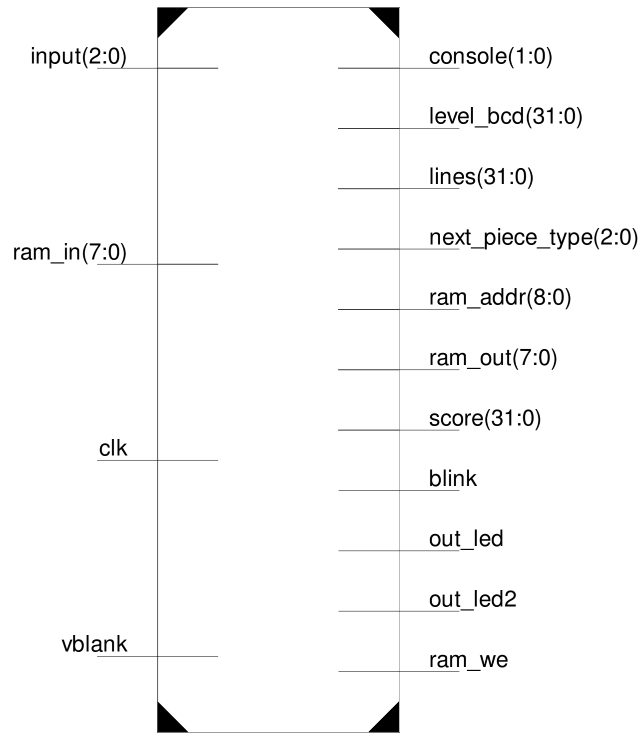


Figura 20: `tetris_logic`

segnale `move_reg`. Quando la mossa è stata effettuata la FSM si risveglia e controlla leggendo i segnali `collision_found` e `stopped` rispettivamente se si è verificata una collisione oppure se il tetramino è stato posizionato. In particolare se non risulta possibile generare un nuovo teramino per via di una collisione, allora la partita termina e la FSM si porta nello stato `game_over`. Se invece un tetramino è stato appena posizionato la prossima operazione da effettuare sarà l'eliminazione di eventuali linee piene e la FSM lo fa portandosi nello stato `clear_full_lines`. In caso contrario non è necessaria alcuna operazione, e la FSM torna quindi in `idle`. Per una descrizione più dettagliata delle possibili mosse del gioco rimandiamo alla Sezione 6.2.

clear_full_lines
L'entrata in questo stato ha l'effetto di attivare il modulo `line_clearer`, descritto nella Sezione 6.3, il cui compito è quello di controllare se il tetramino appena posizionato ha causato il riempimento di una o più linee. In tal caso il modulo provvede a cancellare le linee piene e a operare lo scorrimento delle linee non vuote soprastanti ad esse. La FSM rimane in attesa che `line_clearer` termini il suo compito cioè quando il segnale `line_clearer_done` viene alzato. Quando ciò avviene la FSM si sposta nello stato `update_score`. In caso contrario la macchina non fa nulla e si porta in `idle`.

update_score
In questo stato si attiva il modulo `score_counter` che provvede ad aggiornare tutti i contatori della partita, in particolare il punteggio totalizzato, il numero di linee cancellate e il livello di gioco raggiunto. La FSM rimane in attesa che il segnale `update_score_done` venga alzato, cioè quando il modulo `score_counter` ha terminato il suo compito, e si porta nello stato `new_piece` pronta per generare un nuovo tetramino.

idle
In questo stato la FSM è in attesa di operazioni da eseguire. La FSM effettua automaticamente una

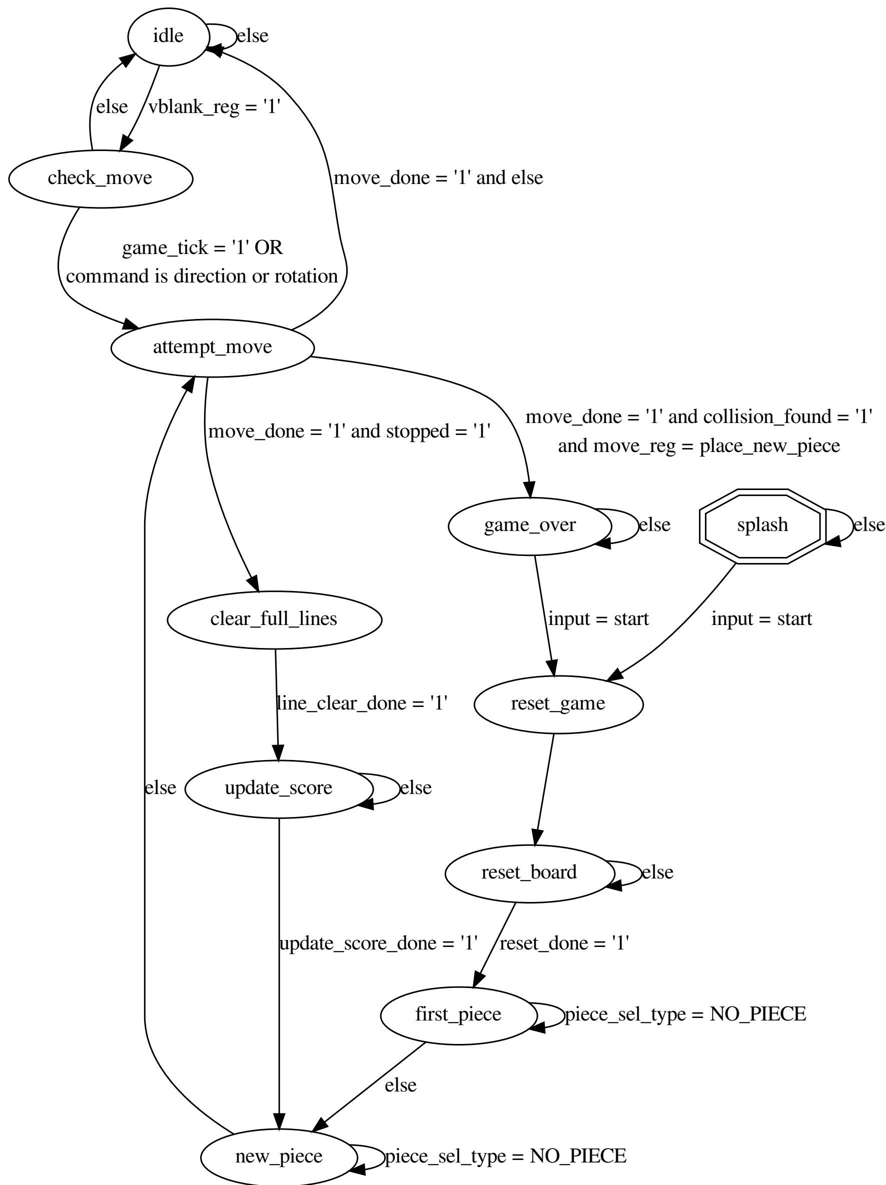


Figura 21: Macchina a stati finiti di `tetris_logic`

transizione verso `check_move` a ogni refresh verticale dello schermo, ossia quando il segnale `vblank` va alto. Tale transizione innescherà la verifica della presenza di una mossa, movimento dei pezzi e in generale tutte le operazioni di aggiornamento dello stato del gioco. La transizione avviene all'inizio del vertical blanking interval per evitare interferenze con il modulo grafico. In particolare vogliamo evitare che il modulo della logica del gioco modifichi lo stato di gioco in RAM o nei registri mentre il modulo grafico legge gli stessi dati per disegnare sullo schermo. Visto il tempo impiegato dalla logica ad aggiornare lo stato rispetto al tempo di attesa dovuto naturalmente all'uso della VGA, abbiamo

notato che tutte le operazioni di aggiornamento hanno abbondante tempo per venire eseguite prima che i dati di gioco siano necessari per la visualizzazione a schermo.

check_move
In questo stato si verifica se c'è una mossa da eseguire. In tal caso il registro `move_reg` viene impostato di conseguenza e la FSM si porta in `attempt_move` per risolvere effettivamente la mossa, altrimenti torna in `idle`. Se il segnale di `game_tick` è alto il gioco effettua automaticamente una mossa verso il basso indipendentemente dall'input del giocatore, implementando così il meccanismo di caduta dei tetramini. In caso contrario si verifica se c'è un comando proveniente dal circuito di gestione dell'input in attesa di essere eseguito.

game_over
La FSM giunge in questo stato al termine della partita e vi rimane finchè il giocatore preme nuovamente il tasto *START* per iniziare una nuova partita e la FSM si porta nello stato `reset_game`.

All'interno di questo modulo è presente un multiplexer il cui scopo è quello di selezionare, in base allo stato corrente della FSM, i segnali corretti da collegare ai segnali RAM in uscita. In particolare, se la FSM si trova nello stato:

- `attempt_move`, l'indirizzo di memoria in cui scrivere sarà dettato dal segnale in uscita dal `piece_mover` (`ram_addr_mover`);
- `clear_full_lines`, sarà il `line_clearer` a decidere le locazioni di memoria in cui scrivere;
- `reset_board`, si occuperà il `board_reset` a gestire la memoria per cancellare la griglia di gioco.

6.2 Movimento del tetramino

Come già accennato in precedenza, la macchina a stati di `tetris_logic` seleziona (talvolta genera automaticamente) e gestisce le mosse che si possono effettuare in Tetris. Queste sono rappresentate nel codice dal tipo enumerativo `move_type`. Ognuna di esse ha un effetto diverso sul tetramino e, a seconda della mossa, può essere necessario passare segnali di controllo diversi al modulo `tetris_logic` affinchè la sua FSM possa cambiare lo stato di gioco correttamente. Di seguito esponiamo le varie mosse in dettaglio:

down Si tratta del movimento del tetramino verso il basso di una singola cella. Può essere generato dall'input del giocatore oppure automaticamente dal gioco come parte del meccanismo di caduta dei tetramini. Nel caso in cui non sia possibile effettuare la mossa in quanto una o più celle sottostanti il tetramino sono occupate oppure il pezzo tocca il bordo inferiore della griglia, il tetramino va fermato. In questo caso il risultato dell'azione sarà `stopped`, che porterà la logica del gioco a effettuare tutte le operazioni di controllo sulle linee ed eventualmente a introdurre un nuovo tetramino in cima alla griglia.

left, right Si tratta del movimento laterale del tetramino, a destra o sinistra di una singola cella. Questa mossa può essere esclusivamente generata da un comando in input dal giocatore. Se la mossa non può essere eseguita perchè le celle coinvolte dal movimento sono occupate oppure si sono toccati i bordi laterali della griglia il risultato sarà `collision_found` che porterà la logica di gioco ad ignorare semplicemente la mossa.

rotate Si tratta del movimento di rotazione del tetramino, già illustrato in Sezione 6. Come per le mosse verso destra e sinistra, l'impossibilità di compiere il movimento genera `collision_found` che porterà il gioco semplicemente a scartare la mossa.

place_new_piece Si tratta del posizionamento di un nuovo pezzo in cima alla griglia. Questa mossa può essere generata esclusivamente dal gioco in automatico all'inizio di una partita oppure al momento in cui un tetramino viene fermato. Se le celle in cui si dovrebbe posizionare il nuovo tetramino sono già occupate il risultato della mossa è `collision_found`. In questo caso la partita non può procedere e la logica del gioco provvede a gestire il *game over*. È l'unico modo per concludere una partita di Tetris.

no_move Si tratta del semplice valore di default per ogni segnale che descrive una mossa.

Il compito di effettuare la mossa richiesta data la posizione corrente del tetramino è affidato al modulo `piece_mover`, mostrato in Figura 22, che gestisce lo spostamento, rotazione, eventuali collisioni del tetramino e l'aggiornamento dello stato della griglia in memoria RAM.

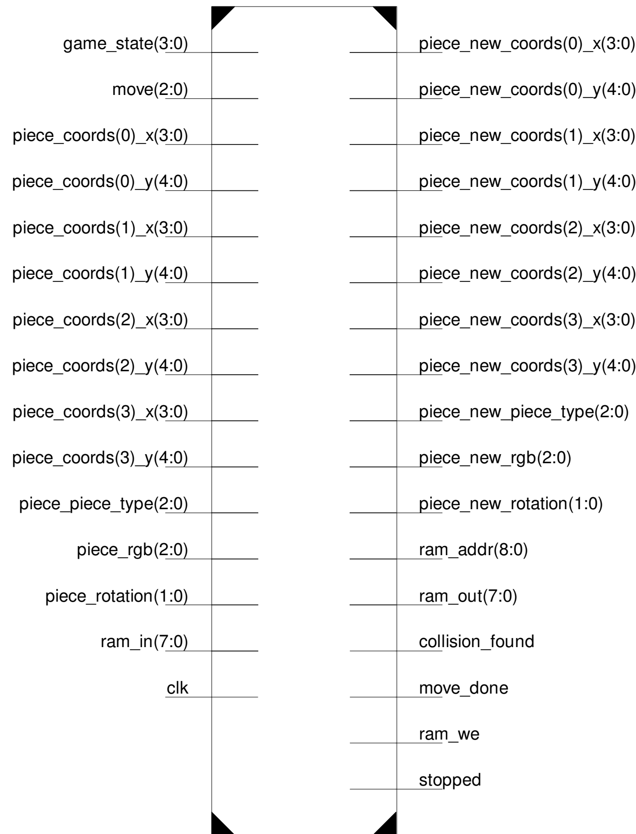


Figura 22: `piece_mover`

Il modulo prende in ingresso lo stato della logica del gioco `game_state`, il record che rappresenta il tetramino corrente `piece_*` e la mossa `move`. Fornisce in uscita il record che rappresenta la posizione del tetramino dopo la mossa (eventualmente uguale alla precedente) `piece_new_*`, i segnali di stato che rappresentano il risultato della mossa `collision_found` e `stopped` e il proprio segnale di terminazione `move_done`. È inoltre fornito dei segnali necessari a leggere e scrivere in RAM.

Per effettuare le sue operazioni, il modulo comprende una macchina a stati finiti, il cui grafo delle transizioni è mostrato in Figura 23.

idle (stato iniziale)
: In questo stato la macchina è semplicemente in attesa che la logica di gioco effettui una transizione verso `compute_next_pos`, che genera una transizione della FSM del modulo stesso verso `compute_next_pos`.

compute_next_pos
: In questo stato si imposta il registro `piece_new_reg` con il risultato della mossa selezionata sul tetramino in ingresso, operando opportune traslazioni sui quadretti che lo compongono, oppure mantenendo il tetramino in ingresso invariato (nel caso di `generate_new_piece` o di `no_move`). Lo spostamento verso destra, sinistra o verso il basso è molto semplice: è sufficiente infatti traslare tutte le celle che

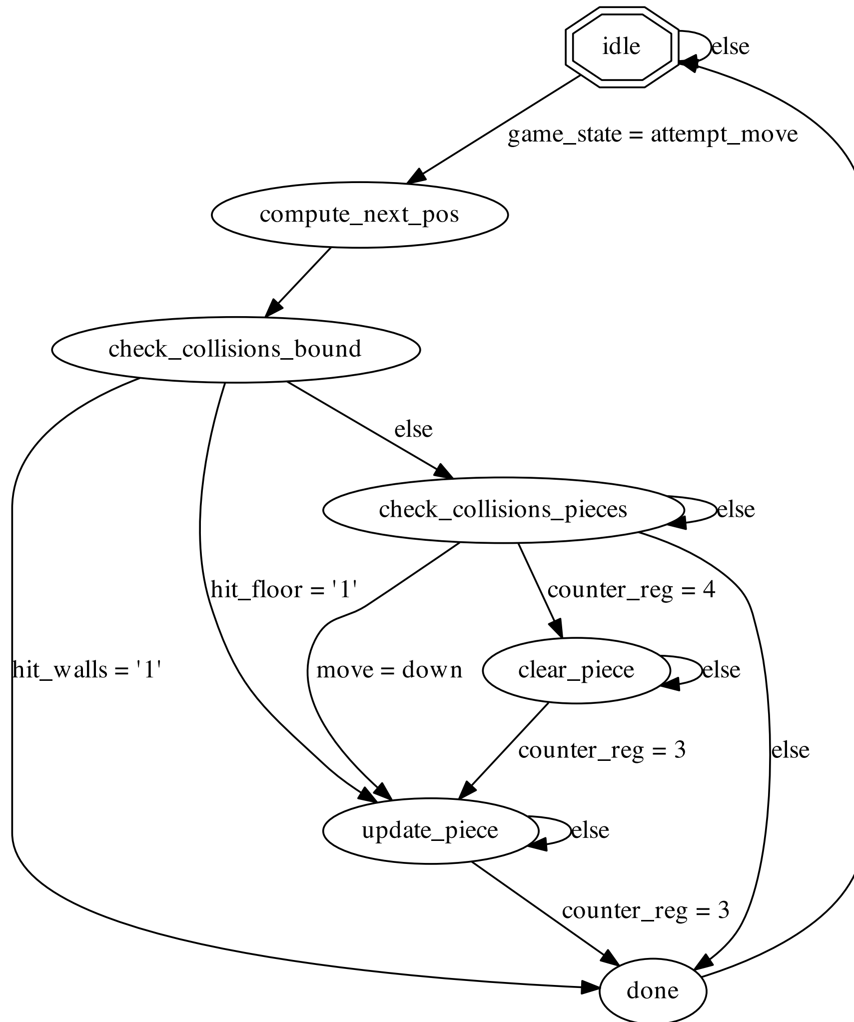


Figura 23: Macchina a stati finiti di `piece_mover`

compongono il tetramino di una posizione sull'asse x o y. La gestione della rotazione invece non è altrettanto immediata anche se l'idea di base è esattamente la stessa. L'unica differenza risiede nel fatto che in questo caso ogni quadretto avrà una diversa traslazione sull'asse x e y a seconda del tipo del tetramino e del suo stato di rotazione precedente. Per mantenere la leggibilità del file `piece_mover.vhd` abbiamo deciso di spostare la gestione della rotazione in una `function` chiamata `rotate_piece` definita in `logic_pkg.vhd`. La funzione implementa opportune traslazioni in modo che i tetramini ruotino esattamente come mostrato in Figura 19 a pagina 23. Per esempio, per far ruotare il tetramino T dallo stato 1 allo stato 2 occorrerà traslare il quadretto 3 di -1 sull'asse x e -1 sull'asse y.

check_collisions_bound

In questo stato si controlla se la mossa appena calcolata porterebbe il tetramino ad entrare in collisione con i bordi della griglia e si modificano opportunamente le relative variabili di controllo. In particolare, `hit_floor` viene posta a 1 se una o più celle del tetramino appena generato hanno oltrepassato il bordo inferiore della griglia durante un movimento verso il basso, mentre `hit_walls` viene posta a 1 se il tetramino appena generato si trova parzialmente o totalmente oltre i bordi laterali o il bordo inferiore della griglia durante un altro movimento. Nel caso di `hit_floor` si procede a impostare il registro `stop_reg` a 1, che verrà poi utilizzato per emettere il segnale di `stopped` al modulo che gestisce la logica e si procede ad aggiornare il contenuto della RAM (`update_piece`), mentre nel caso di `hit_walls` si lascia lo stato di gioco invariato e si passa direttamente in stato di `done`. Altrimenti occorre verificare se il tetramino appena mosso è parzialmente o totalmente sovrapposto a celle già occupate, per cui si passa allo stato `check_collisions_pieces`.

check_collisions_pieces

In questo stato la macchina mantiene un registro `counter_reg` che va da 0 a 4 e inizialmente posto a 0 per tenere traccia della cella che si sta controllando. Legge le coordinate della cella di indice `counter_reg` dal registro `piece_new_reg` e le pone come indirizzo in lettura dalla RAM, rimanendo nello stato `check_collisions_pieces`. Al successivo colpo di clock, quando il dato è pronto in `ram_out` verifica se la cella memorizzata è vuota oppure se il flag di `moving` per quella cella è impostato. In questi casi rimane nello stesso stato e procede a leggere la cella successiva del tetramino, altrimenti significa che si è verificata una collisione, quindi occorrerà impostare `collision_found_reg` a 1, lasciare il pezzo invariato e concludere le operazioni portando la FSM in `done`. Analogamente al caso della collisione con il bordo inferiore della cella, se si verifica una collisione mentre il tetramino si sta muovendo verso il basso si dovrà invece impostare `stop_reg` a 1 e procedere con l'aggiornamento del tetramino (`update_piece`). Se invece tutte le celle sono state controllate e non si è rilevata alcuna collisione, si passa a `clear_piece` e si reimposta il registro `counter_reg`.

clear_piece

In questo stato la FSM si comporta in modo analogo a `check_collisions_pieces`, nel senso che utilizza il registro `counter_reg` per scandire le quattro celle che compongono il tetramino, ma anzichè verificarne lo stato si limita a impostare il contenuto degli indirizzi corrispondenti in RAM a 0 (cella vuota). Al termine, passa in `update_piece` reimpostando `counter_reg` a 0.

update_piece

In questo stato la FSM si comporta in modo analogo a `clear_piece` ma anzichè scrivere 0 in memoria la macchina scrive il tetramino appena generato in RAM. Nel caso in cui il tetramino vada fermato (`stop_reg` = 1) il flag di `moving` viene impostato a 0, altrimenti a 1. Al termine, il registro `counter_reg` viene reimpostato come di consueto e la macchina passa in `done`.

done

In questo stato si forniscono opportunamente i segnali in uscita `stopped` e `collision_found` a seconda del valore dei registri `stop_reg` e `collision_found_reg`. La macchina si riporta quindi in `idle`, reimpostando adeguatamente tutti i registri a 0.

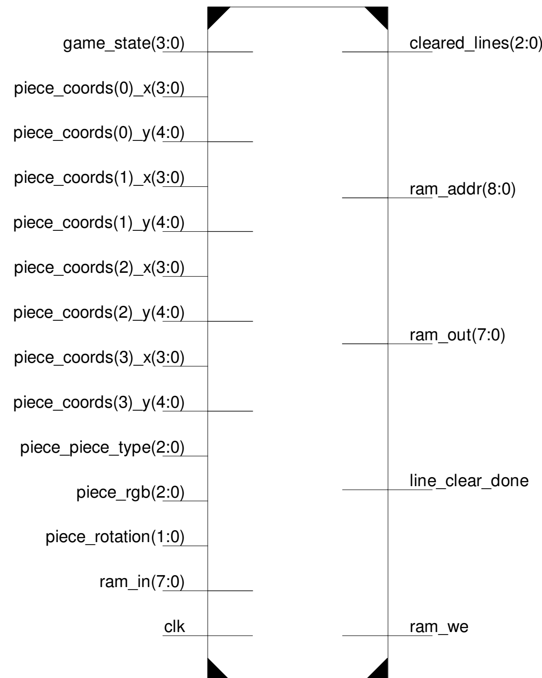


Figura 24: `line_clearer`

6.3 Eliminazione delle linee piene

Ogni volta che un tetramino si ferma nel suo movimento di caduta, è necessario verificare se una o più linee della griglia sono completamente piene, e in quel caso occorre eliminarle e spostare le linee sovrastanti verso il basso. Questo è il compito del modulo `line_clearer`, mostrato in Figura 24 e dei moduli `line_checker` e `line_shifter` che istanzia.

Il lavoro di questo modulo è sostanzialmente sequenziale e si articola in tre parti fondamentali:

Verifica delle linee Ogni linea viene scandita cella per cella. Nel caso in cui la linea risulti piena viene marcata per l'eliminazione. Si nota che in questo passaggio occorre verificare esclusivamente le linee interessate dal tetramino appena fermato, visto che sono le uniche che sono effettivamente cambiate rispetto allo stato precedente.

Cancellazione delle linee piene Ogni linea così rilevata viene eliminata, cioè sostituita con una linea vuota.

Spostamento delle altre linee verso il basso Quando tutte le linee piene sono state marcate per l'eliminazione e svuotate, un modulo dedicato (`line_shifter`) provvede a spostare tutte le altre verso il basso in accordo con le regole di Tetris.

In particolare, abbiamo deciso di mantenere separate le ultime due parti, anche se avrebbero potuto venire svolte entrambe da `line_shifter`, in quanto una struttura di questo genere rende possibile l'introduzione di una piccola animazione al momento di cancellazione delle linee. Lo sviluppo di un modulo con questa funzionalità è stato valutato in fasi avanzate del progetto, ma, vista la buona giocabilità del Tetris anche senza questo componente, poi non realizzato.

Il modulo, oltre ai consueti segnali di clock, RAM e allo stato della macchina di `tetris_logic` prende anche in ingresso le quattro coordinate y del tetramino che è appena stato fermato sulla griglia di gioco, rappresentate da `piece_coords(i)_y`. La Figura 24 mostra anche altri segnali, la cui linea non entra però

nel modulo. Si tratta di segnali *unconnected* che sono comunque presenti per via del fatto che in VHDL abbiamo specificato il parametro di ingresso come `record`, anche se non ne utilizziamo tutti i campi. In fase di sintesi il software ottimizzerà questo modulo eliminando i segnali non necessari.

Questo modulo implementa una macchina a stati, rappresentata in Figura 25. Di seguito sintetizziamo le operazioni eseguite in ciascuno stato.

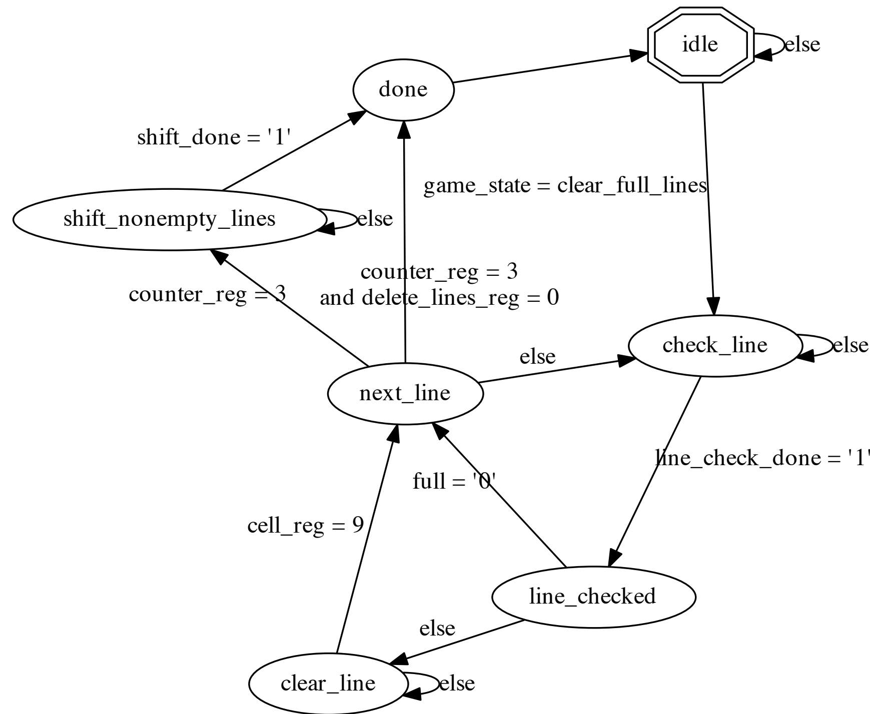


Figura 25: Macchina a stati finiti di `line_clearer`

idle (stato iniziale)

Normalmente la macchina si trova in questo stato e attende che la FSM di `tetris_logic` raggiunga lo stato, letto tramite `game_state`, di `clear_full_lines` per avviarsi. In questo stato il contenuto di tutti i registri e counter utilizzati è 0.

check_line

L'ingresso in questo stato attiva il modulo `line_checker`, che prende in ingresso la coordinata y della cella `counter_reg` del tetramino (`piece_coords(counter_reg)_y`) e, al termine delle sue operazioni, alzerà il segnale `line_check_done` e porterà il segnale `full` a 1 se la linea corrispondente è piena, 0 altrimenti.

line_checked

La macchina entra in questo stato quando la linea è stata scandita. Se la linea non è piena, si passa a verificare la prossima linea (stato `next_line`), altrimenti si aggiorna lo stato del registro `delete_lines_reg` che serve a marcare le linee per l'eliminazione. Questo registro è grande 18 bit, uno per linea, e viene aggiornato semplicemente ponendo a 1 il bit corrispondente alla coordinata y della linea piena e si raggiunge lo stato `clear_line`.

next_line

Se non ci sono più linee da controllare (cioè se `counter_reg` è uguale a 3) e nessuna linea è stata eliminata, si passa semplicemente in `done`. Se invece il registro `delete_lines_reg` ha alcuni bit a 1 significa che alcune linee sono state eliminate e sarà necessario far scorrere la griglia verso il basso. Per questo si raggiunge lo stato `shift_nonempty_lines`. Se invece `counter_reg` non ha ancora raggiunto 3, vi sono ancora linee da controllare e dunque la macchina torna in `check_line` dopo aver incrementato `counter_reg`.

clear_line

Quando la macchina è in questo stato, il segnale di write enable della RAM viene alzato, sul bus dati viene caricato 0 e la parte dell'indirizzo corrispondente a y viene collegata alla coordinata y della linea appena controllata. Per quanto riguarda la parte x la si collega al registro `cell_reg`, inizializzato a 0 e incrementato ad ogni colpo di clock in cui si mantiene la macchina nello stato `clear_line`. Quando il registro raggiunge 9 (la coordinata x massima della cella) il registro viene reimpostato a 0 e la macchina passa a `next_line`.

shift_nonempty_lines

L'ingresso in questo stato attiva il modulo `line_shifter` che provvede ad aggiornare la griglia rimuovendo le linee eliminate (marcate con 1 in `delete_lines_reg`) e facendo scorrere opportunamente le altre. Quando il segnale `line_shift_done` va alto, il modulo ha completato il suo lavoro e la FSM può a sua volta entrare nello stato `done`.

done

Il segnale di `line_clear_done` di questa macchina è portato a 1, i registri vengono reimpostati e la macchina torna in `idle`.

6.4 Controllo delle linee

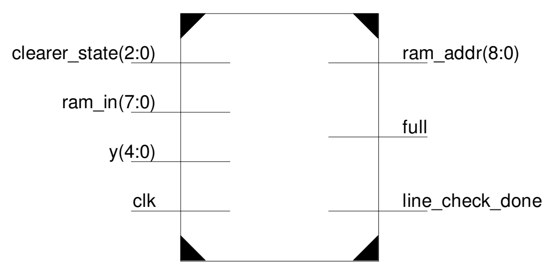


Figura 26: `line_checker`

Il modulo `line_checker`, mostrato in figura 26, si occupa di verificare se una linea è piena, cioè se tutte le sue celle sono occupate oppure no.

Riceve in ingresso lo stato della macchina di `line_clearer`, più la coordinata `y` della linea da controllare, in output fornisce il segnale `full` che indica se la linea è piena e il proprio segnale di completamento `line_check_done`. È inoltre fornito dei segnali per l'uso della RAM in lettura.

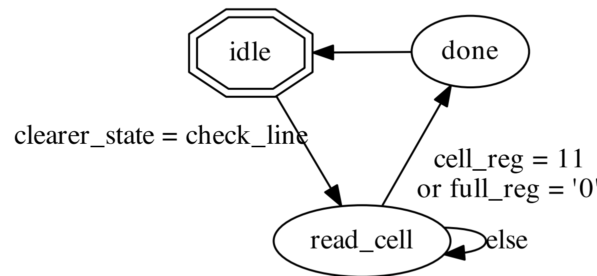


Figura 27: Macchina a stati finiti di `line_checker`

idle (stato iniziale)
La macchina attende che `line_clearer` arrivi in stato `check_line`. Quando ciò accade, porta il valore del registro `full_reg` a 1 e si porta in `read_cell`. In questo stato il registro `cell_reg` è a 0.

read_cell
In questo stato viene portato sul bus indirizzo della RAM il valore relativo alle coordinate (`cell_reg`, `y`). Si ricorda che `y` è un valore che il modulo riceve in ingresso, impostato da `line_clearer`. Se il valore di `cell_reg` è 11, cioè uno più del totale delle celle che compongono la linea, significa che la scansione è terminata. La macchina può dunque procedere in stato `done`. Idem dicasi per il caso in cui al colpo di clock precedente si sia trovata una cella vuota (`full_reg` = 0). Si può concludere che la linea non è piena e dunque terminare la scansione. Se nessuna delle due condizioni non si verifica, invece, si va a leggere il contenuto del bus dati della RAM. Il suo valore al primo colpo di clock (`cell_reg` = 0) è scartato in quanto la lettura dalla RAM non è istantanea e richiede un ciclo di clock, mentre per gli altri istanti un valore di 0 fa portare `full_reg` a 0 mentre altri valori fanno semplicemente incrementare il registro `cell_reg` di uno per il controllo della cella successiva.

done

Il segnale `line_check_done` viene alzato e il registro `cell_reg` viene reimpostato a 0.

6.5 Spostamento delle linee

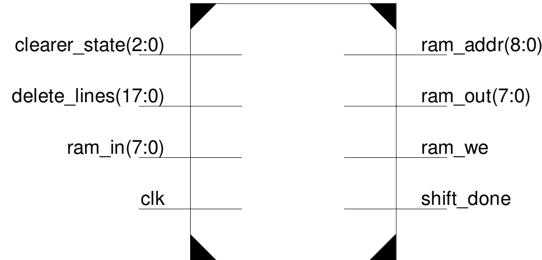


Figura 28: `line_shifter`

Quando una o più linee vengono eliminate occorre spostare tutte le altre verso il basso, come previsto dalle regole di Tetris. La gestione di questa funzionalità è compito del modulo `line_shifter`, mostrato in Figura 28.

Il modulo riceve in ingresso lo stato della macchina di `line_clearer`, lo stato del registro `delete_lines_reg` descritto in precedenza (qui chiamato `delete_lines`) ed è provvisto dei segnali per l'accesso in scrittura alla RAM e del proprio segnale di completamento `shift_done`.

L'idea di base è che il modulo riscriva completamente la griglia quando lo stato della macchina di `line_clearer` entri in `shift_nonempty_lines`. Per fare questo, anche `line_shifter` implementa una macchina a stati, il cui funzionamento è mostrato in dettaglio in Figura 29, che una volta avviata procede a trascrivere linea per linea tutto il contenuto della griglia tralasciando le linee vuote, cioè quelle marcate con 1 in `delete_lines`. Di conseguenza, una volta entrata nel ciclo, la FSM semplicemente leggerà una cella alla volta e la riscriverà al ciclo di clock successivo, eventualmente leggendo e scrivendo da e verso linee diverse. Per assolvere il suo scopo, la macchina opera su tre registri: `y_reg` che rappresenta la coordinata y della cella da scrivere, `cell_reg` che rappresenta la coordinata x della cella da leggere e scrivere e `delta_reg` che rappresenta la differenza, in numero di linee, tra la linea che legge e la linea che scrive. Questa differenza corrisponde esattamente al numero di linee bianche incontrate fino a quel momento durante la scansione della griglia. Il registro `delta_reg` dovrà quindi essere aggiornato prima della lettura di ogni linea, cosa che succede quando la FSM è nello stato `update_delta`. Visto che la macchina esce dallo stato di `idle` solo se ci sono linee da spostare, `delta_reg` al termine delle operazioni assumerà un valore da 1 a 4. Arrivati alle ultime linee sulla cima della griglia quindi si otterrà un valore di `y_reg` + `delta_reg` maggiore o uguale al numero di linee totale (18). In qual caso, anzichè andare a leggere il valore della cella dalla RAM in quel punto si inserirà semplicemente una linea vuota.

Al termine, anche in questo caso si alza il relativo segnale di `done`, si riporta la macchina in `idle` e si reimpostano i registri.

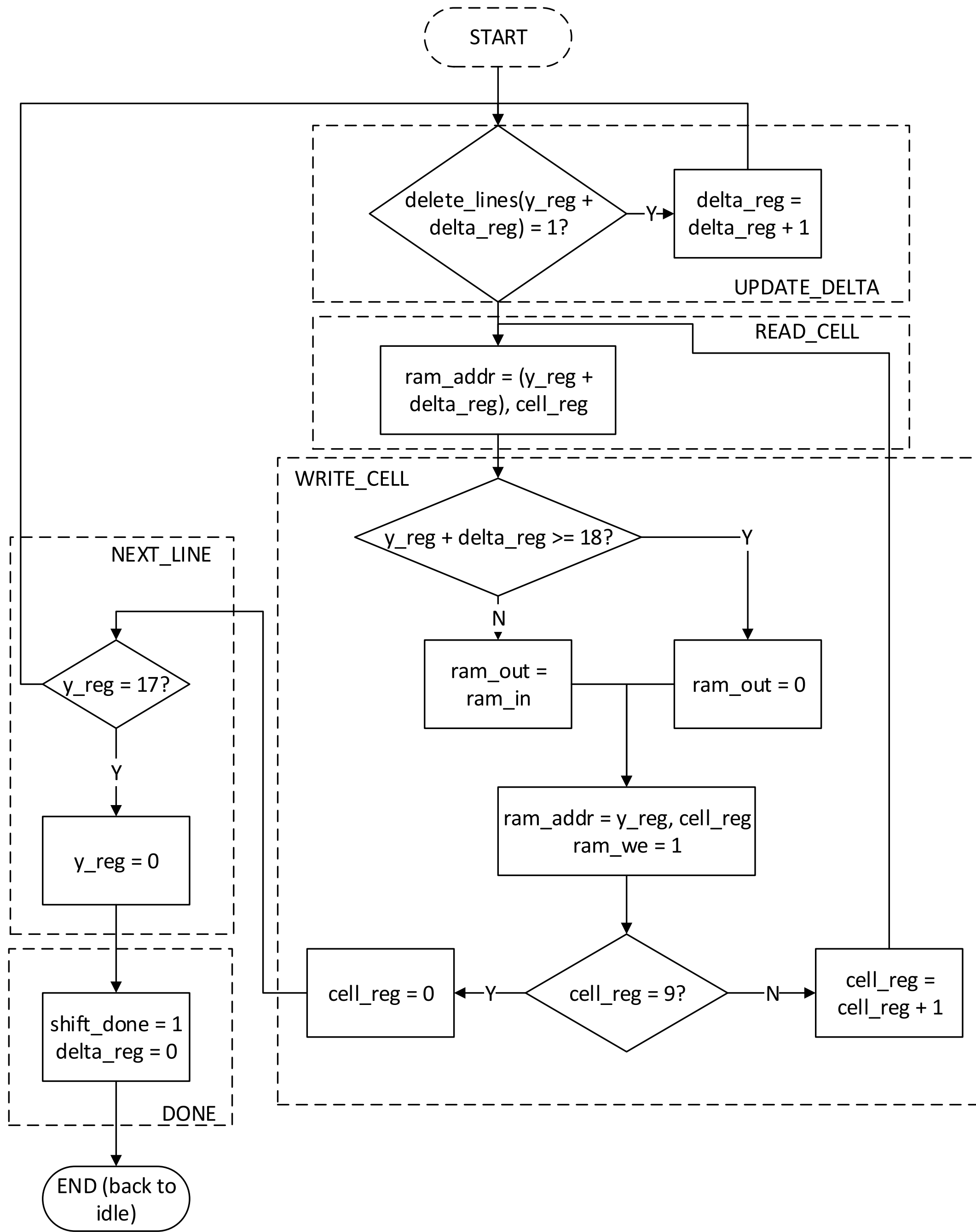


Figura 29: Diagramma di funzionamento di `line_shifter`

6.6 Punteggio

Il calcolo del punteggio si ispira al metodo di calcolo adottato dalla Nintendo. Il punteggio dipende dal numero di linee riempite contemporaneamente e dal livello raggiunto e viene calcolato in base alla Tabella 3.

Livello	**1 Linea**	**2 Linee**	**3 Linee**	**4 Linee**
0	40	100	300	1200
1	80	200	600	2400
2	120	300	900	3600
...	...	...	...	...
9	400	1000	3000	12000
n	$40 * (n + 1)$	$100 * (n + 1)$	$300 * (n + 1)$	$1200 * (n + 1)$

Tabella 3: Calcolo del punteggio nel Tetris

Il modulo `score_counter`, mostrato in Figura 30, si occupa di aggiornare il punteggio totalizzato dal giocatore ogni volta che vengono cancellate delle linee. Inoltre, il modulo incrementa il contatore delle linee cancellate e, in base al valore di esso, viene aumentato il livello di gioco. In particolare, il livello aumenta ogni 10 linee cancellate. Anche per questa scelta implementativa ci siamo ispirati al meccanismo di *leveling* adottato dalla Nintendo. Il gioco si compone di 16 livelli, ognuno associato ad una differente velocità. All'aumentare del livello i tetramini scendono più velocemente. Una volta raggiunto il livello più elevato il gioco procede mantenendosi alla velocità massima.

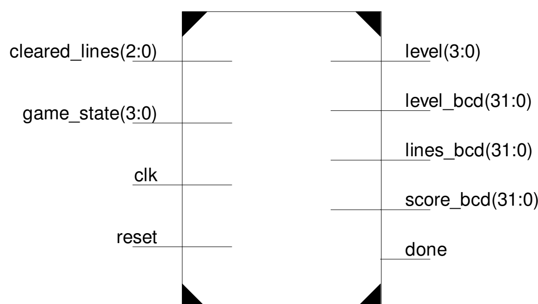


Figura 30: `score_counter`

Il modulo, oltre al segnale di clock e al segnale di stato della macchina `tetris_logic` prende in input anche il numero di linee (`cleared_lines`) eventualmente cancellate con la mossa appena effettuata e un segnale di `reset` che viene attivato ogni volta che si inizia una nuova partita e ha come effetto l'azzeramento dei contatori.

In uscita dal modulo si ottengono i segnali `score_bcd`, `lines_bcd`, `level_bcd` che sono le rappresentazioni in BCD rispettivamente dei contatori del punteggio, delle linee cancellate e del livello raggiunto. Questi segnali vengono ottenuti dal modulo `bin2bcd` istanziato da `score_counter` e andranno a costituire l'input del modulo `font_gen` che si occuperà di visualizzare sullo schermo i contatori in decimale.

Viene fornito in uscita anche un segnale `level` che rappresenta il livello raggiunto. Questo segnale è collegato al modulo `tetris_logic` il quale regola la velocità della partita in base al valore assunto dal segnale stesso.

Nel modulo viene implementata una macchina a stati, il cui grafo di transizione degli stati viene illustrato in Figura 31. Di seguito illustriamo quali operazioni vengono effettuate in ciascuno stato.

idle (stato iniziale)

La macchina si trova principalmente in questo stato, in attesa di leggere da `game_state` che la FSM

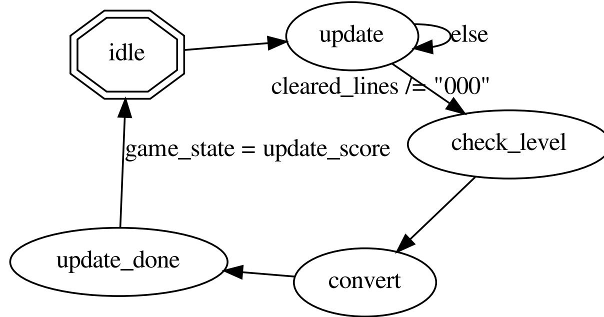


Figura 31: Macchina a stati finiti di `score_counter`

di `tetris_logic` sia entrata nello stato `update_score`. Quando questa condizione è verificata, la macchina si avvia.

update
In questo stato vengono aggiornati i contatori del punteggio e delle linee cancellate, ma solo se è opportuno farlo cioè quando sono state effettivamente cancellate delle linee (`cleared_lines` $\neq 0$). In tal caso, dopo l'aggiornamento, la macchina procede a controllare se è necessario aumentare il livello di gioco (stato `check_level`) altrimenti rimane nello stato corrente.

check_level
In questo stato viene implementato un contatore modulo 10 che tiene traccia delle linee cancellate. Quando il contatore torna a 0, il livello (`level_reg`) viene incrementato di un'unità.

convert
L'ingresso in questo stato attiva il modulo `bin2bcd` che si occupa di convertire in BCD i valori dei contatori del punteggio, delle linee e del livello. La FSM va immediatamente in `update_done` senza attendere che il convertitore termini la propria esecuzione poiché non necessita del risultato di esso e rallenterebbe lo svolgersi del gioco.

update_done
Il segnale `done` viene alzato e la macchina torna in `idle`.

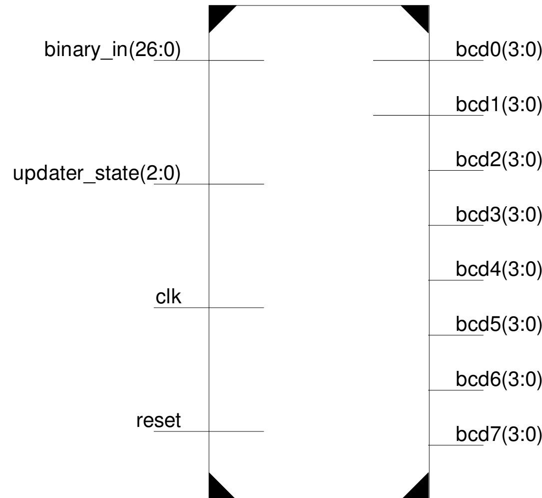


Figura 32: `bin2bcd`

6.7 Convertitore da binario a BCD

Per poter essere visualizzati come testo, i numeri a più cifre devono essere convertiti in BCD. Per questo motivo è necessario un convertitore *binary-to-*BCD che si occupa di convertire opportunamente i segnali `score`, `lines` e `level`.

Il modulo `bin2bcd` in Figura 32 realizza la conversione in BCD di un segnale binario dato in ingresso, fornendo in uscita le 8 cifre BCD corrispondenti. Dato che questo modulo prende un solo segnale in ingresso, lo `score_counter` istanzia tre moduli `bin2bcd` ognuno dei quali provvederà a convertire un segnale specifico.

La conversione viene effettuata utilizzando un algoritmo "*Shift and add 3*", descritto a pagina 147 di [3], che opera nel modo seguente:

1. I bit vengono shiftati verso sinistra, così che il MSB del numero binario diventa il LSB della prima cifra BCD (`bcd0`), quindi il MSB di `bcd0` diventa il LSB di `bcd1` e così via.
2. Dopo ogni shift, vengono controllati tutti i registri BCD. Se il valore nel registro è maggiore di 4, allora viene sommato 3.
3. Si ripete dal passo 1 finché tutti i bit sono stati shiftati.

La conversione viene controllata da una macchina a stati finiti il cui grafo delle transizioni è illustrato in Figura 33. La macchina è costituita da tre stati, descritti di seguito.

start (stato iniziale)
: La macchina rimane in attesa che la FSM di `score_counter` entri nello stato `update_score` per essere attivata. Quando si avvia, in questo stato vengono azzerati tutti i registri e precarica il numero binario da convertire leggendo `binary_in`.

shift
: In questo stato viene controllato il numero di bit che sono stati shiftati. Se tutti i bit sono stati shiftati (`shift_counter = N_BIN`) allora la macchina passa nello stato `done`. Altrimenti rimane nello stato corrente e prosegue con le operazioni di shift. Le somme con 3 vengono effettuate utilizzando il codice concorrente fuori dalla FSM.

done
: La conversione è terminata e la macchina può tornare nello stato iniziale pronta per una nuova conversione.

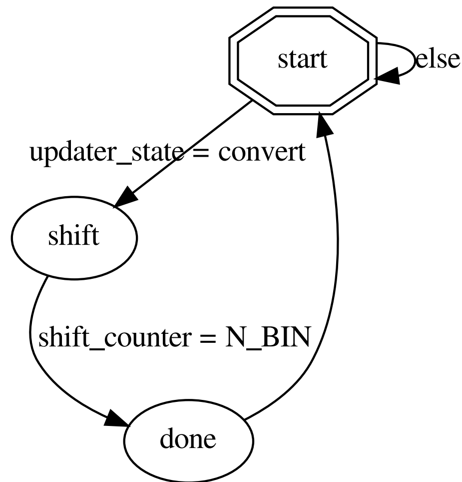


Figura 33: Macchina a stati finiti di `bin2bcd`

6.8 Reset della griglia

Il modulo `board_reset` in Figura 34 si occupa di pulire la griglia di gioco prima di ogni partita. Esso prende in ingresso sul segnale `game_state` lo stato della FSM di `tetris_logic` e fornisce in uscita i segnali di gestione della RAM e un segnale di `reset_done`.

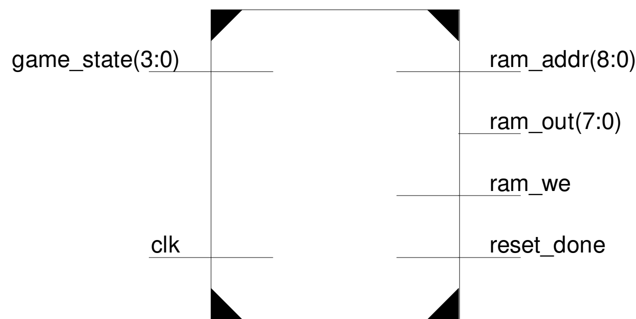


Figura 34: `board_reset`

Questo modulo implementa una macchina a stati finiti, il cui grafo di transizione degli stati è illustrato in Figura 35. Descriviamo di seguito le operazioni svolte in ciascuno stato.

idle (stato iniziale)
La macchina si attiva quando la FSM di `tetris_logic` entra nello stato `reset_board`. Una volta attivata, si pone in `writing`.

writing
In questo stato viene azzerato il contenuto di ogni cella, scorrendo la griglia per righe. Una volta terminata la procedura di pulizia della griglia, la macchina entra nello stato `done`.

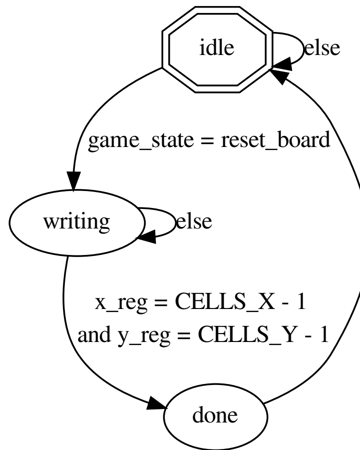


Figura 35: Macchina a stati finiti di `board_reset`

done

L'ingresso in questo stato causa il risveglio della FSM di `tetris_logic` per mezzo del segnale `reset_done` che viene alzato, dopo di che la macchina si porta in stato `idle`.

6.9 Generazione di numeri casuali

La sequenza dei tetramini che compaiono durante una partita deve essere generata in maniera pseudocasuale per rendere ogni partita il più possibile differente dalle precedenti. Il modulo `random_gen` illustrato in Figura 36 si occupa di generare tale sequenza, prendendo in ingresso sul segnale `input` il comando dato dal giocatore attraverso il controller e restituendo il successivo numero casuale (`rnd`) della sequenza, il quale corrisponderà ad un particolare tipo di tetramino.

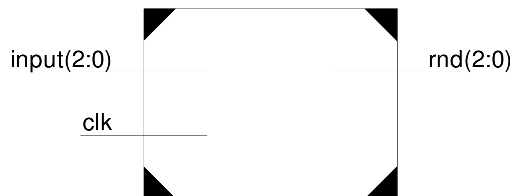


Figura 36: `random_gen`

Dato che non è presente alcun segnale sulla Spartan-3 che può essere direttamente utilizzato come generatore di numeri casuali, abbiamo deciso di implementare un PRNG utilizzando *Linear Feedback Shift Register*. Un LFSR è un registro a scorrimento il cui bit di input è ottenuto dalla combinazione lineare di alcuni bit dello stato precedente. Il modulo `lfsr`, in Figura 37, implementa un registro a 8 bit di questo tipo. Il `linear_feedback` è dato dallo XOR tra i bit 7 e 3 del registro. La formula di aggiornamento del registro sarà dunque

$$b' = b_{[6..0]} \| (b_7 \oplus b_3)$$

dove b_i indica l'i-esimo bit del registro, $b_{[i..j]}$ indica tutti i bit del registro dall'i-esimo al j-esimo (inclusi), $\|$ rappresenta l'operatore di concatenazione e b' rappresenta il valore del registro all'istante successivo.

Figura 37: `lfsr`

Dato che l'output del modulo è di soli 3 bit avremmo potuto decidere di utilizzare direttamente un *LFSR* a 3 bit ma, per via della breve durata del periodo di un *LFSR* a 3 bit abbiamo deciso di utilizzarne uno a 8 bit e poi fornire in output una ulteriore combinazione lineare dei bit che compongono il registro utilizzando la formula

$$rnd = (b_6 \oplus b_4 \oplus b_2) \| (b_7 \oplus b_3 \oplus b_1) \| (b_5 \oplus b_0 \oplus b_4)$$

Il registro continua a cambiare stato ad ogni colpo del clock da 50 MHz, e l'output verrà emesso di conseguenza. Questo accorgimento però non basta per ottenere una sequenza pseudocasuale abbastanza soddisfacente, infatti i tetramini nella sequenza tendono a ripetersi troppo spesso diminuendo significativamente la giocabilità. Il motivo di questo comportamento probabilmente risiede nel fatto che il registro è sincronizzato con lo stesso clock che stabilisce l'intervallo di tempo che intercorre tra ogni verifica dello stato di gioco e pilota la macchina a stati che genera i nuovi tetramini. La periodicità dell'aggiornamento del registro e dello stato potrebbe portare il registro a entrare "in fase" con la logica di gioco, sbilanciando la probabilità dei vari stati di output in favore di alcuni valori.

Per ovviare a questo problema il numero in uscita dal modulo `random_gen` è ottenuto combinando l'output di ben due moduli `lfsr`, uno dei quali viene pilotato, oltre che dal clock, anche dalla presenza di input fornito dal giocatore. La presenza di input viene collegata al segnale `ce` che diventa alto quando `input /= no_command` abilitando così il secondo `lfsr`. Il numero random viene quindi ottenuto effettando lo XOR tra le uscite dei due moduli `lfsr`.

Il primo modulo quindi produce un valore che è solo funzione del tempo per cui è rimasta accesa la macchina, mentre il secondo produce un valore che dipende da quante volte il giocatore ha premuto pulsanti del controller e per quanto tempo (in termini di cicli di clock da 50 MHz). Una volta implementata, questa combinazione ha dato risultati soddisfacenti dal punto di vista della giocabilità.

7 Musica

L'ultimo componente che a questo punto rimane da sviluppare per ottenere una versione di Tetris fedele a quella originale è la musica di sottofondo. La scelta del tema musicale da riproporre in questa versione del Tetris ricade sulla canzone popolare russa *"Korobeiniki"*, utilizzata come tema principale nella versione per Game Boy.

La Spartan-3 non è equipaggiata di altoparlanti, per questo motivo è stato realizzato un connettore che si collega da un lato agli *Expansion Connector* della board e dall'altro si collega ad un paio di speaker per mezzo di un'uscita audio jack da 3,5 mm.

La riproduzione della musica trova la sua realizzazione nel modulo `music` illustrato in Figura 38, il quale prende in ingresso il clock a 50MHz e divide la frequenza del clock generando un nuovo segnale ad onda quadra di minor frequenza che viene collegato in uscita allo *speaker*. Cambiando la frequenza del segnale in uscita, la FPGA produce suoni differenti. Questa è l'unica soluzione adottabile che non prevede l'uso di un *Digital to Analog Converter*.

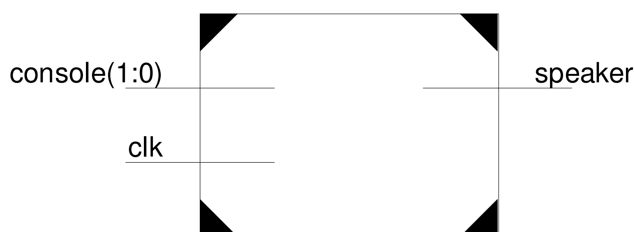


Figura 38: `music`

Lo spartito [7] delle note che compongono il tema musicale viene trascritto in una ROM. Le singole note vengono estratte in ordine dalla ROM istanziando un contatore a 30 bit (`tone_reg`) ed utilizzando gli 8 bit più significativi come indice per la ROM. La nota estratta si trova sul segnale `fullnote`.

Dividendo per 12 il segnale `fullnote` si ottengono l'*ottava* della nota da riprodurre e la *nota* stessa leggendo rispettivamente il quoziente e il resto della divisione. Qui entra in gioco il modulo `divide_by12`, il quale prende in ingresso il segnale `fullnote` e ne calcola la divisione per 12, dividendo prima per 4 e successivamente per 3. La divisione per 4 risulta banale in hardware poiché consiste nel rimuovere i 2 bit meno significativi di `fullnote` e copiarli in `remainder[1:0]`. La divisione per 3 viene effettuata per mezzo di una lookup table. In uscita dal modulo si ottiene sui segnali `quotient` e `remainder` rispettivamente l'ottava e la nota da riprodurre.

Nota	`clkdivider`
A	512
A♯ / B♭	483
B	456
C	431
C♯ / D♭	406
D	384
D♯ / E♭	362
E	342
F	323
F♯ / G♭	304
G	287
G♯ / A♭	271

Tabella 4: Valori di `clkdivider` in funzione della nota

La nota viene quindi riprodotta istanziando un contatore `counter_note` modulo `clkdivider` il cui valore cambia in funzione della nota che si vuole far suonare, e un contatore `counter_octave` che divide ulteriormente il segnale di clock per generare l'ottava. Per ottenere l'ottava più bassa si divide `counter_note` per 256, per l'ottava superiore si divide per 128 e così via. Il passaggio da un semitono al successivo si ottiene moltiplicando la frequenza della nota per 1.0594, ma essendo difficoltoso da implementare in hardware, viene implementato per mezzo di una lookup table con valori precomputati come descritto in [6]. La Tabella 4 mostra i valori assunti da `clkdivider` in funzione della nota desiderata. In questo modo dividendo il clock per 512 si ottiene la nota LA, dividendo il clock per 483 si ottiene la nota LA♯ e così via. Si noti che più basso è il divisore del clock e più è alta la nota ottenuta.

Infine, abbiamo adottato la convenzione che quando il segnale `fullnote` assume il valore 0, questo rappresenti una pausa e quindi per la durata della pausa non viene generato un suono in uscita.

Lo schema in Figura 39 mostra il funzionamento del componente e riassume quanto detto finora.

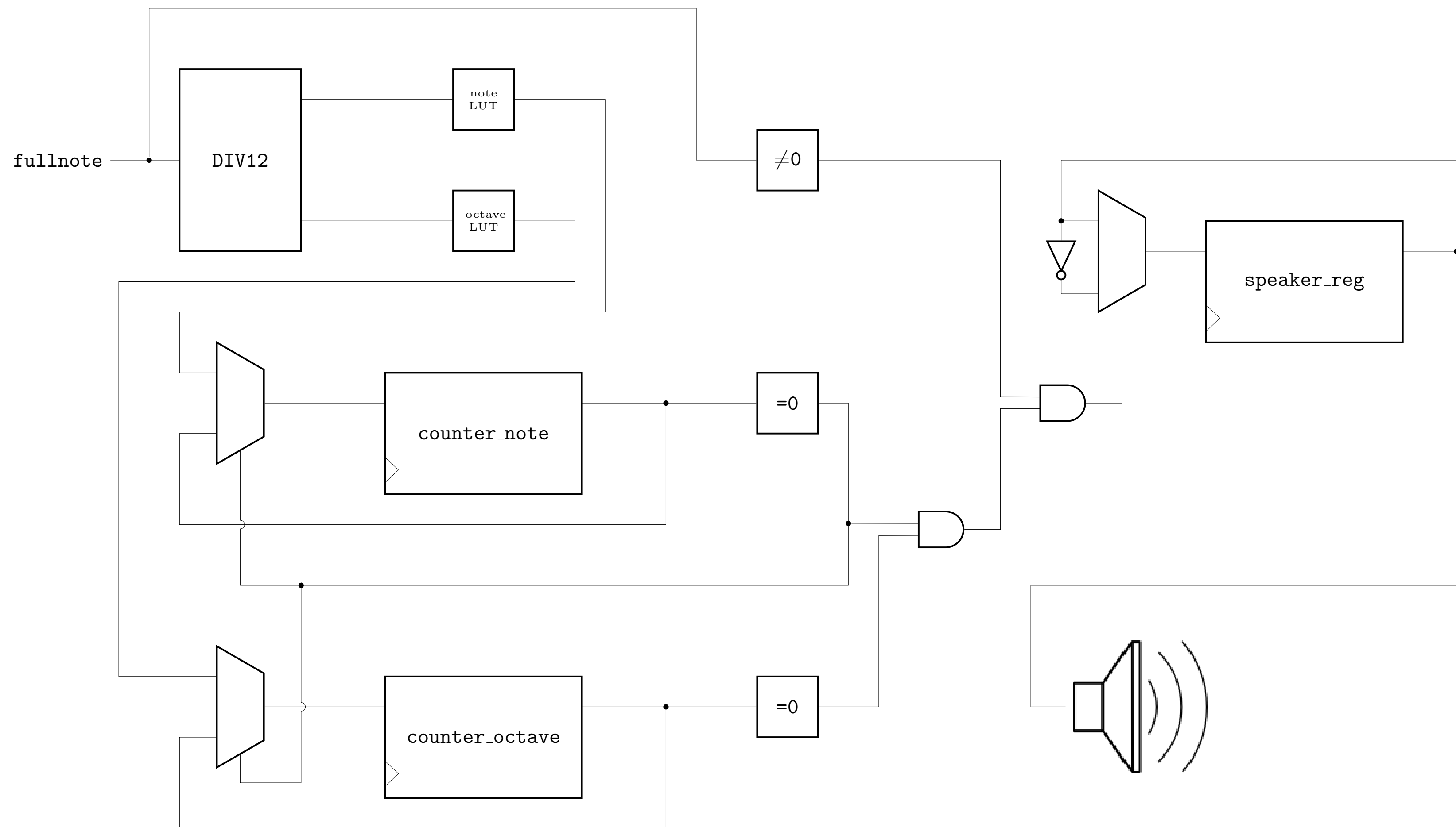


Figura 39: Struttura del sintetizzatore

8 Conclusioni e sviluppi futuri

Il progetto è stato portato a termine con successo e tutti gli obiettivi preposti sono stati raggiunti, realizzando tutte le funzionalità previste in fase di progettazione.

Prima della realizzazione del progetto stesso, sono state sviluppate semplici applicazioni di test su FPGA per meglio comprendere il funzionamento delle singole componenti equipaggiate sulla board, permettendoci così di gestirle in maniera ottimale.

Possibili sviluppi futuri si possono trovare nello sviluppo di un'animazione durante la cancellazione delle linee piene. La modularità del progetto stesso permette di aggiungere questa funzionalità senza la necessità di modificare significativamente i sorgenti che lo costituiscono. Una seconda possibile miglioria è quella di collegare un *Digital to Analog Converter* (DAC) per ottenere una miglior gestione della musica riprodotta dal circuito, consentendo in questo modo di riprodurre un tono polifonico sicuramente più piacevole all'udito a differenza del tono monofonico su onda quadra riprodotto in questa versione. Analogamente, si potrebbe costruire un piccolo circuito alternativo di uscita video munito di DAC in modo da ottenere una profondità di colore superiore a 3 bit.

Rendiamo nota anche una possibile idea per la realizzazione di un'indicatore di "criticità" della situazione della partita in corso, che può cambiare in base al livello raggiunto o al numero di linee della griglia occupate. Più precisamente l'idea è la realizzazione di un'*emoji* che rappresenti uno stato emotivo più preoccupato al crescere del livello raggiunto o al numero di linee occupate. Questa idea prende spunto dal comportamento dei *Bubble Dragon* nel famoso gioco arcade *"Puzzle Bobble"* che cambiano stato d'animo in base all'altezza raggiunta dalle bolle che compongono il puzzle nella griglia di gioco.

Infine, possiamo affermare che nonostante alcune difficoltà che si sono poste durante lo sviluppo del codice, la realizzazione di questo progetto è stata impegnativa ma piacevole oltre ad averci permesso di acquisire esperienza nel campo della progettazione digitale su FPGA e sull'uso del linguaggio VHDL.

Ringraziamenti

Desideriamo ringraziare il Prof. Pedersini per averci messo a disposizione la scheda Spartan-3, aver realizzato i connettori indispensabili per collegare il gamepad e gli speaker alla FPGA e per la sua disponibilità durante tutto lo sviluppo del progetto.

Un ringraziamento va a Lollacci, per averci prestato il gamepad facendoci giurare solennemente di conservarlo con estrema cura, e a Chiara per averci ospitato nel suo laboratorio e averci conseguentemente sopportato durante la realizzazione, soprattutto nelle fasi più critiche in cui noi stessi avremmo fatto fatica a sopportarci.

Infine, ringraziamo tutti coloro che si sono prestati per la fase di *testing* del progetto.

Erik Calligari, *Luca Guerra*

Riferimenti bibliografici

[1] Xilinx Inc., *Spartan-3 FPGA Family Data Sheet*, 2013.

[2] V. A. Pedroni, *Circuit Design With VHDL*. Mit Press, 2004.

[3] P. P. Chu, *FPGA Prototyping by VHDL Examples*. Wiley, 2008.

[4] Xilinx Inc., *Spartan-3 Starter Kit Board User Guide*, 2005.

[5] `http://www.cs.cmu.edu/~chuck/infopg/segasix.txt`.

[6] `http://www.fpga4fun.com/MusicBox.html`.

[7] `http://pianosquall.com/wp-content/uploads/2012/11/Tetris-Korobeiniki1.pdf`.

A Note tecniche/operative sulla scheda e sull'ambiente Xilinx ISE

Durante lo svolgimento del progetto abbiamo utilizzato la scheda FPGA Spartan-3 XC3S200 messa a disposizione dal Laboratorio di Architetture Digitali (DALab). Si tratta di un modello non recentissimo che richiede alcuni accorgimenti per essere programmato tramite computer moderni. Di seguito riportiamo alcune note sull'uso della scheda nella speranza che possano essere utili a chi desiderasse utilizzare la board per altre applicazioni.

Ambiente di sviluppo

L'ambiente di sviluppo da utilizzare per programmare la FPGA in dotazione è Xilinx ISE. Tuttavia, a partire dal 2012, Xilinx Inc. offre la suite Vivado in sostituzione a ISE come prodotto software per la programmazione di FPGA/CPLD. Vivado non è però compatibile con le schede della serie Spartan, la cui produzione è stata interrotta con la serie Spartan-6 nel 2010. Rimane comunque la possibilità di scaricare ISE dalla pagina `http://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/design-tools.html`. La soluzione più ovvia è quella di provare la versione più recente del software, cosa che suggeriamo di fare per prima cosa. Noi abbiamo utilizzato con successo sia la versione 14.6 sia la notevolmente più datata 10.1 SP3. Di seguito forniamo istruzioni per l'installazione e il collegamento dell'hardware per la versione 14.6. Se si dovessero riscontrare problemi con tale versione, come a noi è accaduto su una macchina di sviluppo, aggiungiamo anche le istruzioni per ISE 10.

Collegamento dell'hardware con ISE 14.6

La scheda Spartan-3 che abbiamo utilizzato è dotata di un cavo con connettore DB25 per porta parallela (LPT), di cui le moderne schede madri non sono più fornite. Per utilizzare la scheda è quindi necessario essere in possesso di una vecchia macchina con connettore parallelo. Noi abbiamo utilizzato un computer con processore Pentium 4 e sistema operativo Windows XP messoci a disposizione dal laboratorio CLUB. Fortunatamente, visto anche il notevole sforzo computazionale richiesto per eseguire la sintesi e il Place-and-Route da ISE, non è necessario svolgere lo sviluppo sul computer dotato di porta parallela ma lo si può utilizzare semplicemente come server e collegarci una macchina di sviluppo più potente via rete. Per impostare in questo modo l'ambiente sono necessarie le seguenti operazioni:

- Sulla/e macchine di sviluppo installare ISE WebPACK (da `http://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/design-tools.html` seguire il link "Full installer for Linux" o Windows). Il software installato occupa circa 15 GB.
- Sulla macchina server installare ISE LabTools (da `http://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/design-tools.html` seguire il link Lab Tools - All platforms), verificando che la versione di LabTools sia la stessa di ISE installato sui client. Il software installato occupa circa 2 GB.
- Collegare la FPGA al server tramite la porta parallela.
- Sul server (si suppone Windows) eseguire `$XILINX_INSTALL_DIR$\LabTools\LabTools\bin\nt\cse_server.exe [-port <portnumber>]`.
- Sul client, una volta avviato ISE e raggiunta la fase di configurazione dispositivo (iMPACT, Boundary Scan) selezionare Output → Cable setup... e impostare sotto "Communication mode" Parallel III, Parallel IV o MultiPRO, sotto "Cable location" selezionare "remote" e inserire l'indirizzo IP del server e la porta su cui e' avviato il server (visibile come messaggio di stato sul server).
- Un messaggio di stato sulla parte inferiore della finestra di iMPACT segnala il corretto collegamento al server. È ora possibile inizializzare la *chain* con CTRL+I.

Prima di programmare la scheda occorre assicurarsi che i jumper `M0` ed `M2` siano disinseriti mentre il jumper `M1` sia inserito. Il jumper `JP1` deve inoltre essere posto in configurazione `Default`. Al contrario, se si desidera avviare la scheda con la configurazione salvata sulla *PlatformFlash* occorre inserire tutti e tre i

jumper `M0`, `M1` e `M2`. Per salvare i dati sulla *PlatformFlash* occorre invece selezionare le adeguate opzioni di iMPACT mantenendo i jumper in configurazione di programmazione.

Finché si tratta di utilizzare ISE su un solo client non dovrebbero verificarsi particolari problemi, a parte che occasionalmente iMPACT perde la preferenza del server. In quel caso è necessario semplicemente tornare in Output → Cable setup... e cambiare l'impostazione del cavo (una tra Parallel III, Parallel IV e MultiPRO). Diverso è il caso in cui si utilizzi più di una macchina come client di sviluppo. In tal caso non è possibile avere iMPACT aperto sullo stesso cavo su due macchine diverse. Occorre chiuderlo su una macchina prima di aprirlo su un'altra, ma spesso anche con questo accorgimento il server potrebbe mantenere il lock sul sistema chiuso, ponendosi nell'imbarazzante situazione di rimanere bloccato su un computer disconnesso e richiedere di venire riavviato (semplicemente rilanciando `cse_server.exe`). Per evitare per quanto possibile la situazione suggeriamo di selezionare Output → Disconnect ogni volta che sia necessario passare il controllo da un computer all'altro.

Installazione di ISE 10.1 SP3

È possibile scaricare ISE 10.1 WebPACK SP3 da questa pagina `http://www.xilinx.com/webpack/classics/wpclassic/`. Scaricare sia il pacchetto 10.0 sia il service pack 3, da installare in sequenza. Abbiamo riscontrato problemi durante la procedura di installazione, in particolare il codice di attivazione di WebPACK generato dal sito non sbloccava il programma di installazione. Dopo una richiesta al servizio clienti Xilinx abbiamo ottenuto un codice funzionante per ISE 10.1.

Windows 32 bit È sufficiente eseguire setup.exe dopo aver scompattato l'archivio e seguire le istruzioni a schermo.

Windows 64 bit Siamo stati tentati a utilizzare setup.exe anche su questa piattaforma, ma sulle nostre macchine l'installatore andava in crash all'inizio della procedura. Abbiamo trovato invece un programma di installazione funzionante sempre nello stesso archivio, sotto `webpack\bin\nt\setup.exe` o `_setup.exe`.

Linux L'installazione di questa versione su Linux ha presentato non pochi problemi, per alcuni dei quali non siamo riusciti a trovare una soluzione. Questa versione del software richiede che i pacchetti `openmotif`, `portmap` e `libstdc++` siano già installati. Una volta scompattato l'archivio, l'installazione può procedere lanciando lo script `xsetup` e seguire le istruzioni a schermo. Terminata l'installazione, abbiamo riscontrato la presenza di due errori, il fallimento del processo di *Place & Route* e l'FPGA editor (*PACE*) che non si apre. Il primo problema pare dovuto alla versione recente di `glibc` non più supportata da ISE 10.1. Per risolvere il problema è sufficiente creare un wrapper al programma `trce` seguendo i passaggi seguenti:

1. scaricare l'archivio all'indirizzo `http://forums.xilinx.com/xlnx/attachments/xlnx/EDK/19927/1/trce-wrapper.tar.gz` e scompattarlo (ad esempio in `/tmp`), verrà creata la cartella `trce-wrapper`
2. entrare nella cartella con `cd /tmp/trce-wrapper` e lanciare `make`
3. `cd $XILINX_INSTALL_DIR$/bin/lin`
4. `mv -i trce trce.real`
5. creare un link simbolico a `trce` con il comando `ln -s trce unwrapped/trce.real`
6. `cp /tmp/trce-wrapper/trce-wrapper trce`

Per quanto riguarda il problema di *PACE*, siamo riusciti solo a risolvere l'errore "`libXm.so.3 not found`" creando un link simbolico alla nuova versione di libXm con il comando `ln -s libXm.so.XX libXm.so.3` (dove `XX` è la versione di tale libreria installata sul sistema) nella cartella `/usr/lib`.

ISE 10 non distingue tra versione completa e Lab Tools. Se si programma la scheda in configurazione client-server occorre quindi installare sul server e client la stessa versione (service pack compreso). Le

versioni del server e di iMPACT di due versioni di ISE differenti non sono compatibili tra di loro. Sul server (si suppone Windows) eseguire `$ISE_INSTALL_DIR$\bin\nt\CableServer.exe [-P <porta>]` e procedere sui client come per la versione 14.6.

Elenco delle figure

1 Tetris su FPGA . . . 2
2 I differenti tetramini nel gioco del Tetris . . . 3
3 Struttura dei moduli . . . 5
4 Schema implementativo delle macchine a stati . . . 7
5 sram_module . . . 10
6 Schema della sincronizzazione orizzontale VGA . . . 12
7 vga_sync . . . 13
8 graphic_gen . . . 14
9 Schema di funzionamento del modulo graphic_gen . . . 14
10 board_gen . . . 15
11 Schema del funzionamento di board_gen . . . 16
12 font_gen . . . 17
13 Atari font . . . 17
14 Font pattern della lettera A . . . 18
15 Estrazione dei segnali utili alla Font ROM . . . 18
16 next_indicator_gen . . . 19
17 Connettore DB-9 del gamepad. . . . 20
18 sega_controller_driver . . . 21
19 Rotazione dei tetramini . . . 23
20 tetris_logic . . . 25
21 Macchina a stati finiti di tetris_logic . . . 26
22 piece_mover . . . 28
23 Macchina a stati finiti di piece_mover . . . 29
24 line_clearer . . . 31
25 Macchina a stati finiti di line_clearer . . . 32
26 line_checker . . . 34
27 Macchina a stati finiti di line_checker . . . 34
28 line_shifter . . . 35
29 Diagramma di funzionamento di line_shifter . . . 36
30 score_counter . . . 37
31 Macchina a stati finiti di score_counter . . . 38
32 bin2bcd . . . 39
33 Macchina a stati finiti di bin2bcd . . . 40
34 board_reset . . . 40
35 Macchina a stati finiti di board_reset . . . 41
36 random_gen . . . 41
37 lfsr . . . 42
38 music . . . 43
39 Struttura del sintetizzatore . . . 45