

SoundPlayer: Lettore walkman di file Wav da scheda SD

Progettazione di Sistemi Digitali

Luca Ceragioli, Marco Ferrini

Anno Accademico 2022/2023



Indice

Indice.....	1
Introduzione.....	2
Caratteristiche principali.....	2
Sfide tecniche.....	2
Quick start guide.....	2
Dettagli implementativi.....	3
Schede SD.....	3
File system FAT32.....	3
File Audio WAV.....	3
Buffer Audio.....	4
Debug con Logic Analyzer.....	4
Architettura.....	5
Schema a Blocchi.....	5
SDcard Reader.....	6
SD Card main fsm.....	8
SD Card cmd fsm.....	11
SPI master.....	13
FAT32 Reader.....	15
RAM dualport.....	18
Codec Interface.....	19
Codec Config.....	21
I2S Master.....	22
Main PLL.....	23
Codec audio WM8731.....	23
Debug e funzionamento.....	24
Protocollo SPI scheda SD.....	24
Protocollo I2C codec.....	24
Protocollo I2S codec.....	25
Lettura di un blocco della scheda SD.....	25
Lettura di un cluster FAT32.....	26
Funzionamento a regime.....	27
Conclusioni.....	28

Introduzione

Il progetto consiste nel prototipo di un riproduttore portatile di file musicali, comunemente chiamato “walkman”. I file audio sono memorizzati su scheda SD, utilizzando la scheda di sviluppo TERASIC DE2. Il lettore permette di ascoltare brani audio, di metterli in pausa e di saltare da un brano all’altro scorrendoli in ordine.

- Link ai **sorgenti** del progetto: <https://github.com/VirtuContraFurore/SoundPlayer>
- Video dimostrativo: https://www.youtube.com/watch?v=giQW_ChssqA

Caratteristiche principali:

- Compatibile con schede SDHC o SDXC formattate FAT32
- Riproduce file audio di tipo WAV a 44.1 kHz sia mono che stereo
- Uscita audio su jack stereo da 3.5 mm
- Bottone “**Play/Pause**”
- Bottone “**Next Track**”
- Bottone “**Restart Track/Previous Track**”¹
- Permette di navigare fino a 80 brani diversi



Sfide tecniche:

- Implementare i protocolli di comunicazione con periferiche **I2C**, **I2S** e **SPI**
- Realizzare la sequenza di inizializzazione di schede SD di nuova generazione
- Rilevamento e lettura di una partizione FAT32 valida
- Lettura dei file attraverso la *File Allocation Table* del file system **FAT32**
- Sincronizzazione del **buffer audio** con la lettura del file dalla scheda SD
- Debug real-time attraverso l’uso di un **logic analyzer**

Quick start guide

1. Formattare una scheda SD da almeno 4GB col file system FAT32, scegliendo, se possibile, *allocation unit* di 8 kbytes o superiore.
2. Ottenere dei file audio WAV. Se si possiede il file in un altro formato, per convertirlo si può usare lo script allegato al progetto, che sfrutta il programma open source `ffmpeg` per ottenere file audio compatibili.²
3. Copiare i file nella scheda SD. Non è necessario riformattare la scheda ogni volta che si aggiunge o rimuove un file: questo perchè il file system viene effettivamente letto e non si sfrutta l’allocazione contigua tipica delle partizioni vergini.
4. Inserire la scheda SD, programmare la board DE2 e collegare delle cuffie all’uscita audio!

¹ Se premuto dopo 5 secondi dall’inizio della traccia fa ripartire la stessa canzone, altrimenti, se premuto entro 5 secondi dall’inizio della traccia, va alla canzone precedente.

² File compatibili: file con estensione WAV, struttura RIFF, encoding pcm16s con sampling frequency di 44.1 kHz, traccia mono o stereo.

Dettagli implementativi

Schede SD

Non sono supportate le schede di vecchia generazione (quelle con capacità minore od uguale a 2GB) in quanto la sequenza di startup legacy non è stata implementata. Le schede SD più recenti, ossia quelle SDHC o SDXC hanno una nuova sequenza di inizializzazione come riportato dalle specifiche *Physical Layer* disponibili sul sito <https://www.sdcards.org>.

La lettura delle schede SD avviene tramite interfaccia SPI e con blocchi di 512 bytes. Da quando viene richiesto un blocco a quando esso viene mandato al master passa un certo intervallo di tempo; per questo è meglio, quando possibile, leggere blocchi contigui: in questo caso il tempo di attesa è nullo.

Si riportano alcune risorse che di fatto riassumono le specifiche e ne agevolano la comprensione:

1. http://elm-chan.org/docs/mmc/mmc_e.html
2. <http://chlazza.nfshost.com/sdcardinfo.html>
3. <https://www.pjrc.com/tech/8051/ide/fat32.html>

File system FAT32

Probabilmente il punto centrale di tutto il progetto, dato che, assieme alle caratteristiche di lettura della scheda SD, definisce le modalità e le tempistiche per accedere ai file al suo interno, il file system FAT32 è stato scelto in quanto diffusissimo e largamente supportato da diversi sistemi operativi. Va notato che una partizione FAT32 può essere grande al massimo 32 GB, per questo alcuni programmi potrebbero rifiutarsi di formattare una scheda SD più grande con questo tipo di file system. Il problema è facilmente risolvibile accettando di avere parte della scheda inutilizzata – anche se, nella parte inutilizzata, vi si può mettere agilmente un'altra partizione.

Brevemente, il file system FAT32 possiede una tabella - detta ‘allocation table’, appunto - che permette di frammentare lo spazio di archiviazione in *cluster* di dimensione fissata (raccomandati 8 kbyte o superiore) il quale ordine di lettura viene ricostruito leggendo le catene di cluster dalla tabella stessa.

Il nostro progetto riproduce solo i file WAV contenuti della cartella “root” della scheda SD³. Per i dettagli sul sistema FAT32 si rimanda alla pagina:

https://it.wikipedia.org/wiki/File_Allocation_Table

File Audio WAV

I file audio supportati devono avere le seguenti caratteristiche:

- Container di tipo “RIFF”
- Nessun metadato
- Codifica pcm16s

³ In realtà, riproduce solo i file contenuti nel primo cluster della directory “root”, che corrispondono ad un massimo di circa 80 canzoni. La lettura dei cluster successivi al primo è implementabile estendendo la macchina a stati di lettura dei file; per i nostri scopi abbiamo ritenuto 80 canzoni sufficienti a dimostrare la funzionalità del progetto.

- Sampling rate 44.1 kHz
- Canali stereo o mono

Non tutti i file di tipo “WAV” vanno bene, in quanto ci sono diversi container che possono ospitare i campioni in codifica pcm. Si raccomanda di usare lo script “convert_audio_file.sh” allegato al progetto per avere un file compatibile. È uno script bash, quindi occorre avere una shell UNIX. Su windows è possibile usare *Cygwin* o *WSL* per avere una shell Linux e installare il programma *ffmpeg*.

Risorse utili:

- <http://soundfile.sapp.org/doc/WaveFormat/>

Buffer Audio

Viene impiegata la tecnica del *double buffering*: sono presenti due buffer di ugual dimensione: mentre uno viene riprodotto sull’uscita audio, l’altro viene riempito con i dati letti dalla scheda SD. È importante notare che, data la lettura a blocchi da 512 bytes della scheda SD, il buffer dovrà essere grande quanto un multiplo intero della dimensione del blocco di lettura. La memoria del buffer sfrutta i blocchi di tipo M4K disponibili dentro lo FPGA Altera.

Debug con Logic Analyzer

È stato utilizzato un fondamentale strumento in grado di farci vedere le forme d’onda digitali sui pin dello FPGA. Sono stati connessi, attraverso il verilog, i segnali di interesse ai GPIO della board DE2. In questo modo è stato possibile visualizzare ogni protocollo e controllarne il corretto funzionamento.

Lo strumento è un Saleae Logic a 8 canali. Con interfaccia USB, si adopera col programma messo a disposizione dall’azienda.

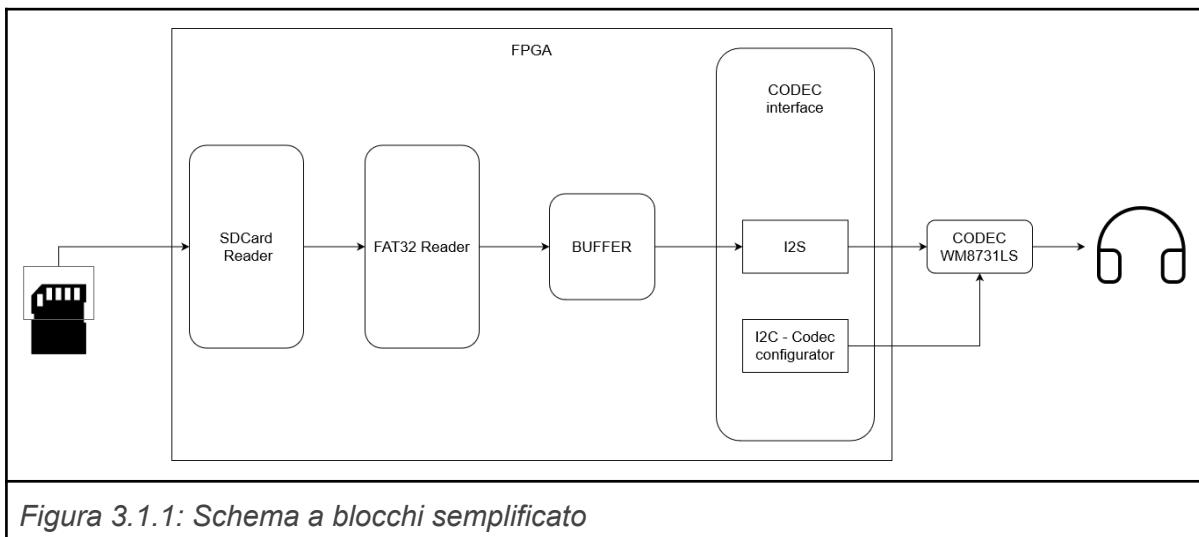
Sotto, in figura: analizzatore logico e cattura di segnali col tool *Logic2*.



Architettura

In questa sezione viene spiegata l'architettura di alto livello del progetto e, nelle sottosezioni seguenti, viene descritto ciascun sottoblocco.

Schema a Blocchi



In figura è illustrato lo schema a blocchi generale del sistema, ponendo l'attenzione sul flusso di dati:

- **SD Reader:** Preleva i dati dal lettore di schede SD attraverso il protocollo SPI.
- **FAT32 Reader:** Riceve i dati da SD Reader guidandolo nel file system della scheda SD.
- **Buffer:** Memoria RAM dual-channel, permette la sincronizzazione delle parte destra dello schema con la parte sinistra.
- **Codec Interface:** wrapper che contiene tutti i moduli responsabili della comunicazione con il Codec.
 - I2S module: Preleva i dati dal buffer e li invia al codec secondo il protocollo I2S.
 - Codec Configurator (I2C): Configura il Codec audio tramite il protocollo I2C.
- **Codec Audio WM8731:** Riceve il flusso di dati digitali dal blocco I2S e genera un segnale analogico per le cuffie.

Nel file Verilog `globals.v` ci sono le configurazioni del progetto, ossia:

- frequenze del clock globale e di ogni protocollo usato
- mapping dei bottoni alle loro funzioni

in questa maniera è possibile modificare agevolmente le frequenze di lavoro dei vari blocchi senza dover aprire file differenti.

SDcard Reader

SDcard Reader gestisce l'inizializzazione e la comunicazione con la scheda SD.

Esso è costituito in realtà da 3 sottomoduli ovvero:

- SDCard_main_fsm: macchina a stati principale che gestisce inizializzazione e comunicazione con la scheda SD.
- SDCard_cmd_fsm: macchina a stati che sia durante l'inizializzazione che poi nella successiva comunicazione con la scheda SD, decodifica le informazioni riguardanti i comandi da inviare alla scheda SD e ne gestisce la trasmissione.
- SPI_master: macchina a stati finiti che gestisce la comunicazione SPI con la scheda SD implementando il physical layer del protocollo.

Iniziamo un'analisi approfondita del modulo in questione partendo dal suo pinout, riferimento figura 3.2.1.

Input:

- **clk, rst_n**: clock e reset attivo basso.
- **block_read_block_addr[31:0]**: bus a 32 bit per specificare l'indirizzo del blocco da leggere nel file system della scheda SD.
- **block_read_trigger**: segnale che avvia la lettura dei blocchi della scheda SD.
- **block_read_continuous_mode**: flag che avvia la modalità di lettura continua della scheda SD.
- **sd_do**: SPI miso.

Output:

- **sd_clk**: clock inviato da SPI master alla scheda SD.
- **sd_di**: SPI mosi
- **sd_cs_n**: chip select relativo alla scheda SD
- **block_read_data_new_flag**: flag che si attiva quando viene ricevuto un nuovo byte relativo ad un blocco.
- **card_configured**: segnale che viene messo ad 1 quando il processo di inizializzazione, della scheda SD è terminato.
- **block_read_card_ready**: segnale che viene attivato quando SD card reader è pronto a leggere un nuovo blocco.
- **block_read_data_out[7:0]**: Bus che trasmette i dati ricevuti dalla scheda al FAT32_reader.
- **block_read_data_idx[8:0]**: Contatore che indica il numero di byte ricevuti relativi ad un blocco.

Procediamo con l'analisi dei sottomoduli.

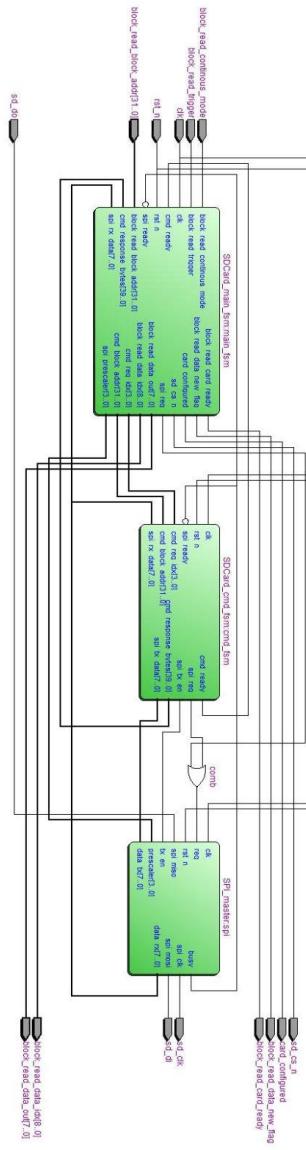


Figura 3.2.1: Schema a blocchi, SDCard Reader.

Table 3.

SD Card main fsm

Analisi del pinout del modulo.

Input:

- **clk, rst_n**: clock e reset attivo basso.
- **block_read_trigger**: vedi pinout SD reader
- **block_read_continuos_mode**: vedi pinout SD reader
- **block_read_block_addr**: vedi pinout SD reader
- **cmd_ready**: segnale di uscita della SD Card cmd fsm che viene attivato quando quest'ultima è nello stato di riposo e dunque quando è pronta a decodificare un comando
- **spi_ready**: segnale di uscita dell'SPI master che viene attivato quando quest'ultimo non è impegnato in trasmissione/ricezione.
- **spi_rx_data**: bus dei dati ricevuti da SPI master
- **cmd_response_bytes**: bus dati in uscita dall'SD Card cmd fsm in cui viene riportata la risposta ad un comando inviato, ricevuta tramite la periferica SPI.

Output:

- **block_read_card_ready**: vedi pinout SD reader
- **sd_cs_n**: vedi pinout SD_reader
- **spi_req**: segnale con il quale viene richiesto l'ultimo dell'interfaccia SPI.
- **cmd_req_idx**: bus dati in cui viene riportato il codice identificativo del comando da inviare
- **block_read_data_idx**: registro di uscita in cui viene riportato il numero di bytes inviati relativi al blocco in lettura
- **cmd_block_addr**: registro di uscita in cui viene riportato l'indirizzo del blocco della scheda SD da leggere ovvero block_read_block_addr.
- **block_read_data_out[7:0]**: vedi pinout SD reader
- **card_configured**: vedi pinout SD reader
- **block_read_data_new_flag**: vedi pinout SD reader
- **spi_prescaler**: bus in cui viene indicato il codice che identifica il fattore di divisione in frequenza per ottenere il clock di SPI master a partire dal master clock

La macchina a stati è organizzata in un unico blocco always e gli stati possono essere divisi in due macrogruppi:

- Relativi all'inizializzazione della scheda
- Relativi alla lettura dei blocchi della scheda.

Nel processo di inizializzazione lo stato di partenza è FSM_CARD_NOT_READY, stato in cui si approda a seguito di reset sincrono, qui viene settato cmd_req_idx con il codice relativo al comando NOCMD, ovvero un comando fittizio che serve a mantenere la macchina a stati SD Card cmd fsm nello stato di idle. Viene impostato spi_prescaler in modo che, durante la fase di inizializzazione, il clock di SPI master abbia l'opportuna frequenza. Infine si passa allo stato successivo: FSM_CONFIGURING_CARD_1.

Questo è uno stato di setup, si disattiva sd_cs_n, si setta spi_req e successivamente si aspetta che il SPI master risponda ad spi_req per passare allo stato successivo: FSM_CONFIGURING_CARD_2.

In FSM_CONFIGURING_CARD_2 vengono inviati 80 impulsi di clock con il CS disattivato, per permettere il reset della scheda successivamente viene attivato sd_cs_n.

Successivamente inizia la configurazione vera e propria della scheda SD con il susseguirsi di 5 stati, (da FSM_CONFIGURING_CARD_3 a FSM_CONFIGURING_CARD_7) nel quale vengono inviati opportuni comandi alla scheda SD. Ciò viene ottenuto settando opportunamente cmd_req_idx e passando da due stadi (FSM_RUN_CMD_0 e FSM_RUN_CMD_1) nel quale si aspetta l'esecuzione del comando.

I 5 comandi inviati sono, in ordine: CMD0, CMD8, CMD58, CMD55, ACMD41, per capirne la funzione si faccia riferimento alla sezione relativa alle schede SD a pagine 3.

Terminato questa prima fase di configurazione, si approda nello stato FSM_CONFIGURING_CARD_8, dove viene valutato il contenuto del LSB del registro R0, nel quale viene posto il segnale di ingresso costituito da cmd_response_bytes. Se il bit è nullo si può procedere con la configurazione altrimenti vengono rinvolti i comandi CMD55, e ACMD41.

Propriamente l'inizializzazione della scheda SD termina con lo stato FSM_CONFIGURING_CARD_9 nel quale, in modo analogo a prima, viene inviato il comando CMD58.

Infine la macchina evolve nello stato FSM_CONFIGURING_CARD_10 nel quale viene settato card_configured, e viene impostato il valore di spi_prescaler opportuno per quando la scheda è configurata.

Con l'evoluzione verso lo stato FSM_CARD_READY la macchina a stati inizia ad occuparsi della lettura del contenuto della scheda SD. Questo è lo stato da cui inizia la lettura di un nuovo blocco, infatti viene azzerato il contenuto di byte counter, contatore usato per tenere traccia del numero di bytes inviati relativi al blocco in lettura, e si attende l'attivazione di block_read_trigger. Giunta quest'ultima, si pone in cmd_block_addr l'indirizzo del blocco dati da leggere e si invia, dipendentemente dal valore di block_read_continuos_mode, il comando CMD18 o CMD17 che impongono rispettivamente lettura di più blocchi dati o di blocchi singoli.

Nello stato FSM_CARD_BUSY_0 la macchina valuta la corretta esecuzione dell'ultimo comando inviato, per poi eventualmente passare al successivo.

Negli stati FSM_CARD_BUSY_1 e FSM_CARD_BUSY_2 la macchina a stati attiva il flag spi_reg e, una volta che l'interfaccia SPI è pronta, aspetta che sul bus dati in ingresso siano presenti dati validi, per poi evolvere nello successivo stato.

FSM_CARD_BUSY_3 è uno stato di attesa.

Iniziano ora una serie di stati dediti alla gestione dei dati in ingresso partendo da FSM_CARD_BUSY_4, nel quale i bytes in ingresso (spi_rx_data) vengono riportati sul bus di uscita block_read_data_out, viene assegnato il contenuto di byte_counter a block_read_data_idx per poi essere incrementato. Nel frattempo viene settato il flag block_read_data_new_flag. Infine abbiamo uno statement condizionale che discrimina il caso in cui la ricezione del blocco dati è incompleta, dunque si torna allo stato

FSM_CARD_BUSY_3 per ricevere i bytes successivi, e il caso in cui la ricezione è completa, in quel caso la macchina evolve verso gli stati FSM_CARD_BUSY_CRC_1 e FSM_CARD_BUSY_CRC_2 in cui gestisce la ricezione dei due byte CRC.

Nello stato FSM_CARD_BUSY_CRC_3 abbiamo un nested conditional statement per determinare l'evoluzione della macchina, in cui se è non stata selezionata la continuos read mode si torna nello stato FMS_CARD_READY in cui si aspetta un nuovo segnale di trigger per la lettura di un nuovo blocco dati. Se invece la suddetta modalità di funzionamento è stata selezionata, se è richiesta la lettura di un nuovo blocco dati si torna allo stato FSM_CARD_BUSY_1 procedendo con la lettura del blocco contiguo o, viceversa, la macchina evolve verso lo stato FSM_CARD_BUSY_STOP_1.

Gli stati FSM_CARD_BUSY_STOP_1, FSM_CARD_BUSY_STOP_2, FSM_CARD_BUSY_STOP_3 sono dediti all'interruzione della modalità di lettura continua, la macchina, infatti, invia il relativo comando CMD12, e aspetta l'opportuna risposta da parte della scheda SD per poi evolvere in FSM_CARD_READY.

SD Card cmd fsm

Analisi pinout del modulo:

Input:

- **clk, rst_n**: clock e reset sincrono
- **cmd_req_idx**: bus di ingresso nel quale viene riportato il codice relativo al comando da inviare.
- **spi_ready**: vedi pinout SD Card main fsm.
- **spi_rx_data**: vedi pinout SD Card main fsm.
- **cmd_block_addr**: bus in cui viene riportato l'indirizzo del blocco dati da leggere.

Output:

- **cmd_ready**: flag con il quale la macchina a stati segnala che è nello stato di riposo e quindi pronta a ricevere nuove istruzioni.
- **cmd_response_bytes**: bus multi byte nel quale viene riportata la risposta della scheda ad un comando precedentemente inviato.
- **spi_tx_data**: bus dei dati da inviare alla scheda tramite l'interfaccia SPI.
- **spi_req**: vedi pinout SD Card main fsm.
- **spi_tx_en**: flag che abilita la trasmissione dati dell'SPI master.

La macchina a stati è organizzata in due blocchi always entrambi che descrivono logica sincrona e gli stati possono essere divisi in due macrogruppi:

- dediti alla trasmissione dei comandi.
- dediti alla ricezione della risposta da parte della scheda SD.

Il primo blocco always, quando la macchina stati è riposo, decodifica segnale riportato in cmd_req_idx inizializzando in modo opportuno la struttura dati del comando da inviare. Quest'ultima è card_cmd_packet, ovvero 6 bytes in cui i primi 5 sono dedicati agli argomenti del comando da inviare mentre l'ultimo byte è dedicato al codice di Cyclic Redundancy Check. Inoltre per ciascun comando decodificato viene indicato il numero di bytes della risposta corrispondente e viene attivato il flag pending_request che segnale la presenza di un comando da inviare.

Il secondo blocco always, invece, gestisce trasmissione dei comandi e ricezione delle risposte.

CMD_FSM_IDLE costituisce lo stato di riposo, viene inizializzato cmd_counter, ovvero un contatore con triplice funzione: tiene traccia del numero di byte che sono stati inviati relativi ad un comando, indica il byte da inviare e segnala il numero di bytes ricevuti relativi alla risposta ad un comando. Nello stato, inoltre, vengono settati a zero spi_req e spi_tx_en. Successivamente si aspetta che sia presente un comando da inviare ed eventualmente si procede verso lo stato CMD_FSM_SEND_CMD_1.

Questo costituisce uno stato di setup della trasmissione, infatti vengono settati spi_tx_en e spi_req per poi evolvere in CMD_FSM_SEND_CMD_2 nel quale si ha l'effettiva trasmissione dei dati. Qui, finché non è stato trasmesso l'intero comando viene mantenuto spi_tx_en alto e quando spi-ready è alto viene incrementato il numero di bytes inviati e si controlla di non aver terminato la trasmissione per poi eventualmente evolvere nello stato CMD_FSM_RECEIVE_CMD_RESP_2.

CMD_FSM_RECEIVE_CMD_RESP_2 e CMD_FSM_RECEIVE_CMD_RESP_1 costituiscono due stati di attesa nel quale la macchina aspetta di ricevere qualcosa dalla scheda SD per poi evolvere in CMD_FSM_RECEIVE_CMD_RESP_3 nel quale effettivamente si gestisce la ricezione. Qui infatti i bytes ricevuti in risposta vengono posti in uscita al blocco tramite il bus cmd_response_bytes, quando poi il numero di bytes egualgia il numero di bytes che costituiscono la risposta ad un comando si torna allo stato di riposo CMD_FSM_IDLE.

SPI master

Analisi del pinout del modulo:

Input

- **clk, rst_n**: clock e reset asincrono attivo basso.
- **spi_miso**: MISO dell'interfaccia SPI
- **req**: flag che segnala la presenza di una richiesta di utilizzo dell'interfaccia SPI, è controllato, tramite un OR dai due segnali spi_req delle due macchine a stati del SD Card reader.
- **tx_en**: segnale che abilita la trasmissione dati.
- **data_tx**: bus dei dati da trasmettere.
- **prescaler**: bus che indica il modulo del prescaler da utilizzare per ottenere il clock da inviare a SPI slave.

Output:

- **spi_mosi**: MOSI dell'interfaccia SPI.
- **spi_clk**: slave clock dell'interfaccia SPI.
- **busy**: segnale che indica quando il modulo non è nello stato di riposo.
- **data_rx**: bus dei dati ricevuti.
- **_spi_rising_edge**: vedi capitolo debug.
- **_spi_clock_counter**: vedi capitolo debug.
- **_spi_sub_counter**: vedi capitolo debug.
- **_spi_done**: vedi capitolo debug.
- **_spi_fsm**: vedi capitolo debug.

Inoltre sono presenti dei parametri da settare quando istanziamo il modulo:

- **SPI_PACKET_SIZE**: dimensione dei bus data_rx e data_tx.
- **SPI_MOSI_IDLE**: livello logico di riposo del MOSI.
- **SPI_CPOL**: livello logico di riposo di SPI clock.
- **SPI_CPHA**: regola la fase del clock, ovvero il fronte di clock in cui il ricevente campiona il segnale in ingresso.

il modulo è costituito da una funzione e due blocchi always che descrivono logica sincrona.

La funzione opera la decodifica del codice prescaler.

Un blocco always genera il clock per l'interfaccia SPI e i segnali di fronte in salita dello stesso clock. Se il flag interno spi_clk_en è attivo, il contatore sub_counter viene usato come divisore in frequenza del master clock, quando questo è uguale a sub_counter_cap, ovvero il fattore di divisione in frequenza decodificato dalla funzione sopra citata, allora viene decrementato il contatore clock_counter il cui LSB, concordemente a i valori di SPI_CPOL e SPI_CPHA, costituisce il clock del SPI. Quando invece spi_clk_en è disattivato vengono resettati entrambi i contatori.

Si nota che clock_counter viene inizializzato con il doppio del valore di SPI_PACKET_SIZE in modo tale che quando il contatore si azzerà si è completato trasmissione e ricezione e si può attivare il segnale done e _spi_done utile per il debug.

Il secondo blocco always invece costituisce la macchina a stati vera e propria che gestisce trasmissione e ricezione. Qui oltre allo stato di reset abbiamo lo stato di riposo FSM_IDLE nel quale se è richiesto l'utilizzo del SPI si attiva spi_clk settando l'opportuno valore del prescaler e se necessario si abilita la trasmissione.

Lo stato FMS_START gestisce la ricezione dati campionando i bit su spi_miso nei fronti in salita di spi_clk che vengono messi nel registro rx_buffer. Quest'ultimo è un registro di dimensione $2 \times \text{SPI_PACKET_SIZE}$ così facendo posso scrivere in una metà mentre il byte precedentemente ricevuto è disponibile in uscita tramite il bus data_rx. Quando poi il flag done è attivo si disabilita spi_clk si invertono le metà di rx_buffer destinate a lettura e scrittura e si ritorna in FSM_IDLE.

FAT32 Reader

Macchina a stati che gestisce la lettura del file WAV al livello di astrazione più alto. Quest'ultima offre un'interfaccia utente molto semplice attraverso il quale è possibile gestire la riproduzione dei vari file WAV presenti nella SD, e ricevere dei feedback sullo stato di lettura del file.

Analisi del pinout del modulo.

Input:

- **clk, rst_n**: master clock e reset sincrono attivo basso.
- **block_read_card_ready**: vedi pinout SD card reader.
- **block_read_data_new_flag**: vedi pinout SD card reader.
- **block_read_data_in**: bus dei dati letti dalla scheda SD.
- **block_read_data_idx**: vedi pinout SD card reader.
- **audio_buffer_empty_i**: Segnale per la sincronizzazione con il modulo Codec che legge dal buffer dati, viene messo a 1 quando il buffer è vuoto.
- **player_next_song_req_i**: richiesta di riproduzione di una nuova canzone, proveniente dall'interfaccia utente.
- **player_next_song_forward_i**: segnale che discrimina il caso in cui la nuova canzone riprodotta deve essere la precedente o la successiva nel file system.

Output:

- **block_read_trigger**: vedi pinout SD card reader.
- **block_read_continous_mode**: vedi pinout SD card reader.
- **block_read_block_addr**: vedi pinout SD card reader.
- **error_no_fat_found**: segnale di errore attivato quando il tipo di partizione utilizzato nella scheda SD non è FAT32.
- **audio_buffer_data_o**: bus dei dati da scrivere nel buffer.
- **audio_buffer_addr_o**: bus degli indirizzi in scrittura del buffer.
- **audio_buffer_wren_o**: segnale di write enable.
- **audio_buffer_filled_o**: Segnale per la sincronizzazione con il modulo Codec che legge dal buffer dati, viene messo a 1 quando il buffer è pieno.
- **audio_buffer_empty_ack_o**: segnale di acknowledgement relativo al segnale audio_buffer_empty_i.
- **wav_info_sampling_rate**: bus dati che trasmette le informazioni sul sampling rate del file WAV.
- **wav_info_audio_channels**: bus dati che specifica se il file WAV è mono o stereo.
- **player_next_song_req_ack_o**: segnale di acknowledgement relativo al segnale audio_buffer_empty_i.

La macchina a stati è costituita da un unico blocco always che descrive logica sincrona.

Gli stati di quest'ultima possono essere divisi in quattro macro gruppi .

- **FSM_READ_MBR**: dediti alla lettura del Master Boot Record della scheda.
- **FSM_READ_FAT**: dediti alla lettura del Boot Sector del FAT32.
- **FSM_PARSE_FILE_ENTRY**: dediti alla lettura della Root Directory Region.

- **FSM_PARSE_WAV_FILE**: dediti alla lettura della Data Region e in particolar modo di file WAV trovati.

Analizzando il primo macro blocco, nello stato **FSM_READ_MBR_1** si richiede la lettura del primo settore della memoria, ovvero il master boot record, si passa allo stato successivo, **FSM_READ_MBR_2**, e quando sono presenti dati letti dalla scheda si controlla che il file system sia il FAT32 e si salva nel registro interno `far32_start` il numero di byte presenti tra MBR e il primo settore del file system. Se il tipo di partizione utilizzata non è FAT32 la macchina si blocca e viene segnalato un errore tramite il segnale `error_no_fat_found`.

Analogamente al precedente, il secondo macro blocco inizia con **FSM_READ_FAT_0** che richiede la lettura del primo settore del file system, ovvero il boot sector. Di questo settore vengono salvati in appositi registri interni le informazioni contenute nel BIOS parameter block, ovvero informazioni sul file system del tipo:

- Il numero di Bytes per settore.
- Il numero di settori per cluster.
- Il numero di settori riservati all'inizio del file system.
- Il numero di File Allocation Tables.
- Il numero di settori per ogni FAT.
- Il numero di root cluster.

Una volta lette tali informazioni, la macchina evolve verso nello stato **FSM_READ_FAT_2** dove si aspetta che il buffer sia vuoto tramite gli stati **FSM_WAIT_BUFFER_0** e **FSM_WAIT_BUFFER_1** in cui, rispettivamente, la macchina aspetta il flag `audio_buffer_empty_i`, manda il corrispondente acknowledgement, aspetta che il flag sopracitato si disattivato e infine evolve verso lo stato **FSM_PARSE_FILE_ENTRY_0**.

La macchina evolve, in questo modo, nel macro gruppo dedito alla lettura della Root Directory Region. Qui si scansiona le entries della regione ricercando quella corrispondente ad un file WAV riproducibile. Per farlo, nello stato **FSM_PARSE_FILE_ENTRY_0**, è richiesta la lettura del settore corrispondente ad una entry e nello stato **FSM_PARSE_FILE_ENTRY_1** si salvano, in appositi registri, le informazioni riportate in tale settore, ovvero:

- Il nome del file.
- L'estensione del file.
- Gli attributi del file
- L'indirizzo del primo cluster del file.
- La dimensione del file in bytes.

Successivamente si controlla che il filename non sia 0x00, in quanto sta ad indicare che è l'ultimo file entry nella Root Directory Region, in quel caso si riparte dall'inizio della regione stessa con la scansione. Al contrario se il filename è diverso si incrementa o decrementa `dir_entry_idx` (il contatore che guida la scansione) dipendentemente dal segnale `search_backwards`. Infine si esegue un ulteriore controllo sulle informazioni ottenute sul file, il cui risultato confluiscce sul flag `file_good` tramite una porta AND. Questi controlli consistono nella verifica sulla corretta estensione del file (.wav), un controllo sugli attributi del file (si controlla che non si tratti di una subdirectory) e un'ulteriore verifica sul file name (si controlla che sia diverso da 0xE5 in quanto corrispondente a file precedentemente cancellati).

Se `file_good` è alto, è stato trovato un file WAV valido e dunque possiamo procedere

con la lettura.

La lettura del file WAV inizia con la richiesta dei settori della scheda SD relativi al primo cluster del file WAV, per fare ciò viene abilitata la modalità di lettura continua tramite block_read_continuos_mode successivamente, nello stato FSM_PARSE_WAV_FILE_1 si procede con la lettura dell'header del file ovvero 44 bytes, dal quale si ricavano informazioni sul file stesso:

- ID del file, 4 byte che contrassegnano il file come RIFF file.
- File type header: nel nostro caso deve essere WAVE.
- Format chunk header: nel nostro caso “fmt”
- Type of audio format.
- Number of channels: sono supportati solo file mono o stereo.
- Frequenza di campionamento: supportati segnali campionati a 44.1Khz.
- Data chunk header: nel nostro caso “data”.
- Dimensione del file.

Si verificano le informazioni ricevute affinché siano coerenti con quanto sopra riportato, se il controllo fallisce, la macchina evolve in FSM_PARSE_WAV_FILE_7 e da lì ritorna alla fase di parsing dei file, in caso contrario si continua con la lettura. Durante la ricezione dei dati relativi all'header del file nel buffer vengono scritti degli zeri.

Nello stato FSM_PARSE_WAV_FILE_2 avviene l'effettiva lettura della regione dati del file e la scrittura nel buffer. Quest'ultima è accompagnata da una serie di controlli: Quando è stato ricevuto un intero blocco della scheda SD la lettura continua solo se non sono stati ricevuti tutti i file relativi ad un cluster del file system e se il buffer non è pieno. Se il buffer è pieno viene attivato l'apposito flag audio_buffer_filled_o e passando per lo stato FSM_WAIT_BUFFER_0 si aspetta che il buffer sia svuotato. Se invece sono stati letti tutti i settori relativi ad un cluster la macchina evolve nello stato FSM_PARSE_WAV_FILE_3.

In questo stato si ha la richiesta settore della file allocation table relativo al cluster attualmente in lettura ottenendo in questo modo l'indirizzo del cluster successivo.

Se tale indirizzo corrisponde all'EOF si ritorna al parsing dei file, se così non è si procede con la lettura del file richiedendo i settori del cluster selezionato e si torna allo stato FSM_PARSE_WAV_FILE_2.

Infine si nota che tutte le volte che la macchina evolve nello stato FSM_WAIT_BUFFER_0 vengono letti i segnali di ingresso player_next_song_req_i e player_next_song_forward_i che implementano le funzionalità di “Next Track” e “Restart Track/Previous Track”.

RAM dualport

Modulo ottenuto per tramite MegaWizard Plug-In Manager, per questo motivo verrà illustrato esclusivamente il pinout, rimandando per maggiori informazioni alla documentazione Altera.

Input:

- **clock**.
- **data**: bus dati in scrittura
- **raddress**: bus indirizzi in lettura
- **wraddress**: bus indirizzi in lettura
- **wren**: segnale di abilitazione lettura

Output:

- **q**: dati in lettura.

Il clock di questo modulo è una versione sfasata di 180° rispetto al clock di sistema per garantire un corretto campionamento dei dati in ingresso.

Inoltre la gestione degli indirizzi è stata realizzata in modo tale che la RAM è come fosse divisa in due buffer distinti: in uno scrive FAT32_reader mentre nell'altro legge Codec, quando uno è pieno e l'altro è vuoto si invertono. Per ottenere questa funzionalità il MSB di entrambi raddress e wraddress è controllato dal modulo codec.

Codec Interface

Macchina a stati che costituisce l'interfaccia con il codec audio gestendo inizializzazione dello stesso e trasmissione dati. Quest'ultimo presenta dei sottomoduli istanziati al suo interno:

- **Codec_config**: modulo I2C dedicato alla corretta configurazione del Codec audio.
- **I2S_master**: modulo che gestisce la trasmissione dati tramite il protocollo I2S.

Analisi pinout del modulo.

Input:

- **clk, rst_n**: clock e reset sincrono attivo basso.
- **codec_pause_i**: segnale proveniente dall'interfaccia utente che permette di mettere in pausa la riproduzione del brano.
- **codec_buffer_filled_i**: Segnale per la sincronizzazione con il modulo FAT32 che scrive nel buffer dati, viene messo a 1 quando il buffer è pieno.
- **codec_buffer_empty_ack_i**: acknowledgement del segnale **codec_buffer_empty_o**.
- **codec_buffer_data_i**: bus dati provenienti dal buffer.
- **wav_info_audio_channels_i**: bus dati che specifica se il file audio è mono o stereo.
- **wav_info_sample_rate_i**: bus dati che specifica il sample rate del file audio.

Inout:

- **codec_i2c_sdat_io**: linea dati in/out proveniente dall'interfaccia I2C.

Output:

- **codec_aud_xck_o**: master clock line dell'interfaccia I2S.
- **codec_aud_bclk_o**: bit clock line dell'interfaccia I2S
- **codec_aud_dacdat_o**: data line dell'interfaccia I2S
- **codec_aud_daclrck_o**: word clock line dell'interfaccia I2S
- **codec_i2c_sclk_o**: slave clock interfaccia I2C.
- **codec_buffer_addr_o**: indirizzo in lettura del buffer dati.
- **codec_buffer_sel_o**: flag di selezione del buffer dati in cui leggere e di conseguenza anche di quello in cui scrivere.
- **codec_buffer_empty_o**: Segnale per la sincronizzazione con il modulo FAT32 che scrive nel buffer dati, viene messo a 1 quando il buffer è vuoto.

La macchina a stati è descritta tramite un unico blocco always che implementa logica sincrona.

Oltre allo stato di reset abbiamo due stati di wait: **FSM_WAIT_BUFFER_0** e **FSM_WAIT_BUFFER_1**. Qui la macchina azzerà il segnale **codec_buffer_empty_o** in caso sia stato ricevuto il corrispondente ack, se il modulo I2S invia il segnale **I2S_done** se ne azzerà il trigger e successivamente si aspetta che il buffer dati attualmente in scrittura sia pieno e si inverte con quello in lettura alzando il flag **codec_buffer_empty_o** per poi evolvere nello stato **FSM_CONSUMING_BUFFER**.

Qui si ha lo switch dei buffer se il buffer in scrittura è pieno e quello in lettura è vuoto (informazione contenuta nel flag **swap_buffer**) e si inizia la lettura aggiornando gli indirizzi.

Il registro interno sample_count è un contatore che ha un valore massimo pari a 1 se il file è mono, mentre se è stereo è pari a 3 questo permette di inviare opportunamente i campioni letti all'interfaccia I2S. Quindi in pratica in questo stato si inviano al codec 1 o 2 sample dipendentemente se il file è mono o stereo, si valuta se il buffer è stato letto completamente e si continua con la lettura o si torna allo stato FSM_WAIT_BUFFER_1.

Lo stato FSM_WAIT_I2S serve per aspettare che il modulo I2S abbia terminato la trasmissione dei sample prima di continuare con la lettura. Questo inoltre è lo stato in cui si aspetta nel caso in cui dall'interfaccia utente arrivi il comando di “Pause”

Si procede con l'analisi dei sottomoduli.

Codec Config

Analisi pinout del modulo.

Input:

- **clk, rst_n:** clock e reset sincrono attivo basso

Inout:

- **i2c_sdat:** data line interfaccia I2C

Output:

- **i2c_sclk:** slave clock.

Modulo che istanzia al suo interno il sottomodulo I2C_controller che implementa il physical layer del protocollo I2C, quest'ultimo è stato ripreso dagli esempi forniti da Altera per questo motivo se ne analizza solo il pinout e si rimanda alla documentazione Altera per maggiori informazioni.

Input:

- **CLOCK, RESET.**
- **I2C_DATA:** registro di ingresso che conterrà l'indirizzo del dispositivo slave, l'indirizzo di un registro interno dello slave e i dati da scriverci dentro.
- **GO:** variabile di sblocco della scrittura dei registri interni di un certo dispositivo slave.

Inout:

- **I2C_SDAT:** linea dati.

Output:

- **I2C_SCLK:** linea per il clock del dispositivo slave.
- **END:** variabile che indica, se attiva, il termine della trasmissione
- **ACK:** variabile che indica, se a'0', se la comunicazione è avvenuta correttamente

Il modulo Codec Config è costituito da 3 blocchi always, il primo presenta nella sensitivity list il clock di sistema e genera il clock di per il modulo I2C_controller.

Il secondo, invece, implementa una LUT nel quale sono salvati i dati da scrivere nei registri per l'opportuna configurazione del codec e gli indirizzi degli stessi.

Il terzo e ultimo blocco always è sincrono con I2C controller e semplicemente invia i dati da scrivere e aspetta una risposta positiva dal I2C master per inviare il successivo.

I2S Master

Analisi pinout del modulo.

Input:

- **clk, rst_n**: clock e reset sincrono attivo basso.
- **i2s_sample_data_L_i**: bus dati in ingresso relativo al canale sinistro.
- **i2s_sample_data_R_i**: bus dati in ingresso relativo al canale destro.
- **i2s_send_i**: Segnale che fa evolvere la macchina a stati dalla condizione di riposo.

Output:

- **codec_aud_xck_o**: vedi pinout di Codec.
- **codec_aud_bclk_o**: vedi pinout di Codec.
- **codec_aud_dacdat_o**: vedi pinout di Codec.
- **codec_aud_daclrck_o**: vedi pinout di Codec.
- **i2s_done_o**: flag che segnala il termine della trasmissione dei campioni ricevuti.

Il modulo è costituito da due tre blocchi always.

Il primo, sincrono con il clk, è dedicato alla generazione di codec_aud_xck_o e codec_aud_bclk_o a partire dal clock di sistema. Per farlo utilizza due contatori utilizzati da divisorì in frequenza, ovvero xck_counter e bclk_counter. Il primo divide il clock di un fattore XCK_CNT_TOP mentre l'altro di un ulteriore BCLK_CNT_TOP in modo tale che alla fine codec_aud_bclk_o abbia una frequenza sufficiente affinché sia possibile campionare tutti i bit di due sample del file audio (nel caso sia un file stereo) i quali devono essere inviati con la frequenza più possibile simile a quella di campionamento.

Il secondo blocco always, triggerato nei fronti in discesa di codec_aud_bclk_o invia i bit (MSB First) dei campioni ricevuti in ingresso.

Il terzo blocco always, sensibile ai fronti in salita del clock di sistema, costituisce la funzione di stato del modulo. Oltre che lo stato di reset abbiamo uno stato di riposo FSM_IDLE nel quale si aspetta il segnale i2s_send_i per evolvere nello stato FSM_SEND.

In questo stato si commuta il segnale codec_aud_daclrck_o quando sono stati inviati tutti i bit relativi ad un sample, tale segnale, ricordiamolo, deve presentare una frequenza pari alla frequenza di campionamento del file audio e quando è alto determina l'invio del campione sinistro mentre quando è basso di quello destro.

Quando sono stati inviati sia un campione destro e sinistro, se i2s_send_i è ancora attivo, si continua ad inviare i nuovi campioni ricevuti. Mentre se il segnale i2s_send_i viene disattivato la macchina ritorna nello stato di IDLE

Il segnale i2s_done_o viene attivato quando la macchina è nello stato FSM_IDLE o quando sono stati inviati sia il campione destro e sinistro.

Main PLL

Modulo ottenuto dalle librerie Altera, si faccia dunque riferimento all'apposita documentazione per maggiori informazioni.

Il PLL è usato per ottenere un clock di sistema con frequenza di 100MHz a partire da un clock di 50MHz.

Codec audio WM8731

Il codec audio WM8731 si occupa della conversione A/D e D/A e dell'amplificazione del segnale. Lo schema a blocchi è mostrato in figura:

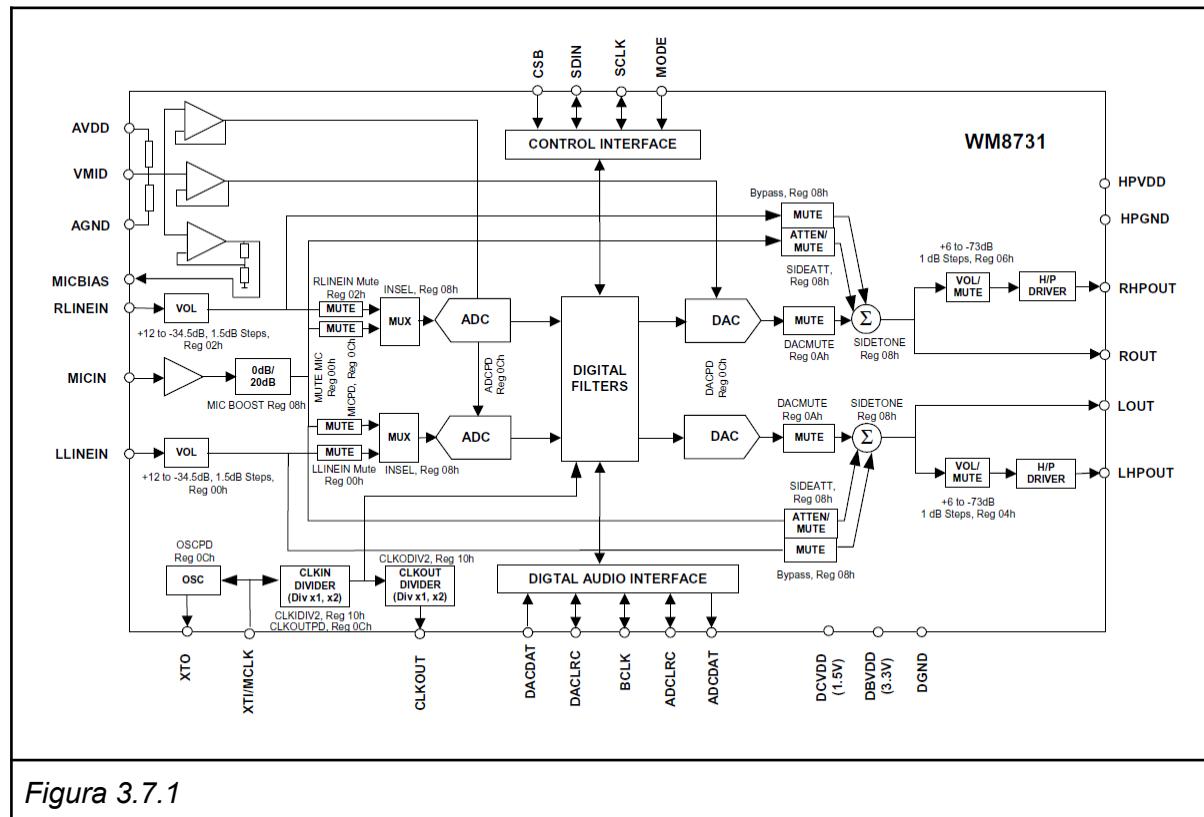


Figura 3.7.1

Caratteristiche:

- Stereo Audio Codec with Integrated Headphone Driver.
- Stereo ADC di tipo - con SNR pari a 90 dB.
- Frequenza di campionamento impostabile da 8 kHz a 96 kHz.
- Master or slave clocking mode.
- Word Length impostabile da 8 bit a 32 bit.
- Programmabile tramite interfaccia I2C o SPI. La scelta tra le due è determinata dallo stato logico di un pin esterno.
- Programmable Audio Data Interface: I2S, Left/Right Justified o DSP. Output Volume and Mute Controls.

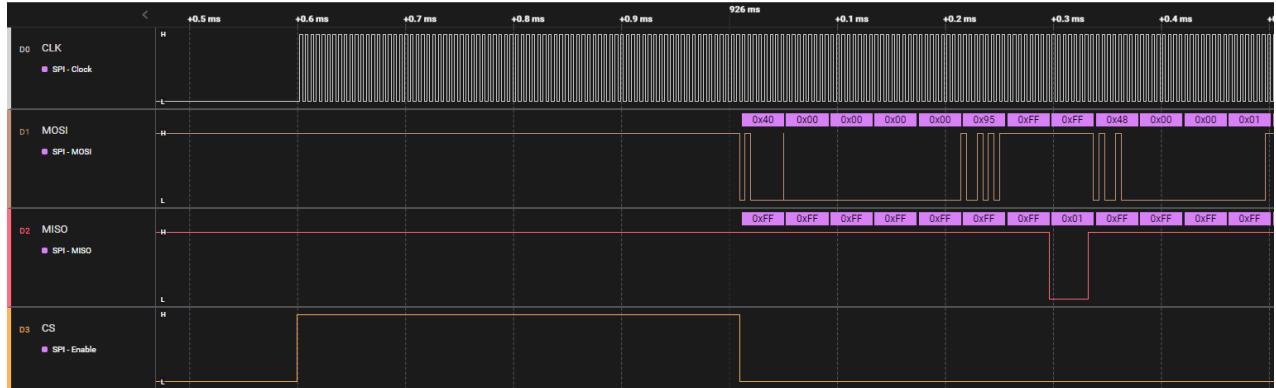
Datasheet: <http://cdn.sparkfun.com/datasheets/Dev/Arduino/Shields/WolfsonWM8731.pdf>.

Debug e funzionamento

In questa sezione vengono mostrate waveforms acquisite con l'analizzatore logico, per mostrare il funzionamento *live* - ossia non simulato - del sistema.

Protocollo SPI scheda SD

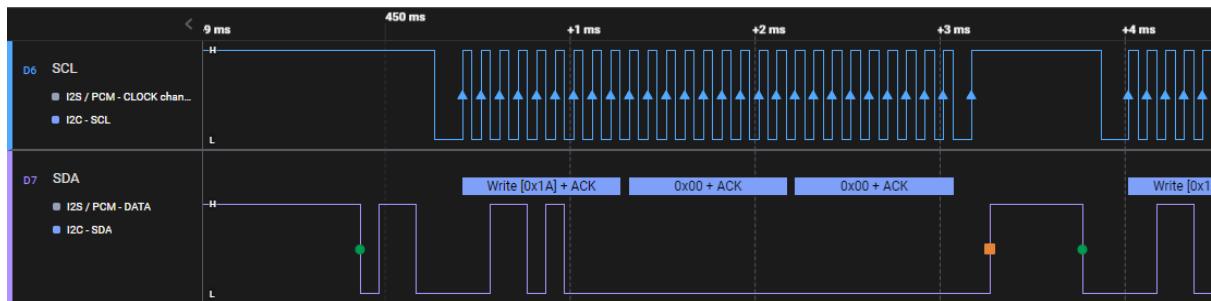
Dopo il reset, avvenuto a +0.6 ms, si osserva il chip select andare alto per 80 cicli di clock. Questa procedura segnala alla scheda SD che deve uscire dal suo stato di power down (in cui riduce praticamente a zero i consumi) e prepararsi a comunicare.



Viene poi mandato il primo comando, ossia il CMD0, a cui la scheda risponde con una parola di 8 bit “0x01”. L'ultimo bit segnala che la scheda non è ancora pronta a leggere/scrivere i dati al suo interno e necessita prima di terminare la sua configurazione.

Protocollo I2C codec

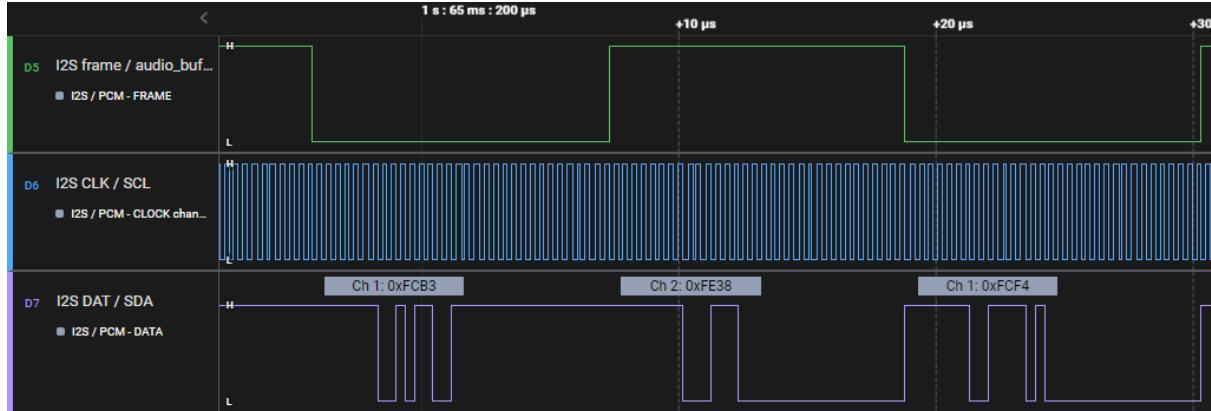
Dopo il reset vengono anche configurati i registri del codec attraverso il protocollo I2C.



In figura, sono evidenziati con pallini verdi i bit di *start* e in rosso i bit di *stop* del protocollo I2C. È mostrata la scrittura del registro 0x0: viene prima mandato l'indirizzo del codec e poi due bytes, contenenti i 7 bit di indirizzo del registro e i 9 bit di dati del registro; in questo caso entrambi zero.

Protocollo I2S codec

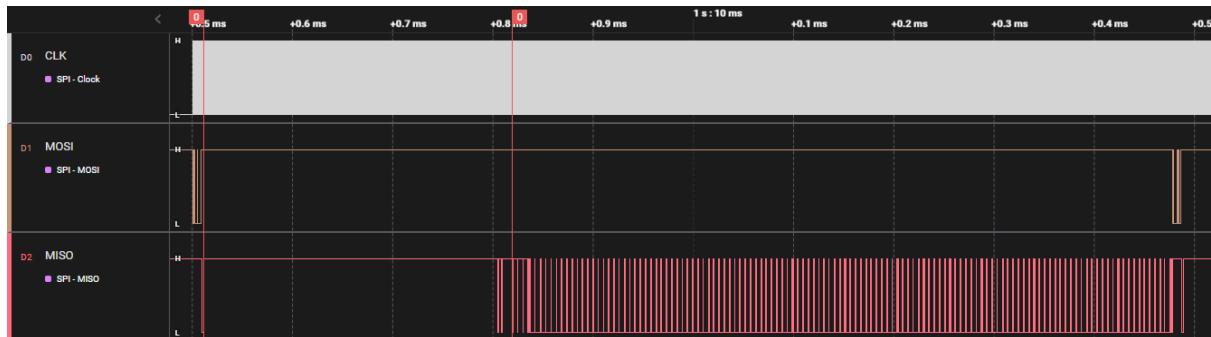
Il protocollo I2S si compone di tre segnali, mostrati in figura.



I campioni, in formato pcm16s, sono rappresentati come interi con segno in complemento a due. Possiamo quindi concludere che i dati in figura, che iniziano con '0xF', sono numeri negativi.

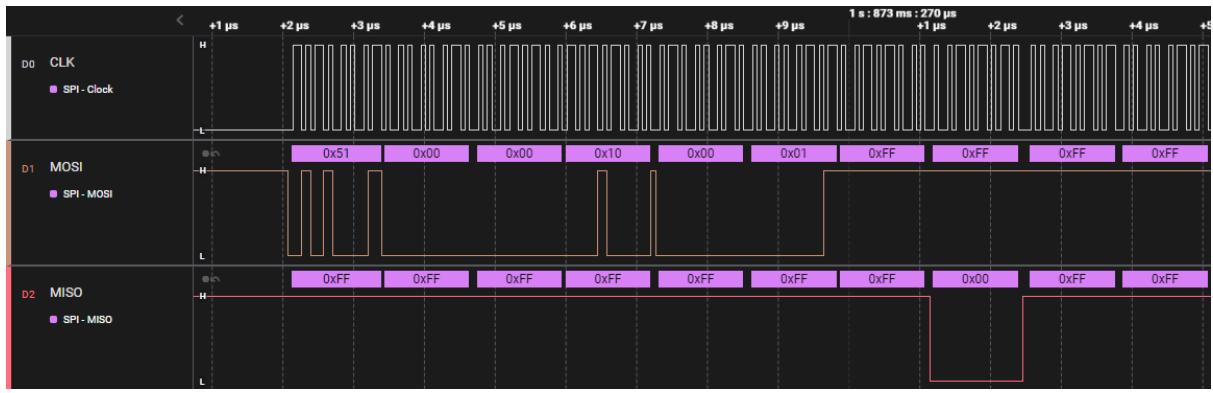
Lettura di un blocco della scheda SD

La lettura di un blocco inizia mandando il comando di lettura, contenente l'indirizzo del blocco da 512 bytes da leggere. Si aspetta poi che i dati siano pronti. Con questa scheda SD il tempo di attesa è di circa 308 us, e viene misurato con la coppia di marker rossi in figura.



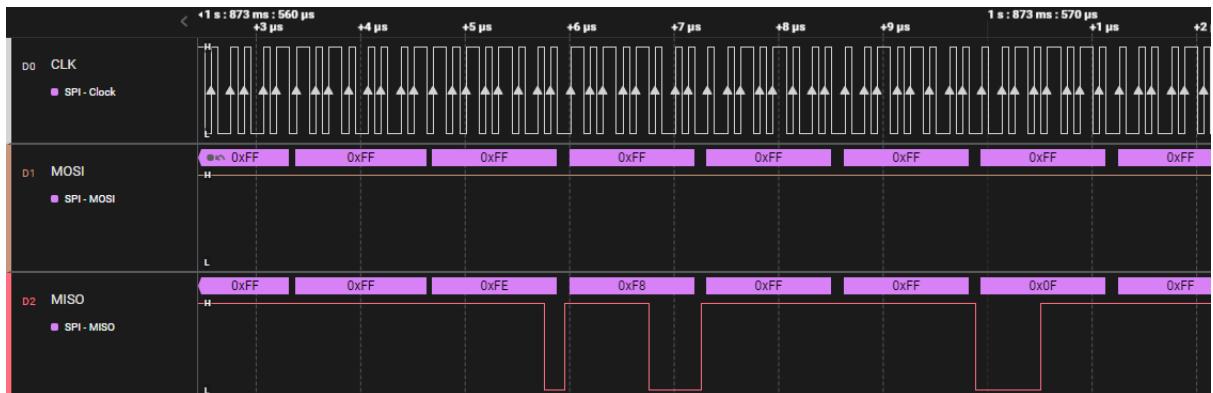
Quando i dati sono pronti, vengono trasmessi i 512 bytes del blocco uno dopo l'altro. Al termine del blocco in figura si vede che viene immediatamente mandato il comando di lettura successivo.

Mostriamo ora un zoom sul comando di lettura:



Il comando si compone di 6 bytes. Il primo byte è il numero del comando. I bytes 2-5 contengono invece il blocco da leggere: in figura viene richiesto il blocco '0x00001000'.

I dati sono pronti quando la scheda risponde '0xFE'



Il byte successivo, ossia '0xF8', è il primo byte valido del blocco da 512 bytes richiesto.

Lettura di un cluster FAT32

Per la lettura di un cluster bisogna fare alcune operazioni:

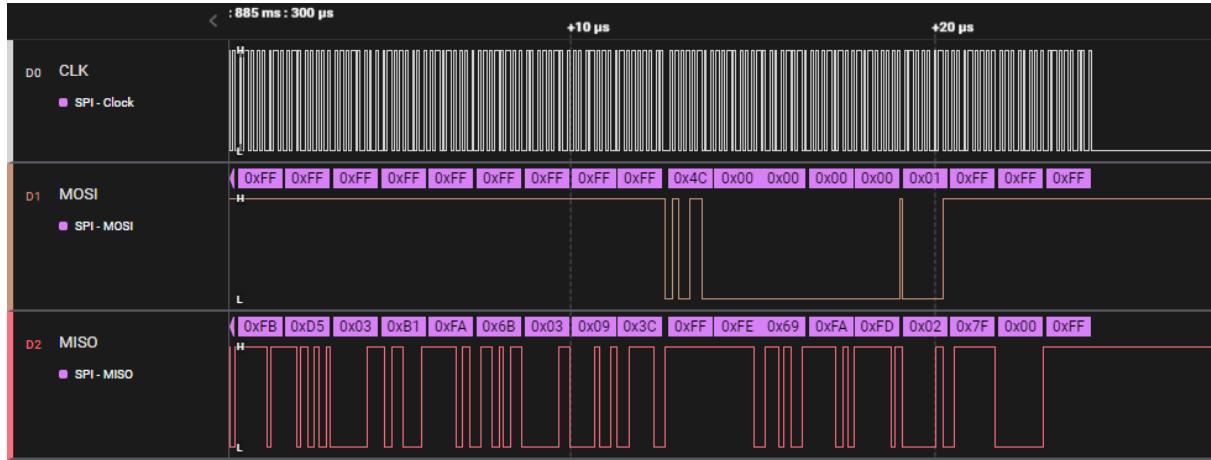
1. leggere la FAT per scoprire in quale settore inizia il cluster successivo
 2. legger tutto il cluster mandando il comando 'continuous block read'
 3. dopo aver letto 16 blocchi, ossia la dimensione di un cluster, mandare il comando di 'stop continuous block read'



Si notano due intervalli in cui il MISO rimane in *idle* sul livello logico alto: sono gli intervalli in cui si aspetta che siano pronti rispettivamente il blocco della fat e il primo blocco del cluster.

La lettura sequenziale offre il vantaggio di essere più rapida: non c'è da aspettare il tempo di fetch del blocco poiché la scheda SD lavora con la pipeline piena.

Mostriamo ora l'invio del comando di stop:

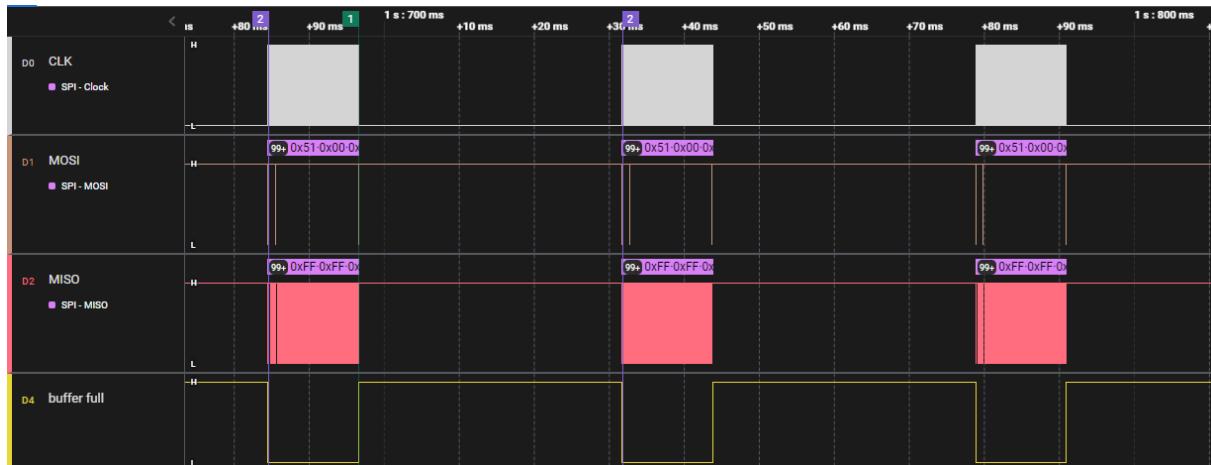


Notare che la scheda SD ha già iniziato a mandare il blocco successivo, che inizia col byte '0x69', poiché '0xFF' e '0xFE' sono rispettivamente il byte di idle e di start data block.

Dopo aver ricevuto il comando di stop, la scheda risponde con '0x00' e cessa di inviare dati.

Funzionamento a regime

In figura sono mostrate le ripetute letture della scheda SD, per riempire il buffer coi campioni audio da mandare al codec.



La frequenza con cui si ripetono le letture coincide dipende dal tempo impiegato dal codec per riprodurre tutti i campioni dentro il buffer.

La traccia gialla mostra il segnale 'buffer full': esso viene messo a 1 quando il buffer viene riempito leggendo i dati dalla scheda SD. Scende poi a 0 quando il codec segnala di avere bisogno di un altro buffer pieno; inizia così il processo di riempimento del buffer successivo. Ricordiamo che viene impiegato il double buffering: in ogni istante c'è un buffer che viene letto e uno che viene riempito.

Conclusioni

Il sistema illustrato implementa un lettore walkman di file WAV funzionante, ciò nonostante sarebbe comunque possibile apportare delle migliorie, come ad esempio un'interfaccia utente più completa, sfruttando anche il display LCD di cui è fornita la scheda di sviluppo per mostrare il nome della canzone in riproduzione.

È stato molto interessante il debug con l'analizzatore logico: sfruttare i GPIO per portare 'nel mondo esterno' i segnali che normalmente sono sepolti dentro il chip offre possibilità di debug estremamente flessibili e immediate, che non sono assolutamente disponibili quando si lavora invece con i microcontrollori, a cui eravamo abituati.

In conclusione, il progetto si è rivelato molto interessante e gratificante nella sua realizzazione.