# POLITECNICO DI TORINO

Dipartimento di Elettronica e Telecomunicazioni

Laurea Magistrale in Ingegneria Elettronica

---



## Tesi di Laurea Magistrale

# MFIO HV module: analisi della scheda e realizzazione firmware per il test automatizzato di segnali elettrici avionici

**Relatore**

Prof. Maurizio Martina

**Correlatore**

Ing. Antonio Grasso (Leonardo Company)

**Candidato**

Marta Di Matteo

Aprile, 2021

# Elenco delle tabelle

# Elenco delle figure

# Indice

# CAPITOLO 1

# Introduzione

La tesi proposta è stata realizzata in collaborazione con **Leonardo S.p.A. Divisione Velivoli** (**Leonardo Velivoli**), principale azienda italiana operante nei settori di aerospazio, difesa e sicurezza, sia in ambito civile che militare, con l'obiettivo di rafforzare la sicurezza globale, la protezione delle persone, i territori, le infrastrutture e la sicurezza informatica.
Oggetto del lavoro di tesi è lo sviluppo di un firmware in VHDL e C da installare su una scheda **MFIO (Multi Function Input Output) HV (High Voltage)**, ovvero una tipologia di schede che vengono utilizzate, tra le altre applicazioni, all'interno di apposite apparecchiature denominate **AGE (Aerospace Ground Equipment)**, prodotte da **Leonardo S.p.A.** al fine di supportare lo sviluppo e la manutenzione dei velivoli. Il firmware sviluppato ha seguito le best practice ed ha avuto come baseline di partenza un firmware già sviluppato in passato da **Leonardo S.p.A.** per uso in ambito laboratorio. Le suddette schede, devono soddisfare un'elevata affidabilità nell'utilizzo e una altrettanto elevata manutenibilità, sia dell'hardware sia del firmware in esse installato. Le schede MFIO HV hanno lo scopo di stimolare e misurare segnali in DC e AC, in particolare 28V in continua e 115 V a 400Hz in alternata.
Considerato che schede come la MFIO HV possono essere parte della baseline di diversi sistemi di test automatico in ambito avionico, un'errata segnalazione di avaria può ingenerare un impatto significativo sul "TAT" (Turn Around Time) e una ricaduta sulla "availability" del sistema di test e del sistema under test, sia ad un potenziale Cliente sia a **Leonardo S.p.A**.

Infatti, nell'industria aeronautica sono sempre più rilevanti i fattori di "availability", "maintainability" e "TAT". Un prodotto manutenibile e affidabile genera un virtuoso processo di qualità e di economia di attività che sono la frontiera tecnica, sempre in movimento, alla ricerca del sistema di "domani".
Il mercato sempre più "mondiale" di velivoli e sistemi di test per la loro manutenzione, ordinaria e straordinaria, spingono dunque un'azienda come **Leonardo S.p.A.** a non fermare mai la ricerca e lo sviluppo di ogni aspetto e dettaglio, hardware, software, firmware, che possano incidere positivamente sui parametri per creare prodotti migliori, sempre più appetibili dal mercato e al passo con i tempi della tecnologia che negli ultimi lustri ha subito accelerazioni che portano a prevedere per i prossimi anni ulteriori e ad oggi impensabili sviluppi.

## 1.1 Obiettivi

Gli obiettivi della presente tesi sono:

- Analizzare la scheda MFIO HV messa a disposizione da **Leonardo Velivoli**
- Analizzare in modo critico il firmware installato sulla scheda
- Progettare e sviluppare un nuovo firmware da poter installare sulla scheda che sia in grado di misurare e stimolare segnali in DC

Il progetto della scheda e il firmware erano stati realizzati circa un decennio addietro. Si è dunque resa necessaria un'analisi dettagliata dei codici VHDL e C per comprenderne il funzionamento.
A conclusione della fase di analisi della scheda, si è convenuto di sviluppare una nuova struttura hardware in VHDL e un nuovo firmware in C, rendendoli più semplici, essenziali e manutenibili (in previsione di utilizzi futuri), in grado di supportare la lettura e la generazione di segnali elettrici in continua dei velivoli.
I dispositivi programmabili presenti sulla scheda sono due:

- **CPLD (Complex Programmable Logic Device**)
- **FPGA (Field Programmable Gate Array)**

Nello sviluppo posto in essere, si è mantenuto inalterato il firmware del CPLD, mentre si è realizzato un nuovo firmware per l'FPGA.
A conclusione del processo di progetto hardware e sviluppo del firmware, si sono verificate le prestazioni del sistema prima mediante simulazioni utilizzando il software **Modelsim** e, successivamente, utilizzando un banco di test rappresentativo messo a disposizione da **Leonardo Velivoli** presso i propri laboratori, comprendente l'hardware necessario a riprodurre in laboratorio le interfacce di un AGE generico (rack di alimentazione, scheda CPU, scheda target MFIO HV).

## 1.2 Organizzazione dei capitoli

La tesi è organizzata in 9 capitoli, di cui il primo (*Capitolo 1*) costituisce l'introduzione, mentre i restanti 8 sono dedicati alla descrizione del lavoro svolto. Le informazioni esposte saranno organizzate nel seguente modo:

- *Capitolo 2, Leonardo S.p.A.*: Si forniscono cenni sulla descrizione generale dell'azienda Leonardo S.p.A. e sull'attività svolta dall'unità aziendale Avionic test and Support Systems, con la cui collaborazione è stata sviluppata la tesi in esame. Si descrive, inoltre, l'AGE generico in cui sono state utilizzate le schede MFIO HV, oggetto del lavoro di tesi.

- *Capitolo 3, Analisi della scheda*: Contiene la descrizione del modulo Multi Function Input Output High Voltage.

- *Capitolo 4, Struttura firmware originale*: Si occupa di descrivere in modo generale la struttura del firmware originale, sia dell'FPGA che del CPLD.

- *Capitolo 5, Analisi dei comandi e simulazioni CPLD*: descrive in dettaglio i comandi che il CPLD può ricevere e mostra le simulazioni effettuate per ogni blocco interno al CPLD, che dimostrano un comportamento conforme a quello atteso.

- *Capitolo 6, Interfaccia Avalon e Standard ANSI/VITA 4-1995*: E' dedicato alla descrizione dello standard **ANSI/VITA 4-1995** e dell'interfaccia "**Avalon**", definita da **Intel/ Altera** per connettere in modo semplice i diversi componenti all'interno delle FPGA Intel/Altera.

- *Capitolo 7, Struttura firmware finale*: Descrive in modo dettagliato il firmware realizzato: inizialmente illustra il flusso di progetto adottato e successivamente analizza sia la struttura dell'hardware che del firmware.

- *Capitolo 8, Prove sperimentali:* E' suddiviso in tre sezioni:

    1. La prima è dedicata alla descrizione della strumentazione utilizzata per testare il firmware sulla scheda MFIO HV

2. La seconda descrive le prove sperimentali e i risultati ottenuti per la verifica del corretto funzionamento della parte di inizializzazione
3. La terza descrive le prove sperimentali realizzate per effettuare il "testing" dell'"instructions scheduler".
4. *Capitolo 9, Conclusioni*: E' dedicato all'analisi critica dei risultati ottenuti e agli sviluppi futuri del progetto.

# CAPITOLO 2

# Leonardo S.p.A.

Come accennato nel paragrafo "Introduzione", **Leonardo S.p.A.** è Azienda leader in Italia e una delle maggiori al mondo operante nei settori di Aerospazio, Difesa e Sicurezza, sia in ambito civile che militare. Questa posizione di prestigio suffragata da oltre 70 anni di storia, fa di **Leonardo S.p.A.** un partner di fiducia sia per aziende private che per governi e istituzioni, italiane ed internazionali.
L'azienda è attualmente organizzata in cinque divisioni: Elicotteri, Velivoli, Aerostrutture, Elettronica e Cyber Security, ognuna delle quali è a sua volta suddivisa in "Funzioni" e "Discipline" sia in ambito tecnico che in ambiti organizzativi, commerciali, legali, qualità, sicurezza, ecc. Nello specifico, il presente lavoro di tesi è stato supportato dal team di sviluppo *AGE complessi*, inquadrato nella disciplina aziendale **Avionic Test & Support System**, appartenente all'unità aziendale **ASYS (Avionic System)**, a sua volta parte dell'"Ingegneria", della divisione "Velivoli".
Alcuni dei principali progetti in cui AT&SS ed in particolare il team di sviluppo *AGE complessi* è coinvolto, riguardano a titolo di esempio i velivoli "Eurofighter Typhoon" (EFA) e "M346", rispettivamente un velivolo da combattimento (EFA) e da addestramento (M346). Per alcuni dei sistemi avionici dei sopracitati velivoli, *Avionic Test & Support System* ha il compito, tra gli altri, dello sviluppo, della qualifica e della manutenzione post-vendita di "AGE" (Aerospace Ground Equipments), progettando e realizzando sia l'elettronica che il software di bordo (nei vari layers che un sistema complesso come un AGE presenta).
Gli AGE risultano di particolare interesse ai fini della presente tesi: si tratta di

complesse macchine di test utilizzate per verificare il funzionamento (manutenzione programmata o manutenzione straordinaria) di una specifica sottoparte del velivolo (sotto sistema avionico), fornendo come risultato del test PASS o FAIL. Dal momento che un velivolo è un sistema complesso, per ogni velivolo sono necessari decine (spesso centinaia) di AGE, ognuno dei quali progettato e realizzato per verificare il funzionamento di una diversa parte del velivolo o anche per poter accedere a parti di velivolo per esigenze operative e manutentive.
Le schede MFIO HV oggetto di questa tesi, sono state installate in uno specifico AGE per un velivolo di addestramento.
Questo tipo di AGE ha l'obiettivo di testare e verificare il corretto funzionamento di uno specifico sottosistema avionico del quale, tra gli altri requisiti funzionali da soddisfare vi sono:

- acquisizione e controllo di segnali 28V DC
- acquisizione e controllo di segnali 115V AC a 400 Hz

La risposta a tali requisiti, porta alla necessità di installare nel sistema AGE le schede MFIO HV che devono avere a bordo un firmware in grado di acquisire e/o stimolare segnali in continua e in alternata (*ai fini della tesi si è realizzato il firmware per la gestione di segnali elettrici in continua*).
In avionica, la 28V DC ha la funzione di fornire alimentazione a tutti gli apparati avionici di bordo come ad esempio computer di bordo, radar, pannelli comandi ecc. La tensione 115V AC @400Hz, invece, viene utilizzata per alimentare una parte degli apparati avionici o come sorgente per generare altre tensioni ausiliarie.
Tra i requisiti che un AGE deve poter soddisfare, risulta importante il range di temperatura in cui l'AGE deve essere in grado di lavorare correttamente, dal momento che un velivolo e tutti gli AGE utilizzati per supportare la verifica di idoneità al volo (fitness for purpose/ready for flight) vengono venduti da **Leonardo S.p.A.** a clienti in diverse parti del mondo, da climi secchi e torridi a climi umidi e glaciali. Un medesimo modello di AGE deve operare indistintamente in condizioni ambientali così diverse, senza limitazioni o modifiche nelle prestazioni.
Pertanto, l'AGE e tutti i componenti sono sottoposti a un processo di qualifica di Part Number di tipo ambientale. Fra le caratteristiche ambientali principali è possibi-

le citare il range di temperatura operativa molto ampio -20°C ÷ +55°C, la resistenza alla pioggia, alla sabbia e alla salsedine e la compatibilità elettromagnetica (con variazioni legate al velivolo oggetto di test).
Prima di essere venduti, gli AGE vengono sottoposti alla procedura denominata "Acceptance Test Procedure" (ATP), che mira a verificare che i requisiti imposti per l'AGE siano soddisfatti. Al fine di garantire maggiore oggettività nella valutazione, tale procedura coinvolge diversi gruppi aziendali appartenenti alla divisione "Velivoli", ovvero il gruppo "Ingegneria", "Qualità", "Qualifiche Ambientali", "Qualifiche EMC" e "Customer Support". Se l'ATP fornisce come risultato "PASS", l'AGE viene dichiarato adatto a collaudare il velivolo.

## 2.1 AGE Generico per Addestratori

In questo paragrafo si fornisce una descrizione di un AGE generico per velivolo addestratore.
Esso è composto da un sistema computerizzato dotato di software (Software real time con sistema operativo real time) in grado di eseguire in tempo reale acquisizione dati, verifica e stimolazione di dati, emulazione e simulazione di componenti fisici per effettuare test rappresentativi propedeutici alle missioni di volo. Le componenti hardware di I/O (segnali analogici, discreti, bus avionici, seriali, segnali RF, segnali video ecc) utilizzate sono anch'esse controllate in tempo reale.
L'AGE si interfaccia con l'avionica del velivolo ed è composto da schede elettroniche presenti in commercio ("COTS", Commercial OFF The Shelf) e da schede progettate ad hoc (Custom Boards).
Un AGE come quello descritto è composto da 5 parti principali, illustrate in figura 2.1:

- **Human Machine Interface (HMI)**
- **Main Processing Unit (MPU)**
- **Front Accessory Box**
- **Central Accessory Box**
- **Trolley**

In particolare, il trolley a quattro ruote viene utilizzato per trainare l'AGE ed è progettato per sostenere i principali componenti hardware, oltre ai cavi/loom di interfaccia elettro-meccanica tra AGE e velivolo e ai cassetti per gli accessori. Infatti, tipicamente l'AGE deve poter essere trainato a bassa velocità (massimo 25 Km/h) lungo superfici asfaltate proprie di Hangar e piste di volo da un opportuno mezzo trainante. Inoltre, come si nota dalla figura 2.1, l'AGE è dotato di due Accessory Box, uno frontale e uno centrale, utilizzati per conservare cavi/loom e accessori.
La **Main Processing Unit (MPU)** è la parte principale del sistema, composta dal sistema di test, unità di alimentazione (Power Distribution Unit, "PDU") e cavi di

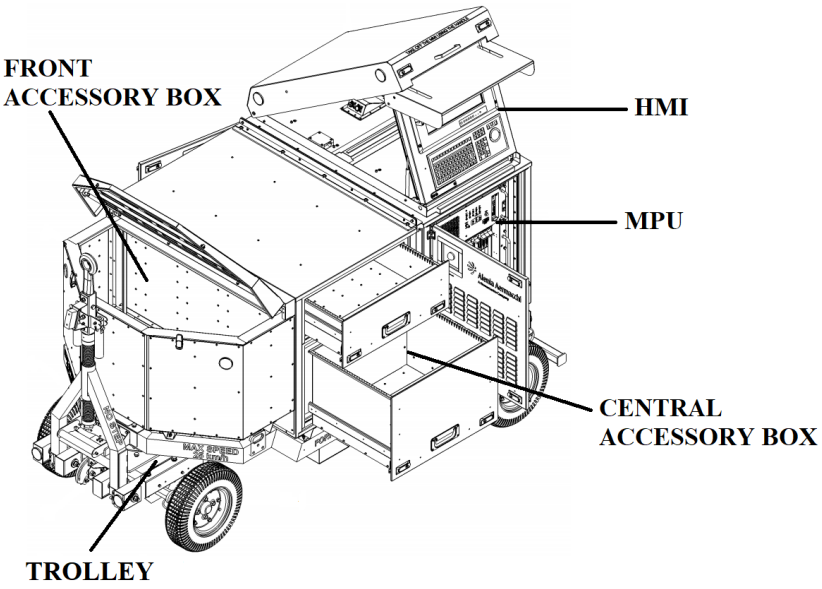


Figura 2.1: Overview di un AGE generico

interconnessione interna. Essa può essere utilizzata per acquisire/stimolare segnali di vario tipo oppure come Bus Controller, Bus Monitor e Remote Terminal con bus avionici standard (come il bus 1553), simulando ed emulando pacchetti di dati (sia segnali analogici che discreti) per realizzare il test del sistema avionico.
In particolare, il sistema di test è a sua volta composto da una "cassa", una connessione ethernet e un sistema di ventilazione. La "cassa" è un crate metallico contenente tutte le schede di elaborazione e gestione dei segnali del TEST SET. Esso può contenere fino a 20 schede elettroniche in altrettanti slots VME (Versa Module Eurocard)[1]. Un esempio del crate VME è illustrato in figura 2.2.
Esso riceve da una delle Power Supply Units le tensioni utili ai componenti interni ( che vengono generate a partire dalla Power Distribution Unit (PDU)) e contiene anche un sistema di ventilazione montato nella parte inferiore, mediante il quale è possibile ottenere un raffreddamento dei PCBs che, come detto precedentemente, possono essere costretti ad operare in scenari particolarmente sfidanti dal punto di vista ambientale.
In particolare, sono presenti due unità di alimentazione: quella principale fornisce alimentazione al rack VME mentre l'unità di alimentazione di supporto fornisce ali-

[1]Riferimento all'appendice B

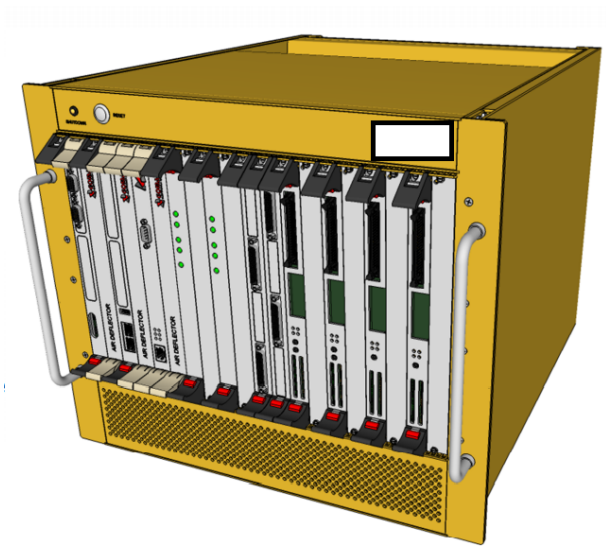

Figura 2.2: Esempio di crate AGE

mentazione al sistema di ventilazione, alle USB e alla comunicazione ethernet.
Un esempio di uno dei moduli dell'unità di alimentazione è mostrato in figura 2.3: questa è in grado, nel caso di esempio in fase di descrizione, di fornire 8 diverse alimentazioni corrispondenti agli 8 slot:

- $\pm 60V$, rispettivamente negli slot 1 e 2;
- 5V nello slot 3;
- 3.3V nello slot 4;
- $\pm 12V$, rispettivamente negli slot 5 e 6;
- 24V nello slot 7;
- 28V nello slot 8.

Infine, con riferimento alla figura 2.1, la **Human Machine Interface (HMI)** rappresenta l'interfaccia con l'utente. E' composta da un monitor, da una tastiera e da un trackball, che servono per gestire le attività di test attraverso le istruzioni che il software dell'AGE deve fornire (e ricevere) dall'operatore.
Tra i requisiti che l'AGE deve soddisfare, figurano anche quelli relativi alle condizioni climatiche. E' infatti richiesto che l'AGE funzioni correttamente al variare delle

Figura 2.3: Unità di alimentazione

condizioni ambientali, dal momento che deve poter essere utilizzato, come già accennato, in qualsiasi parte del mondo a seconda della locazione del cliente, nel range di temperatura operativo -20°C ÷ +55°C.
Infine, deve essere resistente alla muffa e, per taluni AGE, deve garantire le prestazioni anche in presenza di pioggia, sabbia e polvere.

# CAPITOLO 3

# Analisi della scheda

In questo capitolo si analizzerà nel dettaglio la struttura della scheda utilizzata per supportare l'acquisizione e verifica di segnali elettrici in continua (28V DC) di un velivolo.
Come accennato nel capitolo "Introduzione", si tratta di una scheda di I/O che fa parte di una famiglia di schede che possono realizzare quattro diversi moduli:

1. MFIO (Multi Function Input Output) High Voltage
2. MFIO Low Voltage
3. MFIO Resistant Sensor
4. PWM Pulse Width Modulator

*Ai fini del lavoro di tesi, l'unico modulo di interesse è quello "High Voltage" e pertanto si utilizzerà solo la scheda che implementa questo specifico modulo.*

## 3.1 MFIO High Voltage Module

Il Multi Function Input Output High Voltage Module è un modulo **IP**[8] compatibile, composto da una scheda principale denominata "**Baseboard**" e una secondaria denominata "**Piggy Board**", montate una sull'altra. In figura 3.1 è mostrata la parte superiore della scheda, ovvero la Baseboard.

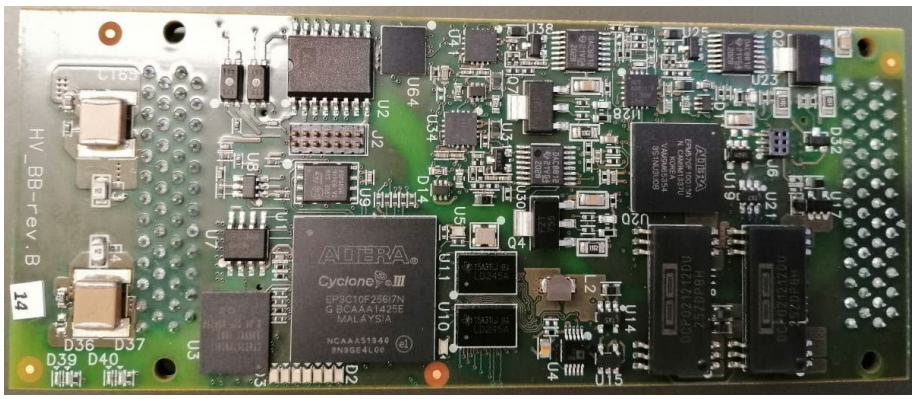


Figura 3.1: MFIO High Voltage Module: Baseboard

Tale modulo è in grado di stimolare e/o acquisire tensioni fino a $\pm 50V$ in DC e in AC fino 20kHz, sui sei canali disponibili che possono essere usati contemporaneamente in modo indipendente. Per ogni canale è presente sia un ADC (Analogue to Digital Converter) che un DAC(Digital to Analogue Converter), al fine di supportare rispettivamente la lettura e la scrittura di tensioni. La modalità di lettura è denominata "monitor mode", mentre la modalità di scrittura è denominata "simulation mode". Nello specifico, i dispositivi di conversione A/D e D/A e i rispettivi circuiti di condizionamento dei canali 1, 2 e 3 sono integrati sulla baseboard, mentre quelli relativi ai canali 4,5 e 6 sono integrati sulla Piggy Board.
Inoltre, il modulo è alimentato da una scheda "IP carrier"[8] a 5V e 12V e riceve dalla PSU anche la $\pm 60V$ per l'interfaccia con l'esterno. L'IP carrier si occupa di trasformare l'interfaccia elettrica del modulo (segnali IP) in un'interfaccia VME compatibile. Infatti, le schede in questione vengono inserite all'interno di un crate di gestione e alimentazione che contiene un "cestello" in grado di ospitare fino a venti schede, il quale è uno standard avionico denominato VME[1]. Nonostante il VME sia un bus di comunicazione diventato uno standard **ANSI (American National Standard Institute)/IEEE (institute of Electrical and Electronics Engineers) 1014** nel 1987, è ancora utilizzato in quanto alcuni apparati lavorano ancora con questo tipo di bus di comunicazione. Pertanto, sebbene le schede attuali non usino questo tipo di logica di controllo e siano più "performanti", è necessario mantenere la stessa struttura in modo da garantire la retro-compatibilità con i vecchi sistemi (uno dei

[1]Per approfondimenti si rimanda all'appendice B

criteri basilari dei sistemi di test avionici è proprio la retro-compatibilità o "non regression").

Inoltre la scheda contiene due connettori, uno per i segnali IP e l'altro per i segnali di interfaccia ed è anche dotata di due dispositivi programmabili entrambi selezionati tra quelli forniti da Intel/Altera: un **CPLD (Complex Programmable Logic Device)** [3] e una **FPGA (Field Programmable Gate Array)** [1], mediante la quale è possibile programmare il modulo programmando gli appositi registri messi a disposizione e descritti nell' "Interface Control Data"[2].

Da tale documento, si evince che lo spazio dei registri di identificazione del modulo è comune a tutti e quattro i moduli sopraelencati, in quanto contiene informazioni che non variano a seconda del modulo scelto (ad esempio il codice del produttore), mentre i registri di configurazione per ciascun canale (controllo, dati e registri di stato), sono specifici per ogni modulo. In tal caso si è fatto riferimeneto allo spazio dei registri dell' HV module.

[2]Si tratta di uno dei documenti che costituiscono il cosiddetto "data package" della scheda, proprietà di Leonardo S.p.A. I registri utilizzati ai fini della tesi sono dettagliati nell'appendice A

# CAPITOLO 4

# Struttura del FW originale su FPGA

Come previsto dagli obiettivi, parte di questo lavoro di tesi, consiste nell'auto acquisire la conoscenza sul funzionamento della scheda a partire dai codici VHDL e C originali.
La scheda oggetto dell'analisi contiene due dispositivi programmabili: un CPLD della della famiglia *MAXII Altera* e una FPGA della famiglia *Cyclone III Altera*.
Di seguito si descriverà la struttura dell'FPGA derivata dall'analisi e, a seguire, quella del CPLD.

## 4.1 Struttura FPGA

In figura 4.1 si riporta lo schema a blocchi della struttura originale dell'FPGA.
Nello specifico, i macro blocchi presenti sono cinque:

1. **IP_BLOCK**
2. **EX_SPI**
3. **AV_FIFO_DEC**
4. **HV_MODULE_TOP**
5. **NIOS WRAPPER**

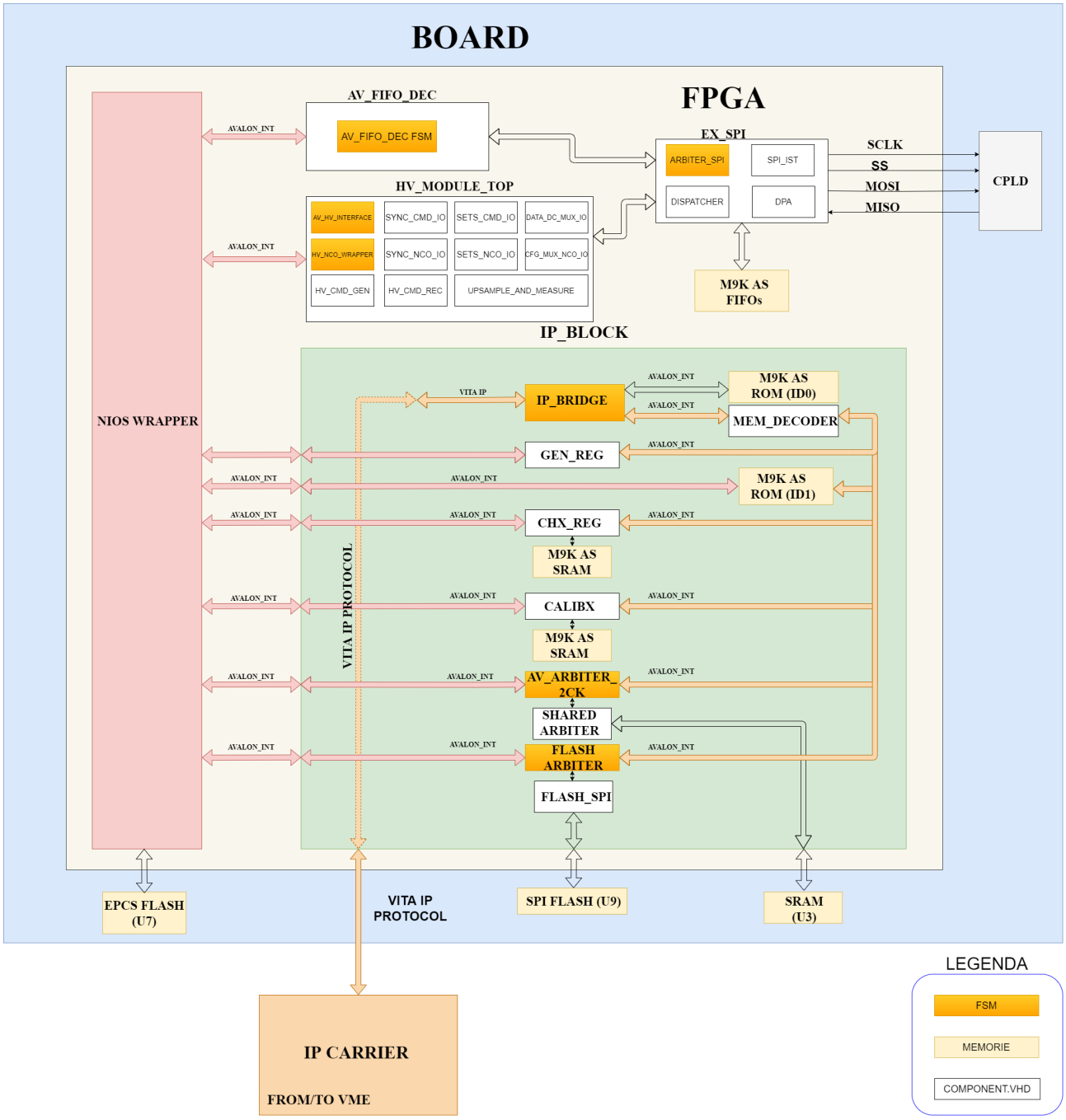


Figura 4.1: Struttura originale dell'FPGA

### 4.1.1 IP_BLOCK

Il blocco denominato **IP_BLOCK** ha lo scopo di interfaccia tra il processore NIOS e le periferiche con le quali comunica. Infatti, scambia con NIOS i segnali dell'interfaccia Avalon (la quale sarà oggetto di un paragrafo dedicato nel capitolo 6) per tutte le periferiche con cui NIOS scambia dati; in aggiunta, riceve da NIOS il contenuto dei registri di stato che entrano nel blocco "GEN_REG" (in questo caso l'interfaccia non è di tipo Avalon) e i segnali di cancellazione di una SPI Flash integrata sulla scheda. Inoltre, **IP_BLOCK** riceve e fornisce segnali da/verso l'esterno. Nello specifico, essi costituiscono i segnali di interfaccia con una SRAM presente sulla scheda e i segnali dello standard ANSI/VITA 4-1995 (il quale è oggetto di un paragrafo dedicato nel capitolo 6).
Le periferiche contenute nel blocco "IP_BLOCK" sono accessibili sia dal processore NIOS, sia dall'esterno tramite "IP_BRIDGE" e "MEM_DECODER".
All'interno di queste periferiche è contenuta una memoria che viene utilizzata per lo scambio di informazioni tra il processore NIOS e il mondo esterno.
Il blocco "MEM_DECODER" riceve tramite il blocco "IP_BRIDGE" i segnali dello standard ANSI/VITA 4-1995 che arrivano dall'esterno e si occupa di leggere o scrivere dati da/in ciascuna periferica mediante interfaccia Avalon. In particolare, "MEM_DECODER" decodifica la periferica con cui scambiare dati mediante il controllo di 6 bit all'interno dell'address a 22 bit che arriva da "IP_BRIDGE", che è costituito dalla concatenazione dell'ip_address a 6 bit e dell'ip_data a 16 bit (questi segnali fanno parte dei segnali IP e verranno analizzati in dettaglio nel capitolo 6).
La figura 4.2 rappresenta l'address del blocco "MEM_DECODER", con i bit di interesse evidenziati in arancione, mentre si riporta di seguito in tabella 4.1 la decodifica operata da "MEM_DECODER" per le diverse periferiche.

| MEM_DECODER AVALON ADDRESS | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| IP_DATA | | | | | | | | | | | | | | | IP_ADDRESS | | | | | | | |
| b21 | b20 | b19 | b18 | b17 | b16 | b15 | b14 | b13 | b12 | b11 | b10 | b9 | b8 | b7 | b6 | b5 | b4 | b3 | b2 | b1 | b0 |

Figura 4.2: MEM_DECODER Avalon Address

Infine, con riferimento alla figura 4.1, i blocchetti in giallo rappresentano tutte le memorie utilizzate. In particolare, quelle denominate U3, U7 e U9 sono memorie esterne all'FPGA e integrate su scheda mentre quelle denominate M9K sono memorie interne

| **b20 b19** | **b10 b9 b8 b7** | **Periferica** |
|---|---|---|
| 00 | 0000 | ROM ID1 |
| 00 | 0001 | GEN_REG |
| 00 | 0010 | CHX_REG |
| 00 | else | CALIBX |
| 01 | - | FLASH_ARBITER |
| 10 | - | AV_ARBITER_2CK |

Tabella 4.1: Decodifica periferiche firmware originale

all'FPGA stessa. Le memorie M9K[2] sono array di memorie inglobate nelle FPGA Altera e la loro densità varia a seconda della famiglia di FPGA utilizzata. Nel caso in esame, in cui l'FPGA è una Cyclone III contenente 10320 elementi logici, sono presenti 46 blocchi di memoria da 9kbit ciascuno, includendo anche i parity bits. Questi array di memoria hanno la particolarità di poter essere configurati come RAM, FIFO o ROM a seconda della necessità. Infatti, come si può notare dalla figura 4.1, vengono utilizzati in tutte e 3 le modalità.

### 4.1.2 EX_SPI

Il blocco **EX_SPI** è quello che si occupa della comunicazione SPI col CPLD. E' a sua volta composto da 4 blocchi interni e da 4 FIFO, ovvero blocchi di memoria M9K configurata come FIFO. Nello specifico, due FIFO sono usate per salvare il comando ricevuto da NIOS tramite i blocchi "AV_FIFO_DEC" o "HV_MODULE_TOP" e che deve essere inviato al CPLD, mentre le rimanenti due FIFO vengono utilizzate per salvare il dato che il CPLD restituisce in risposta al comando ricevuto.
Indicando le due FIFO per la scrittura del comando come FIFO_0 e FIFO_1 e quelle per la scrittura del dato del CPLD come FIFO_2 e FIFO_3, il blocco "AV_FIFO_DEC" può *solo* scrivere nella FIFO_0 e leggere dalla FIFO_2, mentre il blocco "HV_MODULE_TOP" può *solo* scrivere o leggere dalle due rimanenti FIFO.
"ARBITER_SPI", ovvero uno dei blocchi interni di **EX_SPI**, ha il duplice compito di leggere il comando salvato in una delle due FIFO e di mandarlo ad un altro blocco (denominato "SPI") insieme al numero della FIFO da cui il comando è stato letto.
Il blocco SPI è quello che si occupa di scambiare col CPLD i segnali SPI, ovvero:

1. **SS_n**
2. **Clock**
3. **MOSI** (Master Output Slave Input)
4. **MISO** (Master Input Slave Output)

Nel caso specifico, il blocco "SPI" attiva SS_n del CPLD per 26 colpi di clock, dato che il comando scambiato tra FPGA e CPLD è costituito da 26 bit ed il CPLD deve rimanere attivo per tutta la durata della comunicazione.
Inoltre, $f_{clock_{cpld}} = \frac{1}{4} \cdot f_{clock_{fpga}}$.
In aggiunta, il comando che l'FPGA inoltra al CPLD tramite il suddetto blocco "SPI", costituisce il MOSI dell'interfaccia SPI mentre il dato ricevuto dal CPLD rappresenta il MISO. Quest'ultimo, arriva al blocco "SPI" tramite un blocco denominato "DPA" che a sua volta lo inoltra al blocco "DISPATCHER" che ha il compito di salvarlo in una delle due FIFO di lettura (FIFO_2 e FIFO_3). Si precisa che il "DPA" sovracampiona il dato che arriva dal CPLD in risposta al comando ricevuto (dato

che l'FPGA lavora con $f_{clock_{fpga}} = 4 \cdot f_{clock_{cpld}}$ ) e comunica al blocco "SPI" l'istante corretto per salvarlo.

### 4.1.3 AV_FIFO_DEC

Il blocco **AV_FIFO_DEC**, come precedentemente accennato, è una FSM in grado di scrivere o leggere un dato dalle FIFO contenute all'interno di "EX_SPI". In particolare, il suo scopo è quello di gestire la procedura di inizializzazione dell'FPGA. Un esempio di comando utilizzato in questa procedura, consiste nell'accensione e spegnimento di opportuni LEDs, in base allo stato delle alimentazioni del CPLD. Questo comando viene preso da NIOS e salvato nella FIFO_0, in modo che il blocco "EX_SPI" possa inviarlo al CPLD. Il CPLD, a sua volta, restituisce in risposta il valore dei segnali che indicano lo stato delle alimentazioni. Il dato del CPLD viene quindi salvato nella FIFO_2 (in quanto il dato deve essere letto dal blocco **AV_FIFO_DEC** che può *solo* leggere dalla FIFO_2) nominata nel precedente paragrafo, viene poi letto dal blocco **AV_FIFO_DEC** che a sua volta lo inoltra al processore NIOS che contiene al suo interno una funzione per l'opportuna accensione o spegnimento dei LEDs. Questa procedura viene effettuata solo al momento dell'accensione della scheda. Questo giustifica il fatto che il blocco "ARBITER_SPI" menzionato nel precedente paragrafo, effettui la lettura dei comandi salvati nella FIFO_0 e FIFO_1 con priorità di lettura alla FIFO_1 che è quella che contiene il comando che arriva dal blocco "HV_MODULE_TOP" per leggere o scrivere una tensione da uno dei sei canali disponibili della scheda. Al momento dell'accensione, invece, *solo* il blocco **AV_FIFO_DEC** sarà l'unico blocco ad aver scritto un comando nella FIFO_0, mentre la FIFO_1 rimarrà ancora vuota. Pertanto, "ARBITER_SPI" leggerà dalla FIFO_0, procedendo così alla lettura dei segnali di stato delle alimentazioni.

### 4.1.4 HV_MODULE_TOP

Si tratta del blocco specifico per la scheda HV MODULE, ovvero quella utilizzata per questo lavoro di tesi. Questo blocco, infatti, ha il compito di salvare il comando che si vuole inviare al CPLD all'interno della FIFO_1, mentre può leggere dalla FIFO_3 il dato che il CPLD manda all'FPGA come risposta al comando ricevuto. In particolare, questo blocco riceve un dato da NIOS ed ha il compito di interpretarlo in modo da inviare al CPLD il comando corretto. Oltre a generare e salvare il comando per il CPLD nella FIFO_1, il blocco
**HV_MODULE_TOP** si occupa anche di leggere il dato che il CPLD restituisce come risposta al comando ricevuto. In particolare, dato che il firmware in dotazione è in grado di leggere tensioni sia in DC che in AC oltre che la frequenza del segnale, un opportuno blocco all'interno di **HV_MODULE_TOP** si occupa di comprendere la categoria a cui appartiene il valore letto (DC,AC,frequenza), mentre un altro blocco ha il compito di salvare il valore letto in una locazione specifica di una memoria (dove la locazione è determinata dal canale e dalla categoria di appartenenza del dato) dalla quale poi verranno effettuate le letture quando richieste.

### 4.1.5 NIOS WRAPPER

Tutte le FPGA Altera sono dotate di un processore RISC general-purpose a 16 bit o 32 bit. In particolare, la famiglia di FPGA Cyclone III contiene la seconda generazione di processore NIOS, ovvero NIOS II a 32 bit. Esso utilizza l'interfaccia AVALON per comunicare con le periferiche.
Al fine di progettare un sistema digitale di questo tipo, che includa tra i componenti il processore NIOS, delle memorie, interfacce di input/output, timers e altre periferiche personalizzate con cui il processore comunica, sono necessarie tre fasi:

1. Progetto "**Qsys**"
2. Sviluppo dell'hardware in HDL
3. Creazione del firmware

Lo strumento di progettazione **"Qsys"** è messo a disposizione da Intel/Altera all'interno del tool **"Quartus Prime"**.
Mediante interfaccia grafica, **"Qsys"** permette di selezionare tutti i componenti che si vogliono includere nel sistema (in tal caso il processore NIOS, le periferiche standard e personalizzate e i blocchi di I/O) e genera automaticamente il sistema hardware che connette tutti i componenti tra loro.
Infatti, nel caso specifico di questo progetto, tutte le periferiche mostrate in figura 4.1 con cui NIOS comunica, sono periferiche create appositamente per svolgere il compito richiesto.
Lo schema **"Qsys"** del progetto originale è mostrato in figura 4.3. La figura 4.3 rappresenta l'interfaccia grafica del tool **"Qsys"**. Come accennato, esso permette di selezionare i componenti che si vogliono includere nel sistema mediante il menu visibile a sinistra in figura. Si nota che è possibile scegliere tra una vasta gamma di componenti, suddivisi in base alla classe di appartenenza. Inoltre, per poter aggiungere le periferiche personalizzate, è necessario realizzarle e aggiungerle tra i componenti, in modo da poterle selezionare. In questo caso, sono state create due tipologie di periferiche, visibili all'interno del rettangolo in rosso in figura: *av_connect_mod* e *av_burst_mode* .

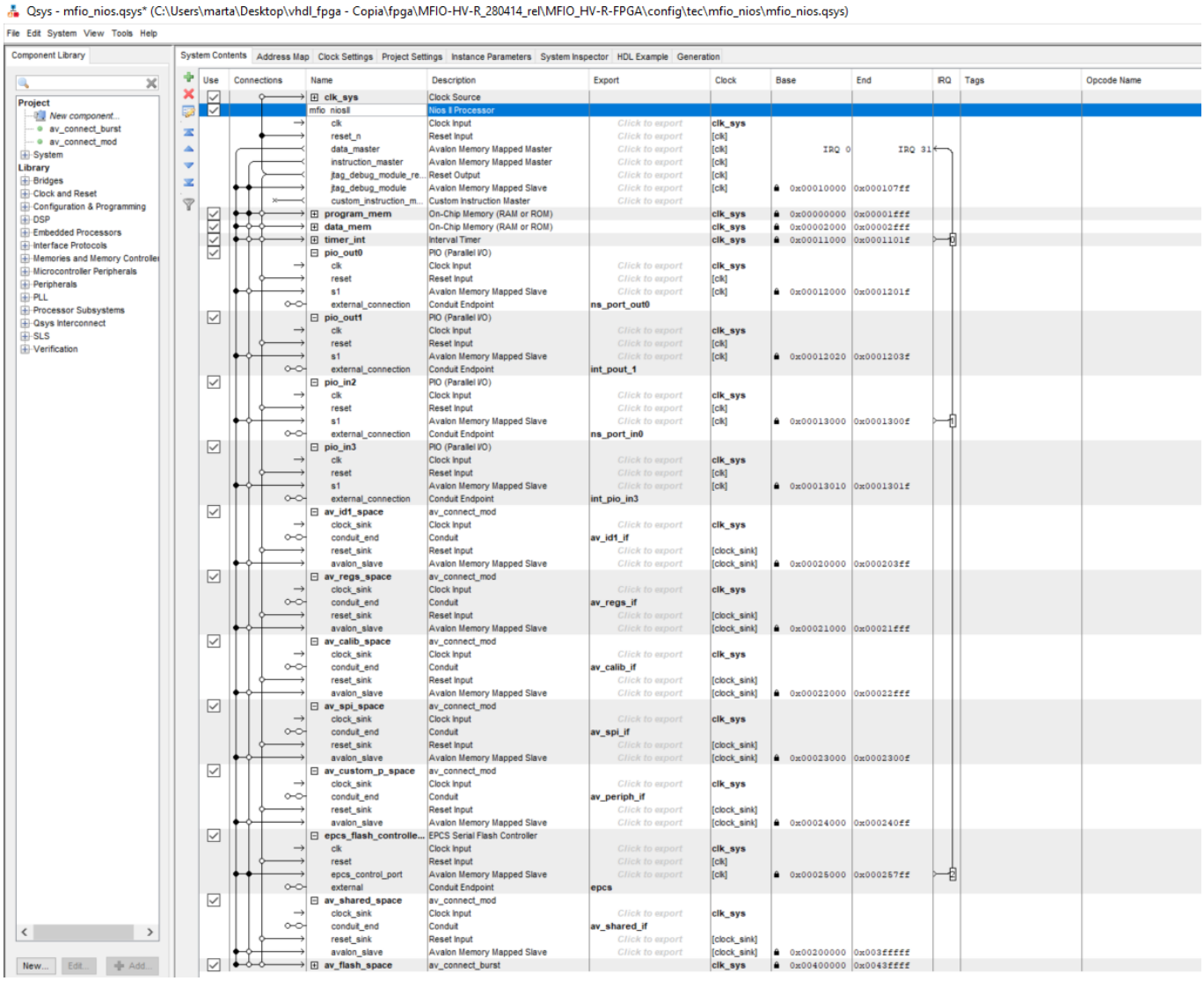


Figura 4.3: Qsys: sistema originale

Esse rappresentano l'interfaccia da creare per le periferiche personalizzate che comunicano con NIOS mostrate in figura 4.1. In particolare, sono tutte di tipo *av_connect_mode*, tranne la flash che è di tipo *av_burst_mode* perché in tal caso la flash supporta anche la modalità burst e dunque sono necessari segnali Avalon appositi.
Le periferiche, invece, sono state inserite nella parte destra di figura 4.3 e per ognuna di esse è possibile selezionare le impostazioni in base alle specifiche. Ad esempio, per le periferiche PIO (Parallel Input Output), è possibile configurare la direzione (input/output/bidirezionale), la dimensione (32 bit in questo caso), la generazione o meno di interrupt e altre impostazioni.

Infine, si nota che a ogni periferica è associato uno spazio di indirizzamento, in quanto NIOS comunica con le diverse periferiche mediante indirizzi.
Dopo aver selezionato tutti i blocchi necessari a comporre il sistema finale, viene generato l'HDL del sistema basato su NIOS. Questo viene poi incluso nella top entity del progetto **"Quartus Prime"**, considerandolo un componente.
In riferimento alla figura 4.1, il blocco **NIOS WRAPPER** rappresenta il sistema creato in "Qsys" e basato sul processore NIOS, il quale comunica sia con le periferiche interne, sia con periferiche di input/output (i cosiddetti altera_avalon_PIO, attraverso le quali è possibile, ad esempio, accendere o spegnere i led presenti su scheda, implementando opportune funzioni nel software del processore).
Dopo aver generato l'hardware, si procede con la creazione del firmware utilizzando il tool **"NIOS II Software Build tools for Eclipse (SBT)"**, ovvero un IDE per lo sviluppo del firmware di NIOS, che include un compilatore C/C++. Nel caso specifico, il firmware è stato realizzato in C.

## 4.2 Struttura CPLD

### 4.2.1 Introduzione al CPLD

Il CPLD viene principalmente utilizzato nella scheda MFIO HV come interfaccia verso i 6 canali di lettura/scrittura della scheda.
Ogni canale è composto da un DAC e un ADC che comunicano col CPLD e con le seguenti caratteristiche:

- **ADC ADS8327**[4]
  - Risoluzione: 16 bit
  - Tipo: Capacitor-based SAR con integrato sample and hold
  - Compatibile SPI con SCLOCK fino a 50MHz
  - Segnale convst_n per avviare la conversione
  - INL Max:± 2LSB
  - DNL Max:± 1LSB
- **DAC 8812**[5]
  - 2 uscite in corrente a 16 bit
  - SPI compatibile con SCLOCK fino a 50MHz
  - Accuratezza: 1 LSB Max
  - DNL: 1 LSB Max

Sulla baseboard e sulla piggy board sono inoltre presenti due componenti MAX14803A [6], utilizzati per connettere le uscite del DAC e l'ingresso dell'ADC tra loro o con l'esterno. La gestione di questi due componenti è anch'essa compito del CPLD, tramite la periferica GPO.
Infine, il CPLD controlla anche lo stato di alcune alimentazioni tramite la periferica GPI.
In figura 4.4 si riporta lo schema a blocchi della struttura originale del CPLD. Nello specifico, i macro blocchi principali sono 5:

1. **DECODER**
2. **ADCs_INTERFACE**
3. **DACs_INTERFACE**
4. **GPI (General Purpose Input)**
5. **GPO (General Purpose Output)**

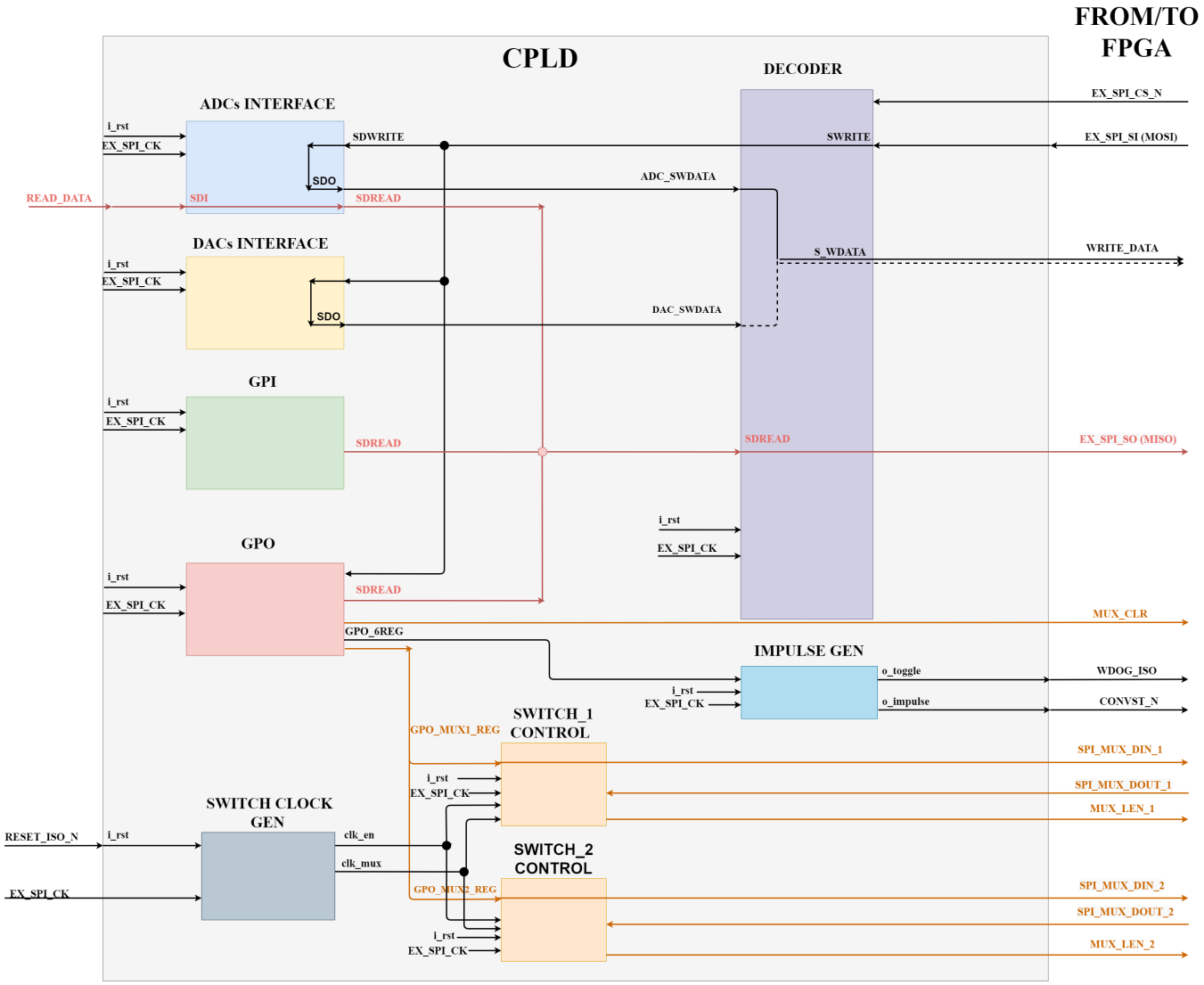


Figura 4.4: Struttura originale CPLD

### 4.2.2 Protocollo di comunicazione

Analizzando il CPLD, si è scoperto che FPGA e CPLD comunicano tramite un pacchetto di 26 bit, inviati in maniera seriale con interfaccia SPI.
Il comando inviato dall'FPGA al CPLD è illustrato in figura 4.5.


Figura 4.5: Dato scambiato tra FPGA e CPLD

In particolare, il significato di ciascun bit è il seguente:

- **R_n/W**: indica se si vuole eseguire una lettura o una scrittura;
- **p1,p0**: servono per identificare con quale blocco comunicare all'interno del CPLD, secondo la decodifica mostrata in tabella 4.2;

| R*/W | p1 | p0 | Periferica |
|---|---|---|---|
| x | 0 | 0 | **ADC** |
| 1 | 0 | 1 | **DAC** |
| X | 1 | 0 | **GPO** |
| 0 | 1 | 1 | **GPI** |

Tabella 4.2: Decodifica Periferica del CPLD

- **c2,c1,c0**:indicano il canale in cui effettuare l'operazione di lettura/scrittura;
- **a1,a0**: nel caso di comunicazione con uno dei 6 DACs, indicano quale dei due canali interni ai DACs utilizzare per la scrittura della tensione. Nel caso di comunicazione con il GPO, invece, vengono utilizzati dal GPO per comprendere se il dato presente negli ultimi 16 bit del comando arrivato dall'FPGA indica il nuovo pattern da impostare ai segnali di controllo degli SWITCHES o al segnale utilizzato da IMPULSE_GEN per generare il segnale *convst_n*, usato dagli ADCs per avviare una nuova conversione;
- **d16...d0**: indicano il dato a 16 bit che viene scambiato tra FPGA e CPLD. In particolare, nel pacchetto a 26 bit che arriva dall'FPGA, questi 16 bit assumono un reale significato solo nel caso in cui si voglia effettuare la scrittura di una

tensione, in quanto in tal caso rappresentano il dato che si vuole scrivere, o nel caso in cui si comunichi con il GPO, in quanto in tal caso rappresentano i segnali di controllo per il componente SWITCH, oppure il segnale usato per generare il *convst_n*, usato dagli ADCs per avviare la conversione di un dato. Nel caso, invece, in cui l'FPGA mandi una richiesta di lettura dagli ADCs, oppure il comando per GPI, questi bit sono "don't care".
Nel caso in cui, invece, sia il CPLD a trasmettere il pacchetto a 26 bit all'FPGA, questi ultimi 16 bit rappresentano il dato che il CPLD legge da una delle periferiche e che inoltra all'FPGA. Nel caso di comando di scrittura di una tensione (effettuata dai DACs) o scrittura dei segnali di controllo per SWITCH e ADCs (che viene effettuata dal GPO), il CPLD restituisce come risposta gli stessi 16 bit presenti nel comando inviatogli dall'FPGA.

### 4.2.3 DECODER

Il **DECODER** si occupa inizialmente di capire con quale periferica interna al CPLD deve comunicare. A tal fine, come mostrato precedentemente, analizza i bit R*/W,p1,p0, effettua la decodifica mostrata in tabella 4.2 e abilita *solo* la periferica da considerare, inoltrando ad essa i 3 bit identificativi del canale e il dato da scrivere, mentre riceve da essa il dato da leggere.
In caso di comando di scrittura (*comando GPO* e *comando DAC*) restituisce come MISO nei 16 bit meno significativi tutti '1' e nei 10 più significativi gli stessi bit del comando ricevuto.
In caso di comando di lettura (*comando GPI* e *comando ADC*) il MISO è formato dal dato letto da uno degli ADC (corrispondente al canale richiesto), oppure dalla sequenza "0010 & power_state" nel caso di comando GPI.
Infine, il decoder gestisce la scrittura del dato nel DAC o nell'ADC (nel caso di ADC si tratta di dati di configurazione).

### 4.2.4 ADCs_INTERFACE

Questo blocco rappresenta l'interfaccia tra il CPLD e i 6 ADCs presenti su scheda (uno per ogni canale). Esso riceve dal decoder il bit R/W* che indica se leggere o

scrivere (anche negli ADCs è possibile scrivere per impostare il registro di configurazione) e riceve inoltre il numero del canale su cui effettuare l'operazione richiesta e lo *Start*. In aggiunta, riceve dall'esterno il dato convertito da uno dei 6 ADCs (quello corrispondente al canale selezionato). Questo blocco, inoltre, si occupa di generare il clock e i chipselect per i 6 ADCs presenti su scheda e manda questi segnali in uscita al CPLD in modo che possano raggiungere il componente specificato. In particolare, attiva solo il chipselect dell'ADC corrispondente al canale specificato nei 3 bit del comando c2,c1 e c0.
In caso di scrittura, questo blocco riceve il dato dal DECODER e lo ritrasmette per poterlo scrivere negli ADC. In caso di lettura, invece, l'ADC_interface manda al DECODER il dato ricevuto dall'opportuno ADC e il DECODER, a sua volta, lo manda in maniera seriale in uscita dal CPLD come MISO. Questo dato sarà allineato agli ultimi 16 bit del pacchetto a 26 bit scambiato tra CPLD e FPGA.

### 4.2.5 DACs_INTERFACE

Questo blocco rappresenta l'interfaccia tra il CPLD e i 6 DACs presenti su scheda (uno per ogni canale). Esso riceve dal decoder il bit R/W* che indica se leggere o scrivere (*si precisa che nei DACs si può effettuare solo una scrittura*) e riceve inoltre il numero del canale in cui effettuare la scrittura di un determinato valore, ovvero quello indicato negli ultimi 16 bit del pacchetto a 26 bit che l'FPGA manda al CPLD e che arriva a questo blocco tramite il blocco DECODER. Infine, il **DACs_INTERFACE** rimanda indietro al DECODER il valore da scrivere nell'opportuno DAC ed il decoder, a sua volta, lo manda in uscita dal CPLD in modo che possa essere scritto nel DAC specificato. Inoltre, questo blocco si occupa di generare il clock e i chipselect per i 6 DACs presenti su scheda e manda questi segnali in uscita dal CPLD. In particolare, attiva solo il chipselect del DAC corrispondente al canale specificato nei 3 bit del comando c2,c1 e c0.

### 4.2.6 GPI

Questo blocco viene utilizzato al momento dell'inizializzazione della scheda, in quanto è collegato ai segnali che indicano lo stato delle alimentazioni. Dunque, quando nel comando che arriva dall'FPGA viene richiesta una lettura dal GPI, il decoder si occuperà di attivare questa periferica mediante il segnale *Start*. Il suo compito, sarà quello di mandare al DECODER in maniera seriale la sequenza di bit "0010 & power_state" e il DECODER, a sua volta, manderà questo pattern all'FPGA come MISO. La dicitura power_state indica in modo generico i segnali che indicano lo stato delle alimentazioni presenti su scheda e che sono ingressi del componente GPI.
In seguito, questi segnali verranno letti dal processore NIOS il quale, tramite un'opportuna funzione, accenderà i LEDs sulla scheda solo nel caso in cui la corrispondente alimentazione sia corretta.

### 4.2.7 GPO

Questo blocco manda ai due componenti SWITCH presenti sulla scheda i segnali necessari per effettuare l'opportuna connessione tra ADC e DAC, in base al comando richiesto.
Per quanto riguarda i segnali inoltrati agli SWITCHES, il GPO genera due segnali a 16 bit che entrano rispettivamente in "SWITCH_1_CONTROL" e in "SWITCH_2_CONTROL" (riferimento alla figura 4.4) che a loro volta li mandano allo SWITCH corrispondente. SWITCH_1_CONTROL si occupa di smistare i segnali allo SWICTH che gestisce i canali 1,2,3 mentre SWITCH_2_CONTROL si occupa di smistare i segnali allo SWITCH che gestisce i canali 4,5,6.
Il GPO riceve quindi dal DECODER il bit R/W* che indica se si vuole leggere o scrivere, oltre al dato che si vuole scrivere negli SWITCH (ovvero il comando per generare le opportune connessioni tra i segnali mediante i componenti SWICTH).
Oltre a mandare i comandi a 16 bit per i due SWICTH, il GPO si occupa anche di generare il *write_enable* per gli SWITCH (nello specifico, solo il *write_enable* dello SWITCH che contiene i segnali relativi al canale desiderato verrà abilitato) e a sua volta questo segnale verrà inviato ai componenti SWITCH_i_CONTROL,
con i=1,2 , i quali lo manderanno allo SWITCH corrispondente. Si occupa anche di

generare il segnale *clear* per gli SWITCH, che è un uscita del CPLD.
Oltre a generare i comandi per gli SWITCHES, il GPO si occupa anche di impostare un segnale che entra nel blocco IMPULSE_GEN, il quale lo utilizza per generare il segnale *convst_n* che esce dal CPLD e serve agli ADCs per iniziare una nuova conversione.

### 4.2.8 SWITCH_i_CONTROL

Come accennato nel paragrafo precedente, questi blocchi sono utilizzati per comunicare con i due SWITCHES presenti sulla scheda. In particolare, ricevono dal blocco GPO il *write_enable* e il comando per lo SWITCH ed inoltrano il comando solo nel caso in cui il *write_enable* sia abilitato, abilitando o meno il *load_enable* dello SWITCH con cui si interfacciano. Il comando, inoltre, viene mandato agli SWITCHES in maniera seriale, come avviene per tutti i segnali scambiati nel sistema in questione, in quanto tutti i blocchi comunicano mediante protocollo SPI.
Gli SWITCHES sono utili perché, in base ai segnali ricevuti, sono in grado di generare la connessione desiderata tra quelle possibili. La figura 4.6, mostra lo schema a blocchi delle connessioni che lo SWITCH è in grado di realizzare per l'iesimo canale. Con riferimento alla figura 4.6 [1], i segnali denominati *IN_OUT_i +* e *IN_OUT_i -*, sono le tensioni positiva e negativa che si desidera misurare sull'iesimo canale. Gli ADCs, infatti, ricevono in ingresso una tensione differenziale.
Se si vuole effettuare una lettura di tensione (modalità *Monitor*), è necessario chiudere gli switches 1 e 2 che collegano questi due segnali all'iesimo ADC, che si occuperà poi di convertire il segnale analogico in una sequenza di bit, la quale sarà infine inviata all'FPGA. Nel caso si voglia, invece, imporre una certa tensione su un canale (modalità *Simulation*), si utilizza il componente DAC. In questo caso è necessario chiudere gli switches 3 e 4. Quindi, prima di effettuare un'operazione di lettura o scrittura in un determinato canale, è necessario eseguire gli opportuni collegamenti dei segnali, pilotando gli switches in maniera corretta mediante il comando GPO.

[1] Si precisa che i possibili collegamenti ammessi sono stati desunti dagli schemi elettrici della scheda, proprietà di Leonardo S.p.A.

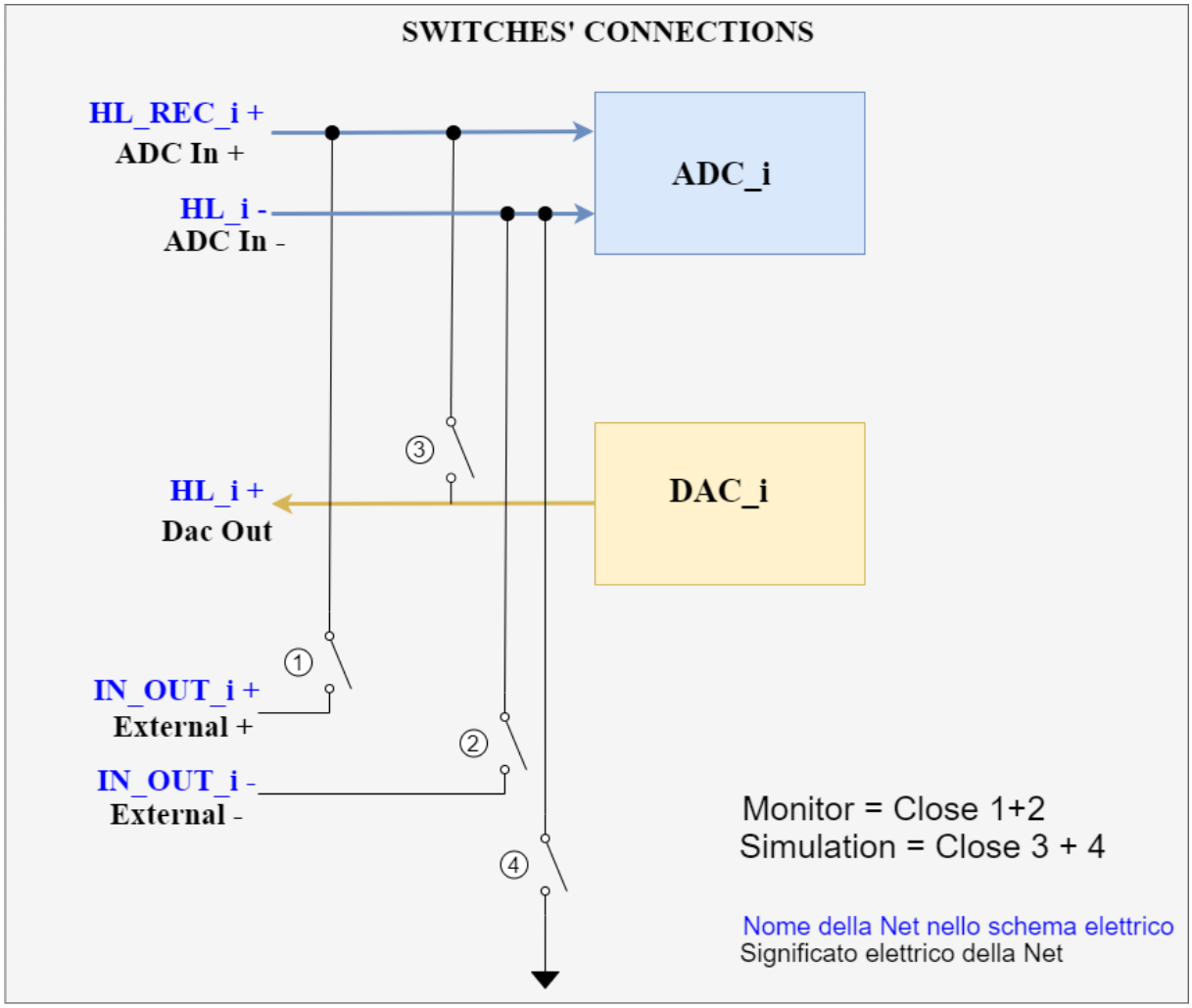


Figura 4.6: Schema a blocchi connessioni SWITCH

# CAPITOLO 5

# Analisi dei comandi e simulazioni CPLD

## 5.1 Analisi dei comandi

Come dettagliato nel paragrafo 4.2.2, l'FPGA e il CPLD comunicano attraverso un pacchetto di 26 bit, come illustrato in figura 4.5.
Il dato è scambiato via seriale tra master (FPGA) e slave (CPLD) e la risposta che il CPLD invia all'FPGA è istantanea: per ogni bit che l'FPGA invia al CPLD, segue un bit che il CPLD invia come risposta all'FPGA. In particolare, la risposta del CPLD è composta dai bit del comando stesso nei 16 bit più significativi, mentre i 16 bit meno significativi sono tutti a '1' nel caso di comando di sola scrittura oppure contengono il dato richiesto nel caso di comando di lettura.
Dall'analisi effettuata sul firmware del CPLD, ne è emerso che i comandi che esso può ricevere sono i seguenti:

1. **Lettura di una tensione da uno dei sei canali presenti sulla scheda**. *Per semplificare la trattazione, si farà riferimento a questo comando con la denominazione "comando_ADC".*
   Un esempio di *comando_ADC* riferito al canale 2 è mostrato in tabella 5.1. Per tutti i comandi che verranno illustrati nel seguito, a seconda del canale di riferimento, i bit che cambiano sono i bit in posizione 22, 21, 20 (considerando LSB in posizione 0). Si precisa che i canali sono numerati dallo 0 al 5, motivo per cui il canale 2 corrisponde alla sequenza "001".
   Nel caso di "comando_ADC", il CPLD restituirà come risposta un dato a 16

| R_n/W | p1 | p0 | CHAN | | | PAD | | a1 | a0 | Dato | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Tabella 5.1: Esempio comando ADC canale 2

bit, che è generato dall'ADC del canale richiesto. Infatti, il dato generato come uscita dai sei ADC presenti sulla scheda, è un ingresso del CPLD.

2. **Scrittura di una tensione in uno dei sei canali presenti sulla scheda**. *Per semplificare la trattazione, si farà riferimento a questo comando con la denominazione "comando_DAC".*
   Un esempio di *comando_DAC* riferito al canale 3 è mostrato in tabella 5.2.

| R_n/W | p1 | p0 | CHAN | | | PAD | | a1 | a0 | Dato | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

Tabella 5.2: Esempio comando DAC canale 3

In tal caso, per forzare una tensione su un canale, si utilizza il DAC collegato al canale specificato nel comando. Il comando contiene nei 16 bit meno significativi il valore di tensione che si vuole imporre sul canale di interesse (tutti '1' nell'esempio, che corrisponde al fondo scala). Questo valore verrà letto dal CPLD che a sua volta si occuperà di scriverlo nel DAC del canale selezionato. Inoltre, i bit in posizione 17 e 16 indicano il canale interno al DAC da selezionare. Nel caso dell'esempio, si seleziona il canale 1.

3. **Lettura dei segnali che indicano lo stato delle alimentazioni**.
   *Per semplificare la trattazione, si farà riferimento a questo comando con la denominazione "comando_GPI"*, in quanto la periferica usata all'interno del CPLD per leggere i segnali di stato delle alimentazioni è proprio il GPI.
   Un esempio di *comando_GPI* riferito al canale 1 è mostrato in tabella 5.3. Si tratta dello stesso comando menzionato nel paragrafo dedicato al blocco "AV_FIFO_DEC", il quale si occupa di leggere i segnali di stato delle alimentazioni che sono collegati al CPLD. In tal caso, la periferica all'interno del CPLD

| R_n/W | p1 | p0 | CHAN | | | PAD | | a1 | a0 | Dato | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Tabella 5.3: Esempio comando GPI canale 1

con cui si comunica è il GPI.

Si tratta, inoltre, dell'unico comando per il quale è prevista una risposta attesa nel caso di corretto funzionamento del sistema. In particolare, nel caso in cui le alimentazioni siano funzionanti, ci si aspetta che il CPLD invii all'FPGA come risposta il pattern "00100101" negli 8 bit meno significativi. In particolare, i 4 bit meno significativi indicano lo stato delle alimentazioni. Se le alimentazioni sono corrette, ci si aspetta uno '0' per le alimentazioni negative (-60V e -12V) e un '1' per le alimentazioni positive (+12V e +60V). Infine, dato che si tratta dell'unico comando per il quale è prevista una risposta attesa, questo viene utilizzato da NIOS per verificare il corretto allineamento della risposta del CPLD o per correggerlo in caso che la risposta del CPLD sia disallineata. Questo argomento verrà trattato in dettaglio nel capitolo 7.

4. **Attivazione conversione dato ADC**

   *Per semplificare la trattazione, si farà riferimento a questo comando con la denominazione "comando_GPO_ADC"*, in quanto la periferica usata all'interno del CPLD è il GPO, che in questo caso viene usato per attivare il segnale *convst_n* che avvia la conversione del dato negli ADC.

   Un esempio di *comando_GPO_ADC* riferito al canale 4 è mostrato in tabella 5.4.

   Si precisa che nel caso di questo di comando il segnale *convst_n* viene attivato non solo per l'ADC corrispondente al canale selezionato, ma per gli ADC di tutti i canali in quanto lo stesso segnale è collegato ai 6 ADC presenti su scheda. Questo non comporta anomalie di funzionamento in quanto si legge il dato convertito solo dell'ADC che ha il *chip_select* attivo. Come mostrato nell'esempio in tabella 5.4, per distinguere tra il comando *comando_GPO_ADC* e *comando_GPO_SWITCH* vengono utilizzati i bit in posizione 17 e 16. Nel caso di comando *comando_GPO_ADC* essi sono impostati a '10'. Infine, nei 16

| R_n/W | p1 | p0 | CHAN | | | PAD | | a1 | a0 | Dato | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |

Tabella 5.4: Esempio comando GPO ADC canale 4

bit di dato si scrive LSB='1', il quale permette di generare un impulso per il *convst_n* e avviare la conversione del dato nell'ADC del canale richiesto (canale 4 nell'esempio).

5. **Segnali di comando per il componente SWITCH**
   *Per semplificare la trattazione, si farà riferimento a questo comando con la denominazione "comando_GPO_SWITCH"*, per distinguerlo dal *comando_GPO_ADC*, anch'esso realizzato dalla periferica GPO.
   Un esempio di *comando_GPO_SWITCH* riferito al canale 4 è mostrato in tabella 5.5. In questo caso, in base al valore dei bit che riceve, lo SWITCH è

| R_n/W | p1 | p0 | CHAN | | | PAD | | a1 | a0 | Dato | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |

Tabella 5.5: Esempio comando GPO SWITCH canale 4

in grado di effettuare connessioni differenti tra ADC e DAC dei diversi canali. Nell'esempio mostrato in tabella 5.5, lo SWITCH riceve '1111' per il canale 4. Questa sequenza di segnali di controllo permette di impostare sia la modalità *Simulation* che la modalità *Monitor* sul canale 4. In pratica, con riferimento alla figura 4.6, questa sequenza permette di chiudere tutti e 4 gli switches per il canale 4.

## 5.2 Simulazioni

Conclusa la fase di analisi, è stato necessario creare delle simulazioni dell'intero blocco CPLD per confermare le ipotesi fatte e le conoscenze acquisite.
Come si evince dalle simulazioni riportate nel seguito, esse hanno confermato i risultati attesi. Si è pertanto deciso di utilizzare il progetto del CPLD come punto di

partenza per la creazione del nuovo firmware dell'FPGA. Per effettuare le simulazioni del firmware del CPLD si è utilizzato il software "Modelsim". A tal fine, si è realizzato un testbench in grado di testare i 5 possibili comandi previsti per il CPLD, considerando nel progetto il firmware in dotazione per il CPLD e il nuovo firmware realizzato per l'FPGA.
Si riportano di seguito le simulazioni relative ai 5 possibili comandi, che coinvolgono le periferiche ADC, DAC, GPI e GPO presenti all'interno del CPLD.

1. **Simulazione blocco "ADC interface"**.

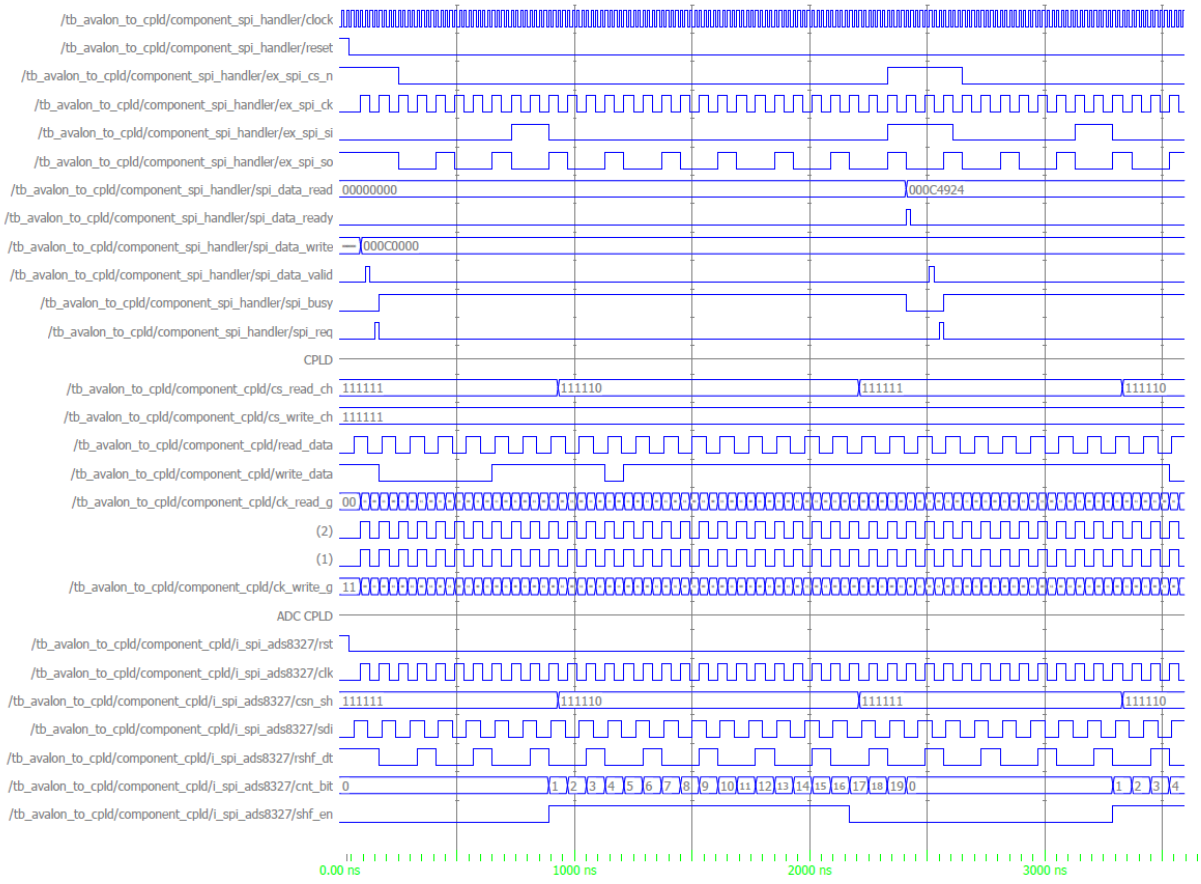


Figura 5.1: Simulazione blocco ADC interface

La figura 5.1 mostra la simulazione effettuata per il *comando_ADC*, riferito al canale 1.
In questo caso, la periferica coinvolta all'interno del CPLD è l'ADC interface. Dalla figura, si evince che questo blocco attiva il *chip_select_n* dell'ADC corrispondente al canale 1, conforme col comportamento atteso. Si nota infatti,

che il segnale *cs_read_ch*, formato da 6 bit che rappresentano i *chip_select* per ognuno dei 6 canali, ha uno '0' in posizione 0, che indica proprio il *chip_select* relativo al primo canale (*chip_select* è attivo col valore logico basso). Inoltre, dalla figura si può notare che il dato mandato dal CPLD in risposta al comando ricevuto contiene nei 16 bit più significativi i 16 bit più significativi del comando (0x000C nell'esempio in figura) e nei 16 bit meno significativi il dato convertito dall'ADC del canale selezionato. In questo caso, si è forzato da testbench in valore 0x4924 che corrisponde, infatti, ai 16 bit meno significativi del segnale *spi_data_read*. Infine, si evince che nel momento in cui arriva il dato dal CPLD, viene attivato il segnale data_ready dal blocco "SPI_INTERFACE" interno a "SPI_HANLDER", che verrà descritto più nel dettaglio nel capitolo 7.
Si può dunque concludere che questo blocco rispetta il comportamento atteso.

2. **Simulazione blocco "DAC interface"**.

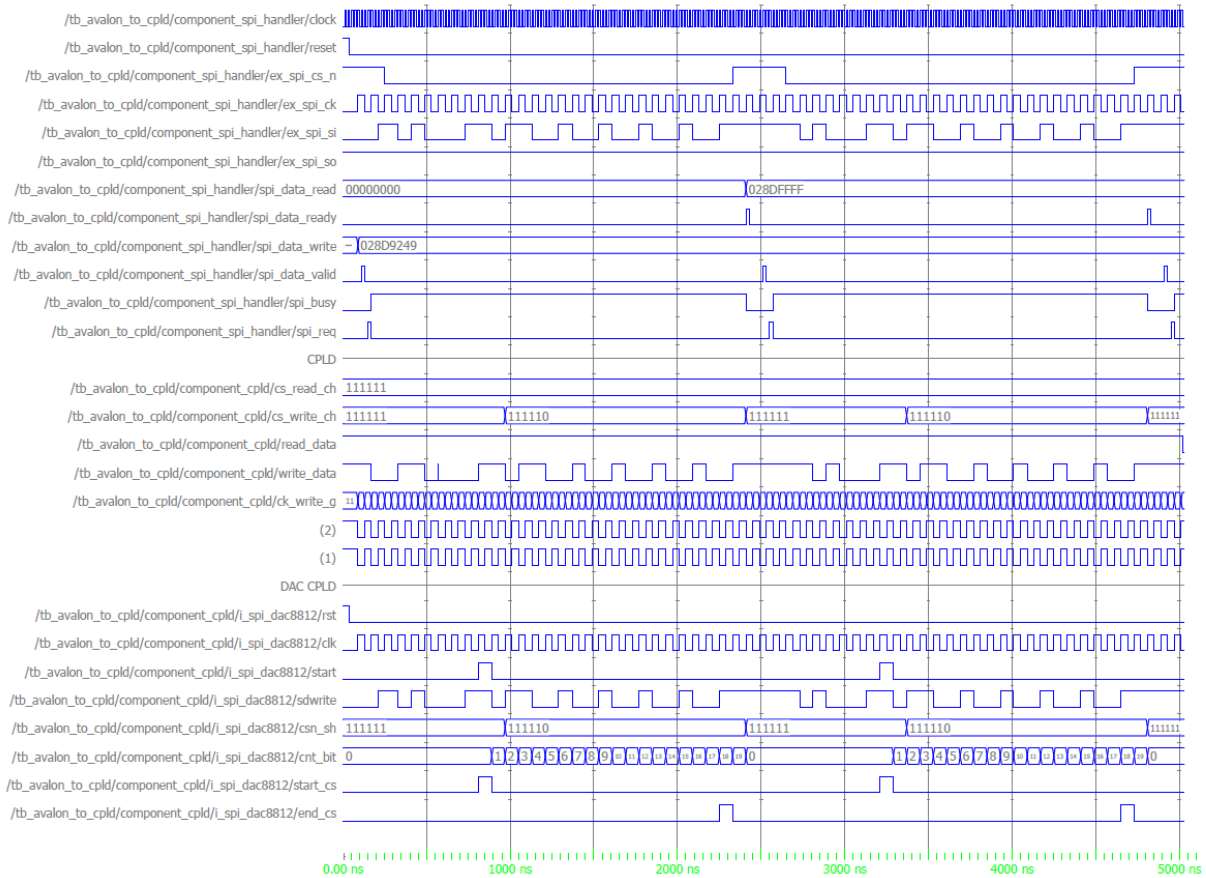


Figura 5.2: Simulazione blocco DAC interface

In figura 5.2 è mostrata la simulazione effettuata per il *comando_DAC*, riferito al canale 1 della scheda e al canale 1 interno al componente DAC. In tal caso, la periferica coinvolta all'interno del CPLD è il DAC interface.
Dal segnale *cs_write_ch* si evince che questo blocco attiva il *chip_select* del DAC corrispondente al canale 1, rispettando il comportamento atteso. Inoltre, trattandosi di un comando di sola scrittura, si può notare che il CPLD restituisce nei 16 bit di dato tutti '1', come spiegato all'inizio del presente capitolo. Infine, nel momento in cui arriva il dato valido dal CPLD, viene attivato il segnale data_ready dal blocco "SPI_INTERFACE" interno a "SPI_HANLDER", che verrà descritto più nel dettaglio nel capitolo 7. Questo segnale viene sempre attivato quando arriva una risposta dal CPLD, a prescindere dal tipo di comando.

3. **Simulazione blocco "GPI"**.

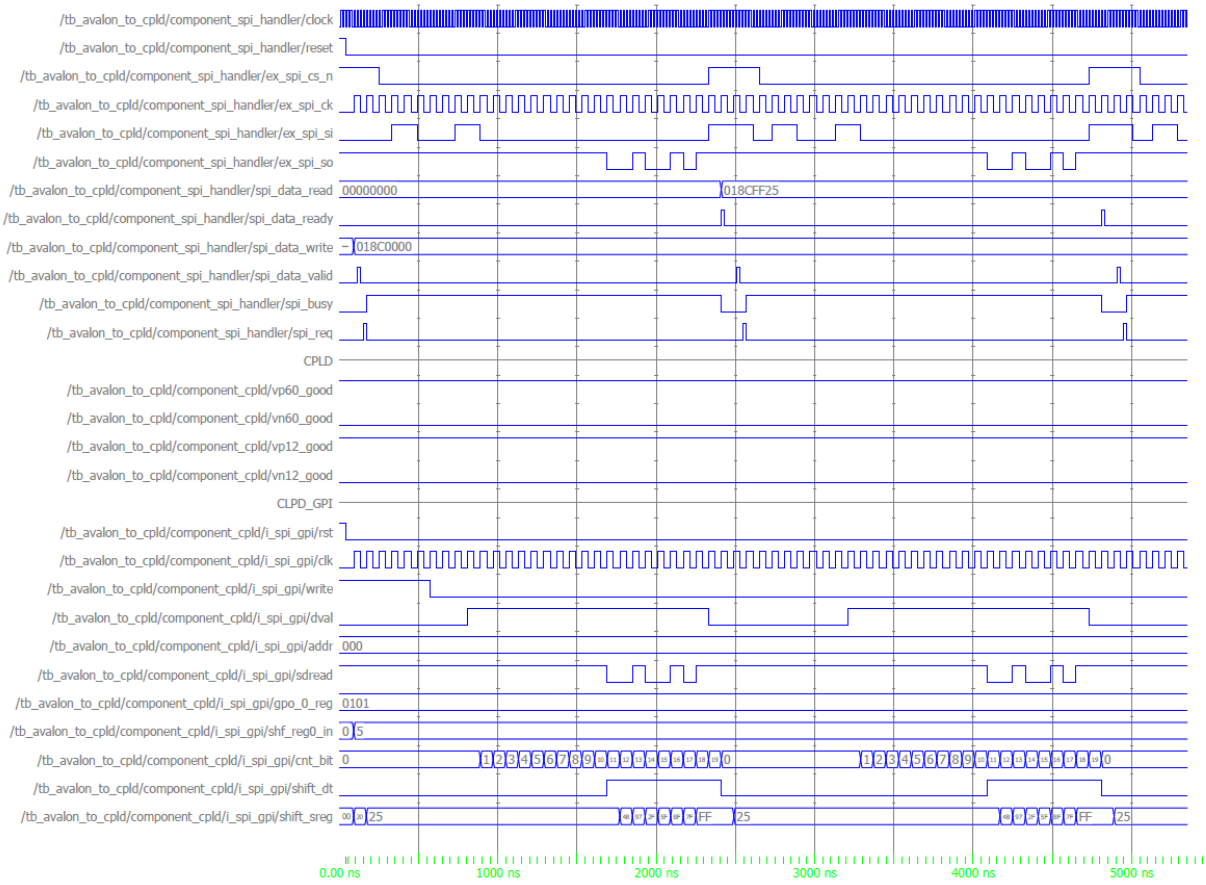


Figura 5.3: Simulazione blocco GPI

La figura 5.3 mostra la simulazione corrispondente al *comando_GPI*, riferito al canale 1 della scheda. Come descritto nel paragrafo 5.1, la risposta attesa per questo comando corrisponde alla sequenza '00100101' (0x0025 in esadecimale). Ai fini della simulazione, i segnali di alimentazione sono stati impostati pari a '0101', in quanto si tratta del valore che devono assumere in caso di corretto funzionamento.

Dalla figura 5.3, si evince che il CPLD manda come risposta al *comando_GPI* la sequenza attesa (riferimento agli 8 bit meno significativi del segnale *spi_data_read*). Si può dunque affermare che anche questo blocco si comporta nella maniera corretta.

4. **Simulazione blocco "GPO" (ADC).**

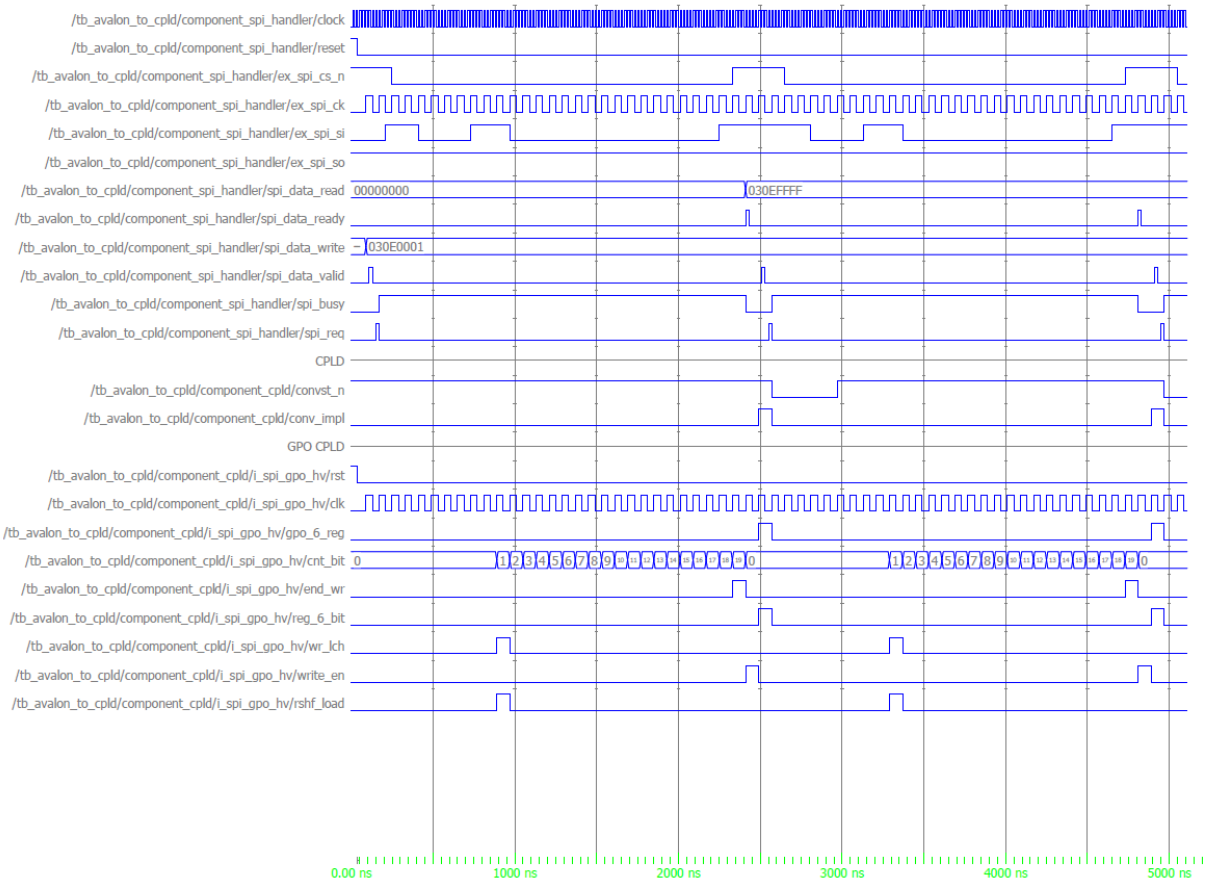


Figura 5.4: Simulazione blocco GPO (ADC)

La figura 5.4 mostra la simulazione corrispondente al *comando_GPO_ADC*, corrispondente al canale 1 della scheda.
In tal caso, dalla lettura dei bit di comando in posizione 17 e 16 (pari a '10'), la periferica GPO comprende di dover abilitare il segnale che genererà poi il *convst_n* per gli ADC. Il segnale che viene inviato al blocco del CPLD incaricato di generare l'impulso, è il segnale denominato *reg_6_bit* in figura.

5. **Simulazione blocco "GPO" (SWITCH)**.

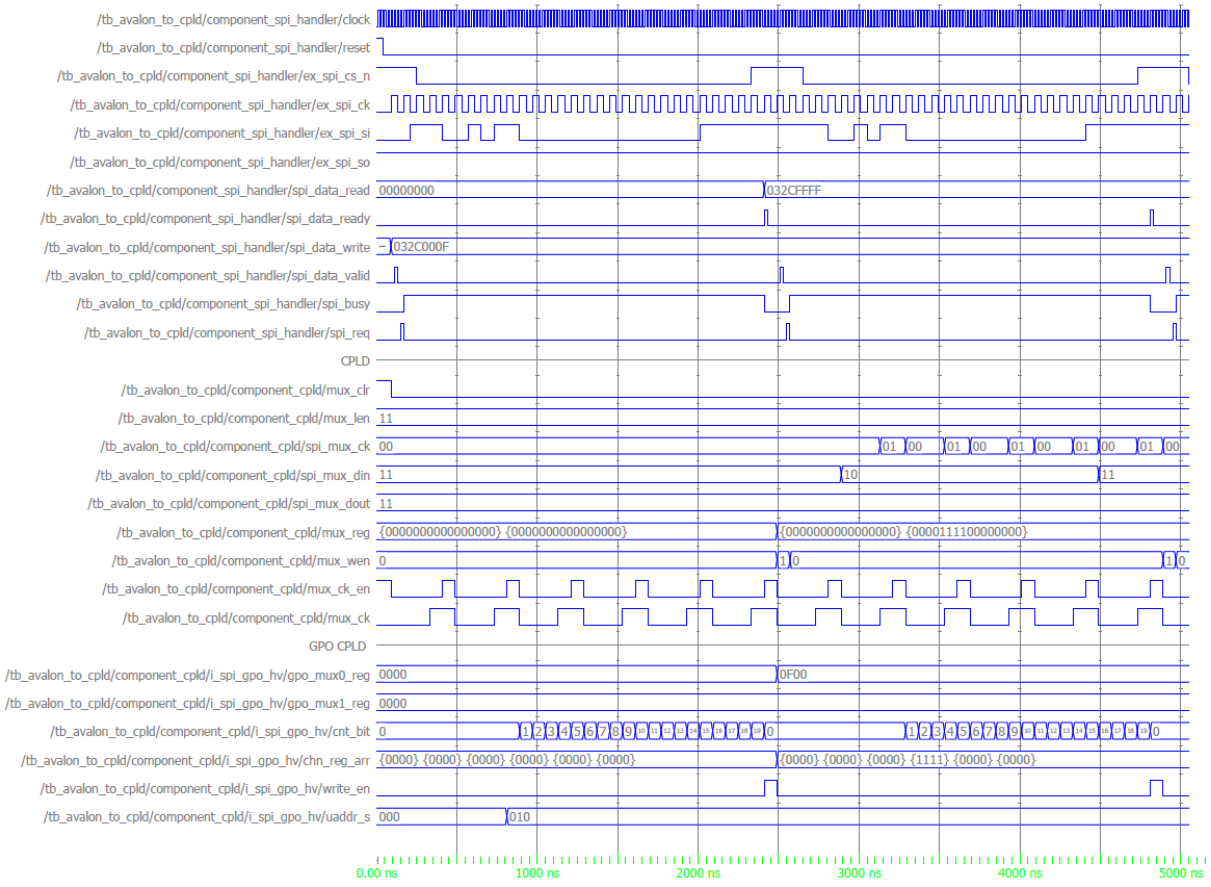


Figura 5.5: Simulazione blocco GPO (SWITCH)

In figura 5.5 è riportata la simulazione corrispondente al *comando_GPO_SWITCH*, riferito al canale 3 della scheda. Nell'esempio in figura si vuole impostare sia la modalità *Simulation* che la modalità *Monitor*, che corrisponde al comando '1111' per lo switch che gestisce il canale 3.
Infatti, dalla figura si evince che il blocco GPO genera un array di 6 vettori (ri-

ferimento a *chn_reg_arr*), uno per ogni canale, il quale contiene nella posizione del canale indicato i comandi per gli switch. In tal caso, poiché nel *comando_GPO_SWITCH* si è selezionato il canale 3, si nota che il vettore in posizione 3 nell'array *chn_reg_arr* contiene la sequenza '1111'. Questa verrà poi inoltrata alla periferica "SWITCH_i_CONTROL" all'interno del CPLD, la quale si occupa di inoltrare i segnali di controllo allo SWITCH corrispondente (in questo caso al primo SWITCH dato che è quello che gestisce i primi 3 canali).
Infine, si può notare che anche in questo caso il CPLD restituisce i 16 bit di dato pari a '1', dal momento che anche il *comando_GPO_SWITCH* è un comando di sola scrittura.
Per concludere questo capitolo, si può affermare che la comunicazione tra FPGA e CPLD avviene in maniera corretta e che i blocchi interni al CPLD si comportano nel modo atteso.

CAPITOLO 6

# Interfaccia Avalon e Standard ANSI/VITA 4-1995

## 6.1 Interfaccia Avalon

Come accennato nei precedenti paragrafi, il processore NIOS comunica con le diverse periferiche mediante l'**interfaccia Avalon** [7], la quale è utilizzata anche per la comunicazione tra i componenti interni (ad esempio tra il blocco "IP_BRIDGE" e il blocco "MEM_DECODER" con riferimento alla figura 4.1). I componenti disponibili in " **Qsys**", utilizzati nella fase iniziale di progettazione dell'hardware di NIOS, sono dotati di questo tipo di interfacce standard. Tuttavia è anche possibile inglobare le interfacce Avalon nei componenti personalizzati, come è stato fatto nel presente progetto. Infatti, le interfacce Avalon definite da Intel/Altera, permettono di semplificare il progetto ammettendo una semplice connessione tra i diversi componenti nelle FPGA Intel/Altera.

Esistono sette diversi tipi di interfaccia Avalon, ma solo una di esse è utilizzata nel progetto in esame: la cosiddetta "**Avalon Memory Mapped Interface**"
**(Avalon - MM)**. Si tratta di un tipo di interfaccia per lettura e scrittura basata su indirizzi, tipica delle connessioni master - slave.

Come si legge da [7], un singolo componente può includere un numero qualsiasi di questo tipo di interfacce e può anche includere più istanze dello stesso tipo di interfaccia.

Con riferimento al progetto illustrato nel capitolo 4, tale situazione si verifica ad esempio nel caso del blocco MEM_DECODER che comunica mediante interfaccia Avalon - MM con sette componenti diversi.

### 6.1.1 Segnali Avalon-MM

Ogni tipo di interfaccia Avalon definisce una serie di segnali, specificandone per ciascun segnale il ruolo e il comportamento. I segnali minimi richiesti sono *readdata* e *read* per una interfaccia di sola lettura e *writedata* e *write* per una interfaccia di sola scrittura. Quindi, in base alla funzionalità richiesta, è possibile scegliere di utilizzare solo i segnali necessari, tuttavia tenendo conto del fatto che i segnali sopra citati sono obbligatori nel caso si voglia rispettivamente effettuare una lettura e una scrittura.
Nel caso specifico del lavoro di tesi, si vuole effettuare sia un'operazione di lettura che di scrittura dalle diverse memorie (fatta eccezione per le ROM che supportano, come noto, solo la lettura). Pertanto, nel nostro caso, saranno sempre presenti i quattro segnali sopra riportati, con aggiunta di altri segnali che vengono spiegati in tabella 6.1.
*Si precisa che il pedice "n" indica che il segnale corrispondente è attivo col valore logico basso. Per semplificare la trattazione, si riporta in tabella solo questa notazione, in quanto i corrispondenti segnali nel progetto vengono attivati col valore logico basso, sebbene l'interfaccia Avalon ammetta l'attivazione dei segnali anche col valore logico alto. Inoltre, il campo "Dimensione" in tabella, indica il numero di bit ammessi per il corrispondente segnale dall'Interfaccia Avalon. Nel progetto, vengono utilizzate solo alcune dimensioni tra quelle possibili (tipicamente 32 bits/16 bits per il readdata e writedata e 4 bits/2 bits per il byteenable)* .
Infine, in aggiunta ai segnali riportati in tabella 6.1, nel progetto è stato aggiunto un ulteriore segnale denominato *chipselect*, attivo col valore logico alto. Esso ha la funzione di abilitare o disabilitare lo slave con cui il master comunica. Ad esempio, se il blocco "MEM_DECODER" vuole richiedere un'operazione di lettura o scrittura alla periferica denominata "CHX_REG", dovrà attivare il *chipselect* di questa periferica, oltre agli opportuni segnali in base all'operazione da svolgere.

| Segnale | Dimensione | Direzione | Descrizione |
|---|---|---|---|
| ***address*** | 1-64 | Master → Slave | Seleziona la locazione in memoria da cui/in cui effettuare una lettura/scrittura |
| ***byteenable*** | 2,4,8,16, 32,64,128 | Master → Slave | Abilita uno o più bytes nel caso di lettura/scrittura di un byte specifico in dati con più di 8 bits |
| ***read_n*** | 1 | Master → Slave | Indica una richiesta di lettura |
| ***readdata*** | 8,16,32,64 128,256, 512, 1024 | Slave → Master | E' il dato che lo slave invia al master in risposta a una richiesta di lettura |
| ***write_n*** | 1 | Master → Slave | Indica una richiesta di scrittura |
| ***writedata*** | 8,16,32,64 128,256, 512, 1024 | Master → Slave | E' il dato che il master invia allo slave per il trasferimento in scrittura. Deve avere la stessa dimensione di *readdata*, se entrambi i segnali sono presenti |
| ***waitrequest*** | 1 | Slave → Master | Uno slave asserisce il *waitrequest* quando non può rispondere a una richiesta di lettura/ scrittura del master, in quanto impegnato in altre . operazioni. In questo modo, lo slave forza il master ad aspettare finché è pronto per iniziare un nuovo trasferimento. |

Tabella 6.1: Segnali Interfaccia Avalon-MM usati nel progetto

### 6.1.2 Timing

In figura 6.1 è riportato il timing diagram definito da Intel/Altera in [7], il quale è stato utilizzato come riferimento per la progettazione della FSM incorporata nel blocco "AVALON_TO_SPI_BLOCK" nel nuovo firmware (la cui descrizione sarà oggetto del capitolo 7).
*Si precisa che il segnale "response" presente in figura, è uno dei possibili segnali dell'interfaccia Avalon-MM, il quale tuttavia non è stato riportato in tabella 6.1 in quanto non utilizzato nel progetto.*

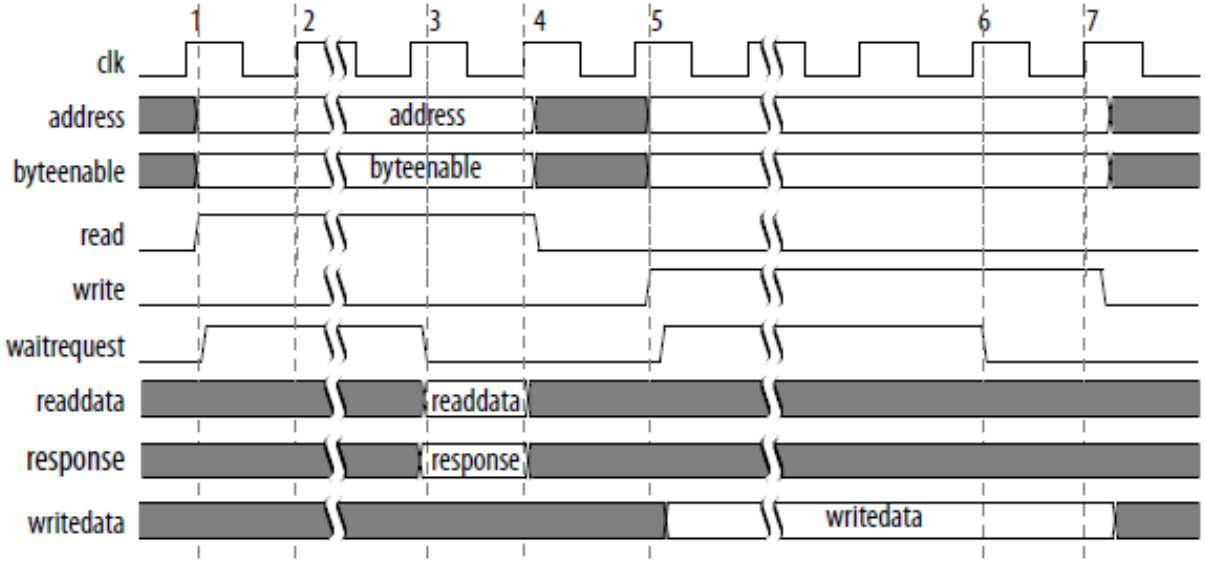


Figura 6.1: Timing diagram presente in [7], riferito a un ciclo di lettura/scrittura con il segnale *Waitrequest*

Il timing sopra riportato costituisce il timing da rispettare nel momento in cui si effettua un'operazione di lettura o scrittura. Nel nostro caso, entrambe le operazioni sono ammesse e, come richiesto dal protocollo Avalon, il segnale *waitrequest* viene utilizzato dallo slave per entrambe le operazioni per ritardare il trasferimento del comando.
Quando il master vuole iniziare un trasferimento, tipicamente invia allo slave i segnali *address*, *byteenable* e *read* o *write*, nel caso voglia effettuare rispettivamente una richiesta di lettura o scrittura. In quest'ultimo caso, deve anche inviare allo slave *writedata*, ovvero il dato che si vuole scrivere.

Nel caso in cui lo slave non abiliti il *waitrequest*, il trasferimento del comando dura un colpo di clock. In caso contrario, invece, il master rende disponibile il comando allo slave finché lo slave disabilita il segnale *waitrequest*. Il trasferimento del comando, quindi, si conclude al fronte di salita del clock successivo alla disabilitazione di *waitrequest* da parte dello slave. Inoltre, quando il *waitrequest* è abilitato, l'*addres*, il *byteenable* e altri segnali di controllo sono mantenuti costanti.

### Ciclo di lettura

Nel caso specifico in cui il master voglia effettuare una lettura da una determinata locazione di memoria, invia allo slave l'*address* che corrisponde alla locazione di memoria da cui si vuole leggere il dato, insieme al *byteenable* per indicare quale/i byte/s vuole leggere e attiva il segnale *read* (attivo col valore logico alto in figura 6.1).
Tuttavia, se il segnale *waitrequest* è attivo (attivo col valore logico alto in figura 6.1), significa che lo slave non può rispondere a tale richiesta del master. Infatti, nella parte a sinistra della figura 6.1, si nota che lo slave restituisce al master il segnale *readdata* contenente il dato richiesto, solo nel momento in cui *waitrequest* viene disabilitato dallo slave stesso. Inoltre, *readdata* rimane disponibile al master per un solo colpo di clock.
Nel caso specifico del progetto, affinché il processore NIOS possa inviare alle periferiche con cui comunica una richiesta di lettura, è necessario utilizzare nel codice C una MACRO specifica definita da Intel/Altera. Essa è definita nel seguente modo:

**IORD_32DIRECT(BASE, OFFSET)**

Si nota che questa MACRO permette di effettuare in modo semplice una richiesta di lettura da una periferica, su un numero di bit specificato (32 nel caso dell'esempio). Il numero di bit supportato dalla MACRO può essere 8, 16 o 32. Nel progetto realizzato, che verrà descritto in dettaglio nel capitolo 7, si è utilizzata la MACRO per la lettura di 16 bit nel caso della periferica "CHX_REG" e 32 bit per la periferica "AVALON_TO_SPI_BLOCK" (riferimento alla figura 7.3).
In riferimento alla definizione della MACRO sopra riportata, è necessario fornirle due parametri:

- ***base***: indica l'indirizzo base della periferica, definito nel progetto "**Qsys**" del sistema.

- ***offset***: indica l'offset rispetto all'indirizzo base della periferica in cui si vuole effettuare una lettura.

In riferimento alla memoria [1] realizzata nel progetto finale per salvare le risposte che il CPLD manda all'FPGA, se si vuole leggere dalla locazione 4, si deve utilizzare la MACRO con i seguenti parametri:

- ***BASE***= indirizzo di base della periferica "AVALON_TO_SPI_BLOCK" definito nel progetto "**Qsys**" (riferimento alla figura 7.4)

- ***OFFSET***=4, per leggere dalla locazione 4 della memoria

### Ciclo di scrittura

Nel caso specifico in cui il master voglia effettuare una richiesta di scrittura in una determinata locazione di memoria, esso invia allo slave l'*address* corrispondente alla locazione di memoria in cui si vuole scrivere il dato, oltre al *byteenable* per indicare in quale/i byte/s si vuole scrivere il dato e al *writedata* (il dato che si vuole scrivere) e attiva il segnale *write* (attivo col valore logico alto in figura 6.1).
Se il segnale *waitrequest* è attivo, il master mantiene il *writedata* disponibile per lo slave finché lo slave disabilita il *waitrequest*.
Infatti, nella parte a destra della figura 6.1, si nota che il master invia allo slave il dato da scrivere *writedata* e questo rimane costante finché *waitrequest* è abilitato dallo slave. Il trasferimento del dato si conclude solo dopo il fronte di salita del clock in cui *waitrequest* viene disabilitato dallo slave.
Anche in questo caso, per permettere a NIOS di inviare una richiesta di scrittura alle periferiche con cui comunica, è necessario utilizzare nel codice C una MACRO specifica definita da Intel/Altera. Essa è definita nel seguente modo:

**IOWR_32DIRECT(BASE, OFFSET, DATA)**

[1] Descritta in dettaglio nel paragrafo 7.3.2

Anche in tal caso si è riportata la MACRO per effettuare una scrittura su 32 bit, ma i formati supportati sono 8, 16, 32 bit. La struttura è la stessa della MACRO per la richiesta di lettura, fatta eccezione per il parametro aggiuntivo che bisogna fornire nel caso di scrittura.
In particolare, si nota che è necessario fornire anche il DATO che si vuole scrivere.
Sempre in riferimento al firmware finale descritto nel capitolo 7, se si vuole inviare un comando al CPLD, NIOS lo deve inviare alla periferica "AVALON_TO_SPI_BLOCK" che è quella con cui comunica. In tal caso, i parametri da fornire sono i seguenti:

- ***BASE***= indirizzo di base della periferica "AVALON_TO_SPI_BLOCK" definito nel progetto "**Qsys**" (riferimento alla figura 7.4)
- ***OFFSET***= 4, se si vuole che la risposta del CPLD al comando inviato venga salvata nella locazione di memoria 4
- ***DATA***: è il comando (su 32 bit) che si vuole inviare al CPLD, la cui struttura è definita nel paragrafo 5.1

## 6.2 ANSI/VITA 4-1995

**ANSI/VITA 4-1995** [8] è uno standard per i cosiddetti "Moduli IP", approvato nel 1996 dall'**American National Standards Institute (ANSI)/VITA Standards Organisation (VSO)**. In particolare, l'ANSI [2] è un ente Americano che si occupa di supervisionare gli standards negli Stati Uniti, mentre il VSO è un ente accreditato dall'ANSI per sviluppare e promuovere gli standards.
Questo standard definisce i "Moduli IP" e le sue specifiche, ovvero dei moduli usati per implementare una vasta gamma di funzioni di I/O, di controllo, funzioni analogiche o digitali. Questi moduli vengono montati su una scheda denominata "Carrier", che permette di connettere il Modulo IP col mondo esterno ed inoltre realizza la conversione dei segnali IP in un'interfaccia compatibile VME. Infatti, un esempio tipico di schede "Carrier" è costituito da schede basate sul VME, che è proprio il caso della scheda "Carrier" usata in questo progetto di tesi. Per questo motivo, la scheda "Carrier" include anche la definizione meccanica, elettrica e logica dello spazio di I/O, dello spazio di memoria, dello spazio di identificazione e funzioni di reset.
Il modulo IP ("Industry Pack") fu creato dalla **GreenSpring Computers** verso la fine degli anni '80, quando il VME era diventato il bus industriale prescelto. Il modulo e le sue specifiche divennero uno standard nel 1996, a seguito della validazione delle specifiche da parte dell'ANSI/VITA.
Come si legge da [8], lo standard ha l'obiettivo di definire alcuni elementi, tra cui:

- L'interfaccia elettrica del connettore logico, includendo i livelli di tensione, la definizione dei pin, i diversi tipi di cicli ammessi, oltre ai timing diagrams specifici per ogni tipo di ciclo e gli state diagrams;
- La dimensione fisica dei Moduli IP;
- I requisiti meccanici ed elettrici della scheda "Carrier";
- Il connettore meccanico degli I/O;
- La minima dimensione della memoria ID ROM;

[2] Per ulteriori informazioni sull'ANSI, si rimanda al sito ufficiale disponibile al seguente link: *Ansi.org*

- L'assegnazione dei pin I/O

Ogni modulo IP comunica con la scheda "Carrier" mediante un connettore a 50 pin che include l'address, il dato, i segnali di controllo e le linee di alimentazione. Essi costituiscono l'interfaccia logica che verrà descritta nel successivo paragrafo.
Inoltre, tutti i trasferimenti di dati tra il Modulo IP e la scheda "Carrier" avvengono in maniera sincrona, come descritto in maniera dettagliata dai timing per i diversi cicli di lettura/scrittura disponibili nel documento che definisce lo standard. Questo permette di realizzare un progetto semplice e affidabile.
Infine, è richiesto che ogni Modulo IP sia dotato di una ID ROM, la quale può essere utilizzata dal software per l'autoconfigurazione. La dimensione minima richiesta per questa ROM è 12 x 8. Essa viene letta dal segnale *IDsel** [3] e viene indirizzata dai 5 bit di indirizzo (*Address* A1 to A5). Essa contiene al suo interno alcune informazioni tra cui il numero del modello del modulo IP, il codice di identificazione del produttore e il numero di bit usati.

[3] Vedasi tabella 6.3

### 6.2.1 Segnali IP

In tabella 6.3 vengono riportati i segnali definiti dallo standard IP, con relativa descrizione e precisazione del numero di pin dedicato a ciascun segnale. *Tuttavia, per semplificare la trattazione, vengono mostrati solo i segnali IP che sono utilizzati nel progetto, mentre per la trattazione completa di tutti i segnali ammessi dallo standard, si rimanda a [8].*

*Si specifica, inoltre, che il simbolo "*" significa che il corrispondente segnale è attivo col valore logico basso.*

*Infine, durante l'indirizzamento in memoria, l'indirizzo è complessivamente composto da 22 bit, di cui i 16 più significativi sono formati dai 16 bit del Data Bus, mentre i 6 meno significativi sono composti dai 6 bit dell'Address.*

Questo è il motivo per cui l'indirizzo del blocco "MEM_DECODER" mostrato in figura 4.2 è formato da 22 bit.

La tabella 6.2, invece, riporta le linee di alimentazione definite dallo standard. In particolare, i 5V costituiscono l'alimentazione principale per le funzioni logiche digitali sul Modulo IP. Sono dedicati 2 pin che forniscono fino a un massimo di 2A di corrente.

Le alimentazioni di ± 12 V, invece, sono principalmente usate per alimentare le funzioni analogiche sui Moduli IP. E' disponibile un pin per ogni alimentazione, ognuno dei quali fornisce un massimo di corrente di 1A.

| **alimentazione** | **Nome** | **No of Pins** |
|---|---|---|
| ***Ground*** | GND | 4 |
| ***+5 Volt*** | +5V | 2 |
| ***± 12 Volt*** | ± 12 V | 2 |

Tabella 6.2: Alimentazioni IP

| Funzione | Nome | No of Pins | Descrizione |
|---|---|---|---|
| ***Data Bus*** | D00..D15 | 16 | Usato per leggere/scrivere dati tra la scheda Carrier e il modulo IP. |
| ***Address*** | A1..A6 | 6/22 | E' usato per indirizzare le locazioni di memoria. Solo la scheda Carrier può pilotare le linee di *Address*. |
| ***Reset*** | Reset* | 1 | E' pilotato dalla scheda Carrier per resettare il circuito a uno stato noto. |
| ***Memory Select*** | MemSel* | 1 | E' una delle 4 linee pilotate dalla scheda Carrier per abilitare il Modulo IP. E' usato per i cicli di lettura/scrittura in memoria. |
| ***Clock*** | CLK | 1 | E' il clock a 8MHz o 32MHz,con DC=50%, pilolato dalla scheda Carrier. |
| ***Module Id.*** | IDSel* | 1 | E' una delle 4 linee pilotate dalla scheda Carrier per per abilitare il Modulo IP. E' usato per leggere la ROM che contiene le informazioni di identificazione del Modulo IP. |
| ***Data Direction*** | R/W* | 1 | E' pilotato dalla scheda Carrier per indicare la direzione del flusso delle linee di dato. |
| ***Data Ack.*** | Ack* | 1 | E' asserito dal Modulo IP per terminare il trasferimento di ogni dato. |

Tabella 6.3: Segnali IP usati nel progetto

### 6.2.2 Timing

Nelle figure 6.2 e 6.3 vengono riportati i timing diagrams riferiti rispettivamente ad un ciclo di lettura e un ciclo di scrittura in memoria, definiti da [8]. Essi verranno utilizzati come riferimento per la progettazione della FSM incorporata all'interno del blocco IP_BRIDGE nel nuovo firmware (la cui trattazione sarà affrontata nel capitolo 7).

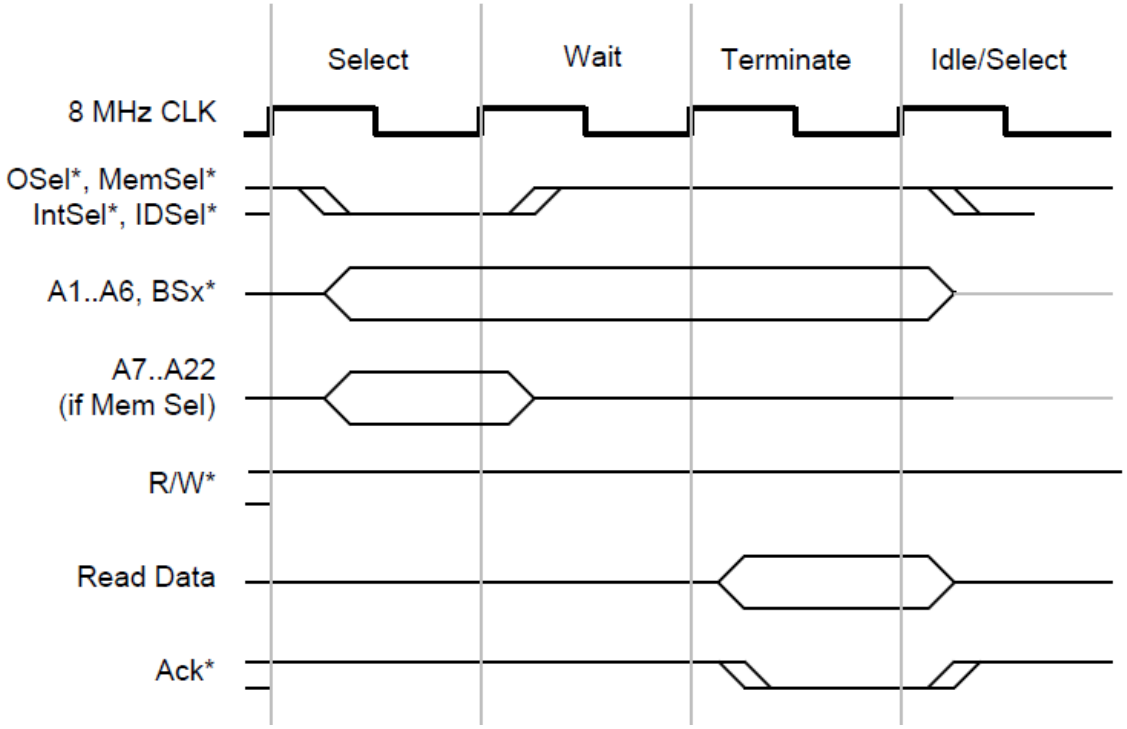


Figura 6.2: Timing diagram presente in [8], riferito a un ciclo di lettura con lo stato *Wait*

*Si precisa che gli stati di Wait, Hold e Idle sono opzionali, mentre gli stati di Select e Terminate devono essere sempre presenti per tutti i trasferimenti di dati.*
Un Modulo IP può ritardare un trasferimento asserendo *Ack**='1'.
Nel caso specifico di questo progetto, si è deciso di utilizzare solo lo stato di *Wait*, mentre non è presente lo stato di *Hold*.
Come si evince dalle figure 6.2 e 6.3, i cicli di lettura e scrittura in memoria utilizzano

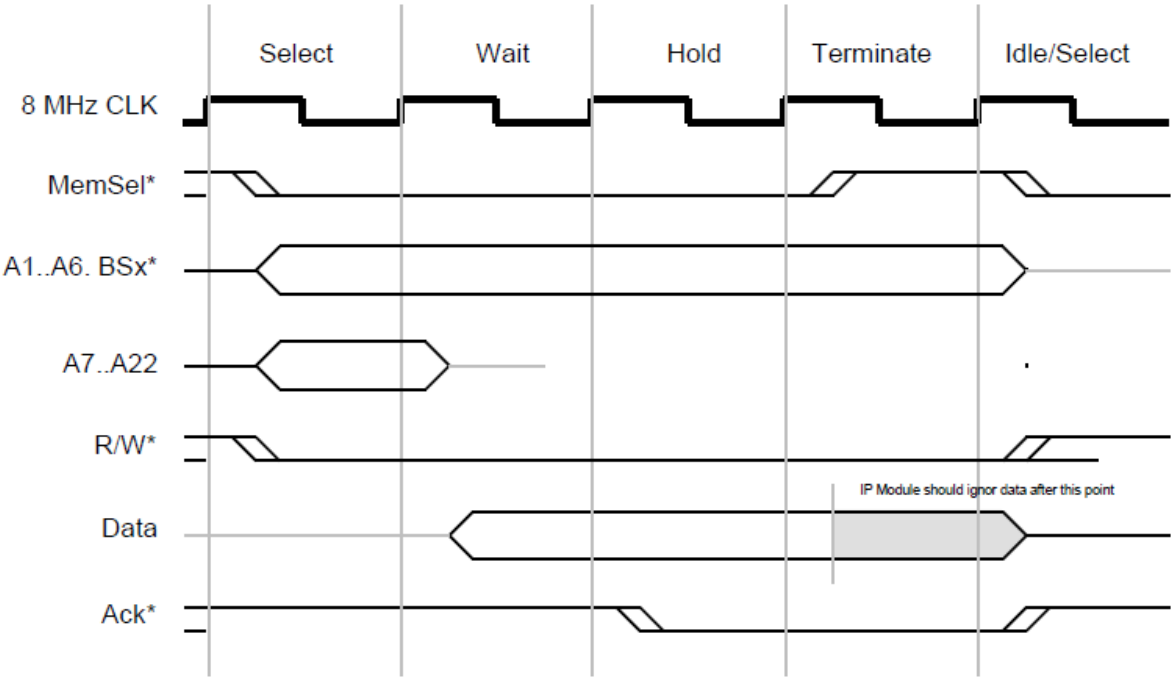


Figura 6.3: Timing diagram presente in [8], riferito a un ciclo di scrittura in memoria con lo stato *Wait* e *Hold*

il *data_bus* per trasmettere i 16 bit più significativi dell'indirizzo (complessivamente a 22 bit come specificato nel paragrafo precedente).

### Ciclo di lettura

Nel caso di ciclo di lettura, il *data_bus* contiene il valore letto dall'opportuna locazione di memoria indirizzata dall'*address* solo nel momento in cui il Modulo IP abilita il segnale *Ack** nello stato *Terminate*. Quindi, il *data_bus* si trova inizialmente in alta impedenza, successivamente contiene i 16 bit più significativi dell'*address* per un colpo di clock nello stato *Select* e torna in alta impedenza fino a che il Modulo IP abilita il segnale *Ack**. A tal punto, *data_bus* contiene il dato letto dall'opportuna locazione di memoria e il dato rimane valido per un colpo di clock nello stato *Terminate*. Il ciclo di lettura è concluso e dunque il data_bus ritorna in alta impedenza.

### Ciclo di scrittura

Anche nel caso di ciclo di scrittura in memoria, il *data_bus* è inizializzato in alta impedenza e contribuisce a formare i 16 bit più significativi dell'*address* per un colpo di clock nello stato *Select*. Al colpo di clock successivo, il *data_bus* cambia il suo valore, che in questo momento contiene il dato che si vuole scrivere in memoria. Il dato da scrivere rimane valido sul *data_bus* fino a quando il Modulo IP abilita e disabilita di nuovo il segnale *Ack**, come mostrato in figura 6.3.
*Si specifica che il timing diagram raffigurato in [8], è generico per un ciclo di scrittura in memoria che contiene tutti gli stati. Nel caso specifico, non si considera quindi lo stato Hold, dato che non è stato inserito nel progetto.*

# CAPITOLO 7

# Struttura firmware finale

Come anticipato nel capitolo "Introduzione", il firmware originale dell'FPGA, la cui struttura è stata illustrata nel capitolo 4, presentava problemi di eccessiva complessità e difficile manutenibilità. Nello specifico, risultava essere eccessivamente complesso per il solo scopo di essere in grado di svolgere misurazioni in DC e in AC dei segnali elettrici. Per questo motivo, si è deciso di realizzare un nuovo firmware, in grado di gestire le misure di segnali elettrici in DC per i fini della tesi. I requisiti che deve soddisfare sono principalmente quelli di essere *semplice, chiaro ed essenziale*, oltre a volgere il compito richiesto.
Si è dunque passati dalla struttura illustrata in figura 7.2 alla struttura illustrata in figura 7.3.
Come si può notare da un confronto tra i due schemi a blocchi, la struttura finale dell'FPGA è composta da un minor numero di blocchi e di memorie allocate e contiene solo 3 FSM (che verranno descritte nel seguito in paragrafi dedicati). In particolare, la FSM all'interno di "IP_BLOCK" si occupa di gestire l'interfaccia con l'esterno (segnali IP), quella all'interno di "AVALON_TO_SPI_BLOCK" gestisce, invece, la comunicazione col processore NIOS e, infine, quella all'interno di "SPI_HANDLER" trasmette al blocco "SPI_INTERFACE" il comando che arriva da NIOS.
Infine, si è deciso di spostare la complessità della generazione dei comandi per il CPLD al solo processore NIOS (nel codice originale i comandi di inizializzazione erano gestiti da NIOS, mentre tutti i comandi successivi venivano creati da appositi blocchetti VHDL dentro "HV_MODULE_TOP").

Per quanto riguarda, invece, il firmwre del CPLD, si è mantenuto quello la cui struttura è illustrata nel capitolo 4.2, che rimane quindi quella di riferimento. Per quanto concerne la struttura dell'FPGA, dalla figura 7.3, si evince che per semplificare la trattazione, l'organizzazione di alcuni blocchi si è mantenuta inalterata rispetto alla versione originale (figura 7.2), ovvero il blocco denominato "IP_BLOCK" ed alcuni blocchi presenti al suo interno. Tuttavia, la struttura interna di ciascun blocco si è riprogettata, al fine di svolgere il compito richiesto.
Il progetto del nuovo sistema basato sul processore NIOS è stato suddiviso in tre fasi, come brevemente descritto nel paragrafo 4.1.5:

1. progetto "**Qsys**" [10]
2. progetto "**Quartus II**" scritto in VHDL;
3. sviluppo del firmware per NIOS II [13]: codice C nell'IDE "NIOS II SBT per Eclipse".

Lo schema riassuntivo per lo sviluppo del progetto è illustrato in figura 7.1.

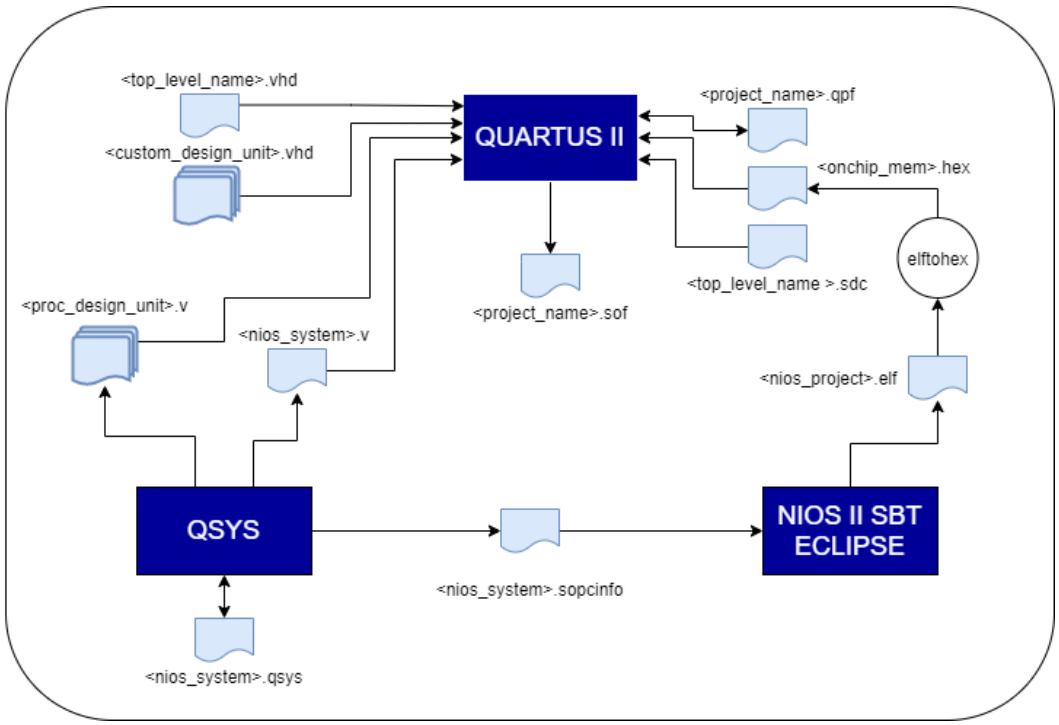


Figura 7.1: Fasi del progetto

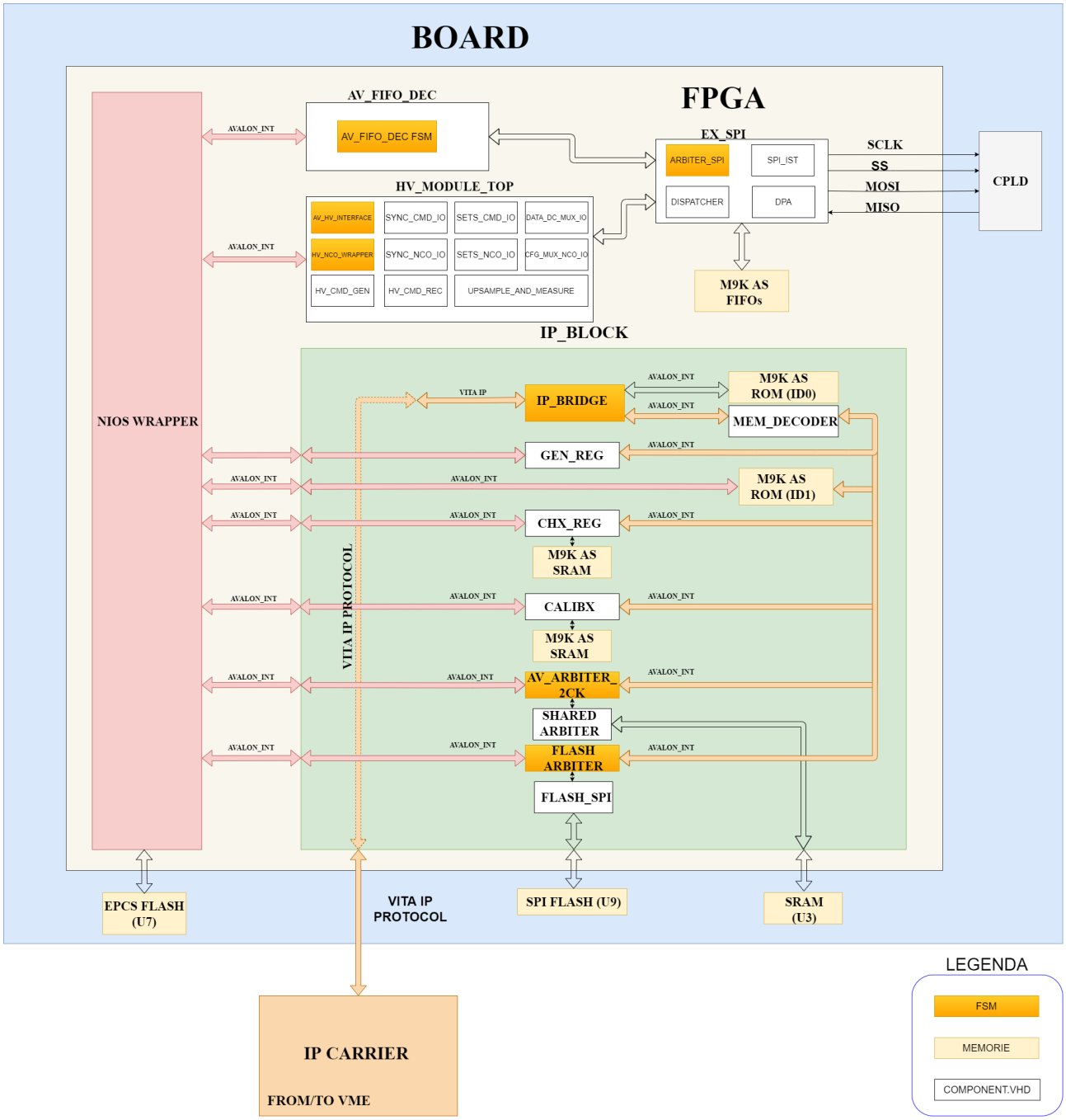


Figura 7.2: Struttura originale dell'FPGA

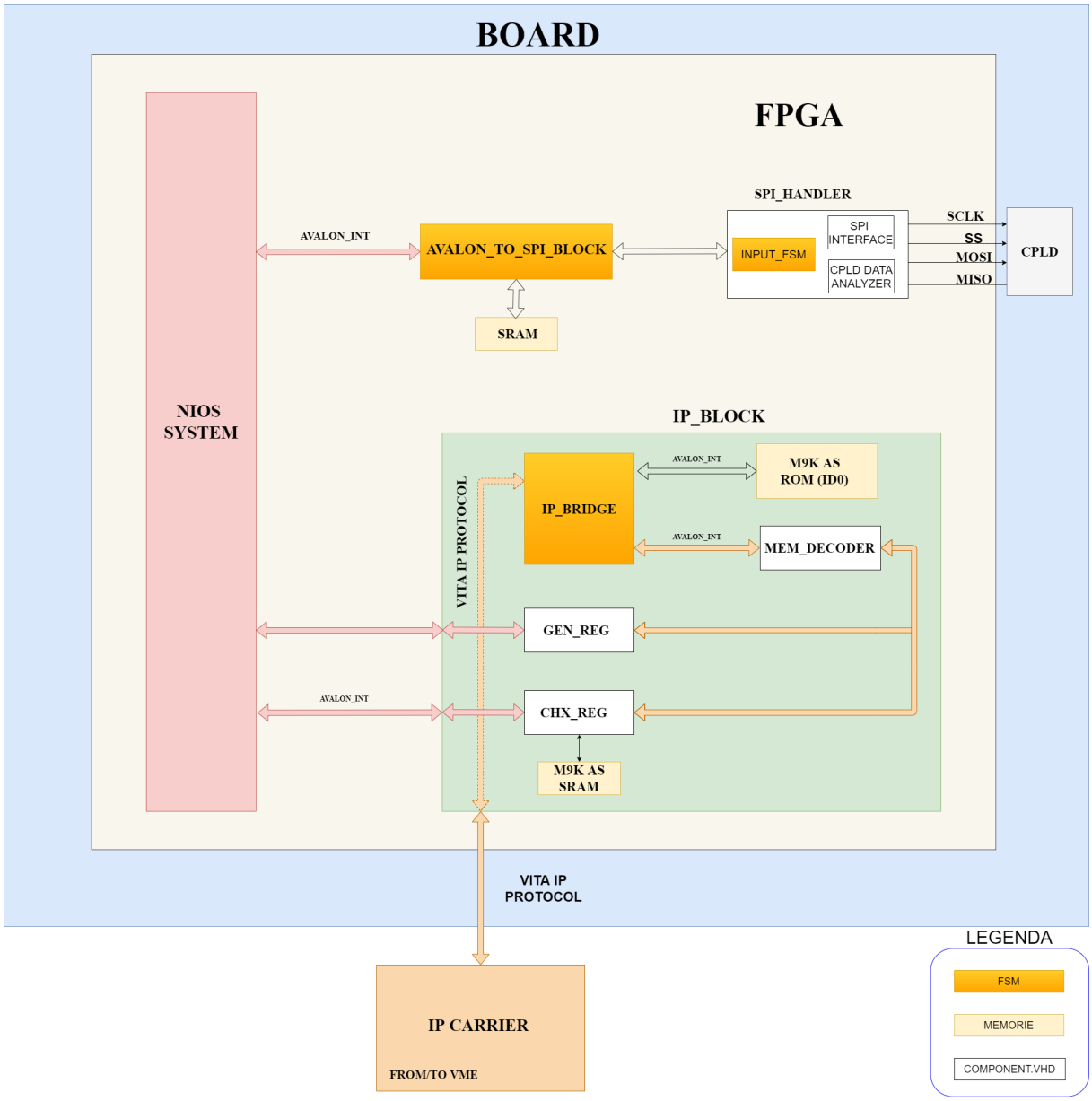


Figura 7.3: Struttura finale dell'FPGA

## 7.1 Progetto Qsys

Per realizzare il progetto dell'hardware di questo sistema basato sul processore NIOS, si è utilizzato il tool **"Qsys"** messo a disposizione da Intel/Altera all'interno del software **"Quartus II"** (nel caso specifico di questo lavoro di tesi si è utilizzato "**Quartus II 13.1 Web Edition**").
Nel momento in cui si crea un nuovo progetto su "**Quartus II**" selezionando l'FPGA d'interesse (nel caso specifico, Cyclone III EP3C10F256I7N) e includendo tutti i files *.vhd* di tutti i componenti inclusi nel progetto, "**Quartus II**" genera diversi files tra cui uno con estensione *.qsys* che il tool "**Qsys**" si occupa di aggiungere al progetto che si sta creando.
**"Qsys"** [11] è uno strumento che, tramite interfaccia grafica, permette di selezionare tutti i componenti che si vogliono utilizzare per realizzare il sistema e genera automaticamente il sistema hardware che connette tutti i componenti tra loro. Nel caso in esame, il sistema è composto dal processore NIOS, da periferiche standard e personalizzate e da periferiche di input/output.
In figura 7.4 è mostrato lo schema **"Qsys"** del progetto realizzato ai fini della tesi. **"Qsys"** permette di selezionare i componenti standard dal menu visibile a sinistra in figura 7.4. Tuttavia, nel caso in cui il sistema includa periferiche personalizzate con cui il processore NIOS deve comunicare, è anche possibile creare un nuovo componente, con una delle tipologie di interfaccia Avalon. Nel caso specifico, si è realizzato un nuovo componente [12] denominato "CUSTOM_PERIPHERALS" (visibile nel menu a sinistra in figura) con interfaccia Memory Mapped, unico tipo di interfaccia Avalon presente nel progetto (fatta eccezione per il clock e il reset).
I blocchi denominati "NIOS_TO_SPI_BLOCK" E "NIOS_TO_CHXREG" in figura 7.4, rappresentano i blocchetti denominati rispettivamente
"AVALON_TO_SPI_BLOCK" e "CHX_REG" illustrati in figura 7.3, i quali comunicano col processore NIOS. Tutti gli altri blocchetti presenti in figura 7.4, invece, fanno parte dei blocchi standard selezionabili dalla libreria. Tra questi, quelli denominati "PIO (Parallel Input Output)", sono periferiche di input/output, utilizzati come input per ricevere i segnali di alimentazione direttamente collegati all'FPGA e come output per pilotare, ad esempio, i LEDs presenti sulla scheda.

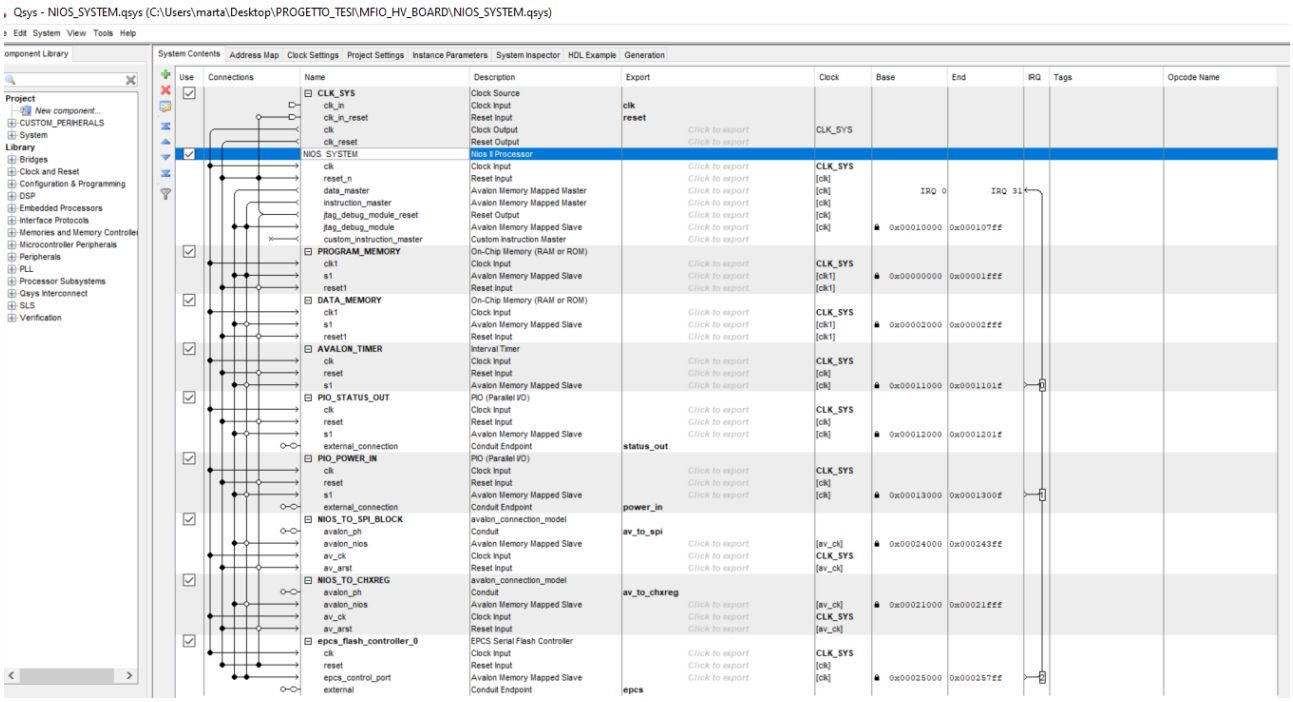


Figura 7.4: Qsys: sistema realizzato

I due blocchetti di memoria “On-Chip Memory (RAM or ROM)”, invece, sono utilizzati per salvare i dati di configurazione dell’FPGA e i dati di programmazione di NIOS, nel momento in cui viene caricato il firmware.

Infine, i tre blocchetti rimanenti rappresentano il clock, il processore NIOS e l’EPCS Serial Flash Controller. Quest’ultimo, è utilizzato per caricare il firmware in una EPCS flash integrata su scheda, in modo che l’FPGA non debba essere riprogrammata ogni qual volta si spenga l’alimentazione.

*Si specifica che il progetto è stato predisposto per poter caricare sia il codice di programmazione dell’hardware sia il firmware per NIOS nell’EPCS flash esterna presente sulla scheda MFIO HV. Tuttavia, questa procedura richiede il possesso della licenza IP Intel/Altera. Ai fine della tesi, durante le prove sperimentali, si è testato il progetto caricandolo nella on-chip RAM e non nella EPCS flash esterna.*

Dalla figura 7.4 si nota, inoltre, che ad ogni periferica è associato uno spazio di indirizzamento. Questa informazione è stata utilizzata durante la generazione del firmware di NIOS, che verrà illustrato in un paragrafo dedicato. Infatti, se si vuole gestire la comunicazione tra NIOS e una specifica periferica, è necessario comunicare a NIOS l’indirizzo di base della periferica desiderata.

Infine, è anche possibile notare che il tool **"Qsys"** permette di definire i collegamenti tra i segnali delle diverse periferiche in maniera grafica, visibile a sinistra dei blocchetti selezionati in figura. Nello specifico, il pallino nero indica la presenza di una connessione, mentre il pallino bianco indica assenza di connessione tra i corrispondenti segnali.
Dopo aver definito la struttura dell'intero sistema basato su NIOS, **"Qsys"** è in grado di generare l'HDL del sistema. Questo è stato poi incluso come componente nella top entity del progetto **"Quartus II"**.
In riferimento alla figura 7.3, il blocco denominato "NIOS_SYSTEM" rappresenta proprio il sistema basato su NIOS, generato tramite il tool **"Qsys"**.
Tra i files generati da "Qsys", quello con estensione *.sopcinfo* è di particolare interesse.
Si tratta di un file di testo ASCII che contiene tutte le informazioni su ogni modulo istanziato nel progetto, sui parametri impostati e sullo spazio di indirizzamento di ogni blocco, oltre alle informazioni su tutte le connessioni tra i diversi blocchi. Questo file permette di sviluppare il firmware per il processore NIOS nella fase successiva del flusso di progetto, in quanto ogni progetto firmware per NIOS necessita della descrizione del corrispondente sistema hardware basato su NIOS.

## 7.2 Progetto Quartus II

La struttura finale dell'FPGA è riportata in figura 7.3.
I principali macro blocchi sono quattro:

1. **AVALON_TO_SPI_BLOCK**
2. **SPI_HANDLER**
3. **IP_BLOCK**
4. **NIOS_SYSTEM**

## 7.3 AVALON_TO_SPI_BLOCK

### 7.3.1 Entity VHDL

```
entity AVALON_TO_SPI_BLOCK is
port (
  clock: in std_logic;
  reset: in std_logic;
--Avalon interface with NIOS II --
  av_chipselect   : in std_logic;
  av_address      : in std_logic_vector(7 downto 0);
  av_byteenable   : in std_logic_vector(3 downto 0);
  av_read_n       : in std_logic;
  av_readdata     : out std_logic_vector(31 downto 0);
  av_write_n      : in std_logic;
  av_writedata    : in std_logic_vector(31 downto 0);
  av_waitrequest  : out std_logic;
  --From/to SPI_HANDLER--
  spi_data_read   : in std_logic_vector(31 downto 0);
  spi_data_ready  : in std_logic;
  spi_data_write  : out std_logic_vector(31 downto 0);
  data_valid      : out std_logic;
  cs_alignment    : out std_logic_vector(1 downto 0)
);
end entity AVALON_TO_SPI_BLOCK;
```

### 7.3.2 Descrizione del blocco

Il blocco **AVALON_TO_SPI_BLOCK** comunica mediante interfaccia *Avalon* col processore NIOS, mentre scambia col blocco "SPI_HANDLER" il comando da inoltrare al CPLD e il dato ricevuto dal CPLD in risposta alla richiesta effettuata.
In particolare, i dati che arrivano dal CPLD vengono salvati all'interno di una memoria RAM contenente 48 locazioni da 16 bit ciascuna, indirizzabile mediante l'*address*

generato da NIOS. L'indirizzamento per ogni comando è stato gestito nel codice C, inserendo l'opportuno offset in base al canale e al comando inviato.
A titolo di esempio, si riporta in tabella 7.1 la struttura della memoria riferita ad un singolo canale, mentre quella completa contiene tale struttura replicata per sei.

| DC_val | AC_val | f_val | DAC_value | GPI_power_state | GPO_command | Not used | Not used |
|---|---|---|---|---|---|---|---|

Tabella 7.1: Struttura RAM contenente i dati provenienti dal CPLD

*Si precisa che le locazioni assegnate ad ogni tipologia di dato sono state scelte in modo arbitrario. Inoltre, la RAM è stata già predisposta per supportare eventualmente anche le misurazioni in AC e frequenza, mentre due locazioni sono state predisposte per eventuali usi futuri.*
Ad esempio, se si vuole effettuare una lettura della tensione in DC relativa al canale 2, sarà innanzi tutto necessario inviare il comando con tale richiesta. In risposta, il CPLD manderà il dato binario generato dall'ADC collegato al canale 2, il quale verrà salvato nella seconda riga di tale memoria nella locazione corrispondente alla dicitura "DC_val". A tal punto, NIOS manderà una richiesta di lettura alla specifica locazione di memoria, corrispondente alla "seconda riga/prima colonna" (in quanto la seconda riga contiene i dati relativi al secondo canale) di tale memoria e leggerà infine il valore desiderato che arriva a NIOS nel *readdata*.
Oltre alla memoria, il blocco **AVALON_TO_SPI_BLOCK** contiene al suo interno due registri per salvare il comando (*writedata*) e l'*address* che arrivano da NIOS. Contiene inoltre un registro utilizzato per allineare la risposta del CPLD nel caso in cui essa sia traslata di 1 o 2 colpi di clock. A tal fine, nella fase di inizializzazione della scheda, NIOS manda un *comando_GPI* per la lettura delle alimentazioni, comando per il quale è prevista una risposta attesa corrispondente al pattern "00100101", dove i 4 bit meno significativi rappresentano i segnali di stato delle alimentazioni (lo '0' è previsto per le alimentazioni negative -12V e -60V, l'1' per le alimentazioni positive +12V e +60V). Nelle misurazioni sperimentali effettuate, si è constatato che la risposta del CPLD è traslata di 1 colpo di clock (in media 9 volte su 10) o di 2 colpi di clock (in media 1 volta su 10). Per questo motivo, durante la procedura di inizializzazione, NIOS analizza la risposta che riceve dal CPLD e se è disallineata

provvede ad inviare al blocco **AVALON_TO_SPI_BLOCK** l'opportuna correzione di allineamento. Essa viene poi passata al blocco "SPI_HANDLER", che la inoltra a sua volta al blocco "CPLD_DATA_ANALYZER" che riceve il dato dal CPLD.
Infine, il blocco **AVALON_TO_SPI_BLOCK** contiene un contatore a 10 bit utilizzato per riportare la FSM nello stato noto di "IDLE", nel caso in cui non dovesse mai arrivare il segnale *data_ready*. Questo segnale viene generato dal blocco "SPI_INTERFACE" nel momento in cui viene ricevuto il dato dal CPLD. Il CPLD restituisce all'FPGA sempre una risposta, anche nel caso in cui il comando inviatogli sia una scrittura. In tal caso, infatti, il CPLD restituisce all'FPGA lo stesso comando che ha ricevuto. Pertanto, se il segnale *data_ready* non viene mai abilitato, significa che si è verificato un malfunzionamento del sistema.
Il contatore, quindi, viene utilizzato per rilevare anomalie di funzionamento del sistema e riportare la macchina nello stato di *IDLE*, in cui si attende un nuovo comando valido.
In figura 7.5 si riporta la FSM realizzata per la gestione della ricezione di un nuovo comando proveniente da NIOS e per il salvataggio dei dati ricevuti dal CPLD nella RAM. Tale FSM, garantisce il rispetto del timing richiesto dall'interfaccia Avalon per la scrittura e la lettura di un dato, illustrato in figura 6.1.
Nello specifico, il blocco **AVALON_TO_SPI_BLOCK** rappresenta in questo caso lo slave del processore NIOS. NIOS può inviare un nuovo comando quando il segnale *waitrequest*[1] è attivo, ma le specifiche sul timing definite in [7], prevedono che NIOS renda disponibile il *writedata* allo slave finché esso disattiva il *waitrequest*. Quando lo slave disattiva il *waitrequest*, significa che ha letto il dato e NIOS può dunque concludere il trasferimento.
In riferimento alla figura 7.5, NIOS II può dunque iniziare il trasferimento di un nuovo comando nello stato di *IDLE*. Insieme al segnale di *writedata* che costituisce il comando che l'FPGA deve inviare al CPLD, NIOS invia al blocco
**AVALON_TO_SPI_BLOCK** anche il chipselect e il segnale *write_n='0'*.
Come si nota dalla figura 7.5, quando arriva chipselect='1' si effettua una verifica sul tipo di operazione richiesta (scrittura di un comando o lettura di un dato) e si evolve negli stati.

[1]per il suo significato, si rimanda alla tabella 6.1

Nel caso arrivi una richiesta di scrittura di un comando, si salvano in due appositi registri l'*address* e il *writedata* nello stato denominato *LOAD_COMMAND*. Essi saranno disponibili in uscita dai rispettivi registri nello stato *COMMAND_AVAILABLE* e rimangono disponibili per il blocco fino al salvataggio di un nuovo comando, che si verifica quando la FSM ritorna nello stato *LOAD_COMMAND*.
A questo punto, è necessario effettuare il controllo sul comando per verificare se corrisponde o meno al comando di allineamento del dato per il CPLD. Questo comando infatti è fondamentale per una corretta comunicazione tra FPGA e CPLD.
Pertanto, nel caso in cui il comando sia quello di allineamento, si carica nell'apposito registro il valore di allineamento (che NIOS deduce in base alla risposta che riceve dal CPLD per il *comando_GPI*) nello stato *LOAD_MAN_REG* e si prosegue nello stato *END_ALIGNMENT_COMMAND*, in cui si disattiva il *waitrequest* per comunicare a NIOS che si è letto il comando. A questo punto NIOS conclude il trasferimento e si ritorna in "IDLE", in attesa di un nuovo comando.
Nel caso in cui, invece, il comando non corrisponda al comando di allineamento, si prosegue nello stato *END_COMMAND_TRANSFER*, nel quale viene disattivato il *waitrequest* e pertanto NIOS può concludere il trasferimento del comando. In questo stato viene inoltre attivato il segnale *data_valid*, il quale viene inoltrato al blocco "SPI_INTERFACE" tramite il blocco "SPI_HANDLER" per indicare la presenza di un nuovo comando valido da inoltrare al CPLD.
Si procede, poi, nello stato *WAIT_FOR_END_COMMAND*, in cui si riattiva il *waitrequest* e da questo momento NIOS può quindi inviare un nuovo comando.
Si attende in questo stato finché arriva il segnale *dataready='1'*, il quale indica la presenza di un dato valido arrivato dal CPLD. Se questo segnale è pari a '0', si verifica che il contatore di controllo non sia giunto al termine del conteggio (1023 nel caso di contatore a 10 bit). Infatti, se si verifica questa condizione, significa che c'è stato un malfunzionamento del sistema e dunque si torna nello stato di *IDLE* in cui si attende un nuovo comando.
Nel momento in cui arriva il segnale *dataready='1'*, si prosegue nello stato *SAVE_CPLD_RESPONSE*, nel quale si salva in memoria la risposta del CPLD al comando ricevuto. La locazione di memoria è definita in tabella 7.1 e la corretta locazione viene decisa da NIOS in base al comando e al canale. Successivamente allo

stato *SAVE_CPLD_RESPONSE*, si ritorna in *IDLE* in attesa di un nuovo comando. Invece, nel caso in cui NIOS invii una richiesta di lettura, si procede nello stato *SAVE_READ_ADDRESS*, nel quale si salva l'*address* che indica la locazione di memoria da cui effettuare la lettura. Si procede poi nello stato *READ_CPLD_RESPONSE* in cui si attiva il *RD_enable* della memoria. Avendo realizzato una memoria sincrona sia in scrittura che in lettura, il dato sarà disponibile al colpo di clock successivo, nello stato *CPLD_RESPONSE_AVAILABLE*. In questo stato si disattiva quindi il *waitrequest* e si invia a NIOS il dato richiesto nel segnale *readdata*, in modo da rispettare il timing mostrato in figura 6.1. Infine, si ritorna nello stato di *IDLE* per aspettare un nuovo comando.

Per fare un esempio di funzionamento della FSM, si consideri il caso in cui da NIOS arrivi un *comando_ADC*, relativo al canale 3 della scheda. Tale comando, come spiegato nel paragrafo 5.1, indica la richiesta di lettura di una tensione dal canale 3 della scheda. Il comando arriva come *writedata* e pertanto corrisponde a una richiesta di scrittura (*write_n='0'*). La FSM evolverà quindi nel ramo di sinistra, in riferimento alla figura 7.5. Per quanto riguarda l'*address*, esso deve corrispondere alla locazione di memoria 16, in quanto il valore di tensione inviato dal CPLD all'FPGA dovrà essere salvato in questa locazione in base a quanto definito in tabella 7.1.

Dato che il comando ADC ha come risposta il valore di tensione sul canale indicato, canale 3 in questo esempio, dopo aver mandato il *comando_ADC*, NIOS manderà un comando di lettura, inviando *read_n='0'*, oltre all'*address*=16. Infatti, il dato richiesto è stato precedentemente salvato proprio in questa locazione. La FSM evolverà quindi nel ramo di destra (in riferimento alla figura 7.5) e si restituirà a NIOS il dato richiesto tramite il segnale *readdata*.

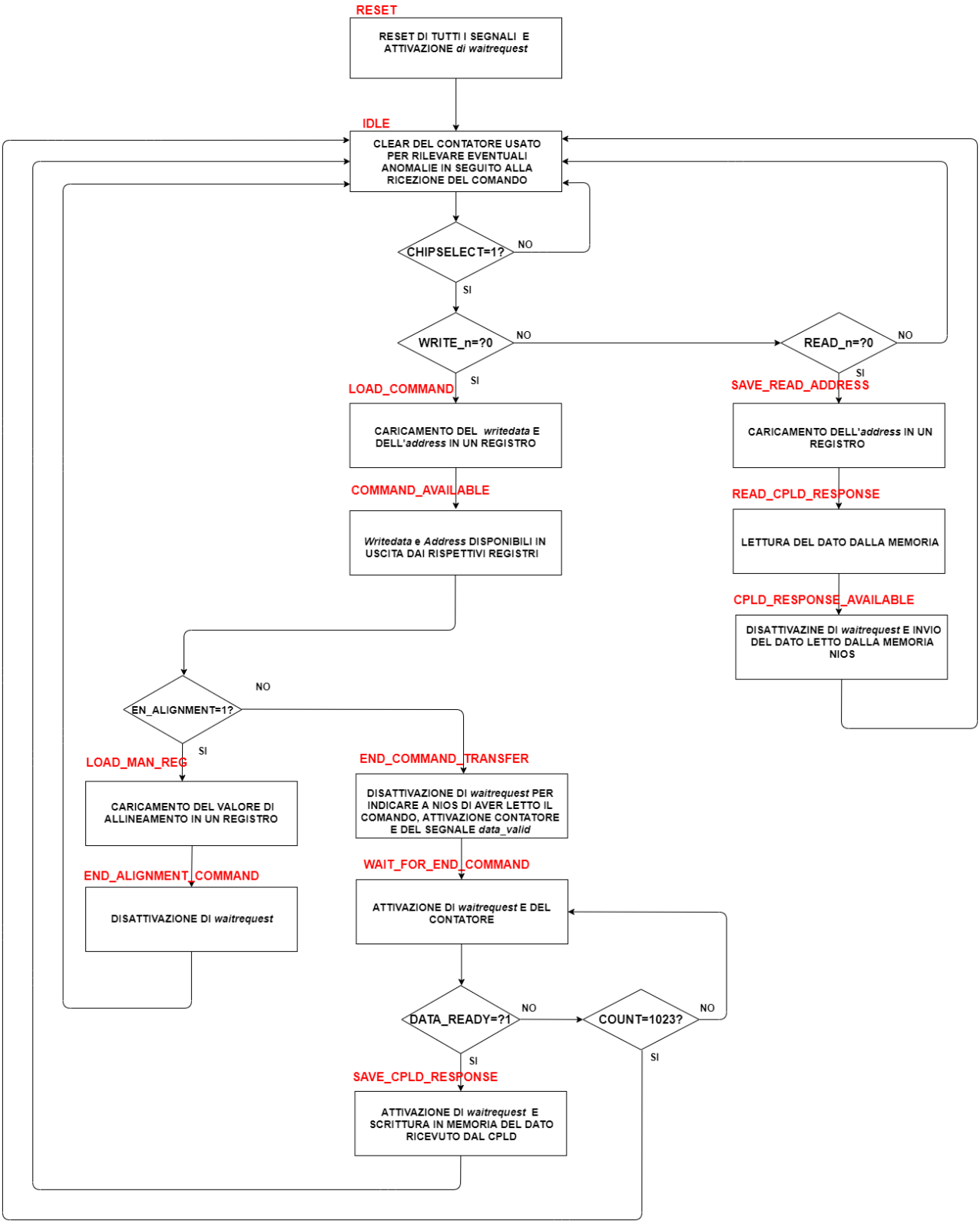


Figura 7.5: FSM AVALON_TO_SPI_BLOCK

### 7.3.3 Simulazioni

Al fine di verificare il corretto comportamento di questo blocco, si è utilizzato il software "Modelsim" per simulare il firmware realizzato. A tal fine, si è realizzato un testbench in grado di testare i 5 possibili comandi previsti per il CPLD, oltre al comando di allineamento utile all'FPGA per ricevere in maniera corretta il dato inviatole dal CPLD.
Poiché l'obiettivo di questo paragrafo è dimostrare il corretto funzionamento della macchina a stati realizzata per questo blocco, si è riportano nel seguito le simulazioni riferite ai seguenti casi:

1. **Scrittura di un comando**;
2. **Lettura di un dato**;
3. **Scrittura del comando di allineamento**.

Essi, infatti, mostrano le 3 possibili evoluzioni all'interno della FSM.

1. **Scrittura di un comando**.
   In tal caso si è scelto di utilizzare il *comando_GPI* caratterizzato dal pattern binario su 32 bit "00000001100011000000000000000000"= 0x018C0000.
   *Per semplificare la trattazione, nelle simulazioni è stato selezionato il valore in esadecimale.*
   In figura 7.6 si riporta la parte iniziale della simulazione per la scrittura di questo comando. Come si può notare dalla figura, nel momento in cui viene disattivato il segnale di *reset* la FSM evolve nello stato di *IDLE* e successivamente, dato che *chipselect='1'* e *write_n='0'*, si procede negli stati *LOAD_COMMAND*, *COMMAND_AVAILABLE*, *END_COMMAND_TRANSFER* e *WAIT_FOR_END_COMMAND*. Infatti, dal momento che si tratta di una scrittura di un comando da inviare al CPLD, la FSM evolve nel ramo di sinistra (riferimento alla figura 7.5). Inoltre, a titolo di esempio, si è selezionato l'*address=7*. Nella successiva simulazione si dimostrerà che la risposta del CPLD verrà salvata in memoria proprio nella locazione numero 7, dalla quale NIOS potrà

successivamente leggere il dato.

Infine, si nota che nel momento in cui si giunge nello stato *WAIT_FOR_END_COMMAND*, viene attivato l'enable del contatore a 10 bit, che inizia dunque a contare come mostrato nell'ultima riga in figura. *In caso di corretto funzionamento del sistema, esso non deve mai raggiungere la condizione count=1023.*

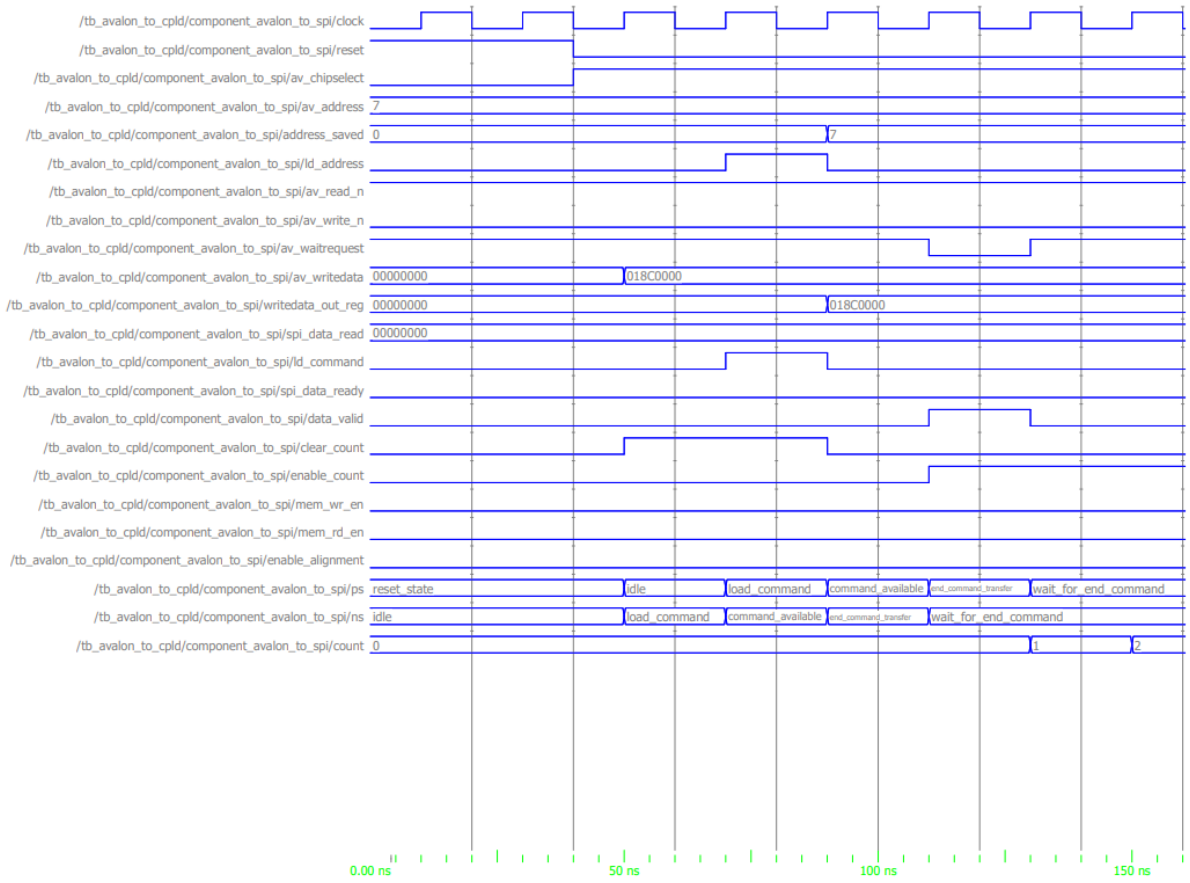


Figura 7.6: Simulazione AVALON_TO_SPI_BLOCK: parte iniziale scrittura comando

In figura 7.7 si riporta la parte finale della simulazione per la scrittura del comando GPI. Dalla figura si evince che la FSM attende nello stato *WAIT_FOR_END_COMMAND* fino all'arrivo del segnale *data_ready='1'*, che indica la presenza di un nuovo dato valido arrivato dal CPLD. Nel momento in cui *data_ready='1'*, si evolve nello stato *SAVE_CPLD_RESPONSE*, in cui si abilita il *wr_en* della memoria e si salva nella locazione numero 7 il dato inviato dal CPLD , denominato *indata* in figura 7.7.


Figura 7.7: Simulazione AVALON_TO_SPI_BLOCK: parte finale scrittura comando

Poiché il dato inviato dal CPLD all'FPGA è su 16 bit, esso corrisponde al valore esadecimale 0xFF25. Tuttavia, solo gli 8 bit LSB sono quelli significativi, in quanto contengono lo stato delle alimentazioni e corrispondono al pattern binario 00100101.

Dopo aver salvato il dato in memoria, si ritorna nello stato di *IDLE* dove viene anche resettato il contatore a 10 bit.

Si nota che in caso di corretto funzionamento esso arriva a contare fino a quando arriva il segnale *data_ready='1'* e non raggiunge la condizione count=1023, fatto che dimostra il corretto funzionamento del sistema.

Infine, per concludere le simulazioni riguardanti la scrittura di un comando e il salvataggio della risposta del CPLD, si riporta in figura 7.8 il contenuto della memoria nello stato di *IDLE*. Infatti, nello stato *SAVE_CPLD_RESPONSE* si abilita il *wr_en* della memoria e trattandosi di una memoria sincrona sia

in scrittura che in lettura, il dato viene memorizzato al successivo colpo di clock, quando la FSM si trova nello stato di *IDLE*. La risposta del CPLD al *comando_GPI* è cerchiata in rosso in figura ed è salvata nella locazione 7 della memoria, dato che in questo esempio si era inviato l'address=7 insieme al *comando_GPI*.

Si può dunque concludere che la scrittura di un comando rispetta le specifiche sul timing ed il comportamento corrisponde a quello atteso.

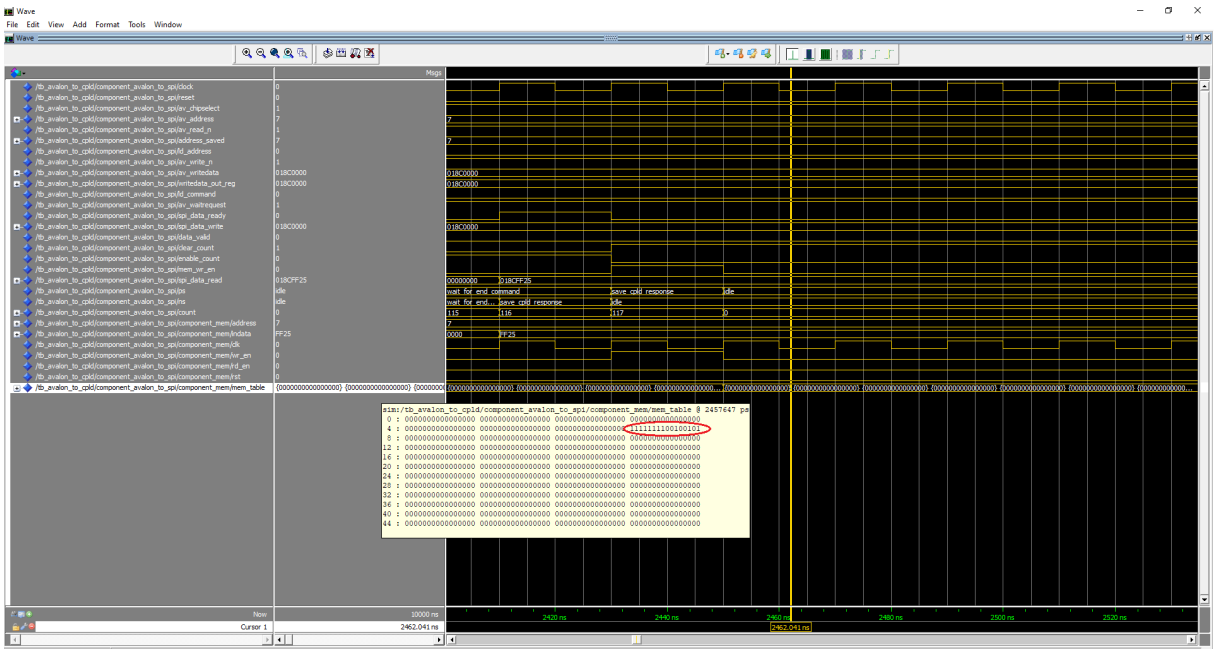

Figura 7.8: Simulazione AVALON_TO_SPI_BLOCK: memorizzazione risposta CPLD

2. **Lettura di un dato**.

   La figura 7.9 mostra la simulazione relativa al comando di lettura da parte di NIOS. *Si precisa che in tal caso il writedata è privo di significato in quanto i comandi Avalon strettamente necessari per una lettura sono solo il read_n e il readdata*. Per quanto riguarda l'*address*, invece, si è mantenuto invariato rispetto alla simulazione precedente ed è stato scelto in maniera arbitraria.

   Dalla figura si evince che la FSM evolve correttamente nello stato *SAVE_READ_ADDRESS*, dopo aver valutato che *chipselect='1'* e *read_n*='0' (che corrisponde a una richiesta di lettura). Successivamente, la FSM evolve negli stati *READ_CPLD_RESPONSE* e *CPLD_RESPONSE_AVAILABLE* e ri-

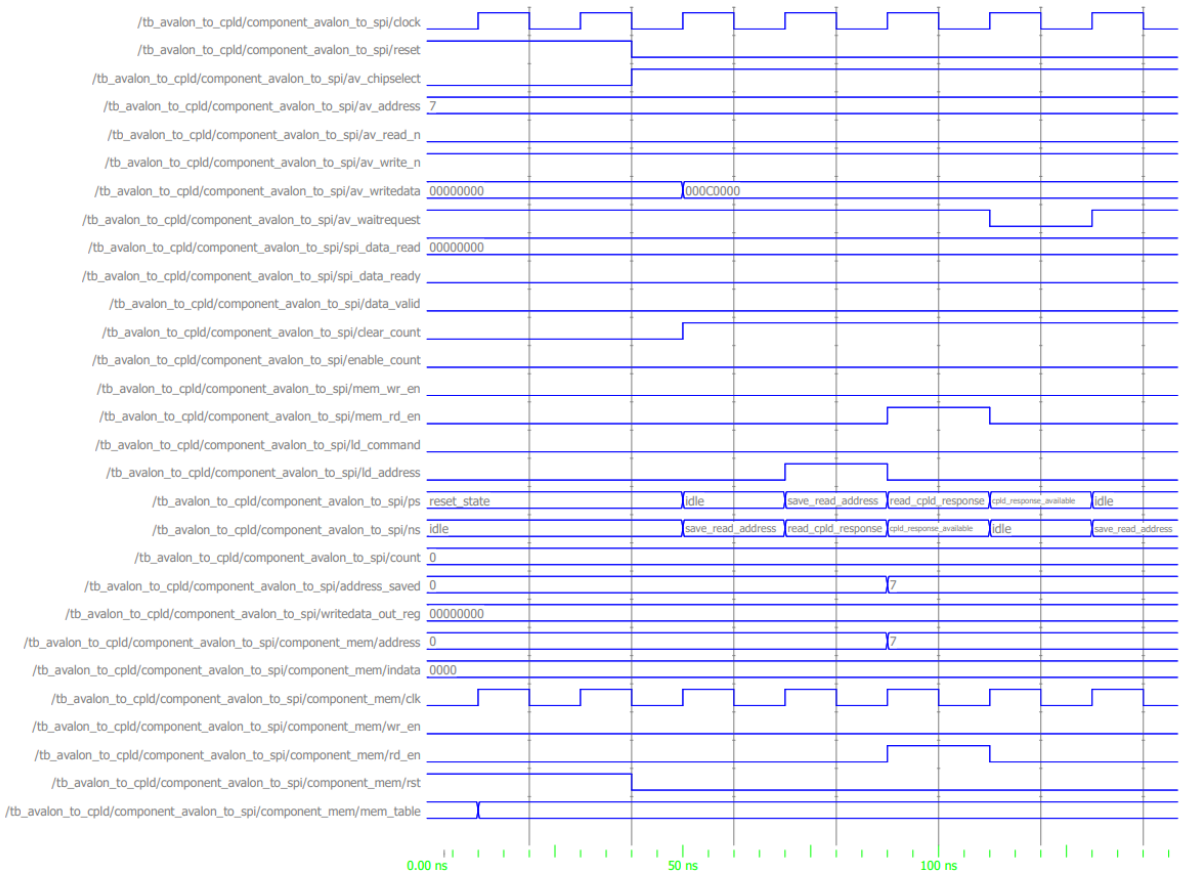


Figura 7.9: Simulazione AVALON_TO_SPI_BLOCK: comando di lettura di un dato

torna, infine nello stato di *IDLE* in attesa di un nuovo comando.
In riferimento alla figura 7.5, la FSM evolve quindi nel ramo di destra, fatto che dimostra la corretta interpretazione dei segnali giunti da NIOS e il corretto funzionamento del blocco **AVALON_TO_SPI_BLOCK** anche nel caso di richiesta di lettura.

3. **Scrittura del comando di allineamento**.
   La figura 7.10 mostra la simulazione relativa alla scrittura del comando di allineamento, inviato da NIOS all'FPGA in caso NIOS si accorga di un disallineamento della risposta inviata dal CPLD.
   Dalla figura, si può constatare che nel caso in cui NIOS mandi questo comando, la FSM evolve nello stato *LOAD_COMMAND* successivamente allo stato di "IDLE", dopo aver verificato che *chipselect='1'* e *write_n='0'*. Si prosegue, poi, nello stato *COMMAND_AVAILABLE* e nello stato *LOAD_MAN_REG*, come

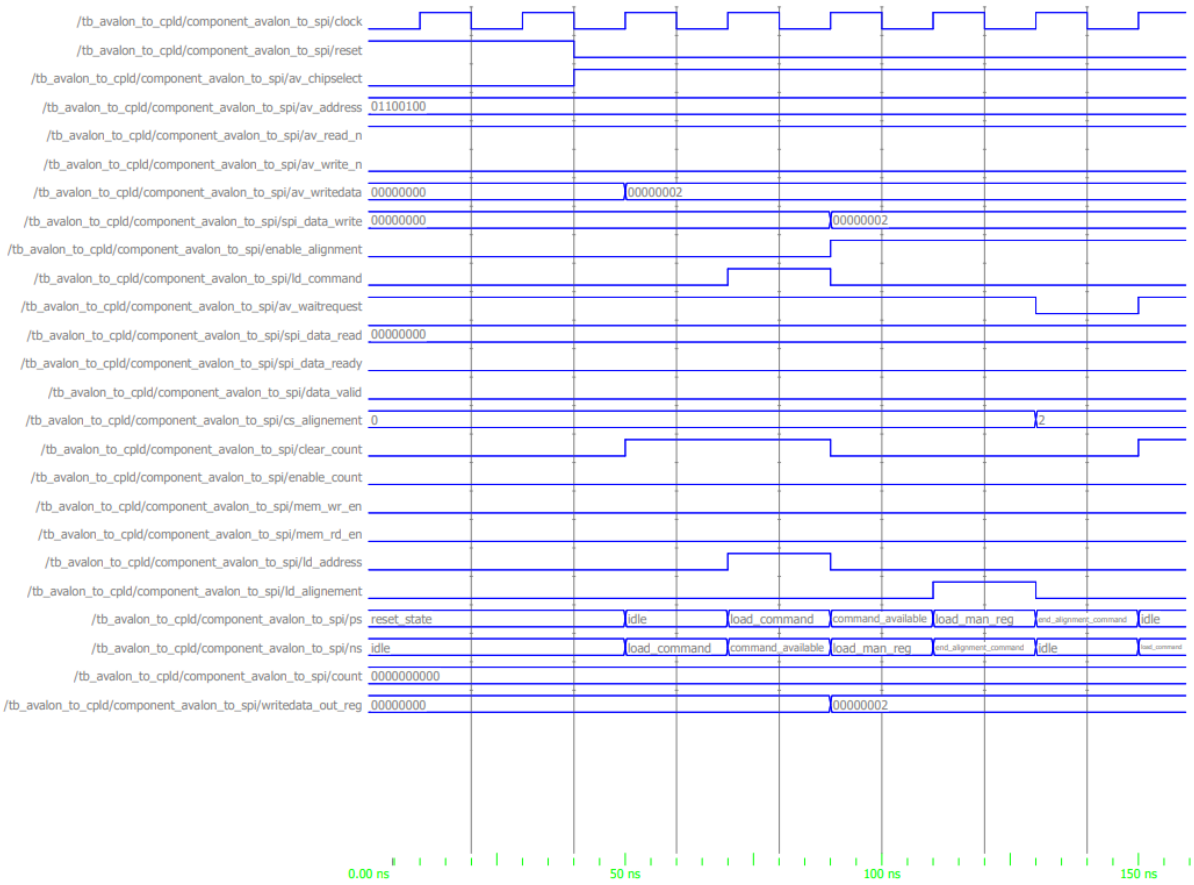


Figura 7.10: Simulazione AVALON_TO_SPI_BLOCK: scrittura comando di allineamento

conseguenza dell'attivazione del segnale *enable_alignment*. In questo stato, si carica in un apposito registro il valore di allineamento pari a '2' poiché, come spiegato precedentemente, si è notato che nella maggior parte dei casi la risposta del CPLD è disallineata di un colpo di clock e solo in rare volte di due colpi di clock. Si è dunque scelto di utilizzare come condizione di default l'allineamento pari a '1' e, solo nel caso in cui esso non risulti sufficiente, NIOS invia il comando di allineamento che permette di modificare il registro di allineamento col valore '2'. I valori di allineamento in uscita da tale registro, vengono mandati al blocco "DPA" interno al blocco "SPI_HANDLER" ed indicano al "DPA" di campionare il dato che viene mandato dal CPLD al primo o al secondo colpo di clock rispetto a quando il chipselect del CPLD viene abilitato per la prima volta.

Successivamente allo stato *LOAD_MAN_REG* si procede in *END_ALIGNMENT_COMMAND*, in cui viene disabilitato il segnale di *load* del registro di allineamento e il segnale *waitrequest*, che indica a NIOS che si è letto il comando e che si può dunque concludere il trasferimento. Infine, si ritorna in *IDLE* in attesa di un nuovo comando.

Anche in tal caso, la simulazione dimostra il corretto funzionamento del blocco in risposta al comando di allineamento e la corretta evoluzione della FSM.

Con questa simulazione si conclude la trattazione delle simulazioni relative al blocco **AVALON_TO_SPI_BLOCK**, in quanto si è dimostrata la correttezza dell'evoluzione della FSM nei tre possibili percorsi.

## 7.4 SPI_HANDLER

### 7.4.1 Entity VHDL

```
entity SPI_HANDLER is
port ( clock                    : in    std_logic;
       reset                    : in    std_logic;
       -- EXTERNAL SPI
       ex_spi_cs_n              : out   std_logic:
       ex_spi_ck                : out   std_logic;
       ex_spi_si                : out   std_logic;
       ex_spi_so                : in    std_logic;
       --  READ DATA FROM CPLD + READ ENABLE
       spi_data_read            : out   std_logic_vector(31 downto 0);
       spi_data_ready           : out    std_logic;
       -- WRITE DATA TO CPLD + WRITE ENABLE
       spi_data_write           : in    std_logic_vector(31 downto 0);
       spi_data_valid           : in    std_logic;
       cs_alignment             : in    std_logic_vector(1 downto 0)
);
end entity SPI_HANDLER;
```

### 7.4.2 Descrizione del blocco

Il blocco **SPI_HANDLER** è quello che si interfaccia con il CPLD, a cui ha il compito di inviare il comando a 26 bit contenente una delle cinque richieste mostrate in 5.1 e da cui riceve, a sua volta, la risposta corrispondente alla richiesta effettuata.
Il comando da inviare al CPLD e la risposta ricevuta da esso, vengono scambiati col blocco denominato "AVALON_TO_SPI_BLOCK" (riferimento figura 7.3), col quale vengono scambiati anche due flag che servono per determinare l'avanzamento negli stati delle FSM presenti sia all'interno di "AVALON_TO_SPI_BLOCK", sia all'interno di **SPI_HANDLER**. In particolare, "AVALON_TO_SPI_BLOCK" invia a **SPI_HANDLER** un segnale di *data_valid* che indica la presenza di un nuovo comando valido da inviare al CPLD. Questo segnale è utilizzato dalla FSM presente in uno dei blocchi interni a **SPI_HANDLER**, denominato "INPUT_FSM", che verifica che questo segnale sia pari a '1' prima di inoltrare il comando al blocco "SPI_INTERFACE" che si occupa di mandarlo in maniera seriale al CPLD.
A sua volta, invece, "AVALON_TO_SPI_BLOCK" riceve da **SPI_HANDLER** il segnale di *data_ready*, generato da un altro blocco interno a **SPI_HANDLER**, denominato "SPI_INTERFACE". Questo flag, indica che è stato ricevuto il dato trasmesso dal CPLD all'FPGA e pertanto, è pronto per essere letto.
Per concludere la descrizione di **SPI_HANDLER**, esso è dunque composto da tre blocchi interni, ovvero "INPUT_FSM" e "SPI_INTERFACE" precedentemente menzionati, più un altro blocco denominato "CPLD_DATA_ANALIZER", il quale sovra campiona il dato ricevuto dal CPLD (che lavora a $f_{clock_{cpld}} = \frac{f_{clock_{fpga}}}{4}$) e asserisce un flag che invia al blocco "SPI_INTERFACE", per indicargli il momento corretto in cui salvare il bit di dato (ovvero, circa a metà del tempo di bit).
Il blocco "INPUT_FSM", invece, si occupa di inoltrare al blocco "SPI_INTERFACE" il comando da inviare al CPLD (che a sua volta riceve da "AVALON_TO_SPI_BLOCK"), solo nel caso in cui tale blocco non sia occupato in un'altra operazione. Per questo motivo, il blocco "SPI_INTERFACE" invia il segnale *spi_busy*, che quando è '0' indica che può avvenire la comunicazione tra i due blocchi.
Un'altra condizione necessaria per iniziare la comunicazione con "SPI_INTERFACE", è che il segnale di *data_valid* sia pari a '1'. Pertanto, solo nel caso in cui questi segnali

valgano rispettivamente '0' e '1', "INPUT_FSM", invia al blocco "SPI_INTERFACE" la richiesta di poter avviare la comunicazione e gli inoltra il comando.
Il blocco "SPI_INTERFACE" è dunque quello che scambia col CPLD i segnali dell'interfaccia SPI, di cui si riporta uno generico schema a blocchi in figura 7.11.
Come si può notare dalla figura 7.11, l'interfaccia SPI (Serial Peripheral Interface) utilizza solo quattro segnali: il clock, due linee di dato (una per ogni direzione) e un chip select. Considerando la comunicazione tra FPGA e CPLD, il master è costituito dall'FPGA, mentre il CPLD rappresenta lo slave. Il master controlla tutti i trasferimenti, attivando il chip select dello slave con cui intende comunicare (in tal caso l'FPGA ha solo uno slave che è costituito dal CPLD) ed inviando allo slave il clock e il MOSI, mentre riceve da esso il cosiddetto MISO.

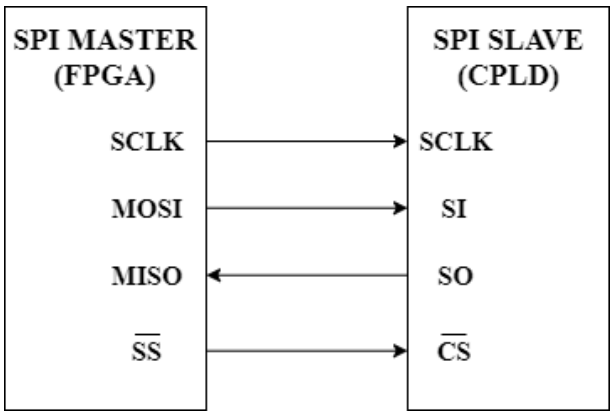


Figura 7.11: Schema a blocchi interfaccia SPI

Con riferimento al progetto, l'FPGA genera tramite il blocco "SPI_INTERFACE" i seguenti segnali per il CPLD:

- il *clock* con $f_{clock_{cpld}} = \frac{f_{clock_{fpga}}}{4} = 20MHz$;
- lo *slave_select*, attivo col valore logico basso, deve rimanere attivo per 26 colpi di clock, ovvero fino a quando viene trasmesso l'ultimo bit del comando;
- il *MOSI*: è il comando a 26 bit che arriva dal blocco "AVALON_TO_SPI_BLOCK" mediante "INPUT_FSM" e che il blocco "SPI_INTERFACE" manda in maniera seriale al CPLD;

A sua volta, il blocco "SPI_INTERFACE", riceve dal CPLD il *MISO*, ovvero il dato proveniente dal CPLD e nel momento in cui riceve tutti i bit di dato, attiva il segnale

*data_ready* che va in ingresso al blocco "AVALON_TO_SPI_BLOCK", dal quale viene utilizzato per salvare il dato inviato dal CPLD in quanto indica la presenza di dato valido arrivato dal CPLD.

### 7.4.3 Simulazioni

Al fine di verificare il corretto comportamento di questo blocco, si è utilizzato il software "Modelsim" per simulare il firmware realizzato. Si è realizzato un testbench per testare i 5 possibili comandi previsti per il CPLD.
Tuttavia, a titolo di esempio, si riportano di seguito le simulazioni per il *comando_GPI*. Esse hanno l'obiettivo di dimostrare il corretto funzionamento dei 3 blocchi interni a **SPI_HANDLER**, ovvero "INPUT_FSM", "SPI_INTERFACE" e "CPLD_DATA_ANALYZER".

1. **Simulazione blocco "INPUT_FSM"**.

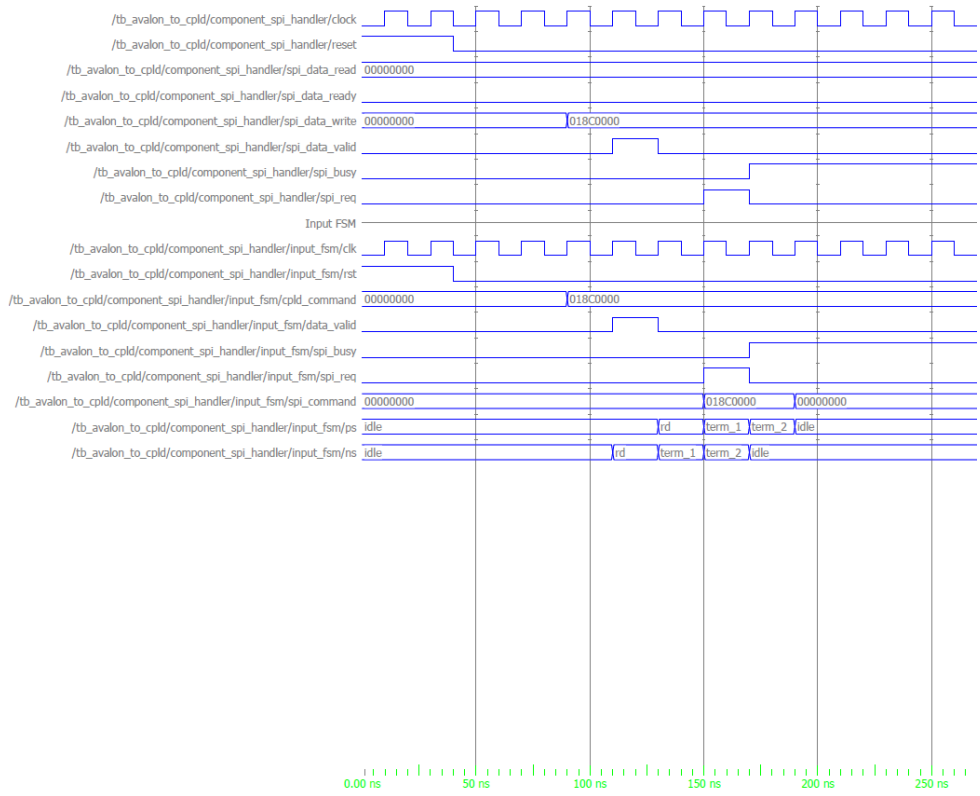


Figura 7.12: Simulazione INPUT_FSM

Dalla figura 7.12, si evince che la FSM interna a questo blocco evolve negli stati solo quando si verifica la condizione *data_valid='1'* e *spi_busy='0'*. In particolare, nello stato "term_1" viene salvato il comando che arriva dal blocco "AVALON_TO_SPI_BLOCK" e questo viene poi inoltrato al blocco "SPI_INTERFACE" che si occuperà di mandarlo in maniera seriale al CPLD. Dopo aver inoltrato il comando, la FSM torna in "idle", in attesa di un nuovo comando valido. Il comportamento ottenuto è dunque conforme a quello atteso.

2. **Simulazione blocco "SPI_INTERFACE"**.

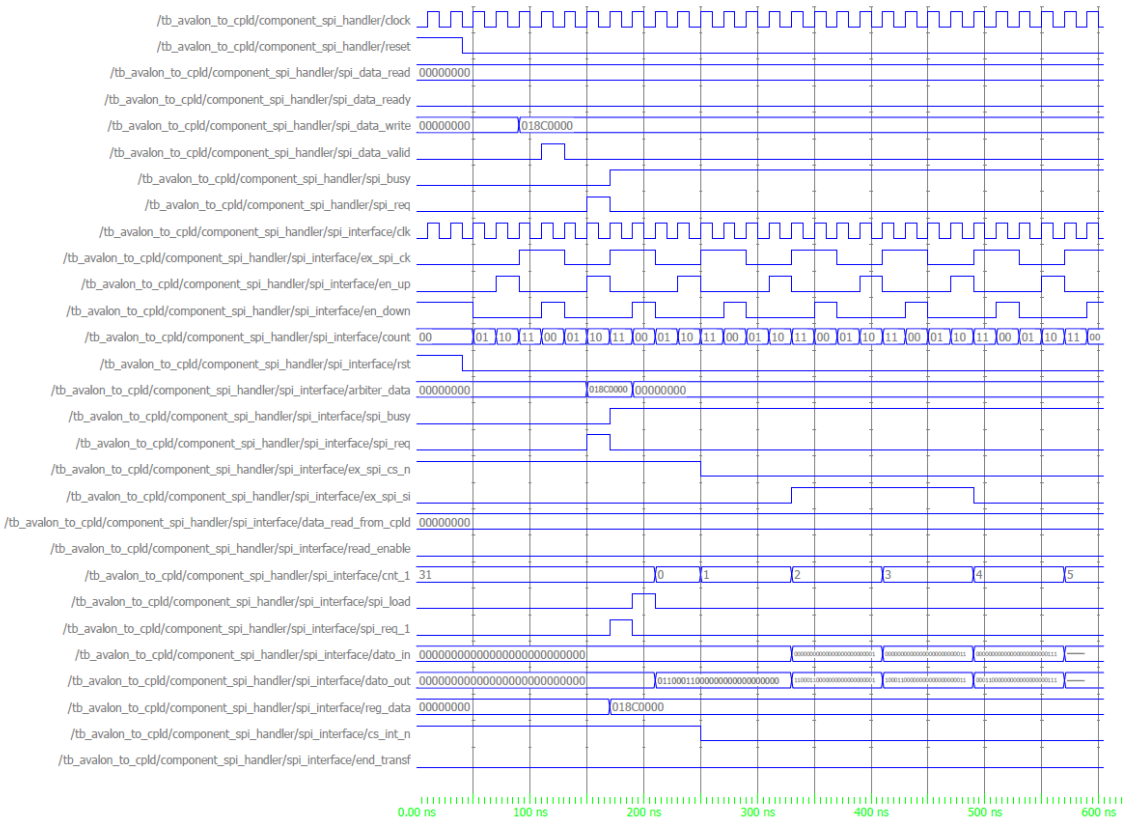


Figura 7.13: Simulazione blocco SPI_INTERFACE

La figura 7.13 mostra la simulazione relativa al blocco "SPI_INTERFACE". Dalla figura si evince che questo blocco genera il clock per il CPLD ad una frequenza pari a $\frac{1}{4}$ di quella del clock dell'FPGA (segnale ex_spi_ck in figura). Esso inoltre attiva il segnale *SS* per il CPLD quando si abilita il contatore (segnale *cnt_1* in figura) che tiene conto dei 26 colpi di clock necessari a inviare tutti i

26 bit di dato al CPLD. Infine, questo blocco salva il comando che gli arriva dal segnale "ARBITER_DATA" nel segnale denominato *dato_out*, e manda al CPLD un bit per ogni colpo di colpo tramite il segnale SPI *ex_spi_si*.

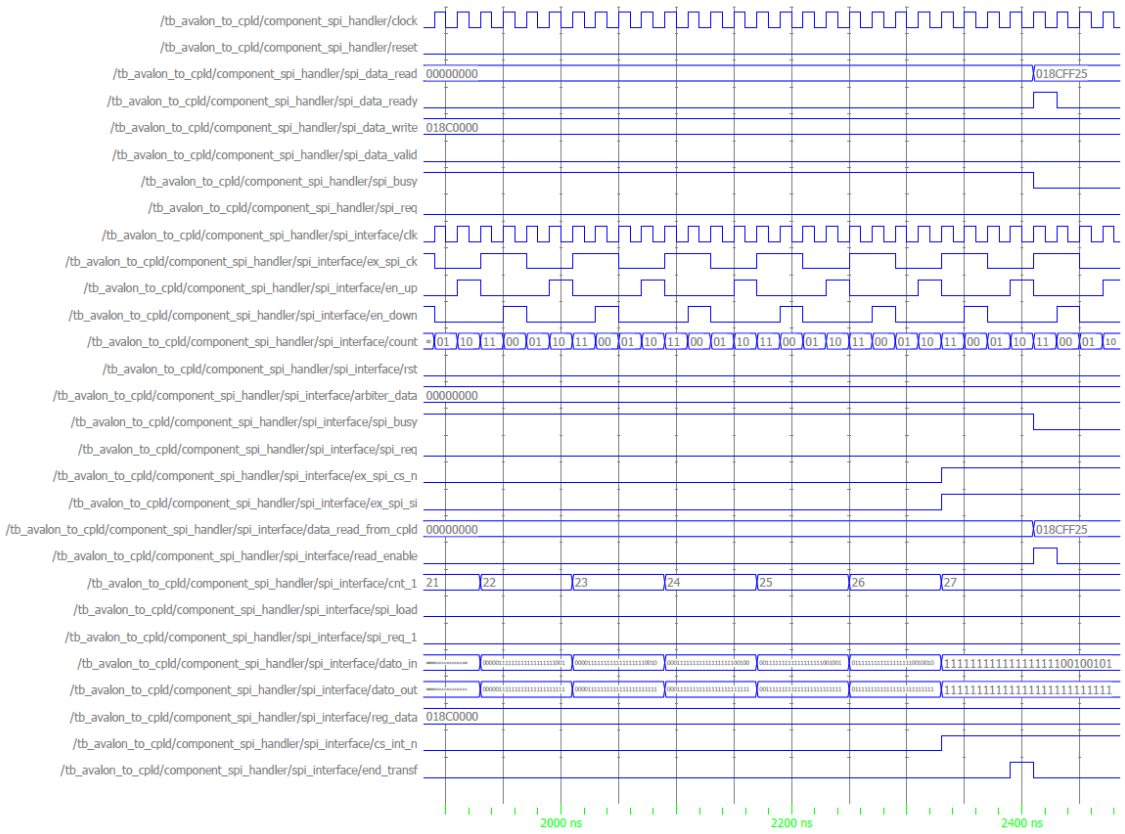


Figura 7.14: Simulazione blocco SPI_INTERFACE - fine della comunicazione

Per quanto riguarda il blocco "SPI_INTERFACE", si riporta anche la simulazione relativa alla fine della comunicazione. Dalla figura 7.14, si evince che il blocco "SPI_INTERFACE" disattiva *SS* per il CPLD nel momento in cui il contatore raggiunge il valore 27. Questo significa, infatti, che l'FPGA ha inviato al CPLD tutti i 26 bit del comando e ha ricevuto i 26 bit di risposta del CPLD. Il segnale *dato_in* in figura, mostra negli 8 bit meno significativi la risposta attesa per il *comando_GPI*, ovvero il pattern binario "00100101". Questo dimostra il corretto funzionamento del blocco "SPI_INTERFACE", oltre alla corretta comunicazione tra FPGA e CPLD.

3. **Simulazione blocco "CPLD_DATA_ANALYZER".**

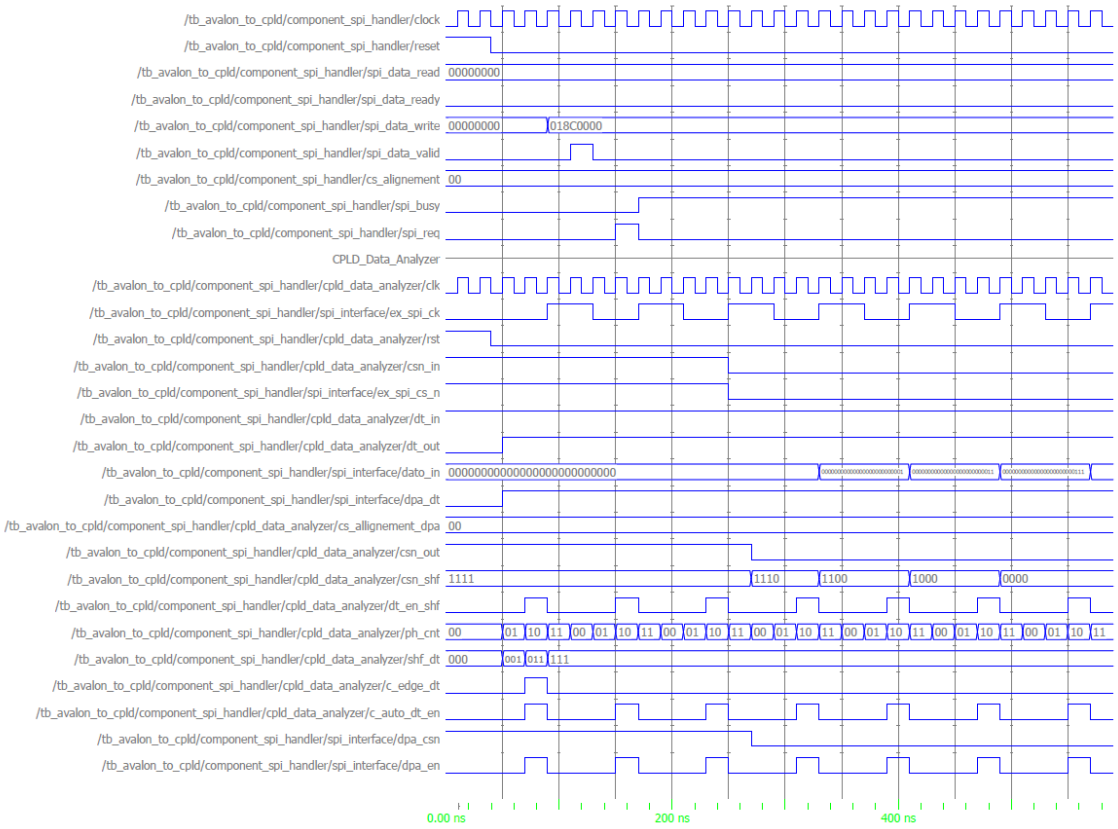


Figura 7.15: Simulazione blocco CPLD_DATA_ANALYZER

Per concludere le simulazioni relative al blocco "SPI_HANDLER", si riporta in figura 7.15 la simulazione relativa al blocco "CPLD_DATA_ANALYZER".
Esso riceve in maniera seriale il dato che arriva dal CPLD (segnale dt_in) e questo viene salvato nel segnale *dato_in*. Infatti, dalla figura si nota che nel momento in cui si attiva il *SS* del CPLD, il segnale *dato_in* inizia a salvare un bit alla volta del segnale *dt_in* (che nell'esempio è stato fissato a '1' per semplificare la trattazione). Il segnale *dt_out*, invece, è il dato che arriva dal CPLD e che questo blocco manda a "SPI_INTERFACE". Infine, il segnale *csn_shf* è quello utilizzato per l'allineamento della risposta del CPLD. Si nota che tale segnale ha LSB=0 nel momento in cui viene attivato *SS* del CPLD (il segnale *SS=slave_select* è attivo col valore logico basso). Questo valore viene poi shiftato per 4 colpi di clock. Come spiegato nel paragrafo 7.3.2, per garantire l'allineamento della risposta del CPLD, il "CPLD_DATA_ANALYZER" campiona di default il dato in

ingresso al primo colpo di clock successivo a quello in cui viene attivato *SS* del CPLD (che corrisponde alla condizione *csn_shf=1100*). Nel caso in cui questa correzione non sia ancora sufficiente, il "CPLD_DATA_ANALYZER" campiona il dato del CPLD al colpo di clock successivo, che corrisponde alla condizione *csn_shf=1000*)

## 7.5 IP_BLOCK

### 7.5.1 Entity VHDL

```
entity ip_blk is
generic(
    ip_addr_size              : natural := 6;
    gen_addr_size             : integer := 7;
    ip_mem_addr_size          : natural := 22;
    ip_bus_size               : natural := 16;
    data_width                : integer := 16
);
port(
    --ip I/F side
    ck0                       : in    std_logic; --ip clock
    ck1                       : in    std_logic; --clk=80MHz
    a_rst                     : in    std_logic;
    sw_rst                    : out   std_logic;
        -- IP interface
    ip_data                   : inout std_logic_vector(15 downto 0);
    ip_add                    : in    std_logic_vector( 5 downto 0);
    ip_iosel_n                : in    std_logic;
    ip_memsel_n               : in    std_logic;
    ip_idsel_n                : in    std_logic;
    ip_rw_n                   : in    std_logic;
    ip_ack_n                  : out   std_logic;
```

```
    ip_bs_n                     : in    std_logic_vector( 1 downto 0);
    ip_dmareq_n                 : out   std_logic_vector( 1 downto 0);
    ip_dmack_n                  : in    std_logic;
    ip_dmaend_n                 : inout std_logic;
    ip_error_in_n               : in    std_logic;
    ip_error_out_n              : out   std_logic;
    oe_n_i                      : out   std_logic;
    dir_i                       : out   std_logic;
    -- board and power state -
        gr_mode                     : in    std_logic_vector(1 downto 0);
    gr_60nvi_fault              : in    std_logic;
    gr_60vi_fault               : in    std_logic;
    gr_12nvi_fault              : in    std_logic;
    gr_12vi_fault               : in    std_logic;
    gr_5vi_fault                : in    std_logic;
    gr_pll_unlock               : in    std_logic;
    gr_33vi_fault               : in    std_logic;
    gr_12v_fault                : in    std_logic;
   --Avalon interface between NIOS II and CHX_REG
    nios_av_chxreg_address      : in    std_logic_vector(10 downto 0);
    nios_av_chxreg_be           : in    std_logic_vector( 1 downto 0);
    nios_av_chxreg_write_n      : in    std_logic;
    nios_av_chxreg_read_n       : in    std_logic;
    nios_av_chxreg_chipselect   : in    std_logic;
    nios_av_chxreg_writedata    : in    std_logic_vector((ip_bus_size -1) downto 0);
    nios_av_chxreg_waitrequest  : out   std_logic;
    nios_av_chxreg_readdata     : out   std_logic_vector((ip_bus_size -1) downto 0);
    NIOS_wait1_reg0_if          : in    std_logic;
    NIOS_wait1_reg1_if          : in    std_logic;
    NIOS_wait1_reg2_if          : in    std_logic
);
end ip_blk;
```

### 7.5.2 Descrizione del blocco

Il blocco denominato **IP_BLOCK** funziona da intermediario tra NIOS e le periferiche con cui esso comunica. In questa nuova versione del firmware si è scelto di mantenere solo due periferiche, ovvero le uniche necessarie per soddisfare le specifiche richieste. In particolare, il processore NIOS scambia alcuni segnali riguardanti lo stato della scheda e delle alimentazioni con il blocco denominato "GEN_REG" (*Generic Registers*) il quale, comunicando anche col blocco "MEM_DECODER", è in grado di fornire al mondo esterno le informazioni ricevute da NIOS.
Il blocco "CHX_REG", invece, si occupa di scrivere e leggere i dati in una memoria interna dell'FPGA, configurata come SRAM. Dato che, in questo caso, sia NIOS che il blocco "MEM_DECODER" possono richiedere al blocco "CHX_REG" di effettuare un'operazione di lettura o scrittura in memoria, per evitare un conflitto sulla comunicazione, si è scelto di impedire ad uno dei due blocchi di fare una richiesta a "CHX_REG", se esso è già occupato a eseguire una richiesta precedentemente pervenuta dall'altro blocco. Quindi, la condizione necessaria per avviare la comunicazione con uno dei due blocchi (NIOS o MEM_DECODER), è che sia attivo il *chipselect* di uno solo dei due mentre l'altro sia disattivo.
La ROM denominata ID0, invece, può essere utilizzata per l'auto-configurazione del sistema e la sua presenza è imposta dallo standard ANSI/VITA 4-1995, illustrato in 6.2.
Il blocco "MEM_DECODER", invece, riceve da "IP_BRIDGE" i segnali dello standard ANSI/VITA 4-1995 e comunica mediante interfaccia *Avalon* con una delle due periferiche ("GEN_REG" o "CHX_REG"), dopo aver opportunamente decodificato i bit 7,8,9,10 dell'*address* [2]. In particolare, quando questi 4 bit dell'address sono pari a "0001" "MEM_DECODER" comunica col blocco "GEN_REG", mentre quando sono pari a "0010" "MEM_DECODER" comunica con "CHX_REG".

[2] L'*address* che arriva a "MEM_DECODER" è illustrato in figura 4.2

### 7.5.3 IP_BRIDGE

### 7.5.4 Entity VHDL

```
entity ip_bridge is

generic(
    ip_addr_size      : natural := 6;
    ip_mem_addr_size  : natural := 22;
    ip_bus_size       : natural := 16;
    rom_addr_size     : natural := 5;
    rom_bus_size      : natural := 16
    );
port(
    --IP CARRIER SIDE
    clk               : in    std_logic;
    rst_h             : in    std_logic;
    id_sel            : in    std_logic;
    mem_sel           : in    std_logic;
    data_direction    : in    std_logic;
    address           : in    std_logic_vector((ip_addr_size -1) downto 0);
    data_bus          : inout std_logic_vector((ip_bus_size -1) downto 0);
    data_acknowledge_n: out   std_logic;
    dir               : out   std_logic;
    oe_n              : out   std_logic;
    --AVALON INTERFACE WITH IDO_RO
    av_id_chipselect  : out  std_logic;
    av_id_address     : out  std_logic_vector((rom_addr_size -1) downto 0);
    av_id_read_n      : out  std_logic;
    av_id_readdata    : in   std_logic_vector((rom_bus_size -1) downto 0);
    av_id_waitrequest : in   std_logic;
    --AVALON INTERFACE WITH MEM_DECODER
    av_mem_chipselect : out  std_logic;
```

```
    av_mem_address    : out std_logic_vector((ip_mem_addr_size -1) downto 0);
    av_mem_read_n     : out  std_logic;
    av_mem_readdata   : in   std_logic_vector((ip_bus_size -1) downto 0);
    av_mem_write_n    : out  std_logic;
    av_mem_writedata  : out  std_logic_vector((ip_bus_size -1) downto 0);
    av_mem_waitrequest: in   std_logic
    );
end entity ip_bridge;
```

### 7.5.5 Descrizione IP_BRIDGE

In questo paragrafo, si rivolge una particolare attenzione al blocco **IP_BRIDGE** che si trova all'interno di "IP_BLOCK" e che si occupa di trasferire i segnali dello standard ANSI/VITA 4-1995 a due periferiche. In particolare, esso contiene al suo interno la FSM che rispetta il timing definito dallo standard ANSI/VITA 4-1995, mostrato in figura 6.2 e 6.3.
Inoltre, questo blocco contiene al suo interno 5 registri, utilizzati per memorizzare i 6 bit di *ip_address* e i 16 bit di *ip_data* (sia quelli che servono per completare l'*address* di memoria a 22 bit, sia quelli contenenti il dato vero e proprio), mentre altri due registri sono usati per salvare il dato restituito dalla memoria ID0-ROM e dal blocco "CHX_REG", che giunge a **IP_BRIDGE** passando tramite il blocco "MEM_DECODER". Infine, questo blocco contiene anche un contatore a 8 bit, utilizzato per monitorare il funzionamento della macchina. Infatti, in condizioni di corretta esecuzione, il *terminal_count* di tale contatore non dovrebbe essere asserito a '1', permettendo in tal modo alla FSM di evolvere negli stati. Nel caso in cui il conteggio arrivi alla fine, invece, significa che si è verificato un problema e dunque viene forzato il passaggio della macchina nello stato di *IDLE*, dove vengono resettati tutti i segnali.

### 7.5.6 Premessa sui BUS transceivers

Prima di procedere con la descrizione della FSM progettata, si propone una breve premessa sui BUS transceivers SN74LVC16245A[9] a cui l'FPGA e i segnali dello standard ANSI/VITA 4-1995 sono collegati.Essa servirà per comprendere meglio la necessità di inserire nella FSM uno stato di *IDLE* oltre lo stato di *RESET*.
I BUS transceivers sono due componenti integrati su scheda, i quali sono progettati per la comunicazione asincrona tra BUS dati. In figura 7.16 si riporta lo schema a blocchi che descrive i collegamenti tra l'FPGA e i segnali dello standard.

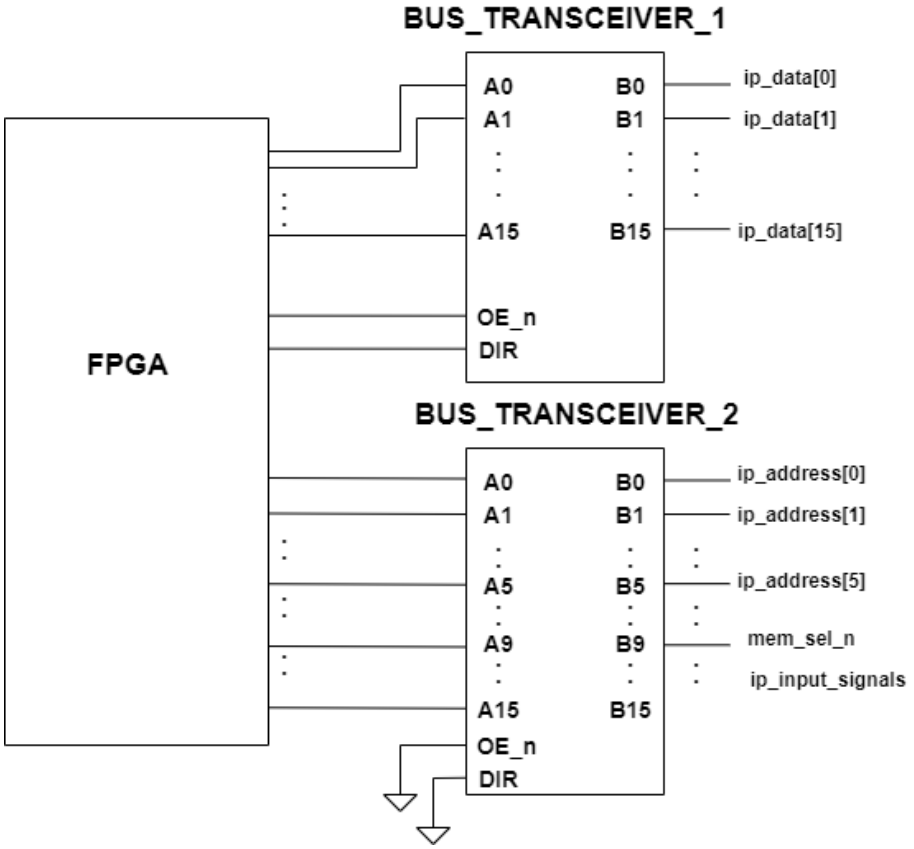


Figura 7.16: Schema a blocchi FPGA e BUS transceiver

In particolare, si nota che sono presenti due BUS transceivers, che sono caratterizzati da due ingressi **OE_n** e **DIR** che determinano la direzione della comunicazione, secondo la tabella di verità 7.2.
Nel nostro caso, il BUS_transceiver_2 è collegato sul lato A all'FPGA e sul lato B ai soli inputs dello standard IP (tra cui l'*ip_address*, i selettori *id_sel* e *mem_sel*, *RW_n*).

| OE_n | DIR | OPERATION |
|---|---|---|
| L | L | B to A bus |
| L | H | A to B bus |
| H | X | Isolation |

Tabella 7.2: Tabella di verità dei BUS transceivers

Pertanto, per questi segnali, si vuole il passaggio in un'unica direzione, ovvero dal BUS B al bus A, in quanto sono tutti ingressi dell'FPGA. Per tale motivo, l'**OE_n** e il segnale **DIR** sono collegati a GND, che corrisponde alla prima configurazione mostrata in tabella 7.2.

Il BUS_transceiver_1, invece, è collegato sul lato A all'FPGA e sul lato B all'*ip_data*, che è un segnale bidirezionale. Questo implica che può essere pilotato sia dal mondo esterno che dall'FPGA. Per questo motivo, si rende necessario asserire nella maniera corretta i segnali **OE_n** e **DIR** a seconda dell'operazione che si sta svolgendo. In particolare, nel caso in cui dall'esterno si richieda la lettura di un dato, sarà l'FPGA a pilotare l'*ip_data* ; è dunque necessario attivare l'**OE_n** e e impostare **DIR='1'**. Questa condizione corrisponde alla seconda riga della tabella 7.2, secondo cui la direzione della comunicazione è dal BUS A al BUS B, ovvero nel nostro caso dall'FPGA all'esterno.

Nel caso che dall'esterno arrivi una richiesta di scrittura, invece, l'*ip_data* sarà pilotato dall'esterno; è dunque necessario attivare l'**OE_n** e impostare **DIR='0'**. Questa condizione corrisponde alla prima riga della tabella 7.2, secondo cui la direzione della comunicazione è dal BUS B al BUS A, ovvero nel nostro caso dal mondo esterno all'FPGA.

### 7.5.7 FSM IP_BRIDGE

In figura 7.17 si riporta la FSM realizzata per gestire lo scambio di segnali dello standard ANSI/VITA 4-1995 col mondo esterno e con le periferiche interne.
In base alla richiesta che arriva dall'esterno, infatti, **IP_BRIDGE** ha il compito di attivare la periferica corretta tra "ID0_ROM" e "MEM_DECODER", che a sua volta inoltrerà la richiesta a "GEN_REG" o "CHX_REG" in base alla decodifica degli opportuni bit dell'*address*.
Come si evince dalla figura 7.17, nello stato di *RESET* vengono resettati tutti i segnali, mentre lo stato di *IDLE* è necessario per poter attivare l'**OE_n** del BUS_Transceiver_1 integrato sulla scheda. In questo modo, nello stato di *RESET* è possibile disabilitare l'**OE_n** garantendo l'isolamento tra i BUS (vedasi tabella 7.2), mentre è necessario attivare l'**OE_n** prima che la FSM evolva negli stati, in modo da permettere la comunicazione con l'esterno.
In seguito allo stato di *IDLE* si procede nello stato *SELECT_STATE*, nel quale si salvano in due appositi registri l'*ip_address* e l'*ip_data* (che in questa fase iniziale contribuisce a formare l'address a 22 bit in base al timing riportato in figura 6.2 e 6.3). Infatti, dai timing definiti dallo standard, si nota che che l'*ip_address* e l'*ip_data* sono disponibili nello stato di *SELECT* e l'*ip_data* è valido per un solo colpo di clock, motivo per cui è necessario memorizzarlo.
Successivamente, si verifica che solo uno dei due selettori tra *id_sel* e *mem_sel* sia attivo, altrimenti si attende nello stato *SELECT_STATE* finché questa condizione è verificata, così da evitare conflitti nella comunicazione.
Nel caso in cui sia attivo solo *id_sel*, l'unica operazione possibile è la lettura di un dato dalla "ID0_ROM". Per questo motivo, si verifica che il segnale *data_direction* che indica se effettuare una lettura o una scrittura sia pari a '1' (*data_direction*='1' significa LETTURA, mentre *data_direction*='0' significa SCRITTURA) e si procede quindi nello stato di WAIT specifico per la "ID0_ROM" (denominato *ID_WAIT* in figura 7.17). Infatti, essendo la ROM in sola lettura, non è possibile effettuare su di essa un'operazione di scrittura. Pertanto, nel caso in cui *data_direction* sia diverso da '1', si attende nello stato *SELECT_STATE* una richiesta valida.
Nello stato *ID_WAIT*, si attiva il *chipselect* della "ID0_ROM" e il segnale Avalon

*read_n*, che indica una richiesta di lettura dalla "ID0_ROM" all'indirizzo indicato dall'*address*.

Nello stato *ID_WAIT*, inoltre, si abilita il contatore utilizzato per verificare il corretto funzionamento del sistema. Infatti, nel caso in cui esso funzioni correttamente, il *terminal_count* non si dovrebbe mai attivare e si dovrebbe procedere negli stati successivi. In caso esso si attivi, invece, significa che si è verificato un problema e dunque si fa ritornare la FSM nello stato di *IDLE*, in cui tutti i segnali vengono impostati a un valore noto. A questo punto, la macchina riparte a ricevere i nuovi segnali IP che eventualmente giungono dall'esterno.

In aggiunta, in questo stato si carica in un apposito registro il dato che arriva dalla "ID0_ROM"; tuttavia, l'unico dato valido è quello che viene inviato dalla "ID0_ROM" quando essa disattiva il segnale di *waitrequest*, in quanto significa che non è impegnata in un'altra operazione e dunque può fornire il dato richiesto tramite il *readdata*.

Per questo motivo, dopo lo stato di *ID_WAIT*, si effettua la verifica sul valore del *terminal_count* e se questo è pari a '0' , si verifica che solo il *waitrequest* che arriva dalla "ID0_ROM" sia disattivo, mentre quello che arriva da "MEM_DECODER" sia attivo, a significare che solo la "ID0_ROM" è disponibile per la comunicazione.

Se questa condizione è verificata, significa che l'ultimo dato salvato nel registro è quello corretto e si può dunque inviare all'esterno tramite l'*ip_data*.

Infatti, in seguito allo stato *ID_WAIT* si procede nello stato *ID_TERMINATE* in cui **IP_BRIDGE** invia a sua volta all'esterno tramite l'*ip_data* il dato che ha letto dalla "ID0_ROM".

Per questo motivo, viene attivato il segnale IP *data_acknowledge* ed è necessario impostare **DIR='1'** dato che è l'FPGA a pilotare l'*ip_data*.

Invece, se il segnale *waitrequest* che arriva dalla "ID0_ROM" non è disattivo, si attende nello stato di *ID_WAIT* che essa si liberi in modo da realizzare l'operazione richiesta.

Nel caso, invece, in cui sia attivo solo *mem_sel*, è necessario distinguere tra le due possibili operazioni di lettura o scrittura. Infatti, *mem_sel* implica la richiesta di comunicare con il blocco "GEN_REG" o con il blocco "CHX_REG", che contiene al suo interno blocchi M9K configurati come SRAM, in cui è possibile effettuare entrambe le operazioni di lettura e scrittura. E' il blocco "MEM_DECODER" che, in base all'ad-

dress, comprende a quale dei due blocchi inoltrare la richiesta, mentre **IP_BRIDGE** comunica mediante interfaccia Avalon *solo* col blocco "MEM_DECODER" e pertanto i segnali Avalon che vengono attivati o controllati in questa FSM sono *solo* quelli relativi a questo blocco.
Dunque, nel caso in cui *data_direction* sia pari a '1', ovvero nel caso in cui arrivi una richiesta di lettura, si procede in maniera analoga a quanto descritto per il blocco "ID0_ROM".
L'unica differenza è che in tal caso viene attivato il *chipselect* e il segnale *read_n* del "MEM_DECODER" ed il dato che proviene dal "MEM_DECODER" ( che arriva a sua volta da "GEN_REG" o da "CHX_REG"), viene memorizzato in un registro dedicato diverso da quello usato per salvare il dato letto dalla "ID0_ROM". Esso viene infine inviato all'esterno tramite l'*ip_data*, nello stato di *MEM_READ_TERMINATE*.
Nel caso in cui *data_direction* non sia pari a '1', significa che si vuole effettuare una scrittura. Per questo motivo si procede nello stato denominato *MEM_GET_DATA*, in cui si salva in un altro registro dedicato la nuova sequenza di *ip_data* che contiene ora il dato vero e proprio da scrivere. Questo sarà dunque disponibile al colpo di clock successivo, ovvero nello stato *MEM_WRITE_WAIT*. In questo stato si abilita il *chipselect* di "MEM_DECODER" e il segnale *write_n* e si abilita il contatore.
Si precisa, inoltre, che il segnale **DIR** è stato impostato a '0' di default, pertanto in tal caso il dato che arriva dall'esterno può entrare nell'FPGA (OE_n='0' e DIR='0' corrisponde alla prima riga della tabella 7.2).
Si procede quindi con la verifica del valore del *terminal_count* per lo stesso motivo descritto nel caso della "ID0_ROM" e solo quando il *waitrequest* proveniente dal blocco "MEM_DECODER" è disattivo, si procede a inviare a questo blocco il *writedata* che contiene il dato da scrivere.

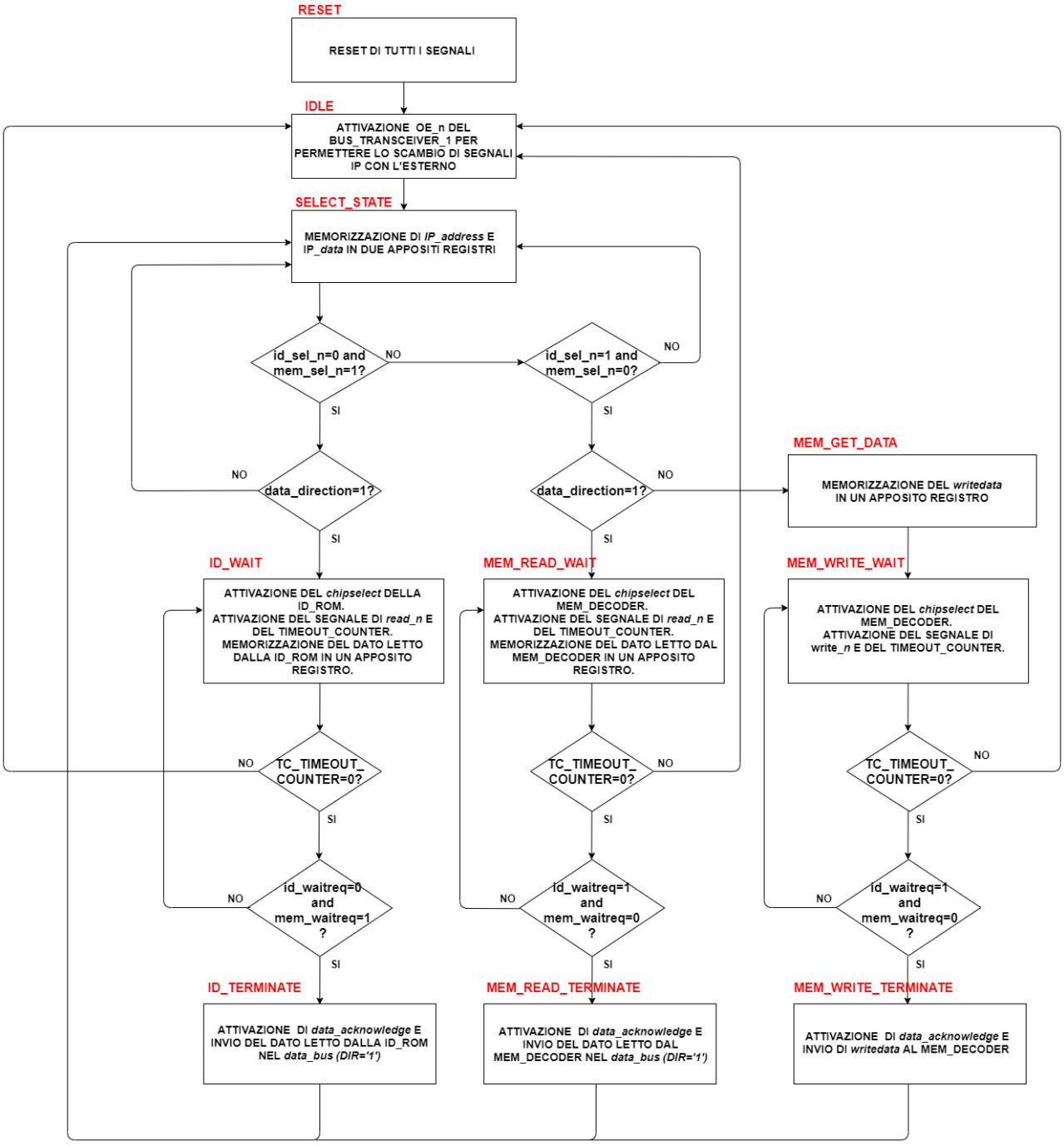


Figura 7.17: FSM IP_BRIDGE

### 7.5.8 Simulazioni

Anche in questo caso, per dimostrare il corretto comportamento della FSM all'interno del blocco "IP_BRIDGE", si riportano le simulazioni effettuate mediante il software "Modelsim" relative ai 3 seguenti casi:

1. **Scrittura in "CHX_REG"**;
2. **Lettura da "CHX_REG"**;
3. **Lettura da ID_ROM**;

Questi 3 casi, infatti, mostrano le 3 possibili evoluzioni della FSM.

1. **Scrittura in "CHX_REG"**.
   Come spiegato nel paragrafo precedente, nella FSM compaiono i segnali *Avalon* relativi al blocco "MEM_DECODER"; è poi compito di questo blocco interpretare la richiesta che arriva dall'esterno e comunicare col blocco "GEN_REG" o "CHX_REG".
   La figura 7.18 mostra la simulazione relativa alla richiesta di scrittura di un dato nella memoria interna a "CHX_REG". Come si evince dalla figura, la FSM evolve nello stato *SELECT_STATE*, in cui si salvano in due appositi registri l'*ip_address* e l'*ip_data*. Infatti, l'*address* completo per la memoria è formato dalla concatenazione dei suddetti segnali, come mostrato dal segnale *ip_av_mem_address* in figura.
   Successivamente, si procede nel percorso più a destra della FSM (riferimento alla figura 7.17), dopo aver verificato che è attivo solo il selettore della memoria e che è arrivata una richiesta di scrittura (*dir_i='0'*). La FSM evolve quindi negli stati *MEM_GET_DATA*, in cui si salva il dato da scrivere in memoria e si procede in *MEM_WRITE_WAIT* e *MEM_WRITE_TERMINATE*, in cui si attiva il segnale *ip_ack_n* per concludere il trasferimento del dato. Si ritorna poi nello stato *SELECT_STATE*, in cui si attende l'arrivo di un nuovo comando.

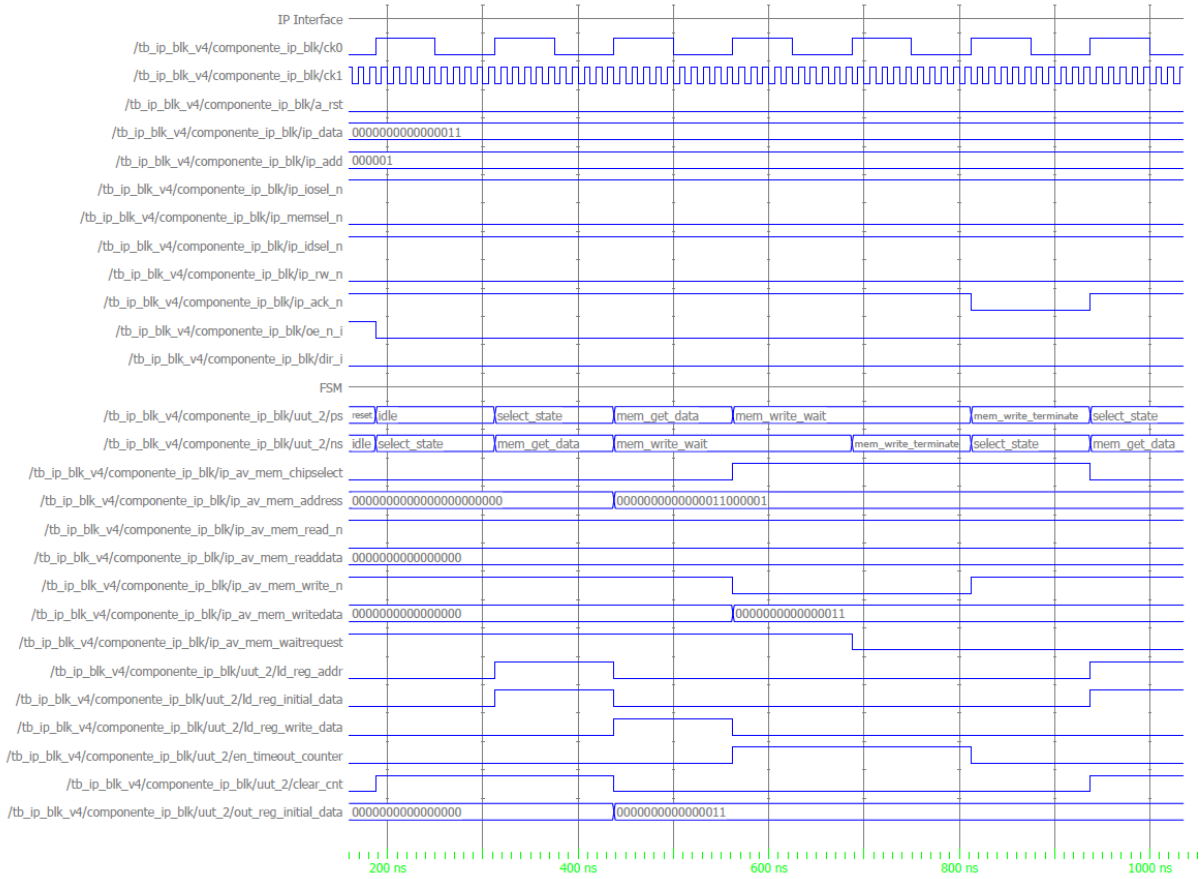


Figura 7.18: Simulazione IP_BRIDGE: scrittura in memoria

2. **Lettura in "CHX_REG"**.

   La simulazione relativa al comando di lettura dalla memoria è riportata in figura 7.19.

   In tal caso, come si può notare dalla figura, la FSM procede in *SELECT_STATE* e, successivamente, in *MEM_READ_WAIT*, dopo aver verificato che è attivo solo il selettore della memoria e che è arrivata una richiesta di lettura. La FSM evolve dunque nel percorso centrale, con riferimento alla figura 7.17.

   Si procede, quindi, in *MEM_READ_TERMINATE*, nel quale si abilita per un colpo di clock il segnale *ip_ack_n* e si conclude il trasferimento del dato inviando all'esterno il dato letto dalla memoria "CHX_REG" ( che arriva in questo blocco tramite il segnale *Avalon readdata*).

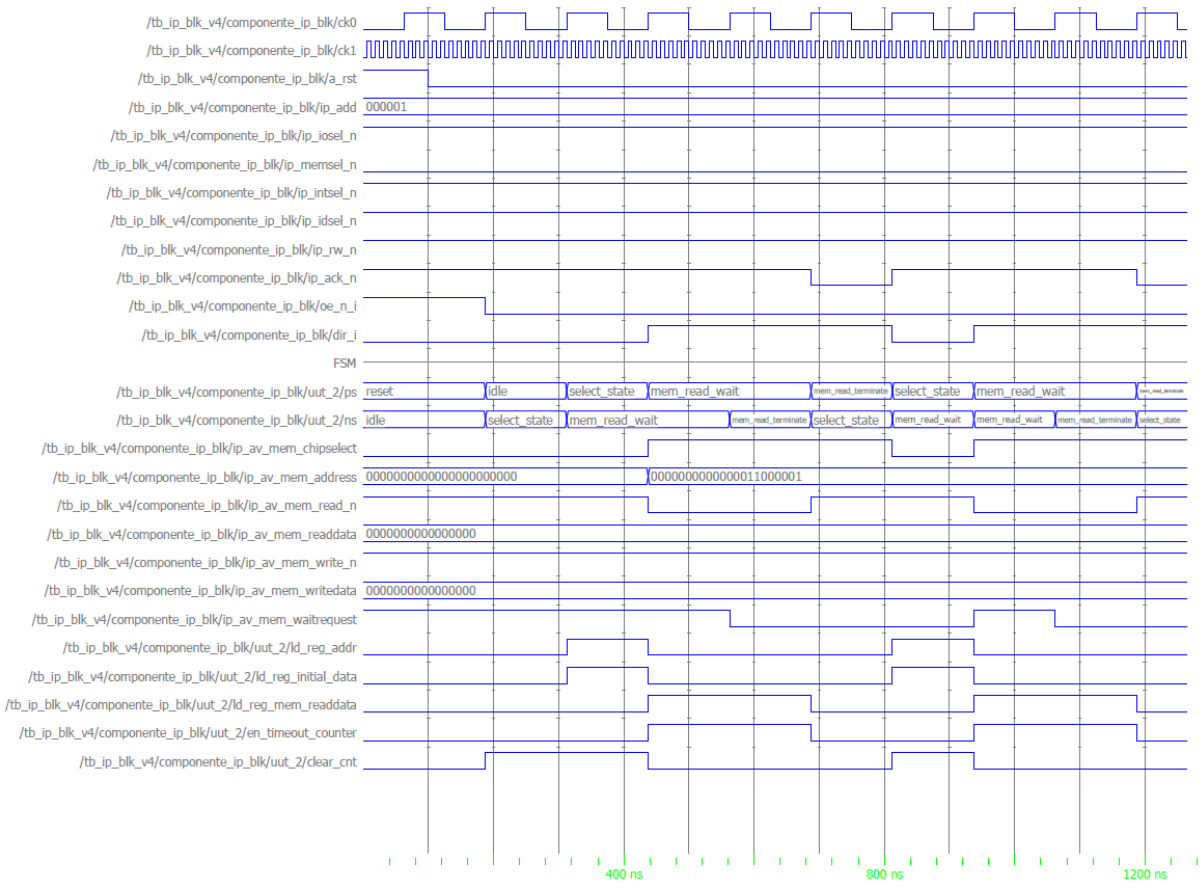


Figura 7.19: Simulazione IP_BRIDGE: lettura in memoria

3. **Lettura da "ID_ROM"**.

   La figura 7.20 mostra la simulazione relativa alla richiesta di lettura dalla memoria ID_ROM. In tal caso la FSM evolve nel ramo a sinistra con riferimento alla figura 7.17. Concettualmente, gli stati e le operazioni svolte in ogni stato sono le stesse che avvengono nel caso di richiesta di lettura da "CHX_REG", con l'unica differenza che, in questo caso, nello stato *ID_TERMINATE* viene mandato all'esterno il dato letto dalla ID_ROM dall'indirizzo selezionato da *ip_address*. In tal caso si è inizializzata la ROM con valori diversi da zero. Il la sequenza 0000000001010000 corrisponde all'*ip_address*=000001 ed è quindi il valore che viene mandato in uscita nello stato *ID_TERMINATE*, in cui si attiva il segnale *ip_ack_n* e si conclude il trasferimento del dato.

Si può dunque concludere che anche questo blocco rispetta il timing imposto dallo standard ANSI/VITA 4-1995 e il comportamento atteso.

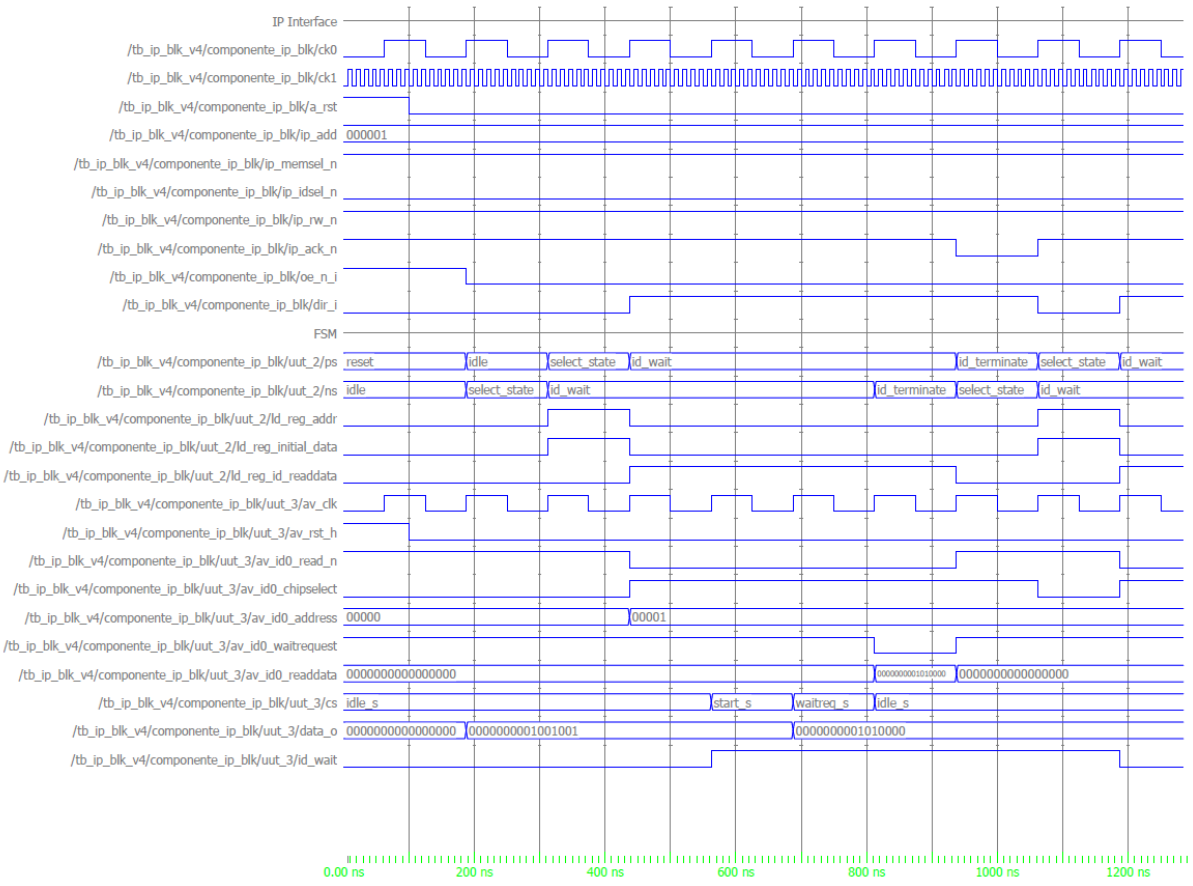


Figura 7.20: Simulazione IP_BRIDGE: lettura da ID_ROM

## 7.6 NIOS_SYSTEM

Si tratta del componente generato da "**Qsys**" che viene poi incluso nel progetto "**Quartus II**". Esso contiene tutti i segnali di interfaccia tra NIOS e le periferiche con cui comunica, oltre alle sorgenti di clock e reset. Nel caso specifico di questo progetto, contiene i segnali di interfaccia *Avalon* scambiati con "AVALON_TO_SPI_BLOCK" e con "CHX_REG" e i segnali SPI scambiati con l'EPCS flash. Inoltre, riceve in ingresso i segnali di stato delle alimentazioni tramite la periferica "PIO_POWER_IN" (riferimento alla figura 7.4). Infine, manda in uscita i segnali di controllo per i LED gialli (che sono anche uscite dell'FPGA), mentre invia al blocco "GEN_REG" (riferimento a figura 7.3)i segnali di stato di tutte le alimentazioni. Infatti, anche quest'area di memoria è accessibile in lettura dal computer, in modo che anche l'utente possa visualizzare lo stato delle alimentazioni.

## 7.7 Sviluppo del firmware per NIOS

Dopo aver realizzato il progetto dell'hardware, è necessario utilizzare un altro tool messo a disposizione di Intel/Altera per poter sviluppare il firmware. A tal fine, è stato utilizzato **"NIOS II Software Build tools for Eclipse (SBT)"**, ovvero un IDE per lo sviluppo del firmware di NIOS, che include un compilatore C/C++. Nel caso specifico, il firmware è stato realizzato in C.
In particolare, quando si crea un'applicazione mediante "**NIOS II SBT**", l'IDE crea due progetti:

- Progetto dell'applicazione: contiene i source e gli header files dell'applicazione che si sta realizzando;
- Progetto "BSP (Board Support Package)": è una libreria specializzata che contiene il codice di supporto specifico per il sistema HW corrispondente, le cui informazioni sono contenute nel file *.sopcinfo* generato da "**Qsys**". Esso include diversi files tra cui il *system.h*, che è l'header file che descrive il sistema hardware e *io.h* che contiene le MACRO definite da Intel/Altera per accedere ai registri delle periferiche.

Infine, il codice sorgente realizzato viene compilato e collegato al "**BSP**" e alle librerie e viene realizzato il file *.elf* (Executable and Linkable Format).
Successivamente, il *.elf* si converte nel formato *.hex* tramite shall di NIOS mediante il seguente comando:
*elf2hex –base=0x00000 –end=0x1FFF –width=32 –input=Mfio_HV_board_Q13.elf –record=4 –output=init_mem.hex*
Questo file è il file di inizializzazione della memoria "PROGRAM_MEM" in figura 7.4. E' dunque necessario compilare il progetto su "**Quartus II**" in modo che Quartus generi il file *.sof* che verrà poi caricato nell'FPGA, che tiene conto anche del file di inizializzazione della memoria precedentemente creato tramite "**NIOS II SBT**" che contiene il firmware per NIOS.

### 7.7.1 Codice C

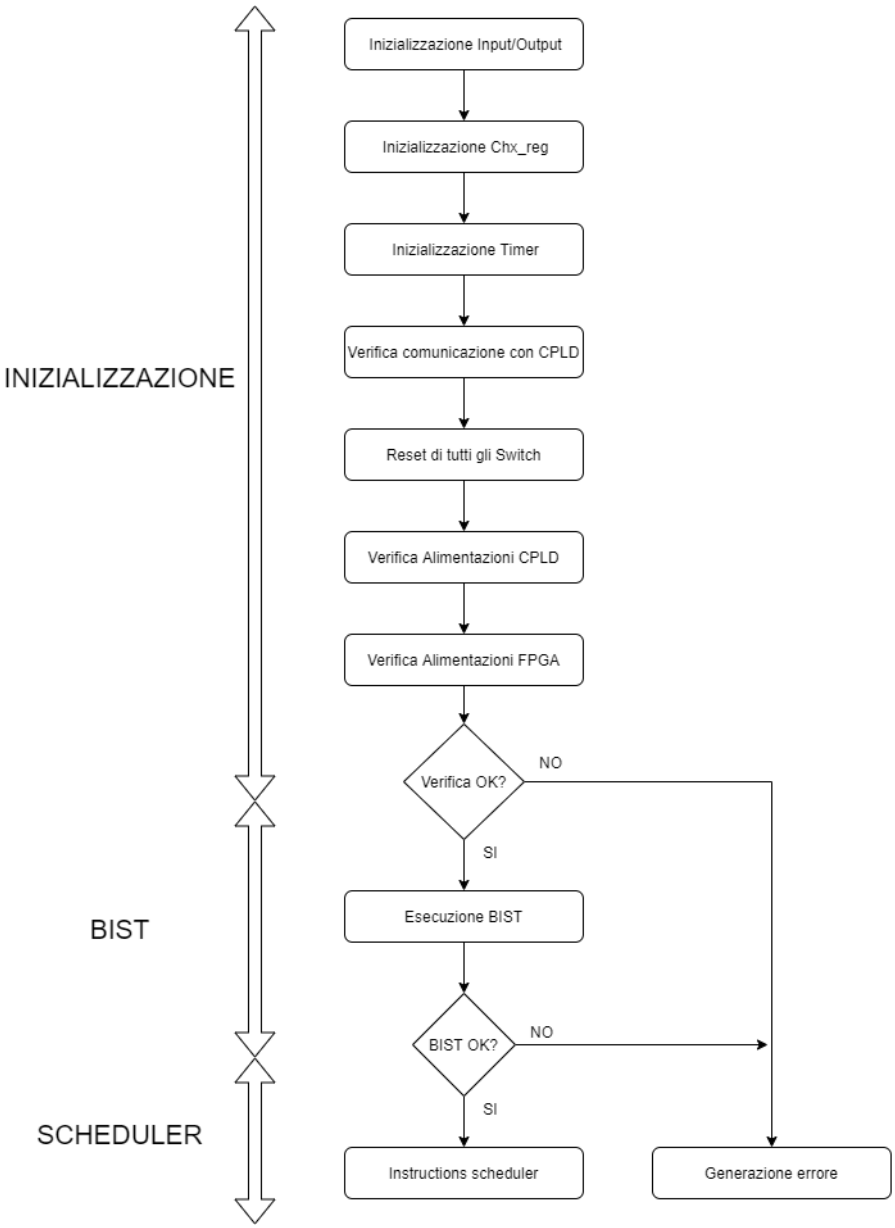


Figura 7.21: Struttura codice C

Il flusso di progetto del codice realizzato è illustrato in figura 7.21. Esso è suddiviso in una parte di inizializzazione eseguita una sola volta e da una parte eseguita in maniera continuativa, che verrà denominata nel seguito "instructions scheduler".

- **Codice di inizializzazione**:

  In questa porzione di codice si utilizzano i led presenti sulla scheda come indicatori del corretto funzionamento dei segnali di stato delle alimentazioni e dei 6 canali della scheda. In particolare, sulla scheda sono presenti 7 led gialli più due coppie di led verdi o rossi. Quelli gialli vengono utilizzati per indicare lo stato delle alimentazioni: alimentazione corretta implica led giallo acceso, alimentazione non funzionante implica led giallo spento. Le due coppie di led verdi e rossi, invece, vengono usate per segnalare errori nelle alimentazioni e malfunzionamento di almeno uno dei 6 canali.

  Per prima cosa, si verifica l'allineamento della risposta del CPLD ed in caso essa non sia allineata, si manda un comando di allineamento. Successivamente, NIOS manda un *comando_GPI* seguito da un comando di lettura della risposta mandata dal CPLD per controllare i segnali di stato delle alimentazioni collegati al CPLD. In caso le alimentazioni siano corrette, si accende il corrispondente led giallo. Questa procedura viene eseguita anche per i segnali di stato delle alimentazioni collegati direttamente all'FPGA.

  Inoltre, nel caso in cui una o più alimentazioni non siano funzionanti, si accende di rosso una delle due coppie di led.

  Successivamente, si inizializzano i registri "CHX_REG" accessibili anche tramite computer e si imposta la modalità *busy*, ad indicare che la scheda non può essere ancora utilizzata in quanto occupata nella procedura iniziale di verifica.

  Se le alimentazioni sono correttamente funzionanti, si procede con l'esecuzione del *BIST (Built in self test)* per la verifica del corretto funzionamento dei canali della scheda. Se almeno uno dei canali non funziona correttamente, si accende di rosso l'altra coppia di led. Inoltre, il risultato del test per ogni canale viene anche scritto nei registri "CHX_REG", accessibili anche dal computer.

  Infine, solo per i canali che risultano funzionanti, si esegue l' "instructions scheduler".

- **BIST(Built in self test)**:

  Come accennato nel precedente paragrafo, il BIST mira a verificare il corretto funzionamento dei sei canali presenti sulla scheda. La figura 7.22 mostra il diagramma di flusso di questo test. In particolare, per ogni canale viene fatta

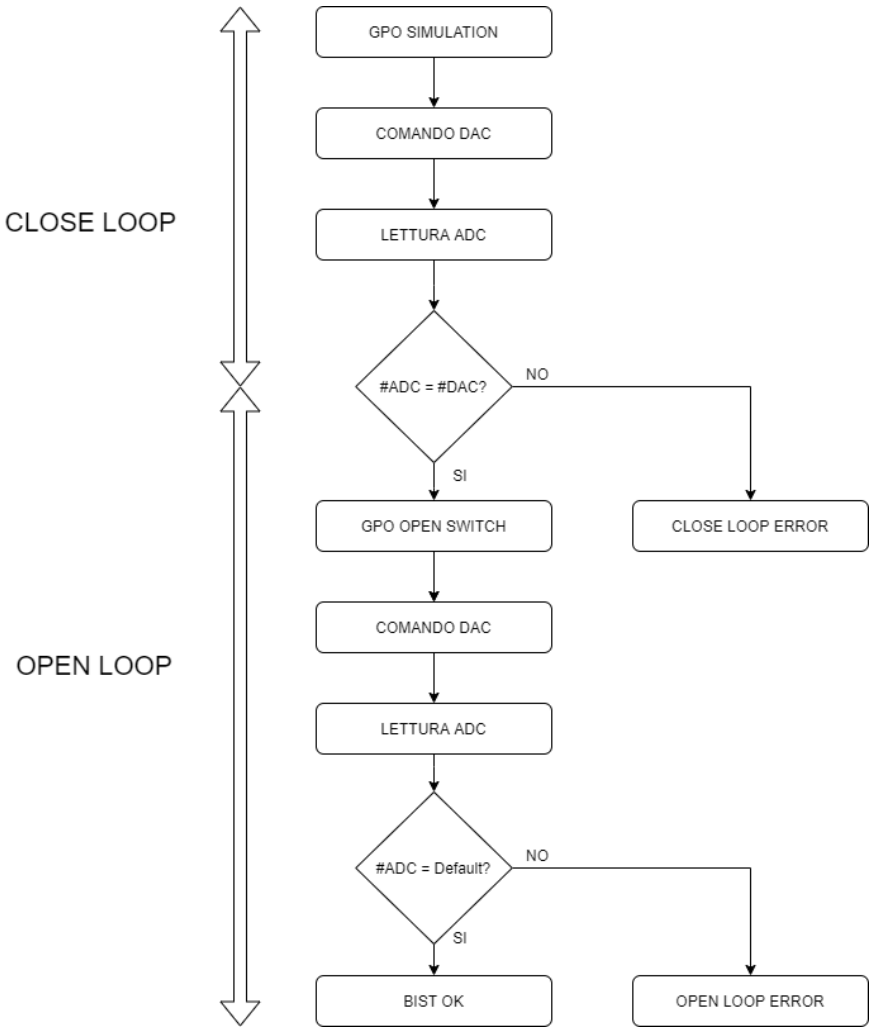


Figura 7.22: Sequenza del BIST

la verifica sia in close loop che in open loop, ovvero impostando gli switches in modalità *simulation* (si chiudono gli switches 3 e 4 con riferimento alla figura 4.6) e impostandoli tutti aperti.

Per quanto riguarda il test in close loop, si invia un *comando_GPO* per impostare gli switches, seguito da un *comando_DAC* che imposti un valore di tensione basso, circa pari a 6V. Si procede poi con un *comando_ADC* seguito da un comando di lettura che permette di leggere il valore restituito dall'ADC. Se questo è compreso nel range $\pm 5\%$ del valore impostato nel DAC, il risultato del test è PASS, altrimenti è FAIL e il canale viene considerato non funzionante.

Per il test in open loop la sequenza di operazioni è identica alla prima, fatta eccezione per il *comando_GPO* che in questo caso imposta gli switches tutti

aperti. Dato che gli switches sono aperti, ci si aspetta che l'ADC restituisca un valore prossimo allo 0. Infatti, il risultato del test in open loop è PASS se il valore restituito dall'ADC è compreso nel range $0,5\%$. In caso questa condizione non si verifichi, il test è FAIL e il canale viene considerato non funzionante.

- **Instructions scheduler**:
  Questa funzione viene eseguita in maniera continuativa solo per i canali che risultano funzionanti in seguito all'esecuzione del BIST. Il diagramma di flusso di questa porzione di codice è sintetizzato in figura 7.23.

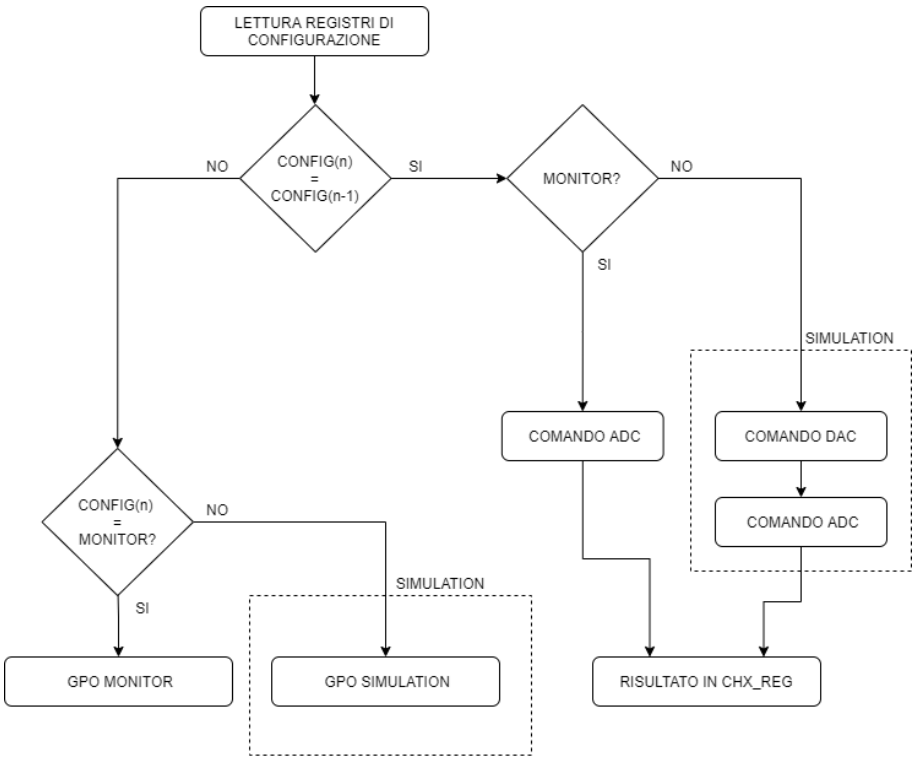


Figura 7.23: Instructions scheduler

Inizialmente si legge la configurazione impostata dall'utente tramite i registri "CHX_REG". Se questa è uguale alla configurazione precedente, si procede col verificare la modalità richiesta (*monitor* o *simulation*). Se *monitor*, si esegue il *comando_ADC* e si scrive il valore restituito dall'ADC in "CHX_REG", in modo che l'utente possa leggerlo tramite computer. Se invece la modalità richiesta è *simulation*, prima del *comando_ADC* si manda il *comando_DAC* per impostare la tensione specificata dall'utente tramite "CHX_REG".

Se, invece, la configurazione richiesta dall'utente non corrisponde a quella precedente, si impostano gli switches in base alla modalità richiesta.

# CAPITOLO 8

# Prove sperimentali

Per verificare il corretto funzionamento del firmware realizzato, si sono effettuate delle prove sperimentali utilizzando la strumentazione messa a disposizione da **Leonardo Velivoli**. Questo capitolo è dedicato alla descrizione delle prove effettuate, ai risultati ottenuti e ai possibili sviluppi del firmware.

## 8.1 Strumentazione utilizzata

Come prima cosa, si presenta di seguito la strumentazione utilizzata per effettuare il test del firmware. Essa ha l'obiettivo di simulare un AGE o un sistema complesso ed è composta da:

1. **Rack di alimentazione**, illustrato in figura 8.1. Si nota che nella parte centrale esso contiene il rack VME 19 inches, con 21 slots VME [1]. Esso ha la stessa struttura e funzione del rack di esempio mostrato in figura 2.2, che rappresenta la Main Processing Unit di un AGE preso ad esempio. In particolare, ai fini della tesi è stata utilizzata una scheda master CPU installata nel primo slot a sinistra del rack, più una scheda slave di servizio detta "carrier" contenente due moduli MFIO HV. Tale scheda è posizionata nello slot numero 7.

[1] Per approfondimenti sul bus VME si rimanda all'appendice B

Figura 8.1: Rack di alimentazione e 21 slots VME

2. **Scheda Carrier**, illustrata in figura 8.2. Essa costituisce la scheda slave VME e rappresenta inoltre la scheda "Carrier" che trasforma l'interfaccia VME nell'interfaccia IP. Le schede MFIO HV sono moduli IP compatibili con lo standard VME e i segnali che l'FPGA delle schede riceve in ingresso sono i segnali IP. Per quanto riguarda la scheda "Carrier", dall'immagine si può vedere che essa è predisposta per contenere fino a 4 moduli MFIO HV, denominati con le lettere A,B,C,D partendo dal basso. Nel caso specifico della tesi, si sono utilizzati due moduli MFIO HV, a cui ci si riferirà nel seguito del documento come *modulo_a* (in basso in figura) e *modulo_c*.

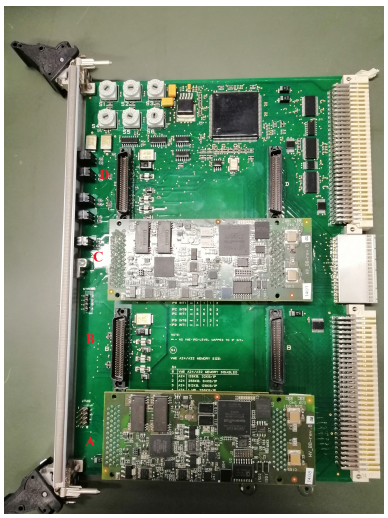

Figura 8.2: Scheda Carrier con due moduli MFIO HV

3. **Computer 1**, utilizzato per programmare il firmware nell'FPGA delle due schede MFIO HV. Il software utilizzato è "**Quartus II 13.1 Web Edition**". In particolare, si è utilizzato il tool "Programmer" all'interno di "**Quartus II**" in modalità "JTAG" e si è programmato il file *.sof* generato da "**Quartus II**" in seguito alla compilazione del progetto.

4. **Computer 2**, collegato tramite cavo Ethernet al rack di alimentazione. Questo computer è stato utilizzato per accedere all'aerea di memoria "CHX_REG" e "GEN_REG" accessibili anche dal processore NIOS. Ciò ha consentito di impostare per un determinato canale della scheda la configurazione desiderata (modalità DC, *monitor* o *simulation*, Dac_value) e di acquisire le misure effettuate dall'ADC del canale selezionato.

5. **Oscilloscopio e multimetro** per verificare la correttezza dei valori della tensione impostata tramite il *Computer 2*.

### 8.1.1 Script dei comandi

Come illustrato nel paragrafo precedente, si è utilizzato il *computer 2* come "user interface", per richiedere specifici comandi alla scheda e per acquisire la risposta da essa restituita.

I comandi sono stati definiti da **Leonardo Velivoli** e sono divisi in due parti: la prima parte contiene i comandi utilizzati per avviare una comunicazione con le due schede MFIO HV e per definire lo spazio di indirizzamento di queste schede e dei registri HV, definiti nell'*Interface Control Data della scheda* [2]. *Questa parte è sempre uguale, a prescindere dal tipo di configurazione che si vuole successivamente impostare per la scheda.* La seconda parte, invece, riguarda i comandi veri e propri che vengono inviati alla scheda.

I comandi per l'avvio della comunicazione tra schede MFIO HV e computer e per l'impostazione degli indirizzi sono mostrati in figura 8.3.

Per visualizzare il contenuto dei registri HV per ogni canale di entrambe le schede, è necessario utilizzare il comando

*d chx_y_addr*, dove *y=a* oppure *y=c* a seconda che si voglia accedere al contenuto dei registri HV del *modulo_a* o del *modulo_c*, mentre x=1,2,...,6 indica il canale che si vuole selezionare.

Infine, per accedere al registro denominato "GEN_REG", che contiene informazioni sullo stato delle alimentazioni, è necessario inviare il seguente comando:

*d module_y_cfg_addr*, dove *y=a* oppure *y=c* a seconda che si voglia accedere all'area di memoria del *modulo_a* o del *modulo_c*.

[2]Si tratta di un documento che fa parte del data package della scheda. Descrizione dettagliata in appendice A

```
telnet 172.31.16.114
ld < io.out
vmeAddress = 0x80000000

//MFIO HV module (module a)
mod_a_addr = vmeAddress + 0x14000000
ch1_a_addr = mod_a_addr + 0x0200
ch2_a_addr = mod_a_addr + 0x0220
ch3_a_addr = mod_a_addr + 0x0240
ch4_a_addr = mod_a_addr + 0x0260
ch5_a_addr = mod_a_addr + 0x0280
ch6_a_addr = mod_a_addr + 0x02A0

//gen_reg address module a
module_a_cfg_addr= mod_a_addr + 0x0100

//MFIO HV module (module c)
mod_c_addr = vmeAddress + 0x15000000
ch1_c_addr = mod_c_addr + 0x0200
ch2_c_addr = mod_c_addr + 0x0220
ch3_c_addr = mod_c_addr + 0x0240
ch4_c_addr = mod_c_addr + 0x0260
ch5_c_addr = mod_c_addr + 0x0280
ch6_c_addr = mod_c_addr + 0x02A0

//gen_reg address module c
module_c_cfg_addr= mod_c_addr + 0x0100

//HV registers from ICD
CH_CMD  = 0x0000
CH_STAT = 0x0002
CH_FRS  = 0x0004
CH_VT   = 0x0006
CH_TT   = 0x0008
CH_VS   = 0x000a
CH_VR   = 0x000c
CH_FS   = 0x000e
CH_FR   = 0x0010
```

Figura 8.3: Comandi impostazione indirizzi

## 8.2 Inizializzazione scheda MFIO HV

Inizialmente si è verificato il corretto funzionamento del codice C di inizializzazione, la cui sequenza è schematizzata in figura 7.21.
In particolare, era attesa la visualizzazione dei 7 led gialli illuminati, in quanto tutte le alimentazioni fornite dal rack sono funzionanti a requisito nominale. Invece, per quanto riguarda le due coppie di led di stato, una coppia era attesa illuminata in verde in quanto indicante lo stato delle alimentazioni, mentre l'altra coppia era attesa illuminata in rosso in quanto le due schede utilizzate hanno almeno un canale non funzionante (in particolare, il *modulo_a* ha i canali 1,4,5,6 non funzionanti e il *modulo_c* ha i canali 1,2,3 non funzionanti). Per questo motivo, nel registro HV denominato "CH_STAT" [3] in figura 8.3, ovvero il secondo registro di ogni canale, ci si aspetta di visualizzare il valore 0x0004 per i canali funzionanti e uno dei valori 0x0101/0x0201/0x0301 per un canale non funzionante rispettivamente in close loop, in open loop e in entrambe le configurazioni.
La figura 8.4 mostra il risultato ottenuto dopo aver eseguito la sequenza di inizializzazione della scheda. Si può notare che il risultato **ottenuto** corrisponde al risultato **atteso**, in quanto i led gialli sono tutti illuminati in entrambe le schede e le coppie di led verdi o rossi sono una verde e una rossa, per indicare rispettivamente che le tensioni di alimentazione funzionano correttamente e che almeno uno dei 6 canali è non funzionante. Questo dimostra il corretto funzionamento della sequenza di inizializzazione. In aggiunta, si è anche visualizzato il registro "CH_STAT" per tutti i canali di entrambi i moduli, il quale contiene il risultato del BIST. A titolo di esempio, si riporta in figura 8.5 il valore ottenuto per il *modulo_a*.
Come si evince dalla figura, il contenuto dei registri HV per ogni canale viene stampato a video in esadecimale. I 6 registri cerchiati in giallo in figura rappresentano il registro "CH_STAT" per ognuno dei 6 canali. Si nota che essi assumono uno dei valori attesi. Gli unici due registri che contengono il valore 0x0004, che significa canale funzionante, sono quelli relativi al canale 2 e 3. Infatti, si tratta degli unici due canali funzionanti per la scheda MFIO HV *modulo_a*.

[3] Riferimento all'appendice A per la descrizione dettagliata del significato di ciascun bit del registro "CH_STATE", definito nell'*Interface Control Data*

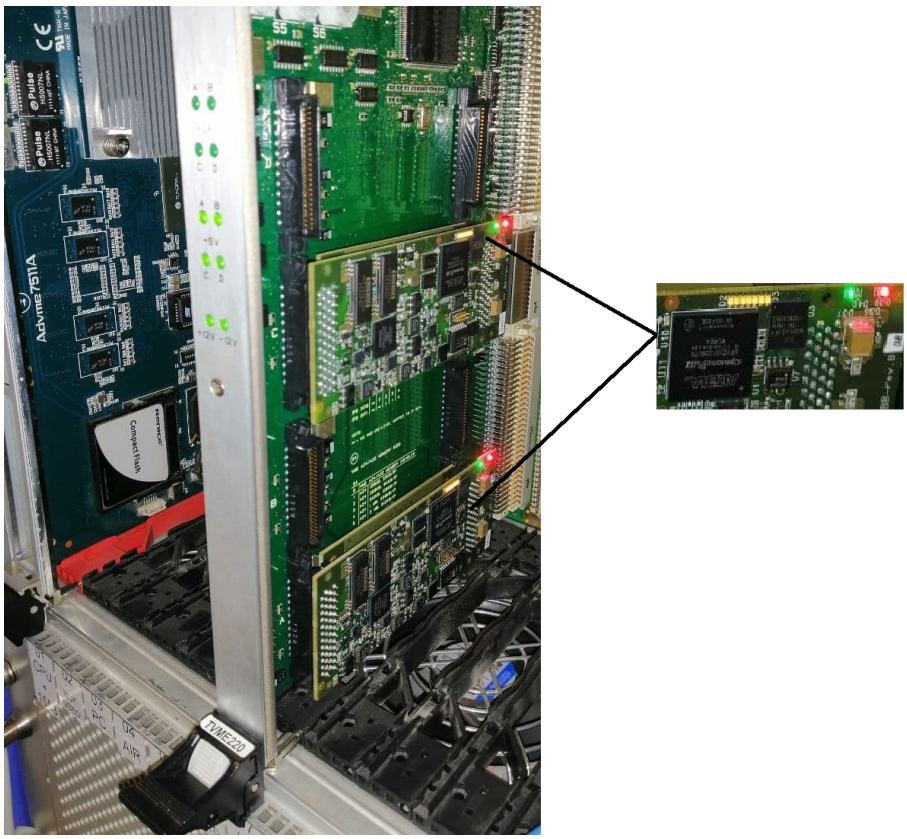

Figura 8.4: Sequenza di inizializzazione su schede MFIO HV

```
-> d ch1_a_addr
NOTE: memory values are displayed in hexadecimal.
0x94000200:   0000 0101 0000 0000 0000 0000 0000 0000
0x94000210:   0000 0000 0000 0000 0000 0000 0000 0000
0x94000220:   0000 0004 0000 0000 0000 0000 000c 0000
0x94000230:   0000 0000 0000 0000 0000 0000 0000 0000
0x94000240:   0000 0004 0000 0000 0000 0000 0000 0000
0x94000250:   0000 0000 0000 0000 0000 0000 0000 0000
0x94000260:   0000 0301 0000 0000 0000 0000 0000 0000
0x94000270:   0000 0000 0000 0000 0000 0000 0000 0000
0x94000280:   0000 0301 0000 0000 0000 0000 0000 0000
0x94000290:   0000 0000 0000 0000 0000 0000 0000 0000
0x940002a0:   0000 0301 0000 0000 0000 0000 0000 0000
0x940002b0:   0000 0000 0000 0000 0000 0000 0000 0000
```

Figura 8.5: Risultato BIST per *modulo_a*

Da queste prove sperimentali si può dunque concludere il corretto funzionamento della sequenza di inizializzazione, in linea con i risultati attesi.

## 8.3 Instructions Scheduler

Dopo aver verificato il corretto funzionamento della sequenza di inizializzazione, si è proceduto col verificare il comportamento dell' *instructions scheduler*.
In particolare, si sono testate le seguenti configurazioni su entrambe le schede MFIO HV:

- Modalità *monitor*: Il setup di misura utilizzato è schematizzato in figura 8.10, ad eccezione del collegamento illustrato con la linea tratteggiata in arancione, assente in questo caso.
  Inizialmente è necessario avviare la comunicazione tra schede MFIO HV e computer 2 inviando tramite il prompt dei comandi i comandi definiti in figura 8.3. In questo esempio sono riportati i comandi che permettono di impostare la modalità monitor nel canale 2 del *modulo_a* e il fondo scala:

  1. *prova (ch2_a_addr + CH_FRS, 0x7FFF)*
  2. *prova (ch2_a_addr + CH_CMD, 0x0004)*

  Il primo comando imposta il fondo scala a metà della dinamica, per considerare solo i valori di tensione positivi. Infatti, la dinamica totale è compresa tra $\pm 50V$, mentre in tal caso si considerano soltanto i valori di tensione positiva compresa tra 0V e 50V.
  Il secondo comando, invece, imposta il valore esadecimale 0x0004 nel registro "CH_CMD", in giallo in figura 8.6. Questo è utilizzato per richiedere la modalità "enable field", che permette di abilitare il segnale di tensione verso l'esterno, in modo che possa essere impostata una tensione di riferimento tramite un'alimentazione esterna o tramite un'altra scheda MFIO HV.
  In figura 8.6, il registro "CH_VR" cerchiato in verde riporta il valore 0x000C in quanto, nella prova sperimentale riportata come esempio, non si era collegata alcuna alimentazione alla scheda. Infatti, il valore letto è pari a 0 (al netto della tolleranza di misura) e, in particolare, è compreso nel range $(0 \pm 0.05)$V.

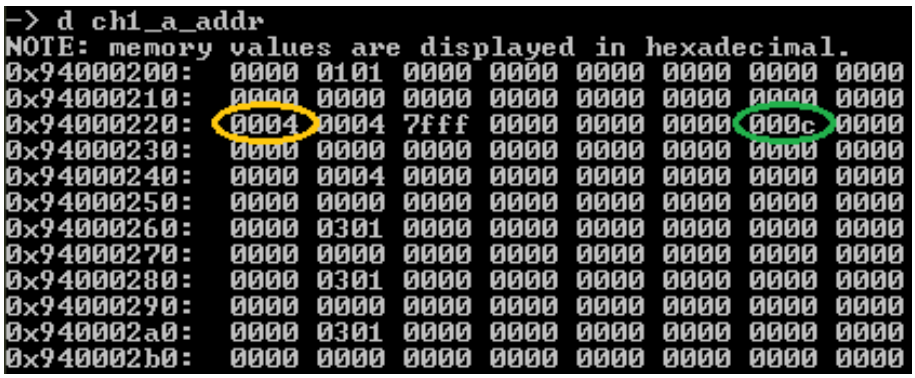


Figura 8.6: Test modalità monitor

- Modalità *simulation*:Il setup di misura utilizzato è schematizzato in figura 8.10, ad eccezione del collegamento illustrato con la linea tratteggiata in arancione, non effettuato per questo test.
  Anche in questo caso, prima di inviare i comandi specifici per impostare questa configurazione alla scheda e per impostare un valore di tensione sul DAC del canale selezionato, è necessario inviare i comandi illustrati in figura 8.3. Dopo aver avviato la comunicazione tra schede MFIO HV e aver impostato gli indirizzi, si inviano i comandi per impostare la modalità
  *monitor + simulation*. Si riportano di seguito i comandi per impostare questa modalità e per impostare 5V sul canale 2 del *modulo_a* :

  1. *prova (ch2_a_addr + CH_FRS, 0x7FFF)*
  2. *prova (ch2_a_addr + CH_VS, 0x1999)*
  3. *prova (ch2_a_addr + CH_CMD, 0x0006)*

  In particolare, il significato dei comandi è il seguente:

  1. Si imposta il fondo scala a metà dinamica
  2. Si impostano 5V sul DAC del canale 2. 5V corrisponde al valore esadecimale 0x1999 secondo la seguente relazione, descritta nel documento *Interface Control Data*[4] della scheda:

$$\frac{5V \cdot (2^{16} - 1)}{50V}$$

[4]Approfondimento in appendice A

3. Si abilitano la modalità DC e la modalità *monitor + simulation* che permette di erogare la tensione anche verso l'esterno, in modo che possa essere visualizzata tramite oscilloscopio o multimetro e che possa essere utilizzata come tensione di riferimento per un'altra scheda.

Dopo aver impostato questa configurazione, si invia il comando *d ch1_a_addr* per visualizzare il contenuto dei registri tutti i canali. In particolare, la terza e la quarta riga contengono i registri del canale 2. Ci si aspetta di visualizzare "CH_CMD"=0x0006 per indicare la configurazione impostata, "CH_STAT"=0x0004 in quanto il canale 2 è funzionante, "CH_VR"=0x7FFF che indica metà della dinamica, "CH_VS"=0x1999 in quanto contiene la tensione impostata, ed infine "CH_VR" con un valore prossimo a 0x1999. Infatti, "CH_VR" contiene il valore restituito dall'ADC che ha ai suoi ingressi la tensione di uscita del DAC, impostata a 5V col comando 2.
La figura 8.7 mostra il contenuto dei registri col colore corrispondente.

```
-> prova (ch2_a_addr + CH_FRS, 0x7FFF)
value = 78625416 = 0x4afba88
-> prova (ch2_a_addr + CH_VS,  0x1999)
value = 78625416 = 0x4afba88
-> prova (ch2_a_addr + CH_CMD, 0x0006)
value = 78625416 = 0x4afba88
-> d ch1_a_addr
NOTE: memory values are displayed in hexadecimal.
0x94000200:   0000 0101 0000 0000 0000 0000 0000 0000
0x94000210:   0000 0000 0000 0000 0000 0000 0000 0000
0x94000220:   0006 0004 7fff 0000 0000 1999 196a 0000
0x94000230:   0000 0000 0000 0000 0000 0000 0000 0000
0x94000240:   0000 0004 0000 0000 0000 0000 0000 0000
0x94000250:   0000 0000 0000 0000 0000 0000 0000 0000
0x94000260:   0000 0301 0000 0000 0000 0000 0000 0000
0x94000270:   0000 0000 0000 0000 0000 0000 0000 0000
0x94000280:   0000 0301 0000 0000 0000 0000 0000 0000
0x94000290:   0000 0000 0000 0000 0000 0000 0000 0000
0x940002a0:   0000 0301 0000 0000 0000 0000 0000 0000
0x940002b0:   0000 0000 0000 0000 0000 0000 0000 0000
```

Figura 8.7: Test modalità monitor + simulation

Infine, si è inviato il comando *d ch1_a_addr* per 31 volte consecutive, in modo da prendere 31 misure della tensione 5V impostata. Si è successivamente realizzato un grafico per verificare che le misure ottenute fossero comprese nel range $(5 \pm 0.25)$V. Il grafico ottenuto è riportato in figura 8.8. Inoltre, in questo grafico sono state riportate anche 31 misure che erano state effettuate col

firmware originale. Da un rapido confronto visivo, si evince che sia i valori delle misure ottenute col firmware originale che quelli ottenuti col nuovo firmware sono compresi nel range $(5 \pm 0.25)V$. Va tuttavia sottolineato come dal grafico si possa osservare che le misure ottenute col nuovo firmware risultino più vicine al valore di tensione nominale impostato.

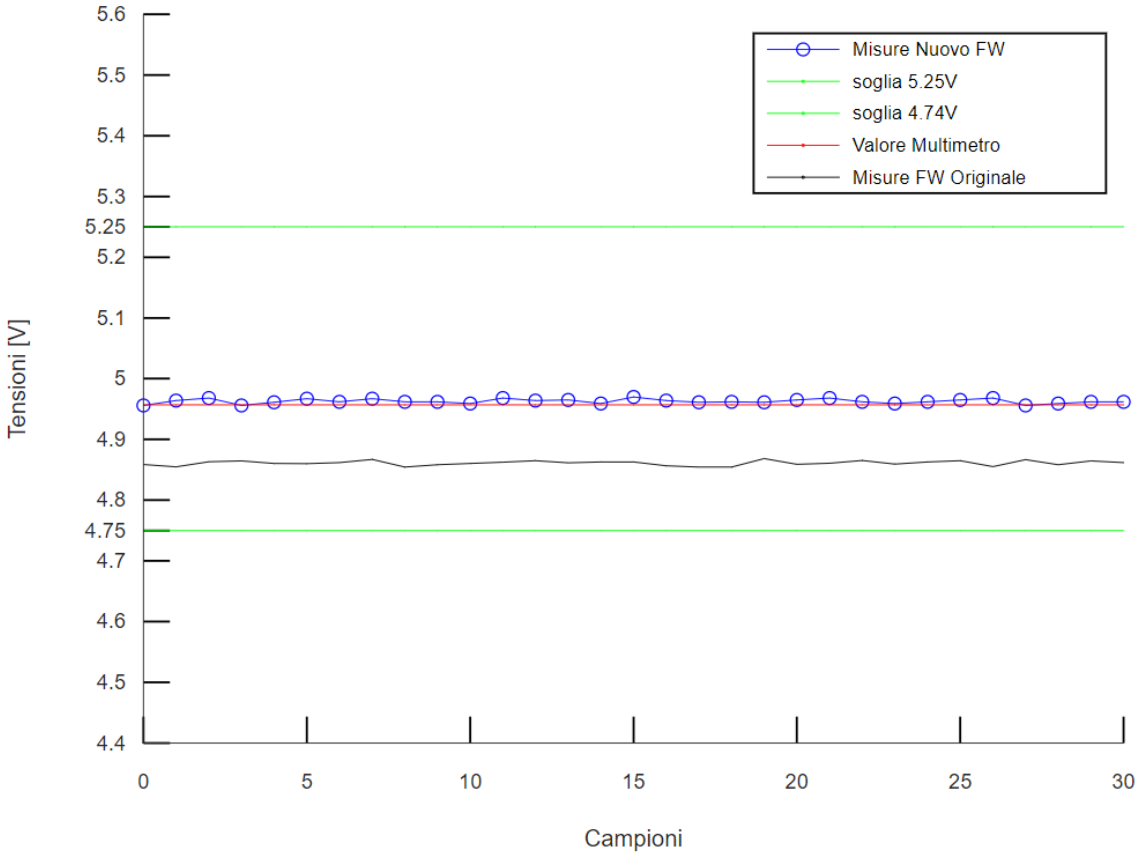


Figura 8.8: Plot 31 misure 5V

Infine, si è ripetuta la sequenza di comandi 1-3 per impostare la modalità *monitor + simulation* sul canale 2 del *modulo_a*, impostando 28V nel comando 2. Infatti, i 28V rappresentano proprio la tensione di interesse, in quando è l'alimentazione per la maggior parte dell'elettronica di bordo nei velivoli.
Applicando la formula precedentemente citata, 28V corrispondono al valore esadecimale 0x8F5B. Anche in tal caso, si sono poi effettuate 31 misurazioni, inviando il comando *d ch1_a_addr* per 31 volte consecutive. Esse sono state elaborate nel grafico riportato in figura 8.9. Anche in questo caso, si evince che

le misure ottenute rientrano nel range $(28V \pm 1.4)V$ e sono molto prossime al valore impostato. Si può dunque concludere che il firmware realizzato funziona correttamente, svolge il compito richiesto (misura e generazione di tensioni in DC) e rispetta i requisiti.

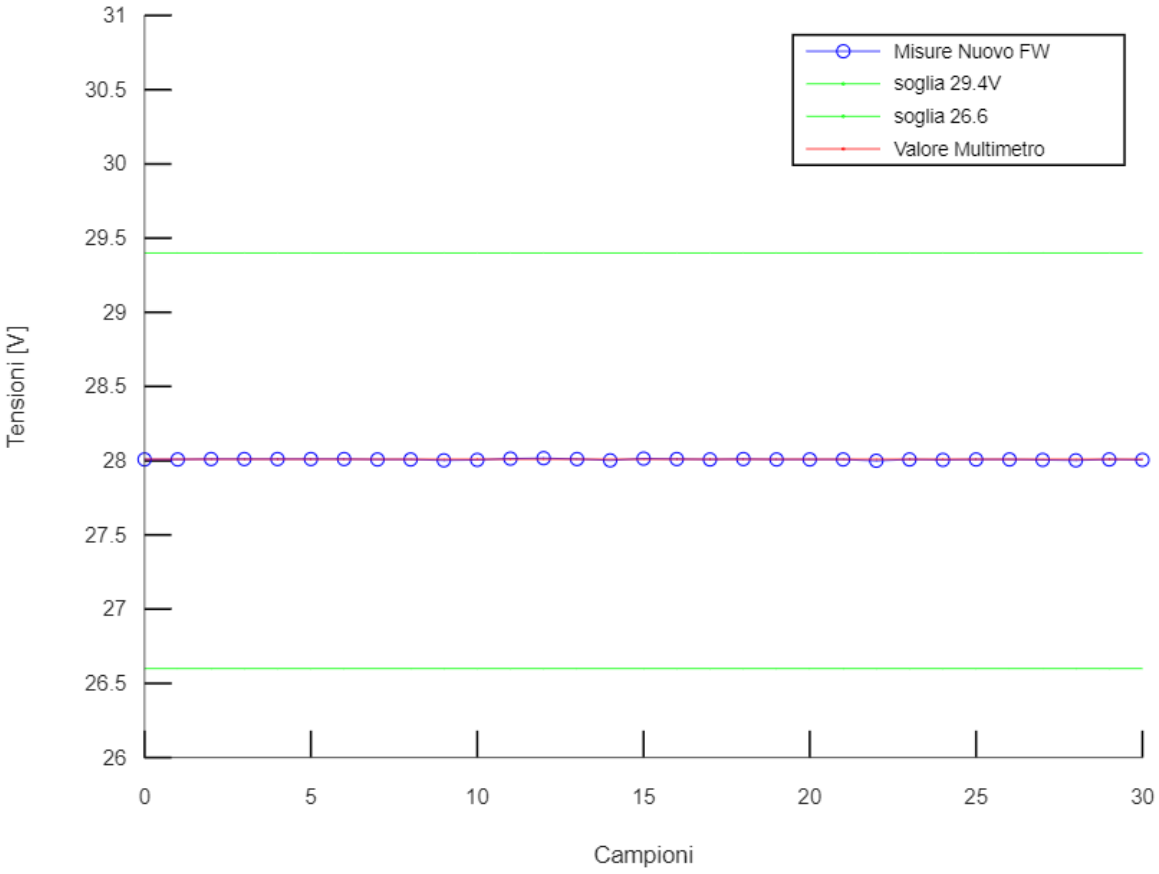


Figura 8.9: Plot 31 misure 28V

Dopo aver verificato su entrambe le schede in maniera indipendente il corretto funzionamento delle modalità monitor e simulation, si è utilizzata una scheda in modalità monitor e l'altra in modalità monitor + simulation. In questo modo, la scheda impostata in modalità monitor + simulation è stata usata per generare una tensione collegata in input alla scheda impostata in modalità monitor.
Il setup di misura utilizzato per effettuare il test con le schede collegate, è schematizzato in figura 8.10. La linea tratteggiata in arancione in figura indica il collegamento effettuato tra l'output di una scheda (*modulo_a*) e l'input dell'altra (*modulo_c*).

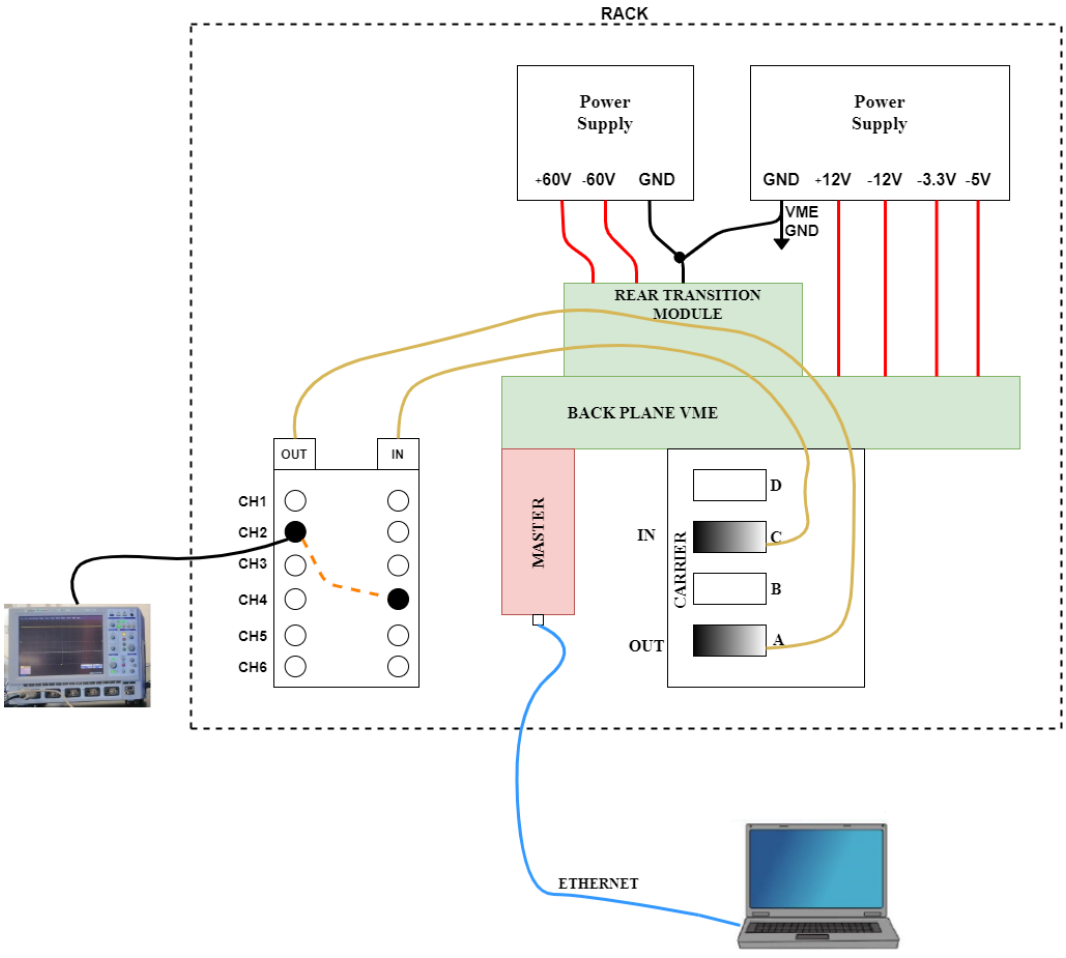


Figura 8.10: Schema setup di misura schede collegate

A titolo di esempio, si riportano di seguito i risultati ottenuti impostando il *modulo_a* in monitor + simulation sul canale 2 in modo che eroghi 5V e il *modulo_c* in monitor sul canale 4. Per poter fornire al *modulo_c* la tensione generata dal *modulo_a*, si sono

effettuati i collegamenti nel rack schematizzati in figura 8.10 e mostrati in figura 8.11, la quale illustra il setup di misura utilizzato nei laboratori di **Leonardo Velivoli** per svolgere il test in analisi.

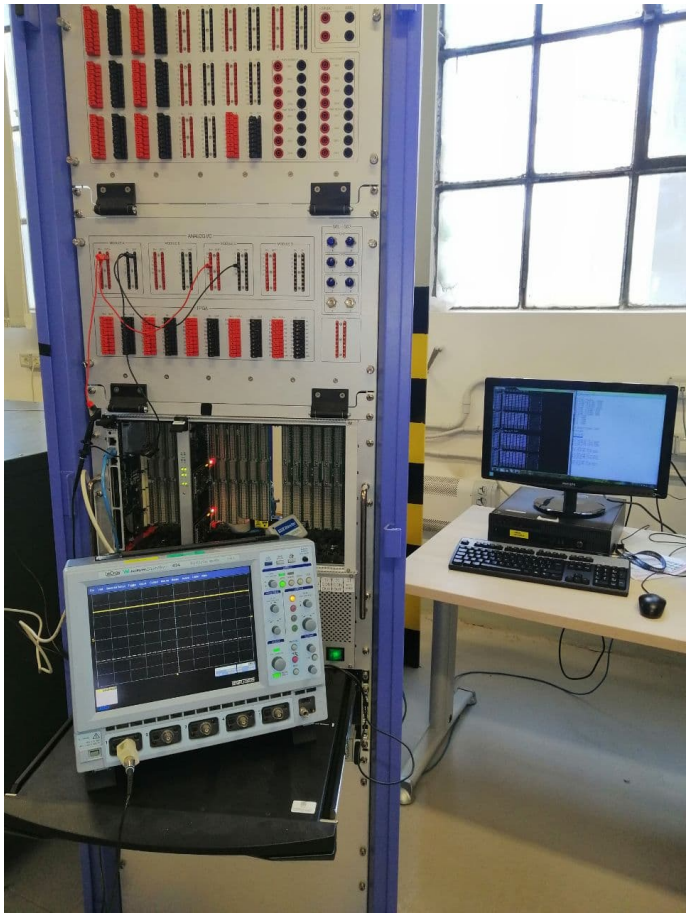

Figura 8.11: Setup di misura schede collegate

Dalle figure 8.10 e 8.11 è inoltre possibile osservare che il canale 2 del *modulo_a* è stato anche collegato all'oscilloscopio per verificare che la tensione erogata corrispondesse a quella impostata pari a 5V.
La linea in giallo nell'oscilloscopio rappresenta 5V DC, considerando che nell'oscilloscopio sono stati impostati 2V/div e che l'offset è pari a 0 V/div.
Si riporta, inoltre, il contenuto dei registri di ogni canale per le due schede *modulo_a* e *modulo_c*.
La figura 8.12 mostra il contenuto dei registri per i 6 canali del *modulo_a*. In particolare, quelli nelle righe 3 e 4 sono relativi al secondo canale. Dalla figura si può osservare che "CH_VS"=0x1999 contiene la tensione impostata pari a 5V, mentre

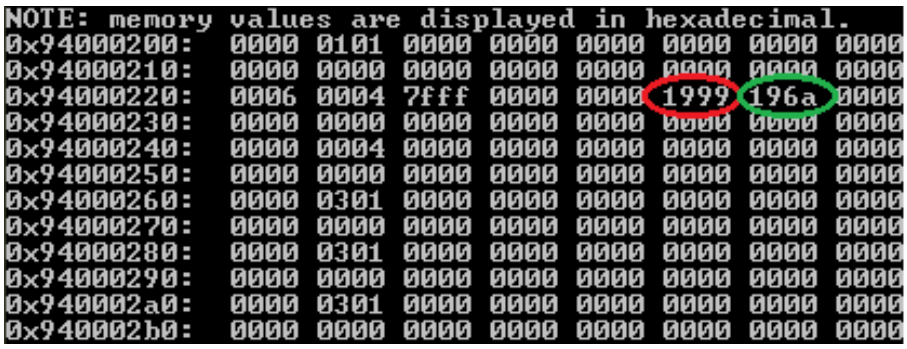


Figura 8.12: Test schede collegate: registri *modulo_a*

"CH_VR=0x196A", contiene il valore di tensione restituito dal canale 2 della stessa scheda. Questo è compreso nel range $(5 \pm 0.25)$V.

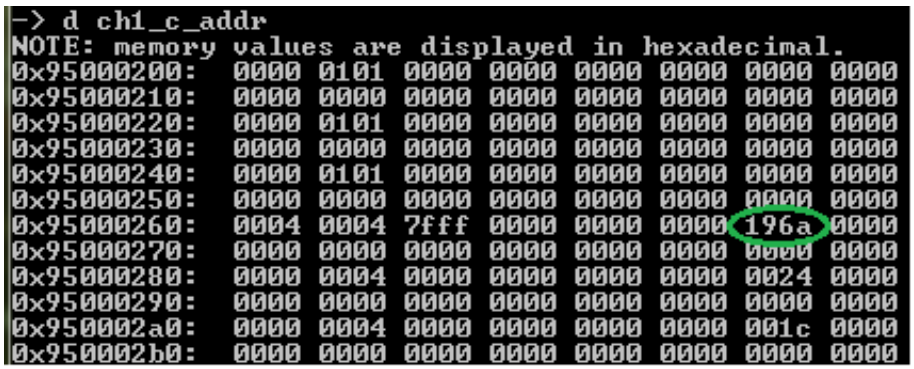


Figura 8.13: Test schede collegate: registri *modulo_c*

In figura 8.13, invece, è riportato il contenuto dei registri relativi al *modulo_c* impostato in modalità monitor sul canale 4. I registri relativi a questo canale sono contenuti nelle righe 7 e 8.
Dall'immagine riportata si evince che "CH_VR=0x196A". Nel caso di questa prova sperimentale si è ottenuto esattamente lo stesso valore di tensione misurato anche dal *modulo_a*. Il valore misurato è corretto in quanto, tramite i collegamenti effettuati nel rack, questa scheda riceve in ingresso la tensione 5V erogata dal *modulo_a*.
Si può pertanto constatare il corretto funzionamento del sistema anche nel caso di utilizzo di schede collegate tra loro.

# CAPITOLO 9

# Conclusioni

Da un'analisi critica delle prove sperimentali effettuate presso i laboratori di **Leonardo Velivoli**, si può constatare il corretto funzionamento della struttura hardware e del firmware sviluppati.
In particolare, la fase di inizializzazione del codice C è in grado di rilevare eventuali anomalie nelle tensioni di alimentazione ed eventuali mal funzionamenti nei 6 canali disponibili sulla scheda. E' anche in grado, inoltre, di segnalare tali malfunzionamenti mediante un codice di errore "scritto" in opportuni registri che, come descritto in precedenza, sono accessibili anche dal mondo esterno e visualizzabili dall'utente.
In aggiunta, si può osservare che anche l'"instruction scheduler" svolge il compito richiesto e rispetta le specifiche. Infatti, è in grado di leggere la configurazione impostata dall'utente tramite opportuni registri, elaborare il contenuto dei registri in modo automatico e inviare alla scheda gli opportuni comandi per soddisfare la richiesta giunta dal mondo esterno. Come descritto nel capitolo precedente, le due possibili configurazione di interesse sono :

- Modalità *monitor*: in questo caso si richiede che la scheda sia in grado di acquisire segnali in DC. Le prove sperimentali illustrate in figura 8.6, 8.7, 8.12 e 8.13 dimostrano che il firmware sviluppato è in grado di supportare questa configurazione e di realizzare misure nel range del 5% rispetto al valore nominale.

- Modalità *simulation*: in questo caso si richiede che la scheda sia in grado di stimolare una tensione tramite i componenti DAC e di acquisirla tramite i com-

ponenti ADC presenti su scheda. Si specifica che nelle prove sperimentali riportate nel capitolo precedente, si è associata a questa modalità anche la modalità *monitor* per poter essere in grado di visualizzare la tensione impostata anche tramite oscilloscopio e multimetro digitale. Le figure 8.7, 8.12 8.11 dimostrano il corretto funzionamento del firmware anche per questa configurazione.

E' pertanto possibile concludere che il progetto hardware e il firmware realizzato svolgono correttamente l'acquisizione e /o stimolazione di segnali in DC e sono più semplici ed essenziali rispetto a quelli inizialmente installati sulle schede MFIO HV: in quanto tali, ammettono anche una migliore manutenibilità.
Infine, per quanto riguarda gli sviluppi futuri, essi riguardano una possibile estensione del firmware realizzato in modo che sia in grado di supportare anche la gestione di segnali elettrici in alternata. Infatti, come menzionato nel capitolo 2, l'altra tensione di alimentazione utilizzata per fornire alimentazione a una parte degli apparati avionici è costituita dai 115V AC @400Hz.

# APPENDICE A

# Interface Control Data (ICD)

Come accennato durante la trattazione, si tratta di un documento che fa parte del "data package" della scheda.
In particolare, contiene la descrizione dettagliata di tutti i registri delle schede MFIO, esplicitando il significato di ciascun bit di ogni registro e l'offset del registro rispetto all'indirizzo base VME della scheda.
Di seguito si riporta la descrizione dei registri utilizzati ai fini della tesi, ovvero i registri del modulo MFIO HV e il registro utilizzato all'interno dell'area di memoria denominata "GEN_REG", che è condivisa da tutte le tipologie di schede MFIO menzionate nel capitolo 3. In particolare, il registro utilizzato è denominato "GR_STATE" ed è utilizzato per salvare in memoria il risultato ottenuto sul test delle tensioni di alimentazione. Pertanto, ai fini della tesi, i soli bit utilizzati all'interno di questo registro sono i bit in posizione 13/12/11/6/5/4/3/1/0. Inoltre, si precisa che in tabella A.1 si è riportato l'offset rispetto all'indirizzo base VME della scheda. In particolare, l'area di memoria "GEN_REG" ha un offset pari a 0x0100 rispetto all'indirizzo base VME della scheda MFIO HV, mentre il registro "GR_STATE" ha offset pari a 0x0002 rispetto all'indirizzo base di "GEN_REG".

## A.1 GEN_REG

| **Register Name** | **GR_STATE** | | |
|---|---|---|---|
| **Offset** | **0x0102** | | |
| **Field** | **Description** | **Dim.[bit]** | **Position** |
| GR_FLASH_STATE | flash update state | 1 | [15] |
| GR_CAL_STATE | calibration data download state | 1 | [14] |
| GR_MODE | module operative state | 1 | [13] |
| GR_60NVI_FAULT | 1=-60V fault<br>0=-60V ok | 1 | [12] |
| GR_60VI_FAULT | 1=60V fault<br>0=60V ok | 1 | [11] |
| GR_PWMCH4_FAULT | PWM CH4 state | 1 | [10] |
| GR_PWMCH3_FAULT | PWM CH3 state | 1 | [9] |
| GR_PWMCH2_FAULT | PWM CH2 state | 1 | [8] |
| GR_PWMCH1_FAULT | PWM CH1 state | 1 | [7] |
| GR_12NVI_FAULT | 1=-12V fault<br>0=-12V ok | 1 | [6] |
| GR_12VI_FAULT | 1=12V fault<br>0=12V ok | 1 | [5] |
| GR_33VI_FAULT | 1=3.3V fault<br>0=3.3V ok | 1 | [4] |
| GR_12V_FAULT | 1=12V fault<br>0=12V ok | 1 | [3] |
| NU | | 1 | [2] |
| GR_5V_FAULT | 1=5V fault<br>0=5V ok | 1 | [1] |
| GR_PLL_UNLOCK | local clock PLL state | 1 | [0] |

Tabella A.1: ICD: Registro GR_STATE in GEN_REG

## A.2 Registri HV

| **Register Name** | **CH_CMD** | | |
|---|---|---|---|
| **Offset** | **0x0000** | | |
| **Field** | **Description** | **Dim.[bit]** | **Position** |
| CH_NU | Not Used | 8 | [15:8] |
| CH_CONFIG | 0x0=DC(default)<br>0x01=AC 400 | 4 | [7:4] |
| CH_IBIT_REQ | 0=normal operation (default) | 1 | [3] |
| CH_FIELD_CTRL | 0=no connection to field<br>1=connection to field | 1 | [2] |
| CH_MODE_CTRL | 0=only monitor mode (default)<br>1=monitor + simulation mode | 1 | [1] |
| CH_ST_TOGGLE | 1=start toggling | 1 | [1] |

Tabella A.2: ICD: Registro CH_CMD

| **Register Name** | **CH_STATE** | | |
|---|---|---|---|
| **Offset** | **0x0002** | | |
| **Field** | **Description** | **Dim.[bit]** | **Position** |
| CH_ERR_CODE | 0x00=NO ERR.<br>0x01=CLOSE LOOP ERR.<br>0x02=OPEN LOOP ERR. | 8 | [15:8] |
| CH_NU | Not Used | 5 | [7:3] |
| CH_MODE_STATE | 0=CH. NOT ACTIVE<br>1=CH. ACTIVE | 1 | [2] |
| CH_BIST_STS | 0=BIST TERMINATED<br>0=BIST IN PROGRESS | 1 | [1] |
| CH_BIST_RES | 0=CH. OK<br>1=CH. FAULT | 1 | [0] |

Tabella A.3: ICD: Registro CH_STATE

| **Register Name** | **CH_FRS** | | |
|---|---|---|---|
| **Offset** | **0x0004** | | |
| **Field** | **Description** | **Dim.[bit]** | **Position** |
| CH_FRS | Set full scale | 16 | [15:0] |

Tabella A.4: ICD: Registro CH_FRS

| **Register Name** | **CH_VT** | | |
|---|---|---|---|
| **Offset** | **0x0006** | | |
| **Field** | **Description** | **Dim.[bit]** | **Position** |
| CH_VT | Set voltage output value | 16 | [15:0] |

Tabella A.5: ICD: Registro CH_VT

| **Register Name** | **CH_TT** | | |
|---|---|---|---|
| **Offset** | **0x0008** | | |
| **Field** | **Description** | **Dim.[bit]** | **Position** |
| CH_TT | Set toggle time value | 16 | [15:0] |

Tabella A.6: ICD: Registro CH_TT

| **Register Name** | **CH_VS** | | |
|---|---|---|---|
| **Offset** | **0x000A** | | |
| **Field** | **Description** | **Dim.[bit]** | **Position** |
| CH_VS | Set voltage value | 16 | [15:0] |

Tabella A.7: ICD: Registro CH_VS

| **Register Name** | **CH_VR** | | |
|---|---|---|---|
| **Offset** | **0x000C** | | |
| **Field** | **Description** | **Dim.[bit]** | **Position** |
| CH_VR | Read voltage monitor | 16 | [15:0] |

Tabella A.8: ICD: Registro CH_VR

| **Register Name** | **CH_FS** | | |
|---|---|---|---|
| **Offset** | **0x000E** | | |
| **Field** | **Description** | **Dim.[bit]** | **Position** |
| CH_FS | Frequency set | 16 | [15:0] |

Tabella A.9: ICD: Registro CH_FS

| **Register Name** | **CH_FR** | | |
|---|---|---|---|
| **Offset** | **0x0010** | | |
| **Field** | **Description** | **Dim.[bit]** | **Position** |
| CH_FS | Period monitor | 16 | [15:0] |

Tabella A.10: ICD: Registro CH_FR

# APPENDICE B

# VME Bus

Il Bus VME (Versa Module Eurocard Bus) [14] [15] è un bus di comunicazione originariamente sviluppato negli anni '80 da Motorola per la famiglia di microprocessori 68000. Esso divenne uno standard ANSI/IEEE 1014-1987, venne adottato per diverse applicazioni in ambito industriale e militare ed è attualmente ancora ampiamente utilizzato.

E' basato su moduli di dimensione Eurocard, un formato standard Europeo. Essi possono essere montati in un rack 19-inch, contentente 21 slots VME. L'altezza delle schede viene misurata in *rack units 'U'*, con la conversione 1U=1.75in=44.45mm. In riferimento all'AGE descritto nel paragrafo 2.1, il rack ha la dimensione 19 inch e le schede "Carrier"su cui sono montati i moduli MFIO HV hanno il formato standard 6U=266.7mm. La scheda carrier VME utilizzata ai fini della tesi è mostrata in figura B.1.

Il bus di comunicazione VME è basato su un tipo di architettura master-slave, con un sistema di interrupt asincrono e uno schema memory mapped, in cui ogni dispositivo può essere visto come un indirizzo.

Il VME utilizza due connettori Eurocard denominati P1 e P2 (mostrati in figura B.1), utilizzati per implementare i pin del bus dati, del bus indirizzi e dei segnali di controllo. Il bus è controllato da un insieme di nove linee denominate *arbitration bus*. Inoltre, tutte le comunicazioni sono gestite da una scheda denominata *arbiter module*, posizionata nel primo slot del rack 19 in (con riferimento alla figura 2.2, l'*arbiter module* è la prima scheda a sinistra in figura). Oltre alla scheda *arbiter*, dal punto di

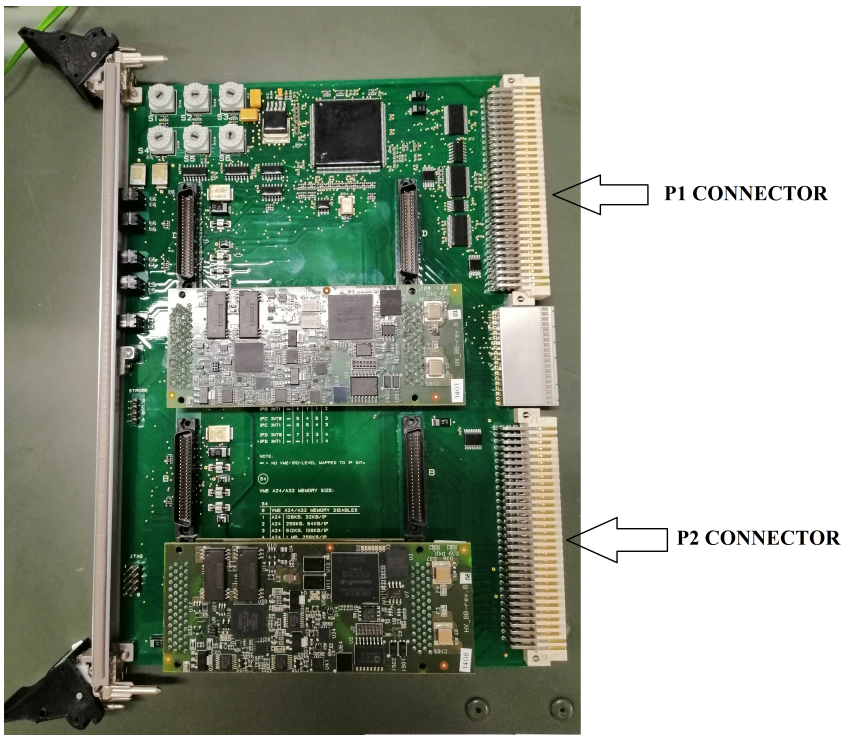


Figura B.1: Esempio scheda VME formato 6U

vista logico le schede possono appartenere a una delle seguenti classi:

- **Master**: è il modulo che inizia un trasferimento;
- **Slave**: è il modulo che risponde alla scheda master;
- **Interrupter**: è la scheda che può fare una richiesta di interrupt (generalmente è uno slave);
- **Interrupt handler**: è il modulo che riceve e gestisce tutte le richieste di interrupts.

Una scheda master può richiedere di pilotare il bus pilotando una delle quattro linee di *Bus Request*(BR0-BR3) col valore logico basso. L'*arbiter module* campiona le richieste in attesa sulle linee BR0-BR3 e, con un meccanismo di priorità, decide quale scheda può pilotare il bus una volta che il bus è libero. Si parte da BR3 con priorità maggiore per arrivare a BR0 che ha la priorità più bassa. Se due master usano lo stesso livello di *bus request*, quello più vicino allo slot 1 (in cui si trova l'*arbiter module*) ha intrinsecamente priorità maggiore. Quando l'*arbiter module* determina

quale richiesta deve essere servita, attiva il corrispondente segnale di riconoscimento (segnale *grant_signal*) sulla linea corrispondente alla linea *Bus Request* (BG0-BG3). A questo punto, la scheda master che pilota il bus ha il compito di pilotare la linea *Bus Busy* **(BBSY*)** col valore logico basso, per indicare che il bus è in uso. Fino al momento in cui questa linea è pilotata col valore logico basso, nessun altro master può pilotare il bus.
I tipi principali di trasferimento dati sono i seguenti:

- **Single Cycle**: consiste nel trasferimento di 8,16, o 32 bit di dato col controllo della CPU sul master;
- **Block transfers**: consiste nel trasferimento di dati da 32 o 64 bit in modalità continuativa, fino a 256 o 2048 bytes. In questa modalità il trasferimento è gestito da un DMA controller, indipendente dalla CPU;
- **Interrupts**: usati tipicamente da una scheda slave per segnalare, ad esempio, la presenza di un dato valido o un errore nel trasferimento.

Durante un trasferimento di un dato, i segnali significativi sono i seguenti:

- **data_bus** (D00-D31): nel caso di operazione di scrittura, contiene il dato che il master vuole scrivere durante un trasferimento oppure , in caso di operazione di lettura, contiene il dato che lo slave restituisce al master;
- ***LWord**: indica la dimensione del dato che può essere 8, 16,32 o 64 bit nel caso di VME64 (con riferimento all'AGE descritto nel paragrafo 2.1 la dimensione è 64 dato che il bus di comunicazione è di tipo VME64). Nel caso di ciclo *block transfer*, trasporta anche i dati;
- **address (A01-A31)**: rappresenta l'indirizzo dello slave con cui il master vuole scambiare un dato.Inoltre, le linee di *address* possono trasportare anche dati nei trasferimenti D64 multiplexed;
- **address_modifier (AM00-AM05)**: definiscono la lunghezza dell'*address* in termini di bit e il tipo di trasferimento;
- **address_strobe (AS)**: è usato dal master per segnalare allo slave la presenza di un indirizzo valido;

- **data_strobes (DS0-DS1)**: sono pilotati dal master col valore logico basso per segnalare la presenza di un dato valido durante un'operazione di scrittura, oppure sono pilotati dallo slave per indicare la presenza di un dato valido durante un trasferimento di lettura;

- **write***: definisce la direzione del trasferimento del dato. E' asserito dal master in base al tipo di operazione che vuole svolgere (scrittura o lettura);

- **data_transfer_acknowledge (DTACK)**: è utilizzato dallo slave per comunicare al master che il trasferimento è stato completato. Nel caso in cui il trasferimento non possa essere completato, lo slave pilota la linea **Bus Error (BERR*)** col valore logico basso.

Inoltre, il VME ha anche un *Bus Interrupt* a 7 pin. Un modulo che vuole effettuare una richiesta di interrupt (*Interrupter*) asserisce una delle 7 linee di **Interrupt Request (IRQ1-IRQ7)** che vengono gestite con un meccanismo di priorità. Quando il modulo che gestisce gli interrupt (*interrupt handler*) accetta una richiesta di interrupt, abilita l'**interrupt acknowledge (IACK*)** e recupera l'interrupt vector dall'*interrupter*.
Nel caso specifico dell'AGE descritto nel paragrafo 2.1, il bus VME utilizzato è il cosiddetto VME64x, ovvero un'estensione dello standard VMEbus sviluppato nel 1997. Mentre le schede slave VME tradizionali usano jumpers o interruttori su scheda per l'inizializzazione dell'indirizzo base, lo standard VME64x usa un meccanismo:

- Ogni slave ha una finestra di 512kB formati da una **Configuration ROM (CR)** e da un **Control and Status Register (CSR)**

- Nel VME64x, l'indirizzo di questa finestra viene derivato dal numero dello slot

Inoltre, generalmente la scheda VMEbus single board computer, posizionata nel primo slot del cestello VME, rappresenta l'unica scheda master e l'interrupt handler e fornisce anche le funzionalità dell'arbitro. Essa si comporta come un normale computer, supporta sistemi operativi come Linux e Windows ed è accessibile tramite RS232, VGA o Ethernet. Nel caso specifico dell'AGE descritto in 2.1 e della strumentazione utilizzata in laboratorio per simulare un AGE, essa è accessibile mediante Ethernet.

# Bibliografia

[1] Altera, “Cyclone III Device Family Overview”, in Cyclone III Device Handbook, Volume 1, July 2012

[2] Altera Corporation, “Memory Blocks in the Cyclone III Device Family”, in Cyclone III Device Handbook, Volume 1, December 2011

[3] Altera Corporation, MAX II Device Handbook, August 2009

[4] Texas Instruments, ADS8327 datasheet, update January 2011

[5] Texas Instruments, DAC8812 datasheet, update June 2016

[6] Maxim Integrated, MAX14803A datasheet, update June 2013

[7] Intel Corporation, “Avalon Interface Specifications”, update 20.1, May 2020

[8] American National Standards Institute, American National Standard for IP Modules , “ANSI/VITA 4-1995”,July 1996.

[9] Texas Instruments, SN74LVC16245A 16-Bit Bus Transceiver Datasheet, update June 2014

[10] Intel, “Intel Quartus Prime Pro Editin User Guide, Platform Designer”, update 20.4, January 2021

[11] Intel Corporation - FPGA University Program, “Introduction to the Qsys System Integration Tool”, November 2016

[12] Altera, “Quartus II Handbook, Creating Qsys Components”, Volume 1, “Design and Synthesis”, update 12.1, November 2012

[13] Intel, “NIOS II Gen2 Software Developer’s Handbook”, update 17.0, May 2017

[14] Wikipedia,VME Bus Description

[15] Markus Joos, CERN, “An introduction to VMEbus”