

POLITECNICO DI TORINO

Corso di Laurea Magistrale in
Ingegneria Elettronica

Tesi di Laurea Magistrale

Sviluppo Software Per Dispositivi Medici Impiantabili Per Applicazioni Prostatiche



**Politecnico
di Torino**

Relatori

Prof. Maurizio MARTINA

Prof. Paolo MOTTO ROS

Dott. Fabiana DEL BONO

Candidato

Federico DE SARIO

Ottobre 2023

Abstract

Grandi passi sono stati fatti nel campo della neuroproestetica, permettendo di sviluppare dispositivi che possono sostituire una modalit  motoria, sensoriale o cognitiva, che potrebbe essere stata danneggiata in seguito ad un infortunio o una malattia.

L'obbiettivo della tesi   sviluppare il software per un sistema, che permetta di interfacciarsi con un ASIC il quale si occupa di stimolazione e registrazione di segnali neurali, permettendo una comunicazione bidirezionale che consenta il bypass della lesione aggiungendo all'attuale stato dell'arte il recupero di sensi come ad esempio il tatto. Il sistema, inoltre, dovr  essere totalmente impiantabile per ridurre al minimo il rischio di infezioni. A questo proposito   stato scelto il protocollo Bluetooth 5, in quanto permette di minimizzare i consumi (massimo 16.6 mA in trasmissione e 12.9 mA in ricezione) in modalit  Bluetooth Low Energy (BLE) con una velocit  di trasmissione ideale di 2 Mbps.

Il sistema sar  composto da: un PC; un dongle; un Microcontrollore (MCU) munito di modulo BLE; una FPGA; l'ASIC che consiste in un dispositivo a 32 canali, tramite cui stimolare elettricamente altrettanti elettrodi o da cui   possibile registrare segnali in ingresso.

Dato questo hardware, in questa tesi si punta a sviluppare un codice per il MCU e per l'FPGA che permetta di inviare un comando (di stimolazione o registrazione neurale) dal PC al MCU tramite BLE, sfruttando il dongle; il MCU (Master 1) a sua volta invier  il comando all'FPGA (Slave) tramite SPI con un clock di 8 MHz; l'FPGA (Slave) in risposta, richieder  la comunicazione all'ASIC (Master 2) per inviargli il comando ricevuto dall'MCU tramite SPI con un clock di 16 MHz. Allo stesso tempo dovr  consentire la comunicazione in verso opposto quando l'ASIC invier  i segnali registrati.

Il software per il MCU e l'FPGA   stato sviluppato a step intermedi ed   stato svolto uno studio sull'ottimizzazione dei parametri del BLE per massimizzare la velocit  di trasmissione, raggiungendo 1.4 Mbps, sufficienti a gestire il flusso di dati dell'ASIC se si effettua un trade-off tra precisione e datarate, riducendo ad esempio il numero di canali o utilizzando l'ASIC in modalit  "spike detection".

Il codice del MCU   stato ultimato ed   capace di ricevere e trasmettere dati tramite BLE al dongle e tramite SPI all'FPGA; il MCU inoltre,   capace di ricevere e gestire tramite un interrupt, il segnale di IRQ che l'FPGA sfrutter  per richiedere la comunicazione. Per quanto riguarda il codice dell'FPGA,   stata implementata con successo l'interfaccia SPI verso il MCU, riducendo anche problemi di metastabilit  sfruttando l'oscillatore interno del chip (48 MHz), il PLL interno per avere un clock di 144 MHz e 2 flip-flop sugli ingressi per sincronizzarli con la logica interna. La stessa interfaccia SPI   stata testata a 8 MHz e 16 MHz e pu  essere quindi utilizzata anche come interfaccia tra il MCU e l'ASIC.

In conclusione, si   arrivati a sviluppare un software che tramite un contatore implementato su FPGA, simula un flusso dati ricevuto dall'ASIC. Quando il MCU riceve via BLE il comando di "START", lo inoltra all'FPGA tramite SPI, il contatore inizia a contare e inviare il risultato al MCU seguito da un impulso di IRQ, che genera nel MCU un interrupt che manda una nuova frame e fa ripartire il loop. Ricevuto invece, via BLE, il comando di "STOP" il ciclo viene interrotto.

Questo lavoro consentir , con l'aggiunta di un registro FIFO tra le due interfacce SPI, di ottenere il risultato prefissato.

Indice

Elenco delle tabelle	VI
Elenco delle figure	VII
Glossario	XII
1 Introduzione	1
1.1 Obbiettivi della tesi	2
1.2 Struttura della tesi	3
2 Stato dell'arte delle tecnologie per feedback sensoriale	5
2.1 Verso una mano che possa "sentire"	5
3 Descrizione del sistema	10
3.1 Bluetooth 5	12
3.1.1 Fondamenti del BLE	12
3.2 MCU	16
3.3 FPGA	17
3.4 ASIC	18
3.5 Interfaccia BLE	19
3.6 Hardware e software esterni	20
4 Sviluppo e Test del Software	23
4.1 Sviluppo del codice per il Microcontrollore	23
4.1.1 Dati generati internamente al Microcontrollore e inviati via BLE	24
4.1.2 Implementazione interfaccia SPI (Master) con trasmissione Bluetooth	41
4.2 Sviluppo del codice per l'FPGA	55
4.2.1 Interfaccia SPI (Slave)	56

4.2.2	Dati generati internamente all'FPGA e inviati al Microcontrollore tramite SPI	61
4.2.3	Sistema MCU-FPGA	76
4.2.4	Test del sistema MCU-FPGA	78
5	Conclusioni	91
5.1	Ottimizzazione e sviluppi futuri	93
	Bibliografia	95

Elenco delle tabelle

3.1	Throughput per una singola connessione al variare dei parametri di connessione, estratto dalla documentazione del Softdevice s132 [23]	15
4.1	Alcuni esempi di combinazioni di Connection Interval e dimensioni dei pacchetti testate con relativi throughput, ritrasmissioni e Packet Error Rate (PER)	40
4.2	Le ritrasmissioni diventano praticamente nulle con pacchetti da 244 byte e CI da 200 ms in su	41
4.3	Collegamento dei pin del Master e dello Slave.	52
4.4	Corrispondenza dei pin tra SPIDriver e FPGA.	71
4.5	Corrispondenza dei pin tra Microcontrollore (Implant) e FPGA.	83
4.6	Valori del data rate del dispositivo per diverse combinazioni di dimensione dei pacchetti e Connection Interval. Gli asterischi "*" indicano le misure in cui non tutti i pacchetti sono stati trasmessi con successo	89

Elenco delle figure

2.1	Controllo di una mano protesica basato su segnali sEMG [17] © 2016 IEEE	6
2.2	Confronto di protocolli wireless low-power [17] © 2016 IEEE	6
2.3	Protesi con sistema tattile installata sul volontario [10] © 2016 IEEE	7
2.4	Collegamento dei nervi del braccio al computer [10] © 2016 IEEE	7
2.5	Funzionamento del dispositivo durante le due fasi operative [18] © 2018 IOP Publishing Ltd.	8
2.6	Il dispositivo impiantabile, SenseBack, prima e dopo essere incapsulato [11]	9
3.1	Schema a blocchi del sistema	10
3.2	Singolo pacchetto per Connection Interval	13
3.3	Più pacchetti per Connection Interval	13
3.4	Formato del pacchetto BLE [22]	13
3.5	nRF52832 [24]	16
3.6	nRF52 DK [27]	16
3.7	ice40 Ultra (iCE5LP4K) [28]	17
3.8	iCE40 Ultra Breakout Board [29]	17
3.9	Schema a blocchi dell'ASIC [30]	18
3.10	Layout dell'ASIC [30]	18
3.11	Dongle nRF52840 [33]	19
3.12	Segger Embedded Studio	20
3.13	nRF Connect for Desktop	20
3.14	Wireshark	21
3.15	MATLAB	21
3.16	iCEcube2	21
3.17	Diamond Programmer	22
3.18	ModelSim	22
3.19	SPIDriver	22
4.1	Schema a blocchi del sistema	24

4.2	Set-up per il test di velocità di trasmissione dati	37
4.3	Schermate di SES dell'Implant e del Controller nei test di velocità di trasmissione dati	37
4.4	Schermate di MATLAB e Wireshark nei test di velocità di trasmissione dati	38
4.5	Debug Terminal del Controller a fine test	38
4.6	Grafico dei dati ricevuti durante il test di velocità. Sulle ascisse è segnato l'n-esimo dato ricevuto e sulle ordinate il suo valore. Si vede come i dati vengono ricevuti in sequenza da 0 a 255.	39
4.7	Grafico della velocità di trasmissione all'aumentare del CI per diverse dimensioni di pacchetto	40
4.8	Schema a blocchi del sistema	41
4.9	Set-up per il test di trasmissione SPI e BLE	52
4.10	Schermate di SES di Master e Slave nel test di comunicazione SPI e BLE	53
4.11	Schermata di nRF Connect for Desktop nel test di comunicazione SPI e BLE	53
4.12	Debug Terminal del Master dopo l'invio del comando di STOP via BLE e il successivo invio della frame di STOP "00CC" via SPI	54
4.13	Debug Terminal dello Slave dopo l'invio del comando di STOP via BLE e la ricezione della frame di STOP "00CC" via SPI	54
4.14	Debug Terminal del Master dopo l'invio del comando di START via BLE e il successivo invio della frame di START "00AA" via SPI	54
4.15	Debug Terminal dello Slave dopo l'invio del comando di START via BLE e la ricezione della frame di START "00AA" via SPI	54
4.16	Schema a blocchi del sistema completo dell'FPGA	55
4.17	Modulo SPI_SLAVE_IRQ	56
4.18	Simulazione del modulo con clock di sistema a 129 MHz e SCK a 16 MHz	60
4.19	Simulazione del modulo con clock di sistema a 159 MHz e SCK a 8 MHz	61
4.20	Modulo COUNTER SYNCHRO	62
4.21	Schema a blocchi di 03_SPI_SLAVE_IRQ_COUNTER	65
4.22	Ricezione di una frame dummy con contatore OFF, clock di sistema a 159 MHz e SPI Clock a 8 MHz	66
4.23	Ricezione della frame di START, con clock di sistema a 129 MHz e SPI Clock a 16 MHz	67
4.24	Ricezione di una frame dummy mentre il contatore è in stato ON, con clock di sistema a 129 MHz e SPI Clock a 16 MHz	68
4.25	Ricezione della frame di STOP, con clock di sistema a 159 MHz e SPI Clock a 8 MHz	69

4.26	Ricezione di una frame dummy con contatore in stato OFF, clock di sistema a 159 MHz e SPI Clock a 8 MHz	69
4.27	Schermata di iCEcube2 durante la modifica dei pin constraint	70
4.28	I log su Diamond Programmer indicano che il programma è stato caricato correttamente sull'FPGA	71
4.29	Set-up per il test con SPIDriver	71
4.30	Interfaccia grafica per PC dell'SPIDriver	72
4.31	Invio della frame "0000" con contatore in stato OFF	72
4.32	Invio della frame di accensione "00AA"	73
4.33	Invio della frame "00FF" con contatore in stato ON	73
4.34	Invio della frame di accensione "00AF" con contatore in stato ON	74
4.35	Invio della frame "0019" con contatore in stato ON	74
4.36	Invio della frame di spegnimento "00CC"	75
4.37	Invio della frame "0011" con contatore in stato OFF	75
4.38	Invio della frame anomala "00CC"	76
4.39	Sistema finale	77
4.40	Placement&Route eseguito su iCEcube2	78
4.41	Flashing del binario sull'FPGA	78
4.42	Finestra di SES nel test funzionale	79
4.43	Flashing del binario sull'FPGA	79
4.44	Set-up del test funzionale	80
4.45	Debug Terminal del MCU dopo l'invio del comando di START. I log rossi sono i log che non sono stati stampati a causa della velocità con cui vengono generati	81
4.46	Terminale di nRF Connect dove vengono stampati, sotto forma di log, tutti i dati ricevuti dopo l'invio del comando di START. Si può notare come vengono ricevuti correttamente in sequenza	81
4.47	Schermate di SES dell'Implant e del Controller durante il test di velocità	82
4.48	Set-up per il test del data rate	83
4.49	Schermate di SES dell'Implant e del Controller durante il test di velocità	84
4.50	Debug Terminal del Controller a fine test, nel caso di pacchetti di dimensione variabile, con massimo a 244 byte e CI da 200 ms	84
4.51	Grafico dei dati ricevuti durante il test di velocità. Sulle ascisse è segnato l'n-esimo dato ricevuto e sulle ordinate il suo valore. Si vede come i dati vengono ricevuti in sequenza da 0 a 255.	85
4.52	Schermata di Wireshark con visualizzati i pacchetti ritrasmessi	85
4.53	Debug Terminal del Controller a fine test, nel caso di pacchetti di dimensione fissa a 244 byte e CI da 200 ms	87

4.54	Grafico dei dati ricevuti durante il test di velocità. Sulle ascisse è segnato l'n-esimo dato ricevuto e sulle ordinate il suo valore. Si vede come i dati vengono ricevuti in sequenza da 0 a 255.	88
4.55	Grafico a barre dell'andamento del data rate del dispositivo finale al variare di Connection Interval e dimensione dei pacchetti	90
5.1	Grafico a barre dell'andamento del throughput al variare del Connection Interval con pacchetti nei best-case del sistema con il solo MCU (pacchetti da 244 byte) e del sistema completo (pacchetti da 200 byte)	92
5.2	Grafico a barre del throughput con i log abilitati/disabilitati, utilizzando gli stessi parametri di connessione (pacchetti fissi da 244 byte e CI da 200 ms)	92

Glossario

ASIC

Application Specific Integrated Circuit

BLE

Bluetooth Low Energy

FPGA

Field Programmable Logic Array

MCU

Micro Controller Unit

SPI

Serial Peripheral Interface

VHDL

Very high speed Hardware Description Language

PHY

Physical Layer

IFS

Inter Frame Space

IDE

Integrated Development Environment

SES

Segger Embedded Studio

GPIO

General Purpose Input Output

ISR

Interrupt Service Routine

sEMG

Surface Electromyography

Capitolo 1

Introduzione

Nel corso degli anni la medicina ha fatto molti progressi nel campo riabilitativo, specialmente nella branca delle protesi neurali (neuroprotesi), dispositivi in grado di restituire in parte, completamente o potenziare capacità sensoriali o cognitive oltre che parti dell'apparato motorio. La differenza dalle normali protesi è che le neuroprotesi si interfacciano direttamente con il sistema nervoso periferico (PNS) dal quale ricevono dei segnali elettrici o verso il quale li propagano [1].

I campi di applicazione sono numerosi: oltre a neuroprotesi motorie, sensoriali e cognitive infatti, sono state sviluppate delle protesi neurali per il trattamento del dolore [2] o per investigare e trattare disturbi neurologici [3] portando a continui miglioramenti nel campo della "medicina bioelettronica" [4].

Si pensi ad esempio all'impianto cocleare, sempre più diffuso e in costante miglioramento [5, 6]: questa non è una normale protesi acustica ma una neuroprotesi, in quanto non amplifica semplicemente i suoni provenienti dall'esterno, ma captato un suono, lo converte in un segnale elettrico, tramite il quale stimola direttamente il nervo ottico, rendendo possibile la percezione e il riconoscimento del suono anche in soggetti parzialmente o completamente sordi.

Molti progressi sono individuabili anche quando si parla delle protesi motorie. Si pensi alle protesi degli arti superiori dove partendo da semplici moduli statici (come ad esempio un braccio in plastica o silicone) in sostituzione dell'arto lesso o mancante, si è andati sempre più verso protesi funzionalmente più complesse fino ad arrivare a delle protesi neurali, le quali ricevendo uno stimolo dalle terminazioni nervose restanti, propagano un segnale alla neuroprotesi la quale riesce ad eseguire movimenti come piegare un braccio o raccogliere un oggetto [7].

Col tempo ha acquisito sempre più importanza la restituzione del senso del tatto in quanto non solo è di particolare rilevanza per portare ad un livello successivo il controllo, l'efficacia e il realismo delle protesi degli arti superiori [8], ma la stimolazione del sistema nervoso periferico sembra addirittura ridurre il dolore derivante dalla "sindrome dell'arto fantasma" [2], una patologia a causa della quale

si ha una viva percezione, spesso spiacevole o dolorosa, di un arto che non è più presente a causa di un'amputazione e che affligge la qualità della vita di circa l'85% dei soggetti che hanno subito questo tipo di operazione. Il senso del tatto, inoltre, è strettamente correlato alla proprioccezione dell'arto stesso, ovvero al senso della posizione e del movimento degli arti indipendente dalla vista, fondamentale per il controllo dei movimenti [8] (soprattutto quelli più naturali). Il tatto inoltre, non è ristretto ai soli arti superiori ma è distribuito attraverso tutto il corpo e si stima che la sua perdita possa arrecare gravi disagi, quali ad esempio la perdita della destrezza manuale e dell'abilità di camminare [9].

Questo ha portato gli studi a concentrarsi sullo sviluppo di dispositivi capaci di restituire questa sensazione tramite stimolazioni elettriche [2, 10–12]. Tuttavia, i meccanismi che si celano dietro le sensazioni elettro-tattili sono ancora poco chiari e ciò presenta un ostacolo, che rende ancora difficile il raggiungimento del controllo stabile di queste sensazioni [9]. Per questo motivo, si fa sempre più importante lo sviluppo di dispositivi in grado di consentire studi cronici ex-vivo e in-vivo, con l'obiettivo di raccogliere dati e colmare le lacune che ancora si presentano sulla conoscenza di tali meccanismi. Poiché si parla di studi cronici di durata superiore a 6 mesi [11], la ricerca ha spinto verso dispositivi completamente impiantabili, cercando di massimizzare la flessibilità di utilizzo e di minimizzare i consumi e il rischio d'infezione [13]. Considerando l'importanza dell'argomento e il notevole contributo che lo sviluppo delle tecnologie nel campo del feedback tattile può apportare al miglioramento della qualità della vita delle persone colpite da lesioni di questo genere, con questa tesi si vuole contribuire a fare un passo in avanti verso il ripristino del senso del tatto in soggetti lesi, senza porre un limite a ulteriori potenziali applicazioni future.

La tesi è stata svolta in collaborazione con il team Next Generation Neural Interfaces (NGNI) Lab dell'Imperial College di Londra, i quali hanno sviluppato il progetto "SenseBack" [11] (descritto nello stato dell'arte, Cap. 2) e hanno fornito gli schemi del loro prototipo impiantabile (Fig. 3.1) e i sorgenti del codice per il Microcontrollore [14, 15], da cui si è partiti per lo sviluppo del software svolto in questa tesi.

1.1 Obiettivi della tesi

L'obiettivo della tesi consiste nello sviluppo di un software destinato a un sistema che permetta di interagire con un ASIC incaricato della stimolazione neurale e della registrazione dei segnali neurali tramite elettrodi collegati ai nervi, permettendo una comunicazione bidirezionale tra il sistema e l'ASIC stesso (e in futuro la neuroprotesi), consentendo il recupero di sensi come ad esempio il tatto. Il sistema hardware, lo stesso del progetto "SenseBack", è composto da un Microcontrollore,

una FPGA e l'ASIC e dovrà essere completamente impiantabile per ridurre al minimo il rischio di infezioni. A tal proposito sarà altrettanto necessario che supporti una comunicazione wireless con l'esterno, la quale massimizzi la velocità di trasmissione per gestire il flusso di dati dell'ASIC verso l'esterno e permetta l'invio dei comandi di configurazione dall'esterno verso l'ASIC. I consumi del dispositivo inoltre, dovranno essere minimizzati in modo che sia alimentabile con un modulo wireless senza però dissipare quantità dannose di energia nei tessuti circostanti. In questa maniera il sistema potrà funzionare per lunghi periodi, considerato che per questo tipo di studi, la mole di segnali neurali che è necessario registrare e analizzare, può portare il dispositivo a rimanere impiantato e funzionante per più di 6 mesi [11].

Nello specifico, l'hardware del dispositivo è quello del progetto "SenseBack" [11], descritto nel capitolo 2, fornito dal Next Generation Neural Interfaces (NGNI) Lab dell'Imperial College di Londra, mentre il lavoro di questa tesi punta invece a sviluppare un codice in C++ per il Microcontrollore (partendo da un sorgente per MCU fornito con l'hardware attraverso una repository Github [14, 15]) e un codice in VHDL per l'FPGA (sviluppato da zero) procedendo per fasi intermedie, le quali saranno analizzate più nel dettaglio nei capitoli successivi.

In questa maniera si punta allo sviluppo di un prototipo, che sfruttando lo stesso hardware di "SenseBack" e il software sviluppato in questa tesi (soprattutto la parte per FPGA, senza la quale non sarebbe possibile continuare lo sviluppo), permetta di ottenere una piattaforma di partenza per effettuare test sull'ASIC e studi a lungo termine sul miglioramento delle prestazioni, con l'obiettivo finale di arrivare alla sperimentazione sugli umani.

1.2 Struttura della tesi

Per una chiara e completa esposizione del progetto, questo scritto sarà organizzato come segue:

- Capitolo 2: Uno sguardo sullo stato dell'arte della neuro-prostetica, con la rapida descrizione di alcuni progetti precedenti in linea con il progetto di questa tesi, tra cui "SenseBack" [11] da cui si è partiti;
- Capitolo 3: Una presentazione del sistema completo, con la successiva descrizione approfondita dei componenti chiave che lo costituiscono, comprensiva delle funzioni che dovranno svolgere e delle specifiche che dovranno rispettare;
- Capitolo 4: Un'esposizione della metodologia di lavoro applicata, con descrizione delle varie fasi intermedie, dei codici sviluppati, del set-up utilizzato ad ogni passo e degli studi svolti, al fine dell'ottenimento dei risultati finali;

- Capitolo 5: Una raccolta dei risultati ottenuti durante questo lavoro con relativa discussione in relazione alle specifiche richieste e uno sguardo alle possibili ottimizzazioni future.

Capitolo 2

Stato dell'arte delle tecnologie per feedback sensoriale

Gli studi sulle neuroprotesi procedono da diverso tempo. Si pensi, ad esempio, che il primo impianto cocleare è stato impiantato nel 1961, sebbene i primi studi su questo impianto risalgano al XIX secolo [16]. Nel corso degli anni, diversi sono stati i campi di studio della neuro-prostetica e diverse le sensazioni che si è cercato di recuperare: vista, udito, equilibrio, tatto, propiocezione, ecc...

Le ultime due, strettamente correlate, sono l'obiettivo a cui ci si vuole avvicinare con il lavoro esposto in questa tesi e per questo motivo, nel prossimo paragrafo, si darà uno sguardo a implementazioni e studi precedenti che hanno ispirato questo progetto, facendo un punto sull'attuale stato dell'arte.

2.1 Verso una mano che possa "sentire"

Diverse figure nel mondo delle scienze hanno fatto passi avanti verso questo obiettivo, introducendo progetti innovativi sempre più vicini all'obiettivo.

Nel 2016, veniva progettato un dispositivo indossabile capace di acquisire segnali sEMG [7], ovvero segnali elettromiografici superficiali prodotti dalla contrazione dei muscoli, con l'obiettivo di riconoscere il movimento generato e controllare una mano protesica di conseguenza. Sebbene qui non si parli di un dispositivo impiantabile, in quanto la tecnologia sEMG è un metodo non invasivo di acquisizione di segnali generati dall'attività muscolare [17], vengono approcciate diverse sfide comuni all'obiettivo di questa tesi, quali: la miniaturizzazione del dispositivo e la scelta di un protocollo di trasmissione wireless e a basso consumo. Per quanto

riguarda l'ultimo punto in particolare, sono stati confrontati diversi protocolli di trasmissione (Fig. 2.2) tra cui il Bluetooth 4.1 e il Low Power WiFi (LpWiFi) che hanno restituito i risultati migliori.

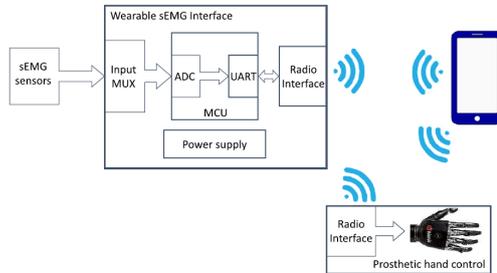


Figura 2.1: Controllo di una mano protesica basato su segnali sEMG [17]

© 2016 IEEE

TABLE I
LOW-POWER WIRELESS PROTOCOLS [41], [42]

Protocol	Max throughput (application)	Max throughput (physical layer)	Protocol efficiency
BLE	305 kbps	1 Mbps	0.31
Zig Bee	115.2 kbps	250 kbps	0.46
6LoWPAN	72 kbps	250 kbps	0.29
ANT	20 kbps	1 Mbps	0.02
LpWiFi (UDP)	16 Mbps	54 Mbps	0.29

Figura 2.2: Confronto di protocolli wireless low-power [17]

© 2016 IEEE

Nello stesso anno, Dustin J. Tyler e il suo team, testano su un volontario il sistema tattile sviluppato nel Functional Neural Interface Lab alla Case Western Reserve University di Cleveland [10]. Questo sistema è collegato a 3 nervi del braccio destro del volontario in 20 punti e alla sua protesi artificiale dall'altro lato (Fig. 2.3). Sono stati usati dei sensori di forza sul pollice e sull'indice della protesi, i cui segnali generati vengono usati per produrre la corrispondente stimolazione nei nervi collegati.

Grazie a questo sistema il volontario riesce a svolgere diversi compiti con più precisione. Ad esempio, nei test del dispositivo, gli viene chiesto di raccogliere una ciliegia e di staccargli il picciolo:

- con il sistema tattile disattivato riusciva a esaudire la richiesta con successo, senza rompere il frutto, solo nel 43% dei tentativi;
- una volta attivato il sistema tattile, la percentuale di successo sale al 93%.

Tuttavia, il sistema è ancora rudimentale e può essere usato solo in laboratorio considerando i cavi che collegano il braccio del volontario al computer da cui viene controllata la stimolazione (Fig. 2.4). Inoltre, c'è bisogno ancora di studio sui segnali acquisiti per capire con quali pattern di stimolazione si possono produrre sensazioni più naturali.



Figura 2.3: Protesi con sistema tattile installata sul volontario [10]
© 2016 IEEE

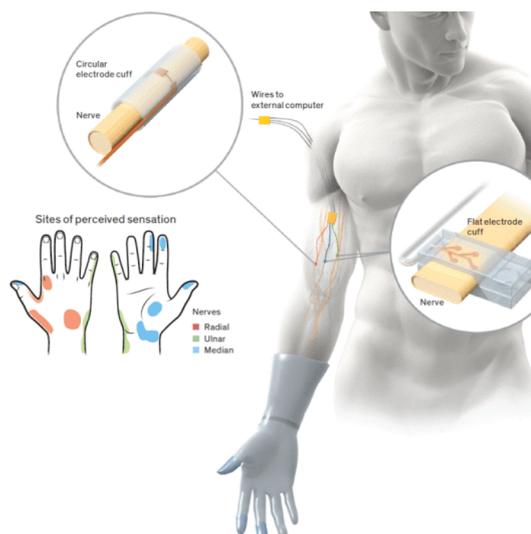


Figura 2.4: Collegamento dei nervi del braccio al computer [10]
© 2016 IEEE

Nel 2018, viene fatto un ulteriore passo avanti nella registrazione di segnali neurali, grazie allo sviluppo, di una piattaforma capace di registrare in tempo reale segnali neurali su 32 canali [18]. Queste registrazioni generano però una grande mole di dati che può essere difficile da trasmettere e immagazzinare, soprattutto in dispositivi compatti. Per questo motivo il dispositivo lavora in due fasi (Fig. 2.5):

- la prima è una fase di training in cui, con l'utilizzo di un computer, vengono registrati i segnali neurali e generate delle maschere di riconoscimento degli Spike;
- nella seconda fase viene effettuata una classificazione in tempo reale delle forme d'onda analizzate, sfruttando i template generati nella fase di training.

In questa maniera si riduce lo spazio di archiviazione richiesto di circa 3 ordini di grandezza e permette di ridurre i consumi e la banda richiesti per trasmettere i dati.

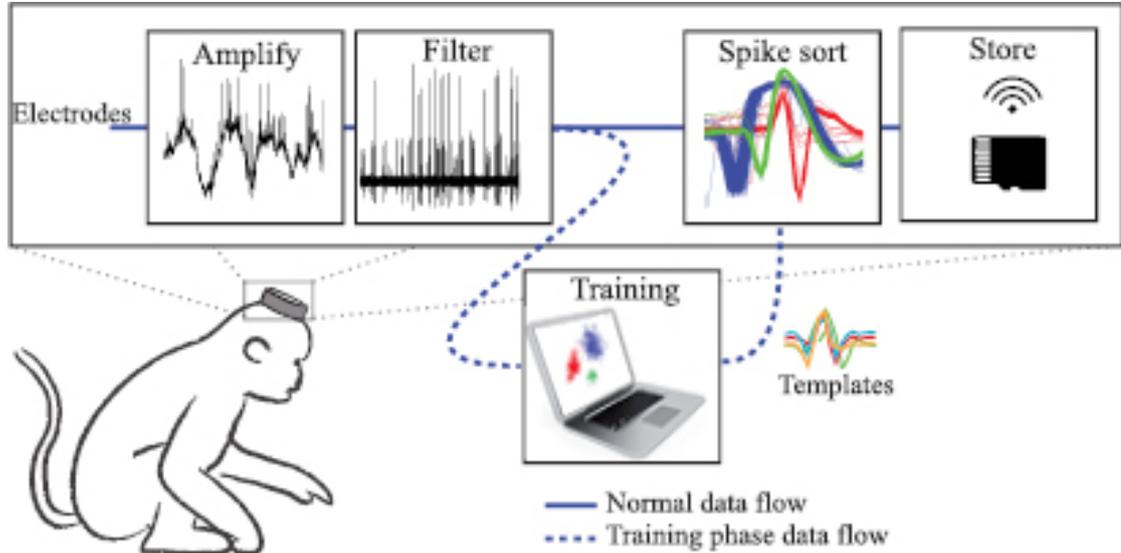


Figura 2.5: Funzionamento del dispositivo durante le due fasi operative [18]
© 2018 IOP Publishing Ltd.

Successivamente, nel 2020, in collaborazione con l'Imperial College di Londra, viene sviluppato il progetto "SenseBack" [11], il cui hardware (Fig. 2.6) è stato reso disponibile, insieme ai codici sorgenti per il Microcontrollore [14, 15], per procedere con il lavoro svolto in questa tesi. Il dispositivo descritto è capace di effettuare stimolare e registrare segnali neurali su 32 Canali [11], scambiando dati via BLE con un PC dal quale vengono inviate le configurazioni di stimolazione e collezionati i dati registrati. Questo sistema sarà descritto meglio nel capitolo 3.

Nei test a banco il dispositivo ha raggiunto una velocità massima di 1.3 Mbps via BLE con consumi tra 17 mW in idle-state e 78 mW se si registrano 4 canali in modalità *Raw Data*, consumo che scende a 21 mW se si registrano tutti i 32 canali in modalità *Spike Detection*.

"SenseBack" è stato testato a banco, *ex-vivo* ed *in-vivo* su roditori. Tuttavia, i test *in-vivo* sono stati effettuati solo in *acuto* [11], ovvero con esperimenti condotti su un breve periodo, con un set-up sperimentale e limitato. I risultati dei test *in-vivo* quindi rappresentano solo una prova dell'efficacia e della fattibilità dei concetti su cui si basa il sistema ma non rappresentano con certezza informazioni dettagliate sull'efficacia a lungo termine e in situazioni reali.

Resta quindi il bisogno di effettuare test, sul lungo periodo, per colmare queste lacune e migliorare la flessibilità delle stimolazioni. Motivo per il quale, partendo

da questo sistema hardware, si è sviluppato il software, argomento di questa tesi, per rendere possibile la riproduzione di un prototipo di questo progetto e il suo futuro miglioramento.

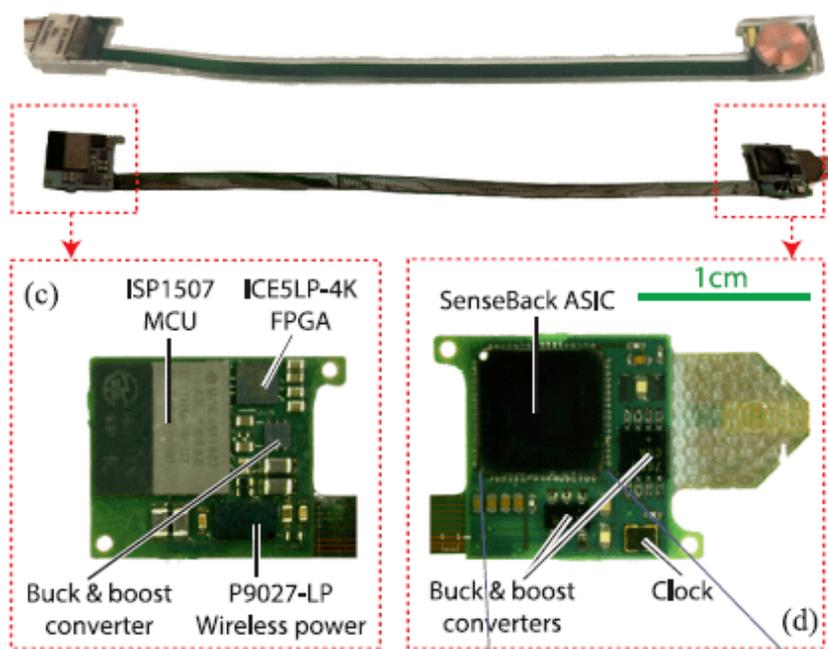


Figura 2.6: Il dispositivo impiantabile, SenseBack, prima e dopo essere incapsulato [11]

Capitolo 3

Descrizione del sistema

Il sistema complessivo, è quello del progetto "SenseBack" [11] presentato nel capitolo 2. E' composto da più elementi in comunicazione tra loro, come si può vedere in Fig.3.1.

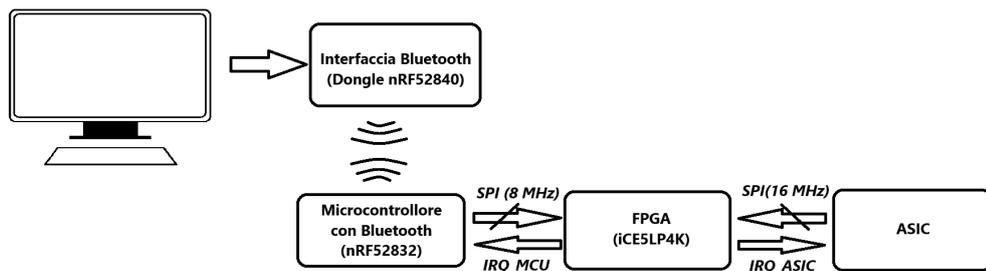


Figura 3.1: Schema a blocchi del sistema

In particolare il sistema comprende:

- un PC dal quale verranno inviate le configurazioni di stimolazione e registrazione per l'ASIC e con cui verranno collezionati ed elaborati i dati registrati;
- un'interfaccia Bluetooth che permetterà al PC di inviare e ricevere dati via BLE;
- un Microcontrollore che supporta il protocollo Bluetooth 5, che gestisce la comunicazione tra il PC e l'ASIC comunicando via BLE con l'interfaccia Bluetooth;

- un'FPGA che farà da buffer tra il Microcontrollore e l'ASIC, permettendo ai due di comunicare tra di loro nonostante siano entrambi SPI master e abbiano due regimi di SPI Clock differenti;
- l'ASIC che si occupa della registrazione e della stimolazione di segnali neurali.

Nei prossimi paragrafi vengono descritti i modelli e le versioni di questi dispositivi che sono stati utilizzati per questo progetto.

Il sistema presentato, supporta una comunicazione wireless, che massimizzerà la velocità di trasmissione e ridurrà al minimo i consumi, instaurando una comunicazione bidirezionale tra il PC e l'ASIC, che funzionerà in questo modo:

- Partendo dal PC verso l'ASIC:
 1. attraverso il PC, viene mandato un comando di configurazione per l'ASIC (che può essere una modalità di registrazione o stimolazione);
 2. sfruttando l'interfaccia Bluetooth, il comando viene inviato al Microcontrollore;
 3. il Microcontrollore a sua volta, trasmette il comando alla FPGA tramite SPI, con un SPI clock di 8 MHz;
 4. l'FPGA inoltra il comando all'ASIC, richiedendo a quest'ultimo la comunicazione, sfruttando il pin IRQ_ASIC;
 5. l'ASIC vede la richiesta di comunicazione sul pin IRQ_ASIC, legge tramite SPI, con SPI clock a 16 MHz, il comando che gli è stato inviato dall'FPGA ed effettua l'operazione richiesta (stimolazione/inizio registrazione).
- Partendo dall'ASIC verso il PC:
 1. se l'ASIC sta registrando dati, questi vengono inviati tramite SPI, con SPI clock a 16 MHz, all'FPGA;
 2. l'FPGA bufferizza i dati ricevuti e richiede la comunicazione al Microcontrollore sfruttando il pin IRQ_MCU;
 3. il Microcontrollore vede la richiesta di comunicazione sul pin IRQ_MCU e legge tramite SPI, con SPI clock a 8 MHz, i dati inviati dall'FPGA;
 4. infine il Microcontrollore invia i dati ricevuti al PC, sfruttando l'interfaccia Bluetooth, dove verranno successivamente raccolti ed elaborati.

L'apparato sarà completamente impiantabile in modo da ridurre al minimo il rischio di infezioni dovuto a cablaggi percutanei [13], permettendo studi *ex-vivo* ed *in-vivo* [11], sebbene questi ultimi siano stati effettuati solo in *acuto*, come anticipato nel capitolo 2.

3.1 Bluetooth 5

Come si evince dalla Fig.3.1 è stato scelto il protocollo Bluetooth per soddisfare i requisiti di trasmissione wireless. In particolare, è stato scelto il protocollo Bluetooth 5 che, grazie alla modalità Bluetooth Low Energy (BLE), permette di minimizzare i consumi pur mantenendo una velocità di trasmissione nominale di 2 Mbps [19], sebbene la reale velocità massima di trasmissione si riduce a circa l'80% del valore dichiarato, a causa dei protocolli di background del BLE [20]. Per comprendere le successive scelte in merito ai parametri di connessione del BLE, nei test del capitolo 4, viene qui descritto in breve il protocollo Bluetooth con particolare riguardo ai parametri che più influenzano la velocità di trasmissione dati [21, 22].

3.1.1 Fondamenti del BLE

Si ritiene utile introdurre qualche definizione basilare.

Una connessione BLE è una connessione senza fili che si instaura tra due dispositivi, definiti Central e l'altro Peripheral.

Ogni evento di comunicazione tra i due dispositivi prende il nome di Connection Event (CE) e il tempo tra due Connection Event è chiamato Connection Interval e può durare da un minimo di 7.5 ms ad un massimo di 4 s, ad incrementi di 1.25 ms [7].

La Central inizialmente è in fase di Scanning per cercare delle Peripherals, queste ultime invece avvertono i dispositivi vicini della loro presenza tramite la funzione di Advertising. La connessione tra i due dispositivi parte dalla Central e una volta connessi, la Peripheral esce dalla fase di Advertising e la Central interrompe lo Scanning, permettendo di risparmiare energia.

Un Connection Event, esattamente come la connessione, inizia dalla Central, la quale trasmette un pacchetto alla Peripheral, che, una volta ricevuto, risponde a sua volta inviando un pacchetto. Ad ogni Connection Event è possibile inviare più pacchetti (Fig.3.2-3.3), ma sebbene lo standard Bluetooth non imponga un numero massimo di pacchetti per CE, nelle applicazioni pratiche questi sono limitati da requisiti di tempo e risorse hardware limitate [7, 21, 22].

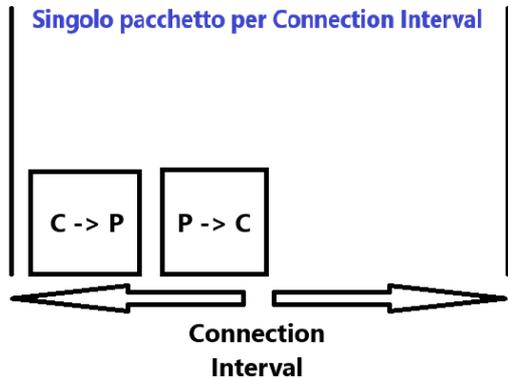


Figura 3.2: Singolo pacchetto per Connection Interval

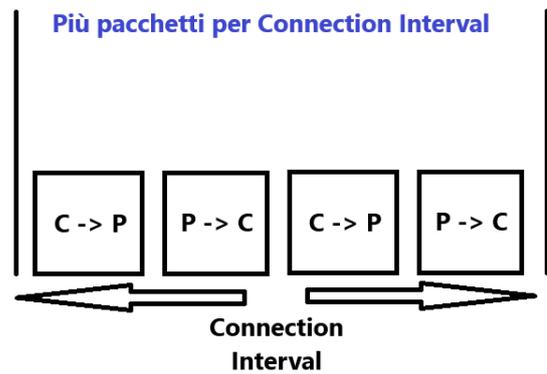


Figura 3.3: Più pacchetti per Connection Interval

Per mantenere la connessione, Central e Peripheral si scambiano dei pacchetti anche quando non c'è effettivamente un dato utile da inviare. Questi sono chiamati Empty Link Layer PD.

In un Connection Event è possibile trasmettere più di due pacchetti e questo aumenta considerevolmente il throughput, ovvero la velocità di trasmissione dei "dati utili".

Infatti ogni pacchetto è formato da diversi byte, ma di questi alcuni sono definiti Overhead e sono implicitamente compresi nel protocollo BLE (Fig. 3.4).

Preamble	Access Address	PDU (2-257 bytes)					CRC	
		LL Header	Payload (0-251 bytes)			MIC (Optional)		
			L2CAP Header	ATT Data (0-247 bytes)				
1 byte (1M PHY) 2 bytes (2M PHY)	4 bytes	2 bytes	4 bytes	Op Code 1 byte	Attribute Handle 2 bytes	Up to 244 bytes	4 bytes	3 bytes

Figura 3.4: Formato del pacchetto BLE [22]

Di tutti i byte del pacchetto, quelli che rappresentano i dati utili sono compresi nell'ATT Payload che, nel Bluetooth 5, può avere una dimensione massima di 244 byte.

Come anticipato, con l'avvento del Bluetooth 5, la velocità massima di trasmissione dichiarata è salita a 2Mbps [19], tuttavia questa è la velocità di trasmissione dell'intero pacchetto, non il throughput. Quest'ultimo sarà sicuramente più lento della velocità dichiarata a causa delle seguenti ragioni [20]:

- parte dei dati del pacchetto sono Overhead (Fig. 3.4);

- c'è un minimo ritardo tra i pacchetti (150 μ s), chiamato Inter Frame Space (IFS);
- c'è un limite al numero di pacchetti che si possono inviare in un Connection Interval;
- quando non ci sono dati utili da trasmettere, per mantenere la connessione c'è comunque bisogno di inviare dei pacchetti vuoti.

Parametri e throughput

Ci sono però diversi parametri che influenzano il throughput di un'applicazione BLE e che possono essere modificati per massimizzarlo [21, 22]. I più comuni sono:

- il Physical Layer (PHY);
- l'ATT Maximum Transmission Unit (ATT MTU);
- il Connection Interval;
- il tipo di Operazione;

Il Physical Layer nel Bluetooth 5 ha tre configurazioni base: 1 Mbps PHY, 2 Mbps PHY (entrambi definiti Uncoded PHY perché usano una rappresentazione a singolo simbolo per bit) e il Coded PHY (che invece utilizza da 2 a 8 simboli per bit, frazionando dello stesso valore la velocità ideale). Per la nostra applicazione verrà scelto il 2 Mbps PHY in quanto è quello che permette la massima velocità di trasmissione e il minimo consumo, in quanto trasmette la stessa quantità di dati in meno tempo, senza aumentare la potenza di trasmissione.

L'ATT MTU è la massima mole di dati (Fig. 3.4) che la Central e la Peripheral possono gestire. Non è la massima dimensione dell'ATT Data, ovvero 247 bytes (Fig. 3.4), ma è consigliato tenerlo uguale. Se infatti viene scelto un valore più grande di ATT Data, il pacchetto successivo da inviare sarà spezzato in due pacchetti, rallentando di molto il throughput a causa dell'aggiunta degli Overhead e dei ritardi tra i pacchetti. L'effettivo valore di ATT MTU scelto per la connessione sarà il minimo valore ammesso fra quello della Central e quello della Peripheral.

Il Connection Interval determina quanto tempo passa fra i Connection Event. Più questa finestra è larga, più pacchetti riuscirò ad inviare al suo interno. Tuttavia, un Connection Interval troppo lungo è controproducente, in quanto introduce latenza. Inoltre, ogni dispositivo ha un massimo numero di pacchetti ammesso per Connection Interval, se quindi durante un Connection Event viene raggiunto il massimo numero di pacchetti trasmessi o se ne perde uno, più è grande il Connection Interval e più bisognerà aspettare per l'arrivo dei prossimi pacchetti. Di contro, ridurre molto il Connection Interval, puntando ad aumentare il numero

dei Connection Event, può ridurre la latenza ma aumenta i consumi del dispositivo. Questo porta ad un trade-off tra latenza e throughput.

Infine, il throughput dipende dal tipo di Operazione richiesto. Utilizzando la funzione Notification anziché Indication, è possibile inviare dati senza attendere la risposta del dispositivo connesso, cosa che invece avviene con la funzione Indication e rallenta il throughput.

Nella tabella 3.1, estratta dalla documentazione del Microcontrollore utilizzato [23], è evidente come i parametri menzionati alterano il throughput.

Protocol	ATT MTU size [byte]	Connection Interval [ms]	Method	Max data throughput (LE 1M PHY) [kbps]	Max data throughput (LE 2M PHY) [kbps]
GATT Server	158	7.5	Send Notification	248.0	330.6
			Receive Write Command	248.0	330.6
			Simultaneous send Notification and receive write command	165.3 (each direction)	275.5 (each direction)
GATT Server	247	50	Send Notification	702.8	1327.5
			Receive Write Command	702.8	1327.5
			Simultaneous send Notification and receive write command	390.4 (each direction)	780.8 (each direction)
GATT Server	247	400	Send Notification	771.1	1376.2
			Receive Write Command	760.9	1376.2
			Simultaneous send Notification and receive write command	424.6 (each direction)	800.4 (each direction)

Tabella 3.1: Throughput per una singola connessione al variare dei parametri di connessione, estratto dalla documentazione del Softdevice s132 [23]

3.2 MCU

Il Microcontrollore scelto per questo progetto è l'nRF52832 della Nordic Semiconductor [24]. Questo Microcontrollore (Fig. 3.5), è già stato largamente utilizzato in diverse applicazioni nel campo dei dispositivi impiantabili [11, 25, 26] grazie alle sue dimensioni ridotte (36 mm^2), al suo sistema di gestione dei consumi ma soprattutto grazie al supporto del protocollo Bluetooth 5, con la funzione BLE e il 2 Mbps PHY.



Figura 3.5: nRF52832 [24]

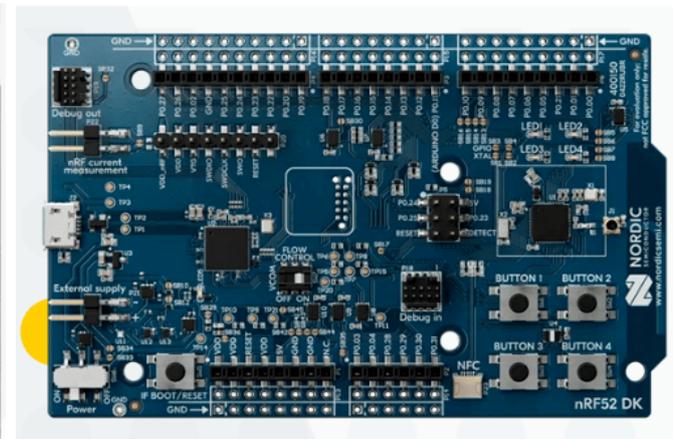


Figura 3.6: nRF52 DK [27]

Durante il lavoro descritto in questa tesi, viste le esigenze di sviluppo, è stato utilizzato, nello specifico, il Design Kit di questo processore, l'nRF52 DK [27], potendo così avere un accesso agevolato ai pin del dispositivo e la disponibilità di LED e pulsanti utili in fase di progettazione del software.

Vedremo nel prossimo capitolo (Sez. 4.2.3) come il Microcontrollore si interfacerà con l'FPGA.

3.3 FPGA

Per quanto riguarda l'FPGA, come per il Microcontrollore, l'hardware è lo stesso di "SenseBack" [11], in cui è stata scelta una iCE40 Ultra (iCE5LP4K) [28], un chip dalle dimensioni ridotte (4.3 mm^2) e i bassi consumi ($71\ \mu\text{A}$ in condizioni statiche).



Figura 3.7: ice40 Ultra (iCE5LP4K) [28]

Figura 3.8: iCE40 Ultra Breakout Board [29]

Per le stesse esigenze del Microcontrollore, in questo progetto sarà utilizzato il kit di sviluppo di questa FPGA (Fig. 3.7), l'iCE40 Ultra Breakout Board [29].

3.4 ASIC

L'ASIC è il cuore del sistema (Fig. 3.9-3.10). Si tratta di un'interfaccia neurale bidirezionale, capace di effettuare, su ognuno dei suoi 32 canali e relativi elettrodi, stimolazioni ad alta tensione e registrazioni di segnali neurali. Il tutto racchiuso in un circuito integrato di 15.2 mm^2 [30].

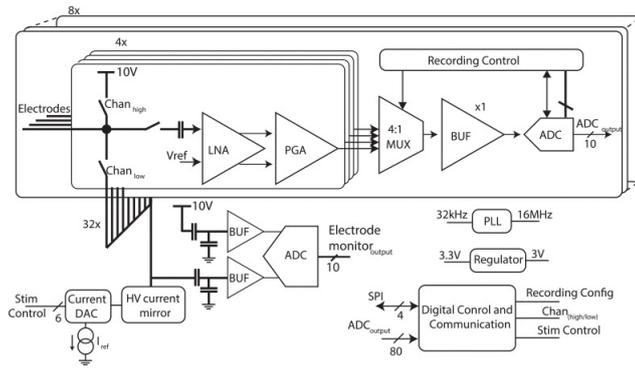


Figura 3.9: Schema a blocchi dell'ASIC [30]

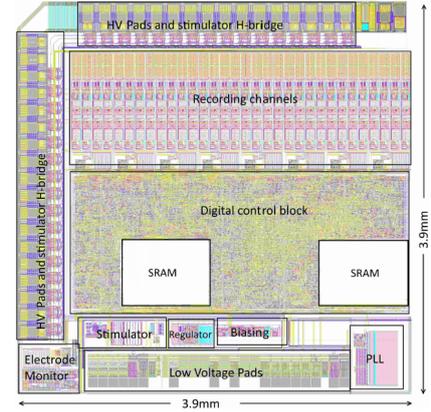


Figura 3.10: Layout dell'ASIC [30]

Ognuno dei canali può essere attivato, disattivato e configurato e ogni 4 canali condividono un ADC da 10 bit, con frequenza di campionamento di 64 kHz (16 kHz per canale). L'ASIC però, manda in uscita 16 bit, 10 bit per il dato registrato e i restanti sono bit di identificazione del canale [31]. Supponendo quindi di utilizzare tutti i canali del chip per uno streaming continuo, si avrebbe un data rate in uscita dall'ASIC di circa 8 Mbps (Eq. 3.1, dove N_{canali} è il numero di canali utilizzati, $N_{bit_{ASIC}}$ è il numero di bit per dato in uscita dall'ASIC e $f_{camp_{ADC}}$ è la frequenza di campionamento dell'ADC).

$$Datarate_{ASIC} = \frac{N_{canali}}{4} \cdot N_{bit_{ASIC}} \cdot f_{camp_{ADC}} = \frac{32}{4} \cdot 16 \text{ bit} \cdot 64 \text{ kHz} \approx 8.2 \text{ Mbps} \quad (3.1)$$

La comunicazione col circuito integrato avviene su un collegamento SPI a 16 MHz, dove il chip è l'SPI Master. Nel momento in cui c'è bisogno di inviare dati all'ASIC, come ad esempio per mandare le configurazioni di stimolazione o registrazione, l'SPI Slave potrà richiedere la comunicazione settando alto il pin di IRQ.

L'alto data rate può diventare un problema se si pensa che il Bluetooth, a fine

catena, può trasmettere idealmente a massimo 2 Mbps. Inoltre non tutti i dati di uno streaming continuo saranno segnali neurali d'interesse. Per questo motivo il chip possiede la logica necessaria per effettuare la Neural Spike Detection, che abbassa considerevolmente il data rate e i consumi del dispositivo [11]. In questa modalità di registrazione se un segnale neurale supera una soglia, determinata dall'utilizzatore, per più di 3 campioni, dopo i successivi 9 campioni un flag avverte che uno spike è stato individuato e salvato in memoria. Successivamente vengono inviati in uscita 16 campioni, 4 prima e 12 dopo il superamento della soglia. Questo meccanismo è spiegato meglio in [32], dispositivo con cui questo ASIC condivide il front-end [30].

3.5 Interfaccia BLE

Per permettere che il Microcontrollore e il PC possano comunicare, è stato utilizzato, come interfaccia BLE, il Dongle nRF52840 [33] della Nordic Semiconductor (Fig. 3.11), il quale supporta tutti gli standard wireless utilizzati dai dispositivi della Nordic ed è progettato per lavorare in simbiosi con il software nRF Connect for Desktop, presentato più avanti (Fig. 3.13). Può inoltre supportare applicazioni custom e possiede un led RGB, un pulsante programmabile e 15 GPIO che possono tornare utili in fase di sviluppo.



Figura 3.11: Dongle nRF52840 [33]

Inoltre, può essere utilizzato con Wireshark (Fig. 3.14) e il Packet Sniffer [34] della Nordic Semiconductor per osservare i pacchetti inviati via BLE dai dispositivi vicini.

3.6 Hardware e software esterni

A supporto dell' hardware presentato, sono stati utilizzati dei software e dell'hardware aggiuntivo, che hanno permesso lo sviluppo del codice per il Microcontrollore e per l'FPGA, l'esecuzione dei vari test. Vengono qui esposti a conclusione del capitolo, indicando la versione utilizzata durante questo lavoro.

Segger Embedded Studio

Come ambiente di sviluppo integrato (IDE) è stato scelto Segger Embedded Studio (SES) [35] in quanto si integra perfettamente con l'SDK 15.2 della Nordic [36], ovvero il Software Development Kit contenente le librerie con cui è stato programmato il Microcontrollore in tutte le fasi di questo lavoro. Permette inoltre di caricare i binari e avviare la procedura di debug in modo semplice e veloce. Nello specifico è stata utilizzata la versione 5.42a.



Figura 3.12:
Segger Embedded Studio

nRF Connect for Desktop



Figura 3.13:
nRF Connect for Desktop

Per connettersi al Microcontrollore dal PC, sfruttando il Dongle nRF52840 come interfaccia Bluetooth, è stato utilizzato nRF Connect for Desktop [37], della Nordic Semiconductor. Questo comprende diversi strumenti, dei quali sono stati utilizzati:

- Bluetooth Low Energy, per lo sviluppo e il test di applicazioni BLE;
- Programmer, per programmare i chip della Nordic (in questo caso solo il Dongle, visto che le nRF52 sono state programmate da Segger Embedded Studio).

E' consigliabile utilizzare sempre l'ultima versione di questi software.

Wireshark

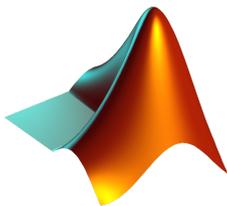
Wireshark [38] è stato un altro software molto utile, che in abbinamento al Packet Sniffer della Nordic Semiconductor [34], ha permesso durante i test sul BLE di individuare, in tempo reale, i pacchetti inviati e ricevuti da un dispositivo, consentendo uno studio della trasmissione.

La versione di Wireshark utilizzata in questo progetto è la 4.0.4, mentre per il Packet Sniffer è stata usata la 4.1.1.



Figura 3.14:
Wireshark

MATLAB



Per l'esecuzione dei test di velocità, che vedremo nel capitolo 4 è stato utilizzato MATLAB [39], nella sua versione R2023a, tramite il quale sono stati inviati i comandi ed elaborati i dati ricevuti.

Figura 3.15:
MATLAB

iCEcube2

Passando invece ai software per l'FPGA, come Ambiente di Sviluppo Integrato è stato utilizzato iCEcube2 [40] della Lattice Semiconductor, fornito con l'FPGA iCE40 Ultra [28]. Questo permette di caricare i file Verilog o VHDL del proprio progetto, aiuta nella verifica dei moduli implementati e consente di compilare i codici, di effettuare il Placement&Route e infine di generare l'immagine (.bmp) da caricare sull'FPGA.

La versione del software utilizzata durante questo lavoro è la 2020.12.



Figura 3.16:
iCEcube2

Diamond Programmer



Figura 3.17:
Diamond
Programmer

Per effettuare operazioni di erase, flashing e verifica del programma sul kit di sviluppo dell'FPGA iCE40 Ultra 3.8 è stato usato, come consigliato dal manuale del kit, Diamond Programmer [41], sempre della Lattice Semiconductor.

La versione utilizzata in questo progetto è stata la 3.12.1.

ModelSim

Infine, per la simulazione dei moduli VHDL, è stato utilizzato ModelSim 3.18, della Mentor. Grazie a questo software è possibile controllare ogni forma d'onda, in ingresso e in uscita da ogni modulo implementato (non solo del top-level), il che è molto utile in fase di debug per individuare errori sistematici insiti nel codice.

Nello specifico, è stato usato ModelSim Lattice FPGA Edition, nella versione 2020.3, compreso nell'installazione di iCEcube2 3.16.



Figura 3.18:
ModelSim

ModelSim



Figura 3.19:
SPIDriver

Per comprendere il protocollo SPI del Microcontrollore e per testare il corretto funzionamento del modulo SPI Slave implementato sull'FPGA (Sez. 4.2.1), è stato utilizzato l'SPIDriver della Excamera Labs, che permette di pilotare dispositivi SPI tramite un'interfaccia grafica su PC.

Capitolo 4

Sviluppo e Test del Software

In questo capitolo si analizzano le diverse fasi dello sviluppo del software che sarà caricato sul Microcontrollore e sull'FPGA. Prima di arrivare ai codici finali (nelle Sez. 4.1.2 e 4.2.2) sono stati progettati e testati dei codici intermedi, per verificare ogni parte del sistema Microcontrollore-FPGA.

Innanzitutto, è stato studiato nel dettaglio il codice per Microcontrollore fornito con "Senseback" [11] dal NGNI Lab dell'Imperial Collage di Londra attraverso le relative repository Github [14, 15]. Dopo averlo compreso, compilato e commentato in modo da rendere chiaro lo scopo di ogni funzione, è servito da spunto per l'implementazione dei codici successivi, che analizzeremo a breve.

Per quanto riguarda invece i codici per l'FPGA, non ci sono stati riferimenti. Il codice è stato sviluppato da zero, utilizzando il linguaggio descrittivo VHDL. Sono prima stati implementati, simulati e testati i singoli blocchi per poi arrivare al codice descritto in Sez. 4.2.2.

In ultima analisi, verrà analizzato il sistema MCU-FPGA risultante (Sez. 4.2.3) in cui vengono utilizzati i codici definitivi presentati nelle sezioni 4.1.2 e 4.2.2.

4.1 Sviluppo del codice per il Microcontrollore

Nello sviluppo del Microcontrollore, dopo l'implementazione di diversi codici di prova, utili per settare la toolchain (ma che non sono d'interesse per questa tesi), sono state due le fasi più importanti di questa parte del lavoro:

- nella prima si simula un flusso di dati proveniente dall'ASIC, generando i dati internamente al Microcontrollore, in risposta ad un comando inviato da PC via BLE;

- nella seconda si implementa il codice finale che si interfacerà con l’FPGA (Fig. 3.8) e con la periferica BLE (Fig. 3.11) tramite il quale il PC invierà i comandi (come si vedrà nella Sez. 4.2.3).

4.1.1 Dati generati internamente al Microcontrollore e inviati via BLE

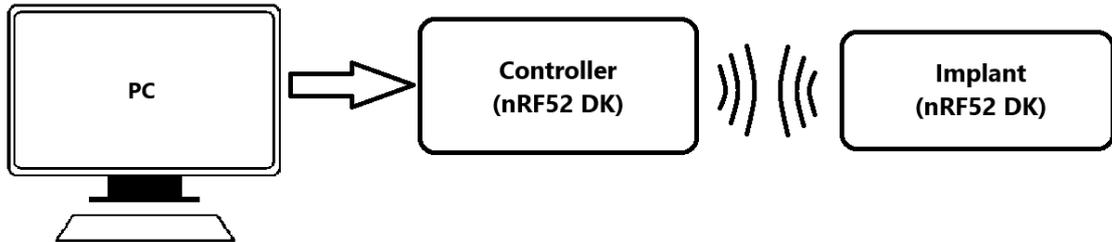


Figura 4.1: Schema a blocchi del sistema

In questa parte del lavoro, è stato implementato un programma che risponde ai comandi di START e STOP, inviati dal PC via BLE, azionando o arrestando la generazione di dati interna al Microcontrollore (Fig. 4.1). Per lo sviluppo di questo codice si è partiti da un esempio base, presente nell’SDK 15.2 della Nordic [36], chiamato *ble_app_uart*. Partendo da questo esempio e modificandolo ispirandosi a "SenseBack" [11, 14], è stato sviluppato il codice *01_Implant_dim_pacchetti_fissa*. In aggiunta, partendo da un altro esempio della Nordic (*ble_app_uart_c*) è stato sviluppato il codice *01_Controller_per_testing_throughput* che ricevendo tramite UART il comando di START dal PC, lo invia al precedente dispositivo via BLE. Il Controller inoltre, riceverà via BLE i dati generati dall’Implant e calcolerà il data rate con cui questi dati vengono ricevuti.

01_Implant_dim_pacchetti_fissa

Rispetto all’esempio *ble_app_uart*, un programma che invia via UART quello che riceve via BLE e viceversa, è stata aggiunta la libreria esterna *ringbuf.h* (Cod. 4.1) che permette l’utilizzo, appunto, dei ring buffer. Questi, data la loro struttura ad anello, si prestano bene come registri FIFO (First In First Out) e sono stati sfruttati per immagazzinare i dati generati, in attesa di essere trasmessi via BLE. Permettono inoltre di sovrascrivere gli elementi più vecchi nel caso i buffer fossero pieni ma ci siano nuovi elementi da ricevere.

```
1 include "ringbuf.h"
```

Cod. 4.1: Inclusione della libreria ringbuffer

E' stata poi definita l'istanza del *timer1* (Cod. 4.2), il quale viene configurato per essere utilizzato come contatore per generare i dati. *TIMER1_INTERVAL_US* definisce l'intervallo di tempo in μs passato il quale viene generato un interrupt. Poiché si vuole simulare un flusso di dati in ingresso a frequenza 80 kHz, a *TIMER1_INTERVAL_US* viene assegnato il valore 12.5 (Eq. 4.1), sottintendendo μs come unità di misura.

```
1
2 const nrfx_timer_t timer1 = NRFX_TIMER_INSTANCE(1);
3 #define TIMER1_INTERVAL_US 12.5
4
5 static void timers_init(void)
6 {
7 nrfx_timer_config_t timer1_config = NRFX_TIMER_DEFAULT_CONFIG;
8 timer1_config.frequency = NRF_TIMER_FREQ_2MHz; //risoluzione del
9 timer
10 /*Viene inizializzato il timer, con la configurazione precedente
11 e l'indicazione dell'Interrupt Service Routine (timer1_handler)*/
12 err_code = nrfx_timer_init(&timer1, &timer1_config, timer1_handler);
13 APP_ERROR_CHECK(err_code);
14
15 /*Qui viene convertito il tempo da us in ticks (clock alla frequenza
16 di risoluzione del timer, impostata a 2MHz nelle righe prima)*/
17 uint32_t time_ticks = nrfx_timer_us_to_ticks(&timer1,
18 TIMER1_INTERVAL_US) + 1;
19 NRF_LOG_DEBUG("ticks timer1: %u", time_ticks);
20
21 /*Viene attivato l'interrupt*/
22 nrfx_timer_extended_compare(&timer1, NRF_TIMER_CC_CHANNEL1,
23 time_ticks, NRF_TIMER_SHORT_COMPARE1_CLEAR_MASK, true);
24 }
```

Cod. 4.2: timer1

$$TIMER1_INTERVAL_US = \frac{1}{80 \text{ kHz}} = 12.2 \text{ ms} \quad (4.1)$$

Nell'inizializzazione di questo timer (Cod. 4.2), viene richiamata l'ISR (Interrupt Service Routine) *timer1_handler* (Cod. 4.3), che è la funzione che si occuperà, ad ogni interrupt del *timer1*, di incrementare il contatore *CNT* e di inserire il

risultato del contatore (il dato) nel ring buffer *uartRx*, dove vengono conservati i dati in attesa di essere trasmessi via BLE. La variabile *data_arrived* conta quanti dati sono stati inseriti nel buffer *uartRX* e raggiunto il numero di dati da inviare (definito da *DATA_TO_SEND*), vengono resettati i contatori, inserito l'ultimo elemento in coda a *uartRX* e disattivata l'ISR.

```
1
2 static void timer1_handler(nrf_timer_event_t event_type, void*
3     p_context)
4 {
5     switch(event_type)
6     {
7         case NRF_TIMER_EVENT_COMPARE1: //In caso di interrupt del
8             timer1
9                 if(transmission) //con transmission attivo/disattivo
10                    questa routine
11                        {
12                            CNT++; //ad ogni richiamo conto
13                            ringbuf_put(&uartRx,CNT); //metto il dato in coda
14                            per l'invio via BLE
15
16                            /*incremento il contatore dei dati (simulati)
17                            ricevuti dall'FPGA (simulati)*/
18                            data_arrived++;
19
20                            /*Quando ricevo l'ultimo dato*/
21                            if(data_arrived > (((DATA_TO_SEND-2)/2)-1))
22                            {
23                                data_arrived = 0; //resetto il contatore dei
24                                dati (simulati) ricevuti
25                                CNT = 0;
26                                ringbuf_put(&uartRx,65535); //metto l'ultimo
27                                dato in coda per l'invio via BLE
28                                transmission = false; //disattivo
29                                questa routine
30                                    }
31                                }
32                                break;
33
34                            default:
35                                break;
36                    }
37 }
38 }
```

Cod. 4.3: ISR del timer1

I dati generati ed inseriti in *uartRx* vengono poi gestiti dalla funzione *spiBuffProcess* (Cod. 4.4) che viene richiamata ogni volta che il dispositivo sta andando in idle state (per risparmiare energia). In questa funzione ci si accerta che i pacchetti inviati, la cui dimensione è definita da *SENSEBACK_MTU* (Cod.4.5) siano completamente pieni. In questa maniera viene massimizzato il throughput.

Successivamente, in questa routine, viene assegnato il valore logico *true* alla variabile *txActive*, per segnalare che da questo momento sta partendo la trasmissione Bluetooth e subito dopo, viene richiamata la funzione *notification_send* (Cod. 4.6), la quale si occupa di inviare il pacchetto (*nusTx*), appena riempito, via BLE. Raggiunto il numero di dati da inviare, viene gestito l'ultimo pacchetto, che potrebbe non venire riempito completamente. Questo dipende da quanti dati sono rimasti in *uartRx*, quindi dal valore di *queueLength* (Cod. 4.4).

```

1
2 static void spiBuffProcess ()
3 {
4     uint16_t element;
5
6     //Gestione del dato generato dai timer
7
8     if (!txActive)      //se non si sta già inviando un pacchetto via
9     BLE
10    {
11        uint16_t queueLength = ringbuf_elements(&uartRx);    //vedo
12        quanti elementi ci sono nel ring buffer
13
14        /*se ho i dati per riempire un pacchetto li invio (diviso due
15        perche' gli elementi nel ringbuffer sono uint16 mentre quelli nel
16        buffer di trasmissione BLE sono uint8). In questo modo mi accerto
17        di inviare pacchetti pieni e massimizzare il throughput*/
18        if (queueLength >= (SENSEBACK_MTU/2))
19        {
20            for (int i=0; i < (SENSEBACK_MTU/2); i++) //riempio
21            completamente il pacchetto
22            {
23                element = ringbuf_get(&uartRx);
24                nusTx[i*2] = (uint8_t)element;    //riempio i buffer
25                di trasmissione BLE con i dati nel ringbuffer
26                nusTx[i*2+1] = element >> 8;
27            }
28            txActive = true;    //inizio trasmissione BLE
29            txSize = SENSEBACK_MTU; //dimensione del pacchetto da
30            inviare
31            txRdPtr = 0;    //posizione del vettore da cui iniziare
32            ad inviare i dati
33            notification_send();    //invio i dati via BLE

```

```

25     }
26     /*Se raggiungo il numero di dati da inviare, invio l'ultimo
pacchetto*/
27     else if((queueLength>0) && (data_sent>(DATA_TO_SEND-
SENSEBACK_MTU)))
28     {
29         /*Non è detto che l'ultimo pacchetto sia completamente
pieno*/
30         NRF_LOG_INFO("Last packet size /2: %u", queueLength);
31         for(int i=0; i<queueLength; i++)
32         {
33             element = ringbuf_get(&uartRx);
34             nusTx[i*2] = (uint8_t)element; //riempio i buffer
di trasmissione BLE con i dati nel ringbuffer
35             nusTx[i*2+1] = element>>8;
36         }
37
38         txActive = true; //inizio trasmissione BLE
39         txSize = 2*queueLength; //dimensione del pacchetto da
inviare
40         txRdPtr = 0; //posizione del vettore da cui iniziare
ad inviare i dati
41         notification_send(); //invio i dati via BLE
42     }
43 }
44 }

```

Cod. 4.4: Gestione dei dati generati

```

1
2 #define SENSEBACK_MTU 244 //dimensione del pacchetto scelta
3 static uint8_t nusTx[SENSEBACK_MTU]; //buffer per trasmissione BLE

```

Cod. 4.5: Definizione del numero di elementi nel pacchetto

Nella funzione *notification_send* viene controllata la dimensione del pacchetto da inviare. Se è più grande di *SENSEBACK_MTU* viene spezzato in 2 pacchetti:

- uno della dimensione prefissata;
- l'altro contenente gli elementi residui.

Poiché in *spiBuffProcess* ci siamo accertati di riempire pacchetti della dimensione di *SENSEBACK_MTU*, nessun pacchetto verrà spezzato, in quanto tutti i pacchetti saranno della dimensione scelta, tranne l'ultimo che potrebbe contenere meno elementi.

Successivamente il pacchetto viene inviato via BLE, come Notification (si veda "Parametri e throughput" nella Sez. 3.1.1), tramite la funzione *ble_nus_data_send*,

predefinita dalle librerie Nordic. Ad ogni invio con successo viene incrementata la variabile `data_sent` che tiene il conto dei dati inviati.

Come ultimo step, a trasmissione completata, viene reimpostata `txActive` su `false`, in modo da riconsentire a `spiBuffProcess` di impacchettare i dati salvati in `uartRx` (Cod. 4.4).

```

1
2 void notification_send()
3 {
4     uint16_t length;
5     uint32_t err_code = NRF_SUCCESS;
6     while (err_code == NRF_SUCCESS && txRdPtr < txSize)
7     {
8         /*Se il pacchetto è più grande di SENSEBACK_MTU, viene
9         spezzato*/
10        length = ((txSize - txRdPtr) > SENSEBACK_MTU) ? SENSEBACK_MTU
11        : (txSize - txRdPtr);
12
13        /*Invio del dato via BLE*/
14        err_code = ble_nus_data_send(&m_nus, &nusTx[txRdPtr], &length
15        , m_conn_handle);
16
17        if ((err_code != NRF_ERROR_INVALID_STATE) &&
18            (err_code != NRF_ERROR_RESOURCES) &&
19            (err_code != NRF_ERROR_NOT_FOUND))
20        {
21            APP_ERROR_CHECK(err_code);
22        }
23        if (err_code == NRF_SUCCESS)
24        {
25            txRdPtr += length;
26            data_sent += length;
27        }
28        if (err_code == NRF_ERROR_RESOURCES) bleTxBusy = true;
29    }
30    if (txRdPtr >= txSize) txActive = false;
31 }

```

Cod. 4.6: Invio dei dati via BLE

Per quanto riguarda, invece, i comandi ricevuti via BLE, questi vengono gestiti nella funzione `nus_data_handler` (Cod. 4.7). In particolare, il comando ricevuto viene salvato in `dongle_command` in modo da poter essere identificato:

- se si tratta del comando di START, viene assegnato il valore `true` alla variabile `transmission` in modo da far partire il flusso dati (Cod. 4.3);

- se il comando corrisponde a quello di RESET, viene resettato il contatore *CNT*, ma non viene alterato lo stato di *transmission* (se si stavano generando dati, questi continueranno ad essere generati);
- se viene ricevuto il comando di STOP viene disabilitato il *timer1* (Cod. 4.2) che si occupa della generazione di dati;
- qualsiasi altro comando viene semplicemente ignorato.

```

1
2 static void nus_data_handler(ble_nus_evt_t * p_evt)
3 {
4     if (p_evt->type == BLE_NUS_EVT_RX_DATA)
5     {
6         uint32_t err_code;
7         /*Salvo il comando ricevuto via ble nella variabile
8         dongle_command*/
9         char dongle_command[BLE_NUS_MAX_DATA_LEN];
10        memset (dongle_command,0,BLE_NUS_MAX_DATA_LEN);
11        memcpy (dongle_command, p_evt->params.rx_data.p_data, p_evt->
12        params.rx_data.length);
13        ... ..
14        if(strcmp(dongle_command, "START\n") == 0 ) //Se ricevo il
15        comando di START
16        {
17            transmission = true; //faccio partire il flusso dati
18            memcpy (dongle_command, 0, 1); //resetto il comando
19        }
20        else if(strcmp(dongle_command, "RESET\n") == 0 ) //Se ricevo
21        il comando di RESET
22        {
23            CNT = 0;
24        }
25        else if(strcmp(dongle_command, "STOP\n") == 0 ) //Se ricevo
26        il comando di STOP
27        {
28            nrfx_timer_disable(&timer1); //disabilito il timer che
29            genera dati
30        }
31    }
32 }

```

Cod. 4.7: Identificazione del comando ricevuto

01_Controller_per_testing_throughput

Come anticipato, il Controller è utilizzato per far partire la trasmissione dall'Implant e per misurare il data rate con cui i dati vengono trasmessi via BLE. Si è partiti dall'esempio Nordic *ble_app_uart_c* il quale implementava un programma capace di inviare via BLE i dati ricevuti sull'UART (percorso sfruttato per inviare il comando di START) e viceversa, scrivere sull'UART i dati ricevuti via BLE. Sono poi state apportate modifiche per permettere il calcolo del data rate.

Come prima cosa è stato inizializzato il *timer1* (Cod. 4.8) che si occuperà di misurare i ms trascorsi dall'inizio alla fine della trasmissione dei dati. Il timer è inizializzato in maniera analoga al precedente, ma in questo caso nell'ISR del timer ci sarà esclusivamente l'incremento del contatore dei millisecondi.

```
1
2 const nrfx_timer_t timer1 = NRFX_TIMER_INSTANCE(1);
3 #define TIMER1_INTERVAL_MS 1
4
5 static void timer1_handler(nrf_timer_event_t event_type, void*
6     p_context)
7 {
8     switch(event_type)
9     {
10         case NRF_TIMER_EVENT_COMPARE1:
11             time_elapsed_ms += 1;
12             break;
13
14         default:
15             break;
16     }
17 }
18 static void timer_init(void)
19 {
20     ret_code_t err_code = app_timer_init();
21     APP_ERROR_CHECK(err_code);
22
23     /*Timer per il calcolo del datarate*/
24     nrfx_timer_config_t timer1_config = NRFX_TIMER_DEFAULT_CONFIG;
25     timer1_config.frequency = NRF_TIMER_FREQ_2MHz;
26     err_code = nrfx_timer_init(&timer1, &timer1_config,
27     timer1_handler);
28     APP_ERROR_CHECK(err_code);
29
30     uint32_t time_ticks = nrfx_timer_ms_to_ticks(&timer1,
31     TIMER1_INTERVAL_MS);
32     NRF_LOG_INFO("ticks timer1: %u", time_ticks);
33 }
```

```

32     nrfx_timer_extended_compare(&timer1, NRF_TIMER_CC_CHANNEL1,
time_ticks, NRF_TIMER_SHORT_COMPARE1_CLEAR_MASK, true);
33 }

```

Cod. 4.8: Timer per calcolo data rate

Alla funzione `ble_nus_c_evt_handler` è stata poi aggiunta una porzione di codice (Cod. 4.9) in corrispondenza dell'evento `BLE_NUS_C_EVT_NUS_TX_EVT`. In questo modo, al primo dato ricevuto dal Controller il timer viene abilitato e una volta ricevuti tutti i dati, viene calcolato il data rate complessivo (in bpms), come il rapporto tra i bit ricevuti e il tempo trascorso tra il primo e l'ultimo (Eq. 4.2, $N_{byte_{ricevuti}}$ è il numero di byte ricevuti e $t_{passato}$ è il tempo trascorso tra il primo e l'ultimo dato ricevuto).

```

1
2 static void ble_nus_c_evt_handler(ble_nus_c_t * p_ble_nus_c,
ble_nus_c_evt_t const * p_ble_nus_evt)
3 {
4     ret_code_t err_code;
5
6     switch (p_ble_nus_evt->evt_type)
7     {
8         case BLE_NUS_C_EVT_DISCOVERY_COMPLETE:
9             ...
10            break;
11        case BLE_NUS_C_EVT_NUS_TX_EVT:
12            {
13                /*Incremento il numero di dati ricevuto*/
14                received_bytes += p_ble_nus_evt->data_len;
15                if (first)
16                {
17                    first = false;
18                    time_elapsed_ms = 0;
19                    nrfx_timer_enable(&timer1); //avvio il timer
20                }
21                /*Riempio senseback_buff coi dati ricevuti*/
22                for (int i=0;i<p_ble_nus_evt->data_len;i++)
23                    senseback_buff[senseback_idx][i] = * (p_ble_nus_evt->p_data + i);
24                /*Quando ricevo gli ultimi dati calcolo il data rate*/
25                if ((senseback_buff[senseback_idx][p_ble_nus_evt->data_len
-1] == 255)&&(senseback_buff[senseback_idx][p_ble_nus_evt->
data_len-2] == 255))
26                {
27                    uint32_t ticksVal = nrfx_timer_capture(&timer1,
NRF_TIMER_CC_CHANNEL1);
28                    float tVal = (float) ticksVal*0.0005;
29                    float time_elapsed = (float) time_elapsed_ms + tVal;

```

```

29         NRF_LOG_INFO("Time elapsed total: "
NRF_LOG_FLOAT_MARKER " ms", NRF_LOG_FLOAT(time_elapsed));
30         NRF_LOG_INFO("Received bytes: %u", received_bytes);
31         float datarate = (float)received_bytes*8/(
time_elapsed);
32         NRF_LOG_INFO("Datarate: " NRF_LOG_FLOAT_MARKER " bits
per ms", NRF_LOG_FLOAT(datarate));
33         received_bytes = 0;
34     }
35     senseback_buff_length[senseback_idx] = p_ble_nus_evt->
data_len;
36     if (((senseback_idx+1) % SENSEBACK_BUFFER_COUNT) ==
prev_senseback_idx) NRF_LOG_ERROR("Data lost");
37     senseback_idx = (senseback_idx+1) %
SENSEBACK_BUFFER_COUNT;
38
39     break;
40 }
41 case BLE_NUS_C_EVT_DISCONNECTED:
42     ...;
43     break;
44 }
45 }

```

Cod. 4.9: Calcolo data rate

$$Datarate = \frac{8 \cdot N_{byte_{ricevuti}}}{t_{passato}} 10^4 \quad (4.2)$$

Infine i dati ricevuti via BLE dal Controller, vengono inviati attraverso l'UART al PC tramite la funzione `ble_nus_chars_received_uart_print` (Cod. 4.10) richiamata nel `main` del codice. In questo modo sarà poi possibile rappresentarli con MATLAB per controllare che siano stati ricevuti correttamente.

```

1
2 static void ble_nus_chars_received_uart_print(uint8_t * p_data,
uint16_t data_len)
3 {
4     ret_code_t ret_val;
5
6     for (uint32_t i = 0; i < data_len; i++)
7     {
8         do
9         {
10             data_sent_uart++;
11             ret_val = app_uart_put(p_data[i]);

```

```

12         if ((ret_val != NRF_SUCCESS) && (ret_val !=
NRF_ERROR_NO_MEM))
13             {
14                 NRF_LOG_ERROR("app_uart_put failed for index 0x%04x."
, i);
15                 APP_ERROR_CHECK(ret_val);
16             }
17         } while (ret_val == NRF_ERROR_NO_MEM);
18     }
19     if (ECHOBACK_BLE_UART_DATA) //non di interesse perche'
disabilitata
20     {
21         ...;
22     }
23
24 int main(void)
25 {
26     ...;
27     // Enter main loop.
28     for (;;)
29     {
30         while(prev_senseback_idx != senseback_idx)
31         {
32             /*Invio dati tramite UART*/
33             ble_nus_chars_received_uart_print(senseback_buff[
prev_senseback_idx], senseback_buff_length[prev_senseback_idx]);
34             prev_senseback_idx = (prev_senseback_idx+1) %
SENSEBACK_BUFFER_COUNT;
35         }
36
37         idle_state_handle();
38     }
39 }

```

Cod. 4.10: Invio dati tramite UART

Test per ottimizzazione della trasmissione Bluetooth

Dopo aver implementato i due codici, è stato condotto uno studio sui parametri che influenzano maggiormente il throughput, i quali, come anticipato nella Sez. 3.1.1, risultano essere:

- il Physical Layer (PHY);
- l'ATT MTU;
- il Connection Interval;

- il tipo di operazione.

Il tipo di operazione selezionato è Notification, scelta implicita con l'uso del Nordic UART Service (NUS), ovvero il dato viene inviato dall'Implant al Controller come notifica, senza attendere una risposta.

Il Physical Layer scelto è il 2Mbps PHY, quello con la velocità teorica più alta e viene impostato dall'Implant, nella funzione *ble_evt_handler*, nel momento in cui si verifica un evento di connessione (Cod. 4.11).

```

1
2 static void ble_evt_handler(ble_evt_t const * p_ble_evt, void *
   p_context)
3 {
4     uint32_t err_code;
5     ble_gap_evt_t const * p_gap_evt = &p_ble_evt->evt.gap_evt;
6
7     switch (p_ble_evt->header.evt_id)
8     {
9         /*Evento di connessione*/
10        case BLE_GAP_EVT_CONNECTED:
11            NRF_LOG_INFO("Connected");
12            err_code = bsp_indication_set(BSP_INDICATE_CONNECTED);
13            APP_ERROR_CHECK(err_code);
14            m_conn_handle = p_ble_evt->evt.gap_evt.conn_handle;
15            err_code = nrf_ble_qwr_conn_handle_assign(&m_qwr,
16            m_conn_handle);
17            APP_ERROR_CHECK(err_code);
18            ble_gap_phys_t const phys =
19            {
20                .rx_phys = BLE_GAP_PHY_2MBPS,           //Physical layer
21                .tx_phys = BLE_GAP_PHY_2MBPS,
22                };
23            err_code = sd_ble_gap_phy_update(p_gap_evt->conn_handle,
24            &phys);
25            APP_ERROR_CHECK(err_code);
26
27            nrfx_timer_enable(&timer1);
28
29            break;
30
31        case ...:
32            ...;
33            break;
34
35        default:
36            break;
37    }

```

 36 }

Cod. 4.11: Impostazione del PHY

Questi due parametri restano fissi durante tutta l'esecuzione del test, così come anche il numero di dati da inviare che sarà sempre 32400, numero oltre il quale MATLAB non è più in grado di elaborare i dati correttamente.

Per quanto riguarda i restanti due parametri, sono state provate diverse combinazioni di ATT MTU e Connection Interval (Cod. 4.12) in modo da individuare le combinazioni che consentono di avere il throughput maggiore.

```

1
2 /*Numero di dati da inviare*/
3 #define DATA_TO_SEND 32400
4
5 /*Dimensione del pacchetto*/
6 #define SENSEBACK_MTU 244
7
8 /*Connection interval (impostare min e max allo stesso valore)*/
9 #define MIN_CONN_INTERVAL MSEC_TO_UNITS(200, UNIT_1_25_MS)
10 #define MAX_CONN_INTERVAL MSEC_TO_UNITS(200, UNIT_1_25_MS)

```

Cod. 4.12: Impostazione ATT MTU e Connection Interval

I test sono stati eseguiti sfruttando due nRF52 DK (Fig. 3.6), uno per l'Implant e uno per il Controller. I due codici, sono stati caricati sulle due board in modalità Debug, tramite SES (Fig. 3.12), in modo da poter leggere i log nel Debug Terminal. In aggiunta è stato utilizzato un Dongle nrf52840 (Fig. 3.11), con caricato il Packet Sniffer [34], con il supporto di Wireshark (Fig. 3.14). Insieme hanno permesso di osservare i pacchetti inviati ed individuare eventuali ritrasmissioni dello stesso pacchetto, dovute a eventuali errori di trasmissione, che rallentano il data rate. Infine è stato usato MATLAB come interfaccia tra il PC e l'UART del Controller, per l'invio del comando di START e per la rappresentazione dei dati ricevuti. Il set-up complessivo è rappresentato in Fig. 4.2 mentre in Fig. 4.3-4.4 è possibile vedere le schermate di SES, MATLAB e Wireshark.

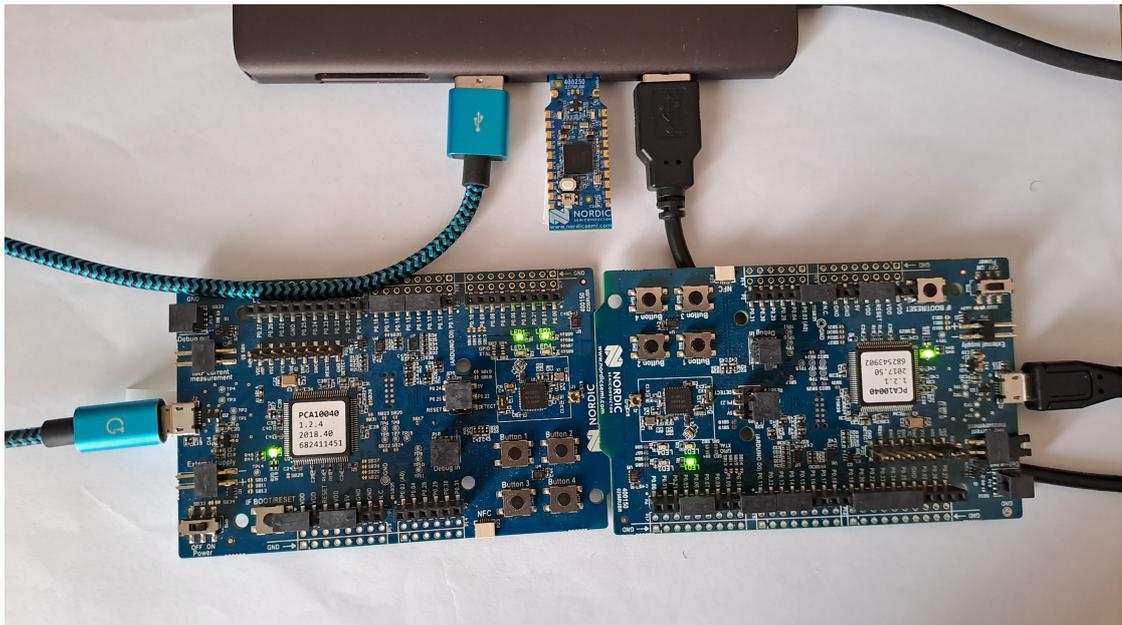


Figura 4.2: Set-up per il test di velocità di trasmissione dati

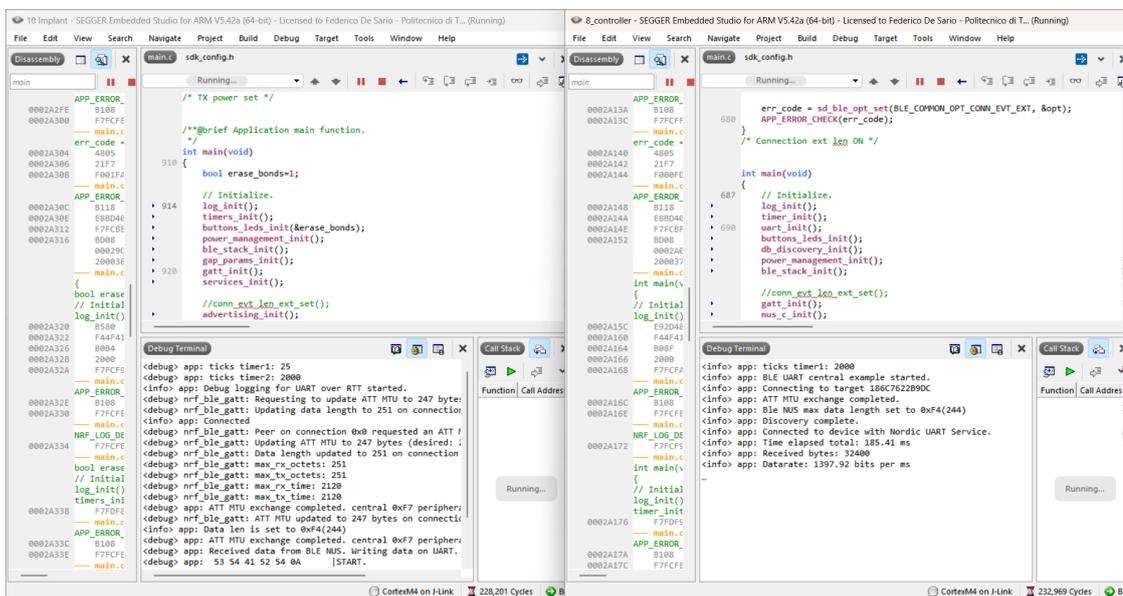


Figura 4.3: Schermate di SES dell'Implant e del Controller nei test di velocità di trasmissione dati

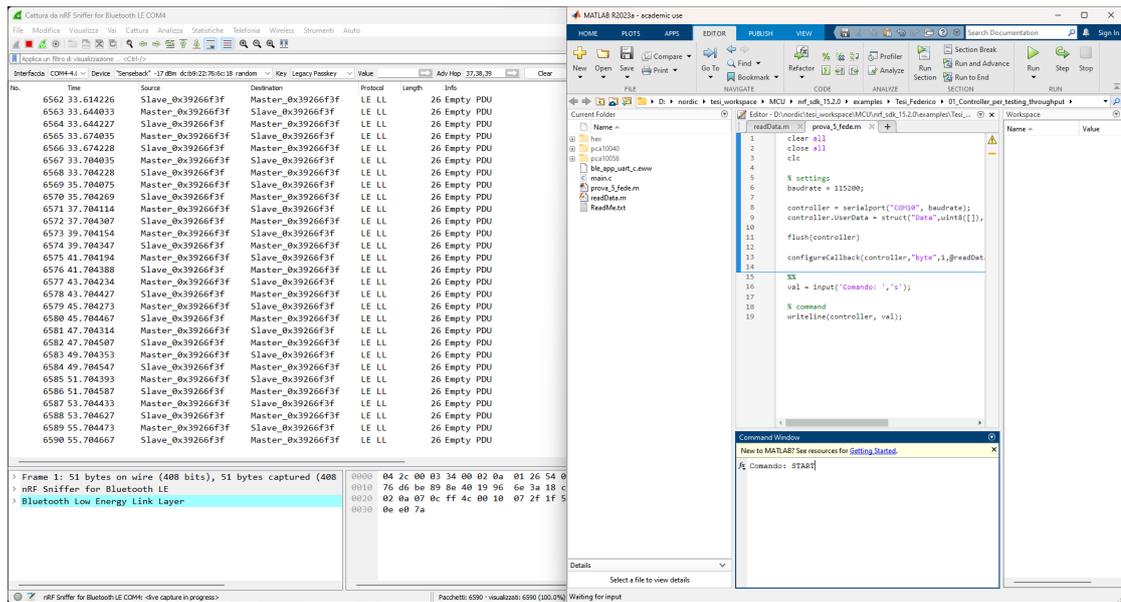


Figura 4.4: Schermate di MATLAB e Wireshark nei test di velocità di trasmissione dati

Anche la posizione, come la distanza tra le board influisce sul throughput. La situazione migliore, con la massima velocità di trasmissione, si verifica posizionando le board come in Fig. 4.2.

Sul Debug Terminal del Controller (a destra in Fig. 4.3) è possibile leggere il data rate espresso in bit per millisecondi (Fig. 4.5). Per ottenere il risultato in Mbps basterà moltiplicare per 10^{-3} questo risultato.

```

Debug Terminal
<info> app: ticks timer1: 2000
<info> app: BLE UART central example started.
<info> app: Connecting to target 186C7622B9DC
<info> app: ATT MTU exchange completed.
<info> app: Ble NUS max data length set to 0xF4(244)
<info> app: Discovery complete.
<info> app: Connected to device with Nordic UART Service.
<info> app: Time elapsed total: 185.41 ms
<info> app: Received bytes: 32400
<info> app: Datarate: 1397.92 bits per ms

```

Figura 4.5: Debug Terminal del Controller a fine test

Sul grafico generato da MATLAB invece, è possibile verificare che durante il test sono stati ricevuti correttamente tutti i dati (Fig. 4.6). Infatti si vede che questi

vengono ricevuti in sequenza, da 0 a 255, in quanto il contatore che li genera è da 16 bit e dunque il massimo valore senza segno rappresentabile è 255, dopodiché si resetta.

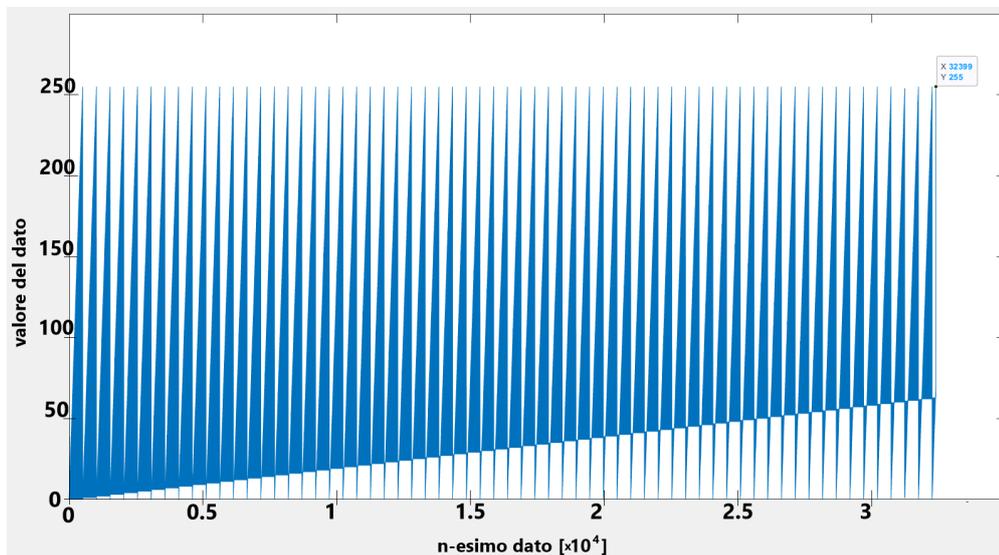


Figura 4.6: Grafico dei dati ricevuti durante il test di velocità. Sulle ascisse è segnato l'n-esimo dato ricevuto e sulle ordinate il suo valore. Si vede come i dati vengono ricevuti in sequenza da 0 a 255.

Come anticipato sono state provate diverse combinazioni di ATT MTU e Connection Interval. Per ogni combinazione sono stati effettuati 6 test, calcolando in ogni test il throughput in kbps, il numero di ritrasmissioni ed il Packet Error Rate (PER), ovvero il rapporto tra ritrasmissioni e il totale dei pacchetti ricevuti. Infine come valore finale è stata considerata la media delle 6 misure. In Tab. 4.1 sono state riportate, a titolo di esempio, alcune combinazioni utilizzate con i relativi risultati. In Fig. 4.7 sono stati riassunti tutti i risultati in un grafico e si vede come i risultati migliori si hanno in questo caso con la dimensione massima dei pacchetti (244 byte) e con Connection Interval compreso tra 200 ms e 4 s. Per questi valori infatti oltre ad avere ottenuto la massima velocità di trasmissione dei test, 1.4 Mbps, si ha che le ritrasmissioni sono praticamente nulle (Tab. 4.2).

Pacchetti da 2 byte	Connection Interval 7.5 ms			Connection Interval 12.5 ms		
	Throughput	Ritrasmissioni	PER	Throughput	Ritrasmissioni	PER
1	23.48	432.00	0.03	28.27	881.00	0.05
2	23.48	422.00	0.03	28.25	888.00	0.05
3	23.48	427.00	0.03	28.22	877.00	0.05
4	23.48	421.00	0.03	28.24	861.00	0.05
5	23.48	457.00	0.03	28.27	823.00	0.05
6	23.44	396.00	0.02	28.27	855.00	0.05
Media	23.47	425.83	0.03	28.17	864.17	0.05

Tabella 4.1: Alcuni esempi di combinazioni di Connection Interval e dimensioni dei pacchetti testate con relativi throughput, ritrasmissioni e Packet Error Rate (PER)

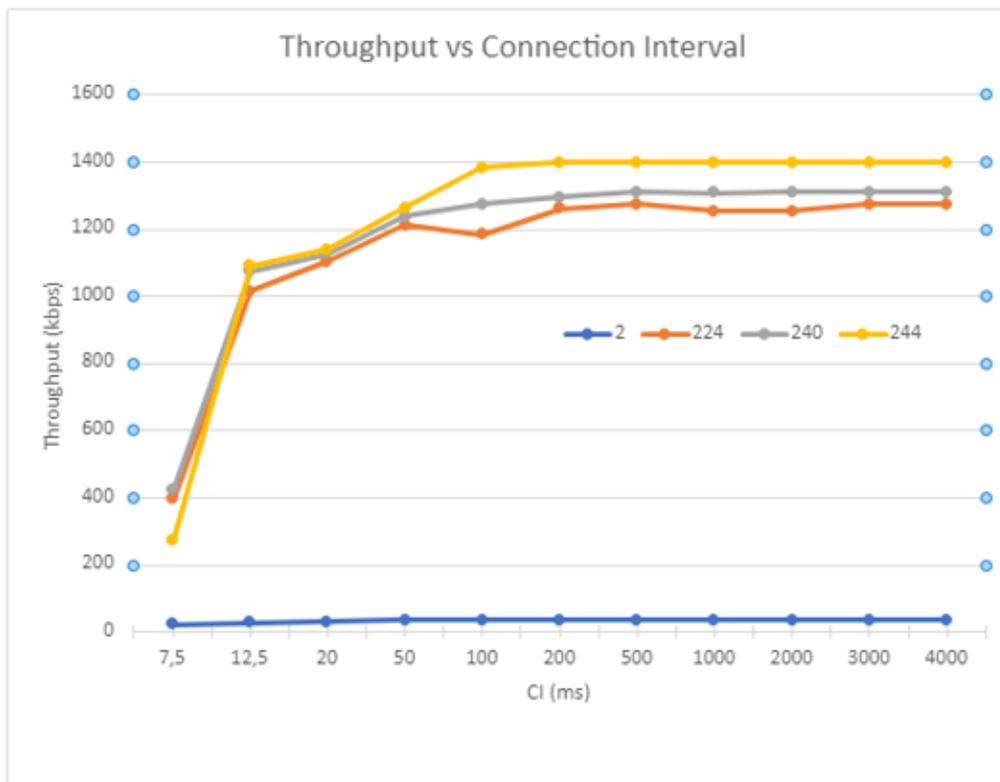


Figura 4.7: Grafico della velocità di trasmissione all'aumentare del CI per diverse dimensioni di pacchetto

Pacchetti da 244 byte	Connection Interval 200 ms		
Iterazione	Throughput	Ritrasmissioni	PER
1	1397.91	0.00	0.00
2	1397.79	1.00	0.01
3	1397.91	0.00	0.00
4	1397.78	0.00	0.00
5	1397.89	0.00	0.00
6	1397.89	0.00	0.00
Media	23.47	0.20	0.00

Tabella 4.2: Le ritrasmissioni diventano praticamente nulle con pacchetti da 244 byte e CI da 200 ms in su

4.1.2 Implementazione interfaccia SPI (Master) con trasmissione Bluetooth

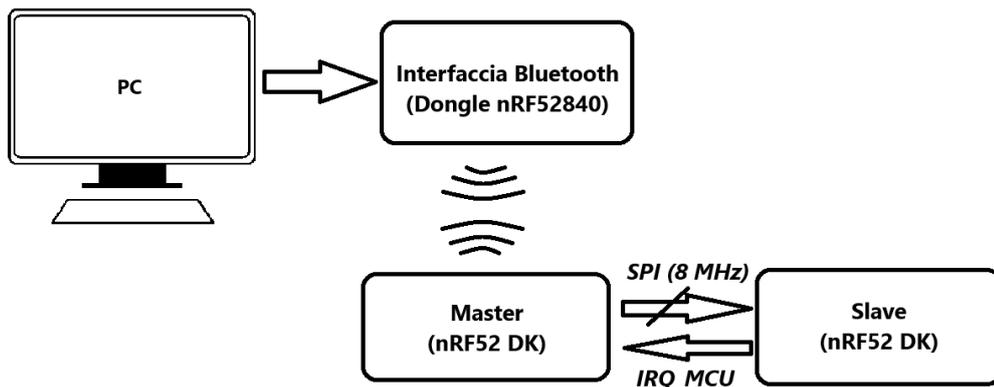


Figura 4.8: Schema a blocchi del sistema

In questa fase è stato implementato il codice `03_MCU_FPGAcounter`. Il programma ideato è simile al precedente, con la differenza che i dati non vengono generati internamente al Microprocessore, ma è stata implementata un'interfaccia SPI Master grazie alla quale è possibile comunicare con l'FPGA, la quale si occuperà di generare i dati nel sistema MCU-FPGA che verrà analizzato nella Sez. 4.2.3. Oltre all'interfaccia SPI è stata aggiunta un'Interrupt Service Routine (ISR) richiamata dagli impulsi sul pin di IRQ, che verranno inviati dall'FPGA. In questa maniera ricevuti il comando di START o di STOP via BLE, il Microcontrollore inoltra il comando via SPI all'FPGA, la quale risponderà come descritto nella Sez. 4.2.2. Viene poi effettuato un test del programma sfruttando un'apposita interfaccia SPI Slave, caricata su un altro Microcontrollore, in modo da poter sfruttare i Debug

Terminal di SES (Fig. 3.12) per verificare il corretto funzionamento del programma principale (Fig. 4.8).

03_MCU_FPGAcounter

Anche in questo caso, si è partiti dall'esempio della Nordic *ble_app_uart*, ma al posto della libreria esterna *ringbuf.h* è stata utilizzata la libreria della Nordic *nrf_queue.h* (Cod. 4.13) la quale permette a sua volta di lavorare con i ringbuffer, restando però sempre all'interno dello stesso SDK ([36]).

A questa libreria si aggiungono quella per implementare l'interfaccia SPI (*nrf_drv_spi.h*) e quella per gestire gli interrupt sulle GPIO (*nrf_drv_gpiote.h*).

```

1
2 #include "nrf_drv_spi.h" //aggiunto per usare l'spi
3 #include "nrf_drv_gpiote.h" //aggiunto per usare l'interrupt su
  IRQ_PIN
4 #include "nrf_queue.h" //aggiunto per implementare i FIFO register
  (ringbuffer)

```

Cod. 4.13: Inclusione delle librerie aggiuntive

Successivamente vengono inizializzati i ringbuffer, tramite le funzioni della libreria appena aggiunta (Cod. 4.14). Nello specifico:

- *m_spiTx_fifo* è il FIFO in cui vengono salvati i comandi ricevuti via BLE in attesa di essere trasmessi via SPI;
- *m_spiTx_fifo* è, invece, quello in cui vengono salvati i dati ricevuti via SPI, in attesa di essere inviati via BLE.

```

1
2 /*Registri FIFO per la trasmissione SPI*/
3
4 #define FIFO_BUFFER_SIZE 2048
5
6 NRF_QUEUE_DEF(uint16_t, m_spiTx_fifo, FIFO_BUFFER_SIZE,
  NRF_QUEUE_MODE_OVERFLOW);
7 NRF_QUEUE_INTERFACE_DEC(uint16_t, spiTx_fifo);
8 NRF_QUEUE_INTERFACE_DEF(uint16_t, spiTx_fifo, &m_spiTx_fifo);
9
10 NRF_QUEUE_DEF(uint16_t, m_spiRx_fifo, FIFO_BUFFER_SIZE,
  NRF_QUEUE_MODE_OVERFLOW);
11 NRF_QUEUE_INTERFACE_DEC(uint16_t, spiRx_fifo);
12 NRF_QUEUE_INTERFACE_DEF(uint16_t, spiRx_fifo, &m_spiRx_fifo);

```

Cod. 4.14: Inizializzazione dei FIFO (ringbuffer)

I comandi inviati via BLE vengono gestiti dalla funzione `nus_data_handler` (Cod. 4.15):

- se viene ricevuta la stringa START seguita da `\n` (in esadecimale "53 54 41 52 54 0A"), viene inserito nel *FIFO* `m_spiTx_fifo` quello che sarà il comando di START per l'FPGA ("00AA" in esadecimale);
- se invece, viene ricevuta la stringa STOP seguita da `\n` (in esadecimale "53 54 4F 50 0A"), viene inserito nel *FIFO* `m_spiTx_fifo` quello che sarà il comando di STOP per l'FPGA ("00CC" in esadecimale).

```

1
2 static void nus_data_handler(ble_nus_evt_t * p_evt)
3 {
4     if (p_evt->type == BLE_NUS_EVT_RX_DATA)
5     {
6         ret_code_t err_code;
7
8         /*Salvo localmente il comando ricevuto via BLE (presente in
9         in p_data)*/
10        char dongle_command[BLE_NUS_MAX_DATA_LEN];
11        memset (dongle_command,0,BLE_NUS_MAX_DATA_LEN);
12        memcpy (dongle_command, p_evt->params.rx_data.p_data, p_evt->
13        params.rx_data.length);
14
15        /*Comando da inviare via SPI all'FPGA*/
16        uint16_t cmd; //Comando che sara' inviato via SPI all'FPGA
17        uint16_t data_watch; //Variabile per leggere se il comando e
18        ' stato scritto bene nel fifo di trasmissione dell'SPI
19
20        NRF_LOG_DEBUG("Received data from BLE NUS. Writing data on
21        RIT. ");
22        NRF_LOG_HEXDUMP_DEBUG(p_evt->params.rx_data.p_data, p_evt->
23        params.rx_data.length);
24
25        if(strcmp(dongle_command, "START\n") == 0 )
26        {
27            cmd = 0x00AA; //comando di START per l'FPGA
28            err_code = spiTx_fifo_push(&cmd); //Accodo il comando
29            nel fifo di trasmissione SPI
30            APP_ERROR_CHECK(err_code); //Vedo se ci sono errori
31            err_code = spiTx_fifo_peek(&data_watch); //Leggo (
32            senza cancellare) cosa e' stato scritto nel fifo di trasmissione
33            SPI (per conferma)
34            NRF_LOG_DEBUG("dato messo nel fifo spiTx:");
35            NRF_LOG_HEXDUMP_DEBUG(&data_watch, 2); //Lo leggerò '
36            al contrario stavolta, sara' capovolto dopo, prima di essere
37            effettivamente trasmesso via SPI

```

```

28         ble_tx_active = false;
29     }
30     else if(strcmp(dongle_command, "STOP\n") == 0 )
31     {
32         cmd = 0x00CC; //comando di STOP per l'FPGA
33         err_code = spiTx_fifo_push(&cmd); //Accodo il
comando nel fifo di trasmissione SPI
34         APP_ERROR_CHECK(err_code); //Vedo se ci sono errori
35         err_code = spiTx_fifo_peek(&data_watch); //Leggo (
senza cancellare) cosa e' stato scritto nel fifo di trasmissione
SPI (per conferma)
36         NRF_LOG_DEBUG("dato messo nel fifo spiTx:");
37         NRF_LOG_HEXDUMP_DEBUG(&data_watch, 2); //Lo leggerò
al contrario stavolta, sarà capovolto dopo, prima di essere
effettivamente trasmesso via SPI
38         ble_tx_active = false;
39     }
40     else
41     {
42         //niente
43     }
44 }
45
46 //Attivo e disattivo led 2 se attivo le notifiche da PC
47 if (p_evt->type == BLE_NUS_EVT_COMM_STARTED)
48 {
49     nrf_gpio_pin_clear(LED_2);
50 }
51
52 if (p_evt->type == BLE_NUS_EVT_COMM_STOPPED)
53 {
54     nrf_gpio_pin_set(LED_2);
55 }
56 }

```

Cod. 4.15: Gestione del dato ricevuto via BLE

Il comando viene poi inviato via SPI, non appena il Microcontrollore va in idle state e viene richiamata la funzione *spi_fifo_process* (Cod. 4.16). Nella stessa funzione vengono gestiti anche i dati ricevuti via SPI dallo Slave (l'FPGA successivamente) e precedentemente salvati in *m_spiRx_fifo*, i quali vengono trasferiti nel buffer *nusTx* e mandati via BLE richiamando la funzione *ble_data_send*.

A fine trasmissione SPI, viene richiamata *spi_event_handler*, che salva il dato, appena ricevuto via SPI, nel ringbuffer *m_spiRx_fifo*.

```

1
2 /*Pin SPI*/
3 #define SPI_SS      28    //Slave select

```

```
4 #define SPI_SCK    29    //SPI Clock
5 #define SPI_MISO  30    //Master Input Slave Output
6 #define SPI_MOSI  31    //Master Output Slave Input
7
8 /*Buffers SPI*/
9 #define SPI_BUFSIZE  2 //Dimensione dei buffer SPI (in byte)
10
11 uint8_t spi_tx_buf[SPI_BUFSIZE];
12 uint8_t spi_rx_buf[SPI_BUFSIZE];
13 uint8_t spi_tx_buf_lenght = sizeof(spi_tx_buf);
14 uint8_t spi_rx_buf_lenght = sizeof(spi_rx_buf);
15
16 static volatile bool spi_xfer_done; //Flag per trasmissione
    completata
17
18 static void spi_fifo_process()
19 {
20     uint16_t element;
21     uint16_t queue_elements;
22     ret_code_t err_code;
23
24     /*Gestione del dato ricevuto via BLE e salvato in m_spiTx_fifo*/
25
26     queue_elements = spiTx_fifo_utilization_get(); //vedo quanti
    elementi sono nel fifo di trasmissione
27
28     for (int i=0; i<queue_elements; i++) //Invio tutti gli elementi
    nel fifo via SPI all'FPGA
29     {
30         err_code = spiTx_fifo_pop(&element); //Tolgo un dato dal
    fifo di trasmissione SPI (verso FPGA) e lo salvo in element
31         APP_ERROR_CHECK(err_code);
32
33         /*Ogni elemento del fifo e' da 2 byte ma la trasmissione SPI
    e' a 1 byte quindi devo spezzare l'elemento in due*/
34         spi_tx_buf[0] = element >> 8;
35         spi_tx_buf[1] = (uint8_t)element; //Inoltre scambio i 2 byte,
    tra loro, come gia' anticipato per trasmetterli nell'ordine
    giusto.
36
37         NRF_LOG_DEBUG("spiTx_fifo element put in spi_tx_buf");
38         NRF_LOG_HEXDUMP_DEBUG(spi_tx_buf, spi_tx_buf_lenght);
39
40         spi_xfer_done = false; //Indica che parte la trasmissione (
    torna true quando alla fine della trasmissione SPI parte l'SPI
    handler)
```

```
41     err_code = nrf_drv_spi_transfer(&m_spi, spi_tx_buf,
spi_tx_buf_lenght, spi_rx_buf, spi_rx_buf_lenght); //Trasmissione
SPI (nel frattempo ricevo anche un dato dall'FPGA, che sara'
salvato in spi_rx_buf)
42     APP_ERROR_CHECK(err_code);
43
44     while(!spi_xfer_done) __WFE(); //Finche' non parte l'handler
di fine trasmissione SPI, aspetto qui
45 }
46
47 /*Gestione del dato rivevuto dalla FPGA da salvare in
m_spiRx_fifo e mandare via BLE*/
48
49 if (!ble_tx_active) //Se sto gia' trasmettendo via BLE salto
questo blocco
50 {
51     queue_elements = spiRx_fifo_utilization_get(); //Vedo quanti
elementi ho ricevuto nel buffer spiRx
52     for (int i=0; i<queue_elements; i++)
53     {
54         err_code = spiRx_fifo_pop(&element); //Tolgo un dato dal
fifo di ricezione SPI (da FPGA) e lo salvo in element
55         nusTx[i*2] = element >> 8;
56         nusTx[i*2+1] = (uint8_t)element; //Ogni due elementi
consecutivi di nusTx sono un elemento di element (ovvero un dato
da 2 byte ricevuto via SPI dall'FPGA)
57     }
58
59     if (queue_elements > 0) //Se effettivamente sono stati
ricevuti elementi dall'FPGA
60     {
61         NRF_LOG_DEBUG("Ready to send data over BLE NUS");
62
63         ble_tx_active = true; //Una volta Ricevuti i dati, setto
ble_tx_active su true che vuol dire che sto mandando dati via BLE
(durante la trasmissione BLE ignoro i dati ricevuti dall'FPGA)
64         txSize = 2*queue_elements; //La dimensione del buffer
che viene inviato via ble (nusTx) e' pari al doppio di quella
spiRx (come detto prima)
65         txRdPtr = 0; //Questo dice da dove partire a mandare i
dati in ble_data_send (ovviamente parte da zero essendo l'inizio
della trasmissione BLE)
66         ble_data_send(); //Si occupa della trasmissione BLE del
dato ricevuto via SPI, gestendolo come una notifica che viene
mandata via BLE (al controller o dongle) sfruttando il NUS
67     }
68 }
69 }
70
```

```

71 /*Event handler dell'spi (eseguito a fine trasmissione SPI)*/
72 void spi_event_handler(nrf_drv_spi_evt_t const * p_event, void *
    p_context)
73 {
74     uint16_t element;
75     ret_code_t err_code;
76     spi_xfer_done = true; //trasmissione spi completata
77     NRF_LOG_INFO("SPI transfer completed.");
78
79     /*Stampo il dato ricevuto via SPI dall'FPGA*/
80     NRF_LOG_INFO(" Received:");
81     NRF_LOG_HEXDUMP_INFO(spi_rx_buf, spi_rx_buf_lenght);
82
83     /*Scrivo il dato ricevuto via SPI sul FIFO di ricezione (
    m_spiRx_fifo)*/
84     element = ((uint16_t)spi_rx_buf[0]<<8) + spi_rx_buf[1];
85     err_code = spiRx_fifo_push(&element);
86     APP_ERROR_CHECK(err_code);
87 }

```

Cod. 4.16: Gestione dei dati da inviare e ricevuti via SPI

In *ble_data_send* (Cod. 4.17) viene inviato via BLE il contenuto del buffer *nusTX*. I pacchetti da inviare hanno dimensione massima *REWIRE_MTU* e nel caso nel buffer ci sia un numero di byte maggiore, questo verrà spezzato in più pacchetti della dimensione massima (tranne l'ultimo che conterrà gli elementi restanti). Viene poi sfruttato il servizio NUS, della libreria Nordic, per inviare il dato come Notifica, essendo l'operazione di trasmissione più veloce (Tab. 3.1).

Nel caso in cui si verifichi l'errore *NRF_ERROR_RESOURCES*, perché i buffer del Softdevice che si occupa del BLE sono pieni, allora si sfrutta il flag *bleTxBusy*. Se questo ha valore *true*, alla fine della trasmissione del pacchetto, quando viene richiamata la funzione *tx_complete*, viene rieseguita *ble_data_send* in modo da riprendere l'invio dei dati da dove si era interrotto.

```

1
2 /*Dimensione massima dei pacchetti*/
3 #define REWIRE_MTU 244
4
5 void ble_data_send()
6 {
7
8     uint16_t length;
9     uint32_t err_code = NRF_SUCCESS;
10
11     /*Invia dati finche' ce ne sono e finche' err_code dice che la
    trasmissione precedente e' riuscita*/
12     while (err_code == NRF_SUCCESS && txRdPtr < txSize)

```

```

13     {
14         /*Se il numero di byte da mandare meno il numero di byte
mandati e' maggiore della dimensione massima del pacchetto allora
length assume il valore massimo di pacchetto (REWIRE_MTU)
altrimenti assume il valore txSize - txRdPtr*/
15         length = ((txSize - txRdPtr) > REWIRE_MTU) ? REWIRE_MTU : (
txSize - txRdPtr);
16
17         /*Qui mandiamo, via BLE, una stringa di byte lunga "length"
partendo da nusTx[txRdPtr] (dove txRdPtr all'inizio parte da 0
ovviamenteemente)*/
18         err_code = ble_nus_data_send(&m_nus, &nusTx[txRdPtr], &length
, m_conn_handle);
19
20         if ((err_code != NRF_ERROR_INVALID_STATE) &&
21             (err_code != NRF_ERROR_RESOURCES) &&
22             (err_code != NRF_ERROR_NOT_FOUND))
23         {
24             APP_ERROR_CHECK(err_code);
25         }
26         if (err_code == NRF_SUCCESS)
27         {
28             /*Se la trasmissione riesce allora aggiorno txRdPtr al
valore di length (all'iterazione successiva partiro' dal byte
successivo all'ultimo byte mandato)*/
29             txRdPtr += length;
30         }
31         if (err_code == NRF_ERROR_RESOURCES)
32         {
33             /*Se si verifica NRF_ERROR_RESOURCES vuol dire che i
buffer nel softdevice sono pieni e non posso mandare altri dati (
Allora metto bleTxBusy = true) finche non si realizza l'evento
BLE_GATTS_EVT_HVN_TX_COMPLETE. Comparso questo errore vuol dire
che usciro' prima dal ciclo while ma non perdo il pacchetto
perche' quando si realizza l'evento BLE_GATTS_EVT_HVN_TX_COMPLETE
parte la funzione resume_tx che rimette bleTxBusy=false e richiama
questa funzione (ble_data_send) che riprende a mandare dati dal
pacchetto che non si era riuscito ad inviare dando appunto questo
errore (NRF_ERROR_RESOURCES).*/
34             bleTxBusy = true;
35             NRF_LOG_DEBUG("Buffer del Softdevice pieni. bleTxBusy =
true");
36         }
37     }
38     if (txRdPtr >= txSize)
39     {
40         /*Una volta finito di trasmettere i dati via ble rimetto
ble_tx_active su false*/
41         ble_tx_active = false;

```

```

42     NRF_LOG_DEBUG("Dati interamente trasmessi da MCU a Dongle.
Pronto per inviare un nuovo pacchetto.");
43     }
44 }
45
46 /*Funzione che parte quando avviene l'evento
BLE_GATTS_EVT_HVN_TX_COMPLETE a trasmissione completata*/
47 void tx_complete()
48 {
49     if (bleTxBusy)
50     {
51         /*Rimetto bleTxBusy= false e rimando la trasmissione BLE */
52         bleTxBusy= false;
53
54         NRF_LOG_DEBUG("La trasmissione e' stata completata e riprendo
da dove ho lasciato. bleTxBusy = False");
55
56         /*Riprendo la trasmissione da dove avevo lasciato*/
57         ble_data_send();
58     }
59 }

```

Cod. 4.17: Invio dei dati via BLE

Infine viene gestito l'impulso di IRQ che lo Slave invierà per richiedere la trasmissione (Cod. 4.18) Per farlo è stato impostato un interrupt sulle transizioni low-to-high rilevate sul pin *IRQ_PIN*. Ad ogni interrupt verrà richiamata la funzione *irq_pin_handler* la quale mette in coda al ringbuffer *m_spiTx_fifo* una frame spuria (per esempio "0000" in esadecimale). Questa verrà inviata non appena vengono terminate queste operazioni e richiamata *spi_fifo_process* prima di andare in idle state.

```

1
2 #define IRQ_PIN 26
3
4 /* IRQ_PIN ISR*/
5 void irq_pin_handler(nrf_drv_gpiote_pin_t pin, nrf_gpiote_polarity_t
action)
6 {
7     NRF_LOG_DEBUG("Interrupt partito. Mando frame dummy.");
8     ret_code_t err_code;
9     uint16_t data = 0x0000; //dato dummy
10    err_code = spiTx_fifo_push(&data);
11 }
12
13 /* inizializzazione IRQ_PIN */
14 void gpio_init()
15 {

```

```

16     ret_code_t err_code;
17
18     nrf_drv_gpiote_in_config_t spiIrq = GPIOTE_CONFIG_IN_SENSE_LOTOHI
19     (true);
20     spiIrq.pull = NRF_GPIO_PIN_PULLDOWN; //in idle resta a zero, se
21     va alto scatta l'interrupt
22
23     err_code = nrf_drv_gpiote_in_init(IRQ_PIN, &spiIrq,
24     irq_pin_handler);
25     APP_ERROR_CHECK(err_code);
26
27     nrf_drv_gpiote_in_event_enable(IRQ_PIN, true); //abilita l'
28     interrupt
29 }

```

Cod. 4.18: ISR sul pin di IRQ

03_SPISlave

Per testare la comunicazione SPI e BLE nel codice *03_MCU_FPGAcounter* implementata è stata utilizzata un'interfaccia SPI Slave per un altro Microcontrollore (*03_SPISlave*).

L'interfaccia è pensata per ricevere i dati via SPI dal master e se il dato ricevuto è il comando di START, il dispositivo passa in uno stato (definito dalla variabile *fsm*) in cui ad ogni fine trasmissione verrà richiesto l'invio di una nuova frame con un impulso sul pin di IRQ (Cod. 4.19).

```

1
2 /*Richiamata a fine trasmissione SPI*/
3 void spis_event_handler(nrf_drv_spis_event_t event)
4 {
5     if (event.evt_type == NRF_DRV_SPIS_XFER_DONE)
6     {
7         spis_xfer_done = true;
8
9         /*Comando di START*/
10        if (m_rx_buf[0] == 0x00 && m_rx_buf[1] == 0xAA)
11        { fsm = true; }
12        /*Comando di STOP*/
13        else if (m_rx_buf[0] == 0x00 && m_rx_buf[1] == 0xCC)
14        { fsm = false; }
15    }
16 }
17
18 int main(void)
19 {

```

```

20  ... ..
21
22  while (1)
23  {
24      spis_xfer_done = false;
25      APP_ERROR_CHECK(nrf_drv_spis_buffers_set(&spis, m_tx_buf,
m_length, m_rx_buf, m_length));
26      if (fsm)
27      {
28          NRF_LOG_DEBUG("Siamo entrati in fsm=true");
29          nrf_gpio_pin_set(IRQ_PIN);
30          nrf_delay_ms(500);
31          NRF_LOG_DEBUG("Pin IRQ abbassato");
32          nrf_delay_ms(500);
33      }
34      else
35      {
36          NRF_LOG_DEBUG("Siamo entrati in fsm=false");
37          APP_ERROR_CHECK(nrf_drv_spis_buffers_set(&spis, m_tx_buf,
m_length, m_rx_buf, m_length));
38      }
39      while (!spis_xfer_done)
40      { __WFE(); }
41  }
42 }

```

Cod. 4.19: Interfaccia SPI Slave con generazione di impulsi su IRQ

Test della comunicazione SPI e BLE

In questo test si verifica il funzionamento della trasmissione BLE per l'invio di comandi al Microcontrollore e la ricezione di dati dallo stesso.

Per eseguire il test verranno utilizzati due nRF52 DK (Fig. 3.6), uno per il Master (*03_MCU_FPGAcounter*) e uno per lo Slave (*03_SPISlave*). Fanno parte del set-up anche un Dongle nRF52840 (Fig. 3.11) e i software Segger Embedded Studio (Fig. 3.12) e nRF Connect for Desktop (Fig. 3.13).

Innanzitutto bisogna collegare i pin delle due board come in Tab. 4.3 e collegare queste ultime al PC tramite i relativi cavi USB. Collegare infine anche il Dongle per ottenere il set-up in Fig. 4.9.

I due codici sono stati caricati sulle due board in modalità Debug, tramite SES (Fig. 4.10), in modo da poter leggere i log nel Debug Terminal ed individuare eventuali bug.

nRF52 DK Master	Nome pin	nRF52 DK Slave
GND	Ground	GND
28	CS	28
29	SCK	29
30	MISO	30
31	MOSI	31

Tabella 4.3: Collegamento dei pin del Master e dello Slave.

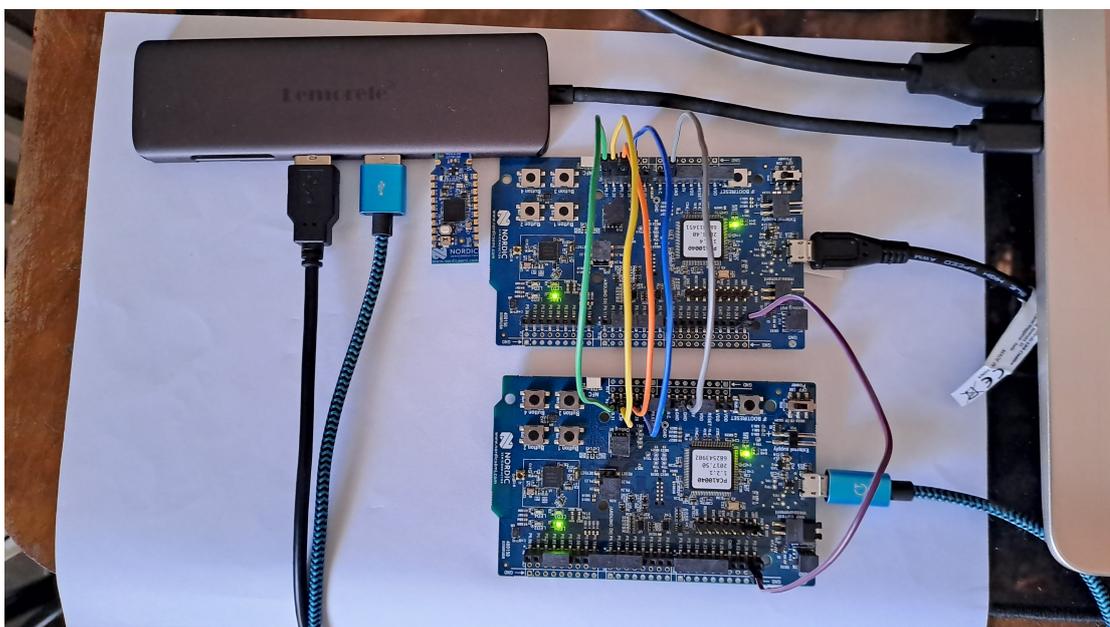


Figura 4.9: Set-up per il test di trasmissione SPI e BLE

E' stata poi utilizzato nRF Connect for Desktop con il Dongle per inviare i comandi al Master via BLE.

Se ad esempio viene inviata la stringa STOP da PC (scritta in esadecimale nella caratteristica UART RX, come si vede in Fig. 4.11), questa verrà ricevuta dal Master che risponderà inserendo nel FIFO la frame, da 16 bit, di STOP ("00CC" scritta in esadecimale) ed inviandola via SPI allo Slave.

Lo Slave poi, risponde alla trasmissione inviando la frame OK ("4F4B" in esadecimale).

Grazie ai log sul Debug Terminal del Master e dello Slave, viene confermata la corretta esecuzione di queste istruzioni (Fig. 4.12 - 4.13).

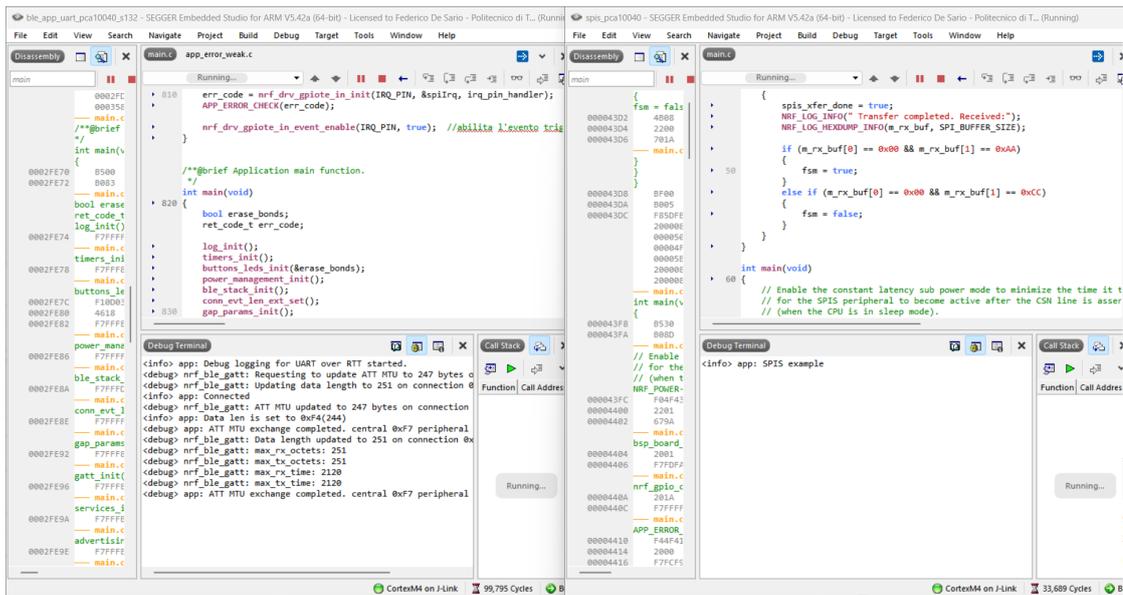


Figura 4.10: Schermate di SES di Master e Slave nel test di comunicazione SPI e BLE

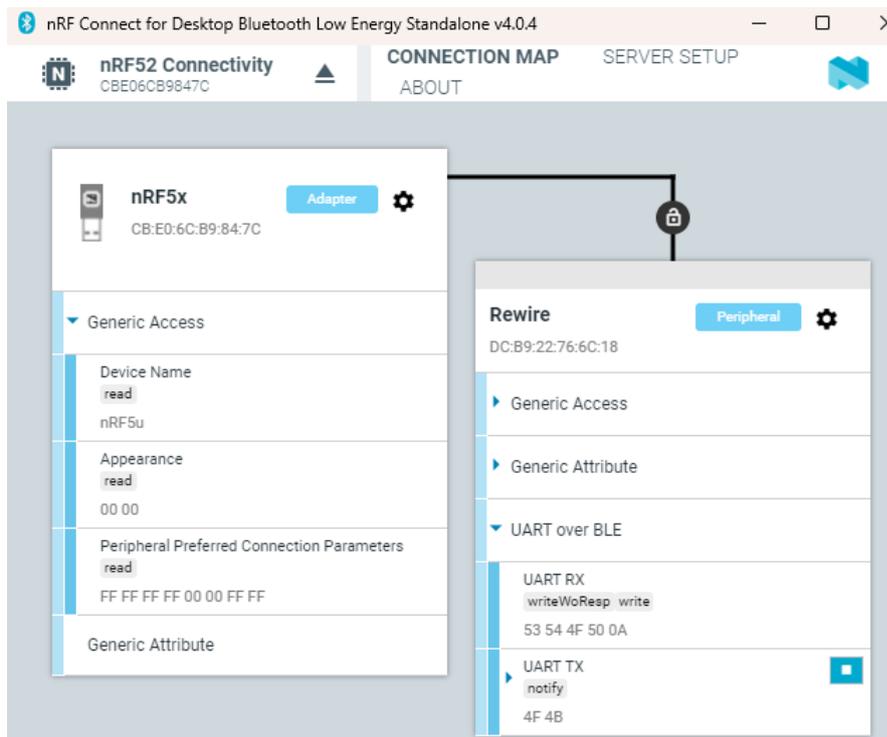


Figura 4.11: Schermata di nRF Connect for Desktop nel test di comunicazione SPI e BLE

```

<info> app: Data len is set to 0xF4(244)
<debug> app: ATT MTU exchange completed. central 0xF7 peripher
<debug> nrf_ble_gatt: Data length updated to 251 on connection
<debug> nrf_ble_gatt: max_rx_octets: 251
<debug> nrf_ble_gatt: max_tx_octets: 251
<debug> nrf_ble_gatt: max_rx_time: 2120
<debug> nrf_ble_gatt: max_tx_time: 2120
<debug> app: ATT MTU exchange completed. central 0xF7 peripher
<debug> app: Received data from BLE NUS. Writing data on RTT.
<debug> app: 53 54 4F 50 0A |STOP.
<debug> app: dato messo nel fifo spiTx:
<debug> app: CC 00 |..
<debug> app: spiTx_fifo element put in spi_tx_buf
<debug> app: 00 CC |..
<info> app: SPI transfer completed.
<info> app: Received:
<info> app: 4F 4B |OK
<debug> app: Ready to send data over BLE NUS
<debug> app: Dati interamente trasmessi da MCU a Dongle.

```

Figura 4.12: Debug Terminal del Master dopo l'invio del comando di STOP via BLE e il successivo invio della frame di STOP "00CC" via SPI

```

<info> app: SPIS example
<info> app: Transfer completed. Received:
<info> app: 00 CC |..

```

Figura 4.13: Debug Terminal dello Slave dopo l'invio del comando di STOP via BLE e la ricezione della frame di STOP "00CC" via SPI

Inviando invece la stringa START via BLE (scritta in esadecimale nella caratteristica UART RX), questa verrà ricevuta dal Master che risponderà inserendo nel FIFO la frame, da 16 bit, di START ("00AA" scritta in esadecimale) ed inviandola via SPI allo Slave. Lo Slave quindi cambia stato e risponde alle frame ricevute inviando la frame OK ("4F4B" in esadecimale) e un impulso sul pin di IRQ. Il Master a sua volta riceve l'impulso e risponde inviando una nuova frame, procedendo in questo loop di sollecitazione e risposta con lo Slave. Il tutto è confermato dai log dei due Debug Terminal (Fig. 4.14 - 4.15).

```

, <debug> app: Received data from BLE NUS. Writing data on RTT.
<debug> app: 53 54 41 52 54 0A |START.
<debug> app: dato messo nel fifo spiTx:
<debug> app: AA 00 |..
<debug> app: spiTx_fifo element put in spi_tx_buf
<debug> app: 00 AA |..
<info> app: SPI transfer completed.
<info> app: Received:
<info> app: 4F 4B |OK
<debug> app: Ready to send data over BLE NUS
<debug> app: Dati interamente trasmessi da MCU a Dongle.
<debug> app: Interrupt partito. Mando frame dummy.
<debug> app: spiTx_fifo element put in spi_tx_buf
<debug> app: 00 00 |..
<info> app: SPI transfer completed.
<info> app: Received:
<info> app: 4F 4B |OK
<debug> app: Ready to send data over BLE NUS
<debug> app: Dati interamente trasmessi da MCU a Dongle.

```

Figura 4.14: Debug Terminal del Master dopo l'invio del comando di START via BLE e il successivo invio della frame di START "00AA" via SPI

```

<info> app: SPIS example
<info> app: Transfer completed. Received:
<info> app: 00 CC |..
<info> app: Transfer completed. Received:
<info> app: 00 AA |..
<debug> app: Siamo entrati in fsm=true
<debug> app: Pin IRQ alzato
<info> app: Transfer completed. Received:
<info> app: 00 00 |..
<debug> app: Pin IRQ abbassato
<debug> app: Siamo entrati in fsm=true
<debug> app: Pin IRQ alzato
<info> app: Transfer completed. Received:
<info> app: 00 00 |..
<debug> app: Pin IRQ abbassato
<debug> app: Siamo entrati in fsm=true
<debug> app: Pin IRQ alzato
<info> app: Transfer completed. Received:
<info> app: 00 00 |..
<debug> app: Pin IRQ abbassato

```

Figura 4.15: Debug Terminal dello Slave dopo l'invio del comando di START via BLE e la ricezione della frame di START "00AA" via SPI

Il codice (*03_MCU_FPGAcounter*) verrà poi ulteriormente testato con il sistema completo nella sezione 4.2.3 di questo capitolo.

4.2 Sviluppo del codice per l'FPGA

Come esposto nel capitolo 3, l'FPGA dovrà fare da ponte di comunicazione tra il Microcontrollore e l'ASIC, permettendo una comunicazione bidirezionale tra i due chip. Entrambi i dispositivi comunicano su SPI ma il Microcontrollore ha un SPI clock massimo di 8 MHz mentre l'ASIC pilota l'SPI clock a 16 MHz. Motivo per il quale l'FPGA dovrà incrociare i due domini di clock.

Si è optato per una strategia sincrona, sfruttando l'oscillatore interno della iCE40 Ultra, che genera un clock a 48 MHz, e il PLL, sempre interno, che triplica questa frequenza, generando un clock da 144 MHz. Il sistema completo pensato per l'FPGA in questo lavoro di tesi è rappresentato in Fig. 4.16

Comprende due moduli SPI SLAVE muniti del pin di IRQ, che permetteranno di comunicare via SPI con il Microcontrollore e con l'ASIC. Tra i due si è pensato di interporre un FIFO bidirezionale, che permetta di bufferizzare i dati in ingresso da ambo i lati, per poi inviarli su richiesta all'ASIC o al Microcontrollore.

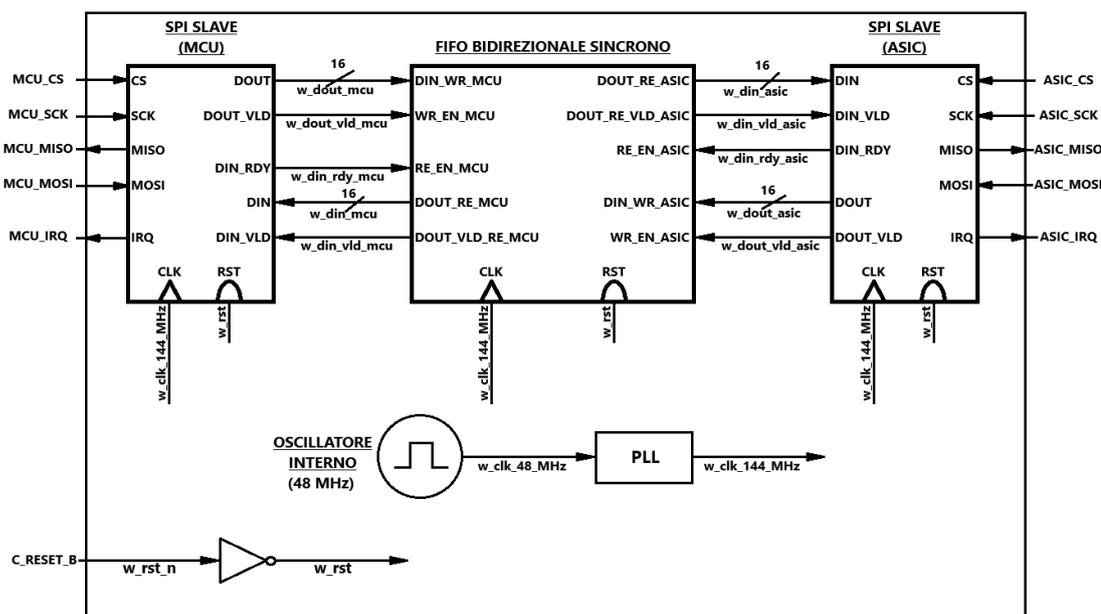


Figura 4.16: Schema a blocchi del sistema completo dell'FPGA

Durante questo lavoro di tesi sono state sviluppate e simulate le interfacce SPI SLAVE a 8 MHz e 16 MHz. Dopodiché sono state testate su hardware, stimolando

il kit di sviluppo dell'FPGA iCE40 Ultra con un SPIDriver (Fig. 3.19) ed infine facendole interfacciare con il Microcontrollore.

Il FIFO bidirezionale invece non è stato ancora implementato e si rimanda questa parte del lavoro agli sviluppi futuri di questo progetto. E' stato però utilizzato un contatore per simulare, internamente all'FPGA, un flusso di dati proveniente dall'ASIC.

Successivamente sono stati fatti dei test qualitativi di velocità, per verificare se i parametri ottimi individuati durante lo studio nella Sez. 4.1.1 garantiscono ancora un data rate prossimo a 1.4 Mbps.

4.2.1 Interfaccia SPI (Slave)

Nella prima fase dello sviluppo del codice per l'FPGA è stato implementato il modulo SPI_SLAVE_IRQ. Questo è in grado, oltre che di essere pilotato da un Master, di generare un impulso di IRQ ogni volta che riceve dei dati validi, lato user, da trasmettere serialmente sul MISO.

SPI_SLAVE_IRQ

Il blocco (Fig 4.17) presenta 12 pin:

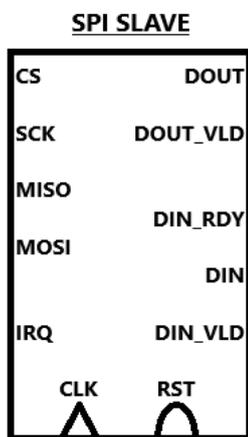


Figura 4.17:
Modulo
SPI_SLAVE_IRQ

- ci sono i 4 pin per la comunicazione SPI, ovvero il Chip Select (CS), l'SPI Clock (SCK), il Master Input Slave Output (MISO) e il Master Output Slave input (MOSI), a cui si aggiunge il pin di IRQ per richiedere la comunicazione al Master;
- a questi, si aggiungono le uscite DOUT (un bus da 16 bit contenente il dato ricevuto serialmente sul MOSI) e DOUT_VLD, il cui valore, se alto, segnala quando il dato su DOUT è valido;
- il dato in ingresso da trasmettere serialmente sul MISO, anche questo da 16 bit, viene ricevuto sul pin di ingresso DIN ed è valido quando l'input DIN_VLD assume il valore alto;
- il pin di uscita DIN_RDY avverte che il modulo è pronto a ricevere un nuovo dato;
- infine i pin di ingresso CLK e RST, rappresentano rispettivamente il Clock e il Reset del modulo.

Il numero di bit trasmessi è facilmente programmabile grazie al Generic con cui si apre il codice VHDL (Cod. 4.20) e alla parametrizzazione della dimensione degli input e degli output. In questa maniera sarà possibile, in eventuali studi futuri, modificare il numero di bit da trasmettere in base alle necessità del momento.

```

1
2 entity SPI_SLAVE_IRQ is
3   Generic (
4     WORD_SIZE : natural := 16); —dimensione della frame
   trasmessa
5
6   Port (
7     CLK      : in  std_logic;    — Clock
8     RST      : in  std_logic;    — Reset asincrono (active high)
9
10    SCLK     : in  std_logic;    — SPI clock
11    CS_N     : in  std_logic;    — Chip Select (active low)
12    MOSI     : in  std_logic;    — Master Output Slave Input
13    MISO     : out std_logic;    — Master Input Slave Output
14
15    DIN      : in  std_logic_vector(WORD_SIZE$-1 downto 0); —
   Dato ricevuto lato user, da trasmettere via SPI al master
16    DIN_VLD  : in  std_logic;    — Quanto e' alto, allora il dato
   presente su DIN, da trasnettere al master, e' valido
17    DIN_RDY  : out std_logic;    — Quando e' alto, il modulo SPI
   e' pronto a ricevere un altro dato da lato user
18    DOUT     : out std_logic_vector(WORD_SIZE$-1 downto 0); —
   Dato ricevuto via SPI da lato master
19    DOUT_VLD : out std_logic;    — Quando e' alto, allora il dato
   ricevuto dal master e presente su DOUT e' valido
20
21    IRQ      : out std_logic);    — Pin che verra' usato dallo
   slave per richiedere la comunicazione al master
22 end entity;
```

Cod. 4.20: Entity del modulo *SPI_SLAVE_IRQ*

Per permettere la sincronizzazione dei segnali in ingresso dall'esterno (CS, SCK e MOSI), sono stati utilizzati 2 flip-flop (Cod. 4.21). In questa maniera si prevengono problemi di metastabilità ma si introduce del ritardo. Per non compromettere il funzionamento del modulo a causa del ritardo introdotto, il clock da 48 MHz dell'oscillatore interno, viene moltiplicato sfruttando il PLL e portato a 144 MHz. Come conseguenza il ritardo di 2 colpi di clock, introdotto dai flip-flop, diventa trascurabile anche quando il modulo viene pilotato con un SPI Clock a 16 MHz (come riscontrabile in simulazione).

```

1
2 signal sclk_ff1           : std_logic;
3 signal cs_n_ff1          : std_logic;
4 signal mosi_ff1          : std_logic;
5 signal sclk_ff2          : std_logic;
6 signal cs_n_ff2          : std_logic;
7 signal mosi_ff2          : std_logic;
8
9 ... ..
10
11 — Flip-Flops per ridurre al minimo la metastabilita'.
12 ff_meta_p : process (CLK)
13 begin
14     if (rising_edge(CLK)) then
15         sclk_ff1 <= SCLK;
16         cs_n_ff1 <= CS_N;
17         mosi_ff1 <= MOSI;
18         sclk_ff2 <= sclk_ff1;
19         cs_n_ff2 <= cs_n_ff1;
20         mosi_ff2 <= mosi_ff1;
21     end if;
22 end process;

```

Cod. 4.21: Flip-Flops di sincronizzazione

Il modulo è pensato per funzionare in SPI MODE 0, quindi l'SPI Clock in stato idle è basso, dopodiché:

- il modulo campiona il dato sul MOSI e lo inserisce nello shift register su ogni fronte di salita dell'SPI Clock;
- su ogni fronte di discesa invece, setta il MISO al valore dell'MSB dello shift register (Cod. 4.22).

```

1
2 — Lo shift register ha in ingresso il dato da mandare al master, e
   man mano che manda i bit al master shifta a sinistra e sostituisce
   l'LSB con il bit ricevuto sul MOSI (alla fine restera' solo il
   dato inviato dal master)
3 data_shreg_p : process (CLK)
4 begin
5     if (rising_edge(CLK)) then
6         if (load_data_en = '1') then
7             data_shreg <= DIN;
8         elsif (sclk_reedge_en = '1' and cs_n_ff2 = '0') then
9             data_shreg <= data_shreg(WORD_SIZE-2 downto 0) & mosi_ff2
10            ;
11         end if;

```

```

11     end if;
12 end process;
13
14 — Scrivo il dato sul miso sui falling edge dell'SPI clock, mentre il
    CS abbassato
15 miso_p : process (CLK)
16 begin
17     if (rising_edge(CLK)) then
18         if (load_data_en = '1') then
19             MISO <= DIN(WORD_SIZE-1);
20         elsif (sclk_fedge_en = '1' and cs_n_ff2 = '0') then
21             MISO <= data_shreg(WORD_SIZE-1);
22         end if;
23     end if;
24 end process;

```

Cod. 4.22: Utilizzo dello shift register

Infine, viene generato un impulso di IRQ qualora ci sia un dato valido in ingresso dal lato user (DIN) e se lo Slave è pronto a ricevere un nuovo dato (Cod. 4.23).

```

1
2 — Se il dato in ingresso (lato user) e' valido e posso accettare un
    nuovo dato (slave_ready alto) allora posso scrivere il dato in
    ingresso nello shift register (alzando questo flag)
3 load_data_en <= slave_ready and DIN_VLD;
4
5 ... ..
6
7 — Lo slave richiede la comunicazione al master se c'e' un dato
    valido sul bus DIN
8
9 irq_p : process (CLK)
10 begin
11     if (rising_edge(CLK)) then
12         if (RST='1') then
13             IRQ <= '0';
14         else
15             IRQ <= load_data_en;
16         end if;
17     end if;
18 end process;

```

Cod. 4.23: Generazione dell'impulso di IRQ

Simulazione del modulo SPI_SLAVE_IRQ

Il modulo così implementato è stato simulato con Modelsim con frequenza di SPI Clock di 8 MHz e 16 MHz, e un clock di sistema tipico a 144 MHz, minimo a 129 MHz e massimo a 159 MHz, in quanto l'incertezza sulla frequenza dell'oscillatore interno è del $\pm 10\%$ (valore indicato sul datasheet dell'FPGA [28]). Per impostare queste frequenze, basterà assegnare i valori desiderati alle corrispondenti costanti nel testbench (Cod. 4.24).

```

1
2 constant CLK_FREQ : natural := 159e6;    — Frequenza del clock di
      sistema
3 constant SPI_FREQ : natural := 8e6;      — Frequenza dell'SPI Clock

```

Cod. 4.24: Frequenza del clock di sistema e dell'SPI Clock nel testbench

In Fig. 4.18-4.19 è riportata, per motivi rappresentativi, una delle 3 trasmissioni simulate da testbench, nei due casi peggiori.

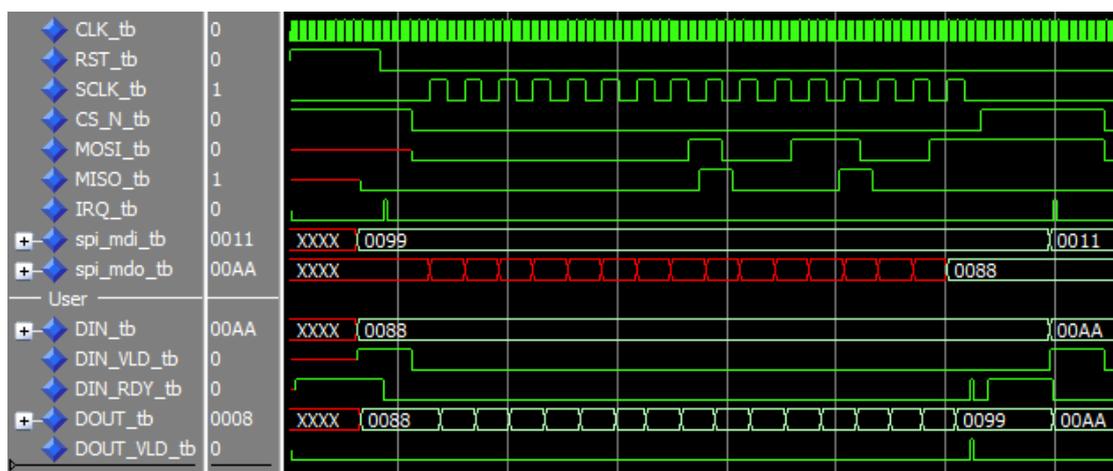


Figura 4.18: Simulazione del modulo con clock di sistema a 129 MHz e SCK a 16 MHz

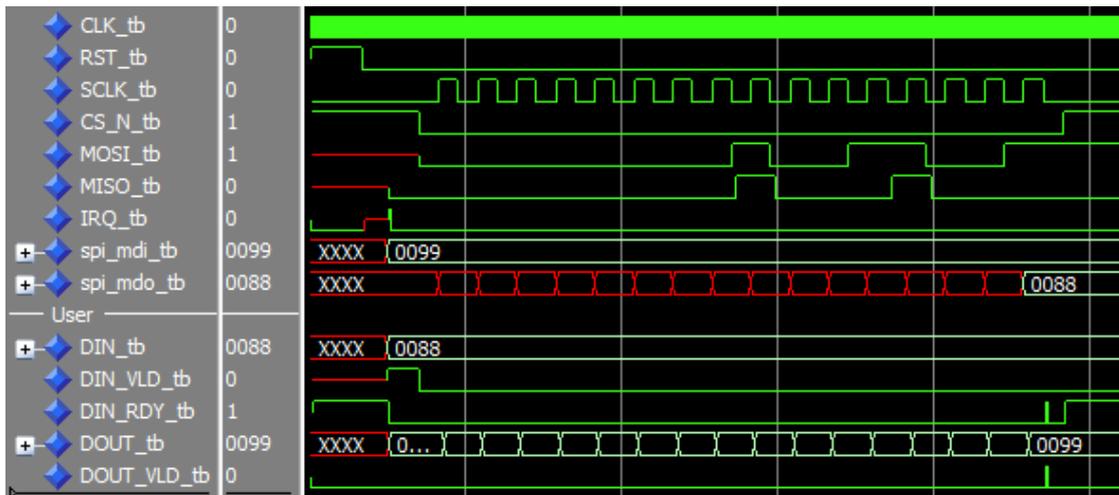


Figura 4.19: Simulazione del modulo con clock di sistema a 159 MHz e SCK a 8 MHz

Come si può notare, inserito il dato da inviare al Master sul bus *DIN* e reso il dato valido con un impulso sul *DIN_VLD*, questo viene correttamente trasferito serialmente sul MISO quando il Master pilota il CS, l'SPI Clock e il MOSI. Per verificarlo, basta vedere i bit campionati sui fronti di salita dell' SPI Clock (SCLK), o semplicemente leggere il vettore *spi_mdo_tb* composto da questi bit e accertarsi che il vettore ottenuto sia uguale a *DIN_tb*.

Allo stesso modo, si può verificare che il dato inviato dal Master (*spi_mdi_tb*) viene ricevuto correttamente dal modulo SPI Slave e lo si legge uguale sul *DOUT*. Infine un impulso di *DOUT_VLD* rende il dato valido appena individuato un fronte di discesa di *sclk_ff2*, ovvero l'SPI clock dopo i 2 flip-flop (Cod. 4.21).

4.2.2 Dati generati internamente all'FPGA e inviati al Microcontrollore tramite SPI

Nell'ultima fase dello sviluppo del codice per l'FPGA, si è implementato un counter che in abbinamento al modulo SPI_SLAVE_IRQ permette di simulare la lettura di dati provenienti dall'ASIC. Tramite il sistema composto dai due elementi sarà possibile verificare che il modulo SPI presentato precedentemente trasmette correttamente i dati verso lo user (in questo caso il counter) e verso il Master. Il sistema presentato in questa sezione permetterà inoltre di effettuare dei test su hardware utilizzando la breakout board dell'iCE40 Ultra , l'SPIDriver e l'nRF52 DK (Fig. 3.8, 3.19, 3.6).

COUNTER_SYNCHRO

Il contatore implementato (Fig. 4.20) è un modulo sincrono, con 7 pin:

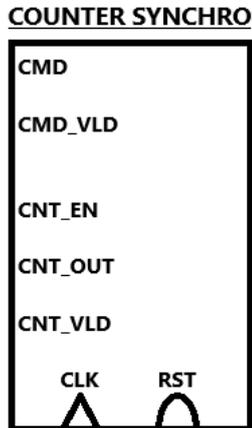


Figura 4.20:
Modulo COUNTER
SYNCHRO

- *CMD* è il bus d'ingresso da 16 bit, sul quale riceve i comandi di ON e OFF;
- *CMD1_VLD* è il pin che se va alto, rende il comando su *CMD* valido;
- *CNT_EN* è l'enable che fa incrementare il contatore di uno (ammesso che questo sia in stato ON);
- *CNT_OUT* è il bus di uscita da 16 bit dove viene inviato il risultato del contatore;
- risultato considerato valido quando il pin *CNT_VLD* va alto;
- il pin *RST* è il reset del contatore;
- infine *CLK* è il clock del sistema.

Come per il modulo SPI, anche qui il numero di bit trasmessi è modificabile attraverso un Generic nell'Entity del contatore (Cod. 4.25).

```

1
2 entity COUNTER_SYNCHRO is
3   Generic (
4     WORD_SIZE : natural := 16); — dimensione della stringa trasmessa
5     e ricevuta
6   Port (
7     CMD      : in  std_logic_vector(WORD_SIZE-1 downto 0); — decide
8     CMD_VLD  : in  std_logic;    — valida il comando ricevuto, se
9     alto
10    CNT_EN   : in  std_logic;    — enable che fa contare di uno il
11    contatore
12    CNT_OUT  : out std_logic_vector(WORD_SIZE-1 downto 0); —
13    Risultato del contatore
14    CNT_VLD  : out std_logic;    — Valida l'uscita del contatore

```

```

15     CLK      : in  std_logic;    —clock
16     RST      : in  std_logic;    — reset (active-high)
17 end entity COUNTER_SYNCHRO;

```

Cod. 4.25: Entity del modulo *SPI_SLAVE_IRQ*

Il contatore è implementato in modo da lavorare in due stati: ON e OFF. Lo stato in cui si trova è definito dal signal *state* (Cod. 4.26). Inizialmente il contatore si trova in stato OFF dopodiché:

- ricevuta la frame di accensione (*00AA* in esadecimale), il contatore passa in stato ON e incrementa il valore di CNT su ogni fronte di salita di CNT_EN;
- ricevuta invece la frame di spegnimento (*00CC* in esadecimale, il contatore passa in stato OFF e ignora qualsiasi comando che non sia quello di accensione.

```

1
2 — purpose: Leggendo il comando ricevuto in ingresso viene deciso lo
3           stato del contatore (ON=1/OFF=0)
3 state_p: process (CLK, RST) is
4 begin — process CMD_FSM
5 if RST = '1' then
6     state <= '0';           — Stato OFF
7 elsif rising_edge(CLK) then
8     if (cmd_vld_r_edge = '1' and cmd_reg = x"00AA") then           — 00AA
9         è la frame che accende il counter
10        state <= '1';       — Stato ON
11    elsif (cmd_vld_r_edge = '1' and cmd_reg = x"00CC") then       — 00CC
12        è la frame che spegne il counter
13        state <= '0';       — Stato OFF
14    else
15        state <= state;     — Per qualsiasi altra combinazione lo
16        stato non cambia
17    end if;
18 end if;
19 end process state_p;
20
21 ... ..
22 — purpose: se CNT_EN va alto mentre è nello stato ON allora il
23           contatore conta 1
24 counter_p: process (CLK, RST) is
25 begin — process counter_p
26 if RST = '1' then
27     cnt <= (others => '0');
28 elsif rising_edge(CLK) then
29     case state is
30     when '0' =>
31         cnt_vld_flag <= '0';

```

```
29  when '1' =>
30      if cnt_en_r_edge = '1' then
31          cnt <= cnt + 1;
32          cnt_vld_flag <= '1';
33      else
34          cnt <= cnt;
35          cnt_vld_flag <= '0';
36      end if;
37  when others => null;
38 end case;
39 end if;
40 end process counter_p;
```

Cod. 4.26: Processi che gestiscono lo stato del contatore e l'incremento

03_SPI_SLAVE_IRQ_COUNTER

I due moduli presentati precedentemente, sono stati poi messi insieme in quello che è l'ultimo codice per FPGA sviluppato in questa tesi (Fig. 4.21): un sistema formato da un'interfaccia SPI Slave che riceve frame da 16 bit dall'esterno e le trasmette parallelamente ad un contatore, il quale incrementa o meno il risultato a seconda della frame ricevuta e invia questo risultato al modulo SPI che lo trasmette all'esterno, serialmente, attraverso il MISO.

Per permettere il corretto funzionamento del sistema, viene utilizzato il clock da 48 MHz dell'oscillatore interno, moltiplicato sfruttando il PLL e portato a 144 MHz (come in Fig. 4.16).

Il sistema funzionerà nel seguente modo:

- se l'SPI Slave riceve il comando di START ("00AA" in esadecimale), questo viene trasmesso al bus *CMD* del contatore insieme ad un impulso sul *CMD_VLD* e subito dopo sul *DIN_RDY*;
- ricevuto il comando di START (o accensione), e il seguente impulso sul *CMD_VLD*, il contatore passa in stato ON, incrementa di uno il valore su *CNT_OUT* e manda un impulso su *CNT_VLD* per rendere il dato valido;
- quando il modulo SPI Slave riceve sul bus *DIN* il risultato del contatore e l'impulso su *DIN_VLD*, genera in uscita un impulso sul pin di IRQ che servirà a richiedere la comunicazione al Master (Sez. 4.2.3 per i dettagli);
- finché è nello stato ON, ad ogni frame ricevuta (a prescindere dal contenuto, purché non sia la frame di STOP) viene incrementato il contatore e generato un impulso sul pin di IRQ;

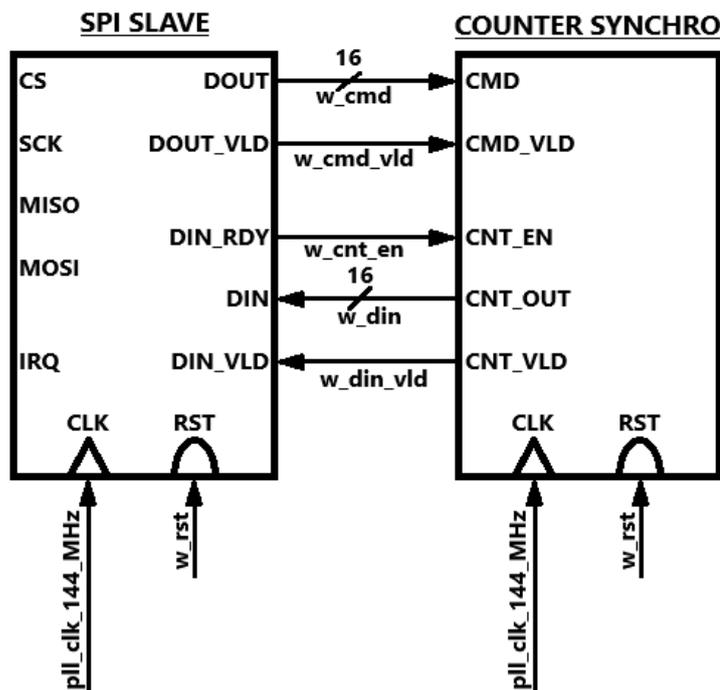


Figura 4.21: Schema a blocchi di 03_SPI_SLAVE_IRQ_COUNTER

- ricevuto il comando di STOP ("00CC" in esadecimale), questo viene trasmesso al contatore che passerà allo stato OFF;
- in questo stato, qualsiasi frame venga inviata via SPI (che non sia la frame di START), verrà ignorata dal contatore e non verrà generato nessun impulso sul pin di IRQ.

Simulazione del sistema 03_SPI_SLAVE_IRQ_COUNTER

Il sistema è stato simulato con Modelsim (Fig. 3.18), con il clock di sistema minimo (129 MHz), tipico (144 MHz) e massimo (159 MHz), in quanto l'incertezza sulla frequenza dell'oscillatore interno è del $\pm 10\%$ (come indicato sul datasheet dell'iCE40 Ultra [28]) e con SPI Clock a 8 MHz e 16 MHz.

In Fig. 4.22 è possibile vedere l'invio di una frame non specifica (d'ora in poi definita "dummy").

Le frame inviate dal Master in simulazione sono quelle presenti in *spi_mdi_tb* (in questo caso "0000" in esadecimale). Queste vengono correttamente trasmesse serialmente sul *MISO*, per poi finire nel bus *DOUT* (collegato direttamente al bus *CMD*) sul quale si leggerà lo stesso valore (nel primo caso quindi, ancora "0000") a fine trasmissione. Viene poi generato l'impulso su *DOUT_VLD* (collegato al

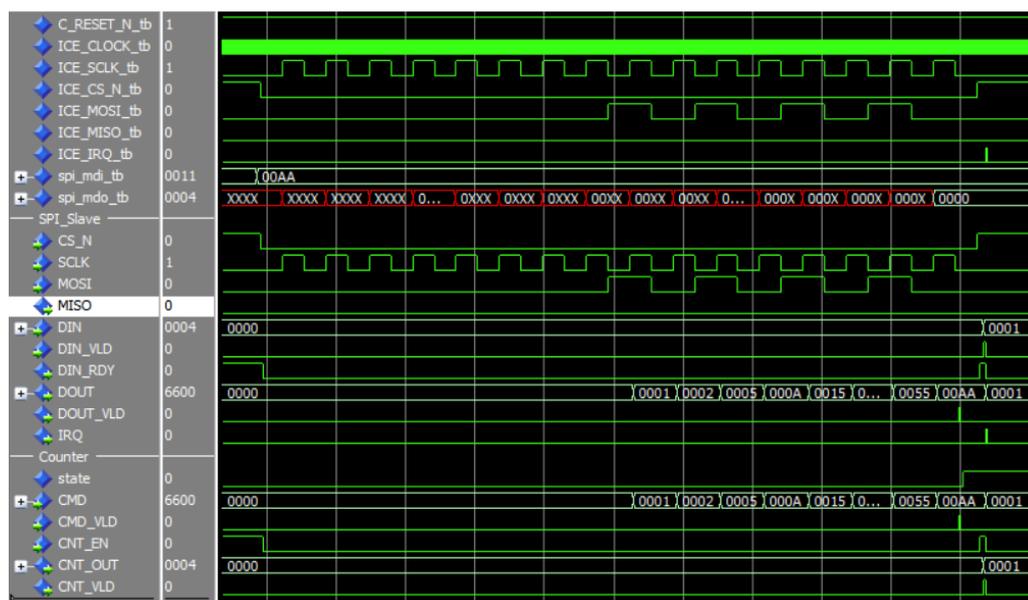


Figura 4.23: Ricezione della frame di START, con clock di sistema a 129 MHz e SPI Clock a 16 MHz

In Fig. 4.24 viene inviata una frame dummy mentre il contatore è in stato ON (*state* ha valore logico alto). Come si vede dalla figura, la differenza dal caso precedente (Fig. 4.22) è che quando il contatore è in stato ON, una volta ricevuta la frame dummy e generato l'impulso sul pin *DIN_RDY*, il contatore viene incrementato e viene generato un nuovo impulso di *IRQ*, per richiedere nuovamente la trasmissione al Master.

In questa maniera sarà possibile instaurare un loop tra Master e Slave, nel quale:

- lo Slave richiede la comunicazione con un impulso sul pin di *IRQ*;
- il Master invia una frame dummy e legge il dato dello slave, inviato serialmente sul *MISO*;
- lo Slave, ricevuta la frame dummy, genera un nuovo dato e un nuovo impulso di *IRQ* facendo ripartire il loop.

In questa maniera è possibile simulare la lettura dei dati dall'ASIC o, più precisamente, dal FIFO che li bufferizzerà (Fig. 4.16).

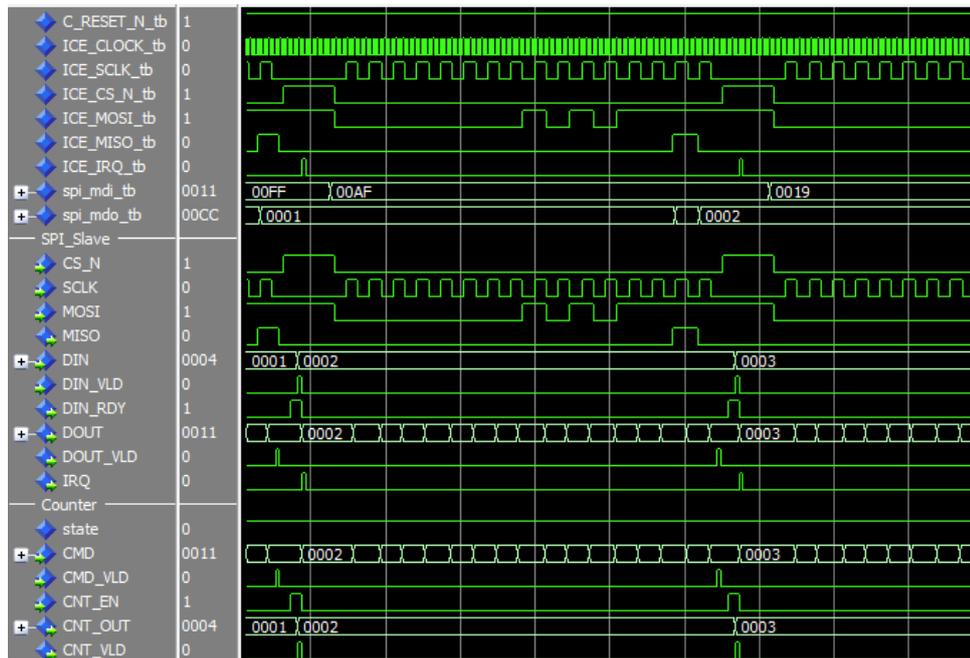


Figura 4.24: Ricezione di una frame dummy mentre il contatore è in stato ON, con clock di sistema a 129 MHz e SPI Clock a 16 MHz

Il loop instaurato sarà interrotto dall'invio della frame di STOP ("00CC" in esadecimale), come mostrato in Fig. 4.25. In seguito alla ricezione di questa frame, è possibile vedere come il contatore passa in stato OFF (*state* assume valore logico basso) come desiderato. Di conseguenza il contatore non viene incrementato e non viene generato alcun impulso sul pin di IRQ a fine trasmissione.

Come ultima prova di questa simulazione è stata inviata nuovamente una frame dummy (Fig. 4.26) per verificare che venga ignorata proprio come la prima (Fig. 4.22).

Test con SPIDriver

E' stato effettuato poi un test su hardware, con l'ausilio di un SPIDriver (Fig. 3.19), tramite il quale sono state inviate le frame di prova viste in simulazione (con un SPI Clock di 500 kHz) e ci si è assicurati di ricevere le stesse risposte, sfruttando il display dell'SPIDriver.

Per effettuare il test innanzitutto sono stati compilati i codici VHDL ed è stato effettuato il posizionamento dei pin e il routing (Fig. 4.27), tramite iCEcube2 (Fig. 3.16).

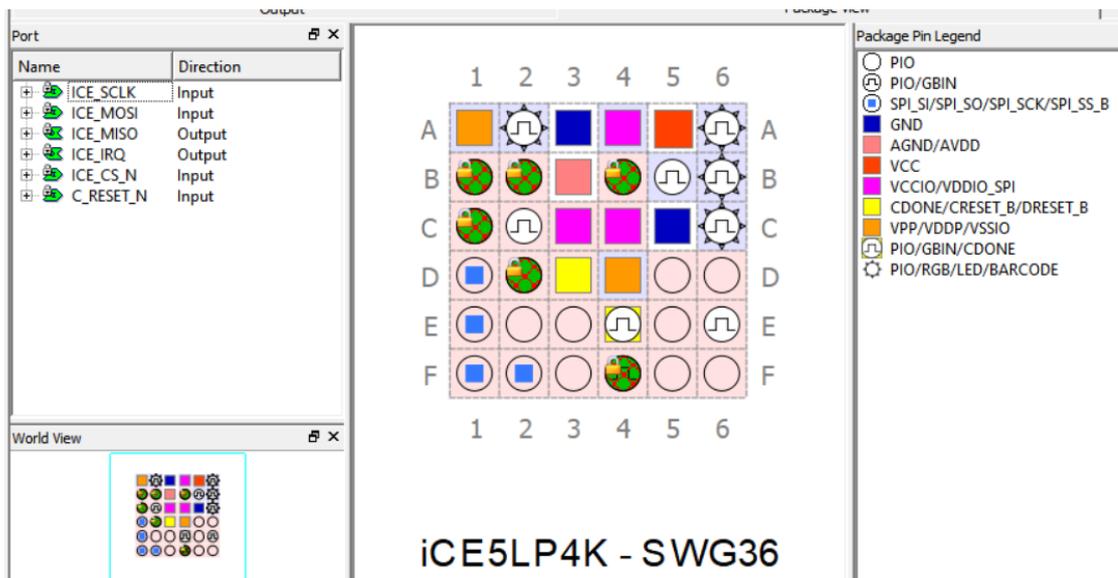


Figura 4.27: Schermata di iCEcube2 durante la modifica dei pin constraint

Tramite lo stesso software è stata poi generata la Bitmap (il file *.bin*) da caricare sull'iCE40 Ultra con l'ausilio di Diamond Programmer (Fig. 3.17). Sui log nella finestra del programmer (Fig. 4.28) è possibile vedere se l'operazione di flash del programma ha avuto esito positivo.

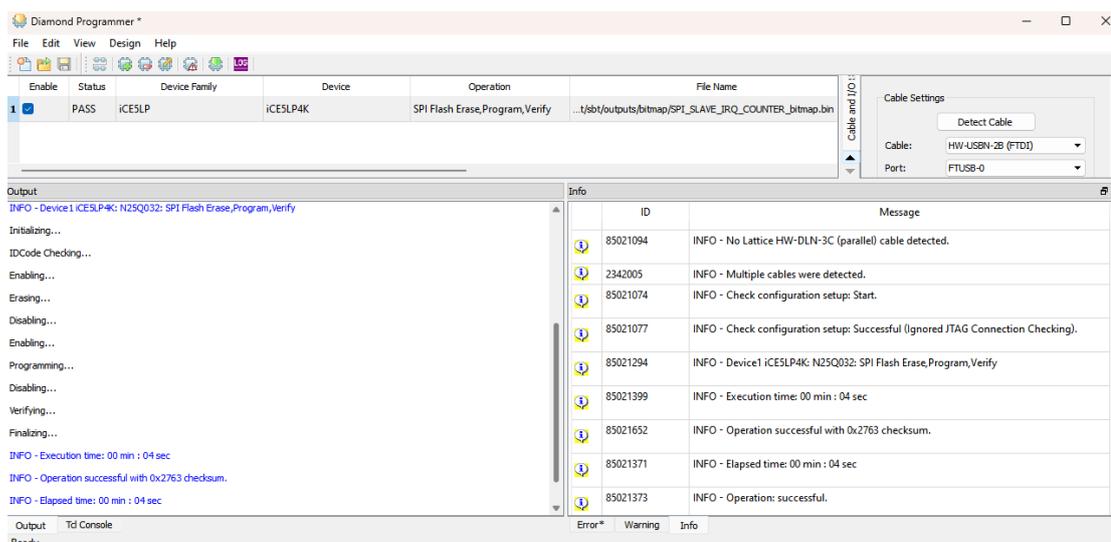


Figura 4.28: I log su Diamond Programmer indicano che il programma è stato caricato correttamente sull’FPGA

Dopo aver caricato il programma sulla breakout board 3.8, va composto il set-up collegando i pin dell’SPI driver a quelli dell’FPGA, come descritto nella Tab. 4.4, utilizzando dei jumper (Fig. 4.29).

SPIDriver	FPGA
SCK	F4
MOSI	B4
MISO	C1
CS	D2
GND	GND

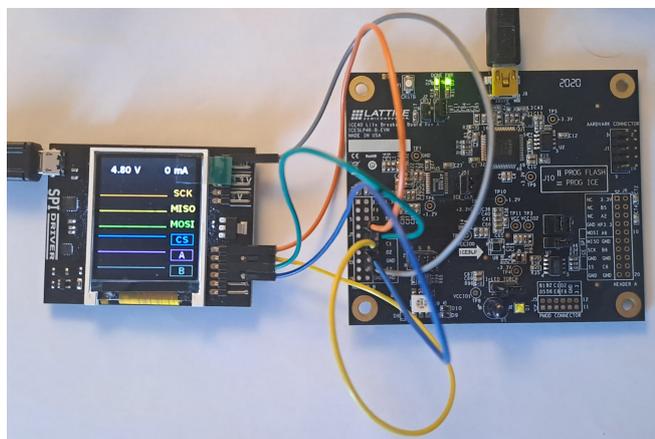


Tabella 4.4: Corrispondenza dei pin tra SPIDriver e FPGA.

Figura 4.29: Set-up per il test con SPIDriver

Tramite l'apposito software fornito con l'SPIDriver (Fig. 4.30), vengono inviate da PC le stesse frame inviate in simulazione per verificare che, ricevute dall'FPGA, generino gli stessi risultati.

L'SPIDriver invia queste frame con un SPI Clock di 500 kHz, il che permette una ulteriore caratterizzazione dei limiti del modulo SPI Slave implementato.

Successivamente verrà effettuato un test col Microcontrollore per verificare il funzionamento anche a 8 MHz.

Dalle immagini nelle figure successive (Fig. 4.31-4.37) si evince come il test ha confermato quanto visto in simulazione. Dal display dell'SPIDriver sarà infatti possibile leggere quanto ricevuto sul MISO in risposta alle frame inviate e quanto letto rispecchia i risultati simulati, confermando il funzionamento del dispositivo.

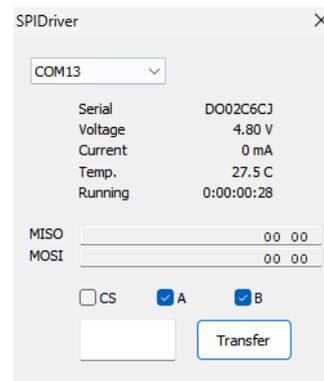


Figura 4.30: Interfaccia grafica per PC dell'SPIDriver

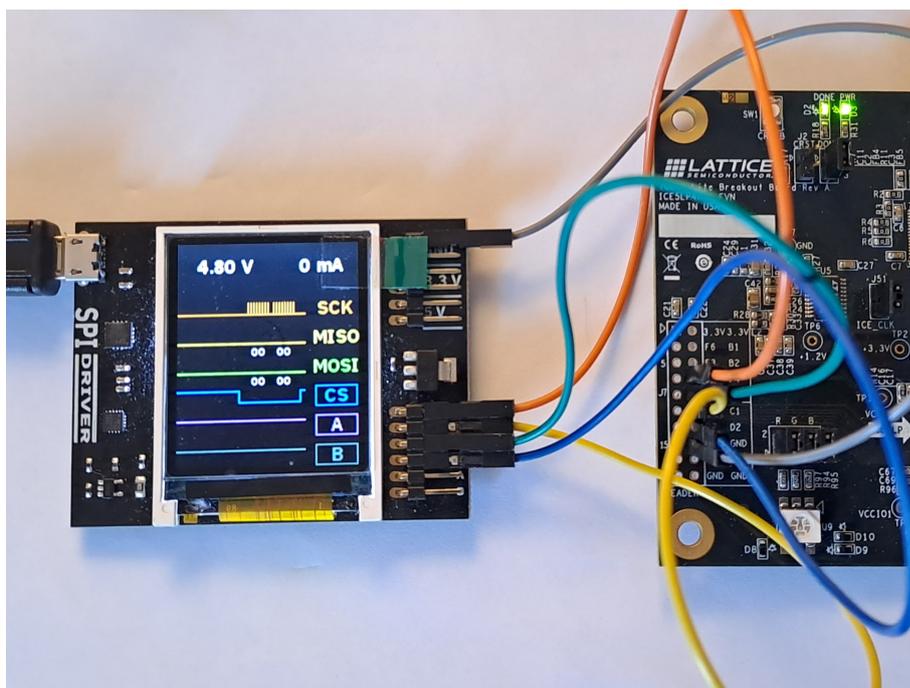


Figura 4.31: Invio della frame "0000" con contatore in stato OFF

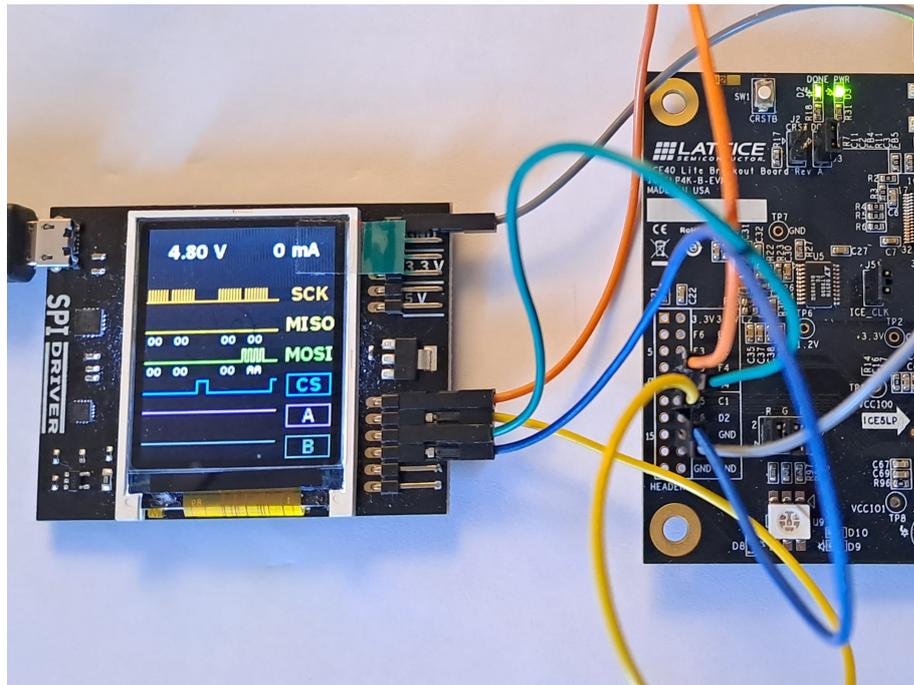


Figura 4.32: Invio della frame di accensione "00AA"

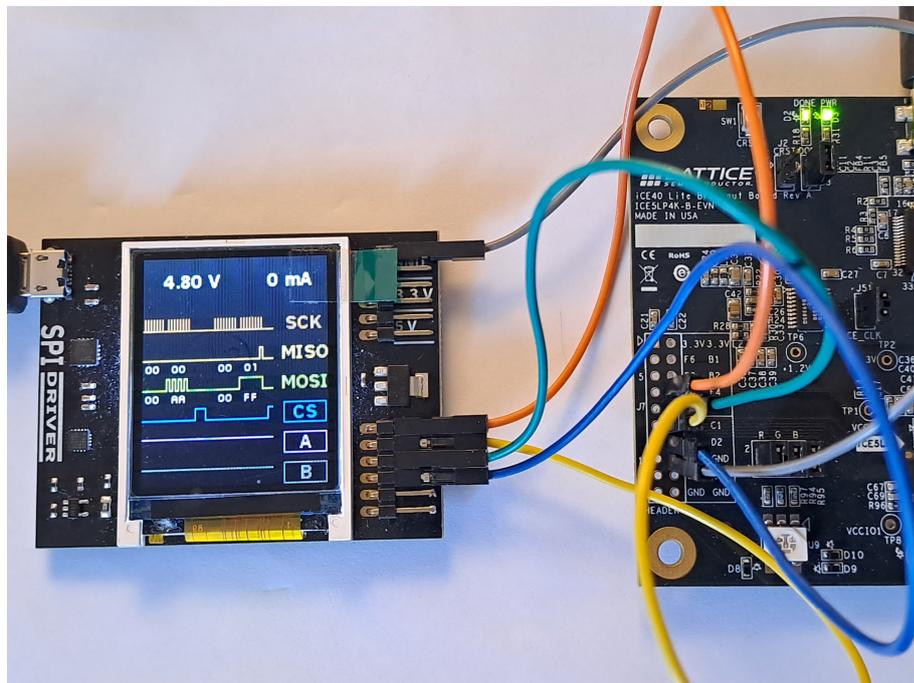


Figura 4.33: Invio della frame "00FF" con contatore in stato ON

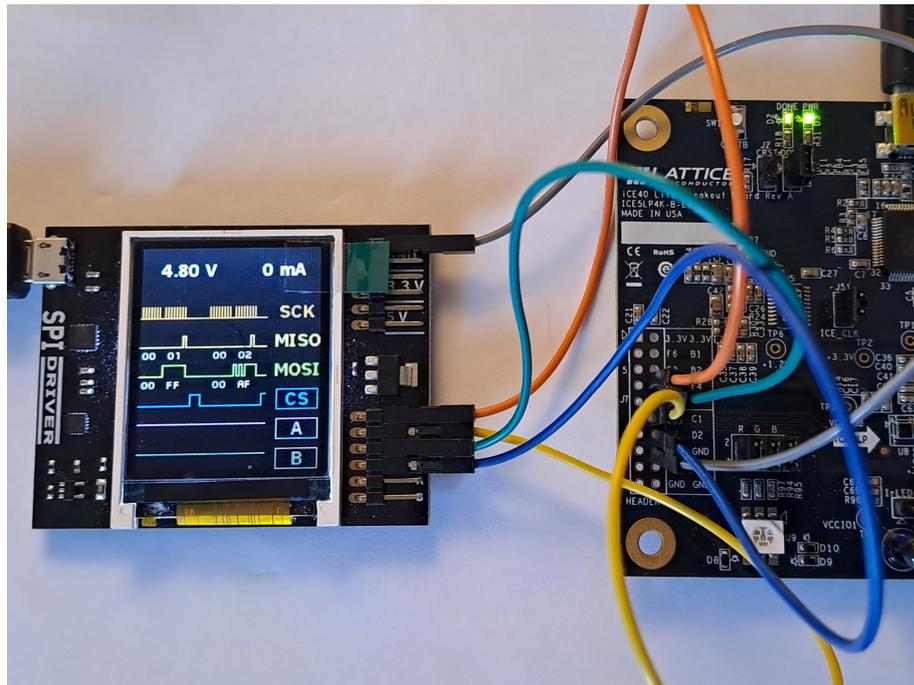


Figura 4.34: Invio della frame di accensione "00AF" con contatore in stato ON

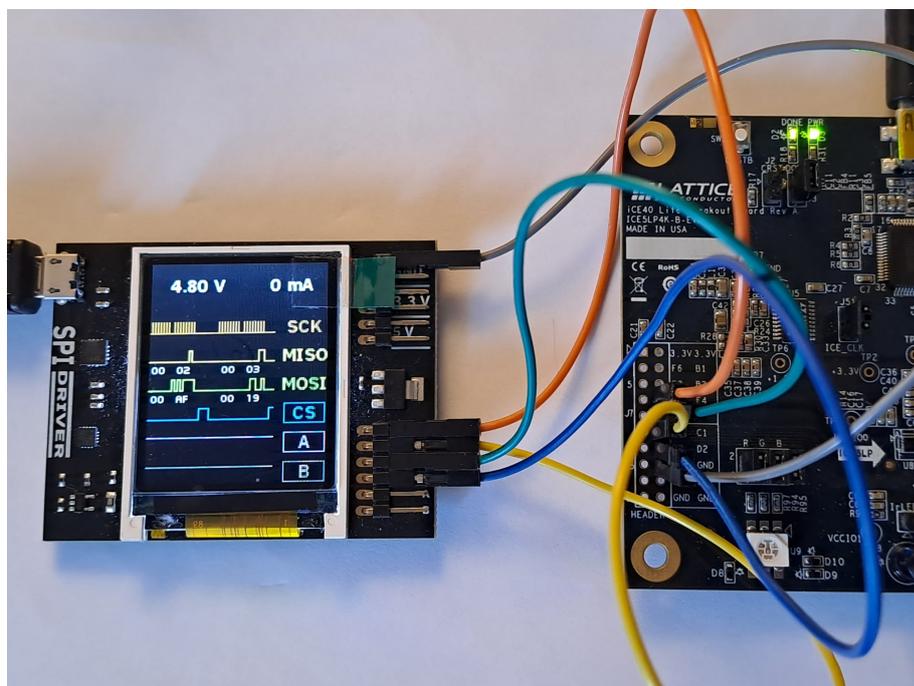


Figura 4.35: Invio della frame "0019" con contatore in stato ON

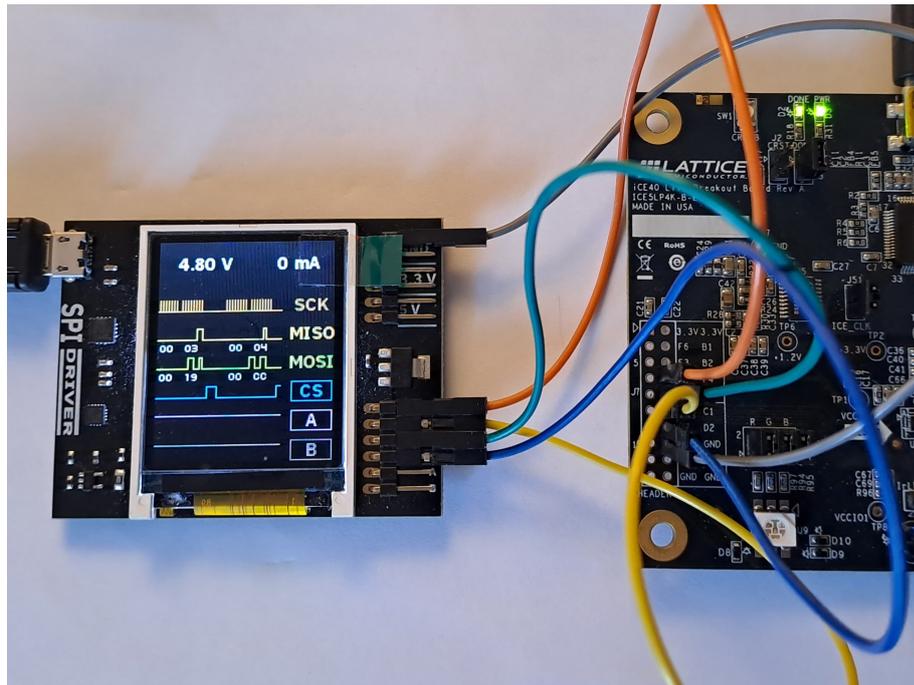


Figura 4.36: Invio della frame di spegnimento "00CC"

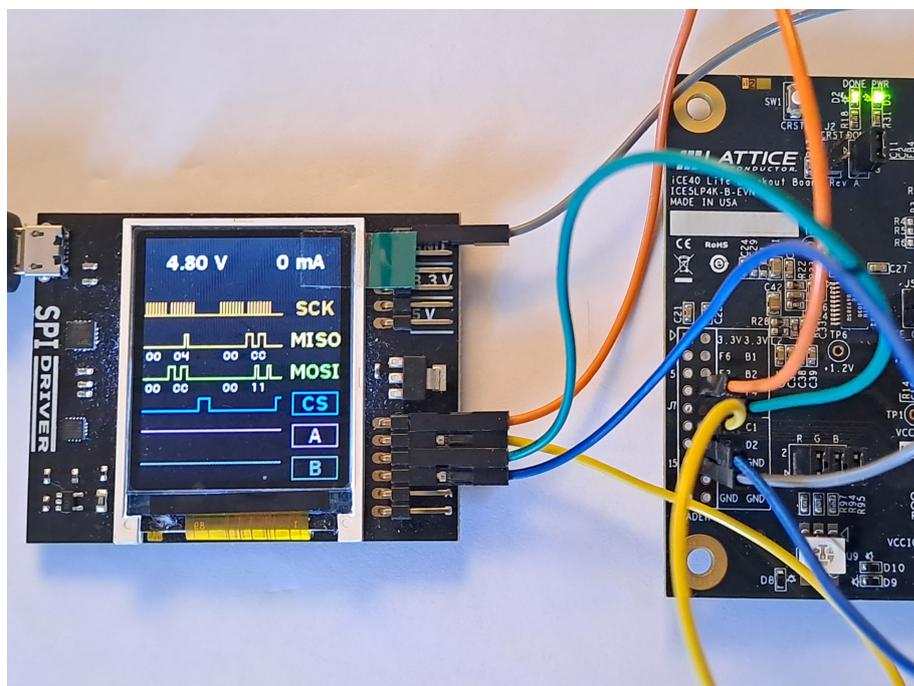


Figura 4.37: Invio della frame "0011" con contatore in stato OFF

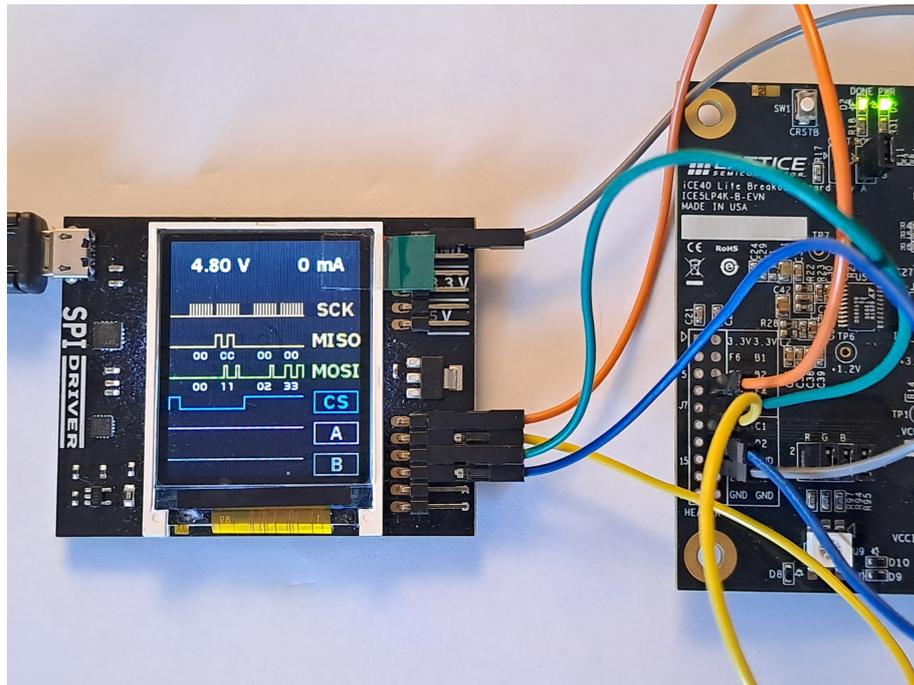


Figura 4.38: Invio della frame anomala "00CC"

Infine per testare la robustezza del codice ad eventuali errori, si è provato ad inviare delle frame anomale, ad esempio mantenendo il CS alto (Fig. 4.38), per vedere come risponde il sistema. Tuttavia il codice non ha mostrato bug di alcun tipo, continuando a restituire "0000" sul MISO se il contatore è in stato OFF, oppure il risultato precedente se il contatore si trova in stato ON.

4.2.3 Sistema MCU-FPGA

In questa sezione viene analizzato quello che rappresenta il punto d'arrivo di questa tesi. I software definitivi, sviluppati per il Microcontrollore (Sez. 4.1.2) e l'FPGA (Sez. 4.2.2), vengono caricati sui relativi dispositivi, a formare il sistema descritto in Fig. 4.39. Questo apparato, vicino a quello completo (Fig. 3.1), è composto da:

- un PC;
- un'interfaccia BLE (Dongle nRF52840, Fig. 3.11);
- un Microcontrollore che supporta il protocollo Bluetooth 5 (nRF52832, Fig. 3.5);
- un'FPGA (iCE40 Ultra, Fig. 3.7).

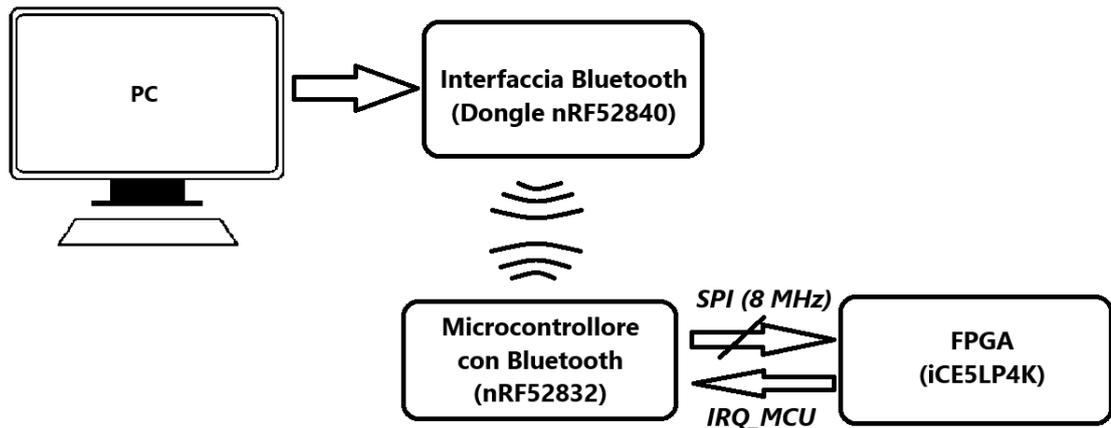


Figura 4.39: Sistema finale

Il sistema permette di instaurare la comunicazione bidirezionale ricercata, a meno dell'ASIC. In questo sistema infatti, i dati saranno generati internamente all'FPGA per mezzo del contatore presentato in precedenza (Sez. 4.2.2), in modo da simulare l'invio dei dati da parte dell'ASIC. Inoltre la comunicazione senza fili implementata, permette di rendere il dispositivo impiantabile nelle future applicazioni pratiche. Il dispositivo, infatti, permetterà di comunicare dal PC verso l'FPGA e viceversa in questo modo:

- partendo dal PC verso l'FPGA:
 1. attraverso il PC, viene mandato un comando di START o STOP;
 2. sfruttando l'interfaccia Bluetooth, il comando viene inviato al Microcontrollore;
 3. il Microcontrollore a sua volta, trasmette il comando alla FPGA tramite SPI, con un SPI clock di 8 MHz;
 4. l'FPGA risponde al comando ricevuto, cambiando lo stato di funzionamento (ON/OFF) del contatore, avviando o arrestando la trasmissione dati;
- partendo dall'FPGA verso il PC:
 1. nel caso in cui il contatore sia nello stato ON, ad ogni frame ricevuta, che non sia quella di STOP, l'FPGA risponde incrementando il contatore e richiedendo la comunicazione al Microcontrollore per mezzo di un impulso sul pin IRQ_MCU;
 2. il Microcontrollore capta la richiesta di comunicazione sul pin IRQ_MCU e legge tramite SPI, con SPI clock a 8 MHz, i dati inviati dall'FPGA;

- infine, il Microcontrollore invia i dati ricevuti al PC, sfruttando l'interfaccia Bluetooth, dove verranno successivamente raccolti ed elaborati.

4.2.4 Test del sistema MCU-FPGA

In questa sezione vengono presentati 3 test eseguiti sul sistema presentato:

- il primo, puramente funzionale, verifica che collegati i dispositivi tra di loro, il sistema risponda correttamente ai comandi di START e STOP inviati via BLE, come verificato nei test precedenti a dispositivi separati;
- il secondo è un test qualitativo di velocità in cui, utilizzando i parametri individuati nello studio sul throughput del BLE (Sez. 4.1.1), si quantifica il data rate in caso di trasmissione di pacchetti variabili, inviati non appena i dati sono disponibili;
- il terzo e ultimo test è identico al secondo ma in questo caso vengono trasmessi pacchetti di dimensione fissa (244 byte), inviati solo dopo averli riempiti completamente.

1 Test Funzionale

In questo test ci si serve di un nRF52 DK (Fig. 3.6), un iCE40 Ultra breakout board (Fig. 3.8), un Dongle nRF52840 (Fig. 3.11) e i software Segger Embedded Studio (SES, Fig. 3.12) e nRF Connect for Desktop (Fig. 3.13).

Sull'FPGA viene compilato e caricato il codice `03_SPI_SLAVE_IRQ_COUNTER`, con l'ausilio di iCEcube2 e Diamond Programmer (Fig. 4.40 - 4.41), in modo analogo a quello nel test in Sez. 4.2.2.

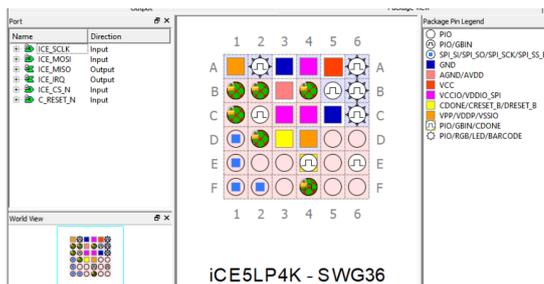


Figura 4.40: Placement&Route eseguito su iCEcube2

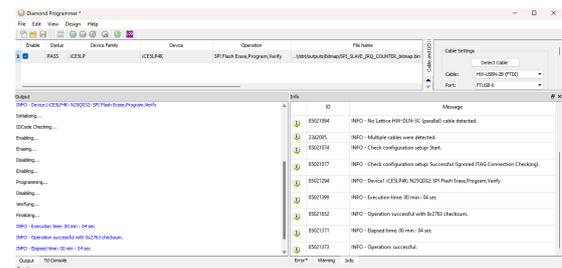


Figura 4.41: Flashing del binario sull'FPGA

Il codice per il Microcontrollore invece, viene caricato sull'nRF52 DK in modalità Debug, tramite SES, in modo da sfruttare il Debug Terminal per leggere i log ed individuare eventuali anomalie (Fig. 4.42).

Infine viene utilizzato il Dongle con il supporto di nRF Connect for Desktop (Fig. 4.43) per inviare i comandi di START e STOP e ricevere i dati inviati dal Microcontrollore.

Il set-up complessivo che ne risulta è quello in Fig. 4.44.

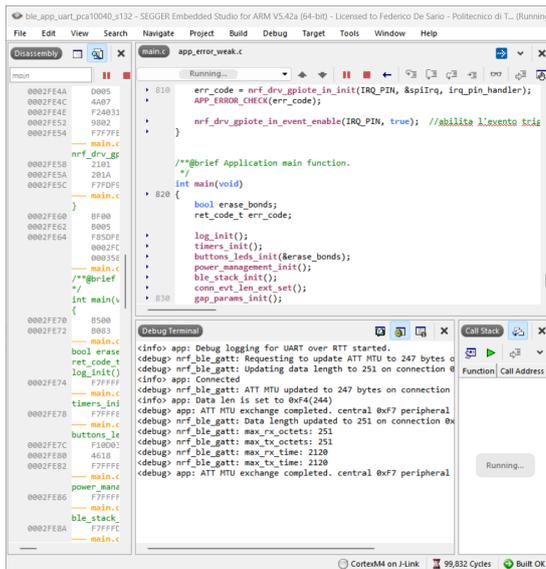


Figura 4.42: Finestra di SES nel test funzionale

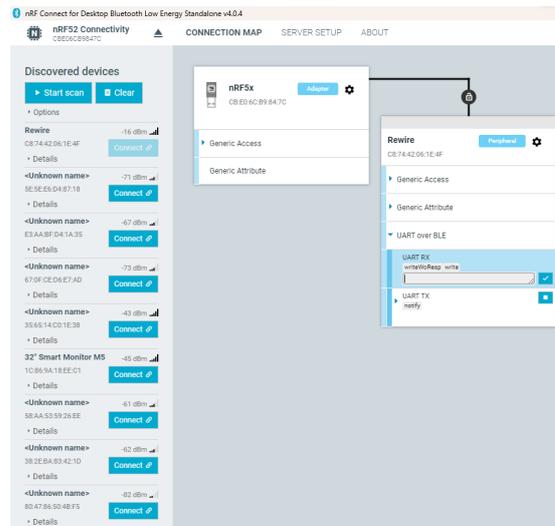


Figura 4.43: Flashing del binario sull'FPGA

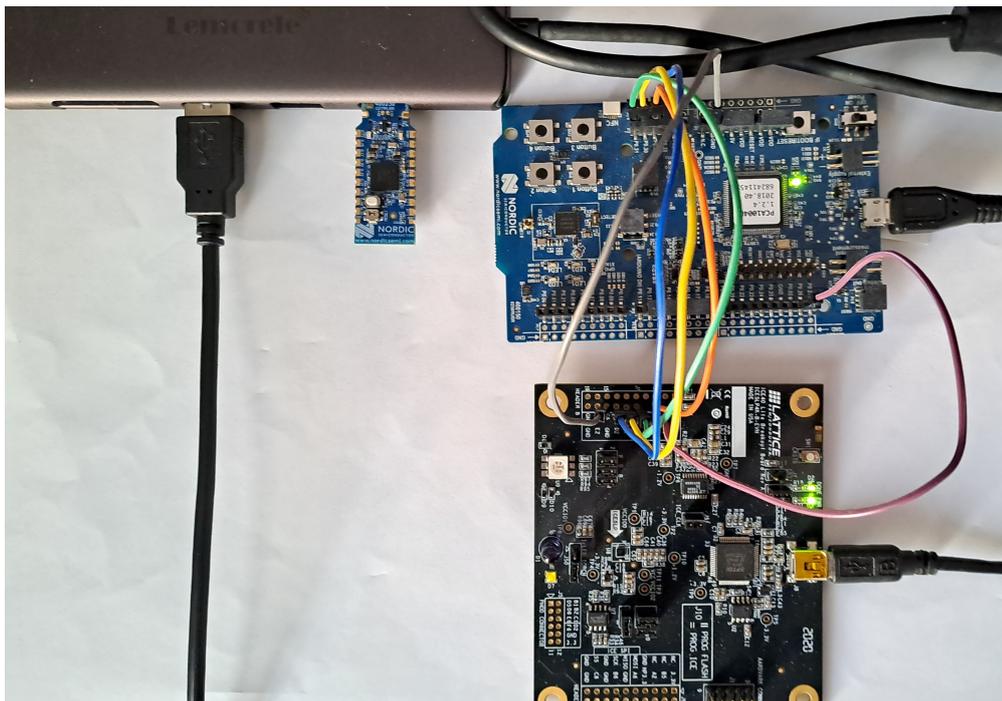


Figura 4.44: Set-up del test funzionale

Se si manda, via BLE, la stringa START, questa viene correttamente ricevuta dal Microcontrollore che invia il comando di ON ("00AA" in esadecimale) all'FPGA. Come si può notare dai log nel Debug Terminal (Fig. 4.45), si instaura il loop desiderato, in cui :

- ad ogni frame ricevuta l'FPGA incrementa il contatore e invia il risultato via SPI insieme ad un impulso sul pin IRQ_MCU;
- ad ogni impulso sul pin IRQ_MCU il Microcontrollore risponde inviando una frame dummy all'FPGA, per leggere il nuovo dato.

Come si può vedere dai log su nRF Connect for Desktop (Fig. 4.46), i dati che l'FPGA invia al Microcontrollore, vengono correttamente inoltrati dallo stesso al Dongle. Infatti, si potrà notare come i dati ricevuti sono in successione, attestando il funzionamento del contatore e della trasmissione SPI e BLE.

```

<debug> app: Interrupt partito. Mando frame dummy.
<debug> app: Interrupt partito. Mando frame dummy.
do frame dummy.
Logs dropped (6)
<debug> app: Interrupt partito. Mando frame dummy.
Logs dropped (6)
<debug> app: Interrupt partito. Mando frame dummy.
Logs dropped (6)
<debug> app: Interrupt partito. Mando frame dummy.
Logs dropped (11)
<debug> app: Interrupt partito. Mando frame dummy.
Logs dropped (15)
<debug> app: Interrupt partito. Mando frame dummy.
Logs dropped (10)
<debug> app: Interrupt partito. Mando frame dummy.

```

Figura 4.45: Debug Terminal del MCU dopo l'invio del comando di START. I log rossi sono i log che non sono stati stampati a causa della velocità con cui vengono generati

```

18:52:57.161 Attribute value changed, handle: 0x0F, value (0x): 01-BE
18:52:57.162 Attribute value changed, handle: 0x0F, value (0x): 01-BF
18:52:57.163 Attribute value changed, handle: 0x0F, value (0x): 01-C0
18:52:57.164 Attribute value changed, handle: 0x0F, value (0x): 01-C1
18:52:57.165 Attribute value changed, handle: 0x0F, value (0x): 01-C2
18:52:57.165 Attribute value changed, handle: 0x0F, value (0x): 01-C3

```

Figura 4.46: Terminale di nRF Connect dove vengono stampati, sotto forma di log, tutti i dati ricevuti dopo l'invio del comando di START. Si può notare come vengono ricevuti correttamente in sequenza

2 Test di velocità con dimensione variabile dei pacchetti BLE

In questo test, in maniera simile a quello in Sez. 4.1.1, avremo 3 componenti principali:

- l'Implant che è il dispositivo su cui caricheremo il codice definitivo per il Microcontrollore (Sez. 4.1.2), che si occuperà di ricevere i comandi e inviare i dati ricevuti dall'FPGA;
- il Controller, che è il dispositivo su cui caricheremo il codice per l'invio dei comandi all'Implant e il calcolo del data rate (simile al Controller nel test eseguito nella Sez. 4.1.1);
- l'FPGA con caricato il suo codice definitivo (Sez. 4.2.2).

L'esperimento consisterà nello sfruttare il Controller per inviare il comando di START all'Implant, il quale a sua volta invierà la frame di ON all'FPGA. Successivamente, come visto nel test precedente a questo, si entrerà in un loop che permetterà al Microcontrollore di ricevere i dati generati dall'FPGA e inviarli al Controller. Una volta inviati 32400 dati, la trasmissione SPI e BLE si interrompe e il Controller calcola il data rate dividendo il numero di dati ricevuto (32400 appunto) per il tempo trascorso tra la ricezione del primo e dell'ultimo dato (Eq. 4.3).

In questo test in particolare non viene forzata la dimensione del pacchetto BLE, ma ne viene solamente fissata la dimensione massima a 244 byte di ATT Payload (Fig. 3.4).

$$Datarate = \frac{N_{datiinviati}}{\Delta t_{1 \rightarrow N}} \quad (4.3)$$

Per l'esecuzione del test, verranno utilizzate due board nRF52 DK (Fig. 3.6), una iCE40 Ultra breakout board (Fig. 3.8), un Dongle nRF52840 (Fig. 3.11) e i software SES (Fig. 3.12), Wireshark (Fig. 3.14) e MATLAB (Fig.3.15).

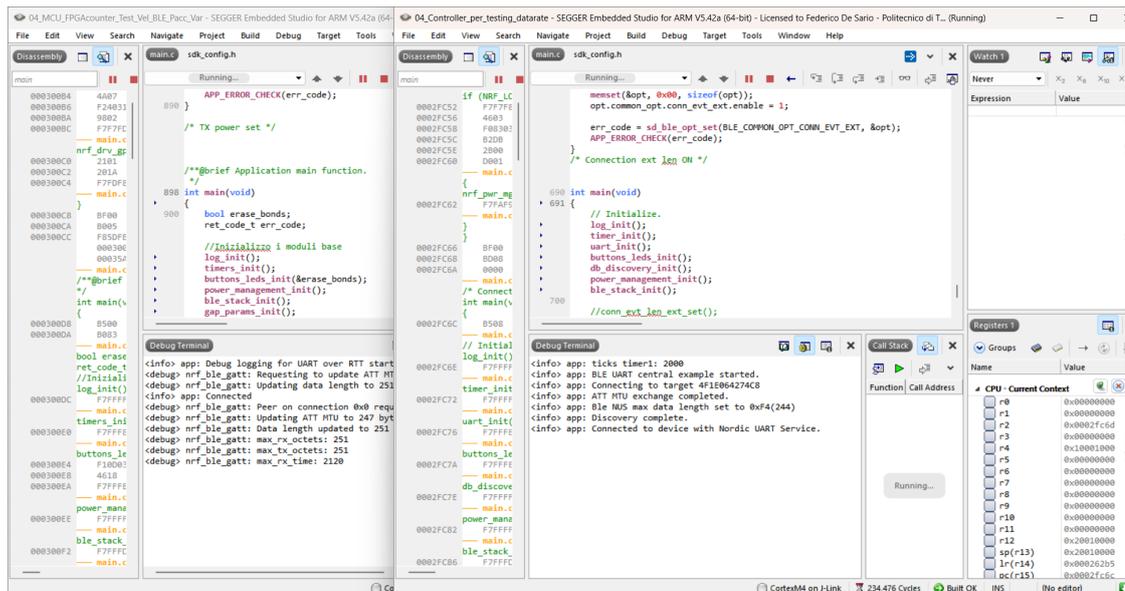


Figura 4.47: Schermate di SES dell'Implant e del Controller durante il test di velocità

I due codici sono stati caricati sulle board in modalità Debug, tramite SES (Fig. 4.47), in modo da poter leggere i log nel Debug Terminal. In questa maniera è possibile verificare il funzionamento del sistema e leggere il valore del data rate alla fine del test. Per programmare l'FPGA invece sono stati usati iCEcube2 e Diamond Programmer come durante il test nella Sez. 4.2.2

Sfruttando il Dongle, con caricato il Packet Sniffer [34] e Wireshark è stato possibile osservare i pacchetti inviati (Fig. 4.49) e individuare possibili errori di trasmissione che portano alla ritrasmissione dello stesso pacchetto, rallentando il data rate.

Tramite MATLAB invece (Fig. 4.49), è stato possibile inviare il Comando di START al Controller (sfruttando l'UART) e rappresentare i dati ricevuti dall'FPGA.

Collegati infine i pin dell'Implant e dell'FPGA come in Tab. 4.5, si ottiene il set-up complessivo (Fig. 4.48), riutilizzato anche nel test successivo.

Pin MCU	Nome PIN	Pin FPGA
29	SCK	F4
31	MOSI	B4
30	MISO	C1
28	CS	D2
26	IRQ	B1
GND	GND	GND

Tabella 4.5: Corrispondenza dei pin tra Microcontrollore (Implant) e FPGA.

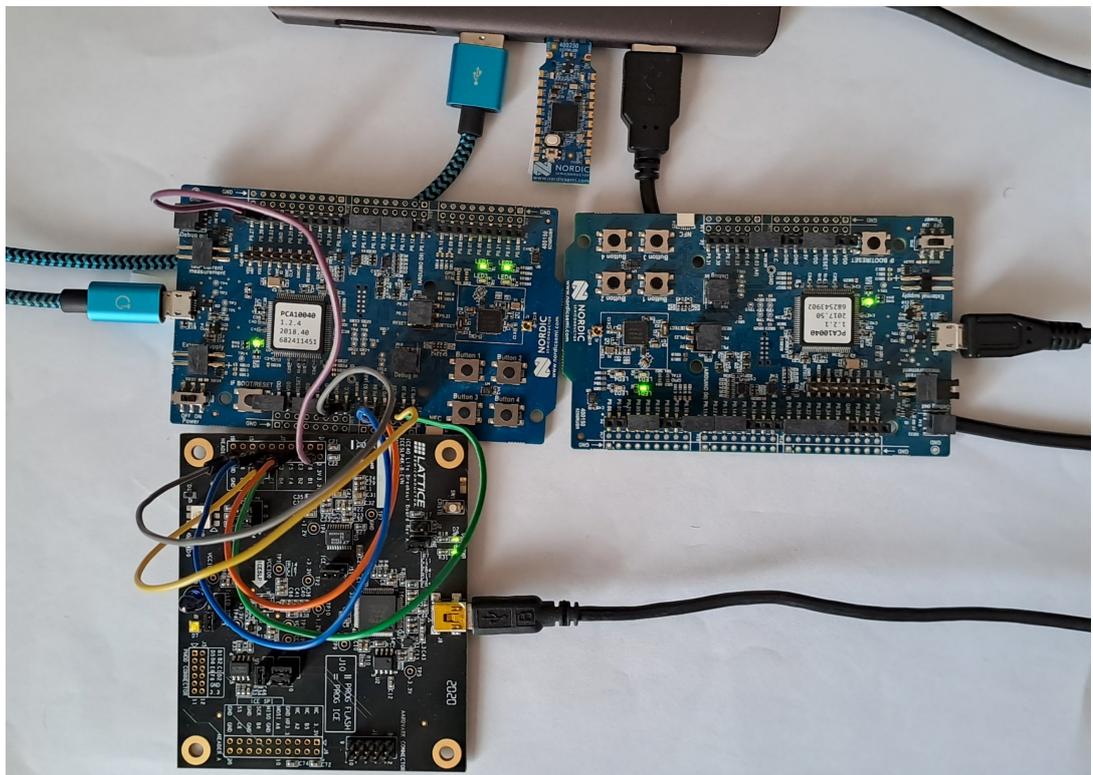


Figura 4.48: Set-up per il test del data rate

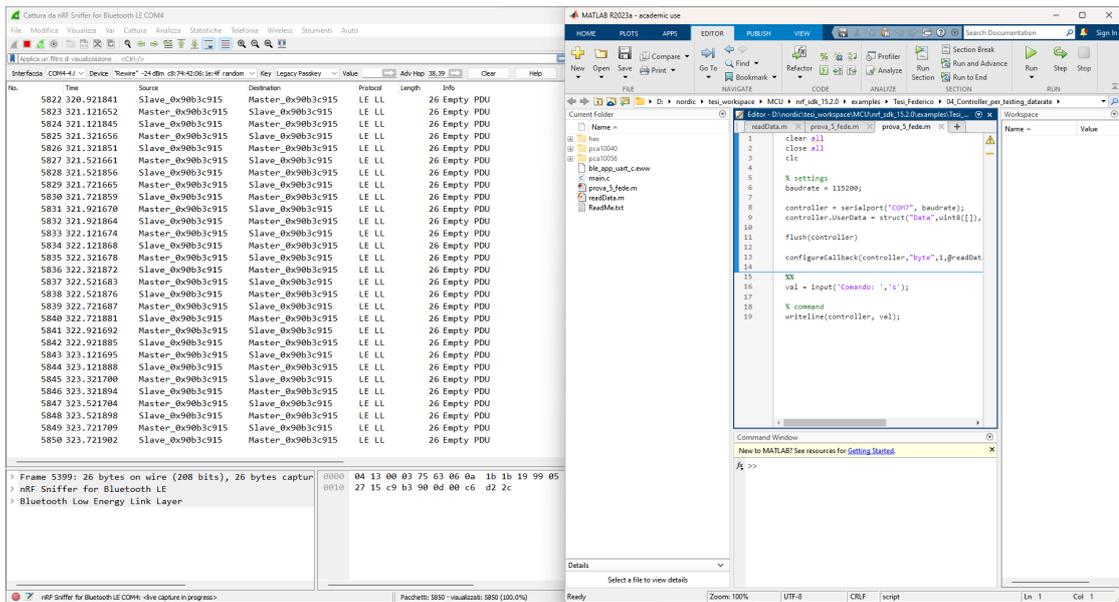


Figura 4.49: Schermate di SES dell'Implant e del Controller durante il test di velocità

Dopo aver inviato il comando di START da MATLAB, l'FPGA inizia a generare dati e l'Implant a leggerli e inviarli via BLE al Controller. Dopo qualche secondo, quando tutti i dati sono stati inviati sul Debug Terminal del Controller (Fig. 4.50) sarà possibile leggere il data rate calcolato.

In Fig. 4.51 è presente il grafico di MATLAB da cui verificare che i dati siano stati inviati in sequenza. Infatti, viene ricevuta una sequenza numerica da 0 a 255, valore per cui il contatore si resetta.

```
<info> app: ticks timer1: 2000
<info> app: BLE UART central example started.
<info> app: Connecting to target 4F1E064274C8
<info> app: ATT MTU exchange completed.
<info> app: Ble NUS max data length set to 0xF4(244)
<info> app: Discovery complete.
<info> app: Connected to device with Nordic UART Service.
<info> app: Time elapsed total: 16793.01 ms
<info> app: Received bytes: 32400
<info> app: Datarate: 15.43 bits per ms
```

Figura 4.50: Debug Terminal del Controller a fine test, nel caso di pacchetti di dimensione variabile, con massimo a 244 byte e CI da 200 ms

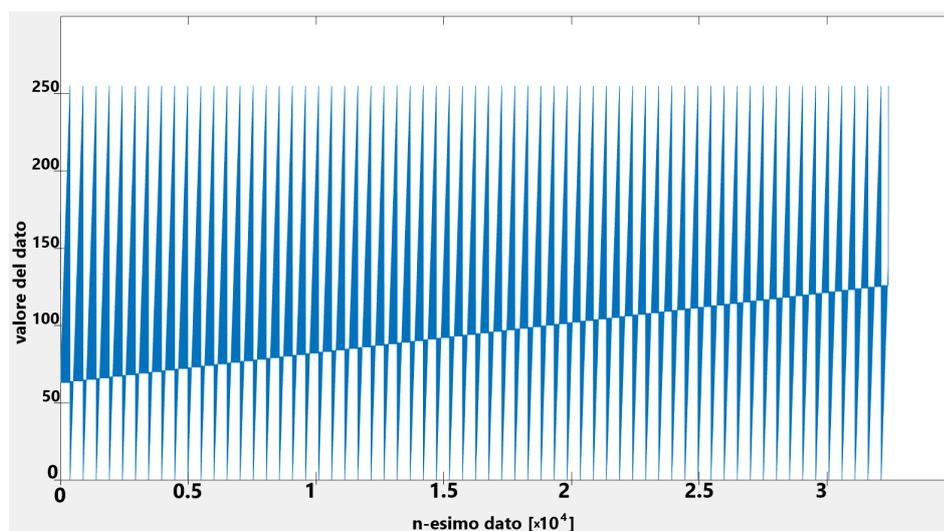


Figura 4.51: Grafico dei dati ricevuti durante il test di velocità. Sulle ascisse è segnato l'n-esimo dato ricevuto e sulle ordinate il suo valore. Si vede come i dati vengono ricevuti in sequenza da 0 a 255.

Il data rate ottenuto in questo test è inferiore di circa 2 ordini di grandezza a quello ottenuto nel test in Sez. 4.1.1 (15.4 kbps contro circa 1.4 Mbps). Questo potrebbe essere legato alla dimensione dei pacchetti inviati, in quanto, come è possibile notare dalla schermata di Wireshark (Fig. 4.52), molti dei pacchetti vengono ritrasmessi e la dimensione di questi è quasi sempre 35 byte, con solo alcuni pacchetti da 277 byte, di cui 244 di ATT Payload (Fig. 3.4).

6445	39.263749	Slave_0x1b75633f	Master_0x1b7563...	ATT	35	Rcvd	Handle	Value	Notification
6460	39.268404	Slave_0x1b75633f	Master_0x1b7563...	ATT	35	Rcvd	Handle	Value	Notification
6484	39.277798	Slave_0x1b75633f	Master_0x1b7563...	ATT	35	Rcvd	Handle	Value	Notification
6526	39.467989	Slave_0x1b75633f	Master_0x1b7563...	ATT	35	Rcvd	Handle	Value	Notification
6539	39.472219	Slave_0x1b75633f	Master_0x1b7563...	ATT	277	Rcvd	Handle	Value	Notification
6590	39.668393	Slave_0x1b75633f	Master_0x1b7563...	ATT	35	Rcvd	Handle	Value	Notification
6613	39.675895	Slave_0x1b75633f	Master_0x1b7563...	ATT	35	Rcvd	Handle	Value	Notification

Figura 4.52: Schermata di Wireshark con visualizzati i pacchetti ritrasmessi

3 Test di velocità con dimensione fissa dei pacchetti BLE

Per i motivi esposti in precedenza, il test è stato effettuato nuovamente con lo stesso set-up (Fig. 4.48), con la differenza che questa volta si forza il numero di byte per pacchetto al valore di ATT MTU scelto (Cod. 4.27), in questo caso 244. In questa maniera ci si assicura di inviare sempre pacchetti interamente pieni, tranne l'ultimo che sarà riempito con i dati rimanenti nell'ultima trasmissione. Questo

potrebbe portare ad un incremento del data rate come visto nel test nella Sez. 4.1.1.

```

1
2 static void spi_fifo_process()
3 {
4     ... ..
5     /*Gestione del dato ricevuto via FPGA da salvare in m_spiRx_fifo
6     (e mandare via BLE)*/
7     if (!ble_tx_active) //se sto gia' trasmettendo via BLE salto
8     questo blocco
9     {
10        /*Vedo quanti elementi ho ricevuto nel buffer spiRx*/
11        queue_elements = spiRx_fifo_utilization_get();
12        /*Se ho riempito un pacchetto allora procedo a inviarlo*/
13        if (queue_elements >= (REWIRE_MTU/2))
14        {
15            for (int i=0; i < (REWIRE_MTU/2); i++)
16            {
17                /*Tolgo un dato dal fifo di ricezione SPI (da FPGA) e
18                lo salvo in element*/
19                err_code = spiRx_fifo_pop(&element);
20                /*Ogni due elementi consecutivi di nusTx sono un
21                elemento di element (ovvero un dato da 2 byte ricevuto via SPI
22                dall'FPGA)*/
23                nusTx[i*2] = element >> 8;
24                nusTx[i*2+1] = (uint8_t)element;
25            }
26            /*Una volta Ricevuti i dati, setto ble_tx_active su true
27            che vuol dire che sto mandando dati via BLE*/
28            ble_tx_active = true;
29            /*La dimensione del buffer che viene inviato via ble (
30            nusTx) e' pari alla dimensione del pacchetto scelta*/
31            txSize = REWIRE_MTU;
32            /*Puntatore che decide quale elemento di nusTx inviare*/
33            txRdPtr = 0;
34            NRF_LOG_DEBUG("Ready to send data over BLE NUS");
35            /*Si occupa della trasmissione BLE del dato ricevuto via
36            SPI, gestendolo come una notifica che viene mandata via BLE (al
37            controller o dongle) sfruttando il NUS*/
38            ble_data_send();
39        }
40        /*Se invece ho raggiunto il numero di dati da inviare,
41        riempio l'ultimo pacchetto coi dati rimanenti (meno di REWIRE_MTU)
42        */
43        else if ((queue_elements > 0) && (data_sent > (DATA_TO_SEND-
44        REWIRE_MTU)))
45        {
46            NRF_LOG_DEBUG("Last packet size /2: %u", queue_elements);

```

```

35     for(int i=0; i < queue_elements; i++)
36     {
37         ... ..
38     }
39     ... ..
40     /*La dimensione del buffer che viene inviato via ble (
nusTx) e' pari al doppio di quella spiRx*/
41     txSize = 2*queue_elements;
42     ... ..
43 }
44 }
45 }

```

Cod. 4.27: Forzo i pacchetti ad essere di REWIRE_MTU elementi

Come nel test precedente, vengono caricati i codici in Debug Mode sulle board nRF52 DK (Fig. 4.47) e caricato il file binario sull'FPGA. Anche qui vengono sfruttati il Dongle con Wireshark e il Packet Sniffer per visualizzare i pacchetti inviati e MATLAB per inviare il comando di START.

Rieseguito il test, si nota un lieve miglioramento, passando da 15.4 kbps a 17.5 kbps (Fig. 4.53), ma ancora lontani dagli 1.4 Mbps.

```

<info> app: ticks timer1: 2000
<info> app: BLE UART central example started.
<info> app: Connecting to target 4F1E064274C8
<info> app: ATT MTU exchange completed.
<info> app: Ble NUS max data length set to 0xF4(244)
<info> app: Discovery complete.
<info> app: Connected to device with Nordic UART Service.
<info> app: Time elapsed total: 14947.61 ms
<info> app: Received bytes: 32768
<info> app: Datarate: 17.53 bits per ms

```

Figura 4.53: Debug Terminal del Controller a fine test, nel caso di pacchetti di dimensione fissa a 244 byte e CI da 200 ms

Stavolta non ci sono state ritrasmissioni a rallentare il data rate, il che spiega il leggero miglioramento.

Anche in questo caso, il grafico elaborato da MATLAB (Fig. 4.54) attesta la corretta ricezione dei dati in sequenza. Come si può vedere infatti i dati ricevuti sono sequenze di numeri da 0 a 255, valore dopo il quale il contatore nell'FPGA si resetta.

Il motivo del rallentamento invece, potrebbe essere legato al cambio di sistema. Nel test di ottimizzazione del BLE (Sez. 4.1.1) i dati venivano generati internamente al Microcontrollore a frequenza costante (16 bit a frequenza 80 kHz). Ora invece vengono generati nell'FPGA, in risposta alle frame inviate dal Microcontrollore, ad

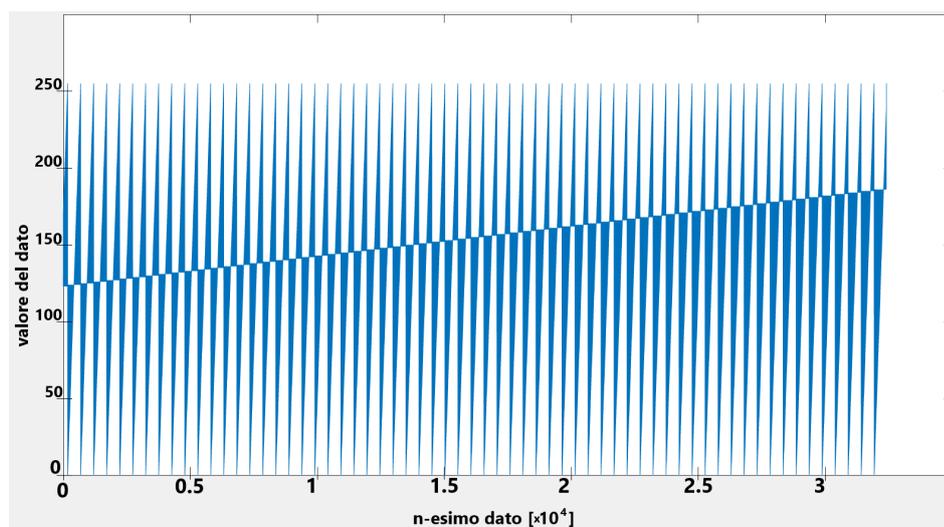


Figura 4.54: Grafico dei dati ricevuti durante il test di velocità. Sulle ascisse è segnato l'n-esimo dato ricevuto e sulle ordinate il suo valore. Si vede come i dati vengono ricevuti in sequenza da 0 a 255.

una frequenza non costante in quanto l'interrupt sul pin di IRQ, ha una priorità inferiore a quella degli interrupt della comunicazione SPI, che a sua volta ha una priorità inferiore rispetto agli interrupt del Softdevice che gestisce la comunicazione BLE. Ci sono quindi 3 operazioni da eseguire, alle quali si aggiunge la generazione dei vari log di debug, utili a verificare il funzionamento del sistema. Operazioni che si interrompono a vicenda, rispettando le loro priorità ma rallentando la velocità di generazione e trasmissione dati.

A dimostrazione di ciò, se si eliminano, ad esempio, tutte le operazioni di log di debug, compilando il codice in versione Release tramite SES, la situazione migliora passando da 17.5 kbps a 76.2 kbps utilizzando gli stessi parametri di connessione (pacchetti da 244 byte e Connection Interval di 200 ms) come si può verificare in Tab. 4.6. Questo lascia intendere che snellire il codice può essere una strategia per velocizzare la trasmissione dati.

E' stato poi effettuato uno studio qualitativo della velocità di trasmissione al variare dei parametri, calcolando il data rate sia tramite il Controller che manualmente dalla finestra di Wireshark ottenendo i risultati in Tab. 4.6 e riassunti nel grafico in Fig. 4.55.

Al momento il miglior risultato è 85.6 kbps senza perdita di dati, utilizzando pacchetti da 200 byte e Connection Interval da 500 ms.

Le misure contrassegnate dall'asterisco "*" sono state misurate manualmente con Wireshark in quanto, per quei valori, non tutti i pacchetti sono stati inviati con successo. Questo non ha permesso al Controller di arrivare al punto di calcolare in

Pacchetti da 50 byte		Pacchetti da 200 byte	
CI [ms]	Data Rate [kbps]	CI [ms]	Data Rate [kbps]
10	76.6	10	73
50	77.5	50	75.1
100	76	100	76
200	76.1	200	57
500	74	500	85.6
1000	84.9*	1000	85.7*
2000	99.8*	2000	128*
Pacchetti da 100 byte		Pacchetti da 244 byte	
CI [ms]	Data Rate [kbps]	CI [ms]	Data Rate [kbps]
10	75	10	73.4
50	76.2	50	75.6
100	76	100	75.8
200	76.1	200	76.2
500	74.11	500	85.3
1000	85.43*	1000	85.81*
2000	118.6*	2000	123.9*

Tabella 4.6: Valori del data rate del dispositivo per diverse combinazioni di dimensione dei pacchetti e Connection Interval. Gli asterischi "*" indicano le misure in cui non tutti i pacchetti sono stati trasmessi con successo

maniera automatica il data rate, alla fine della trasmissione dei 32400 byte.

Inoltre, nelle misure con pacchetti da 50 byte il numero di ritrasmissioni si aggira tra 2 e 54, mentre, per le altre dimensioni di pacchetto, oscillano tra 0 e 1 (praticamente nulle). Ciò lascia intendere che con pacchetti più piccoli, il Packet Error Rate (PER) tende a crescere.

Seppure qualitativo, visto il numero esiguo di combinazioni testate, lo studio dimostra che il data rate può essere migliorato anche con un'opportuna scelta dei parametri di connessione (ATT MTU e Connection Interval), visto che quelli trovati nello studio in Sez. 4.1.1 non sono più ottimali per questa applicazione.

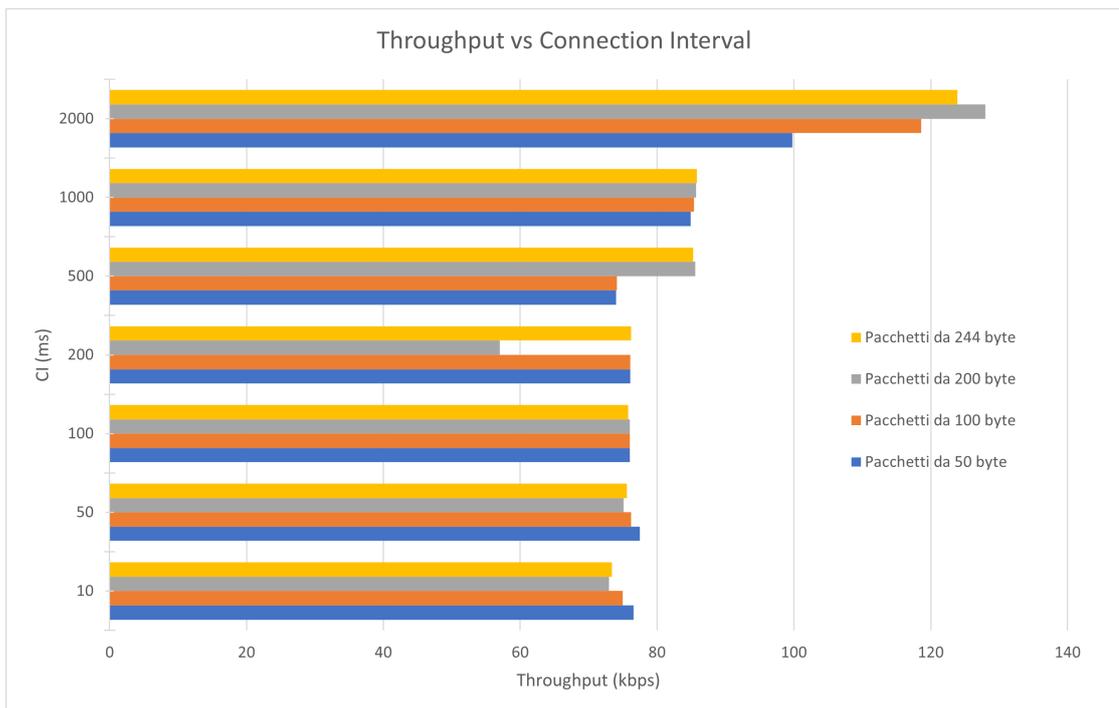


Figura 4.55: Grafico a barre dell'andamento del data rate del dispositivo finale al variare di Connection Interval e dimensione dei pacchetti

Capitolo 5

Conclusioni

In questa tesi è stato sviluppato un sistema che consente una comunicazione BLE, bidirezionale, tra un PC e un dispositivo formato da un Microcontrollore e un'FPGA. Il sistema in questione è capace di inviare, sfruttando un'interfaccia BLE, i comandi di START e STOP dal PC al Microcontrollore il quale a sua volta invia le frame corrispondenti all'FPGA via SPI. L'FPGA risponderà avviando o arrestando un flusso di dati, i quali saranno inviati via BLE al PC (da qui la bidirezionalità del sistema ricercata).

Il software, sviluppato in questa tesi, per realizzare questo sistema è stato analizzato nel capitolo 4, in cui sono stati descritti anche i test effettuati sull'apparato.

I test funzionali garantiscono il funzionamento dei singoli blocchi del sistema, mentre i test di velocità danno una stima di quello che potrebbe essere il data rate del solo Microcontrollore, in cui il collo di bottiglia è rappresentato dal protocollo BLE e del sistema finale, in cui il protagonista principale del rallentamento è la convivenza di più protocolli che concorrono tra loro: il BLE, la comunicazione SPI, l'interrupt sul pin di IRQ e i vari log di debug generati (questi ultimi citati anche dal team del NGNI Lab, nella repository Github resa disponibile a inizio lavoro [14, 15]).

I test garantiscono che il Microcontrollore riesce a trasmettere dati fino a 1.4 Mbps (con pacchetti da 244 byte e CI di 200 ms) quando i dati sono generati internamente ad esso. Tuttavia, questo data rate si abbassa raggiungendo un massimo di 85.6 kbps (con pacchetti da 200 byte e CI di 500 ms) nel momento in cui viene introdotta l'FPGA, la comunicazione SPI e l'interrupt sul pin di IRQ (Fig. 5.1).

I motivi del rallentamento sembrano legati principalmente all'architettura del sistema. La sola rimozione dei log di debug dal codice dell'Implant (MCU) nel test di velocità, ha permesso infatti di passare da circa 17.5 kbps (Fig. 4.53), con pacchetti fissi a 244 byte e CI a 200 ms, a 76.2 kbps (Tab. 4.6) mantenendo gli stessi parametri di connessione (Fig. 5.2).

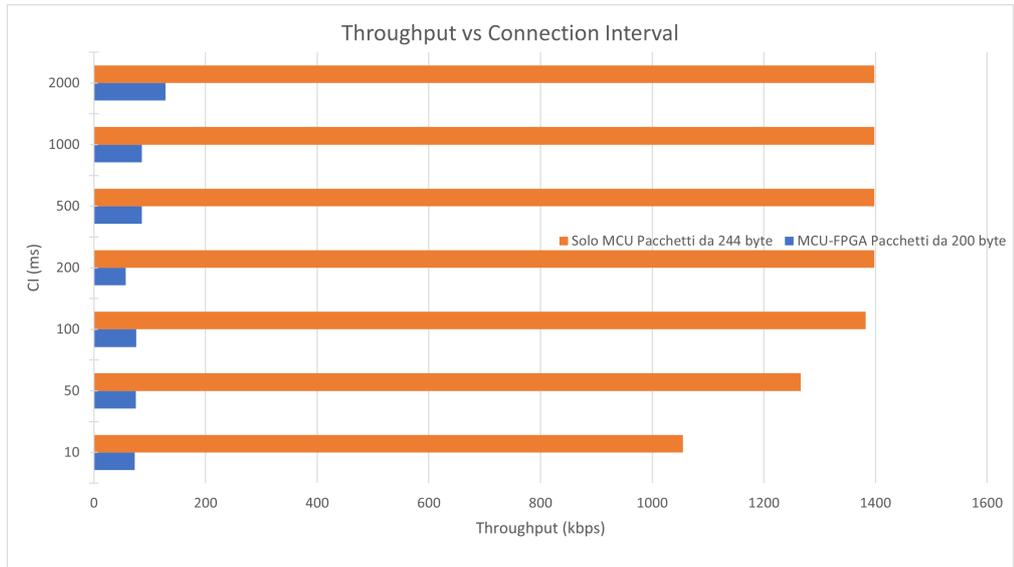


Figura 5.1: Grafico a barre dell'andamento del throughput al variare del Connection Interval con pacchetti nei best-case del sistema con il solo MCU (pacchetti da 244 byte) e del sistema completo (pacchetti da 200 byte)

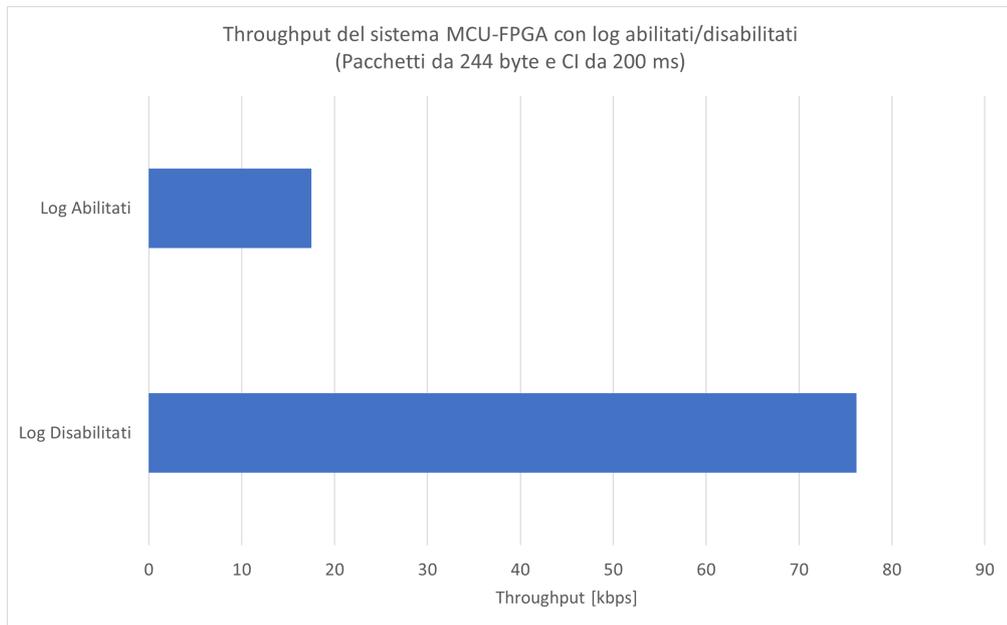


Figura 5.2: Grafico a barre del throughput con i log abilitati/disabilitati, utilizzando gli stessi parametri di connessione (pacchetti fissi da 244 byte e CI da 200 ms)

Ulteriori miglioramenti possono essere apportati con un nuovo studio sui parametri di trasmissione, in quanto con l'ultimo test effettuato, seppur qualitativo, si dimostra come i parametri trovati con il test sul solo MCU (pacchetti da 244 byte e CI di 200 ms) non sono più quelli ottimi (Tab 4.6). Infatti, provando diverse combinazioni si è riusciti a raggiungere gli 85.6 kbps citati precedentemente (con pacchetti da 200 byte e CI di 500 ms).

Sebbene manchi ancora l'implementazione del FIFO bidirezionale nell'FPGA, il sistema sviluppato rappresenta un punto di partenza per i prossimi sviluppi, considerando che:

- il software del Microcontrollore può già comunicare via SPI con l'FPGA e ricevere dati da quest'ultima nel momento in cui riceve una richiesta di comunicazione sul pin di IRQ, esattamente come farà nel sistema completo. Bisognerà sostituire i generici comandi di START e STOP con gli specifici comandi di configurazione dell'ASIC;
- dal lato FPGA, l'interfaccia SPI è stata testata su hardware solo con SPI Clock da 8 MHz, ma è stata simulata anche a 16 MHz, quindi si potrebbe pensare di utilizzare lo stesso modulo anche dal lato ASIC e testarlo su hardware, una volta implementato il FIFO bidirezionale.

5.1 Ottimizzazione e sviluppi futuri

Gli ultimi test svolti, seppur qualitativi, dimostrano che la velocità di trasmissione può ancora essere incrementata agendo sui parametri di connessione, ovvero la dimensione dei pacchetti e il Connection Interval (Fig. 4.55 e Tab. 4.6).

Inoltre l'eliminazione dei log di debug, ha introdotto considerevoli miglioramenti nella velocità di trasmissione, passando da 17.4 kbps a 76.2 kbps con gli stessi parametri di connessione. Ciò suggerisce che un ulteriore snellimento del codice possa incrementare ulteriormente il valore del data rate. Probabile che adottare la stessa strategia anche sul codice del Controller possa incrementare ulteriormente il data rate (restituendo inoltre una misura più precisa).

A conferma di ciò, anche nella repository Github contenente i sorgenti per MCU, forniti dal team del NGNI Lab [14, 15] e da cui si è partiti, viene specificato come i log di sistema rallentano il data rate di "SenseBack" [11].

Si potrebbe inoltre pensare di migrare il software sviluppato utilizzando un SDK più recente della Nordic, il che potrebbe apportare migliorie legate ad un'ottimizzazione delle librerie dell'SDK.

Per quanto riguarda invece l'FPGA, resta da implementare il FIFO bidirezionale, posto tra le due interfacce SPI. La logica è già molto semplificata, in quanto fa solo da buffer bidirezionale, tuttavia, per massimizzare la velocità di trasmissione, si

potrebbe pensare ad un cambio di direzione, sviluppando una logica totalmente combinatoria. Questo tuttavia, potrebbe complicare di molto il design, considerando che l'FPGA, nel sistema finale, dovrà comunicare con due master indipendenti, il Microcontrollore e l'ASIC, e con due SPI Clock differenti. Per questo motivo si è optato, in questa tesi, per una logica sincrona, che permettesse di sincronizzare i segnali in ingresso da ambo i lati evitando problemi di metastabilità.

Bibliografia

- [1] Ankur Gupta, Nikolaos Vardalakis e Fabien B Wagner. «Neuroprosthetics: from sensorimotor to cognitive disorders». In: *Nature Communications biology* 6.1 (2023), pp. 14–14. DOI: 10.1038/s42003-022-04390-w (cit. a p. 1).
- [2] Gurgen Soghoyan, Mikhail Sintsov, Artur Biktimirov, Ilya Chekh e Mikhail Lebedev. «Peripheral nerve stimulation for tactile feedback and phantom limb pain suppression». In: *2022 Fourth International Conference Neurotechnologies and Neurointerfaces (CNN)*. 2022, pp. 162–164. DOI: 10.1109/CNN56452.2022.9912538 (cit. alle pp. 1, 2).
- [3] Scott Stanslaski et al. «A Chronically Implantable Neural Coprocessor for Investigating the Treatment of Neurological Disorders». In: *IEEE Transactions on Biomedical Circuits and Systems* 12.6 (2018), pp. 1230–1245. DOI: 10.1109/TBCAS.2018.2880148 (cit. a p. 1).
- [4] P. S. Olofsson e C. Bouton. «Bioelectronic medicine: an unexpected path to new therapies». In: *Journal of Internal Medicine* 286.3 (2019), pp. 237–239. DOI: <https://doi.org/10.1111/joim.12967> (cit. a p. 1).
- [5] Ria Ghosh e John H. L. Hansen. «Bilateral Cochlear Implant Processing of Coding Strategies With CCi-MOBILE, an Open-Source Research Platform». In: *IEEE/ACM Transactions on Audio, Speech, and Language Processing* 31 (2023), pp. 1839–1850. DOI: 10.1109/TASLP.2023.3267608 (cit. a p. 1).
- [6] Zachary M. Smith, Wendy S. Parkinson e Christopher J. Long. «Multipolar current focusing increases spectral resolution in cochlear implants». In: *2013 35th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*. 2013, pp. 2796–2799. DOI: 10.1109/EMBC.2013.6610121 (cit. a p. 1).
- [7] Davide Brunelli, Elisabetta Farella, Davide Giovanelli, Bojan Milosevic e Ivan Minakov. «Design Considerations for Wireless Acquisition of Multichannel sEMG Signals in Prosthetic Hand Control». In: *IEEE Sensors Journal* 16.23 (2016), pp. 8338–8347. DOI: 10.1109/JSEN.2016.2596712 (cit. alle pp. 1, 5, 12).

-
- [8] Amy Blank, Allison M. Okamura e Katherine J. Kuchenbecker. «Identifying the Role of Proprioception in Upper-Limb Prosthesis Control: Studies on Targeted Motion». In: *ACM Trans. Appl. Percept.* 7.3 (2008). DOI: 10.1145/1773965.1773966 (cit. alle pp. 1, 2).
- [9] Ziliang Zhou, Yicheng Yang, Jinbiao Liu, Jia Zeng, Xiaoxin Wang e Honghai Liu. «Electrotactile Perception Properties and Its Applications: A Review». In: *IEEE Transactions on Haptics* 15.3 (2022), pp. 464–478. DOI: 10.1109/TOH.2022.3170723 (cit. a p. 2).
- [10] Dustin J. Tyler. «Restoring the human touch: Prosthetics imbued with haptics give their wearers fine motor control and a sense of connection». In: *IEEE Spectrum* 53.5 (2016), pp. 28–33. DOI: 10.1109/MSPEC.2016.7459116 (cit. alle pp. 2, 6, 7).
- [11] Ian Williams, Emma Brunton, Adrien Rapeaux, Yan Liu, Song Luan, Kianoush Nazarpour e Timothy G. Constandinou. «SenseBack - An Implantable System for Bidirectional Neural Interfacing». In: *IEEE Transactions on Biomedical Circuits and Systems* 14.5 (2020), pp. 1079–1087. DOI: 10.1109/TBCAS.2020.3022839 (cit. alle pp. 2, 3, 8–11, 16, 17, 19, 23, 24, 93).
- [12] Byunghun Lee, Yaoyao Jia, S. Abdollah Mirbozorgi, Mark Connolly, Xinyuan Tong, Zhaoping Zeng, Babak Mahmoudi e Maysam Ghovanloo. «An Inductively-Powered Wireless Neural Recording and Stimulation System for Freely-Behaving Animals». In: *IEEE Transactions on Biomedical Circuits and Systems* 13.2 (2019), pp. 413–424. DOI: 10.1109/TBCAS.2019.2891303 (cit. a p. 2).
- [13] Tommaso Campi, Silvano Cruciani, Francesca Maradei, Andrea Montalto, Francesco Musumeci e Mauro Feliziani. «Wireless Powering of Next-Generation Left Ventricular Assist Devices (LVADs) Without Percutaneous Cable Driveline». In: *IEEE Transactions on Microwave Theory and Techniques* 68.9 (2020), pp. 3969–3977. DOI: 10.1109/TMTT.2020.2992462 (cit. alle pp. 2, 11).
- [14] Imperial Collage (NGNI Lab). *Repository Github contenente il sorgente per MCU di Senseback Implant*. URL: <https://github.com/ilw/SenseBackImplantV2> (cit. alle pp. 2, 3, 8, 23, 24, 91, 93).
- [15] Imperial Collage (NGNI Lab). *Repository Github contenente il sorgente per MCU di Senseback Controller*. URL: <https://github.com/ilw/SenseBackControllerV2> (cit. alle pp. 2, 3, 8, 23, 91, 93).
- [16] Albert Mudry e Mara Mills. «The Early History of the Cochlear Implant: A Retrospective». In: *JAMA Otolaryngology–Head & Neck Surgery* 139.5 (mag. 2013), pp. 446–453. DOI: 10.1001/jamaoto.2013.293 (cit. a p. 5).

- [17] Claudio Castellini e Patrick van der Smagt. «Surface EMG in advanced hand prosthetics». In: *Biological Cybernetics* 100.1 (2009), pp. 35–47. DOI: 10.1007/s00422-008-0278-1 (cit. alle pp. 5, 6).
- [18] Song Luan, Ian Williams, Michal Maslik, Yan Liu, Felipe De Carvalho, Andrew Jackson, Rodrigo Quian Quiroga e Timothy G Constandinou. «Compact standalone platform for neural recording with real-time spike sorting and data logging». In: *Journal of Neural Engineering* 15.4 (2018), p. 046014. DOI: 10.1088/1741-2552/aabc23 (cit. alle pp. 7, 8).
- [19] Mario Collotta, Giovanni Pau, Timothy Talty e Ozan K. Tonguz. «Bluetooth 5: A Concrete Step Forward toward the IoT». In: *IEEE Communications Magazine* 56.7 (2018), pp. 125–131. DOI: 10.1109/MCOM.2018.1700053 (cit. alle pp. 12, 13).
- [20] Muhyi Bin Yaakop, Izwan Arief Abd Malik, Zubir bin Suboh, Aizat Faiz Ramli e Mohd Azlan Abu. «Bluetooth 5.0 throughput comparison for internet of thing usability a survey». In: *2017 International Conference on Engineering Technology and Technopreneurship (ICE2T)*. 2017, pp. 1–6. DOI: 10.1109/ICE2T.2017.8215995 (cit. alle pp. 12, 13).
- [21] Ashish Derhgawen. *Maximizing BLE Throughput Part 4: Everything You Need to Know*. Disponibile on-line. 2020. URL: <https://punchthrough.com/ble-throughput-part-4/> (cit. alle pp. 12, 14).
- [22] MOHAMMAD AFANEH. *Bluetooth 5 speed: How to achieve maximum throughput for your BLE application*. Disponibile on-line. 2023. URL: <https://novelbits.io/bluetooth-5-speed-maximum-throughput/> (cit. alle pp. 12–14).
- [23] Nordic Semiconductor. *S132 SoftDevice Specification*. Disponibile on-line. 2019. URL: https://infocenter.nordicsemi.com/index.jsp?topic=%2Fsds_s140%2FSDS%2Fs1xx%2Fble_data_throughput%2Fble_data_throughput.html (cit. a p. 15).
- [24] Nordic Semiconductor. *nRF52832. Versatile Bluetooth 5.4 SoC supporting Bluetooth Low Energy, Bluetooth mesh and NFC*. Disponibile on-line. URL: <https://www.nordicsemi.com/products/nrf52832> (cit. a p. 16).
- [25] Bryan Hsieh, Edward C. Harding, William Wisden, Nicholas P. Franks e Timothy G. Constandinou. «A Miniature Neural Recording Device to Investigate Sleep and Temperature Regulation in Mice». In: *2019 IEEE Biomedical Circuits and Systems Conference (BioCAS)*. 2019, pp. 1–4. DOI: 10.1109/BIOCAS.2019.8918722 (cit. a p. 16).

- [26] Jiaxin Lei, Shimeng Wang, Weining Li, Deng Luo, Xiaoyan Ma, Dandan Hui, Zhe Zhao, Xiong Zhong e Milin Zhang. «Design of a Multi-Mode Animal Behavior Analysis System with Dual-View Video and Wireless Bio-Potential Acquisition». In: *2022 IEEE International Symposium on Circuits and Systems (ISCAS)*. 2022, pp. 2705–2709. DOI: 10.1109/ISCAS48785.2022.9937952 (cit. a p. 16).
- [27] Nordic Semiconductor. *nRF52 DK. Bluetooth Low Energy and Bluetooth mesh development kit for the nRF52810 and nRF52832 SoCs*. Disponibile on-line. URL: <https://www.nordicsemi.com/Products/Development-hardware/nRF52-DK> (cit. a p. 16).
- [28] Lattice Semiconductor. *iCE 40 Ultra (iCE5LP4K). Industry-Leading Small Footprint, Low Power FPGA for High Volume Applications*. Disponibile on-line. URL: <https://www.latticesemi.com/Products/FPGAandCPLD/iCE40Ultra> (cit. alle pp. 17, 21, 60, 65).
- [29] Lattice Semiconductor. *iCE40 Ultra Breakout Board*. Disponibile on-line. URL: <https://www.latticesemi.com/products/developmentboardsandkits/ice40ultrabreakoutboard> (cit. a p. 17).
- [30] Ian Williams, Adrien Rapeaux, Yan Liu, Song Luan e Timothy G. Constandinou. «A 32-ch. bidirectional neural/EMG interface with on-chip spike detection for sensorimotor feedback». In: *2016 IEEE Biomedical Circuits and Systems Conference (BioCAS)*. 2016, pp. 528–531. DOI: 10.1109/BioCAS.2016.7833848 (cit. alle pp. 18, 19).
- [31] Ian Williams, Adrien Rapeaux, Jack Pearson, Kianoush Nazarpour, Emma Brunton, Song Luan, Yan Liu e Timothy G. Constandinou. «SenseBack – Implant considerations for an implantable neural stimulation and recording device». In: (2019), pp. 1–4. DOI: 10.1109/BIOCAS.2019.8919046 (cit. a p. 18).
- [32] Yan Liu, Song Luan, Ian Williams, Adrien Rapeaux e Timothy G. Constandinou. «A 64-Channel Versatile Neural Recording SoC With Activity-Dependent Data Throughput». In: *IEEE Transactions on Biomedical Circuits and Systems* 11.6 (2017), pp. 1344–1355. DOI: 10.1109/TBCAS.2017.2759339 (cit. a p. 19).
- [33] Nordic Semiconductor. *nRF52840 Dongle. Designed for nRF Connect for Desktop*. Disponibile on-line. URL: <https://www.nordicsemi.com/Products/Development-hardware/nRF52840-Dongle> (cit. a p. 19).
- [34] Nordic Semiconductor. *nRF Sniffer for Bluetooth LE. Bluetooth LE packet sniffer and learning tool*. Disponibile on-line. URL: <https://www.nordicsemi.com/Products/Development-tools/nrf-sniffer-for-bluetooth-le> (cit. alle pp. 19, 21, 36, 82).

- [35] SEGGER. *Segger Embedded Studio*. Disponibile on-line. URL: <https://www.segger.com/downloads/embedded-studio/> (cit. a p. 20).
- [36] Nordic Semiconductor. *nRF5 SDK v15.2.0*. Disponibile on-line. URL: <https://infocenter.nordicsemi.com/index.jsp?topic=%2Fcom.nordic.infocenter.sdk5.v15.2.0%2Findex.html> (cit. alle pp. 20, 24, 42).
- [37] Nordic Semiconductor. *nRF Connect for Desktop*. Disponibile on-line. URL: <https://www.nordicsemi.com/Products/Development-tools/nrf-connect-for-desktop> (cit. a p. 20).
- [38] *Wireshark*. Disponibile on-line. URL: <https://www.wireshark.org/download.html> (cit. a p. 21).
- [39] MathWorks. *MATLAB*. Disponibile on-line. URL: https://it.mathworks.com/products/matlab.html?s_tid=hp_products_matlab (cit. a p. 21).
- [40] Lattice Semiconductor. *iCEcube2 Design Software*. Disponibile on-line. URL: <https://www.latticesemi.com/iCEcube2> (cit. a p. 21).
- [41] Lattice Semiconductor. *Lattice Diamond Programmer and Deployment Tool*. Disponibile on-line. URL: <https://www.latticesemi.com/en/Products/DesignSoftwareAndIP/ProgrammingAndConfigurationSw/Programmer> (cit. a p. 22).