POLITECNICO DI TORINO

Dipartimento di Elettronica e Telecomunicazioni

Corso di Laurea in Ingegneria Elettronica

Tesi di Laurea Magistrale

Progettazione RTL di una IP Digitale Riconfigurabile

Relatore

Prof. Maurizio MARTINA

Correlatore

Ing. Alberto BONANNO (Eggtronic)

Candidato

Giacomo CATENA

8/4/2024

Indice

Elenco delle tabelle

Elenco delle figure

Acronimi

AMBA

Advanced Microcontroller Bus Architecture

APB

Advanced Peripheral Bus

ASIC

Application Specific Integrated Circuit

CSR

Control and Status Register

DMA

Direct Memory Access

DFT

Design For Test

DSP

Digital Signal Processing

FFT

Fast Fourier Transform

FSM

Finite State Machine

HDL

Hardware Description Language

IC

Integrated Circuit

IP

Intellectual Property

I2C

Inter-Integrated Circuit

LSB

Least Significant Bit

MSB

Most Significant Bit

RTL

Register Transfer Level

SoC

System on Chip

SPI

Serial Peripheral Interface

UART

Universal Asynchronous Receiver-Transmitter

UVM

Universal Verification Methodology

Capitolo 1

Introduzione

La tesi qui proposta è stata realizzata in collaborazione con **Eggtronic S.p.A.**, azienda specializzata nella progettazione di convertitori di potenza ad alte prestazioni, sia in versione cablata che wireless. Da alcuni anni ha avviato lo sviluppo del proprio micro-controllore **ASIC** portato avanti dal **reparto IC** che si occupa dello studio e dello sviluppo di circuiti integrati per applicazioni in ambito industriale (elettronica di potenza, automazione, automotive). Fondata a Modena nel 2012, nella quale è anche ubicato il Quartier Generale e reparto tecnico di progettazione, possiede riferimenti in diverse parti del mondo come Nord America e Sud-Est asiatico. **EPIC (Eggtronic Power Integrated Controller)** è la famiglia di controllori mixed-signal, basati su un'architettura RISC-V a basso consumo che va da una soluzione a basso costo di 8 bit fino ad una ad alte prestazioni di 32 bit, grazie alle periferiche digitali e analogiche di cui questi chip sono dotati. Lo scopo di questo lavoro di tesi è quello di progettare una **IP digitale riconfigurabile** che sia in grado di supportare tre tipi diversi di protocolli di comunicazione seriale tra cui **UART, SPI e I^2C** usati dal micro per comunicare con tutte le sue periferiche. L'obiettivo finale è quindi quello di risparmiare una considerevole area di silicio e mantenere allo stesso tempo un alto grado id flessibilità; in questo modo, invece di allocare in un chip tre IP differenti ciascuna della quali in grado di supportare un singolo sistema di comunicazione seriale, avremo una sola interfaccia la quale potrà essere configurata dall'utente in base alle necessità della sua applicazione attraverso il protocollo standard **APB**, rendendo quindi il sistema più flessibile, portabile, compatto e versatile. Questa IP farà poi eventualmente parte della gamma di periferiche di cui il chip di futura generazione **EPIC 3.0** potrà fare affidamento. Il progetto verrà sviluppato tramite linguaggio **HDL** (Hardware Description Language) nello specifico **SystemVerilog** e verrà simulato e verificato tramite **Cadence Xcelium**. Una volta testata correttamente, ci occuperemo di

sintetizzare la nostra IP con **Cadence Genus** per approfondire e valutare gli aspetti inerenti al consumo di potenza e all'area occupata e facendo il confronto con delle IP che implementano singolarmente i tre protocolli di interesse, arrivando così alle dovute conclusioni.

1.1 Obiettivi

Nella presente tesi, verranno coperti i seguenti obiettivi:

- Analisi dello **stato dell'arte delle le architetture riconfigurabili** per qualunque applicazione.
- Studio dettagliato delle specifiche dei tre protocolli di interesse (I^2C, *SPI* e *UART*), dell'hardware necessario e delle relative **macchine a stati finiti** per il controllo.
- Breve introduzione e comprensione del funzionamento del **bus APB**.
- Ottimizzazione e condivisione della parte hardware richiesta.
- Sviluppo di una **macchina a stati condivisa** che supporti tutti e tre gli Standard seriali.
- Stesura di una **Register-Map** contenente i registri CSR e di dato.
- **Sviluppo codice RTL, simulazione e verifica.**
- **Sintesi del progetto: stima di area e di potenza**

1.2 Organizzazione dei capitoli

La tesi è costituita da sette capitoli organizzati nella seguente maniera:

- **Capitolo 1, Introduzione**.
- **Capitolo 2, Architetture riconfigurabili**: in questo capitolo andremo a cercare nella letteratura scientifica tutto ciò che può essere di nostro interesse per perseguire il nostro scopo finale. Nello specifico analizzeremo progetti passati, di qualsiasi applicazione, nei quali è stata portata avanti una soluzione dotata di riconfigurabilità. Cercheremo di comprendere la metodologia di progetto seguita, le scelte che sono state prese e di investigare eventuali architetture proposte, per poi terminare con le conclusioni sui numeri ottenuti dalle sintesi.
- **Capito 3, Protocolli e Bus**: proseguiremo nell'approfondire il tema riguardante protocolli di comunicazione seriale in generale, per poi addentrarci nelle caratteristiche specifiche dei tre Standard seriali che vogliamo supportare nella nostra IP. Forniremo una panoramica completa che descriva il principio di funzionamento e i punti di criticità nei vari tipi di comunicazione seriale, soprattutto per quel che riguarda il lato gestionale e quindi la **Macchina a Stati** necessaria per il controllo. Questa fase servirà per capire laddove sarà

possibile usare le medesime risorse hardware per l'implementazione al fine di ottemperare all'obiettivo finale di questo progetto, ovvero massimizzare l'ottimizzazione e il risparmio di area di silicio nel chip. Verso la fine del capitolo daremo anche una piccola introduzione del protocollo APB e andremo a vedere qualche esempio di trasferimento sul bus.

- **Capitolo 4, FSM, condivisione hardware e Register-Map**: compresi i principi di funzionamento dei tre protocolli seriali, qui inizieremo il processo di condivisione dei blocchi funzionali e delle macchine a stati. Mostreremo i componenti hardware che allocheremo e il modo in cui opereranno in funzione dello Standard seriale selezionato. Degna di nota sarà la parte di interfacciamento APB, ovvero la parte in cui definiamo accuratamente il tipo di parametri attraverso i quali andremo a configurare il sistema ma anche il registro di stato, tipi di **flag** da inserire e la gestione degli **interrupt**, definendo cosi una **Register-Map** della IP. Nel far ciò, utilizzeremo un programma chiamato **Kactus** che ci permette di descrivere e di visualizzare comodamente e con facilità tutti i registri di configurazione (**CSR**) della Register-Map, definendo nomi, indirizzi, modalità di accesso e i campi (bit-field) che li compongono.

- **Capitolo 5, Simulazione RTL**: una volta sviluppato il relativo codice RTL, in questo capitolo ci occuperemo di simulare e verificare il corretto funzionamento della nostra IP con Cadence Xcelium. In questa fase, usufruiremo di un ambiente di verifica **UVM** già sviluppato dal team di Eggtronic per comunicare con la nostra e verificare così di rispettare le regole di ciascun protocollo seriale in maniera sicura. In particolare potremo sfruttare gli **agenti** dell'ambiente UVM per simulare la comunicazione tra due sistemi (uno dei quali sarà proprio la nostra di IP) con ciascuno dei tre protocolli di nostro interesse. Anche la Register-Map sarà testata in questo ambiente, nel quale è predisposto un tipo di test automatico e specifico sui CSR che compara il **file.xml** (generato da Kactus) con l'RTL che descrive l'interfaccia APB, confrontando le operazioni di lettura e scrittura su ciascun registro verificando che le operazioni effettuate sul codice RTL combacino con il file.xml.

- **Capitolo 6, Sintesi**: verificato correttamente il funzionamento nelle diverse modalità, si passa infine alla sintesi logica con Cadence Genus per stimare l'area allocata e il consumo di potenza. Prima però daremo una panoramica del diagramma a blocchi per mostrare la gerarchia interna con i rispettivi segnali di clock e l'inserimento manuale delle celle di **clock gating**. I valori generati dalla sintesi li andremo poi a confrontare con quelli ottenuti da sei differenti IP, ciascuna della quali supportante un singolo protocollo di comunicazione seriale e un singolo lato, valutando così gli effettivi guadagni ottenuti con la soluzione riconfigurabile rispetto alle altre.

- **Capitolo 7, Conclusioni**: per concludere l'elaborato, ci dedicheremo all'analisi del lavoro svolto e degli obiettivi raggiunti, con particolare enfasi al confronto finale tra la nostra IP e quelle Standard dal punto di vista delle celle allocate e del consumo di potenza per comprendere concretamente i reali vantaggi (e svantaggi) di una soluzione riconfigurabile. Daremo anche un breve spunto per possibili migliorie, soprattutto in senno a un possibile futuro in cui questa IP venga integrata all'interno del sistema di EPIC.

Capitolo 2

Architetture riconfigurabili

In questo capitolo analizzeremo lo stato dell'arte delle architetture riconfigurabili, cercando di estrapolare dalle letteratura scientifica a riguardo gli aspetti cruciali dello studio e della progettazione di esse, oltre che ai vantaggi e svantaggi riscontrati adoperando questa nuovo approccio rispetto a quello Standard. In particolare, i due articoli presi come spunto trattano dell'uso di soluzioni riconfigurabili in due campi di applicazione differenti: **Interfacce Audio/Seriali** e **processori FFT**. Verso la fine, analizzeremo i risultati ottenuti per quel che riguarda l'area allocata e il consumo di potenza.

2.1 Reconfigurable Architecture for IP Peripherals

L'articolo [1] affronta il tema di come i **SoC (System On Chip)**, a seconda del campo di applicazione industriale, dispongano di un certo numero di interfacce seriali e audio per ottemperare a specifici compiti con diversi vincoli da rispettare, portando così alla creazione di varie tecniche di ottimizzazione del consumo di potenza e alla variazione del numero di Pin necessari. Inoltre, con l'avvento di nuove tecnologie e applicazioni sul mercato, la necessità di sviluppare nuovi SoC e migliorare quelli già esistenti è aumentata vertiginosamente. Questa è una nuova sfida di oggi che le aziende del settore devono affrontare a causa di una sempre più ridotta finestra di mercato, rendendo più difficoltoso stare al passo con le nuove specifiche dei clienti sempre più complesse e stringenti. Nasce così l'esigenza di trovare un nuovo approccio per ottimizzare il processo di sviluppo, senza incorrere troppo in un aumento delle spese e del tempo necessario per lanciare un nuovo prodotto sul mercato.

Una soluzione proposta è quella di realizzare progetti con componenti altamente ottimizzati e parametrizzati, rendendoli così facilmente portabili in futuri propositi. Viene dunque mostrata un'**architettura riconfigurabile per periferiche IP** e una metodologia di sviluppo per la sua realizzazione per poi procedere allo studio una possibile implementazione ad alto livello. Infine vengono analizzati i risultati ottenuti.

2.1.1 Metodologia

La metodologia qui proposta espone un flusso di progetto nel quale vari protocolli di comunicazione seriale, audio e wireless possono essere sistematicamente implementati in un singolo blocco hardware riconfigurabile. Questo metodo si concentra sulla classificazione del protocollo, i requisiti delle porte e dei PAD di I/O e delle loro proprietà, per estrarre le informazione più importanti per il corretto funzionamento della IP. Di seguito è riportata una mappa concettuale del flusso:

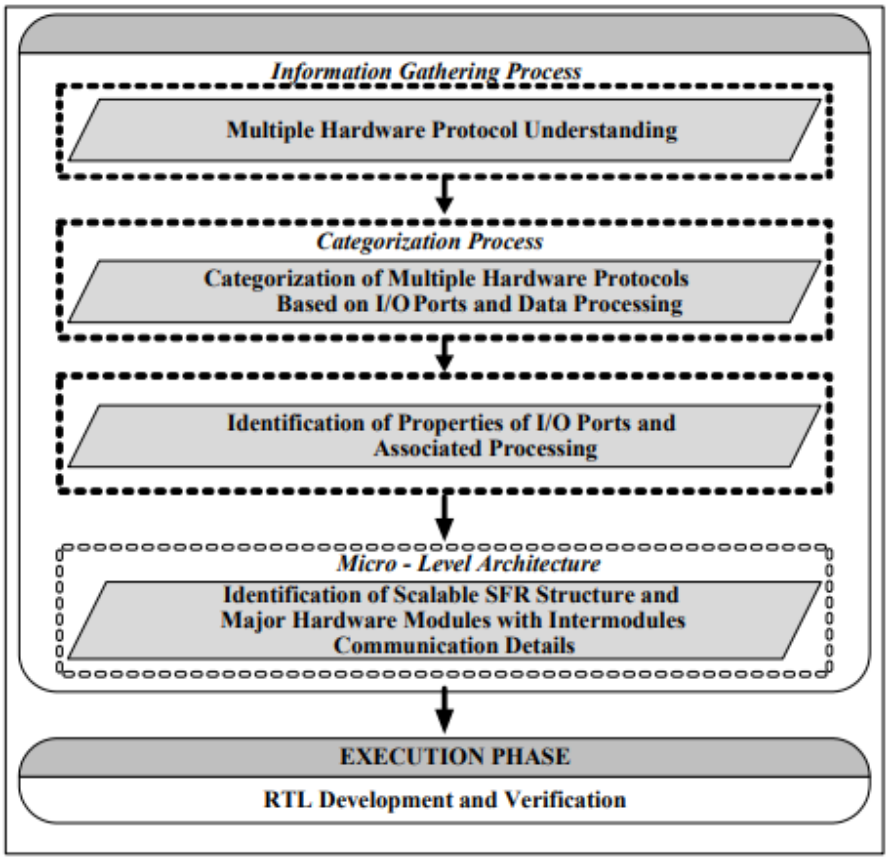


Figura 2.1: Flusso e Metodologia proposta per la realizzazione della IP riconfigurabile

In Figura 2.1 vediamo come al primo posto venga menzionata la comprensione dell'hardware di protocolli differenti. Questo passaggio è cruciale perchè è proprio qui che il progettista inizia a fare una prima sintesi a livello hardware, distinguendo caratteristiche comuni dei protocolli coinvolti, avanzando quindi una prima fase di ottimizzazione. Si passa poi alle categorizzazione dei protocolli basata sulle porte di I/O e all'elaborazione dei dati. In questa fase, vengono analizzate le distinzioni per quel che riguarda caratteristiche come Baud-rate, formattazione dei dati, numero di porte di I/O richieste, segnali di hand-shake e cosi via. Conclusa questa prima parte, si passa al livello micro-architetturale identificando i blocchi principali e intermedi in ottica di scalabilità e riconfigurabilità. Infine c'è la fase di esecuzione nella quale viene sviluppato lo script RTL e il suo relativo testbench per la verifica.

2.1.2 Architettura hardware proposta

Qui verrà discussa l'architettura ad alto livello realizzata seguendo la metodologia appena esposta. La tecnica del riutilizzo dell'hardware è implementata tramite l'uso di un **blocco di registri configurabili** che possono essere programmati a secondo della sequenza stabilita dalle specifiche del protocollo come formato del frame, struttura pin ecc. Nella figura seguente è mostrato lo schema a blocchi ad alto livello:

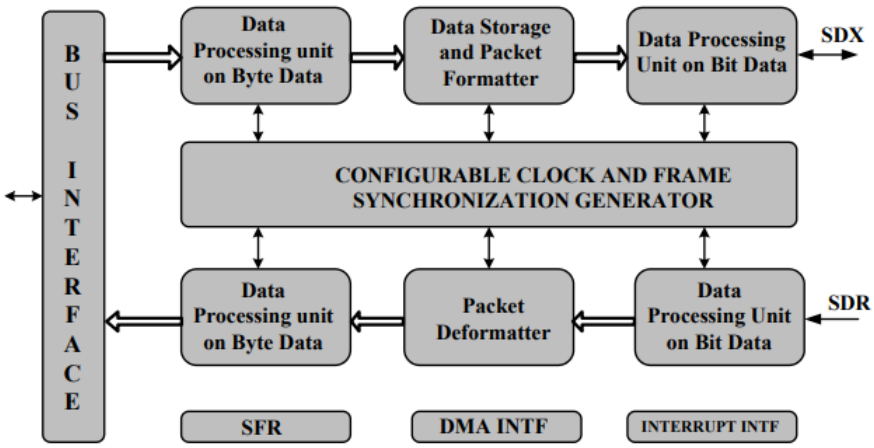


Figura 2.2: Diagramma a blocchi ad alto livello dell'architettura proposta

Blocco Dati

La prima cosa che possiamo notare dalla Figura 2.2 è la presenza di un **Blocco Dati** con due percorsi distinti per la trasmissione (TX) e per la ricezione (RX), come era ovvio aspettarci. Analizzando più in profondità, notiamo che alcune

unità operano chiaramente al livello di bit (dovendo trattare con protocolli seriali) mentre altre, una volta formattato/deformattato il pacchetto, elaborano il dato a livello di Byte per la concatenazione della parola (word) ricevuta.

Blocco di Controllo

Il Blocco di Controllo è responsabile della generazione del clock secondo il Baud-Rate configurato dall'utente e dei segnali di sincronizzazione del frame.

SFR: Special Function Register

Questo è il blocco dei registri precedentemente citato grazie al quale possiamo riconfigurare tutto il nostro hardware in tempo reale, eseguendo funzioni importanti come il controllo del clock, controllo periferica seriale, controllo lunghezza-word, registri di stato ... È da questo modulo che tutti gli altri blocchi ricevono i parametri di configurazione forniti dall'utente in una prima fase antecedente all'esecuzione vera e propria di una operazione con il mondo esterno.

Interfaccia esterna

Questa interfaccia include un bus AMBA-compliant per la comunicazione con l'utente, ovvero lo scambio di dati da trasmettere/ricevere, indirizzi, interrupts DMA e altri tipi di segnali.

2.1.3 Risultati e conclusioni

Per concludere, è stato fatto un paragone tra la soluzione riconfigurabile e una IP nella quale i protocolli industriali sono stati implementati separatamente dal punto di vista dell'utilizzo di Pin e di porte logiche allocate. I risultati sono mostrati nella seguente tabella:

IP	Pin Count	Gate Count
Multiple Serial and Audio Interface IP	6	34722.7
Percentage Saving	71.43%	43.8%

Tabella 2.1: Risparmio totale di Pin e porte logiche

Come possiamo osservare dalla Tabella 2.1, il risparmio di Pin e porte logiche è significativo. Tuttavia, l'articolo menziona anche uno scenario limite in cui

non venissero usati tutti i protocolli supportati dalla IP. Questo porterebbe a un **consumo di potenza maggiore rispetto ad una IP che implementa un singolo protocollo**. Chiaramente, la situazione appena citata è un caso in cui verrebbe fatto un utilizzo di hardware riconfigurabile poco efficiente.

2.2 A Novel Low-Power Reconfigurable FFT Processor

In questo scritto [2] viene proposta una soluzione riconfigurabile per una applicazione DSP: invece di avere un processore ASIC FFT di tipo convenzionale, l'algoritmo viene implementato con un'architettura configurabile che va da 16 punti fino a 1024 punti. Non entreremo nei dettagli dei meccanismi di ottimizzazione e del tipo di flessibilità ottenuta non essendo questa il genere di applicazione strettamente correlata con l'oggetto finale di questa tesi. Cercheremo solo di comprendere le motivazioni di questa scelta e il paragone in termini di prestazioni e consumi con altre soluzioni.

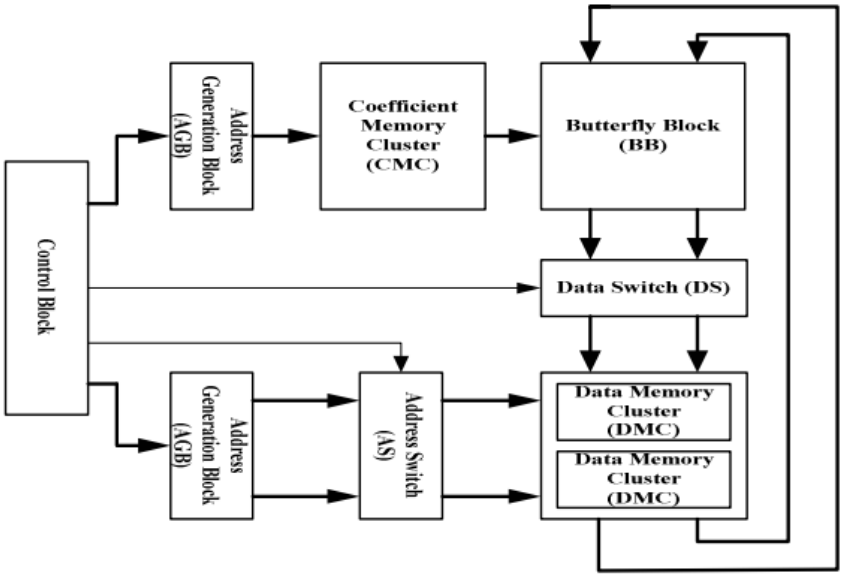


Figura 2.3: Architettura del processore FFT riconfigurabile

Nella Figura 2.3 è raffigurato lo schema a blocchi specificamente mirato ad eseguire l'algoritmo della trasformata di Fourier veloce. Infatti, una soluzione di tipo general-purpose avrebbe portato a delle penalità a livello di prestazione; invece, una soluzione function-specific ci permette di bilanciare in maniere ottimale flessibilità

e consumo di potenza. Per quanto riguarda la scelta del numero di punti della FFT, questa è legata alla norma operativa; rendere quindi la dimensione della FFT configurabile è un grado di flessibilità che possiamo sfruttare in funzione dell'ambiente di operazione.

Componenti

Un processore di questo tipo richiede molte operazioni di tipo butterfly, come possiamo osservare dalla presenza del "Butterfly Block (BB)" sempre in Figura 2.3, i coefficienti sono forniti da un "Coefficient Memory Cluster (CMC)" mentre le uscite del BB sono convogliate da un "Data Switch (DS)" nella giusta "Data Memory Cluster (DMC)". Gli indirizzi per le due memorie vengono generati da due "Address Generation Block" ed è proprio qui che entra in gioco la riconfigurabilità del sistema. Infatti, a differenza di un prodotto FFT a virgola fissa, questi blocchi posso generare indirizzi per diverse dimensione della FFT in base alla loro configurazione con un metodo efficiente proposto da Cohen (che non andremo però ad analizzare essendo questo un argomento che va oltre i nostri scopi).

2.2.1 Risultati

Il processore FFT è stato progettato tramite Verilog HDL e sintetizzato con Synopsis Design Compiler usando una libreria "UMC 0.18μm CMOS standard cell technology". I risultati riguardanti il consumo di potenza e l'area allocata sono mostrati nella seguente tabella:

FFT Size	16	32	64	128	256	512	1024
Power Consumption (mW)	4.7	7.9	8.3	13.0	26.1	49.7	81.6
Area (mm^2)	2.9	2.9	2.9	2.9	2.9	2.9	2.9

Tabella 2.2: Consumo di potenza e area occupata Reconfigurable FFT processor

È chiaro che all'aumentare della dimensione dell'algoritmo corrisponde un incremento della potenza dissipata; l'area è invece fissa trattandosi dello stesso oggetto che effettua diverse operazioni. Più interessante è invece il seguente istogramma dove è stato confrontato il consumo della soluzione proposta operante in tutte le modalità con quello di processori FFT non riconfigurabili:

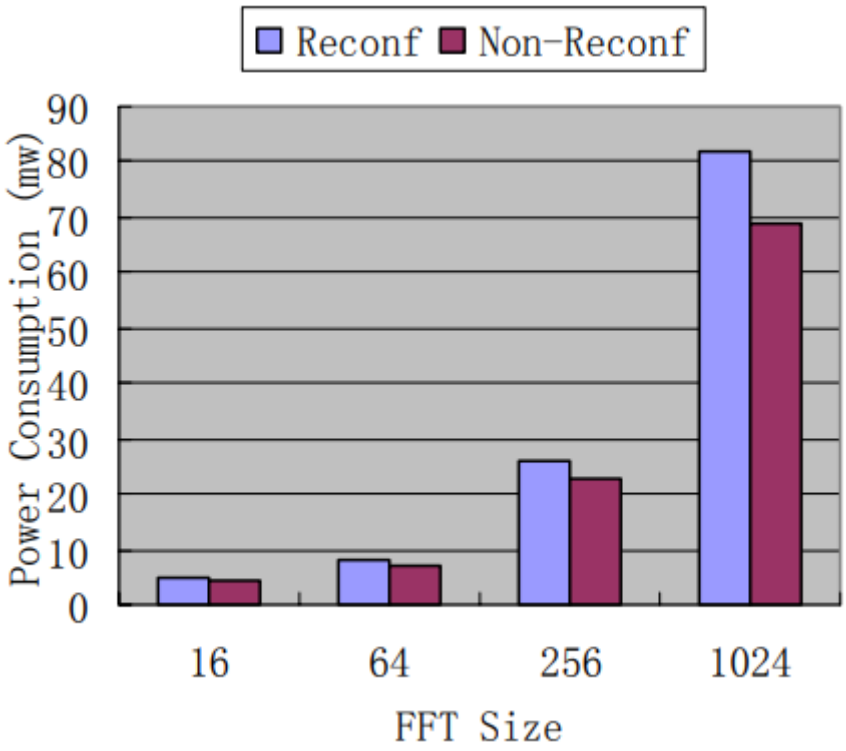


Figura 2.4: Consumi Non-reconf. VS Reconf.

Anche in questa applicazione riscontriamo un maggior dispendio energetico di circa il 12-19% rispetto all'oggetto non riconfigurabile. D'altro canto pero, questo processore può operare in diverse modalità di funzionamento; se volessimo coprire lo stesso numero di dimensioni dell'algoritmo senza ricorrere a soluzioni riconfigurabile significherebbe allocare quattro oggetti, ciascuno operante con un numero di punti diverso, con un occupazione dell'area di gran lunga maggiore.

Capitolo 3

Protocolli e Bus

Abbiamo visto dunque cosa porta i progettisti di micro-sistemi digitali a muoversi verso delle architetture capaci di riconfigurarsi a seconda della funzione che si vuole svolgere o delle nuove specifiche di progetto. Abbiamo analizzato una possibile metodologia di lavoro per l'implementazione di una IP di questo tipo insieme ai vantaggi e alle penalità che incorriamo.
In questo capitolo inizieremo ad affrontare più nel dettaglio le tematiche riguardanti i protocolli di comunicazione di interesse, cercando di comprendere in maniera approfondita l'idea alla base, i punti in comune e le relative differenze, dando quindi un quadro generale di ciascun Standard seriale in vista poi una più facile comprensione della realizzazione del nostro progetto finale. Iniziamo però dando una panoramica di che cosa sia una **trasmissione seriale** e il perchè sia spesso preferita rispetto alla sua contro partita parallela, per poi addentrarci più nei particolari dei tre che vogliamo implementare. Verso la fine, daremo anche una breve introduzione del bus e del protocollo APB che useremo per configurare la nostra IP; descriveremo i segnali di interfacciamento presenti e mostreremo anche due esempi di trasferimento sul bus, uno in lettura e uno in scrittura.

3.1 Trasmissione seriale

La trasmissione seriale è una modalità di interazione tra dispositivi digitali che richiedono uno scambio di dati. In essa, ciascun bit componente un pacchetto dati viene trasmesso da un dispositivo (chiamato **trasmettitore**) attraverso un singolo canale di comunicazione uno di seguito all'altro e verranno ricevuti sequenzialmente da un secondo sistema (chiamato **ricevitore**) nello stesso ordine in cui il mittente li ha disposti. Questo implica che sarà necessario definire diversi aspetti di carattere gestionale a seconda del protocollo e della velocità di trasmissione il che comporterà

per forza di cose un incremento della complessità architetturale riguardante la parte hardware necessaria a supportare tale comunicazione. Tuttavia, la modalità seriale è una delle più diffuse nel campo delle Telecomunicazioni dato che:

- richiede un minor numero fili riducendo notevolmente i costi.
- è più tollerante rispetto alle interferenze e agli errori di trasmissione.

Riportiamo un'immagine raffigurante le differente dal punto di vista strutturale tra una trasmissione seriale e una parallela:

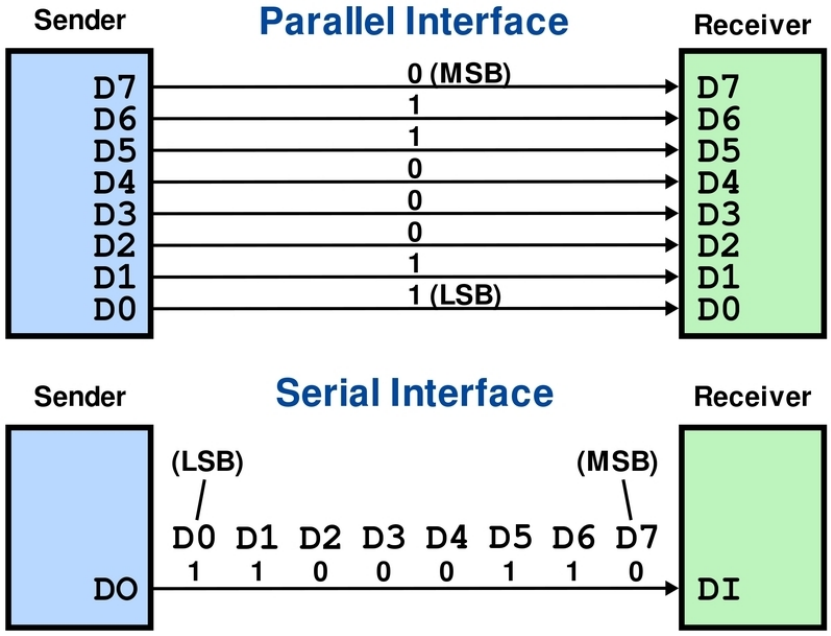


Figura 3.1: Trasmissione seriale e parallela

Le principali modalità in cui avviene una comunicazione seriale sono due: **modalità sincrona** e **asincrona**. Nella prima, tra cui rientrano gli Standard I^2C e *SPI*, è presente un altro filo nel quale scorre un segnale di temporizzazione periodico chiamato **Clock** necessario per la sincronizzazione di tutti i bit in sequenza definendo così anche il Baud-rate. Nella seconda invece, ricevitore e trasmettitore si sincronizzano usando i dati stessi; nello specifico, il trasmettitore per inizializzare una transizione invierà un bit di partenza (chiamato **START-BIT**) e per terminarla un altro bit di stop (**STOP-BIT**). In quest'ultima categoria rientra l'interfaccia *UART*.

3.2 Inter-Integrated-Circuit (I^2C)

Il bus I^2C (talvolta abbreviato *I2C* o *IIC*) è l'acronimo di **Inter-Integrated-Circuit** e come già accennato nella sezioni precedenti, è un protocollo seriale sincrono sviluppato dalla **Philips** [3] (oggi **NXP**) ed è utilizzato tra circuiti integrati di vario tipo. Al giorno d'oggi è uno Standard mondiale ed è presente nelle architetture di oltre mille aziende produttrici di circuiti integrati in vari ambiti applicativi che vanno da System Management Bus (SMBus), Power Management Bus (PMBus), Intelligent Platform Management Interface (IPMI) e tanti altri. Esso si basa sull'uso di due fili:

- Linea di dato seriale **SDA**.
- Linea del clock seriale **SCL**.

La situazione più frequente prevede la presenza di un singolo dispositivo chiamato ***Master*** (è possibile tuttavia avere anche una disposizione ***Multi-Master*** in sistemi più complessi) e diversi dispositivi detti ***Slave*** (**o periferiche**). A titolo di esempio riportiamo un'immagine di come questi sono collegati al bus:

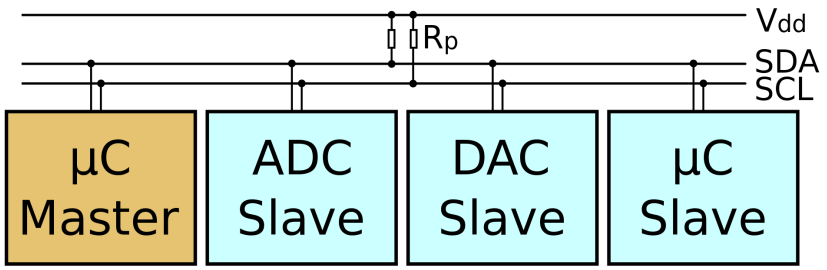


Figura 3.2: Bus I^2C con un singolo *Master*

Il *Master* si occuperà di generare il clock seriale SCL per tutte le periferiche collegate e solo lui ha la possibilità di inizializzare una transizione mentre tutti i dispositivi connessi possono ricevere e trasmettere dati. Possiamo notare dalla Figura 3.2 come siano presenti due **resistori di Pull-up** sulle due linee seriali. Infatti, sia SDA che SCL sono linee cosiddette **Open-Drain** [4]: quando un dispositivo vuole pilotare la linea forzerà su di essa un livello logico basso tramite il suo driver (Figura 3.3); se invece vuole trasmettere un 1 logico, rilascerà la linea e sarà il resistore a tenerla a livello alto (o a riportarla su). Questo comporta il vantaggio di non avere in nessun caso conflitti dato che nessuno potrà forzare un 1 su uno 0 logico; infatti è proprio grazie a questa proprietà del bus che possiamo usare un solo filo per trasmettere e ricevere dati, rendendo così SDA (e SCL) una linea **bidirezionale**. Questo è

uno aspetto molto rilevante e sarà qualcosa da tenere in considerazione in vista del nostro progetto.

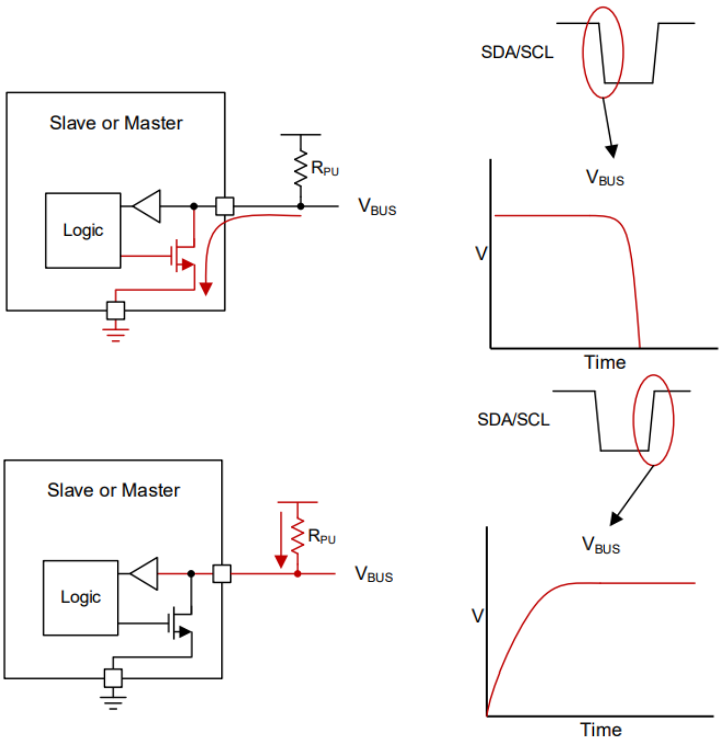


Figura 3.3: Scrittura di uno zero/uno logico in una connessione Open-Drain

Velocità di trasmissione

Con questo tipo di bus, si posso usare diverse frequenze di trasmissione. La modalità Standard prevede un trasferimento di **100 Kbit/s**, ma nulla impedisce di scendere a velocità inferiori. Ci sono poi altre modalità che raggiungono frequenze superiori come **Fast-Mode (400KHz)** e **Fast-Mode Plus (1MHz)** ma con alcune condizioni riguardanti la capacità della linea.

3.2.1 I^2C: Temporizzazione dei segnali

Ogni dispositivo connesso al bus I^2C ha un indirizzo specifico di identificazione tipicamente a 7 bit (è stato ampliato fino a 10 bit per aumentare il numero di periferiche connesse). Il bus quando è libero (IDLE), si trova con entrambi i fili a livello logico alto. Tutte le transizioni sono sempre generate dal *Master* e iniziano con una condizione di START (S) e sono terminate da uno STOP (P). Il primo si verifica quando il *Master* forza la linea SDA a 0 e al tempo stesso SCL è fissa a 1,

viceversa per lo STOP dove rilascerà SDA (fronte di salita), come si può osservare nella Figura 3.4.

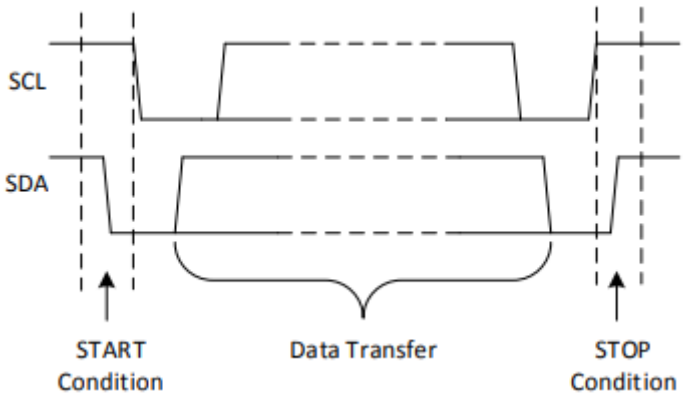


Figura 3.4: Condizioni di START e STOP

Segnali di Acknowledge (ACK) e Not-Acknowledge (NACK)

Una volta iniziata la transizione, il primo dato trasmesso sarà sempre l'indirizzo dello *Slave* con il quale si vuole comunicare, seguito da un bit comando R/W per indicare alla periferica che tipo di operazione si vuole eseguire. L'ordine seriale dei bit parte da quello più significativo fino al meno significativo (MSB-first).
Successivamente, il *Master* rilascerà la linea di dato per attendere una risposta dall'altra parte; se è presente una periferica a quell'indirizzo, essa prenderà il controllo di SDA forzando uno 0 per inviare un **segnale di riconoscimento (Acknowledgment ACK)**. Se invece non c'è nessuno *Slave* con quell'indirizzo, SDA rimarrà alta (**Not-Acknowledge NACK**) e il *Master* interromperà la comunicazione sul bus. Questo processo avviene dopo ogni Byte inviato e permette al ricevitore di segnalare al trasmettitore che la ricezione del Byte ha avuto successo e che quindi si può passare al prossimo.
Ci sono cinque possibili situazioni che portano alla generazione di un NACK:

- Nessun ricevitore è presente all'indirizzo trasmesso.
- Il ricevitore non può ricevere o trasmettere essendo occupato ad eseguire altre operazioni.
- Durante il trasferimento, il ricevitore non comprende i comandi inviati dal *Master*.
- Durante il trasferimento, il ricevitore non può più accogliere altri Byte.

- Il *Master* ricevente deve segnalare la fine di un trasferimento alla periferica trasmittente.

A questo punto il Master può abortire inviando uno STOP oppure re-iniziare una nuova comunicazione tramite un **RE-START** (REPEATED-START).

Validità del dato

Durante ogni ciclo di SCL, viene trasmesso un bit di dato su SDA. Il dato che viene scambiato può trattarsi di indirizzo della periferica, indirizzo del registro della periferica che vogliamo accedere oppure un dato da scrivere o leggere dallo *Slave*. La linea SDA deve però rispettare un certo tempismo:

- SDA può cambiare durante il periodo basso di SCL.
- SDA deve essere stabile durante il periodo alto di SCL.

Le uniche eccezioni a questo sono proprio le condizioni di START e STOP. Di seguito (Figura 3.5) è mostrato il diagramma temporale di un corretto invio di un Byte.

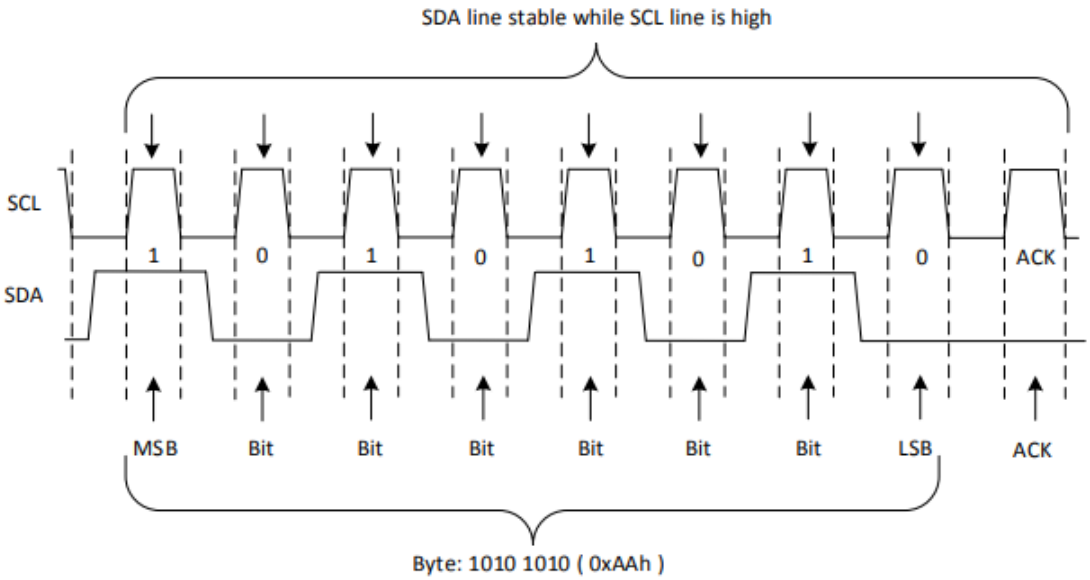


Figura 3.5: Esempio di trasferimento di un Byte

3.2.2 I^2C: Accesso ad una periferica

Come già anticipato in precedenza, quando accediamo ad uno Slave possiamo sia scrivere un dato all'interno di un suo registro oppure leggere un dato dallo stesso.

Le due operazioni però non sono simmetriche dal punto di vista del flusso di passi da seguire per completarle correttamente.

Scrittura

Una volta ricevuto il segnale di ACK dopo aver trasmesso sul bus l'indirizzo del dispositivo di interesse insieme al comando $R/W = 0$ (zero significa scrittura), proseguiremo con l'invio dell'indirizzo del registro che vogliamo scrivere. Dopo aver ricevuto il secondo ACK, passeremo all'invio del dato effettivo da trasferire alla periferica e una volta inviati tutti e otto i bit, al solito lo *Slave* ci farà sapere se ha ricevuto il dato correttamente con un terzo ACK. Se abbiamo terminato la transizione, il *Master* genererà uno STOP altrimenti si può proseguire con l'invio di un altro Byte (o anche di più). Per una migliore comprensione, riportiamo la completa sequenza di dati trasmessi (Figura 3.6) sulla linea SDA per trasferire un singolo Byte [4].

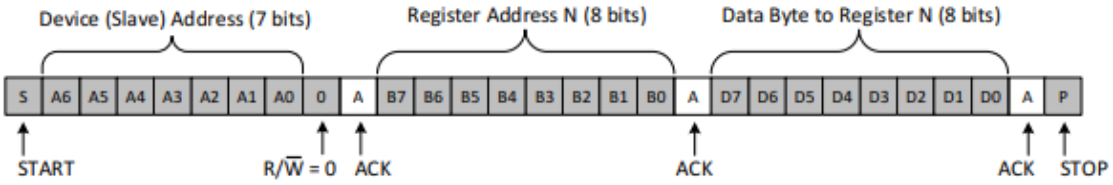


Figura 3.6: Esempio di scrittura

Osserviamo quindi come nel caso di una scrittura, il flusso di piccole operazioni da eseguire per portare a termina la transazione è abbastanza lineare e regolare.

Lettura

La situazione cambia radicalmente nel caso in cui volessimo effettuare una lettura ad una locazione di memoria di una periferica. I primi due passaggi sono analoghi al caso della scrittura ovvero: invio indirizzo *Slave* con comando $R/W = 0$ più invio indirizzo del registro che si vuole accedere. Nonostante il nostro scopo finale sia quello di leggere un dato, è necessario prima configurare il bus in modalità scrittura nella fase iniziale di indirizzamento per memorizzare all'interno della periferica l'indirizzo del registro di interesse. Fatto ciò, lo *Slave* comunicherà di aver ricevuto correttamente i dati con un ACK. A questo punto, il *Master* re-inizierà una transizione sul bus con un **REPEATED-START** e trasmetterà di nuovo l'indirizzo della medesima periferica ma questa volta con il comando $R/W=1$ per girare il bus in modalità lettura; solo ora lo *Slave* prenderà il controllo di SDA per inviare il dato contenuto nel registro indirizzato nella prima fase.

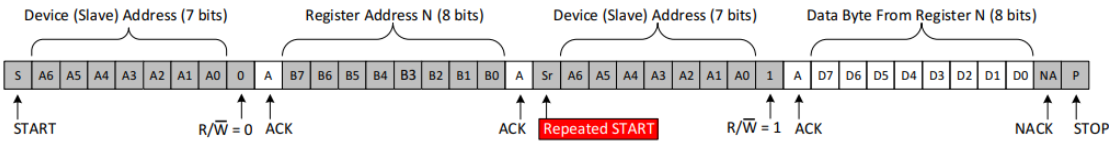


Figura 3.7: Esempio di lettura

Alla fine di ogni Byte, il *Master* può inviare un ACK per far sapere alla periferica che è pronto a ricevere un nuovo dato. In alternativa, una volta che il *Master* ha ricevuto il numero di Byte desiderato, trasmetterà un NACK per segnalare di voler terminare la comunicazione. Seguirà quindi una condizione di STOP, come mostrato in Figura 3.7 dove viene eseguita la lettura di un singolo Byte.

3.2.3 Macchina a Stati: Master

Le due operazioni di lettura e scrittura sono dunque asimmetriche e per leggere/-scrivere un singolo Byte, nel primo caso verrà complessivamente trasmesso sul bus un Byte in più per completare la transizione. Questo si riflette nel meccanismo gestionale del flusso delle operazioni in una **Macchina a Stati** lato *Master*. Per una più facile visualizzazione, abbiamo riportato qui di seguito un pallogramma semplificato raffigurante il flusso degli stati possibili di una ipotetica macchina a stati.

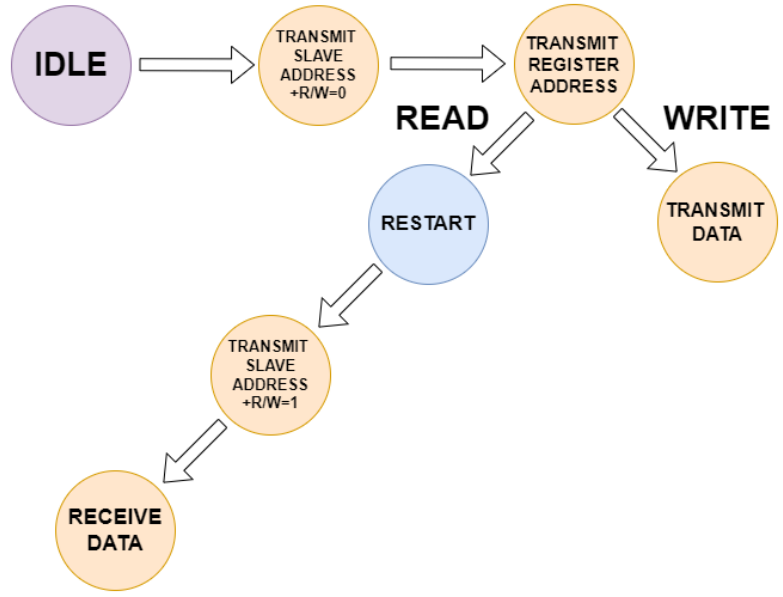


Figura 3.8: Pallogramma I^2C *Master*

Per ragioni di semplicità abbiamo omesso gli stati di START/STOP e di riconoscimento tra *Master* e *Slave*.

3.2.4 Caso Multi-Master

Degno di nota è il caso in cui nel bus fossero collegati più dispositivi *Master* che hanno la possibilità di far partire una transizione. Uno scenario di questo tipo è sempre più diffuso è può essere visto anche come un punto di forza per il protocollo I^2C, il quale può supportare questa modalità operativa grazie ai suoi nodi Open-Drain (a differenza del caso *SPI* dove solo un dispositivo alla volta può essere configurato come *Master*). Tuttavia, il fatto di avere più dispositivi capaci di iniziare una comunicazione sul bus, implica di dover assicurare un certo meccanismo di **sincronizzazione del clock seriale** e un **arbitraggio**; entrambi avvengono contemporaneamente e anche qui la connessione Open-Drain rende questo processo possibile.

Sincronizzazione del clock

Per sincronizzazione del clock [5] si intende il procedimento mediante il quale si permette a due dispositivi Master con velocità di funzionamento differenti di operare senza incorrere in una perdita di dati. Un esempio è riportato nell'immagine seguente:

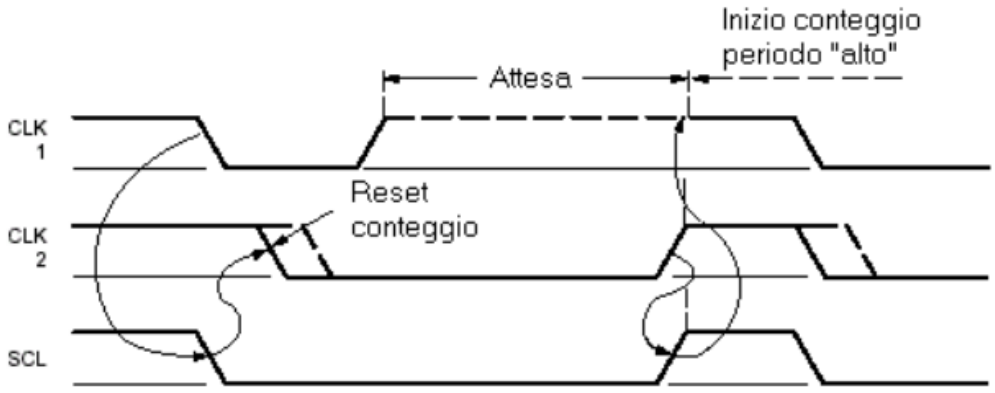


Figura 3.9: Sincronizzazione del clock

Grazie alla proprietà della connessione Open-Drain in cui prevalgono i valori '0' sui valori '1', il meccanismo della sincronizzazione è automatico e prevede che un elemento inizi il conteggio del periodo alto del suo clock seriale quando la linea SCL ritorni al valore 1; fintanto che questo non avviene, il dispositivo si troverà

in uno stato di attesa. In questo modo, una linea SCL sincronizzata avrà il suo periodo basso determinato dal *Master* avente il periodo basso più lungo; viceversa la durata del livello alto sarà decisa dal dispositivo con il periodo alto più breve. Di conseguenza, abbiamo che la linea SCL sul bus è ha un comportamento differente sia da CLK1 che CLK2, come possiamo osservare nella Figura 3.9.

Arbitraggio

Un dispositivo può iniziare una comunicazione con una periferica nel momento in cui il bus è libero. Ciononostante, potrebbe capitare che due *Master* facciano partire una transizione entro il **tempo di hold minimo** della condizione di START ($\mathbf{t_{HD,STA}}$), risultando quindi in un inizio valido per entrambi. È qui che subentra l'arbitraggio per determinare chi porterà a termine la propria operazione. Questo processo prosegue bit a bit: durante la trasmissione, mentre SCL è alto, ciascun *Master* controlla se il livello di SDA coincide con ciò che è stato inviato. Il primo dispositivo che prova ad inviare un uno logico ma rileva un livello di SDA basso, capirà di aver perso il controllo del bus e interromperà la comunicazione disabilitando la propria uscita. L'altro invece proseguirà fino alla fine. Questo lo possiamo osservare bene nella Figura 3.10:

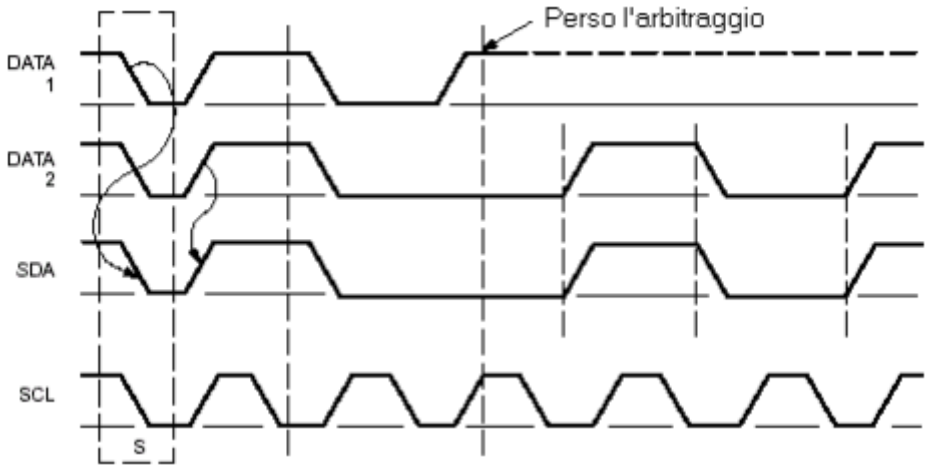


Figura 3.10: Arbitraggio

Sostanzialmente, vincerà sempre il dispositivo che trasmette il dato o l'indirizzo numericamente più piccolo. In linea di principio, se due *Master* vogliono comunicare con la stessa periferica e inviare il medesimo dato, potrebbero entrambi portare a compimento la transazione.

3.3 Serial-Peripheral-Interface (SPI)

Il **Serial Peripheral Interface** [6], abbreviato *SPI*, è un altro sistema di comunicazione seriale sincrono usato tra un micro-controllore (il *Master*) e delle periferiche (*Slave*). Analogamente al protocollo I^2C, anche qui è il *Master* a emettere il clock seriale e a decidere quando iniziare e terminare una transizione. D'altro canto però, lo Standard SPI è di tipo **full-duplex** in quanto la comunicazione può avvenire contemporaneamente in trasmissione e ricezione. I fili allocati in questo caso sono infatti almeno quattro:

- **SCLK**: Clock seriale generato dal *Master*.
- **MISO**: Dato seriale in ingresso al *Master* e uscita dallo *Slave*.
- **MOSI**: Dato seriale in uscita dal *Master* e in ingresso allo *Slave*.
- **SS**: Segnale di *Slave Select* attivo-basso emesso dal *Master* per selezionare il dispositivo con il quale si vuole comunicare.

Il segnale di SS è un segnale dedicato a ciascuna periferica per l'abilitazione della stessa e teoricamente ce ne dovrebbero essere tanti quanti sono le periferiche con il quale è possibile comunicare, come mostrato nel seguente esempio di connessione diretta tra un *Master* e altri tre dispositivi:

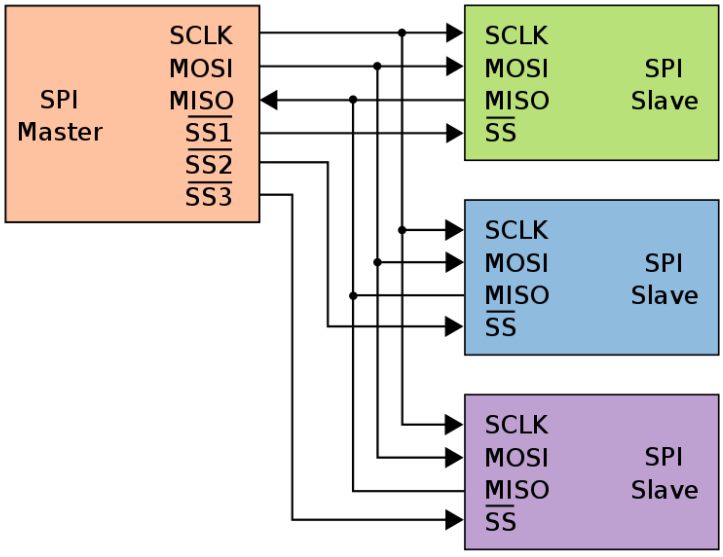


Figura 3.11: Connessione SPI tra un *Master* e più dispositivi *Slave*

I vantaggi di questa configurazione sono una più rapida comunicazione grazie al fatto che non serve indirizzare la periferica di interesse. Inoltre, rispetto al bus I^2C, non è previsto nessun segnale di riconoscimento tra un Byte e l'altro. La frequenza del clock può raggiungere livelli molto elevati (anche MHz) in funzione della velocità supportabile dai singoli dispositivi.

3.3.1 Modalità di funzionamento

Lo Standard *SPI*, a differenza del I^2C, non stabilisce una polarità e una fase del clock seriale con i quali andiamo a campionare e scorrere i dati all'interno dei nostri registri. Infatti, sono possibili quattro diverse modalità definite da due parametri che andranno impostati su ciascun dispositivo:

- **CPOL**: regola la polarità e se CPOL = 0, il clock è a livello logico basso nello stato di riposo.
- **CPHA**: regola la fase ovvero il fronte di clock in cui il ricevente campiona il segnale in ingresso.

Se CPOL = 0 allora con CPHA possiamo scegliere di campionare il dato sul fronte di salita della linea di clock impostando CPHA = 0 essendo il fronte primario, oppure sul fronte di discesa (fronte secondario) impostando CPHA a 1. Se vogliamo l'esatto opposto dovremo impostare CPOL pari a 1 e in questo caso se CPHA = 0, campioneremo sul fronte di discesa essendo questo il fronte primario. Il tutto viene riassunto nella seguente immagine:

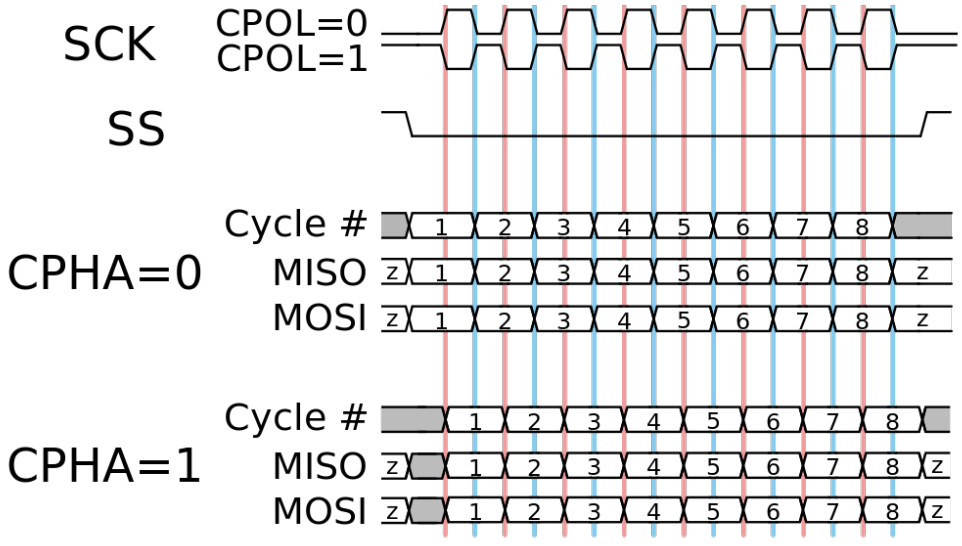


Figura 3.12: CPOL e CPHA

Anche la direzione dello scorrimento non viene definita dal protocollo (se **MSB-first** o **LSB-first**). Per questo motivo, prima di iniziare a comunicare, sarà necessario stabilire in che verso trasmettere i dati.

3.3.2 SPI: Accesso ad una periferica

Come abbiamo già anticipato, per comunicare con uno *Slave*, il *Master* deve abbassare la sua linea di selezione per portarlo fuori dall'alta impedenza. Successivamente, il protocollo non stabilisce il tipo e la sequenza di dati da trasferisce per eseguire una particolare operazione (come accedere ad una locazione di memoria o ripulire una pagina intera se il dispositivo dovesse trattarsi di una EEPROM) ma ciò dipende da periferica e periferica. Questo aspetto del protocollo *SPI* è molto più flessibile rispetto al bus I^2C, che invece prevede una sequenza predefinita di indirizzi e dati per portare a termine una determinata operazione sul bus.

3.4 Universal Asynchronous Receiver-Transmitter

Lo Standard **UART** [7] è un sistema di comunicazione che converte flussi di bit da un formato parallelo a uno seriale e viceversa. Ad oggi, ogni micro-controllore possiede la sua *UART* dedicata e a differenza dei due protocolli appena descritti, quest'ultimo è asincrono e non prevede quindi una linea nella quale scorre un segnale di temporizzazione. Può essere disposto sia in modalità che **full-duplex** nella quale vengono allocati entrambi i dispositivi **trasmettitore TX** e **ricevitore RX** che lavorano in autonomia, oppure **half-duplex** in cui è presente un singolo blocco hardware capace di alternarsi tra una modalità in trasmissione o in ricezione in base alle necessità del momento. Riportiamo un esempio (Figura 3.13) di due dispositivi *UART full-duplex* nel quale possiamo notare come il trasmettitore del primo dispositivo è collegato al ricevitore del secondo e viceversa:

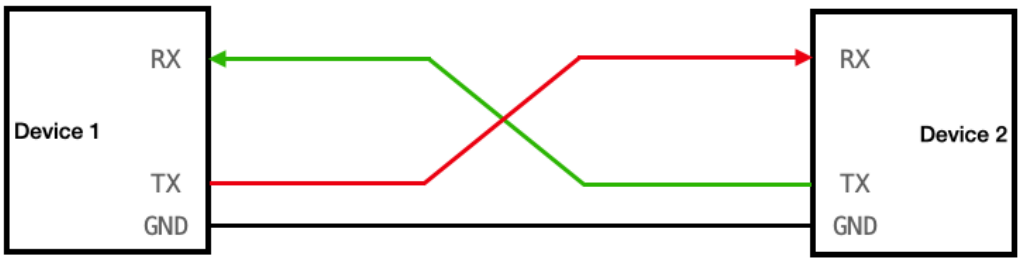


Figura 3.13: Diagramma a blocchi UART

La velocità è definita dal **Baud-Rate** e valori tipici sono 4800, 9600, 19200, 38400, 57600 e 115200 bit/s. È fondamentale che sia il trasmettitore che il ricevitore dall'altra parte siano impostati per lavorare con il medesimo Baud-Rate per evitare la perdita di dati.

3.4.1 Formato frame e sincronizzazione

Poichè stiamo parlando di un protocollo asincrono, non è previsto un filo da utilizzare per sincronizzare il flusso di dati trasmesso. Di conseguenza la sincronizzazione avviene attraverso la linea di dato stessa; a riposo, la linea di dato è ferma a livello alto e per segnalare al ricevitore l'inizio di una transizione, il trasmettitore abbasserà la linea di dato inviando uno **START-BIT**. Successivamente si prosegue con l'invio seriale di bit che compongono il pacchetto dati (tipicamente LSB-first) e per terminare si invia uno **STOP-BIT** per riportare la linea all'uno logico. I dati vengono trasferiti sotto forma di frame con una lunghezza variabile che va da 5 a 9 bit per frame con possibilità di aggiungere un bit di parità opzionale che può essere utilizzato per il rilevamento degli errori. Il valore del bit di parità dipende dal tipo di parità utilizzato (pari o dispari):

- **Parità pari**: questo bit è impostato in modo tale che il numero totale di 1 nel frame sarà pari.
- **Parità dispari**: questo bit è impostato in modo tale che il numero totale di 1 nel frame sarà dispari.

Questo bit di ridondanza è escluso nel caso in cui i bit che compongono il frame dati sono 9. Inoltre è possibile l'invio di un doppio **STOP-BIT** nel caso in cui si voglia trasmettere più pacchetti consecutivi. Un esempio di Frame UART è mostrato qui di seguito:

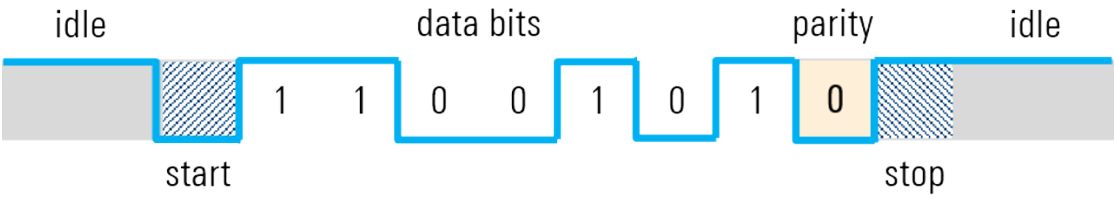


Figura 3.14: Esempio frame UART

Oltre ad avere lo stesso Baud-Rate, entrambi i lati di una connessione UART devono anche utilizzare la stessa struttura di frame e gli stessi parametri per una corretta

comunicazione, altrimenti si può andare incontro ad errori di frame/sincronizzazione

3.5 Protocollo APB

Il protocollo **APB** (Advanced-Peripheral-Bus) [8] è una branca del più ampia famiglia dei protocolli **AMBA** (Advanced Microcontroller Bus Architecture) ed è un bus sincrono a basso costo ottimizzato per ridurre la complessità dell'interfaccia hardware e ottenere un consumo minimo di potenza. A differenza di altri, non possiede una **pipeline** e il modello a 32 bit è stato sviluppato per la comunicazione tra dispositivi *Master* e *Slave*; tipicamente, l'APB si usa per far comunicare il *SoC* con le sue periferiche esterne. Infatti, si può interfacciare con altri protocolli nel mezzi tramite dei cosiddetti **APB-bridge**, che andranno poi fatto ad essere i *Master*, come mostrato nel seguente esempio:

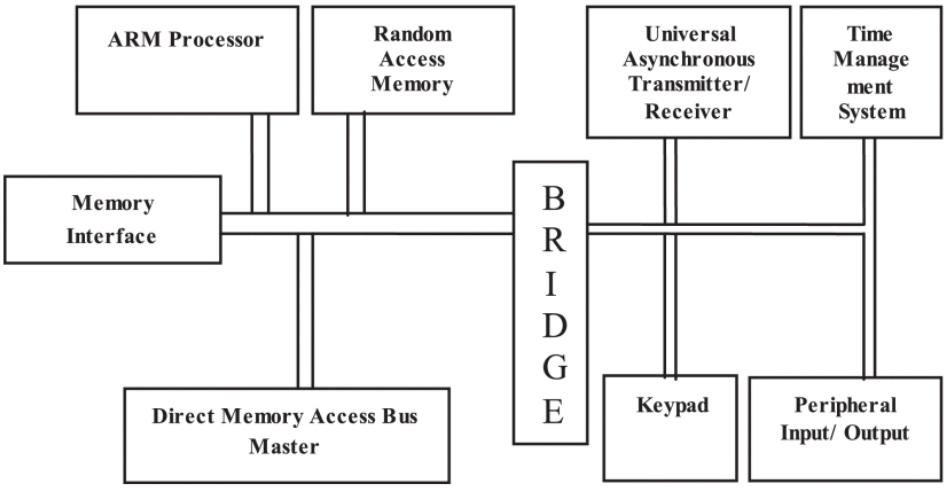


Figura 3.15: Esempio di APB-bridge

3.5.1 Descrizione dei segnali

I segnali scambiati sul bus sono:

- **System bus slave interface**: questa è l'interfaccia di sistema che trasferisce le transazioni AMBA all'APB-bridge.
- **PCLK**: clock di sistema.
- **PRESETn**: reset asincrono attivo basso.

- **PADDR[31:0]**: indirizzo per accedere ad un registro presente nell'interfaccia APB dello *Slave*.
- **PWDATA[31:0]**: bus di scrittura dal *Master* allo *Slave*.
- **PRDATA[31:0]**: bus di lettura dallo *Slave* al *Master*.
- **PSELx**: segnale di *slave select* usato dal *Master* per selezionare una periferica; ce ne sono tanti quanti *Slave* sono connessi al bus APB (simile al bus *SPI*).
- **PENABLE**: indicata il secondo e successivo ciclo di trasferimento. Quando è asserito, la fase di accesso del trasferimento inizia.
- **PWRITE**: comando di lettura/scrittura, quando è alto indica una scrittura mentre quando è basso una lettura.
- **PREADY**: è una uscita dello *Slave* ed è usato per includere degli stati di attesa quando non è pronto prima di completare una operazione.
- **PSLVERR**: indica il successo (1 logico) o il fallimento di un trasferimento (0 logico).

Qui di seguito è raffigurato un esempio di diagramma a blocchi con APB-bridge e *Slave* APB:

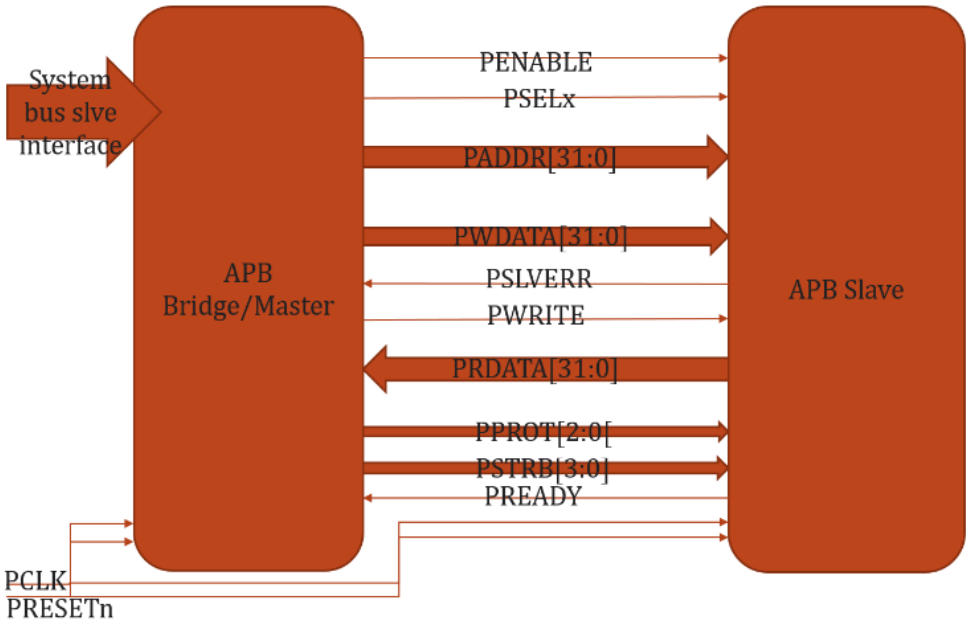


Figura 3.16: Diagramma a blocchi APB

3.5.2 Esempio di scrittura APB senza stati di attesa

Vediamo il diagramma temporale di una transazione sul bus per effettuare una scrittura:

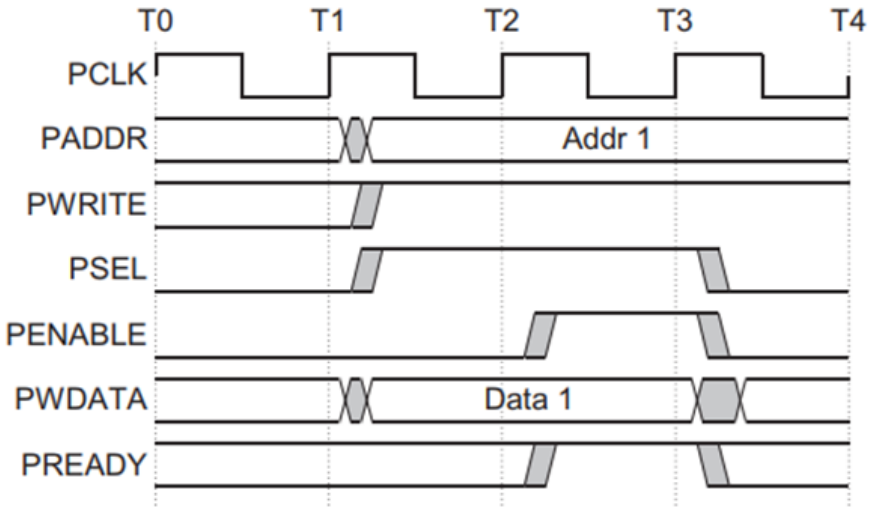


Figura 3.17: Scrittura APB

- Nell'istante **T1**, parte il trasferimento sul bus per effettuare una scrittura (PWRITE = 1) con l'indirizzo PADDR, il dato PWDATA e il segnale di selezione PSEL.
- Verranno campionati al successivo fronte del clock nell'istante **T2**.
- Successivamente, PENABLE e PREADY sono campionati al fronte **T3**.
- Quando è asserito PENABLE significa l'inizio della fase di accesso. Invece, quando PREADY è asserito, significa che lo *Slave* può portare a compimento il trasferimento al fronte successivo.
- PADDR, PDATA e i segnali di controllo devo rimanere stabili fino al completamento dell'operazione (T3).
- PENABLE sarà de-asserito alla fine della transizione. Se il *Master* vuole continuare a comunicare con lo stesso *Slave*, allora PSEL rimarrà alto anche nel prossimo trasferimento.

3.5.3 Esempio di lettura APB senza stati di attesa

Infine passiamo ad un esempio di lettura APB:

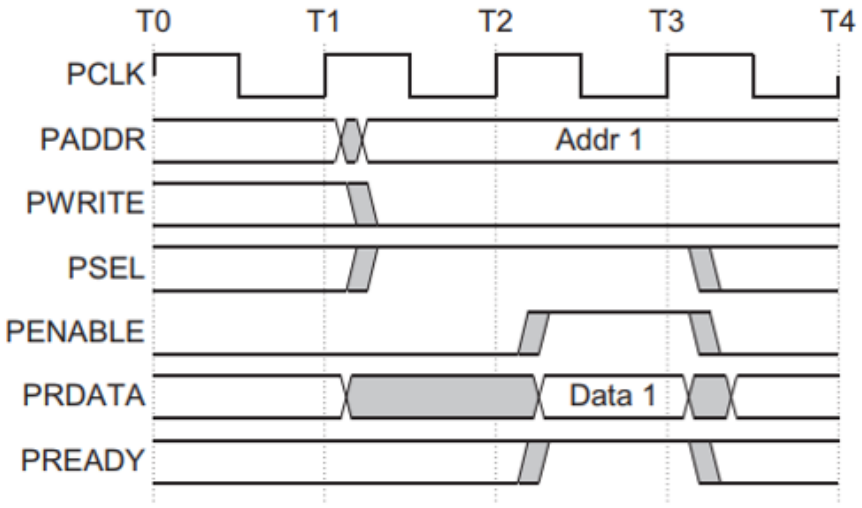


Figura 3.18: Lettura APB

Il tempismo è analogo al caso della scrittura, ovviamente questa volta il comando PWRITE sarà allo zero logico trattandosi di una lettura. Quando lo *Slave* è pronto, asserirà il segnale PREADY e insieme, piloterà la linea PRDATA nella quale scorrerà il dato indirizzato nella fase iniziale dal *Master*.

Capitolo 4

FSM, condivisione hardware e Register-Map

Conclusa l'analisi e lo studio dei tre protocolli, possiamo ora dedurre quali e che tipo di risorse hardware sono necessarie per l'implementazione di ciascuno di essi. In questo capitolo ci dedicheremo a cercare di comprendere come ottimizzare al nostro meglio il meccanismo di condivisione della parte hardware e lo sviluppo di una macchina a stati che gestisca tutto il flusso di operazioni specifiche, dalla fase di configurazione a quella operativa. Parleremo di quali e quanti registri e contatori allocheremo per supportare tutti e tre i bus seriali e di come questi vengano usati in un modo piuttosto che in altro a seconda di come è stata configurata l'IP. In parallelo, inizieremo a definire qualche parametro di configurazione di componenti hardware in funzione proprio del protocollo seriale con il quale vogliamo lavorare, come delle diverse modalità operative disponibili in essi. Mostreremo infine anche qualche esempio di registro di configurazione e di stato, analizzandoli a livello di **campo** (bit-field) e specificando le modalità di accesso da parte dell'utente.

4.1 Macchina a stati generale

L'oggetto più complesso di questo progetto è senza ombra di dubbio la macchina a stati che dovrà gestire e organizzare tutto il flusso di lavoro necessario per configurare la macchina e in seguito portare a termine una transizione. L'idea alla base è quindi quella di avere un'unica FSM che sia in grado di supportare tutte e tre i protocolli uno alla volta, a seconda di come viene configurata; questo ha un doppio riscontro positivo:

- Maggiore flessibilità e semplicità nel controllo delle risorse hardware come registri e contatori.

- Condivisione del registro di stato e quindi anche della logica interna alla macchina a stati.

D'altro canto però, volendo implementare una *UART* full-duplex, non sarà possibile gestire sia il trasmettitore che il ricevitore da una unica FSM essendo questi indipendenti l'uno dall'altro. Per questo motivo faremo supportare dalla nostra FSM generale solo un lato della *UART* che comprensibilmente sarà quello più costoso in termini di area dato che le risorse utilizzate saranno le medesime di quelle usate per I^2C e SPI; per l'altro lato invece stanzieremo una piccola FSM dedicata.
Detto ciò la macchina, dopo un **reset**, partirà da uno **stato di configurazione** nel quale sostanzialmente non è operativa ma è in attesa di ricevere tutti i parametri e appena questo processo è stato completato dall'utente programmando tutti i registri di configurazione APB opportuni, verrà asserito un segnala chiamato `enable_configuration` il quale farà uscire la macchina dal suo stato per poi seguire un unico ramo degli stati appartenenti ad un determinato Standard seriale. Nel particolare, questo sarà deciso da due parametri (che saranno due **campi** provenienti dallo stesso registro di configurazione) dei quali uno selezionerà il protocollo mentre l'altro se configurare la periferica come *Master* o come *Slave*. Ovviamente, quest'ultimo parametro è ignorato nel caso in cui venisse scelto di operare come *UART*. Uno schema esemplificativo è mostrato qui di seguito:

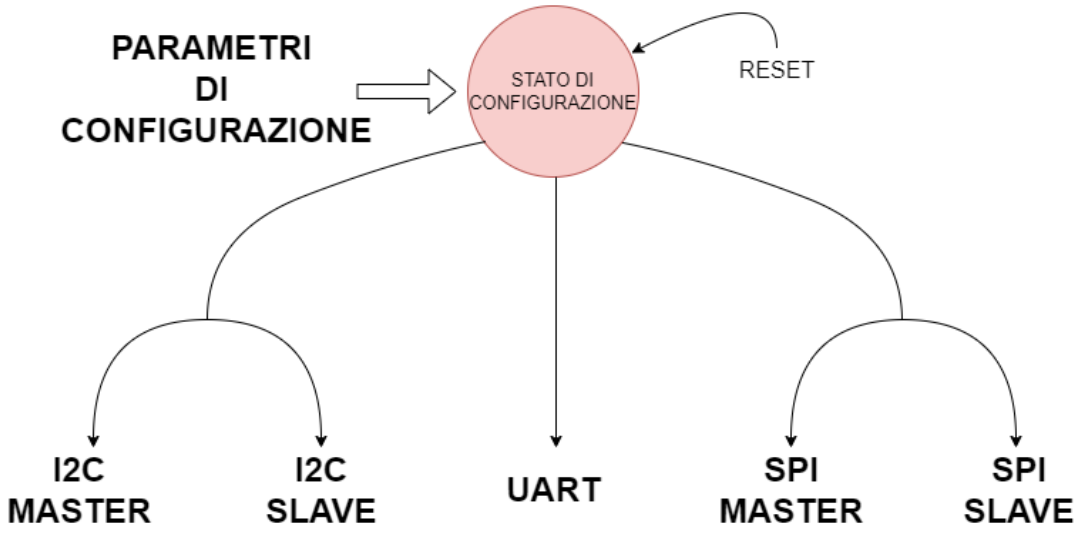


Figura 4.1: Configurazione FSM generale

Oltre ai due appena menzionati, alcuni parametri di configurazioni definisco caratteristiche più operative come **Baud-Rate**, **Word-Lenght**, comando **Read/Write** e altri. È logico dedurre che a seconda del protocollo, la macchina a stati non sarà sensibile a tutti i parametri a disposizione. A titolo di esempio, i parametri CPOL e CPHA riguardano solo il bus *SPI* e non influiscono sul funzionamento degli altri due.
Successivamente, la macchina è pronta ad eseguire una transizione ed attenderà dall'utente un altro segnale di `enable_transaction` che la farà partire per eseguire una specifica operazione che una volta portata a termine, ritornerà nello stato di IDLE dello Standard seriale selezionato. Diversa è la situazione in cui avessimo configurato la periferica come *Slave* o *UART-RX*; in quel caso, saremo noi ad attendere l'inizio di una transizione dall'esterno. Se l'utente volesse poi riconfigurare la macchina operante con un altro bus, asserirà il reset che la riporterà nello stato di configurazione, come mostrato in Figura 4.1.

4.2 Buffer in trasmissione e ricezione

Nel capitolo precedente abbiamo studiato i principi e le regole degli Standard seriali ed è chiaro dunque come tutte e tre i protocolli ruotano attorno all'utilizzo di registri a scorrimento usati sia per trasmettere che per ricevere dati. La Figura 4.2 mostra come il nostro oggetto conterrà al suo interno almeno due buffer dei quali uno verrà caricato in parallelo con una uscita seriale; l'altro registro invece sarà caricato serialmente e preleveremo parallelamente la sua uscita che sarà il dato al suo interno una volta terminato il trasferimento.

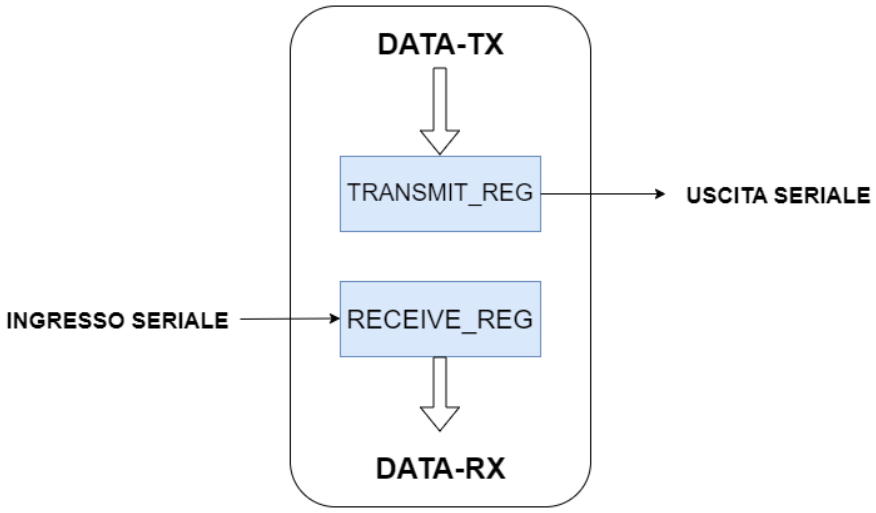


Figura 4.2: Buffer TX e RX

È stato deciso che sarà la FSM generale a contenere direttamente al suo interno questi buffer (che chiameremo `transmit_reg` e `receive_reg`) di modo che sarà estremamente semplice gestire il caricamento seriale o parallelo in un determinato stato o a seguito di un particolare evento. Questa idea di includere all'interno della macchina a stati le risorse necessarie sarà cruciale per gestire in semplicità le diverse specifiche tra un bus e l'altro.
Per quanto riguarda la capacità di questi registri, essendo la grandezza del dato con il quale vogliamo lavorare una specifica configurabile della nostra IP, saranno stanziati a seconda della massima capacità disponibile della periferica (per esempio 32 bit).

Verso di scorrimento

Ora è importante capire in che direzione questi registri faranno scorrere i dati, se MSB-first o LSB-first. In primis, questo sarà determinato ovviamente dal protocollo selezionato che come abbiamo visto differiscono tutti e tre sotto questo aspetto. Di conseguenza, sarà la macchina a stati a gestire il verso di scorrimento in funzione dello stato in cui si trova e nella situazione in cui stessimo operando con un *SPI* (nel quale non è predefinito un verso di scorrimento), sarà un parametro di configurazione a determinarlo al quale la macchina sarà sensibile solo nel caso in cui avessimo configurato l'IP come tale.

4.2.1 Secondo buffer in ricezione

Oltre ai due sopracitati buffer, all'interno della nostra FSM è presente anche un altro registro in ricezione (che chiameremo `receive_reg2`) il quale è utilizzato in maniera differente a seconda della modalità scelta dall'utente.

Caso SPI e I^2C

In una situazione in cui stessimo operando con i due protocolli sincroni, il buffer `receive_reg2` sarà utilizzato per memorizzare l'indirizzo di un registro o di una locazione di memoria da parte di uno *Slave*. Queste locazioni di memoria saranno poi mandate dalla FSM in uscita verso l'interfaccia APB come registri di sola lettura (**read-only**) da parte dell'utente. Bisogna però menzionare il fatto che questa fase di indirizzamento avviene solo nel caso in cui avessimo configurato l'interfaccia come *Slave* e quindi per simulare il caso in cui un *Master* volesse accedere ad una memoria connessa al bus con uno spazio disponibile di indirizzamento e che quindi, a seconda dei comandi ricevuti dell'altra parte, effettui delle operazioni piuttosto che altre. In ottica però di una futura integrazione della IP nel sistema di EPIC, è verosimile che questa fase di indirizzamento sia superflua e che quindi venga rimossa. Uno schema di principio semplificato è mostrato nella Figura 4.3:

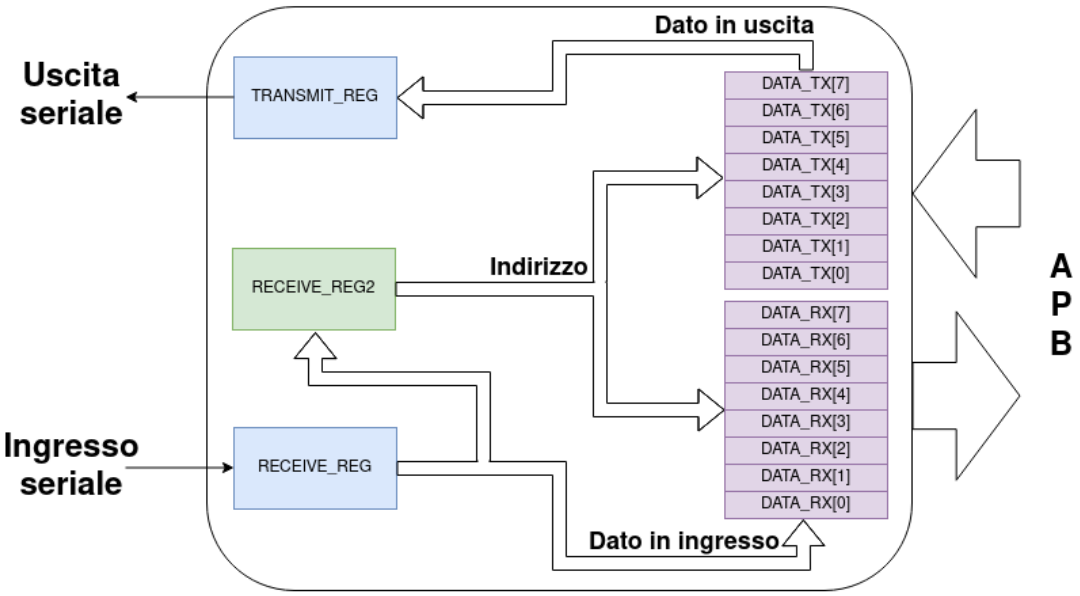


Figura 4.3: Utilizzo di `receive_reg2` con bus sincroni

In questo modo, se il *Master* volesse effettuare una scrittura o una lettura con una periferica, lo *Slave* potrà cosi memorizzare questo indirizzo per poi poter puntare alla locazione di memoria (per ragioni di semplicità ogni *Slave* ha a disposizione uno spazio di memoria di 16 locazioni da 32 bit ciascuna, 8 in scrittura e altrettante in lettura). A quel punto, passerà il dato della locazione indirizzata da `receive_reg2` al `transmit_reg` oppure, in caso di scrittura, caricherà in essa il dato ricevuto su `receive_reg`. Se l'interfaccia è invece configurata come *Master*, questo spazio di memoria sarà usato per salvare dati in arrivo dal bus, partendo dal primo dei otto registri dedicati al salvataggio delle parole ricevute dall'esterno fino all'ultimo per poi ripartire dal primo. Ogni locazione avrà il suo flag dedicato per segnalare che c'è un dato pronto e che quindi deve essere letto dall'utente.

Caso UART-RX

Nell'implementazione con la quale è stato realizzato il ricevitore *UART*, il secondo registro di ricezione è stato pensato per introdurre un fattore di **sovra-campionamento** dell'ingresso rispetto al Baud-Rate. Dal capitolo precedente abbiamo appurato il fatto che a riposo, la linea di dato è ferma all'uno logico e l'inizio di una transizione è definita da uno **START-BIT**, ovvero il passaggio dall'uno allo zero logico. Grazie al sovra-campionamento, nello specifico di un fattore 8, acquisiamo un certo livello di sicurezza per quel che riguardano possibili **spikes** (o in generale disturbi) sulla linea di dato seriale che ne fanno cambiare il livello.

Infatti, il riconoscimento dello START-BIT avviene prendendo parallelamente 8 bit del `receive_reg2` e inviandoli in ingresso ad una porta AND con i primi 4 ingressi negati; l'uscita di questa porta andrà ad 1 nel momento in cui riconoscerà la sequenza quattro uni seguiti da quattro zeri, ovvero l'inizio di una transizione.

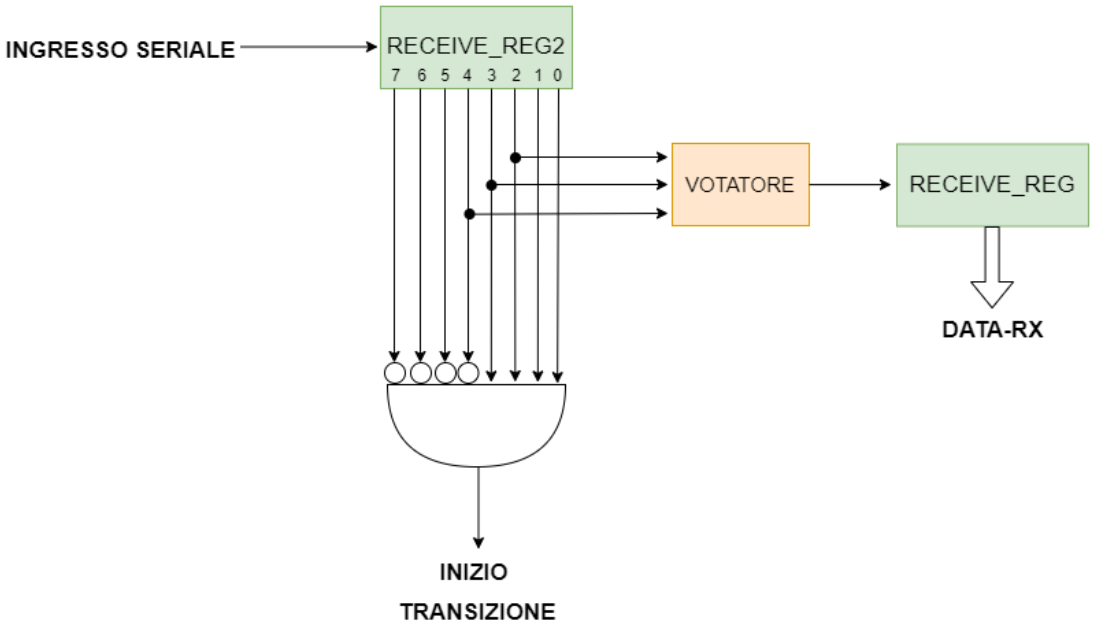


Figura 4.4: Utilizzo di `receive_reg2` con UART-RX

Successivamente, basterà attendere di posizionarci al centro del primo simbolo della sequenza in arrivo, come mostrato nell'immagine seguente:

Figura 4.5: Posizionamento "al centro" del simbolo

A quel punto, prenderemo i tre più centrali degli otto in uscita da `receive_reg2` e li manderemo ad un **votatore a maggioranza**, un componente logico-combinatorio composto da 3 porte AND le cui uscite confluiscono in una OR che deciderà il valore finale del simbolo. Questo aumenterà ancor di più il grado di sicurezza rispetto a possibili disturbi sull'ingresso alla *UART*; il fatto di scegliere i tre più centrali deriva invece dalla possibilità che i campioni più esterni potrebbero essere corrotti o non stabili a causa del transitorio che avviene passando da un simbolo

all'altro. Le decisioni del votatore saranno poi inviate in ingresso all'altro registro che le scorrerà mano a mano che si avanza con i simboli per ritrovarci alla fine con tutta la sequenza di bit contenuta in esso.

4.3 Contatori

La nostra periferica conterrà al suo interno un numero di **contatori** necessario ad assicurarci il corretto funzionamento di ciascuno dei Standard seriali disponibili. Infatti, in tutti e tre i casi, attraverseremo uno o più stati nei quali effettueremo lo scorrimento seriale dei dati caricati in un registro per scodarli o per caricarli serialmente e ci resteremo fintanto che un contatore avrà raggiunto un limite prefissato da noi. Un altro contatore è richiesto per il calcolo del Baud-Rate che farà avanzare il processo operativo non appena raggiungerà un altro limite, fissato questa volta da un parametro di configurazione, per esempio un fattore di divisione per il clock della periferica con il quale verrà calcolato il clock seriale da mandare sul bus (`divider`).
Questi contatori, analogamente ai buffer di trasmissione e ricezione, saranno integrati direttamente all'interno della macchina a stati e questo faciliterà di molto la loro abilitazione e resettaggio a seconda ovviamente dello stato in cui si trova la macchina.

Divider-counter

Il `divider-counter` è, come suggerisce il nome, un contatore a 16 bit con il quale verrà ricavata la frequenza del clock seriale che un eventuale periferica configurata come *Master* si dovrà occupare di generare a partire dalla frequenza del clock della periferica stessa attraverso di una sua divisione, ma definirà anche il Baud-Rate con il quale verranno trasmessi/ricevuti i pacchetti *UART*. Infatti, il suo conteggio verrà incrementato ad ogni fronte positivo del clock e quando raggiungerà un valore pari al fattore di divisione accennato precedentemente, il conteggio verrà ripulito e al tempo stesso verrà generato un impulso (`divider_tick`) che farà avanzare il processo e che al colpo successivo verrà di nuovo de-asserito fino a quando il conteggio non raggiunge nuovamente il fattore di divisione. Questo è stato fatto per assicurarci di rispettare un certo sincronismo tra la linea di dato seriale e quella del clock definiti da i due protocolli sincroni (*SPI* e I^2C), grazie ad una maggiore granularità temporale all'interno di uno stato della FSM. Di fatto, il periodo del clock seriale è stato diviso in quattro parti definite da un altro piccolo contatore chiamato `process_counter` che avanzerà come detto precedentemente ad ogni impulso di `divider_tick`. La durata di ciascun quarto di periodo è quindi definita dal fattore di divisione e la frequenza della linea del clock seriale (per esempio SCL) è calcolata come:

$$f_{SCL} = \frac{f_{clk}}{4(1 + divider)} \tag{4.1}$$

Il termine `divider` nell' Equazione 4.1 è il fattore di divisione fin quì citato e proverrà da un registro di configurazione.
Per comprendere meglio questo meccanismo della gestione del sincronismo tra il dato e il segnale temporizzante riportiamo un esempio di una simulazione del bus I^2C in cui usiamo la nostra periferica:

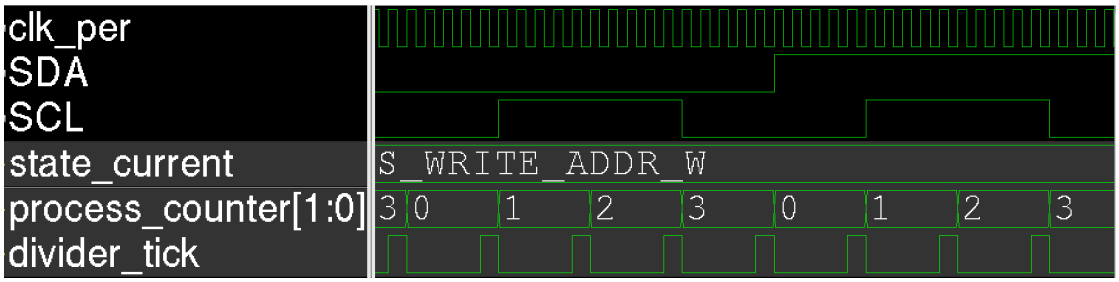


Figura 4.6: Sincronismo SDA e SCL in un bus I^2C

Notiamo subito come ad ogni impulso del segnale `divider_tick`, il `process_counter` viene incrementato. Inoltre la linea SDA può cambiare solo nel periodo basso di SCL e questo è molto importante per rispettare le specifiche del protocollo I^2C, mentre SCL ha i suoi fronti solo nel passagio di `process_counter` da 0 a 1 e da 2 a 3. Discorso simile vale anche nel caso *SPI*, dove tutto questo sarà invece determinato dai parametri CPHA e CPOL che definisco su quali fronti andare a campionare la linea di dato e su quali invece scorrere serialmente i dati nel buffer di trasmissione.
Il codice *SystemVerilog* del `divider_counter` della FSM generale è riportato qui di seguito:

```
1   always_ff @(posedge clk, negedge rstn) begin
2    if (~rstn) begin
3        divider_counter <= 16'd0;
4        divider_tick <= 0;
5    end
6    else begin
7        if (protocol_i == 2'b10) begin //UART-RX Selected
8            if (divider_counter >= (divider_i>>3)) begin
9                divider_counter <= 16'd0;
10           end
11           else begin
12               divider_counter <= divider_counter + 1;
13           end
14       end
15       else begin //EITHER I2C or SPI
16           if (state_current == S_IDLE_I2C_MASTER || state_current ==
             ↪ S_IDLE_SPI_MASTER) begin
17               divider_counter <= 16'd0;
18               divider_tick <= 0;
19           end
20           else if (divider_counter >= divider_i) begin
21               divider_counter <= 16'd0;
22               divider_tick <= 1;
23           end
24           else begin
25               divider_counter <= divider_counter + 1;
26               divider_tick <= 0;
27           end
28       end
29   end
30  end
```

Possiamo notare come il funzionamento di questo blocco procedurale sia funzione dell'ingresso `protocol_i` il quale consiste in un campo di un registro di configurazione (aspetto che analizzeremo più nel dettaglio nella sezione successiva). Sostanzialmente abbiamo due situazioni: nel caso in cui `protocol_i = 2'b10` significa che l'IP è stata configurata come *UART* e che quindi la macchina a stati (Figura 4.1) si occuperà del lato ricevitore e come menzionato in precedenza,

andremo a sovra-campionare l'ingresso per un fattore 8 e la verifica del raggiungimento del conteggio sarà dunque fatta su `divider/8`; sarà in quel momento infatti che campioneremo la linea di dato. Inoltre, non essendo presente un segnale di temporizzazione, non ci sarà necessità di preoccuparci di rispettare un certo sincronismo tra la linea del dato e quella del clock. Per questo motivo, nel ramo degli stati appartenenti alla configurazione *UART*, il contatore `process_counter` non verrà usato (come `divider_tick`), mentre `divider_counter` detterà la frequenza con la quale andremo a scodare i bit sull'uscita (lato trasmettitore) o a campionare l'ingresso (lato ricevitore). Se invece `protocol_i != 2'b10`, allora la modalità selezionata sarà una tra I^2C e *SPI*; in tal caso, la verifica della condizione `divider_counter >= divider_i` non è valida solo quando la macchina si trova nello stato di IDLE ma vengono fissati sia `divider_counter` che `divider_tick` a zero. Questo è stato aggiunto per far si che nel momento in cui venga abilitata l'inizio di una transizione sul bus, il conteggio parta proprio da zero e la durata della condizione di START non sia cosi deterministica.

Bit-Counter e Second-Counter

Nella nostra implementazione sono necessari due ulteriori contatori chiamati `bit-counter` e `second-counter`. Il primo, come si può evincere dal nome, terrà il conto del numero di bit scodati e/o caricati serialmente. In questo modo, saremo in grado di regolare correttamente la permanenza della FSM all'interno di uno stato dove effettueremo lo scorrimento dei dati all'interno di uno o più registri. Una volta raggiunto un certo limite, usciremo da quello stato. Questo vale chiaramente per tutti e tre i protocolli. Diverso è il discorso riguardante il secondo; infatti, analogamente a `receive_reg2`, questo contatore avrà funzioni differenti in base alla comunicazione seriale configurata:

- *UART*: lato ricevitore conterà il numero di campioni nel `receive_reg2` fino ad 8 per decidere il momento esatto (quello in cui siamo al centro del bit) in cui caricare in `receive_reg` la decisione del votatore che rappresenterà il valore finale del simbolo corrente.
- I^2C: essendo la nostra IP capace di configurare la lunghezza di una parola, in questo caso `second_counter` si occuperà di contare il numero di Byte inviati.
- *SPI*: in modalità burst, terremo il conteggio del numero di parole inviate/ricevute con il secondo contatore.

Memory-counter

Un ultimo contatore a 3 bit `memory_counter` è stato allocato per indirizzare lo spazio di memoria dedicato al salvataggio in parallelo dei dati provenienti dal bus,

nel caso in cui configurassimo la periferica come *Master*. Questo verrà incrementato ogni volta che verrà passato un nuovo dato pronto in uscita puntando alla locazione successiva, evitando così di dover riscrivere quella appena scritta immediatamente al prossimo dato pronto in arrivo. Questo permetterà all'utente di avere più tempo per leggere il corrispettivo registro prima che questo venga sovrascritto, perdendo così il dato contenuto in esso. In parallelo, il valore del contatore selezionerà anche il flag corrispondente al registro scritto per settarlo (approfondiremo questo aspetto nella prossima sezione sulla Register-Map).

4.4 Register-Map

Ci dedichiamo ora alla **Register-Map** di **RDIP** (**Reconfigurable-Digital-IP**) contenente tutti i registri di configurazione, dato e di stato che sono stati individuati per il corretto funzionamento del progetto e per l'utente saranno tutti accessibili (alcuni solo in lettura) dall'**interfaccia APB**. Qui di seguito riportiamo la tabella di tutta la Register-Map con una breve descrizio della funzionalità di ciascun registro:

Nome	**Indirizzo**	**Dimensione**	**Reset**	**Descrizione**
REG_PROT	`0x0000`	32	`0x0000`	Configura protocollo e modalità Master/Slave.
REG_BR	`0x0004`	32	`0x0000`	Configura il fattore di divisione per il Baud-Rate.
REG_WL	`0x0008`	32	`0x0000`	Configura lunghezza di una parola e dell'indirizzo da accedere.
REG_SPI	`0x000C`	32	`0x0000`	Configura i parametri SPI.
REG_UART	`0x0010`	32	`0x0000`	Configura i parametri UART.

REG_ADDR_SLAVE	0x0014	32	0x0000	Indirizzo Slave I2C e operazione finale
REG_ADDR_REG	0x0018	32	0x0000	Indirizzo registro da accedere
REG_DATA_TX[0]	0x001C	32	0x0000	Dato da trasmettere [0]
REG_DATA_TX[1]	0x0020	32	0x0000	Dato da trasmettere [1]
REG_DATA_TX[2]	0x0024	32	0x0000	Dato da trasmettere [2]
REG_DATA_TX[3]	0x0028	32	0x0000	Dato da trasmettere [3]
REG_DATA_TX[4]	0x002C	32	0x0000	Dato da trasmettere [4]
REG_DATA_TX[5]	0x0030	32	0x0000	Dato da trasmettere [5]
REG_DATA_TX[6]	0x0034	32	0x0000	Dato da trasmettere [6]
REG_DATA_TX[7]	0x0038	32	0x0000	Dato da trasmettere [7]
REG_DATA_RX[0]	0x003C	32	0x0000	Dato ricevuto dall'esterno [0]
REG_DATA_RX[1]	0x0040	32	0x0000	Dato ricevuto dall'esterno [1]
REG_DATA_RX[2]	0x0044	32	0x0000	Dato ricevuto dall'esterno [2]
REG_DATA_RX[3]	0x0048	32	0x0000	Dato ricevuto dall'esterno [3]
REG_DATA_RX[4]	0x004C	32	0x0000	Dato ricevuto dall'esterno [4]
REG_DATA_RX[5]	0x0050	32	0x0000	Dato ricevuto dall'esterno [5]

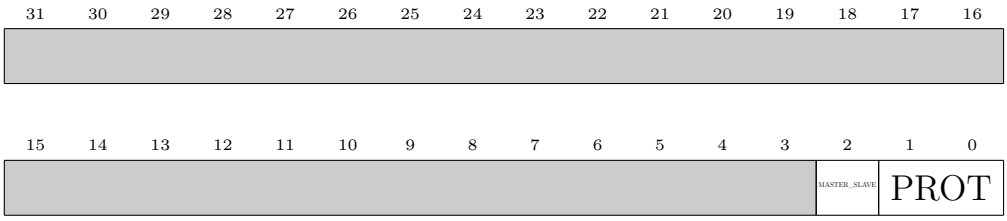
REG_DATA_RX[6]	0x0054	32	0x0000	Dato ricevuto dall'esterno [6]
REG_DATA_RX[7]	0x0058	32	0x0000	Dato ricevuto dall'esterno [7]
REG_STATUS_TX	0x005C	32	0x0000	Registro di stato lato TX.
REG_STATUS_RX	0x0060	32	0x0000	Registro di stato lato RX.
REG_INT_ENABLE	0x0064	32	0x0000	Registro di abilitazione degli interrupt.
REG_INT_STATUS	0x0068	32	0x0000	Registro di stato degli interrupt
REG_EN_CONFIGURATION	0x006C	32	0x0000	Abilitazione configurazione.
REG_EN_TRANSACTION	0x0070	32	0x0000	Abilitazione transazione.

Tabella 4.1: RDIP Register-Map

I registri sono riportati avanti la medesima dimensione, ovvero 32, anche se non tutti i bit sono utilizzati. Sarà poi il sintetizzatore a rimuovere le porzioni inutilizzate per ottimizzare l'area.
Scendiamo ora più nel dettaglio a livello di campo e partiamo dal primo registro di configurazione **REG_PROT** che è anche quello in cui andiamo a configurare il protocollo con cui vogliamo lavorare:

4.4.1 REG_PROT

Indirizzo: 0x0000
Valore di reset: 0x0000



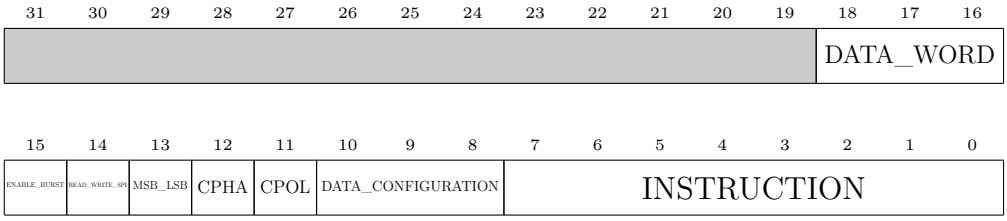
`Bit 2` **MASTER_SLAVE** *(RW)* Configura la periferica come Master/Slave:
-1'b0: Slave.
-1'b1: Master.

`Bit 1 - 0` **PROT** *(RW)* Configura il protocollo:
-2'b00: I2C.
-2'b01: SPI.
-2'b10: UART.

Entrambi i due campi contenuti in questo registro sono chiaramente accessibili all'utente sia in lettura che in scrittura (RW). Ovviamente, se configurassimo la IP come UART, il `Bit 2` che decide se operare in modalità *Master* o *Slave*, verrà ignorato dalla FSM.
Vediamo un altro esempio più specifico ad un protocollo in se:

4.4.2 REG_SPI

Indirizzo: `0x000C`
Valore di reset: `0x0000`



`Bit 18 - 16` **DATA_WORD** *(RW)* Numero di parole interessate in modalità Burst.

`Bit 15` **ENABLE_BURST** *(RW)* Abilita o meno modalità Burst:
-1'b0: non abilitata.
-1'b1: abilitata.

`Bit 14` **READ_WRITE_SPI** *(RW)* Abilita o meno il flag di ricezione per SPI:
-1'b0 = non abilitato.
-1'b1 = abilitato.

`Bit 13` **MSB_LSB** *(RW)* Configura il verso di scorrimento:
-1'b0: LSB-first.
-1'b1: MSB-first.

`Bit 12` **CPHA** *(RW)* Configura il parametro CPHA (fase del clock seriale):
-1'b0: CPHA=0.
-1'b1: CPHA=1.

`Bit 11` **CPOL** *(RW)* Configura il parametro CPOL (polarità del clock seriale):
-1'b0: CPOL=0.
-1'b1: CPOL=1.

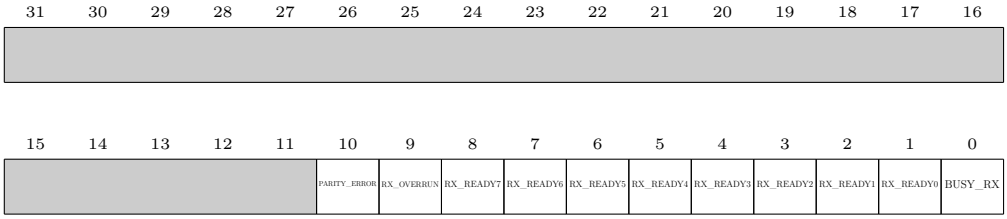
`Bit 10 - 8` **DATA_CONFIGURATION** *(RW)* Configura il tipo di dato da inviare allo *Slave SPI*:
-DATA_CONFIGURATION[0]: dato.
-DATA_CONFIGURATION[1]: indirizzo registro.
-DATA_CONFIGURATION[2]: istruzione.

`Bit 7 - 0` **INSTRUCTION** *(RW)* Istruzione.

Questo registro è esclusivamente inerente al bus *SPI* e non va scritto nel caso in cui non fossimo interessati ad usare questo protocollo. Esso contiene i parametri principali della configurazione del bus, da CPHA e CPOL alla configurazione del tipo di dato da usare e a un eventuale istruzione da trasmettere ad una periferica esterna se l'IP è configurata come *Master*. Anche in questo registro ci sono alcuni parametri a cui la macchina a stati è sensibile in funzione della modalità in cui opera, se *Master* o *Slave*. Come ultimo esempio mostriamo anche un registro di stato di tipo **READ-ONLY**, ovvero **REG_STATUS_RX**:

4.4.3 REG_STATUS_RX

Indirizzo: `0x000C`
Valore di reset: `0x0000`



`Bit 10` **PARITY_ERROR** *(RO)* Il flag segnala che il dato ricevuto dalla *UART* contiene un errore.

`Bit 9` **RX_OVERRUN** *(RO)* Il flag segnala che che il dato ricevuto precedentemente non è stato letto prima di sovrascrivere il registro REG_DATA_RX contenente il dato ricevuto.

`Bit 8` **RX_READY7** *(RO)* Il flag segnala che c'è un dato pronto per essere letto dall'utente nel registro REG_DATA_RX[7].

`Bit 7` **RX_READY6** *(RO)* Il flag segnala che c'è un dato pronto per essere letto dall'utente nel registro REG_DATA_RX[6].

`Bit 6` **RX_READY5** *(RO)* Il flag segnala che c'è un dato pronto per essere letto dall'utente nel registro REG_DATA_RX[5].

`Bit 5` **RX_READY4** *(RO)* Il flag segnala che c'è un dato pronto per essere letto dall'utente nel registro REG_DATA_RX[4].

`Bit 4` **RX_READY3** *(RO)* Il flag segnala che c'è un dato pronto per essere letto dall'utente nel registro REG_DATA_RX[3].

`Bit 3` **RX_READY2** *(RO)* Il flag segnala che c'è un dato pronto per essere letto dall'utente nel registro REG_DATA_RX[2].

`Bit 2` **RX_READY1** *(RO)* Il flag segnala che c'è un dato pronto per essere letto dall'utente nel registro REG_DATA_RX[1].

`Bit 1` **RX_READY0** *(RO)* Il flag segnala che c'è un dato pronto per essere letto dall'utente nel registro REG_DATA_RX[0].

`Bit 0` **BUSY_RX** *(RO)* Il flag indica che la parte di ricezione è occupata.

Tutti questi flag eccetto il `Bit 0` generano un interrupt e vengono puliti esclusivamente dopo una lettura APB da parte dell'utente. Infatti, questo registro è realizzato con flip-flop di tipo **SET-RESET** dove l'impulso di SET è generato dalla FSM generale, mentre quello di RESET dall'interfaccia APB. Per quanto riguarda i campi **RX_READY[7:0]**, questi verranno ripuliti per l'appunto solo se l'utente accede in lettura al registro corrispondente. Se questo non viene fatto e durante la transizione venisse riscritto un registro prima che il dato venisse letto, allora verrà generato il flag **RX_OVERRUN** per segnalare la perdita di un dato.

Capitolo 5

Simulazione RTL

Passiamo ora alla fase di simulazione in cui finalmente testeremo il corretto funzionamento della nostra IP appena sviluppata. L'ambiente nel quale lavoreremo sarà un ambiente di verifica **UVM** realizzato dal **reparto IC di Eggtronic**. In esso, possiamo sfruttare la presenza dei cosiddetti **"agenti"**, ciascun dei quali si comporterà come un dispositivo comunicante con uno dei tre standard seriali che vogliamo supportare; nello specifico, comunicheremo con:

- **UART agent**: simulatore di una *UART full-duplex*.
- **I²C Master agent**: emulatore di un *Master* I^2C.
- **I²C Slave agent**: simulatore di uno *Slave* I^2C.

Per quanto riguarda l'*SPI*, faremo comunicare la nostra IP con un *SPI-Master* sviluppato dal team open-source **PULP**, nato da una collaborazione tra l'Università di Bologna e ETH di Zurigo. Oltre a questo, potremo anche simulare con una certa accuratezza un utente che va a scrivere i registri di configurazione per eseguire una specifica operazione, grazie all'ambiente UVM. È infatti presente in esso il cosiddetto `ral` che consiste ad una copia locale di tutta la Register-Map alla quale possiamo apportare facilmente modifiche per poi copiarle concretamente nella IP attraverso l'interfaccia APB, configurando così la nostra periferica. Nel mentre, il **monitor UVM** verificherà istante per istante che rispettiamo le specifiche di ogni protocollo.

5.1 Simulazione UART

Come primo caso, partiamo dalla simulazione della IP configurata come *UART full-duplex*. Nella figura sottostante abbiamo riportato solo i registri richiesti per

configurare la IP in questa modalità:

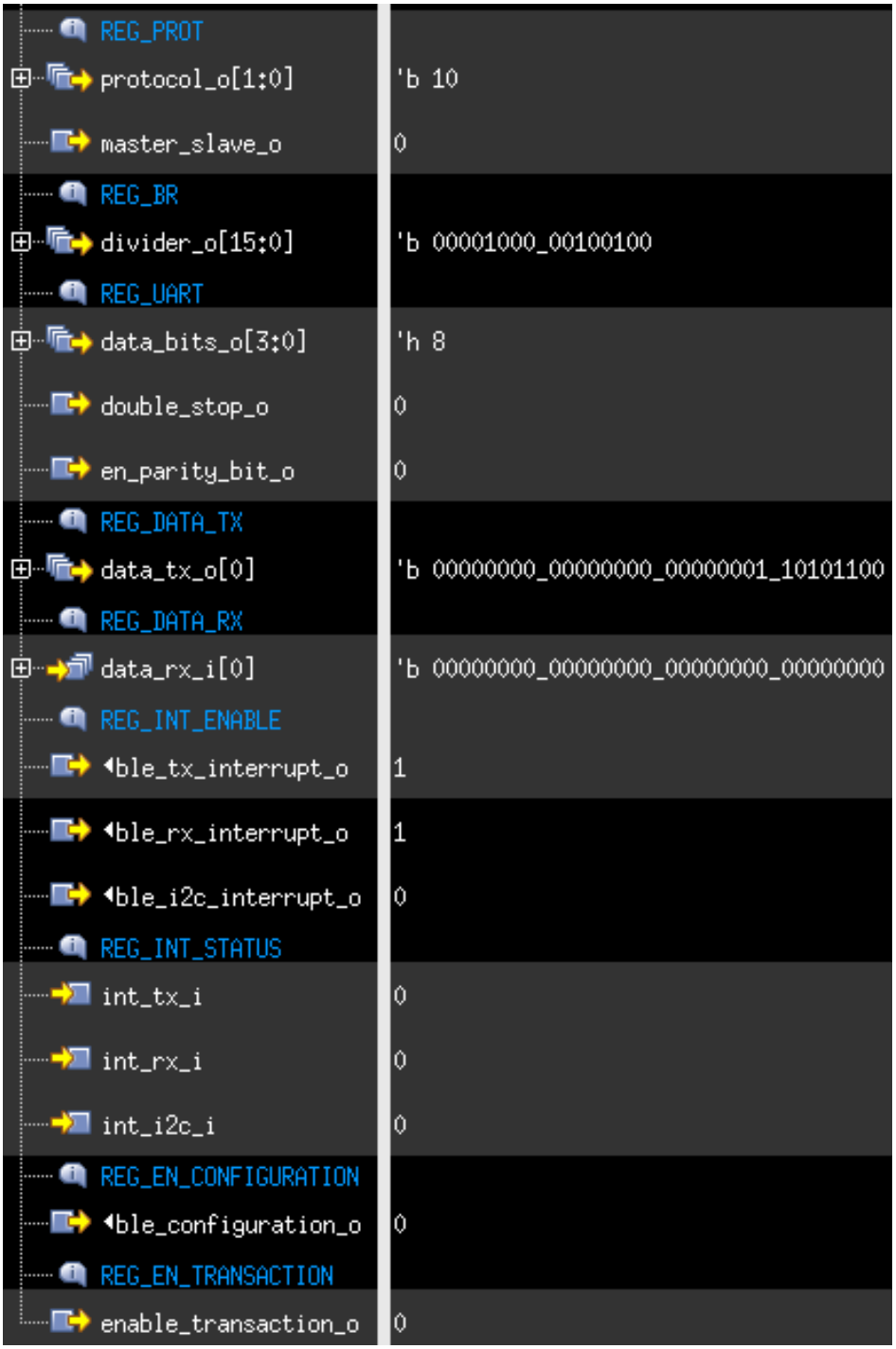


Figura 5.1: Configurazione registri APB come *UART*

Nella Figura 5.1 sotto al nome di ciascun registro sono riportati i campi a partire dalla posizione zero fino a salire e osserviamo come sono stati scritti i seguenti parametri:

- **REG_PROT = 3'b010**: il valore 3'b010 scritto in quel registro porta la macchina a stati ad entrare nello stato di IDLE della *UART-RX* nel momento in cui riceve il segnale `enable_configuration`. Per quanto riguarda il terzo bit in realtà non ci interessa dato che come abbiamo visto nel capitolo precedente, tramite di esso indicheremo se configurare la periferica come *Master* o *Slave* e nel caso corrente non ci interessa.

- **REG_BR = 16'b00001000_00100100**: questo valore del divisore porta il Baud-Rate operativo ad essere 9600 bit/s. La motivazione di questa scelta sta nel fatto che il clock delle periferica è stato impostato a 20 MHz e per rispettare il Baud-Rate scelto dovremo scorrere un bit ogni $\frac{2 \cdot 10^7}{9600} \approx 2084$ colpi di clock. Ovviamente, configureremo anche l'agente *UART* per trasmettere e ricevere i pacchetti dati a quella frequenza.

- **REG_UART = 6'b001000**: in questo modo andiamo a disabilitare sia il doppio STOP-BIT che il bit di parità (MSB e MSB-1 a zero); inoltre andiamo a settare il pacchetto dati a 8 bit.

- **REG_DATA_TX = 9'b1_10101100**: in realtà questo registro è composto da 32 bit ma siccome il numero massimo di bit che possono comporre un pacchetto dati *UART* è nove, scriveremo solo quelli. Inoltre, avendo configurato il pacchetto dati a 8 bit, quello più significativo dei nove scritti verrà ignorato dalla *UART-TX* e non verrà caricato in `transmit_reg`.

- **REG_INT_ENABLE = 3'b011**: operando con una *UART full-duplex*, abilitiamo gli interrupt dovuti sia alla trasmissione che alla ricezione.

E verso la fine abilitiamo la configurazione per poi far partire una transizione per quel che riguarda il trasmettitore (il ricevitore invece è in attesa di riconoscere una transizione avviata dall'agente *UART* che partirà più o meno contemporaneamente alla nostra). Passiamo ora a vedere i segnali della *UART interface* con la quale facciamo comunicare il **DUT** (la nostra IP) e l'agente *UART*:

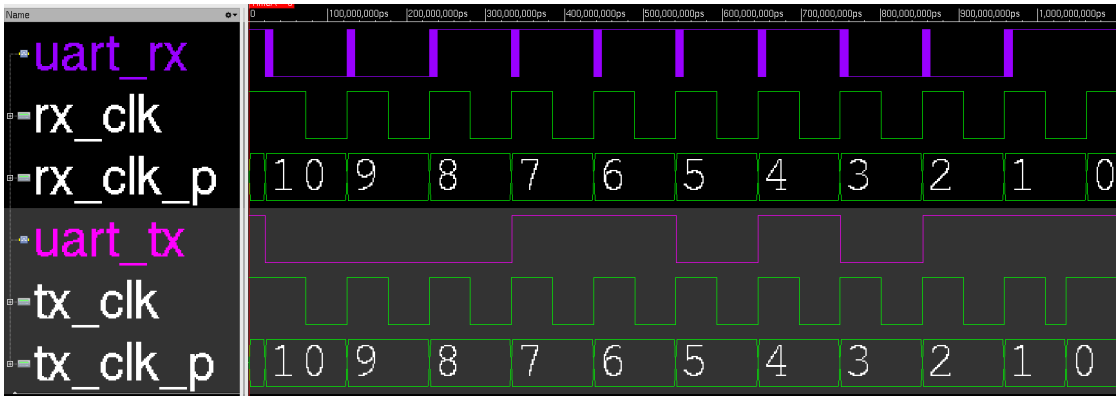


Figura 5.2: *UART* interface

I nomi dei segnali prendono come riferimento il punto di vista del DUT. Di conseguenza, i segnali UART_RX e UART_TX si riferiscono rispettivamente al nostro ingresso del ricevitore e all'uscita del trasmettitore. Partiamo da quest'ultimo: sempre nella Figura 5.2, notiamo come la linea UART_TX all'inizio sia a riposo e poco dopo parte lo START-BIT. Successivamente possiamo identificare, partendo dal LSB, ogni simbolo costituente del nostro pacchetto dati ovvero **8'b10101100** (in esadecimale **'hAC**) per poi terminare con lo STOP-BIT, e lo possiamo vedere direttamente dalla *UART_TX* interna a RDIP:

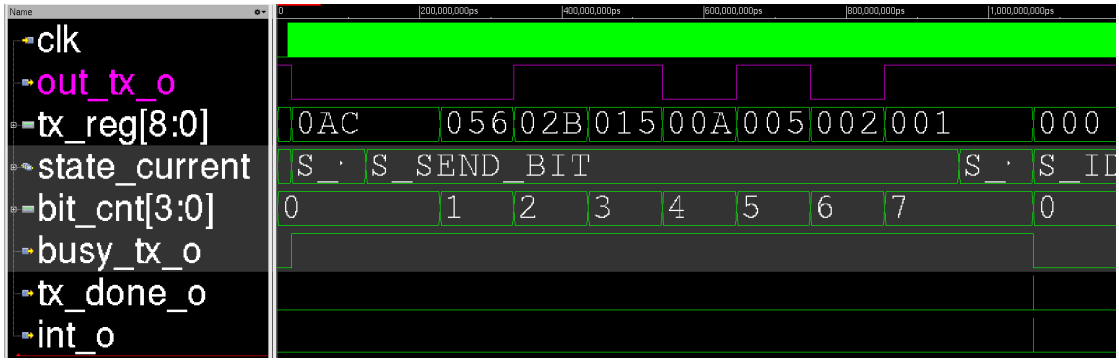


Figura 5.3: *UART-TX*

Come prova del nove, andiamo a vedere il log di simulazione che riporta il seguente dato raccolto dall'agente *UART* per confermare la sua corretta ricezione:

```
----------------------------------
Name       Type        Size  Value
----------------------------------
item       uart_item   -     @11862
  data     integral    8     'hac
  parity   integral    1     'h0
----------------------------------
```

Figura 5.4: *log* di simulazione: caso *UART-TX*

Nel nostro ingresso RX invece osserviamo come ci sia un certo **jitter** tra un simbolo e l'altro; questa è una proprietà dell'agente che ci permette di rendere più realistica questa linea di dato nella transizione da un simbolo all'altro.

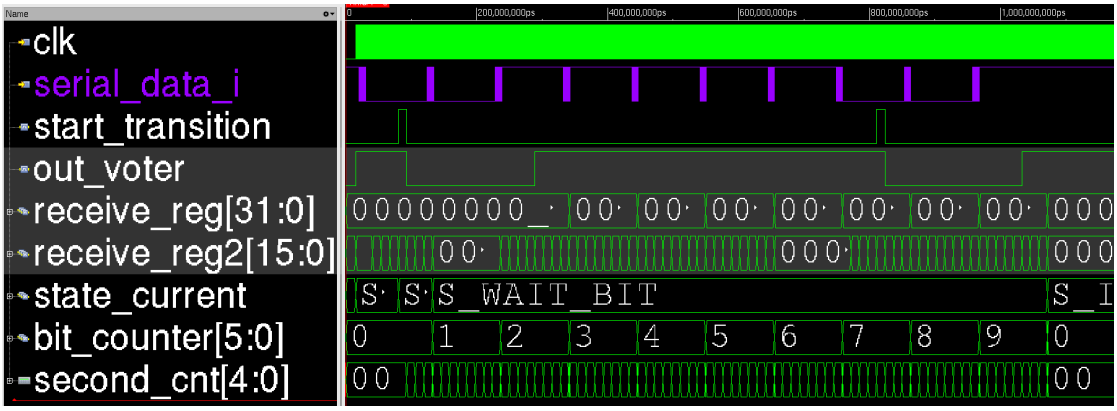


Figura 5.5: FSM generale operante come *UART-RX*

Nella Figura 5.5 riportiamo i segnali della macchina a stati generale configurata come *UART*-ricevitore e notiamo in ordine cronologico i seguenti eventi:

- Passaggio allo stato **S_WAIT_MIDDLE** nel momento in cui riconosciamo correttamente l'inizio di una transizione (segnale `start_transition` alto).
- Attendiamo di ricevere altri quattro campioni nel `receive_reg2` per assicurarsi di posizionarsi al centro del simbolo e quindi passere allo stato **S_WAIT_BIT** in cui attendiamo di ricevere tutti i bit componenti il pacchetto dati.
- Ritorno in IDLE e scrittura del registro **REG_DATA_RX** con il pacchetto appena ricevuto, mostrato nella Figura 5.6.

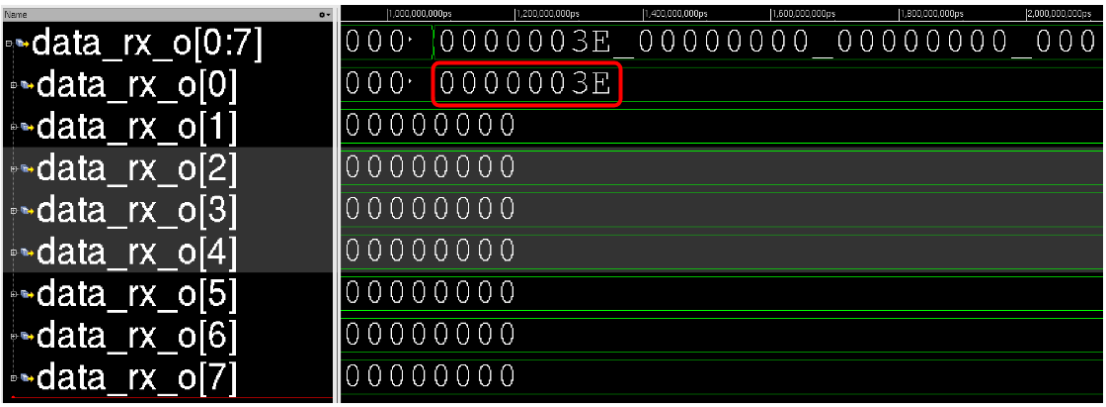


Figura 5.6: Dato salvato nel registro in uscita

Nonostante la presenza del *jitter*, grazie al sovra-campionamento e all'utilizzo del votatore a maggioranza, il dato ricevuto finale è pari a **'h0000003E** e corrispondente a quello inviato dall'agente *UART*, confermato anche dal *log* di simulazione:

```
----------------------------------
Name       Type        Size  Value
----------------------------------
item       uart_item   -     @11838
  data     integral    8     'h3e
  parity   integral    1     'h0
----------------------------------
```

Figura 5.7: *log* di simulazione: caso *UART-RX*

5.2 Simulazione I2C

Passiamo ora alla simulazione del bus I^2C; per simulare il comportamento di un filo di tipo Open-Drain con il resistore di pull-up, è stato inserito nel test-bench un convertitore di segnali in ingresso e **output enable** in porte three-state **inout** per una facile integrazione di oggetti connessi al bus.

5.2.1 I2C-Master

Proviamo ora ad effettuare una lettura come *Master* I^2C ad un dispositivo *Slave*. Analogamente alla simulazione precedente, andiamo a vedere come sono stati configurati tutti i parametri di interesse per eseguire l'operazione desiderata:

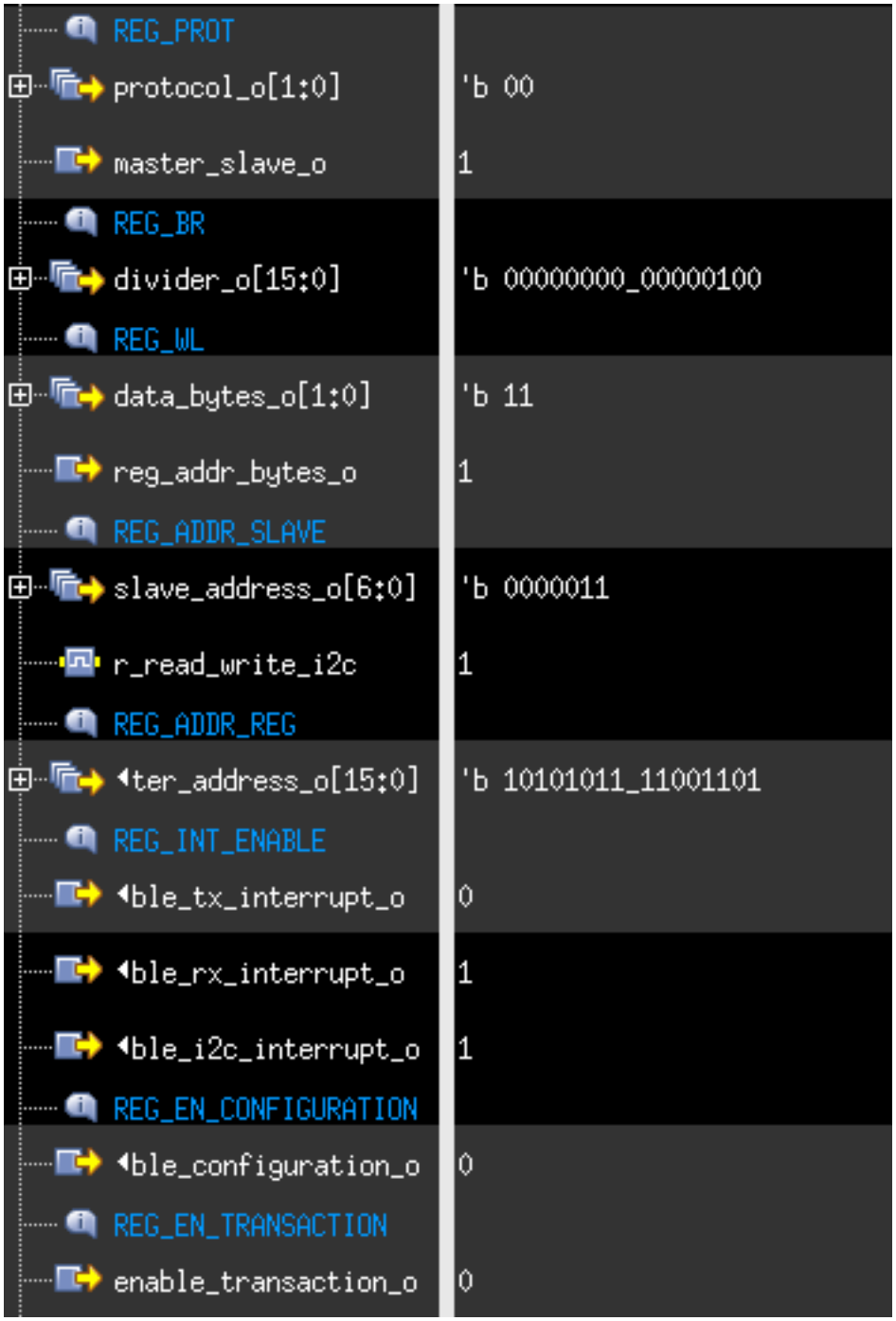


Figura 5.8: Configurazione registri APB come I^2C *Master*

Dalla Figura 5.8 vediamo che:

- **REG_PROT = 3'b100**: a differenza della UART, la FSM generale è ora sensibile al terzo bit che la porterà ad entrare nello stato **S_IDLE_I2C_MASTER** una volta asserito `enable_configuration`.
- **REG_BR = 16'b00000000_00000100**: il valore binario scritto corrisponde a 4 in decimale e rifacendoci alla Equazione 4.1 otteniamo che $f_{SCL} = \frac{20MHz}{20} = 1MHz$ che corrisponde alla modalità **Fast-Mode-Plus**.
- **REG_WL = 3'b111**: con MSB=1 andiamo a configurare a due Byte l'indirizzo del registro/locazione di memoria che vogliamo accedere mentre gli altri due pari a 'b11='d3 andremo a configurare il dato vero e proprio con quattro Byte.
- **REG_ADDR_SLAVE = 8'b10000011**: qui MSB=1 significa che l'operazione finale che andremo ad eseguire sarà una lettura; i rimanenti sette bit sono invece l'indirizzo con il quale vogliamo comunicare (3).
- **REG_ADDR_REG = 16'b10101011_11001101**: questo è l'indirizzo del registro dello slave al quale vogliamo accedere.
- **REG_INT_ENABLE = 3'b110**: siccome siamo interessati ad una lettura e stiamo comunicando con un I^2C, abilitiamo gli interrupt dovuti al lato ricevitore e al bus corrispondente.

Tralasciamo la parte in cui abilitiamo la configurazione selezionata. Riportiamo ora i segnali della macchina a stati generale mostrando le sequenze più importanti:

Figura 5.9: Invio indirizzo *Slave* + R/W = 0

Per una semplice identificazione, abbiamo evidenziato con diversi colori le linee scl_o e sda_o che hanno la funzione di indicare le uscite SCL e SDA, rispettivamente. La

simulazione inizia (Figura 5.9) ovviamente dal momento in cui inviamo la condizione di START sul bus e subito dopo la FSM entra nello stato **S_WRITE_ADDR_W** nel quale invia l'indirizzo dello *Slave* con comando R/W=0. Fatto ciò, si passa in uno stato (**S_CHECK_ACK**) in cui la macchina lascia il controllo della linea SDA e subentra l'agente *I2C Slave* che abbasserà la linea per inviare l'ACK; da notare infatti come sda_o != sda_i. Una volta trasmesso l'indirizzo della locazione di memoria da accedere, si procede con il RESTART, mostrato nella Figura 5.10:

Figura 5.10: Invio indirizzo *Slave* + R/W = 1

Da questo momento in poi, sarà lo *Slave* a pilotare SDA per trasmettere il dato che vogliamo leggere e noi ad inviare l'ACK dopo ogni Byte:

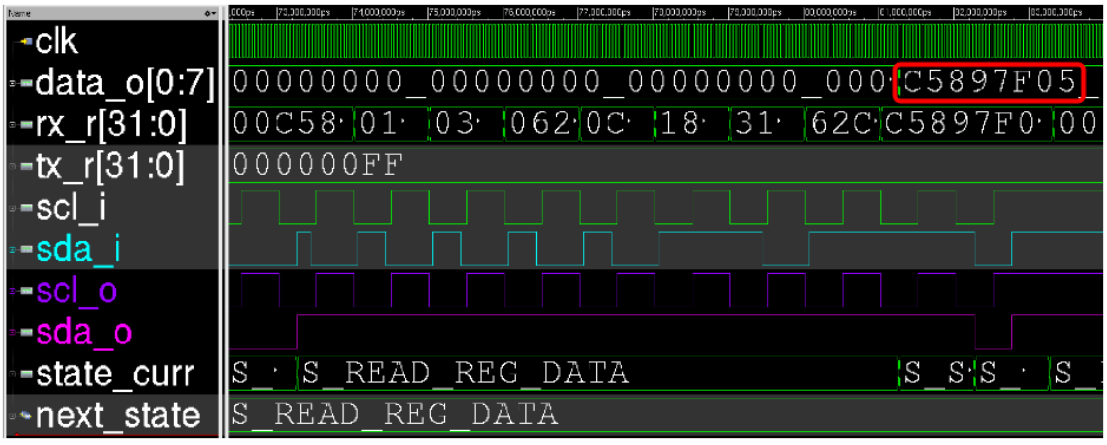


Figura 5.11: Lettura ultimo Byte e invio dato in uscita

Nella Figura 5.11 viene mostrata la ricezione dell'ultimo dei quattro Byte che

compongono la parola e infine il passaggio del contenuto di `receive_reg` (indicato con `rx_r`) in uscita pari a **'hC5897F05**. Per confermare di aver ricevuto il dato corretto, riportiamo le linee del *log* di simulazione che indica il dato inviato sul bus:

```
[uvm_test_top.env.i2c_agent_d.driver] Send readback data c5 1°
[uvm_test_top.env.i2c_agent_d.driver] Send readback data 89 2°
[uvm_test_top.env.i2c_agent_d.driver] Send readback data 7f 3°
[uvm_test_top.env.i2c_agent_d.driver] Send readback data 5  4°
```

Figura 5.12: *log* di simulazione: dato inviato dall'agente *I2C Slave*

5.2.2 I2C-Slave

Per completare questo protocollo, simuliamo il caso in cui la nostra IP venga configurata come *Slave*. In questa situazione, i registri da configurare sono solo due: REG_PROT e REG_WL: nel primo verrà settato il bit più significativo a zero invece di uno, nell'altro configureremo la lunghezza del dato sempre a quattro Byte ma quella dell'indirizzo a singolo Byte. L'operazione che vogliamo eseguire è la scrittura in una locazione di memoria della nostra macchina a stati. Nel capitolo precedente è stata menzionata la presenza di una piccola memoria ad 8 locazioni proprio per andare a simulare uno *Slave* di qualsiasi tipo dotato di uno spazio di indirizzamento. Siccome per ragioni pratiche è questo spazio è limitato a otto, allora andremo a configurare l'agente *I2C-Master* in modo tale che possa inviare Byte contenuti nell'intervallo [0:7]. Di seguito riportiamo le forme d'onda dove dopo lo START, la macchina a stati passa da riposo (IDLE) allo stato **S_READ_SLAVE_ADDRESS** in cui legge l'indirizzo dello *Slave* trasmesso sul bus dall'agente UVM:

Figura 5.13: Fase di indirizzamento: FSMg configurata come I^2C *Slave*

Ricevuti gli otto bit, la macchina riconosce che l'indirizzo corrisponde al proprio e verso la fine della Figura 5.13 invia il segnale di ACK (sda_o = 0). La transizione si conclude con la ricezione da parte nostra dell'ultimo Byte inviato dal *Master* e il passaggio del dato in uscita pari a **'h03050005** (segnale `out_d[5]` nella Figura 5.14).

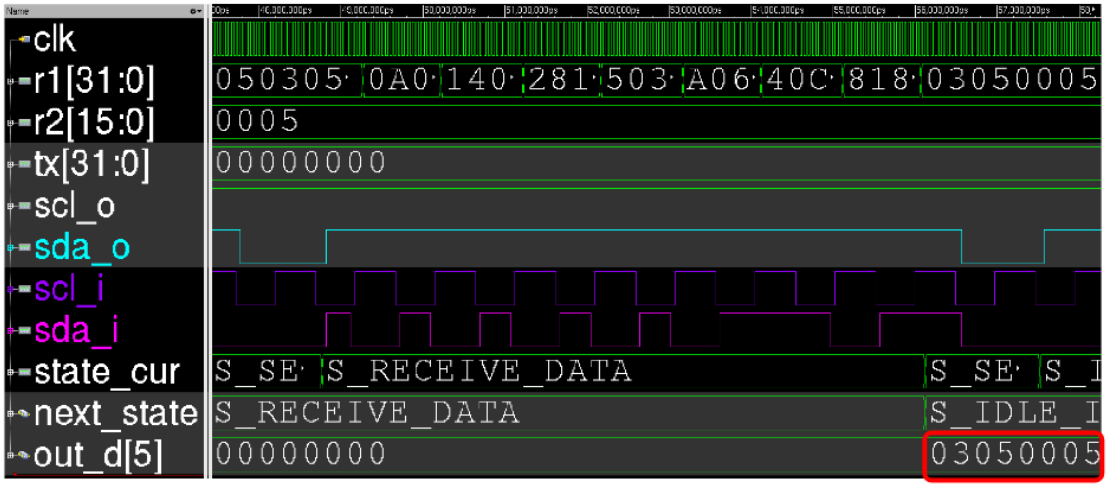


Figura 5.14: Ricezione ultimo Byte: FSMg configurata come *I²C Slave*

Come nel caso precedente, controlliamo il *log* di simulazione per verificare che il dato ricevuto sia quello corretto:

```
[uvm_test_top.env.i2c_agent_h.driver] Driving host item 0x03 1°
[uvm_test_top.env.i2c_agent_h.driver] wait_cycles 0x00000008
[uvm_test_top.env.i2c_agent_h.driver] drv: HostData
[uvm_test_top.env.i2c_agent_h.driver] Driving host item 0x05 2°
[uvm_test_top.env.i2c_agent_h.driver] wait_cycles 0x00000008
[uvm_test_top.env.i2c_agent_h.driver] drv: HostData
[uvm_test_top.env.i2c_agent_h.driver] Driving host item 0x00 3°
[uvm_test_top.env.i2c_agent_h.driver] wait_cycles 0x00000008
[uvm_test_top.env.i2c_agent_h.driver] drv: HostData
[uvm_test_top.env.i2c_agent_h.driver] Driving host item 0x05 4°
[uvm_test_top.env.i2c_agent_h.driver] wait_cycles 0x00000008
[uvm_test_top.env.i2c_agent_h.driver] drv: HostStop
```

Figura 5.15: Ricezione ultimo Byte: FSMg configurata come *I²C Slave*

5.3 Simulazione SPI

Concludiamo questa parte con l'ultimo protocollo da verificare, ovvero l'*SPI*. Prima di iniziare, dobbiamo fare una piccola introduzione di come è stato pensato il funzionamento della nostra periferica *SPI Slave*. Infatti, a differenza del bus precedente, il tipo di informazione inviata sulle linee di dato seriale MOSI e MISO non è specificato (nel bus I^2C si partiva subito con l'indirizzo dello *Slave* e il comando R/W), come è stato già menzionato nel capitolo 3. Di conseguenza, c'è molta più varietà dal punto di vista del tipo di informazioni scambiate tra due oggetti e di come questi comunichino tra loro. Proprio per questo motivo nel registro di configurazione **REG_SPI** 4.4.2 è stata aggiunta la possibilità di configurare il tipo di dato da inviare sul bus (lato *Master*) con il campo **DATA_CONFIGURATION**. Dall'altra parte, questo sarà definito a priori e dipenderà da periferica a periferica. Nella nostra implementazione, si è pensato dunque di definire un piccolo set di istruzioni a otto bit eseguibili dal nostro *Slave* e sono:

- **ERASE MEMORY** codificata con la sequenza **8'b00000001**: questa istruzione setta tutti i bit di tutte le locazioni di memoria ad 1 non appena il segnale `slave_select` ritorna all'uno logico (dopo ovviamente aver decodificato l'istruzione appena ricevuta).

- **READ MEMORY LOCATION** codificata con la sequenza **8'b00000010**: questa istruzione permette al *Master* di accedere ad una specifica locazione di memoria interna. Infatti, ricevuta correttamente l'istruzione, lo *Slave* si aspettera di ricevere successivamente un indirizzo.

- **WRITE MEMORY LOCATION** codificata con la sequenza **8'b00000011**: diversamente da quella precedente, questa istruzione accedere ad una locazione di memoria per una scrittura.

- **READ BURST** codificata con la sequenza **8'b00000100**: in questo caso effettueremo una lettura **burst** a partire dalla locazione 0. Infatti, non è previsto l'invio di un indirizzo di partenza. La comunicazione sarà poi interrotta al fronte di salita dello `slave_select`.

- **WRITE BURST** codificata con la sequenza **8'b00000101**: analogamente all'istruzione precedente, la scrittura delle locazioni di memoria avviene una dopo l'altra a partire dalla prima e si interromperà al fronte di salita dello `slave_select`.

Andiamo a vedere come sono stati configurati i registri APB in questa simulazione:

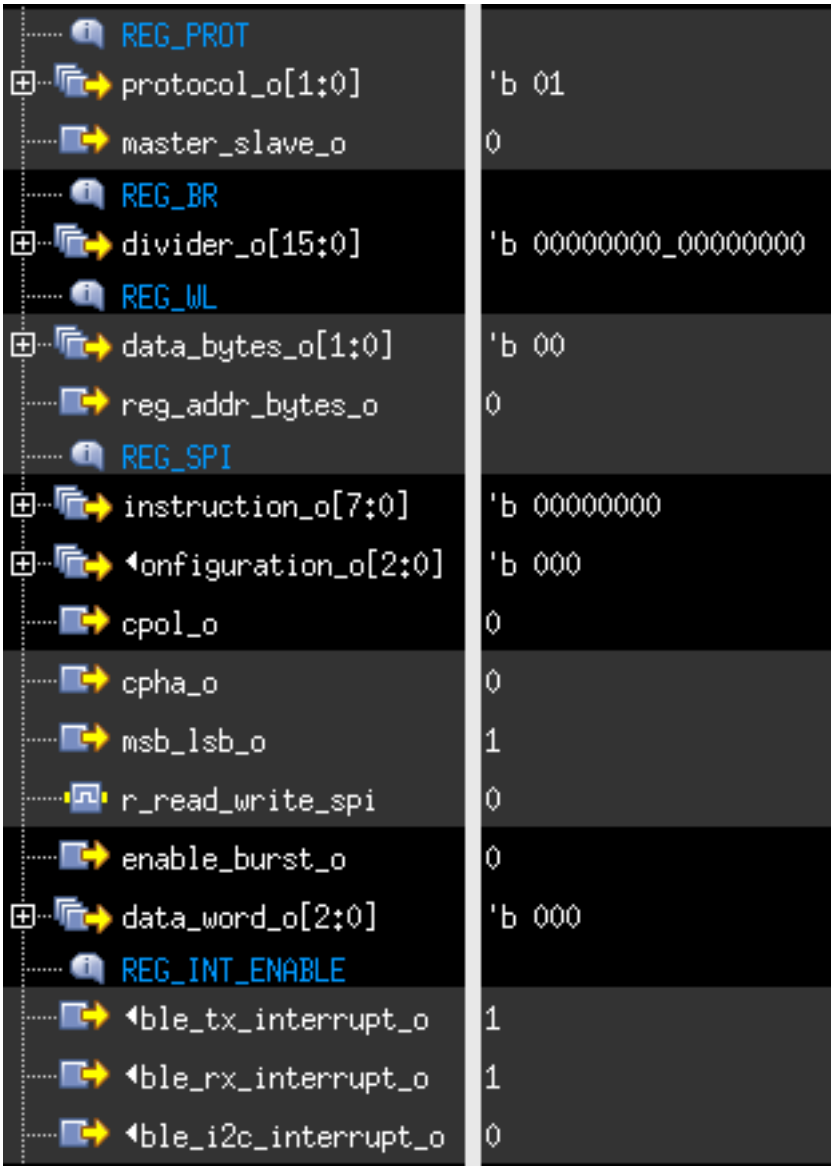


Figura 5.16: Configurazione registri APB come *SPI Slave*

Dalla Figura 5.16 abbiamo:

- **REG_PROT = 3'b001**: i primi due bit selezionano la modalità *SPI* mentre con il terzo a zero configuriamo la IP come *Slave*.
- **REG_BR = 16'b00000000_00000000**: essendo *Slave*, non saremo noi a pilotare la linea del clock seriale ma sarà il *Master* esterno ad occuparsene. In

virtù di questo, il valore scritto in questo registro non avrà alcun effetto sul Baud-Rate della comunicazione.

- **REG_WL = 3'b000**: questa volta selezioniamo un singolo Byte sia lunghezza dell'indirizzo del registro che la lunghezza del dato.
- **REG_SPI = 19'b000_00100000_00000000**: nel caso in questione, siamo interessati solo a configurare tre campi, ovvero i parametri **CPHA = 0**, **CPOL = 0** e **MSB_LSB = 1** per selezionare la modalità operativa. Infatti, il dispositivo *SPI-Master* di PULP con il quale comunichiamo può operare solamente in questo modo.
- **REG_INT_ENABLE = 3'b011**: abilitiamo gli interrupt da entrambi i lati.

L'operazione che si vuole effettuare è una lettura dalla nostra IP del primo Byte della locazione di memoria **5** tra le seguenti:

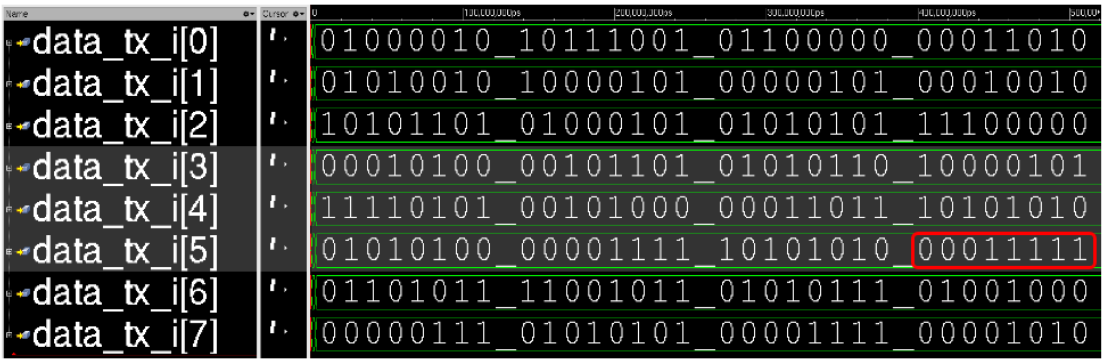


Figura 5.17: Locazioni di memoria

Figura 5.18: FSMg operante come *SPI-Slave*

È possibile osservare subito come il clock seriale (indicato nella Figura 5.18 come sclk_i) inizi a commutare dopo che il *Master* dall'altro lato abbassa lo `slave_select` in ingresso. Successivamente, la FSM generale entrerà nello stato **S_INSTRUCTION** dove per l'appunto si aspetta di ricevere l'istruzione a 8 bit dalla linea MOSI (indicata come mosi_i); dopo infatti otto colpi di clock, l'istruzione decodificata è proprio **8'b00000010**. Si passa dunque allo stato **S_ADDRESS** dove ci aspettiamo l'indirizzo della locazione interna da leggere e una volta ricevuto quest'ultimo (nel registro `receive_reg2` indicato come `r2` viene scritto il valore 5), entriamo in **S_DATA** dove possiamo vedere sulla linea MISO (indicata come miso_o) venir trasmesso il dato **'b00011111**, vale a dire il primo Byte della locazione 5 della memoria interna. Il dato raccolto dall'altra parte risulta essere:

Figura 5.19: Dato ricevuto dal *SPI Master*

Il che coincide con quello memorizzato nella memoria interna della nostra IP.

Capitolo 6

Sintesi logica

Una volta verificato il corretto funzionamento della IP appena sviluppata, terminiamo il lavoro di questa tesi con la sintesi logica per valutare l'area occupata e il consumo di potenza. Tuttavia, per contestualizzare i numeri ottenuti e attribuirgli un valore concreto, sarà necessario paragonarli a quelli riguardanti la sintesi di IP singole che implementano un solo Standard seriale, giungendo cosi ad un raffronto il più possibile valido tra la nostra soluzione riconfigurabile e quella Standard.
Infatti, prima di iniziare la fase di sviluppo RTL di RDIP, sono stati implementate singolarmente delle IP dedicate appunto a ciascun protocollo, separando anche lato Master/Slave per I^2C e *SPI* e TX/RX per *UART*. Da queste, una volta verificato il loro corretto funzionamento tramite simulazione, è stato poi preso spunto per la stesura del codice RTL del progetto finale; in questo modo, rafforziamo la validità del confronto dei numeri delle varie IP dato che queste avranno la medesima logica interna per il controllo e gli stessi componenti interni per l'implementazione.
La sintesi è stata effettuata adoperando una tecnologia a **180nm** con la cella base **NAND2** che occupa uno spazio di $\mathbf{8{,}78}\boldsymbol{\mu m^2}$; questo ci servirà per capire il numero di porte allocate per stanziare ciascun macro-blocco in ottica di una futura scalabilità. Inoltre, il clock utilizzato per questa sintesi è 40MHz per il sistema e 20MHz per la periferica, utile per l'analisi della tempistica statica (static timing analysis) e controllare che non ci siano violazioni dei tempi di **setup** nelle situazioni in cui il processo tecnologico e le condizioni di temperatura e la tensione di alimentazione siano meno favorevoli (**worst-case**).

6.1 Descrizione blocchi interni

Prima di procedere, descriviamo come è stata organizzata la gerarchia dei blocchi interni dell'IP.

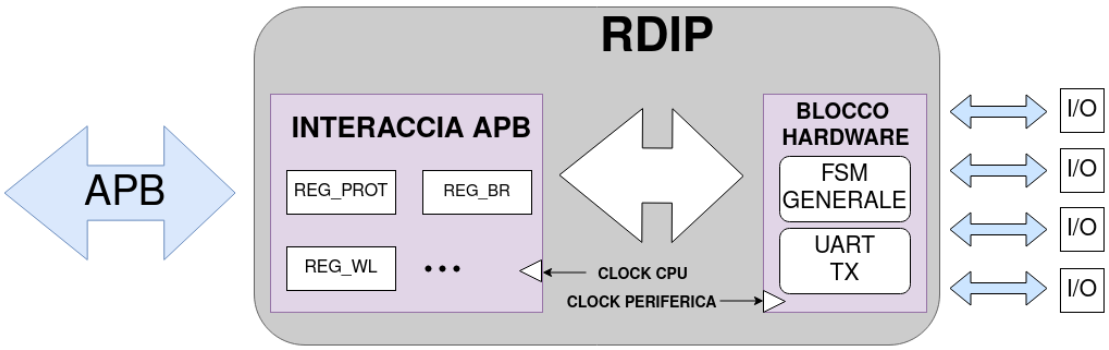


Figura 6.1: Diagramma a blocchi RDIP

Dalla Figura 6.1 vediamo come siano presenti tre macro blocchi:

- **Interfaccia APB**: contiene tutti i registri CSR e di dato, oltre che al generatore di interrupt.
- **FSM generale**: macchina a stati riconfigurabile per supportare tutti i e tre i protocolli di comunicazione (eccetto il trasmettitore *UART*).
- **UART-TX**: piccola macchina a stati per supportare la parte mancante del trasmettitore abilitata solo nel caso in cui volessimo comunicare con una *UART* esterna.

Notiamo anche come questi lavorino con due segnali di clock: **clock cpu** e **clock periferica** (quest'ultimo per il blocco hardware che include la FSM e la *UART-TX*). Il rapporto tra le loro frequenze sarà imponibile dall'utente e per evitare di incombere a possibili malfunzionamenti della IP, il clock cpu deve essere maggiore o uguale a clock periferica.
Rimanendo in tema, nel capitolo 2 abbiamo visto come l'uso di un sistema dotato di architettura riconfigurabile può portare ad un incremento del consumo di potenza rispetto ad uno non riconfigurabile che esegue una medesima operazione. Per far fronte a ciò, nel nostro progetto sono state inserite anche delle celle di **clock gating** ed è stato fatto a ciascuno dei tre macro blocchi. Il codice *SystemVerilog* di queste celle è il seguente:

```
module pulp_clock_gating (
    input  logic clk_i,
    input  logic en_i,
    input  logic test_en_i,
    output logic clk_o
);

  logic clk_en;

  always_latch begin
    if (clk_i == 1'b0) clk_en <= en_i | test_en_i;
  end

  assign clk_o = clk_i & clk_en;

endmodule
```

Il segnale di abilitazione delle celle (`en_i`) è diverso per ciascuno dei tre macro blocchi. Per l'interfaccia APB questo proviene direttamente dal lato utente; diverso è invece il discorso per la FSM generale alla quale arriverà il clock solo nel momento in cui abiliteremo la configurazione appena selezionata dall'utente. Infine, la `uart_tx` verrà abilitata nel caso in cui venga riconosciuta la scrittura del campo **PROT = 2'b10** nel registro 4.4.1 e l'abilitazione di una transizione. Per quanto riguarda l'altro segnale `test_en_i`, non lo useremo dato che questo è usato in ambito **DFT** (Design for Test) che va oltre lo scopo di questo elaborato. In parallelo alla inserzione "manuale" delle celle di clock-gating, anche il sintetizzatore introdurrà a sua volta della celle a livello di singolo flip-flop laddove riconosca un effettivo risparmio di potenza consumata rispetto all'incremento di area, raggruppando gruppi di flip-flop che hanno la stessa condizione di abilitazione con un'unica cella. Questa opzione è abilitata a tutte le sintesi che porteremo avanti da qui in poi, anche alle periferiche che implementano un singolo bus.

6.2 Report Area

Iniziamo dai "**report area**" che ci danno un resoconto dell'area totale e di come questa sia suddivisa tra logica combinatoria, inverter, flip-flop e clock-gate nei vari blocchi che compongono il modulo sintetizzato.

6.2.1 RDIP

I valori di area occupata dalla nostra periferica e da ciascun macro-blocco al suo interno sono riportati nella seguente tabella in **numero di porte logiche equivalenti**, unità di misura che ha più senso nel contesto di una futura scalabilità con un'altra tecnologia:

Modulo	Area totale	Combinatoria	Inverter	Flip-Flop	Clock-Gate
`rdip`	9014	3986	156	4536	336
`/apb_int`	2696	736	8	1884	68
`/clk_gating_apb`	4	0	0	0	4
`/clk_gating_fsm`	4	0	0	0	4
`/clk_gating_uart_tx`	4	0	0	0	4
`/fsm`	5650	2976	135	2310	229
`/reg_int_status`	16	0	0	16	0
`/reg_status_rx`	83	24	0	55	4
`/reg_status_tx`	21	5	0	16	0
`/uart_tx`	487	223	12	231	21

Tabella 6.1: Report area: RDIP

Dalla Tabella 6.1 scopriamo che il numero di porte logiche necessarie per implementare questa IP è dunque **9014** che corrispondono a $\mathbf{79143{,}545 \mu m^2}$ con la tecnologia usata per la sintesi. È interessante anche notare come 298 porte costituiscono le celle di clock gating ma solo 12 sono inserite da noi manualmente a livello di blocchi, tutte le altre solo allocate automaticamente dal sintetizzatore. Per quanto riguarda i contributi di ciascun modulo al totale, è chiaro che `fsm` (che corrisponde alla FSM generale nella Figura 6.1) è quella che da il contributo maggiore, soprattutto la logica combinatoria risulta maggiore (2973 porte) di quella sequenziale (2312 porte) dato che la macchina è composta da 36 stati, nonostante inglobi al suo interno tutti registri e i contatori usati per supportare tutte le modalità di funzionamento.

6.2.2 IP singole

Sintetizzata RDIP, passiamo ora alle piccole FSM che implementano un singolo protocollo. Nella seguente tabella, sono riportati i valori estratti dal report area:

Modulo	Area totale	Combinatoria	Inverter	Flip-Flop	Clock-Gate
`i2c_master`	2580	512	24	1980	63
`i2c_slave`	2197	484	35	1617	59
`spi_master`	3555	1217	55	2096	187
`spi_slave`	3548	1284	40	2036	188
`uart_tx`	486	217	15	237	17
`uart_rx`	1175	340	16	760	59

Tabella 6.2: Report area: IP singole

Per prima cosa, dalla tabella soprastante notiamo subito come il fatto di aver deciso di allocare una `uart_tx` all'interno del blocco hardware di RDIP e far supportare il ricevitore dalla `fsm` invece che il contrario si è rilevata essere una scelta accurata, essendo il numero totale di porte logiche equivalenti della `uart_rx` più del doppio rispetto all'altra.
Sommando l'area totale di tutte queste IP otteniamo un numero di porte logiche pari a **13541**. Di conseguenza, la percentuale di area allocata per stanziare RDIP rispetto all'area totale occupata da tutte le IP singole corrisponde al:

$$\frac{\mathbf{9014}}{\mathbf{13541/100}} \approx \mathbf{66{,}6\%} \tag{6.1}$$

Ovvero risparmiamo il **33,4%** di porte logiche. Tuttavia, le periferiche singole riportate nella Tabella 6.2 sono state sintetizzate senza tenere in considerazione un eventuale interfaccia APB contente i registri CSR e di dato. Infatti, ciascuna macchina a stati nella Tabella 6.2 è dotata delle medesime configurazioni disponibili per RDIP inerenti allo specifico protocollo che supportano. È plausibile quindi immaginare che anche loro, se integrate in un sistema con un micro-controllore, avranno a disposizione un'interfaccia APB dedicata contenete i registri di configurazione e di dato. Per questo motivo, è più corretto fare un paragone andando a sottrarre l'area dovuta all'interfaccia APB e ai registri di stato e considerare solo quella della `fsm` e della `uart_tx` nella Tabella 6.1. Dunque, ricalcolando il rapporto otteniamo:

$$\frac{\mathbf{6137}}{\mathbf{13541/100}} \approx \mathbf{45{,}3\%} \tag{6.2}$$

Che si traduce in un risparmio del **54,7%**, un risultato molto importante in senno all'idea di principio di questo progetto che come scopo ultimo ha proprio quello di voler ottimizzare l'area occupata al fine di minimizzarla.

6.2.3 Confronto con interfacciamento APB

Se invece volessimo includere lo spazio allocato per l'interfacciamento APB, in RDIP anche questo è unico e alcuni registri di dato e di configurazione, come ovviamente quelli di stato, sono condivisi tra i tre Standard seriali. Tenendo in considerazione come è stato ideato il salvataggio dei dati ricevuti dall'esterno e la fase di indirizzamento di uno *Slave* in RDIP, il contributo principale nel caso di periferiche singole verrebbe proprio dai registri di dato dove andiamo a salvare sia i dati ricevuti che quelli da trasmettere sul bus, e come abbiamo visto dalla Tabella 4.1 nel Capitolo 4, questi sono in totale 16 da 32 bit. È stata dunque fatta una stima il più possibile fedele e coerente di ipotetici numeri di flip-flop contenuti in un eventuale interfacciamento APB per le IP singole che **supporti le medesime funzionalità di RDIP** e li abbiamo riportati nella seguente tabella:

Modulo	Numero di Flip-Flop
`rdip`	606
`i2c_master`	350
`i2c_slave`	523
`spi_master`	580
`spi_slave`	529
`uart_tx`	57
`uart_rx`	289

Tabella 6.3: Numero di Flip-Flop nell'interfaccia APB

Come in RDIP, sia `i2c_master` che `uart_tx` ricevono in ingresso i dati da inviare sul bus da un solo registro. Per lo *Slave* è presente invece uno spazio di indirizzamento per la scrittura e uno per la lettura. Per questo, il numero di flip-flop di `i2c_slave` è nettamente maggiore di `i2c_master`. Il totale delle singole macchine è **2328** flip-flop contro i **606** di RDIP. Questo risulterebbe in un ulteriore risparmio di risorse stanziate e il vantaggio di una soluzione riconfigurabile sarebbe di molto maggiore, considerando anche il fatto che in questo paragono riguardante l'interfaccia APB stiamo paragonando il **numero di flip-flop** e non di porte logiche equivalenti.

6.3 Report Power

Concludiamo questo capitolo con il "**report power**", ovvero la stima di potenza consumata durante una particolare simulazione. Cercheremo di coprire tutti e tre i protocolli e faremo un confronto del consumo tra RDIP e le IP singole nell'esecuzione delle stesse operazioni e ovviamente alla stessa frequenza di modo che il paragone sia il più oggettivo possibile. Il report suddivide il totale nei vari macro-blocchi della gerarchia di un oggetto e distingue tre tipi di potenze:

- **Leakage**: potenza dovuta al consumo statico, ovvero alle correnti di perdita che i transistori presentano tra *Drain* e *Source* quando sono in teoria "spenti".
- **Internal**: potenza dovuta alla commutazione dei transistori interni alle celle.
- **Switching**: potenza dovuta alla carica e alla scarica della capacità di carico (equivalente) in uscita.

L'unità di misura con la quale sono riportate è il $\boldsymbol{\mu W}$. Eviteremo di includere nelle successive tabelle i contributi di potenza dovuti alle celle di clock gating per ragioni di semplicità e leggeliblità delle tabelle dato l'elevato numero.

6.3.1 Caso UART

Come per la simulazione RTL, partiamo dalla stima di potenza nel caso in cui configurassimo RDIP come *UART full-duplex*. Nel caso specifico, il `file.saif` (che tiene traccia dell'attività di tutti i nodi) è stato generato da una simulazione nella quale abbiamo eseguito **nove transizioni**, sia con il trasmettitore che con il ricevitore con i seguenti parametri:

- clock_cpu = 40MHz.
- clock_per = 20MHz.
- Baud-Rate = 9600 bit/s.
- Pacchetto dati a 8 bit, senza doppio stop e senza bit di parità.

I risultati delle due simulazioni sono riportati di seguito:

Modulo	Total	Leakage	Internal	Switching
`rdip`	5.33123e+02	2.23520e-01	4.59117e+02	7.37828e+01
`/fsm`	1.95083e+02	1.38589e-01	1.80483e+02	1.44613e+01
`/apb_int`	1.26131e+02	6.58935e-02	1.26050e+02	1.42605e-02
`/uart_tx`	7.32570e+01	1.35000e-02	6.63004e+01	6.94312e+00
`/reg_status_tx`	1.86838e+01	6.35942e-04	1.86831e+01	5.87990e-05
`/reg_int_status`	1.86836e+01	4.58016e-04	1.86831e+01	6.24420e-05
`/reg_status_rx`	1.57887e+01	2.66141e-03	1.57857e+01	3.81110e-04

Tabella 6.4: Potenza consumata RDIP: caso *UART full-duplex*

Il totale di RDIP risulta essere circa $\mathbf{533\mu W}$: di questi circa $\mathbf{86\mu W}$ sono consumati dalle celle di clock gating mentre il contributo principale è, come per l'area, la FSM generale che prende circa il 36% del totale. Per quanto riguarda le IP singole abbiamo che:

Oggetto	Total	Leakage	Internal	Switching
`uart_rx`	9.82131e+01	3.27767e-02	8.44813e+01	1.36991e+01
`uart_tx`	7.97378e+01	1.33158e-02	7.02760e+01	9.44857e+00

Tabella 6.5: Potenza consumata IP singole: caso *UART full-duplex*

Sommando le due IP singole abbiamo in totale $\mathbf{180\mu W}$ contro i $\mathbf{533\mu W}$ di RDIP. Bisogna fare della considerazioni riguardo al fatto che la soluzione riconfigurabile presenti un consumo di potenza nettamente maggiore: innanzitutto, come per l'area, per fare un paragone più equo bisognerebbe escludere l'APB e tutti i registri di configurazione e considerare solo la macchina a stati generale e la *UART-TX*. Infatti, gli ingressi delle due IP singole sono stati semplicemente stimolati tramite il testbench senza passare per una fase di configurazione dato che le interfacce APB per queste due non sono state sviluppate (come per le altre IP). Cosi facendo il consumo scenderebbe a $\mathbf{268\mu W}$. Rimane comunque più grande e questo può essere spiegato con:

- Dimensione del buffer RX della FSM generale molto più grande (32 bit) rispetto a quello della `uart_rx`. Lo scorrimento coinvolge dunque molti più flip-flop, portando ad un incremento di tutti e tre i contributi di potenza (internal, leakage e switching).
- La FSM generale possiede 36 stati a dispetto dei 3 della `uart_rx`; di conseguenza, la logica combinatoria per il calcolo dello stato futuro è molto più grande.

- Dal punto di vista temporale, la simulazione di RDIP prende più tempo dato che prima di iniziare le transizioni attraversiamo una fase di configurazione dove andiamo a scrivere i registri APB, assente invece nelle altre IP.

6.3.2 Caso I2C

Procediamo con la simulazione I2C; nello specifico stimeremo la potenza consumata nel caso in cui volessimo effettuare cinque letture a cinque indirizzi differenti del medesimo *Slave* (nella simulazione di RDIP, questo sarà l'agente UVM). I parametri della simulazione sono:

- clock_cpu = 40MHz.
- clock_per = 20MHz.
- $\boldsymbol{f_{SCL} = 1MHz}$.
- Numero Byte dato = 4.
- Numero Byte indirizzo registro = 2.

Il numero di transazioni compiete è stato ridotto per mantenerci sullo stesso ordine di grandezza per quel che riguarda la potenza dissipata. I valori che otteniamo sono i seguenti:

Modulo	Total	Leakage	Internal	Switching
`rdip`	4.97782e+02	2.24635e-01	4.32323e+02	6.52342e+01
`/fsm`	2.38424e+02	1.39567e-01	2.20150e+02	1.81346e+01
`/apb_int`	1.26603e+02	6.64036e-02	1.26229e+02	3.07014e-01
`/uart_tx`	1.58025e-02	1.31515e-02	1.57952e-03	1.07153e-03
`/reg_status_tx`	1.86812e+01	6.33900e-04	1.86806e+01	0.00000e+00
`/reg_int_status`	1.86821e+01	4.54094e-04	1.86813e+01	3.99006e-04
`/reg_status_rx`	1.58333e+01	2.66035e-03	1.58258e+01	4.88787e-03

Tabella 6.6: Potenza consumata RDIP: caso *I2C-Master*

Mentre per la periferica singola sono:

Oggetto	Total	Leakage	Internal	Switching
`i2c_master`	1.45781e+02	7.01863e-02	1.23317e+02	2.23939e+01

Tabella 6.7: Potenza consumata IP singola: caso *I2C-Master*

Anche questa volta abbiamo una dissipazione di potenza maggiore per FSM generale, vale a dire $\mathbf{238\mu W}$ contro $\mathbf{146\mu W}$ della periferica singola. È interessante sottolineare nella Tabella 6.6 come il consumo di `uart_tx` è quasi nullo (ordine della decina di nW); questo può essere attribuito alla cella di clock gating che abbiamo inserito dall'esterno proprio per fermare il clock a quel modulo quando non vogliamo comunicare tramite *UART*. Quindi, in ottica di risparmio di potenza, le celle di clock gating inserite a livello di macro-blocco quando è presente una gerarchia risultano essere molto valide.

6.3.3 Caso SPI

Terminiamo questa parte con la simulazione SPI: questa volta, configureremo RDIP come *Slave* e faremo effettuare dal dispositivo esterno cinque letture a cinque locazioni di memoria differenti. I parametri della simulazione sono:

- clock_cpu = 40MHz.
- clock_per = 20MHz.
- $\boldsymbol{f_{SCLK} = 200KHz}$
- CPOL = 0.
- CPHA = 0.
- Direzione scorrimento = MSB-first.
- Numero Byte dato = 4.
- Numero Byte indirizzo = 0.

Abbiamo abbassato la frequenza seriale dato che in una comunicazione *SPI* lavorano contemporaneamente sia il buffer TX che RX, portando così un maggior dispendio di energia. I risultati sono i seguenti:

Modulo	Total	Leakage	Internal	Switching
`rdip`	4.72259e+02	2.28878e-01	4.18606e+02	5.34241e+01
`/fsm`	2.13105e+02	1.42370e-01	2.06448e+02	6.51439e+00
`/apb_int`	1.26530e+02	6.80142e-02	1.26257e+02	2.04396e-01
`/uart_tx`	1.35812e-02	1.30276e-02	4.16613e-04	1.37025e-04
`/reg_status_tx`	1.86843e+01	6.34230e-04	1.86831e+01	5.78872e-04
`/reg_int_status`	1.86834e+01	4.52005e-04	1.86826e+01	3.07366e-04
`/reg_status_rx`	1.57842e+01	2.65916e-03	1.57815e+01	0.00000e+00

Tabella 6.8: Potenza consumata RDIP: caso *SPI-Slave*

Nella Tabella 6.8 possiamo apprezzare come il registro di stato `reg_status_rx` non abbia un contributo di potenza di "switching"; il motivo di ciò è che, nonostante la comunicazione sia *full-duplex*, i flag di ricezione sono disattivati (lettura da parte del *Master* esterno) e non pilotano nessuna uscita. Esattamente l'opposto si è verificato nella precedente simulazione dove eseguivamo della letture e i flag di trasmissione erano disattivati, risultando un contributo di "switching" nullo per `reg_status_tx`.

Oggetto	Total	Leakage	Internal	Switching
`spi_slave`	1.48319e+02	9.40749e-02	1.26533e+02	2.16925e+01

Tabella 6.9: Potenza consumata IP singola: caso *SPI-Slave*

Il totale è quindi $\mathbf{213\mu W}$ contro $\mathbf{148\mu W}$, riconfermando il fatto che il consumo di potenza della FSM generale è maggiore rispetto ad una macchina stati che implementa solamente una specifica.

Capitolo 7

Conclusioni

In questo lavoro di tesi abbiamo testato RDIP a livello RTL in diverse configurazioni possibili, dal Baud-Rate con cui comunicare sul bus alla lunghezza di una parola, ma soprattutto verificando che supportasse e che rispettasse tutti e tre i protocolli seriali che avevamo prefissato al principio dell'attività grazie all'ambiente UVM che ha conferito un grado di validità in più alle nostre simulazioni. In particolare, per ciascun Standard seriale, il **monitor UVM** ci ha assicurati che da parte nostra le linee di dato e di clock seriali pilotate dalla periferica non violassero il tempismo e la sincronizzazione fissati in fase di costruzione dell'ambiente di simulazione. Dall'altra parte invece, gli **agenti UVM** ci hanno garantito di portare a termine delle simulazioni più realistiche comunicando con oggetti esterni tramite il medesimo Standard seriale e constatare così che il flusso di controllo portato avanti dalla macchina a stati interna fosse corretto per tutti e tre i protocolli, inclusa la parte di configurazione dei registri APB (di tutta la Register-Map in generale). Tutto questo ha sancito con un certo grado di sicurezza il funzionamento di tutta la struttura hardware che è stata sviluppata in RTL. Fatto ciò, abbiamo proseguito con la sintesi logica per valutare finalmente gli aspetti inerenti all'area occupata e alla consumo di potenza, oltre che verificare che venisse eseguita correttamente anche la simulazione **Gate-level** che sfrutta la *netlist* generata dalla sintesi. Per quanto riguarda il report inerente all'area, la nostra soluzione riconfigurabile si è dimostrata molto più che efficiente rispetto ad una Standard, portando il risparmio di area a ben oltre il 50%, soprattutto se tenessimo in considerazione anche un'interfaccia APB dedicata per ognuna delle IP singole usate per il confronto. La condivisione totale dell'hardware e l'implementazione di una macchina a stati generale che gestisse e supportasse più periferiche seriali si è rivelata una scelta azzeccata sotto questo punto di vista. D'altro canto però, la potenza dissipata nell'esecuzione di una operazione, che si usi un protocollo seriale piuttosto che un altro, risulta

essere generalmente maggiore rispetto alle periferiche singole; questo è dovuto chiaramente alla maggiore complessità della FSM generale, dall'elevato numero di stati possibili a tutta la logica combinatoria interna, oltre che a tutti i registri e contatori usati per portare avanti tutti i tipi di transizioni. Questi risultati ottenuti sono in linea con ciò che è stato studiato e analizzato all'inizio di questa tesi nel capitolo 2; anche li infatti le architetture riconfigurabili permettevano una riduzione considerevole delle risorse hardware utilizzate a costo di un maggior dispendio di potenza, persino in altri tipi di applicazioni (come la FFT). Per quanto riguarda possibili sviluppi futuri, soprattutto in ottica di integrazione all'interno del sistema di EPIC, potremmo avere:

1. Possibili revisioni del codice RTL per ulteriori ottimizzazioni, specialmente per quel che riguarda gli aspetti inerenti al consumo.
2. Riorganizzazione della Register-Map per un eventuale risparmio dello spazio di indirizzamento a livello di sistema.
3. Riformulazione del funzionamento della periferica configurata come *Slave*, adattandolo di più alle esigenze di EPIC.

Bibliografia

[1] Manish Kumar Saxena, Ekansh Bhatnagar, Nitin Jaiswal, Milind Parab e Samir Nagesh Kulkarni. «Reconfigurable architecture for IP peripherals». In: *ICSES 2010 International Conference on Signals and Electronic Circuits*. 2010, pp. 347–350 (cit. a p. 6).

[2] Yutian Zhao, A.T. Erdogan e T. Arslan. «A novel low-power reconfigurable FFT processor». In: *2005 IEEE International Symposium on Circuits and Systems (ISCAS)*. 2005, 41–44 Vol. 1. DOI: 10.1109/ISCAS.2005.1464519 (cit. a p. 10).

[3] A Anagha e M. Mathurakani. «Prototyping of dual master I2C bus controller». In: *2016 International Conference on Communication and Signal Processing (ICCSP)*. 2016, pp. 2124–2129. DOI: 10.1109/ICCSP.2016.7754555 (cit. a p. 15).

[4] *Understanding the I2C Bus*. SLVA704. Application Note. Texas Instruments. 2015 (cit. alle pp. 15, 19).

[5] *I2C-bus specification and user manual*. UM10205. User Manual. NXP. 2021 (cit. a p. 21).

[6] *Serial Peripheral Interface (SPI), User Guide*. SPRUGP2A. User Guide. Texas Instruments. 2012 (cit. a p. 23).

[7] *8-Bit UART Datasheet UART V 5.3*. 001-13628. Datasheet. Cypress. 2015 (cit. a p. 25).

[8] Padmaprabha Jain e Satheesh Rao. «Design and Verification of Advanced Microcontroller Bus Architecture-Advanced Peripheral Bus (AMBA-APB) Protocol». In: *2021 Third International Conference on Intelligent Communication Technologies and Virtual Mobile Networks (ICICV)*. 2021, pp. 462–467. DOI: 10.1109/ICICV50876.2021.9388549 (cit. a p. 27).