ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

CAMPUS DI CESENA
SCUOLA DI INGEGNERIA E ARCHITETTURA

Corso di Laurea in Ingegneria Elettronica e Telecomunicazioni per l'Energia

PROTOTIPAZIONE FPGA DI SOTTOSISTEMI INTEGRATI PER POWER MANAGEMENT BASATI SU ISA RISC-V

Elaborata nel corso di: Calcolatori Elettronici

Tesi di Laurea di:
ANTONIO DEL VECCHIO

Relatore:
Prof. ANDREA BARTOLINI

Correlatore:
ALESSANDRO OTTAVIANO

ANNO ACCADEMICO 2021–2022
SESSIONE II

Indice

Capitolo 1

Introduzione

Fino a 15 anni fa, grazie al Dennard scaling, era possibile aumentare il numero di transistor su un singolo chip e contemporaneamente la sua frequenza di clock mantenendo la densità di potenza costante[18]. Tuttavia, con la continua diminuzione delle dimensioni dei transistor assicurata dalla legge di Moore, a causa di limitazioni fisiche, dal 2004 non è più possibile mantenere invariata la potenza dissipata per unità d'area.

Al fine di aumentare le performance dei processori e di impedire una diminuzione delle frequenze di clock, gli hardware designer furono costretti ad introdurre innovazioni a livello architetturale prevedendo principalmente due strategie: la semplificazione dei core e la specializzazione delle unità computazionali, in modo tale da migliorare contemporaneamente sia le performance che l'efficienza.

Tuttavia, queste soluzioni portarono a design in cui l'elevato numero di core integrati su un singolo die[1], imponeva vincoli stringenti sia per quanto riguarda la dissipazione termica che la gestione della potenza del chip.

Recentemente, a causa dell'avvicinamento al limite massimo di core integrabili su un singolo die, la specializzazione delle unità computazionali sta diventando un trend ancora più importante della semplificazione dei core. Secondo questa strategia, all'interno del processore, vengono integrati una serie di circuiti ottimizzati per eseguire specifiche applicazioni. Una unità hardware progettata ad hoc per eseguire funzioni appartenenti ad un dominio limitato prevede l'utilizzo di molti meno transistor e permette l'ese-

[1] Blocco di materiale semiconduttore su cui è realizzato un circuito elettronico integrato.

cuzione di applicazioni decine o centinaia di volte più velocemente rispetto ad un core general-purpose[18].

I processori moderni integrano on-die dei Power Controller Subsystems (PCS) come risorse hardware dedicate, progettati insieme ad un Power Control Firmware che implementa complesse strategie di gestione della potenza, in modo tale da mantenere il livello delle performance più alto possibile e permettendo contemporaneamente di rispettare i vincoli fisici del chip. La gestione avanzata della temperatura e del power budget all'interno di un chip, prevedono l'implementazione e l'interconnessione di una serie di funzionalità all'interno del PCS come: il controllo dinamico del consumo di potenza da parte della CPU con costanti di tempo brevi[22] in modo tale da prevenire situazioni in cui il chip viene danneggiato dalle temperature troppo elevate della zona attiva (power capping[19]), l'interazione in tempo reale con gli input forniti dalle risorse on-die (Sistema Operativo, interfacce di gestione dell'alimentazione e sensori on-chip) e off-die (Baseboard Management Controller, Voltage Regulator Modules), allocazione dinamica della potenza disponibile da distribuire tra sottosistemi integrati general-purpose (CPU) e altri sistemi, come ad esempio i processori grafici (GPU) [25].

A livello implementativo, i PCS attualmente utilizzati, condividono un design caratterizzato generalmente da un microcontrollore single-core supportato da una serie di macchine a stati dedicate[25] o acceleratori[23]. L'hardware tipicamente sfrutta librerie specifiche per i task del PCS [5], [16] per implementare un ambiente di esecuzione real-time e per implementare le policy di power management caratterizzate da vincoli temporali molto stringenti. Un'eccezione è rappresentata dal controllore termico e di potenza, ControlPULP [21] che, come esposto nel capitolo 2 prevede un'architettura multicore in grado di introdurre un livello di scalabilità e flessibilità maggiori rispetto alle soluzioni attualmente proposte dai produttori commerciali.

Negli ultimi anni, l'aumento della complessità del design dei processori e dei compiti assegnati ai PCS hanno portato alla graduale riconsiderazione dei controllori embedded sia in termini di architettura hardware che di firmware. D'altro canto, mentre le prestazioni computazionali del PCS seguono la crescente complessità ed eterogeneità dei chip moderni, l'interfaccia di comunicazione tra il controllore di potenza e le risorse on-chip e off-chip citate in precedenza, sta diventando un fattore limitante per la capacità d'interazione real-time del power controller e la granularità degli algoritmi

di controllo previsti dal PCF. Questa situazione presenta l'esigenza di una nuova architettura per un'interfaccia di comunicazione capace di connettere il controllore di potenza con le unità hardware integrate all'interno del processore, ottimizzata per: lo scambio di messaggi e di informazioni necessarie al funzionamento del PCF caratterizzato da bassa latenza; modularità e configurabilità, grazie alle quali poter estendere facilmente i domini di controllo del PCS; standardizzazione dei protocolli di comunicazione in modo da garantire un certo livello di compatibilità con l'hardware attualmente esistente.

In questo progetto di tesi vengono affrontati i requisiti di cui sopra ampliando l'architettura del PCS ControlPULP con i seguenti contributi:

1. progettazione e validazione di un'interfaccia di comunicazione per la connessione tra ControlPULP e un processore HPC controllato: la validazione è stata portata a termine attraverso il software Questasim;

2. estensione dell'interfaccia di comunicazione tramite supporto hardware per lo scambio di messaggi previsti dalla specifica SCMI di ARM attraverso l'introduzione di una mailbox SCMI;

3. sviluppo dei driver per la comunicazione di ControlPULP verso la mailbox SCMI;

4. sviluppo e caratterizzazione di un firmware di decodifica dei messaggi SCMI per ControlPULP, attraverso misurazioni dei tempi d'esecuzione su piattaforma ControlPULP;

5. caratterizzazione dell'interfaccia sviluppata sull'implementazione hardware su FPGA attraverso la sintesi su board ZCU102. Sul SoC[2] XCZU9EG della ZCU102[14] viene utilizzato il 92% delle LUT disponibili.

Il codice relativo ai contributi elencati sopra è disponibile nella repository del progetto di ControlPULP[9].

[2]System-on-Chip.

Capitolo 2

Stato dell'Arte

Attualmente, al di fuori dei design open-source, ci sono quattro aziende produttrici di processori principali che prevedono dell'hardware dedicato all'interno dei loro prodotti, finalizzato al controllo della potenza consumata e della temperatura del chip: Intel, AMD, IBM e ARM. Anche se è disponibile una vasta documentazione sui PCS di Intel e AMD, le loro soluzioni sono closed-source sia nell'hardware che nel firmware. IBM ha un firmware open-source ma l'hardware è proprietario. ARM integra un processore M7 e, anche se il firmware è stato rilasciato come open-source, tutta la parte di controllo è nascosta in librerie compilate.

2.1 Intel RAPL

RAPL rappresenta il controllore termico e di potenza ideato da Intel[1], introdotto a partire dall'architettura Sandy Bridge[25]. Le sue principali funzioni sono il "power-capping"[2] e il monitoraggio dell'energia consumata con un un livello di granularità molto fine. Entrambe le funzionalità sono applicate ai cinque domini di potenza supportati: Package, Powerplane 0, Powerplane 1, DRAM e Psys. Ognuno dei power domain estrae e comunica informazioni quali: il suo consumo d'energia, l'impatto sulle performance del power-capping, massima e minima potenza supportata dal dominio[17].

[1] Più precisamente si tratta del nome dell'interfaccia verso il controllore.
[2] Limitazione del consumo di potenza.

RAPL prevede quattro meccanismi di monitoraggio della temperatura e della potenza del processore[13]:

1. Il **Catastrophic Shutdown Detector** obbliga il processore a fermare la sua esecuzione nel momento in cui la temperatura del core supera un certo limite preimpostato. Si tratta dell'unico meccanismo invisibile al software.

2. L'**Automatic/Adaptive Thermal Monitor** consiste in un sensore di temperatura calibrato in modo tale da attivarsi quando la temperatura supera un certo livello stabilito dalle caratteristiche termiche del chip.

3. La **Software Controlled Clock Modulation** consente al Sistema Operativo (SO) di implementare politiche di power-management col fine di ridurre il consumo di potenza; questo meccanismo funziona parallelamente a quelli automatici visti in questo paragrafo.

4. L'**On-die Digital Thermal Sensor** è un sensore digitale posto all'interno di ciascun core. Tale sensore rappresenta uno strumento per avere letture relative alla temperatura del core più affidabili e conservative; questo perchè si trova nella posizione più calda del die. I cambiamenti della temperatura rilevati dal sensore digitale possono essere misurati attraverso due soglie poste rispettivamente al di sotto e al di sopra della temperatura corrente. Nel momento in cui il valore rilevato dal sensore oltrepassa una delle due soglie, se il meccanismo è configurata in maniera opportuna, vengono generati degli interrupt che verranno poi serviti dal software.

Oltre a questi meccanismi, Intel aggiunge altre feature sotto il nome di **Opportunistic Processor Performance Operation**. Questa categoria include tutte le tecnologie di boost come ad esempio la **Dynamic Acceleration Technology** e la **Turbo Boost Technology**[13].

Un'altra feature è rappresentata dall'**Energy-Efficient Turbo (EET)**. Tendenzialmente le frequenze turbo riducono l'efficienza energetica, soprattutto quando il guadagno in performance è trascurabile[15], questa tecnologia punta a ridurre l'utilizzo delle frequenze turbo che non hanno un effetto migliorativo sulle performance.

Un'altro elemento importante introdotto nel firmware di Intel sono i Performance States (P-States), descritti nello stadard Advanced Configuration and Power Interface (ACPI)[8]. Ogni P-State corrisponde ad un punto di lavoro individuato da una coppia frequenza-tensione di funzionamento del core. Il Sistema Operativo può richiedere specifici P-State in base al carico computazionale, il processore a sua volta può accettare o rifiutare tali richieste a seconda del proprio stato. All'aumentare del P-State, sia la frequenza che la tensione di funzionamento diminuiscono.

A partire dall'architettura Haswell, Intel ha aggiunto tre nuovi range di frequenze disponibili per fronteggiare il consumo di potenza superiore causato dall'esecuzione di nuove istruzioni disponibili (AVX2 etc.). Ciascuno di questi range include frequenze che vanno da un valore di riferimento fino ad una possibile turbo frequency che dipende dal numero di core attivi ed è regolata in funzione della distanza dal limite imposto dal TDP (Thermal Design Power)[15], [26].

A livello hardware la Power Controller Unit di Intel (PCU) è rappresentata da una combinazione di macchine a stati dedicate (hardware) e un microcontrollore[25] integrato che si occupano del Dynamic Voltage Frequency Scaling (DVFS) selezionando gli stati di controllo tensione-frequenza.

2.2 AMD

AMD implementa un embedded controller chiamato System Management Unit (SMU), simile al controllore Intel sia a livello di feature disponibili che di algoritmo di controllo [28], [27]. A livello hardware sono presenti una serie di SMU slave posti ognuno nel proprio die e connessi ad un bus che li connette tra di loro permettendo lo scambio di dati. Ogni SMU consiste in un controller capace di eseguire un firmware di controllo[3] delle performance di sistema, soprattutto attraverso la scelta della frequenza della CPU[6]. Uno dei controller, oltre ad eseguire il firmware che condivide con gli altri slave, comprende le funzioni assegnate allo SMU master. L'algoritmo del master consiste nel collezionare le frequenze calcolate dagli slave e i dati che essi hanno misurato, scegliere la frequenza opportuna da applicare e comu-

[3]Le funzioni che realizzano l'algoritmo di gestione delle performance si basano prevalentemente sul controllo PID.

nicarla agli slave. Oltre a questo meccanismo, AMD prevede altre strategie che concorrono nell'algoritmo di gestione della potenza che comprendono:

- Il calcolo di una **mappa termica** del chip basata sui modelli termici interni e le misure sulla potenza consumata (in aggiunta alle informazioni rilevate direttamente dai sensori di temperatura)[20];
- Un **Reliability Tracker** interno che esegue il monitoraggio della tensione di funzionamento e della temperatura del chip e concorre nella decisione della tensione e della frequenza del processore in modo da aumentarne l'affidabilità nel tempo[11];
- Il meccanismo **Workload Aware Power Management**. Al contrario di quanto avviene tipicamente, in cui tutti i task vengono tendenzialmente eseguiti alla massima velocità consumando molta energia, in modo tale da andare prima possibile in uno stato di idle; AMD implementa un algoritmo capace di rilevare i task con vincoli di latenza meno stringenti ed eseguirli in maniera più rilassata determinando quindi un consumo di energia inferiore.
- Un meccanismo di configurazione del comportamento del processore il cui compito è quello di garantire una performance o un consumo di potenza deterministico nonostante le differenze di caratteristiche tra i vari componenti presenti all'interno del processore, dovuti alle tolleranze dei processi di produzione. Tale meccanismo è detto **Determinism Slider**[12].

2.3 IBM

Il power controller di IBM, chiamato **OCC**, è composto da cinque unità: un processore PowerPC 405 con 768 KiB di SRAM e quattro engine general-purpose per la raccolta dei dati dai sensori PVT[4](GPE) e il controllo dello stato delle prestazioni e delle funzioni di arresto della CPU (PGPE e SGPE)

[4]I sensori Process, Voltage and Temperature sono dei circuiti integrati su semiconduttore tipicamente inclusi nei design dei System-on-Chip (SoC). I sensori PVT sono in grado di misurare grandezze dinamiche relative all'ambiente del SoC (per esempio tensione e temperatura), oppure grandezze statiche (per esempio le proprietà del circuito implementato su silicio).

rispettivamente. Il firmware di IBM comprende il controllo della potenza e il controllo termico. Il controllo della potenza viene eseguito ogni $250\mu s$ e include un algoritmi PI per limitare il power budget al livello stabilito e calcolare la massima frequenza permessa. Anche il controllo termico viene eseguito attraverso un algoritmo PI[5], con un coefficiente proporzionale molto alto. Viene eseguito ogni $2ms$ e ha come unico input la frequenza più alta calcolata. L'OCC utilizza diverse istanze di questo algoritmo termico, uno per ogni componente nel chip (per esempio uno per le unità computazionali, uno per ogni tipo di memoria ecc.). Tutte le frequenze calcolate vengono inviate ad una funzione di "voting box" che seleziona la frequenza più bassa da applicare al chip. Gli altri input di questa funzione provengono da altre feature del power controller come gli algoritmi di Idle Power Saver (IPS), Dynamic Power Saving (DPS) e Workload Optimized Frequency (WOF)[16].

2.4 ARM

ARM implementa due PCS indipendenti basati sul microcontrollore ARM Cortex-M7, **SCP** e **MCP** (rispettivamente System e Manageability Control Processor). L'SCP fornisce funzionalità di gestione dell'alimentazione, mentre l'MCP supporta le funzionalità di comunicazione. Nei SoC basati su ARM l'interazione con il Sistema Operativo è gestita dal protocollo System Control and Management Interface (SCMI)[7]. SCMI fornisce una serie di interfacce software e hardware standard, indipendenti dal sistema operativo, per la gestione del dominio dell'alimentazione, della tensione, del clock e dei sensori attraverso un sistema di mailbox condiviso e gestito attraverso interrupt verso il PCS.

2.5 ControlPULP

ControlPULP nasce come fork del progetto PULP (Parallel Ultra Low Power) e dalla collaborazione tra l'ETH di Zurigo e l'Università di Bologna, con l'intento di realizzare un power controller open-source[21].

[5]Proporzionale-Integrale.

Questo embedded controller estende la struttura dei microcontrollori single-core commerciali, fornendo la prima architettura di un PCS RISC-V multi-core. In questo progetto, gli algoritmi di controllo possono essere parallelizzati su un cluster di core CV32E40P connessi ad una RAM L1 di 128 KiB ed ad un DMA engine, ognuno dei quali calcola le azioni di controllo per un sottoinsieme dei core controllati. Oltre alle unità appena presentate, all'interno dell'architettura di ControlPULP, è presente un ulteriore core CV32E40P detto **Manager Core** col compito di delegare le funzionalità relative agli algoritmi di controllo al cluster, di implementare le funzionalità a più alto livello e di gestire le interfacce verso l'esterno.

Il **Power Control Firmware** (PCF) esegue le funzioni di controllo termico e di potenza e gestisce le comunicazioni e i trasferimenti di dati verso risorse on-die e off-die. Per ottenere uno scheduling più preciso e andare incontro alle diverse frequenze di aggiornamento dei sensori, la routine di controllo consiste in due task periodici caratterizzati da frequenze armoniche multiple: il Fast Power Control Task (FPCT) con armonica a 8 kHz e il Periodic Control Task (PCT) con frequenza 2 kHz. Il PCT è il task di controllo principale, riceve la frequenza di clock desiderata per ogni unità computazionale e la controlla per soddisfare i vincoli fisici e imposti del sistema; l'FPCT legge periodicamente il consumo di potenza dei rail di tensione dai VRM (Voltage Regulator Modules) e modifica la soglia del budget di potenza del PCT.

Le scelte di design di ControlPULP sono state eseguite per renderlo adatto all'esecuzione real-time delle istruzioni del firmware per implementare le politiche di controllo termico e di potenza.

La RAM L2 presente, ha una capacità di di 512 KiB, abbastanza da poter contenere sia il firmware che i dati in modo tale che non sia necessaria nessuna operazione di swapping[6]. Per quanto riguarda la gestione degli interrupt è affidata ad un interrupt controller detto **CLIC** (Core Local Interrupt Controller). Questo controller, prevede una gestione degli interrupt vettorizzata, a bassa latenza, basata su livelli di priorità a 8 bit ed è capace di gestire fino a 256 linee di interrupt.

Come già accennato a inizio paragrafo il cluster include un DMA multicanale con accesso diretto alla RAM L1 e una bassa latenza di programmazione. Il compito principale del DMA è quello di rendere possibile la comu-

[6]Se una memoria hardware ha meno capacità di quella necessaria durante l'esecuzione del firmware, sono necessarie operazioni di scambio dei dati con altre unità di memoria.

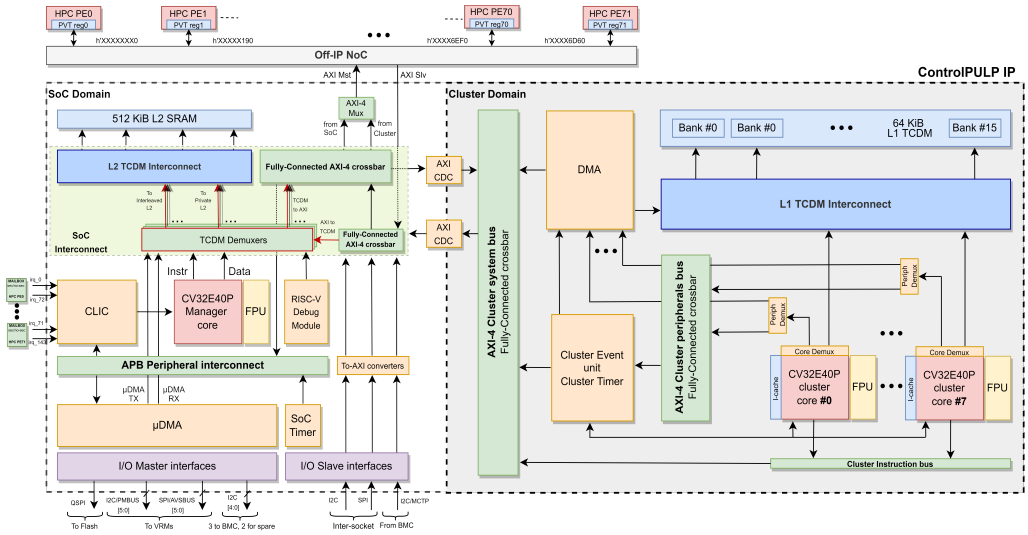

Figura 2.1: Schema dell'architettura di ControlPULP

nicazione diretta tra le memorie L1 e L2 in parallelo e senza l'intervento dei core del cluster o del cluster manager[24]. Tenendo in considerazione che il progetto di ControlPULP nasce per applicazioni di power management, il DMA è stato connesso attraverso la AXI master interface direttamente al cluster e ai sensori PVT. In questo modo si garantisce un elevato grado di flessibilità, grazie alla separazione delle fasi di trasferimento dei dati e della loro computazione.

Per quanto riguarda le interfacce esterne, ControlPULP dispone di due porte AXI4[4], una Master e una Slave, con canali a 64 bit W/R e 32 bit AW/AR. Esse svolgono un ruolo fondamentale nella progettazione dell'embedded controller e garantiscono una comunicazione a bassa latenza con il sistema controllato. Il master AXI4 è il livello di trasporto attraverso il quale il PCS raccoglie i dati dei sensori e le richieste della politica di gestione di potenza. Durante l'applicazione del controllo, viene invece utilizzato per distribuire i dati riguardanti il punto operativo di frequenza ottimale. In particolare, è possibile visualizzare in figura 2.1, come il bus AXI4 rappresenti il mezzo di comunicazione verso i core HPC (High Performance Computing) controllati.

ControlPULP dispone inoltre di un sottosistema di periferiche in cui,

un DMA permette la comuncazione autonoma tra gli elementi off-die e la SRAM L2. Il PCS integra sei interfacce AVSBUS e PMBUS verso i VRM e una interfaccia QSPI per comunicare con una memoria non volatile esterna. oltre a queste sono disponibili cinque interfacce master/slave I2C che gestiscono la comunicazione con il BMC (Board Management Controller).

Come riportato nello schema in gigura 2.1, il bus di gestione dell'alimentazione (PMBUS) e Advanced Voltage Bus (AVSBUS) estendono rispettivamente l'I2C e l'SPI per fornire un monitoraggio digitale di tensione e dei rail d'alimentazione mantenendo contemporaneamente un buon livello sia di velocità di trasmissione che di efficienza energetica.

Capitolo 3

Concetti Introduttivi

3.1 FPGA

Una Field Programmable Gate Array (FPGA) è un circuito integrato su silicio dotato di blocchi logici configurabili (CLB) e altre caratteristiche che possono essere riprogammate dall'utente. A differenza di altri circuiti integrati, in una FPGA possono essere configurate sia le reti logiche che le interconnessioni che le collegano, attraverso linguaggi di descrizione dell'hardware come ad esempio SystemVerilog o VHDL. Anche se il design di una FPGA dipende dal prodotto preso in considerazione, generalmente questo tipo di circuito integrato prevede, all'interno della sua struttura i seguenti blocchi (riportati anche in figura 3.1):

- CLB (Configurable Logic Block);
- IOB (Input/Output Block);
- blocchi di memoria;
- reti di distribuzione del clock;
- connessioni riconfigurabili;

I CLB rappresentano l'elemento utilizzato per implementare le funzioni logiche: questi infatti generalmente contengono un insieme di reti logiche che concorrono, grazie anche alla presenza di memorie di diverso tipo, all'implementazione di una LUT e dunque all'emulazione di una rete logica non

realmente presente in hardware. Una LUT o Look Up Table è una tabella in cui viene associato un valore logico relativo alle uscite di un CLB ad ogni combinazione dei valori dei segnali in ingresso allo stesso blocco. Grazie a questo meccanismo, a parità di design hardware di un CLB, possono essere implementate le funzioni logiche desiderate, purchè si riferiscano ad un numero di ingressi e uscite pari a quelle del blocco riconfigurabile. All'interno di una FPGA, generalmente le funzioni logiche vengono distribuite su più CLB interconnessi tra loro.

Gli IOB sono blocchi che eseguono gli adattamenti elettrici necessari a connettere gli elementi interni di una FPGA con i componenti esterni, attraverso i pin del package[1]. All'interno di una FPGA, sono presenti anche dei blocchi di memoria RAM che vengono generalmente utilizzati quando l'implementazione delle memorie attraverso dei CLB comporterebbe un dispendio troppo grande di ques'ultima tipologia di blocco, quindi per salvare dati che richiedono una capacità abbastanza grande per le risorse presenti sul circuito integrato. Anche per questi blocchi, la loro configurazione e utilizzo può essere gestita attraverso i linguaggi di descrizione dell'hardware.

Le reti di distribuzione del clock sono invece delle connessioni che si occupano di portare i segnali relativi al clock a tutte le aree di una FPGA, andando a gestire in maniera opportuna i ritardi di propagazione e lo skew. Generalmente questo tipo di connessioni viene collegato a dei Phase-Locked Loop (PLL) presenti in hardware che danno la possibilità di ottenere segnali di clock a diverse frequenze e di gestirne la sincronizzazione. Un altro elemento che concorre alla riconfigurabilità di questo tipo di circuiti integrati è rappresentato dalle connessioni riprogrammabili. Come è possibile notare anche in figura 3.1, all'interno di una FPGA, sono presenti diverse aree in cui le connessioni tra i vari blocchi, si incrociano tra loro. In queste aree, oltre alle connessioni, sono presenti dei circuiti che agiscono come degli interruttori programmabili che possono essere dunque configurati per unire due tratti dei collegamenti o meno.

Oltre a queste tipologie di blocchi, possono essere integrate altre risorse hardware, come dei blocchi per Digital Signal Processing (DSP), in funzione della classe di applicazioni per cui è previsto uno specfico modello di FPGA. Data la loro configurabilità, generalmente le FPGA vengono utilizzate nella fase di prototipazione di un sistema digitale in quanto ne permettono l'emulazione e, a differenza di un circuito integrato non riconfigurabile,

[1] Contenitore in cui è racchiuso il circuito integrato

permettono di eseguire delle modifiche al design emulato direttamente da codice.

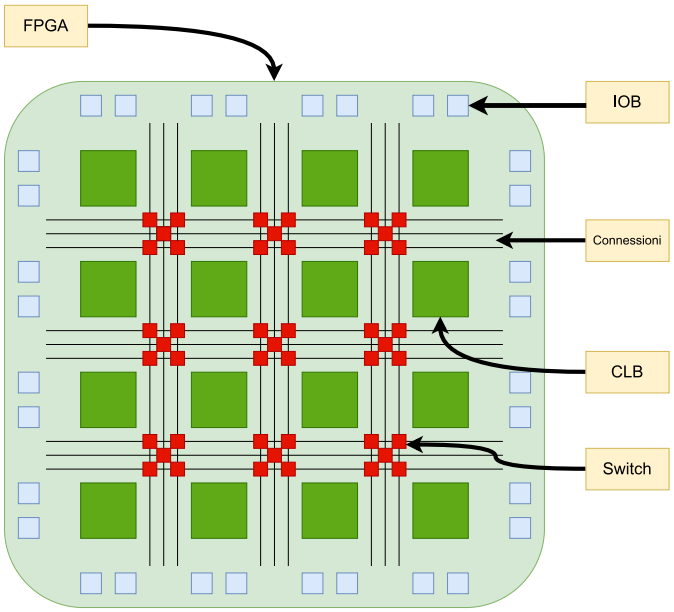

Figura 3.1: Struttura generale di una FPGA

3.2 ARM SCMI

Generalmente, per quanto riguarda la gestione degli aspetti termici e di potenza di un processore, si tende ad astrarre tali funzionalità al di fuori del processore principale, delegando questi compiti ad un microcontrollore esterno. I microcontrollori utilizzati, spesso integrano funzionalità e interfacce molto simili tra loro; ARM, dovendo fornire le IP[2] per tali sistemi, ha cercato di standardizzare le comunicazioni tra i power controller e i processori a cui sono connessi. La specifica System Control and Management Interface (SCMI), dunque, definisce un insieme di interfacce software indipendenti dal sistema operativo, utilizzate per la comunicazione tra il processore principale di un sistema e le restanti unità computazionali. SCMI[7] attualmente fornisce interfacce per la gestione di:

[2]Intellectual Property

- potenza;
- prestazioni;
- clock;
- lettura dai sensori on-chip e off-chip;
- tensioni di alimentazione;
- funzionalità di power capping;
- scoperta delle funzionalità offerte dall'hardware connesso al processore.

In questo paragrafo saranno presentati gli aspetti rilevanti di questa specifica per la successiva comprensione dei contenuti dei capitoli 4 e 5.

3.2.1 La Mailbox ARM SCMI

La specifica SCMI è definita su diversi livelli d'astrazione. Uno di questi è detto livello del trasporto; si tratta di un livello superiore alla struttura hardware che definisce le modalità di scambio dei dati relativi al protocollo. Per quanto riguarda questo layer, vi è la possibilità di scegliere tra due opzioni: trasporto basato sulla memoria condivisa o ACPI-based transport. Nel primo caso viene imposto l'utilizzo di una memoria condivisa tra il processore controllato, detto **agent**, e il power controller, chiamato **platform**, ad esso connesso, per lo scambio di messaggi SCMI. I messaggi SCMI vengono scritti da parte del processore all'interno della memoria condivisa per impartire comandi al power menager, oppure dalla piattaforma per inviare notifiche relative alle politiche di power-management al processore. La memoria condivisa, come verrà approfondito in seguito, risulta divisa in più campi. Il layout dei messaggi fa corrispondere un valore ad ogni campo della memoria condivisa, come imposto da specifica SCMI. Per quanto riguarda la seconda opzione, il layer di trasporto può essere rappresentato da un canale di comunicazione tra il processore e la piattaforma di tipo ACPI. In questo progetto verrà scelta la prima tipologia di layer di trasporto, andando ad implementare la memoria condivisa chiamata in questa specifica **mailbox**. Anche se la specifica SCMI non impone nessun vincolo sulla tipologia del supporto hardware della memoria condivisa, la scelta relativa al layer di

Field	Byte Length	Byte Offset
Reserved	0x4	0x0
Channel status	0x4	0x4
Reserved	0x8	0x8
Channel flags	0x4	0x10
Length	0x4	0x18
Message header	0x4	0x18
Message payload	N	0x1C

Tabella 3.1: Layout Mailbox SCMI

trasporto risulta determinante per andare a gestire le modifiche architetturali della piattaforma in cui questa va integrata. La strategia della memoria condivisa permette di mantenere alto il grado di flessibilità e scalabilità dell'interfaccia che connette agent e platform in quanto si tratta soltanto di una zona di memoria dedicata allo scambio di messaggi del protocollo SCMI ma non impone vincoli sulle funzionalità implementate. Inoltre, per aumentare il numero di core supportati dall'interfaccia è necessario soltanto aumentare la capacità di tale memoria. Come riportato nel Platform Design Document[7], la memoria della mailbox deve seguire uno specifico layout, tale disposizione è espressa nella tabella 3.1. Come è possibile notare viene definito l'ordine in cui i vari campi devono trovarsi in memoria attraverso i valori del Byte Offset ed è già definita la dimensione in byte di ciascun campo attraverso Byte Length ad eccezione di quello relativo al payload del messaggio.

A parte il primo e il terzo campo, che sono riservati e hanno valori dipendenti dall'implementazione o risultano inutilizzabili, tutti gli altri costituiscono i campi del messaggio definito dal protocollo SCMI.

Il campo **Channel status** ha la seguente struttura:

- Bit[31:2] Riservati, devono valere zero.

- Bit[1] errore di canale, questo bit deve essere posto a 1 se il messaggio precedente non è stato trasmesso a causa di un errore di comunicazione. L'agente deve porlo a 0 quando prende il controllo del canale.

- Bit[0] questo bit viene posto a 0 quando l'agent inizia una comunicazione verso la platform, 1 quando la piattaforma risponde e libera il canale.

Per quanto riguarda il campo **Channel flags** la struttura è la seguente:

- Bit[31:1] Riservati.
- Bit[0] vale 1 se il comando deve concludersi con la generazione di un interrupt, 0 altrimenti.

Il **Message header** contiene a sua volta diversi campi che vanno a definire la natura del messaggio (comando, notifica, risposta), il dato richiesto e il sottoinsieme di messaggi di cui fa parte il messaggio da decodificare. Segue il **Message payload** che consiste in un array di lunghezza arbitraria composto da valori a 32 bit utilizzati per i parametri dei comandi e dei valori di ritorno. Il campo **Length**, invece, va a definire la lunghezza in byte del message header più quella del message payload (4+N).

In riferimento al Platform Design Document di ARM[7], viene definito che le comunicazioni tra agent e platform possono essere gestite sia a interrupt che a polling. Nel grafico 3.2 viene presentato il flusso degli eventi di una scrittura interrupt-driven tra *caller* (agente) e *callee* (platform). Per prima cosa il caller aspetta che il canale sia libero, ad esempio leggendo in modalità polling il campo Channel status del messaggio; una volta verificatasi questa situazione passa alla fase di scrittura della memoria condivisa. Come ultima scrittura scrive nel Channel status il valore "0" nella posizione del bit meno significativo. A questo punto, il supporto hardware della mailbox dovrà segnalare al callee l'arrivo del messaggio da parte del caller, quest'operazione viene eseguita mandando un impulso sul segnale di interrupt detto *doorbell* verso la piattaforma. Una volta processato il comando presente nella memoria condivisa, il callee la sovrascrive con i valori di ritorno. A questo punto il callee libera il canale settando a 1 il bit meno significativo del Channel status e triggerando il *completion interrupt* che sarà rilevato dal caller che dovrà quindi andare a leggere i valori di ritorno dalla mailbox. Nello schema 3.3 invece viene esposto il flusso relativo alla gestione a polling dello scambio di pacchetti del protocollo SCMI. Come è possibile notare, il segnale di doorbell viene utilizzato anche in questo caso per segnalare al callee l'arrivo di nuovi dati nella memoria condivisa; la differenza principale rispetto allo schema precedente sta nella gestione a

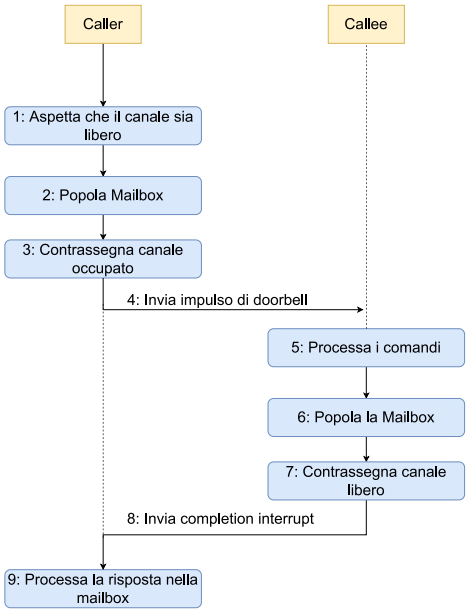

Figura 3.2: Flusso scambio messaggi interrupt-driven

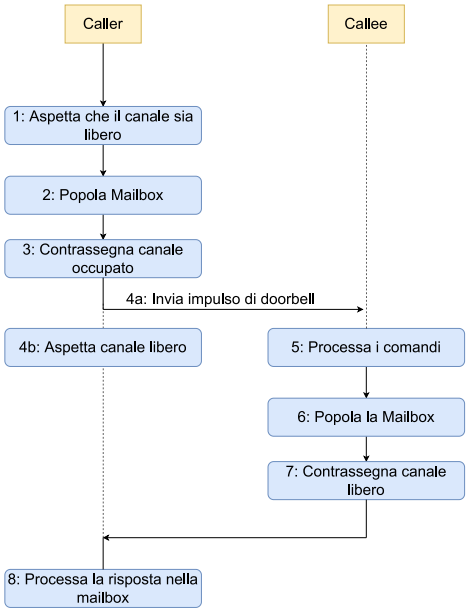

Figura 3.3: Flusso scambio messaggi con lettura a polling

polling della risposta da parte del callee. In questo caso infatti, il caller non aspetta il completion interrupt ma esegue una lettura continua del Channel status fino a quando non trova il valore "1" nel suo bit meno significativo. Poichè le modalità di conclusione dello scambio di dati viene impostata dal caller attraverso il Bit[0] del campo Channel flags, questa può cambiare anche per ogni transazione.

3.2.2 Il firmware SCP

ARM non prevede soltanto la formulazione della specifica SCMI ma, per i suoi prodotti hardware fornisce anche un firmware per l'integrazione delle funzionalità presentate nel paragrafo precedente. Tale firmware risulta un'estensione del software SCP[5] che implementa le funzioni del System Control Processor e del Manageability Control Processor già visti nel capitolo 2. La struttura del firmware SCP, presentata in figura 3.4 riesce a mantenere un alto grado di flessibilità grazie alla sua struttura divisa in layer e alla sua modularità che permette di aggiungere o togliere funzionalità semplicemente attraverso dei file di configurazione. Per quanto riguarda il firmware SCP, la struttura risulta divisa in strati che vanno dal livello di astrazione più basso dei driver, vicino all'hardware, a quello più alto riguardante i protocolli implementabili presentati nel paragrafo sul modulo SCMI che segue. In particolare, di seguito, ci occuperemo della descrizione del codice riguardante le funzionalità del protocollo SCMI spaziando tra i vari livelli d'astrazione.

Framework

Il framework rappresenta quella parte del firmware SCP le cui funzionalità sono presenti ad ogni livello d'astrazione. Il framework definisce l'insieme dei componenti di diverso tipo che possono essere combinati per ottenere un *prodotto*[5]. Un prodotto è la rappresentazione del sistema hardware per cui vengono create le immagini del firmware e comprende l'insieme di tutti i file di configurazione necessari a selezionare le funzionalità implementabili. All'interno del framework vengono istanziati diversi moduli che includono ognuno un set di operazioni invocabili attraverso il firmware. Ciascun modulo ricade in una particolare classe in base al tipo di funzionalità che rende disponibili. I diversi tipi di modulo possibili sono:

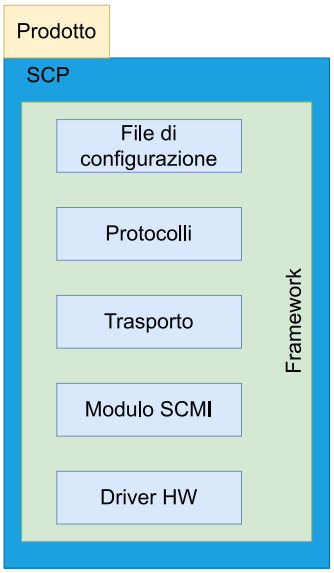

Figura 3.4: Struttura di un prodotto relativa al firmware SCP

- **Hardware Abstraction Layer**: fornisce funzionalità per uno o più tipi di dispositivi hardware attraverso interfacce standardizzate.
- **Driver**: controlla un dispositivo specifico o una classe di dispositivi.
- **Protocol**: implementa un protocollo e fornisce una o più API[3] in modo che altri moduli possano utilizzarlo. Il modulo implementa qualsiasi gestione specifica del protocollo (ad esempio, l'arbitraggio dei canali relativi allo scambio di messaggi).
- **Service**: rende disponibili funzionalità non relative alla gestione dell'hardware. I moduli di questo tipo non devono necessariamente fornire servizi ad altri moduli attraverso delle API.

Da quanto esposto, ogni modulo può fornire funzionalità ad altri moduli attraverso la dichiarazione di Application Programming Interfaces (APIs), ovvero una serie di funzioni definite all'interno del modulo che le fornisce ma che verranno utilizzate da altri moduli. Il framework si occupa anche della gestione degli *eventi*, ossia dei messaggi strutturati che vengono passati da

[3] Application Programming Interface, in questo caso si riferisce a delle librerie software.

un modulo all'altro. Ciascun evento fa riferimento ad un blocco di memoria per memorizzare i parametri utilizzati per passare le informazioni tra l'entità di origine e quella di destinazione.

Trasporto

Il modulo del trasporto è rappresentato da un modulo HAL detto appunto Trasporto, che offre un'interfaccia ai moduli permettendogli di inviare e ricevere messaggi. Questo modulo può essere utilizzato per le comunicazioni di due tipologie *In-Band* e *Out-Band*, in base alle modalità con le quali viene utilizzato l'hardware di cui questo modulo ne rappresenta un'astrazione. Per quanto riguarda la prima tipologia, a livello hardware non viene utilizzata la mailbox ma i messaggi vengono direttamente scambiati attraverso scritture all'interno dei registri di stato dei driver utilizzati in altre modalità per gestire la mailbox. A livello software i messaggi vengono scambiati da un modulo all'altro attraverso delle strutture di dati che contengono i vari campi del messaggio o dei puntatori verso quest'ultimi. Per quanto riguarda la trasmissione in modalità In-Band, ARM definisce una procedura riportata anche in figura 3.5, i cui passaggi sono i seguenti:

1. il modulo a più alto livello invia il messaggio al modulo del trasporto;
2. il trasporto lo inoltra al driver che gestisce l'hardware;
3. il driver lo copia nei registri di stato;
4. dopo la scrittura, il driver risponde al layer di trasporto con lo stato della scrittura appena eseguita;
5. il modulo del trasporto impone al driver di generare un interrupt verso chi riceverà il messaggio (ad esempio un doorbell interrupt);
6. il driver esegue il comando e riporta nuovamente lo stato dell'esecuzione al trasporto che a sua volta lo inoltra al modulo che ha iniziato le transazioni.

Per quanto riguarda la ricezione del messaggio secondo la stessa modalità, come in figura 3.6, i passaggi sono simmetrici:

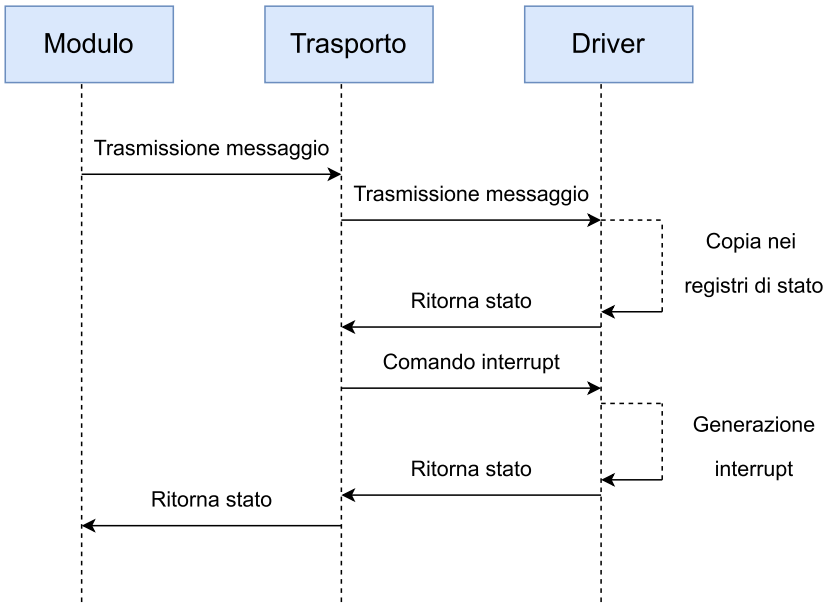

Figura 3.5: Flusso invio messaggio in modalità in-band

1. Il driver riceve un interrupt che segnala l'arrivo di un messaggio ed esegue una Interrupt Service Routine[4] (ISR). A questo punto segnala l'arrivo del nuovo messaggio al modulo del trasporto;

2. il modulo del trasporto elabora la notifica rispondendo con una richiesta del contenuto del messaggio appena ricevuto dal driver;

3. il driver inoltra il messaggio al trasporto;

4. il trasporto esegue dei controlli sulla struttura del messaggio;

5. se questi controlli vengono superati, viene inoltrata dal trasporto al modulo ad alto livello che intende comunicare col driver, una notifica;

6. Il modulo ad alto livello elabora la notifica inviando l'avviso di avvenuta ricezione al layer di trasporto che lo inoltra al driver;

[4]Uno specifico insieme di istruzioni che viene eseguito in seguito alla ricezione di un determinato segnale di interrupt.

7. A questo punto il modulo ad alto livello interroga il trasporto per ricevere i valori dei vari campi che compongono il messaggio ricevuto inizialmente dal driver.

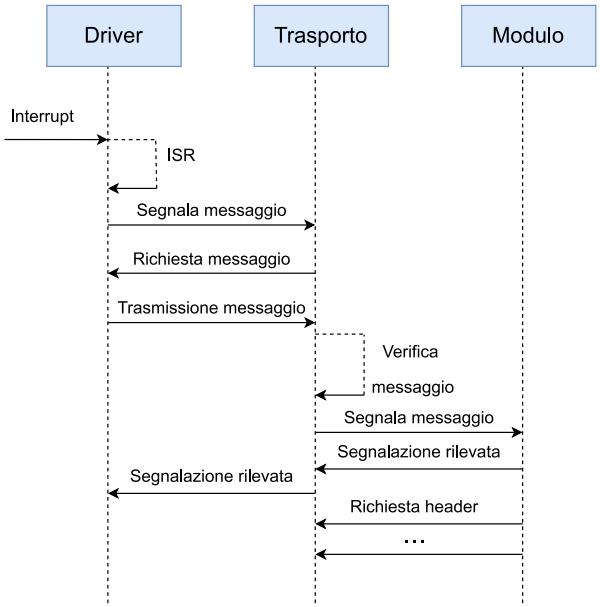

Figura 3.6: Flusso ricezione messaggio in modalità in-band

ARM definisce una procedura anche per quanto riguarda lo scambio di messaggi in modalità Out-Band. Si tratta del caso in cui viene utilizzata la mailbox come memoria condivisa tra agente e piattaforma. Per l'invio di un messaggio da parte di un modulo i passi da seguire sono i seguenti (esposti anche in figura 3.7):

1. il modulo ad alto livello trasmette il contenuto del messaggio al modulo del trasporto.

2. il trasporto copia il messaggio nella memoria condivisa.

3. il trasporto, attraverso un evento del framework, invia un comando verso il driver in modo tale che questo generi un segnale di interrupt verso il destinatario del messaggio.

4. il driver esegue il comando.

5. dopo che il driver ha eseguito queste operazioni ritorna lo stato dell'esecuzione al modulo del trasporto che lo inoltra al modulo iniziale.

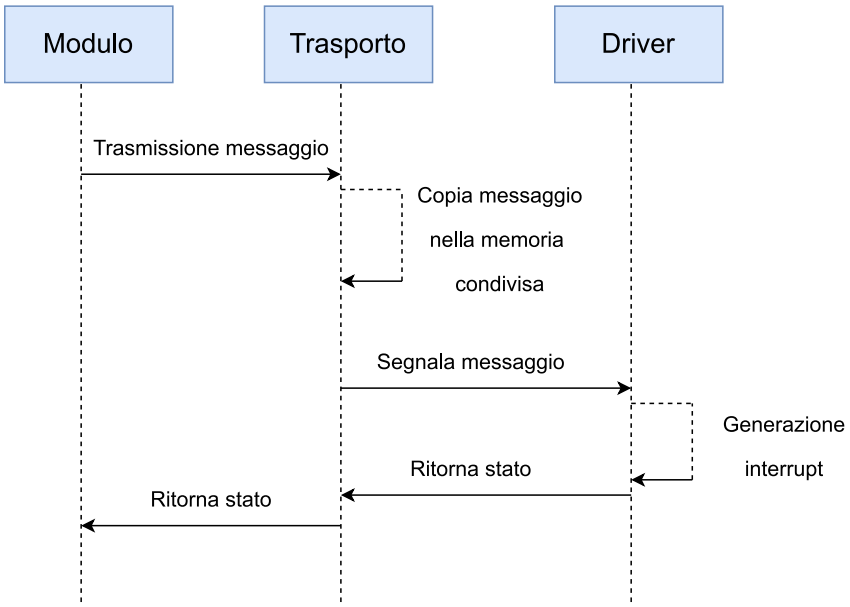

Figura 3.7: Flusso invio messaggio in modalità out-band

Per la ricezione di un messaggio da parte del driver che dovrà successivamente inoltrarlo verso il modulo ad alto livello viene eseguita la procedura riportata di seguito e presentata nello schema 3.8,:

1. il driver riceve un segnale di interrupt ed esegue la ISR. Una volta eseguita, segnala l'arrivo del messaggio al modulo del trasporto;

2. tale modulo va a recuperare i valori dei campi dei messaggi all'interno della mailbox e segnala la loro acquisizione al modulo a livello superiore;

3. il modulo a sua volta invia al trasporto un messaggio di acknowledgement che viene inoltrato al driver;

4. a questo punto il modulo ad alto livello interroga il trasporto per ricevere i valori dei vari campi del messaggio, lo processa ed esegue un handshake finale col trasporto.

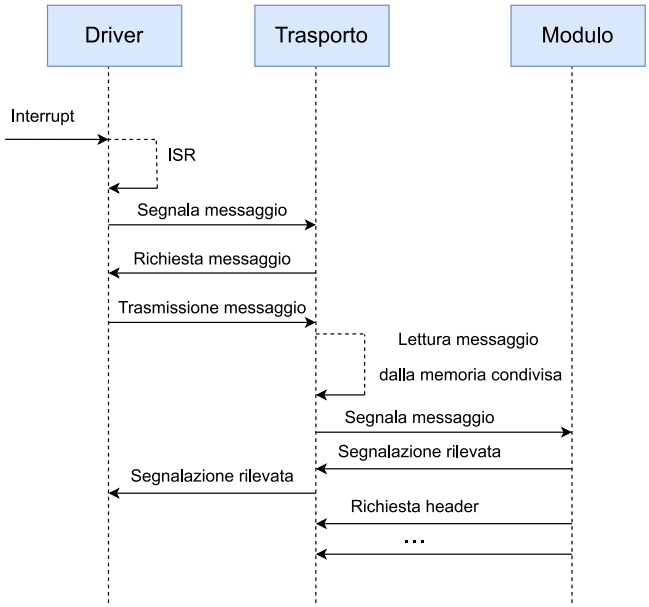

Figura 3.8: Flusso ricezione messaggio in modalità out-band

Modulo SCMI

Uno dei blocchi ad alto livello citati precedentemente può essere ad esempio il modulo SCMI. Questo modulo viene connesso, tramite la procedura di *bind*, direttamente al modulo del trasporto e implementa gran parte delle funzionalità della specifica SCMI. A questo punto è necessario però fare una precisazione: finora si è parlato indistintamente di specifica e di protocollo SCMI riferendosi all'insieme complessivo di direttive imposte da ARM per la gestione delle comunicazioni tra agent e platform e, più in generale, per l'implementazione delle politiche di controllo. In realtà, nella descrizione di tale specifica, ci si riferisce ai **protocolli** come insiemi di messaggi SCMI ognuno relativo ad uno dei domini della specifica (per esempio clock, potenza, ecc.). I vari protocolli condividono tra loro la struttura dei messaggi e il relativo flusso per lo scambio tra agent e platform. Il modulo SCMI,

di tipo *service*, implementa la gestione dei messaggi SCMI comune a tutti i protocolli. Per fare ciò include una serie di funzioni relative all'estrazione dei campi dell'header, di controllo della conformità dei loro valori così come della conformità dei dati all'interno del payload del messaggio. Oltre a questo implementa tutte le funzioni che gli permettono di comunicare con il modulo del trasporto. Al suo interno inoltre contiene un meccanismo che gli permette di eseguire una prima fase di decodifica dell'header del messaggio grazie alla quale può selezionare il protocollo a cui appartiene e per tanto a quale modulo a livello più alto inoltrarlo. Sono inoltre presenti le funzioni necessarie all'inizializzazione e alla configurazione del modulo in questione secondo le opzioni previste dalla specifica SCMI.

Protocolli SCMI

A livello più alto troviamo i moduli di tipo protocollo che possono essere connessi al modulo SCMI per implementare i protocolli desiderati. In questi sono presenti tutte le funzionalità che permettono di configurare e connettere questi moduli a quelli a più basso livello, quelle necessarie a concludere la decodifica dei messaggi e le funzioni che gli permettono di comunicare con i moduli a livello più basso. Naturalmente il tipo di elaborazione dei valori di ritorno dipenderà dal protocollo per cui il modulo è stato implementato.

Driver hardware

Al livello più basso possibile troviamo i driver di gestione dell'hardware. In questo caso, risulta di particolare rilevanza il driver **Arm Message-Handling-Unit** (MHU), detto anche Mailbox Controller Driver. All'interno del codice relativo all'MHU è possibile trovare principalmente funzioni relative all'inizializzazione delle strutture dati necessarie alla gestione della mailbox, al collegamento virtuale al resto dei layer, alla lettura e scrittura di dati all'interno della memoria condivisa. Anche in questo modulo sono presenti le funzioni necessarie al collegamento con il layer di trasporto. Questo driver include anche le funzioni relative alla gestione degli interrupt. Alcune di queste definiscono la routine da eseguire come risposta al doorbell interrupt, altre implementano le funzionalità necessarie alla generazione del completion interrupt dopo aver scritto i valori di ritorno all'interno della memoria condivisa.

Capitolo 4

Metodologia

Questo capitolo è diviso in due parti. All'inizio è presente una descrizione dell'architettura di partenza implementata su FPGA, con particolare focus sugli aspetti architetturali necessari a comprendere lo scambio di dati tra ControlPULP e le risorse a cui esso è connesso. Segue poi l'esposizione delle modifiche apportate all'architettura del power controller a livello di trasporto[1], in maniera coerente con lo standard introdotto SCMI da ARM.

4.1 Implementazione su FPGA

Come è possibile dedurre dallo schema in figura 4.1, l'implementazione su FPGA, può essere schematizzata in tre blocchi principali: ControlPULP, il processore HPC e le interfacce di connessione tra i due sistemi. Per quanto riguarda i primi due blocchi, nel capitolo 2 si è già visto come il power controller nasca come componente da integrare nell'architettura di un processore, le connessioni tra questi invece, sono totalmente dipendenti dall'implementazione. In questo caso si è scelto di dotare l'implementazione su FPGA di ControlPULP di un'interfaccia AXI4 che gli permette di comunicare con un processore ad essa connesso, rappresentato nello schema 4.1 dal blocco centrale delle interfacce.

Poiché le modifiche all'architettura del bus di ControlPULP verso i core del processore costituiscono il contributo del progetto a livello hardware,

[1]Si tratta dell'insieme delle risorse hardware e software che gli permettono di comunicare con le unità computazionali esterne.

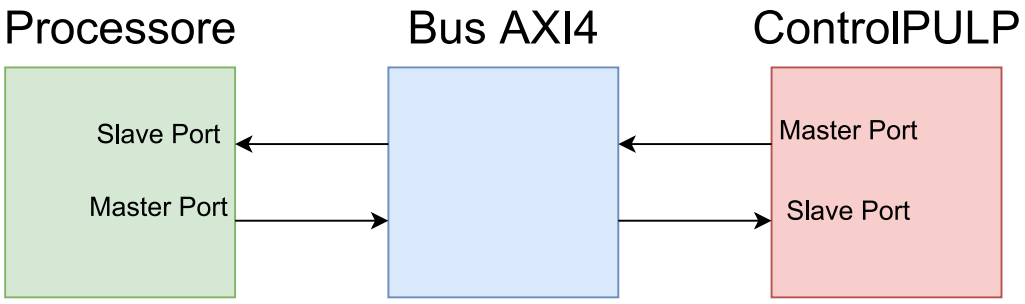

Figura 4.1: Struttura delle connessioni tra ControlPULP e il Processore

viene di seguito esposto il design delle connessioni tra il power controller e quest'ultimi prima dell'integrazione delle nuove funzionalità.

4.1.1 L'interfaccia AXI4

Come visto nel capito 2, il power manager implementato, integra un'interfaccia master/slave AXI4 verso un numero arbitrario di core (nello schema 2.1, 72 core). Secondo questo design, per le connessioni tra il processore che li contiene e ControlPULP, sono presenti due tratti separati del bus ognuno dedicato ad una direzione dello scambio di dati: processore master verso power controller slave e viceversa. L'architettura del bus, nell'implementazione su FPGA presenta non solo le connessioni tra le interfacce di ControlPULP e quelle verso i core, ma anche una serie di IP logici, necessari all'adattamento dei vari campi del bus previsti dalla specifica AXI4. Nella tabella 4.1 vengono riportate le ampiezze delle sezioni del bus lato processore e lato ControPULP (indicato con "CP") in bit.

Bus da processore a controller

Come primo IP connesso a tale interfaccia, come riportato in figura 4.2, troviamo il blocco **CNoC latency**. Questo componente ha il compito di introdurre una latenza nello scambio di dati attraverso il bus in modo da simulare il ritardo di propagazione dei segnali all'interno di un bus reale. Per introdurre questo ritardo, agisce unicamente sui segnali *ready* e *valid* relativi all'handshake tra master e slave, il resto dei segnali viene riportato dal suo ingresso all'uscita immediatamente, a meno dei ritardi di propagazione del

Campo	Proc. Master	CP Slave	CP Master	Proc. Slave
ID	16	6	7	6
USER	16	6	6	1
DATA	64	64	32	64
STROBE	8	8	4	8
ADDRESS	40	32	32	49

Tabella 4.1: Tabella delle ampiezze dei campi del bus AXI4

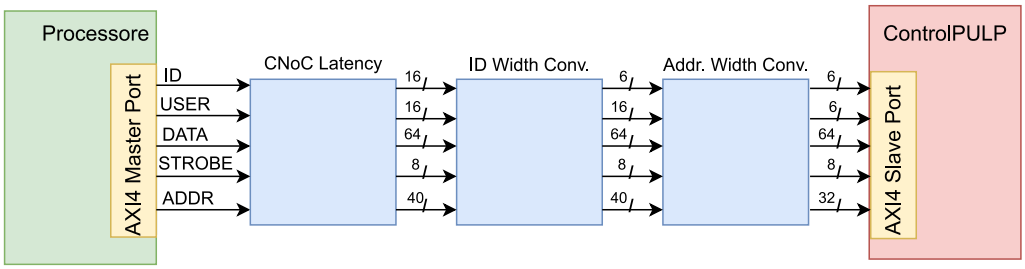

Figura 4.2: Struttura bus da processore a ControlPULP

blocco stesso. In fase di configurazione è possibile scegliere la tipologia di latenza da introdurre, tra *random*, che ritarda i segnali di un numero di cicli di clock casuale, oppure *fixed delay* che introduce un ritardo costante e impostabile.

Direttamente connesso alle uscite di questo blocco, un **ID width converter** adatta l'ampiezza del campo ID del bus da 16 a 6 bit, ovvero dall'ampiezza della porta master lato processore a quella della porta AXI4 slave lato PCS. In questo caso il blocco prevede una configurazione molto semplice in cui è necessario soltanto fornire l'ampiezza dell'ID della sua porta di input e di quella di output.

Come riportato in tabella 4.1 l'ampiezza del campo relativo ai dati (Data) risulta uguale sia per la porta master che per quella slave del bus nella direzione considerata, pertanto non è necessario alcun adattamento.

Come ultimo blocco in cascata verso la porta slave precedentemente citata, troviamo un **Address width converter**. In questo caso il modulo consente di riportare in uscita i segnali del campo Address del bus come combinazione dei segnali relativi allo stesso campo fornitigli in ingresso. Questo permette non solo di troncare o estendere l'ampiezza di tale campo, ma anche di andare a rimappare gli indirizzi. Nel caso dell'implementazione in questione infatti, nella configurazione di questo blocco, si è scelto di troncare l'indirizzo del master da 40 a 32 bit e di rendere possibile una mappatura per adattare lo spazio di indirizzi visibili dal master a quelli che ControlPULP (slave) dovrà interpretare.

Per quanto riguarda i segnali User e Strobe non è necessario un adattamento poiché, il primo segnale non viene sfruttato all'interno del bus implementato, mentre il secondo risulta già di ampiezza uguale per entrambe le interfacce alle estremità del bus.

Bus da controller a processore

Per le connessioni dalla porta master di ControlPULP a quella slave del processore, gli adattamenti necessari sono della stessa tipologia di quelli esposti nel paragrafo precedente, tranne alcune differenze specificate di seguito. Anche in questo caso, è stato istanziato il modulo che permette di riprodurre la latenza di propagazione. Successivamente, il blocco che troviamo nella cascata che costituisce gli adattatori del bus AXI4, è un **Address width converter** che avrà il compito sia di adattare l'ampiezza del campo

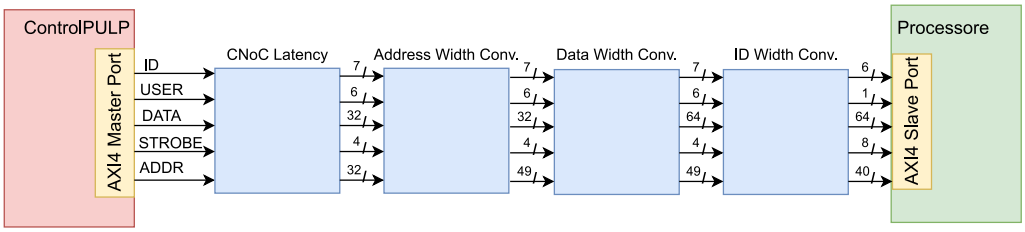

Figura 4.3: Struttura bus da ControlPULP a processore

Address che di rimappare gli indirizzi nello spazio del processore. Come espresso dalla tabella 4.1 in questo caso l'ampiezza viene portata da 32 a 49 bit.

Al contrario di quanto avviene nel tratto del paragrafo precedente, in questo caso è necessario adattare anche l'ampiezza del campo Data; sull'uscita dell'address width converter è presente un IP che ha proprio questo compito. Tale blocco, detto **Data width converter**, risulta molto facile da configurare in quanto, oltre ai collegamenti sulle porte in ingresso e in uscita, richiede poche altre configurazioni di cui, a parte le ampiezze degli altri campi del bus che resteranno uguali sia per gli ingressi che per le uscite, vengono richieste anche quelle del campo Data in modo da poter eseguire il passaggio desiderato da 32 a 64 bit.

Come si nota dallo schema 4.3, risulta evidente come il restante blocco riprenda le operazioni di adattamento già attuate per le due porte descritte in precedenza, andando a modificare l'ampiezza dell'ID in maniera coerente con quanto riportato dalla tabella 4.1.

In questa direzione del bus non è stato necessario adattare i segnali relativi ai campi User e Strb in quanto non direttamente utilizzati dai blocchi descritti in questa versione dell'implementazione.

4.1.2 Core Local Interrupt Controller

Al fine di comprendere le modifiche attuate al design di ControlPULP descritte nei capitoli successivi, risulta di particolare rilevanza una descrizione più approfondita del modulo CLIC rispetto a quanto già visto nella descrizione della piattaforma nel capitolo 2.

Il CLIC prevede tre gruppi di connessioni principali: l'insieme delle porte verso i segnali di interrupt provenienti dall'esterno di ControlPULP, l'inter-

faccia di notifica e l'interfaccia che permette l'accesso ai registri di configurazione da parte del manager core. Per quanto riguarda il primo gruppo, l'interrupt controller dispone di 256 porte a cui è possibile connettere segnali provenienti dall'esterno del power controller, che è possibile campionare sia sul fronte che sul livello in base alle impostazioni dei registri di configurazione. L'interfaccia di notifica dell'avvenuto rilevamento di una condizione di interrupt, verso il manager core, prevede l'utilizzo di cinque segnali: *valid*, *ready*, *id*, *level* e *shv*. I due segnali *valid* e *ready* rappresentano le connessioni attraverso i quali avviene l'handshake tra i due componenti in comunicazione: quando entrambi i segnali hanno il valore di "1" logico, l'handshake risulta concluso e il manager core inizia ad elaborare la richiesta di interrupt. Per quanto riguarda *id*, è il segnale attraverso il quale il CLIC comunica al core il numero identificativo dell'interrupt rilevato. Il segnale *level* viene settato dal CLIC e serve a comunicare il livello di priorità della ISR associata al segnale di interrupt con l'id segnalato durante la stessa comunicazione. In questo caso infatti, al contrario di quanto avviene con altri interrupt controller, la priorità delle routine non viene assegnata ad una porta scegliendo l'id, e quindi la posizione della routine all'interno della Interrupt Vector Table[2](IVT), bensì attraverso il valore del segnale *level*. Per quanto riguarda il segnale *shv*, viene utilizzato dal CLIC per segnalare se l'interrupt relativo all'id che viene segnalato deve essere gestito in modalità vettorizzata o meno. Oltre a questi segnali, per il funzionamento del CLIC, risultano essenziali i registri di controllo e di stato (CSR) presenti sia all'interno dell'interrupt controller stesso, che all'interno del manager core. Di seguito, vengono presentati i registri di stato e di controllo necessari alla comprensione delle configurazioni eseguite nel capitolo 5:

- INT_THRESH: registro mappato all'interno del core, necessario ad impostare il livello di soglia minimo degli interrupt che possono essere serviti. Può essere utilizzato per disabilitare momentaneamente il meccanismo degli interrupt in fase di inizializzazione del sistema.
- CLIC_INT_ATTR: Registro contenuto all'interno del CLIC, necessario alla configurazione di:
  - interrupt rilevato sul fronte o sul livello;

[2]Tabella in cui ciascuna riga associa l'id di un interrupt ad una interrupt service routine.

  - impostazione della gestione vettorizzata o non vettorizzata per ogni interrupt ID.

- CLIC_NLBITS: Registro del CLIC, utilizzato per impostare il numero di bit per la codifica dei livelli di priorità.

- CLIC_INT_CTL: Registro contenuto all'interno del CLIC, utilizzato per assegnare un livello di priorità ad ogni ID.

- CLIC_INT_IE: Registro del CLIC attraverso il quale ogni porta appartenente ai segnali su cui vengono rilevate condizioni di interrupt, può essere abilitata o disabilitata.

4.2 Estensioni architetturali di ControlPULP

Il design di ControlPULP, secondo quanto descritto nella sezione precedente, garantisce già la possibilità di comunicare con un processore col fine di implementare le politiche di controllo. Il bus AXI4 utilizzato però non presenta alcuna risorsa hardware ottimizzata per l'implementazione della parte di trasporto di un protocollo di power-management. Per questo motivo, mentre i prodotti commerciali riescono a trarre vantaggio da risorse hardware specializzate per lo scambio di dati relativi alle politiche di gestione della potenza, nel caso dell'architettura appena presentata, esse possono solo essere emulate ad alto livello. Un'implementazione ad alto livello, se da un lato permette di mantenere un certo grado di flessibilità, scalabilità e una bassa occupazione d'area all'interno del SoC, non garantisce performance paragonabili a quelle dei concorrenti commerciali. In primo luogo bisogna considerare che l'assenza di hardware specializzato comporta un livello di latenza tendenzialmente più alto del caso in cui sia presente un'architettura ottimizzata a basso livello, in contrasto con la natura real-time degli algoritmi di controllo. Successivamente bisogna valutare che un'emulazione andrà necessariamente a fare utilizzo di hardware esterno alla piattaforma di controllo (ad esempio memorie condivise) che in ogni caso andranno a determinare una complicazione anche dell'architettura hardware delle risorse dedicate al protocollo di gestione della potenza. L'obiettivo principale di questo progetto è rappresentato dall'integrazione a livello hardware e software di risorse necessarie alla massimizzazione della flessibilità e della

scalabilità oltre che la minimizzazione della latenza nello scambio di dati relativi all'implementazione di un protocollo di power-management nell'architettura di ControlPULP. Per questo scopo si è fatto riferimento al modello esposto nella specifica di ARM già vista nei capitoli precedenti riguardante il protocollo SCMI[7].

4.2.1 La mailbox in ControlPULP

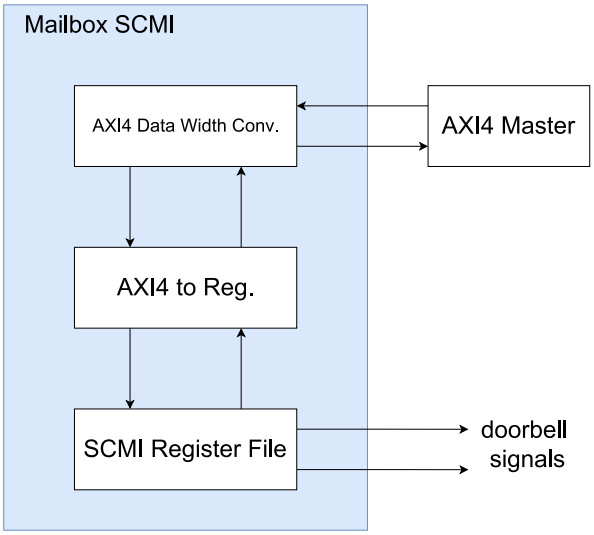

Figura 4.4: Struttura della Mailbox SCMI nel design di ControlPULP

Questo IP è stato progettato come una memoria istanziabile all'interno di un bus AXI4 e che sia internamente mappata secondo il layout imposto da ARM già presentato. Coerentemente alle specifiche, è presente una porta AXI4 che permette alle unità computazionali esterne di effettuare degli accessi sia per la lettura che per la scrittura dei registri interni. Sono inoltre presenti altri due segnali che riguardano le connessioni attraverso le quali vengono inviati rispettivamente gli impulsi relativi al doorbell interrupt e al completion interrupt. Al suo interno, come esposto dallo schema 4.4, troviamo tre blocchi principali: un datawidth converter, un convertitore di protocollo, e il register file che costituisce la memoria condivisa tra agent e platform. Il datawidth converter è necessario per adattare l'ampiezza del campo Data da quella del bus esterno a quella supportata dal register file,

pari a 32 bit. Il convertitore successivo si occupa di tradurre tutti i segnali che riceve in input da protocollo AXI4 ad un secondo protocollo meno complesso e costruito ad hoc per la comunicazione con il register file[29]. Si specifica che l'ultima conversione non avviene direttamente, il blocco in questione, infatti, risulta internamente diviso in due sotto blocchi: un primo convertitore da AXI4 ad AXI-Lite ed uno da AXI-Lite al protocollo supportato dall'interfaccia del register file. Per quanto riguarda le uscite relative agli interrupt, il segnale di doorbell è conesso direttamente al bit meno significativo di un **Registro di Doorbell**: quando in quella posizione viene posto un bit col valore di "1", anche il relativo segnale passa dal livello 0 a 1 logico. Un meccanismo analogo viene previsto per il segnale del completion interrupt, che nell'architettura di ControlPULP è lasciato scollegato, prevedendo una gestione solo a polling per il completamento delle transazioni SCMI tra agent e platform.

4.2.2 Integrazione hardware su FPGA

Per prima cosa è necessario notare come nella nuova struttura del bus AXI4 non siano più presenti solo due direzioni come spiegato nel paragrafo sull'architettura iniziale delle interfacce della piattaforma. In questo caso infatti, oltre alle connessioni tra le interfacce master e slave già presenti, è necessario l'utilizzo di un nuovo set di collegamenti che permettano sia all'agente che alla piattaforma di comunicare con la memoria condivisa. Per questo motivo, oltre all'aggiunta di nuovi sottocircuiti è stato necessario ridistribuire anche le altre risorse già presenti sul bus.

Bus da processore a controller

In questo caso gli adattamenti a livello di ampiezze risultano identici a quelli visti nel paragrafo sull'architettura di partenza; cambia però l'ordine in cui vengono eseguiti. Troviamo nuovamente il blocco necessario ad introdurre la latenza sulle comunicazioni direttamente connesso alla porta master verso il processore. In serie a questo, è presente un address width converter che a differenza di prima esegue solo un troncamento dei bit dell'indirizzo per ragioni che saranno chiarite di seguito. In uscita al convertitore d'ampiezza dell'indirizzo, in questo caso troviamo un'istanza di un nuovo componente chiamato **AXI Crossbar**. Una crossbar è un circuito in grado di connet-

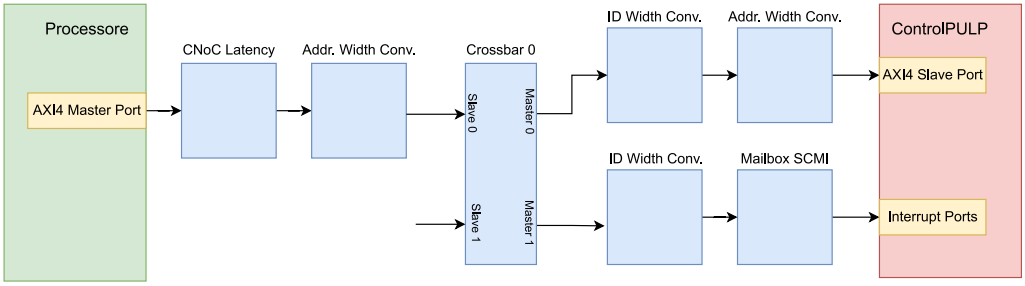

Figura 4.5: Struttura bus da processore a ControlPULP dopo l'introduzione della mailbox

tere una o più porte facenti parte dei suoi ingressi a una o più delle sue porte in uscita. In questo caso le porte rappresentano interfacce AXI4. Per configurare questo tipo di componente, oltre alle varie connessioni sui suoi input e output, bisogna fornire una tabella detta *address rule*: in ogni sua riga si può specificare un insieme di indirizzi contigui da associare ad una porta d'uscita della crossbar. In questo modo, la logica interna della crossbar sarà in grado di inoltrare un pacchetto AXI4 da una qualsiasi sua porta d'ingresso verso la porta in uscita a cui è associato il set di indirizzi di cui fa parte l'address specificato nel pacchetto in questione. Lo smistamento dei pacchetti sulle porte in uscita selezionate avviene agendo soltanto sui segnali di handshake del protocollo AXI4, per quanto riguarda gli altri segnali non è necessario eseguire la stessa selezione. La presenza della crossbar in questo tratto del bus è motivata dalla necessità di connettere la porta master verso il processore sia alla porta slave del power controller, sia alla mailbox. Inoltre, come è possibile notare dallo schema 4.5 è presente una porta in input aggiuntiva dedicata alle connessioni provenienti dalla porta master di ControlPULP che dovrà essere in grado di accedere alla mailbox allo stesso modo. Al posto di istanziare una mailbox si sarebbe potuto optare per un apparentemente più semplice **Multiplexer**. Anche se il circuito in questione potrebbe risultare meno complesso e, per tanto, suggerire un minor utilizzo d'area all'interno dell'FPGA, va specificato che sarebbe comunque stato necessario utilizzarne almeno due in modalità multiplexer e due in modalità demultiplexer per ottenere la stessa configurazione del bus offerta dalla crossbar. La crossbar, ugualmente a quanto sarebbe avvenuto utilizzando direttamente i multiplexer, implementa internamente le stesse

reti logiche ma permette di gestire il forwarding dei pacchetti tramite la address rule piuttosto che attraverso dei segnali di selezione molto piu scomodi da controllare. In uscita dalla crossbar, seguendo le connessioni verso ControlPULP, è connesso un ID width converter le cui funzionalità non sono state modificate rispetto all'architettura di partenza. L'unica differenza che va considerata è che, in questo caso, la crossbar ha aggiunto un numero di bit al campo ID del bus AXI4 che il convertitore vedrà in ingresso, pari al logaritmo in base 2 del numero di porte di output della crossbar stessa. I bit aggiuntivi vengono utilizzati dai multiplexer interni per instradare le risposte provenienti dagli slave.

Direttamente connesso al convertitore d'ampiezza dell'ID, troviamo un ultimo blocco dedicato alla conversione del campo address. In questo caso viene eseguito il remapping che non era stato incluso nel secondo blocco di questo tratto di bus. La separazione delle operazioni di troncamento e di rimappaggio in due convertitori, anche se può sembrare una complicazione porta due vantaggi rilevanti. Andando ad eseguire il remapping dagli indirizzi visti dal processore a quelli visti dalla platform solo dopo la crossbar permette di inserire in maniera più intuitiva all'interno della address rule dei set di indirizzi coerenti con quelli processore. L'esecuzione del troncamento degli indirizzi prima della crossbar permette, allo stesso tempo, di semplificare il routing e le logiche interne di tale blocco, che dovrà gestire indirizzi a 32 bit piuttosto che 40.

Lo stesso tratto di bus si conclude con la connessione verso la porta slave di ControlPULP.

In uscita dall'altra porta output della crossbar, invece, troviamo connesso il convertitore di ID width destinato alla connessione dell'interfaccia AXI4 della mailbox che porta tale campo allo stesso numero di bit previsti per le connessioni nel tratto parallelo verso ControlPULP. L'ampiezza dell'ID in ingresso e in uscita, per questo tratto del bus così come quello verso il power controller risulta identico; nonostante ciò sarebbe stato difficile andare a ridurre il numero di istanze del convertitore. Andando infatti a spostarlo in ingresso alla crossbar, tratto comune ad entrambi gli slave trattati in questo paragrafo, sarebbe stato comunque necessario porne due in uscita per compensare l'aggiunta dei bit portata dalla crossbar sullo stesso campo.

Direttamente connesso all'uscita dell'ultimo convertitore, viene connesso il blocco relativo alla mailbox SCMI. In realtà bisogna precisare che, al

fine di supportare un numero di core paragonabile a quello ipotizzato nello schema 2.1, si è scelto di connettere una sola memoria condivisa con le stesse istanze dei convertitori interni presentati nello schema 2.1 ma connessi ad un register file contenente un numero di registri pari a quello indicato nel layout in tabella 3.1, moltiplicato per 64. Si è inoltre scelto di istanziare due registri per quanto riguarda il payload di ogni mailbox.

Oltre a questi collegamenti, la mailbox espone due set di segnali relativi al doorbell interrupt e al completion interrupt; di questi solo quelli del doorbell interrupt vengono connessi al CLIC controller di ControlPULP prevedendo una gestione a polling per la lettura dell'arrivo dei valori di ritorno. Il set di segnali verso il CLIC prevede una connessione per ogni insieme di registri associati ad una sola mailbox, quindi 64.

Bus da controller a processore

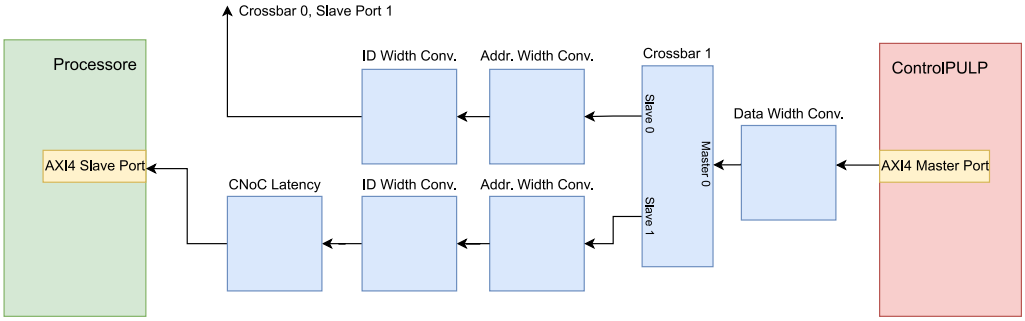

Figura 4.6: Struttura bus da ControlPULP al processore dopo l'introduzione della mailbox

Anche in questo caso sono presenti delle modifiche rispetto all'architettura di partenza. In questo tratto di bus si vuole lasciare integra la connessione dalla porta master di ControlPULP alla porta slave destinata al processore ma è stata prevista anche la possibilità di comunicare con la crossbar introdotta nel paragrafo precedente. Quest'ultima connessione sarà necessaria al power controller per accedere alla mailbox qualora volesse leggere un messaggio o scrivere i valori di ritorno. Come primo blocco troviamo un data width converter, questo avrà le stesse funzioni che aveva nell'architettura antecedente alle modifiche; la sua istanza è però posta prima della nuova crossbar poichè la porta master di ControlPULP è l'unica con un campo

Data a 32 bit. Istanziare il convertitore altrove, avrebbe comportato una semplificazione del routing della crossbar ad esso connessa ma, allo stesso tempo, avrebbe reso necessaria l'istanza di un altro convertitore aumentando l'occupazione d'area complessiva. Connesso in cascata alle sue uscite, è presente una seconda crossbar. Tale crossbar presenta solo una porta in ingresso ma provvede alla connessione sia verso a porta slave del processore che la crossbar riportata nel paragrafo precedente, che a sua volta assicura la connessione alla mailbox. In questo caso i convertitori dell'ampiezza dell'indirizzo sono posti sulle due uscite della crossbar. Quello verso la mailbox ha solo il compito ridurre l'ampiezza a 32 bit (come imposto dagli ingressi della prima crossbar), il restante esegue anche il remapping agli indirizzi coerenti con il processore. Anche l'ID width converter viene istanziato su entrambe le uscite poichè eseguono conversioni verso ampiezze diverse. Come ultimo blocco troviamo ancora il CNoC delayer che porta direttamente alla porta slave destinata alla comunicazione con i core del processore.

4.2.3 Integrazione software

Anche in questo caso, il firmware è stato scritto in modo tale da mantenere un alto grado di riconfigurabilità e flessibilità come nel caso dell'SCP di ARM ed è disponibile nella repository del progetto ControlPULP[9]. Per ottenere queste caratteristiche è stato diviso il codice ancora una volta in moduli, rappresentanti ciascuno un diverso livello di astrazione: driver, common SCMI e protocolli SCMI. In questo caso però non è presente né uno strato di framework né di trasporto proprio perchè il sistema su cui vanno integrate le funzionalità della specifica SCMI, rappresentato da ControlPULP, è soltanto uno e non è necessario prevedere la compatibilità con hardware significativamente diverso da quello utilizzato. Nel caso di questo firmware inoltre, non sono state implementate procedure di connessione dei moduli, piuttosto, le loro funzionalità sono rese disponibili attraverso l'inclusione degli header file relativi ad ognuno di essi all'interno del codice principale. Di seguito vengono presentati i tre layer appartenenti al firmware SCMI di ControlPULP per poi passare ad una descrizione degli aspetti implementativi del firmware a livello di codice.

Driver

Il driver rappresenta l'insieme di funzioni necessarie per leggere e scrivere i valori all'interno delle varie mailbox disponibili. A questo livello del firmware, l'unico controllo che viene eseguito sui valori dei registri della mailbox è quello relativo alla lunghezza del messaggio indicata nel campo *length*, se questa prevede letture al di fuori della dimensione massima allocata al payload all'interno della memoria condivisa, il messaggio non viene inoltrato agli strati superiori ma viene già segnalato un errore al caller. Questo meccanismo evita di andare ad eseguire letture da indirizzi non dedicati alla mailbox dalla quale si sta leggendo il messaggio. Oltre a queste funzioni, il driver si occupa anche di andare ad eseguire il reset relativo al registro di doorbell e del registro del Channel status ogni volta che viene scritto un messaggio da parte di ControlPULP all'interno della mailbox. In questo modo ci si assicura che, ad una nuova scrittura da parte del processore, venga generato l'impulso di doorbell verso il CLIC. Anche in questo caso si è cercato di mantenere un alto grado di riconfigurabilità del firmware, per cui, le funzioni presenti all'interno del driver, vengono esposte ai layer superiori attraverso delle strutture di puntatori che permettono, in fase di configurazione, di cambiare facilmente l'insieme di funzioni che andranno a leggere e scrivere nel supporto hardware della mailbox.

Common SCMI

Al livello d'astrazione successivo, Common SCMI, racchiude l'insieme delle funzioni necessarie alla gestione dei messaggi SCMI e comuni a tutti i protocolli implementabili al livello superiore, e di configurazione del firmware. In fase di configurazione è necessario definire le seguenti impostazioni:

- Scelta del driver di gestione della mailbox.
- Abilitazione dei protocolli SCMI.
- Abilitazione degli agenti.
- Abilitazione dei protocolli SCMI per ciascun agente.
- Inizializzazione dei moduli relativi ai protocolli abilitati.

Oltre alle funzioni per l'inizializzazione della piattaforma di power-management, sono presenti anche le funzioni di estrazione delle feature dei campi dell'header dei messagi e di decodifica del protocolli. Per quanto riguarda l'estrazione dei valori si tratta di semplici operazioni tra maschere definite dalla specifica SCMI e l'header del messaggio, sono però presenti anche le funzioni necessarie a controllare la conformità di tali valori (secondo quanto definito da ARM) e di eventuale riconoscimento di una condizione d'errore prevista dalla specifica. Successivamente ai controlli, viene eseguita una decodifica del tipo di messaggio e del *protocol id* in modo da selezionare il modulo da invocare relativo al protocollo a cui appartiene il messaggio.

Protocolli

I moduli a questo livello si occupano di decodificare il *message id* per comprendere quale tipo di azione o informazione si sta richiedendo attraverso i messaggi nella mailbox. Anche in questo caso è necessario eseguire delle configurazioni per collegare una specifica funzione ad ogni *protocol id*; questo tipo di configurazione viene però eseguita solo nel momento in cui si vuole aggiungere il supporto ad un nuovo tipo di messaggio all'interno di uno specifico protocollo e non, come per il modulo del paragrafo precedente, per definire le modalità d'esecuzione di tutti i protocolli. Poiché alla fine dell'esecuzione dei comandi sarà necessario scrivere i valori di ritorno all'interno della mailbox, una volta elaborato il payload e il resto dei valori di ritorno, il layer common SCMI si occuperà di inoltrare la risposta del modulo del protocollo verso il layer del driver. Le procedure per la lettura, la decodifica dei messaggi, l'esecuzione dei comandi e la scrittura dei valori di ritorno sono ulteriormente riportate nello schema 4.7.

Ottimizzazione del Codice

Una volta compresa la struttura del firmware ad alto livello, risulta evidente come i tempi d'esecuzione e, più in generale, le performance del firmware, risultino strettamente dipendenti dalle elaborazioni eseguite in fase di decodifica dei messaggi. Per decodifica dei messaggi, in questo caso, si intende la selezione del tipo di messaggi e dei protocolli a cui essi appartengono. Per ogni messaggio, infatti, dopo aver estratto i valori dei campi dell'header è necessario almeno:

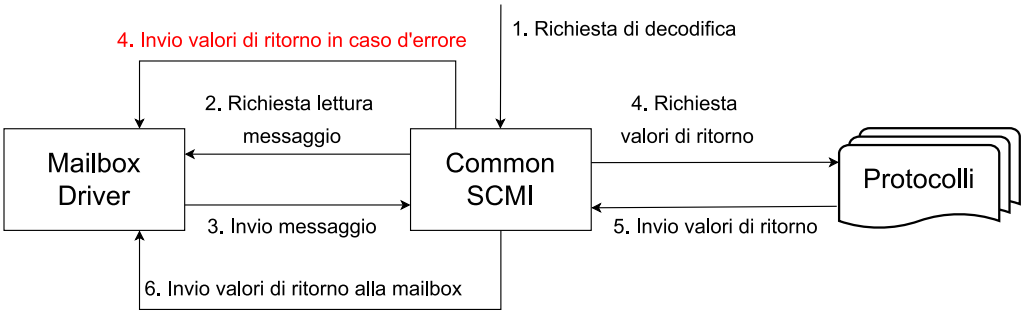

Figura 4.7: Passaggi per la decodifica di un messaggio SCMI

- controllare se l'agente da cui si è ricevuto il messaggio (rappresentato dall'id della mailbox) è abilitato;
- controllare se il protocollo relativo al *protocol id* letto è attivo;
- assicurarsi che il protocollo richiesto sia attivo per l'agente da cui è stato inviato il messaggio SCMI;
- associare al *protocol id* letto, il giusto modulo protocollo a cui inviare eventualmente il messaggio letto dal driver.
- eseguire una nuova associazione tra *message id* e funzione da eseguire per concludere la decodifica.

Questa serie di controlli prevede delle letture sequenziali da strutture di dati in cui si tiene traccia di tutte le informazioni necessarie alla loro esecuzione e che permettono di richiamare le funzioni opportune per portare la decodifica al livello dei moduli dei protocolli. Dato l'impatto che avrebbe avuto sui tempi d'esecuzione della decodifica, si è scelto di non accedere a tali strutture dati con delle letture cicliche. Queste, infatti, anche se avrebbero permesso un utilizzo più parsimonioso della memoria, avrebbero rallentato di molto i tempi di decodifica. Il meccanismo utilizzato invece, prevede soltanto due letture in sequenza da due array[3] separati, per ogni controllo

[3]Struttura di dati del linguaggio C che consiste in una lista di elementi dello stesso tipo, a cui è possibile accedere grazie ad un indice che specifica la posizione dell'elemento cercato.

da eseguire. Risulta dunque chiaro come, per tutte le sequenze di controllo elencate, il punto di partenza sia rappresentato da un identificativo (del protocollo, agente o messaggio) al quale sia necessario associare un'informazione, come l'abilitazione di un protocollo o le funzioni da eseguire per concludere la decodifica. Nella figura 4.8 sono rappresentati i passaggi necessari ad eseguire ognuno di questi check. Per prima cosa, si utilizza l'identificativo a disposizione per accedere ad un array che associa ad ogni sua posizione, un indice. Il risultato della prima lettura sarà dunque un altro indice con il quale poter accedere ad un nuovo array. Questo primo passaggio ha due scopi: dare la possibilità di sapere se un determinato protocollo o tipo di messaggio è abilitato, ridurre la dimensione delle strutture dati a cui si accede successivamente al primo array. Per quanto riguarda il primo obiettivo, si è fatto in modo che ad ogni posizione dell'array in questione fosse associato un valore numerico: se questo valore risulta uguale a "0" allora il protocollo, l'agente o il messaggio il cui id rappresenta la posizione a cui accedere nell'array è disabilitato; altrimenti, il valore riportato si riferisce all'indice con il quale accedere al secondo array che contiene il resto delle informazioni necessarie. In questo modo, oltre ad eseguire già un primo controllo sulla risorsa in analisi, è possibile ridurre la lunghezza degli array contenenti le altre informazioni poichè questi conterranno soltanto elementi relativi a risorse abilitate. La seconda struttura dati, a cui è possibile accedere grazie all'indice ottenuto dal primo array, conterrà anch'essa informazioni dipendenti dal controllo che si sta eseguendo: per quanto riguarda il *protocol id*, una volta accertata l'abilitazione del protocollo attraverso la prima lettura, si può eseguire una nuova lettura da una matrice che tiene traccia di tutti i protocolli attivi per ogni agente (accedendo con il nuovo indice ottenuto sia per il protocollo che dell'agente), per quanto riguarda il *message id* è possibile capire allo stesso modo quale funzione richiamare all'interno di un modulo protocollo per calcolare i valori di ritorno, lo stesso schema può essere utilizzato per tutti gli altri valori estratti dell'header. Il meccanismo di selezione delle funzioni da richiamare attraverso elementi di un array è anche reso possibile grazie all'utilizzo di puntatori a funzione. Un puntatore costituisce un tipo di dati del linguaggio C (utilizzato per questo firmware) che rappresenta la posizione in memoria di un altro elemento come una variabile o una funzione. In questo modo è possibile costruire degli array di puntatori a funzione che in fase di inizializzazione vengono associati alle funzioni da richiamare in base ai valori dell'header, mentre in

fase di decodifica servono ad invocare tali funzioni. Oltre a permettere una scrittura del codice più semplice, i puntatori permettono anche di mantenere alto il grado di modularità in quanto è possibile cambiare le funzioni da richiamare semplicemente cambiando il valore degli elementi nei sopracitati array. Anche se questi meccanismi possono sembrare una complicazione per un codice eseguito a basso livello[4] bisogna considerare che concorrono sia ad un'ottimizzazione delle risorse hardware utilizzate, che a mantenere alto il livello di scalabilità e modularità nel caso in cui si volessero ampliare le funzionalità del firmware. In figura 4.8 vengono presentati i passaggi per ottenere il puntatore alla funzione da richiamare (appartenente al livello dei protocolli) dal modulo Common SCMI partendo dal solo *protocol id*; ciò che va aggiunto a questa rappresentazione è che generalemnte il primo array (sulla sinistra) dovendo contenere gli elementi fino all'id con valore più alto tra quelli supportati, risulterà tendenzialmente più lungo del secondo, che invece contiene i puntatori relativi alle funzioni corrispondenti soltanto ai protocolli abilitati.

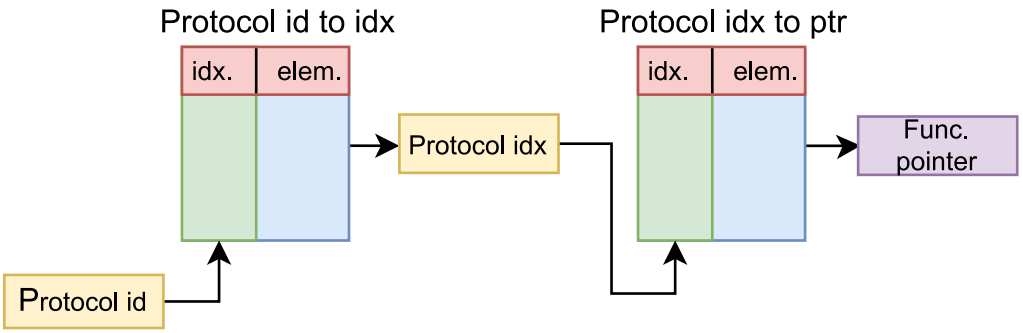

Figura 4.8: Passaggi per il controllo del protocollo e la selezione della funzione associata

[4]Livello d'astrazione basso, più vicino all'hardware

Capitolo 5

Risultati Sperimentali

In questo capitolo viene prima di tutto presentato il setup sperimentale attraverso il quale sono stati eseguiti i test, seguono le descrizioni di validazione e caratterizzazione a livello hardware e software delle nuove funzionalità introdotte in ControlPULP, già viste nel capitolo 4.

5.1 Setup Sperimentale

Nella fase di test, il setup è stato costruito sia a livello software che hardware, in modo tale da permettere la validazione e la caratterizzazione attraverso due tipologie di verifica: la simulazione e l'emulazione.

La simulazione consiste nell'impiego di un software opportuno, in questo caso Questasim[3], per andare a simulare sia il processore su cui applicare le politiche di controllo termico e di potenza, sia l'architettura di ControlPULP e del relativo bus AXI4 che li connette. La simulazione si basa sull'utilizzo di un testbench, un ambiente di verifica nel quale viene istanziato un Device Under Test (DUT) e vengono scritti i driver virtuali per interagire con il DUT. In questo caso, all'interno del testbench verrà istanziato il design di ControlPULP come DUT, mentre verrà sfruttata la capacità del testbench di imporre valori arbitrari ai segnali del bus AXI4 per simulare la presenza di un processore multicore connesso al bus AXI4 di ControlPULP. Alla fine di ogni simulazione, essendo queste cycle accurate[1], è possibile visualizzare

[1]Simulazione con un livello di precisione della descrizione dell'andamento dei segnali pari al singolo ciclo di clock.

le forme d'onda relative ad ogni segnale parte del test. Per quanto riguarda la simulazione di ControlPULP, oltre a simulare il design hardware fornito a Questasim attraverso una descrizione in SystemVerilog, è possibile eseguire una simulazione anche dell'intero stack software supportato dalla sua architettura. Nel caso dei test che seguono, come riportato in figura 5.1, lo stack si compone principalmente di quattro livelli:

- le librerie di ControlPULP, implementate per la gestione delle risorse hardware (in questo caso simulate);
- il setup che permette di eseguire delle Interrupt Service Routine;
- FreeRTOS;
- il firmware SCMI di ControlPULP.

Di questi, è stata già riportata una descrizione approfondita ad eccezione del layer di FreeRTOS. FreeRTOS è un kernel di Sistema Operativo real-time per microcontrollori, che permette di utilizzare metodi per la creazione e l'istanza all'interno di un firmware, di task da eseguire secondo delle regole di scheduling impostabili[2]. In questo progetto, il ruolo di questo sistema operativo è quello di rendere lo svolgimento della decodifica dei messaggi SCMI e la relativa risposta verso gli agenti, eseguibile in maniera concorrente al codice principale implementato dalla piattaforma, incentrato sulla computazione relativa alle politiche di controllo.

Per quanto riguarda l'emulazione, si tratta di un metodo di validazione alternativo rispetto alla simulazione, che consiste nell'implementazione su un supporto hardware del design implementato. Il supporto hardware per la realizzazione di questo progetto, è rappresentato dalla board ZCU102, una evaluation board general-purpose di Xilinx basata sul SoC Zynq UltraScale+ XCZU9EG[14]. Questo SoC contiene al suo interno diverse risorse utilizzate per la fase di test del design di ControlPULP di cui, particolarmente rilevanti, il sistema di elaborazione basato su quattro core Arm Cortex-A53 più due core Arm Cortex-R5F complessivamente denominato "PS", e una unità FPGA composta da logica progammabile e hardware di supporto per diversi tipi di interfacce (PCI, Ethernet etc.) denominata "PL". Nella fase di test, l'insieme delle risorse sotto il nome di PS è stato utilizzato per emulare il processore multicore a cui applicare i comandi di ControlPULP inviati attraverso la mailbox SCMI; mentre il PL per implementare il power

controller in analisi. Per quanto riguarda le connessioni tra lato PS e lato PL del SoC, l'architettura del bus AXI4 vista nei capitoli precedenti è stata implementata sulla logica programmabile. Dal punto di vista del software, mentre per ControlPULP ritroviamo la stessa configurazione già esposta per la simulazione, è necessario presentare anche gli strumenti utilizzati per il PS. In questo caso, il processore è dotato di una versione embedded di Linux il cui File System è stato generato attraverso il tool Buildroot[1]. All'interno di questo Sistema Operativo vengono principalmente utilizzate due applicazioni durante la fase di test: una per eseguire la fase di boot e una per eseguire l'emulazione del processore multicore entrambe approfondite nei paragrafi che seguono.

Sia per quanto riguarda le prove di simulazione che per l'emulazione, il codice è disponibile nella repository del progetto ControlPULP nella cartella relativa ai test del CLIC[10].

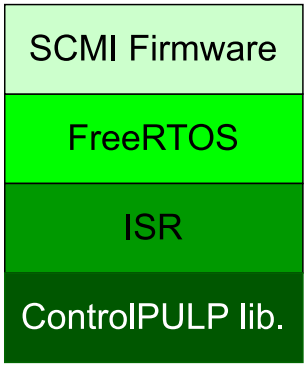

Figura 5.1: Stack software di ControlPULP utilizzato nella fase di sperimentazione

5.2 Validazione

In questo paragrafo vengono presentate le sequenze di test col fine di validare le modifiche architetturali integrate in ControlPULP.

5.2.1 Verifiche iniziali

Per prima cosa, sono state testate le funzionalità dell'architettura di ControlPULP precedente all'integrazione della mailbox, dunque la possibilità da parte del processore di inviare dati attraverso il bus AXI4 al power controller e viceversa. In questo caso, il test è stato eseguito in simulazione per cui, le funzioni di ControlPULP sono state testate attraverso una simulazione del design descritto nel capitolo 2, mentre le azioni del processore (invio e ricezione dei dati) vengono simulate attraverso delle istruzioni di lettura e scrittura sul bus AXI4 riportate nel codice di un testbench. La struttura del testbench prevede le seguenti azioni:

- Fase di boot: scrittura del codice destinato all'esecuzione da parte del manager core di ControlPULP nella sua memoria L2.
- Attesa della fine dell'esecuzione del codice da parte di ControlPULP attraverso la lettura a polling del valore di ritorno del codice eseguito dal manager core ad un indirizzo predefinito nella memoria di ControlPULP.

Il codice in C, compilato e scritto nella memoria del manager core, e da esso eseguito durante la simulazione, ha il compito di far stampare il messaggio "Hello" al manager core stesso e agli 8 core presenti all'interno del cluster di ControlPULP. Come si può vedere dal listato di seguito, le stampe sono state eseguite correttamente confermando il funzionamento del tratto di bus AXI4 in cui il processore è connesso alla porta master e ControlPULP alla porta slave.

```
# [STDOUT-CL31$_$PE0] test csr accesses
# [STDOUT-CL31$_$PE0] set up vector table
# [STDOUT-CL31$_$PE0] set selective hardware vectoring
# [STDOUT-CL31$_$PE0] set trigger type: edge-triggered
# [STDOUT-CL31$_$PE0] set nlbits
# [STDOUT-CL31$_$PE0] set interrupt priority and level
# [STDOUT-CL31$_$PE0] raise interrupt threshold to max
# [STDOUT-CL31$_$PE0] enable interrupt
# [STDOUT-CL31$_$PE0] lower interrupt threshold (interrupt should happen)
# [STDOUT-CL31$_$PE0] someone wrote to mailbox!
```

Successivamente, una volta modificato il bus per integrare la mailbox SCMI e aggiunte le nuove connessioni per accedere ai suoi registri come descritto nel capitolo 4, il test è stato rieseguito per verificare che fossero state mantenute le funzionalità validate nella prova precedente.

5.2.2 Validazione Mailbox

A questo punto, per verificare che gli indirizzi della mailbox fossero accessibili almeno da parte del processore, è stata eseguita una nuova simulazione. In questa, la struttura del testbench prevede la scrittura di valori arbitrari all'interno dei registri della mailbox attraverso l'interfaccia AXI4 e la successiva lettura dagli stessi indirizzi per eseguire un confronto tra i dati letti e quelli scritti.

Una volta verificato che fosse possibile accedere, dall'esterno, ai registri della mailbox SCMI, è stato dunque analizzato il meccanismo di generazione di interrupt da parte della mailbox sui suoi segnali di doorbell connessi al CLIC di ControlPULP. Per analizzare questa capacità della logica interna delle mailbox, è stato preparato un testbench in grado di scrivere attraverso il tratto del bus AXI4 tra il processore e la memoria condivisa in analisi. In questo caso, il compito del testbench è stato quello di scrivere nella prima mailbox del register file nel doorbell register (descritto nel capitolo 4), il valore "1" nella posizione del bit meno significativo, in modo da poter osservare una successiva transizione del segnale di doorbell dal valore "0" a "1" logico. In figura 5.2 è possibile notare come, nelle forme d'onda prodotte dalla simulazione, avvenga una transizione del segnale doorbell durante il ciclo di clock in cui avviene la scrittura all'interno del registro della mailbox.

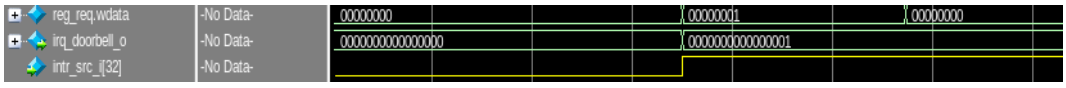

Figura 5.2: Forme d'onda relative al segnale di doorbell interrupt della mailbox

Al fine di verificare sia la possibilità di ControlPULP di accedere alla mailbox che di testare i meccanismi per cui, una volta rilevato il fronte sul segnale di doorbell, fosse in grado di eseguire una Interrupt Service Routine, sono stati predisposti un nuovo testbench e un nuovo codice per il manager core. Il test che è stato eseguito nuovamente in simulazione, prevede che il testbench si occupi soltanto di eseguire la fase di boot e di attendere il valore di ritorno di ControlPULP che indica la fine dell'esecuzione del codice da parte dal manager core. In questo caso dunque, è stato previsto che il manager core accedesse al registro di doorbell della mailbox in modo da

generare un segnale di interrupt verso il CLIC. Per fare ciò, all'interno del codice destinato all'esecuzione da parte del manager core di ControlPULP, è stata prima di tutto inserita una fase di inizializzazione del CLIC in cui:

- viene preparata la Interrupt Vector Table (IVT);
- viene abilitata la gestione vettorizzata per l'interrupt ID relativo alla mailbox in cui viene eseguita la scrittura in questo test;
- viene impostata la rilevazione di una condizione di interrupt sul fronte del segnale (che in questo caso sarà il doorbell di una mailbox);
- si imposta il numero di bit per codificare il livello di priorità degli interrupt gestiti dal CLIC;
- si sceglie il livello di priorità dell'interrupt relativo all'ID in questione;
- viene selezionata la funzione da eseguire come ISR;
- viene abilitato il CLIC in modo che possa funzionare secondo le configurazioni eseguite.

Mentre per la fase di configurazione del CLIC, le procedure sono note dal capitolo 4, è necessario spiegare il meccanismo di gestione delle ISR interno a ControlPULP per comprendere il setup della IVT. La IVT è rappresentata da una tabella in cui, per ogni riga è presente il nome di una funzione di gestione dell'interrupt. A sua volta, la funzione di gestione, oltre a salvare lo stato dei registri della CPU del manager core (per poi ripristinarli alla fine della sua esecuzione), va a richiamare un'ulteriore funzione di ISR impostata nella fase di configurazione descritta sopra. Mentre il resto del firmware destinato a ControlPULP è stato scritto in linguaggio C, la IVT e la funzione di gestione degli interrupt sono state scritte in linguaggio Assembly in modo tale da permettere l'accesso ai registri interni alla CPU in maniera più semplice. Oltre a questa fase di preparazione, sono presenti le funzioni per andare a scrivere all'interno della mailbox subito dopo l'inizializzazione del CLIC e la descrizione della ISR. Per quanto riguarda la fase di scrittura, sono state utilizzate funzioni già presenti all'interno delle librerie di ControlPULP; la ISR invece è stata predisposta per generare una stampa di avviso dell'avvenuta ricezione del segnale di interrupt. Come è

possibile osservare nel listato che segue, le stampe all'interno del simulatore, permettono di tenere traccia di tutte le azioni eseguite da ControlPULP, fino all'esecuzione della routine di interrupt.

```
# [STDOUT-CL31$_$PE0] test csr accesses
# [STDOUT-CL31$_$PE0] set up vector table
# [STDOUT-CL31$_$PE0] set selective hardware vectoring
# [STDOUT-CL31$_$PE0] set trigger type: edge-triggered
# [STDOUT-CL31$_$PE0] set nlbits
# [STDOUT-CL31$_$PE0] set interrupt priority and level
# [STDOUT-CL31$_$PE0] raise interrupt threshold to max
# [STDOUT-CL31$_$PE0] enable interrupt
# [STDOUT-CL31$_$PE0] lower interrupt threshold (interrupt should happen)
# [STDOUT-CL31$_$PE0] someone wrote to mailbox!
```

A questo punto, al fine di avvicinarsi alla situazione del test che verrà eseguito direttamente su FPGA, è stato prevista una verifica ulteriore. In questa, rispetto al test precedente, è stata spostata la funzione di scrittura all'interno del doorbell register della mailbox, dal codice destinato all'esecuzione da parte del manager core di ControlPULP al testbench. Quindi non sarà più il manager core ad eseguire sia le scritture nella mailbox che la ISR come nel test precedente. Il testbench in questo caso dunque provvederà ancora una volta a caricare il codice nella memoria di ControlPULP, eseguirà la scrittura in un registro della mailbox e attenderà la fine dell'esecuzione del codice da parte del manager core. Anche in questo caso sono stati ottenuti i risultati riportati nel listato precedente, confermando il funzionamento del meccanismo di interrupt della mailbox anche in caso di scrittura da parte di un master diverso da ControlPULP, e la possibilità del power controller di servire le relative ISR.

Nonostante siano già stati presentati diversi test, risulta evidente come non siano ancora state verificate tutte le nuove funzionalità integrate nell'architettura di ControlPULP. Infatti, è necessario spiegare come sia stata esaminata la capacità da parte di tale power controller di eseguire letture dai registri della mailbox oltre che scritture. Per andare a indagare questa possibilità, è stato previsto un test in cui il testbench, simulazione del processore da connettere a ControlPULP, esegue delle scritture all'interno dei registri della mailbox seguendo il layout dei messaggi SCMI e riportando i seguenti valori:

- header: 0x4105
- payload[0] : 0x01

- payload[1] : 0x02

Corrispondenti ai seguenti valori dei campi dell'header:

- message type: 0x01
- protocol id : 0x10
- message id: 0x05

Successivamente alle scritture del testbench, ControlPULP esegue delle letture da tali registri per eseguire una prima verifica di conformità dei messaggi rispetto a quanto previsto dalla specifica SCMI. Ancora una volta, il testbench si occupa anche delle fasi di boot e di attesa della fine dell'esecuzione del codice da parte di ControlPULP. Per quanto riguarda il codice caricato nella memoria di ControlPULP, invece, le istruzioni risultano identiche a quelle del test precedente, ad eccezione della routine di interrupt che riporta attraverso delle stampe i valori letti dai registri della mailbox ed esegue un check di conformità dell'header del messaggio. Oltre a queste operazioni, riscrive l'header del messaggio ricevuto all'interno della mailbox, seguendo la procedura descritta all'interno della specifica SCMI. Di seguito si possono osservare le stampe riportate durante la simulazione che attestano la verifica del corretto funzionamento del sistema.

```
# [STDOUT-CL31$_$PE0] test csr accesses
# [STDOUT-CL31$_$PE0] set up vector table
# [STDOUT-CL31$_$PE0] set selective hardware vectoring
# [STDOUT-CL31$_$PE0] set trigger type: edge-triggered
# [STDOUT-CL31$_$PE0] set nlbits
# [STDOUT-CL31$_$PE0] set interrupt priority and level
# [STDOUT-CL31$_$PE0] raise interrupt threshold to max
# [STDOUT-CL31$_$PE0] enable interrupt
# [STDOUT-CL31$_$PE0] lower interrupt threshold (interrupt should happen)
# [STDOUT-CL31$_$PE0] someone wrote to mailbox!
# [STDOUT-CL31$_$PE0] channel flags: 0
# [STDOUT-CL31$_$PE0] header: 4105
# [STDOUT-CL31$_$PE0] message type: 1
# [STDOUT-CL31$_$PE0] protocol id: 10
# [STDOUT-CL31$_$PE0] message id: 5
# [STDOUT-CL31$_$PE0] payload 0: 1
# [STDOUT-CL31$_$PE0] payload 1: 2
```

5.2.3 Test su FPGA

A questo punto, verificato il funzionamento della nuova architettura del bus di ControlPULP a livello di simulazione, le successive prove per la validazione sono state eseguite su FPGA. Come spiegato nel paragrafo precedente, in questo caso, non è presente un processore HPC in hardware, ma un processore ARM multicore destinato alla sua emulazione, connesso alla logica programmabile dell'FPGA. Nei test seguenti, il processore ARM contenuto all'interno del SoC, è stato programmato direttamente per interagire con la mailbox istanziata nel lato della logica programmabile visto nel paragrafo del setup sperimentale. Il linguaggio scelto per programmare il processore in questione è il C++. Il primo test eseguito prevede la stessa struttura dell'ultima verifica eseguita in simulazione: il processore scrive all'interno della mailbox un messaggio, questa fa partire una ISR all'interno di ControlPULP, attraverso la catena di eventi già descritta, e vengono riportati attraverso una stampa i risultati dell'esecuzione. In questo caso le stampe vengono riportate su un display attraverso la porta UART della board ZCU102 e risultano identiche a quelle ottenute attraverso simulazione. Per quanto riguarda il codice da caricare lato processore, esso contiene le stesse istruzioni precedentemente assegnate al testbench che riescono ad eseguire le scritture all'interno della mailbox. Per eseguire tali scritture, in questo caso, vengono utilizzate delle funzioni che richiedono soltanto l'indirizzo a cui eseguire la scrittura (mappato nello spazio degli indirizzi del processore) e il valore da scrivere; di seguito viene riportato il codice per la scrittura all'interno del registro della mailbox necessaria a generare l'impulso di doorbell.

```
std::cout << "Ringing the doorbell\n";
axi_ps_scmi_mailbox_doorbell_address = 0xA6000000 + 0x20;
axi_scmi_mailbox_payload = 0x00000001;
scmi_mailbox.write_u32( axi_ps_scmi_mailbox_doorbell_address,
                        axi_scmi_mailbox_payload );
```

Essendo stati ottenuti gli stessi risultati relativi al test eseguito in simulazione, il design è stato dunque validato anche attraverso l'emulazione.

5.2.4 Test Firmware SCMI di ControlPULP

Una volta testate le funzionalità legate all'architettura hardware anche su FPGA, è stato verificato il corretto funzionamento del firmware presentato

nel capitolo 4. La struttura del test prevede che vi sia ancora una volta un testbench che popola i registri della memoria condivisa. Una volta rilevato il doorbell interrupt, ControlPULP andrà ad eseguire la relativa ISR richiamando la funzione di decodifica appartenente al modulo Common SCMI e avviando la catena di eventi descritta in figura 4.7. Oltre alla verifica del firmware SCMI, in questo caso è stato anche validato il metodo messo a disposizione dalle API di FreeRTOS che permette di eseguire il deferring della funzione per decodificare il messaggio. All'interno della Interrupt Service Routine, infatti, non viene direttamente richiamata la funzione *handle scmi* che si occupa della decodifica; bensì viene fornita attraverso un puntatore al metodo di FreeRTOS *xTimerPendFunctionCallFromISR*. Tale metodo permette di ritornare immediatamente alle istruzioni che sarebbero state eseguite se non fosse arrivato l'impulso di doorbell (e non fosse stata eseguita la ISR) per poi eseguire la decodifica quando permesso dallo scheduler di FreeRTOS. Di seguito viene riportato il codice della ISR; la direttiva *define* viene utilizzata per replicare la stessa routine per ogni mailbox dalla quale sia possibile ricevere un impulso di doorbell.

```
#define IRQ_HANDLE_SCMI(SRC)
void irq##SRC##_handle_scmi(void)
{
        BaseType_t xHigherPriorityTaskWoken = pdFALSE;
        xTimerPendFunctionCallFromISR(handle_scmi, NULL, 0,
                                      &xHigherPriorityTaskWoken);
        portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
}
```

L'esecuzione del codice si conclude con la scrittura da parte del driver SCMI all'interno della mailbox in cui è stato ricevuto il messaggio inizialmente.

Per implementare questo test, per prima cosa è stato necessario andare a inizializzare i moduli del firmware SCMI; tale inizializzazione è stata eseguita nel codice del modulo common SCMI e sull'unico protocollo implementato al momento del test: il base protocol. L'inizializzazione del modulo common, è stata eseguita in modo tale da:

- esporre le API del driver SCMI precedentemente descritto;
- abilitare gli agenti/mailbox con id pari a 0 e 2;
- abilitare il base protocol;

- rendere disponibile il base protocol al primo agent ma non al secondo.

A questo punto è stato inizializzato anche il base protocol andando a fornire l'insieme di funzioni da richiamare in base al *protocol id* rilevato in fase di lettura dalla mailbox.

Successivamente, seguendo le procedure di configurazione del CLIC, è stata compilata la IVT e sono stati abilitati gli interrupt relativi ai due agenti indicati in precedenza tralasciando la procedura per avviare il funzionamento dell'interrupt controller. Infatti, se questa fosse stata eseguita direttamente, non ci sarebbe stato modo di andare ad eseguire le successive impostazioni. Come ultima inizializzazione, viene abilitato il funzionamento dello scheduler di FreeRTOS; se questa operazione fosse stata eseguita prima di eseguire altre impostazioni (come ad esempio quelle relative al CLIC) avrebbe impedito di portare a termine questa fase, facendo partire il task di *idle*, descrivibile come un loop infinito da cui è possibile uscire soltanto attraverso le funzioni dello scheduler stesso e da cui non si sarebbe tornati in ogni caso alla fase di inizializzazione. Per non generare dunque un conflitto tra l'attivazione dello scheduler e l'abilitazione per il funzionamento del CLIC, quest'ultima fase è stata eseguita all'interno di un task che viene richiamato subito dopo l'istruzione per abilitare lo scheduler di FreeRTOS. Per quanto riguarda la fase di test vera e propria, sono state eseguite più prove: per prima cosa è stato verificato che in corrispondenza dell'arrivo di un impulso di doorbell, il meccanismo degli interrupt e l'esecuzione della funzione di decodifica attraverso deferring andassero a buon fine. Per questo primo test, per non aggiungere complicazioni, si è scelto di inviare soltanto un messaggio nella mailbox corrispondente all'agente 0. Una volta verificato il funzionamento di questo meccanismo alla base, sono stati eseguiti diversi tentativi andando a cambiare il *message id* dei messaggi SCMI in modo tale da rivelare se tutte le risposte previste dal base protocol fossero scritte nella mailbox con successo. Successivamente sono stati generati e inviati messaggi all'interno della stessa mailbox con *protocol id* relativi a protocolli non supportati, *header* non conformi alle specifiche SCMI o che non rispettassero i vincoli sulla lunghezza indicata del payload, *message id* non supportati all'interno del protocollo base, in modo tale da generare condizioni d'errore e le risposte da parte del firmware di decodifica. Il valore di tali risposte infatti, è definito all'interno della descrizione della specifica SCMI e prevede un insieme di codici d'errore da riportare

nel primo registro dedicato al payload. Come ultime prove, sono stati inviati messaggi dall'interfaccia AXI4 del processore simulato alla mailbox corrispondente all' agente numero 1 (non abilitato) e all'agente 2 indicando all'interno dell'header l'id del base protocol non supportato da quest'ultimo. Anche in questo caso è stata verificata la correttezza dei messaggi d'errore riportati all'interno della mailbox. Si specifica che il riconoscimento di un agente attivo, in questa versione dell'implementazione del firmware SCMI di ControlPULP, viene eseguita in due passaggi. In primo luogo, se non si intende attivare un agente, può risultare pratico non andare ad abilitare l'ingresso del CLIC corrispondente al suo segnale di *doorbell*; secondariamente, se dovesse verificarsi una situazione in cui il CLIC venisse comunque impostato per rilevare tale interrupt, bisogna considerare che il modulo Common SCMI è in grado di rilevare in ogni caso che il messaggio proviene da un agente non abilitato, perchè tutti i protocolli da esso supportati risultano non inizializzati.

5.3 Caratterizzazione

In questo paragrafo vengono riportati i risultati relativi alla caratterizzazione del nuovo design di ControlPULP, che quindi include le modifiche architetturali viste nel capitolo 4, e all'esecuzione del firmware SCP.

In maniera concorde a quanto introdotto nel capitolo 3, in tabella 5.1 viene riportato l'utilizzo delle risorse all'interno della logica programmabile del SoC XCZU9EG della ZCU102 per l'implementazione del design di ControlPULP che include il bus AXI4 tra PS e PL, e la mailbox. In aggiunta alle risorse già presentate, sono presenti anche voci per i flip-flop[2] anch'essi contenuti all'interno dei blocchi CLB. Come è possibile notare, l'utilizzo delle LUT dei CLB risulta molto alto e pari al 92% di quelle disponibili. Questo percentuale può sembrare eccessiva ma la motivazione di un così grande utilizzo di LUT risiede anche nella natura della mailbox. La mailbox infatti è una memoria, pertanto il software utilizzato per la sintesi, a meno dell'impiego di specifiche direttive, andrà ad occupare prima di tutto delle Look Up Table per implementarla.

Oltre alle statistiche sull'implementazione su FPGA, risultano rilevanti anche le caratterizzazioni eseguite sui tempi di esecuzione del firmware SC-

[2]Tipologia di registri.

Risorsa	Utilizzo	Disponibili	Utilizzo [%]
LUT	252154	274080	**92**
FF	131558	548160	24
BRAM	274	912	30
DSP	101	2520	4
IO	82	328	25

Tabella 5.1: Mapping di ControlPULP sulla Xilinx UltraScale+ ZCU102, utilizzo delle risorse

MI riportate di seguito. Il test utilizzato risulta analogo a quello esposto nel paragrafo precedente in cui è stato validato il funzionamento del firmware SCMI di ControlPULP attraverso il deferring della funzione di decodifica. L'unica differenza rispetto al codice utilizzato nel test di validazione è rappresentata dall'assenza di stampe di debug; esse infatti avrebbero rallentato l'esecuzione delle istruzioni necessarie alla decodifica. Inoltre, tali stampe non risultano utili al di fuori del test di validazione. Anche in questo caso, il test è stato eseguito in simulazione. Al fine di caratterizzare i tempi di esecuzione nel caso peggiore, si è presa in considerazione la combinazione dei valori scritti all'interno del messaggio destinato alla mailbox che riuscisse a passare tutti i controlli eseguiti durante la decodifica. In questo modo, poichè il firmware non rileva condizioni d'errore, si riesce a misurare il tempo di decodifica nella situazione in cui il messaggio riesce ad arrivare fino al livello dei protocolli e ad essere mandato indietro verso la mailbox con i nuovi valori. Inoltre, poichè i tempi d'esecuzione dipendono dalla frequenza di clock di ControlPULP, i valori trovati sono stati riportati anche in cicli di clock normalizzati rispetto ad una frequenza di clock del bus AXI4 e della mailbox, pari a 100MHz e quello del manager core, pari a 20MHz.

Come riportato in tabella 5.2, il tempo di esecuzione totale per la decodifica di un messaggio risulta pari a $17.43\mu s$ quindi 348 cicli a 20MHz. Questo valore si riferisce al tempo che intercorre tra la fine della scrittura sul bus AXI4 da parte del processore (ancora una volta simulato attraverso scritture da testbench) e la fine della scrittura all'interno del registro Channel status della mailbox da parte di ControlPULP. Si specifica che anche se questa misura risulta avere un valore abbastanza alto per un'applicazione

Misura	Tempo	Cicli	Frequenza
Tempo Totale	17.43 μs	872	20 MHz
Tempo Interupt	0.02 μs	100 MHz	

Tabella 5.2: Tabella dei tempi misurati

real-time, bisogna considerare come sia stata eseguita su FPGA, che rappresenta una versione meno performante rispetto al circuito integrato su cui può essere implementata l'architettura di ControlPULP. In figura 5.3, è possibile notare attraverso la posizione del cursore giallo, come l'inizio del tempo d'esecuzione sia stato posto alla fine della procedura di scrittura sul bus AXI4 all'indirizzo 0xA6000024 evidenziato e corrispondente al registro di doorbell della prima mailbox del register file delle mailbox. In figura 5.4 si può invece osservare la configurazione dei segnali corrispondente a quella che si è considerata la fine del tempo d'esecuzione del codice di decodifica. Come già anticipato, questa corrisponde alla fine degli impulsi di handshake presenti sull'interfaccia tra l'ultimo convertitore della mailbox e l'interfaccia col suo register file. In questo caso viene evidenziato l'indirizzo relativo al registro di Channel Status della prima mailbox del register file, pari a 0xA6000004. Infatti, solo successivamente alla scrittura in questo registro da parte di ControlPULP, il processore che nel frattempo sta eseguendo letture a polling allo stesso indirizzo, potrà elaborare la risposta.

Al fine di capire quale percentuale del tempo totale della decodifica fosse rappresentata dal periodo che intercorre tra la scrittura all'interno del registro di doorbell della mailbox da parte del processore e l'inizio dei meccanismi interni al manager core necessari all'esecuzione della ISR è stata eseguita un'ulteriore misurazione. Questa prende come riferimento iniziale ancora una volta la stessa configurazione dei segnali utilizzata per il calcolo del tempo totale ma termina nel momento in cui il manager core invia l'impulso di hacknowledgement verso il CLIC per segnalare la ricezione dell'id relativo al segnale di doorbell di una mailbox. In figura 5.5 sono visibili i due segnali valid e ready relativi ai segnali di handshake tra CLIC e il manager core di ControlPULP che passano dal valore logico "0" a "1" in corrispondenza dell'interrupt id 0x20 relativo all'impulso di doorbell della prima mailbox del register file. Come si può osservare dalla seconda voce della tabella 5.2 questo tempo risulta di soli 0.02μs ovvero 2 cicli di clock,

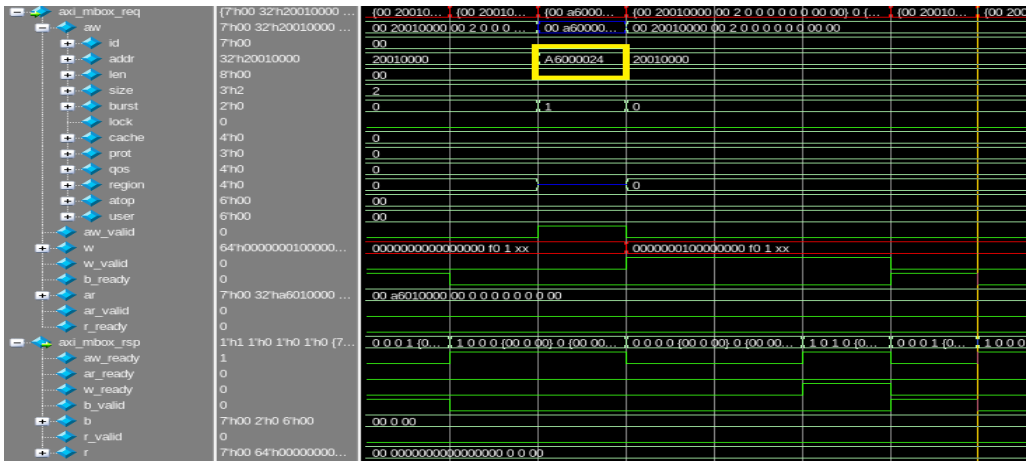

Figura 5.3: Forme d'onda della scrittura su bus AXI4 destinate al registro di doorbell della mailbox

dunque una percentuale minima rispetto al tempo totale precedentemente misurato.

Figura 5.4: Forme d'onda relative al register file della mailbox

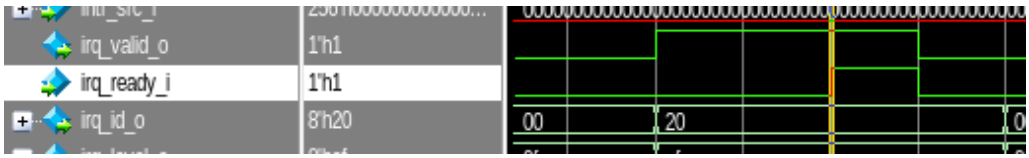

Figura 5.5: Forme d'onda relative all'handshake tra CLIC e manager core di ControlPULP

Capitolo 6

Conclusioni

In questo progetto di tesi è stato modificato il design hardware di ControlPULP per integrare una nuova interfaccia di comunicazione verso il processore HPC controllato, basata sul bus AXI4. Per raggiungere questo obiettivo, è stata introdotta all'interno della nuova interfaccia una memoria condivisa che seguisse le caratteristiche di layout della mailbox espresse dalla specifica SCMI. Oltre a questo, è stata modificata la struttura del bus delle connessioni tra ControlPULP e il processore in modo tale che le nuove risorse hardware fossero accessibili da entrambi i master ad esso connessi. Le nuove funzionalità sono state validate attraverso l'utilizzo di opportuni testbench in simulazione e dei test eseguiti direttamente su supporto FPGA, dimostrando il corretto funzionamento dell'interfaccia di comunicazione di ControlPULP anche in hardware. Oltre all'integrazione hardware sono stati sviluppati i driver necessari per l'accesso alla mailbox SCMI in scrittura e lettura attraverso bus AXI4 da parte di ControlPULP. Ad un livello d'astrazione più alto rispetto ai driver, è stato sviluppato un firmware per la decodifica dei messaggi SCMI da parte di ControlPULP, mantenendo il grado di scalabilità e modularità presentato dal firmware SCP di ARM. Il firmware di decodifica è stato a sua volta caratterizzato attraverso misurazioni relative ai tempi d'esecuzione fissando anche un riferimento per eventuali test successivi eseguiti sulla stessa architettura. Infine, la nuova interfaccia sviluppata è stata caratterizzata attraverso le statistiche della sintesi RTL destinata al SoC ZU9EG.

6.0.1 Prospettive Future

Per quanto concerne le prospettive future, il design di ContolPULP risulta ottimizzabile sia dal punto di vista dell'hardware che del software. Per quanto riguarda i miglioramenti hardware, a livello di implementazione su FPGA, risulta evidente come sia possibile trarre beneficio emulando la mailbox attraverso la RAM presente sul SoC piuttosto che utilizzando le LUT. Questo permetterebbe di rendere disponibili delle LUT per l'implementazione di ulteriori reti logiche sullo stesso SoC. Un altro miglioramento che può essere eseguito a livello di design hardware per migliorare l'occupazione d'area anche di un eventuale ASIC[1] che integri l'architettura di ControlPULP, è rappresentato dalla riorganizzazione dei registri di doorbell. In questo caso, come si è visto nel capitolo 4, per generare un impulso di interrupt relativo ad una mailbox è necessario scrivere il valore "1" per il bit meno significativo del relativo registro di doorbell. In questo modo, viene istanziato un intero registro da 32 bit per usare soltanto un bit per registro. Un primo miglioramento potrebbe consistere nell'utilizzo di una serie di registri in cui vengono utilizzati tutti i bit disponibili per codificare lo stato dei segnali di interrupt in uscita dalla mailbox. Sarebbe poi necessario un blocco decoder per selezionare gli ingressi del CLIC che si intende attivare. Questo meccanismo, oltre a ridurre l'utilizzo di registri all'interno della mailbox per gestire gli impulsi di doorbell andrebbe anche a semplificare il routing tra tale blocco e il decoder posto in prossimità del CLIC. Bisogna però specificare che potrebbe essere necessario sviluppare ulteriori meccanismi di arbitraggio delle scritture nei registri di doorbell e anche che la latenza di trasmissione degli impulsi di interrupt verso il CLIC aumenterebbe inevitabilmente. Come visto all'interno del capitolo 5 però, il tempo di generazione dell'interrupt verso il manager core risulta trascurabile rispetto ai tempi d'esecuzione del firmware di decodifica, per cui anche un aumento della latenza in questione dovuta alle reti logiche appena descritte, sarebbe trascurabile rispetto ai tempi totali di risposta ai messaggi SCMI da parte di ControlPULP.

Anche dal punto di vista del software sarebbe possibile introdurre dei miglioramenti. In primo luogo, come è stato specificato nei capitoli precedenti, è possibile implementare all'interno dei firmware di decodifica, il supporto a più protocolli facenti parte della specifica SCMI. Infatti, nella prima ver-

[1] Application Specific Integrated Circuit

sione del firmware è stato previsto soltanto il base protocol, obbligatorio da specifica SCMI. Secondariamente, è possibile anche esplorare l'insieme dei meccanismi messi a disposizione dalle API di FreeRTOS, in modo tale da sostituire il deferring della funzione di decodifica vista nel capitolo 5, con altre modalità d'esecuzione e arbitraggio delle ISR, in modo tale da ridurre l'utilizzo di RAM a disposizione del manager core e migliorare i tempi d'esecuzione. Entrambi gli obiettivi di miglioramento del firmware possono essere facilmente raggiunti grazie al grado di modularità del firmware sviluppato in questo progetto di tesi, approfondito nel paragrafo 4.

Bibliografia

[1] Buildroot. https://buildroot.org/, 2022.

[2] Freertos. https://www.freertos.org/, 2022.

[3] Questasim. https://eda.sw.siemens.com/en-US/ic/questa/simulation/advanced-simulator/, 2022.

[4] ARM. *AMBA AXI and ACE Protocol Specification*, 2022.

[5] Arm. Scp firmware. https://github.com/ARM-software/SCP-firmware, 2022.

[6] T. Burd, N. Beck, S. White, M. Paraschou, N. Kalyanasundharam, G. Donley, A. Smith, L. Hewitt, and S. Naffziger. "zeppelin": An soc for multichip architectures. *IEEE Journal of Solid-State Circuits*, 54(1):133–143, 2019.

[7] P. D. Document. *Arm System Control and Management Interface*, 2022.

[8] S. Document. *Advanced Configuration and Power Interface Specification*, 2022.

[9] ETH. Controlpulp repository. https://iis-git.ee.ethz.ch/pms/control-pulp/-/tree/arm-scmi/tests/control-pulp-tests/clic, 2022.

[10] ETH. Controlpulp repository. https://iis-git.ee.ethz.ch/pms/control-pulp, 2022.

[11] A. Grenat, S. Sundaram, S. Kosonocky, R. Rachala, S. Sambamurthy, S. Liepe, M. Rodriguez, T. Burd, A. Clark, M. Austin, and S. Naffziger. 4.2 increasing the performance of a 28nm x86-64 microprocessor through system power management. In *2016 IEEE International Solid-State Circuits Conference (ISSCC)*, pages 74–75, 2016.

[12] P. Guide. *High Performance Computing: Tuning Guide for AMD EPYC 7002 Series Processors*, 2020.

[13] P. Guide. *Intel 64 and IA-32 Architectures Software Developer's Manual*, 2022.

[14] U. Guide. *ZCU102 Evaluation Board*, 2022.

[15] D. Hackenberg, R. Schöne, T. Ilsche, D. Molka, J. Schuchart, and R. Geyer. An energy efficiency feature survey of the intel haswell processor. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, pages 896–904, 2015.

[16] IBM. Openpower occ. `https://github.com/open-power/occ`, 2022.

[17] K. N. Khan, M. Hirki, T. Niemi, J. K. Nurminen, and Z. Ou. Rapl in action: Experiences in using rapl for power measurements. *ACM Trans. Model. Perform. Eval. Comput. Syst.*, 3(2), Mar. 2018.

[18] C. E. Leiserson, N. C. Thompson, J. S. Emer, B. C. Kuszmaul, B. W. Lampson, D. Sanchez, and T. B. Schardl. There's plenty of room at the top: What will drive computer performance after moore's law? *Science*, 368(6495), 2020.

[19] Y. Liu and H. Zhu. A survey of the research on power management techniques for high-performance systems. *Software: Practice and Experience*, 40, 2010.

[20] S. Naffziger, B. Liu, and M. Touzelbaev. Performance state boost for multi-core integrated circuit, 2013. US Patent 4,741,207.

[21] A. Ottaviano, R. Balas, G. Bambini, C. Bonfanti, S. Benatti, D. Rossi, L. Benini, and A. Bartolini. Controlpulp: A risc-v power controller for hpc processors with parallel control-law computation acceleration.

In *Embedded Computer Systems: Architectures, Modeling, and Simulation: 22nd International Conference, SAMOS 2022, Samos, Greece, July 3–7, 2022, Proceedings*, page 120–135, Berlin, Heidelberg, 2022. Springer-Verlag.

[22] I. Ripoll and R. Ballester. Period selection for minimal hyper-period in real-time systems. 2014.

[23] T. Rosedahl, M. Broyles, C. Lefurgy, B. Christensen, and W. Feng. Power/performance controlling techniques in openpower. In *International Conference on High Performance Computing*, pages 275–289. Springer, 2017.

[24] D. Rossi, I. Loi, G. Haugou, and L. Benini. Ultra-low-latency lightweight dma for tightly coupled multi-core clusters. In *Proceedings of the 11th ACM Conference on Computing Frontiers*, pages 1–10, 2014.

[25] E. Rotem, A. Naveh, A. Ananthakrishnan, E. Weissmann, and D. Rajwan. Power-management architecture of the intel microarchitecture code-named sandy bridge. *IEEE Micro*, 32(2):20–27, 2012.

[26] R. Schöne, T. Ilsche, M. Bielert, A. Gocht, and D. Hackenberg. Energy efficiency features of the intel skylake-sp processor and their impact on performance. In *2019 International Conference on High Performance Computing Simulation (HPCS)*, pages 399–406, 2019.

[27] T. Singh, S. Rangarajan, D. John, C. Henrion, S. Southard, H. McIntyre, A. Novak, S. Kosonocky, R. Jotwani, A. Schaefer, E. Chang, J. Bell, and M. Co. 3.2 zen: A next-generation high-performance ×86 core. In *2017 IEEE International Solid-State Circuits Conference (ISSCC)*, pages 52–53, 2017.

[28] A. Technology. *AMD64 Architecture Programmer's Manual Volume 2: System Programming*, 2022.

[29] O. Titan. Open titan register tool. `https://docs.opentitan.org/doc/rm/register_tool/`, 2022.