UNIVERSITÀ DEGLI STUDI DI MESSINA

DIPARTIMENTO DI INGEGNERIA

CORSO DI LAUREA TRIENNALE IN

INGEGNERIA ELETTRONICA ED INFORMATICA

Implementazione di meccanismi per la Full e Partial Reconfiguration di FPGA in architetture IoT-Cloud

Tesi di laurea di:
Claudio Di Salvo

Relatore:
Chiar.mo Prof. Francesco Longo

Anno Accademico 2021/2022

Indice

Elenco delle figure

Elenco dei codici

Capitolo 1

Introduzione

Negli ultimi tempi nel panorama informatico globale è diventato d'uso comune il termine *Cloud* che rappresenta un'infrastruttura di rete remota alla quale, tramite Internet, si è in grado di interfacciarsi. All'interno del *Cloud* è possibile trovare varie risorse come spazi d'archiviazione, indirizzi di rete e macchine virtuali.

Con l'avvento dell' *Internet of Things* nei sistemi *Cloud* non è raro trovare anche dispositivi con i quali è possibile interfacciarsi, interconnettersi, acquisire ed elaborare dati. Da questo panorama vengono esclusi dispositivi basati su tecnologia *Field Programmable Gate Array* o *FPGA*, per via della loro integrazione con il sistema *Cloud*.

Tale caratteristica è data dalla natura del dispositivo che, a differenza dei sistemi comunemente usati, necessiterà di un percorso diametralmente opposto per la programmazione, composto da un posizionamento dei blocchi logici che formano le *FPGA* ed una traduzione del comportamento logico del sistema.

Inoltre, la strumentazione per effettuare la progettazione finale delle *FPGA* è proprietaria ed a sorgente chiuso.

L'elaborato si pone come obiettivo quello di creare strumenti, gratuiti ed a codice sorgente aperto, che permettano una più semplice integrazione nei sistemi *Cloud* già esistenti. In particolare, si è progettato l'interfacciamento del sistema con l'ambiente cloud e la sua completa e parziale riprogrammazione. L'implementazione di tale sistema è stata testata tramite la *Zedboard*. La realizzazione dell'ambiente *Cloud* e le politiche di gestione della risorsa rappresentano il lavoro necessario alla finalizzazione del progetto.

Nel primo capitolo si discuteranno gli argomenti di base su cui si incentra l'elaborato.
Introducendo i dispositivi *FPGA* e le loro funzionalità. In seguito verranno trattati i linguaggi di descrizione dell'hardware, di conseguenza gli strumenti tramite i quali è possibile descrivere il comportamento dell'*FPGA*.
Inoltre verrà fornita un'introduzione sulle tecnologie utilizzate nello sviluppo e che saranno necessarie al continuo del lavoro.
Nel Secondo capitolo si discuterà la soluzione ideata al fine del raggiungimento degli obiettivi di progetto.
Nel Terzo capitolo verrà implementato il sistema operativo per le schede *FPGA*, approfondendo le procedure di programmazione del kernel e descrivendo il workflow necessario al fine di avere un sistema connesso al Cloud.
Nel Quarto Capitolo si approfondirà l'interfacciamento interno del *FPGA*, andando a descrivere i vari bus e verrà implementato un driver per l'interfacciamento con il GPIO della scheda tramite il sistema operativo. Questo driver ci permetterà l'acquisizione e l'elaborazione in loco dei dati.
Nel Quinto capitolo si discuterà sull'interfaccia necessaria alla riprogrammazione della scheda, sviluppando uno script che permette la *Full Reconfiguration*. Inoltre, verranno discusse le metodologie di riprogrammazione parziale e le sue problematiche.
Infine, verrà fornita una guida completa ed integrale alla configurazione e all'uso di tutte le componenti trattate.
Per completezza i codici sorgenti inerenti alla trattazione verranno inseriti nella loro totalità nell'appendice, per facilitarne la fruizione.

Capitolo 2

Background

Al fine di fornire le conoscenze necessarie alla comprensione od eventuale continuazione del lavoro svolto in questo elaborato, risulta necessario introdurre gli argomenti che si andranno ad affrontare durante la tesi, partendo dai dispositivi *FPGA*.

2.1 Field Programmable Gate Array

Sin dall'invenzione dei computer, la loro progrettazione è avvenuta con una rapida evoluzione. A metà degli anni '80, un tipo di architettura diverso da quello convenzionale fu sviluppato, le *Field Programmable Gate Array* (*FPGA*). Esse furono costruite tramite un'interconnessione effettuata dal progettista tra ROM/PROM o array di porte NAND. Le *FPGA* moderne sono una soluzione prefabbricata, elettronicamente programmabile, atta alla progettazione di sistemi digitali ad alte prestazioni per basso volume. Le *FPGA* vengono progettate su CAD appositi, come ad esempio *Vivado*, senza la necessità di dover modificare la struttura fisicamente.

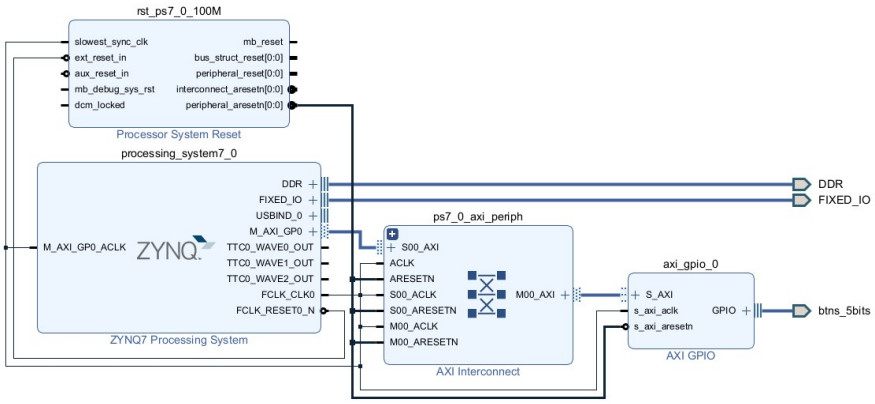


Figura 2.1: Foto di una prototipazione tramite *Vivado* di una *FPGA Zynq-7000* che sfrutta il GPIO

Questa famiglia di dispositivi dev'essere immaginata come un insieme di componenti logici; tali *Configurable Logic Block* (*CLB*), come già detto; possono esser modificati dal progettista. I *CLB* sono composti da Look Up Table (LUT), Full Adders, Registi, Multiplexer e funzioni date dall'implementazioni. Questa divisione in blocchi permette di creare dei cluster interconnessi, rendendo così possibile la sintetizzazione di tutte le possibili operazioni booleane e la realizzazione di sistemi complessi.

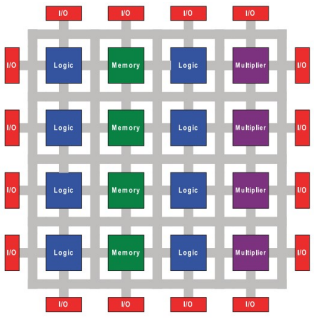


Figura 2.2: Tipica rappresentazione di un *FPGA* [7]

Le *FPGA* per loro natura devono comunicare in qualche modo con altri dispositivi: tale comunicazione è gestita da blocchi detti Input Output Block(IOB); essi garantiscono l'interfacciamento tra le risorse della *Programmable Logic* (PL) ed il mondo esterno. Ogni IOB gestisce un segnale d'input/output ed è in grado di effettuare una conversione, programmabile, tra formati seriali e paralleli, oltre alla gestione di vari standard di I/O e l'eventuale configurazione di Pull-up/Pull-down interni.

Data la sempre più crescente quantità di CLB si può pensare come sia necessario aumentare

l'area di silicio occupata, ma essa è solitamente occupata dai sistemi di interconnessione interna.

Le *FPGA*, essendo completamente riprogrammabili, possono accogliere dei *Cores* custom, detti *Intellectual Property Cores*; essi rappresentano dei dispositivi pre-realizzati al fine di compiere una ben precisa azione, e possono passare da un banale moltiplicatore ad un *Softcore ARM*.

2.1.1 Le FPGA SoC

L'evoluzione dei processi tecnologici ha portato alla nascita di nuovi tipi di FPGA; le *FPGA SoC*, architetture che presentano una interconnessione tra Processore, solitamente della famiglia *ARM*, ed un *FPGA* in un unico dispositivo.

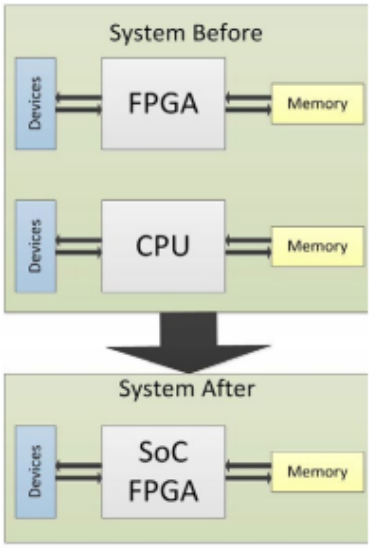


Figura 2.3: Cambio di paradigma con l'introduzione delle FPGA SoC

La loro fusione permette una maggior integrazione, anche lato connettività e quindi Cloud, un minor consumo ed una comunicazione molto più efficiente tra i due grazie ad una larghezza di banda superiore ed una latenza nettamente inferiore a prima.

2.1.2 Architettura Zynq-7000

L'architettura *Zynq-7000* è basata sull'architettura Xilinx All Programmable System On Chip (AP SoC). I dispositivi di questa famiglia sono formati da un dual-single Core ARM

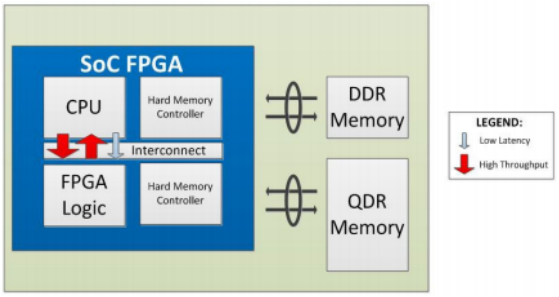


Figura 2.4: Schema rappresentativo FPGA SoC

Cortex A9 su cui si basa il *Processing System* (PS), includendo anche una on-chip memory, interfacce con memorie esterne e le periferiche di connettività con le interfacce, ed una *Programmable Logic* (PL) a 28nm.

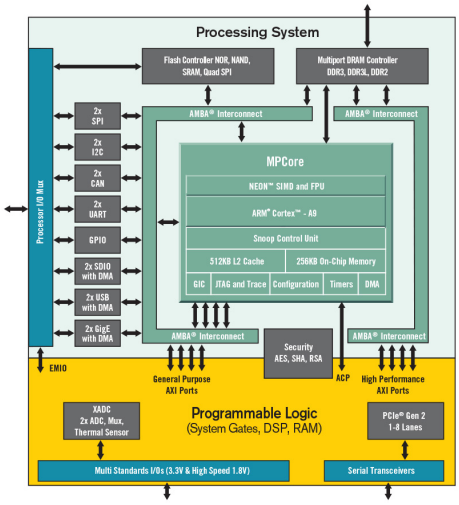


Figura 2.5: Schema rappresentativo ZYNQ-7000[18]

La PS e la PL sono alimentate separatamente e sarà compito del progettista decidere se disabilitare o meno la *Programmable Logic*, il processore eseguirà il boot per primo e sarà sempre necessario anche per usare la parte logica, infatti si usa un approccio software centrico per la PL, ma come vedremo più avanti le due parti non sono scollegate tra di loro e questo approccio ci servirà al fine di effettuare operazioni lato PL.

2.2 Linguaggi di descrizione delle FPGA

Come accennato nel capitolo, 2.1, le *FPGA* vengono implementate tramite l'uso dei *CAD* da un progettista; risulta però utile avere conoscenze dei linguaggi di descrizione dell'hardware quali *VHDL*[5] e *Verilog*[6] . Entrambe le opzioni sono supportate da Vivado e sono valide al fine di descrivere il comporamento di un sistema digitale. Le principali differenze tra i due linguaggi sono le similitudini al linguaggio C, infatti il Verilog ricorda molto la struttura del linguaggio di programmazione, permettendo anche ad un utente novizio la comprensione e l'approccio a questo tipo di linguaggio.
In generale entrambi i linguaggi sono intercambiabili per quanto rigurda la rappresentazione Register Transfer Level, ovvero la schematizzazione del sistema come un flusso di informazioni tra logica e registri. Inoltre, entrambi condividono una struttura gerarchica; è infatti possibile tramite dei costrutti, descrivere un sistema in maniera strutturale, tramite tre possibili approcci:

- Comportamentale: Il programmatore ha il compito di descrivere il comportamento di un sistema tramite un algoritmo.
- Strutturale: Il programmatore descrive singolarmente ogni componente interno al sistema andando poi ad usare una combinazione per rappresentare il sistema.
- DataFlow: Il programmatore scrive il codice emulando il flusso di dati nel sistema.

Ogni approccio necessità la presenza di un *Top Level Entity*, ovvero il file principale che verrà sintetizzato e che permetterà di implementare l'*FPGA*; esso conterrà tutte le componenti, codici ed eventuali librerie o supporti.

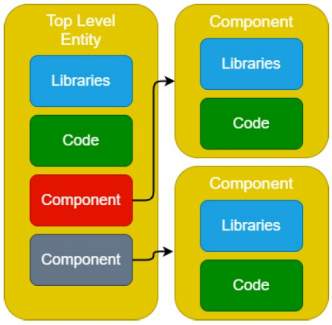


Figura 2.6: Rappresentazione grafica di una Top Level Entity[8]

Una volta terminata la stesura del codice, bisogna specificare le specifiche e limitazioni, i *constraints*, necessarie affinchè il sistema possa esser realizzato sulla scheda.

2.3 Processo di Sviluppo FPGA

Il processo di sviluppo non è riducibile alla scrittura del codice di definizione, come potrebbe avvenire per un Arduino, ma necessità dei passaggi ulteriori dovuti all'architettura, infatti la presenza dell'interconnessione dei blocchi, i blocchi I/O necessariamente configurati e la *Programmable Logic* definita impongono un ulteriore livello di complessità.

L'evoluzione tecnologica ha comportato miglioramenti significativi per quanto riguarda lo sviluppo, infatti ad oggi si parte dalla scrittura del codice[1], passando il tutto ad un sistema di sintesi che effettua la traduzione in una netlist, un file che descrive tutte le interconnessioni logiche da effettuare tra i vari blocchi della scheda.

Per capire meglio il funzionamento dividiamo il processo in tre stadi:

Il primo stadio, che prende il nome di *Technology Mapping*, effettua la generazione di un modello *RTL* (*Register Transfer Level*); questo stadio converte il codice di descrizione in dei blocchi più semplici, ed è importantissimo poichè bisogna garantire che le funzionalità descritte nel codice siano rispettate; esso si verifica tramite delle simulazioni. Alcuni blocchi di logica complessa vengono portati ad un livello composto tra gate logici e registri: questo livello è detto Logic Gate Level.

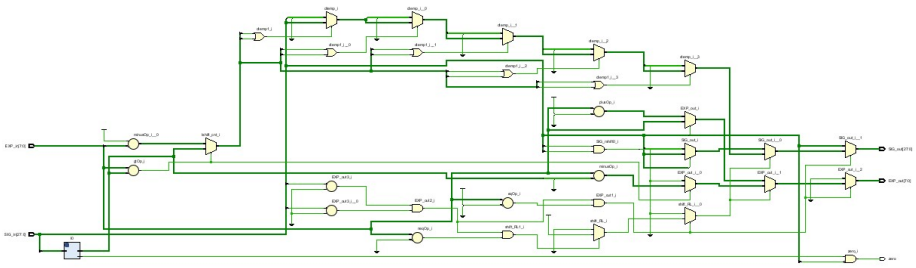


Figura 2.7: Rappresentazione grafica del Tecnology Mapping del FPGA in figura 2.1

[1] O generazione di esso tramite vivado

Tutto questo processo viene ottimizzato come farebbe un compilatore, al fine di occupare meno spazio.

Una volta completato questo passaggio bisogna effettuare la collocazione degli elementi dell'FPGA: il procedimento prende il nome di *Place & Route* (*PnR*), ed è a sua volta composto da tre sotto passaggi.

Il primo è il *Packing*, che analizza le primitive e le organizza in un cluster.

Il secondo è il *Placing*, che prende in ingresso i cluster e decide dove collocare fisicamente i componenti sulla scheda permettendo cosi l'instradamento

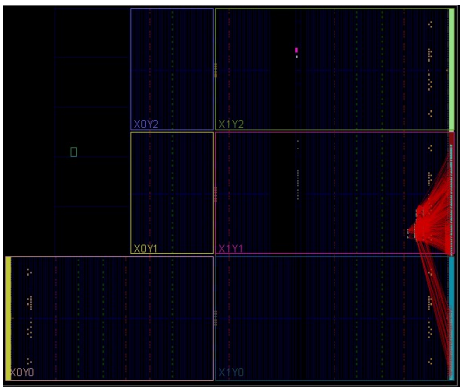


Figura 2.8: Rappresentazione grafica del Placing del FPGA in figura 2.1

Infine il *Routing*, si occuperà di calcolare tutte le possibili connessioni selezionando la migliore, in funzione del ritardo: questo, ovviamente, rende il passaggio oneroso computazionalmente.

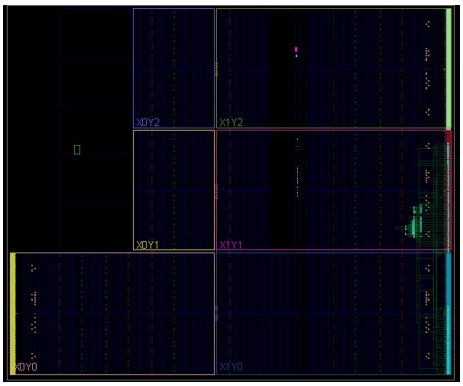


Figura 2.9: Rappresentazione grafica del Routing del FPGA in figura 2.1

Una volta terminati questi passaggi avremo a disposizione l'implementazione del *FPGA*, ma sarà necessario tradurre il tutto in un formato interpretabile al dispositivo; questo processo

verrà ripreso anche in seguito. Questo passaggio di traduzione prende il nome di generazione del bitstream.

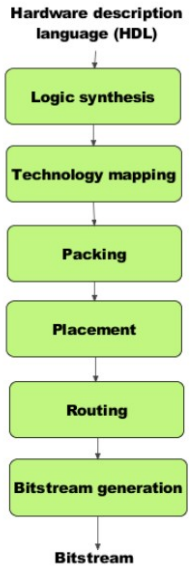


Figura 2.10: Stack per lo sviluppo di un FPGA

Questo tipo di sviluppo è il migliore nel nostro caso, poichè non stiamo effettuando un *High Level Synthesis* (*HLS*), in quel caso esistono dei tools appositi.

2.3.1 Sviluppo tramite Vivado

Lo sviluppo tramite *Vivado* è alquanto semplificato, poichè godendo di un'interfaccia grafica permette l'implementazione di sistemi complessi anche non avendo alcuna conoscenza di *VHDL* o *Verilog*; si rimanda, però, all'appendice A.1 per maggiori chiarimenti su installazione ed inizio del primo progetto.

2.4 Tecnologie implementative

2.4.1 Vitis

Vitis è il *Software Development Kit* (*SDK*) fornito da Xilnix nella sua suite di prodotti la sua installazione è contestuale a quella trattata in A.1, e tramite esso è possibile generare codice in grado di esser eseguito sulla scheda. Questo è possibile per mezzo delle le informazioni

sulla scheda implementata tramite *Vivado*.

Sono presenti due possibilità di programmazione: *Standalone* e *Linux*. La prima applicazione permetterà l'esecuzione di codice C in modalità *Bare-Metal*[2]; in questo modo potremo usare delle librerie della *Xilinx* che contengono delle funzioni standard al fine di effettuare la comunicazione tra PS e PL. Successivamente tratteremo anche la comunicazione tra PS e PL su sistema Linux. Questo tipo di programmazione è discussa approfonditamente B.1

2.4.2 Petalinux

È un tool della Xilnix necessario qualora si volesse usare una distro Linux embedded su di una *FPGA SoC* Xilnix; la sua installazione è trattata in C.1. *Petalinux* offre una grossa flessibilità per la costruzione e compilazione di un kernel Linux, la sua flessibilità ci permette di escludere o includere moduli pre-esistenti, oppure di crearne uno a seconda delle esigenze. Si basa fortemente sull'architettura e sull'esportazione che avviene da *Vivado* al fine di costruire il device tree e gli strumenti necessari al kernel.

Al fine di usufruire di *Petalinux* si rende necessario impostare i Jumpers della scheda *MIO2-6* in configurazione "00110": in questo modo potremo usare il kernel creato e compilato[3], tramite la scheda SD che dovrà rispettare delle specifiche espresse in D.1.

2.4.3 Bootgen

Bootgen è un tool appartenente alla suite di sviluppo della Xilnix, il suo scopo è quello di generare i file binari e tutti gli artefatti necessari. La sua installazione verrà discussa in E.1 Sfortunatamente questo tool è attualmente disponibile solo per architettura x86, per cui come tutti i tool presentati precedentemente, non sarà possibile installarlo sulla piattaforma da noi prescelta.

[2]Livello di programmazione senza astrazioni, come il sistema operativo

[3]Verrà trattato successivamente

2.4.4 OpenStack

OpenStack è un piattaforma Cloud open-source[16] che permette la gestione di risorse in Cloud. Le risorse possono variare: da macchine virtuali a blocchi d'indirizzamento, spazio d'archiviazione ed acceleratori. Tutte le risorse fisiche vengono aggregate in un unico grande pool ed alloca le risorse virtuali, permettendo così la richiesta tramite le API da parte dell'utente finale, *OpenStack* non gestisce la virtualizzazione, ma sfrutta le tecnologie di virtualizzazioni preesistenti, eseguendo un funzionamento assimilabile a quello di un raccoglitore.

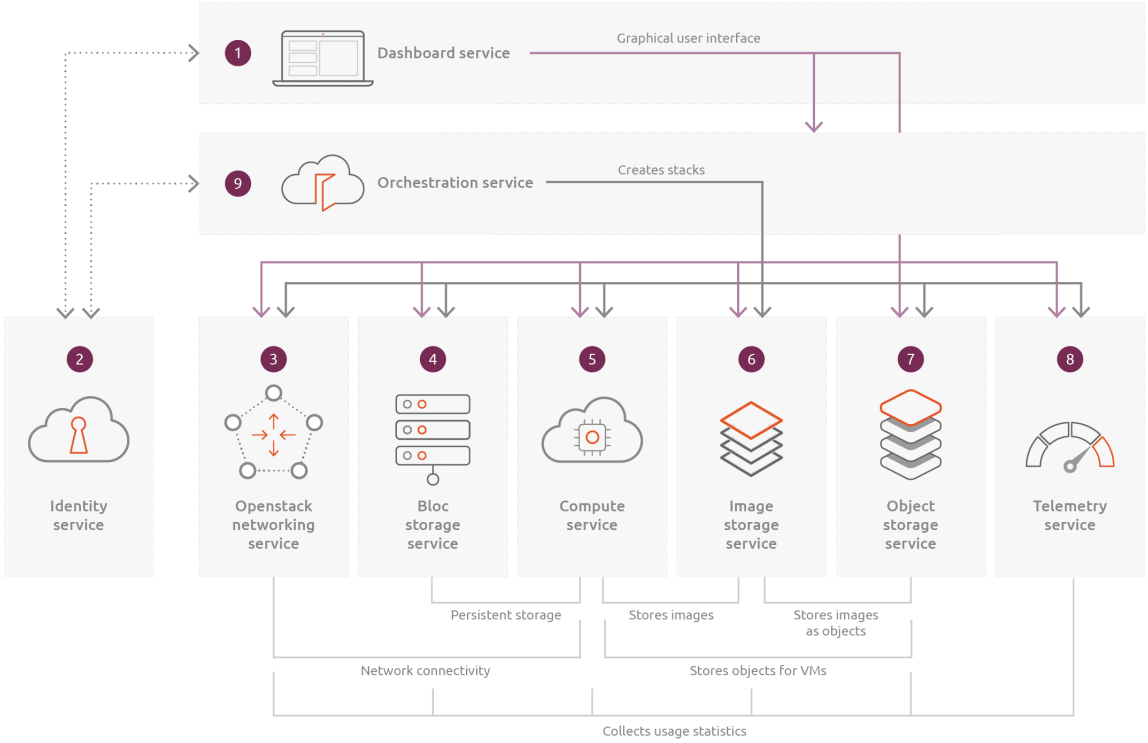


Figura 2.11: Diagramma rappresentativo funzionamento OpenStack[16]

2.4.5 Nova

Nova fa parte del progetto open-source *OpenStack*, e il suo scopo è la gestione delle *Virtual Machine* (*VM*). Esso permette, in simbiosi con Cyborg, di definire gli acceleratori e le FPGA come VM; per maggiori approfondimenti si rimanda a [9].

2.4.6 Cyborg

È un sottosistema di *OpenStack*, il quale si occupa della gestione del ciclo di vita degli acceleratori, il tutto effettuato tramite un'interfaccia *REpresentational State Transfer* (*REST*). Tale approccio permette l'aggiunta e la rimozione di nodi ed informazioni legate ai dispositivi.[4]

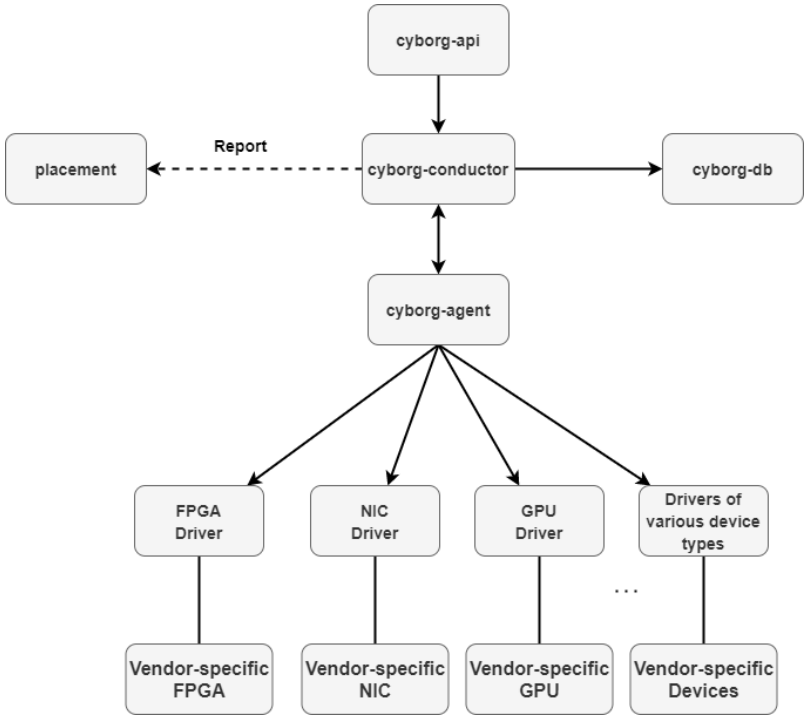


Figura 2.12: Architettura del funzionamento di Cyborg[4]

Tramite i *driver*, *Cyborg* sarà in grado di interfacciarsi con la scheda FPGA ed effettuare le operazioni discusse approfonditamente nel capitolo 3

2.4.7 Blazar

È un servizio fornito da *OpenStack* al fine di effettuare la *Resource Reservation*, ossia la prenotazione delle risorsr disponibili nel sistema Cloud. Le prenotazioni possono esser effettuate per un periodo di tempo specifico, immediato o nel futuro; per risorse intendiamo istanze di VM, risorse di storage e risorse puramente computazionali.[3]

2.4.8 Iotronic - Lightning-Rod

Essi sono due servizi che tramite le *REST API*, permettono la gestione di device *IoT* denominati all'interno del framework come "board"; la struttura dei servizi è conforme a quella dei servizi di *OpenStack*, quindi integrabile nel sistema.
L'architettura del sistema si concentra sulla comunicazione tra gli user ed i nodi *IoT*. I servizi sono due, e si differenziano dipendentemente dal lato dell'infrastruttura che operano; *IoTronic* lavora lato Cloud, mentre *Lighting-Rod* lato *IoT*; la combinazione dei due permette di realizzare varie azioni volte alla gestione dei dispositivi IoT,

- La gestione delle Board, come l'invio dei comandi;
- Plugin;
- Servizi;
- Virtual Network;
- Web Service;

Lightning-rod, come detto, rappresenta il punto di contatto tra il Sistema Operativo, presente sulla Processing System dell'FPGA SoC, ed il Cloud. Permettendo all'utente di interfacciarsi attraverso una comunicazione assicurata tramite WAMP.

Capitolo 3

Soluzione Proposta

L'idea del progetto nasce dalla volontà di fornire un'integrazione degli acceleratori, in un ambiente cloud tramite l'architettura OpenStack. L'intenzione è quella di avere presenti in un'interfaccia web la possibilità di prenotare risorse cloud, tra le quali sono presenti le FPGA, le risorse saranno usabili sia per scopi didattici sia per ricerca.

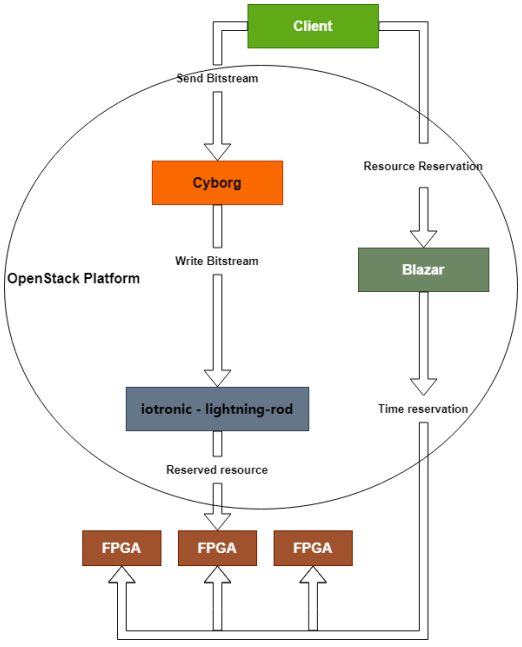


Figura 3.1: Stack dell'architettura

Un esempio applicativo potrebbe essere un laboratorio virtuale, dove ogni utente finale, professori, studenti o utenti privati, hanno accesso ad un'infrastruttura che presenta molte risorse cloud tra cui le FPGA, sulle quali è possibile effettuare il caricamento del proprio bitstream precedentemente prodotto.
Quest'architettura potrebbe esser vista come un sistema di prenotazione delle risorse per il Cloud Computing, portando ad un incremento di performance rispetto alle strutture classiche grazie all'uso delle FPGA. Tramite la loro riprogrammazione si garantisce una maggior flessibilità, una maggior potenza computazionale e soprattutto la divisione in regioni, che permette la possibilità di ospitare più acceleratori in un singolo device in modo da avere più risorse virtuali disponibili nel cloud, ma con lo stesso numero di board.
Le varie fasi del processo verranno effettuate da diversi servizi presenti nella piattaforma *OpenStack*.
Il servizio che orchestra le risorse sarà *Blazar*, come spiegato nella sotto sezione 2.4.7, il quale permette la *Resource Reservation* e tramite un altro servizio che effettua il WebServer sarà possibile invocare le API che permettono la reservation.

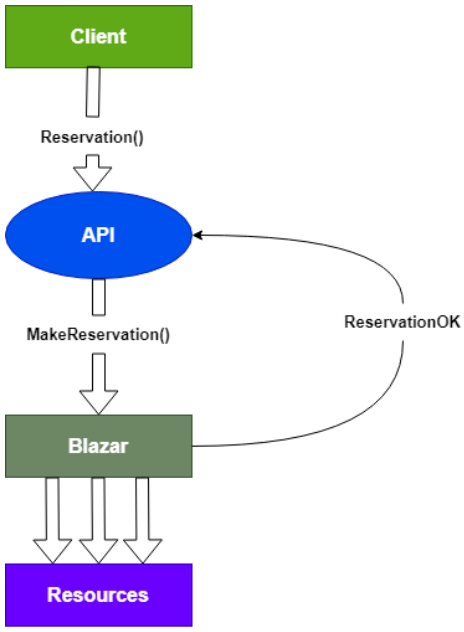


Figura 3.2: Diagramma che rappresenta le fasi di reservation

Una volta avvenuta l'effettiva prenotazione, l'utente finale utilizza il servizio *Cyborg*, che gestisce il ciclo di vita degli acceleratori e tramite la sua integrazione con *Nova*, garantisce

il passthrough degli host accelleratori dall'host alle virtual machine rendendo disponibile la gestione da parte dell'utente dei nodi da quest'ultimo riservati. Al fine di creare il punto di contatto tra il cloud e i dispositivi IoT verranno usati in simbiosi IoTronic e Lightning-rod, integrabili nella piattaforma OpenStack e quindi interfacciabili tramite le API agli altri servizi già presenti nel sistema. Per come sono stati definiti IoTronic sarà l'agente che verrà eseguito lato cloud, mentre Lightning-rod sarà l'agente che verrà eseguito lato FPGA SoC. IoTronic e Lightning-rod saranno connessi tra di loro tramite una connessione *WAMP* in modo da garantire il trasferimento del file binario e l'esecuzione di tutti gli script necessari alla riprogrammazione.

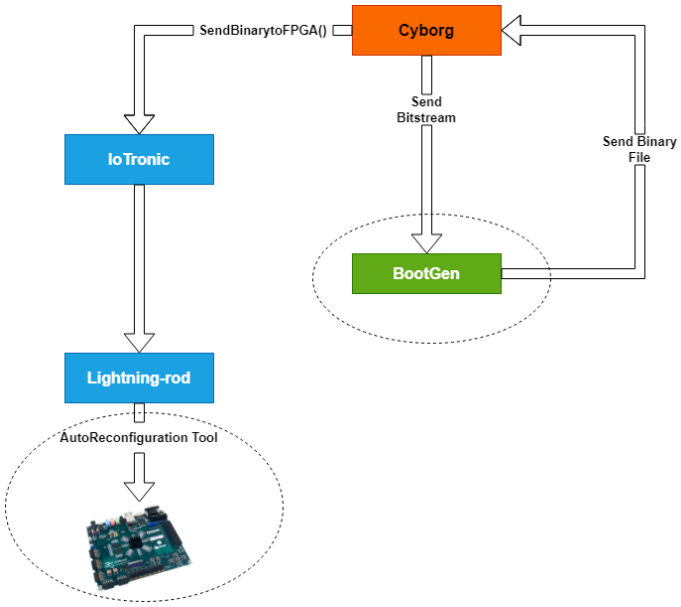


Figura 3.3: Diagramma che rappresenta le fasi post upload del bitstream

La scelta di fornire il bitstream e quindi aggiungere un livello di complessità in più, permette di effettuare un'analisi più accurata della validità del bitstream per l'architettura prenotata tramite la sua decodifica. Il bitstream contiene informazioni riguardo la data di creazione, l'architettura e la dimensione. Questo controllo permette di limitare gli errori che un utente novizio può commettere riservando delle risorse non compatibili alla sua architettura.
Tutto il sistema è gestito dalle API di OpenStack, il quale garantisce questa integrazione tra servizi diversi tra di loro, rendendo il tutto trasparente all'utente finale.
In questa tesi verrà trattato il processo di automatizzazione per la generazione del file binario e per la riprogrammazione automatica tramite Lightning-rod.

Capitolo 4

Interfacciamento IoT-Cloud con FPGA

Il primo passaggio per l'integrazione delle FPGA SoC nel sistema IoT-Cloud è permettere l'interfacciamento con la rete tramite una distro Linux embedded. La distro compatibile con le FPGA SoC della Xilnix è *Petalinux*, che verrà montata sulla parte a microprocessore della board al fine di permettere l'esecuzione di *Lightning-rod* che effettuerà l'interfacciamento con il sistema Cloud. In tal modo sarà garantito l'uso della risorsa nell'architettura precedentemente descritta.

4.1 WorkFlow di Petalinux

Come già visto nell'appendice C.1.3 *Petalinux* è un tool della *Xilinx* che ci permette di creare un kernel linux per sistemi FPGA SoC.
Petalinux tramite l'esportazione del hardware da vivado[1] ed un Board Support Package (BSP)[2] ci da la possibilità di creare il kernel per la scheda di riferimento.
Petalinux oltre a definire il kernel ed il comportamento degli stati di boot, ci permette di costruire il file system, in maniera puramente custom, tramite l'uso di Yocto, esso è

[1] A.1.4

[2] Esso rappresenta un codice di supporto di un implementazione per una determinata scheda

un progetto open-source che permette alla community indipendentemente dall'architettura hardware.

Una volta effettuati i passaggi descritti nel C.1.4, è possibile effettuare tramite console linux l'effettiva creazione del kernel.

4.1.1 Creazione di un progetto

Al fine di creare un progetto è possibile percorrere due strade, usare un template[3] o l'uso di un BSP[4].

```
1 petalinux-create -t project -s <PATH-TO-BSP>
```

Codice 4.1: Comando creazione progetto con BSP

```
1 petalinux-create --type project --template <PLATFORM> --name
    <PROJECT_NAME>
```

Codice 4.2: Comando creazione progetto con il template

```
1 INFO: Create project:
2 INFO: Projects:
3 INFO: * xilinx-zcu102-v<petalinux-version>
4 INFO: has been successfully installed to /home/user/
5 INFO: New project successfully created in /home/user/
```

Codice 4.3: Output atteso

Entrambe le scelte sono intercambiabili, poichè ci daranno solamente lo scheletro sulla quale poi modellare tutto il sistema.

4.1.2 Configurazione del sistema

Dopo la creazione dello scheletro è però necessario configurare il progetto in base alle esigenze progettuali, quindi al fine di far ciò sarà necessario muoversi nella cartella di progetto e

[3] Nel caso di zedboard bisogna usare zynq

[4] Xilinx

definire l'hardware del progetto tramite l'hardware definition file[5], esso si può accumunare ad un archivio al cui interno ci sono tutte le informazioni necessarie per costruire la piattaforma per la struttura che è stata progetta antecedentemente[6]. Al suo interno troveremo le librerie PS7[7], Processing System 7000, come la nostra architettura, essi sono strettamente legali al processo di boot lato PS[8], nel dettaglio i file PS7_init contengono tutte le configurazioni per il clock, la memoria ed il GPIO, inoltre essi vengono usati per creare il First Stage BootLoader (FSBL), il qui compito è quello di puntare al Second Stage BootLoader nella partizione di boot.

```
1 petalinux-config --get-hw-description <PATH-TO-XSA
    Directory>/<PATH-TO-XSA>
```

Codice 4.4: Comando necessario alla definizione dell'hardware

Questo passaggio risulta essere critico ai fini del proseguio corretto dello sviluppo, poichè un errata esportazione del file XSA o l'incompatibilità del sistema operativo con il tool potrebbero causare errori.

Nel caso in cui questo passaggio sia andato a buon fine, si presenterà la seguente schermata Ai fini della tesi sarà necessario abilitare il parametro FPGA Manager, che trattato nel dettaglio al capitolo 6.

Dopo aver configurato l'hardware e quindi i parametri del boot loader[9], è necessario riconfigurare il file system ed eventualmente il kernel.

```
1 petalinux-config -c rootfs
```

Codice 4.5: Comando necessario alla riconfigurazione del file system

Se il comando è andato a buon fine comparirà la seguente schermata Da qui sarà possibile attivare o disattivare packages, al fine della programmazione on-board e per un ottimale interfacciamento con il sistema cloud è fortemente consigliato abilitare i seguenti package:

[5] Xilinx Support Archive, XSA

[6] Per la progettazione si rimanda a A.1.1, mentre per l'export del file XSA a A.1.4

[7] Cambiano per ogni architettura

[8] Processing System

[9] U-BOOT

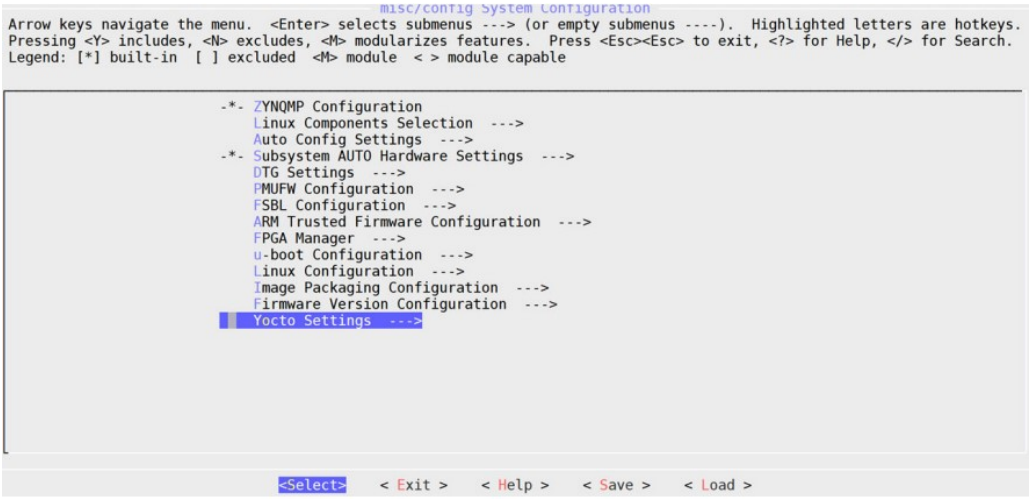


Figura 4.1: Schermata di configurazione del sistema

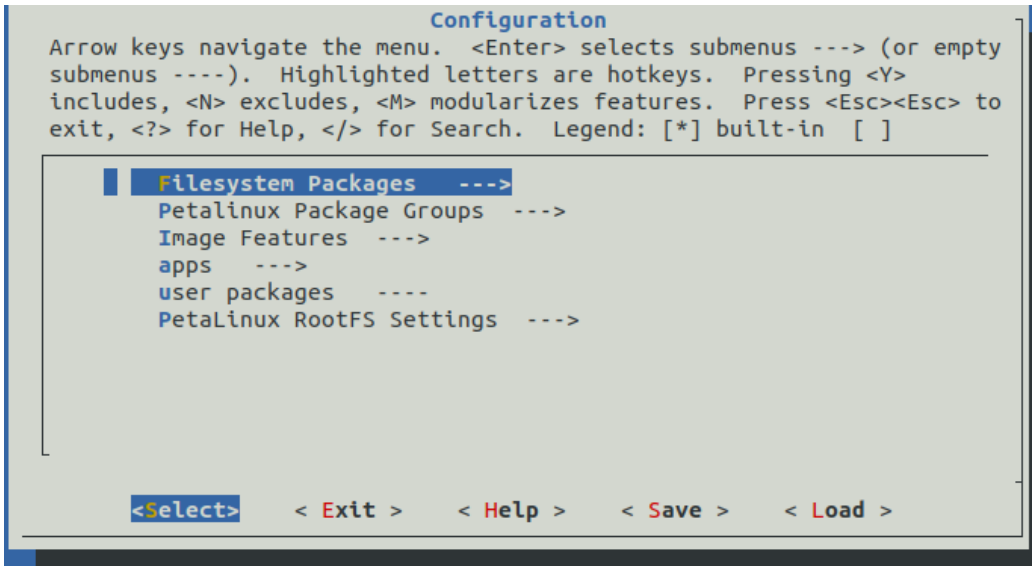


Figura 4.2: Schermata di modifica file system

- *Petalinux Package Groups -> packagegroup-petalinux-utils*, al fine di avere GCC per l'architettura ARM Cortex A9.
- *Petalinux Package Groups -> packagegroup-petalinux-python* e *Filesystem Packages-> devel-> Python*, al fine di avere l'interprete di Python.
- *console-> utils -> git*, al fine di avere il comando git.

Terminato ciò si potrà passare alla modifica del kernel, questa modifica risulta necessaria

per poter effettuare le attività di tesi.

```
1 petalinux-config -c kernel
```

Codice 4.6: Comando necessario alla riconfigurazione del kernel

Dalla schermata, che sarà analoga a quelle precedenti, è necessario abilitare i seguenti moduli[10]:

- *Device Drivers —> FPGA Configuration Framework -> [*]FPGA Region*, l'FPGA Region verrà discussa nel dettaglio nella sezione 6.3
- *Device Drivers —> FPGA Configuration Framework -> [*]FPGA Bridge Framework*, il Bridge è un gate bus che permette la connessione tra Host ed FPGA. Esso dovrà esser disabilitato durante la *Full Reconfiguration*, al fine di evitare possibili segnali malevoli. Mentre per la *Partial Reconfiguration* sarà ininfluente poichè le regioni non interessate non verranno modificate.
- *Device Drivers –> Device Tree and Open Firmware support -> [*] Device Tree Overlays*, il suo scopo è modificare il device tree del kernel mentre è in esecuzione, una sua modifica si ripercuote sul lato FPGA, causandone una modifica.
- *Device Drivers –> Device Tree and Open Firmware support -> [*] Device Tree Overlays ConfigFS Interface*
- *Memory Management options —> [*] Contiguous Memory Allocator*, il Contiguous Memoy Allocator verrà discusso nel dettaglio nel capitolo 5
- *Library routines–> DMA Contiguous Memory Allocator*

4.2 Creazione app e modulo kernel

Creare un app a livello kernel può risultare comodo, al fine di creare l'applicazione ed innestarla al nostro file system sarà necessario eseguire il seguente comando:

[10]Per "[*]" si intende selezionato e si fa posizionandosi sopra il modulo desiderato e premendo la barra spaziatrice

```
1 petalinux-create -t apps --name [nome desiderato] --template [c,
    c++, autoconf, install]
```

Codice 4.7: Comando necessario alla creazione dell'applicazione, va eseguito nella cartella del progetto

Se il comando è stato eseguito correttamente il codice per l'applicazione si troverà nella cartella */project-spec/meta-user/recipes-apps/Nome Desiderato*.

4.2.1 Compilazione

Prima di poter compilare l'applicazione sarà necessario aggiungerla al file system, eseguendo i comandi visti nel codice 4.5, spostandoci nel menu *apps -> [nome desiderato]* sarà possibile selezionarlo.
Effettuato ciò per compilare è necessario eseguire:

```
1 petalinux-build -c [nome desiderato]
```

Codice 4.8: Comando necessario alla compilazione dell'applicazione

4.2.2 Modulo kernel

Ai fini della creazione del modulo per il kernle, sarà necessario eseguire alcuni comandi:

```
1 petalinux-create -t modules --name [nome desiderato] --enable
```

Codice 4.9: Comando necessario alla creazione del modulo kernel

Analogalmente all'applicazione il modulo si troverà nella cartella */project-spec/meta-user/recipes-modules/[nome desiderato]*, esso sarà compilato una volta compilato il kernel.

4.3 Compilazione kernel

Una volta ultimata la progettazione e la configurazione del kernel e del file system è necessaria la compilazione al fine di generare il codice binario, in ambiente *Petalinux* sarà necessario usare il comando

```
1 petalinux-build
```

Codice 4.10: Comando necessario alla compilazione del kernel

Questo passaggio sarà onesoro temporalmente, poichè si dovranno linkare tutte le librerie ed eventualmente scaricare moduli aggiunti in fase di configurazione.

4.3.1 Pacchettizzazione del kernel

Una volta che il kernel è compilato nella cartella *images/linux* si troveranno tutte le componenti del sistema operativo. Quindi eseguendo il seguente comando

```
1 petalinux-package -boot -format BIN -fsbl image/linux/[FSBL] -fpga
    <path/to/bitstream> -u-boot
```

Codice 4.11: Comando necessario alla pacchettizzazione del kernel

Capitolo 5

Interfaccia Soft Core - Gate Array

Nell'ambiente delle FPGA SoC è importante garantire l'alta performace e la bassa latenza. *Processing System* e *Programmable Logic* comunicano tra di loro al fine di permettere il corretto funzionamento della scheda alle sue massime funzionalità.
Tuttavia il metodo di interfacciamento tra le due aree non è univoco. In questo capitolo verrà trattato il metodo più comune e diffuso tra quelli disponibili in tutte le FPGA SoC.

5.1 Interfacce di comunicazione

Le ZYNQ-7000 usano diversi metodi di comunicazione tra PS e PL, tutte basate su tecniche di interconnessione, le interfacce possono essere suddivise in due categorie[13]:

- Functional Interface, le quali rientrano l'AXI, i controller DMA e la Extended MIO.
- Configuration Signals, contenente il Processor Configuration Access Port che verrà approfondito nella sezione 6.3.

Per la comunicazione tra PS e PL è preferibile usare il protocollo AXI, per via della sua diffusione, compatibilità e velocità.

5.2 Advanced Microcontroller Bus Architecture, AMBA

Prima di parlare del protocollo AXI, è necessario esporre il protocollo AMBA, esso è un open-standard per la comunicazione nei system-on-chip SoC. AMBA definisce come i blocchi funzionali comunicano tra di loro, definendo il tutto nel Device Tree,[1]

```
1   amba: amba {
2           compatible = "simple-bus";
3           #address-cells = <2>;
4           #size-cells = <2>;
5           ranges;
6           can0: can@ff060000 {
7                       compatible = "xlnx,zynq-can-1.0";
8                       clock-names = "can_clk", "pclk";
9                       reg =<0x0 0xff060000 0x0 0x1000>;
10          };
11  };
```

Codice 5.1: Questo è un esempio di un device tree che definisce la comunicazione di un core propretario tramite il protocollo CANBus.

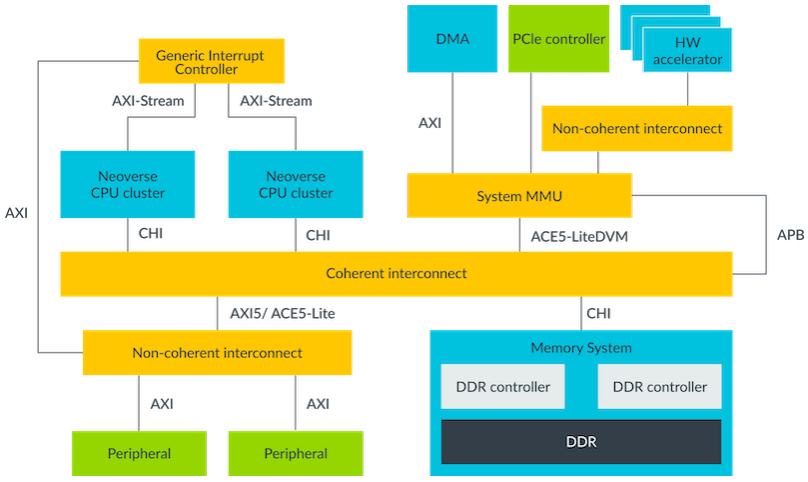


Figura 5.1: Diagramma raffigurante SoC, tutti i collegamenti saranno decisi dal protocollo AMBA, si può notare come figura il protocollo AXI[1]

5.2.1 Perchè e dov'è usato AMBA

Il protocollo AMBA è usato per semplificare lo sviluppo di schede contenenti più processori e un grande numero di periferiche e controllori.
È diventato uno standard data la sua flessibilità, compatibilità, larghezza di banda e latenza.

5.3 Advanced eXtensible Interface

È un protocollo di interfaccia sviluppato sulle architetture ARM e parte del protocollo AMBA. L'AXI è un protocollo master-slave, usato per connessioni con periferiche e memorie,

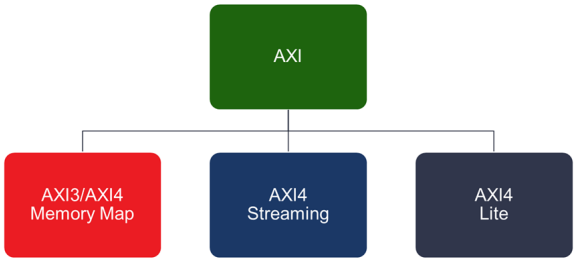


Figura 5.2: I 3 tipi d'interfaccia AXI

essendo presenti più master in base alle priorità si determina quale master potrà usare il bus, mentre un decodere eseguirà la selezione dello slave.
Le operazioni sono eseguite in burst, solitamente da 256bit, che può richiedere più cicli di clock per esser completata, ogni burst consiste in due fasi, la prima di indirizzamento e la seconda di scambio dati, fornendo elevate prestazioni per accessi di tipo memory mapped.
È composto da 5 canali indipendenti tra master e slave[2] :

- Write Address
- Write Data
- Read Address

Quelli direzione Master to Slave, mentre Slave to Master:

- Write Response

- Read Data

Figura 5.3: Esempio di comunicazione tramite protocollo AXI, i blocchi multipli stanno ad identificare i Burst

I dati possono viaggiare simultaneamente in entrambe le direzioni, esso è possibile per via delle due connessioni separate, in indirizzi e dati, sia per lettura che per scrittura. Ogni scambio di dati è detto transazione essa include l'indirizzo, le informazioni di controllo, i dati inviati e qualsiasi informazione di risposta.

5.3.1 AXI nei sistemi ZYNQ

Nell'architettura ZYNQ-7000, ma come tutte le architetture ZYNQ, il protocollo AXI permette l'interfacciamento tra Processing System (PS) e Programmable Logic (PL).

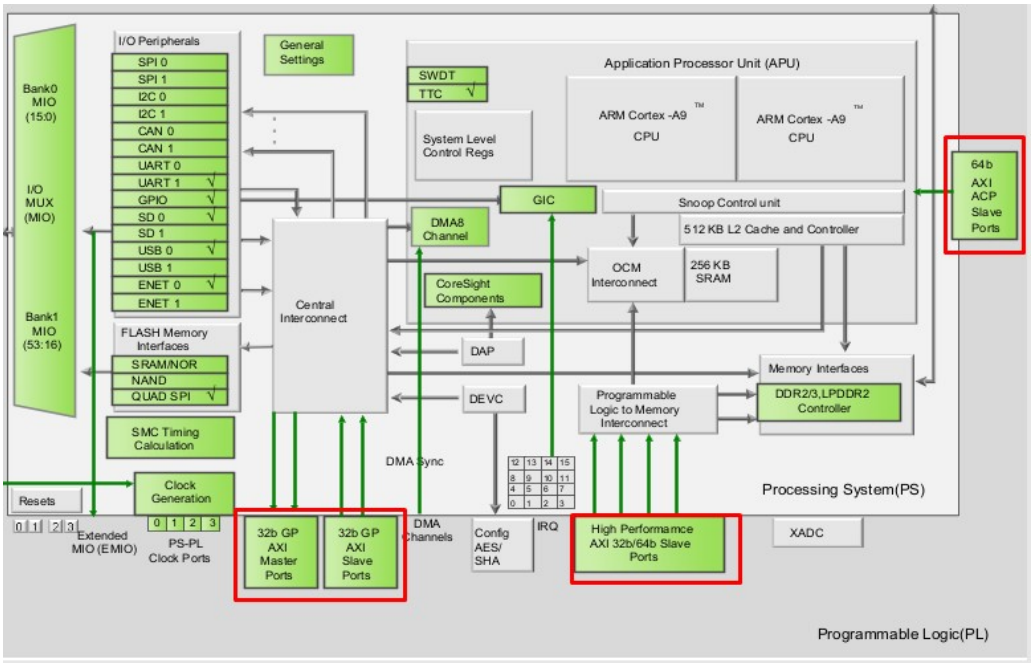


Figura 5.4: Design di un architettura ZYNQ-7000, dove sono evidenziati i blocchi AXI

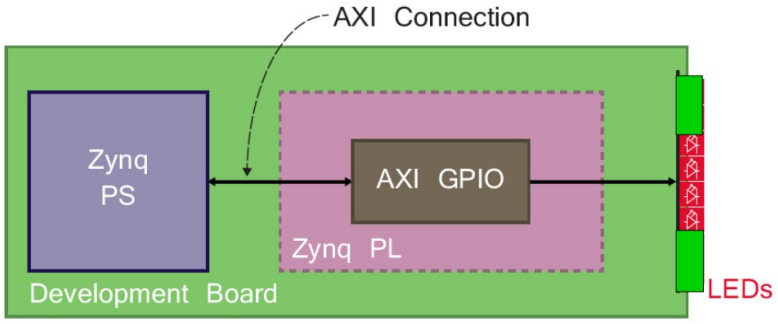


Figura 5.5: Zoom della connessione tra PS e PL

In quest'architettura abbiamo delle interfacce di comunicazione sul bus AXI. Al fine di facilitare lo sviluppo della scheda la xilnix ha prodotto un componente, completamente riprogrammabile, detto AXI Interconnect, esso facilità e gestisce tutte le connessioni tra master e slave.

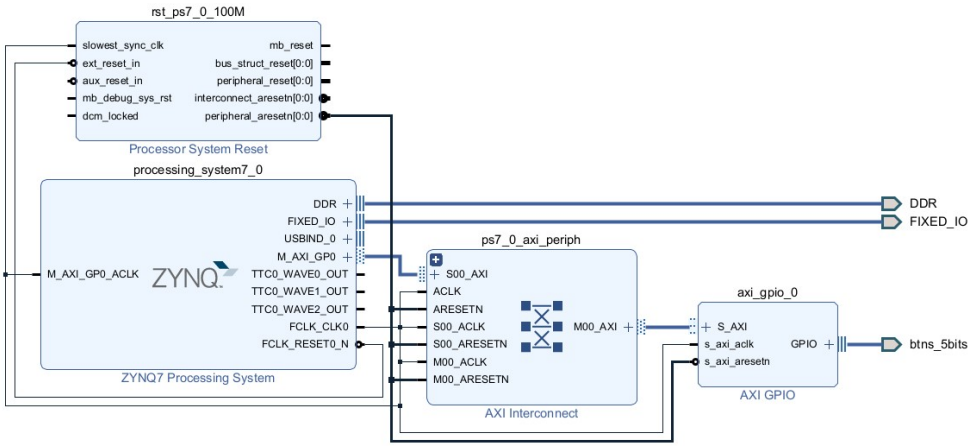


Figura 5.6: Schema contentente l'interconnecter

5.3.2 Comunicazione memory-mapped tra PS e PL

Al fine di far comunicare tramite gli indirizzi definiti in fase di progettazione dobbiamo usare il Central Direct Memory Access (CDMA), esso è un core prodotto da Xilnix che ci permette di avere una connessione tra una sorgente con un indirizzo memory-mapped ed una destinazione con un indirizzo memory-mapped, tramite il protocollo AXI. Quindi è in grado di associare un indirizzo mappato in memoria fisica con un indirizzo mappato in memoria logica, evitando cosi il vincolo imposto dal sistema operativo sull'accesso agli indirizzi fisici tramite la funzione mmap.

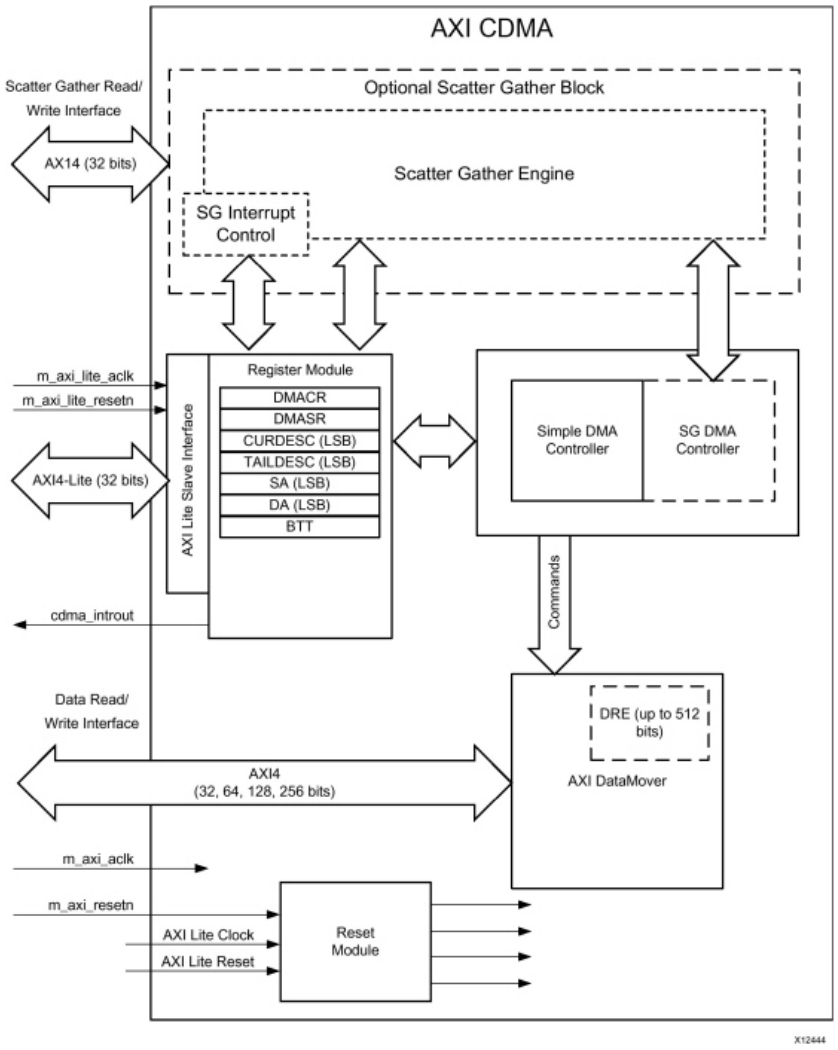


Figura 5.7: Schema del CDMA

Nel caso di un FPGA SoC con un Real Time OS (RTOS), può esser usata per interfacciarsi con il GPIO lato FPGA, con dei core di calcolo tensoriale o per la riprogrammazione sia completa che parziale della PL.

5.4 Driver AXI

Andremo a programmare, leggere e manipolare i dati presenti nella Programmable Logic ed interconnessi con la PS tramite il protocollo AXI e sfruttando il core CDMA ed il funzionamento di esso

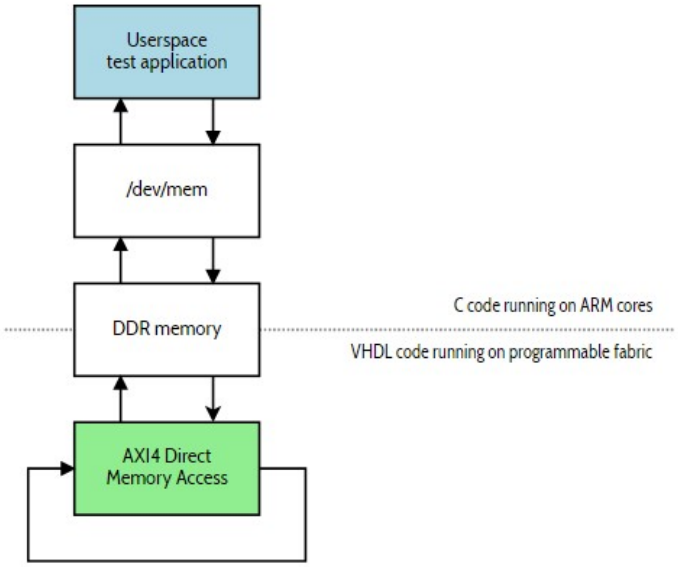


Figura 5.8: Funzionamento minimo del CDMA

Il driver necessita di conoscere gli indirizzi base dei core interconnessi al protocollo AXI e la loro dimensione ed eventuale offset.

```
1    #define GPIO_BASE_ADDRESS       0x42000000
2    #define GPIO_MAP_SIZE           0x1000
3    #define XAXIGPIO_DATA_OFFSET    0x00000000
```

Codice 5.2: Definizione costanti

Poichè necessitiamo di usare */dev/mem/*, file che rappresenta l'immagine della memoria centrale del microprocessore, per effettuare l'associazione tra indirizzo fisico noto ed indirizzo logico, dobbiamo andare ad effettuare un'apertura assicurandoci che sia andato a buon fine

```
1    int memfd;
2    if ( (memfd = open("/dev/mem", O_RDWR | O_DSYNC)) == -1 ){
3        printf("Can't␣open␣/dev/mem.\n");
4        exit(-1);
5    }
```

Codice 5.3: Apertura file mem

Una volta effettuato ciò potremmo andare accedere, tramite la funzione mmap all'indirizzo del core AXI con la quale vogliamo interfacciarci, quest'apertura dovra esser sia in lettura che

scrittura e map shared, quindi visibile anche a tutti i processi mappati nella stessa regione, quindi usiamo

```
1     void *gpioMapAddr = NULL;
2     gpioMapAddr = mmap(0, GPIO_MAP_SIZE, PROT_READ | PROT_WRITE,
          MAP_SHARED, memfd, GPIO_BASE_ADDRESS);
3     if (gpioMapAddr == (void *) -1){
4         printf("Can't map the CDMA-Control-Port to user space.\n");
5         exit(-1);
6     }
```

Codice 5.4: Mapping dell'indirizzo di GPIO

Se il tutto ha avuto esito positivo possiamo effettuare la manipolazione dei dati presenti in esso, andando a sommare l'eventuale offset e l'indirizzo fornitoci dalla funzione mmap

```
1     unsigned long ii;
2     for(ii=0; ii<256; ii++){
3         *((volatile unsigned long*) (gpioMapAddr +
              XAXIGPIO_DATA_OFFSET)) = ii;
4         printf("led = 0x%02x\n", ii);
5         usleep(20000); // sleep 20ms
6     }
```

Codice 5.5: Esempio con un contatore tramite i led

Questo esempio può esser modificato ad hoc al fine di accedere ai core interconnessi con il protocollo AXI, questo da la base alla partial reconfiguration tramite due controller già presenti nei SoC. Questo codice può esser compilato sia on-board, sia tramite la cross compilazione, per la cross compilazione si rimanda a F.1

Capitolo 6

Gate Array Reconfiguration

6.1 Full Reconfiguration

La riprogrammabilità delle FPGA è la chiave per la loro adozione nei sistemi Cloud. In tal modo l'FPGA risulterà essere un sistema versatile, scalabile, efficiente ed ottimo per il cloud computing. Per la riprogrammazione delle FPGA SoC è possibile eseguire delle procedure automatiche, e non, per la riprogrammazione completa della Programmable Logic.

6.1.1 FPGA Manager

Esso bisogna che sia abilitato nella struttura del kernel, poichè è un insieme di API agnostiche che la loro chiamata ci permette di programmare, o riprogrammare, l'FPGA tramite un file binario.

6.1.2 Creazione del file binario

Per la creazione del file binario, si necessita un bitstream la quale generazione è discussa nell'appendice A.1.4.
Questa procedura necessita del tool bootgen, si rimanda a E.1 per l'installazione, il quale al fine di generare il file binario necessità uno file intermedio il Boot Image Format (BIF), questo file è cosi strutturato:

```
1 all:{./path/bitstream.bit}
```

Codice 6.1: template file .bif

Il file solitamente contiene tutte le fasi di boot ed eventuali partizioni dell'immagine.
Al termine di ciò sarà necessario caricare l'environment di lavoro di bootgen, semplicemente spostandosi nella cartella d'installazione ed eseguendo il seguente:

```
1 source  settings64.sh
```

Codice 6.2: setup environment bootgen

In questo modo tramite il comando

```
1 bootgen -image [/path/to/bifFile] -arch zynq -process_bitstream bin
```

Codice 6.3: setup environment bootgen

All' appendice H.1 è disponibile uno script in grado di automatizzare questo processo.

6.2 Invocazione classe FPGA Manager

Dopo la generazione del file binario, esso andrà copiato nella scheda SD in una qualsiasi partizione.
Al fine di abilitare la riprogrammazione completa della scheda sarà sfruttata la classe FPGA Manager, essa per l'abilitazione necessità alcuni comandi da eseguire all'interno della scheda[12]:

```
1 echo 0 > /sys/class/fpga_manager/fpga0/flags
```

Codice 6.4: Abilitazione FPGA_manager

Questo comando imposta le Flags dell'fpga manager per accettare un nuovo bistream, quindi di fatto abilitando la riprogrammazione.
Di seguito bisognerà caricare il bitstream nella Programmable Logic

```
1 mkdir -p /lib/firmware
2 cp [NomeBIN.bin] /lib/firmware/
```

Codice 6.5: Caricamento bitstream nella Programmable Logic

Infine per effettuare la completa riprogrammazione è necessario puntare il nuovo file binario caricato nella PL, per far ciò è necessario comunicarlo al FPGA_Manager

```
1 echo [NomeBIN.bin] > /sys/class/fpga_manager/fpga0/firmware
```

Codice 6.6: Comunicazione nuovo bitstream alla PL

La verifica dell'avvenuta riprogrammazione è visibile tramite un led on-board o tramite l'esecuzione di un sorgente che sfrutta il driver GPIO precedentemente discusso al capitolo 5.

All'appendice I.1 è possibile trovare uno script automatico per la riprogrammazione.

6.3 Partial Reconfiguration

Le FPGA, oltre la loro riprogrammazione, si rendono ancora più versatili tramite l'uso delle regioni. Quest'ultime rappresentano una parte di FPGA che può esser riprogrammata senza intaccare il resto della board, permettendo di avere molteplici acceleratori.

Le regioni definite nel seguente modo

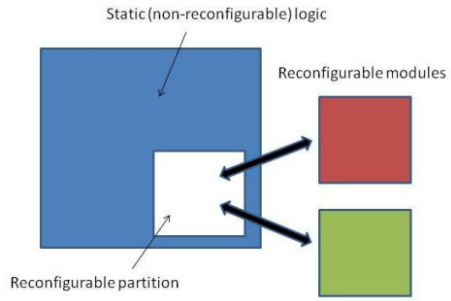


Figura 6.1: Definizione regioni FPGA

Non tutte le regioni sono riprogrammabili, ma solo alcune. Queste vengono dette *Partially Reconfigurable Region* (PRR). Questo modo di definire le FPGA rende ancora più conveniente la loro integrazione in un sistema IoT-Cloud.

6.3.1 Place and Route per il Partial Bitstream

La creazione di un Partial Bitstream, parte prima dalla progettazione dell'hardware, dove dovremmo descrivere, tramite Verilog o VHDL, un Intellectual Property(IP)[17], un soft

core che sia una blackbox, tramite essa sarà possibile effettuare il *floorplanning*, esso è un tentativo di rappresentazione di come saranno piazzati i blocchi funzionali.

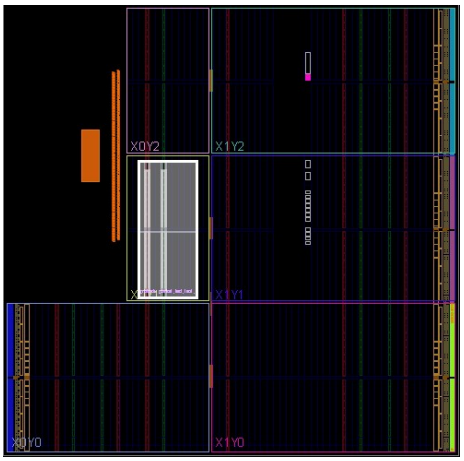


Figura 6.2: FloorPlanning in un FPGA Xilinx

Effettuato ciò potremmo effettuare il *Place and Route*, generando cosi l'implementazione della scheda al fine di generare il *Bitstream*[14].

6.3.2 Interfaccia per la riconfigurazione parziale

Le FPGA SoC permettono la programmazione della Programmable Logic tramite il Processing System, esso si interfaccerà tramite l'interfaccia *Device Configuration interface*(DevC), essa possiede un'interfaccia DMA che tramite il bus AXI trasferirà il partial bitstream ad una delle due *Configuration Access Port*, rispettivamente *Processor Configuration Access Port*(PCAP) e *Internal Configuration Access Port* (ICAP).

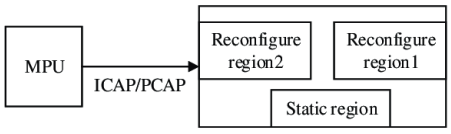


Figura 6.3: Esempio di interfacciamento tra PS e PL per la riprogrammazione[15]

La loro coesistenza non è permessa dall'architettura, è possibile lo scambio tra le due interfacce anche runtime, tramite la modifica di un registro interno alla Processing System.
La *Xilinx* garantisce un IP core *AXI_HWICAP*, che abilita la *Partial Reconfiguration* tramite l'uso di *ICAP*. Il processo di riprogrammazione parziale è stato svolto tramite l'uso di un controller chiamato *ZyCap*[14]

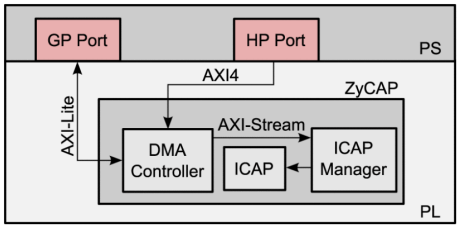


Figura 6.4: Interfaccia tra PS e PL con il controller

6.4 Problemi e possibili soluzioni

Sfortunatamente l'architettura scelta al fine della realizzazione della tesi non supportava nativamente la *Partial Reconfiguration*, rendendo impossibile la realizzazione di quest'ultima, anche causa un'obsoleta letteratura scientifica.
Il principale problema è dovuto dalla non completa integrazione di vivado con i meccanismi di sviluppo dei partial bitstream, poichè risulta impossibile esportare la definizione del hardware contenente tutte le informazioni inerenti al FPGA appena progettata, qualora si riuscisse ad esportare il file XSA sarà necessario creare il progetto su Vitis, ma le librerie usate in [14] sono buona parte deprecate, rendendo inutile il processo di progettazione del codice per la riprogrammazione.

6.4.1 Possibili soluzioni

Alcuni possibili soluzioni potrebbero essere il cambio di architettura come ad esempio il passaggio a *ZynqMP*, ma anche la creazione di un nuovo controller sulla base del *Zycap*, quindi usando un *AXI DMA* ed un controller per gli *ICAP*. Ovviamente la creazione di un nuovo controller comporta la riscrittura del driver di riprogrammazione, che per alcuni tratti può esser basato su quello usato per il controllo del bus AXI, il driver dovrà necessariamente interfacciarsi con l'interfaccia *DevC*, la quale però viene astratta tramite una libreria già fornita dalla *Xilinx*.
Il metodo appena descritto può esser usato per effettuare una riprogrammazione *BareMetal*, quindi senza la presenza di un sistema operativo lato PS, questo comporta l'introduzione di un elemento in più, assimilabile ad un Raspberry, che ospiterà l'agente *Lightning-rod* per la connessione al Cloud.

Capitolo 7

Conclusioni

L'obiettivo di questo elaborato era quello di creare degli strumenti che permettessero l'interfacciamento al *Cloud* e la riprogrammazione completa e parziale dell'*FPGA*.
Lungo la trattazione è stato dimostrato come l'integrazione tra il sistema *Cloud* e le *FPGA* sia possibile e come la loro completa riprogrammazione sia effettuabile ed automatizzabile.

Purtroppo non è stato possibile raggiungere quest'obiettivo.
L'implementazione di un meccanismo per la riconfigurazione parziale non ha riscontrato i risultati attesi, infatti il device scelto per la creazione dell'architettura non supporta la riprogrammazione parziale, In seguito ad un'analisi approfondita del sistema è stato possibile però trovare eventuali soluzioni valide.

L'architettura presentata potrebbe espandersi.
Per questo motivo questo elaborato fornisce anche le conoscenze base per permettere ad un eventuale sviluppatore di continuare il lavoro svolto.

Una possibile soluzione per una prossima implementazione potrebbe essere la creazione di una libreria d'interfacciamento tra *Processing System* e *Programmable Logic*, al fine di permettere la riconfigurazione parziale della scheda.
Un altro esempio di implementazione potrebbe essere la creazione di core proprietari che permettano la definizione delle regioni nella fase di *Floorplanning*.

Appendice A

A.1 Installazione Xilnix Tool

Al fine di effettuare tutta la creazione e riconfigurazione della Zedboard tramite Petalinux, dobbiamo scaricare gli opportuni tools forniti dalla Xilnix.
I tools necessari sono:

- Vivado
- Vitis

Essi ci permettono di creare il bitstream, quindi di definire l'hardware della zedboard, al fine di far ciò scarichiamo il tutto dal sito Xilnix, previa registrazione, facendo attenzione alla versione di linux che si sta usando, per tutta la guida è stato usato Ubuntu 18.04.01, anche se la documentazione riferisce compatibile la versione 20.04.01(fino a .05).
Una volta terminata l'installazione di circa 60gb, dobbiamo importare i file inerenti alla zedboard, al fine di semplificare il lavoro di creazione del progetto. Per far ciò si seguano i seguenti passaggi:

- Aprire la cartella d'installazione di vivado e seguire il path

  path/to/vivado/version/data/boards/

- Scaricare il file zedboard
- Estrarlo in questa cartella

In questo modo avremo installato la scheda su Vivado.

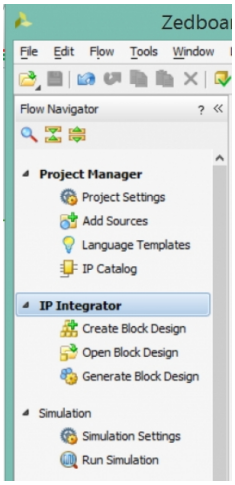
A.1.1 Creazione di un progetto

Al fine di creare il progetto sarà necessario seguire i seguenti step:

- Aprire vivado
- Cliccare su "Create a new vivado project"
- Andare avanti e dare un nome al progetto
- Selezionare il tipo di progetto, che sara **RTL Project**
- Inserire se esistono eventuali risorse di altri progetti già fatti in precedenza
- Selezionare la scheda, quindi selezionare la schermata "Boards" e ricercare "zedboard"
- Concludere la creazione

A.1.2 Creazione modello

Arrivati a questo punto dobbiamo andare a definire il nostro hardware, partendo dalla creazione dello schema, quindi creiamo un nuovo blocco

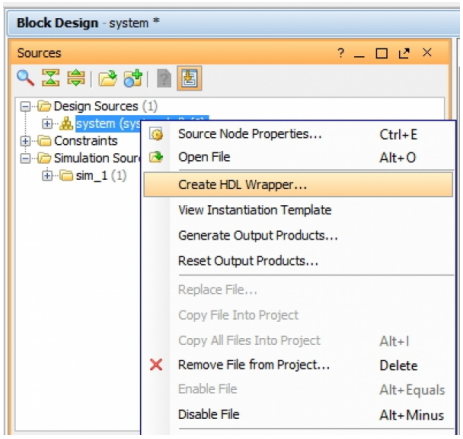


Definiamo il nome e proseguiamo con la scelta dei componenti.
Cliccando il pulsante "Add IP" andiamo a cercare il processing system, che sarà obbligatoriamente **ZYNQ7 Processing System**.
Da questo punto possiamo aggiungere qualsiasi elemento di nostro interesse al fine di creare

la struttura che più necessitiamo, effettuando ad ogni inserimento il comando *Run Connection Automation* che ci sarà consigliato da Vivado.
Una volta inseriti tutti gli elementi richiesti procediamo con la rigenerazione del layout tramite il comando `regenerate_bd_layout`.
Procediamo con le simulazioni e le sintesi al fine di poter generare il bitstream.

Creazione HDL Wrapper

L'HDL Wrapper è un file di High definition Level, che ci permette di definire il nostro progetto, tramite esso effettueremo le sintesi ed il bitstream.
Al fine di crearlo basterà cliccare con il tasto destro sul design creato in precedenza e cliccare la dicitura *Create HDL Wrapper*



Sintesi

Adesso dobbiamo avviare la sintesi, cosi da far tradurre il file HDL in una netlist[1], al fine di farla partire usiamo il pulsante *Run Synthesis* che si trova nella tool bar di sinistra.

Implementazione

Questo processo mappa la sintesi che abbiamo svolto nel chip che abbiamo selezionato, nel nostro caso la zedboard. Anche qui troviamo la possibilità di far partire il tutto tramite il

[1]Contiene tutti i componenti che ci servono

pulsante *Run Implementation* nella tool bar di sinistra.

A.1.3 Generazione del bistream

Una volta terminati tutti questi passaggi andati a buon fine possiamo creare il file ".bit" che ci servirà più avanti per definire il kernel di Petalinux. Al fine di far partire la generazione si usa il comando *Generate Bitstream* presente nella tool bar di sinistra.

A.1.4 Export del file di developing

Al fine di creare sul nostro hardware l'immagine di Petalinux, al fine di far ciò:

- Una volta seguiti tutti gli step precedenti clicco in alto a sinistra il pulsante file
- Export, il terzultimo pulsante
- Export hardware
- Una volta aperta la finestra andiamo avanti e selezioniamo **Include bitstream**
- Diamo un nome ed una directory ed abbiamo esportato tutto il necessario per Petalinux

Appendice B

B.1 Programmazione StandAlone

Al fine di effettuare la programmazione StandAlone, basterà seguire i passaggi visti in precedenza fino al build del progetto, una volta che il progetto sarà buildato sarà necessario aver installati i driver della scheda.

B.1.1 Installazione Driver

Spostiamoci nella cartella d'installazione di vivado e dopodichè nel seguente path

```
1 /data/xicom/cable_drivers/lin64/install_script/install_drivers/
```

da qui dovremmo semplicemente eseguire lo script.

B.1.2 Flash

Al fine di effettuare il flash sulla scheda dovremmo ritornare nel IDE, collegare la scheda nella micro USB J17, quella vicino i connettori audio, ed eseguire i seguenti passaggi:

- Xilnix -> Program Device. In questa schermata che ci comparirà dobbiamo selezionare il bitstream collegato al nostro progetto e clicchiamo program.

- Una volta effettuato questo passaggio, clicchiamo sul progetto tasto destro *Run As -> Launch Hardware*, dopodichè avremo sulla scheda ciò che abbiamo programmato.

B.1.3 Collegamento Tramite UART

Al fine di vedere il risultato a schermo della programmazione che si sta effettuando, bisogna connettersi tramite UART al connettore micro USB J14 e tramite un monitor seriale Tera Term su windows e tio su linux, connettersi alla scheda che solitamente sotto ambiente windows sarà COM* e sotto linux /dev/ttyACM* con un baudrate[1] di 115200, da qui potremmo interfacciarci con la scheda per usufruire del codice bare-metal che è stato caricato.
Questo metodo è equivalente se è presente Petalinux sulla scheda.

[1] velocità di trasmissione

Appendice C

C.1 Petalinux

Per installare Petalinux, assicuriamoci che dopo l'installazione del pacchetto Vivado e Vitis restino al più 3GB.
Soddisfatto questo requisito, abbiamo la possibilità di percorrere due strade, l'installazione tramite self-extracting pack (una sorta di zip), ed l'installer della xilnix. Noi useremo il self-extracting pack, tanto sono equivalenti.

C.1.1 Installazione

Scarichiamo il file presente alla pagina Petalinux, se si usa una versione del sistema operativo compatibile anche l'ultima versione(2021.2) è funzionante, altrimenti usare la versione (2020.3).

C.1.2 Requisiti

Solitamente le dipendenze sono:

```
1 sudo apt-get -y install iproute2 \
2 gcc \
3 g++ \
4 net-tools \
5 libncurses5-dev \
6 zlib1g:i386 \
```

```
7  libssl-dev \
8  flex \
9  bison \
10 libselinux1 \
11 xterm \
12 autoconf \
13 libtool \
14 texinfo \
15 zlib1g-dev \
16 gcc-multilib \
17 build-essential \
18 screen \
19 pax \
20 gawk \
21 python3 \
22 python3-pexpect \
23 python3-pip \
24 python3-git \
25 python3-jinja2 \
26 xz-utils \
27 debianutils \
28 iputils-ping \
29 libegl1-mesa \
30 libsdl1.2-dev \
31 pylint3 \
32 cpio
```

C.1.3 Installazione

Una volta scaricato il file assegnare ad esso i permessi d'esecuzione e spostarlo nella directory desiderata. Una volta effettuati questi passaggi eseguire il seguente comando:

```
1 ./petalinux-v<petalinux-version>-final-installer.run --dir
     /home/<user>/
2 petalinux/<petalinux-version>
```

Aggiungendo l'argomento

```
1 --platform "arm"
```

si può ridurre il peso dell'installazione a solo le zynq arm.

C.1.4 Setup dell'ambiente lavorativo

Al fine di poter usare il tool bisogna eseguire il seguente comando:

```
1 $ source <path-to-installed-PetaLinux>/settings.sh
2 $ source <path-to-installed-PetaLinux>/settings.csh
```

In base al tipo di Shell in cui ci troviamo.

Se l'output dovesse essere di questo tipo:

```
1 PetaLinux environment set to '/opt/pkg/petalinux'
2 INFO: Checking free disk space
3 INFO: Checking installed tools
4 INFO: Checking installed development libraries
5 INFO: Checking network and other services
6 WARNING: No tftp server found - please refer to "UG1144␣
     <petalinuxversion>␣PetaLinux␣Tools␣Documentation␣Reference␣
     Guide" for its impact
7 and solution
```

O al più presentare il seguente warning:

```
1 WARNING: /bin/sh is not bash
```

Tutto è andato a buon fine e si può procedere con la creazione di una build.

Appendice D

D.1 Usare l'SD

- Formattare un SD da almeno 8GB, in due partizioni, la prima che useremo come BOOT in FAT32, con una dimensione di almeno 1GB, la seconda che useremo come root in ext4, da almeno 6GB.
- Ora spostiamo i file :
  - BOOT.BIN
  - image.ub, è l'immagine di linux
  - boot.scr
- Estraiamo rootfs.tar.gz nella partizione di root
- inseriamo l'sd nella scheda assicurandoci che sia impostata in boot da SD, quindi connettiamoci all'usb J14 e tramite un terminale seriale vediamo il boot della scheda.

Appendice E

E.1 Installazione BootGen

Per installare bootgen è necessario usare il tool di download precedentemente scaricato in A.1.4 e nel menu di selezione per l'installazione scegliere BOOTGEN.

Appendice F

F.1 Cross Compilazione

Per effettuare la cross-compilazione la strada più semplice è quella di sfruttare il cross compilatore già presente nella console di vivado, quindi per far ciò si avvii tramite shell vivado e ci si sposti nella cartella contentente il file in C, per compilare eseguire il seguente comando:

```
1 arm-linux-gnueabihf-gcc [nome].c -o [nome].out
```

Codice F.1: Cross-compilazione

Appendice G

G.1 Codice completo driver GPIO

```
1
2  #include <unistd.h>
3  #include <time.h>
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include <string.h>
7  #include <fcntl.h>
8  #include <sys/mman.h>
9  #include <sys/time.h>
10 #include <sys/types.h>
11 #include <sys/stat.h>
12
13 #define GPIO_BASE_ADDRESS       0x42000000
14 #define GPIO_MAP_SIZE           0x1000
15
16 #define XAXIGPIO_DATA_OFFSET    0x00000000
17
18 typedef unsigned long uint32;
19
20
21 int main(int argc, char *argv[]){
22     unsigned long ii;
23     int memfd;
```

```
24      void *gpioMapAddr = NULL;
25
26      if ( (memfd = open("/dev/mem", O_RDWR | O_DSYNC)) == -1 ){
27          printf("Non riesco ad aprire /dev/mem.\n");
28          exit(-1);
29      }
30
31      gpioMapAddr = mmap(0, GPIO_MAP_SIZE, PROT_READ | PROT_WRITE,
            MAP_SHARED, memfd, GPIO_BASE_ADDRESS);
32      if (gpioMapAddr == (void *) -1){
33          printf("Non posso mappare il CDMA.\n");
34          exit(-1);
35      }
36
37      printf("GPIO mappato %p\n", gpioMapAddr);
38
39      for(ii=0; ii<256; ii++){
40          *((volatile unsigned long*) (gpioMapAddr +
                XAXIGPIO_DATA_OFFSET)) = ii;
41          printf("led = 0x%02x\n", ii);
42          usleep(20000); // sleep 20ms
43      }
44  }
```

Codice G.1: Codice completo del driver GPIO

Qualora si volesse usare un GPIO dual channell, bisogna modificare l'offset che di default è

$$GPIO\_BASE\_ADDRESS + 0x08$$

Appendice H

H.1 Generazione automatica del file .bin

Al fine di effettuare la generazione automatica del file sarà necessario eseguire il seguente script, passandogli come parametri:

- path/to/bitstream
- Architettura
- path/to/output

Inoltre sarà necessario creare un file *template.bif* da posizionare nella stessa cartella d'esecuzione dello script

```
1  #!/bin/bash
2  TMP_PATH=$(mktemp -d)
3  print_invalid_usage() {
4          echo "Invalid␣usage."
5          echo "Usage:␣generate␣<input_file_path>␣
               <output_directory_path>"
6  }
7  generate_bin() {
8          BIT_PATH="$1"
9          ARCH="$2"
10         OUTPUT_PATH="$3"
11
12         BIT_NAME=$(basename "$BIT_PATH")
```

```
13          TMP_BIT_PATH="$TMP_PATH/$BIT_NAME"
14          cp "$BIT_PATH" "$TMP_BIT_PATH"
15
16          BIF_TEMPLATE_PATH="$template.bif"
17          TMP_BIF_PATH="$TMP_PATH/bitstream.bif"
18          sed -r "s|template.bit|$TMP_BIT_PATH|"
               "$BIF_TEMPLATE_PATH" > "$TMP_BIF_PATH"
19
20          TMP_BIN_PATH="$TMP_BIT_PATH.bin"
21          bootgen -image "$TMP_BIF_PATH" -arch "$ARCH"
               -process_bitstream bin
22          if [[ $? -ne 0 ]]; then
23                  echo "Bin␣file␣generation␣failed."
24                  exit 1
25          fi
26
27          echo
28          echo "Generating␣bin␣file..."
29          BIN_PATH=$(realpath "$OUTPUT_PATH/fpga-$BIT_NAME.bin")
30          cp "$TMP_BIN_PATH" "$BIN_PATH"
31          echo "bin:␣$BIN_PATH"
32  }
33          BIT_PATH="$1"
34          ARCH="$2"
35          OUTPUT_PATH="$3"
36
37          if [[ "$BIT_PATH" = "" ]]; then
38                  print_invalid_usage
39                  exit 1
40          fi
41
42          if [[ "$OUTPUT_PATH" = "" ]]; then
43                  print_invalid_usage
44                  exit 1
```

```
45          fi
46
47          if [[ ! -f "$BIT_PATH" ]]; then
48                  echo "Invalid␣bit␣file␣path␣provided."
49                  echo "Make␣sure␣the␣path␣to␣the␣bit␣file␣is␣right."
50                  exit 1
51          fi
52
53          if [[ ! -d "$OUTPUT_PATH" ]]; then
54                  echo "Invalid␣output␣directory␣provided."
55                  echo "Make␣sure␣the␣path␣to␣the␣output␣directory␣
                       is␣right."
56                  exit 1
57          fi
58
59          case "$ARCH" in "zynq")          ;;      *)
60                  echo "Invalid␣arch␣provided."
61                  exit 1
62                  ;;
63          esac
64
65          generate_bin "$BIT_PATH" "$ARCH" "$OUTPUT_PATH"
66
67
68  echo
69  echo "Cleaning␣up..."
70  rm -rf "$TMP_PATH"
71
72  echo
73  echo "Finished."
```

Codice H.1: Codice completo generazione automatica file BIN

Appendice I

I.1 Script riprogrammazione

Il seguente script necessita come parametri:

- Il percorso fino al file binario
- il nome del file binario

```
1 echo 0 > /sys/class/fpga_manager/fpga0/flags
2 mkdir -p /lib/firmware/
3 $path = $1
4 $nameBIN = $2
5 cd $path
6 cp $nameBIN /lib/firmware/
7 echo $nameBIN > /sys/class/fpga_manager/fpga0/firmware
```

Codice I.1: Codice completo riprogrammazione automatica FPGA

Bibliografia

[1] *AMBA Technical Document*. https://developer.arm.com/documentation/102202/0200/What-is-AMBA--and-why-use-it-.

[2] *AXI Technical Document*. https://developer.arm.com/documentation/ihi0022/e/AMBA-AXI3-and-AXI4-Protocol-Specification.

[3] *Blazar Documentation*. https://docs.openstack.org/blazar/latest/user/introduction.html.

[4] *Cyborg Documentation*. https://docs.openstack.org/cyborg/latest/.

[5] «IEEE Standard for VHDL Language Reference Manual». In: *IEEE Std 1076-2019* (2019), pp. 1–673. DOI: 10.1109/IEEESTD.2019.8938196.

[6] «IEEE Standard Verilog Hardware Description Language». In: *IEEE Std 1364-2001* (2001), pp. 1–792. DOI: 10.1109/IEEESTD.2001.93352.

[7] Ian Kuon, Russell Tessier e Jonathan Rose. 2008.

[8] Mattia Morabito. «Strumenti di programmazione remota per dispositivi FPGA». In: (2021).

[9] *Nova Documentation*. https://docs.openstack.org/nova/latest/.

[10] *OpenStack Site*. https://www.openstack.org/software/.

[11] Sunita Ramagond, Siva Yellampalli e C Kanagasabapathi. «A review and analysis of communication logic between PL and PS in ZYNQ AP SoC». In: *2017 International Conference On Smart Technologies For Smart Nation (SmartTechCon)*. 2017, pp. 946–951. DOI: 10.1109/SmartTechCon.2017.8358511.

[12] *Solution Zynq PL Programming With FPGA Manager*. https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18841645/Solution+Zynq+PL+Programming+With+FPGA+Manager.

[13] *Technical Reference Manual UG585*. https://docs.xilinx.com/v/u/en-US/ug585-Zynq-7000-TRM.

[14] Kizheppatt Vipin e Suhaib A. Fahmy. «ZyCAP: Efficient Partial Reconfiguration Management on the Xilinx Zynq». In: *IEEE Embedded Systems Letters* 6.3 (2014), pp. 41–44. DOI: 10.1109/LES.2014.2314390.

[15] Zhe Wang et al. «Resource Partitioning and Application Scheduling with Module Merging on Dynamically and Partially Reconfigurable FPGAs». In: *Electronics* 9 (set. 2020), p. 1461. DOI: 10.3390/electronics9091461.

[16] *What is OpenStack?* https://ubuntu.com/openstack/what-is-openstack.

[17] *Zycap Github*. https://github.com/warclab/zycap.

[18] *ZYNQ-7000*. https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html.

Ringraziamenti

Ai professori Francesco Longo e Giovanni Merilino

Per il supporto datomi durante la scrittura della tesi e durante la mia carriera universitaria.

A SIC - Stretto in Carena

Per avermi permesso di crescere sia come persona che come Ingegnere, rendendo il percorso unico.

Ai miei amici e colleghi

Per l'aiuto, il supporto, le risate ed il tempo passato insieme in dipartimento e non.

A Silvia

Per essermi stata affianco in questo percorso nonostante tutte le avversità del tragitto.

Alla mia famiglia

Per il supporto in ogni mia decisione e per essere il paracadute in una vita in caduta libera.