POLITECNICO DI MILANO

Facoltà di Ingegneria dell'Informazione

Corso di Laurea Magistrale in Ingegneria Informatica

Flusso per la generazione automatica di architetture per sistemi riconfigurabili

Relatore: Prof. Marco Domenico SANTAMBROGIO

Correlatore: Dott. Ing. Gianluca Carlo DURELLI

Tesi di Laurea di:

Fabrizio Spada

Matricola n. 787210

Anno Accademico 2013–2014

Desidero innanzitutto ringraziare i miei genitori per la possibilità che mi hanno dato di frequentare l'università e per la comprensione e la fiducia che hanno avuto in me.
Uno speciale ringraziamento a Monica, la persona che più di tutte mi è stata vicina durante questi anni di università riempiendomi le giornate di gioia e amore, e alla sua famiglia che ha saputo spingermi a dare sempre il massimo.
Un caloroso grazie ai miei fratelli che in un modo o nell'altro mi hanno sempre incoraggiato nelle scelte intraprese.
Ci tengo a ringraziare tutti i ragazzi del Necst per aver condiviso con me un anno pieno di impegni ma altrettanto ricco di soddisfazioni. In particolare ringrazio Alberto, Gabriele e Gianluca con i quali mi sono subito trovato in sintonia insieme al mio relatore Marco che ha sempre cercato di sostenermi ed incoraggiarmi nel lavoro di tutti i giorni.
Voglio inoltre ringraziare il mio storico compagno di università Marco per i momenti di studio insieme condividendo ansia e felicità per gli esami superati.

Indice

1 Introduzione **1**

2 Lavori correlati **10**

2.1 Daedalus 10

2.2 RAMPSoC 13

2.3 XPilot 15

2.4 Sommario 18

3 MPSoC e FPGA: Zedboard **20**

3.1 Sistema ZYNQ e protocolli hardware 21

3.2 Flusso di sviluppo per Zedboard 28

3.3 Comunicazione 33

3.4 Sommario 34

4 Formulazione del problema **35**

4.1 Definizione del problema trattato 35

4.2 Modello dell'applicazione 37

4.3 Modello e architettura di comunicazione 39

4.4 Supporto Software 41

4.5 Sommario 42

5 Generazione architettura hardware **44**

5.1 Definizione del flusso per la creazione dell'architettura HW 44

5.2 Implementazione 46

5.3 Sommario 55

6 Generazione supporto software **56**

6.1 Definizione del flusso per la generazione delle API 56

6.2 API Software 58

6.3 AXI DMA driver 62

6.4 Sommario 68

7 Caso di studio **71**

7.1 Filtro Otsu 71

7.2 Descrizione in forma di Taskgraph e SDF 76

7.3 Prototipazione filtro Otsu 77

7.3.1 Soluzione A: computeHistogram in HW 78

7.3.2 Soluzione B: halfProbability in HW 80

7.3.3 Soluzione C: computeHistogram e halfProbability in HW . . 83

7.3.4 Soluzione D: intera fase in HW 85

7.4 Sommario 89

8 Conclusioni e lavori futuri **91**

Bibliography **97**

Elenco delle figure

1.1 Flusso di lavoro da applicazione software a implementazione su scheda . . . 7

2.1 Flusso di lavoro di Daedalus [1] . . . 11
2.2 Daedalus: dettagli implementativi [1] . . . 12
2.3 Flusso di lavoro di RAMPSoC [2] . . . 14
2.4 CAP-OS [3] . . . 16
2.5 XPilot [4] . . . 17

3.1 Zynq-7000 AP SoC Overview . . . 23
3.2 Architettura del canale read del protocollo AXI . . . 27
3.3 Architettura del canale write del protocollo AXI . . . 27
3.4 Traduzione da funzione a IP con interfaccia AXI4-Lite . . . 29
3.5 Xilinx IP AXI Interconnect . . . 30
3.6 Traduzione da funzione a IP con interfaccia AXI4-Stream . . . 31
3.7 Scambio dati tra memoria RAM e funzione stream . . . 31
3.8 Architettura hardware di una funzione stream . . . 32
3.9 Xilinx AXI DMA interfacce . . . 33

4.1 Esempio di un taskgraph gerarchico, il nodo N3 viene espanso in un sottografo . . . 39
4.2 Architettura di comunicazione di un taskgraph . . . 41
4.3 Composizione a livelli di un sistema riconfigurabile . . . 42

5.1 Diagramma del flusso di generazione dell'architettura . . . 45

5.2 Descrizione testuale di un taskgraph 48
5.3 Esempio grafico di un taskgraph base 49
5.4 Schema sintattito della rappresentazione testuale del taskgraph . . 49
5.5 Schema del flusso di esecuzione della toolchain 54

6.1 Diagramma del flusso di generazione delle API 57
6.2 Coerenza tra cache e RAM con trasferimenti tramite AXI DMA . . 63
6.3 Comparazione delle prestazioni di accesso alla memoria tra ARM e DMA 69

7.1 Immagine prima dell'applicazione del filtro Otsu 72
7.2 Immagine dopo l'applicazione del filtro Otsu 72
7.3 Rappresentazione grafica del metodo Otsu 74
7.4 Taskgraph filtro Otsu 77
7.5 Taskgraph hardware della Soluzione A 78
7.6 Partizionamento HW/SW della Soluzione A 79
7.7 Architettura Soluzione A 80
7.8 Taskgraph hardware della Soluzione B 81
7.9 Partizionamento HW/SW della Soluzione B 81
7.10 Architettura Soluzione B 82
7.11 Taskgraph hardware della Soluzione C 83
7.12 Partizionamento HW/SW della Soluzione C 84
7.13 Architettura Soluzione C 85
7.14 Partizionamento HW/SW della Soluzione D 87
7.15 Taskgraph della Soluzione D 88
7.16 Architettura Soluzione D 89

Elenco delle tabelle

7.1 Tabella delle risorse hardware occupate dall'architettura in figura 7.6 80
7.2 Tabella delle risorse hardware occupate dall'architettura in figura 7.9 82
7.3 Tabella delle risorse hardware occupate dall'architettura in figura 7.12 85
7.4 Tabella delle risorse hardware occupate dall'architettura in figura 7.16 89

Elenco delle abbreviazioni

API Application Programming Interface

BRAM Block RAM

DSL Domain Specific Language

DSP Digital Signal Processor

GPP General Purpose Processor

HDL Hardware Description Language

HLS High Level Synthesis

KPN Kahn Process Network

LUT Look-Up Table

RTL Register Transfer Level

SDF Synchronous Data Flow

SoC System on Chip

VHDL VHSIC Hardware Description Language

Sommario

Nell’ambito dei sistemi embedded hanno riscosso una notevole diffusione i cosiddetti MPSoC, ovvero sistemi dotati di più processori sul chip. Per incrementare le prestazioni delle applicazioni embedded tali sistemi vengono in genere affiacanti alle FPGA, ovvero circuiti dotati di logica programmabile che permettono l’implementazione di funzionalità hardware personalizzate.
Le FPGA hanno ottenuto un ruolo sempre più importante per lo sviluppo dei sistemi embedded, vengono utilizzate sia come componenti a se stanti o in collaborazione con General Purpose Processor (GPP), come nel caso del sistema trattato in questa tesi.
Ad oggi le fasi relative allo sviluppo di applicazioni per sistemi riconfigurabili, come MPSoC dotati di FPGA, vanno dalla co-progettazione hw/sw della applicazione fino alla realizzazione su scheda di sviluppo e sono realizzate a mano dal progettista tramite strumenti CAD forniti dal produttore della scheda. Il procedimento di sviluppo può essere complesso e di conseguenza lo sviluppo di queste applicazioni può risultare lento e macchinoso.
L’ obiettivo di questo lavoro di tesi è quello di incrementare il livello di astrazione e nascondere i dettagli implementativi per permettere ai programmatori di lavorare solo al livello applicativo.
Ci si soffermerà sulla definizione del concetto di Taskgraph, la quale permetterà di riuscire a formulare il problema in esame: ovvero, data la rappresentazione di un’applicazione sotto forma di Taskgaph, generare automaticamente l’architettura hardware provvista di driver per il sistema operativo Linux e le interfacce in linguaggio C per le applicazioni che useranno tale architettura.

Sarà spiegata in dettaglio la struttura della toolchain sviluppata per risolvere tale problema illustrando tutti i dettagli implementativi.
Verranno infine approfonditi tutti i passaggi sull'uso della toolchain sviluppata tramite l'analisi di un caso di studio reale.

Capitolo 1

Introduzione

Le applicazioni software, con il passare degli anni, hanno subito un enorme aumento in termini di complessità strutturale e delle risorse computazionali che queste richiedono. Da un lato la quantità di dati da gestire è cresciuta enormemente, grazie anche alla maggior disponibilità e facilità nel recuperarli, e dall'altro i requisiti in termini di performance richieste alle applicazioni non cambiano. Per garantire quindi la stessa qualità di servizio all'aumento dei dati gestiti deve corrispondere un aumento in termini di potenza di calcolo.
Inizialmente, al fine di incrementare le prestazioni fornite dai sistemi computazionali, la soluzione proposta era quella dell'aumento della frequenza di clock del processore con conseguente crescita della potenza consumata. Tale soluzione ha però una evidente limitazione fisica: un aumento della potenza corrisponde ad un incremento del calore da dissipare, fino a superare le capacità delle normali tecniche di raffreddamento, questo fenomeno è noto come *Power Wall* [5].
Per aggirare tale limitazione fisica e migliorare le performance dei sistemi di calcolo, sono state create le architetture multicore. Affiancando più processori sullo stesso chip si sono incrementate le prestazioni riducendo al tempo stesso la frequenza di funzionamento. Infine un sistema multicore apporta nuove problematiche per i programmatori in quanto il parallelismo ottenuto dai sistemi multicore è teorico e necessita di una maggior competenza tecnica per essere sfruttato.
Al fine di ottenere prestazioni sempre più elevate, recentemente le nuove architetture sviluppate sono diventate eterogenee, ovvero vi è la presenza di elementi

di computazione di natura diversa. Lo scopo dell'introduzione di unità eterogenee nel sistema è quello di sfruttarle per ottenere vantaggi in termini di performance e di performance per watt. Per citare qualche esempio, di recente ha avuto successo l'utilizzo della GPU da affiancare alla CPU per un tipo di computazione anche diversa da quella grafica oppure sfruttare le FPGA come acceleratori hardware per le applicazioni. Data la natura di questi strumenti, essi presentano un elevato grado di parallelismo disponibile e nel caso delle FPGA è possibile personalizzare l'hardware in base alle esigenze.
Le FPGA rispetto alle GPU presentano maggiori vantaggi non solo perché permettono la generazione di hardware ad-hoc ma anche perché hanno un consumo minore di potenza, bisogna inoltre tenere in considerazione che la potenza assorbita dalle GPU, in alcuni domini applicativi, non è accettabile. Per citare un esempio le FPGA sono molto utilizzate nell'ambito dei sistemi embedded e cominciano ad essere impiegate anche come acceleratori per il calcolo scientifico [6].
L'uso delle FPGA come acceleratori hardware presenta sia vantaggi che svantaggi, infatti, come già esposto si ottiene un aumento delle prestazioni anche in relazione alla potenza consumata, ma nel contempo presentano un'elevata difficoltà di programmazione e per ottenere una funzionalità in hardware bisogna generare il Register Transfer Level (RTL) da integrare poi con il resto del sistema. Infine è necessario implementare le interfacce hardware e software verso il sistema riconfigurabile.
Nell'ambito delle applicazioni embedded hanno avuto un'enorme diffusione i cosiddetti MPSoC (Multiprocessor System-on-Chip)[7], ossia sistemi dotati di più processori sul chip.
L' MPSoC di per sé non costituisce un sistema riconfigurabile poiché l' hardware è statico e ben noto a runtime. Per renderlo rinconfigurabile bisogna aggiungere al sistema un elemento processante, la cui funzionalità sia di volta in volta personalizzabile in base alle necessità, realizzando di fatto un componente hardware piuttosto che un altro. L'elemento dotato di questa caratteristica si chiama FPGA (Field Programmable Gate Array), ovvero una serie di celle logiche program-

mabili opportunamente interconnesse tra di loro con interconnessioni anch'esse programmabili, al fine di realizzare un circuito hardware che implementi la funzione richiesta. Il blocco base di una FPGA è composto da una memoria chiamata Look-Up Table (LUT) che ha la stessa funzionalità di una tabella della verità, ovvero ad ogni combinazione di valori di bit in ingresso corrispondono dei valori in uscita. Oltre alla LUT vi sono ulteriori elementi che costituiscono una FPGA: le Block RAM (BRAM) o i Digital Signal Processor (DSP); i blocchi di input/output che consentono la comunicazione con il mondo esterno; infine vi sono le interconnessioni programmabili che possono collegare i blocchi logici tra di loro oppure con i blocchi di input/output. La riconfigurazione di una FPGA può essere intesa in due modi differenti, in particolare, ci si riferisce alla riconfigurabilità totale statica nel momento in cui si vuole programmare l'intero dispositivo prima del suo utilizzo per impostare la funzionalità voluta; si fa riferimento invece alla rinconfigurabilità parziale dinamica nel momento in cui si vogliono riconfigurare singole parti del dispositivo lasciando intatte e funzionanti le altre (anche durante il processo di riconfigurazione).
Negli ultimi anni una configurazione degli MPSoC che ha riscosso molto successo, per ragioni di prestazioni e di efficienza energetica, è quella che combina un processore dual-core ARM, una gerarchia di memoria con almeno una RAM e due livelli di cache. Per citare un esempio, Xilinx è uno dei produttori di FPGA più conosciuti e produce delle schede dotate di MPSoC per sfruttare al meglio l'uso delle FPGA. La scheda di sviluppo Zedboard utilizzata in questo lavoro di tesi ne è un esempio, in generale tutta la famiglia Zynq-7000 AP SoC di Xilinx è dotata di processori ARM. Una lista delle schede che fanno parte di questa famiglia è presente in [8]. Questi sistemi possono contenere anche elementi processanti eterogenei e sono inoltre dotati di interfacce di I/O per interagire con l'ambiente esterno (USB, SD card, Ethernet ecc...).
In questo contesto con il passare degli anni le FPGA stanno svolgendo un ruolo cruciale per lo sviluppo dei sistemi embedded; la loro capacità di essere riconfigurate staticamente e dinamicamente permette al progettista hardware una certa flessibilità nello sviluppo e testing di circuiti hardware [9]. Proprio per que-

sta flessibilità con quale si può configurare più volte la logica programmabile le FPGA venivano principalmente utilizzate come strumento di prototipizzazione. Recentemente si è assistito ad un aumento dell'utilizzo delle FPGA come componenti a se stanti o in collaborazione con General Purpose Processor (GPP), come nel caso del sistema oggetto di questa tesi. Una tendenza degli ultimi anni è quella di utilizzare questi strumenti riconfigurabili per realizzare dei circuiti che fungono da acceleratori hardware per applicazioni eseguite su GPP.
Lo sviluppo di applicazioni per sistemi riconfigurabili come MPSoC dotati di FPGA coinvolge varie fasi che vanno, dalla progettazione dell'architettura software, fino alla creazione dei core hardware, passando per la generazione delle interfacce di comunicazione per scambiare dati tra processore e area programmabile. Attualmente tutte le fasi che portano dalla co-progettazione hw/sw dell'applicazione fino alla realizzazione vera e propria su scheda di sviluppo vengono realizzate a mano dal progettista, nella maggior parte dei casi ciò avviene tramite l'utilizzo di strumenti CAD forniti dal produttore della scheda sulla quale si andrà ad implementare il sistema finale. Tuttavia l'utilizzo di questi strumenti di progettazione CAD è in genere molto complicato in quanto richiede una preparazione di base da parte del progettista e una profonda conoscenza del tool di sviluppo usato. La grande complessità di utilizzo di questi strumenti, aggiunta alla quantità di dettagli tecnici di cui bisogna tenere conto se si vuole un' implementazione funzionante, fan si che lo sviluppo di tali applicazioni risulti lento e macchinoso anche per i più esperti.
I principali vendor di applicazioni CAD per FPGA (Xilinx, Altera, ecc.) forniscono strumenti per la scrittura di core hardware direttamente in un linguaggio Hardware Description Language (HDL) e per l'integrazione dei core nell'architettura hardware finale. Vengono inoltre forniti strumenti che tramite High Level Synthesis (HLS) sono in grado di generare il RTL partendo da software scritto in un linguaggio di più alto livello come il C. In particolare tali strumenti di HLS fanno uso di direttive che il progettista può indicare per descrivere alcune proprietà che il core deve avere, ad esempio si può indicare il protocollo per le interfacce oppure se utilizzare ottimizzazioni per incrementare le performance

del core, per esempio il loop unrolling di un ciclo for.
Il problema di questi strumenti CAD è che non sono completamente automatizzati, tutto il lavoro di integrazione dei core nel sistema e la scrittura delle interfacce con le direttive per i core sono da eseguire a mano da parte del progettista dell'applicazione. L'utente che fa uso di tali strumenti deve essere a conoscenza di tutti i dettagli hardware e implementativi, per esempio necessita di conoscere i tipi di interfacce che devono avere i core, i protocolli hardware utilizzati per connettere i core sia tra di loro che con il resto del sistema. Tale basso livello di astrazione presenta parecchie difficoltà nel caso in cui il progettista sia un application designer il quale potrebbe non essere a conoscenza di tutti questi dettagli, viene infatti richiesta una conoscenza approfondita del singolo strumento poiché il flusso di esecuzione potrebbe cambiare in funzione del tool utilizzato.
Dal punto di vista del progettista sarebbe ideale avere a disposizione uno strumento che data un'applicazione generi il software per il GPP e contemporaneamente implementi l'architettura hardware, ovvero sintetizzi le funzioni in hardware se necessario e si occupi dei driver per accedere ai core. In particolare questo strumento dovrebbe possedere le seguenti caratteristiche:

1. realizzazione della fase di HW/SW Codesign
2. creazione dei core HW
3. implementazione dell'architettura
4. integrazione con i tool proprietari in modo da rendere lo strumento del tutto generale
5. generazione dei drivers
6. compilazione dell'applicazione

In questo lavoro di tesi l'obiettivo è quello di incrementare il livello di astrazione nascondendo i dettagli implementativi per consentire ai programmatori di lavorare solo al livello applicativo. Lo scopo del lavoro è quindi quello di evitare che il progettista si occupi della parte di integrazione dell'applicazione con i tool di

sintesi. Infine il programmatore può sfruttare in maniera efficiente gli acceleratori hardware perché non deve occuparsi dei task 2,3,4 e 5 in quanto generati automaticamente dal lavoro realizzato.
La progettazione delle applicazioni per sistemi MPSoC dotati di FPGA, se effettuata con il supporto dello strumento ideale dotato delle caratteristiche sopra citate, risulterebbe semplificata in quanto il flusso di progettazione sarebbe completamente automatico e del tutto trasparente. Allo stato attuale gli strumenti messi a disposizione dai vendor costringono il progettista a tenere in considerazione i seguenti passi (la Figura 1.1 mostra uno schema a cascata del flusso di lavoro):

(A) si parte dall'applicazione di cui si dispone il sorgente e si realizza il partizionamento hardware/software, durante questa fase l'utilizzo di una rappresentazione intermedia come quella in forma di taskgraph può essere di supporto e può semplificare il problema

(B) ottenuto il partizionamento dell'applicazione si procede con lo sviluppo dell'architettura hardware

- si generano moduli hardware con le funzionalità richieste
- si crea tutta l'infrastruttura di comunicazione

(C) si ottiene il bitstream per programmare la logica programmabile

(D) si fa il porting della parte software dell'applicazione sull'MPSoC cross-compilandola per il processore target

(E) si sviluppano i driver e le Application Programming Interface (API) necessarie all'applicazione software per accedere facilmente ai moduli hardware

(F) test dell'intero sistema realizzato su scheda

Il flusso descritto in Figura 1.1 ad oggi viene svolto manualmente dal progettista dell'applicazione. Alcune fasi come quella della creazione dell'architettura hardware e la creazione del bitstream sono difficilmente testabili perché richiedono

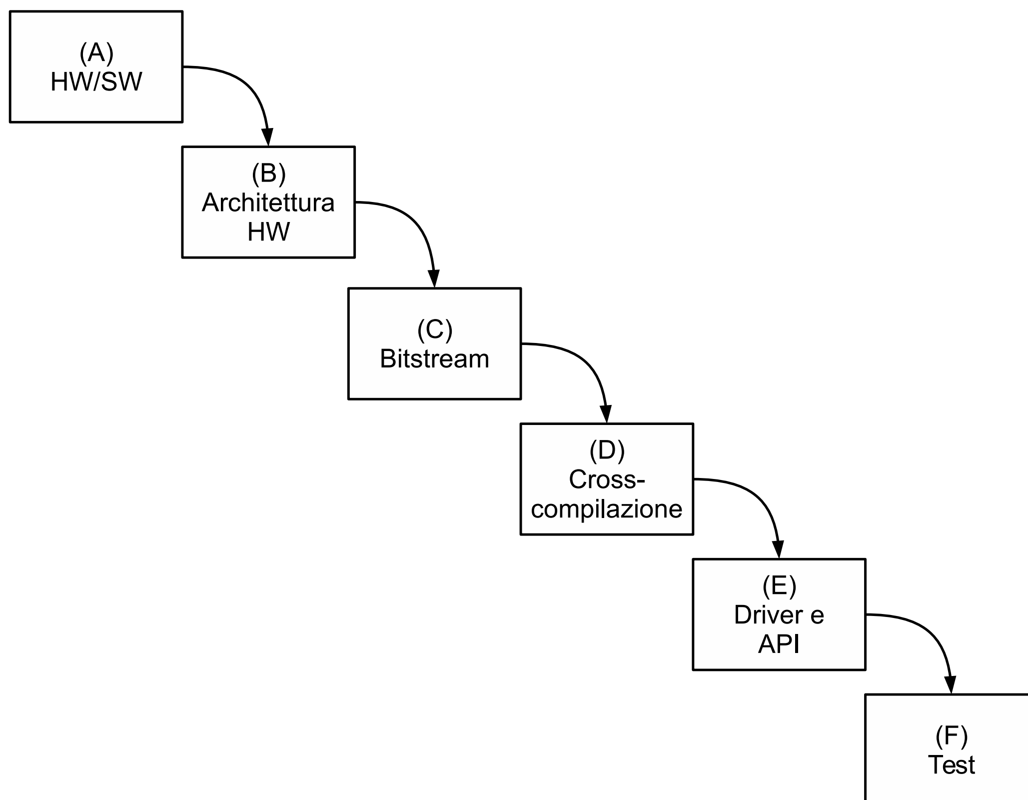


Figura 1.1: Flusso di lavoro da applicazione software a implementazione su scheda

il completamento dell'intero sviluppo per poter essere verificate. Di conseguenza, essendo complicate da debuggare, sono spesso soggette a errori da parte del progettista che si propagano per essere poi scoperti solo a valle dello sviluppo. Commettere uno sbaglio in fase di generazione dell'architettura può quindi ritardatare enormemente lo sviluppo dell'applicazione; arrivare in fondo al processo e accorgersi di tale errore implica il ritorno alla fase di creazione dell'architettura e il rifacimento dell'intero flusso, con conseguente ritardo nella realizzazione del prodotto.

La soluzione proposta da questa tesi mira ad automatizzare questo flusso di sviluppo. La metodologia proposta si pone subito dopo il punto (A), partendo dal punto (B) esegue in maniera automatica tutte le fasi fino al punto (E) compreso. In letteratura vi sono presenti numerosi lavori che si occupano della coprogettazione HW/SW, per questo motivo il partizionamento dell'applicazione è stato lasciato al progettista, così come la verifica funzionale del sistema. Per arrivare a coprire i passi menzionati è stata sviluppata un'applicazione che implementa questa metodologia, sfruttando i tool proprietari rilasciati per la piattaforma target. I contributi proposti in questa tesi sono:

- il metodo di descrizione funzionale dell'applicazione attraverso l'utilizzo della rappresentazione in forma di taskgraph combinata con la descrizione SDF (descritta in dettaglio nel Capitolo 4).
- l'automatizzazione delle seguenti fasi:
  - sintesi ad alto livello delle funzionalità dell'applicazione che si intende implementare in hardware ottenendo l'IP Core da integrare con il resto dell'architettura.
  - composizione di tutti i core generati per ottenere il sistema hardware completo e la generazione del bitstream per programmare l'FPGA.
  - gli IP Core sviluppati devono diventare accessibili da parte dell'applicazione che risiede sul processore, per avere accesso all'FPGA devono essere generati dei driver che gestiscono i core come delle periferiche hardware.

  - infine a supporto dello sviluppatore dell'applicazione si mettono a disposizione delle API software che permettono di interfacciarsi al driver e quindi all'hardware implementato.

In questo capitolo è stata fornita un'introduzione a quello che è il mondo dei sistemi riconfigurabili, presentando il problema e la soluzione proposta per risolverlo.
Il resto della trattazione è organizzata nel modo seguente:

- Capitolo 2: viene quì presentato lo stato dell'arte analizzando i lavori correlati e i framework presenti;
- Capitolo 3: fornisce una descrizione del problema affrontato in questa tesi ed introduce il formalismo utilizzato per rappresentare l'applicazione in questo lavoro;
- Capitolo 4: in questo capitolo vengono riportati i dettagli architetturali della scheda di sviluppo usata in questo lavoro e viene descritto il flusso di lavoro utilizzato per implementare sistemi riconfigurabili su questo dispositivo, attraverso l'uso di sistemi CAD proprietari. Vengono inoltre analizzati i componenti HW che sono di maggior interesse in questo lavoro;
- Capitolo 5: questo capitolo descrive come nell'ambito di questo lavoro di tesi si e' automatizzata la fase di creazione dell'architecttura HW;
- Capitolo 6: contiene la descrizione di come avviene la generazione del supporto software, cioè dei driver linux e le api software;
- Capitolo 7: tale capitolo contiene la presentazione di un caso d'uso reale a supporto del lavoro svolto;
- Capitolo 8: conclusioni e possibili sviluppi futuri.

Capitolo 2

Lavori correlati

In questo capitolo verranno presentati tutti i framework disponibili in letteratura che rappresentano lo stato dell'arte dei lavori correlati al tema affrontato in questa tesi. Per ogni framework verrà illustrato il problema che lo contraddistingue e la soluzione che esso propone. Per primo sarà descritto il funzionamento di Daedalus, verrà poi presentata una descrizione del framework RAMPSoC seguito da XPilot. Quest'ultimo essendo stato acquistato da Xilinx è diventato a tutti gli effetti un framework commerciale, infatti era alla base di AutoESL che negli ultimi anni è stato migliorato e potenziato divenendo quello che tuttora è Vivado [10]. Vivado è anch'esso un tool commerciale utilizzato in questo lavoro di tesi.

2.1 Daedalus

Daedalus [1] è un framework per lo sviluppo di MPSoC sviluppato presso il Leiden Embedded Research Center dell'università di Leiden in Belgio. Questo framework ha l'obiettivo di automatizzare lo sviluppo di architetture MPSoC effettuando in automatico i passi di par- tizionamento Hw/Sw ed assegnamento dei task ai processori, in modo tale da fornire uno strumento in grado di procurare un supporto fondamentale allo sviluppatore nella fase di analisi dello spazio delle soluzioni. Il flusso di lavoro è composto dall'interazione di tre strumenti principali: KPNgen [11], Sesame [12, 13] ed ESPAM [14, 15].
Il flusso realizzato da Daedalus, rappresentato in figura 2.1, parte dall'applicazio-

ne scritta in linguaggio C e arriva alla generazione del codice VHDL per le funzioni da eseguire in hardware, inoltre genera il codice C per le parti da eseguire in software. In sequenza i passi affrontati da Daedalus sono i seguenti:

- KPNgen: analizza il programma sequenziale in ingresso e lo trasforma nella forma di Kahn Process Network (KPN) [16];
- partendo dalla rappresentazione in forma di KPN questo formalismo viene poi analizzato da Sesame che si occupa di esplorare lo spazio delle soluzioni alla ricerca di un insieme di architetture candidate ad eseguire l'algoritmo. Nella fase di partizione Hw/Sw questo strumento supporta il progettista nell'assegnare le diverse parti dell'applicazione alle varie unità di computazione;
- alla fine del flusso di analisi troviamo ESPAM che genera il codice VHDL per i core hardware e il codice C per le funzioni software da eseguire su General Purpose Processor (GPP). Una volta ottenuto il VHDL è possibile generare il bitstream del sistema utilizzando software proprietari come ad esempio Vivado prodotto da Xilinx.

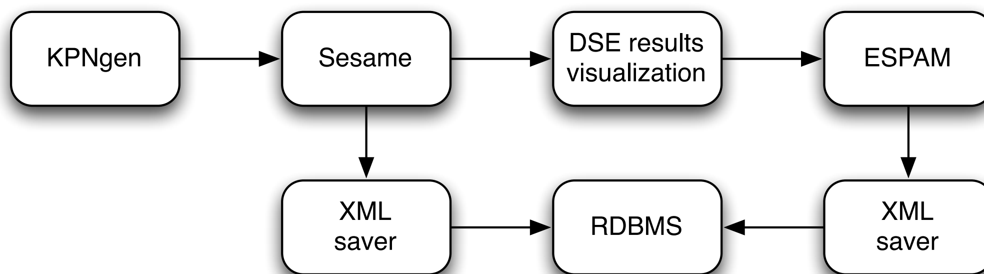


Figura 2.1: Flusso di lavoro di Daedalus [1]

Al fine di aumentare la compatibilità tra le parti che costituiscono Daedalus l'intero flusso si appoggia sul formato XML, come si può notare dalla figura 2.2. Il framework gestisce inoltre un database contenente una libreria di IP-core usata in fase di generazione del codice VHDL.

All'interno dello stesso framework si concentrano svariati lavori: in particolare in [17] viene presentato un algoritmo che lavora su un taskgraph, migliora il

partizionamento Hw/Sw effettuando un *merge* o *split* dei task iniziali allo scopo di ottimizzare il tempo di esecuzione. In questo lavoro di tesi non si affronta il problema del partizionamento dell'applicazione, le tematiche affrontate partono dalla premessa di considerare un'applicazione già partizionata. L'algoritmo presentato in [17] potrebbe quindi essere utilizzato a monte per ottenere il partizionamento Hw/Sw da fornire al framework sviluppato.
Nei lavori [18, 19, 20, 21] gli autori forniscono analisi e metodi per ricavare un modello in forma di *Polyhedric Process Networks*[11] partendo da un'applicazione scritta con un linguaggio di alto livello.
Altri lavori infine si concentrano sulla ottimizzazione dei loop per la loro sintesi su hardware riconfigurabili [22] e sulla definizione di uno scheduler per il sistema da generare [23].

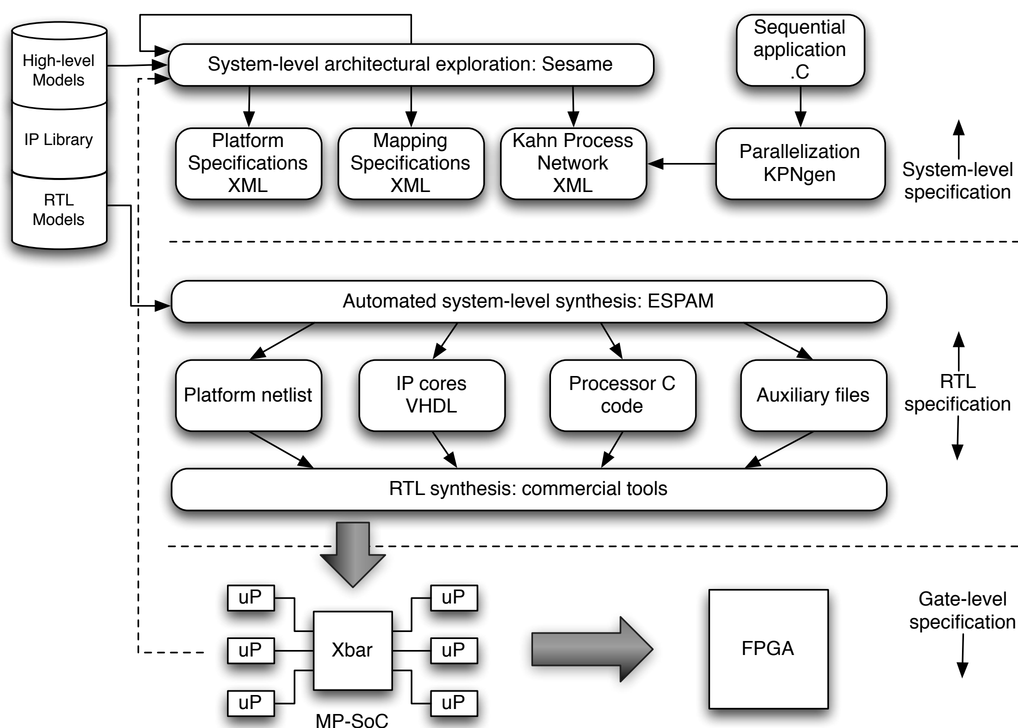


Figura 2.2: Daedalus: dettagli implementativi [1]

2.2 RAMPSoC

Reconfigurable Architecture Multi Processor System on Chip RAMPSoC [2] è un framework per lo sviluppo di sistemi basati su architetture riconfigurabili, nello specifico FPGA, è stato sviluppato in Germania presso il Karlsruher Instisute of Technologie. RAMPSoC è un framework completo che partendo da una decrizione ad alto livello dell'applicazione (C/C++) arriva fino al bitstream pronto ad essere configurato su scheda, è capace di supportare sistemi in grado di adattarsi a runtime al carico di lavoro.
Di seguito viene presentato il flusso di lavoro del framework che è mostrato in figura 2.3 ed è diviso in tre fasi che sono così composte:

- Fase 1:
  - in ingresso riceve un'applicazione sequenziale scritta in linguaggio C/C++;
  - le funzioni presenti vengono profilate e viene analizzato l'*overhead* di comunicazione;
  - viene effettuata una fase di *tracing* al fine di ricavare il call graph [24] dell'applicazione;
  - viene eseguito un algoritmo di clustering volto ad effettuare un partizionamento Software/Software;
  - infine viene definita l'architettura e i moduli applicativi.
- Fase 2:
  - ogni modulo applicativo viene profilato al fine di identificare cicli o blocchi di codice computazionalmente intensivi, delle porzioni di codice identificate viene suggerita l'implementazione in hardware del task. La creazione del modulo hardware viene lasciata al progettista.
- Fase 3:
  - in questa fase vengono generati gli eseguibili software dei moduli applicativi identificati;

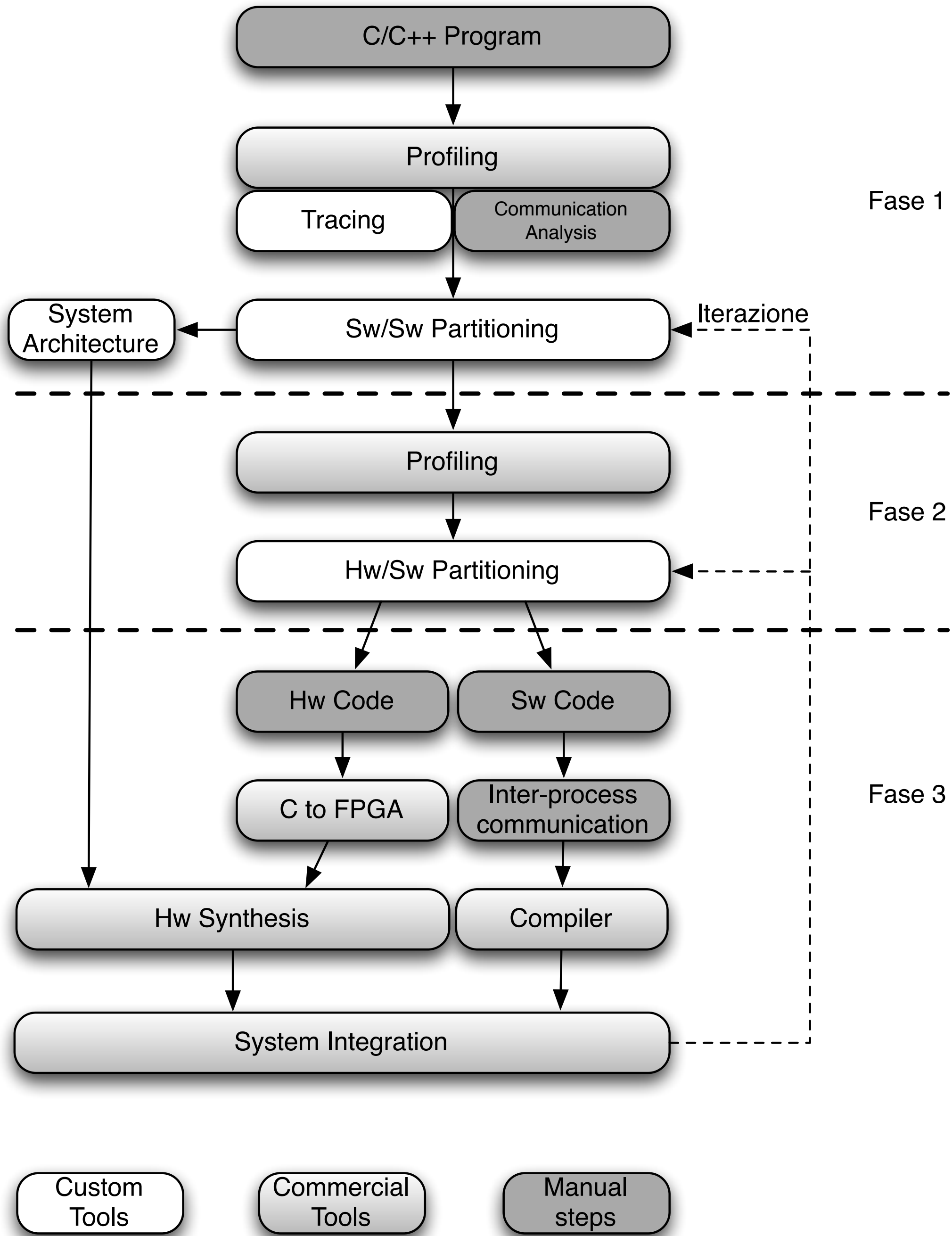


Figura 2.3: Flusso di lavoro di RAMPSoC [2]

- per i moduli implementati in hardware viene usato uno strumento di HLS al fine di ottenere una traduzione in HDL;
- vengono generati infine i bitstream completi e quelli parziali, quest'ultimi usando uno strumento appositamente sviluppato al fine di semplificare il lavoro del progettista [25].

RAMPSoC è l'unico tra i framework in analisi che supporta sistemi che si adattano al carico di lavoro a runtime. Il layer software di questi sistemi necessita di impiegare strategie svolte ad ottimizzare i tempi di riconfigurazione del dispositivo al fine di minimizzare l'overhead dovuto alla riconfigurazione stessa.
CAP-OS [3] è il layer software utilizzato per gestire l'architettura di RAMPSoC; esso ha come input la descrizione, in forma di taskgraph, delle applicazioni che possono essere eseguite e per ognuna di esse le diverse implementazioni disponibili con i relativi tempi di esecuzione e riconfigurazione. CAP-OS sfrutta il riuso dei componenti ed il prefetch delle riconfigurazioni; fornisce inoltre la possibilità di interrompere una riconfigurazione in atto se un task con maggior priorità è pronto per essere riconfigurato; affinché un task possa essere riconfigurato è necessario che tutti i suoi predecessori siano già stati configurati.
L'algoritmo che viene sfruttato da CAP-OS per gestire le riconfigurazioni è rappresentato in Figura 2.4. Analizzando la figura risulta evidente come l'approccio proposto cerchi di minimizzare il numero delle riconfigurazioni necessarie riutilizzando, ove possibile, processori già presenti e liberi oppure che saranno liberi a breve.

2.3 XPilot

XPilot [4] è l'unico prodotto tra quelli presentati ad avere avuto un seguito in ambito commerciale. Inizialmente sviluppato presso il dipartimento di Computer Science dell'università della California è stato poi acquistato e sviluppato da Xilinx arrivando al prodotto AutoESL, divenuto ora Vivado [10].
Il framework segue una struttura generica, partendo da una descrizione dell'applicazione da realizzare in System C o C, arriva ad avere una sua descrizione in

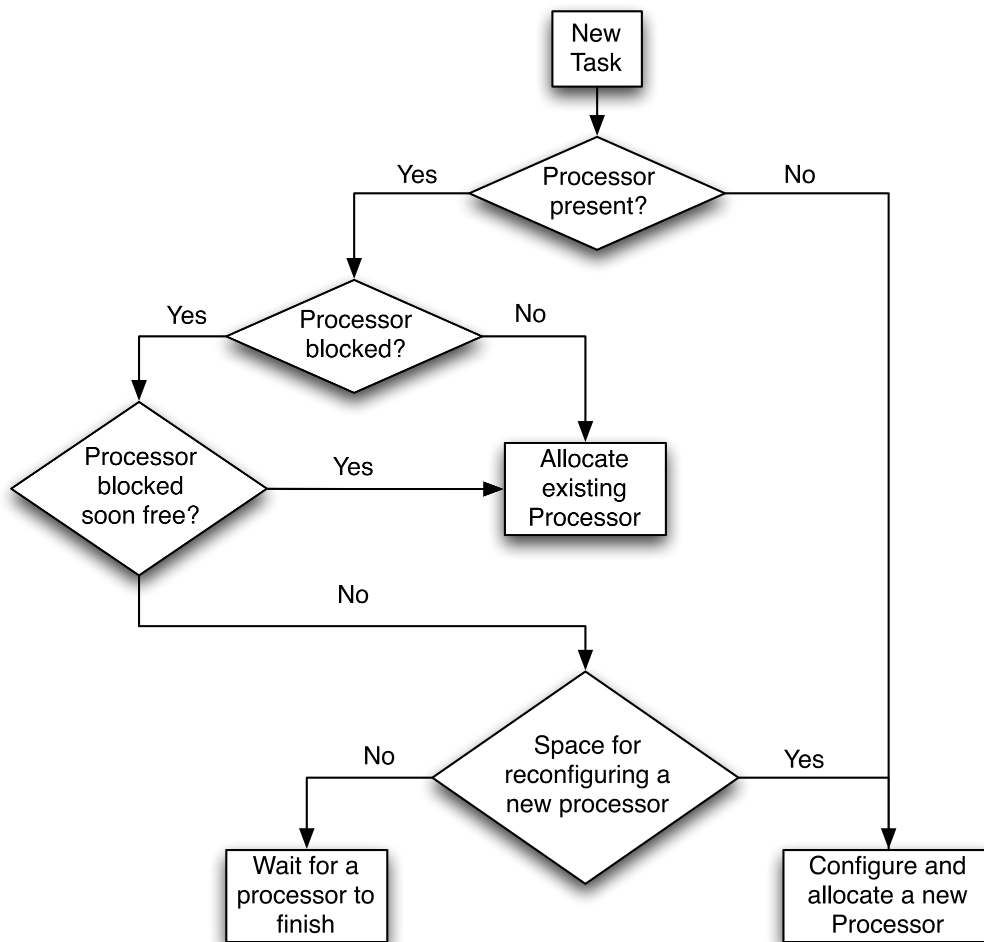


Figura 2.4: CAP-OS [3]

codice HDL sintetizzabile, tale codice è pronto per l'implementazione nel caso di uso su dispositivi hardware.
Il flusso di lavoro seguito dal framework, figura 2.5, prevede come input l'applicazione descritta in C o System C e la descrizione della piattaforma target. Il flusso è diviso nei seguenti passi:

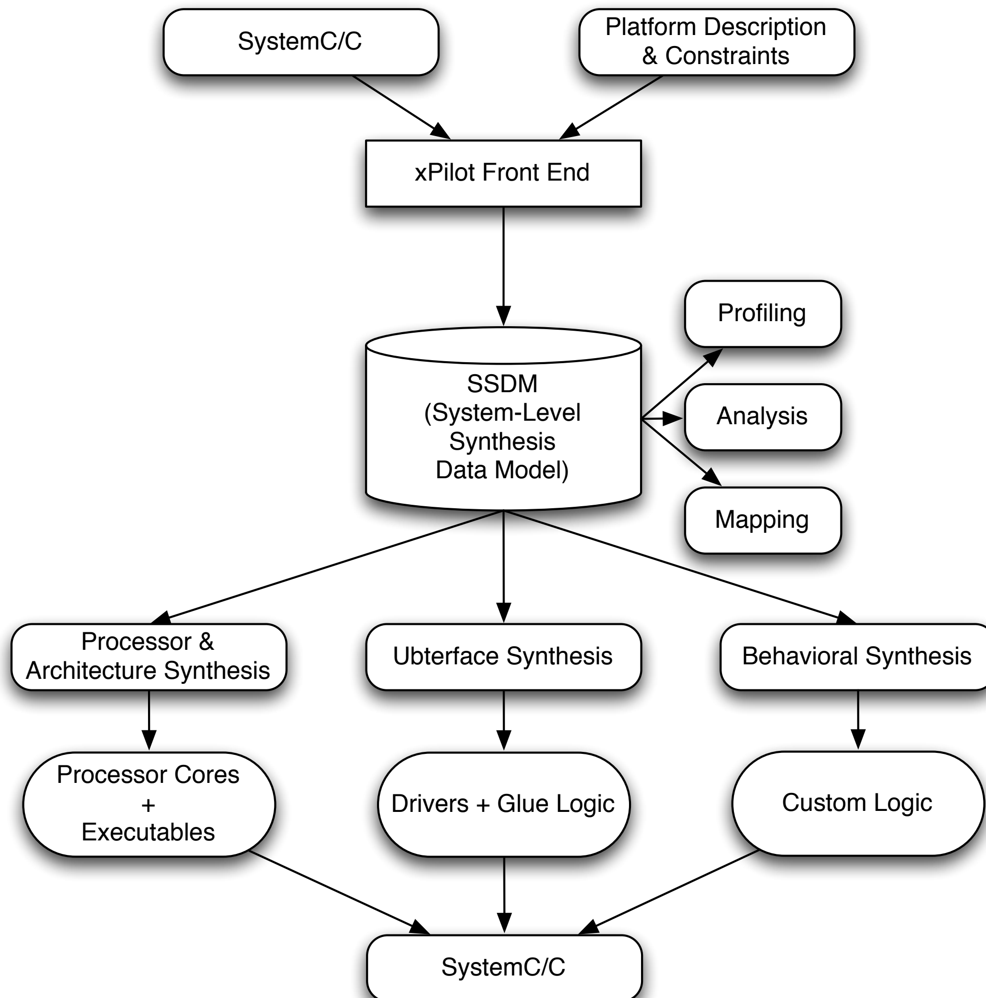


Figura 2.5: XPilot [4]

- Front-End: Analizza il codice dell'applicazione e ne genera una descrizione in forma di grafo (CFG[26], DFG[27]);
- Analisi: in questa fase viene effettuata un'analisi ed una profilazione dell'applicazione e queste informazioni vengono usate per effettuare l'asse-

gnamento delle varie parti dell'applicazione ai processori; le informazioni generate in questa fase vengono salvate in un modello chiamato System-Level Synthesis and Data Model;

- infine vengono generati gli output ed il flusso di lavoro si divide in tre parti indipendenti volte a realizzare:
  - la traduzione del codice per i GPPs nel sistema;
  - la traduzione in HDL delle parti che andranno implementate su hardware dedicato;
  - la generazione delle interfacce e dei drivers per la comunicazione tra i vari componenti.

La parte divenuta commerciale è quella relativa allo sviluppo di componenti hardware dedicati con un'attenzione particolare al problema del HLS. Questa parte è stata affrontata in molti lavori dal gruppo di XPilot [28, 29, 30, 31, 32, 4] ed è stata ulteriormente migliorata da Xilinx per arrivare ad avere non solo una descrizione in Register Transfer Level (RTL), e quindi poco leggibile, dell'applicazione, ma una descrizione behavioral che quindi ne permette un uso molto più flessibile da parte dello sviluppatore.
Per quanto riguarda il prodotto sviluppato da Xilinx, AutoESL (divenuto suite Vivado), la parte di HLS accetta in input un insieme di direttive che esprimono le decisioni dello sviluppatore in relazione all'implementazione dei vari componenti; tali direttive possono riguardare l'assegnamento dei vari task ai diversi processori oppure direttive di traduzione quali ad esempio il livello di loop unrolling desiderato per un particolare ciclo del codice.

2.4 Sommario

In questo capitolo sono stati presentati i principali framework per lo sviluppo di MPSoC presenti nello stato dell'arte. In particolare sono stati presentati strumenti che supportano lo sviluppo di sistemi classici Daedalus e XPilot e strumenti in grado di supportare lo sviluppo di MPSoC riconfigurabili, ovvero RAMP-

SoC. In questo lavoro di tesi è stata sviluppata una toolchain che si differenzia dai tool appena analizzati per le seguenti ragioni: il flusso di lavoro, descritto in dettaglio nel Capitolo 4, si pone a valle delle analisi possibili sull'applicazione e genera l'architettura hardware partendo dalle funzionalità dall'applicazione già ben definite, mentre i lavori citati in questo capitolo hanno una componente di design space exploration che il lavoro svolto non tratta concentradosi sull'integrazione automatica del sistema; rispetto a Daedalus e RAMPSoC viene aggiunta la parte di generazione automatica dei drivers per i core hardware sintetizzati; infine XPilot manca della parte di integrazione automatica degli IP Core nel sistema, la toolchain sviluppata oltre a svolgere il lavoro di integrazione arriva a generare il bitstream.

Capitolo 3

MPSoC e FPGA: Zedboard

Per capire come è stato raggiunto l'obiettivo prefissato è essenziale avere una conoscenza approfondita della piattaforma utilizzata, ovvero è necessario possedere almeno una conoscenza di base di come funzionano i protocolli hardware standard adottati dal dispositivo e una semplice comprensione di come è strutturata l'infrastruttura di comunicazione dei core hardware. Per presentare al meglio il lavoro svolto, questo capitolo presenta tutte le informazioni appena citate:

- la prima sezione è relativa alla Zedboard, ovvero la scheda di sviluppo usata come target, contiene una descrizione di come è composta l'architettura del sistema ZYNQ [33] e quali sono i protocolli utilizzati per interfacciarsi con questo sistema;
- la seconda sezione mostra come avviene il mapping tra i parametri delle funzioni software scritte in linguaggio C e le interfacce hardware corrispondenti che implementano i protocolli presentati nella prima sezione;
- infine la sezione relativa alle prestazioni mette alla luce quali possono essere i punti critici in tema di performance quando si deve fare uso di un'interfaccia piuttosto che un'altra. Un aspetto fondamentale da tenere in considerazione quando si progetta un'applicazione HW/SW è quello della comunicazione e dello scambio dei dati: in questo tipo di applicazioni nasce un naturale compromesso tra quello che è il guadagno nell'avere un compo-

nente dell'applicazione in hardware e il tempo speso nella comunicazione tra le funzioni, in caso di presenza di un'infrastruttura hardware tale tempo è ovviamente maggiore rispetto al caso puramente software.

3.1 Sistema ZYNQ e protocolli hardware

La zedboard è una scheda di sviluppo prodotta da Xilinx che fa parte della famiglia Zynq.
Le caratteristiche tecniche della scheda sono:

- Dual ARM Cortex-A9 MPCore
- 512 KB L2 Cache
- 512 MB DDR3
- Xilinx Artix-7 FPGA XC7Z020-CLG484-1

La famiglia dei sistemi Zynq è basata sull'architettura Xilinx All Programmable SoC (AP SoC). Questi prodotti integrano un dual-core ARM Cortex-A9 MPCore con un sistema ricco di funzionalità di elaborazione basate su un processing system (PS) e una logica programmabile Xilinx (PL) in un unico dispositivo. La famiglia Zynq è costruita sullo stato dell'arte, con alte prestazioni e bassa potenza con tecnologia di processo a 28 nm. Le cpu dell'ARM Cortex-A9 MPCore sono il cuore della PS che comprende anche una memoria on-chip, interfacce di memoria esterne, e un ricco set di periferiche di I/O.
La famiglia Zynq-7000 offre la flessibilità e la scalabilità di una FPGA, fornendo al contempo le prestazioni, la potenza e la facilità d'uso tipicamente associati ad un ASIC. Ogni dispositivo nella famiglia Zynq-7000 contiene la stessa PS, mentre la PL e le risorse di I/O variano tra i dispositivi. Come risultato, le Zynq-7000 AP SoC sono in grado di servire una vasta gamma di applicazioni, tra cui:

- Automotive
- Broadcast video

- Industria del controllo
- Telecomunicazioni
- Diagnosi mediche e immagini
- Stampanti multifunzione

L'architettura Zynq-7000 mappa la logica personale e il software nella PL e nella PS rispettivamente. L'integrazione della PL con la PS fornisce un livello di prestazioni che due chip separati non possono raggiungere per i limiti dovuti alla banda di I/O e al power budget.
I processori nella PS si avviano sempre per primi, consentendo un approccio software per l'avvio della PL e la sua configurazione. La PL può essere configurata al momento dell'avvio oppure successivamente. In aggiunta, la PL può essere completamente riconfigurata o usata per la riconfigurazione dinamica parziale (PR). La PR consente la riconfigurazione di una porzione di PL mentre il resto della logica programmabile continua a funzionare. Questo permette di ottenere cambiamenti al design, come per esempio aggiornare coefficienti di una funzione o sostituirla con un nuovo algoritmo se necessario. Tale capacità è simile a quella del caricamento e scaricamento dei moduli software. I dati per la configurazione della PL sono contenuti nel bitstream. La Figura 3.1 illustra i blocchi funzionali del Zynq-7000 AP SoC. La PL e la PS sono su due domini di potenza separati, consentendo all'utente di spegnere la PL per risparmiare potenza se richiesto.
Il Zynq-7000 AP SoC è composto principalmente dai seguenti blocchi funzionali:

- Processing System (PS)
  - Application processor unit (APU)
  - Memory interfaces
  - I/O peripherals (IOP)
  - Interconnect
- Programmable Logic (PL)

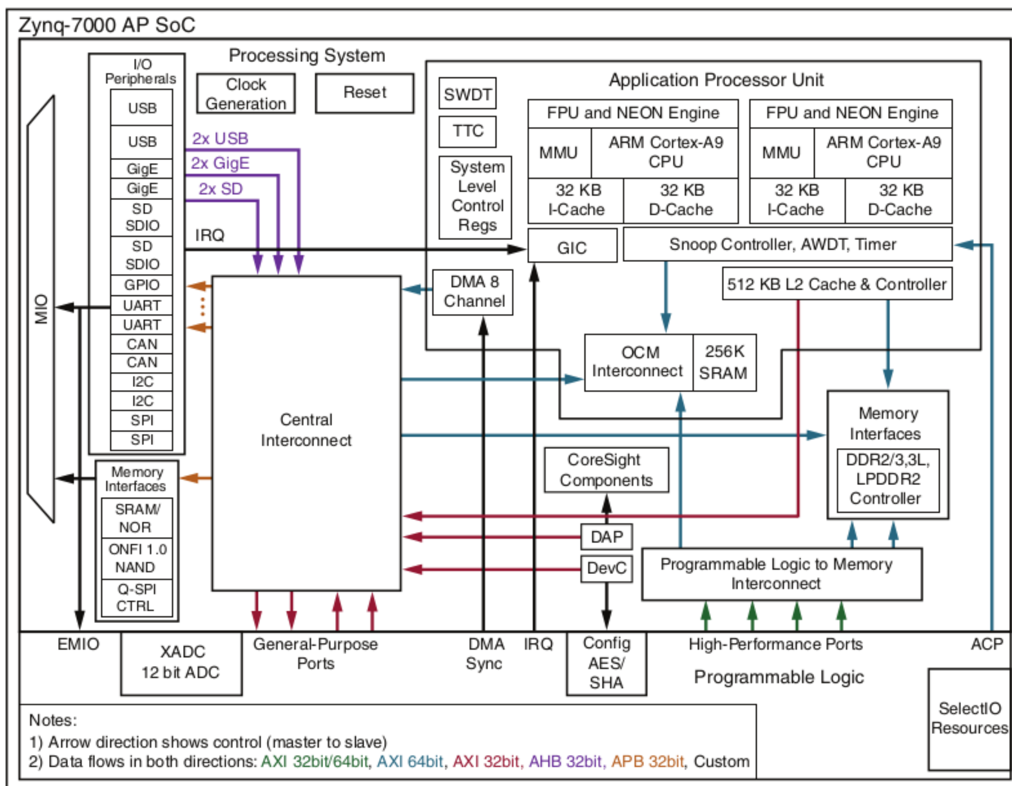


Figura 3.1: Zynq-7000 AP SoC Overview

L'interfaccia PS-PL contiene tutti i segnali disponibili al progettista della PL per integrare le funzioni implementate sulla PL con la PS. Ci sono due tipi di interfacce tra la PL e la PS:

1. interfacce funzionali che includono l'AXI interconnect, interfacce MIO estese per la maggior parte delle periferiche di I/O, interrupts, controlli DMA, clocks e interfacce di debug. Questi segnali possono essere connessi con gli IP creati dall'utente nella PL;

2. segnali di configurazione che includono: processor configuration access port (PCAP), configurazione dello stato e Program/Done/Init. Questi segnali sono connessi ad una logica fissa all'interno del blocco di configurazione della PL, i quali forniscono il controllo della PS.

Tra le interfacce funzionali, per questo lavoro, sono rilevanti quelle sul bus AXI che nello specifico sono:

- AXI_ ACP: una 64-bit cache coherent master port nella PL;
- AXI_ HP: quattro high performance/bandwidth master ports nella PL;
- AXI_ GP: quattro porte general purpose.

Dopo aver visto le interfacce di comunicazione tra i due sistemi verrà affrontato un possibile modo di scambio di dati tra la PS e la PL. Per supportare il progettista nello sviluppo dei core hardware, Xilinx mette a disposizione svariati IP hardware che implementano specifiche funzionalità. Tali IP potranno essere poi in un qualche modo connessi a quelli creati dal progettista. Per avere un modo flessibile e veloce di connessione tra questi IP, tutti gli IP sviluppati da Xilinx, nella versione per la Zedboard, si rifanno ad un unico protocollo hardware, ovvero il protocollo AXI. La PS è dotata di una porta AXI master interfacciata con il bus AXI, che viene direttamente collegato con la PL. Detto ciò, se si vuole scambiare dati con la PS, bisogna implementare nella PL una interfaccia AXI verso questo bus. Per esempio se si implementa una porta slave, quest'ultima potrà ricevere

i comandi dalla porta master collocata sulla PS. Anche il controllore di memoria (DDR) è dotato di un'interfaccia verso questo bus AXI, che quindi può essere controllato anche dagli IP che stanno sulla PL (attraverso una porta master). Tra questi IP forniti da Xilinx, ne esiste uno chiamato AXI DMA, il quale è in grado, se controllato dalla PS, di avere accesso diretto alla memoria e di spostare i dati dalla memoria sulla PL. Nei prossimi paragrafi verrà accennato il funzionamento del protocollo AXI, descrivendo in dettaglio il componente AXI Dma con le sue prestazioni.
Ci sono tre tipi di interfacce AXI:

- AXI4 per requisiti memory-mapped ad alte prestazioni
- AXI4-Lite per comunicazioni memory-mapped semplici a basso throughput (per esempio verso i registri di controllo e di stato)
- AXI4-Stream per streaming dati ad alta velocità

AXI4 fornisce miglioramenti ai prodotti Xilinx e ha portato benefici alla produttività, alla flessibilità e alla disponibilità di nuovi IP.

- Produttività: standardizzando le interfacce sul protocollo AXI, gli sviluppatori devono conoscere un solo protocollo per gli IP
- Flessibilità: fornendo il giusto protocollo per le applicazioni
  - AXI4 per le interfacce memory-mapped, consente burst che vanno fino a 256 cicli di traferimento dati
  - AXI4-Lite per interfacce memory-mapped a singola transizione con una logica di gestione semplificata
  - AXI4-Stream consente un burst di trasferimento illimitato. Le interfacce AXI4-Stream non sono dotate di indirizzo e quindi non sono considerate memory-mapped
- Disponibilità: implementando un protocollo standard il progettista ha a disposizione non solo il catalogo IP di Xilinx ma anche quello della community dei partners ARM

  - Molti fornitori di IP supportano il protocollo AXI
  - Una robusta collezione di tools AXI da parte di vendors di terze parti è disponibile e fornisce tools di sviluppo di sistema, verifica e caratterizzazione delle prestazioni

Fino ad ora sono stati trattati i maggiori benifici della scelta del protocollo AXI, mentre nel resto della sezione verrà presentata una panoramica di come funziona il protocollo AXI.
Per rappresentare lo scambio dati tra due IP core verranno presentate le specifiche AXI di una singola porta master in comunicazione con una singola porta slave. Le interfacce memory-mapped AXI master e AXI slave possono essere connesse tra di loro usando una struttura chiamata blocco di interconnesione. L'AXI Interconnect IP di Xilinx contiene delle interfacce AXI master e slave, conformi al protocollo AXI, che possono essere usate per instradare le transazioni tra due o più AXI master e slave.
Sia le interfacce AXI4 che quelle AXI4-Lite sono formate da cinque differenti canali:

- Read Address Channel
- Write Address Channel
- Read Data Channel
- Write Data Channel
- Write Response Channel

I dati possono viaggiare in entrambe le direzioni (verso la porta master e verso la porta slave) contemporaneamente, e la grandezza di trasferimento può variare. Il limite di un burst per la AXI4 è di una transazione da 256 dati trasferiti, l'AXI4-Lite consente il trasferimento di un dato per transazione. Come mostrato nelle figure 3.2 e 3.3, AXI4 fornisce canali dati separati per le letture e le scritture, consentendo simultaneamente il trasferimento dati bidirezionale. Il protocollo AXI4 contiene una varietà di opzioni che consentono di raggiungere un alto throughput di trasferimento, queste opzioni sono per esempio: il dimensionamento della

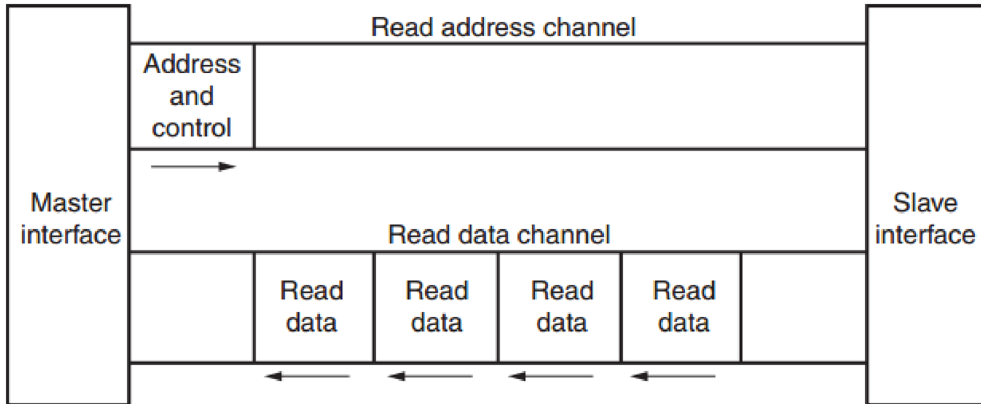


Figura 3.2: Architettura del canale read del protocollo AXI

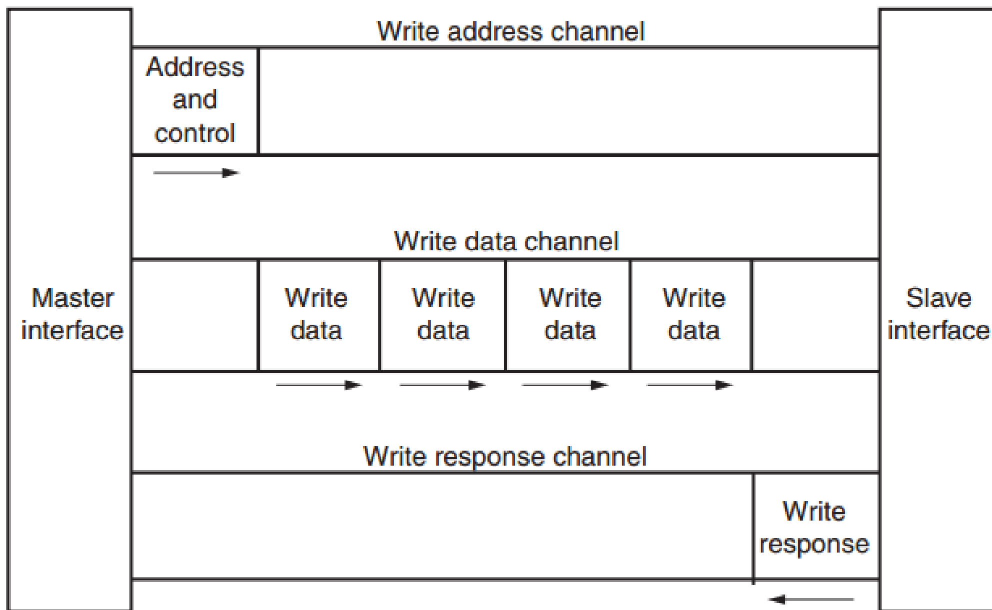


Figura 3.3: Architettura del canale write del protocollo AXI

grandezza dei dati e il processamento delle transazioni in maniera out-of-order.
L'AXI4-Lite è simile all'AXI4 con qualche eccezione, la più importante è che non è supportato un burst con più dati.
Il protocollo AXI4-Stream è composto da un singolo canale per la trasmissione streaming dei dati. Le interfacce AXI4-Stream possono effettuare burst di trasferimento di grandezza illimitata. A differenza di AXI4, i trasferimenti effettuati tramite AXI4-Stream non possono essere riordinati e devono essere eseguiti in ordine. Il protocollo AXI4-Stream è usato da applicazioni che tipicamente sono incentrate su un paradigma data-flow, dove il concetto di indirizzo non è presente o non è richiesto. Ogni cananle AXI4-Stream è singolo ed unidirezionale.
Gli IP basati su AXI4-Stream e quelli basati su AXI memory mapped possono essere combinati insieme, spesso un DMA engine può essere usato per muovere stream di dati sia in entrata che in uscita dalla memoria. Per esempio, un processore può lavorare con un DMA engine per trasferire dati dalla RAM alla FPGA e viceversa.

3.2 Flusso di sviluppo per Zedboard

L'architettura della Zedboard e soprattutto la descrizione delle interfacce hardware adottate da Xilinx rappresentano un punto fondamentale per questo lavoro di tesi, esse costituiscono le conoscenze di base che un progettista di questi sistemi deve avere presente quando sviluppa un'applicazione su tale sistema. Conoscere a fondo i vari protocolli e gli IP presenti che li supportano permette di implementare un sistema finale che sia, non solo corretto e funzionante, ma anche efficiente in termini di tempo speso nello scambio dei dati. Elemento fondamentale di questo lavoro di tesi è quindi l'architettura di comunicazione che attualmente il progettista deve necessariamente creare a mano. L'obiettivo è quello di automatizzare completamente questo processo di generazione accelerando così il tempo di implementazione dell'applicazione. Verranno ora illustrati in dettaglio tutti passi che il progettista del sistema hardware deve eseguire.
Per realizzare un'architettura hardware è necessario prima di tutto avere i co-

re che implementano le funzioni da mappare in hardware. Il progettista deve crearli usando le interfacce corrette, manualmente o tramite il tool di High Level Synthesis (HLS). Una volta che si hanno a disposizione i core bisogna connettere le interfacce di questi core in maniera opportuna. In base al tipo dei parametri in ingresso alle funzioni, si deve utilizzare un protocollo piuttosto che un altro e connettere correttamente la funzione con il resto del sistema. Se la funzione da implementare in hardware ha parametri di tipo primitivo il progettista deve sintetizzare la porta slave del protocollo AXI4-Lite (Figura 3.4). Al fine di

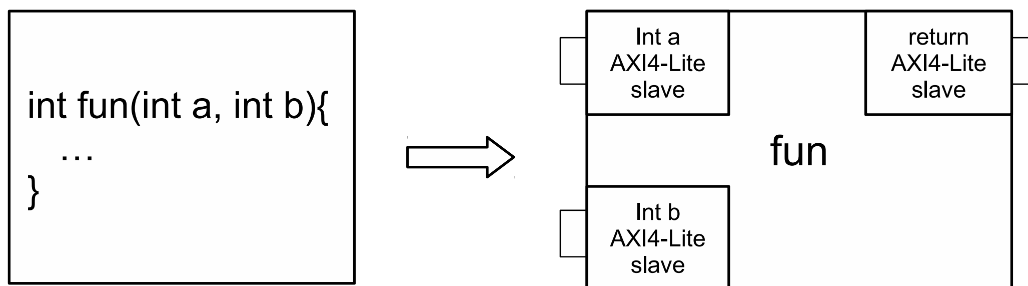


Figura 3.4: Traduzione da funzione a IP con interfaccia AXI4-Lite

implementare una funzione questi sono i passi da seguire:

1. si deve istanziare la funzione hardware implementata e configurare i parametri dell'interfaccia, per esempio assegnare il set di indirizzi che si vuole utilizzare nel caso sia memory mapped

2. bisogna istanziare il componente AXI Interconnect che è in grado di mettere in comunicazione le porte master con le porte slave

3. da un lato dell'Interconnect collegare la porta master del processing system PS e dall'altro la porta slave della funzione realizzata (Figura 3.5).

I punti 1 e 3 vanno ripetuti per ogni core che si interfaccia con il resto del sistema in maniera semplice: parametro di tipo primitivo, ovvero con interfaccia AXI4Lite. L'architettura risultante è quella mostrata in Figura 3.5 dove l'IP Core rappresenta la funzionalità da implementare in hardware.

Automatizzare questo processo può sembrare inizialmente inutile ma, qualora il

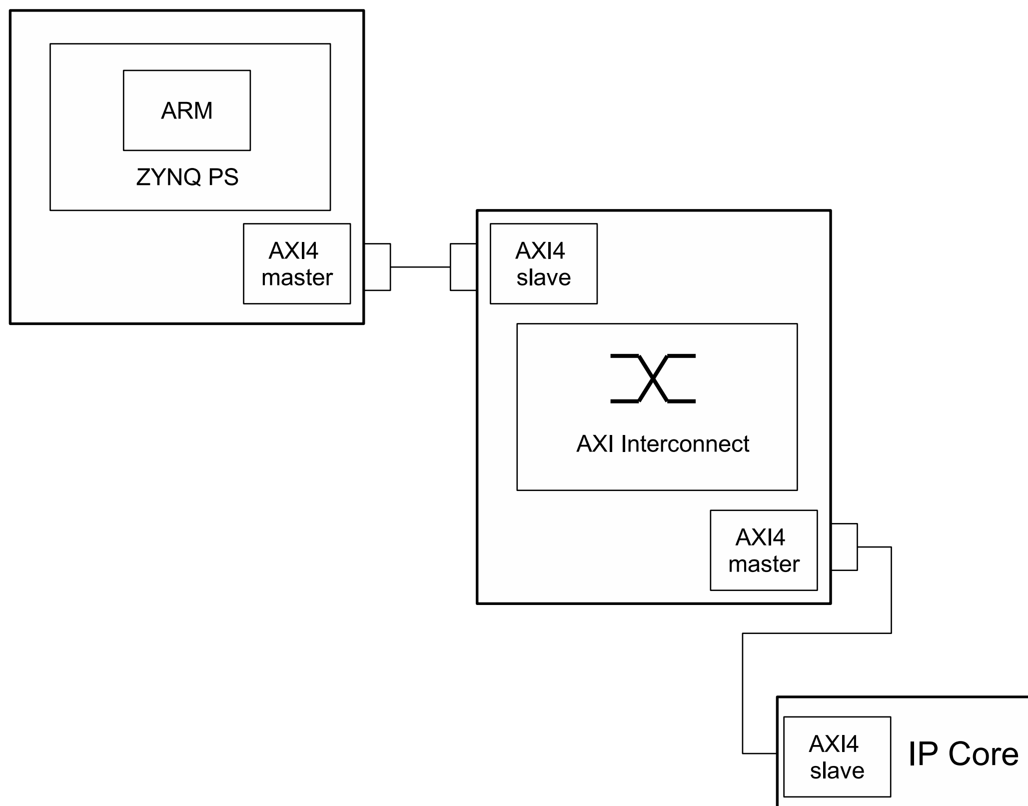


Figura 3.5: Xilinx IP AXI Interconnect

sistema sia formato da decine di funzioni, ci si rende presto conto dell'importanza di avere un sistema automatico per il processo di generazione dell'architettura. Il processo di automazione diventa invece fondamentale quando si vuole implementare una funzione dove il tipo dei parametri non è più primitivo ma è per esempio un array (Figura 3.6). I dati devono fluire dalla RAM alla funzione in

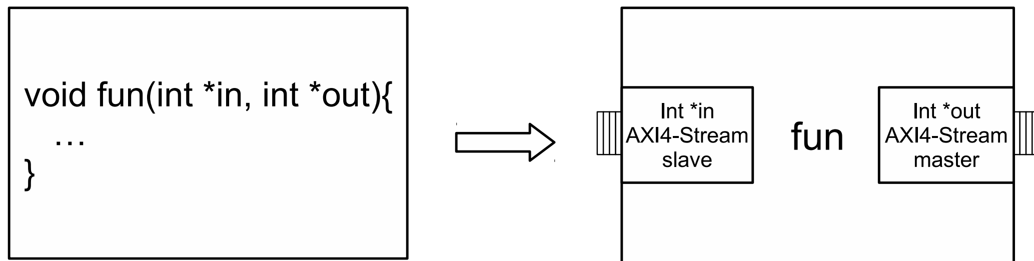


Figura 3.6: Traduzione da funzione a IP con interfaccia AXI4-Stream

maniera stream (Figura 3.7). Per una questione di efficienza è opportuno tenere

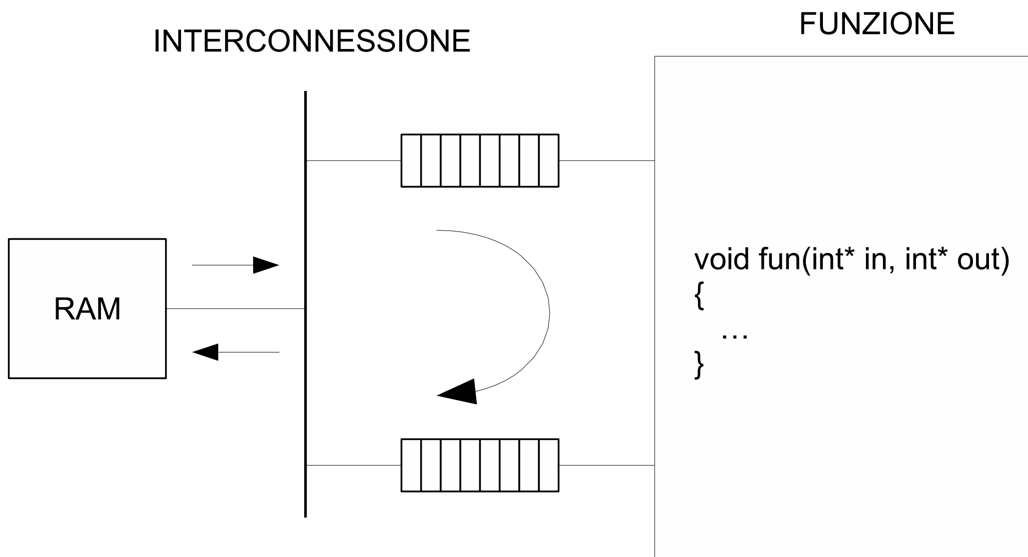


Figura 3.7: Scambio dati tra memoria RAM e funzione stream

libero il più possibile il processore dall'incarico di continuare a trasmettere e occorre convertire i dati dal protocollo memory-mapped a quello stream che non lo è. Esiste l'IP AXI DMA adatto a questo scopo. I passi da eseguire per ottenere l'architettura finale sono:

1. creare ogni funzione hardware che compone l'SDF, ognuna dotata di una interfaccia AXI4-Stream

2. importare nel sistema l'IP AXI DMA che da un lato è dotato dell'interfaccia AXI4-Stream da connettere al nodo di ingresso dell'SDF e dall'altro di una porta AXI4 slave da collegare alla master della parte PS (Figura 3.9)

3. configurare il set di indirizzi assegnati all'AXI DMA

4. importare l'AXI Interconnect per collegare l'AXI4 master della parte PS con la slave dell'AXI DMA

L'architettura realizzata in Figura 3.7, è quella mostrata in Figura 3.8. L'intero processo di generazione dell'architetttura è stato completamente automatizzato fino ad arrivare alla generazione del bitstream.

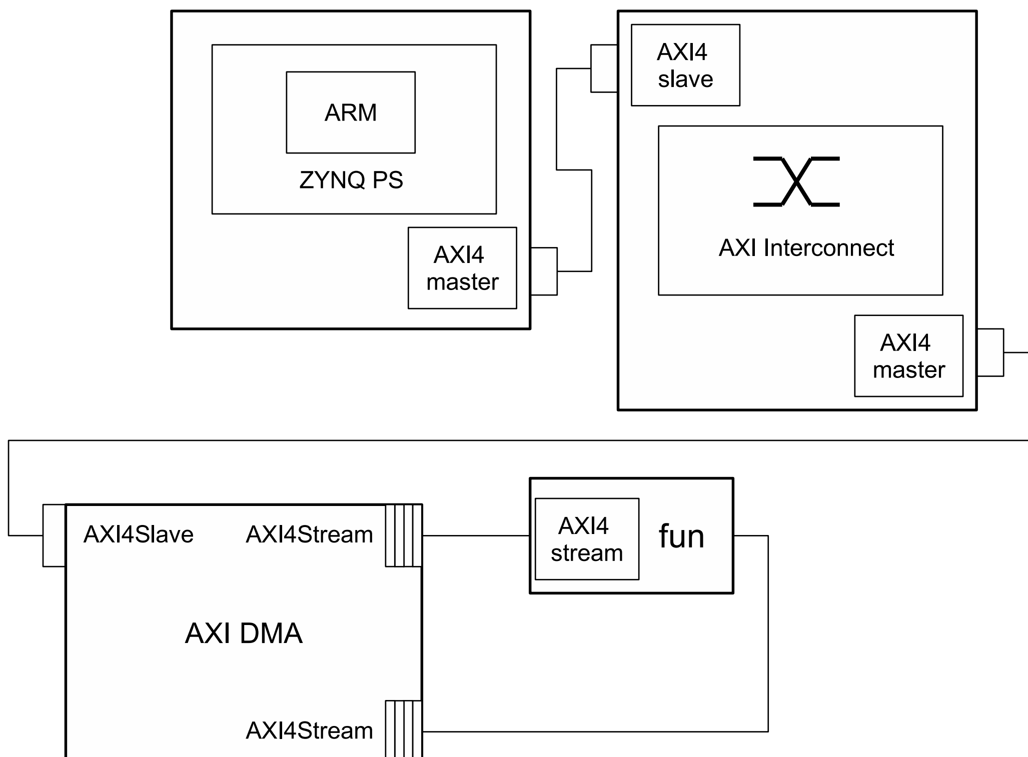


Figura 3.8: Architettura hardware di una funzione stream

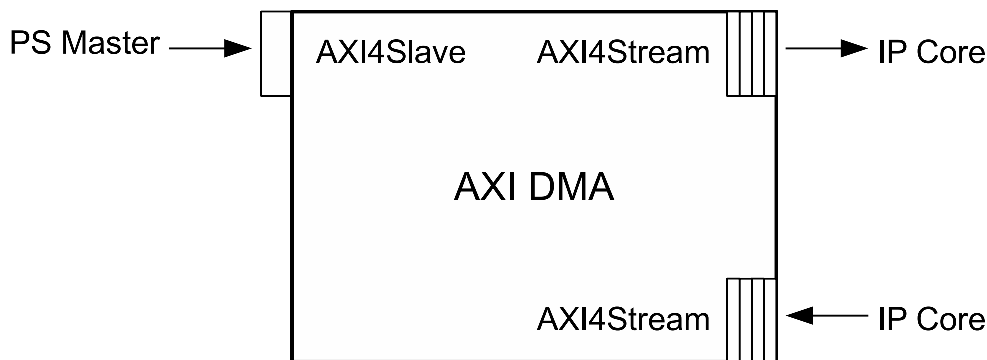


Figura 3.9: Xilinx AXI DMA interfacce

3.3 Comunicazione

Come illustrato nel paragrafo precedente, vi sono sostanzialmente due modi di trasferire i dati dalla memoria RAM alla FPGA. Consideriamo ora un'applicazione, che è in esecuzione sui processori nella parte PS del sistema Zynq, che vuole chiamare una funzione hardware che è stata programmata invece sulla parte PL. Per iniziare l'esecuzione la funzione ha bisogno dei dati che rappresentano i parametri, in base al tipo del parametro è opportuno scegliere un protocollo di comunicazione piuttosto che un altro. Se quest'ultimo è un tipo primitivo (intero, float, double, ecc...) viene salvato su un registro locale alla FPGA e verrà poi utilizzato dall'IP. Scrivere il valore del parametro sul registro è una operazione semplice che può essere effettuata con una singola transazione e il protocollo ottimo per eseguire questa operazione è il protocollo AXI4-Lite. Il registro viene infatti mappato su un porta AXI4-Lite slave per poter essere raggiunto dagli altri IP. Il processore è dotato di una porta AXI4 master che tramite un AXI Interconnect IP viene messa in comunicazione con la porta AXI4-Lite slave del registro. In qualsiasi momento quindi il processore può scrivere un valore sul registro attraverso la sua porta master. Il registro, come mezzo per contenere il parametro di una funzione, non è più sufficiente nel momento cui il parametro non è un tipo primitivo, ma qualcosa di più articolato, come ad esempio una struttura. In questo caso la trasmissione dei dati comprende più trasferimenti all'interno della singola transazione e il protocollo che permette di trasferire tanti dati in maniera

efficiente è il protocollo AXI4-Stream. Tutte le volte che è presente un'applicazione che ha bisogno di streaming di dati, questi vanno portati dalla memoria alla FPGA. L'interfaccia AXI4-Stream non è memory mapped, cioè non ha un indirizzo associato, e quindi non può essere collegata direttamente alla porta AXI4 master del processore che invece lo è. Vi è quindi la necessità di avere un componente che da un lato abbia una porta di tipo AXI4 slave (connessa alla master del processore) con una dello stesso tipo collegata al controllore di memoria, dall'altro invece abbia un'interfaccia AXI4-Stream che può essere collegata con quella stream del core hardware. Il componente con queste caratteristiche si chiama AXI Direct Memory Access (DMA) che effettua la conversione dei dati che stanno in memoria e sono quindi dotati di indirizzo verso dati di tipo stream che invece sono privi di indirizzamento.

3.4 Sommario

La scheda usata per implementare e testare le architetture hardware progettate è la Zedboard, essa è dotata del system on chip chiamato ZYNQ. In questo capitolo sono state presentate le caratteristiche tecniche di tale scheda e sono state mostrate le interfacce e i protocolli hardware di cui è dotato il PS (Processing System) per potersi interfacciare con la parte riconfigurabile della scheda, ovvero la FPGA. Per arrivare ad ottenere un core hardware da una funzione software è necessario generare in maniera appropriata le interfacce hardware che corrispondono ai parametri delle funzioni software, è stato quindi presentato come avviene il mapping tra i parametri delle funzioni e le corrispondenti interfacce hardware.

Capitolo 4

Formulazione del problema

Questo capitolo serve a fornire una descrizione del problema affrontato in questa tesi. Verrà dapprima fornita una descrizione del formalismo utilizzato per rappresentare l'applicazione nel contesto di questo lavoro; poi sarà introdotto il modello di comunicazione supportato ed infine verrà fornita una rapida panoramica dei prerequisiti per la parte relativa alla componente SW di supporto, ovvero drivers e Application Programming Interface (API).

4.1 Definizione del problema trattato

Allo stato attuale la programmazione di sistemi riconfigurabili risulta essere un procedimento lento e complicato, la maggior parte delle funzioni sono svolte a mano dal progettista e in genere richiedono un lungo tempo di sviluppo, la tematica trattata riguarda quindi la semplificazione dello sviluppo di architetture per sistemi programmabili tramite l'automazione del processo di implementazione dell'architettura. I lavori citati nel Capitolo 2 si occupano solo di alcune parti di tutto il processo di sviluppo e lo scopo principale di questa tesi è quello di supportare il progettista nella generazione del sistema.
Dopo la breve introduzione su cosa sono i sistemi riconfigurabili consideriamo le seguenti premesse:

1. data un'applicazione definita secondo una rappresentazione di alto livello;

2. data la suddivisione dei nodi tra nodi che andranno eseguiti in software su un processore e nodi che invece andranno implementati in hardware;

3. data la struttura dell'hardware (MPSoC) su cui implementare l'applicazione.

Viste le premesse appena formulate il problema da risolvere è quello di:

- realizzare automaticamente l'infrastruttura hardware da programmare su FPGA (cioè la creazione dei core che andranno poi programmati sulla FPGA), vedi capitolo 5;

- creare automaticamente tutta l'infrastruttura di comunicazione tra i processori e i core che fisicamente sono implementati nell'area programmabile, vedi capitolo 5;

- al fine di avere un'applicazione in grado di comunicare con la logica programmabile bisogna generare i driver per il sistema operativo Linux, permettendo così alla porzione di software che viene eseguita dal processore di avere accesso ai core hardware, vedi capitoli 5 e 6.

Dal punto di vista del sistema operativo i core sono come delle periferiche in grado di eseguire la funzione per cui sono stati creati. L'applicazione deve poter scambiare dati con queste periferiche e ricevere i risultati della computazione eseguita in hardware su FPGA. Lo scopo del lavoro è quindi quello di realizzare tutto il necessario per avere in maniera automatica un'implementazione dell'applicazione per l' architettura presa in considerazione, facilitando enormemente il lavoro ai progettisti di questi sistemi e riducendo il tempo complessivo di sviluppo delle applicazioni per i sistemi riconfigurabili;
L'architettura MPSoC presa in considerazione in questo lavoro di tesi è la scheda di sviluppo Zedboard prodotta da Xilinx che è stata descritta in dettaglio nel Capitolo 3.
Il resto del capitolo è così strutturato: un primo paragrafo contiene la formulazione della rappresentazione adatta a descrivere l'applicazione; un secondo paragrafo mostra il modello utilizzato per l'infrastruttura di comunicazione ed in-

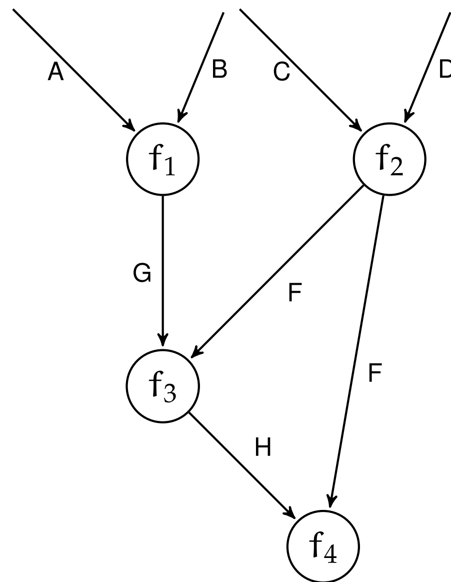
fine è presente un breve riferimento alla funzionalità dei drivers all'interno dei sistemi riconfigurabili.

4.2 Modello dell'applicazione

Al fine di risolvere il problema appena presentato vi è la necessità di avere come punto di partenza l'applicazione che si intende implementare sul sistema programmabile. Analizzare direttamente il codice sorgente risulterebbe inutilmente complicato, l'input ideale da cui partire è una rappresentazione ad alto livello che permette di astrarre dal codice sorgente in cui è scritta l'applicazione. Una rappresentazione che ha avuto molto successo è il modello in forma di taskgraph che è stato combinato a quello di Synchronous Data Flow (SDF) per ottenere il giusto livello di astrazione.
Considerando la seguente definizione di grafo orientato:

Def. 1. *Un grafo orientato G è una coppia* (V, E) *dove V è un insieme non vuoto ed E è una relazione binaria su V,* $E \subseteq VxV$, *cioè un insieme di coppie ordinate di elementi di V.*

Un taskgraph è un grafo orientato (per una trattazione più approfondita sui grafi si rimanda a [34]), i nodi rappresentano le funzioni e i lati le collegano in modo da rispettare l'ordine di esecuzione, infatti una generica applicazione contiene anch'essa funzioni che vengono eseguite in ordine per preservare la semantica del programma.
Per mostrare un esempio di taskgraph consideriamo questa banale applicazione composta da quattro funzioni:
1 - $G = f_1(A, B)$
2 - $F = f_2(C, D)$
3 - $H = f_3(G, F)$
4 - $f_4(H, F)$
Nel taskgraph corrispondente ci sono quattro nodi che rappresentano le funzioni f_i con i lati che rispettano il flusso dell'applicazione.



Ogni parametro della funzione viene tradotto in un arco in ingresso al nodo, tutte volte che il risultato prodotto da una funzione è parametro di un'altra viene tracciato un arco di collegamento tra questi due nodi. Oltre alle caratteristiche già evidenziate, questa rappresentazione è facilmente manipolabile: al taskgraph si possono applicare una serie di trasformazioni che ne modificano la struttura di collegamento, lasciando intatta la semantica del programma. Si può, per esempio, fare l'unione di due nodi (merge) o viceversa un nodo può essere diviso in due nodi diversi (operazione di split). Applicare le trasformazioni al taskgraph è di gran lunga più semplice ed immediato rispetto ad applicarle al codice sorgente, soprattutto se ad eseguirle è un processo automatico.

La nozione di taskgraph utilizzata in questo lavoro di tesi è leggermente più generica di quella appena presentata, ogni nodo può essere infatti o un task (che rappresenta la funzione da eseguire) oppure una fase. Una fase è un sottografo rappresentato come un unico nodo che può essere espanso, dando vita a taskgraph gerarchici (HTG [35, 36]). Una fase, una volta espansa, diventa quindi un grafo chiamato SDF [27] che a differenza del taskgraph rappresenta sempre un'applicazione, ma di tipo diverso: si tratta di applicazioni streaming, dove i dati fluiscono da un nodo all'altro attraverso i nodi/funzioni.

Per capire meglio come funziona un taskgraph gerarchico si osservi la figura 4.1

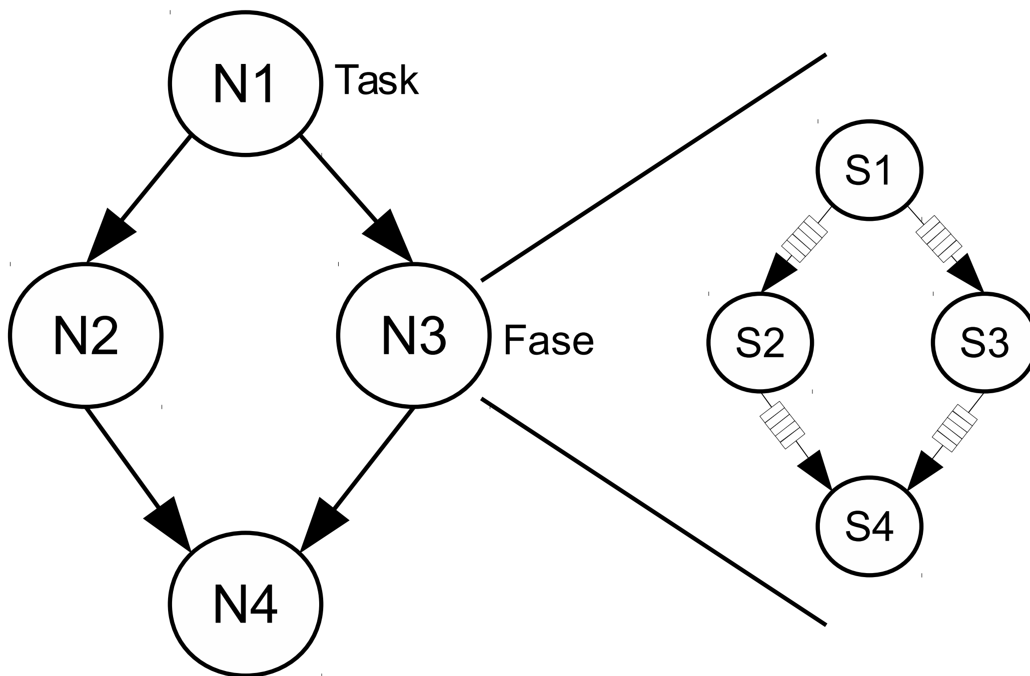


Figura 4.1: Esempio di un taskgraph gerarchico, il nodo N3 viene espanso in un sottografo

in cui il nodo N1, che è effettivamente un task, è collegato al nodo N3, che come si può osservare è una fase poiché contiene un sottografo che è un SDF e i nodi da S1 a S4 sono a loro volta dei task con la caratteristica di processare i dati in maniera streaming.

4.3 Modello e architettura di comunicazione

Nella rappresentazione in forma di taskgraph dell'applicazione risulta chiaro che tra i nodi funzione vi è una dipendenza legata ai dati che devono essere in qualche modo scambiati tra di loro. Nei comuni processori general purpose il meccanismo che permette di scambiare dati tra una funzione software e un' altra è quello di usare la memoria centrale come mezzo di comunicazione, ovvero i dati vengono prima salvati in memoria dal chiamante e sono poi recuperati e utilizzati dalla funzione chiamata. Nel caso del taskgraph sia che i nodi siano parti software sia che siano core hardware il meccanismo di scambio dati non cambia: i dati che servono a un core hardware vengono prima recuperati dalla

memoria e trasferiti alla FPGA tramite il bus, questi vengono utilizzati dai core e quando essi producono a loro volta dei dati per un altro nodo del taskgraph questi ultimi vengono ritrasferiti da FPGA a memoria centrale. Scambiare dati verso la FPGA e viceversa introduce un ritardo dovuto al trasferimento sul bus, questo implica dover raggiungere un compromesso tra il tempo di trasferimento dei dati tra memoria centrale e core e il tempo di esecuzione della funzione in hardware; questo è un problema da considerare in fase di progettazione dell'applicazione. Un' eccezione a questo meccanismo di comunicazione la fanno le fasi nella quale i dati fluiscono all'interno in maniera continua e devono essere prelevati e portati al primo nodo dell' SDF, alla fine poi saranno prelevati dall'ultimo nodo e riportati in memoria, si ottiene così un' infrastruttura hardware dove il primo e l'ultimo nodo sono direttamente collegati al bus. I nodi intermedi non necessitano di essere collegati alla memoria centrale poiché i dati fluiscono direttamente all'interno dei nodi. Infatti i dati devono essere salvati localmente in una coda gestita con logica FIFO (first in first out). Salvare temporaneamente i dati implica l'utilizzo di memoria locale che sulla FPGA è chiamata Block RAM (BRAM). La risorsa BRAM è indispensabile perché permette ai core hardware di accedere velocemente ai dati evitando numerosi trasferimenti sul bus con conseguente diminuzione del tempo totale di computazione dell'intera fase. Data la scarsa quantità di BRAM disponibile sul chip un passo importante nello sviluppo dell'architettura hardware è la giusta allocazione di questa risorsa che è di gran lunga inferiore rispetto a quella della memoria RAM, questa limitazione è dovuta principalmente a problemi di costo e spazio occupato. La quantità di memoria locale disponibile è un altro aspetto da considerare in fase di realizzazione delle funzioni hardware che andranno poi implementate sulla logica programmabile. Un corretto utilizzo di RAM e BRAM permette di raggiungere prestazioni migliori.

L'architettura illustrata nella figura 4.2 corrisponde al taskgraph mostrato nella figura 4.1 dove i nodi N1 ed N4 sono mappati per essere eseguiti in software dal processore mentre i rimanenti nodi sono implementati in hardware.

La figura 4.2 riassume quanto detto sul metodo di scambio dati ipotizzando che

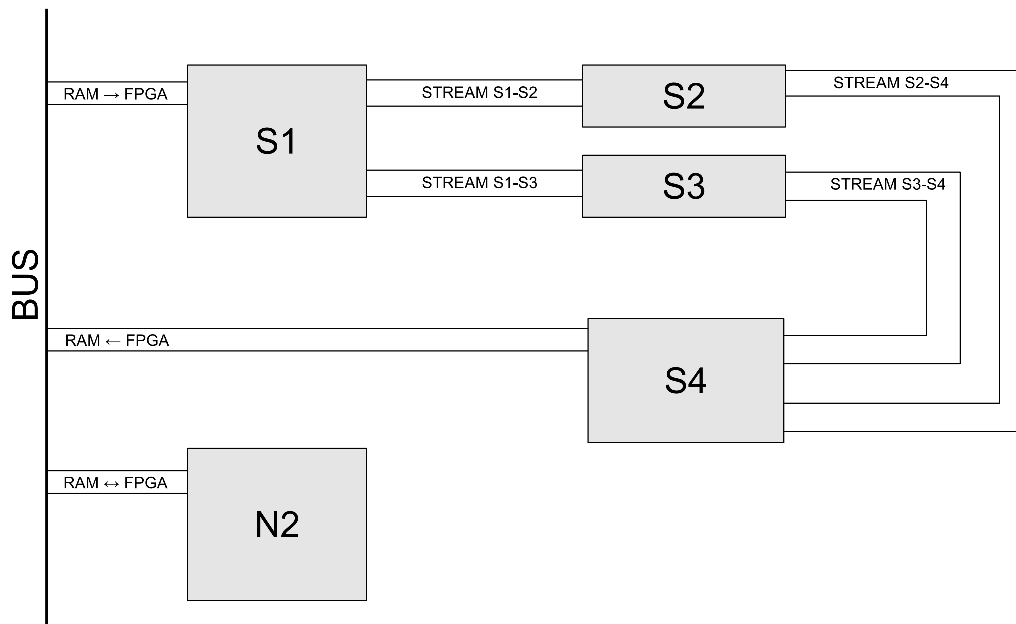


Figura 4.2: Architettura di comunicazione di un taskgraph

le funzioni da implementare in hardware siano quelle che corrispondono ai nodi S1-S4 e N2. Come si può osservare sia il nodo iniziale che quello finale dell' SDF sono connessi direttamente al bus insieme al nodo N2, i rimanenti nodi S2 e S3 ricevono e producono dati tramite un' interfaccia stream.

4.4 Supporto Software

Nella sezione riguardante lo scambio dei dati manca un particolare importante: cioè chi si occupa della gestione di tutta l'infrastruttura di comunicazione. Al fine di scambiare correttamente i dati tra processore e FPGA serve un sistema operativo dotato di uno strato software composto dai driver. Qualora vi sia la presenza di una periferica hardware che cambia di volta in volta in base alla funzione implementata, come nel caso dei sistemi riconfigurabili, il driver deve tenere conto della riconfigurazione e deve essere scritto in modo da gestire tutti i core hardware che si possano creare. La gestione deve avvenire tramite l'utilizzo di un'interfaccia comune. Dalla figura 4.3 è evidenziato il lavoro svolto dai driver che permette all'applicazione l'utilizzo delle funzioni hardware esportando delle

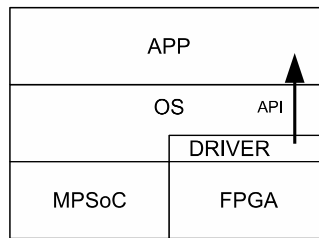


Figura 4.3: Composizione a livelli di un sistema riconfigurabile

API semplici da utilizzare. I dettagli su come funziona la generazione e l'adattabilità dei driver ai tipi di periferiche sono spiegati in maniera approfondita nel Capito 6.

4.5 Sommario

Nella sezione introduttiva del capitolo sono state formulate una serie di premesse e sono stati elencati i punti fondamentali che costituiscono il problema proposto.
Un paragrafo è stato dedicato alla descrizione del modello usato per rappresentare un'applicazione. Il modello dell'applicazione che permette di astrarre dal codice sorgente e facilitare l'implementazione in hardware è quella del rappresentazione sotto forma di taskgraph combinato con il SDF. Dopo aver dato la definizione del concetto di taskgraph usato nell'ambito di questo lavoro di tesi, è stato mostrato come i task e le fasi si scambiano dati tra di loro. Un ulteriore paragrafo è dedicato al modello di comunicazione utilizzato: i nodi task scambiano dati tra di loro utilizzando la memoria centrale come canale di comunicazione, invece le fasi scambiano dati tra di loro servendosi di una coda gestita con logica FIFO. Il concetto di taskgraph è necessario per riuscire a formulare il problema risolto: data un'applicazione rappresentata sotto forma di nodi interconnessi da archi, generare automaticamente la piattaforma hardware provvista di driver per il sistema operativo Linux e le interfacce in linguaggio C per le applicazioni che useranno l'architettura implementata.
Infine il paragrafo 4.4 contiene un breve accenno alla funzionalità dei drivers

all'interno di sistemi riconfigurabili, il sistema operativo può accedere alle periferiche hardware collocate su FPGA (come illustrato in figura 4.3), i drivers fanno quindi da tramite tra l'applicazione software e l'IP Core implementato.

Capitolo 5

Generazione architettura hardware

Per riuscire a generare automaticamente l'intera infrastruttura hardware il software sviluppato fa uso di strumenti proprietari Xilinx, in particolare Vivado e Vivado HLS. Questo capitolo contiene una panoramica su come siano stati utilizzati questi due strumenti per riuscire a fornire un flusso di lavoro completamente automatizzato. Oltre a questi due strumenti utili per generare il bitstream ed effetturare la sintesi ad alto livello vi è anche il compilatore C per ARM, quest'ultimo fornito da Xilinx consente di compilare le proprie applicazioni per il sistema operativo Linux embedded allo scopo di essere eseguite sul processore della scheda. Il capitolo è dedicato alla descrizione di tutto il software sviluppato nell'ambito del lavoro di tesi, verrà presentata la struttura delle classi di cui è composta l'applicazione mostrando tutte le fasi intermedie che costituiscono il flusso di esecuzione della toolchain.

5.1 Definizione del flusso per la creazione dell'architettura HW

Questa sezione illustra, a livello teorico, la procedura che dalla rappresentazione in forma di Taskgraph arriva ad ottenere il bitstream per configurare la

logica programmabile. Il flusso per la creazione dell'architettura è mostrato in Figura 5.1.
Il punto di partenza è quindi il Taskgraph usato per descrivere l'applicazione.

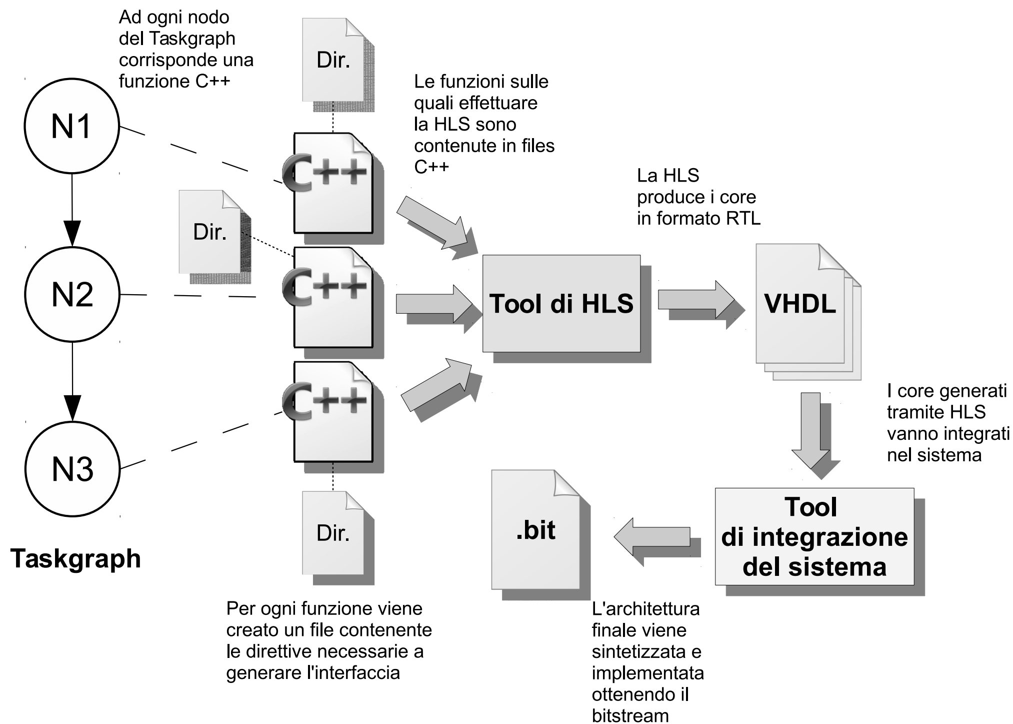


Figura 5.1: Diagramma del flusso di generazione dell'architettura

Come detto ogni nodo del grafo si mappa su una funzione, in questo caso scritta in linguaggio C++, alla quale deve corrispondere un IP Core che realizzi la medesima funzionalità.
Inizialmente è quindi necessario generare, tramite strumenti di sintesi ad alto livello, il Register Transfer Level (RTL) del core per ogni funzione da integrare nel sistema. Oltre alla sintesi della funzionalità vanno generate correttamente anche le interfacce che permettono al core di scambiare dati con il resto del sistema. Al fine di integrare il core nell'architettura, le interfacce da sintetizzare devono implementare un protocollo che dipende dal tipo dei parametri della funzione. I parametri che costituiscono le funzioni sono fondamentalmente di due tipi, ovvero tipo primitivo (int, float, ecc.) o tipo strutturato (array, struct, ecc.), il protocollo varia da un caso all'altro. Una volta terminata la generazione di tutti i core,

questi vanno integrati con il resto del sistema tramite uno strumento in grado di generare il bitsteam dell'architettura. Seguendo la struttura del Taskgraph i core vanno interconnessi mantenendo le dipenze presenti.
Durante il lavoro svolto, vista la piattaforma scelta come target, sono stati presi in considerazione strumenti proprietari Xilinx. Per effettuare il processo di sintesi ad alto livello è stato utilizzato Vivado HLS, invece come strumento per la generazione del bitstream la scelta è ricaduta necessariamente su Vivado. Infine i protocolli adottati sono AXI4Lite per le interfacce composte da parametri di tipo primitivo e AXI4Stream per quelle con parametri di tipo strutturato.

5.2 Implementazione

Nel capitolo 4 è stato presentato il problema da risolvere: si vuole realizzare un processo completamente automatico che, data un'applicazione descritta mediante taskgraph, di cui è stata fatta la co-progettazione hardware/software, generi automaticamente tutta l'infrastruttura hardware da implementare su scheda. Inoltre verrà generato tutto il software necessario per permettere all'applicazione di avere accesso alle funzioni hardware (Driver per Linux con le API lato utente). Successivamente verrà spiegato in dettaglio quale è il punto di partenza, nonchè l'implementazione finale che si vuole ottenere descrivendo l'architettura dell'applicazione sviluppata e come utilizzarla. Il punto di partenza è quindi la porzione di taskgraph delle sole funzionalità che andranno portate in hardware. La rappresentazione mediante taskgraph presentata nel capitolo tre è puramente grafica ed in qualche modo deve essere tradotta in una corrispondente espressione testuale, per poter poi essere data in ingresso all'applicazione, la figura 5.2 ne mostra un esempio che verrà spiegato successivamente.
L'applicazione è stata scritta nel linguaggio di programmazione Scala (Scalable Language), questo perchè tale linguaggio ha la caratteristica di poter realizzare in maniera semplice e veloce un DSL (Domain Specific Language). Questo linguaggio permette quindi di scrivere codice creando una propria sintassi personalizzata che si integri bene con il resto del linguaggio. Quindi per questo lavoro

è stata realizzata una espressione testuale ad hoc che rispecchi il modello grafico del taskgraph. Tale nuova rappresentazione è stata creata in modo da essere incorporata nella sintassi del linguaggio, consentendo la scrittura di un taskgraph in linguaggio SCALA. Di fatto questa realizzazione del taskgraph, per come è stata costruita, permette di essere direttamente eseguita come normale codice SCALA, anzichè essere soggetta ad una analisi mediante parsing, al pari di una qualsiasi altra normale descrizione testuale, come può essere per esempio il formato XML. Questa nuova descrizione è composta da due parti: nella prima parte vi è la lista dei nodi del taskgraph con la lista delle interfacce, ovvero degli archi entranti e degli archi uscenti per ciascun nodo; la seconda parte invece contiene le informazioni su come i nodi sono connessi, cioè elenca gli archi presenti, indicando il nodo sorgente e il nodo di destinazione.
Nell'esempio in Figura 5.2 vi sono due nodi chiamati *mul* e *add* che rappresentano le funzioni aritmetiche di moltiplicazione e somma; nella Figura 5.3 viene mostrato lo stesso taskgraph di esempio ma usando la rappresentazione grafica: in particolare vi sono elencati i due nodi *mul* e *add* connessi tra di loro con a fianco la lista delle interfacce che ogni nodo possiede, in questo caso vi sono tre interfacce per nodo come è evidenziato nella descrizione testuale riportata in figura 5.2. La sintassi della rappresentazione testuale è riassunta dallo schema in figura 5.4 ed è così composta:

- Lista nodi: la sezione si apre con *tg nodes;* e si conclude con *tg end_nodes;* in mezzo vi è un elenco di tutti i nodi del taskgraph, per ogni nodo è presente l'elenco delle interfacce con questa sintassi *tg node* ***nodo*** *i* ***interf_semplice*** *end;*. Dopo il nome del nodo si usa la lettera *i* di interfaccia se quella che segue è un'interfaccia semplice, nel caso il nodo sia composto da interfacce stream si usa la parola **is** che sta per interfaccia stream;

- Lista connessioni: proprio come la sezione precedente questa ha delle marcature di apertura e chiusura descrizione indicate rispettivamente con *tg edges;* e *tg end_edges;*, nel mezzo è contenuta la lista delle connessioni presenti tra i nodi. La sintassi per indicare una connesione è diversa nel caso la connessione sia di tipo semplice piuttosto che stream. Nel caso sempli-

ce lo scambio dei dati avviene tramite l'utilizzo della memoria che funge da mezzo di comunicazione e la sintassi da utilizzare è *tg connect **nodo*** che serve per indicare che tutte le interfacce di tale nodo sono da collegare alla memoria centrale. Per descrivere un canale per lo scambio di dati in maniera stream la sintassi da usare è *tg link (**nodo**, **interf_stream**) to (**nodo**, **interf_stream**) end;*.

Per capire come l'esecuzione di questo codice produca il sistema finale bisogna comprendere come è realizzata l'infrastruttura software dell'applicazione. L'applicazione è composta principalmente dalle seguenti classi:

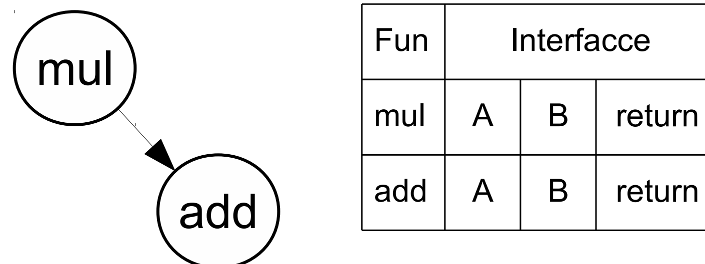
1. TaskGraph
2. Node
3. Interface
4. Link
5. BootZedboard
6. ZedApiGenerator

```
tg nodes;
  tg node "mul" i "A" i "B" i "return" end;
  tg node "add" i "A" i "B" i "return" end;
tg end_nodes;

tg edges;
  tg connect "mul"
  tg connect "add"
tg end_edges;
```

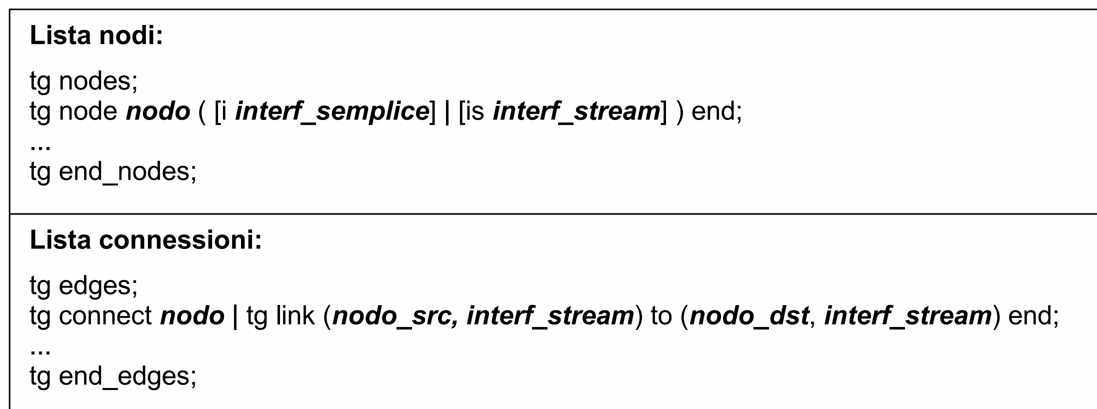
Figura 5.2: Descrizione testuale di un taskgraph

I nomi delle classi sono stati scelti tenendo in considerazione le loro funzionalità e nel complesso devono rispecchiare la rappresentazione in forma di taskgraph formulata precedentemente. Verrà ora presentata in dettaglio la struttura di ogni



Fun	Interfacce		
mul	A	B	return
add	A	B	return

Figura 5.3: Esempio grafico di un taskgraph base



Lista nodi:

tg nodes;
tg node ***nodo*** ([i ***interf_semplice***] **|** [is ***interf_stream***]) end;
...
tg end_nodes;

Lista connessioni:

tg edges;
tg connect ***nodo*** **|** tg link (***nodo_src, interf_stream***) to (***nodo_dst***, ***interf_stream***) end;
...
tg end_edges;

Figura 5.4: Schema sintattito della rappresentazione testuale del taskgraph

classe per arrivare infine a mostrare come complessivamente la toolchain sia in grado di risolvere il problema precedentemente formulato.

1. Classe **Taskgraph**: contiene tutte le informazioni significative per rappresentare la struttura generale di un taskgraph, per esempio consente di risalire alle informazioni riguadanti i nodi. La struttura dei nodi invece è rappresentata dalla classe Node descritta successivamente. Per avere un riferimento ai nodi che costituiscono il taskgraph, la classe Taskgraph mantiene una variabile chiamata *nodeList* contenente una lista di istanze della classe Node. Esattamente come un taskgraph contiene i nodi, la classe Taskgraph tiene traccia delle istanze della classe Node. Oltre alla variabile *nodeList* questa classe contiene alcune funzioni che permettono la corretta gestione del taskgraph, con l'obiettivo di arrivare ad una corretta architettura finale. Prima di mostrare le funzioni principali da cui è composta la classe Taskgraph occorre una premessa, ovvero che la toolchain sviluppata non crea direttamente l'archiettettura progettata ma genera automaticamente il codice tcl che permette di farlo; è possibile quindi eseguire il codice tcl tramite gli strumenti di progettazione di Xilinx per ottenere il bistream finale. Detto questo, la maggioranza delle funzioni di seguito elencate producono, come risultato della loro esecuzione, il codice tcl con le istruzioni da eseguire. Le principali funzioni della classe Taskgraph sono:

    - *nodes*: si occupa di generare un file tcl chiamato *script_sys* con le istruzioni per creare un nuovo progetto in Vivado, istanziare il blocco hardware che contiene il System on Chip (SoC) e preparare il progetto all'importazione del core hardware generati dalla High Level Synthesis (HLS).
    - *node*: viene eseguita per ogni nodo del taskgraph che corrisponde ad una funzione da implementare in hardware. Il codice sorgente di tale funzione, scritta in linguaggio C++, deve essere contenuto in un file che riporta lo stesso nome della funzione. Esiste quindi un file .cpp per ogni nodo del taskgraph. La funzione node crea il tcl, che eseguito

tramite Vivado HLS avvia la sintesi ad alto livello ed ottiene infine l'IP da importare nel progetto. Vivado HLS per arrivare effettuare la sintesi ad alto livello necessita di un altro file chiamato *directives* contenente tutte le informazioni sulle interfacce del core, tale file viene generato da una funzione che appartiene alla classe Node che verrà presentata successivamente. Le ultime due istruzioni della funzione node creano una nuova istanza della classe Node e la aggiungono alla lista dei nodi, tale istanza è pronta ad eseguire le funzioni per creare il file delle direttive con le informazioni sulle interfacce (corrispondono alle funzioni *i* per le interfacce semplici e *is* per quelle stream e che verranno descritte in seguito).

- *end_nodes*: esegue un ciclo for sulla lista dei nodi e all'interno del file *script_sys* crea il codice tcl che importa l'IP precedentemente sintetizzato all'interno del progetto. Un dettaglio fondamentale è che tutte le funzioni citate fino ad ora si limitano alla sola creazione del codice tcl e non sono responsabili della sua esecuzione. Al termine della descrizione delle interfacce un'apposita funzione della classe Node si occuperà di lanciare il codice tcl per sintetizzare l'IP, per quanto riguarda l'esecuzione del file *script_sys* questo viene invocato al termine della descrizione delle interconnessioni dalla funzione end_edges.
- *edges*: questa funzione non contiene istruzioni ed è stata inserita solo per una questione puramente sintattica.
- *connect*: è responsabile della connessione tra l'interfaccia semplice (AXI4Lite) e il SoC. All'interno del file *script_sys* inserisce il codice tcl per connettere tutte le interfacce AXI4Lite dell'IP al bus di sistema.
- *link*: al pari della connect che gestisce le interfacce AXI4Lite questa si occupa dei collegamenti tra il SoC e le interfacce AXI4Stream di un core e della comunicazione stream tra IP diversi. La gestione di dette interfacce avviene generando all'interno del file *script_sys* le apposite istruzioni tcl.

- *end_edges*: una volta terminata la descrizione dei core e delle loro interconnessioni questa funzione ha il compito di lanciare Vivado per eseguire finalmente il codice tcl fino ad ora generato. Al termine della produzione del bitstream lascia il flusso di esecuzione alla classe BootZedboard per la generazione dei file di boot, in modo da avere tutto il necessario per testare l'applicazione su scheda.

2. Classe **Node**: è la classe utilizzata per rappresentare i nodi del taskgraph, ogni nodo è caratterizzato dalla presenza di una o più interfacce di comunicazione, proprio per questo tale classe contiene una variabile chiamata *interfList* con lo scopo di tenere traccia della lista di interfacce presenti (ovvero delle istanze della classe Interface che rappresenta l'interfaccia). A questa classe appartengono tre funzioni principali:

   - funzione *i*: usata solo per descrivere interfacce semplici di tipo AXI4ite, all'interno del file che elenca le direttive viene inserito il codice che descrive il tipo di interfaccia, in particolare il nome che la caratterizza e il tipo di protocollo (AXI4Lite).
   - funzione *is*: in maniera equivalente alla funzione *i* lo scopo di questa funzione è quello di descrivere la presenta di un'interfaccia appartenente al nodo, mentre il metodo precedente si occupa dell'interfaccia AXI4Lite quest'ultimo è responsabile della gestione delle interfacce stream. Nel file delle direttive viene quindi aggiunto il codice per descrivere l'interfaccia di tipo AXI4Stream.
   - funzione *end*: al termine della generazione del file con le direttive, questa funzione contiene le istruzioni per eseguire Vivado HLS ed effettuare la sintesi ad alto livello.

3. Classe **Interface**: non contiene funzioni ma solo attributi che descrivono l'interfaccia, ovvero il nome, il tipo, l'indirizzo base del mapping in memoria e l'insieme di indirizzi occupati.

4. Classe **Link**: l'istanza di questa classe è utile per descrivere il singolo collegamento di tipo stream, non ha attributi rilevanti e contiene la funzione

di nome *to*. Il metodo *to* crea all'interno del file *script_sys* il codice tcl per collegare l'interfaccia al SoC o ad un altro IP Core utilizzando il protocollo AXI4Stream. La funzione *end* all'interno di questa classe è anch'essa inutile ed è presente solo per ragioni di sintassi.

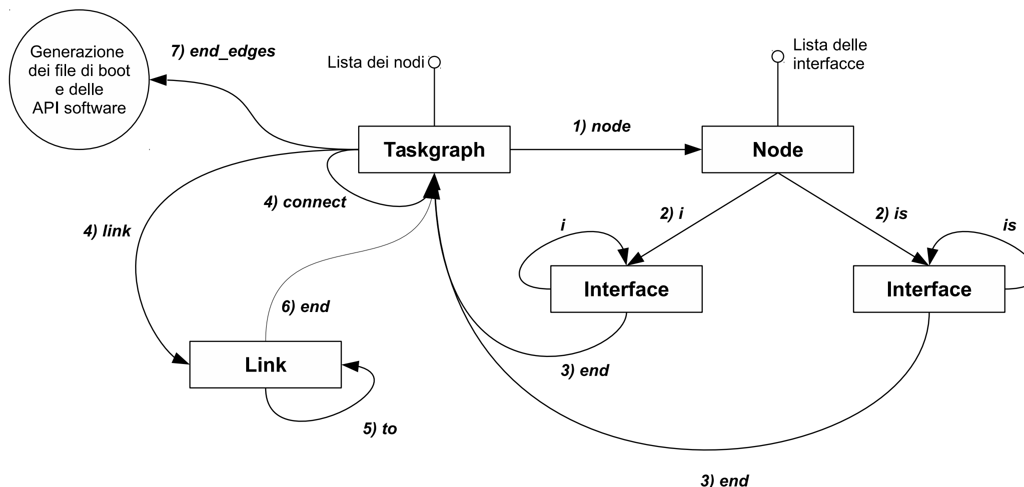
5. Classe **BootZedboard**: non contiene attributi rilevanti ma raccoglie tutti metodi responsabili della generazione dei file di boot che servono per avviare Linux sulla scheda di sviluppo. I metodi presenti in questa classe sono:

    - *devicetree*: in base alla lista dei nodi e alla lista delle interfacce per ogni nodo genera il file di testo contenente il devicetree.
    - *dtc*: in fasi di avvio di Linux il devicetree, oppurtunamente compilato va passato come parametro al kernel che può così associare il driver corretto alle periferiche. Questa funzione compila il file di testo con il devicetree nel formato utilizzato dal kernel, per farlo si serve del tool *dtc* (devicetree compiler) messo a disposizione da Xilinx.
    - *bootgen*: tramite l'uso del tool *bootgen* crea il file *boot.bin* necessario per eseguire il boot della scheda.
    - *toSD*: tutti i file generati vengono riuniti in una cartella chiamata *toSD* pronti per avviare Linux sulla Zedboard.
    - *genZedboardApi*: al termine della generazione dei file di boot, il flusso della toolchain passa alla classe *ZedApiGenerator* invocata da questa funzione.

6. Classe **ZedApiGenerator**: è responsabile della generazione delle Application Programming Interface (API) software e si basa sulla lista dei nodi e delle interfacce, non è presente nessun attributo ma contiene solo le funzioni per la generazione dei file header (.h) e dei sorgenti (.c).

Tenendo presente la struttura di tutte le classi, il flusso di esecuzione della toolchain (illustrato anche in Figura 5.5) è il seguente.
La variabile tg presente nel taskgraph, visibilie anche in figura 5.2, è un'istanza

della classe Taskgraph, invocando il metodo *node* viene creata una nuova istanza della classe Node e questa viene aggiunta alla lista dei nodi. Il controllo passa ora alla classe Node che tramite invocazione dei metodi *i* e *is* genera il file con le direttive che descrivono l'interfaccia. Al termine della gestione del singolo nodo il flusso ritorna alla classe Taskgraph per gestire il nodo successivo. Il controllo passa ciclicamente dalla classe Taskgraph alla classe Node e viceversa per tutta la parte superiore del taskgraph, fino ad esaurire la lista dei nodi. Durante l'esecuzione della seconda parte del taskgraph, in cui si descrivono le connessioni, il flusso di controllo rimane alla classe principale Taskgraph nel caso delle interfacce semplici e si alterna con la classe Link quando vi è la presenza di una connessione stream.



Legenda funzioni:
- Funzione ***node***: crea una nuova istanza della classe Node per ogni nodo del taskgraph e lo aggiunge alla lista dei nodi, questa funzione genera il codice tcl per creare un nuovo progetto Vivado.
- Funzione ***i*** : crea una nuova istanza della classe Interface e la aggiunge alla lista delle interfacce, questa funzione popola il file di direttive con la descrizione dell'interfaccia ed è utilizzata solo per interfacce semplici.
- Funzione ***is*** : crea una nuova istanza della classe Interface e la aggiunge alla lista delle interfacce, questa funzione popola il file di direttive con la descrizione dell'interfaccia ed è utilizzata solo per interfacce stream.
- Funzione ***end*** della classe Interface: lancia Vivado HLS per effettuare la sintesi ad alto livello della funzione C++, il flusso di controllo ritorna alla classe Taskgraph.
- Funzione ***connect*** : genera il tcl per connettere le interfacce semplici (AXI4Lite) di una funzione al SoC.
- Funzione ***link e to***: la funzione link crea una nuova istanza della classe Link che contiene la funzione to. Tutte e due le funzioni generano codice tcl per collegare le interfacce stream tra di loro oppure al SoC.
- Funzione ***end*** della classe Link: restituisce il controllo alla classe Taskgraph.
- Funzione **end_edges** : avvia Vivado eseguendo il tcl generato ottenendo il bitstream dell'architettura. Successivamente lascia il controllo alle classi responsabili della creazione dei file di boot e delle API software.

Figura 5.5: Schema del flusso di esecuzione della toolchain

5.3 Sommario

Il punto di partenza per arrivare ad ottenere una soluzione in hardware dell'applicazione progettata è come detto il taskgraph, vi è quindi la necessità di svilupppare un applicativo in grado di prendere in input tale taskgraph esaminandolo per estrarre tutte le informazioni che descrivono le funzioni.
La rappresentazione in forma di taskgraph presentata nel capitolo 4 è un metodo grafico che descrive le funzionalità dell'applicazione e per essere esaminato da un software deve avere una corrispondente forma testuale. La rappresentazione che probabilmente risulta più immediata e intuitiva è quella che usa il formato XML per descrivere nodi e archi, quest'ultima ha però lo svantaggio che deve essere parsata dall'applicazione per estrarne i contenuti.
Nell'applicazione sviluppata e presentata in questo capitolo viene illustrato un modo alternativo di descrivere un taskgraph in maniera testuale, cioè quello di usare un Domain Specific Language (DSL) sviluppato ad hoc per l'obiettivo che si vuole raggiungere: evitare il parsing del taskgraph eseguendolo direttamente essendo codice eseguibile SCALA. Per ottenere un taskgraph eseguibile è stata mostrata l'infrastruttura software che costituisce l'applicazione: la lista delle classi principali con le relative funzioni.
Per arrivare a generare il bitstream la toolchain fa uso degli strumenti messi a disposizione da Xilinx, ovvero Vivado HLS per generare il VHSIC Hardware Description Language (VHDL) partendo dal software C++ e Vivado per assemblare tutto il sistema formato dagli IP, sintetizzare e generare il bitstream dell'architettura.

Capitolo 6

Generazione supporto software

Questo capitolo contiene i driver sviluppati a supporto dell'applicazione per avere accesso alla piattaforma generata: sia il driver per i core di tipo stream che permette l'accesso al componente AXI DMA da parte di un'applicazione eseguita con i privilegi di utente, sia il driver per le interfacce semplici costituite da registri. Un paragrafo è invece dedicato alla descrizione delle Application Programming Interface (API) generate automaticamente e messe a disposizione del programmatore per consentire l'uso dei driver in maniera semplice ed immediata.

6.1 Definizione del flusso per la generazione delle API

Al termine della sintesi dell'architettura e della generazione del bitstream, la toolchain procede con la generazione delle API per facilitare l'accesso alle funzioni hardware da parte dell'applicazione. Questa sezione contiene le fasi che partendo dalla funzione scritta in C++ arrivano a generazione le intefacce software verso i core.
Il flusso di generazione delle API è mostrato in Figura 6.1.
Al fine di sostituire una computazione software con una hardware, occorre generare una funzione, con prototipo identico a quello della funzione sintetizzata, da sostituire all'interno dell'applicazione. Tale funzione serve ad eseguire le istruzioni necessarie per comunicare con la periferica hardware, lo scopo è quello di

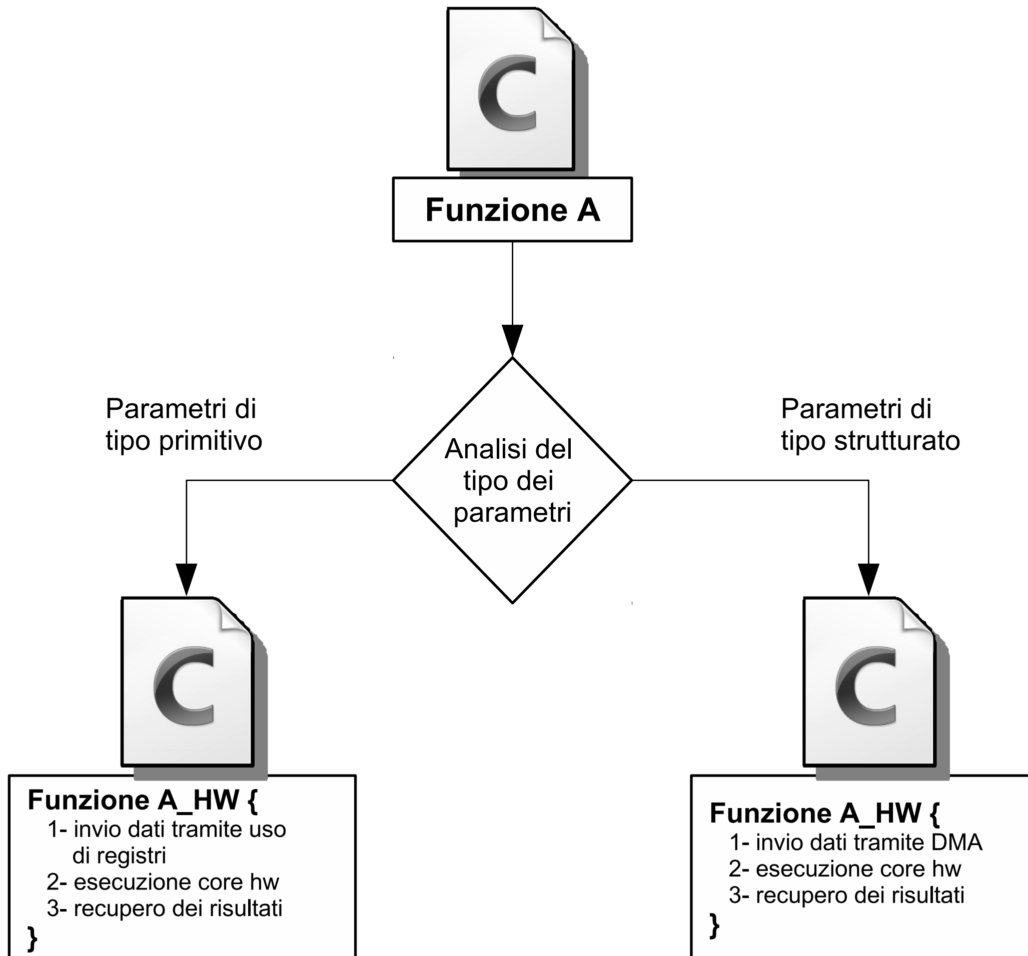


Figura 6.1: Diagramma del flusso di generazione delle API

inviare i dati al core, avviare l'esecuzione hardware corrispondente alla funzione chiamata ed infine recuperare i risultati prodotti e restituirli al chiamante. Così facendo si esonera di fatto il programmatore dalla scrittura del codice di interfacciamento verso il core.
Per le funzioni considerate in questo lavoro di tesi esistono due protocolli di interfaccia differenti, di conseguenza le istruzioni da eseguire si differenziano in base al tipo di protocollo.
Ricapitolando, il flusso per arrivare dalle funzioni C++ alle API è quello di analizzare i parametri delle funzioni, generare una funzione da sostituire a quella sintetizzata contenente le istruzioni per comunicare con il core hardware, portare a termine l'esecuzione hardware e restituire i risultati ottenuti. I dettagli su come è stato implementato questo flusso sono riportati nel Paragrafo 6.2.

6.2 API Software

In questa sezione verrà illustrato come sfruttare i driver per creare le API da rendere disponibili al programmatore che andrà ad utilizzare la piattaforma creata. Come mostrato nel Capitolo 5 durante l'esecuzione del codice Scala che rappresenta il taskgraph viene mantenuta dal software una lista di tutte le funzioni implementate in hardware, al termine della generazione del bitstream, per ogni funzione vengono generati i sorgenti contenenti le API.
Il principio generale è quello di creare una funzione che possa sostituire completamente la funzione software mantenendo quindi gli stessi nomi dei parametri, tale funzione prende il posto di quella software originale e nasconde tutte le istruzioni da eseguire per comunicare con l'IP Core generato. Di seguito sarà ora presentata la struttura dei sorgenti generati sia nel caso di nodi task, quindi con parametri di tipo primitivo, sia nel caso di nodi che rappresentano fasi. Verranno analizzate prima le API per le funzioni con parametri di tipo primitivo.
Per ogni IP Core con interfacce di tipo AXI4Lite Vivado HLS genera le funzioni fondamentali per sfruttare il framework UIO. Per quanto riguarda la generazione automatica delle API per questo tipo di interfacce, il contributo dato da questo

lavoro è la generazione del software dotato di un livello di astrazione maggiore rispetto a quello ottenuto con Vivado HLS. Nei core di tipo memory mapped il parametro fondamentale che permette di utilizzarlo è l'indirizzo base dell'interfaccia slave, per questo motivo Vivado HLS crea una struttura in C contenente due campi: l'indirizzo base della porta slave ed una variabile che indica se il core è in fase di esecuzione o meno, se per esempio la funzione hardware è un sommatore chiamato *add* la struttura risultante sarà la seguente:

```
1 typedef struct {
2     u32 Slv_BaseAddress;
3     u32 IsReady;
4 } XAdd;
```

Il primo passo che un'applicazione deve compiere è quello di istanziare questa struttura, dopo averla dichiarata deve essere opportunamente inizializzata con il valore corretto dell'indirizzo. Nella normale progettazione di un sistema hardware gli indirizzi di BUS sono in genere decisi e assegnati dal progettista. Un modo di inizializzare la struttura è quindi quello di scrivere a mano l'indirizzo conosciuto nel campo *Slv_BaseAddress*, questo approccio ovviamente è sconsigliato in quanto ostacola la generazione automatica del software che decide autonomamente gli indirizzi da assegnare.

Vi è la necessità di ottenere automaticamente l'indirizzo corretto, il valore cercato è contenuto negli attributi esportati dal driver UIO: in particolare nella cartella */sys/class/uio/uioX/maps/map0/*. Il secondo passo da eseguire per inizializzare la struttura *XAdd* è quello di recuperare le informazioni tramite il driver UIO, a farlo ci pensa la funzione *XAdd_Initialize* prodotta da Vivado HLS.

Oltre alla funzione di inzializzazione, Vivado HLS crea le interfacce software che permettono, tramite la tecnica di memory mapping, di settare i registri dei parametri da passare alla funzione. Per semplificare ulteriormente il lavoro al programmatore, tutti questi passi di inizializzazione e passaggio di parametri prima di eseguire la funzione, sono prese in carico dalle API che vengono effettivamente generate dal software sviluppato. Le API risultanti che sono esposte all'applicazione permettono di effettuare una chiamata a funzione hardware nello stesso

modo in cui questa viene effettuata in software. L'obiettivo del supporto software generato è quello di nascondere tutte le complessità che si presentano per effettuare una chiamata in hardware. In particolare nel caso di interfacce AXI4Lite una tipica chiamata a funzione hardware contiene:

- una variabile booleana per inizializzare una sola volta la struttura formata da indirizzo base *Slv_BaseAddress* e variabile *IsReady*
- per ogni parametro presente nel prototipo invoca la corrispettiva funzione che inizializza il valore corretto nei registri prima della chiamata di *start*
- scrive il valore 1 nel registro di *start* iniziando l'esecuzione del core
- controlla in polling la variabile *IsReady* fino a computazione terminata
- raccoglie il valore ritornato dal core e lo restituisce al chiamante

Il risultato finale visto dall'applicazione è quello di invocare la funzione sostitutiva a quella hardware che svolge tutto il lavoro necessario, risulta quindi trasparente al programmatore l'invocazione della funzionalità hardware.
Rimane da analizzare le API generate nel caso di componente hardware funzionante in maniera streaming. Il driver che l'applicazione deve essere in grado di sfruttare in questo caso è quello che gestisce il componente AXI DMA. Il driver svolge la maggior parte delle operazioni necessarie a trasferire i dati da RAM a FPGA, l'interfaccia che il driver espone è semplice ed estremamente intuitiva, ovvero esporta il componente AXI DMA come se fosse un device a caratteri (*/dev/axi_dmaN*). Per la gestione dei componenti streaming come per esempio il core AXI DMA Xilinx non fornisce nessun supporto software, l'implementazione delle API sono completamente lasciate a carico dello sviluppatore. Dove il supporto Xilinx non arriva interviene l'applicazione sviluppata generando le interfacce utili a eseguire una funzionalità hardware che processa dati in streaming come se fosse una normale funzione software. Di fatto, dal punto di vista del programmatore, non vi è nessuna distinzione tra i due tipi di invocazione. Sia muovere dati in streaming che scrivere sui registri sono opportunamente nascosti da una semplice chiamata a funzione, allo stesso modo di una esecuzione in software.

Per ottenere quanto detto anche per le funzionalità con interfacce streaming, le API esposte, nel momento in cui avviene l'invocazione software, nascondono la seguente procedura:

1. eseguire la funzione di *open* sul device corrispondente alla funzione hardware
2. chiamare la funzione *write* per far arrivare i dati sulla FPGA
3. attendere l'esecuzione hardware
4. effettuare una *read* per riportare i risultati in RAM
5. ritornare i risultati alla funzione chiamante

Il punto 1 della lista precedente necessita di un chiarimento, in generale i core che processano dati in streaming sono più di uno e vengono collegati a catena, ognuno esegue la funzione per cui è stato creato e componendoli insieme costituiscono l'intera chiamata a funzione hardware. Il punto fondamentale per avere un'interfaccia software che estrae completamente i dettagli implementativi è la presenza di un unico punto di ingresso dei dati nella catena dei core streaming, e di conseguenza un unico flusso di dati in uscita. I due flussi di dati in ingresso e in uscita sono opportunamente collegati al canale MM2S e S2MM del componente AXI DMA, responsabile del trasferimento verso la memoria. Non è esclusa ovviamente la presenza di più catene di core, in tal caso ognuna verrà collegata al proprio componente AXI DMA.
Per quanto riguarda il driver per il componente AXI DMA, usato dall'applicazione per mandare i dati ai core con interfaccia AXI4Stream, il modulo kernel presente all'interno della distribuzione Linux di Xilinx è risultato non funzionante. Di conseguenza durante il lavoro di tesi si è resa necessaria la scrittura di un driver in sostituzione a quello fornito da Xilinx. Il driver chiamato *ds_axi_dma* è disponibile online a questo link [37] e i dettagli implementativi sono contenuti nel Paragrafo 6.3.

6.3 AXI DMA driver

Per consentire alle applicazioni un accesso al componente AXI DMA in maniera più semplice possibile l'interfaccia utilizzata è quella dei device a caratteri di Linux. Esporre il DMA come se fosse un device a caratteri (contenuto in */dev*) permette l'utilizzo delle librerie standard di accesso ai file, questo semplifica enormemente il lavoro svolto dalle applicazioni lato utente. Nello specifico nel momento in cui si vuole scambiare dati dalla memoria alla FPGA i passi da eseguire diventano:

1. aprire il device tramite le usuali interfacce a file di linux, utilizzando per esempio la *open* sul percorso */dev/axi-dma0*.
2. per trasmettere da RAM a FPGA si utilizza la comune *write*
3. per trasmettere in direzione opposta alla *write*, cioè da FPGA a RAM, si usa la funzione *read*
4. conclusa l'esecuzione dell'applicazione rilasciare il device tramite la *close*

L'AXI DMA driver è stato scritto per supportare la presenta simultanea di più DMA.
L'IP Core AXI DMA è pensato per scambiare dati con la memoria in maniera autonoma ovvero senza l'intervento del processore. L'uso di tale componente fa sorgere problemi di coerenza dovuti alla presenza della cache.
Il problema della coerenza della cache si presenta nel momento in cui i dati vengono recuperati dalla FPGA e salvati in memoria RAM. Il processore manda il comando al DMA per iniziare il trasferimento e i dati vengono così salvati direttamente in RAM, a questo punto la copia in cache diventa obsoleta e deve essere invalidata (figura 6.2). Per risolvere il problema della coerenza dei dati esiste una funzione chiamata *dma_zalloc_coherent* che si occupa di chiedere al kernel un buffer di memoria il cui contenuto è sempre coerente con la cache. Oltre a mantenere la coerenza la funzione *dma_zalloc_coherent* consente di associare l'indirizzo fisico reale del buffer con l'indirizzo virtuale usato dal driver. Nei registri

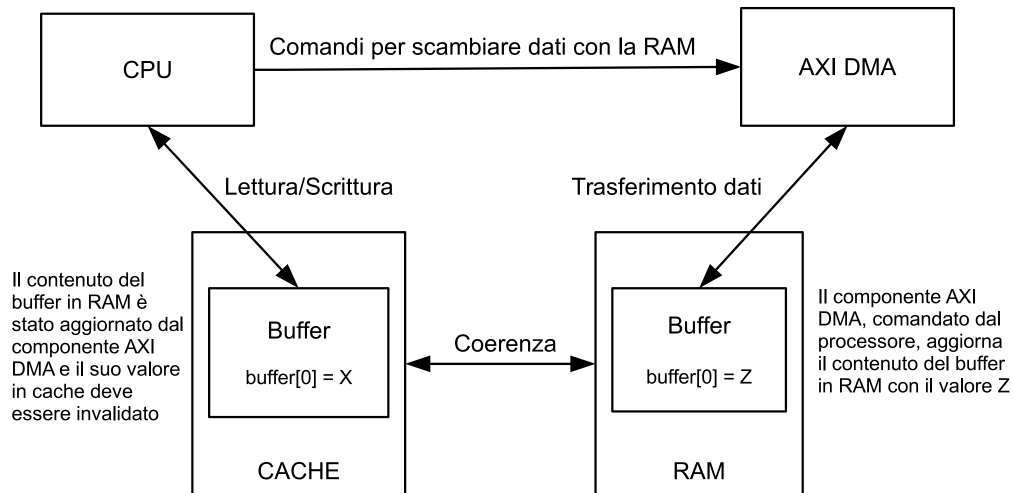


Figura 6.2: Coerenza tra cache e RAM con trasferimenti tramite AXI DMA

del DMA verrà di conseguenza scritto l'indirizzo fisico che corrisponde a quello sorgente/destinazione del trasferimento in memoria, mentre quello ritornato dalla funzione *dma_zalloc_coherent* è il corrispondente indirizzo virtuale.
Per completare la descrizione del driver AXI DMA rimane da verificare l'implementazione delle funzioni open, write, read e close. Al momento della chiamata della funzione open da parte del programma applicativo viene invocata una chiamata di sistema che manda in esecuzione la funzione *ds_axidma_open*, questo avviene perché tale funzione è stata registrata dal driver per questo scopo. La *ds_axidma_open* è composta sostanzialmente da tre funzioni principali:

1. *get_elem_from_list_by_inode*: in base all'inode del device a cui è associata la open recupera dalla lista dei device la struttura *ds_axidma_device* corrispondente precedentemente salvata in fase di probe.

2. *request_mem_region*: chiede al sistema operativo di riservare un particolare spazio di indirizzi di memoria, dato dall'indirizzo base e dalla dimensione. Questo avviene dopo aver fatto un controllo di quell'aria di memoria per verificarne la disponibilità.

3. *ioremap_nocache*: così come avviene per il buffer DMA l'associazione tra virtuale e fisico va eseguita anche per gli indirizzi che rappresentano i re-

gistri di controllo dell'AXI DMA, questa funzione consente di associare l'indirizzo virtuale *virt_bus_addr* all'indirizzo fisico *bus_addr*.

Tutto quanto allocato nella funzione di open deve essere deallocato in quella di close che esegue le istruzioni inverse:

1. trova l'elemento nella lista dei device
2. *iounmap* funzione inversa della *ioremap_nocache*
3. *release_mem_region* opposta rispetto alla *request_mem_region*

Le ultime due funzioni rappresentano la parte essenziale del driver perché sono quelle che inviano i comandi al componente hardware AXI DMA. Al fine di controllare il funzionamento del core scrivono dei valori ben precisi nei registri di controllo del componente. I valori da scrivere per far funzionare correttamente l'AXI DMA sono riportati nella guida utente che Xilinx mette a disposizione per ogni IP proprietario [38]. La funzione *ds_axidma_write* è così composta:

1. recuperare la struttura dati del device corretto (*get_elem_from_list_by_inode*)
2. i dati forniti dall'applicazione utente da trasferire sulla FPGA sono passati alla funzione di write e questi andranno copiati nel buffer DMA allocato precedentemente
3. lanciare il trasferimento dei dati dal buffer DMA a FPGA settando i registri di controllo tramite la funzione *iowrite32*, tale funzione scrive il valore passato come primo parametro all'indirizzo del registro contenuto nel secondo
4. attendere la fine del trasferimento chiamando la funzione *dmaSynchMM2S*

In maniera duale alla *ds_axidma_write* la funzione *ds_axidma_read* deve trasferire i risultati della computazione avvenuta in hardware nella memoria RAM in modo da renderli disponibili al resto dell'applicazione software. Le istruzioni eseguite dalla read sono:

1. recuperare la struttura dati dalla lista dei device come avviene in qualunque operazione eseguita dal driver
2. lanciare il trasferimento dalla FPGA verso la RAM
3. attendere la fine del trasferimento, in questo caso l'operazione *dmaSynchS2MM* è strettamente necessaria per avere i dati pronti da passare all'applicazione utente. Nel caso invece della write l'esecuzione potrebbe proseguire in quanto il trasferimento avviene in hardware in parallelo e non vi è la necessità di avere dei dati pronti
4. infine copiare i dati dal buffer DMA al buffer utente

Per completezza si riporta di seguito il codice sorgente delle operazioni appena descritte.

```
1  static struct ds_axidma_device *get_elem_from_list_by_inode(struct
       inode *i)
2  {
3      struct list_head *pos;
4      struct ds_axidma_device *obj_dev = NULL;
5      list_for_each( pos, &full_dev_list ) {
6          struct ds_axidma_device *tmp;
7          tmp = list_entry( pos, struct ds_axidma_device, dev_list );
8          if (tmp->dev_num == i->i_rdev)
9          {
10             obj_dev = tmp;
11             break;
12         }
13     }
14     return obj_dev;
15 }
16
17 static int dmaSynchMM2S(struct ds_axidma_device *obj_dev){
18     unsigned int mm2s_status = ioread32(obj_dev->virt_bus_addr +
           MM2S_DMASR);
19     while(!(mm2s_status & 1<<12) || !(mm2s_status & 1<<1) ){
20         mm2s_status = ioread32(obj_dev->virt_bus_addr + MM2S_DMASR);
21
22     }
```

```
23	    return 0;
24	}
25
26	static int dmaSynchS2MM(struct ds_axidma_device *obj_dev){
27	    unsigned int s2mm_status = ioread32(obj_dev->virt_bus_addr +
	        S2MM_DMASR);
28	    while(!(s2mm_status & 1<<12) || !(s2mm_status & 1<<1)){
29	        s2mm_status = ioread32(obj_dev->virt_bus_addr + S2MM_DMASR);
30	    }
31	    return 0;
32	}
33
34	static int ds_axidma_open(struct inode *i, struct file *f)
35	{
36	    /* printk(KERN_INFO "<%s> file: open()\n", MODULE_NAME); */
37	    struct ds_axidma_device *obj_dev = get_elem_from_list_by_inode(i);
38	    if (check_mem_region(obj_dev->bus_addr, obj_dev->bus_size))
39	    {
40	        return -1;
41	    }
42	    request_mem_region(obj_dev->bus_addr, obj_dev->bus_size,
	        MODULE_NAME);
43	    obj_dev->virt_bus_addr = (char *)
	        ioremap_nocache(obj_dev->bus_addr, obj_dev->bus_size);
44	    return 0;
45	}
46
47	static int ds_axidma_close(struct inode *i, struct file *f)
48	{
49	    /* printk(KERN_INFO "<%s> file: close()\n", MODULE_NAME); */
50	    struct ds_axidma_device *obj_dev = get_elem_from_list_by_inode(i);
51	    iounmap(obj_dev->virt_bus_addr);
52	    release_mem_region(obj_dev->bus_addr, obj_dev->bus_size);
53	    return 0;
54	}
55
56	static ssize_t ds_axidma_read(struct file *f, char __user * buf, size_t
57	            len, loff_t * off)
58	{
59	    /* printk(KERN_INFO "<%s> file: read()\n", MODULE_NAME); */
```

```
60      struct ds_axidma_device *obj_dev;
61      if (len >= DMA_LENGTH)
62      {
63          return 0;
64      }
65      obj_dev = get_elem_from_list_by_inode(f->f_inode);
66      iowrite32(1, obj_dev->virt_bus_addr + S2MM_DMACR);
67      iowrite32(obj_dev->ds_axidma_handle, obj_dev->virt_bus_addr +
            S2MM_DA);
68      iowrite32(len, obj_dev->virt_bus_addr + S2MM_LENGTH);
69      dmaSynchS2MM(obj_dev);
70      memcpy(buf, obj_dev->ds_axidma_addr, len);
71      return len;
72  }
73
74  static ssize_t ds_axidma_write(struct file *f, const char __user * buf,
75                size_t len, loff_t * off)
76  {
77      /* printk(KERN_INFO "<%s> file: write()\n", MODULE_NAME); */
78      struct ds_axidma_device *obj_dev;
79      if (len >= DMA_LENGTH)
80      {
81          return 0;
82      }
83      obj_dev = get_elem_from_list_by_inode(f->f_inode);
84      memcpy(obj_dev->ds_axidma_addr, buf, len);
85      iowrite32(1, obj_dev->virt_bus_addr + MM2S_DMACR);
86      iowrite32(obj_dev->ds_axidma_handle, obj_dev->virt_bus_addr +
            MM2S_SA);
87      iowrite32(len, obj_dev->virt_bus_addr + MM2S_LENGTH);
88      dmaSynchMM2S(obj_dev);
89
90      return len;
91  }
```

In questo paragrafo è stato illustrato nel dettaglio il codice sorgente del driver in grado di far fluire i dati in streaming attraverso una nodo del taskgraph che rappresenta una fase. Come si può notare da quanto descritto, questo è un driver che è generico per qualunque AXI DMA e non dipende dalla particolare imple-

mentazione della funzionalità hardware che lo usa, al tempo stesso supporta la presenza simultanea di più DMA contemporaneamente. Essendo generico e indipendente dal resto del sistema non vi è la necessità di crearlo automaticamente ad ogni piattaforma hardware generata. Risulta invece strettamente dipendente dall'architettura implementata il file contenente l'albero dei device su cui il driver si appoggia. L'albero dei device con la lista dei DMA collegati al sistema deve necessariamente essere generato automaticamente.
Il componente AXI DMA è dotato di due AXI4-FIFO che fungono da buffer e salvano temporaneamente i dati trasferiti. Il DMA è quindi un componente fondamentale per le applicazioni, le sue prestazioni in termini di tempo speso e velocità nel trasferimento possono impattare enormemente sul tempo di esecuzione dell'applicazione che ne fa uso. Proprio per questo motivo è utile illustrare le prestazioni di tale componente.
Le prestazioni che riguardano il trasferimento di dati tra la memoria DDR e la logica programmabile sono state verificate attraverso un test. Il test è stato eseguito copiando un crescente numero di dati fino a 40 MB di blocchi. La dimensione del singolo burst di trasferimento è di 4KB che è la dimensione della FIFO configurata al momento del test. I dati sono stati trasferiti in parallelo usando fino a un massimo di 4 DMA. Per ogni test è stato misurato il tempo necessario per il trasferimento ed è stata così calcolata la banda in MB/s. Per ogni test è stata presa la media di dieci misure effettuate. La figura 6.3 contiene i risultati ottenuti durante i test e mostra come i trasferimenti effettuati con il DMA siano migliori di quelli ottenuti con il processore ARM quando si trasferiscono più di 40KB di dati.

6.4 Sommario

Nel capitolo corrente è stato presentato il lavoro svolto in termini di software generato a supporto del programmatore, a valle del processo di generazione delle interfacce software scrivere l'applicazione per l'architettura generata risulta estremamente semplice e intuitivo. L'obiettivo finale raggiunto è quello di nascondere completamente i dettagli implementativi del portare una funzionalità

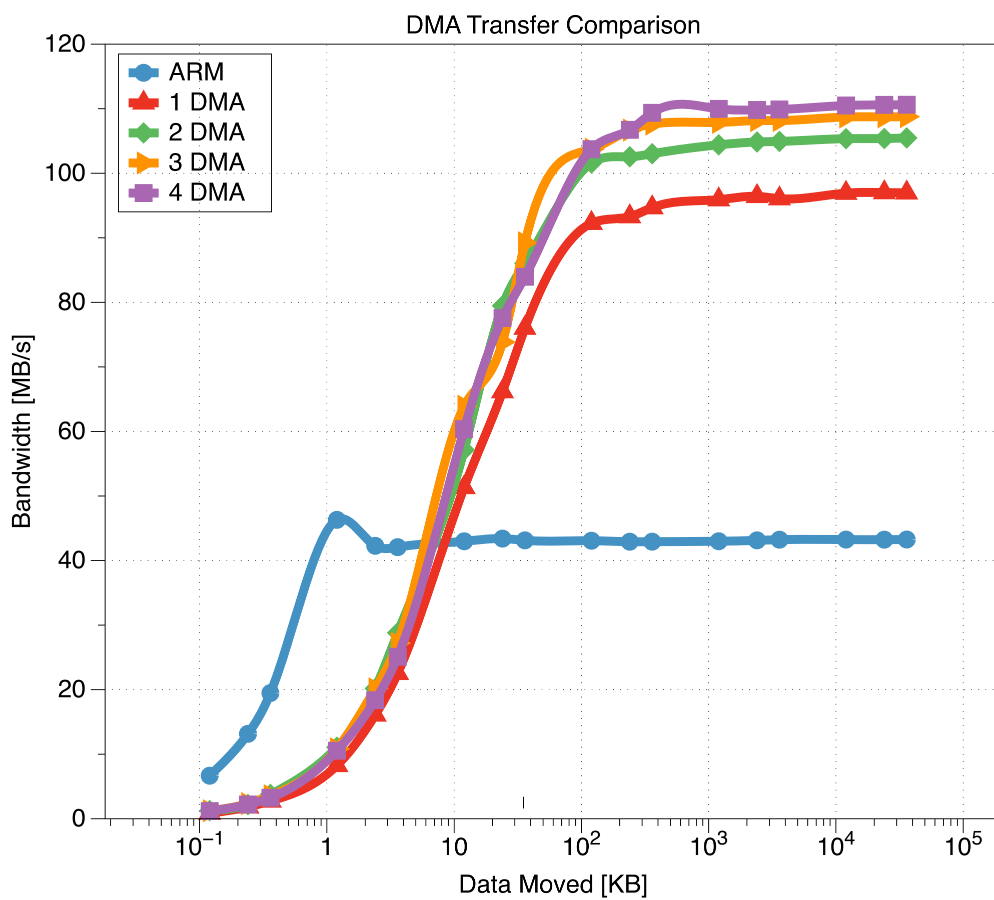


Figura 6.3: Comparazione delle prestazioni di accesso alla memoria tra ARM e DMA

in hardware.
Inizialmente è stato presentato il flusso di esecuzione per la generazione delle API, ovvero la procedura che permette partendo dalla funzione di creare l'interfaccia software verso i core. Successivamente è stata descritta l'implementazione di tale procedura mostrando nel dettaglio le API generate in funzione del tipo dei parametri. Le interfacce software sono differenti in base al fatto che i parametri siano di tipo primitivo piuttosto che strutturato.
Infine per comunicare con i core dotati di interfaccia stream si è reso necessario lo sviluppo di un driver per l'IP AXI DMA responsabile del trasferimento dei dati tra i core e la memoria. I dettagli implementativi del driver sono contenuti nell'ultimo paragrafo. Il componente AXI DMA influisce sulle prestazioni ottenute dall'applicazione progettata, per ottimizzare il tempo di esecuzione è bene tenere conto di tali prestazioni che sono state riportate mediante un grafico.

Capitolo 7

Caso di studio

In questo capitolo verranno ripercorsi tutti i passi sull'uso della toolchain sviluppata tramite un caso di studio reale, lo scopo è quello di mostrare e mettere in pratica tutte le informazioni raccolte nei capitoli precedenti. Il punto di partenza è quindi il taskgraph di un'applicazione reale, si mostrerà in dettaglio le funzioni principali da cui è formata l'applicazione e il risultato intermedio prodotto da ogni funzione. Successivamente ogni nodo/fase del taskgraph verrà sintetizzato tramite la High Level Synthesis (HLS) ottenendo l'IP hardware da importare e integrare nell'architettura finale. Una volta composto il sistema finale verrà generato il bitstream per configurare la logica programmabile. Infine a valle della toolchain saranno generati e disponibili da usare i driver e le Application Programming Interface (API) per sfruttare l'hardware generato.

7.1 Filtro Otsu

il caso di studio preso in esame per dare risalto al lavoro svolto è un filtro per immagini digitali, partendo da un'immagine a colori il filtro estrae come risultato un'altra immagine dove vi sono presenti solo due colori: il bianco e il nero, in particolare il filtro identifica gli oggetti presenti nella scena e li mette in risalto colorandoli di nero o di bianco secondo un procedimento presentato di seguito. La figura 7.1 mostra un esempio di un'immagine prima dell'applicazione del me-

todo Otsu, mentre la figura 7.2 contiene la stessa immagine dopo l'applicazione del filtro Otsu.

Figura 7.1: Immagine prima dell'applicazione del filtro Otsu

Figura 7.2: Immagine dopo l'applicazione del filtro Otsu

Come si osserva l'immagine risultato è binaria e contiene solo il colore bianco o nero. Seguirà ora la descrizione di tutte le fasi per arrivare ad ottenere l'immagine binaria:

1. lettura file immagine: il primissimo passo da eseguire è caricare in memoria l'immagine a cui applicare il filtro, bisogna quindi aprire il file immagine in formato bitmap (BMP) scartare l'intestazione del formato e caricare in memoria solo le informazioni sul colore dei pixels, nel formato bitmap ad

ogni pixel sono associati tre colori uno per ogni canale red, green e blue (RGB);

2. filtro a scala di grigi: partendo dall'immagine in memoria ottiene come risultato un'immagine a scala di grigi, ovvero tutti e tre i componenti di ogni pixel hanno il medesimo valore dato dalla media dei tre valori RGB originali;

3. istogramma: per ogni pixel del formato in scala di grigi si estrae il colore che è un numero intero che va da 0 a 255 (8 bit per ogni pixel in scala di grigi), per ogni colore trovato si tiene un contatore che tiene traccia di quante volte questo appare nell'immagine, così facendo si ottiene un istogramma;

4. calcolo della soglia: funzione fondamentale che partendo dall'istogramma calcola un valore di soglia da usare per discriminare i pixel bianchi da quelli neri, tale funzione costituisce una proprietà del filtro in quanto possono essere usate diverse funzioni per caratterizzare l'immagine risultato ottenuta;

5. partizionamento: partendo dal formato a scala di grigi e preso in ingresso il valore soglia calcolato precedentemente, questa funzione si occupa di assegnare il colore corretto per ogni pixel in base al valore di grigio originale. Per ogni pixel si confronta il colore in scala di grigi con il valore di soglia, se minore allora si assegna al pixel risultante il colore nero, altrimenti se maggiore si assegna il colore bianco;

6. immagine risultante: l'immagine risultato, ottenuta al termine dell'applicazione del metodo Otsu, deve essere nuovamente salvata su file; questa fase converte tale immagine nel formato bitmap, ovvero lo stesso formato di partenza.

Il caso di studio analizzato è quindi una sequenza di funzioni che partono da un'immagine e in maniera autonoma e indipendente calcolano il risultato, l'unica dipendenza presente si trova lungo la catena di chiamate a funzioni, ovvero ogni funzione lavora con i risultati prodotti da quella precedente. La figura 7.3

racchiude quanto detto fino ad ora evidenziando le dipendenze presenti.

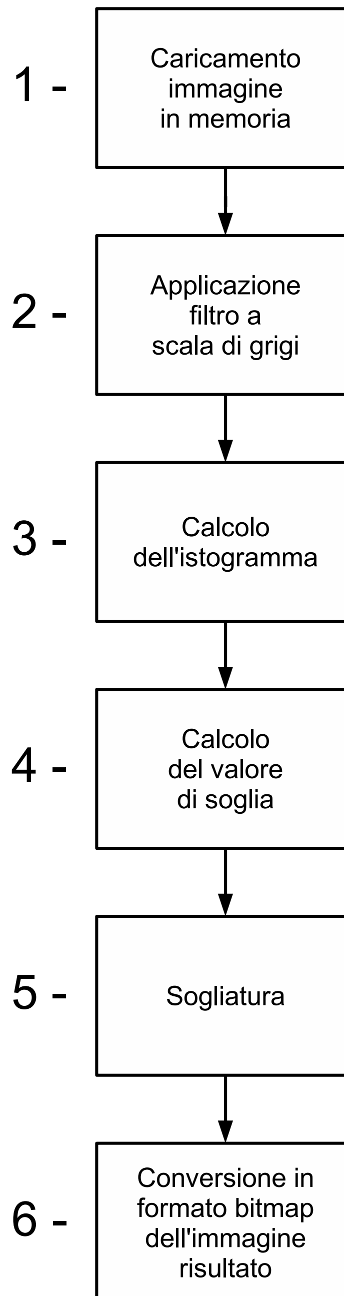


Figura 7.3: Rappresentazione grafica del metodo Otsu

Per completezza si riporta di seguito il listato di codice C contenente la funzione *main* dell'applicazione Otsu dal quale si può notare chiaramente la sequenza mostrata nella figura 7.3.

```
1  int main(int argc, char **argv){
2      if(argc!=5){
3          printf("USAGE: ./caseStudy INPUT.BMP rows cols OUTPUT.BMP\n");
4          exit(0);
5      }
6
7      char *fileNameIn = argv[1];
8      int rows = atoi(argv[2]);
9      int cols = atoi(argv[3]);
10     char *fileNameOut = argv[4];
11
12     unsigned char *header;
13     unsigned char *inputImage;
14
15     unsigned char *grayScaleImage;
16     unsigned char *segmentedGrayImage;
17     unsigned char *outputImage;
18
19     allocateImageBuffer(&grayScaleImage, rows, cols, 1);
20     allocateImageBuffer(&segmentedGrayImage, rows, cols, 1);
21     allocateImageBuffer(&outputImage, rows, cols, 1);
22
23     unsigned char otsuThreshold;
24     unsigned int histogram[256];
25
26     readImage(fileNameIn, rows, cols, &inputImage, &header);
27
28     setImageProperties(rows, cols);
29
30     //ACTUAL COMPUTATION FROM HERE
31
32     grayScale(inputImage, grayScaleImage);
33
34     computeHistogram(grayScaleImage, histogram);
35
36     otsuThreshold = halfProbability(histogram);
37     segment(grayScaleImage, otsuThreshold, segmentedGrayImage)
38
39     gs2rgb(segmentedGrayImage, outputImage);
```

```
40
41     //COMPUTATION DONE
42
43     writeImage(fileNameOut, rows, cols, outputImage, header);
44
45 }
```

7.2 Descrizione in forma di Taskgraph e SDF

La sezione precedente contiene la descrizione dell’applicazione Otsu e come è strutturata, in questo paragrafo invece verrà presentato il Taskgraph dell’intera applicazione Otsu a prescindere dalla fase di partizionamento e saranno forniti i dettagli riguardo i task e le fasi delle funzionalità di Otsu. A monte dell’utilizzo della toolchain realizzata, per implementare il caso di studio, va effettuato il partizionamento HW/SW, lo strumento sviluppato non si occupa di questa tematica ma riceve in ingresso l’applicazione già partizionata: in particolare il taskgraph delle funzioni che andranno poi istanziate in hardware. A tale scopo è quindi necessario illustrare il taskgraph e l’SDF dell’applicazione Otsu.
L’immagine in Figura 7.4 mostra i nodi da cui è composta l’applicazione, come si può notare il taskgraph è composto dai nodi 1 e 2 che sono effettivamente dei task e da un ulteriore nodo che rappresenta una fase. Il nodo fase può essere espanso in un SDF composto a sua volta da 4 nodi che rappresentano le funzioni principali del filtro.

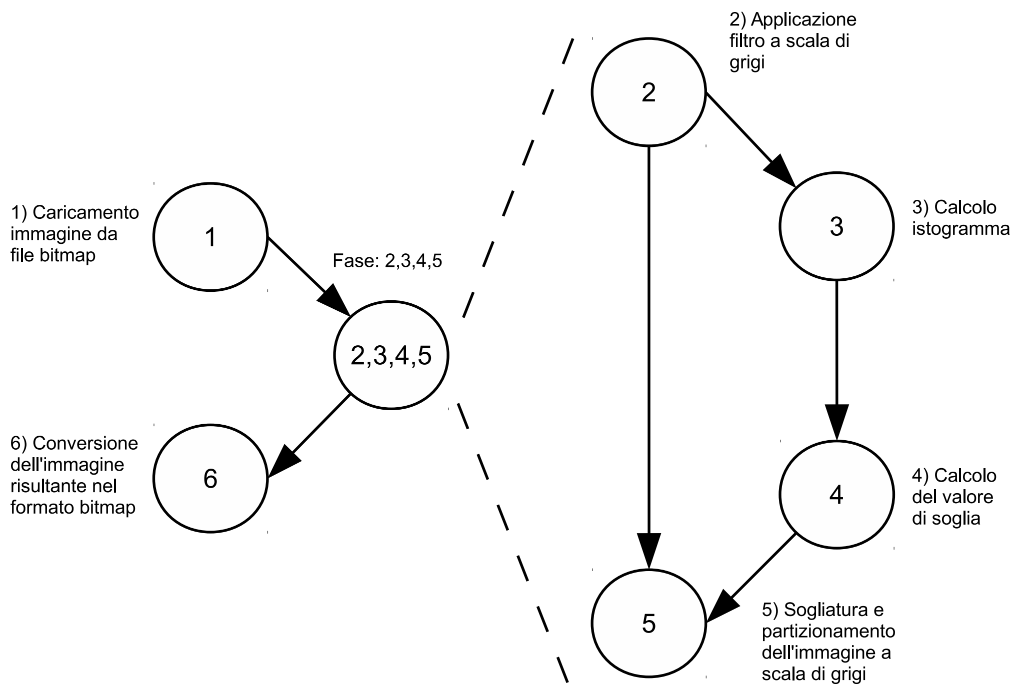


Figura 7.4: Taskgraph filtro Otsu

Oltre alla generazione automatica di architetture per sistemi riconfigurabili, la toolchain sviluppata può essere utilizzata per effettuare un'esplorazione dello spazio delle soluzioni, ovvero trovare la soluzione hardware migliore tenendo conto delle risorse hardware occupate su FPGA e del tempo di esecuzione dell'applicazione. Di seguito è riportato il prototipo di Otsu tramite l'utilizzo della toolchain sviluppata.

7.3 Prototipazione filtro Otsu

Analizzando in dettaglio l'applicazione si può notare che le funzioni 1 e 6 di Otsu sono interfaccie di I/O poiché leggono e scrivono da file, tali funzioni devono essere necessariamente eseguite in software. Il caricamento dell'immagine avviene in software utilizzando le usuali interfacce a file del sistema operativo Linux, si ottiene quindi un buffer in memoria contenente tutte le informazioni sul colore dei pixel. Per arrivare ad eseguire le funzioni hardware istanziate, l'applicazione deve utilizzare le interfacce software generate per permettere il

trasferimento dei dati verso la FPGA.
Per quanto riguarda le funzioni rimanenti, che costituiscono una fase, queste possono essere eseguite sia in software che in hardware, vi sono quindi diversi modi di mappare il filtro Otsu in hardware. Nelle sezioni seguenti saranno presentate alcune soluzioni di implementazione della fase.

7.3.1 Soluzione A: computeHistogram in HW

La Soluzione A sfrutta la FPGA per eseguire in hardware solo la funzione di calcolo dell'istogramma, ovvero la funzione numero 3 della lista in figura 7.3. Il taskgraph prevede quindi una sola funzione di nome *computeHistogram* che ha due canali di comunicazione stream: il primo è in ingresso al nodo e contiene l'immagine a scala di grigi mentre il secondo è in uscita e contiene l'istogramma appena calcolato. L'intero taskgraph è riportato in figura 7.11.

```
tg nodes;
  tg node "computeHistogram" is "grayScaleImage" is "histogram" end;
tg end_nodes;

tg edges;
  tg link 'soc to ("computeHistogram","grayScaleImage") end;
  tg link ("computeHistogram","histogram") to 'soc end;
tg end_edges;
```

Figura 7.5: Taskgraph hardware della Soluzione A

La figura 7.6 mostra il partizionamente HW/SW nel caso di implementazione in hardware della sola funzione *computeHistogram*.

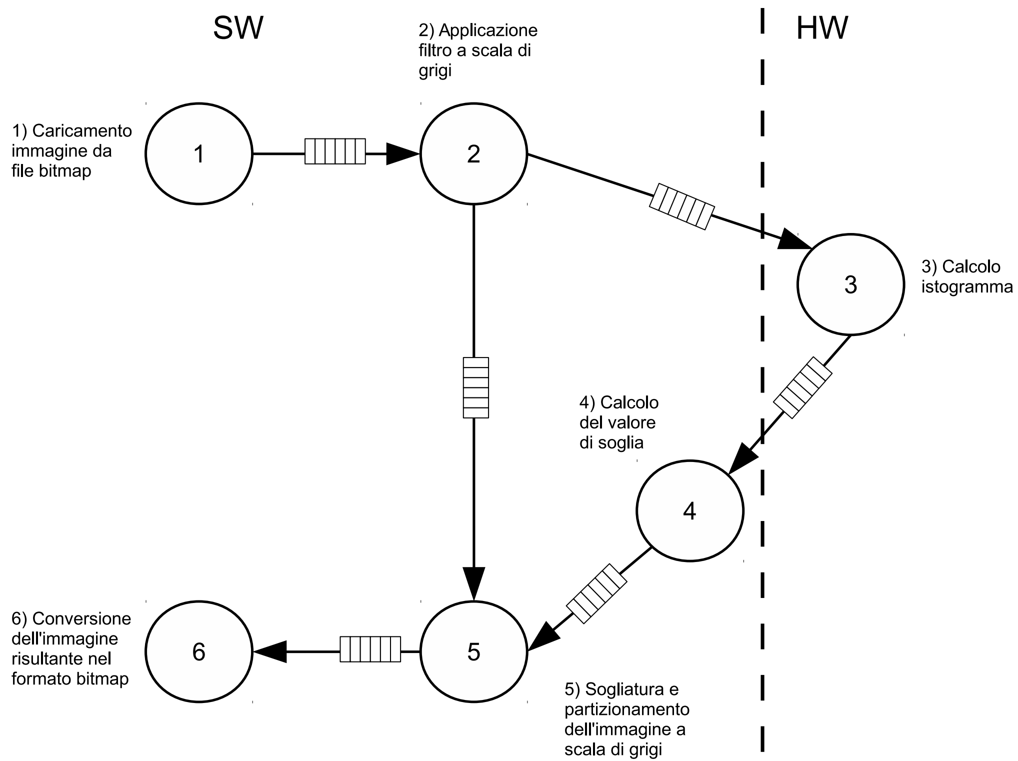


Figura 7.6: Partizionamento HW/SW della Soluzione A

Integrando il core con il resto del sistema si ottiene l'architettura hardware in Figura 7.7.

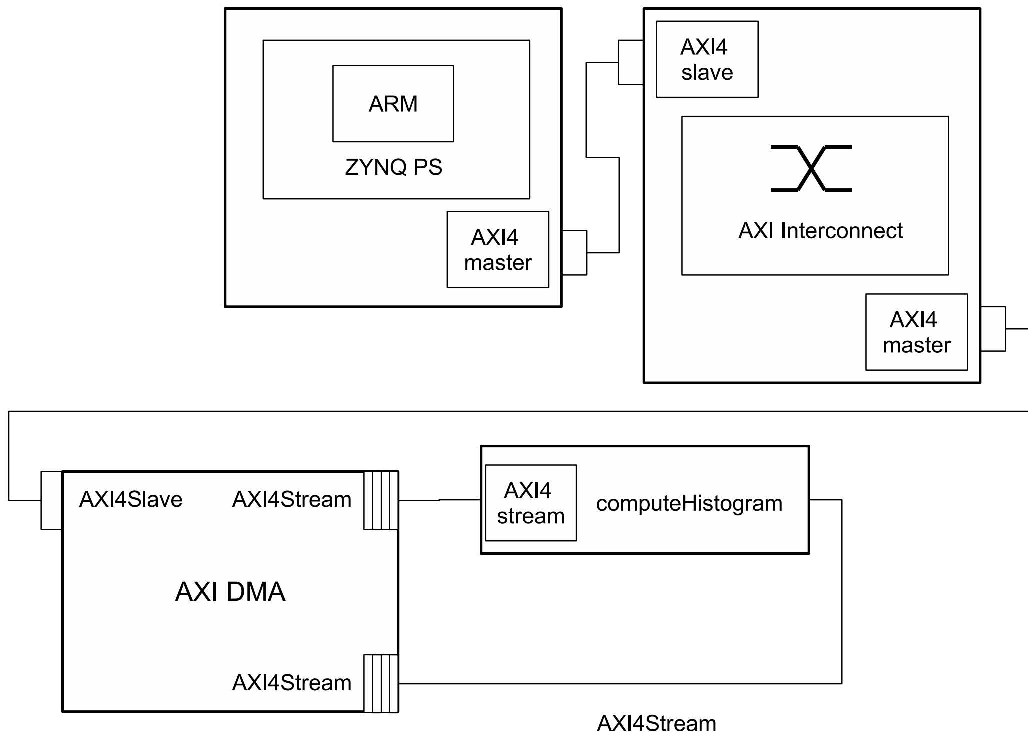


Figura 7.7: Architettura Soluzione A

L'elenco delle risorse occupate su FPGA dalla funzione *computeHistogram* sono riportare di seguito nella tabella 7.1

Risorsa	Utilizzo	Disponibilità	Utilizzo %
LUT	3809	53200	7.15
FF	4562	106400	4.28
RAMB36	2	140	1.42
RAMB18	1	280	0.35
DSP	0	220	0.00

Tabella 7.1: Tabella delle risorse hardware occupate dall'architettura in figura 7.6

7.3.2 Soluzione B: halfProbability in HW

La Soluzione B implementa in hardware solo la funzione *halfProbability*, ovvero quella incaricata di estrarre il valore di soglia da usare per partizionare l'immagine a scala di grigi. Al pari dei sottoparagrafi precedenti viene inserito il taskgraph che rappresenta questo scenario ed è visibile in figura 7.8.

```
tg nodes;
  tg node "halfProbability" is "histogram" is "probability" end;
tg end_nodes;

tg edges;
  tg link 'soc to ("halfProbability","histogram") end;
  tg link ("halfProbability","probability") to 'soc end;
tg end_edges;
```

Figura 7.8: Taskgraph hardware della Soluzione B

Esattamente come la soluzione precedente, questa implementazione prevede un unico nodo collegato al componente AXI DMA incaricato di muovere dati da e verso il core *halfProbability*. Per completare l'analisi di questa soluzione è stato di seguito riportato il grafico di partizionamento HW/SW (figura 7.9).

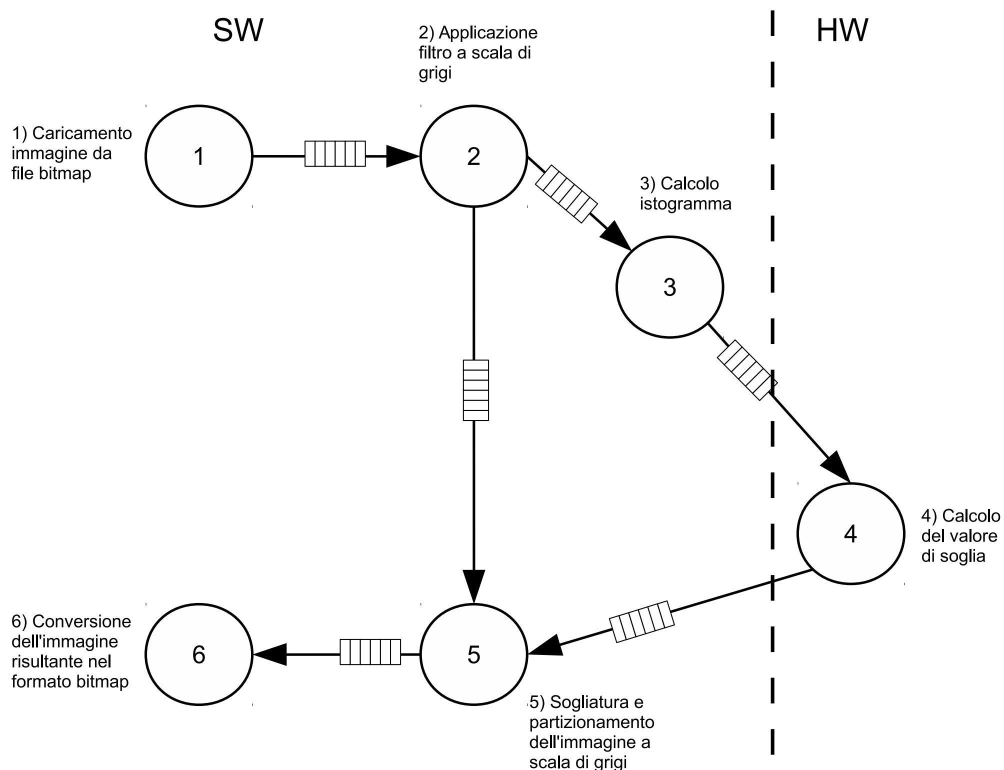


Figura 7.9: Partizionamento HW/SW della Soluzione B

Di seguito viene riportata l'architettura hardware implementata nella Soluzione B 7.10.

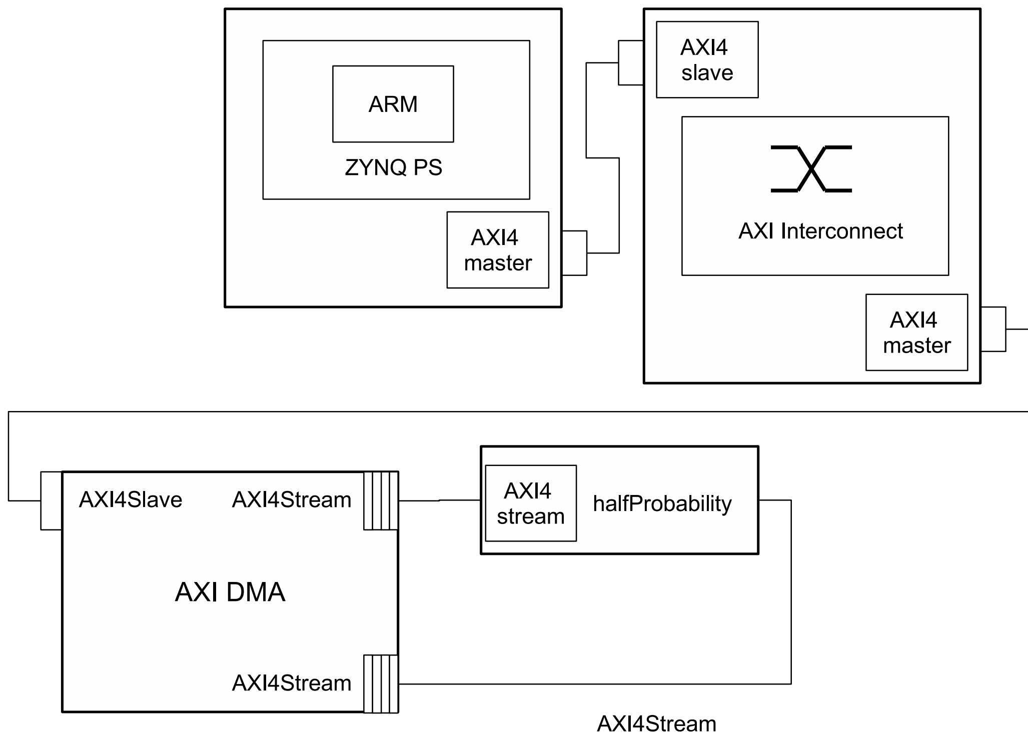


Figura 7.10: Architettura Soluzione B

Infine viene riportata la tabella riassuntiva che mostra l'occupazione su scheda della funzione *halfProbability* (tabella 7.2) compresa di componente AXI DMA. Tutte le informazioni sono riportate in termini di risorse hardware utilizzate, ovvero numero di risorse e percentuale.

Risorsa	Utilizzo	Disponibilità	Utilizzo %
LUT	7834	53200	14.72
FF	9951	106400	9.35
RAMB36	2	140	1.42
RAMB18	0	280	0.00
DSP	2	220	0.90

Tabella 7.2: Tabella delle risorse hardware occupate dall'architettura in figura 7.9

7.3.3 Soluzione C: computeHistogram e halfProbability in HW

Nella Soluzione C proposta è prevista l'implementazione in hardware di due delle funzioni di Otsu elencate in figura 7.3: in particolare verranno istanziate sulla FPGA le funzioni di calcolo dell'istogramma partendo dall'immagine a scala di grigi e di computazione del valore di soglia, ovvero le funzioni 3 e 4. Il taskgraph in ingresso alla toolchain è mostrato in figura 7.11.

```
tg nodes;
  tg node "computeHistogram" is "grayScaleImage" is "histogram" end;
  tg node "halfProbability" is "histogram" is "probability" end;
tg end_nodes;

tg edges;
  tg link 'soc to ("computeHistogram","grayScaleImage") end;
  tg link ("computeHistogram","histogram") to ("halfProbability","histogram") end;
  tg link ("halfProbability","probability") to 'soc end;
tg end_edges;
```

Figura 7.11: Taskgraph hardware della Soluzione C

Il componente AXI DMA in questo caso è connesso direttamente alla funzione *computeHistogram* che rappresenta il punto di ingresso dei dati da elaborare su FPGA. Al termine della computazione hardware i risultati tornano in memoria attraverso il DMA che li recupera dalla funzione *halfProbability*, Lo schema di partizionamento è presente in figura 7.12.

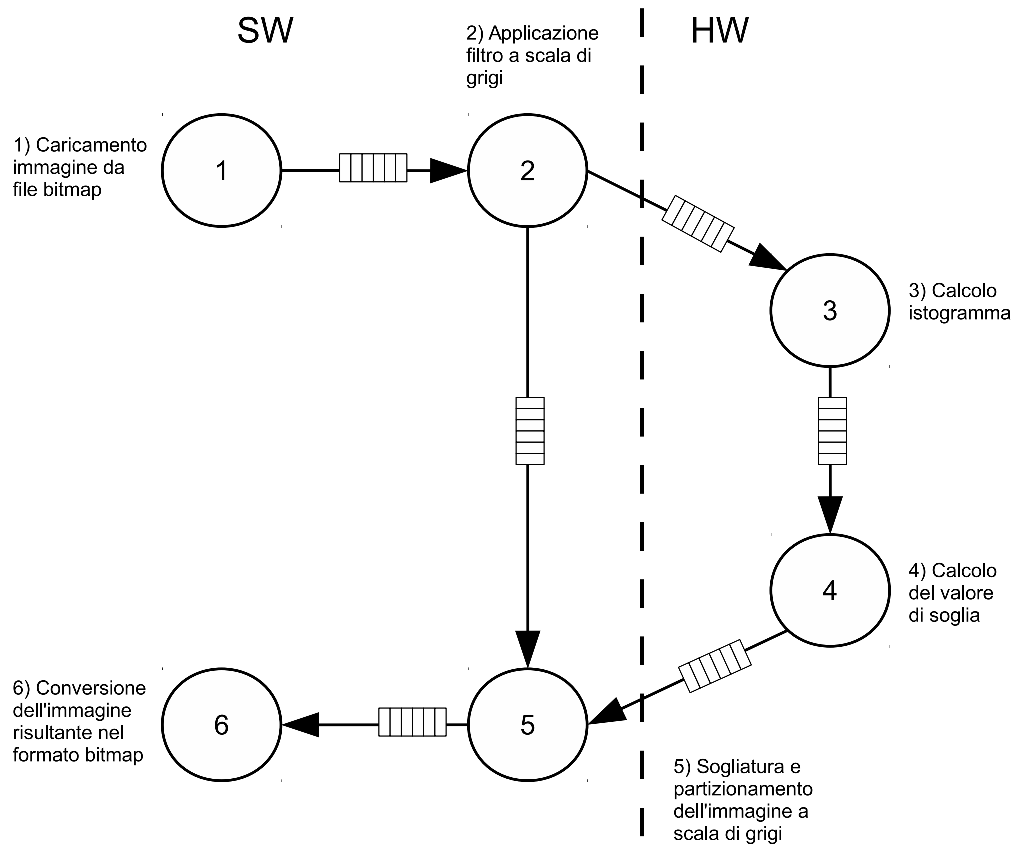


Figura 7.12: Partizionamento HW/SW della Soluzione C

In figura 7.13 è riportata l'architettura della Soluzione C.

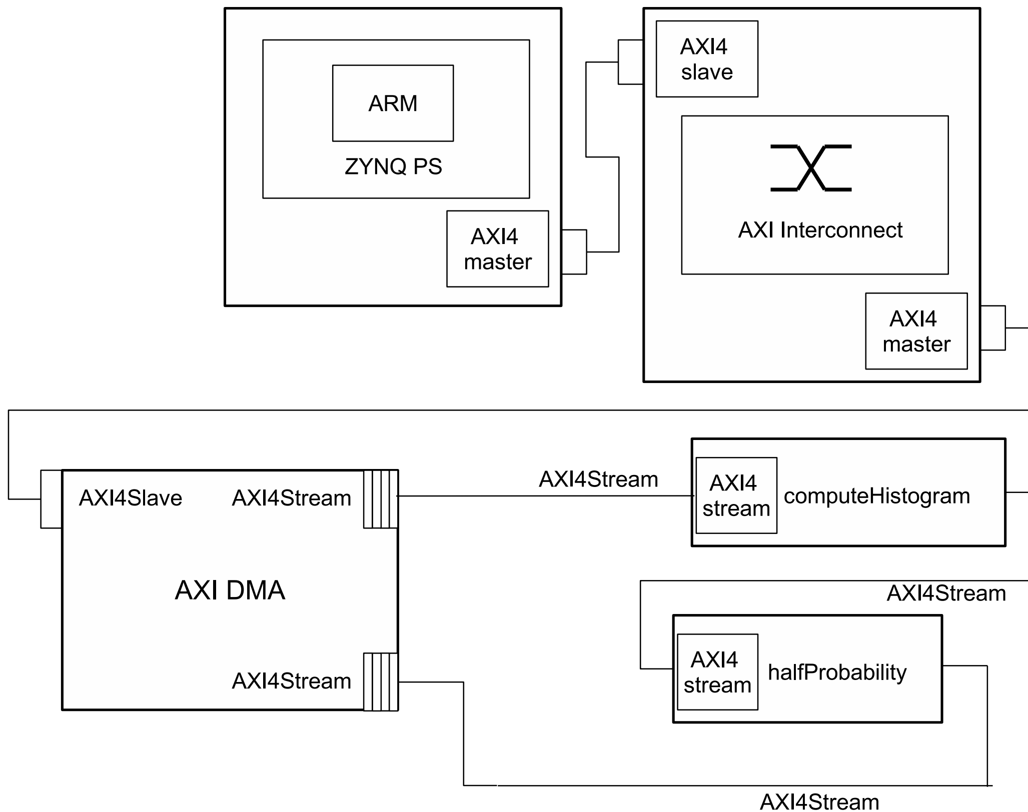


Figura 7.13: Architettura Soluzione C

I risultati in termini di occupazione delle risorse sono riportati in tabella 7.4

Risorsa	Utilizzo	Disponibilità	Utilizzo %
LUT	8190	53200	15.39
FF	10234	106400	9.61
RAMB36	2	140	1.42
RAMB18	1	280	0.35
DSP	2	220	0.90

Tabella 7.3: Tabella delle risorse hardware occupate dall'architettura in figura 7.12

7.3.4 Soluzione D: intera fase in HW

La suddivisione HW/SW scelta per implementare questa Soluzione prevede l'esecuzione in hardware di tutte le funzioni principali: i punti 2,3,4,5 della lista, ovvero le funzioni di calcolo dell'immagine a scala di grigi, dell'istogramma, del valore di soglia e del partizionamento dell'immagine.

La figura 7.14 mostra il risultato del partizionamento hardware/software. Il punto di ingresso del taskgraph è la funzione *grayScale*, tale funzione è dotata di un parametro che non è di tipo primitivo in quanto rappresenta il buffer con l'immagine in ingresso, le sue interfacce hardware saranno quindi le interfacce AXI4Stream. I dati vanno trasferiti dalla DDR alla FPGA e vi rimangono per l'intera esecuzione composta dalle funzioni principali, bisogna tenere conto che l'esecuzione del flusso hardware avviene in maniera streaming e quindi sulla FPGA, in un dato istante di tempo, non è presente l'intera immagine ma solo la porzione processata in quel momento. Al termine dell'esecuzione delle funzioni hardware i dati devono ritornare a disposizione dell'applicazione per eseguire la parte rimanente dell'algoritmo: la scrittura su file dell'immagine risultante. Il punto di uscita del taskgraph è la funzione di partizionamento dell'immagine, l'uscita di questa funzione è quindi collegata all'AXI DMA che si occupa di trasferire in memoria il risultato in maniera duale al metodo usato per fare arrivare l'immagine dalla memoria all'area programmabile.

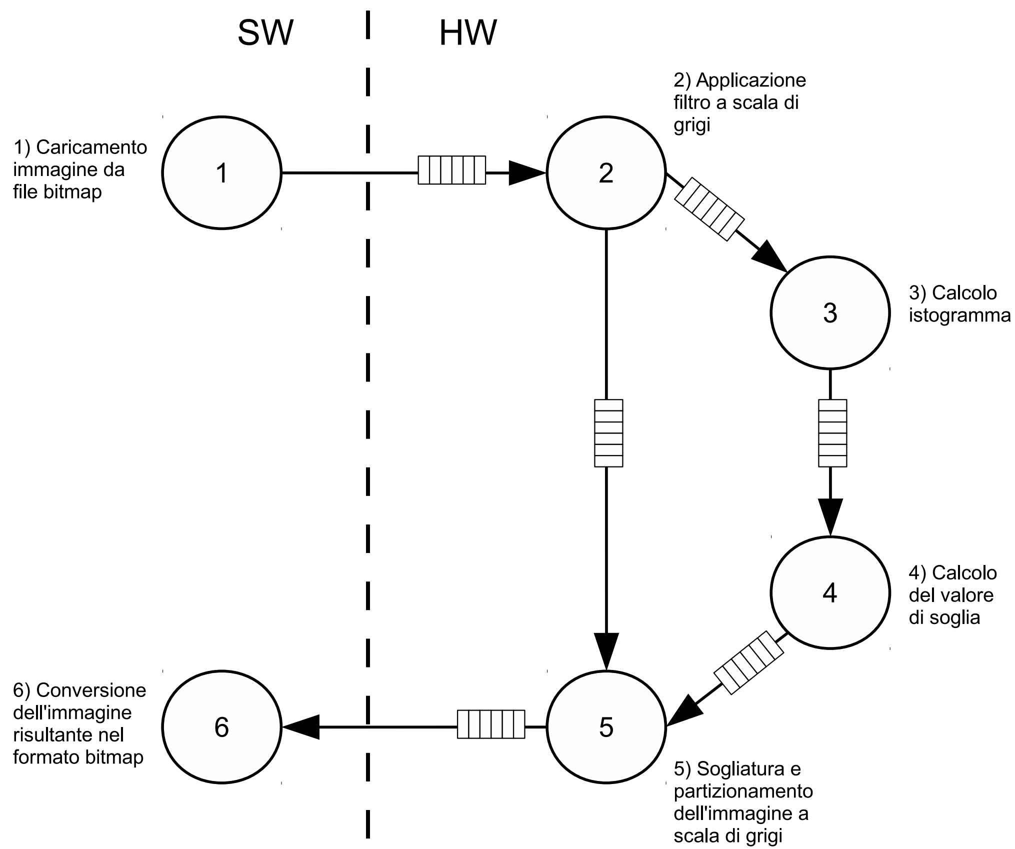


Figura 7.14: Partizionamento HW/SW della Soluzione D

Al fine di generare l'intera architettura hardware con il relativo software, lo schema di coprogettazione illustrato in figura 7.14 va tradotto nel corrispettivo Domain Specific Language (DSL) creato in linguaggio Scala per l'utilizzo della toolchain sviluppata (vedi paragrafo 5.2).

```
tg nodes;
  tg node "grayScale" is "imageIn" is "imageOutCH" is "imageOutSEG" end;
  tg node "computeHistogram" is "grayScaleImage" is "histogram" end;
  tg node "halfProbability" is "histogram" is "probability" end;
  tg node "segment" is "grayScaleImage" is "otsuThreshold" is "segmentedGrayImage" end;
tg end_nodes;

tg edges;
  tg link 'soc to ("grayScale","imageIn") end;
  tg link ("grayScale","imageOutCH") to ("computeHistogram","grayScaleImage") end;
  tg link ("grayScale","imageOutSEG") to ("segment","grayScaleImage") end;
  tg link ("computeHistogram","histogram") to ("halfProbability","histogram") end;
  tg link ("halfProbability","probability") to ("segment","otsuThreshold") end;
  tg link ("segment","segmentedGrayImage") to 'soc end;
tg end_edges;
```

Figura 7.15: Taskgraph della Soluzione D

La figura 7.15 illustra il risultato di questa traduzione mostrando il taskgraph ottenuto, le funzioni riportate sono *grayScale*, *computeHistogram*, *halfProbability* e *segment*. Nella sezione relativa ai link si può osservare come queste siano collegate tra di loro in maniera coerente a quanto mostrato dalla figura 7.14: l'input in ingresso alla funzione *grayScale* arriva direttamente dalla memoria tramite l'uso dell'AXI DMA, questa funzione produce in output l'immagine in formato a scala di grigi che andrà in ingresso sia alla funzione *computeHistogram*, per il calcolo dell'istogramma, sia alla funzione di segmentazione che crea l'immagine binaria a partire da quella a scala di grigi. L'istogramma prodotto dalla funzione *computeHistogram* arriva in ingresso alla funzione *halfProbability* che si occupa di estrarre un valore di soglia usato dalla funzione *segment*.
Confrontando la figura 7.15 con la 7.14 si nota che i nodi supportati, da descrivere in forma di taskgraph, sono quelli da implementare poi in hardware.
Infine la figura 7.16 mostra l'architettura hardware della Soluzione D da programmare su FPGA una volta ottenuto il bitstream tramite Vivado.

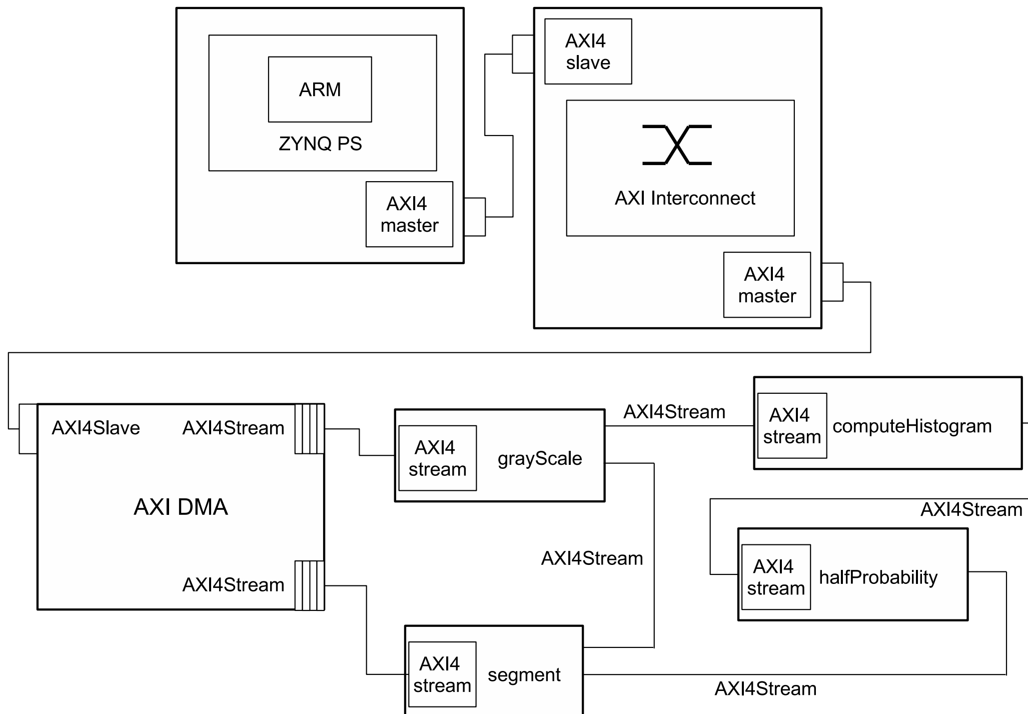


Figura 7.16: Architettura Soluzione D

A valle dell'implementazione dell'architettura in Figura 7.16, la tabella riportata in Figura 7.4 mostra le risorse occupate sulla FPGA, indicando una percentuale di utilizzo in relazione alla loro disponibilità totale.

Risorsa	Utilizzo	Disponibilità	Utilizzo %
LUT	9312	53200	17.50
FF	11265	106400	10.58
RAMB36	2	140	1.42
RAMB18	1	280	0.35
DSP	3	220	1.36

Tabella 7.4: Tabella delle risorse hardware occupate dall'architettura in figura 7.16

7.4 Sommario

L'applicazione scelta come caso di studio è un filtro per immagini chiamato Otsu, la toolchain è stata utilizzata per realizzare un prototipo di tale filtro mappando diverse funzionalità in hardware con lo scopo di fare design space

exploration tramite l'implementazione di diverse soluzioni architetturali.
La toolchain ha permesso di astrarre dai dettagli implementativi passando attraverso una descrizione in forma di taskgraph e generando in automatico le architetture.
I risultati prodotti dalle diverse implementazioni possono essere facilmente osservati, analizzati e confrontati tra loro dal progettista che ha il compito di scegliere quella che risulta migliore secondo le specifiche del progetto.
Rimane in carico al progettista la scrittura del runtime per eseguire l'applicazione anche se il tool mette a disposizione tutte le API di cui ha bisogno.

Capitolo 8

Conclusioni e lavori futuri

Il lavoro di tesi proposto si colloca nell'ambito dell'automazione della progettazione di sistemi programmabili, in particolare fornisce al progettista uno strumento che lo assiste nella generazione della piattaforma hardware/software e si colloca a valle della scrittura dell'applicazione. Per riuscire a fornire Il supporto e l'automazione proposti in questa tesi è stato necessario lo sviluppo di una toolchain che in maniera automatica arriva a generare l'intera piattaforma hardware, di fatto nasconde al progettista tutte le difficoltà, compresi i dettagli implementativi, che caratterizzano la realizzazione di questi sistemi. Esistono diverse fasi che portano un'applicazione software ad essere eseguita su un dispositivo embedded con parte di essa eseguita in hardware sul sistema programmabile. Queste fasi comprendono:

- lo sviluppo dell'applicazione e la coprogettazione hardware/software: ovvero quali parti di essa vanno eseguite sul processore e quali invece come componente hardware su logica programmabile;
- generazione della piattaforma hardware: sintesi ad alto livello dei core hardware, implementazione dell'architettura e generazione del bitstream per programmare la FPGA;
- generazione dell'architettura software: creazione delle Application Programming Interface (API) per sfruttare i driver di basso livello creati contem-

poraneamente all'architettura hardware e accedere alle funzionalità fornite come IP core;

- testing dell'applicazione: dopo aver riscritto l'applicazione per utilizzare le interfacce software messe a disposizione l'ultimo passo da eseguire è il test dell'applicazione sul dispositivo embedded.

Se escludiamo il primo e l'ultimo punto della lista appena citata, la toolchain fornisce al progettista dell'applicazione la completa automazione di tutti gli altri punti rimanenti.
L'intera tesi è stata strutturata in modo da presentare in maniera approfondita come dette fasi vengono realizzate. Nei primissimi capitoli viene data una contestualizzazione del problema tramite un capitolo introduttivo sui temi trattati e una presentazione dello stato dell'arte riguardo i lavori correlati a quello svolto in questa tesi. Nel capitolo introduttivo vengono descritti gli elementi che caratterizzano un sistema riconfigurabile elencandone i pregi e difetti, in particolare viene messo in risalto che tali sistemi sono ottimi per sviluppare rapidamente prototipi per testare la corretta funzionalità dell'architettura progettata. Un terzo capitolo è dedicato alla formulazione approfondita del tema trattato, definisce il concetto di taskgraph insieme alla sua forma gerarchica chiamata Synchronous Data Flow (SDF), infine illustra con chiarezza il problema risolto e le possibilità offerte dal sistema sviluppato. L'obiettivo raggiunto con questo lavoro di tesi, partendo da una descrizione grafica dell'applicazione in forma di taskgraph e dei relativi sorgenti, è quello di effettuare la High Level Synthesis (HLS), generare automaticamente l'infrastruttura sia hardware producendo il bitstream (file di configurazione della logica programmabile) e infine supportare il progettista con la generazione dell'infrastruttura software (driver per le interfacce verso l'hardware e le API per l'utilizzo delle funzionalità hardware direttamente dall'applicazione software sviluppata). Dopo aver formulato il problema i rimanenti capitoli sono dedicati a descrivere nel dettaglio come è strutturata la toolchain e la scheda di sviluppo utilizzata per lo scopo, con tutti i dettagli implementativi che ne conseguono, in particolare sono stati dedicati tre capitoli così organizzati:

- un capitolo per la descrizione della piattaforma hardware presa in esame e dei protocolli di comunicazione utilizzati da Xilinx;
- un ulteriore capitolo che illustra in dettaglio il software sviluppato nell'ambito di questa tesi, ovvero della toolchain sviluppata e della generazione automatica dell'architettura e dei driver.
- infine è stato dedicato un capitolo per mostrare il caso di studio trattato e i risultati ottenuti su un'applicazione reale.

Un possibile sviluppo futuro potrebbe concentrarsi nel cercare di alzare il livello di astrazione ora raggiunto. Si potrebbe estendere la toolchain al fine di ottenere uno strumento simile ad un compilatore che preso in ingresso il sorgente C sia in grado di scegliere, in base all'architettura target, le funzioni da accelerare. Tale compilatore deve essere in grado di generare automaticamente l'architettura e i drivers in modo da integrare la loro invocazione direttamente in fase di compilazione. Tale estensione appena descritta porterebbe questo strumento ad avvicinarsi a quello ideale dal punto di vista del programmatore, ovvero che rende il processo di accelerazione di un'applicazione in hardware completamente trasparente e automatico.
Un ulteriore lavoro futuro potrebbe riguardare il supporto per architetture riconfigurabili con la generazione dei bitstream parziali e l'ampliamento delle API software generate per gestire la riconfigurabilità parziale, ampliando nel contempo le schede hardware supportate. Ad oggi l'architettura target è rappresentata unicamente dalla Xilinx Zedboard.

Bibliografia

[1] Hristo Nikolov, Mark Thompson, Todor Stefanov, Andy D. Pimentel, Simon Polstra, R. Bose, Claudiu Zissulescu, and Ed F. Deprettere. Daedalus: toward composable multimedia mp-soc design. In Limor Fix, editor, *Proceedings of the 45th Design Automation Conference, DAC 2008, Anaheim, CA, USA, June 8-13, 2008*, pages 574–579. ACM, 2008.

[2] Diana Gohringer and Jürgen Becker. High performance reconfigurable multiprocessor-based computing on fpgas. In *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pages 1–4. IEEE, 2010.

[3] Diana Gohringer, Michael Hubner, Etienne Nguepi Zeutebouo, and Jürgen Becker. Cap-os: Operating system for runtime scheduling, task mapping and resource management on reconfigurable multiprocessor architectures. In *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pages 1–8. IEEE, 2010.

[4] Jason Cong, Yiping Fan, Guoling Han, Wei Jiang, and Zhiru Zhang. Platform-based behavior-level and system-level synthesis. In *SOC Conference, 2006 IEEE International*, pages 199–202. IEEE, 2006.

[5] David A Patterson and John L Hennessy. *Computer organization and design: the hardware/software interface*. Newnes, 2013.

[6] Oliver Pell, Lee W Howes, Kubilay Atasu, Olav Beckmann, and Oskar Mencer. Accelerating scientific computations using fpgas. In *The Advanced Maui Optical and Space Surveillance Technologies Conference*, volume 1, page 97, 2006.

[7] Wayne Wolf, Ahmed Amine Jerraya, and Grant Martin. Multiprocessor system-on-chip (mpsoc) technology. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 27(10):1701–1713, 2008.

[8] Xilinx Zynq-7000 Silicon Devices. `http://www.xilinx.com/products/silicon-devices/soc/zynq-7000/silicon-devices.html`.

[9] Rainer Kress, Andreas Pyttel, and Alexander Sedlmeier. Fpga-based prototyping for product definition. In *Field-Programmable Logic and Applications: The Roadmap to Reconfigurable Computing*, pages 78–86. Springer, 2000.

[10] Xilinx Vivado. `http://www.xilinx.com/products/design-tools/vivado/integration/esl-design/`.

[11] Sven Verdoolaege, Hristo Nikolov, and Todor Stefanov. Pn: a tool for improved derivation of process networks. *EURASIP journal on Embedded Systems*, 2007(1):19–19, 2007.

[12] Andy D Pimentel, Cagkan Erbas, and Simon Polstra. A systematic approach to exploring embedded system architectures at multiple abstraction levels. *Computers, IEEE Transactions on*, 55(2):99–112, 2006.

[13] Cagkan Erbas, Andy D Pimentel, Mark Thompson, and Simon Polstra. A framework for system-level modeling and simulation of embedded systems architectures. *EURASIP Journal on Embedded Systems*, 2007, 2007.

[14] Hristo Nikolov, Todor Stefanov, and Ed Deprettere. Systematic and automated multiprocessor system design, programming, and implementation. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 27(3):542–555, 2008.

[15] Todor Stefanov, Ed Deprettere, and Hristo Nikolov. Multi-processor system design with espam. In *Hardware/Software Codesign and System Synthesis, 2006. CODES+ ISSS'06. Proceedings of the 4th International Conference*, pages 211–216. IEEE, 2006.

[16] KAHN Gilles. The semantics of a simple language for parallel programming. In *In Information Processing'74: Proceedings of the IFIP Congress*, volume 74, pages 471–475, 1974.

[17] Sjoerd Meijer, Hristo Nikolov, and Todor Stefanov. Combining process splitting and merging transformations for polyhedral process networks. In *Embedded Systems for Real-Time Multimedia (ESTIMedia), 2010 8th IEEE Workshop on*, pages 97–106. IEEE, 2010.

[18] Sjoerd Meijer, Hristo Nikolov, and Todor Stefanov. Throughput modeling to evaluate process merging transformations in polyhedral process networks. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 747–752. European Design and Automation Association, 2010.

[19] Dmitry Nadezhkin, Hristo Nikolov, and Todor Stefanov. Translating affine nested-loop programs with dynamic loop bounds into polyhedral process networks. In *Embedded Systems for Real-Time Multimedia (ESTIMedia), 2010 8th IEEE Workshop on*, pages 21–30. IEEE, 2010.

[20] Dmitry Nadezhkin and Todor Stefanov. Identifying communication models in process networks derived from weakly dynamic programs. In *Embedded Computer Systems (SAMOS), 2010 International Conference on*, pages 372–379. IEEE, 2010.

[21] Dmitry Nadezhkin and Todor Stefanov. Automatic derivation of polyhedral process networks from while-loop affine programs. In *Embedded Systems for Real-Time Multimedia (ESTIMedia), 2011 9th IEEE Symposium on*, pages 102–111. IEEE, 2011.

[22] Ozana Silvia Dragomir, Todor Stefanov, and Koen Bertels. Loop unrolling and shifting for reconfigurable architectures. In *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*, pages 167–172. IEEE, 2008.

[23] Mohamed Bamakhrama and Todor Stefanov. Hard-real-time scheduling of data-dependent tasks in embedded streaming applications. In *Proceedings of the ninth ACM international conference on Embedded software*, pages 195–204. ACM, 2011.

[24] Susan L Graham, Peter B Kessler, and Marshall K Mckusick. Gprof: A call graph execution profiler. In *ACM Sigplan Notices*, volume 17, pages 120–126. ACM, 1982.

[25] D Gohringer, Jan Luhmann, and Jürgen Becker. Generatercs: A high-level design tool for generating reconfigurable computing systems. In *Very Large Scale Integration (VLSI-SoC), 2009 17th IFIP International Conference on*, pages 159–164. IEEE, 2009.

[26] Frances E Allen. Control flow analysis. In *ACM Sigplan Notices*, volume 5, pages 1–19. ACM, 1970.

[27] Alex Orailoglu and Daniel D Gajski. Flow graph representation. In *Proceedings of the 23rd ACM/IEEE Design Automation Conference*, pages 503–509. IEEE Press, 1986.

[28] Deming Chen, Jason Cong, Yiping Fan, and Junjuan Xu. Optimality study of resource binding with multi-vdds. In *Proceedings of the 43rd annual Design Automation Conference*, pages 580–585. ACM, 2006.

[29] Jason Cong, Yiping Fan, Guoling Han, Yizhou Lin, Junjuan Xu, Zhiru Zhang, and Xu Cheng. Bitwidth-aware scheduling and binding in high-level synthesis. In *Proceedings of the 2005 Asia and South Pacific Design Automation Conference*, pages 856–861. ACM, 2005.

[30] Jason Cong, Guoling Han, and Zhiru Zhang. Architecture and compilation for data bandwidth improvement in configurable embedded processors. In *Proceedings of the 2005 IEEE/ACM International conference on Computer-aided design*, pages 263–270. IEEE Computer Society, 2005.

[31] Jason Cong, Yiping Fan, Guoling Han, Ashok Jagannathan, Glenn Reinman, and Zhiru Zhang. Instruction set extension with shadow registers for configurable processors. In *Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*, pages 99–106. ACM, 2005.

[32] Jason Cong, Yiping Fan, Guoling Han, Xun Yang, and Zhiru Zhang. Architecture and synthesis for on-chip multicycle communication. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 23(4):550–564, 2004.

[33] Xilinx ZYNQ. `http://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html`.

[34] T.H. Cormen. Introduction to algorithms. 2001.

[35] Milind Girkar and Constantine D Polychronopoulos. The hierarchical task graph as a universal intermediate representation. *International Journal of Parallel Programming*, 22(5):519–551, 1994.

[36] Constantine D Polychronopoulos. The hierarchical task graph and its use in auto-scheduling. In *Proceedings of the 5th international conference on Supercomputing*, pages 252–263. ACM, 1991.

[37] AXI DMA Driver. `https://github.com/durellinux/ZedBoard_Linux_DMA_driver`.

[38] Xilinx AXI DMA. `http://www.xilinx.com/support/documentation/ip_documentation/axi_dma/v7_1/pg021_axi_dma.pdf`.

1 dicembre 2014

Document typeset with LaTeX