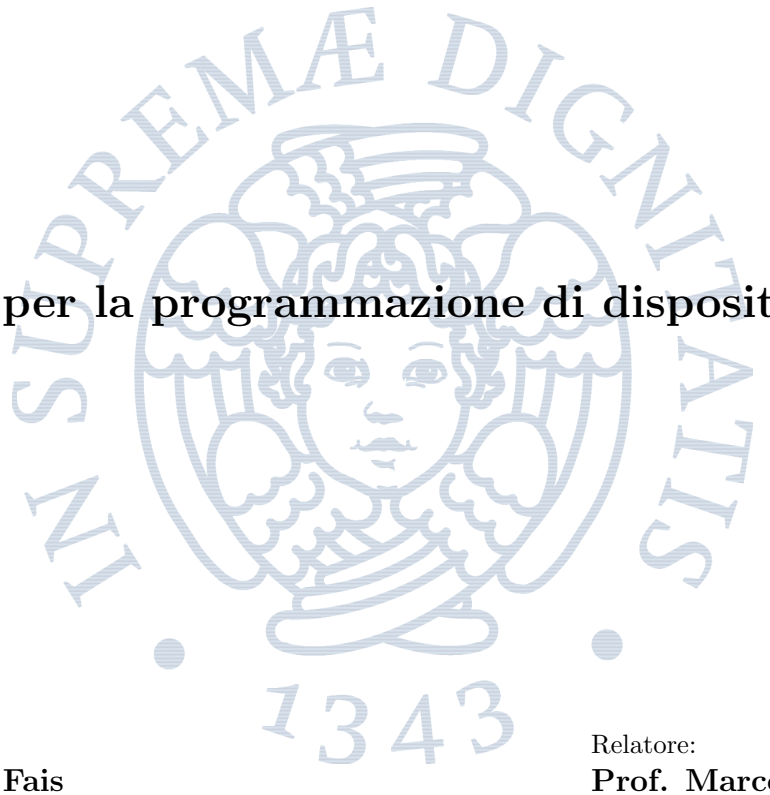
UNIVERSITÀ DI PISA

DIPARTIMENTO DI INFORMATICA

Corso di Laurea in Informatica

RELAZIONE FINALE

Ambienti per la programmazione di dispositivi FPGA



Candidato:
Alessandra Fais

Relatore:
Prof. Marco Danelutto

Anno Accademico 2014/2015

Indice

Ringraziamenti

È arrivato il momento in cui poter dire di essere riuscita a concludere un altro percorso, il più difficile che ho intrapreso fino ad ora: ho sempre pensato alla laurea come a una cosa lontana, e invece eccomi qua!

Credo sia doveroso tirare un po' le somme di questi ultimi anni, che tra cose sia positive che negative mi hanno permesso di crescere e diventare (credo) una persona migliore. Il fatto di uscire fuori di casa è stata una spinta per imparare a cavarmela da sola. Le difficoltà che ho incontrato durante il corso di studi sono state parecchie, a partire da quelle iniziali dovute a tante lacune pregresse, ma anche quelle insite del corso di laurea stesso. Nonostante questo, a parte qualche periodo di scoraggiamento, ho capito sempre meglio che davvero l'ambito che ho scelto era quello giusto per me, e sono contenta di questa scelta.

Le persone che più di tutte vorrei ringraziare sono i miei genitori, che hanno sempre avuto tanta fiducia in me e mi hanno sostenuto in ogni situazione, quindi grazie mamma e babbo! Ovviamente c'è anche mio fratello Marco, il primo informatico di casa. Sicuramente è un po' merito suo se ho iniziato a maturare l'interesse e la curiosità per questo ambito, fino a decidere di intraprendere lo stesso percorso di studi. Grazie per essermi stato sempre vicino e per avermi spronato a cercare di fare sempre meglio! Infine l'università e la città mi hanno permesso di conoscere tante persone, che mi hanno arricchito, ognuno a suo modo. Vorrei ringraziare in particolare Orlando, bravissimo collega, più di un amico e fervido sostenitore morale, il suo contributo in questi anni è stato senza dubbio molto importante. Un grazie enorme va anche alla combriccola di amici toscani e alle amicizie che sono rimaste immutate nonostante la lontananza fisica.

L'ultimo ringraziamento - ma non di certo per importanza! - lo vorrei riservare al Prof. Danelutto, che ha avuto la pazienza di seguirmi durante questo lavoro in modo puntuale e scrupoloso, consigliandomi e correggendomi.

Grazie a tutti per aver contribuito a far sì che raggiungessi questo importante traguardo.

Alessandra

Capitolo 1

Introduzione

La tendenza a programmare hardware sta prendendo sempre più piede anche tra gli appassionati, basti pensare a dispositivi come Arduino o Raspberry Pi; il lavoro realizzato ha l'obiettivo di descrivere il panorama attuale delle tecnologie hardware riprogrammabili, come gli FPGA. La loro diffusione è in costante aumento, sia per le loro caratteristiche di programmabilità che per la loro versatilità d'uso. È possibile impiegarli in ambiti diversi, da quelli più specifici, come l'elaborazione di segnali digitali, ad altri che sfruttano più in generale la loro capacità di prestarsi bene all'esecuzione di calcoli paralleli.

Le metodologie utilizzate per la programmazione di dispositivi FPGA sono raggruppabili in due tipologie: la prima, quella classica, utilizza linguaggi di descrizione dell'hardware a basso livello; la seconda, di più recente introduzione, cerca di descrivere l'hardware elevando il livello di astrazione (si potrebbe parlare di linguaggi di programmazione dell'hardware ad alto livello).

Entrambi i tipi di linguaggi permettono di descrivere il comportamento di un programma, creando uno schema che ne rappresenti il funzionamento e sia traducibile sull'hardware sfruttando le risorse che si hanno a disposizione. La differenza sta nella difficoltà di utilizzo (quindi nei tempi di sviluppo richiesti) e nell'accessibilità delle due tecnologie: l'approccio classico richiede ai programmatori una conoscenza profonda dei dettagli dell'hardware, cosa che invece viene in parte mascherata da un livello di astrazione maggiore. L'utilizzo di uno stile di programmazione più simile ai linguaggi ad alto livello permette a una fetta maggiore di sviluppatori di approcciarsi alla programmazione di FPGA.

Da una parte si hanno quindi linguaggi di descrizione dell'hardware, caratterizzati da un livello più o meno alto di astrazione, che consentono di definire moduli e descrivere le connessioni tra essi; dall'altra si è iniziato a lavorare alla creazione di framework che permettano di raggiungere un ulteriore ampliamento della fascia di sviluppatori in grado di accedere al-

la tecnologia degli FPGA. Tali framework consentono di sviluppare codice parallelo (utilizzando il linguaggio C per scrivere le funzioni da eseguire), delegando totalmente al compilatore tutti quei compiti che in realtà dipendono dalla struttura dell'hardware del dispositivo (che può essere un FPGA, ma anche una GPU o una CPU), e dunque risultano normalmente essere molto gravosi per il programmatore.

Capitolo 2

FPGA - Field Programmable Gate Array

La ricerca di piattaforme di esecuzione in grado di supportare parallelismo e concorrenza (per migliorare la performance dei programmi eseguiti) continua ad avere un notevole impatto. Il *parallelismo* sfrutta la possibilità di suddividere un problema iniziale in più sotto-problemi semplici e risolvibili in modo indipendente l'uno dall'altro; il risultato di ognuno di questi può essere calcolato nello stesso istante di tempo mediante l'uso di risorse diverse. La *concorrenza* permette di sovrapporre nel tempo l'esecuzione di più operazioni; l'impiego di risorse condivise tra più computazioni viene regolato mediante accessi in mutua esclusione. L'utilizzo di entrambi questi approcci computazionali unisce l'esecuzione contemporanea delle operazioni di un programma alla possibilità di iniziare ad eseguire un task anche se i precedenti non sono ancora conclusi. L'incremento della performance ottenibile in questo modo è una spinta alla continua evoluzione e ricerca nell'ambito.

Scegliere di implementare il software su un circuito integrato che ne rappresenti la traduzione hardware ha dei costi elevati, cui si può far fronte solo in caso di produzione di massa. I circuiti tradizionali, una volta configurati al momento della fabbricazione, mettono a disposizione un'architettura immutabile; ciò li rende adatti per la distribuzione su larga scala di programmi testati e funzionanti.

L'alternativa è creare delle configurazioni personalizzate a partire da componenti logici programmabili. Questo è l'approccio tipico di un **Field Programmable Gate Array** (FPGA). Tale tipologia di dispositivo risponde al problema dei costi permettendo la produzione di poche unità, usate come prototipi, e garantisce livelli di parallelismo simili a quelli dei circuiti integrati di tipo tradizionale. Il modello di programmazione di un dispositivo FPGA è basato su descrizioni di programmi di tipo **Register Transfer Level** (RTL). La progettazione diretta di un circuito è molto dispendiosa in termini di tempi di sviluppo; per questo motivo stanno emergendo nuovi

approcci, grazie ai quali è possibile utilizzare linguaggi ad alto livello per scrivere il software da utilizzare per gli FPGA. La configurazione per il dispositivo FPGA verrà poi prodotta automaticamente da un compilatore a partire dal programma ad alto livello. Questo approccio permette di astrarre dai dettagli della piattaforma sulla quale verrà eseguito il programma.

2.1 Cos'è un FPGA

La caratteristica principale di un dispositivo FPGA è quella di essere programmabile, cosa che lo rende adatto all'utilizzo come prototipo per testare il funzionamento di algoritmi diversi. Si è detto in precedenza che un dispositivo FPGA garantisce un certo livello di parallelismo. Ciò significa che i componenti base della sua architettura, combinati opportunamente, consentono di eseguire in parallelo tutte le parti del codice originale che sono state specificate come parallele dal programmatore. Il design dell'architettura di un FPGA può essere prodotto "a mano" oppure compilato a partire da un programma ad alto livello. Nel primo caso la decisione su quali e quante risorse utilizzare è a cura del programmatore, nel secondo è onere del compilatore. Un esempio di compilatore è il **Vivado High-Level Synthesis** (HLS), per modelli di FPGA della Xilinx. Esso, dato un software scritto in un sottoinsieme del C/C++, è in grado di derivarne una descrizione RTL. Si vedrà in seguito in che modo vengono tradotti i vari costrutti del linguaggio di programmazione e i componenti architetturali usati per farlo.

2.1.1 Componenti e architettura

L'architettura di qualsiasi dispositivo FPGA contiene i seguenti componenti, organizzati in una struttura di interconnessione regolare e configurabile:

- **Tabelle di look-up (LUT)**

  Il loro scopo principale è quello di implementare la logica delle operazioni, ma possono essere utilizzate anche come elementi di memoria ROM. Il vantaggio di usare le LUT come memorie distribuite, nonostante abbiano una piccola capacità, è che possono essere posizionate in qualsiasi punto del circuito fornendo un accesso veloce ai dati. Il loro contenuto è deciso al momento della configurazione del dispositivo; per questo motivo non possono essere modificate durante l'utilizzo dello FPGA. Dal punto di vista dell'implementazione, le LUT possono essere viste come delle tabelle di verità, in cui a ogni combinazione degli N ingressi corrisponde una diversa funzione logica. Dati N valori in ingresso, le locazioni di memoria accedute dalla tabella sono 2^N (numero di tutte le possibili combinazioni degli N ingressi), e le funzioni logiche implementate saranno in totale 2^{N^N} (al variare dei valori degli

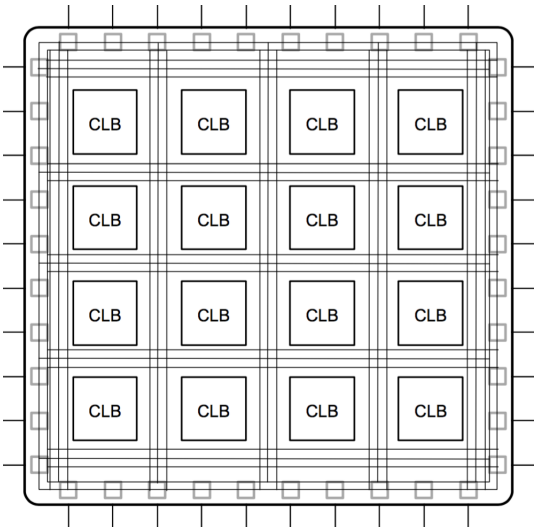


Figura 2.1: La struttura di un FPGA. I CLB (Configurable Logic Block) sono celle contenenti LUT, registri, ... I collegamenti avvengono mediante una rete di interconnessione configurabile.

N ingressi). Di solito ogni cella di un FPGA ha una LUT di pochi ingressi (possono variare dai 5 ai 10) e un bit di uscita.

- **Registri flip-flop (FF)**

  Sono capaci di memorizzare informazioni, per esempio possono mantenere dati generati durante l'esecuzione. Questi registri sono dotati di tre ingressi: uno per i dati, uno per il `clock` e uno per il segnale di abilitazione alla scrittura. Quando i segnali di `clock` e `enable` sono entrambi alti allora il valore in ingresso al registro viene salvato e reso disponibile in uscita, in caso contrario l'uscita non viene aggiornata. Il segnale di abilitazione permette quindi di mantenere un valore per più di un ciclo di `clock`, anziché effettuare una nuova scrittura a ogni ciclo. Di solito ogni cella di un FPGA contiene un registro flip-flop da un bit.

- **Connessioni (C)**

  Permettono di connettere i vari componenti del dispositivo, mettendoli in comunicazione e consentendo eventuali scambi di dati. Anche la configurazione del grafo delle connessioni avviene durante la fase di programmazione dello FPGA.

- **Porte per l'I/O**

  Sono utilizzate per comunicazioni e scambio di dati con l'esterno, per esempio con delle memorie.

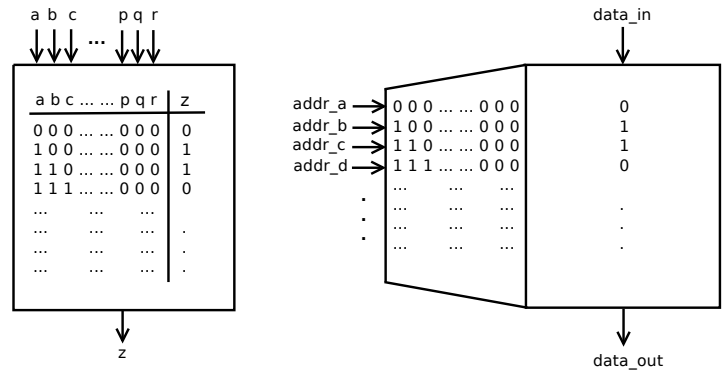


Figura 2.2: La struttura di una tabella di look-up. La LUT può essere pensata come una memoria indirizzabile mediante gli ingressi della tabella di verità. Nelle locazioni di memoria corrispondenti è contenuto il valore dell'uscita prodotta dalla tabella di verità.

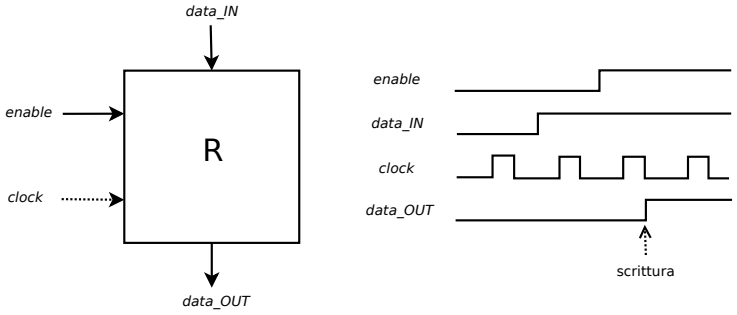


Figura 2.3: La struttura di un registro flip-flop.

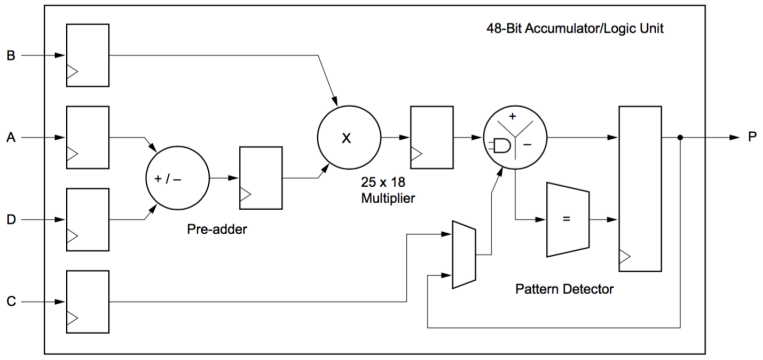


Figura 2.4: La struttura del blocco DSP48.

Col passare del tempo e l'evolversi della tecnologia, si è avuta la necessità di aumentare la capacità computazionale e la memoria disponibile su questi dispositivi. Per questo motivo, nell'architettura dei moderni FPGA si possono trovare altri componenti, in aggiunta a quelli elencati in precedenza. Questi possono essere:

- **Componenti per il calcolo**

  Sono specializzati nell'esecuzione di operazioni particolari e la loro struttura può essere anche abbastanza complessa. Un caso d'uso reale è il *blocco DSP48*, incluso in alcuni dispositivi FPGA della Xilinx, la cui importanza emerge nel campo dell'elaborazione dei segnali digitali; esso implementa vari tipi di operazioni aritmetiche, come l'accumulazione di prodotti o la somma di prodotti, e diversi tipi di operazioni logiche.

- **Componenti per la memoria**

  Sono in maggioranza moduli *Block-RAM* (BRAM). Un BRAM è una memoria dedicata (questo componente non può essere utilizzato per implementare altre funzionalità), utilizzata nei casi in cui sia necessario memorizzare un gran numero di dati. L'impiego di LUT in queste situazioni non è possibile poiché sono memorie ROM; in realtà, se anche si fossero comportate come memorie RAM, il loro utilizzo sarebbe stato sconsigliato a causa degli eccessivi ritardi che deriverebbero dal gran numero di connessioni necessarie tra le singole memorie distribuite. Allo stesso modo non è consigliato l'utilizzo di memorie BRAM per piccole quantità di dati, poiché lo spazio rimasto libero andrebbe sprecato.

  Tra gli ingressi di questi moduli vi sono un segnale di abilitazione della memoria (`enable`), il `clock`, un ingresso dati, un ingresso per l'indirizzo (indica la locazione che si vuole leggere o scrivere), più altri opzionali.

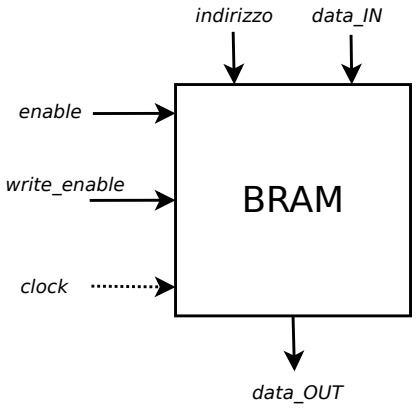


Figura 2.5: La struttura di una memoria BRAM.

L'accesso in memoria mediante operazioni di lettura e scrittura viene abilitato settando il valore dell'ingresso `enable`. Quando il modulo è attivo, qualsiasi operazione viene effettuata in modo sincrono col segnale di `clock`.

Tali moduli possono comportarsi sia come memorie RAM che come memorie ROM. Le due tipologie di memoria differiscono sul numero di scritture che è possibile effettuare; una memoria RAM è riscrivibile più volte (in ogni momento dell'esecuzione del circuito), mentre una memoria ROM può essere scritta un'unica volta (al momento della programmazione del dispositivo). La lettura dei dati è invece sempre possibile in entrambi i casi.

Un'operazione di scrittura viene abilitata, nel caso di una configurazione RAM, settando il valore dell'ingresso `write-enable`. Se la scrittura è consentita, i dati in ingresso vengono memorizzati nella locazione di memoria individuata dall'indirizzo. L'uscita dati assume valori diversi a seconda dell'implementazione della memoria: alcune supportano solo il *Read-After-Write* (in questo caso l'uscita assume come nuovo valore il dato appena scritto), altre permettono di scegliere, oltre alla *Read-After-Write*, tra altre due modalità di scrittura (*Read-Before-Write* fa assumere all'uscita il valore presente nella locazione di memoria prima della scrittura, *No-Read-On-Write* lascia invariato il valore in uscita).

Sui modelli FPGA della Xilinx si possono trovare sia moduli BRAM *single-port* (che permettono un singolo accesso alla memoria per ciclo di clock), sia moduli *dual-port* (che grazie a due differenti ingressi per gli indirizzi consentono l'accesso parallelo a due locazioni dello stesso componente di memoria).

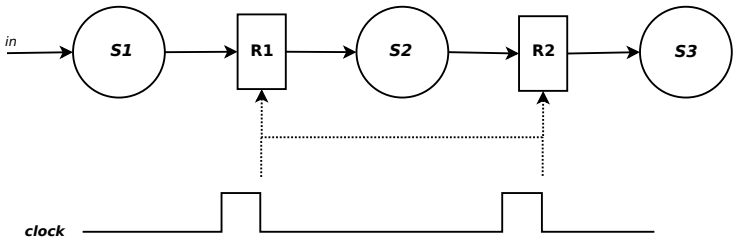


Figura 2.6: La struttura pipeline.

2.2 Il compilatore High-Level Synthesis

Il compilatore HLS è utilizzato sui modelli FPGA della Xilinx; il suo compito è partire da un algoritmo scritto in un sottoinsieme del C/C++ per poi generare un'architettura a livello RTL, basata sulle risorse disponibili. Ogni operazione del programma originale viene implementata mediante l'utilizzo di hardware dedicato; ciò, unito al fatto di avere un numero finito di risorse a disposizione, impone delle ottimizzazioni sul loro utilizzo e una ricerca continua del circuito migliore per l'algoritmo. Tale circuito viene ottenuto mediante i processi di scheduling, pipelining e dataflow.

Lo *scheduling* consiste nell'analizzare le dipendenze tra operazioni successive o sovrapposte nel tempo; per fare ciò si raggruppano quelle che possono essere eseguite nello stesso ciclo di clock e si crea dell'hardware adatto a permettere sovrapposizioni nell'esecuzione di computazioni diverse. In questo modo non è necessario che l'esecuzione di un'operazione sia vincolata alla fine di quella precedente sullo stesso insieme di dati.

Il *pipelining* permette di implementare le computazioni con dipendenze lineari sui dati e di aumentare il livello di parallelismo. Per fare ciò si divide la computazione in più stadi, che eseguono parallelamente durante lo stesso ciclo di clock; ognuno di essi riceve come dato in ingresso il risultato calcolato dallo stadio prima di lui al ciclo di clock precedente. Dopo un certo numero di cicli di clock, necessario per riempire il pipeline, si inizia a lavorare a regime, riuscendo a produrre un risultato ad ogni ciclo di clock.

Il modello *dataflow* consente l'esecuzione parallela di più funzioni in un programma. Mediante l'analisi degli ingressi e delle uscite delle funzioni, si è in grado di capire se tra esse esistono delle dipendenze; ciò permette di ricavare il massimo grado di parallelismo raggiungibile. Nel caso in cui le funzioni operino su insiemi di dati diversi, si potranno allocare risorse tali da consentire esecuzioni parallele e indipendenti. In caso contrario una funzione, per eseguire correttamente, avrà bisogno di avere il risultato prodotto da un'altra; questo rapporto produttore-consumatore si traduce, nel circuito, mediante una coda FIFO che implementa la connessione tra i due moduli funzione. L'esecuzione del consumatore può avvenire dopo che i dati siano stati generati tutti, oppure con l'utilizzo di risultati parziali dal produttore.

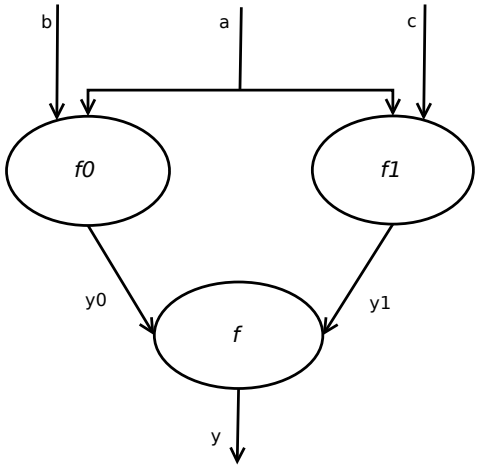


Figura 2.7: Il grafo dataflow di un modulo che opera su uno stream di triple di valori con tipo (a, b, c) e generera uno stream di valori $y = f(f_0(b, a), f_1(a, c))$. Tra le funzioni f_0 ed f_1 non ci sono dipendenze, mentre ognuna di esse è un produttore per la funzione f.

È fondamentale trovare un compromesso tra risorse utilizzate, complessità del pipeline, e numero di moduli indipendenti con esecuzione parallela.

2.2.1 Algoritmi Computation-Centric e Control-Centric

Gli algoritmi eseguibili da un dispositivo FPGA possono essere suddivisi in due categorie, a seconda del tipo delle operazioni da cui sono composti. Da esse dipende il comportamento durante l'esecuzione del task, corrispondente a una chiamata di funzione in un linguaggio ad alto livello.

Un algoritmo è *Computation-Centric* se la sua traduzione hardware determina una funzionalità non modificabile durante l'esecuzione del task corrispondente. Le operazioni definite vengono applicate a tutti i dati in ingresso ed eventuali modifiche ad esse possono essere apportate soltanto al termine del task.

Un algoritmo è *Control-Centric* se l'esecuzione delle sue operazioni dipende dai dati ricevuti in ingresso. È il caso in cui il programma contiene cicli, istruzioni condizionali o switch; questo tipo di istruzioni vincola l'esecuzione di un ramo del programma, piuttosto che quella di un altro, alla valutazione di guardie booleane. Questo è il motivo per cui non può essere fissato a priori l'insieme delle operazioni che verranno sicuramente eseguite; è necessario perciò implementare tutte le operazioni e la logica di controllo.

A seconda dei casi può essere più efficiente eseguire un algoritmo Control-Centric su un processore, piuttosto che su un FPGA; tale scelta dipende dal tempo di esecuzione richiesto e dalle risorse che sarebbero necessarie per un'implementazione su un FPGA. Se l'esecuzione richiede un gran numero di cicli di clock, è preferibile un processore; in questo caso l'implementazione

del circuito su un FPGA richiederebbe infatti un numero di risorse elevato. Se invece sono necessari pochi cicli di clock, è preferibile usare un FPGA; tale dispositivo infatti può riuscire a eseguire un maggior numero di operazioni nell'unità di tempo rispetto a un processore. Il caso medio comprende solitamente la situazione in cui l'algoritmo contiene una funzione implementabile come modulo hardware; in tale scenario il programma principale è eseguito su un processore e l'esecuzione della funzione è demandata a un modulo esterno, come un FPGA, che funge da co-processore per il suo calcolo.

2.3 Analisi della performance

Per analizzare la performance si può tracciare un parallelo tra il comportamento di un FPGA e quello di un processore tradizionale, e i loro rispettivi compilatori. Un compilatore per dispositivi FPGA e un compilatore per un processore tradizionale partono entrambi da un programma scritto in un linguaggio ad alto livello, come il C o il C++. Ad essere diversi sono ovviamente i risultati delle due compilazioni: nel primo caso viene generata un'architettura a livello RTL, nel secondo una sequenza di istruzioni eseguibili dal processore. Gli scopi dei due compilatori possono essere messi in contrapposizione: il compilatore di un FPGA deve generare un circuito che implementi l'algoritmo iniziale, mentre il compilatore di un processore deve di fatto tradurre il programma iniziale nel linguaggio macchina dell'architettura.

Nel caso di FPGA l'esecuzione di un programma avviene sempre mediante un circuito creato ad-hoc; perciò è possibile ottenere un alto livello di parallelismo tra le operazioni, anche con una frequenza di clock bassa. Nel caso di un processore l'insieme delle istruzioni viene processato tramite un ciclo di `fetch-decode-execute`, per il caricamento da memoria, la decodifica e l'esecuzione delle operazioni; la performance può essere migliorata utilizzando una maggiore frequenza del ciclo di clock. La frequenza del clock non è perciò una buona metrica per il confronto delle performance delle due tipologie di dispositivo; il raffronto può essere meglio espresso in termini di latenza, tempo di servizio, banda e considerazioni sull'organizzazione della memoria e sulla gestione dei costrutti di programmazione.

Per un FPGA, la *latenza* è espressa come il numero di cicli di clock necessari affinché sia generato un risultato, la *banda* è il numero medio di task eseguibili nell'unità di tempo; essa può essere anche espressa come l'inverso del *tempo di servizio*, definito come il numero di cicli di clock necessari affinché si possano accettare nuovi dati in input nel circuito. Una minore latenza e un incremento della banda conducono a una miglior performance.

Il processo di pipelining, sia sui processori che sui dispositivi FPGA, permette di migliorare la performance. Nell'architettura FPGA i componenti possono essere connessi direttamente o mediante registri; l'inserimento

di registri genera nuovi stadi nel pipeline, frammentando la computazione. L'aumento degli stadi del pipeline provoca una crescita della latenza (più cicli di clock per calcolare il risultato), ma porta a un incremento della banda. Spezzettando la computazione, ogni stadio del pipeline andrà a svolgere operazioni più semplici, impiegando meno tempo per calcolare i risultati: l'esecuzione parallela degli stadi conduce a una diminuzione del tempo di servizio e del ciclo di clock. Se il pipelining è completo, la logica del circuito viene suddivisa nel massimo numero di stadi, ognuno dei quali è impiegato per eseguire un task diverso. In questo scenario ogni stadio del pipeline sarà sempre occupato durante l'esecuzione, migliorando la banda (e consentendo così di accettare nuovi dati in input con una frequenza maggiore).

La performance viene influenzata anche dal tipo di *organizzazione dei dati in memoria*. Una parte delle istruzioni Assembler prodotte dal compilatore di un processore tradizionale si occupa del caricamento dei dati nei registri e del salvataggio in memoria dei risultati calcolati. L'ottimizzazione degli accessi alla memoria si traduce, per un processore, in una buona gestione della cache. Un FPGA non deve far fronte a questo problema: l'architettura viene generata in modo da garantire che i dati possano essere sempre acceduti con la massima banda durante l'esecuzione delle operazioni che li richiedono. Le memorie distribuite usate per questo scopo sono veloci e accessibili in modo indipendente e parallelo. La quantità di dati presenti nel circuito viene tenuta sotto controllo eseguendo le operazioni di memoria il più vicino possibile, in termini di stadi del pipeline, a quelle che consumano il dato. La limitazione di un FPGA sta nel fatto che, dovendo essere noto al compilatore quanta memoria utilizzare nel circuito, questa deve essere necessariamente allocata in modo statico; l'utilizzo di puntatori è comunque supportato, per riferire locazioni che si trovino in una memoria esterna al chip. Questa può essere acceduta mediante opportune interfacce, per esempio DDR.

Il circuito di un FPGA comprende sempre tutti i possibili rami di esecuzione di un programma; ciò è necessario non essendo possibile sapere a priori quale sarà eseguito. Un'istruzione condizionale viene ridotta alla selezione tra i risultati prodotti dai due rami d'esecuzione; le operazioni delle due funzioni avverranno in parallelo e sincronizzate in modo da produrre il risultato nello stesso ciclo di clock. In questo modo non si hanno degradazioni della performance dovute ad istruzioni di salto, come avviene nel caso di un processore.

I cicli vengono gestiti utilizzando una quantità di risorse tale da consentire l'esecuzione in pipeline delle varie iterazioni; verrà determinato un *intervallo di inizializzazione* (II) pari al numero di cicli di clock tra l'inizio di due iterazioni successive, ogni stadio del pipeline conterrà le operazioni del corpo del ciclo e una logica di controllo valuterà il numero di iterazioni necessarie a calcolare il risultato.

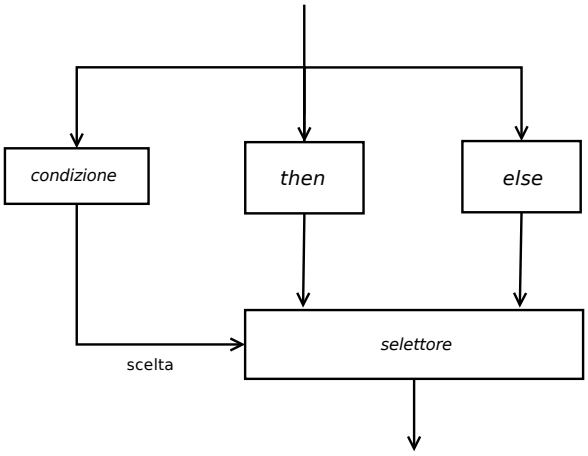


Figura 2.8: L'implementazione di un'istruzione condizionale su un FPGA.

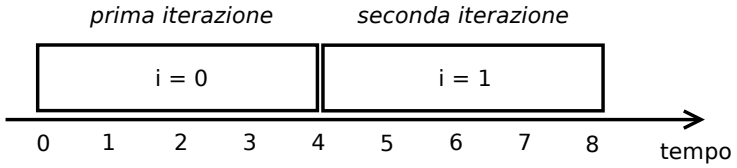


Figura 2.9: Un esempio di scheduling delle iterazioni di un ciclo su un processore.

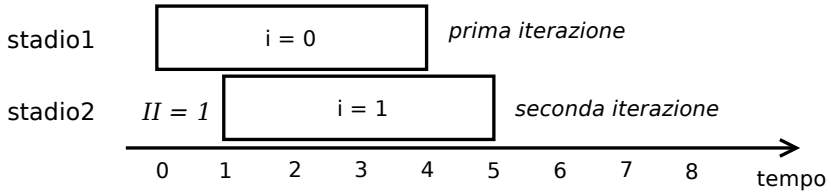


Figura 2.10: Un esempio di scheduling delle iterazioni di un ciclo su un FPGA, dove l'intervallo di inizializzazione è II = 1.

Capitolo 3

Metodologie di programmazione di un FPGA

Quando ci si avvicina al mondo dei dispositivi programmabili (come gli FPGA) ci si trova quasi subito a dover scegliere il metodo da utilizzare come strumento di sintesi sul dispositivo; ci si chiede quale sia l'approccio migliore da poter adottare per costruire l'hardware che implementi un certo algoritmo e permetta la sua esecuzione.

I modelli di programmazione tra cui poter scegliere sono da una parte gli **Hardware Description Language** (HDL), come il Verilog, dall'altra approcci che fanno uso di **linguaggi Orientati agli Oggetti** (OO), come Chisel (un DSL - domain specific language - implementato con il linguaggio Scala). Il punto cruciale sta nel riuscire a caratterizzare la parola "migliore", ossia in base a quali criteri un metodo può essere preferibile rispetto ad un altro, e quali sono i vantaggi e gli svantaggi di entrambi.

L'approccio classico di programmazione per i dispositivi FPGA è quello che utilizza i linguaggi HDL; Verilog e VHDL sono i due linguaggi più diffusi. La complessità odierna dei circuiti può essere ben utilizzata mediante programmi HDL: essi permettono di verificare la funzionalità di possibili implementazioni alternative per lo stesso circuito, tra le quali il programmatore dovrà aver cura di scegliere quella più performante.

Questo tipo di programmazione risulta ostico per certi versi; Verilog e VHDL sono nati come linguaggi per la simulazione dell'hardware e solo in un secondo momento sono diventati una base per la sua sintesi. La *simulazione* è il processo che permette di verificare il funzionamento e la performance di un sistema, che si sta sviluppando su una piattaforma che emuli un FPGA; la sua utilità è da contrapporre all'impossibilità di testare parti del sistema che si interfacciano direttamente con alcuni componenti del dispositivo e al maggiore tempo di calcolo richiesto. La *sintesi* è il processo che a partire da una specifica (descritta per esempio mediante un linguaggio HDL) genera un modello equivalente più a basso livello (in questo caso l'implementazione

hardware del programma).

Dalla natura degli HDL deriva il fatto che molti costrutti sono poco intuitivi da utilizzare, poiché richiedono di ragionare esclusivamente in termini di componenti hardware, non di flusso di esecuzione del programma; per questo motivo pian piano hanno iniziato ad emergere approcci alternativi. Lo scopo di tali approcci è quello di sviluppare nuovi linguaggi per la descrizione dell'hardware, più semplici dei linguaggi HDL e capaci quindi di offrire un nuovo modo di programmare i dispositivi FPGA. L'idea principale è quella di non programmare direttamente il circuito, bensì creare un suo generatore. Il *generatore di un circuito* è un programma capace di produrre automaticamente un design hardware a partire da un insieme di parametri (e loro valori) e vincoli ad alto livello. Esso può essere anche abbastanza complesso, poiché deve essere capace di prendere decisioni che conducano alla migliore ottimizzazione possibile del circuito. Lo scopo di questo tipo di programmi è alleggerire il lavoro del programmatore e aumentare il livello di astrazione, rendendo accessibile l'ambito a un maggior numero di persone.

Rientra in questo secondo approccio alla programmazione di FPGA il *Constructing Hardware In a Scala Embedded Language* (Chisel). Chisel è un linguaggio per la costruzione di hardware totalmente integrato col linguaggio orientato agli oggetti Scala. La scelta di Scala nella ricerca di un approccio ad un più alto livello consente di utilizzare alcuni costrutti mancanti su un HDL; Scala è un linguaggio orientato gli oggetti che incorpora diverse caratteristiche tipiche della programmazione funzionale, come l'inferenza di tipo. Tali proprietà si adattano bene al modello di programmazione parallela. Il paradigma di programmazione orientato agli oggetti (entità che incapsulano dati e operazioni) consente di manipolare in modo concorrente oggetti diversi. Il paradigma di programmazione funzionale è caratterizzato da un flusso di esecuzione del programma descritto mediante la valutazione di funzioni matematiche; la valutazione delle espressioni è indipendente dallo stato del programma, perciò è possibile valutare gli argomenti di una funzione in qualsiasi ordine. Ne consegue che gli argomenti di una funzione possono essere tutti valutati in modo parallelo. Le architetture data driven sfruttano questo paradigma; in esse l'ordine di esecuzione delle istruzioni si basa sulle dipendenze tra i dati (una istruzione viene eseguita solo se tutti gli operandi di cui essa necessita sono disponibili). I grafi dataflow rappresentano programmi per computazioni data driven.

Verilog e VHDL includono dei costrutti primitivi adatti alla generazione di circuiti, ma non sono abbastanza per bilanciare la mancanza di caratteristiche come quelle sopra elencate.

Questo secondo approccio sembra emergere sempre più ormai, a scapito della programmazione mediante HDL, permettendo a chi fa il design dell'hardware di essere più produttivo, lavorando a un livello di astrazione maggiore.

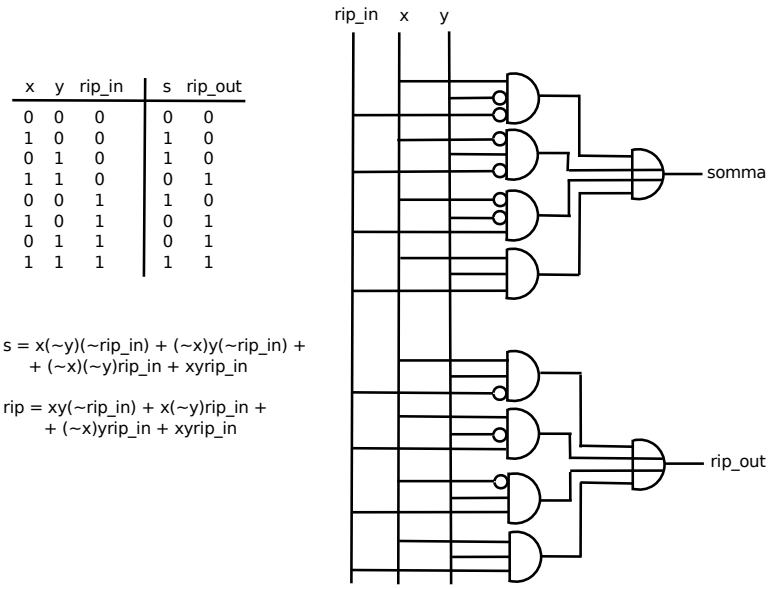


Figura 3.1: La tabella di verità che descrive un modulo addizionatore binario, da cui sono state ricavate le espressioni booleane in forma disgiuntiva (somma di prodotti) per l'uscita somma e per l'uscita riporto. Vengono mostrate anche le reti di porte logiche corrispondenti.

3.1 Esempio

È interessante vedere un utilizzo concreto dei due tipi di programmazione sopra descritti; per questo motivo si è adottata la scelta di partire da un esempio pratico.

In questa sezione viene mostrata l'implementazione di un modulo addizionatore sia mediante l'utilizzo di Verilog che di Chisel. Un addizionatore è un componente capace di effettuare la somma dei due valori in ingresso, restituendone il risultato e l'eventuale riporto. Un addizionatore binario effettua la computazione su ingressi da un bit, producendo un bit di uscita e un bit di riporto. È possibile generalizzare questa definizione a ingressi da più bit; in tal caso anche l'uscita calcolata sarà da più bit.

3.1.1 Implementazione Verilog

Una possibile implementazione di un modulo addizionatore binario in Verilog è la seguente, realizzata come rete di porte logiche. Il codice Verilog viene suddiviso in tre parti: la dichiarazione del modulo e dei suoi ingressi e uscite, fili e registri temporanei per i valori intermedi della computazione e assegnamenti a eventuali registri.

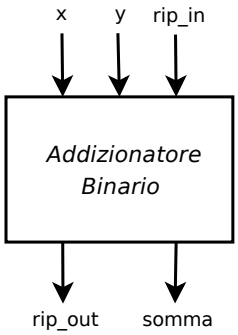


Figura 3.2: La struttura del modulo addizionatore binario. I tre ingressi e le due uscite sono tutti da un bit.

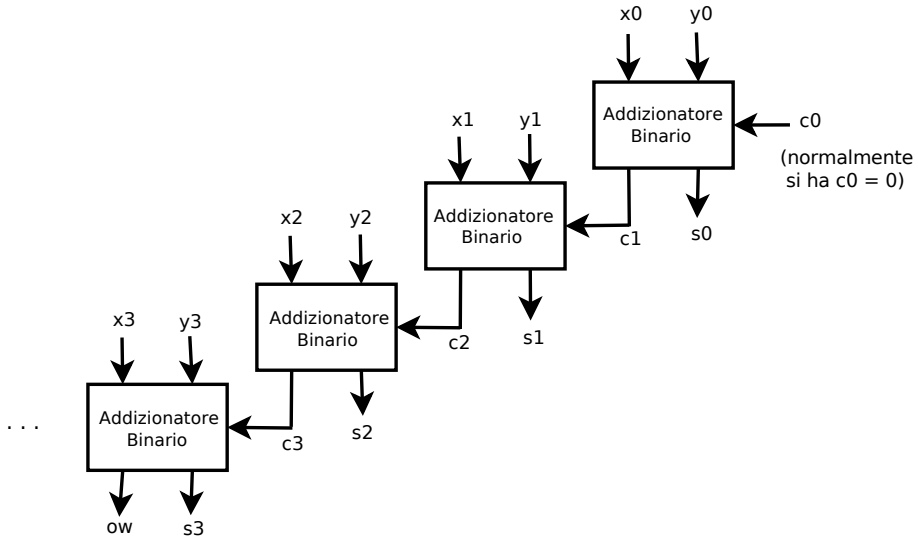


Figura 3.3: La struttura del modulo addizionatore a n bit, ricavata dalla composizione di più moduli addizionatore binario. Il riporto in uscita dell'addizionatore a destra viene passato come ingresso all'addizionatore successivo; il riporto prodotto dall'ultimo addizionatore nella catena sarà il bit di overflow.

Codice 3.1: Modulo addizionatore binario Verilog

```
module addizionatore_binario(somma, rip_out, x, y, rip_in);

  output somma;
  output rip_out;
  input x;
  input y;
  input rip_in;

  wire s;
  wire r1;
  wire r2;
  wire r3;

  // genera la somma
  xor(s, x, y);
  xor(somma, rip_in, s);

  // genera il riporto
  and(r1, x, y);
  and(r2, y, rip_in);
  and(r3, x, rip_in);
  or(rip_out, r1, r2, r3);

endmodule
```

Una sua generalizzazione con ingressi da $N = 32$ bit è la seguente, che sfrutta un costrutto particolare di Verilog, il `generate`. L'implementazione del modulo risulta dalla composizione di più moduli `addizionatore_binario`. Si può notare che il valore del riporto iniziale viene posto a 0, mentre i successivi riporti assumono il valore prodotto da ogni modulo addizionatore binario.

Codice 3.2: Modulo addizionatore n bit Verilog

```
module addizionatore_n_bit_generative(somma, rip_out, x, y);

  parameter N = 32;

  output [N-1:0]somma;
  output rip_out;
  input [N-1:0]x;
  input [N-1:0]y;

  reg [N, 0]rip;

  initial
    begin
      rip[0] = 0;
    end
```

```
16
17    genvar i;
18
19    generate
20      for(i = 0; i < N; i = i +1)
21        begin
22          addizionatore_binario
23            add(somma[i], rip[i+1], x[i], y[i], rip[i]);
24        end
25    endgenerate
26
27    assign rip_out = rip[N];
28
29  endmodule
```

3.1.2 Implementazione Chisel

L'implementazione in Chisel è illustrata nel codice seguente. Anche in questo caso il primo è un modulo addizionatore binario e il secondo è la sua generalizzazione con ingressi a n bit. Per definire un modulo Chisel è necessario importare le librerie che permettono di utilizzare Scala come linguaggio di costruzione dell'hardware. L'import avviene mediante la dichiarazione `import Chisel._`. Il componente che si sta implementando viene definito come una classe Scala, in modo molto simile alla dichiarazione di un modulo in Verilog.

Nell'implementazione del modulo `Addizionatore` a n bit vengono utilizzati tre vettori differenti: un vettore di moduli `AddizionatoreBinario`, un vettore per i riporti e un vettore per i bit di somma. Una particolarità è la dichiarazione di quest'ultimo come vettore di booleani; il motivo è il fatto che Chisel non consente l'assegnamento diretto dei singoli bit, perciò ogni bit di somma viene memorizzato come un valore booleano, per poi ricavare il valore finale della somma trasformando l'array prima in bit e poi effettuando un cast a UInt.

Codice 3.3: Modulo addizionatore binario Chisel

```
1  class AddizionatoreBinario extends Module {
2
3    val io = new Bundle {
4      val x = UInt(INPUT, 1)
5      val y = UInt(INPUT, 1)
6      val rip_in = UInt(INPUT, 1)
7      val somma = UInt(OUTPUT, 1)
8      val rip_out = UInt(OUTPUT, 1)
9    }
10
11
```

```
12    // genera la somma
13    val x_xor_y = io.x ^ io.y
14    io.somma := x_xor_y ^ io.rip_in
15
16    // genera il riporto
17    val x_and_y = io.x & io.y
18    val y_and_rip_in = io.y & io.rip_in
19    val x_and_rip_in = io.x & io.rip_in
20    io.rip_out := x_and_y | y_and_rip_in | x_and_rip_in
21
22  }
```

Codice 3.4: Modulo addizionatore n bit Chisel

```
1   class Addizionatore(val n:Int) extends Module {
2
3     val io = new Bundle {
4       val A = UInt(INPUT, n)
5       val B = UInt(INPUT, n)
6       val Cin = UInt(INPUT, 1)
7       val Sum = UInt(OUTPUT, n)
8       val Cout = UInt(OUTPUT, 1)
9     }
10
11    val adds = Vec.fill(n){ Module(new AddizionatoreBinario()).io }
12    val carry = Vec.fill(n+1){ UInt(width = 1) }
13    val sum   = Vec.fill(n){ Bool() }
14
15    // il primo riporto e' il valore Cin
16    carry(0) := io.Cin
17
18    for (i <- 0 until n) {
19      adds(i).x := io.A(i)
20      adds(i).y := io.B(i)
21      adds(i).rip_in := carry(i)
22      carry(i+1) := adds(i).rip_out
23      sum(i) := adds(i).somma.toBool()
24    }
25
26    io.Sum := sum.toBits().toUInt()
27    io.Cout := carry(n)
28
29  }
```

Una delle caratteristiche più potenti di Chisel è la capacità di generare il codice Verilog per un FPGA a partire dai componenti Scala. La generazione avviene tramite un processo di compilazione. Una volta pagato il costo di compilazione si ottengono due implementazioni dello stesso modulo funzionalmente identiche, perciò i due programmi possono essere considerati

equivalenti. Di seguito viene mostrato il codice Verilog prodotto a partire dal modulo Chisel `AddizionatoreBinario`.

Codice 3.5: Modulo addizionatore binario Verilog generato a partire dal modulo Chisel

```
module AddizionatoreBinario(
    input  io_x,
    input  io_y,
    input  io_rip_in,
    output io_somma,
    output io_rip_out
);

  wire T0;
  wire x_and_rip_in;
  wire T1;
  wire y_and_rip_in;
  wire x_and_y;
  wire T2;
  wire x_xor_y;


  assign io_rip_out = T0;
  assign T0 = T1 | x_and_rip_in;
  assign x_and_rip_in = io_x & io_rip_in;
  assign T1 = x_and_y | y_and_rip_in;
  assign y_and_rip_in = io_y & io_rip_in;
  assign x_and_y = io_x & io_y;
  assign io_somma = T2;
  assign T2 = x_xor_y ^ io_rip_in;
  assign x_xor_y = io_x ^ io_y;

endmodule
```

Il codice del modulo `Addizionatore` per ingressi a N bit, ricavato dal Chisel, non viene riportato qui ma in Appendice A, poiché eccessivamente lungo. Non si utilizza l'approccio iterativo (stile *generative* di Verilog), per questo motivo la quantità di codice necessaria all'implementazione cresce col crescere del parametro N. Viene dichiarato un gran numero di wire per i valori dei risultati intermedi; un certo numero di assegnamenti continui aggiorna il valore dei fili ogni volta che cambia quello dell'espressione sulla destra; infine il codice esplicita le N istanziazioni dei moduli `AddizionatoreBinario`, che hanno come parametri attuali i rispettivi fili definiti e assegnati in precedenza.

3.1.3 Confronto tra i due approcci

A partire dagli esempi riportati possono essere effettuate alcune considerazioni sui due strumenti di sintesi su FPGA a confronto.

La somiglianza tra la struttura del codice Verilog (nativo) e quello Chisel appare abbastanza evidente. C'è quasi una corrispondenza uno a uno tra le due implementazioni date per il corpo del modulo addizionatore binario e quello dell'addizionatore a n bit. Per il primo modulo Verilog utilizza in modo esplicito le porte logiche, cosa che si traduce in Chisel nell'utilizzo dei rispettivi operatori logici. Per il secondo modulo Verilog usa un approccio generativo, che compone più addizionatori binari insieme; anche Chisel utilizza un approccio simile, cambiando solo la sintassi: i moduli vengono inseriti in un *Vecs*, consentendo così di iterare su di essi e assegnare alle porte di ognuno le rispettive connessioni. Il parametro `N` (che determina il numero di moduli binari da istanziare) può essere modificato in entrambi i casi al momento dell'istanziazione del modulo.

La dichiarazione esplicita degli ingressi e delle uscite del modulo Verilog si riflette in Chisel nella creazione di un'interfaccia (il *Bundle* `io`) che contiene tutti gli ingressi e le uscite. Tale interfaccia viene intesa come costruttore del modulo, poiché ne esplicita tutte le interazioni. Nella classe `Addizionatore` definita, si può notare infatti che ogni addizionatore binario è creato come `new AddizionatoreBinario.io`.

Per quanto riguarda il codice Verilog ottenuto a partire dall'implementazione Chisel, si può notare che i suoi ingressi e le uscite sono ricavati con una corrispondenza diretta dai campi dell'interfaccia del modulo Chisel. Il corpo del modulo risulta un po' più verboso rispetto all'implementazione Verilog nativa (viene utilizzato un numero maggiore di componenti) ma le operazioni logiche effettuate sono le stesse.

Si può dire quindi che le implementazioni elencate sopra sono tutte funzionalmente equivalenti e con la stessa complessità di calcolo; a cambiare è solamente la sintassi dei costrutti utilizzati. Ciò significa che Chisel è in grado di generare del codice Verilog senza nessun overhead (a essere pagato è solo il costo di compilazione).

L'istanziazione dei moduli è equivalente sia in Chisel che in Verilog. In Chisel si utilizza la parola chiave di Scala `new`, assegnando poi l'oggetto creato a una variabile per poter essere riferito successivamente. Se al momento dell'istanziazione non vengono connesse in modo corretto le porte di ingresso e uscita, il compilatore potrebbe decidere di fare delle ottimizzazioni eliminando delle porzioni di circuito che ritiene non necessarie, riportando errori o warning.

3.2 Panoramica su Verilog

L'importanza degli Hardware Description Language è correlata alla capacità di esprimere circuiti in modo diretto mediante moduli e connessioni tra essi. La modellazione avviene a livello di fili, porte logiche, operazioni sui bit... Il Verilog appartiene a questa famiglia di linguaggi.

I tipi di dato messi a disposizione sono costanti, fili, registri, array e variabili generiche. Più in dettaglio:

- **Costanti**

  Sono valori numerici rappresentabili con una notazione che specifica il numero di bit in decimale (`<n>`), la base (`<b>`) e il numero espresso in quella base (`xxxx`). La forma assunta dalle costanti (dette anche *literal*) è la seguente: `<n>'<b>xxxx`.

- **Fili**

  Sono i collegamenti che permettono di connettere i vari componenti. La loro dichiarazione, mediante l'utilizzo della parola chiave `wire`, può specificare fili singoli (un solo bit) oppure gruppi di fili. Il loro numero è indicato tra parentesi in questo modo `[a:b]`. Tale notazione permette anche di identificare i singoli bit, riferendo con `a` il più significativo e con `b` il meno significativo.

- **Registri**

  Seguono le stesse convenzioni usate per i fili (se non viene specificata una dimensione sono considerati da un bit, altrimenti essa andrà indicata in modo esplicito). Vengono identificati mediante la parola chiave `reg`.

- **Array**

  Per dichiarare un array è necessario un tipo base (di uno o più bit) e una dimensione per il vettore stesso.

- **Integers**

  Sono variabili generiche. Di fatto possono essere viste come registri il cui valore viene interpretato come un intero con segno.

La panoramica sui tipi di dato permette di identificare i tasselli base con cui si va a definire un modulo. Sui dati sono definiti vari operatori, sia aritmetici che logici, con le rispettive regole di precedenza.

Un generico componente può essere costruito in modo simile a una dichiarazione di procedura, utilizzando il costrutto `module`. Il modulo sarà caratterizzato da un nome, una lista di parametri formali e un corpo.

```
module <nome> (<lista parametri formali>);
  <corpo>
endmodule
```

Al momento della sua istanziazione verranno utilizzati opportuni parametri attuali.

```
<nome> <nome istanza>(<lista parametri attuali>);
```

Ogni parametro viene dichiarato come un ingresso o un'uscita del modulo. In fase di istanziazione si ha la possibilità di ridefinire alcuni parametri; tale meccanismo è utile qualora avessimo definito in modo generico il componente e volessimo decidere di volta in volta il numero di bit dei suoi ingressi.

Un modulo può essere definito mediante diversi tipi di comandi e blocchi di comandi. Un blocco marcato come `initial` verrà eseguito solamente alla partenza della simulazione, mentre un blocco marcato come `always` verrà eseguito continuamente (se si utilizza il modificatore `@(<lista variabili>)` il blocco sarà eseguito ogni volta che una delle variabili in lista cambia valore). Tra i comandi sono presenti i classici `for` e `while` per gestire i cicli, il condizionale e la scelta multipla. Una particolarità si ritrova nel comando di assegnamento. Esso può essere *bloccante* (simbolo `=`) se termina prima che venga eseguita l'istruzione successiva, *non bloccante* (simbolo `<=`) se viene eseguito contemporaneamente ad altri assegnamenti, oppure *continuo* (simbolo `assign <parte sinistra> = <parte destra>`) se ogni volta che cambia la parte destra essa viene rivalutata e assegnata alla parte sinistra.

Per modellare realisticamente i ritardi dei componenti è possibile utilizzare la sintassi `#<ritardo>`, indicando il numero di unità di tempo necessarie per produrre il risultato.

Per testare i componenti implementati è possibile definire dei moduli di test, privi di parametri formali; essi dichiarano al loro interno registri per gli ingressi e fili per le uscite del modulo da testare (verranno passati come parametri attuali al momento dell'istanziazione del modulo). Modificando i registri utilizzati come ingressi è possibile monitorare il comportamento del modulo e analizzarne la correttezza leggendo i valori prodotti sui fili in uscita: il controllo della simulazione avviene mediante l'uso di comandi appositi, come `dumpvars`, `monitor` o `display`.

3.2.1 Definizione di componenti

Un modulo modella una rete logica; il tipo di tale rete determina il modo in cui viene implementato il modulo stesso. Una rete può essere definita come combinatoria o sequenziale. Di seguito viene data una panoramica di entrambi i tipi.

Reti combinatorie

Una rete combinatoria è una rete logica che ha n ingressi binari $x_1, \ldots, x_n$ ed m uscite binarie $z_1, \ldots, z_m$. Ad ogni combinazione degli n ingressi corrisponde una e una sola combinazione dei valori delle uscite: di fatto tale tipo di rete modella una funzione logica.

Un componente di questo tipo viene definito in Verilog mediante un costrutto `primitive`, una rete di porte logiche oppure in modo *behavioural*. Più in dettaglio:

- **Modulo primitive**

  È un modulo il cui corpo è una tabella di verità, definita in modo esplicito mediante un blocco table. Gli ingressi e l'uscita (una sola) di questo tipo di modulo sono tutti da un bit e sono in corrispondenza diretta con gli ingressi e l'uscita della tabella di verità (l'ordine dei parametri in ingresso è influente).

```
primitive <nome> (<lista parametri formali>);
  table
    <definizione tabella di verita'>
  endtable
endprimitive
```

- **Rete di porte logiche**

  È un modulo il cui corpo definisce una rete di porte logiche opportunamente collegate tra loro; l'insieme di tali porte implementa la rete logica.

```
module <nome> (<lista parametri formali>);
  <struttura della rete in termini di porte
    predefinite and, or, not, ...>
endmodule
```

- **Modulo behavioural**

  È un modulo nel quale le uscite (potenzialmente più di una e da più bit) vengono calcolate utilizzando comandi e operatori. Questo tipo di modulo fornisce direttamente l'espressione dell'algebra booleana ricavata dalla rete logica anziché utilizzare in modo esplicito le porte logiche. L'utilizzo di assegnamenti continui consente di aggiornare il valore dell'uscita (o delle uscite) con quello calcolato dall'espressione (a partire dagli ingressi del modulo).

```
module <nome> (<lista parametri formali>);
  ...
  assign z = <espressione booleana per la rete logica>;
endmodule
```

Si mostra di seguito un esempio per i tipi di implementazione presentati, prendendo come riferimento il modulo `addizionatore_binario` definito in precedenza. Nel primo Codice il modulo è descritto dalla sua tabella di verità, nel secondo si utilizza l'approccio behavioural; l'implementazione mediante rete di porte logiche è quella riportata nel Codice 3.1.

Codice 3.6: Addizionatore binario (versione che utilizza tabelle di verità)

```
primitive addizionatore_binario(somma, rip_out, x, y, rip_in);

  output somma;
  output rip_out;
  input x;
  intup y;
  intput rip_in;

  table
    0 0 0 : 0 0;
    1 0 0 : 1 0;
    0 1 0 : 1 0;
    1 1 0 : 0 1;
    0 0 1 : 1 0;
    1 0 1 : 0 1;
    0 1 1 : 0 1;
    1 1 1 : 1 1;
  endtable

endprimitive
```

Codice 3.7: Addizionatore binario (versione behavioural)

```
module addizionatore_binario(somma, rip_out, x, y, rip_in);

  output somma;
  output rip_out;
  input x;
  intup y;
  intput rip_in;

  assign somma = x ^ y ^ rip_in;
  assign rip_out = (x & y) | (x & rip_in) | (y & rip_in);

endmodule
```

La definizione di un modulo che rappresenti una funzione su ingressi a più bit può essere effettuata mediante un approccio *behavioural* oppure affiancando tanti moduli `primitive` da un bit quanti sono i bit degli ingressi. Ogni modulo, indipendentemente da come viene definito, può essere utilizzato dichiarandone una istanza, ossia utilizzando il nome della definizione

e poi assegnando un nome all'istanza e passando una lista di parametri attuali. Verilog mette a disposizione il comando `generate` per permettere la definizione di un modulo complesso componendo più istanze di moduli con ingressi da un bit. La generazione di tali istanze viene regolata mediante l'utilizzo di una variabile generativa che funge da indice.

Lavorando con ingressi a più bit si può sfruttare il meccanismo dei parametri di Verilog; esso consente di impostare ad ogni istanziazione del modulo il valore del parametro che indica il numero di bit da utilizzare.

Reti sequenziali

Una rete sequenziale è una rete logica con uno stato interno, modellabile mediante un automa a stati finiti. Tale automa è una macchina caratterizzata da un certo numero di variabili logiche in ingresso, di variabili logiche in uscita e di variabili logiche dello stato interno. Le transizioni tra gli stati sono controllate da una funzione di transizione dello stato interno $\sigma : X \times S- > S$, che calcola il nuovo stato in base all'ingresso e allo stato corrente. I valori in uscita sono generati da una funzione delle uscite $\omega : X \times S- > Z$, che calcola il nuovo valore in base all'ingresso e allo stato corrente.

Ogni variazione avviene in corrispondenza degli istanti di una sequenza temporale discreta $t_0, \ldots, t_n$. Vengono definiti allora due modelli di automa, di Mealy e di Moore. Per entrambi lo stato interno al tempo $t + 1$ della sequenza temporale dipende sia dagli ingressi che dallo stato al tempo t

$$S(t + 1) = \sigma(X(t), S(t)).$$

La funzione delle uscite cambia: nel caso di un automa di Mealy l'uscita al tempo t dipende sia dagli ingressi che dallo stato al tempo t

$$Z(t) = \omega(X(t), S(t))$$

mentre per un automa di Moore dipende dal solo stato interno al tempo t

$$Z(t) = \omega(S(t)).$$

Da tale definizione ne deriva che per implementare un modulo che calcoli una rete sequenziale si ha bisogno di memorizzare lo stato interno ad ogni istante di tempo e di implementare le due funzioni σ e ω. Lo stato può essere mantenuto da un registro, impulsato dal segnale di clock (tutta la rete è sincrona). La definizione delle funzioni dipende dall'approccio che si sceglie. Utilizzando un approccio composizionale si può realizzare un modulo che definisca al suo interno un modulo per il registro (lo stato) e i moduli σ e ω come reti combinatorie; i parametri formali conterranno gli ingressi, il `clock` e le uscite. L'aggiornamento del registro di stato col valore calcolato dalla σ e del registro in uscita col valore calcolato dalla ω dovranno essere eseguiti mediante assegnamenti non bloccanti (in parallelo ad altri nel blocco)

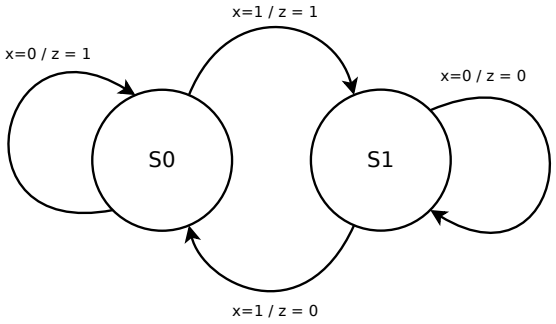


Figura 3.4: Un automa di Mealy con due stati e il seguente comportamento: in $S0$ se riceve uno 0 manda in uscita un 1 (senza cambiare stato), se riceve un 1 manda in uscita un 1 e transita nell'altro stato; in $S1$ se riceve uno 0 manda in uscita uno 0 (senza cambiare stato), se riceve un 1 manda in uscita uno 0 e transita nell'altro stato.

a ogni fronte di discesa del clock (statement `always @(negedge clock)`). L'alternativa è un approccio behavioural che consiste nel definire un modulo che implementi al suo interno sia la funzione σ che la ω.

Codice 3.8: Implementazione del modulo registro

```
1  module registro(r, clk, beta, i0);
2
3    parameter N = 32;
4
5    output [N-1:0]r;
6    input clk;
7    input beta;
8    input [N-1:0]i0;
9    reg [N-1:0]registroN;
10
11   initial
12   begin
13     registroN = 0;
14   end
15
16   always @ (negedge clk)
17     begin
18       if(beta==1)
19         registroN = i0;
20     end
21
22   assign r = registroN;
23
24 endmodule
```

Di seguito si mostrano le due possibili implementazioni per la rete sequenziale relativa all'automa di Mealy in Figura 3.4. Per l'implementazione composizionale è necessario definire i moduli per la rete σ e per la rete ω (sono reti combinatorie) e occorre utilizzare un registro per lo stato (in questo caso è sufficiente un registro interno da un bit, altrimenti si può utilizzare un modulo registro definito come nel Codice 3.8). L'implementazione behavioural è un modulo con due ingressi (l'ingresso vero e proprio e il segnale di clock) e una singola uscita (che rappresenta l'uscita dell'automa): ingressi, uscita e stato interno sono tutti componenti da un solo bit.

Codice 3.9: Implementazione dell'automa di Mealy mediante approccio composizionale

```
1  primitive automa_sigma(output z, input s, input x);
2    table
3      0 0 : 0;
4      0 1 : 1;
5      1 0 : 0;
6      1 1 : 1;
7    endtable
8
9  endprimitive
10
11 primitive automa_omega(output z, input s, input x);
12   table
13     0 0 : 1;
14     0 1 : 1;
15     1 0 : 0;
16     1 1 : 0;
17   endtable
18
19 endprimitive
20
21 module automa_mealy(output reg z, input x, input clock);
22
23   reg stato;
24   wire uscita_omega;
25   wire uscita_sigma;
26
27   automa_omega omega(uscita_omega, stato, x);
28   automa_sigma sigma(uscita_sigma, stato, x);
29
30   initial
31     begin
32       stato = 0;
33       z=0;
34     end
35
36
```

```
37    always @(negedge clock)
38      begin
39        stato <= uscita_sigma;
40        z <= uscita_omega;
41      end
42
43  endmodule
```

Codice 3.10: Implementazione dell'automa di Mealy mediante approccio behavioural

```
1   module automa_mealy(output reg z, input x, input clock);
2
3     reg stato;
4
5     // codifica degli stati
6     `define S0 0
7     `define S1 1
8
9     initial
10      begin
11        stato = `S0;
12      end
13
14    always @(negedge clock)
15      begin
16        case(stato)
17          `S0:
18            begin
19              stato <= (x==0 ? `S0 : `S1);
20              z <= 1;
21            end
22          `S1:
23            begin
24              stato <= (x==0 ? `S1 : `S0);
25              z <= 0;
26            end
27        endcase
28      end
29
30  endmodule
```

3.3 Panoramica su Chisel

La programmazione di dispositivi FPGA ad un livello di astrazione più alto dei singoli bit e componenti base porta a tempi di sviluppo più brevi e ad una maggiore accessibilità della tecnologia. Questo presupposto ha animato i creatori di Chisel, che fonde costrutti adatti alla progettazione dell'hardware con le proprietà del linguaggio Scala.

Chisel consente di lavorare con la logica binaria; esso definisce propri tipi di dato (che vanno ad aggiungersi ai tipi di dato di Scala) utilizzati per specificare il tipo dei valori contenuti nei registri o che passano sui fili. Il tipo *Bits* rappresenta una collezione di bit, che evita la manipolazione diretta di array di bit. I tipi *SInt* e *UInt* rappresentano interi con segno e senza segno rispettivamente. Il tipo *Bool* rappresenta i valori booleani. Oltre a questi tipi di dato semplici Chisel definisce i tipi *Bundle* e *Vecs*, che permettono di aggregare i tipi base. Un Bundle viene utilizzato per collezioni di valori non omogenei (può avere più campi che contengono valori di tipi diversi, ognuno associato a un nome, come una Struct). Si può definire un proprio bundle estendendo la classe Bundle. Un Vecs è una collezione omogenea di elementi riferibili mediante un indice. Per esprimere i valori costanti (*literals*) si specifica il valore mediante i tipi Int e String di Scala, per poi passarlo al costruttore del tipo base Chisel (SInt, UInt, Bool). Il compilatore è capace di ottimizzare il numero di bit necessari a contenere un valore costante. La dimensione di ogni literal può essere specificata esplicitamente, altrimenti viene calcolata come il minor numero di bit capace di esprimere il valore. Le classi primitive (SInt, UInt, Bool) e quelle aggregate (Bundle, Vecs) ereditano tutte dalla superclasse *Data*. Ogni oggetto il cui tipo sia un sottotipo di Data può essere rappresentato a livello hardware come un array di bit.

Per indicare porzioni di codice riutilizzabili più volte è possibile definire delle funzioni mediante la parola chiave di Scala `def`.

```
def nomefunzione(<lista parametri>) : <tipo di ritorno>;
```

Una funzione prende in ingresso una lista di parametri (con i rispettivi tipi) e restituisce come uscita un filo col valore del circuito booleano.

Definiti tutti i componenti primitivi è possibile metterli assieme per andare a comporre moduli di vario tipo.

3.3.1 Come definire un modulo

I moduli Chisel sono molto simili ai moduli Verilog. Essi sono organizzati secondo una gerarchia nel circuito, definita in base alle relazioni reciproche tra ingressi e uscite. La definizione di un modulo corrisponde a quella di una classe che eredita dalla superclasse *Module*; ognuno di essi contiene un'interfaccia (per la comunicazione con l'esterno) e fili per connettere più circuiti (nel proprio costruttore).

L'interfaccia di un modulo è una collezione di porte, realizzata mediante un Bundle; la convenzione è quella di indicare l'interfaccia mediante una variabile `io`. Ogni porta è un oggetto di tipo Data (o sottotipo) per il quale viene esplicitata una direzione (che può essere `INPUT` o `OUTPUT`) e in modo facoltativo il numero di bit.

La gerarchia viene costruita connettendo più moduli semplici mediante l'utilizzo di fili, da collegare in modo opportuno alle porte delle loro interfacce; queste connessioni consentono di derivare per composizione nuovi moduli più complessi. Una volta definito il modulo, il circuito corrispondente viene prodotto chiamando il metodo `chiselMain`; la fase di test avviene utilizzando vettori di test e la classe *Tester*: tale classe collega il tester al modulo sotto esame e fornisce una serie di comandi per controllare man mano il processo di verifica. Il comando `poke` permette di settare i valori delle porte di ingresso, `step` consente di eseguire il circuito per una unità di tempo, `peek` di leggere i valori di porte e registri e `expect` di confrontare i valori prodotti dal circuito con quelli attesi. L'istanza del test viene eseguita in un processo separato, a cui si passano gli ingressi e da cui si ricevono i risultati prodotti.

3.3.2 Componenti e costrutti

Anche Chisel mantiene lo stato di una computazione in componenti registro; ognuno di essi è definito con la parola chiave `Reg`. La dichiarazione

```
val reg = Reg(next = in)
```

istanzia un registro nel quale il valore dell'uscita è una copia del valore in ingresso al ciclo di clock precedente; la dichiarazione

```
val reg = Reg(init = UInt(0))
```

istanzia un registro e ne inizializza il valore a 0 (notare che UInt è il costruttore del tipo Chisel corrispondente, mentre 0 è un valore del tipo Int di Scala); il tipo del registro viene inferito automaticamente dall'ingresso indicato al momento della sua istanziazione (nel primo caso da in, nel secondo caso dal literal UInt(0)).

I segnali di `clock` e `reset` sono trattati come globali, quindi vengono inclusi implicitamente dove necessari.

Per descrivere le operazioni che interessano i registri è utile indicare le condizioni sotto le quali avvengono gli aggiornamenti dei loro valori. Chisel adotta una regola per l'aggiornamento basata sul costrutto condizionale `when`, il cui argomento è un'espressione che calcola un booleano. Il blocco del when può contenere statement di assegnamento, espressioni semplici e variabili che indicano fili. In una sequenza di aggiornamenti ha effetto quello relativo all'ultimo ramo when con condizione soddisfatta; ogni aggiornamento viene ridotto a una scelta dipendente dal valore dell'espressione nel when: se questa è falsa si mantiene invariato il valore dell'ingresso del registro, altrimenti lo si modifica. È necessario impostare un valore di default per gestire

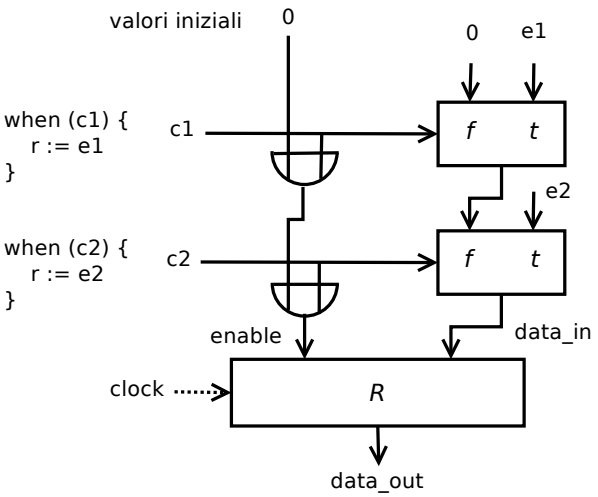


Figura 3.5: Il funzionamento dell'assegnamento condizionale.

il caso in cui nessuna condizione nella catena di aggiornamento condizionale risulti vera; in questa eventualità infatti non verrebbe mai assegnato un valore al registro (lasciandolo con un contenuto indefinito). Le condizioni per i costrutti di aggiornamento condizionale possono essere composte o concatenate (creando una sequenza) con l'utilizzo delle parole chiave `when`, `.elsewhen`, `.otherwhise` (corrispondenti a `if-else`, `if` e `else` in Scala).

Sia le memorie ROM che le memorie RAM possono essere definite e utilizzate da Chisel. Una memoria ROM viene definita utilizzando il tipo Vecs; la dichiarazione

```
val m = Vecs(inits: Seq(T))
```

inizializza la memoria con una sequenza di dati di tipo T. Una memoria RAM viene definita mediante il costrutto *Mem*, che rappresenta un'astrazione adattabile sia alla descrizione behavioural di una memoria in Verilog, sia a moduli di memoria descritti in modo diverso.

Chisel consente anche di avere delle classi interfaccia da utilizzare nella definizione di un modulo complesso; ogni interfaccia è definita come Bundle e può far parte di una gerarchia (ottenuta mediante l'uso di ereditarietà). La comodità di tali classi sta nella possibilità di riuso dell'interfaccia e nel supporto di *connessioni bulk* tra produttore e consumatore; tali connessioni permettono di collegare tra loro le porte di moduli diversi con lo stesso nome (un caso possibile è quello in cui due moduli hanno la stessa interfaccia, ma le direzioni delle porte sono invertite, perciò per uno una porta è di `INPUT` e per l'altro è di `OUTPUT`).

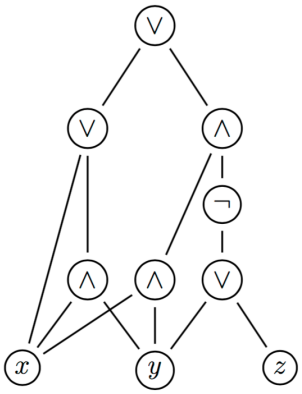


Figura 3.6: Un circuito booleano che rappresenta l'espressione booleana $(x \vee (x \wedge y)) \vee ((x \wedge y) \wedge \neg(y \vee z))$; esso contiene due copie del circuito corrispondente all'espressione $(x \wedge y)$.

3.3.3 Circuiti booleani

Un modulo Chisel definisce un *circuito booleano*, ossia un grafo che rappresenta un'*espressione booleana*; tali circuiti permettono di descrivere le funzioni logiche rappresentate dalle espressioni booleane e possono quindi rappresentare reti combinatorie.

Un circuito booleano è definito formalmente come un grafo diretto aciclico (N, A), i cui nodi $1, \ldots, n$ sono detti porte e i cui archi sono rappresentati come coppie ordinate (i, j). Le porte hanno da zero a due ingressi e sono di sorta (tipo) $s(i) \in \{tt, ff, \neg, \vee, \wedge\} \cup X$, dove X è l'insieme delle variabili; gli ingressi i del circuito sono le porte di sorta $s(i) \in \{tt, ff\} \cup X$ (non hanno ingressi) e l'uscita del circuito è denotata per convenzione dalla porta n (senza uscite). Se $s(i) = \neg$ la porta i ha un solo ingresso, altrimenti se $s(i) \in \{\vee, \wedge\}$ la porta i ha due ingressi. Una volta definito il circuito è necessario trovare un assegnamento di valori di verità che leghi ogni sua variabile a un valore in $\{tt, ff\}$. Il valore di verità per ogni nodo interno viene calcolato in modo induttivo a partire dai valori delle porte e da regole che definiscono il valore per ogni operatore; il processo termina col calcolo del valore della porta di uscita. La formalizzazione data può essere generalizzata permettendo più uscite.

La rappresentazione grafica di un circuito evidenzia il fatto che esso sia interpretabile come un ordinamento parziale. Gli elementi di tale ordinamento sono le porte e esso rappresenta il fluire del segnale nel circuito: le porte che appaiono più in basso sono da considerare minori di quelle che appaiono più in alto.

La struttura dei moduli si può mappare abbastanza bene sulla definizione di circuito data. I valori e gli operatori sono i nodi del grafo; un valore costante è un nodo di ingresso del circuito, un operatore hardware è una porta.

Il circuito è costruito in modo incrementale attraverso l'aggiunta di nodi i cui ingressi sono prodotti da nodi inseriti in precedenza; le connessioni tra i nodi rispecchiano la struttura dell'espressione logica di partenza. Ogni espressione può essere convertita direttamente in un circuito mediante l'utilizzo di fili, porte d'ingresso per le costanti e un nodo interno per ogni operatore. È abbastanza immediato derivare da espressioni semplici la costruzione di circuiti che siano alberi; il vincolo di essere privo di cicli (ottenendo un DAG - Direct Acyclic Graph) viene imposto mediante la definizione di un nome per ogni filo (una nuova variabile), in modo da poter riferire il valore risultante da una sotto espressione in punti diversi del circuito. Chisel riporta un errore nella computazione ogni volta che determina un ciclo nel grafo.

In modo simile al processo induttivo che porta al calcolo del risultato del circuito, Chisel applica un altro processo induttivo, capace di inferire il numero di bit necessario per esprimere il valore di ogni nodo. La dimensione di ogni filo (uscita di un nodo) viene inferita automaticamente a partire dalla dimensione di registri e costanti (porte e ingressi del circuito) che lo precedono. L'algoritmo di inferenza parte dai valori delle porte per poi derivare le dimensioni di ogni uscita; il processo sfrutta l'applicazione di regole che definiscono la dimensione del risultato prodotto da ogni tipo di operando (a partire dalle dimensioni dei suoi ingressi). La terminazione si ha una volta calcolata la dimensione della porta di uscita. Si può dimostrare che la dimensione di ogni uscita è sempre maggiore o uguale delle dimensioni degli ingressi; i valori iniziali delle dimensioni dei registri devono essere specificati dal programmatore o in alternativa vengono derivati dalla dimensione del valore di default per quel registro (costante).

Capitolo 4

Evoluzione degli FPGA

Nei capitoli precedenti è stato sottolineato il grande interesse della ricerca nel campo degli FPGA nell'ampliare la diffusione di questi dispositivi, rendendoli appetibili a un sempre maggior numero di sviluppatori. Gli approcci visti fin qui prevedono di progettare nuovi linguaggi per la descrizione dell'hardware, che permettano di sfruttare costrutti più ad alto livello rispetto a un classico HDL come Verilog; una tendenza più recente consiste nell'utilizzare dei framework che consentano di programmare gli FPGA mediante la scrittura di codice parallelo in C, quindi con un'astrazione ancora maggiore di quella che riesce a fornire un linguaggio HDL ad alto livello come Chisel. Un approccio di questo tipo può sembrare simile al lavoro svolto da alcuni strumenti di sintesi come il compilatore HLS per modelli FPGA della Xilinx (di cui si è parlato nel primo capitolo); in realtà HLS compila del codice sequenziale scritto in C/C++, mentre i framework di cui si va a parlare ora permettono al programmatore di controllare direttamente il parallelismo mediante opportune annotazioni esplicite nel codice.

4.1 Il framework OpenCL

Open Computing Language (OpenCL) è uno standard di programmazione parallela multi piattaforma, utilizzato negli anni passati principalmente per la programmazione su **Graphics Processing Unit** (GPU).

Le GPU sono processori adatti all'esecuzione di funzioni semplici su vettori e matrici di dati di grandi dimensioni; per questo motivo sono adatte all'esecuzione di computazioni di tipo grafico, che manipolano matrici con un gran numero di pixel. È stata proprio la necessità di migliorare la performance per le computazioni data-parallel a spingere la ricerca verso le architetture multicore (fino ad arrivare alle GPU).

Lo scopo principale di OpenCL è quello di cercare di ampliare il range di sviluppatori in grado di usufruire di queste nuove tecnologie; in un primo momento l'interesse era solo sulle GPU, ma ora che si stanno affermando gli

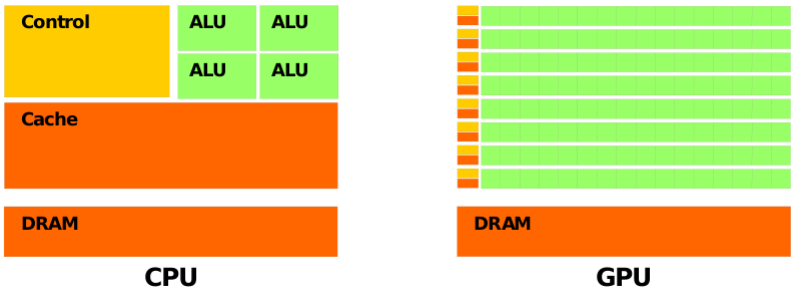


Figura 4.1: L'architettura di una CPU e l'architettura di una GPU a confronto.

FPGA si cerca di fare lo stesso anche per questi dispositivi. L'evoluzione dei dispositivi FPGA, in questo senso, sta ripercorrendo le fasi dell'evoluzione delle GPU, che da tecnologia per pochi ha avuto una continua diffusione fino ai recenti utilizzi anche nell'ambito di computazioni general-purpose.

OpenCL fornisce un'astrazione dall'hardware, consentendo di implementare algoritmi in modo parallelo utilizzando il linguaggio C (con qualche estensione); i dettagli hardware sono manipolati soltanto dal compilatore così che il programmatore si possa focalizzare esclusivamente sulla definizione dell'algoritmo. Il compilatore consente di eseguire il programma OpenCL su piattaforme diverse (CPU, GPU e FPGA): viene garantita la *portabilità funzionale* del codice prodotto (il compilatore genera programmi funzionalmente equivalenti per dispositivi diversi), ma la performance ottenuta non è sempre la stessa. Un codice OpenCL prevede delle allocazioni di memoria sul dispositivo, necessarie per potervi trasferire i dati da utilizzare durante la computazione; il tipo di allocazione dipende dall'organizzazione della memoria del dispositivo che si vuole utilizzare, quindi in realtà il codice OpenCL è ottimizzato per un particolare dispositivo (CPU, GPU, FPGA). Quando si compila questo codice per andarlo ad eseguire su un dispositivo differente, tutte le ottimizzazioni iniziali non hanno più effetto poiché un'architettura diversa implica anche un'organizzazione della memoria differente e per questo motivo si può verificare un degrado della performance. In ogni caso esistono ambiti in cui è possibile rinunciare ad avere la performance ottima in favore di altri fattori più determinanti quali programmabilità e versatilità.

Un sistema OpenCL si basa sull'interazione fra un *host* (un programma scritto in C/C++ capace di essere eseguito da qualsiasi tipo di microprocessore) e uno o più *device* (che possono essere CPU, GPU o FPGA) a cui delegare l'esecuzione di funzioni annotate come parallele dallo sviluppatore; tali funzioni sono dette *kernel*, scritte in linguaggio C con annotazioni che riguardano la memoria e il parallelismo. È compito dell'host gestire la memoria, i trasferimenti dei dati ai device, gli errori e infine ottenere indietro i risultati calcolati dai device.

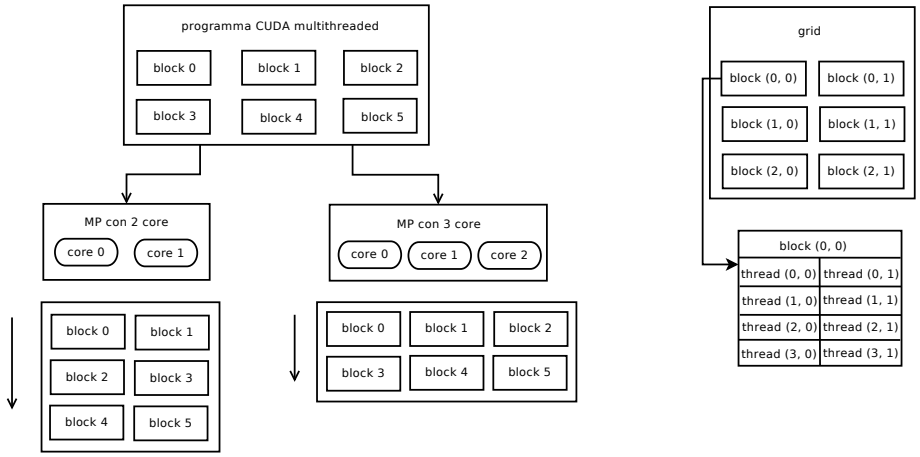


Figura 4.2: L'organizzazione CUDA di thread, block e grid. Un grid è gestito dal chip GPU, organizzato in più multiprocessor (MP), ognuno dei quali gestisce uno o più block.

Un livello di astrazione simile a quello fornito da OpenCL su dispositivi che possono essere CPU, GPU e FPGA viene messo a disposizione (in modo dedicato per le GPU NVIDIA) anche dal framework *CUDA*; esso permette di utilizzare il linguaggio di programmazione C (più un insieme di estensioni) per scrivere algoritmi da eseguire sulle GPU. CUDA si basa sul concetto di suddivisione di un problema in sottoproblemi più semplici: ognuno di essi corrisponde al calcolo di una funzione, delegato a un *warp* (un insieme di *thread*, ossia di computazioni parallele e indipendenti). Il numero di warp sulla GPU determina il numero di compiti differenti che è in grado di svolgere parallelamente; tutti i thread di un warp sono vincolati all'esecuzione della stessa funzione (per essere identificati mediante ID i thread vengono organizzati in *block*, che possono avere da una a tre dimensioni, a loro volta organizzati in *grid*, anch'esse 1D, 2D o 3D). OpenCL e CUDA adottano concetti molto simili, a cambiare sono i nomi utilizzati: i thread CUDA corrispondono ai *work-item* OpenCL, che sono organizzati in *work-group* (corrispondono ai block CUDA).

I flussi di esecuzione di ogni core del dispositivo GPU (thread CUDA o work-item OpenCL) possono accedere a quattro regioni di memoria, che si differenziano per modalità di accesso e scope. Secondo la suddivisione di OpenCL la *memoria globale* può essere utilizzata per operazioni di lettura e scrittura da tutti i work-item del dispositivo, la *memoria locale* restringe l'accesso in lettura e scrittura ai work-item di un work-group, la *memoria costante* consente operazioni di sola lettura a tutti i work-item e non può essere modificata durante l'esecuzione del kernel, infine la *memoria privata* è dedicata ai singoli work-item, che vi possono operare in lettura e scrittura.

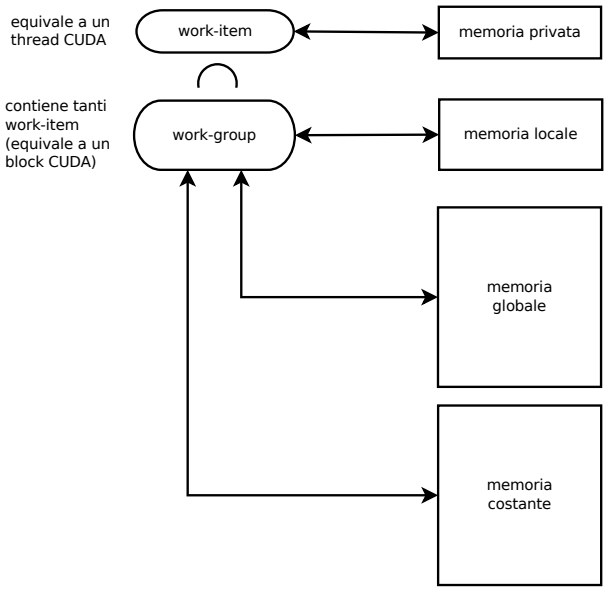


Figura 4.3: L'organizzazione OpenCL della memoria.

4.2 Differenze tra GPU e FPGA nell'esecuzione di un programma OpenCL

Un programma OpenCL viene tradotto dal compilatore in modi diversi, dipendenti dall'architettura del dispositivo scelto.

L'esecuzione di un algoritmo su una GPU prevede una precisa organizzazione dei dati da utilizzare, che vengono trasferiti dall'host alla memoria globale del dispositivo, dalla quale ogni core (che corrisponde a un flusso di esecuzione indipendente) è in grado di prelevarli (accedendoci in modo vettoriale) e utilizzarli per la computazione. I core di una GPU sono organizzati in unità chiamate *multiprocessor* (MP); tutti i core di un MP calcolano la stessa funzione (su porzioni di dati differenti), ma ogni MP può eseguire computazioni diverse dagli altri; l'esecuzione della stessa computazione su ogni core di un MP è controllata da un'unità apposita. Rispetto all'organizzazione CUDA un MP può essere paragonato al concetto di warp; ogni MP gestisce uno o più block e può essere ulteriormente diviso in *stream processor* (SP) che gestiscono uno o più thread del block.

Mentre i tanti core dell'architettura di una GPU sono adatti al calcolo di una funzione semplice su una grande quantità di dati, un FPGA è conveniente per il calcolo di funzioni anche molto complesse su pochi dati. La complessità di una funzione è gestita mediante l'implementazione di circuiti configurati ad hoc per l'algoritmo a partire dalle risorse dell'hardware. L'utilizzo di tanti dati si traduce, su un dispositivo FPGA, in un costo molto alto per i trasferimenti, che avvengono mediante uno stream sequenziale. È

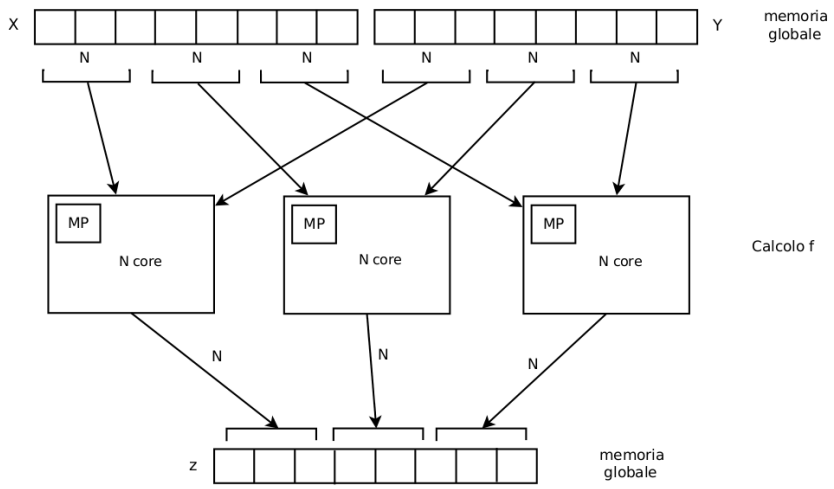


Figura 4.4: L'implementazione della funzione $z[i] = f(x[i], y[i])$ su una GPU. I dati nei vettori in ingresso x e y vengono mappati in gruppi di N (numero di core per MP) su più MP che calcolano f, i valori calcolati sono salvati nelle locazioni del vettore z.

possibile accettare di pagare un costo elevato per il trasferimento dei dati solamente nel caso in cui la funzione da calcolare sia abbastanza complessa da far sì che il guadagno sulla performance ottenuto mediante la sua implementazione sullo FPGA compensi il costo dei trasferimenti.

L'esecuzione di una funzione (kernel) del tipo

```
for(i = 0; i < M; i++)
  z[i] = f(x[i], y[i]);
```

su un dispositivo FPGA e su una GPU, si traduce in implementazioni differenti (generate dal compilatore OpenCL) sulle due architetture. È necessario che le iterazioni del kernel siano indipendenti tra loro, altrimenti non sarebbe possibile eseguire questa computazione su una GPU, visto che il calcolo avviene su ogni core in modo parallelo e indipendente.

Quello che avviene per una GPU è prima di tutto l'organizzazione dei dati nei vettori x e y, che vengono poi trasferiti nella memoria globale del dispositivo; una volta che i vettori sono stati caricati in memoria essi vengono suddivisi in porzioni di dimensione equivalente al numero di core per MP, poniamo N (questo consente di mappare gruppi di N posizioni dei vettori in ingresso su gruppi di N core che utilizzeranno questi dati per calcolare la funzione). Ogni core è capace di calcolare l'i-esima posizione del vettore risultato: tutti trasferiscono il valore calcolato in memoria globale, così che il vettore risultato possa essere trasferito indietro all'host.

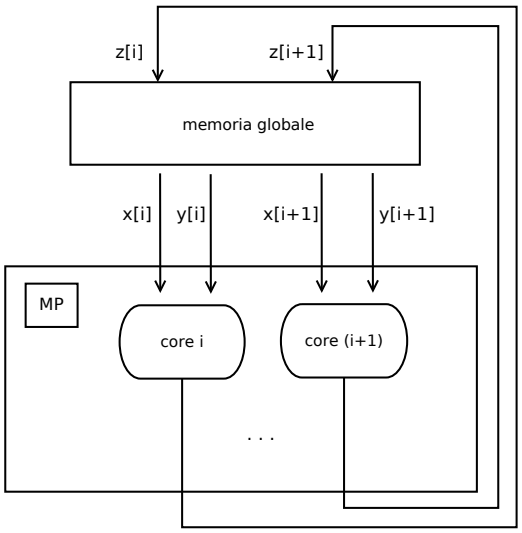


Figura 4.5: L'implementazione della funzione $z[i] = f(x[i], y[i])$ su una GPU. Si mostra il dettaglio degli accessi in lettura e scrittura alla memoria globale da parte di ogni core di un MP che calcola f.

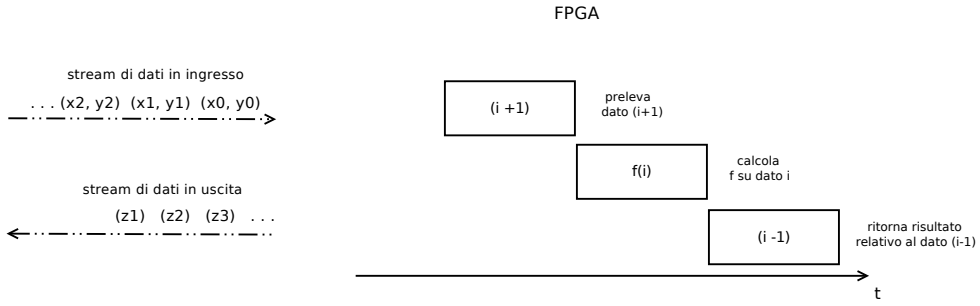


Figura 4.6: L'implementazione della funzione $z[i] = f(x[i], y[i])$ su un FPGA.

Su un FPGA la computazione assume la forma di un pipeline, in cui si ha uno stadio che si occupa di prelevare i dati, uno stadio che effettua il calcolo della funzione e infine un'altro stadio il cui compito è rimandare indietro il risultato calcolato. Il trasferimento dei dati avviene quindi come stream di coppie $(x_0, y_0), (x_1, y_1), \ldots$ che vengono accettate man mano dal primo stadio del pipeline; quando il pipeline è a regime, il primo stadio preleva il dato $(i+1)$-esimo, mentre il secondo stadio calcola la funzione sul dato i-esimo e il terzo stadio restituisce il risultato calcolato relativo al dato $(i-1)$-esimo.

Lo stadio che si occupa del calcolo della funzione al suo interno può essere implementato come un singolo pipeline (nel caso in cui l'implementazione per la computazione della f richieda l'utilizzo di tante risorse, con le modalità descritte nel primo capitolo); altrimenti, in base alla disponibilità delle

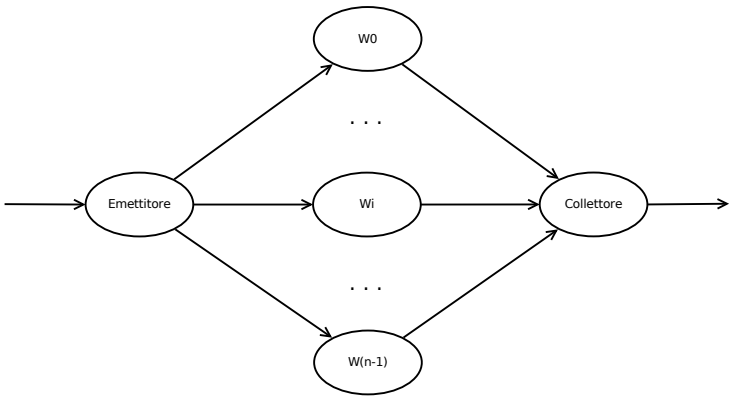


Figura 4.7: La struttura farm.

risorse sullo FPGA, il compilatore può fare alcune ottimizzazioni sul circuito prodotto. La presenza di risorse inutilizzate sul chip consente di procedere alla creazione di più copie del circuito che calcola la funzione; questo processo è chiamato *replicazione funzionale* e ogni copia esegue in maniera indipendente e parallela alle altre. Di fatto mediante la replicazione il compilatore genera un *farm*: tale forma di parallelismo replica l'intera elaborazione di un modulo (indipendentemente dalla sua struttura interna) su n *worker* identici e prevede l'utilizzo di un *emettitore* per la distribuzione dei dati dallo stream di ingresso ai worker e di un *collettore* per la raccolta dei dati elaborati dai worker. Un po' di logica sullo FPGA deve occuparsi quindi di prendere i dati in ingresso e distribuirli alle unità nel farm e inoltre di prelevare i risultati dalle stesse unità: tornando all'esempio, ogni worker riceve in ingresso una coppia $(x[i], y[i])$, che utilizza per il calcolo della funzione, generando un risultato $z[i]$.

L'incremento della performance che si ottiene mediante la replicazione funzionale è lineare col numero di copie della funzione. OpenCL (sui dispositivi della Altera) consente di esplicitare il grado di replicazione di un kernel assegnando il valore all'attributo `num_compute_units`. Va valutato caso per caso se sia preferibile (in termini di performance ottenuta) dedicare meno risorse all'implementazione del pipeline che calcola la singola funzione e ottenere un grado di replicazione maggiore, oppure il viceversa. La riduzione delle risorse richieste per il calcolo della funzione può essere ottenuta eliminando le porzioni di circuito che si occupano di calcoli complessi e demandando l'esecuzione di quelle operazioni all'host, che fornisce il risultato calcolato passandolo come parametro del kernel.

Un caso in cui conviene impiegare un maggior numero di risorse per il singolo pipeline si ha con l'*unroll* di un ciclo: anziché eseguire le iterazioni del ciclo in modo sequenziale si utilizza abbastanza logica (se il numero di risorse sullo FPGA lo permette) per calcolare in parallelo i risultati di tutte le

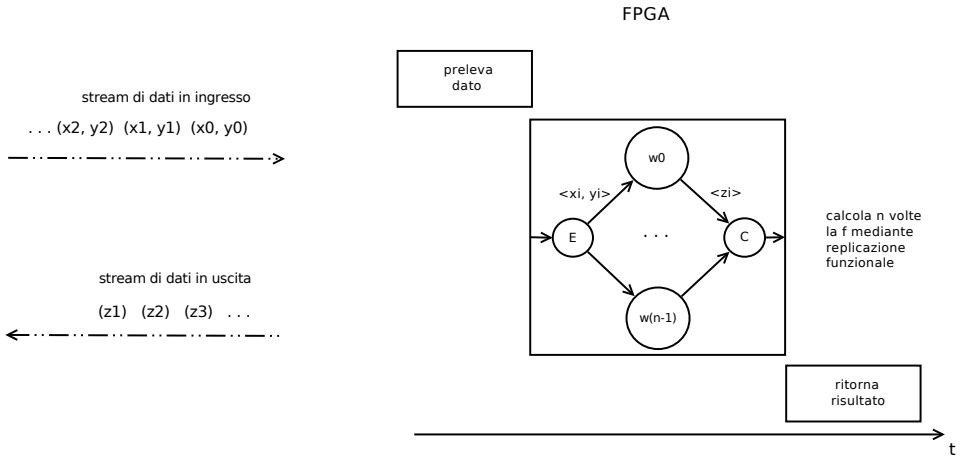


Figura 4.8: L'implementazione della funzione $z[i] = f(x[i], y[i])$ su un FPGA con replicazione funzionale.

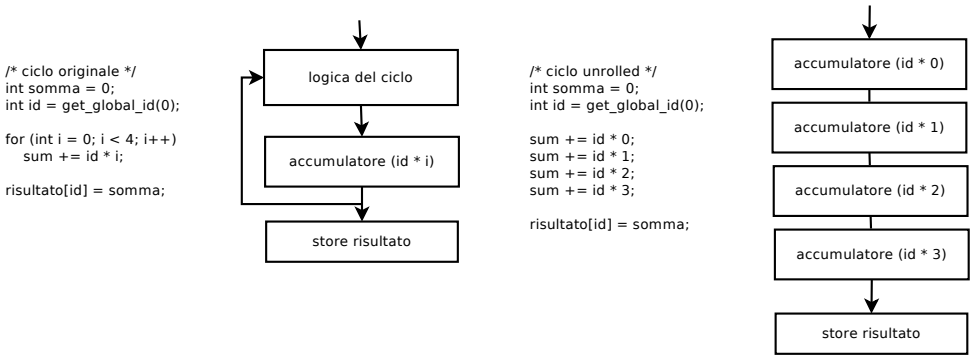


Figura 4.9: Un esempio di loop unrolling.

iterazioni (o di un certo numero di esse); tali risultati vengono messi assieme dopo essere stati calcolati tutti, permettendo di rimuovere il contatore delle iterazioni e la logica che testa la condizione di uscita del ciclo. OpenCL (sui dispositivi Altera) permette di esplicitare il loop unrolling mediante la direttiva `#pragma unroll`.

Un'ottimizzazione alternativa alla replicazione funzionale è la *vettorizzazione* del kernel; viene mantenuta un'unica copia del pipeline per il calcolo della funzione, ma ogni stadio non lavora su un dato alla volta bensì su N (il trasferimento dei dati è effettuato in modo vettoriale e il calcolo della funzione avviene su tutti gli N dati ricevuti).

Trovare la configurazione migliore utilizzando replicazione, vettorizzazione e loop-unrolling non è un compito banale: il compilatore OpenCL esplora più possibili design, in base alle risorse disponibili, per poi scegliere quello che offre la performance maggiore.

L'architettura delle GPU è adatta ad eseguire computazioni del tipo

$$z[i] = f(x[i], y[i])$$

poiché tutti i core eseguono la stessa funzione f in modo parallelo e indipendente, sui due vettori in ingresso x e y, computando ognuno l'i-esima posizione del vettore risultato. Non è possibile calcolare altrettanto bene sulle GPU computazioni del tipo

$$z[i] = (y[i]\%2 == 0\ ?\ f(x[i]) : g(x[i]))$$

dove f e g sono due funzioni distinte. Questo perché il vincolo che tutti i core debbano eseguire necessariamente la stessa funzione non consente di calcolare due funzioni differenti in base al valore assunto dalla guardia del costrutto condizionale: per fare ciò ogni core dovrebbe poter calcolare alternativamente la f o la g, col risultato di avere nello stesso MP alcuni core che eseguono la f e altri che eseguono la g (cosa non consentita). Per rispettare il vincolo che tutti i core eseguano la f oppure tutti eseguano la g ognuno dovrebbe valutare la guardia ($y[i]\%2 == 0$) e a quel punto procedono all'esecuzione solo i core per cui la guardia valuta *true* (se la funzione che eseguono è la f) oppure solo i core per cui la guardia valuta $false$ (se la funzione che eseguono è la g). È abbastanza evidente che una computazione del genere non può sfruttare a pieno il grado di parallelismo offerto da tutti i core, con conseguente perdita in termini di performance raggiungibile. Questo tipo di calcolo è implementato efficientemente sugli FPGA (come spiegato nel primo capitolo) poiché tutti i flussi di esecuzione vengono calcolati da un circuito dedicato (in questo caso vengono implementate entrambe le funzioni f e g) permettendo la produzione in parallelo dei risultati e utilizzando poi la condizione per scegliere il risultato corretto.

Per le GPU l'esecuzione della stessa funzione su tutti i core di un MP è un vincolo molto forte; la possibilità di calcolare più task diversi in parallelo è data mediante l'utilizzo di MP differenti. In passato si aveva un maggior numero di MP, con relativamente pochi core ognuno, dando più enfasi all'esecuzione di task diversi parallelamente. La tendenza attuale è invece quella di diminuire il numero di computazioni differenti eseguibili su una GPU (diminuire il numero di warp), preferendo un incremento del numero di core per warp (dedicati all'esecuzione della stessa funzione).

Tutte le considerazioni fatte fin qui possono essere riassunte in un esempio, che mette a confronto l'esecuzione su un FPGA e quella su una GPU. Si ammetta di dover eseguire una funzione f composta da 4 fasi diverse A, B, C, D; parlando in termini di thread si ha che ognuno esegue il corpo della f in maniera indipendente e parallela.

La computazione della f su un FPGA viene tradotta in un pipeline con 4 stadi, ognuno dei quali esegue una della istruzioni A, B, C, D; durante l'esecuzione, col pipeline a regime, si avrà che il primo stadio esegue l'istruzione

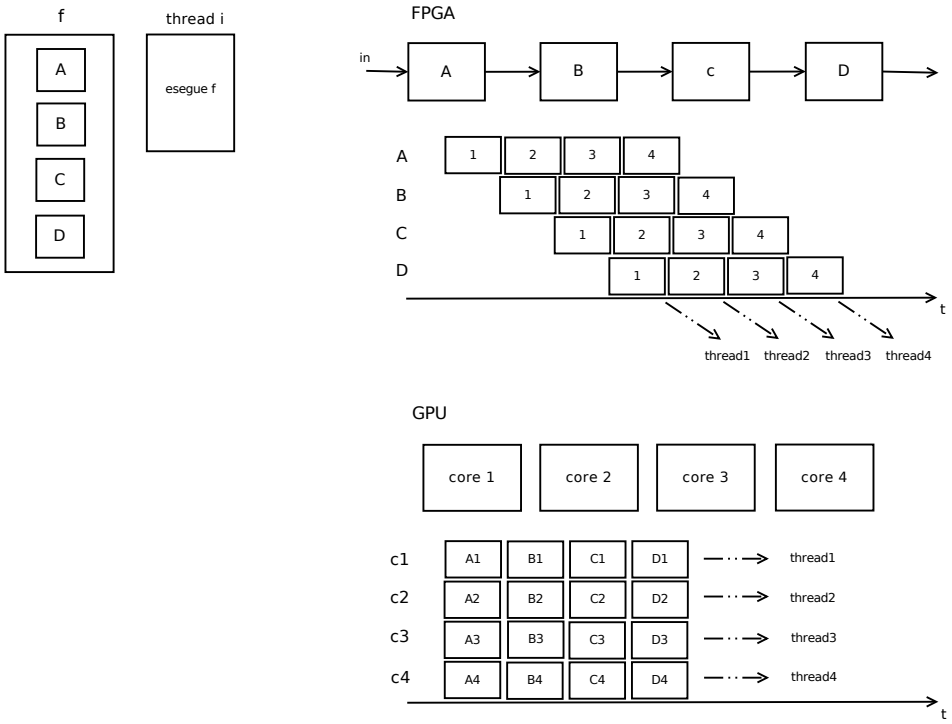


Figura 4.10: Un esempio di esecuzione di una funzione f su un FPGA e su una GPU. La f è composta da 4 istruzioni diverse, che su un FPGA si traducono in 4 differenti stadi di un pipeline, mentre su una GPU vanno a formare il flusso di esecuzione gestito da ogni core.

A del thread t_4 (sul dato $i+3$), mentre il secondo stadio esegue l'istruzione B del thread t_3 (sul dato $i+2$), mentre il terzo stadio esegue l'istruzione C del thread t_2 (sul dato $i+1$), mentre il quarto stadio esegue l'istruzione D del thread t_1 (sul dato i) (che quindi ha attraversato completamente il pipeline e terminato l'esecuzione producendo il suo risultato).

La stessa computazione su una GPU che utilizza 4 core avviene in modo differente, infatti ognuno dei 4 core esegue in parallelo con gli altri un thread (su dati diversi): il core 1 esegue le istruzioni $A-B-C-D$ sul dato i, in parallelo col core 2 che esegue le istruzioni $A-B-C-D$ sul dato $i+1$, in parallelo col core 3 che esegue le istruzioni $A-B-C-D$ sul dato $i+2$, in parallelo col core 4 che esegue le istruzioni $A-B-C-D$ sul dato $i+3$.

Capitolo 5

Conclusioni

Nei capitoli centrali di questo lavoro sono state descritte alcune delle metodologie più comunemente adottate per l'utilizzo dei dispositivi FPGA, cercando di inquadrare le motivazioni che portano a preferire un approccio piuttosto che un altro e l'evoluzione tecnologica sottostante.

La diffusione di framework come OpenCL risponde alla necessità di ampliare la fascia di sviluppatori in grado di programmare su un particolare dispositivo di tipo coprocessore (prima sulle GPU, ora sugli FPGA). Lo sviluppo di codice parallelo mediante un linguaggio come il C migliora la programmabilità di dispositivi come gli FPGA, offrendo di fatto un mezzo per astrarre dall'hardware e delegando al compilatore la generazione di codice a basso livello, anche se il sottoinsieme del C compilabile è tutto sommato abbastanza limitato. Tali standard consentono inoltre di avere programmi che siano portabili, evitando di dover riscrivere il codice per architetture diverse tra loro; questo è utile nel caso in cui non si abbiano particolari esigenze riguardo alla performance ottenibile dall'esecuzione sugli altri dispositivi.

Sono stati analizzati due approcci diversi alla programmazione degli FPGA mediante linguaggi di descrizione dell'hardware; la tendenza è quella di semplificare lo sviluppo dei programmi da eseguire su questi dispositivi.

L'utilizzo di Hardware Description Language come il Verilog è ancora prevalente. La forza di questi linguaggi sta nell'avere costrutti appositi per la programmazione di hardware, ma il basso livello di astrazione rende il loro utilizzo ostico: gli sviluppatori devono ragionare in termini di bit, fili, componenti dell'hardware, necessitando di una profonda conoscenza dei dettagli dell'architettura del dispositivo. Recenti sviluppi hanno cercato di mantenere le caratteristiche proprie degli Hardware Description Language, integrandole con quelle di linguaggi più ad alto livello. Uno dei progetti che si basa su questa idea di fondo è Chisel, un nuovo linguaggio per la descrizione dell'hardware, che grazie all'integrazione col linguaggio Scala riesce a beneficiare di caratteristiche proprie della programmazione a oggetti e funzionale. La programmazione a oggetti si presta bene al calcolo parallelo,

permettendo di considerare ogni oggetto come un'entità capace di eseguire in maniera indipendente e concorrente. La programmazione funzionale vede le funzioni come entità che guidano il flusso di esecuzione; questo approccio permette di esprimere bene modelli di programmazione dataflow, nei quali l'analisi delle dipendenze tra funzioni diverse determina il loro ordine di esecuzione (funzioni indipendenti possono eseguire in maniera parallela mentre le altre vengono schedulate in modo da garantire la produzione dei dati prima dell'esecuzione delle funzioni che li consumano).

Un metodo di programmazione ad un più alto livello migliora l'usabilità dei dispositivi FPGA (riducendo i tempi di sviluppo del software) e rende la tecnologia accessibile a una porzione più ampia di sviluppatori.

Come lavoro futuro si potrebbero sviluppare dei test per effettuare l'analisi della performance di un modulo complesso descritto in Verilog e della corrispondente versione Chisel, andando a riscontrare con dati reali se le due implementazioni sono equivalenti anche dal punto di vista della performance, oltre che da quello della funzionalità.

Appendice A

Implementazione Verilog del modulo Addizionatore

L'implementazione del modulo addizionatore a $N = 8$ bit riportata qui di seguito è stata derivata dalla compilazione del modulo addizionatore a n bit implementato in Chisel e riportato nel Codice 3.4.

Si può notare che viene generato prima il modulo addizionatore binario e successivamente il modulo addizionatore a N bit che lo istanzia. C'è una corrispondenza diretta tra il numero dei bit degli ingressi e dell'uscita e il numero di istanze del modulo addizionatore binario: in questo esempio gli ingressi e le uscite sono da 8 bit e vengono istanziati 8 moduli per eseguire il calcolo (ognuno lavora su unico bit). Da ciò si deduce che il numero di istanziazioni necessario cresce proporzionalmente al numero di bit N. Questo è vero anche nel caso in cui l'implementazione del modulo Verilog utilizzi il costrutto `generate`; la differenza sta nel fatto che nel tipo di implementazione qui riportato si ha un aumento della quantità di codice lineare col parametro N, mentre con un approccio generativo cresce solo il numero di iterazioni necessarie a istanziare tutti i moduli.

```
1  module FullAdder(
2      input  io_a,
3      input  io_b,
4      input  io_cin,
5      output io_sum,
6      output io_cout
7  );
8
9    wire T0;
10   wire a_and_cin;
11   wire T1;
12   wire b_and_cin;
13   wire a_and_b;
14   wire T2;
15   wire a_xor_b;
```

```
16
17
18    assign io_cout = T0;
19    assign T0 = T1 | a_and_cin;
20    assign a_and_cin = io_a & io_cin;
21    assign T1 = a_and_b | b_and_cin;
22    assign b_and_cin = io_b & io_cin;
23    assign a_and_b = io_a & io_b;
24    assign io_sum = T2;
25    assign T2 = a_xor_b ^ io_cin;
26    assign a_xor_b = io_a ^ io_b;
27  endmodule
28
29  module Adder(
30      input [7:0] io_A,
31      input [7:0] io_B,
32      input  io_Cin,
33      output[7:0] io_Sum,
34      output io_Cout
35  );
36
37    wire carry_7;
38    wire T0;
39    wire T1;
40    wire carry_6;
41    wire T2;
42    wire T3;
43    wire carry_5;
44    wire T4;
45    wire T5;
46    wire carry_4;
47    wire T6;
48    wire T7;
49    wire carry_3;
50    wire T8;
51    wire T9;
52    wire carry_2;
53    wire T10;
54    wire T11;
55    wire carry_1;
56    wire T12;
57    wire T13;
58    wire carry_0;
59    wire T14;
60    wire T15;
61    wire carry_8;
62    wire[7:0] T16;
63    wire[7:0] T17;
64    wire[7:0] T18;
```

```
65    wire[3:0] T19;
66    wire[1:0] T20;
67    wire sum_0;
68    wire T21;
69    wire sum_1;
70    wire T22;
71    wire[1:0] T23;
72    wire sum_2;
73    wire T24;
74    wire sum_3;
75    wire T25;
76    wire[3:0] T26;
77    wire[1:0] T27;
78    wire sum_4;
79    wire T28;
80    wire sum_5;
81    wire T29;
82    wire[1:0] T30;
83    wire sum_6;
84    wire T31;
85    wire sum_7;
86    wire T32;
87    wire FullAdder_io_sum;
88    wire FullAdder_io_cout;
89    wire FullAdder_1_io_sum;
90    wire FullAdder_1_io_cout;
91    wire FullAdder_2_io_sum;
92    wire FullAdder_2_io_cout;
93    wire FullAdder_3_io_sum;
94    wire FullAdder_3_io_cout;
95    wire FullAdder_4_io_sum;
96    wire FullAdder_4_io_cout;
97    wire FullAdder_5_io_sum;
98    wire FullAdder_5_io_cout;
99    wire FullAdder_6_io_sum;
100   wire FullAdder_6_io_cout;
101   wire FullAdder_7_io_sum;
102   wire FullAdder_7_io_cout;
103
104
105   assign carry_7 = FullAdder_6_io_cout;
106   assign T0 = io_B[3'h7:3'h7];
107   assign T1 = io_A[3'h7:3'h7];
108   assign carry_6 = FullAdder_5_io_cout;
109   assign T2 = io_B[3'h6:3'h6];
110   assign T3 = io_A[3'h6:3'h6];
111   assign carry_5 = FullAdder_4_io_cout;
112   assign T4 = io_B[3'h5:3'h5];
113   assign T5 = io_A[3'h5:3'h5];
```

```
114   assign carry_4 = FullAdder_3_io_cout;
115   assign T6 = io_B[3'h4:3'h4];
116   assign T7 = io_A[3'h4:3'h4];
117   assign carry_3 = FullAdder_2_io_cout;
118   assign T8 = io_B[2'h3:2'h3];
119   assign T9 = io_A[2'h3:2'h3];
120   assign carry_2 = FullAdder_1_io_cout;
121   assign T10 = io_B[2'h2:2'h2];
122   assign T11 = io_A[2'h2:2'h2];
123   assign carry_1 = FullAdder_io_cout;
124   assign T12 = io_B[1'h1:1'h1];
125   assign T13 = io_A[1'h1:1'h1];
126   assign carry_0 = io_Cin;
127   assign T14 = io_B[1'h0:1'h0];
128   assign T15 = io_A[1'h0:1'h0];
129   assign io_Cout = carry_8;
130   assign carry_8 = FullAdder_7_io_cout;
131   assign io_Sum = T16;
132   assign T16 = T17;
133   assign T17 = T18;
134   assign T18 = {T26, T19};
135   assign T19 = {T23, T20};
136   assign T20 = {sum_1, sum_0};
137   assign sum_0 = T21;
138   assign T21 = FullAdder_io_sum;
139   assign sum_1 = T22;
140   assign T22 = FullAdder_1_io_sum;
141   assign T23 = {sum_3, sum_2};
142   assign sum_2 = T24;
143   assign T24 = FullAdder_2_io_sum;
144   assign sum_3 = T25;
145   assign T25 = FullAdder_3_io_sum;
146   assign T26 = {T30, T27};
147   assign T27 = {sum_5, sum_4};
148   assign sum_4 = T28;
149   assign T28 = FullAdder_4_io_sum;
150   assign sum_5 = T29;
151   assign T29 = FullAdder_5_io_sum;
152   assign T30 = {sum_7, sum_6};
153   assign sum_6 = T31;
154   assign T31 = FullAdder_6_io_sum;
155   assign sum_7 = T32;
156   assign T32 = FullAdder_7_io_sum;
157
158   FullAdder FullAdder(
159        .io_a( T15 ),
160        .io_b( T14 ),
161        .io_cin( carry_0 ),
162        .io_sum( FullAdder_io_sum ),
```

```
163         .io_cout( FullAdder_io_cout )
164     );
165     FullAdder FullAdder_1(
166         .io_a( T13 ),
167         .io_b( T12 ),
168         .io_cin( carry_1 ),
169         .io_sum( FullAdder_1_io_sum ),
170         .io_cout( FullAdder_1_io_cout )
171     );
172     FullAdder FullAdder_2(
173         .io_a( T11 ),
174         .io_b( T10 ),
175         .io_cin( carry_2 ),
176         .io_sum( FullAdder_2_io_sum ),
177         .io_cout( FullAdder_2_io_cout )
178     );
179     FullAdder FullAdder_3(
180         .io_a( T9 ),
181         .io_b( T8 ),
182         .io_cin( carry_3 ),
183         .io_sum( FullAdder_3_io_sum ),
184         .io_cout( FullAdder_3_io_cout )
185     );
186     FullAdder FullAdder_4(
187         .io_a( T7 ),
188         .io_b( T6 ),
189         .io_cin( carry_4 ),
190         .io_sum( FullAdder_4_io_sum ),
191         .io_cout( FullAdder_4_io_cout )
192     );
193     FullAdder FullAdder_5(
194         .io_a( T5 ),
195         .io_b( T4 ),
196         .io_cin( carry_5 ),
197         .io_sum( FullAdder_5_io_sum ),
198         .io_cout( FullAdder_5_io_cout )
199     );
200     FullAdder FullAdder_6(
201         .io_a( T3 ),
202         .io_b( T2 ),
203         .io_cin( carry_6 ),
204         .io_sum( FullAdder_6_io_sum ),
205         .io_cout( FullAdder_6_io_cout )
206     );
207     FullAdder FullAdder_7(
208         .io_a( T1 ),
209         .io_b( T0 ),
210         .io_cin( carry_7 ),
211         .io_sum( FullAdder_7_io_sum ),
```

```
212         .io_cout( FullAdder_7_io_cout )
213   );
214 endmodule
```

Elenco delle figure

Bibliografia

[1] Acceleware. *OpenCL on FPGAs for GPU programmers*. 2014. URL: https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/wp/wp-201406-acceleware-opencl-on-fpgas-for-gpu-programmers.pdf.

[2] Altera Corporation. *Implementing FPGA Design with the OpenCL Standard*. 2013. URL: https://www.altera.com/en_US/pdfs/literature/wp/wp-01173-opencl.pdf.

[3] Marco Danelutto. *Note sull'utilizzo di Verilog per la prima parte del corso di Architettura degli Elaboratori*. 2015. URL: http://didawiki.cli.di.unipi.it/lib/exe/fetch.php/informatica/ae/verilog2.pdf.

[4] Pierpaolo Degano. *Fondamenti di informatica: calcolabilità e complessità*. 2015.

[5] Domenico Talia Giandomenico Spezzano. *Calcolo parallelo, automi cellulari e modelli per sistemi complessi*. A cura di FrancoAngeli. 2008.

[6] John Wawrzynek Jonathan Bachrach Krste Asanovic. *Chisel 2.2 Tutorial*. 2015. URL: https://chisel.eecs.berkeley.edu/latest/chisel-tutorial.pdf.

[7] Vincent Lee Jonathan Bachrach. *Getting Started with Chisel*. 2015. URL: https://chisel.eecs.berkeley.edu/latest/getting-started.html.

[8] NVIDIA. *CUDA C Programming Guide*. 2015. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#hardware-implementation.

[9] Marco Vanneschi. *Architettura degli Elaboratori*. A cura di Plus - Pisa University Press. 2009.

[10] Xilinx. *Dual-Port Block Memory*. 2005. URL: http://www.xilinx.com/support/documentation/ip_documentation/dp_block_mem.pdf.

[11] Xilinx. *Introduction to FPGA Design with Vivado High-Level Synthesis*. 2013. URL: http://www.xilinx.com/support/documentation/sw_manuals/ug998-vivado-intro-fpga-design-hls.pdf.

[12] Xilinx. *Single-Port Block Memory*. 2005. URL: http://www.xilinx.com/support/documentation/ip_documentation/sp_block_mem.pdf.