STUDIO E IMPLEMENTAZIONE DI STRUMENTI PER LO SVILUPPO SOFTWARE NEI SISTEMI BASATI SU FPGA

UNIVERSITA' DI CAGLIARI
DIPARTIMENTO DI INGEGNERIA ELETTRICA ED ELETTRONICA
TESI PER IL CONSEGUIMENTO DEL DOTTORATO DI RICERCA
IN INGEGNERIA ELETTRONICA ED INFORMATICA

Autore:
Giovanni BUSONERA
Marzo 2008

ABSTRACT

Computer market becomes every day more performance-hungry. Nowadays microprocessor based systems are not able to relevant good performances in various application domains. The classic Von Neumann load/store architecture is suitable for some tasks, but seems to have serious scalability issues.

To tackle these issues one of the current trend in the quest for additional execution speed, provides additional processing cores in the central processing unit and to parallelize the execution of as much of the software as possible. This paradigm has potential, but its effectiveness has been limited by the difficulties involved in parallelizing software that was written for sequential execution, resulting in cumbersome dependencies. These dependencies require extensive changes in the software design paradigms if this approach means to reach its full potential.

On the other hand, reconfigurable devices have proven to be very scalable for a large class of applications and they offer potential application performance improvements beyond those predicted by Moore's Law.

Among the different templates proposed in literature in the field or reconfigurable computing, Configurable System-on-a-Chip (CSoC) are emerging as a convincing trade-off between efficiency and flexibility. This kind of systems are often composed by one or more CPUs coupled with a Field Programmable Gate Array (FPGA), and it can be demonstrated that they can run applications two orders of magnitude faster than traditional on CPUs. This performance speedup with respect to microprocessors relies basically in the opportunity of using dedicated reconfigurable hardware to exploit the inherent parallelism of an algorithm.

One of the key issues in using such systems is related to the software development activity. A programmer typically uses a high level language to implement its algorithms but, to exploit the full potential of a reconfigurable system, a deeper knowledge of the target architecture and of digital design techniques are needed.

In this work two tools are developed and implemented to overcome these issues and allow the software developer to use its traditional development flow. The first one deals with code partitioning between CPU and FPGA. The tool takes as input an ANSI C application code and performs the following operations: (i) automatically identifies code fragments suitable for hardware implementation as specialized functional units (ii)for all these segments a synthesizable code is generated and sent to a synthesis tool,(iii) from the synthesis results, the segments to be implemented on FPGA are selected (iv) bitstream to configure the FPGA

and modified C code to be executed on the CPU are generated. In order to validate this tool, it was applied to standard benchmarks obtaining, with respect to state of the art, an improvement of up to 250% in the accuracy of performances estimation related to the selected segments of code.

The second tool developed, named eBug, is a debugging solution for software developed on the eMIPS [1] dynamically-extensible processor. The off-chip portion of eBug is an application that performs tasks that would be too expensive or too inflexible to perform in hardware, such as implementing the communication protocols to interface to the client debuggers. The on-chip hardware portion of eBug is realized with a new approach: rather than being built into the base pipelined data path, it is a loadable logic module that uses the standard Extension interface of the processor. This accomplishes the three goals of area minimization and reuse, security in a general purpose, multi-user environment, and open-ended extensibility.

When not in use, eBug is simply not present on the chip and its area is therefore reused. eBug solves the security issues normally created by a hardware-level debug module because only the process that owns the eBug Extension can be affected by a debugging session. As an Extension, eBug is not compiled into the basic processor design and this makes it easy to add new features without affecting the core eMIPS design. Leveraging the high-visibility extension interface of eMIPS, eBug can realize arbitrarily complex features for high-level monitoring. To show this feature hardware watchpoints support is transparently added to the initial, simpler design. It is also possible to interface eBug with other eMIPS extensions such as those generated by P2V [2] to improve its capabilities. eBug was written in Verilog and is usable both with the Giano [3] system simulator and on the Xilinx ML401 FPGA board.

Sommario

Lista delle Figure

Capitolo 1

Introduzione

La continua richiesta di capacità computazionale unita alla ricerca di altri obiettivi quali la bassa dissipazione di potenza, rende i sistemi a microprocessore sempre meno adatti in molti domini applicativi d'interesse. I microprocessori più moderni implementano generalmente un'architettura di tipo Reduced Instrction Set Computer (RISC) che è basata su un insieme fisso e relativamente limitato di istruzioni. Quest'insieme o, più precisamente, l'Instruction Set Architecture (ISA) [4] del processore, viene definito con l'intento di raggiungere la massima efficienza possibile per un ampio spettro di applicazioni cercando, inoltre, il miglior compromesso in termini di dimensione, costo e potenza dissipata dal chip.
A causa della grande dimensione e della continua crescita dello spazio delle applicazioni, la definizione di un ISA, che sia efficiente in tutti i domini applicativi, risulta essere un'operazione impossibile. In sostanza, l'approccio general purpose alla progettazione dei microprocessori, mostra dei limiti che sono tanto più evidenti quanto più i domini applicativi diventano specifici. L'esempio classico è dato dalle applicazioni multimediali che richiedono notevoli capacità computazionali per l'elaborazione di streaming audio e video. Per far fronte a queste richieste, alcune aziende costruttrici di microprocessori hanno esteso l'ISA con istruzioni adatte all'elaborazione dei flussi di dati. Ad esempio, a partire dal PentiumIII, Intel ha implementato nei suoi cores un insieme di istruzioni SIMD chiamate SSE (poi evolute fino alla versione SSE4) [5] mentre AMD ha utilizzato le 3DNow [6].
Questa situazione è tanto più evidente se si considera il mercato embedded dove, oltre alla richiesta di potenza computazionale, è necessario rispettare delle specifiche stringenti riguardo alla dimensione, al costo e alla potenza dissipata. Per far fronte a queste specifiche, i microprocessori per sistemi embedded sono progettati per funzionare, in modo efficiente, per un ristretto dominio applicativo dove l'uso di una CPU general purpose sarebbe inadeguato a causa di un eccessivo numero di risorse non utilizzate. Ad esempio, se l'applicazione d'interesse è basata su una rappresentazione dei dati di tipo fixed point, la presenza di un'unità floating point comporta un inutile spreco di spazio e di potenza.

Generalmente la specializzazione del microprocessore per applicazioni embedded avviene tramite la definizione, application-driven, di un nuovo ISA, causando un aumento dei costi per la riprogettazione e la realizzazione delle nuove CPU.
Un'altra soluzione, orientata alla ricerca di maggior potenza computazionale, è quella relativa ai sistemi multicore. In questo caso l'obiettivo è suddividere un'applicazione in più tasks paralleli eseguiti da diverse CPU integrate nello stesso chip. Questo paradigma architetturale, pur essendo molto promettente, ha fondamentalmente due problemi. Il primo dipende dalla parallelizzazione del software scritto per essere eseguito in modo sequenziale. Questo comporta una notevole dipendenza dei dati che inficia il potenziale dell'esecuzione parallela. Il secondo problema riguarda la necessità di rendere trasparente al programmatore la comunicazione e la sincronizzazione dei processi eseguiti dai diversi core [7], per fornire un semplice modello di programmazione del sistema.
Nella ricerca di maggiori prestazioni sia la ricerca accademica che l'industria, hanno proposto un nuovo paradigma: le architetture riconfigurabili. Queste sono in grado di modificare le proprie caratteristiche in modo dinamico aumentando, di fatto, l'efficienza per un certo dominio applicativo. Un esempio classico è l'uso di un dispositivo riconfigurabile, come la FPGA, associato ad una CPU per implementare dinamicamente delle unità funzionali specializzate. Quando l'elemento riconfigurabile è integrato nello stesso chip della CPU, si parla tipicamente di Configurable System on Chip (CSoC).
In alcune proposte la riconfigurabilità non avviene a run-time ma a design-time. In sostanza un'architettura base è estesa staticamente e implementata su un chip, a partire dalle esigenze dell'applicazione d'interesse.

1.1 *Architetture Riconfigurabili*

La ricerca accademica ha proposto numerose soluzioni ispirate al paradigma riconfigurabile. Tra queste si è scelto di dare una breve descrizione di quelle più correlate al presente lavoro.
Si consideri ad esempio il sistema PRISC [8]. Pur non essendo stato mai realizzato fisicamente, PRISC è molto interessante perché estende l'ISA di un microprocessore MIPS, permettendo l'esecuzione delle istruzioni estese attraverso un array riconfigurabile chiamato PFU. Una limitazione di questo sistema risiede nell'impossibilità delle PFU di stallare la pipeline del processore per eseguire istruzioni multiciclo e di poter accedere alla memoria. Dal punto di vista dello sviluppo software, l'idea proposta è quella di utilizzare un compilatore ad hoc ma,

negli esperimenti realizzati, le istruzioni estese sono state inserite direttamente nell'eseguibile delle applicazioni considerate.

Un'evoluzione di PRISC è rappresentata da GARP [9] in cui la porzione riconfigurabile del sistema è soggetta al controllo del programma in esecuzione che la utilizza per eseguire alcuni loops e subroutines con l'obiettivo di aumentare la velocità d'esecuzione. Quest'operazione è molto dispendiosa poiché, oltre al trasferimento dei dati da processare, il programma deve gestire anche l'overhead legato alle differenti configurazioni dell'array riconfigurabile. A differenza di PRISC comunque, in GARP la porzione riconfigurabile può accedere alla gerarchia di memoria rendendo così più efficiente l'esecuzione di tasks data-intensive. Tuttavia, l'accesso avviene considerando gli indirizzi fisici e non vi è alcun supporto alla memoria virtuale, il che introduce problemi di sicurezza dei dati in un contesto general purpose e multitasking.

Un passo avanti rispetto a GARP, è proposto in [10] dove è stato utilizzato un approccio misto tra ASIC e FPGA. In ASIC è stata implementata una CPU RISC mentre la FPGA, collegata via bus, è utilizza per implementare dinamicamente sia coprocessori dedicati che periferiche. L'utilizzo della FPGA è tuttavia limitato dal fatto che non è possibile accedere al register file, gli accessi in memoria sono possibili solo verso un buffer locale e non esiste nessun supporto alla memoria virtuale, il che porta agli stessi problemi di sicurezza sui dati, precedentemente osservati per GARP.

eMIPS [1] rimuove gran parte di queste limitazioni inserendo nello stesso dispositivo FPGA una CPU MIPS e un certo numero di slot riconfigurabili dinamicamente per mezzo della partial reconfiguration, caratteristica delle FPGA di nuova generazione.

All'interno degli slot riconfigurabili è possibile inserire della logica custom, chiamata Estensione, che può essere sfruttata sia per implementare delle unità funzionali specializzate che delle periferiche. Le Estensioni sono in grado di accedere alle risorse del processore e, in particolare, l'accesso alla memoria avviene per mezzo del sottosistema di memoria virtuale. La gestione delle Estensioni, inoltre, è sotto il controllo del software di sistema il che risolve il problema di sicurezza nel caso di un utilizzo in ambiente multi programmato.

Un altro approccio, che si basa più sulla possibilità di estendere staticamente dei processori che sulla riconfigurabilità dinamica, è il processore Xtensa proposto da Tensilica [11]. In questo caso viene considerata una CPU base RISC single issue. A partire dal profiling delle applicazioni d'interesse, vengono individuate delle unità funzionali specializzate usate per estendere il processore base che verrà implementato su ASIC. A questo punto l'ISA del processore contiene anche delle istruzioni estese che devono essere chiaramente tenute in considerazione nel

momento in cui si vuole sfruttare il massimo potenziale del sistema generato. A tal fine, è automaticamente generata anche la tool chain per lo sviluppo software.

1.2 *Motivazioni*

La maggior parte delle architetture considerate nel paragrafo precedente, non permette di utilizzare direttamente il tradizionale modello di programmazione sequenziale.
Ad esempio, dal punto di vista dell'interfaccia hardware/software, GARP è assimilabile ad un sistema multiprocessore eterogeneo che necessita di un supporto di compilazione molto complesso e una certa capacità nella programmazione parallela. Questo non rende immediato l'uso di GARP per il programmatore software. Inoltre, a differenza del tool che è stato sviluppato nel presente lavoro, il compilatore proposto per lo sviluppo su GARP [12] è fortemente dipendente dalla particolare struttura dell'hardware.
Un altro esempio è rappresentato da ROCCC [13] in cui il compilatore è progettato per generare dei circuiti da implementare su logica riconfigurabile a partire dal profiling del codice sorgente delle applicazioni. In questo caso, però, la selezione delle parti di codice da eseguire nelle nuove unità funzionali, viene fatta tenendo conto solo del tempo di esecuzione software e non dell'informazione relativa all'accelerazione ottenibile in hardware.
Considerando il sistema di sviluppo proposto da Tensilica, si può osservare come il compilatore (XPRESS) utilizzi il codice sorgente delle applicazioni di interesse, per estendere l'architettura del processore Xtensa in una più efficiente nell'eseguire codice originale. A tal fine il compilatore esplora rapidamente milioni di possibili configurazioni del processore utilizzando diverse tecniche di accelerazione e selezionando quelle che maggiormente aumentano le prestazioni del codice originale. Una volta individuata la configurazione ottimale, viene generata la descrizione del processore esteso e il compilatore con cui sarà possibile ottenere i binari eseguibili sulla la nuova architettura.
Nello sviluppo del software per sistemi riconfigurabili, ricoprono una notevole importanza anche i sistemi per eseguire il debugging del software. In un contesto riconfigurabile, questi strumenti devono far fronte a nuovi vincoli ma possono anche sfruttare le potenzialità offerte dal paradigma riconfigurabile per offrire al programmatore strumenti più efficaci nella fase di testing.
Da questa analisi, è evidente la necessità di avere dei tools sofisticati che permettano allo sviluppatore software di sfruttare i vantaggi delle architetture riconfigurabili. In questo lavoro si è fatto particolare riferimento a quelle basate sui dispositivi FPGA

che, nel corso degli ultimi anni, hanno avuto un notevole incremento di prestazioni. Questo ha portato a vedere la FPGA non più solo come strumento indispensabile per la prototipazione, ma anche come dispositivo adatto all'utilizzo in domini applicativi tradizionalmente propri di microprocessori, DSP e ASIC.

Nel Capitolo 2 vengono descritti i dispositivi FPGA con particolare attenzione verso la famiglia Virtex4 della Xilinx [14]. La scelta di utilizzare questi ultimi piuttosto che quelli di Altera (la quale insieme a Xilinx occupa quasi l'intera quota di mercato) è data esclusivamente dalla facilità di accesso da parte dell'autore ai tools di sviluppo e ai dispositivi Xilinx.

Il Capitolo 3 riguarda lo sviluppo e l'implementazione di un sistema in grado di modificare il codice sorgente di un'applicazione, per sfruttare in modo trasparente il potenziale offerto da un'architettura riconfigurabile.

Nel Capitolo 4 viene descritta la realizzazione di eBug, un approccio al debugging software che sfrutta le potenzialità offerte dal paradigma riconfigurabile ed in particolare dalla partial reconfiguration delle FPGA.

La tesi si conclude con il Capitolo 5 che sottolinea i risultati più importanti e delinea alcuni possibili sviluppi futuri.

Capitolo 2

I Dispositivi FPGA

2.1 *Concetti Generali*

Le Field Programmable Gate Array (FPGA), insieme ai Programmable Array Logic (PAL) e ai Complex Programmable Logic Devices (CPLD), appartengono alla classe dei dispositivi riconfigurabili, hanno cioè la capacità di poter essere configurati per implementare funzionalità di complessità variabile un numero pressoché illimitato di volte. La loro configurazione è generalmente memorizzata su una memoria non permanente per cui il dispositivo ha la necessità di essere inizializzato all'accensione. La flessibilità della FPGA le rende una piattaforma ottimale per la prototipazione, per implementare della glue logic, per lo sviluppo di System on Programmable Chip (SoPC) e per l'implementazione di architetture dinamicamente riconfigurabili.

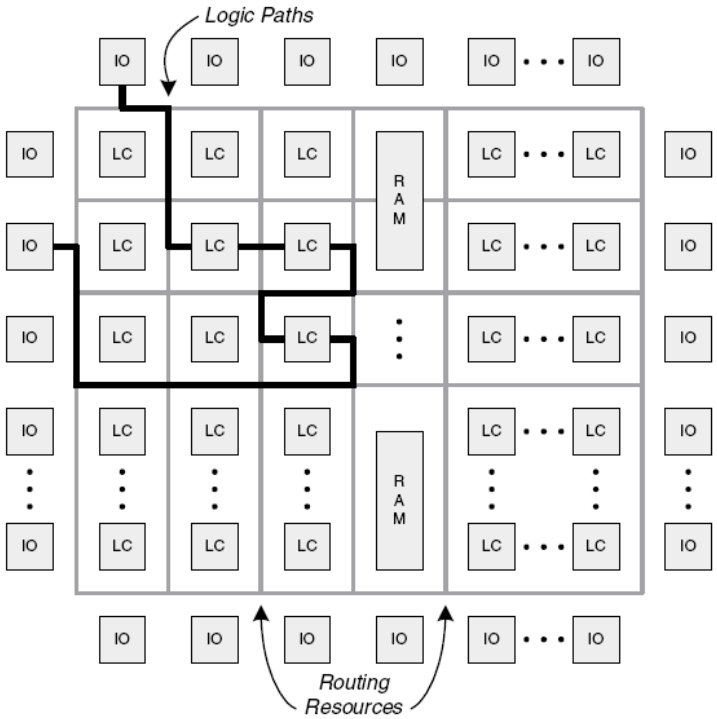


Figura 2.1: Struttura di una FPGA

La struttura di base di una FPGA è rappresentata nella Figura 2.1. Le funzionalità logiche del dispositivo sono fornite da una matrice di logic cell (LC) con la possibilità d'interconnessione tra loro.
Ogni LC possiede una look up table (LUT) ossia una memoria ram a *n* ingressi e un'uscita, che permette di realizzare una qualunque funzione logica di *n* variabili. L'utilizzo delle LUT permette, a livello del singolo LC, di avere delle prestazioni che sono invarianti rispetto alla complessità della funzione logica implementata. Si consideri l'esempio di Figura 2.2. Le variabili O_1 e O_2 rappresentano due diverse funzioni logiche degli ingressi A, B, C. In entrambi i casi è sufficiente l'utilizzo di una LUT 4x1 per cui il ritardo tra ingresso e uscita è sempre il ritardo di propagazione della memoria con cui è realizzata la LUT. Nella parte destra della figura invece si nota come le due funzioni logiche abbiano una differente implementazione nel caso delle standard cell. La funzione O_1 attraversa un livello di logica mentre la O_2 ne attraversa tre. In quest'ultimo caso è quindi evidente che il ritardo di propagazione non sia invariante rispetto alla complessità della funzione implementata.

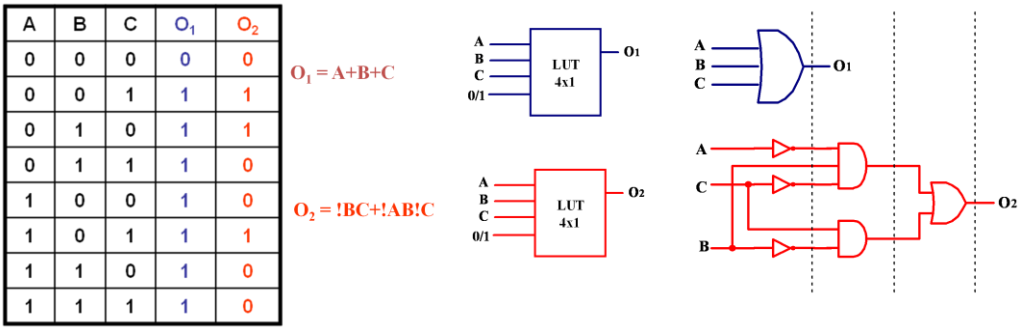


Figura 2.2: Differenza tra LUT e Standard Cells

All'interno di un LC sono inoltre presenti elementi di memorizzazione (flip-flop D) e d'instradamento dei dati (multiplexer).
Una rappresentazione molto semplificata di un LC è quella rappresentata in Figura 2.3.

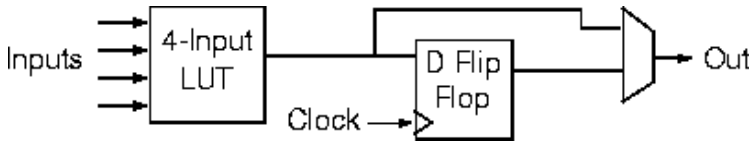


Figura 2.3: Schema di una logic cell

Ogni LC si può connettere agli altri mediante switch box posti alle intersezioni delle risorse di routing. Tutte le connessioni sono configurabili garantendo così la massima flessibilità del dispositivo. Altrettanto configurabili sono le interconnessioni con gli I/O pad, che forniscono il mezzo di comunicazione del chip con l'esterno permettendo, ad esempio, la scelta dello standard elettrico da utilizzare. Le FPGA sono inoltre dotate anche di altre risorse configurabili tra cui come clock manager, blocchi di memoria RAM e moltiplicatori hardwired.

Da questa breve descrizione si possono osservare le notevoli differenze che intercorrono, ad esempio, tra i microprocessori e le FPGA. I primi sono dei dispositivi programmabili, con un'architettura fissata, che variano il loro comportamento in funzione del software in esecuzione in un particolare momento. Le FPGA, invece, sono dispositivi configurabili, cioè variano la funzionalità che implementano modificando la configurazione delle risorse hardware interne. In questo modo è sempre possibile avere un'architettura specializzata per un determinato task con i vantaggi che questo comporta.

La FPGA si pone, quindi, in una posizione intermedia tra la flessibilità estrema di un microprocessore general purpose e i circuiti integrati per applicazioni specifiche (ASIC) mostrando, rispetto a questi ultimi, un difetto per quel che riguarda prestazioni velocistiche, di consumo e di costo su alti volumi di produzione, ma un vantaggio per quel che riguarda la flessibilità.

La progettazione di sistemi basati su FPGA segue un design flow molto simile alla progettazione dei dispositivi ASIC. Comincia con il design entry, ossia la modalità con cui il progettista deve descrivere il progetto da implementare sul dispositivo. Questo avviene tipicamente per mezzo di linguaggi HDL (VHDL e Verilog sono i più usati), schematici o netlist. Il design è successivamente sintetizzato e implementato e sottoposto a simulazioni ai vari stadi per verificare la consistenza con le specifiche. Se le simulazioni hanno dato risultati soddisfacenti, viene generato un file e trasferito nella memoria di configurazione del dispositivo che implementa così la funzionalità richiesta.

Chiaramente la flessibilità delle FPGA ha un costo rispetto alla rigidità degli ASIC. Ad esempio, la frequenza massima di clock ottenibile è, allo stato attuale, di poco superiore ai 500MHz, mentre un microprocessore moderno può raggiungere frequenze che superano i 3GHz. La FPGA inoltre ha un consumo di potenza superiore rispetto allo stesso sistema realizzato in ASIC. Il mercato delle FPGA però è in continua evoluzione e spinge per colmare questo gap fornendo dei dispositivi sempre più complessi e con features orientate verso l'efficienza. Ad esempio, sono in commercio FPGA che integrano, nello stesso chip, anche dei microprocessori o dei

moduli ASIC per il digital data processing e per le trasmissioni seriali veloci. Un esempio è il dispositivo Xilinx Virtex4 che è stato come FPGA di riferimento per il lavoro presentato in questo documento.

2.2 *Xilinx Virtex 4*

Le FPGA Xilinx Virtex4 [14] introducono il concetto di *platform FPGA*, ossia dispositivi le cui risorse a disposizione dell'utente, sono dimensionate in funzione dei domini applicativi d'interesse. La Xilinx fornisce tre differenti famiglie di Virtex4 (LX, SX, FX) realizzate con tecnologia copper-CMOS, 90nm a triplo ossido, che permette al dispositivo di funzionare con clock fino a 500MHz. Le tre famiglie di Virtex4 condividono la medesima architettura a colonne, chiamata ASMBL, che fornisce numerosi vantaggi rispetto ai precedenti dispositivi [15] ma, la LX è ottimizzata per applicazioni logiche, la SX per applicazioni DSP high-end e la FX per l'implementazione di sistemi embedded e trasmissioni seriali veloci. Ad eccezione della FX, che possiede anche uno o due microprocessori PowerPC integrati e blocchi per la trasmissione e ricezione seriale, le tre piattaforme possiedono lo stesso tipo di risorse ma in bilanciate in modo differente (Figura 2.4). Ad esempio, la LX ha mediamente un maggior numero di risorse riconfigurabili su cui implementare logica (Slices), mentre la SX possiede un maggior numero di risorse utili per compiere operazioni di somma, moltiplicazione e accumulazione. Per maggiori dettagli sulle risorse disponibili, si può fare riferimento a [16].

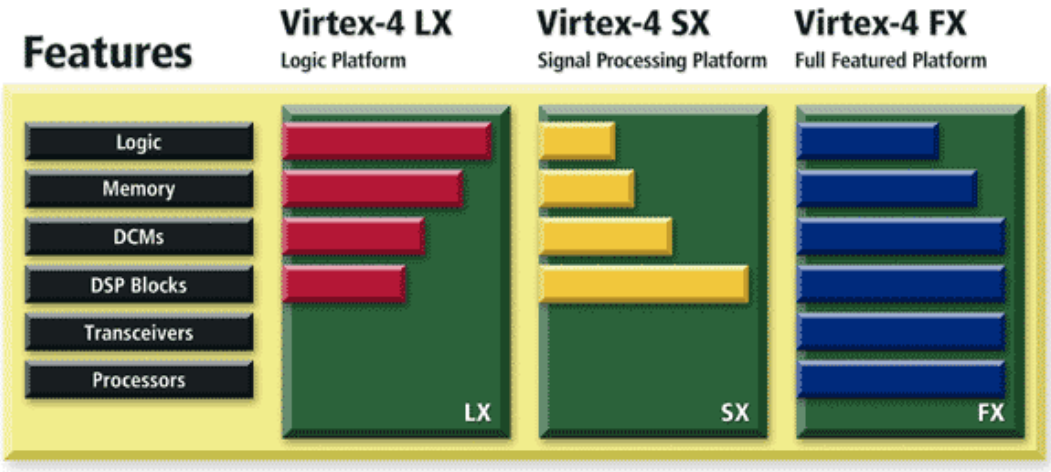


Figura 2.4: Virtex4 LX, SX, FX [17]

Nei seguenti sottoparagrafi è fornita una breve descrizione sulle risorse contenute nelle FPGA Virtex4.

2.2.1 **Interfaccia di I/O**

L'interfacciamento della Virtex4 con il mondo esterno è organizzato in strutture chiamate I/O Tiles (Figura 2.5). Ognuna di queste è costituita da due pads, due I/O blocks (IOB), due logiche di ingresso (ILOGIC) e due logiche d'uscita (OLOGIC). In alternativa alle ILOGIC e OLOGIC, si possono avere delle logiche dedicate (ISERDES e OSERDES) usate la conversione seriale-parallela dei dati nei casi di applicazioni in cui sia necessaria una comunicazione seriale veloce di tipo source synchronous.

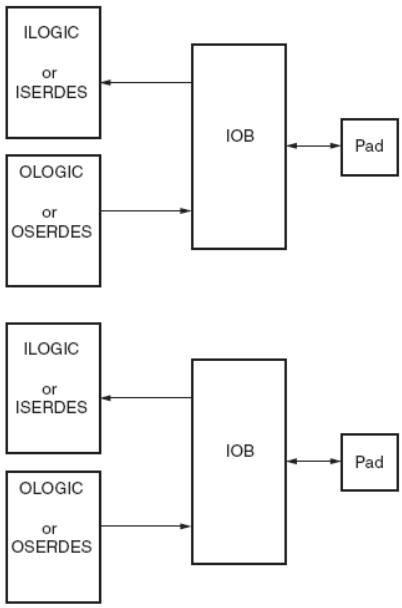


Figura 2.5: I/O Tile

Gli IOB, la cui struttura base è mostrata in Figura 2.6, sono il primo elemento configurabile del sistema e sono collegati direttamente ai pin del dispositivo. La loro configurabilità permette di utilizzare un pin come input o come output, di supportare diversi standard elettrici, differenti livelli di correnti d'uscita, di slew rate e, inoltre, di selezionare l'impedenza desiderata. I blocchi di I/O presenti in una I/O Tile possono, inoltre, essere usati a coppie per operare segnali di tipo differenziale.

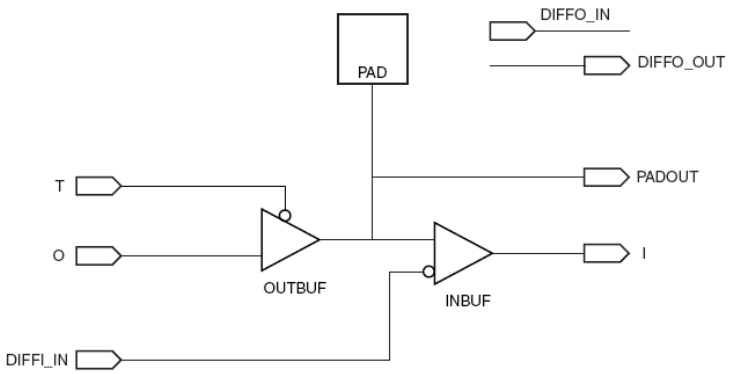


Figura 2.6: Schema base di un blocco di I/O

I moduli ILOGIC (Figura 2.7) possiedono fondamentalmente quattro elementi di memorizzazione e una linea di delay programmabile. Tutti i quattro elementi sono configurabili sensibili al livello o al fronte del segnale. In generale è usato solo un elemento quando si desidera registrare un dato in ingresso. Gli altri tre sono usati per implementare registri d'ingresso di tipo DDR. Il delay programmabile permette di associare un ritardo prefissato e deterministico al dato in ingresso.

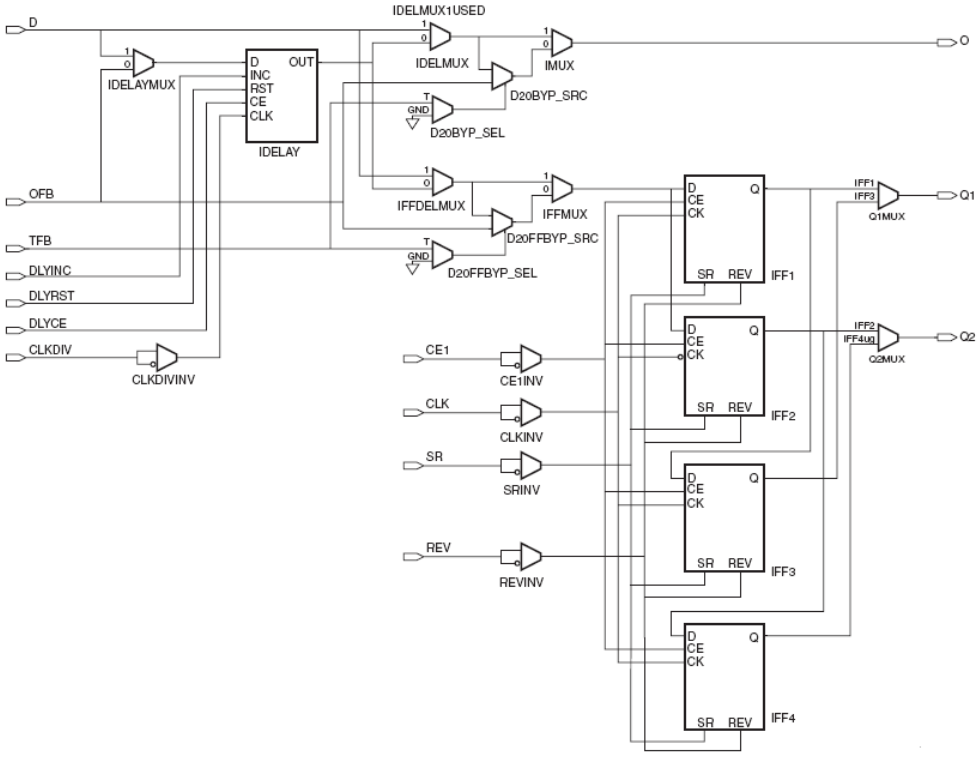


Figura 2.7: Diagramma della logica di Input

I moduli OLOGIC (Figura 2.8) possiedono sei elementi di memorizzazione. Il numero maggiore, rispetto a quelli contenuti negli ILOGIC, è necessario per pilotare

sia i segnali relativi al dato, che quelli relativi allo stato del buffer three-state dell'IOB. Similmente a quanto rilevato in precedenza per la logica d'ingresso, in generale viene usato solo un registro per il dato e uno per pilotare il buffer three-state. Gli altri sono usati per implementare operazioni DDR.

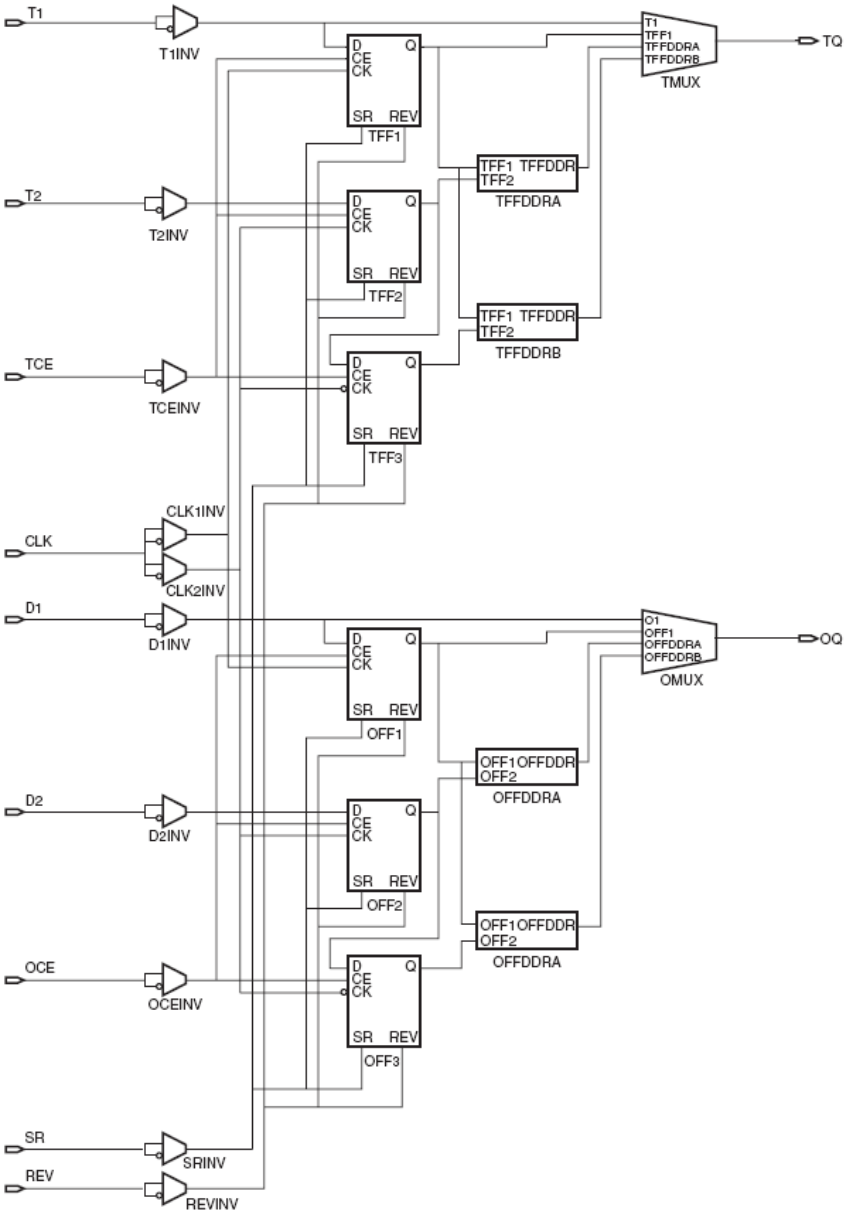


Figura 2.8: Diagramma della logica di output

2.2.2 **Blocchi di logica configurabile (CLBs)**

I Configurable Logic Blocks (CLB) rappresentano la risorsa principale per l'implementazione sia di logica sequenziale che di logica combinatoria. Una FPGA

contiene un array di CLB ognuno dei quali è collegato a una switch matrix ossia un sistema d'interconnessione configurabile che permette di collegare i CLB tra loro e alle altre risorse a disposizione della FPGA.
In Figura 2.9 è possibile vedere la struttura di un CLB. Al suo interno si trovano quattro elementi interconnessi chiamati *slices*. Le slices sono raggruppate a coppie e ogni coppia forma una colonna, quella di sinistra contiene le slice di tipo M (SliceM) e quella di destra le slice di tipo L (SliceL).

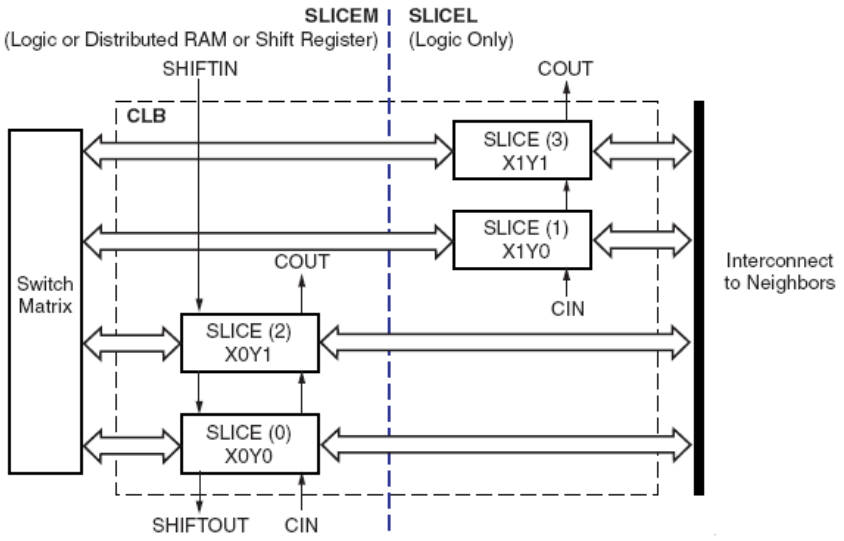


Figura 2.9: Struttura di un CLB

Sia le SliceM (Figura 2.10) che le SliceL (Figura 2.11), contengono due generatori di funzioni logiche (le look-up tables), due elementi di memorizzazione, che possono essere configurati come flip-flop D o come Latch D, alcuni multiplexer, carry chains dedicate, per implementare in modo efficiente funzioni aritmetiche, e porte logiche. Questi elementi sono utilizzati per implementare funzioni logico-aritmetiche e memorie ROM.
Le SliceM, inoltre, sono in grado di implementare delle memorie RAM distribuite (single o dual port, con lettura sincrona o asincrona) e uno shift register a 16 bit in una unica LUT con un notevole risparmio di risorse.
In realtà, l'utilizzo di queste ultime due features ha alcune limitazioni. Ad esempio, per utilizzare lo shift register in una LUT, è necessario che lo shift register non abbia reset altrimenti sarà sintetizzato utilizzando una i flip flop D contenuti nelle slices. Queste limitazioni sono molto importanti da tener presenti soprattutto quando si vuole inserire lo shift register, o in generale una risorsa dedicata, per inferenza utilizzando quindi una descrizione HDL. Alcuni utili consigli rispetto a quest'argomento possono essere trovati in [18].

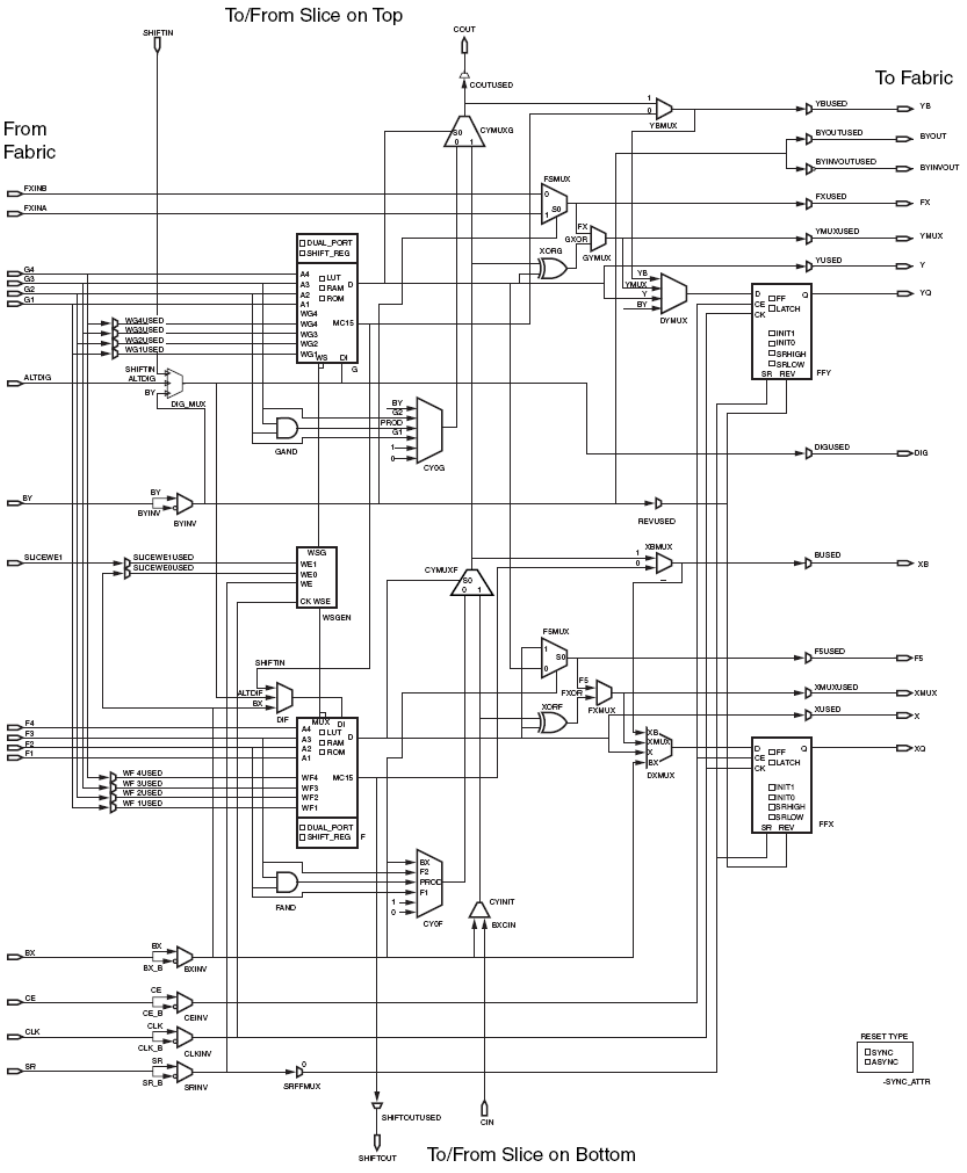


Figura 2.10: SliceM

Oltre alle LUT contenute nelle slices, che da sole possono generare funzioni logiche a quattro ingressi, il CLB contiene un certo numero di multiplexer (MUX5, MUX6, MUX7, MUX8) che permettono di generare funzioni logiche più estese. Questi multiplexer collegano gerarchicamente gli elementi interni di un CLB. In Figura 2.12 si può notare che:

- Il MUX5 collega due LUT permettendo, ad esempio, di realizzare una funzione logica a cinque ingressi la cui uscita è l'uscita del multiplexer stesso.

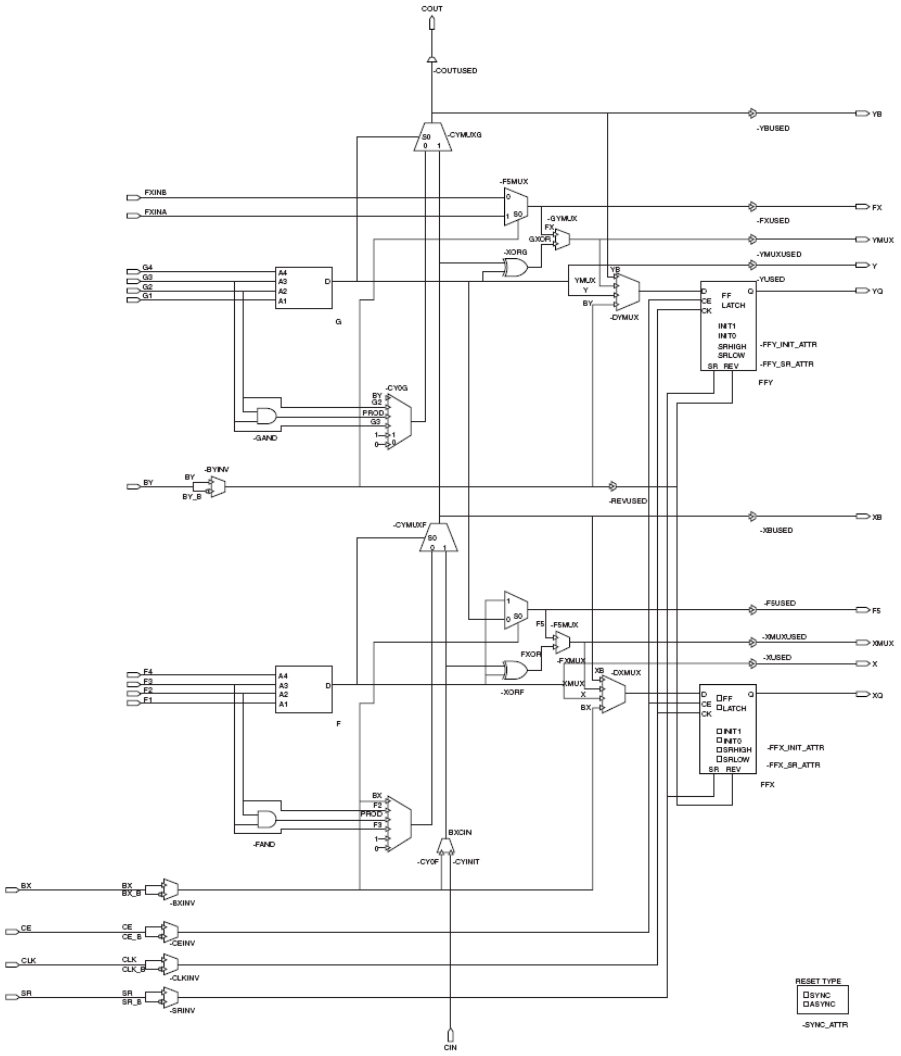


Figura 2.11: SliceL

- Il MUX6 collega le uscite dei due MUX5 appartenenti alla slices di una stessa colonna. In questo caso si ottengono funzioni logiche a sei ingressi.
- Il MUX7 collega le uscite dei due MUX6 contenuti in un CLB estendendo a sette il numero d'ingressi della funzione logica implementabile.
- Il MUX8 che collega le uscite dei MUX7 di due CLB differenti e permette di avere funzioni logiche a otto ingressi.

Ovviamente è possibile implementare funzioni logiche con un numero d'ingressi maggiore di otto collegando ad esempio le uscite dei MUX8 ma il collegamento non avverrebbe più in modo diretto e dedicato e quindi si avrebbe una perdita di efficienza sia dal punto di vista dell'occupazione d'area che del percorso critico.

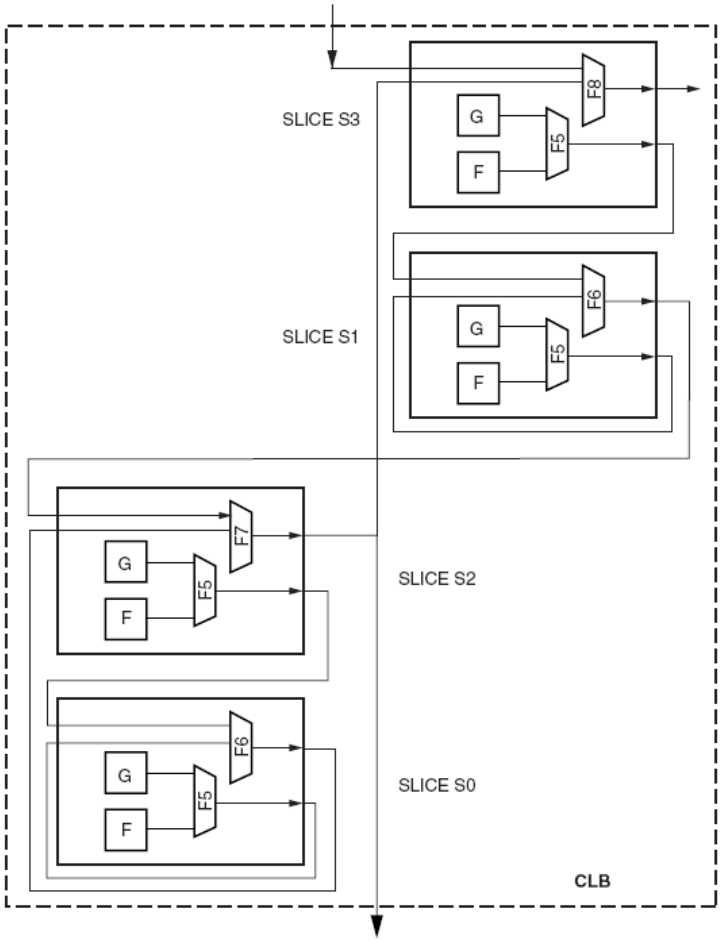


Figura 2.12: Gerarchia dei Multiplexers

2.2.3 **Blocchi di RAM**

Ogni Virtex4 contiene, in misura più o meno maggiore, delle Block RAM (BRAM). Le BRAM sono memorie configurabili da 18Kb ciascuna con cui si possono implementare delle RAM con fattori di forma che vanno da 16K elementi da 1 bit fino a 512 elementi da 36 bit. Chiaramente le BRAM possono essere collegate tra loro per ottenere delle memorie di dimensioni maggiori il cui limite è dato dal numero presente nel dispositivo. Le RAM sono implementabili in configurazione single o dual port e, in quest'ultimo caso, ogni porta è totalmente sincrona (a differenza delle RAM implementate con le LUT in cui si può accedere in lettura anche in modo asincrono) e permette di gestire in tre modi differenti (write first, read first, no operation) la situazione in cui si cerchi di accedere simultaneamente in lettura e scrittura su uno stesso indirizzo.

Le BRAM sono presenti in tutte le FPGA di Xilinx ma nella Virtex4 hanno una frequenza massima di lavoro superiore (500 MHz) e inoltre hanno il supporto built-in per implementare delle FIFO operanti alla stessa frequenza delle BRAM. Questo permette un'implementazione delle FIFO molto efficiente in termini di velocità (si ha circa un fattore 2x rispetto all'implementazione realizzata con CLB), di risparmio di risorse logiche e di semplicità di design.
Un altro elemento molto importante è il supporto per le memorie a correzione d'errore (ECC), molto usate negli ambiti in cui l'integrità dei dati è fondamentale (ad esempio servers e applicazioni aerospaziali). Anche in questo caso, essendo il supporto built-in si ha un incremento di efficienza rispetto alle soluzioni ottenute usando la logica distribuita.

2.2.4 **Xtreme DSP Slices**

Le Xtreme DSP slices (Chiamate anche slice DSP48) sono risorse dedicata per l'implementazione efficiente di applicazioni DSP potendo raggiungere frequenze di clock di 500MHz (nel caso siano configurate fully pipelined). Le slice DSP48 sono organizzate a coppie per formare una DSP Tile che è mostrata in Figura 2.13. Come si può osservare, ogni DSP48 contiene un moltiplicatore dedicato 18x18 bit, un sommatore e un accumulatore a 48 bit che agiscono su operatori senza segno o in complemento a due. Tutti i moduli aritmetici sono hardwired ossia non sono implementati in logica riconfigurabile ma direttamente integrati. La loro interconnessione è tuttavia configurabile come lo è anche l'interconnessione tra differenti slices DPS48.
L'utilizzo di queste risorse può essere fatto sia mediante l'istanza di primitive che per inferenza. Soprattutto nel secondo caso bisogna prestare particolare attenzione nella codifica HDL per ottenere i migliori risultati [19].

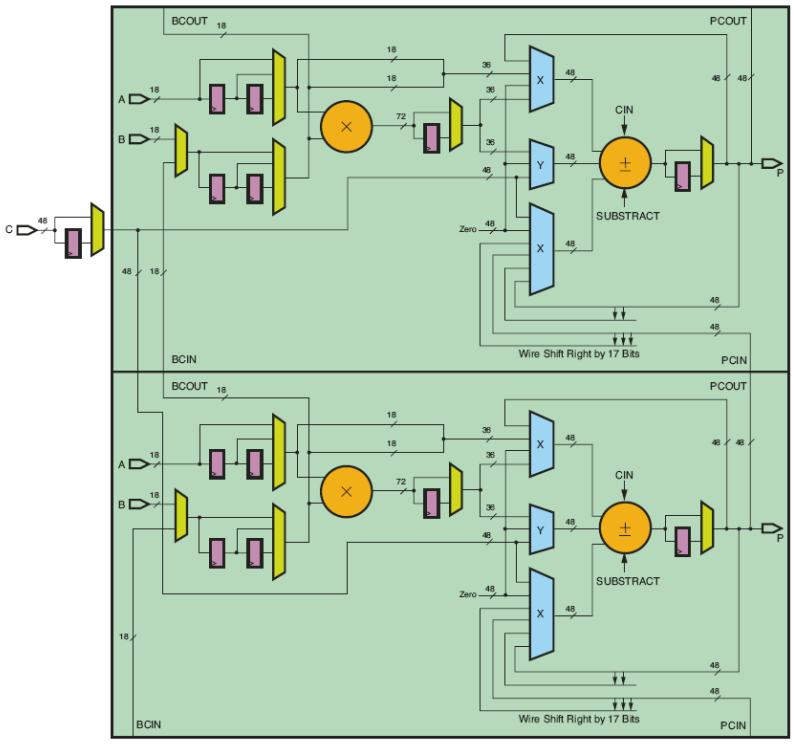


Figura 2.13: Tile DSP che comprende 2 Slice DPS48.

2.2.5 **Risorse per la distribuzione del segnale di clock**

Per fornire un clock preciso sia in termini di fase che di frequenza, la Virtex4 possiede numerose risorse. Oltre ad una struttura di routing dedicata per il trasporto di più segnali di clock all'interno del chip, sono presenti numerosi buffers, multiplexers bufferizzati e fino a venti DCM (Digital Clock Manager). Questi ultimi elementi rappresentano la risorsa fondamentale per avere un segnale di clock stabile e preciso poiché permettono di sintetizzare più clock con frequenze e fasi differenti provvedendo anche all'eliminazione dello skew per mezzo del DLL interno.
Una peculiarità tra risorse per la distribuzione del clock, non presente nelle FPGA precedenti della Xilinx, è il PMCD (Phase-Matched Clock Divider).

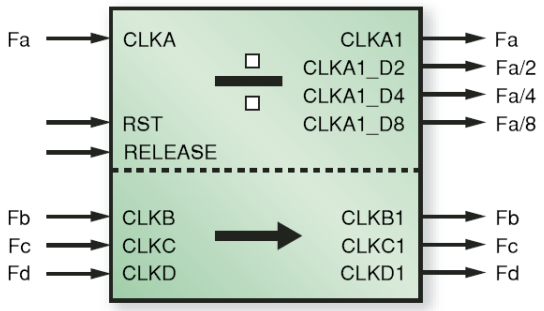


Figura 2.14: PMCD

Il PMCD (Figura 2.14) preserva l'allineamento dei fronti e delle relazioni di fase tra i 4 ingressi di clock considerando come riferimento il CLKA. Come si può vedere dalla Figura 2.14, all'ingresso CLKA corrispondono quattro uscite. La prima, CLKA1, ha la stessa frequenza dell'ingresso ma presenta un certo ritardo, le altre tre hanno frequenza pari alla metà, un quarto e un ottavo di CLKA ma sono allineate in fase con CLKA1.
Allo stesso modo le uscite corrispondenti agli ingressi CLKB, CLKC, CLKD, possiedono lo stesso ritardo, relativamente al loro ingresso, che intercorre tra CLKA e CLKA1. In questo modo il ritardo relativo tra gli ingressi è preservato sulle uscite.

2.2.6 **RocketIO Multi-Gigabit Tranceiver ed Ethernet MAC**

Le Virtex4 FX sono state concepite per fornire una piattaforma completa per applicazioni di tipo embedded che richiedono anche un alto grado di connettività con l'esterno sopratutto per quello che riguarda i protocolli seriali. Questi ultimi, infatti, grazie ai progressi tecnologici, stanno velocemente sostituendo quelli basati su comunicazione parallela, si consideri, ad esempio, il passaggio dal bus PCI parallelo al PCI-Express oppure all'utilizzo nei dischi del protocollo SATA.
La Virtex4 si basa sui RocketIO transceiver che, arrivati alla terza generazione, permettono di implementare protocolli seriali con velocità che vanno da 622 Mb/s fino a più di 10 Gb/s. Ogni tranceiver RocketIO è programmabile ed è in grado di implementare, con una relativa facilità di design, standards come PCI-Express, Serial-ATA, Gigabit Ethernet e altri ancora.
Oltre al RocketIO la Virtex4 Fx include al suo interno più moduli Ethernet MAC funzionanti nelle tre modalità (10/100/1000) rendendo molto semplice la realizzazione di sistemi embedded che hanno l'esigenza di comunicare tramite una connessione Ethernet.

2.2.7 **PowerPC core**

Oltre alla possibilità utilizzo di processori softcore come il Microblaze [20], Le Virtex4 della famiglia FX possiedono uno o due processori PowerPC (PPC) [21] integrati che possono funzionare con una frequenza massima di 450MHz. I PowerPC sono caratterizzati da un'architettura di Harvard, possiedono un ISA RISC a 32 bit con una pipeline standard a 5 stadi. E' inoltre presente una cache dati e una istruzioni di primo livello di 16KB. L'interfacciamento con il resto del dispositivo avviene sia mediante l'infrastruttura a bus CoreConnect di IBM [22] che tramite il controller OCM (On-Chip Memory) che funge da interfaccia dedicata tra il PowerPC e le BRAM permettendo un accesso veloce, non cacheable, alla memoria. Una caratteristica che rende ulteriormente flessibile l'uso del processore integrato nel dispositivo, è il controller APU (Auxiliary Processor Unit) che permette al progettista di estendere l'ISA nativo del PPC con istruzioni custom che sono eseguite da coprocessori implementati con l'utilizzo della logica riconfigurabile della FPGA. Questo controller permette una comunicazione molto più stretta tra i coprocessori e la pipeline del PPC rispetto all'utilizzo di un bus.

2.3 *Partial reconfiguration nelle FPGA Xilinx*

La Partial Reconfiguration (PR) [23] dinamica è una funzionalità delle FPGA di fascia medio-alta che consentono di configurare alcune parti del dispositivo mentre le altre sono in funzione senza interrompere le loro operazioni. Questa caratteristica, presente, ad esempio, nei dispositivi Xilinx della famiglia Virtex e in quelle Altera della famiglia Stratix, rende ulteriormente flessibili le FPGA. La PR è molto utile poiché consente di condividere temporalmente alcune risorse della FPGA. Si può pensare, ad esempio, al caso di un sistema in cui una parte opera continuativamente mentre altre sezioni possono essere incluse o rimosse per fornire nuove funzionalità.
Tradizionalmente quando si vuole cambiare, o modificare parzialmente il funzionamento del dispositivo, si procede a una sua completa riconfigurazione impedendo però che una sua parte possa continuare a operare. Questo diventa un problema per applicazioni in cui è necessario che sia garantita l'operatività di una parte critica. In Figura 2.15, ad esempio, è rappresentato un sistema che implementa un collegamento video, uno radio e una interfaccia per un bus che devono sempre essere in funzione per garantire la comunicazione dei sistemi connessi.

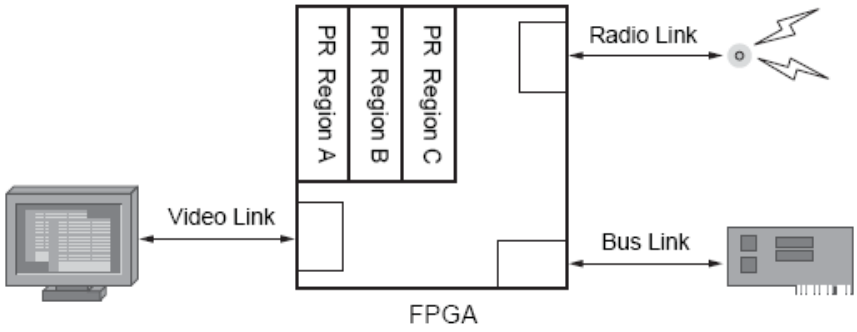


Figura 2.15: Utilizzo delle regioni riconfigurabili nelle FPGA

Utilizzando la PR è possibile generare dei file di configurazione parziali che permettono di modificare solo le regioni riconfigurabili dinamicamente (PR Region) A, B e C.

Come si può vedere in Figura 2.16 la FPGA è partizionabile in una regione base che è statica ossia non viene modificata durante le configurazioni parziali. In questa parte va implementata la logica che deve operare senza interruzioni e che si può occupare anche di gestire il caricamento di file di configurazione parziale per le regioni riconfigurabili (PRR A nella figura).

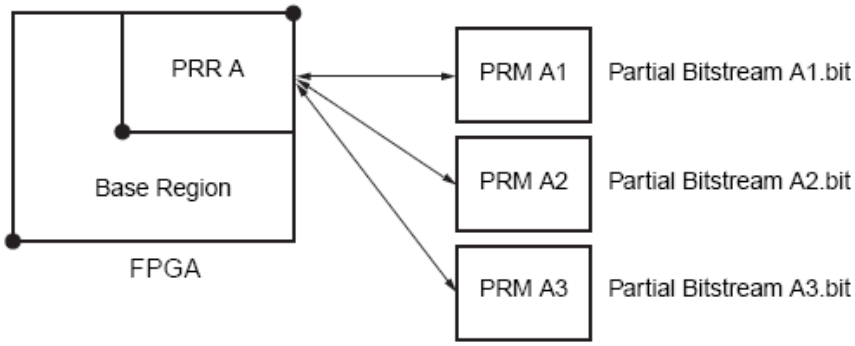


Figura 2.16: Regione statica e regioni riconfigurabili

Le regioni riconfigurabili sono soggette a delle restrizioni in termini di dimensione, placement e routing.

Per comprendere le limitazioni rispetto alla dimensione e al placement bisogna considerare che la memoria di configurazione delle FPGA della Xilinx è organizzata in "frames" che rappresentano la più piccola porzione del dispositivo che può essere configurata singolarmente e dinamicamente.

La suddivisione in frames è molto importante per il partizionamento della FPGA dato che è indispensabile che ogni singola regione (statica o dinamicamente

riconfigurabile) coincida, sia nelle dimensioni che nei confini, con un numero intero di frames.
Nelle FPGA Virtex II un singolo frame è costituito da una colonna di slices che si estende per l'intero dispositivo limitando la granularità delle regioni riconfigurabili dinamicamente. Nelle più moderne Virtex4 il singolo frame è invece costituito da una colonna di sedici slices. In questo modo in un dispositivo ogni colonna è costituita da più frames (ad esempio la Virtex4 LX25, che possiede 192 righe di slices, ha dodici frames per colonna).
Per quello che riguarda il routing, è necessario fare in modo che i segnali che attraversano i confini di una regione riconfigurabile siano consistenti con il resto del progetto. I problemi potrebbero infatti nascere se si riconfigura una di queste regioni introducendo dei segnali che attraversano i confini in punti differenti.

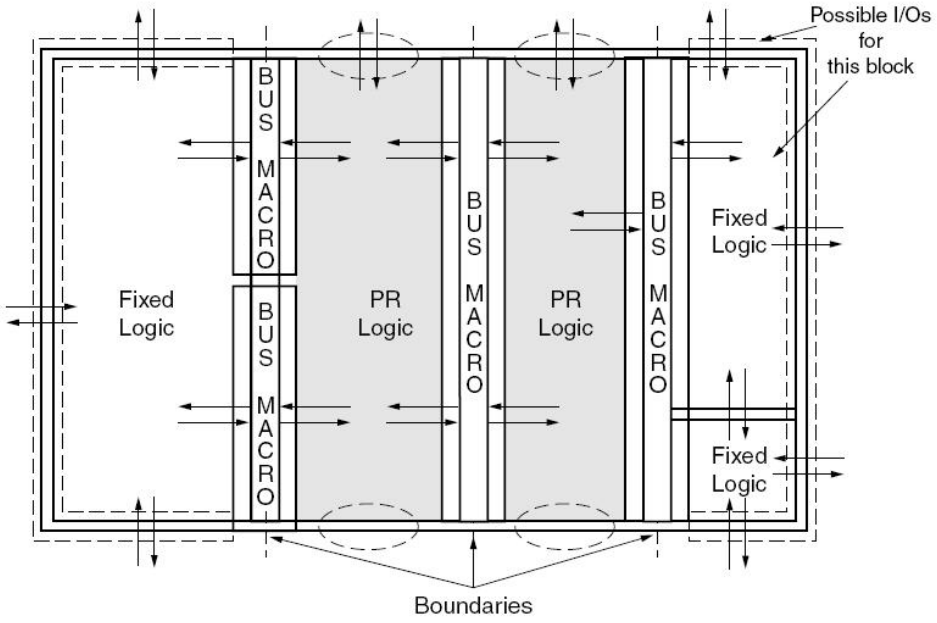


Figura 2.17: Esempio di connessione tra logica statica e logica riconfigurabile

Questo produrrebbe una chiara inconsistenza nel routing che forzerebbe l'eliminazione del segnale. Per superare questo problema, il routing dei segnali di interfaccia viene forzato ad attraversare dei moduli con placement e routing prefissati chiamati "bus macro". Le regioni riconfigurabili sono quindi interfacciate tra loro e con la logica statica mediante queste macro come mostrato in Figura 2.17. L'implementazione dei "bus macro" è differente tra le famiglie di dispositivi della Xilinx. Infatti, fino alla Virtex e VirtexII, venivano usati dei buffer tristate (TBUF) per instradare i segnali tra le regioni adiacenti come mostrato in Figura 2.18.

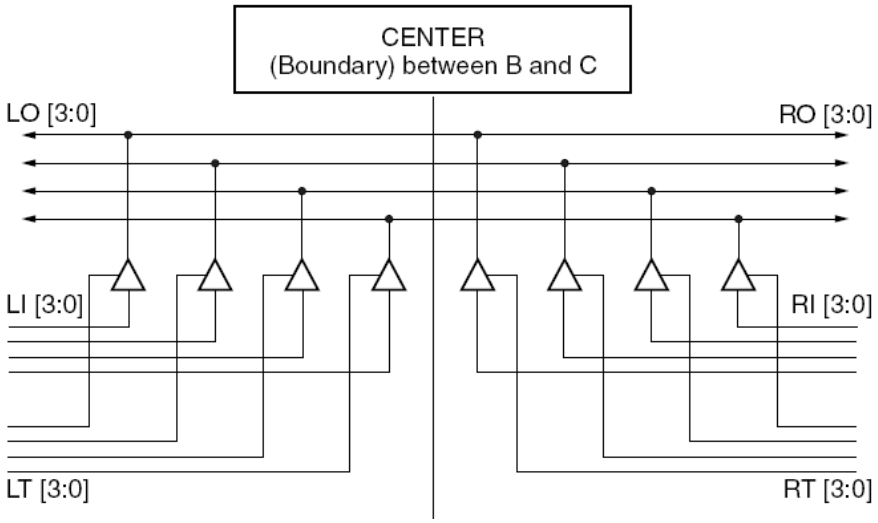


Figura 2.18: Bus Macro basata su TBUF.

Con l'introduzione delle Virtex4, i TBUF sono stati esclusi obbligando gli sviluppatori che avevano intenzione di usare la PR a utilizzare bus macro basati su LUT (Figura 2.19) che hanno un certo spreco di risorse dato che per ogni segnale vengono usate 2 LUT (una per direzione).

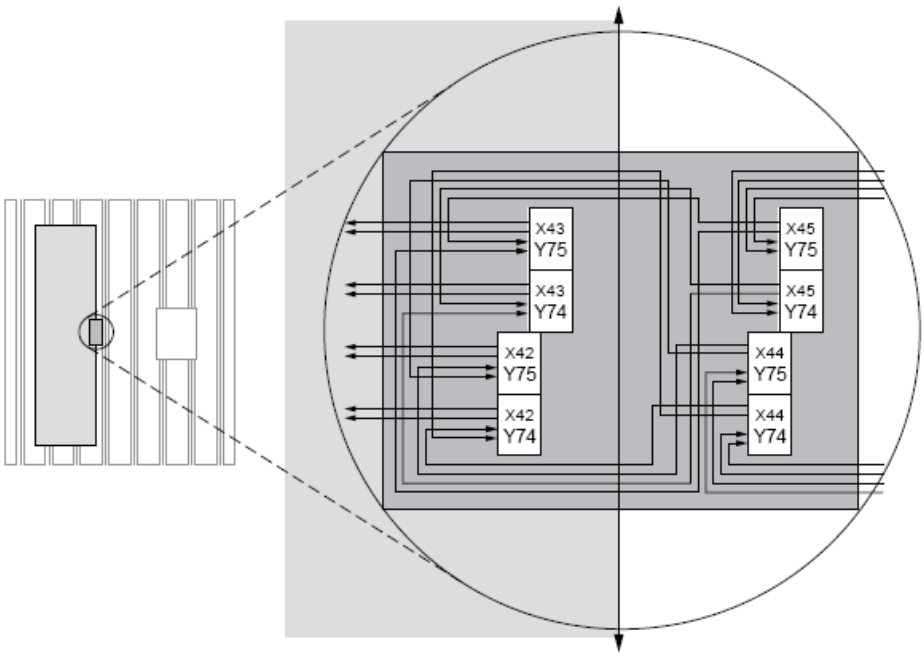


Figura 2.19: Bus Macro basata su LUT.

Un'altro problema nell'uso della PR risiedeva nello scarso supporto fornito dal tool Xılinx ISE. Successivamente la Xilinx ha messo a disposizione un tool chiamato

Planahead [24] che fornisce un supporto migliorato per il flusso di progettazione che utilizza la PR. Le fasi di sviluppo di un progetto basato su PR sono: Design Entry, Initial Budgeting, Active Module e Final Assembly. Una loro breve descrizione è data in [1] mentre una descrizione più dettagliata si trova in [23] [25] [26].

Capitolo 3

Partizionamento hardware/software per architetture riconfigurabili

In questo capitolo è presentato un sistema in grado di ottimizzare l'implementazione di un codice C per un'architettura riconfigurabile composta di una CPU e da una FPGA usata come coprocessore riconfigurabile. In generale i vantaggi nell'uso dei sistemi riconfigurabili si riducono notevolmente se il loro utilizzo obbliga l'utente a doverne conoscere i dettagli. Tenendo in considerazione che la maggior parte degli sviluppatori ha familiarità con linguaggi di programmazione di altro livello e non con linguaggi di descrizione dell'hardware, si è voluto sviluppare un'applicazione che nascondesse al programmatore i dettagli hardware riguardanti l'elemento riconfigurabile del sistema.

Come sarà descritto più approfonditamente nel seguito del capitolo, l'applicazione riceve in ingresso un codice sorgente C individuando e selezionano quelle parti che, eseguiti sul coprocessore piuttosto che sulla CPU, consentano di aumentare la velocità di esecuzione. Le parti di codice sorgente selezionate per l'esecuzione sul coprocessore, sono sostituite con delle chiamate a funzione. L'output dell'applicazione è, quindi, un codice C modificato, una libreria di funzioni, e un file di configurazione per il coprocessore FPGA.

Grazie a questa applicazione il programmatore che vuole sfruttare le potenzialità del sistema riconfigurabile può utilizzare il modello di programmazione che ha sempre usato.

Durante la fase di concezione e sviluppo dell'applicazione si è preso in esame uno studio [27] sull'estensione dell'ISA dei processori, modificandone alcune parti per renderlo più preciso e in particolare più efficiente rispetto all'architettura riconfigurabile considerata. Per l'implementazione sono stati utilizzati dei framework di compilazione molto conosciuti in ambito accademico (SUIF [28] e MachSUIF [29]) di cui verrà fatta una breve descrizione nel paragrafo successivo.

3.1 *SUIF e MachSUIF*

Per fornire un adeguato supporto allo sviluppo software in ambiente riconfigurabile, è stato necessario considerare dei framework che facilitassero l'analisi e la trasformazione del codice sorgente. Si è fatto quindi riferimento a dei tool noti in campo accademico che mettono a disposizione degli strumenti adeguati per lo sviluppo di tutte le parti di un compilatore. In questo paragrafo viene fatta una breve descrizione di SUIF [28] e MachSUIF [29]. Per quel che riguarda i concetti relativi ai principi sui compilatori, sia di base che avanzati, si considerino [30] [31] [32].

3.1.1 **Suif**

Suif [28] è un sistema per lo studio e la ricerca nel campo delle tecniche di compilazione, basata su una rappresentazione intermedia estendibile del codice sorgente (Stanford University intermediate form). Questo tipo di approccio è pensato per favorire la scrittura di passi riutilizzabili e per provvedere un livello di astrazione adatto per una facile collaborazione anche tra gruppi di ricerca differenti.
Il sistema Suif 2 si caratterizza principalmente per due aspetti:

1. Un sistema modulare che permette lo sviluppo di diversi componenti che possono essere costrutti per la rappresentazione del programma, o passi di analisi e trasformazione del programma. Un compilatore può essere quindi realizzato fondamentalmente in due modi: un insieme di programmi a se stanti che leggono, elaborano e riscrivono un file suif (derivato dal programma che si vuole compilare mediante il front-end c2suif), o un programma che importa dinamicamente un insieme di moduli che vengono poi applicati al file suif caricato precedentemente in memoria (Figura 3.1);
2. Una rappresentazione intermedia estendibile, che permette all'utente di creare nuovi nodi al fine di rappresentare nuove istruzioni o nuovi costrutti semantici. La rappresentazione intermedia (IR) è realizzata mediante una gerarchia di oggetti, con differenti livelli d'astrazione, da cui il programmatore può ereditare alcune caratteristiche per creare oggetti più specifici alle sue necessità. Il Suif IR avrà sempre dei nodi base, che contengono le informazioni principali, ma permette di creare un sottoinsieme di nodi estensione dei primi. La scrittura di nuovi nodi IR è fatta mediante un linguaggio ad alto livello, denominato Hoof, che permette al programmatore di disinteressarsi riguardo all'effettiva implementazione degli oggetti creati. Il file Hoof viene, infatti, processato dal tool Smgn (Suif Macro Generator),

che crea automaticamente i files d'interfaccia e il file di implementazione per i nuovi nodi.

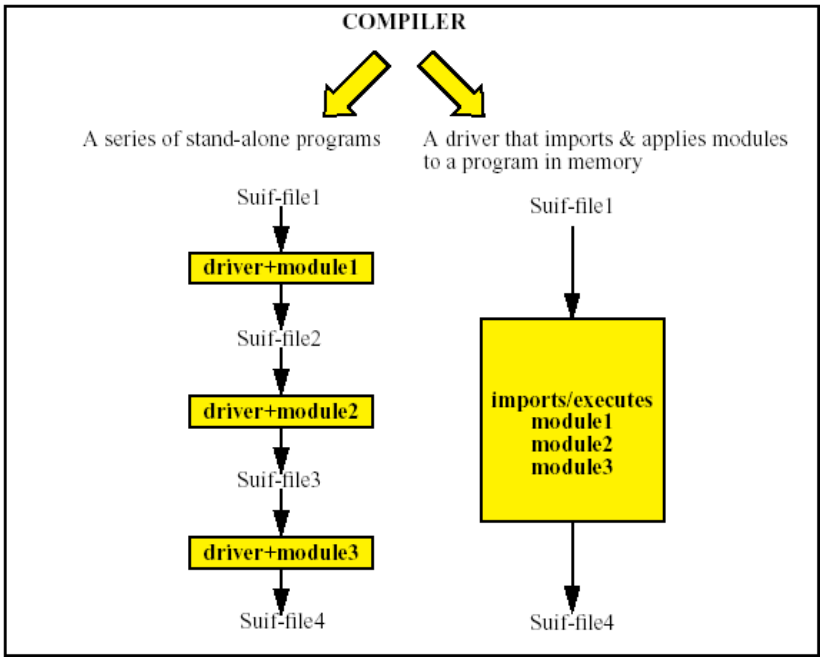


Figura 3.1: Struttura modulare del compilatore Suif

3.1.2 **MachSuif**

Machine SUIF è una struttura flessibile ed estendibile per costruire back end di compilatori. Con esso si possono rapidamente costruire e manipolare rappresentazioni intermedie di livello macchina e produrre in output codici assembly, binario o C. Il sistema fornisce dei back end già pronti per le architetture Aplha e x86. Si possono facilmente aggiungere nuove macchine target e sviluppare ottimizzazioni.
Anche se Machine SUIF si basa sul sistema Stanford SUIF (versione 2.1), i passi di analisi e ottimizzazione scritti per MachSuif non sono specifici per SUIF, ovvero non referenziano direttamente nessun costrutto SUIF e non incapsulano costanti di alcuna architettura specifica.
Le librerie fornite con MachSuif consentono un'agevole manipolazione delle rappresentazioni intermedie e forniscono tutta una serie di metodi per analizzare i dati e le dipendenze intercorrenti tra essi.

Un backend è fondamentalmente uno strumento per tradurre in codice assembly o oggetto le rappresentazioni intermedie. Con MachSuif si può facilmente implementare un blocco equivalente alla serie middle end + backend vista in precedenza. Si tratta sostanzialmente di applicare ai dati rappresentati sotto forma di IR una serie di passi di ottimizzazione, e quindi di produrre l'assembly specifico per l'architettura target. Ad esempio il più semplice backend è il seguente:

```
do_lower  hello.suif hello.lsf
do_s2m    hello.lsf hello.svm
do_gen    -target_lib alpha hello.svm hello.avr
do_il2cfg hello.avr hello.afg
do_raga   hello.afg hello.ara
do_cfg2il hello.ara hello.ail
do_fin    hello.ail hello.asa
do_m2a    hello.asa hello.s
```

Supponiamo di partire da un file "hello.suif" che costituisce l'output del front end fornito dall'ambiente SUIF. Questo file contiene una IR del programma originale.

- Il passo do_lower esegue il cosiddetto lowering sui dati in modo che le IR siano compatibili col sistema MachSuif.
- Il passo do_s2m crea un codice di tipo assembly generico.
- Il passo do_gen crea il codice specifico per il target indicato, in questo caso un'architettura alpha.
- Il passo do_il2cfg trasforma la IR da una instruction list (IL) ad un CFG (Control Flow Graph.)
- Il passo do_raga esegue l'allocazione dei registri.
- Il passo do_cfg2il inverte le IR da CFG a IL
- Il passo do_fin completa il processo di traduzione sostituendo i riferimenti (indirizzi) reali a quelli virtuali e inserendo le sequenze di ingresso e uscita nel codice della procedura.
- Il passo do_m2a traduce la rappresentazione MachSuif in un assembly ASCII specifico (un file .s)

Il framework è, inoltre, distribuito con una serie di passi di ottimizzazione di tipo machine independent.

3.2 *Control Flow Graph, Direct Acyclic Graph e definizione di FU*

Nell'applicazione dei processi di trasformazione e ottimizzazione operati dai compilatori, è spesso richiesta la rappresentazione del codice dell'applicazione mediante Control Flow Graph (CFG). Un CFG è un grafo i cui nodi rappresentano i basic block di una procedura (dove, per basic block, s'intende una lista di istruzioni che termina con una Control Transfer Instruction (CTI)) e i rami, il flusso di controllo tra gli stessi. Nel sistema di partizionamento proposto, sarà usata, oltre ai CFG, un'altra rappresentazione utile ad evidenziare le operazioni e le dipendenze tra i dati. Una simile rappresentazione è offerta dai DAG (Directed Acyclic Graph). I nodi di un DAG rappresentano i dati o le operazioni su di essi, mentre i rami indicano le varie dipendenze.
Per chiarire i concetti introdotti consideriamo un semplice programma C:

```
int main() {
  int a, b, c;
  c = 0;
  for (a = 0; a < 2; a++) {
    b = a * 2;
    c += a + b;
  }
  return c
}
```

L'assembly virtuale prodotto da MachSuif è del tipo:

```
main:
     ldc $vr0.s32 <- 0
     mov main.c <- $vr0.s32
     ldc $vr1.s32 <- 0
     mov main.a <- $vr1.s32
main._provaTmp1:
     ldc $vr2.s32 <- 2
     bge main.a, $vr2.s32, main._provaTmp0
     ldc $vr4.s32 <- 2
     mul $vr3.s32 <- main.a, $vr4.s32
```

```
    mov main.b <- $vr3.s32
    add $vr6.s32 <- main.a, main.b
    add $vr5.s32 <- main.c, $vr6.s32
    mov main._provaTmp3 <- $vr5.s32
    mov main.c <- main._provaTmp3
    mov main._provaTmp2 <- main.a
    ldc $vr8.s32 <- 1
    add $vr7.s32 <- main._provaTmp2, $vr8.s32
    mov main.a <- $vr7.s32
    jmp main._provaTmp1
main._provaTmp0:
    ret main.c
```

Si possono individuare quattro basic blocks. Il CFG corrispondente è mostrato in Figura 3.2.
A ciascuno di questi quattro blocchi base corrispondono altrettanti DAG. In Figura 3.3 è possibile osservare il DAG che si riferisce al basic block più esteso.
Come si può intuire, se si "accorpassero" più operazioni in una sola macroistruzione eseguita da un hardware dedicato, si potrebbero avere dei benefici in termini di prestazioni globali.
Chiameremo **FU** questi gruppi di operazioni logico-aritmetiche elementari, sottintendendo la loro propensione a poter essere implementate come unità funzionali specializzate. Nel DAG in Figura 3.3 un'unità funzionale possibile sarebbe quella data dalla moltiplicazione e le due somme. Ha quattro ingressi e due uscite. Da questo esempio si comprende come sia vantaggioso creare dei DAG – e quindi dei basic block – grandi il più possibile per raggruppare il numero maggiore di operazioni in una singola FU.
Per operare sui CFG sono stati usati dei passi forniti dall'ambiente MachSUIF mentre, per realizzare e analizzare la rappresentazione DAG, indispensabile per l'applicazione dell'algoritmo presentato nel paragrafo 3.5, è stato implementato un nuovo passo MachSUIF.

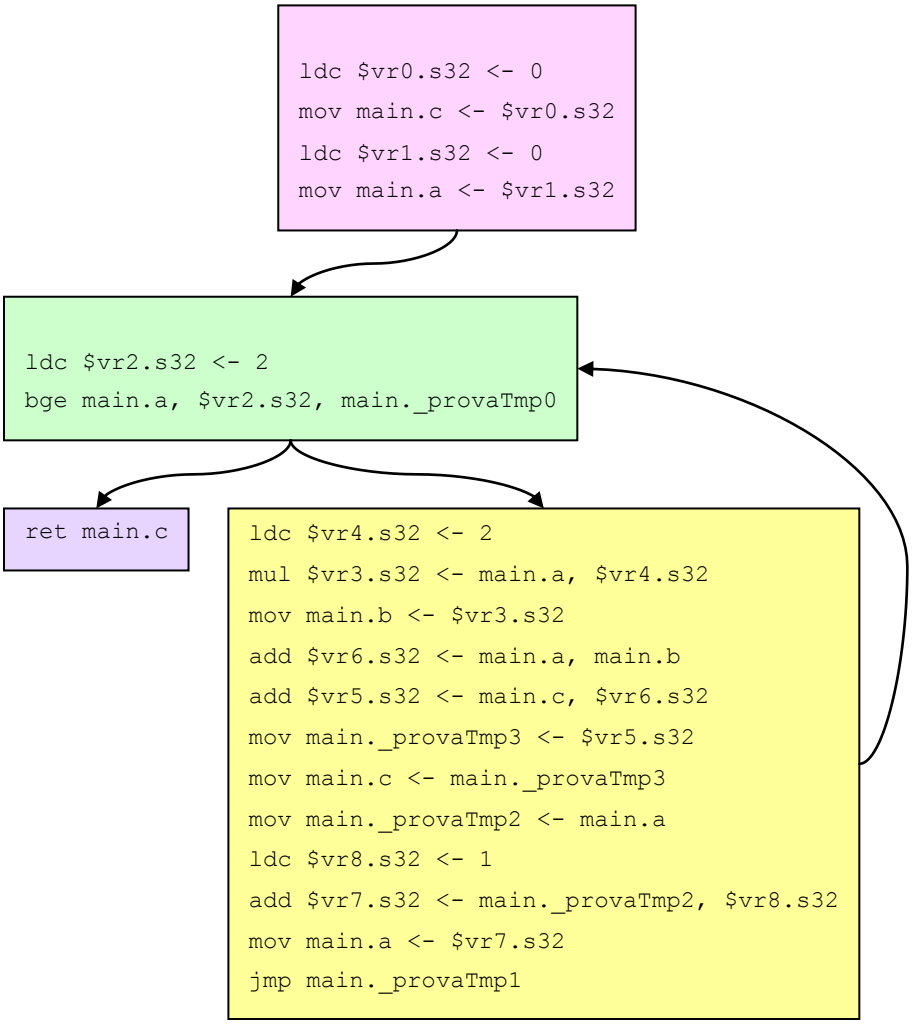


Figura 3.2: Rappresentazione con CFG

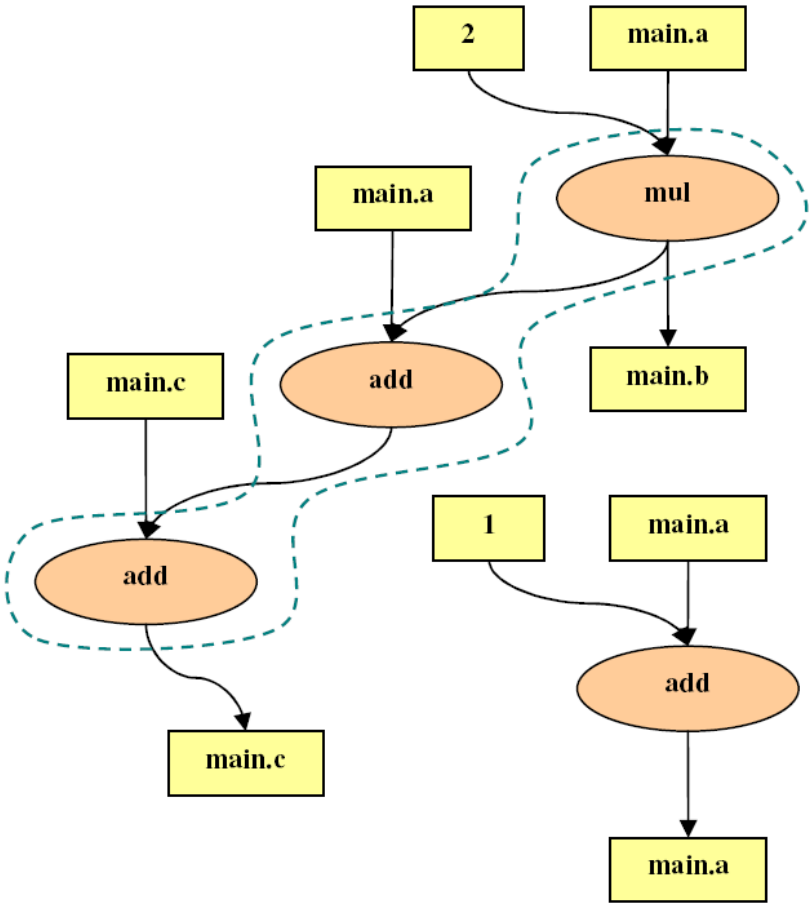


Figura 3.3: DAG

3.3 *Architettura Riconfigurabile di riferimento*

Nel concepire e realizzare il sistema proposto in questo capitolo, ci si è riferiti all'architettura riconfigurabile mostrata in Figura 3.4. Questa è composta da un microprocessore e da una FPGA. Come descritto in [33], è possibile scegliere differenti metodi di interfacciamento tra la parte riconfigurabile del sistema e la CPU ognuno dei quali presenta dei benefici e dei costi. In questo caso si è scelto di utilizzare un collegamento basato su bus e di mappare la FPGA nello spazio di indirizzamento della CPU. Il motivo di questa scelta risiede nel fatto che, pur dovendo fronteggiare un certo overhead per il trasferimento dei dati da elaborare verso la FPGA e il conseguente recupero dei risultati, questo tipo di accoppiamento permette di raggiungere un certo grado di parallelismo tra processore e coprocessore.

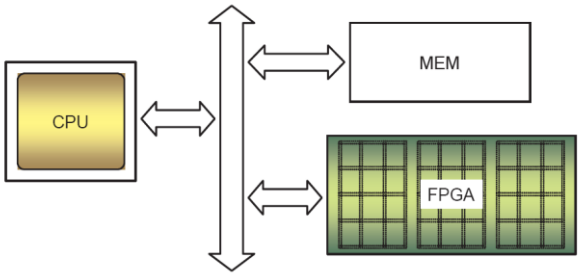


Figura 3.4: Architettura target di riferimento

La FPGA agisce come coprocessore riconfigurabile per eseguire le operazioni selezionate dal tool di partizionamento che ne genera automaticamente la configurazione. Uno schema a blocchi delle funzionalità implementate sulla FPGA è mostrato in Figura 3.5.

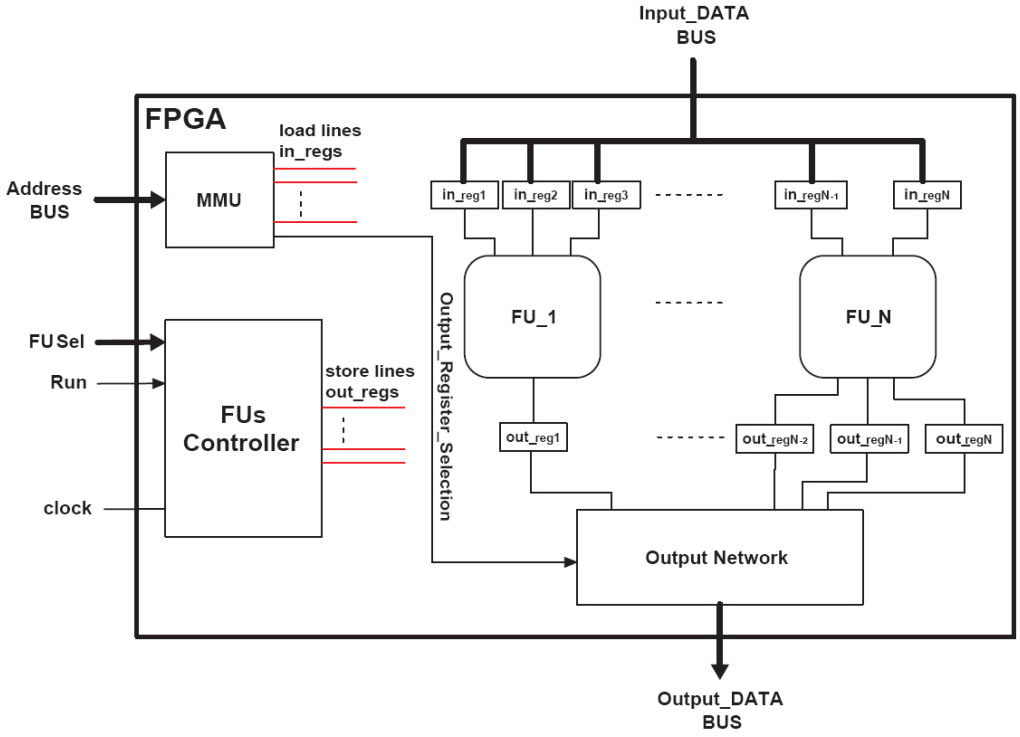


Figura 3.5: Struttura del coprocessore

E' possibile riconoscere i seguenti elementi:

1. *Memory Resources*: Sono costituite da due gruppi di registri mappati nello spazio di indirizzamento del processore. I valori memorizzati nei registri del primo gruppo sono usati come ingressi delle FUs mentre negli altri registri sono memorizzati i risultati delle computazioni.
2. *Memory Management Unit (MMU)*: La MMU decodifica l'indirizzo presente nell'address bus e genera i segnali per memorizzare correttamente i dati nei

registri d'ingresso. Allo stesso modo, nel caso di un'operazione di lettura eseguita dal processore, la MMU seleziona il registro di output desiderato attraverso la output network.

3. *Functional Units*: Rappresentano le implementazioni delle FU selezionate dal tool di partizionamento. Sono costituite da un datapath combinatorio ottenuto, come si vedrà nei capitoli successivi, dalla sintesi logica dei DAG selezionati.
4. *FUs Controller*: Questo modulo è usato per far si che il valore proveniente da una (o più) FU possa essere memorizzato nel relativo registro d'uscita. Nel controller è contenuta l'informazione sui ritardi di propagazione delle singole FUs. Una volta che una FU comincia la sua elaborazione, in base allo stato dei segnali *FUsel* e *Run*, il controller misura il tempo trascorso in modo da poter abilitare la memorizzazione dei registri d'uscita nel momento corretto.
5. *Output Network*: Questo modulo serve per instradare verso il bus il registro di uscita corretto in funzione del risultato è stato richiesto dalla CPU.

Pur considerando come riferimento l'architettura presentata in Figura 3.4, la maggior parte dei moduli funzionali che compongono il tool di partizionamento, sono riusabili anche con altre architetture.
Un esempio è il processore riconfigurabile eMIPS [1] che sarà analizzato nel prossimo capitolo. eMIPS è dotato di un insieme di applicazioni (BBTools) che non agiscono sul codice sorgente ma direttamente sull'eseguibile analizzandolo per identificare e selezionare i basic blocks da implementare poi nelle regioni riconfigurabili prescelte. I basic blocks selezionati diventano input per Mips-to-Verilog (M2V) [34], un compilatore che genera la loro implementazione Verilog.
Il sistema di partizionamento proposto in questo capitolo sarebbe utilizzabile, senza troppe modifiche, con eMIPS perché opera su un codice sorgente e risulta, quindi, sostanzialmente non dipendente dalla tecnologia.
Un test che potrebbe essere interessante realizzare, è quello di comparare le performances ottenute dai basic block selezionati a partire dalle analisi sul codice di alto livello invece che sull'eseguibile.

3.4 *Struttura del Framework*

In Figura 3.6 sono mostrati i vari stadi di cui è costituito il sistema di sviluppo proposto. L'input del sistema è costituito dal codice C del programma che si vuole mappare sul sistema microprocessore-FPGA proposto nel paragrafo 3.3.

Il codice sorgente è inizialmente sottoposto ad un'analisi lessicale e sintattica, al termine della quali viene prodotto un parse tree.
A questo tipo di rappresentazione intermedia (RI) possono essere applicate alcune trasformazioni di alto livello, come il loop unrolling e l'inlining. I passi successivi sono usati per muoversi dalla rappresentazione con parse tree ad un control flow graph i cui basic blocks contengono una sequenza lineare di pseudoistruzioni assembly.
İl processo di partizionamento hardware/software si basa su un algoritmo di ricerca e selezione che ha come ingresso una rappresentazione di tipo direct acyclic graph per singolo basic block. Prima di passare alla rappresentazione DAG viene eseguita una trasformazione di if-conversion che converte un costrutto if-then-else in due computazioni indipendenti al fine di accrescere le dimensioni dei DAG. Dato che il fine ultimo è quello di implementare in una FU il maggior numero di operazioni eseguite dal processore, avere dei DAG di dimensioni maggiori permette il raggiungimento di una migliore efficienza.
Una volta che i DAG sono creati per ogni singolo basic block, si può applicare l'algoritmo di partizionamento hardware/software che esegue le seguenti operazioni:

1. Analizza il DAG per ricerca tutte le possibili FU,
2. genera una la loro descrizione Verilog,
3. seleziona le FU che danno un incremento generale di prestazioni,
4. modifica il codice C originario inserendo delle chiamate di funzione al posto del codice la cui funzionalità è stata sostituita da una FU. La chiamata a funzione si farà carico di passare i dati in ingresso alla FU e a recuperare il risultato dell'elaborazione.

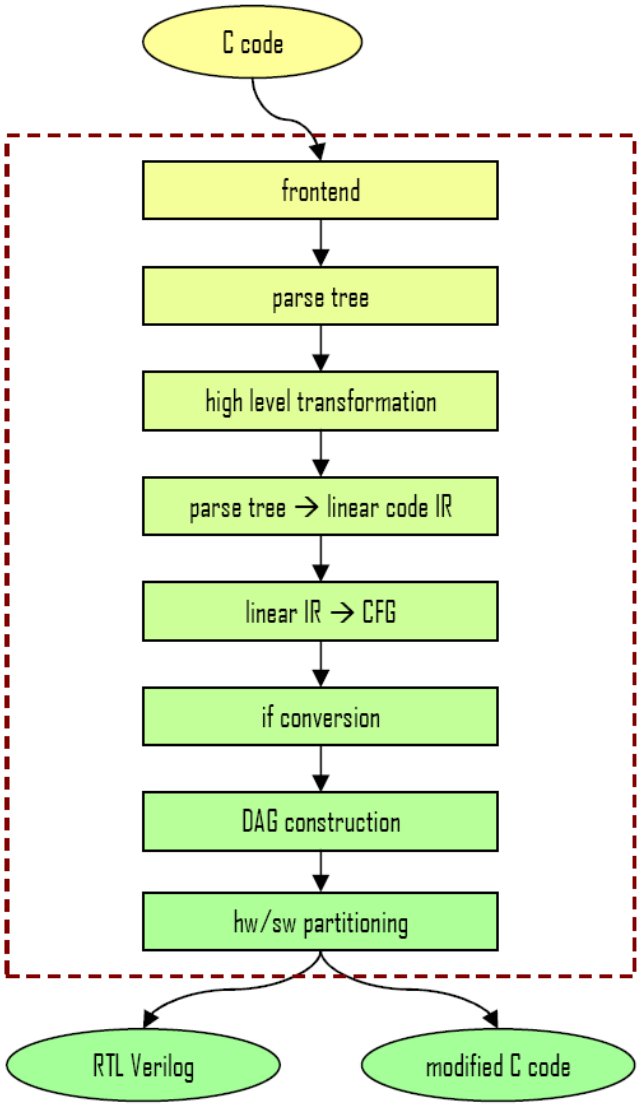


Figura 3.6: Overview del sistema

In Figura 3.7 è mostrata l'implementazione del sistema sottolineando quali parti sono state sviluppate con l'apporto di SUIF e quali con quello di MachSUIF.
SUIF è usato come front-end del compilatore e per eseguire un primo passo di ottimizzazione (il loop unrolling). Lo scopo di questa ottimizzazione è quello di scomporre i cicli for in liste di istruzioni senza il salto condizionato. In questo modo si creano le condizioni per ingrandire già in partenza le dimensioni del basic block che si riferisce al ciclo for. Tutte le successive ottimizzazioni sono implementate per l'ambiente MachSuif.

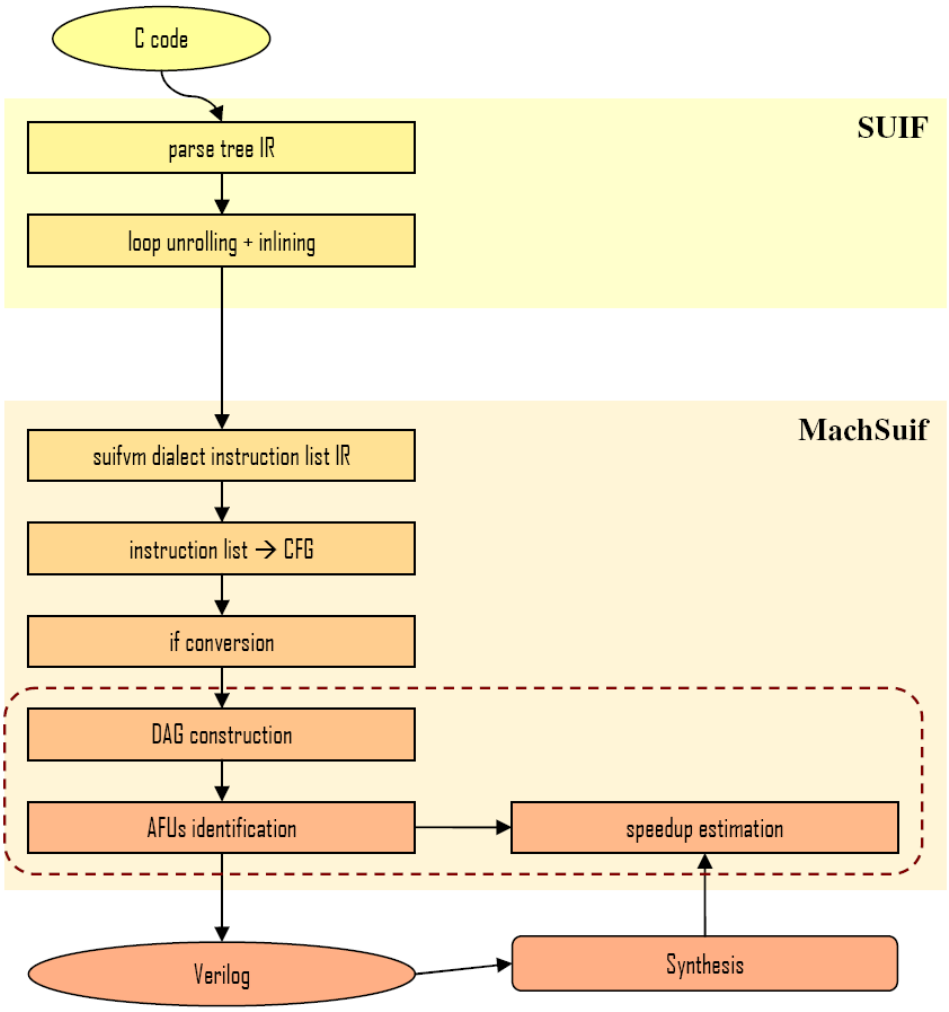


Figura 3.7: Implementazione del sistema

Per adattare la rappresentazione intermedia SUIF a quella usata da MachSUIF occorre un passo di analisi detto di lowering. Questo passo produce una IR formata da una lista di istruzioni assembly di una macchina virtuale, il SUIFvm. MachSUIF fornisce gli strumenti per passare alla rappresentazione CFG e per applicare la trasformazione di if-conversion. Con questa trasformazione la diramazione nel CFG dovuta ad un costrutto if-then-else, è sostituita da due gruppi di istruzioni, che eseguono le operazioni relative ai blocchi then ed else, seguiti da un'istruzione di selezione. La selezione è un'operazione predicata cui sono passati come argomenti i risultati delle computazioni dei due blocchi then ed else e la condizione da verificare. A seconda del valore assunto dal risultato del test, l'istruzione di selezione restituirà il primo o il secondo input. In questo modo si eliminano le operazioni di salto necessarie ad implementare un costrutto if-then-else in assembly, e si aumentano le

dimensioni del basic block. Ricordando che un basic block termina sempre con un'istruzione per il trasferimento del controllo, è evidente che la sua eliminazione permette di unire più basic block ottenendone uno più esteso.
Fino a questo punto tutte le trasformazioni applicate sono state prese tra quelle messe a disposizione dai due tools.
Per arrivare al partizionamento hardware/software è necessario compiere delle altre operazioni. Queste operazioni, indicate dalla linea tratteggiata in Figura 3.7, sono state inserite in un passo MachSUIF la cui implementazione sarà descritta più avanti.
Per prima cosa si cambia rappresentazione interna dei nodi del CFG passando dal codice lineare ai DAG che diventano ingressi per dell'algoritmo di identificazione delle FU. I DAG vengono analizzati e, per i sottografi che soddisfano dei parametri definibili dall'utente, viene generata una descrizione Verilog. Questi sottografi costituiscono le potenziali FU da implementare in hardware (nel paragrafo 3.5 questa parte sarà analizzata nel dettaglio). È quindi eseguita la stima della velocità d'esecuzione delle operazioni che costituiscono i tagli individuati. Questa è stimata, nel caso di esecuzione tradizionale sul processore, sulla base del numero di cicli necessari all'esecuzione, mentre, nel caso dell'esecuzione su FU, è determinata dai risultati della sintesi logica del codice Verilog precedentemente generato. Se dal confronto dei due valori risulta che l'operazione mappata in hardware introduce uno speedup significativo, la FU viene selezionata per l'implementazione su FPGA.

3.5 *Algoritmo di partizionamento*

In questo paragrafo è descritto nel dettaglio l'algoritmo che è stato usato per l'identificazione e la selezione delle FU estraibili da un DAG. L'algoritmo è basato su quello proposto in [27] ma è stata modificata la valutazione della velocità di esecuzione di una FU in hardware. Questo parametro, infatti, è molto importante perché condiziona la scelta delle FU da implementare sul coprocessore e, quindi, anche le prestazioni globali. Poiché solo un loro sottoinsieme è in grado di fornire un incremento di prestazioni e dato che l'area nella FPGA è limitata, è necessario selezionare solo quelle più efficienti. Nel paragrafo 3.5.3 sarà descritto nel dettaglio il criterio di selezione usato.

3.5.1 **Individuazione delle FU**

Una FU può essere classificata rispetto al numero dei suoi input, dei suoi output, della complessità della funzione che implementa e della sua capacità di accedere alla memoria. Possiamo individuare i seguenti tipi di FU:

- Molti Input Singolo Output (MISO)
- Molti Input Molti Output (MIMO)
- MISO/MIMO con unità di load/store
- MISO/MIMO con loops o esecuzione predicata

Il lavoro presentato in [27] affronta il problema dell'estensione di un ISA considerando dei vincoli microarchitetturali imposti dal progettista e delle ipotesi semplificative sulle FU cercate. Queste ultime impongono che le FU addizionali non debbano avere accesso alla memoria (perciò le istruzioni di load/store sono ignorate nella ricerca) e mappano solo operazioni sui dati. In sostanza sono considerate solo FU di tipo MIMO semplice.
Vediamo in che modo funziona l'algoritmo per l'identificazione delle FU.
Chiamiamo *G(V, E)* i DAG che rappresentano i vari basic block; i nodi *V* rappresentano operazioni elementari e i rami *E* rappresentano le dipendenze tra i dati. Ad ogni grafo *G* ne è associato un altro $G^+(V \cup V^+, E \cup E^+)$ che contiene i nodi e rami addizionali V^+ ed E^+. I nodi addizionali rappresentano le variabili d'ingresso e di uscita del basic block. I rami addizionali connettono i nuovi nodi con quelli del grafo *G* di partenza.
Un ***taglio*** *S* è un sottografo di *G*. Ci sono $2^{|V|}$ possibili tagli, dove $|V|$ è il numero di nodi presenti in *G*.
Una funzione arbitraria ***M(S)*** misura il merito di un taglio *S* e rappresenta una stima dello speedup ottenibile implementando *S* come una FU.
Chiamiamo ora *IN(S)* il numero di nodi predecessori dei rami che entrano nel taglio *S* dal resto del grafo G^+. Esso rappresenta il numero di input usati dalle operazioni in *S*. Allo stesso modo *OUT(S)* è il numero di nodi in *S* che sono predecessori dei rami uscenti dal taglio. Esso rappresenta il numero di valori prodotti da *S* e utilizzati da altre operazioni, che siano in *G* o in altri basic block. Infine, definiamo il taglio *S* ***convesso*** se non esistono percorsi da un nodo $u \in S$ ad un altro nodo $v \in S$ che includono qualche nodo $w \notin S$. In Figura 3.8 è rappresentato un esempio di taglio non convesso.

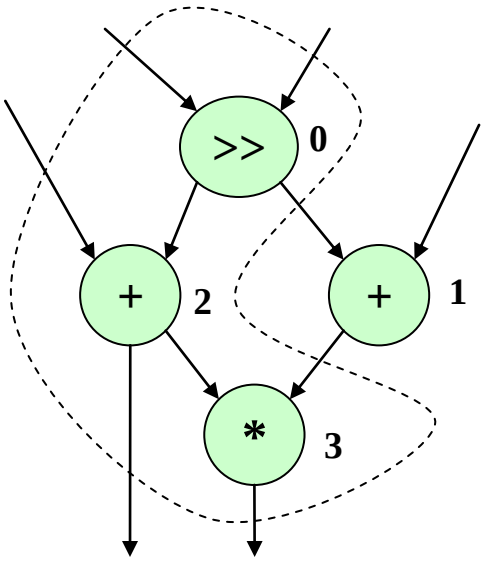


Figura 3.8: Taglio non convesso

Considerando separatamente ciascun basic block il problema di identificazione può essere formalmente enunciato come segue:

PROBLEMA 1: Dato un grafo G^+, trovare il taglio S che massimizza M(S) sotto le seguenti condizioni:

1. $IN(S) \leq N_{in}$
2. $OUT(S) \leq N_{out}$
3. *S è convesso*

I valori N_{in} e N_{out} sono definiti dall'utilizzatore e indicano il numero di porte di lettura e scrittura, rispettivamente, che possono essere utilizzati dall'unità funzionale. Il vincolo della convessità è un controllo di legalità sul taglio *S* necessario per assicurare che sia possibile implementare l'operazione in hardware. Come si vede nella Figura 3.8, se tutti gli input di un'istruzione devono essere disponibili al momento dell'esecuzione della stessa e tutti gli output vengono prodotti alla fine dell'esecuzione, non esisterebbe alcun ordinamento in grado di rispettare le dipendenze di questo grafo una volta che *S* fosse mappato come singola istruzione. Poiché da ciascun DAG in genere vengono estratti molti gruppi di istruzioni, è necessario trovare un massimo di N_{instr} tagli che, insieme, offrono il massimo

vantaggio. Questo problema, detto di selezione, viene in genere risolto risolvendo ripetutamente il Problema 1 su tutti i basic block e semplicemente scegliendo i N_{instr} migliori gruppi di istruzioni. Enunciato formalmente, il problema che vogliamo risolvere è il seguente:

PROBLEMA 2: Dati i grafi G_i^+ di tutti i blocchi base, trovare un massimo di N_{instr} tagli S_j che massimizzino $\Sigma_j M(S_j)$ sotto le stesse condizioni del Problema 1 per ogni taglio S_j.

Analizziamo nelle successive sottosezioni i due problemi sopraesposti.

3.5.2 **L'algoritmo di identificazione**

Considerare tutti i possibili tagli di un basic block è computazionalmente molto oneroso, e talvolta addirittura impraticabile. Per questo motivo l'algoritmo proposto in [27], esplora per intero lo spazio di ricerca, ma elimina nel corso della stessa le regioni che non soddisfano le condizioni viste. I nodi del grafo *G* devono essere ordinati in maniera tale che se *G* contiene un ramo *(u, v)* allora il nodo *u* deve comparire prima di *v* nella numerazione. L'algoritmo usa una funzione di ricerca ricorsiva basata su questo ordinamento per esplorare il DAG. In particolare, è utilizzato un albero binario che rappresenta i possibili tagli. Quest'albero è costituito da un nodo radice che rappresenta un taglio vuoto a cui seguono dei rami marcati con 1 o 0. Ogni coppia di rami 1 e 0 al livello *i* rappresenta rispettivamente l'aggiunta, o l'eliminazione, del nodo di *G* con ordine topologico *i*, al taglio costituito dal nodo predecessore. I nodi dell'albero di ricerca immediatamente successivi ad un ramo 0 rappresentano lo stesso taglio visto al nodo predecessore, e possono essere quindi ignorati nella ricerca. La Figura 3.9 mostra l'albero di ricerca per l'esempio di Figura 3.8, con alcuni nodi etichettati col valore del loro taglio.

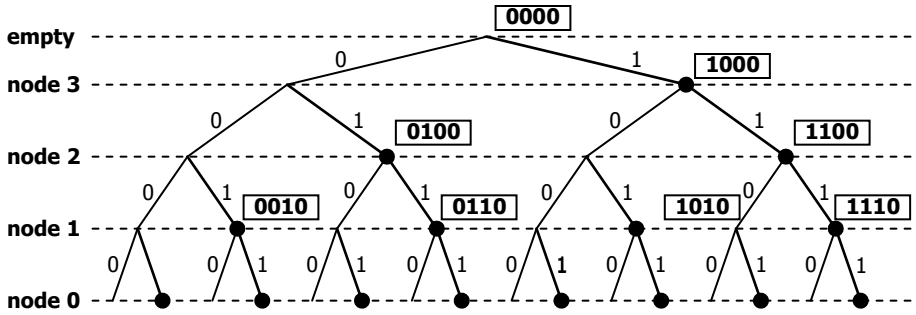


Figura 3.9: Albero di Ricerca

La ricerca prosegue secondo uno schema *preorder* trasversale. In certi casi non c'è bisogno di proseguire la ricerca verso i livelli più bassi, quindi lo spazio di ricerca viene sfrondato dell'intero sottoalbero che ha origine da quel nodo. Si supponga per esempio che il vincolo sul numero di output sia stato violato dal taglio definito da un certo nodo dell'albero. Aggiungendo nodi che compaiono successivamente nell'ordinamento topologico non si può in nessun modo ridurre il numero di output, che possono solo aumentare. Allo stesso modo, se il vincolo della convessità è stato violato ad un certo nodo dell'albero, non c'è modo di rendere utilizzabile il taglio inserendo nodi di *G* che compaiono successivamente nell'ordinamento topologico.
Si consideri per esempio la Figura 3.9 dopo l'inclusione del nodo 0; gli unici modi per riguadagnare la convessità sono quelli di includere il nodo 1 o rimuovere i nodi 0 o 3. A causa dell'ordinamento topologico, entrambe le soluzioni sono impraticabili. Di conseguenza, se raggiungendo un certo nodo dell'albero di ricerca, un vincolo viene violato, il sottoalbero che ha per radice quel nodo viene completamente.
Consideriamo, ad esempio, il DAG raffigurato in Figura 3.8 e poniamo N_{out}=1. In Figura 3.10 è riportato l'albero di ricerca della Figura 3.9, col nuovo vincolo.

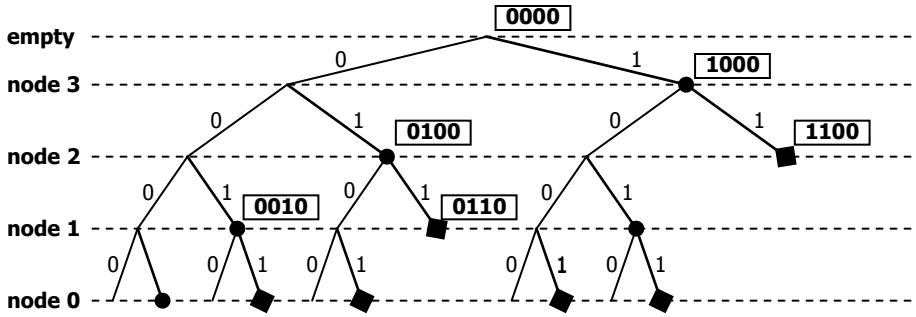


Figura 3.10

I rami che terminano con un rombo rappresentano un taglio non valido. Si può osservare che l'algoritmo, dopo l'inclusione del nodo 3 e del nodo 2, riconosce una violazione del vincolo sul numero massimo di uscite. A quel punto segna il taglio non valido e torna indietro. Solo cinque tagli rispettano tutte le condizioni, mentre sei violano uno dei vincoli, portando all'esclusione di altri quattro tagli che non vengono neppure considerati.
Una volta individuati con questa procedura tutti i tagli che soddisfano tutti i vincoli imposti, occorre un criterio per selezionare i migliori candidati, ovvero quell'insieme limitato di tagli la cui implementazioni come FU massimizza le prestazioni globali del sistema.

3.5.3 **Criterio di selezione dei tagli**

Applicando l'algoritmo di identificazione a un DAG molto grande possono essere individuate diverse centinaia di tagli idonei ad essere trasformati in FU hardware. È evidente che non tutti i tagli possono essere presi in considerazione; esiste un limite fisico al numero di unità funzionali programmabili sulla FPGA. Non solo, potrebbe addirittura capitare che l'operazione software originale eseguita dal processore si riveli più vantaggiosa di quella eseguita dalla relativa FU. Questo, in genere, avviene solo per tagli molto piccoli. Al crescere delle dimensioni del taglio e, quindi, del numero di operazioni, le prestazioni di una FU superano notevolmente quelle della CPU. Ci si trova allora a dover scegliere, tra tutti i tagli, quelli che porterebbero il massimo vantaggio al sistema se fossero implementati sulla FPGA.
La selezione è basata sul calcolo dello speedup, ossia l'incremento di velocità ottenibile implementando le operazioni appartenenti ad un taglio con una FU.
Lo speedup è definito come:

$$Speedup = f(i)(L_{sw}(i) - L_{hw}(i))$$

Dove:

- *f(i)* rappresenta la frequenza di esecuzione del basick block che contiene il DAG a cui appartiene il sottografo i-simo.
- $L_{sw}(i)$ rappresenta la latenza software ossia, il tempo impiegato dalla CPU ad eseguire le operazioni appartenenti al grafo.
- $L_{hw}(i)$ rappresenta la latenza hardware ovvero, il tempo impiegato da una FU per eseguire le operazioni appartenenti al grafo.

Il criterio che si utilizza per il calcolo dei singoli termini è, quindi, fondamentale per avere una selezione ottimale.
In [27] la valutazione dello speedup è calcolata nel seguente modo:

- La frequenza dei singoli basic block è determinata da un profiling statico dell'applicazione.
- Considerato l'ISA del processore di riferimento, la latenza software è determinata considerando il numero di cicli necessario per eseguire ogni operazione appartenente al taglio. Questo valore è stato poi diviso per la frequenza di funzionamento del processore per ottenere un valore espresso in ns.
- Per ogni operazione logico-artimetica eseguibile dall'ISA di riferimento, è stato sintetizzato un datapath in tecnologia CMOS 180um annotando il valore del ritardo del percorso critico (in ns).
- La latenza hardware della FU che implementa il taglio i-simo, è data dalla somma dei ritardi delle singole operazioni appartenenti al percorso ingresso-uscita più lungo.

Tralasciando una discussione sul calcolo del termine *f(i)*, che anche nel presente lavoro è dato dal risultato di un profiling statico, risulta evidente che le stime degli altri due termini sono entrambe approssimative. In software si suppone ottimisticamente che il processore esegua l'operazione senza mai un'interruzione di sistema o uno stallo. In hardware, si è trascurato che una sintesi logica non funziona sintetizzando i singoli operandi ma considerando la funzione logica ingresso-uscita nella sua interezza e, di conseguenza, utilizzando delle ottimizzazioni con uno scope più elevato.
Considerando che risulta molto difficile stimare con esattezza il termine $L_{sw}(i)$, essendo soggetto ad eventi asincroni e non deterministici come gli interrupt, si è concentrata l'attenzione sul calcolo del termine $L_{hw}(i)$.

Per aumentare la precisione, si è esteso l'algoritmo di partizionamento con un modulo per generare automaticamente il codice Verilog che descrivesse i tagli individuati. Si è quindi proceduto alla loro sintesi logica dal cui timing report è stato estratto il termine $L_{hw}(i)$, non più stimato ma reale.
Una volta determinate lo speedup per ogni taglio, si può procedere alla selezione di quelli che possiedono i valori maggiori e implementare le relative FU compatibilmente con l'area messa a disposizione dalla FPGA considerata.

3.5.4 Implementazione dell'algoritmo

L'algoritmo è stato implementato in C++ come passo di ottimizzazione del sistema MachSUIF.
In particolare è stato necessario implementare le classi e i metodi per la creazione dei DAG, per l'individuazione e la selezione automatica delle FU e per la generazione automatica del Verilog dei tagli.

3.5.4.1 DAG in MachSuif

La distribuzione standard di MachSuif non mette a disposizione una libreria che consenta di rappresentare il contenuto dei basic blocks come DAG. Il primo passo che si è affrontato è stato quindi quello di scrivere delle classi che permettessero di manipolare questa rappresentazione. La classe rappresentata in Figura 3.11 è usata per descrivere e manipolare i nodi del DAG.
La prima proprietà necessaria per un nodo di un grafo è il numero identificativo del suo ordinamento topologico. Il membro ***index*** serve a questo scopo. Altro aspetto fondamentale è la capacità di poter agevolmente scorrere il grafo passando da un nodo ai suoi predecessori e successori. Per far ciò utilizziamo le variabili ***preds_size*** e ***succs_size***, che rappresentano il numero di predecessori e successori di un nodo rispettivamente, e i puntatori ***pred*** e ***succ***, che possono essere visti come dei vettori contenenti gli indirizzi di tutti i nodi predecessori e successori. Abbiamo detto che i nodi di un DAG possono essere dati o operazioni sui dati: per distinguere questi due casi usiamo la variabile booleana ***is_opnd***. Se ***is_opnd*** è vera, il nodo rappresenta un dato, se è falsa rappresenta un'operazione. Quando si sta trattando un dato è spesso necessario sapere se è un operando di una data istruzione, o se è piuttosto la variabile nella quale viene memorizzato un risultato, ovvero se è un dato sorgente o destinazione. Il membro booleano ***is_src*** indica un dato sorgente se è vero e un dato destinazione se è falso. Gli altri attributi della classe sono tre tipi di dato dell'ambiente MachSUIF: ***opnd*** è il dato vero e proprio, su cui si può agire coi metodi delle varie classi MachSUIF, ***instr*** è un puntatore all'istruzione, nel caso in

cui il nodo rappresenti un'operazione, e ***type*** rappresenta il tipo di dato e la sua dimensione (es. intero a 32 bit con segno).

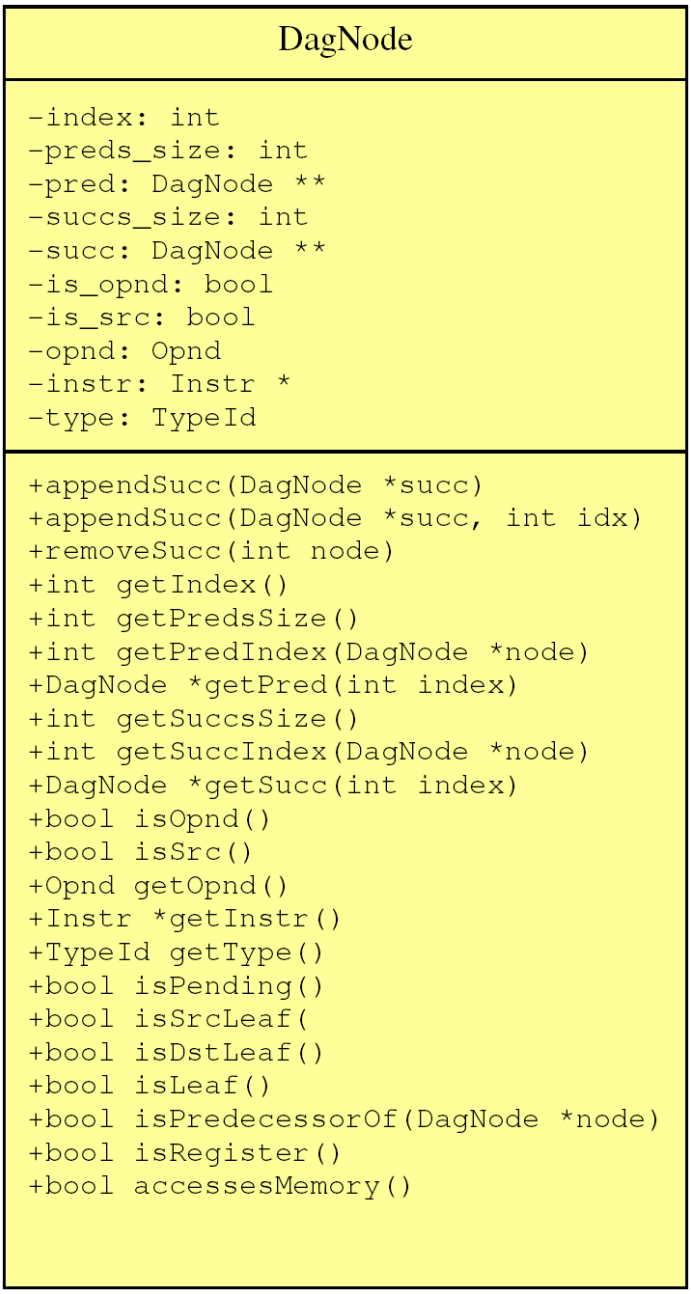


Figura 3.11: Classe DagNode

Passiamo ad analizzare i metodi della classe. Prima esigenza nella costruzione di un grafo qualunque è quella di aggiungere e rimuovere nodi. I metodi ***appendSucc()*** e ***removeSucc()*** servono a questo scopo. Per usare il metodo ***appendSucc()*** è necessario creare ed inizializzare correttamente una nuova istanza di DagNode e

passarla come argomento al metodo. In alternativa si può passare un qualunque nodo già esistente. Esistono due prototipi del metodo. Il primo ha, nella lista degli argomenti, solo il puntatore al DagNode col quale si vuole stabilire la nuova connessione. Il secondo ha anche un intero che rappresenta la posizione in cui si vuole che il nodo su cui chiamiamo il metodo, venga memorizzata tra i predecessori del nodo passato come argomento. Chiariamo con un esempio. Consideriamo i due nodi *A* e *B*, con i predecessori e successori indicati in Figura 3.12(a).

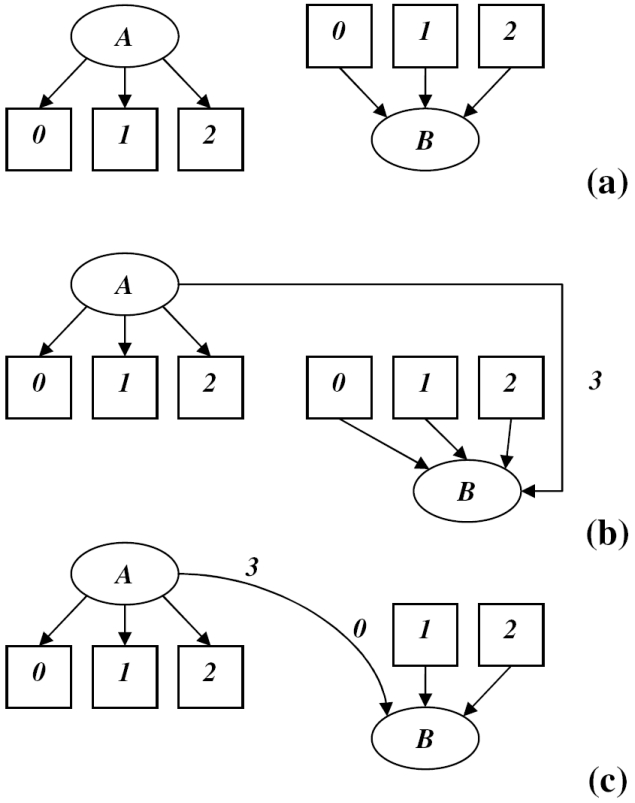


Figura 3.12: (a) Due DagNode isolati. (b) L'unione dei due nodi col metodo *A.appendSucc(&B)*. (c) L'unione dei due nodi col metodo *A.appendSucc(&B, 0)*.

Eseguendo la chiamata `A.appendSucc(&B)`, il riferimento al nodo ***B*** verrà inserito in coda al vettore di successori (***succ***) di ***A*** (Figura 3.12(b)). Questo approccio è comodo se non si è interessati all'ordine in cui i predecessori e i successori vengono elencati, perché il metodo si preoccupa di inizializzare o modificare le dimensioni dei vettori ***succ*** e ***pred*** accodando i nuovi elementi a quelli eventualmente presenti. Spesso, però, può capitare di dover preservare un ordine nella sequenza dei link. Ad esempio, se dobbiamo costruire un DAG che rappresenti un'operazione non commutativa come la sottrazione, è importante che l'ordine degli operandi non venga

invertito. In questo caso bisogna utilizzare `A.appendSucc(&B, 0)` in cui viene specificata la posizione che si vuole far occupare al link (Figura 3.12(c)). Si presti attenzione al fatto che in questo caso non viene creata una posizione nuova nel vettore dei predecessori di ***B***, e quindi è necessario che questa già esista. Se si tenta di chiamare il metodo per l'esempio di Figura 3.12 come `A.appendSucc(&B, 3)`, verrà prodotto un messaggio d'errore per indicare l'incompatibilità della chiamata col numero di predecessori del nodo ***B***.

Per rimuovere un link tra un nodo e un suo successore è invece sufficiente chiamare il metodo ***removeSucc()*** passandogli l'indice del successore. Attenzione a non confondere questo indice – che identifica la posizione in un vettore – con la proprietà ***index***, che invece rappresenta il numero del nodo nell'ordinamento topologico del DAG e che si può recuperare col metodo ***getIndex()***. Sempre con riferimento alla Figura 3.12, la chiamata `A.removeSucc(3)` rimuove il nodo ***B*** dalla lista dei successori di ***A***, e il nodo ***A*** dalla lista dei predecessori di ***B***. ***getPredsSize()*** e ***getSuccsSize()*** restituiscono rispettivamente il numero di predecessori e di successori. Se si vuole conoscere la posizione di un nodo X nel vettore dei predecessori o dei successori di un altro nodo Y è sufficiente usare i metodi ***getPredIndex()*** e ***getSuccIndex()*** passandogli come argomento un puntatore al nodo X. Se X non è predecessore o successore di Y il metodo restituisce -1. Ad esempio, la chiamata `A.getSuccIndex(&B)` restituisce 3, mentre `B.getPredIndex(&A)` restituisce 3 nel caso di Figura 3.12(b) e 0 nel caso di Figura 3.12(c). A.getPredIndex(&B) e B.getSuccIndex(&A) restituiscono -1. Per ottenere un puntatore ad un predecessore di un nodo bisogna usare il metodo ***getPred()*** passando come argomento la posizione del predecessore nel rispettivo vettore. Stesso discorso per i successori usando il metodo ***getSucc()***. Per esempio, `A.getSucc(3)` restituisce un puntatore al nodo ***B*** e `B.getPred(0)` (Figura 3.12(c)) restituisce un puntatore al nodo ***A***. I metodi ***isOpnd()***, ***isSrc()***, ***getOpnd()***, ***getInstr()*** e ***getType()*** restituiscono i dati membro ***is_opnd***, ***is_src***, ***opnd***, ***instr***, e ***type***.

Quando si applicano al DAG degli algoritmi di ottimizzazione, vengono individuati molti nodi da eliminare. Per rimuovere questi nodi dal DAG il primo passo consiste nell'eliminare tutti i link col resto del grafo, lasciandoli isolati (*pending*). Chiamando il metodo ***isPending()*** su un nodo, si può sapere se questo è privo di connessioni (tutti i nodi isolati possono essere rimossi da un'istanza della classe Dag con la funzione ***removePendingNodes()***).

Altra necessità nell'analisi di un grafo è quella di sapere se il nodo che si sta visitando è una foglia e, in particolare, se è una foglia sorgente o destinazione. Per questo scopo sono stati implementati i metodi ***isLeaf()***, ***isSrcLeaf()*** e ***isDstLeaf()*** rispettivamente.

Se si vuole sapere se il nodo in esame è predecessore di un altro nodo si usa il metodo ***isPredecessorOf()***.
Sempre riferendoci alla Figura 3.12(c)., `A.isPredecessorOf(&B)` è *true*, mentre `B.isPredecessorOf(&A)` è *false*. Per finire, se si stanno esaminando dei nodi di tipo dato si può sapere se il dato è un registro piuttosto che una variabile usando il metodo ***isRegister()***, e se accede alla memoria col metodo ***accessesMemory()***.
Un DAG potrebbe essere visto semplicemente come un vettore di DagNode, ma ci sono alcune proprietà e operazioni che è immediato considerare proprie di un oggetto DAG. Risulta conveniente creare una classe apposita la cui struttura è mostrata in Figura 3.13.

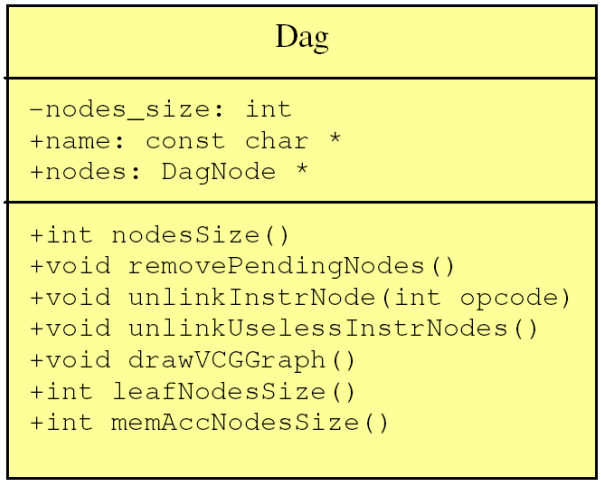


Figura 3.13: Classe Dag

I dati membro della classe sono tre: ***nodes_size,*** che rappresenta il numero di nodi componenti il DAG, ***name*** che è una stringa formata dal nome della procedura da cui dipende il DAG più un numero che identifica il basic block, e ***nodes*** che è il vettore di DagNode che rappresenta il DAG vero e proprio.
I metodi ***nodesSize()***, ***leafNodesSize()*** e ***memAccNodesSize()*** restituiscono rispettivamente il numero di nodi presenti nel DAG, il numero di nodi foglia e il numero di nodi che accedono alla memoria. ***drawVCGGraph()*** è una utility per creare un file VCG per la rappresentazione grafica del DAG. Molti tools commerciali e freeware sono in grado di leggere questo formato di grafo. I restanti metodi sono delle ottimizzazioni sul DAG. ***removePendingNodes()***, come si è accennato, consente di rimuovere dal grafo tutti i nodi precedentemente isolati con le procedure di *merging* di registri e variabili, oppure con i metodi ***unlinkInstrNode()*** e ***unlinkUselessInstrNodes()***. ***unlinkInstrNode()*** elimina tutte le connessioni dei nodi istruzione il cui opcode viene passato come argomento al metodo, trasformandoli in nodi isolati. Il metodo consente di isolare tutti i nodi istruzione a un solo operando,

ma l'effettiva utilità della procedura è limitata a poche operazioni che non hanno significato nella ricostruzione del dataflow di un basic block. Per esempio l'istruzione MOV è indispensabile per spostare il risultato di un'operazione su un registro, ma sicuramente è inutile nella rappresentazione grafica dell'operazione. Anche l'istruzione NOT ha un solo operando, il dato su cui calcolare il complemento, ma eliminando questa operazione dal DAG non si preserverebbe il calcolo originale. Chiariamo con un esempio. Consideriamo questo semplicissimo calcolo in C:

```
a =  !(a >> 16);
```

In una notazione pseudo assembly avrebbe questa forma:

```
lsr $vr1, a, 16
not $vr2, $vr1
mov a, $vr2
```

a cui corrisponderebbe il DAG elementare in Figura 3.14(a).

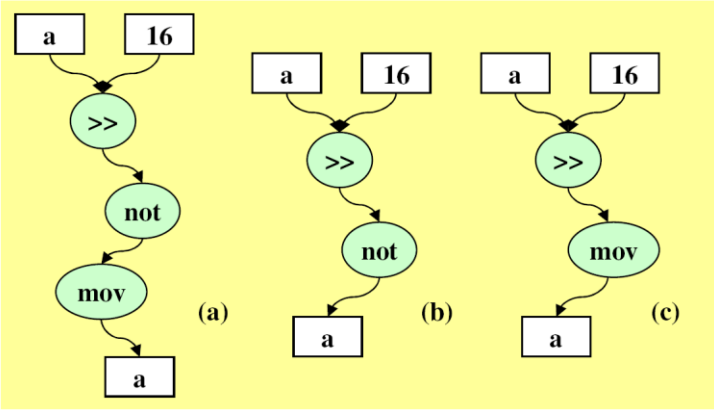


Figura 3.14

Come si può notare, il nodo del DAG corrispondente all'istruzione MOV è del tutto inutile ai fini del calcolo dell'operazione. Ha senso, quindi, applicare il metodo ***unlinkInstrNode(24)***, dove 24 è l'opcode corrispondente all'istruzione MOV per il dialetto SUIFVM di MachSUIF [29], per ottenere il DAG di Figura 3.14(b), più adatto ai nostri scopi. Al contrario, non ha senso chiamare il metodo sulle istruzioni NOT. L'operazione è consentita, perché l'unico controllo effettuato è sul numero di

operandi e non sulla preservazione del significato del calcolo originale, ma il DAG prodotto (Figura 3.14(c)) non rappresenta l'operazione descritta dal codice C di partenza. Sta all'utilizzatore stabilire i casi in cui è corretto e vantaggioso eliminare i nodi di un certo tipo di istruzione.
Alcuni algoritmi di ottimizzazione, come l'if-conversion, possono introdurre un certo numero di operazioni inutili, ovvero operazioni il cui risultato viene memorizzato su variabili o registri virtuali. Queste sono locazioni di memoria usate dal compilatore per scrivere temporaneamente i dati, che verranno successivamente spostati nelle variabili previste dal codice originale. È evidente che se un dato viene memorizzato su una simile locazione di memoria e mai spostato in una variabile reale, l'operazione è inutile e lo sono anche tutte le istruzioni che concorrono al calcolo. ***unlinkUselessInstrNodes()*** elimina tutte le connessioni verso i nodi istruzione il cui risultato viene messo su variabili o registri temporanei o comunque non presenti nel codice originale. Vengono anche rimossi i collegamenti del grafo verso tutti i nodi che contribuiscono al calcolo dello stesso risultato e non hanno altre connessioni.
I metodi ***unlinkInstrNode()*** e ***unlinkUselessInstrNodes()*** non eliminano i nodi, ma li isolano soltanto. Quindi, ogni volta che si usa uno dei due metodi, è necessario usare il metodo ***removePendingNodes()*** per eliminare effettivamente i nodi dal DAG.
Diamo adesso una sommaria descrizione del processo di costruzione del DAG a partire dal basic block.

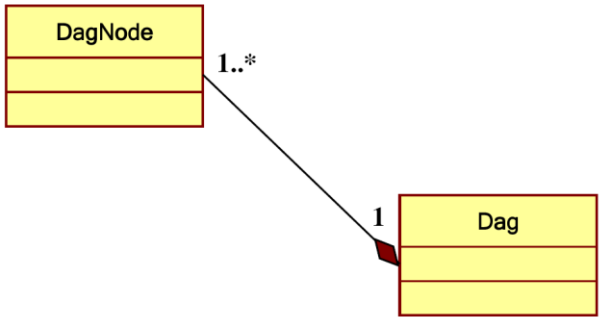


Figura 3.15: Class Diagram per un DAG

Quando si crea un'istanza della classe Dag tramite l'operatore **new** viene chiamato il costruttore, a cui si passa il basic block e il nome del DAG col formato già descritto.

*Dag *dag = new Dag(CfgNode *basic_block, char *dag_name);*

Come si può vedere, il basic block è un nodo del CFG. CfgNode è una classe MachSUIF che implementa e permette di modificare la rappresentazione CFG. Il nome del DAG verrà utilizzato nella creazione dei vari file VCG e Verilog.
Per costruire il DAG a partire dalla lista di istruzioni contenute nel CfgNode, si procede come segue:

1. Si esaminano tutte le istruzioni del basic block e si eliminano le *nop*, le *label* e le *CTI*.
2. Si stima il numero iniziale di nodi contando tutti gli operandi e le operazioni presenti nel basic block.
3. Si individuano le istruzioni di load e store, che verranno escluse dallo spazio di ricerca dell'algoritmo di identificazione delle AFU.
4. Si esegue un'analisi di tipo reaching definitions / liveness per collegare tra loro le istruzioni che leggono o scrivono sugli stessi registri o variabili, e ricostruire il dataflow del codice originale.
5. Si elimina il link dai nodi con le istruzioni MOV, CVT, LDC che sono inutili per ricostruire le dipendenze sui dati.
6. Si eliminano eventuali altri nodi inutili.
7. Si rimuovono i nodi isolati.

3.5.4.2 Implementazione dell'algoritmo di identificazione

Le classi create per l'implementazione dell'algoritmo di ricerca sono due: la classe **Cut** e la classe **PathFinder**. La classe Cut contiene tutti i membri e i metodi che caratterizzano un taglio, oltre alla funzione di ricerca ricorsiva già descritta. Per verificare la convessità dei tagli è necessario poter individuare tutti i percorsi che legano i vari nodi. A questo scopo è stata ideata una classe apposita, PathFinder, che contenesse dei membri rappresentanti i vari percorsi tra i nodi del DAG, e dei metodi per individuarli.
Consideriamo per prima la classe **Cut**. Le FU rappresentano il corrispettivo hardware di una operazione software complessa, ovvero una serie di operazioni elementari dell'ISA che vengono eseguite in diversi cicli di clock. Abbiamo visto come la IR di tipo DAG consenta di visualizzare questo flusso di operazioni sui dati, e di individuare agevolmente i cluster di operazioni che possono costituire l'AFU. Topologicamente definiamo questo cluster di operazioni un *taglio* del DAG, ovvero un suo sottografo. È sembrato vantaggioso implementare una descrizione indipendente di questi *tagli* per poter incapsulare nella struttura di una classe le operazioni quali il controllo della convessità, del numero di input e di output. La struttura della classe è mostrata in Figura 3.16.

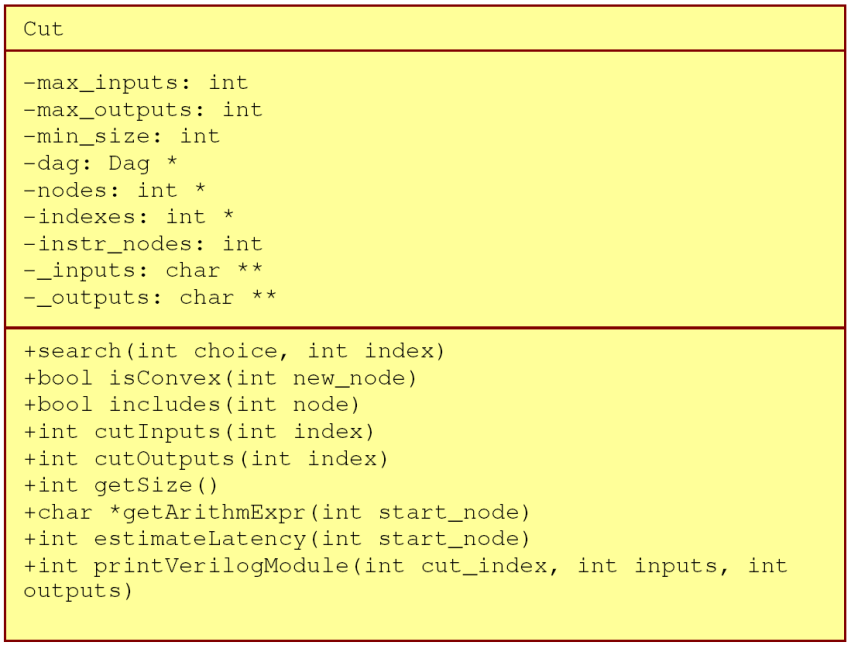


Figura 3.16: Classe Cut

Esaminiamo i dati membro nel dettaglio: ***max_inputs***, ***max_outputs*** e ***min_size*** rappresentano i vincoli architetturali imposti al taglio (numero massimo di input e output e numero minimo di nodi dell'AFU), ***dag*** è un puntatore al DAG su cui si sta applicando l'algoritmo, e ***nodes*** è il vettore che rappresenta il taglio. Come abbiamo detto in precedenza ad ogni elemento *i* del vettore ***nodes*** corrisponde un nodo del dag: se ***nodes[i]*** = 1 il nodo i-esimo è incluso nel taglio, viceversa è escluso. Poiché non tutti i nodi del DAG possono essere considerati nella ricerca (ad esempio quelli che indicano un'operazione di accesso alla memoria non vengono considerati), si è implementato un meccanismo di reindirizzamento ai nodi validi tramiti il vettore ***indexes***. Gli elementi di questo vettore sono gli ***index*** dei nodi del DAG che possono essere correttamente inclusi nella scelta dei tagli. Il numero di elementi di questo vettore, ossia il numero di nodi operatore adatti all'implementazione hardware, è memorizzato nella variabile ***instr_nodes***. Infine, sui vettori ***_inputs*** e ***_outputs*** sono annotate delle stringhe rappresentanti i nodi di input e output del taglio, rilevati durante la ricerca. Queste stringhe sono usate per costruire una stringa descrittiva dell'operazione eseguita dal taglio.

Passiamo ai metodi pubblici, gli unici accessibili all'utente.

search(choice, index) è l'algoritmo di ricerca vero e proprio. È una funzione ricorsiva alla quale si passano due interi. Il secondo – *index* – rappresenta il nodo che si vuole aggiungere al taglio esistente. Il primo, *choice*, è un valore booleano: se è pari a uno il nodo *index* verrà aggiunto al taglio, in caso contrario la ricerca

proseguirà ignorando il sottoalbero avente per radice il nodo *index*. Ogni volta che un nuovo nodo è aggiunto al taglio esistente, vengono richiamati i metodi ***isConvex()***, ***cutInputs()*** e ***cutOutputs()*** che eseguono i controlli di validità dei vincoli descritti nel Problema 1 della sezione 3.5.1. Se anche solo uno dei vincoli è violato, l'inclusione del nuovo nodo non va a buon fine e viene eliminato dallo spazio di ricerca tutto il sottoalbero che da quel nodo si dirama. A questo punto si è sicuri che il taglio rappresentato dal vettore ***cut*** sia un buon candidato per l'implementazione su FPGA. Per prima cosa si usa il metodo ***getSize()*** per conoscere il numero di nodi che compongono il taglio. Se questo valore non è inferiore al numero minimo di nodi per AFU definito nella macro MIN_CUT_SIZE, si passa alla generazione del file Verilog che descrive il taglio in questione col metodo ***printVerilogModule()***. All'interno di questa funzione viene richiamato il metodo ***getArithmExpr(start_node)*** per ricavare la stringa che descrive l'operazione effettuata dal taglio. Poiché un taglio può avere più di un output, e cioè eseguire più di una computazione, è necessario specificare quale espressione si vuole. Un metodo semplice per distinguere tra le espressioni è quello di fornire l'indice del nodo che rappresenta l'ultimo operando dell'operazione globale. L'argomento ***start_node*** passato è esattamente l'indice di questo nodo. Il metodo ***estimateLatency(start_node)*** restituisce il numero di cicli di clock necessari al processore per eseguire l'insieme di operazioni costituenti il taglio. Il significato del parametro ***start_node*** è lo stesso già visto per il metodo precedente. ***includes(node)*** è una funzione booleana che restituisce vero se ***node*** è incluso nel taglio, dove ***node*** è l'indice di un nodo del DAG.

Consideriamo ora la classe ***PathFinder***. Il vincolo della convessità è uno dei più difficili da verificare. Ricordiamo la definizione di sottografo convesso:

Un sottografo G di un DAG G^+ si dice convesso se, dati due nodi u e v appartenenti a G, non esiste nessun percorso che collega u a v costituito da nodi di G^+ non appartenenti a G.

In altre parole, per verificare la convessità di un taglio *S* dobbiamo accertarci che qualunque connessione tra due nodi qualunque di *S* non includa nodi che non appartengono a *S*. Risulta quindi immediato immaginare un test sulla convessità che, per ogni coppia di nodi appartenenti a *S*, cerchi tutti i possibili percorsi tra i due nodi e verifichi che nessuno di questi includa un nodo non appartenente a *S*.

La classe PathFinder, la cui struttura è mostrata in Figura 3.17, assolve a questo compito. Vediamo come:

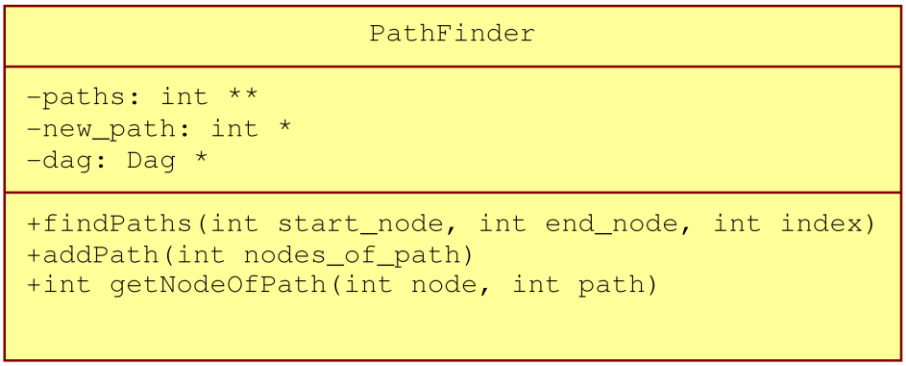


Figura 3.17: Classe PathFinder

Dati due nodi *u* e *v* di un DAG, chiamiamo *path* una sequenza di nodi che inizia con *u* e termina con *v*. Possono esistere diversi *path* che collegano due nodi tra loro. Il metodo ***findPaths(start_node, end_node, index)*** avvia la ricerca di tutti i possibili percorsi che collegano tra loro i nodi ***start_node*** e ***end_node*** del DAG puntato da ***dag***. Partendo dal nodo ***index*** tutti i nodi incontrati vengono annotati sul vettore ***new_path***. Quando il percorso collega effettivamente i nodi di partenza e di arrivo, ***new_path*** è memorizzato sulla struttura ***paths*** tramite il metodo ***addPath(nodes_of_path)***, dove il parametro ***nodes_of_path*** è il numero di nodi costituenti il percorso memorizzato in ***new_path***.

3.6 *Risultati*

Al fine di valutare l'incremento di prestazioni, introdotte dal partizionamento hardware/software del codice effettuato dal sistema fin qui descritto, si è proceduto ad effettuare dei test sul codice C che implementa tre noti algoritmi: ADPCM, DCT, IDCT. I test sono stati effettuati imponendo e variando dei vincoli sul numero massimo di ingressi e uscite e il numero minimo dei nodi che i singoli tagli dovevano possedere.
In Tabella 3.1 sono riportati i risultati più significativi relativi ai tagli selezionati per i tre algoritmi considerati. Per ognuno sono riportati il numero di porte di input e di output del taglio, la latenza software, la latenza hardware, lo speedup sia in termini di tempo che di fattore moltiplicativo e, infine, la latenza hardware calcolata con il metodo proposto in [27]. Per quel che riguarda il calcolo della latenza software ci si è riferiti all'ISA del MIPS mentre, per quello che riguarda la latenza hardware, è stato considerato il critical path delay delle sintesi logiche del codice HDL generato per la descrizione dei tagli identificati. Le sintesi sono state eseguite con il sintetizzatore

Xilinx XST 7.1 considerando una Virtex 4 (in particolare la XC4VLX25-10ff668) come dispositivo FPGA.

3.6.1 **Analisi dei risultati**

La DCT è spesso usata nel processing di segnali e immagini, in particolare per la compressione dei dati, perché la maggior parte dell'informazione del segnale tende ad essere concentrata in poche componenti a bassa frequenza della DCT. Trova impiego in numerosi algoritmi di compressione quali JPEG, MPEG.
Formalmente, la DCT è una funzione lineare e invertibile, rappresentabile come una matrice N x N.
La DCT si caratterizza per alcune sezioni di codice ricorrenti che sono relative al calcolo dei coefficienti. A queste sezioni corrispondono dei basic block relativamente grandi da cui il sistema di partizionamento è stato in grado di estrarre quattro DAG, contenenti gruppi di operazioni simili, che eseguiti dalle FU dedicate forniscono un'accelerazione significativa nell'esecuzione della DCT.
A differenza della DCT, l'IDCT mostra un risultato leggermente inferiore in termini di speedup raggiungibile, a causa della frammentazione del DAG principale in otto tagli da cui si è potuto ottenere otto FU più piccole in termini di operazioni elementari implementate.
L'ADPCM consiste fondamentalmente di due routines principali per la codifica e la decodifica dei dati in ingresso. Queste due routines sono rappresentate da due DAG principali. Il primo DAG, relativo all'operazione di codifica è composto da pochi nodi rendendo inefficace l'algoritmo di identificazione dei sottografi. Al contrario, l'algoritmo di identificazione e selezione si applica in modo efficiente al DAG relativo alla decodifica ADPCM permettendo di selezionare un sottografo che ha dimensioni molto simili al DAG di partenza. Questo avviene grazie all'applicazione della trasformazione di if-conversion che elimina i branches dovuti a sei costrutti *if-then-else* e permette la formazione di un unico DAG di dimensioni estese invece che sei differenti piccoli DAG. La FU corrispondente al taglio selezionato è circa tre volte più veloce della CPU nell'eseguire le operazioni rappresentate dal DAG.

DCT							
Cluster	**In**	**Out**	**Lat. SW(ns)**	**Lat. HW(ns)**	**M(Ci)(ns)**	**Speedup**	**Est.Lat.**
dag_fdct_8x8_5_cut0	9	4	91,11	17,23	73,88	5,29	43,38
dag_fdct_8x8_5_cut32	9	4	91,11	17,23	73,88	5,29	43,38
dag_fdct_8x8_9_cut44	10	4	100,00	17,22	82,78	5,81	47,15
dag_fdct_8x8_9_cut100	10	4	100,00	17,22	82,78	5,81	47,15
ADPCM							
Cluster	**In**	**Out**	**Lat. SW(ns)**	**Lat. HW(ns)**	**M(Ci)(ns)**	**Speedup**	**Est.Lat.**
dag_adpcm_decoder_6_cut0	16	4	44,44	13,41	31,03	3,31	26,57
IDCT							
Cluster	**In**	**Out**	**Lat. SW(ns)**	**Lat. HW(ns)**	**M(Ci)(ns)**	**Speedup**	**Est.Lat.**
dag_idctcol_3_cut281	13	4	88,89	19,15	69,74	4,64	45,19
dag_idctcol_3_cut1364	13	4	88,89	19,13	69,76	4,65	67,78
dag_idctcol_3_cut1936	12	4	75,56	19,13	56,42	3,95	45,19
dag_idctcol_3_cut2202	12	4	71,11	19,15	51,96	3,71	52,74
dag_idctcol_3_cut2305	12	4	71,11	19,13	51,98	3,72	52,74
dag_idctcol_3_cut2320	9	4	66,67	19,22	47,45	3,47	38,51
dag_idctrow_3_cut3	10	4	60,00	19,56	40,44	3,07	31,22
dag_idctrow_3_cut141	10	4	60,00	19,16	40,84	3,13	48,96

Tabella 3.1

3.6.2 Valutazione della precisione nel calcolo della latenza hardware

Il calcolo della latenza hardware fatto in questo lavoro non è una stima ma un dato reale ottenuto da una sintesi logica. Come già osservato nel paragrafo 3.5.3, in [27] la valutazione della latenza hardware viene eseguita nel seguente modo:

- Per ogni operazione logioco-aritmetica dall'ISA della CPU di riferimento, viene creato e sintetizzato un datapath hardware.
- Viene, quindi, creato un database che contiene i ritardi del percorso critico dei datapath sintetizzati.

- La latenza hardware di un taglio selezionato è quindi calcolata come somma dei ritardi delle singole operazioni che appartengono al cammino ingresso-uscita più lungo.

Questo approccio porta, inevitabilmente, a delle discrepanze dal risultato effettivo della sintesi dell'hardware che implementa il taglio. Il sintetizzatore, infatti, è in grado di considerare la funzione logica ingresso-uscita nella sua interezza applicando delle ottimizzazioni.

Per evidenziare questa differenza si è provato a fare il calcolo della latency hardware considerando l'approccio proposto in [27] confrontandolo con i risultati mostrati in Tabella 3.1. A tal fine è stato necessario effettuare le sintesi delle singole operazioni dell'ISA del MIPS implementate in datapath dedicati. A partire da questi risultati (mostrati in Tabella 3.2) si sono ricostruite le latenze hardware dei tagli selezionati e mostrati in Tabella 3.1.

SINGLE OPERAZIONS LATENCY (XST 7.1)						
FPGA: *XC4VLX25-10FF668*						
Operation	Slices	FFDs	LUTs	IOBs	GCLKs	Delay ns
Shifter >>>	109	99	185	97	1	4,851
Shifter >>	128	109	213	97	1	5,073
Shifter <<<	140	113	215	97	1	4,599
Shifter <<	140	113	215	97	1	4,599
32-bit Unsigned Adder	53	96	32	97	1	3,772
32-bit Signed Adder	53	96	32	97	1	3,772
32-bit (32 bit out) Signed Multiplier	326	153	512	97	1	10,201
32-bit (32 bit out) Unsigned Multiplier	326	154	512	97	1	10,201
32-bit (64 bit out) Signed Multiplier	712	357	1088	129	1	12,073
32-bit (64 bit out) Unsigned Multiplier	711	354	1057	129	1	11,611
32-bitwise and	55	96	32	97	1	1,711
32-bitwise or	55	96	32	97	1	1,711
32-bit mux_2_1	55	96	32	98	1	1,658

Tabella 3.2: Latency hardware delle singole operazioni dell'ISA del MIPS

I valori calcolati sono stati riportati nelle curve (indicate dal simbolo a forma di quadrato) di Figura 3.18 e Figura 3.19 dove sono confrontati ai valori ottenuti sintetizzando interamente i tagli selezionati (curve con il simbolo a forma di rombo).

In entrambe le figure la distanza tra le curve rappresenta l'errore tra la stima ottenuta con il calcolo della latenza hardware basata sulla somma dei valori della Tabella 3.2 e il reale ritardo del percorso critico ottenuto dalla sintesi del codice Verilog che rappresenta il taglio. In particolare in Figura 3.18, relativa all'algoritmo di IDCT, si osserva un andamento dell'errore variabile mentre nella Figura 3.19, relativa alla

DCT, l'andamento è costante. Questa differenza è di particolare interesse dato che la funzione di merito assume valori diversi con una conseguente differenza nella selezione dei tagli. Considerando che il calcolo usato nel sistema proposto in questo lavoro, si basa su un valore più preciso della latenza hardware delle FU, è possibile che la selezione risultante possa essere maggiormente efficiente.

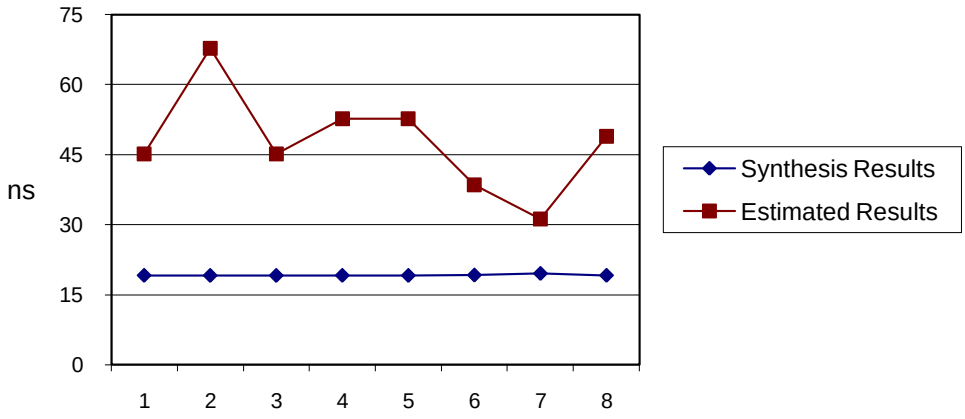


Figura 3.18: IDCT

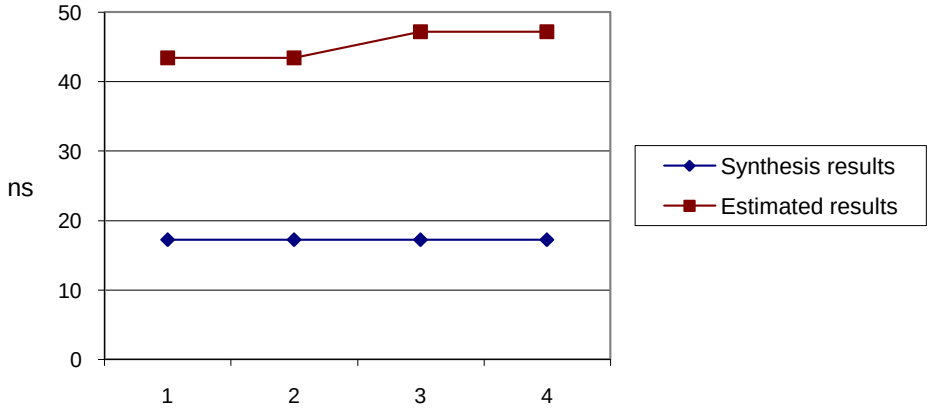


Figura 3.19: DCT

Capitolo 4

Strumenti per il software debugging in sistemi dinamicamente riconfigurabili

L'attività di debugging è una parte importante nel processo di sviluppo software ma anche molto delicata e soggetta ad errore ad essere inefficiente se non supportata dagli strumenti appropriati.
È molto utile, ad esempio, dare la possibilità all'utente di analizzare lo stato dell'applicazione non appena il baco si presenta in modo da poter isolare più facilmente le cause che hanno portato il programma ad avere il comportamento osservato.
Nei sistemi embedded l'attività di debugging viene eseguita usando un client debugger remoto che si connette al processo da debuggare mediante un protocollo di comunicazione. Il debugger può accedere alle risorse del processore seguendo un approccio software o un hardware. Nel primo caso una porzione di codice chiamata "debug stub" viene eseguita sul processore del sistema embedded nel momento in cui è necessario interfacciarsi con il debugger remoto (ad esempio perché è stata rilevata una trap instruction). Lo stub si connette quindi al debugger e comunica implementando via software il protocollo. In questo caso non è necessario nessun hardware addizionale ma è presente un certo overhead software. Vi è inoltre l'impossibilità di osservare lo stato del processo esattamente quando avviene una trap. Nel secondo caso un modulo hardware (ad esempio un'interfaccia JTAG) è accoppiato con il processore e può accedere direttamente alle sue risorse comunicandole poi al debugger remoto. Il modulo hardware può essere più o meno complesso in funzione delle caratteristiche implementate, ma è comunque intrinsecamente poco flessibile poiché, l'aggiunta di una funzionalità, impone una reimplementazione dell'intero sistema presente sul chip. Un altro aspetto, che rappresentare una minaccia per la sicurezza, risiede nel fatto che il modulo hardware può accedere alle risorse del processore in modo incondizionato e senza un controllo del software di sistema. Questo non permette un suo uso esteso in un contesto general-purpose multi-utente dove un approccio comune è quello di fornire in

hardware un supporto minimale per il single-step e lasciare che il software di sistema si occupi del resto.

In questo capitolo viene descritto eBug, un sistema per il supporto al debugging delle applicazioni per il processore eMIPS [1]. eMIPS è composto da un processore base che non può essere modificato ma le cui funzionalità possono essere estese e potenziate sfruttando la PR dei dispositivi FPGA. In questo modo è possibile caricare e scaricare dinamicamente dei moduli aggiuntivi di logica che vengono chiamati *estensioni*. Questi moduli, accedendo alla pipeline del processore e ad altre sue risorse, permettono di realizzare dei task specifici estendendo, di fatto, le capacità del modulo base di eMIPS. Le funzionalità aggiuntive possono essere di vario tipo come, ad esempio, delle unità computazionali specifiche o delle periferiche di I/O.

Nel caso di eBug, la capacità di eMIPS di poter essere esteso, è stata utilizzata per sviluppare un sistema di software debugging flessibile, con basso overhead e pensato anche per far fronte ai problemi di sicurezza di cui si è parlato in precedenza.

La flessibilità si evidenzia sia dal punto di vista hardware che da quello software. eBug, infatti, è concepito per essere facilmente estendibile in modo da fornire funzionalità complesse alle sessioni di debugging. Chiaramente un incremento di funzionalità impatta sul footprint del modulo hardware ma, dato che il suo impatto sul sistema non è permanente ma solo contingente al periodo di sviluppo del software, questo non risulta essere un problema. Durante la fase di sviluppo sono state realizzate, in tempi brevi, un certo numero di estensioni per fornire diversi livelli di supporto al debugging dimostrando la flessibilità dell'approccio da un punto di vista hardware. La flessibilità software è fornita da un'applicazione che si occupa delle operazioni che sarebbe troppo complesso implementare direttamente in hardware come, ad esempio, l'implementazione del protocollo remoto per comunicare con il client debugger.

eBug ha un basso (se non addirittura nullo) impatto in termini di overhead per il processore. Quando si presenta un errore in un'applicazione in esecuzione su eMIPS, eBug ferma immediatamente il processore. In questo modo si evita l'esecuzione del codice usato per la gestione di una trap da parte del sistema operativo e si può osservare lo stato del processore esattamente nel momento in cui è avvenuto l'errore. Anche l'accesso alle risorse del processore e la comunicazione con il debugger remoto non implica l'esecuzione di codice estraneo all'applicazione come nel caso dell'utilizzo di uno stub.

Il problema della sicurezza è affrontato facendo in modo che l'attività di debugging, e quindi l'accesso di eBug alle risorse, sia sotto il controllo del sistema operativo. Quando un processo deve essere testato, il sistema operativo carica sia il suo eseguibile che il modulo hardware di eBug passando poi il controllo al processo

stesso. Finché il processo rimane attivo eBug può accedere alla pipeline e alle altre risorse del processore ma, quando il sistema operativo schedula un nuovo processo, eBug viene disabilitato e la sessione di debugging viene congelata evitando così che possa interferire con la restante parte del sistema. Tutti gli accessi in memoria operati da eBug sono inoltre filtrati tramite una MMU. In questo modo è garantita l'impossibilità di corrompere dati non relativi al processo attivo dato che, in occasione di un accesso ad un'area riservata, verrebbe immediatamente rilevata un'eccezione gestirà dal sistema operativo. Questi accorgimenti fanno di eBug un sistema di debugging con approccio hardware sicuro anche in un sistema general-purpose multiutente.
Come ultimo aspetto va evidenziato che eBug è stato concepito per poter cooperare con altri moduli al fine di rendere ancora più potenti gli strumenti a cui l'utente può fare riferimento per le attività di debugging. Un esempio è l'utilizzo insieme a P2V [2], un sistema di verifica per la correttezza basato su un supporto hardware alle asserzioni. Con P2V il programmatore può creare delle asserzioni sia prima che dopo la compilazione del programma. Queste asserzioni vengono trasformate in estensioni hardware per il processore eMIPS e osservano il comportamento specificato dalla semantica delle asserzioni segnalando un'eventuale violazione. Questo segnale è poi usato da eBug per può fermare l'esecuzione e passare il controllo ad una sessione di debugging nel momento in cui è stata violata l'asserzione. Il tutto avviene senza alcun overhead software.
In questo capitolo verrà descritta la prima implementazione di eBug, analizzate le sue caratteristiche architetturali, di sicurezza e di estendibilità. In particolare verrà mostrato com'è stato possibile aggiungere, con poco sforzo, il supporto ai watchpoints e breakpoints hardware quantificando il miglioramento delle prestazioni a fronte di un'aumentata complessità dell'hardware.

4.1 *Esempi di sistemi di supporto al debugging*

Un primo esempio di supporto on-chip per il debugging del software, è quello realizzato per il processore Leon [35], una CPU RISC a 32 bit, il cui codice HDL è open source, progettata dalla Gaisler Research e l'Agenzia Spaziale Europea. Leon è un'implementazione dell'ISA del processore Sparc V8 [36] e, nella sua seconda revision (Leon2) è stato dotato di un'unità di supporto onchip per il debugging (DSU). La DSU fornisce un'interfaccia per creare una connessione tra GDB [37] e il processore reale o un suo modello per la simulazione. La comunicazione tra il Leon2 e il PC remoto, in cui è eseguito GDB, avviene mediante porta seriale. In una più

recente versione del processore Leon (il Leon3) la DSU è connessa al bus di sistema come periferica slave, utilizzabile con diverse tipologie di interfaccia quali la UART, JTAG, USB o Ethernet.
Un altro sistema per il supporto al debugging remoto su sistemi embedded è XMD (Xilinx Microprocessor Debugger) [38], proposto dalla Xilinx per i processori Microblaze [20] e PPC [39]. XMD è un tool software usato per interfacciare una sessione remota di GDB con un processore implementato su FPGA o con un simulatore cycle-accurate dell'ISA del PPC o del Microblaze. Nel caso del PPC, il processore, essendo integrato nel chip della FPGA, possiede un modulo built-in che consente di stabilire un collegamento con XMD mediante un'interfaccia JATG. Nel caso del Microblaze invece si può avere o un debug stub o un modulo hardware aggiuntivo che però, una volta inserito, non può essere escluso se non riconfigurando l'intera FPGA. Questo modulo, chiamato MDM [40], si connette a XMD mediante un'interfaccia JTAG.
Sia il supporto al debugging per il Leon che quello proposto da Xilinx, non si avvantaggiano delle capacità di riconfigurabilità dei dispositivi FPGA. Ad esempio, Leon è stato sviluppato per essere principalmente implementato in ASIC e, anche se alcune implementazioni su FPGA sono state realizzate, queste ultime non utilizzano la PR per inserire e escludere la DSU a runtime. Una volta che la DSU viene inserita, una parte dell'area del chip, o della FPGA, è utilizzata anche quando nessuna attività di debugging è effettivamente richiesta impedendo, quindi, l'uso di quella porzione del dispositivo per altri scopi. Nel caso si volesse escludere o modificare la DSU, si dovrebbe procedere ad una nuova sintesi ed implementazione dell'intero processore, il che implica una nuova fase di validazione.
Al contrario, l'hardware dell'estensione eBug, si avvantaggia della PR delle FPGA per riutilizzare dinamicamente l'area del dispositivo quando il debugging del software non è necessario. Utilizzando l'interfaccia comune a tutte le estensioni di eMIPS, è possibile modificare ed estendere eBug riducendo la fase di test solo al modulo per il debugging evitando quindi di dover rivalidare tutto il sistema. In questo modo risulta molto semplice aggiungere delle nuove funzionalità ad eBug.
Considerazioni simili a quelle fatte per la DSU del Leon, si applicano anche al supporto al debugging proposto da Xilinx. Il modulo hardware per il debugging del PPC è integrato direttamente nel chip e, quindi, la sua area non è recuperabile. Per quel che riguarda il Microblaze, l'inserimento del modulo MDM è opzionale ma va deciso prima di configurare la FPGA poiché, ogni successiva modifica, impone una riconfigurazione totale del dispositivo con conseguente blocco di tutte le applicazioni in esecuzione su Microblaze.

L'utilizzo del bus JTAG può, inoltre, essere causa di problemi di sicurezza. Essendo un sistema proprietario, non è chiaro se il protocollo JTAG venga usato dal MDM solo per comunicare con un PC remoto o anche per accedere alle risorse del processore come, ad esempio, il register file. In quest'ultimo caso il JTAG, avendo accesso a bassissimo livello alle risorse del dispositivo, non è controllabile dal software di sistema. Ad esempio, qualora il processore operi in un contesto multitasking, il sistema operativo potrà eseguire un certo numero di context-switching durante la sessione di debugging. Se MDM non venisse usato correttamente, potrebbe corrompere lo stato di altri processi o di altre parti del sistema. Questa eventualità è impossibile con eBug dato che una estensione è caricata dal sistema operativo insieme al un processo a cui è subordinata. In sostanza, l'estensione non può "vedere" o modificare lo stato di altri processi poiché, quando il processo che la possiede viene disattivato per un cambio di contesto o per qualche altro motivo, l'estensione stessa viene disattivata.

4.2 *Il Processore eMIPS*

eMIPS è un microprocessore estendibile dinamicamente sviluppato dall'Embedded System Group di Microsoft Research. Per estendibilità del processore si intende la possibilità di aggiungere dinamicamente della logica addizionale a qualsiasi stadio della pipeline del processore base.
La logica addizionale, che viene chiamata *Estensione*, può essere usata per aumentare l'efficienza del processore su task specifici incrementando le prestazioni globali del sistema. Le Estensioni possono essere caricate dinamicamente sul dispositivo durante il funzionamento del processore e solo quando sono effettivamente richieste.
La Figura 4.1 mostra un diagramma a blocchi dell'architettura del processore eMIPS. La pipeline di base, il register file e l'interfaccia alla memoria sono le stesse di una CPU RISC classica [41] e sono rappresentate con un colore più chiaro. In particolare, il progetto eMIPS implementa come base l'ISA del MIPS R4000. Questa porzione del core di eMIPS viene chiamata Trusted ISA o TISA ed è richiesta per le operazioni iniziali e per garantire l'esecuzione di un codice compilato per il processore MIPS. Questa parte, che non può essere rimossa o disabilitata, deve essere presente allo startup del sistema operativo e contiene anche il coprocessor-0 del MIPS. Il TISA, inoltre, include tutte le funzionalità per l'utilizzo e il controllo delle Estensioni come, ad esempio, le istruzioni per il loro caricamento, scaricamento, abilitazione e disabilitazione. A livello funzionale i blocchi della

pipeline lavorano come un classico processore RISC eccetto che per l'interconnessione tra loro e gli slot in cui possono risiedere le Estensioni.

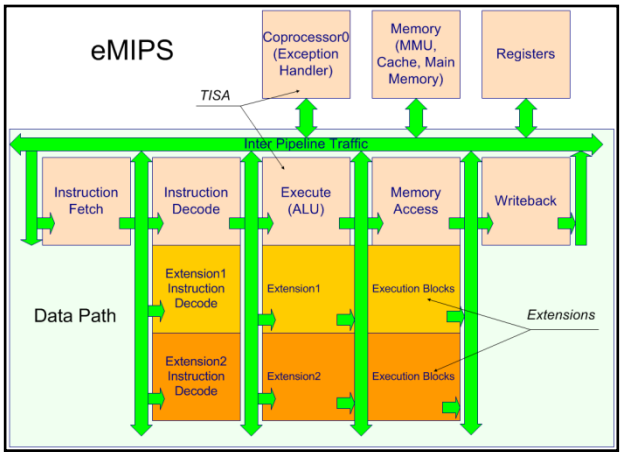


Figura 4.1: Diagramma a blocchi di eMIPS

Una semplice Estensione, come quella mostrata con colore più scuro in Figura 4.1, include uno stadio di instruction decode (ID) e uno stadio di Execute che si può estendere fino allo stadio di Memory Access del TISA. Questo accorgimento permette ad un'Estensione di eseguire operazioni con latenza di due cicli, senza influenzare il funzionamento della pipeline del TISA con l'eventuale inserimento di bolle. Se un'Estensione ha bisogno di più di due cicli di clock per compiere un'operazione, può richiedere al controllo di stallare la pipeline del TISA mantenendo il possesso di tutte le risorse delle pipeline. Le operazioni multiciclo sono necessarie, ad esempio, quando l'Estensione deve accedere al sottosistema di memoria. In quest'ultimo caso, l'accesso avviene sempre tramite una MMU e, quindi, l'Estensione può indirizzare solo lo spazio di memoria del processo che la possiede. Per motivi di sicurezza, l'estensione non ha accesso alle risorse privilegiate (ad esempio i registri del coprocessor-0) ad eccezione del caso in cui è posseduta da un processo eseguito in modalità privilegiata. Tutte le altre risorse non privilegiate come il register file e la memoria sono invece accessibili.

In generale ogni Estensione osserva le istruzioni durante la fase di decode in parallelo all'unità di decode della pipeline "classica". Un'istruzione, quindi, può essere riconosciuta dal TISA, da una o più Estensioni oppure da nessun modulo di decode. Nel caso in cui un'istruzione appartenente al TISA sia riconosciuta anche da una o più Estensioni, ad esempio per implementare un overlay dell'istruzione stessa, un meccanismo basato sulle priorità passa il controllo al sottosistema con priorità più elevata. Nel caso che il controllo venga trasferito ad un'Estensione, quest'ultima esegue le sue operazioni per poi restituirlo scrivendo sul PC l'indirizzo

dell'istruzione successiva. Quest'ultimo non è necessariamente sequenziale al PC dell'istruzione riconosciuta dall'Estensione.
Sebbene un'Estensione sia usata spesso per potenziare le capacità computazionali del processore, può anche essere utilizzata, ad esempio, per implementare una periferica o come, come descritto in questo capitolo, per creare un supporto per il debugging remoto del software in esecuzione sul processore.

4.3 *eBug Overview*

eBug è composto di due componenti fondamentali: uno software (l'applicazione **emips2gbd**) e uno hardware (l'estensione **eBug**). Le due componenti cooperano fornendo il supporto necessario al debugging remoto per le applicazioni in esecuzione su eMIPS. L'utilizzo di una parte software ha permesso di ridurre la complessità dell'hardware permettendo la realizzazione di un'Estensione a basso impatto d'area. Nel software sono state realizzate alcune funzionalità complesse come l'implementazione del protocollo remoto usato dal client debugger. Nel caso specifico del debugger GDB, utilizzato durante lo sviluppo di eBug, il protocollo è composto da uno stream di caratteri che necessita un parsing per decodificare il comando desiderato. Implementare un parser in hardware risultava essere una soluzione dispendiosa in termini d'area e, sopratutto, non flessibile. Se, ad esempio, venisse deciso di cambiare client debugger la porzione di hardware relativa al parsing andrebbe riprogettata. Il componente *emips2gdb*, implementato come una singola applicazione che viene eseguita sotto il sistema operativo di un PC host remoto, è mostrato in Figura 4.2 e Figura 4.3. Come verrà spiegato nel paragrafo successivo, *emips2gdb* si comporta come una interfaccia tra il PC in cui è eseguito il client debugger e il sistema target. Quest'ultimo può essere sia un'effettiva implementazione su FPGA (Figura 4.2) che un modello per la simulazione (Figura 4.2) di un sistema eMIPS. Nel caso della simulazione di sistema è stato utilizzato il simulatore Giano [3] [42].
Il componente hardware è implementato come un modulo Verilog che può essere sintetizzato separatamente come Estensione oppure caricato, insieme al resto dei moduli e delle periferiche eMIPS, all'interno di una simulazione Giano. L'estensione eBug è stata sviluppata per raggiungere tre obiettivi principali:

1. *Riutilizzo dell'area*: eBug occupa una parte di area del dispositivo solo quando un programma in esecuzione deve essere debuggato. Se nessuna attività di debugging è necessaria, l'area viene rilasciata e può essere usata da altre Estensioni.

2. *Sicurezza*: Anche se un'Estensione è presente in uno slot riconfigurabile, il processore eMIPS può abilitarla e disabilitarla dinamicamente facendo si che possa accedere o meno alle risorse del processore. Questa caratterista viene usata per attivare eBug solo quando il processo oggetto di debugging è schedulato dal sistema operativo. In questo caso, infatti, i comandi del client debugger hanno effetto solo sul processo a cui è associata l'estensione. Nessuno degli altri processi presenti nel sistema può essere modificato dall'attività di debugging. eBug, inoltre, ha accesso ai registri ed alla memoria mediante l'interfaccia comune a tutte le Estensioni e non tramite un canale di basso livello come il JTAG. Questo permette al sistema operativo di avere il completo controllo dell'attività di eBug e di prevenire, quindi, accessi indesiderati, alle risorse del sistema, da parte del client debugger.
3. *Estendibilità*: il componente hardware di eBug è stato concepito e realizzato per essere un'Estensione estendibile. La sua struttura, molto modulare, permette di aggiungere in modo molto semplice nuove funzionalità alla versione base. In questo modo eMIPS non è limitato ad un supporto hardware rigido ma, in funzione delle necessità dell'utilizzatore, può evolvere e fornire caratteristiche più complesse. L'unico vincolo nell'espandibilità del componente hardware è dato dalle sue dimensioni in termini di area occupata. Quest'ultima deve rimanere minore di quella disponibile in un Extension Slot, cioè nella porzione del dispositivo a disposizione per l'inserimento dinamico delle estensioni.

Figura 4.2: Collegamento all'Hardware

Figura 4.3: Collegamento al simulatore

4.4 *Il Componente Software di eBug: emips2gdb*

Come introdotto in precedenza, il componente software di eBug è costituito da un'unica applicazione, *emips2gdb*, eseguita sul PC host che si collega al sistema

target eMIPS. Come mostrato in Figura 4.2 e in Figura 4.3 *emips2gdb* permette di collegare GBD ad un sistema eMIPS implementato su FPGA oppure ad una suo modello eseguito sul simulatore Giano. Nel primo caso *emips2gdb* si collega al sistema hardware mediante la linea seriale del PC mentre, nel secondo caso, il modello possiede un'interfaccia, basata sulle PLI [43], per simulare il comportamento della linea seriale di eBug. Quest'interfaccia crea una namepipe alla quale *emips2gdb* può collegarsi.

Per descrivere il funzionamento di *emips2gdb*, consideriamo una tipica sessione di debugging eseguita su eMIPS. La sessione comincia con l'esecuzione di *emips2gdb* che, come prima operazione, crea un server GDB da un lato e, in funzione del tipo di utilizzo che vuol fare l'utente, una connessione alla porta seriale o alla namepipe dall'altro. A questo punto è possibile eseguire GDB e connetterlo al server che, dal punto di vista del debugger, è il target con cui comunicare per mezzo del suo protocollo remoto.

Una volta che la comunicazione è stabilita, *emips2gdb* tradurrà i comandi di GDB in un protocollo semplificato utilizzato dall'Estensione eBug per eseguire le operazioni richieste.

Allo stato attuale *emips2gdb* supporta solo GDB come client debugger ma è possibile fornire il supporto ad altri, ad esempio WinDbg [44], semplicemente aggiungendo una classe al codice di *emips2gdb* che implementi la traduzione tra il protocollo remoto del nuovo debugger ed il protocollo seriale usato dall'Estensione.

Il protocollo per la comunicazione tra *emips2gdb* e l'Estensione eBug è stato concepito per essere semplice da decodificare e per non richiedere un volume di trasmissione dati elevato al fine di non pesare eccessivamente sulla linea seriale. Utilizzando questo protocollo *emips2gdb* può eseguire le seguenti operazioni di base:

1. Sospendere e Ripristinare l'esecuzione di un processo che possiede l'estensione eBug.
2. Leggere e scrivere i registri di Emips.
3. Accedere alla memoria in lettura e scrittura.

Il debugger può usare queste operazioni di base per eseguirne altre più complesse come, ad esempio, il single step e l'utilizzo di breakpoints e watchpoints software. Per eseguire il single step è compiuta una serie di operazioni che coinvolgono solo i punti 1 e 3. In sostanza quando viene richiesto uno step, viene letto e memorizzato il contenuto dell'istruzione successiva a quella in cui il programma è stato sospeso. Questa istruzione è poi sostituita con una Break per mezzo di una scrittura in memoria istruzioni. In seguito il programma viene ripristinato ma la prima istruzione incontrata sarà proprio la Break che, riconosciuta dall'Estensione eBug, causerà

l'immediata sospensione del processo e la sostituzione della Break con l'istruzione originale. A questo punto si può anche procedere con un successivo comando di step.
Il protocollo usto da *emips2gdb* è formato da un flusso di byte che comincia sempre con il *command byte*. Come mostrato in Figura 4.4, il *command byte* può avere due formati possibili. Il primo utilizza tre campi di bit ed è utilizzato per inviare comandi che si riferiscono all'accesso dei registri del processore eMIPS. Il secondo formato utilizza invece due soli campi di bit ed è utilizzato per le operazioni di accesso alla memoria e di controllo. In entrambi i formati, il campo *opcode* identifica l'operazione che deve essere eseguita.

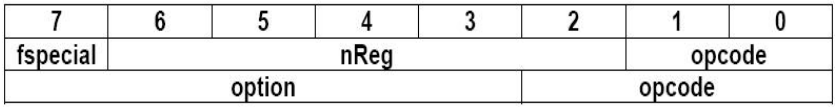


Figura 4.4: Formato del *Command byte*

L'insieme dei valori possibili del campo *opcode* sono mostrati nella prima colonna della Tabella 4.1. La seconda colonna indica l'intervallo dei valori assumibili, quando viene utilizzato dal campo *option*. Infine, nell'ultima colonna è indicato il numero di byte attesi come risposta dell'Estensione al commando inviato.

opcode	*option*	*Operation*	*Bytes returned*
x00	**N/A**	**Read from an eMIPS register**	**4**
x01	**N/A**	**Write to an eMIPS register**	**1 (Ack)**
010	**0x0-0x1F**	**Fetch byte from memory**	**variable**
011	**0x0-0x1F**	**Store byte to memory**	**1 (Ack)**
110	**00000**	**Suspend**	**1 (Ack)**
110	**00001**	**Continue**	**1 (Ack)**
111	**-----**	**Future Expansion**	**-----**

Tabella 4.1: Comandi base del protocollo eBug

4.4.1 Operazioni di Controllo

Le operazioni di controllo sono quelle con cui è possibile richiedere ad eBug di sospendere o ripristinare l'esecuzione del processo associato all'Estensione. Come si può vedere dalla Tabella 4.1, esistono due comandi: *Suspend* e *Continue*.
Una volta che GDB si connette al server creato da *emips2gdb*, quest'ultima applicazione invia un byte di *Suspend* all'Estensione Ebug che richiede al processore eMIPS, mediante l'interfaccia di comunicazione processore-Estensione, di andare in uno stato di stallo. Una volta che eMIPS è stallato, viene inviato un byte di

acknoledge ad *emips2gdb* ed è possibile accedere alle risorse del processore tramite il debugger.
Quando il debugger invia un comando per indicare di ripristinare l'esecuzione, *emips2gdb* invia un byte di *Continue* per richiedere che il processo sospeso torni in esecuzione. Questa situazione può presentarsi non solo per una richiesta dell'utente di ripristinare l'esecuzione del programma ma anche, per esempio, durante la gestione del single step o dei breakpoints senza supporto hardware.

4.4.2 Operazioni per l'accesso ai registri

Come osservato in precedenza, un'operazione di accesso ai registri utilizza un formato del *command byte* con tre campi: *l'opcode, nReg* e fSpecial. L'*opcode* ha il bit uno sempre a zero mentre il bit zero indica il tipo di accesso richiesto ovvero una lettura o una scrittura. I campi *nReg* e *fSpecial* sono utilizzati per identificare il registro destinazione seguendo la convenzione descritta nella Tabella 4.2.
Nel caso di un'operazione di lettura viene inviato un unico byte all'Estensione eBug che, una volta identificato il registro da leggere, ne richiede il valore al processore. I 32 bit contenenti il valore letto dal registro e inviati come quattro byte con ordine big-endian, vengono ricevuti da *emips2gdb* che trasferisce l'informazione a GDB.

fSpecial	*nReg*	*Register*
0	**0-31**	**GPR file register number**
1	**0**	**PC**
1	**1**	**hi**
1	**2**	**lo**
1	**3**	**sr**
1	**4**	**bad**
1	**5**	**cause**
1	**6**	**fsr**
1	**7**	**fir**

Tabella 4.2: Codifica per l'accesso ai registri

Nel caso in cui è richiesta un'operazione di scrittura, il protocollo cambia leggermente. L'applicazione *emips2gdb* invia inizialmente il *command byte* seguito da quattro byte, con ordine big-endian, che contengono il valore da scrivere sul registro. Una volta che il valore è stato memorizzato correttamente nel registro eMIPS richiesto, l'Estensione invia un byte di acknoledge per informare *emips2gdb* che l'operazione richiesta è avvenuta con successo.

In questa prima implementazione di eBug è possibile accedere in lettura e scrittura su tutti i registri del general purpose register file e sul PC mentre, i registri del coprocessor-0, lo e hi, sono accessibili in sola lettura.

4.4.3 **Operazioni sulla memoria**

Le operazioni di accesso alla memoria si distinguono per un numero variabile di byte che possono essere inviati e ricevuti da *emips2gdb*. Come osservato in precedenza, il primo byte inviato è il *command byte* ma, in questo caso, il formato usato contiene solo due campi: *opcode* e *option*. Il campo *opcode*, oltre ad indicare che si tratta di un'operazione di accesso alla memoria, contiene anche l'informazione relativa al tipo di accesso in memoria (lettura o scrittura). Il campo *option* è usato per fornire l'informazione sulle dimensioni del blocco di memoria al quale s'intende accedere. In funzione del suo valore cambia il numero di byte inviati verso l'Estensione:

- Il valore è 0: I due byte (ordinati big-endian) che seguono il *command byte* indicano la dimensione del blocco di memoria che deve essere scritto o letto. Avendo a disposizione 16 bit per indicare questa dimensione, è possibile accedere a blocchi di 64KB con una singola transazione. In realtà è stato osservato che il protocollo remoto GDB scompone gli accessi alla memoria in blocchi con dimensione massima di 400 byte che, comunque, non consentono di utilizzare il solo campo *option* per specificare la dimensione.
- Il valore è compreso tra 1 e 31: In questo caso il campo *option* specifica direttamente la dimensione del blocco da processare e non è necessaria la spedizione di altri byte. Questo permette di rendere più efficienti gli accessi alle singole locazioni o per piccoli blocchi di memoria.

Al *command byte*, o agli eventuali byte per la specifica delle dimensioni del blocco, seguono sempre altri quattro byte per specificare l'indirizzo iniziale dell'operazione in memoria.

Un'altra differenza nel numero di byte inviati o ricevuti da *emips2gdb* si ha nel caso in cui l'operazione sia una lettura oppure una scrittura.

Nel caso di una lettura in memoria la trasmissione si ferma dopo i quattro byte relativi all'indirizzo e *emips2gdb* attende dall'Estensione un flusso di byte con i valori del blocco di memoria letto. Una volta che l'ultimo byte è stato ricevuto, la transazione è conclusa e le informazioni vengono inviate a GDB. In questo caso non viene inviato nessun byte di acknoledge da eBug.

Nel caso di una scrittura, ai byte che contengono le informazioni sull'indirizzo, seguono i byte con i dati da scrivere in memoria. Dato che eBug possiede l'informazione sulle dimensioni del blocco, non è necessario alcun byte che ne

indichi la fine. Una volta che l'estensione ha scritto tutti i byte in memoria, viene inviato un byte di acknoledge verso *emips2gdb*.

4.5 *Il Componente Hardware di eBug*

L'Estensione eBug possiede alcune differenze rispetto ad una tipica Estensione del processore eMIPS. Non esegue, infatti, alcuna istruzione appartenente ad un'estensione dell'ISA e non implementa nessun task computazionale, ma si occupa di assumere il controllo del processore se accade uno di questi due eventi:

1. Un istruzione *break* si trova nello stadio di Instruction Decode, o
2. Il client debugger richiede la sospensione del processo che possiede l'Estensione eBug.

In entrambi i casi eBug stalla il TISA e prende il controllo del processore. Chiaramente qualora le funzionalità di eBug venissero estese, come descritto in 4.6 , anche altri eventi potranno causare una sospensione del processo.
eBug restituisce il controllo al TISA in una delle seguenti condizioni:

- Il sistema operativo schedula un altro processo, oppure
- Il client debugger invia un comando *Continue.*

A parte le differenze descritte, la struttura dell'Estensione è del tutto simile quella di altre Estensioni eMIPS.
eBug è stato sviluppato totalmente in Verilog e, nella
Figura 4.5, è evidenziata la relazione tra i moduli che compongono la sua struttura interna. Il modulo che si trova gerarchicamente più in alto, che per una convenzione del sistema eMIPS deve essere chiamato extension0, è un modulo wrapper che espone tutti i segnali con cui il TISA si interfaccia ad una generica Estensione. Di questi solo alcuni segnali di input sono utilizzati dal modulo principale dell'Estensione eBug (*debug_extension*) mentre tutti i segnali di output, anche quelli non utilizzati, sono tenuti ad un certo valore costante.
Il modulo *debug_extension* istanzia al suo interno tre moduli. Il primo modulo, chiamato *reset_manager*, si occupa della gestione del segnale di reset globale dell'Estensione. Gli altri due (*ext_debug_control* e *Top_debug*, mostrati in Figura 4.6) gestiscono l'interfacciamento con il pipeline arbiter, i registri del TISA e il sottosistema di memoria e verranno descritti nei paragrafi successivi.

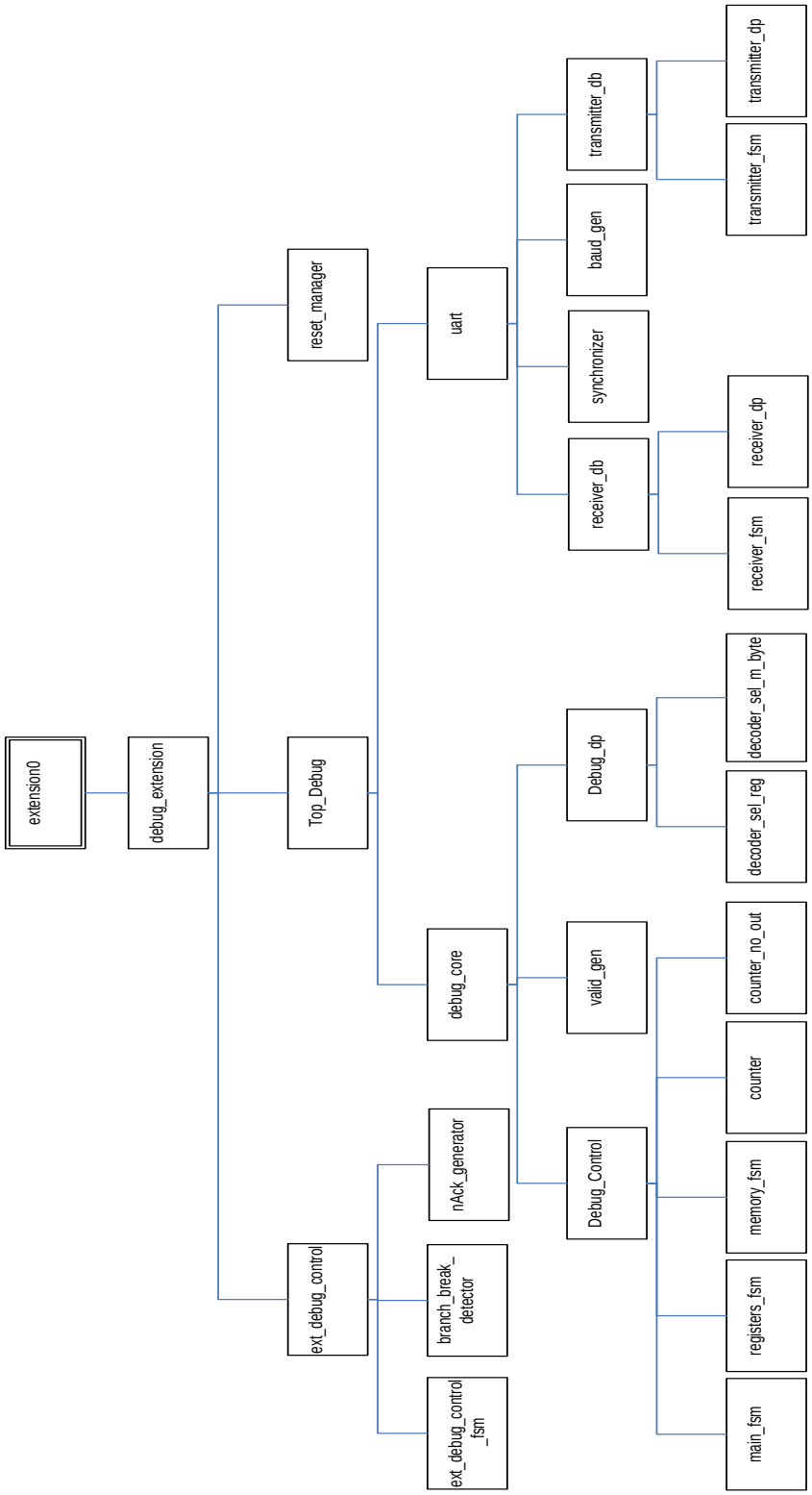


Figura 4.5: Gerarchia dei moduli

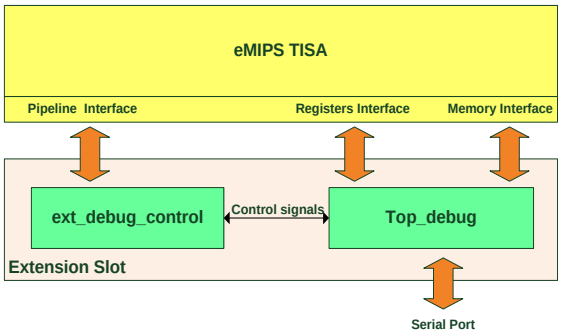


Figura 4.6: Interfaccia Esterna di eBug

4.5.1 Interfacciamento al Pipeline Arbiter

Il processore eMIPS è stato concepito per eseguire sia istruzioni appartenenti all'ISA del MIPS [45] che istruzioni estese, cioè istruzioni non appartenenti all'ISA originale. Nel primo caso il datapath "classico" si occupa della loro esecuzione mentre, nel secondo caso, le istruzioni sono eseguite da unità funzionali implementate da Estensioni eMIPS.

Il riconoscimento di un tipo di istruzione avviene durante lo stadio di Instruction Decode. In questa fase sia il TISA che le Estensioni possono riconoscere un'istruzione e, quando questo avviene, abbassano un segnale chiamato *recognized instruction* (RI). Il sottosistema di eMIPS, che si occupa di gestire tutti i segnali RI provenienti dalle unità ID del TISA e delle Estensioni, è il *Pipeline Arbiter*. In generale il Pipeline Arbiter trasferisce il controllo al modulo che ha abbassato il segnale RI. Dato che più moduli possono decidere di eseguire una stessa istruzione, il Pipeline Arbiter utilizza un meccanismo di priorità che stabilisce il "proprietario" dell'istruzione e quindi il modulo a cui verrà dato il controllo delle risorse (registri, memoria, etc.). Il TISA ha, per default, una priorità maggiore rispetto alle Estensioni ma è possibile modificarla permettendo l'override delle istruzioni del TISA.

eBug utilizza questo meccanismo per richiedere il controllo quando viene decodificata un'istruzione *break* provocando così lo stallo del processore come se si trattasse di un'istruzione multi ciclo. Questo fa si che il TISA non riconosca la *break* e quindi non trasferisca il controllo al sistema operativo per la gestione di una trap software, eventualità che avrebbe come conseguenza indesiderata la modifica dello stato del processore.

Lo stesso meccanismo è utilizzato quando il client debugger richiede una sospensione del processo, ad esempio quando si connette per la prima volta ad eMIPS. In questo caso l'Estensione eBug riconosce incondizionatamente l'istruzione

che si troverà nello stadio ID al ciclo successivo di pipeline e sospenderà l'esecuzione esattamente in quel punto. Dato che questa istruzione non viene eseguita, quando verrà dato un comando *Continue,* il PC del processore assumerà il valore dell'indirizzo di questa stessa istruzione. L'unica eccezione a quanto detto riguarda una caratteristica relativa alla gestione dei branch nel processore MIPS e quindi in eMIPS. Nel MIPS un'istruzione che segue un branch è detta *delay-slot instrunction* e viene eseguita indipendentemente dal fatto che il salto sia fatto o no. Questo è un problema nel caso in cui eMIPS venga sospeso proprio in corrispondenza di una *delay-slot instrunction.* Quando l'esecuzione viene ripristinata proprio dall'istruzione presente nel delay slot, l'informazione sul PC dell'istruzione successiva sarà pari al PC corrente più quattro, indipendentemente dall'esito del salto. Per superare questo problema è stato implementato (nel modulo *branch_break_detector*) un meccanismo che rileva una *delay-slot instrunction* nella fase di Instruction Fetch, evitando la sospensione del processore al ciclo successivo e rinviandola all'istruzione successiva. In realtà, poiché il TISA permette alle Estensioni di interfacciarsi solo con l'Instruction Register (IR) dello stadio ID, viene verificata l'istruzione presente nel IR. Se questa è un'istruzione di branch o di salto incondizionato, viene asserito un segnale che impedisce ad RI di assumere uno stato logico basso in modo che, l'istruzione successiva a quella nel delay slot, non venga considerata un'istruzione dell'Estensione.

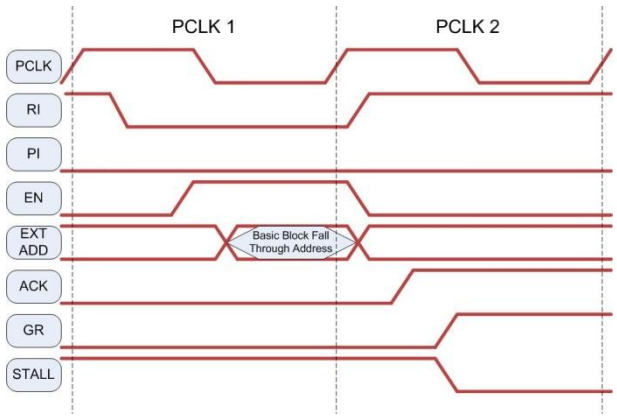


Figura 4.7: Protocollo di Comunicazione con il Pipeline Arbiter

La Figura 4.7 mostra l'handshaking dei segnali tra eBug ed il Pipeline Arbiter. Dopo che l'Estensione abbassa il segnale RI, il Pipeline Arbiter alza in segnale di enable (EN) per indicare all'estensione l'acquisizione dei diritti di accesso alle risorse del processore. Al successivo fronte positivo del clock di sincronizzazione della pipeline (PCLK), l'Estensione deve rilasciare il segnale RI rimettendo il suo valore logico a uno. A questo punto EN viene posto a zero e l'Estensione può richiedere l'accesso

alle risorse di sistema per più cicli di clock (come fa eBug) mettendo ad uno il segnale ACK a cui il Pipeline Arbiter risponde asserendo un segnale di grant (GR). Il processore risulta, quindi, in uno stato di sospensione fino a che l'Estensione non rilasci le risorse abbassando il segnale ACK o il sistema operativo riprenda il controllo (ad esempio per una timeout interrupt) disabilitando l'estensione.
Tutte le operazioni descritte fin qua sono gestite dal modulo *ext_debug_control* che si interfaccia sia al Pipeline Arbiter che al modulo *Top_debug* o, più specificatamente, al suo sottomodulo *main_fsm* come evidenziato in Figura 4.8.
Uno dei compiti più importanti del modulo *main_fsm* è implementare la comunicazione con *emips2gdb*. Sempre facendo riferimento alla Figura 4.8, ogni volta che un'istruzione *break* si trova nello stadio ID, il segnale **break** viene asserito e la *main_fsm* comunica questo evento ad *emips2gdb* per ripristinare la sessione di debugging. In modo simile, quando *emips2gdb* invia un comando di *Suspend*, la *main_fsm* alza il segnale **suspend** per richiedere il controllo delle risorse eMIPS. Una volta che il processore viene stallato, la macchina a stati del modulo *ext_debug_control*, manda ad uno il valore del segnale **suspend_Ack** per avvisare la *main_fsm* che, a sua volta, invierà un byte di acknoledge ad *emips2gdb*.

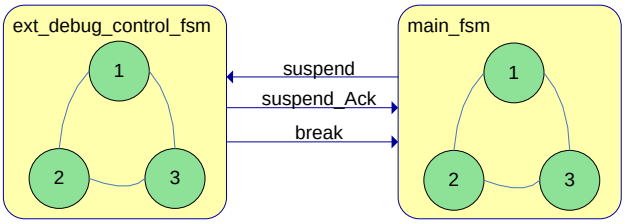


Figura 4.8: Moduli e segnali usati per la sospensione del processore.

Il modulo *ext_debug_control* è composto da tre sottomoduli:

- Una macchina a stati implementata nel modulo *ext_debug_control_fsm*. Nella Figura 4.9 e nella Figura 4.10 è possibile vedere rispettivamente un diagramma semplificato e il diagramma completo della macchia a stati.
- Un modulo (*branch_break_detector*) per decodificare le istruzioni che si trovano nello stadio ID in modo da identificare sia le istruzioni *break*, le istruzioni di branch e di salto incondizionato.
- Un modulo per generare un byte di nAck.

Come precedentemente osservato nella descrizione delle *delay-slot-instructions*, il secondo modulo è responsabile del corretto funzionamento del sistema nel momento in cui una sessione di debugging viene cominciata e rilasciata.

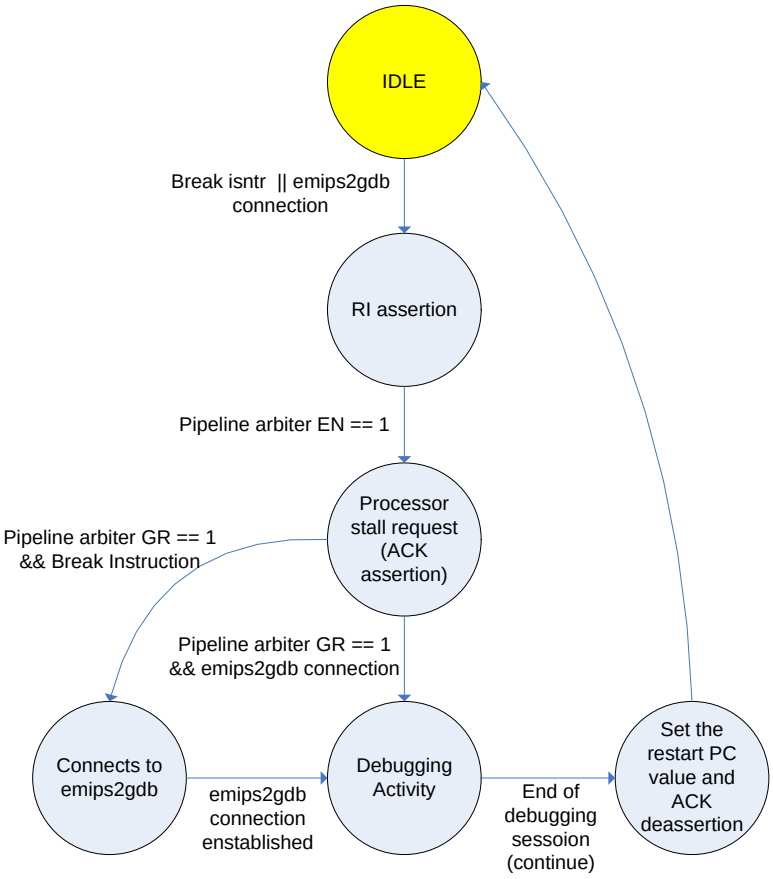


Figura 4.9: ext_debug_control_fsm

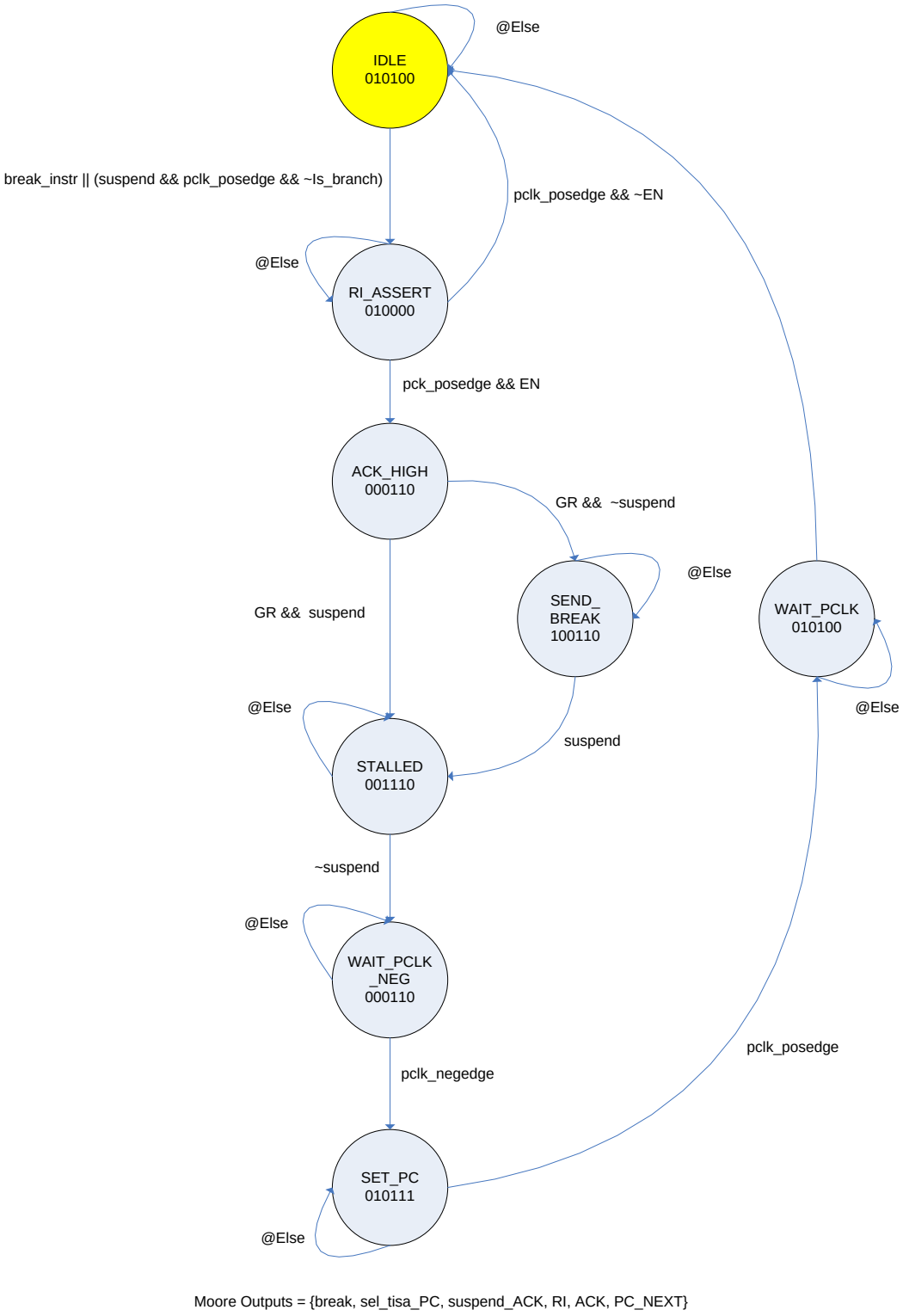


Figura 4.10: Diagramma a stati dettagliato del modulo ext_debug_control_fsm detailed

Il terzo modulo è utilizzato per generare un byte di nACK che è mandato ad *emips2gdb* per rispondere sia ad un comando non supportato che nel caso in cui venga sospeso il processore a causa di un'istruzione di *break*. In Tabella 4.3 si può osservare che vengono usati differenti valori del byte nACK per indicare diversi tipi di istruzioni di *break* eseguibili dal processore eMIPS.

Event	*nACK*
***Command byte* not supported**	**0**
Breakpoint	**0**
Load software module	**1**
Unload software module	**2**
Other break codes	**3**

Tabella 4.3: codifica del byte nACK

4.5.2 Datapath

Come mostrato in Figura 4.6, il modulo *Top_debug* si occupa:

- della comunicazione con il PC host tramite una linea seriale,
- della gestione dell'interfaccia con i registri e la memoria,
- dell'interazione con il modulo *ext_debug_control*.

La struttura interna di *Top_debug* è composta da due moduli: il modulo *uart* e il modulo *debug_core*. Il modulo *uart* è un'implementazione del protocollo di comunicazione seriale RS232 con un baud rate parametrico e modificabile mediante una reimplementazione dell'Estensione eBug. Questo modulo è stato progettato per essere molto semplice e piccolo in termini di area occupata (occupa circa 50 slices di una FPGA Xilinx Virtex4) e quindi non è configurabile via software. Il suo baud rate di default è di 115,200 baud.

Il modulo *debug_core* rappresenta il nucleo di tutta l'Estensione eBug. Il suo datapath è mostrato in Figura 4.11 ed è implementato nel modulo *debug_dp*. La parte superiore del datapath comunica con il modulo *uart* entra la parte inferiore è collegata all'interfacciata TISA-Estensione. Il datapath è stato concepito e sviluppato per minimizzare il footprint in termini di area e, dato che la frequenza di clock massima non era la constraint primaria del progetto avendo nella linea seriale il collo di bottiglia del sistema, non sono stati usati registri di pipeline o altre tecniche per la riduzione dei percorsi critici.

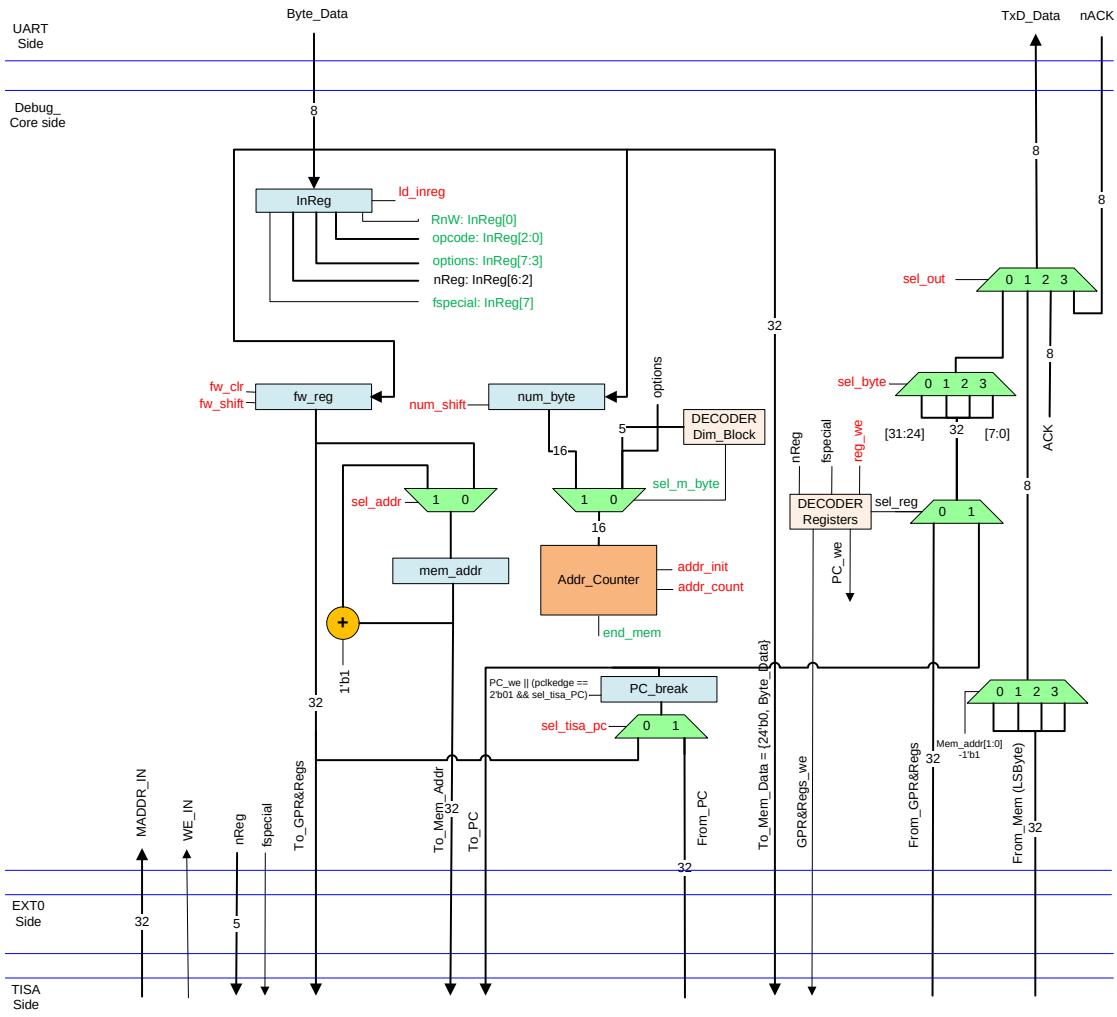


Figura 4.11: Debug_dp module

I registri presenti nel datapath sono cinque:

- *InReg* è usato per memorizzare il *command byte* comunicato dal modulo *uart*.
- *fw_reg* è utilizzato per raggruppare quattro byte in una word da 32 bit. Come evidenziato in precedenza, i bytes sono attesi con ordine big-endian. Questo registro è usato per le operazioni di scrittura nei registri e in memoria.
- *PC_Break* è usato per memorizzare l'indirizzo dell'istruzione che si trova nello stadio di ID. Una volta che una sessione di debugging comincia, *PC_Break* può essere modificato solo dal client debugger. Questo registro è un'immagine del PC del processore e il suo valore viene usato quando il programma viene ripristinato.

- *mem_addr* è il registro in cui è memorizzato l'indirizzo iniziale per un'operazione di accesso alla memoria.
- *num_byte* memorizza il numero di bytes del blocco di memoria a cui accedere.

I multiplexers presenti hanno le seguenti funzioni:

- *sel_addr* e *sel_m_byte* sono usati per le operazioni di accesso in memoria. In particolare il primo fornisce al registro *mem_addr* l'indirizzo iniziale o l'indirizzo successivo a cui accedere. Il secondo inizializza *Addr_counter*, un contatore usato per indicare l'avvenuto trasferimento di un blocco. Il contatore può essere inizializzato dal campo *option* del *command byte* o dal registro *num*_byte in funzione di quanto specificato dallo stesso campo *option* che viene decodificato dal decoder *Dim_Block*.
- *sel_tisa_pc*: questo mux è usato per selezionare il valore con cui il registro PC_Break viene aggiornato. Questo può essere il valore corrente del PC proveniente dal TISA o un valore fornito dal client debugger.
- *sel_reg*: questo mux è pilotato da un decoder che, considerando i campi *fSpecial* e *nReg*, stabilisce se deve passare il valore del registro PC_Break o quello proveniente dai registri del TISA. Il decoder utilizza, inoltre, il segnale *reg*_we proveniente dal controllo, per settare opportunamente i segnali di scrittura dei registri.
- *sel_out*: è il mux di uscita che invia al modulo *uart* il valore da trasmettere.
- *sel_byte*: è usato per serializzare una parola da 32 bit in 4 byte.
- *Mem_Addr[1:0]-1'b1*: Questo mux è usato per accedere ai singoli byte di una word letta dalla memoria, tenendo conto del suo allineamento. Il TISA ha un'interfaccia di memoria che, nel caso di un accesso a singoli byte (come fa eBug), non allinea a destra il valore letto. E' pertanto necessario selezionare il byte in funzione dell'indirizzo letto.

4.5.3 **Il Controllo**

Il controllo del *debug*_core è implementato dal modulo *Debug_control* (
Figura 4.5), utilizzando tre machine a stati che interagiscono tra loro: *main_fsm, register_fsm, memory_fsm*. La macchina a stati implementata da *main_fsm*, di cui è possibile vedere un diagramma semplificato e uno dettagliato rispettivamente in Figura 4.12 e Figura 4.13, gestisce la sincronizzazione tra il modulo *ext_debug_control_fsm* e l'applicazione software *emips2gdb* relativa allo stallo del processore.

Quando la *main_fsm* si trova nello stato IDLE possono accadere solo due possibili eventi:

1. Un'istruzione di *break* è nello stadio ID,
2. L'applicazione *emips2gdb* richiede una connessione al processore.

Nel primo caso il processore viene prima messo in stallo e, in seguito, il debugger viene informato con un opportuno codice di nACK inviato ad *emips2gdb*. Nel secondo caso la *main_fsm* alza il segnale **suspend** per richiedere lo stallo del processore. In entrambi i casi la macchina a stati va nello stato "wait for *emips2gdb* commands" (Figura 4.12). Una volta che un comando proveniente da *emips2gdb* è ricevuto e riconosciuto, ad esempio per un'operazione di accesso ai registri, la macchina a stati esegue l'operazione richiesta per poi tornare in questo stato. Se il comando riconosciuto è *Continue*, al modulo *ext_debug_control_fsm* viene comunicato di rilasciare le risorse del TISA (e quindi di ripristinare l'esecuzione del processore) per poi tornare allo stato IDLE. Se il comando non viene riconosciuto la *main_fsm* risponde con un byte di nACK di valore 0 e si mette in attesa per ricevere un nuovo comando.

Un evento che è stato necessario gestire riguarda il caso in cui l'applicazione *emips2gdb* viene chiusa mentre il processore eMIPS è in stallo e la *main_fsm* è in attesa di un comando. Se l'applicazione viene rieseguita e riconnessa, invierà un nuovo comando di *Suspend* verso eBug che risponderà con un byte ACK per indicare che il processore risulta già in stallo. La sessione di debugging viene quindi correttamente ripristinata.

Le altre due machine a stati che compongono il controllo sono la *registers_fsm* e la *memory_fsm* (Figura 4.14 e Figura 4.15). Questi moduli implementano il protocollo usato dal TISA per accedere ai registri ed al sottosistema di memoria. La *memory_fsm* è più complessa della *register_fsm* poiché le operazioni di accesso in memoria implicano un protocollo di comunicazione con un numero di byte variabile. La macchina a stati, infatti, deve gestire la memorizzazione dell'indirizzo iniziale e della dimensione del blocco ed inoltre, il protocollo usato dall'interfaccia per l'accesso alla memoria. Queste differenti fasi sono evidenziate, con colori diversi, nel diagramma di Figura 4.15. Poiché eMIPS è un progetto ancora in fase di sviluppo, in futuro è probabile che questi protocolli possano essere modificati. In questo caso le macchine a stati *register_fsm* e *memory_fsm* andranno modificate opportunamente. Ad esempio, l'accesso in lettura ad un registro del general purpose register file, impiega quattro cicli del clock di sistema. Questo valore è parametrizzato nella *register_fsm* e se dovesse cambiare in futuro, sarà necessario modificarlo e reimplementare eBug.

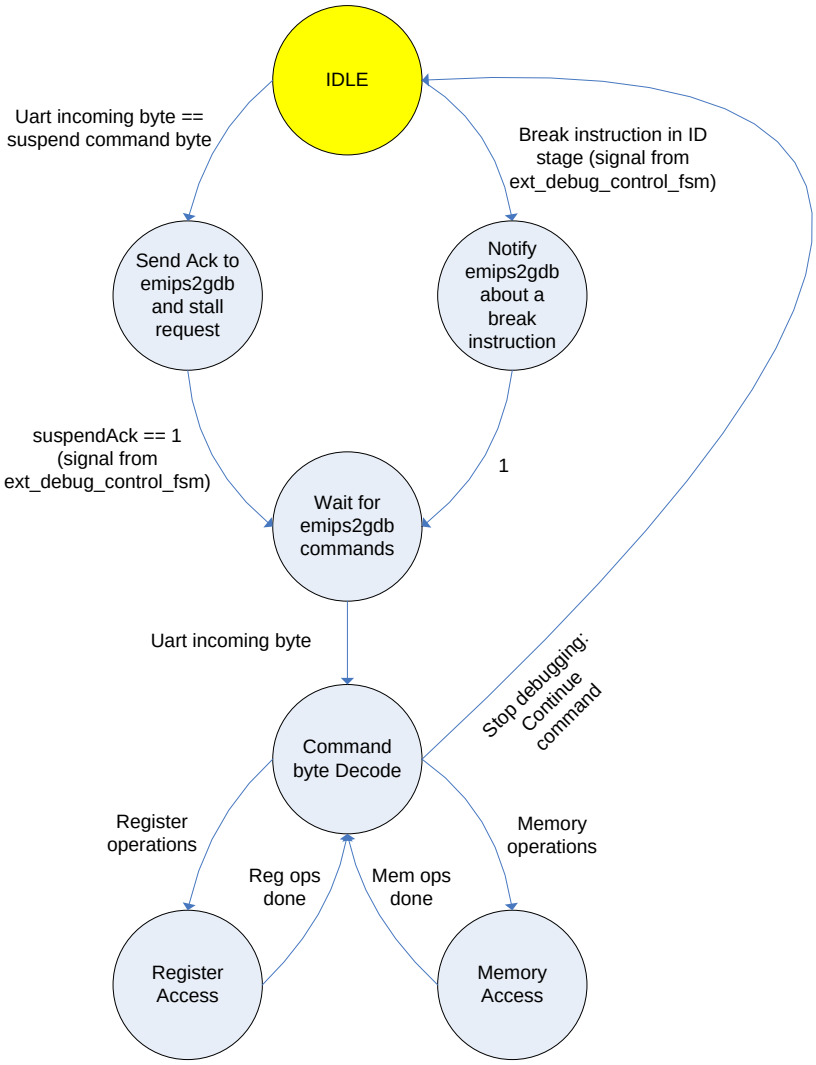
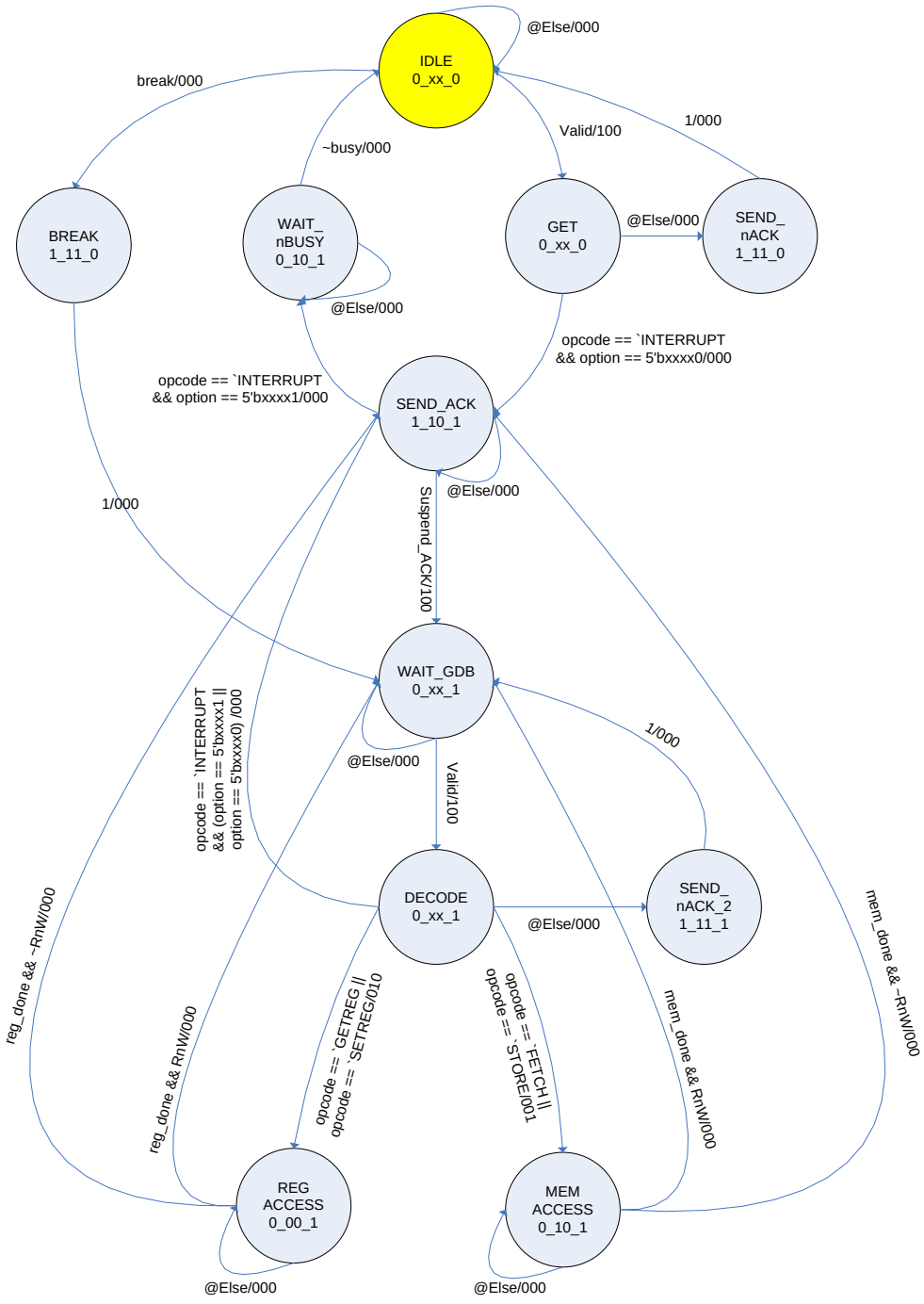
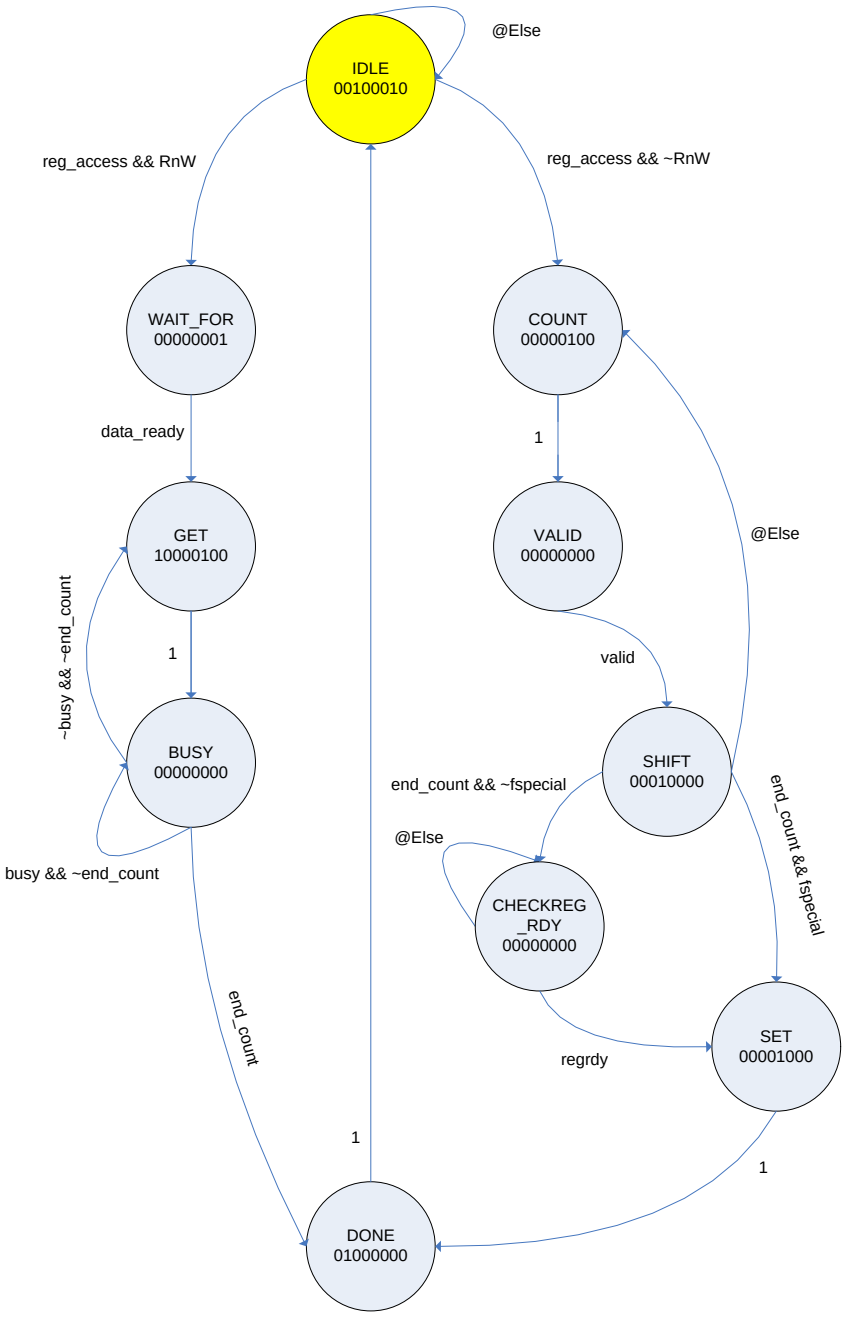


Figura 4.12: main_fsm



Moore Outputs = {TxD_Start_main, sel_out, suspend}
Mealy Outputs = {ld_inreg, reg_access, mem_access}

Figura 4.13: Diagramma completo della main_fsm



Moore Outputs = {TxD_Start_reg, reg_done, fw_clr, fw_shift, reg_we, count_reg, init_latency, count_latency}

Figura 4.14: registers_fsm

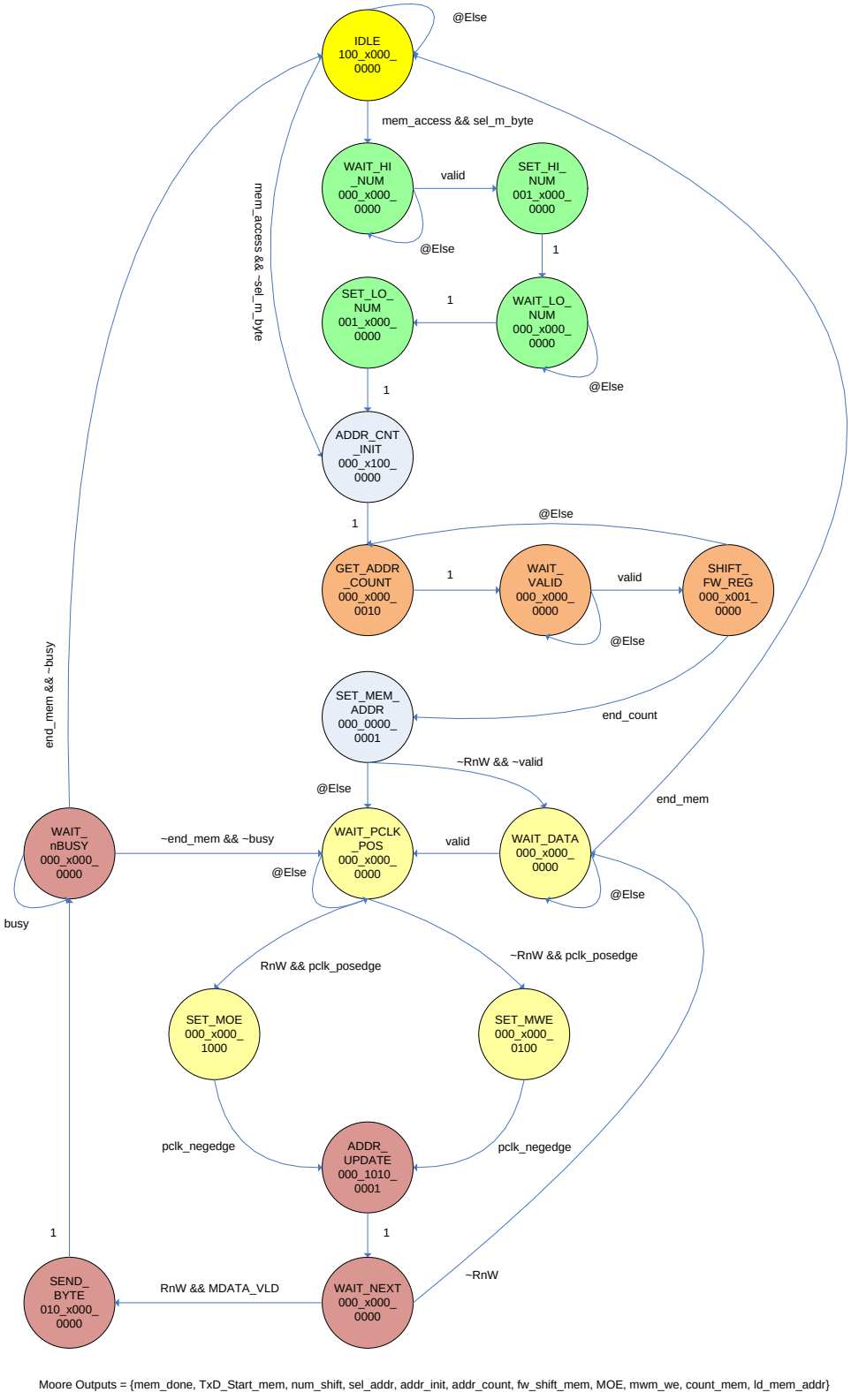


Figura 4.15: memory_fsm

4.6 *Estendere le funzionalità di eBug*

In questo paragrafo sono descritti due esempi relativi all'aggiunta di nuove caratteristiche ad eBug. Il primo sfrutta la struttura modulare del sistema hardware-software di eBug, per aggiungere, in modo semplice, il supporto hardware ai breakpoints e ai watchpoints. Solo un piccolo sottoinsieme dei moduli, chiaramente identificabili, è stato modificato per l'aggiunta delle nuove caratteristiche lasciando inalterata la struttura preesistente. In Figura 4.16 sono evidenziati, in giallo, i moduli che sono stati aggiunti e, con il bordo rosso, quelli sono stati modificati. Si possono inoltre osservare anche delle linee rosse di interconnessione tra i moduli a cui sono state aggiunte delle porte, o delle istanze di sottomoduli, per mantenerli coerenti con il resto delle modifiche.
Il secondo esempio presentato si basa invece sulla connessione di eMIPS con altre estensioni progettate per il processore eMIPS ed in particolare P2V [2]. Utilizzando eBug in cooperazione con P2V è possibile fornire delle facilities per il debugging di alto livello che sono molto utili soprattutto per applicazioni embedded e real-time.

4.6.1 Aggiunta del supporto hardware ai breakpoints e ai watchpoints

Il debugger GDB, come del resto altri client debugger, prevede che il sistema da debuggare possa avere un supporto hardware per rendere più efficiente l'utilizzo dei breakpoints e dei watchpoints. Quando questo support manca, GDB utilizza (o vanno settati alcuni parametri per farlo) i cosiddetti breakpoints e watchpoints software. I Breakpoint software vengono implementati tramite l'uso dei comandi di step. In sostanza il programma è fatto avanzare istruzione per istruzione e, ad ogni passo, si verifica che il PC abbia raggiunto il valore relativo ad uno dei breakpoints che sono stati impostati dall'utente. Se questo avviene, allora il programma è sospeso. I watchpoints software funzionano in modo simile ma, l'informazione che il debugger controlla passo per passo è il contenuto della locazione di memoria corrispondente alla variabile osservata. Per una serie di motivi legati a come GDB gestisce i simboli, il funzionamento dei watchpoints richiede un elevato numero di accessi alla memoria da parte del debugger. Per questi motivi si è ritenuto opportuno considerare come primo esempio di estensione di eBug, l'aggiunta del supporto hardware ai breakpoints ed ai watchpoints.

Come primo passo è stato analizzato il protocollo utilizzato da GDB nelle due differenti situazioni e si è osservato che GDB usa comandi differenti per impostare i breakpoints (e i watchpoints) rispetto al fatto che siano hardware o software. Per supportare questi nuovi comandi del protocollo remoto di GDB, l'applicazione *emips2gdb* è stata modificata, in particolare, nella classe che implementa il server GDB. E' stato inoltre necessario estendere anche il protocollo di comunicazione tra *emips2gdb* e l'Estensione eBug. Per questo motivo, è stato aggiunto un nuovo valore possibile per il campo *opcode* (111) del *command byte* che indicasse un nuovo set di comandi possibili specificabili dal contenuto del campo *option* (00001 nel caso in esame). Per la gestione dei breakpoints e watchpoints è necessaria un'ulteriore quantità di informazioni, per cui è stata prevista la trasmissione di un ulteriore byte chiamato Control Byte le cui informazioni sono codificate come specificato nella Tabella 4.4.

Bits	***Significato***
3-0	Slot number
4	Watchpoint (1) or Breakpoint (0)
5	Enable(1) or Disable (0)
7-6	Access (00-write, 01-read, 11-all)

Tabella 4.4: ControlByte

I quattro bits meno significativi indicano lo slot hardware dedicato al breakpoint (watchpoint) al quale si vuole accedere. In questa implementazione sono consentiti un massimo di 16 slots. Va osservato che quando GDB inserisce o elimina un breakpoint (o watchpoint), lo identifica solo con l'indirizzo della variabile osservata e non con un ID numerico. Se quest'informazione fosse mandata direttamente all'hardware, servirebbe una logica complessa per identificare lo slot hardware dedicato a quel particolare breakpoint (watchpoint). Per evitare questa situazione, *emips2gdb* è stato dotato di una semplice struttura dati che tiene traccia degli indirizzi e di tutte le altre informazione relative agli slots hardware e che permette di tradurre l'indirizzo fornito da GDB nel numero dello slot associato. Alla prima connessione *emips2gdb* sincronizza la sua struttura dati con le informazioni presenti negli slot.
Il bit quattro indica l'intenzione di operare su un watchpoint o su un breakpoint e, il bit 5, se si desidera abilitarlo o disabilitarlo. Gli ultimi due bit sono usati solo nel caso del watchpoint per indicarne il tipo. Un watchpoint, infatti, può controllare che l'accesso alla variabile osservata avvenga in lettura, scrittura o entrambe. Nel caso in cui si disabilita uno slot, solo il bit 5 è significativo mentre, quando si procede con un'abilitazione, il ControlByte è seguito da altri 4 byte che contengono l'indirizzo da

osservare. Questo indirizzo andrà confrontato con il PC nel caso di un breakpoint e con l'address bus nel caso di un watchpoint.

Per implementare queste funzionalità il datapath originale è stato modificato come mostrato in Figura 4.17. La zona evidenziata in giallo è un'istanza del modulo *wbpoints_dp* e si può notare che aggiunge delle nuove porte e dei segnali di controllo al datapath. Questo modulo è composto da un registro di controllo (CR), un decoder e, in funzione del numero di slots che si vuole implementare, una o più istanze del modulo *wp_bel*. Il registro CR è usato per memorizzare il ControlByte inviato da *emips2gdb*. Il decoder instrada i segnali di controllo verso lo slot specificato nel registro CR.

Il modulo *wp_bel* è il modulo base che implementa le funzionalità di un breakpoint o di un watchpoint. La sua struttura logica è mostrata in Figura 4.18. Si possono osservare al suo interno quattro registri:

- *Wp_reg* è usato per memorizzare l'indirizzo che deve essere monitorato.
- *En_reg* è usato come enable globale. Se contiene uno zero logico, il segnale che indica il mathcing degli indirizzi sarà sempre forzato a zero.
- *Sel_addr_reg* contiene l'informazione relativa all'uso dello slot (watchpoint or breakpoint); questo valore logico è usato per decidere se osservare il bus indirizzi o il PC. E' anche usata dalla logica di hit.
- *Wp_type_reg* è significativo solo se lo slot è usato per un watchpoint perché ne memorizza il tipo (read, write and read/write).

I blocchi *watchpoint_enable_logic* e *breakpoint_enable_logic* sono usati per abilitare o disabilitare un address match. In entrambi i moduli i segnali di ingresso *en_reg* e *sel_addr_reg* si comportano come segnali di enable. Inoltre, nel modulo relativo al watchpoint, viene usato il dato *wp_type_reg* e controllato il segnale *write_enable*. In questo modo si può avere un hit per il watchpoint solo se il tipo di accesso in memoria è quello che si desiderava osservare. Quando c'è un hit dell'indirizzo osservato e una delle logiche di enable ha l'uscita alta, allora, uno tra segnali *bp_hit* e *wp_hit* va alto, richiedendo che venga iniziato l'handshaking per lo stallo del processore.

Per quello che riguarda il controllo, si sono dovuti fare dei piccoli cambiamenti ai moduli *main_fsm* e *ext_debug_control_fsm* ed è stata aggiunta una nuova macchina a stati. Come è mostrato in Figura 4.20, è stato aggiunto uno stato alla *main_fsm* per decodificare il nuovo comando inviato da *emips2gdb*. La Figura 4.21 mostra le modifiche apportate a *ext_debug_control_fsm* in cui, l'evento che porta dallo stato IDLE allo stato RI_ASSERT, è stato modificato per considerare i segnali *bp_hit* and *wp_hit*.

La macchina a stati mostrata in Figura 4.19 è quella implementata nel nuovo modulo *wbpoints_fsm* ed è usata per gestire l'effettivo inserimento, o cancellazione, di un breakpoint o un watchpoint.

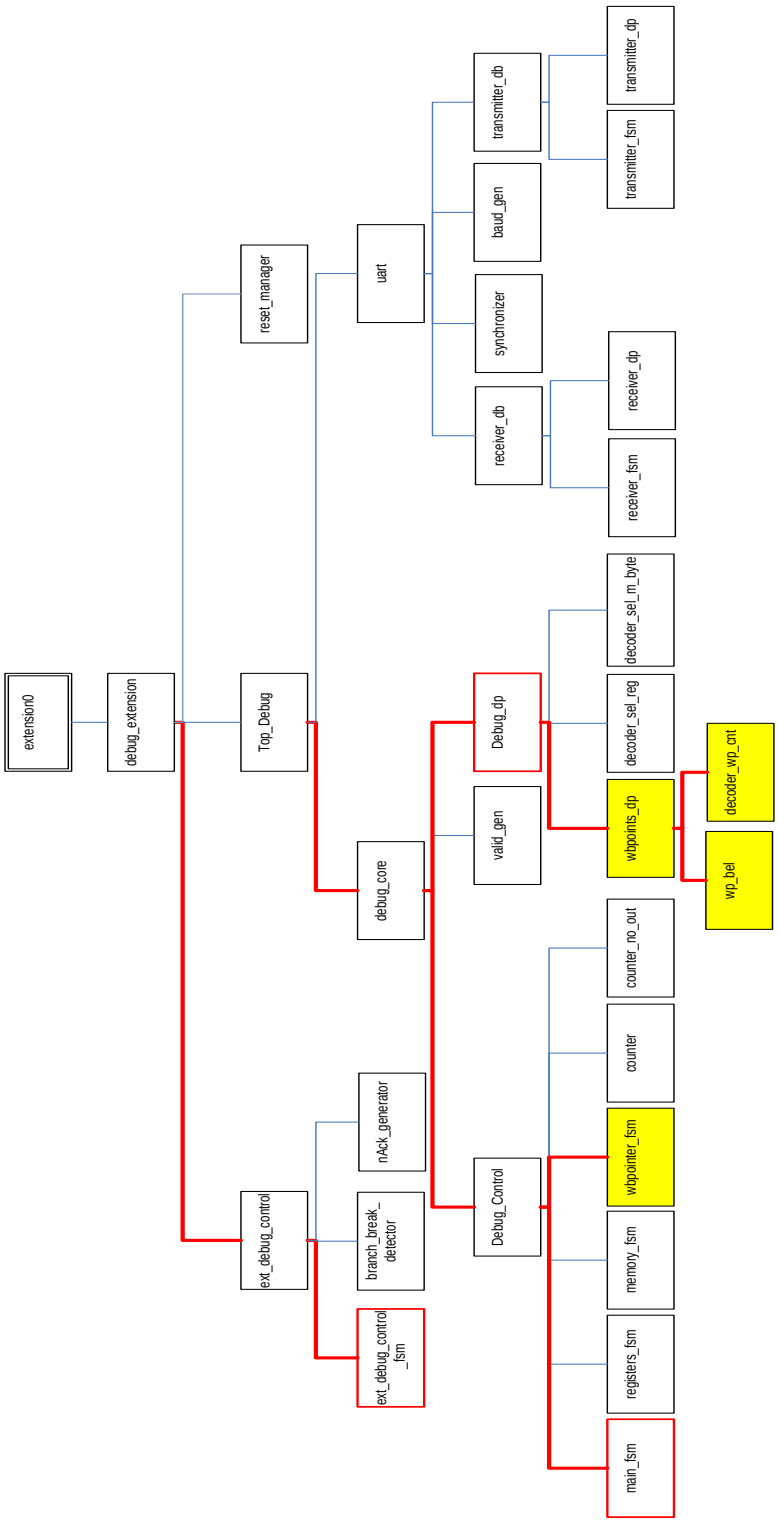


Figura 4.16: Struttura dei moduli dopo l'aggiunta del support hardware ai watchpoint

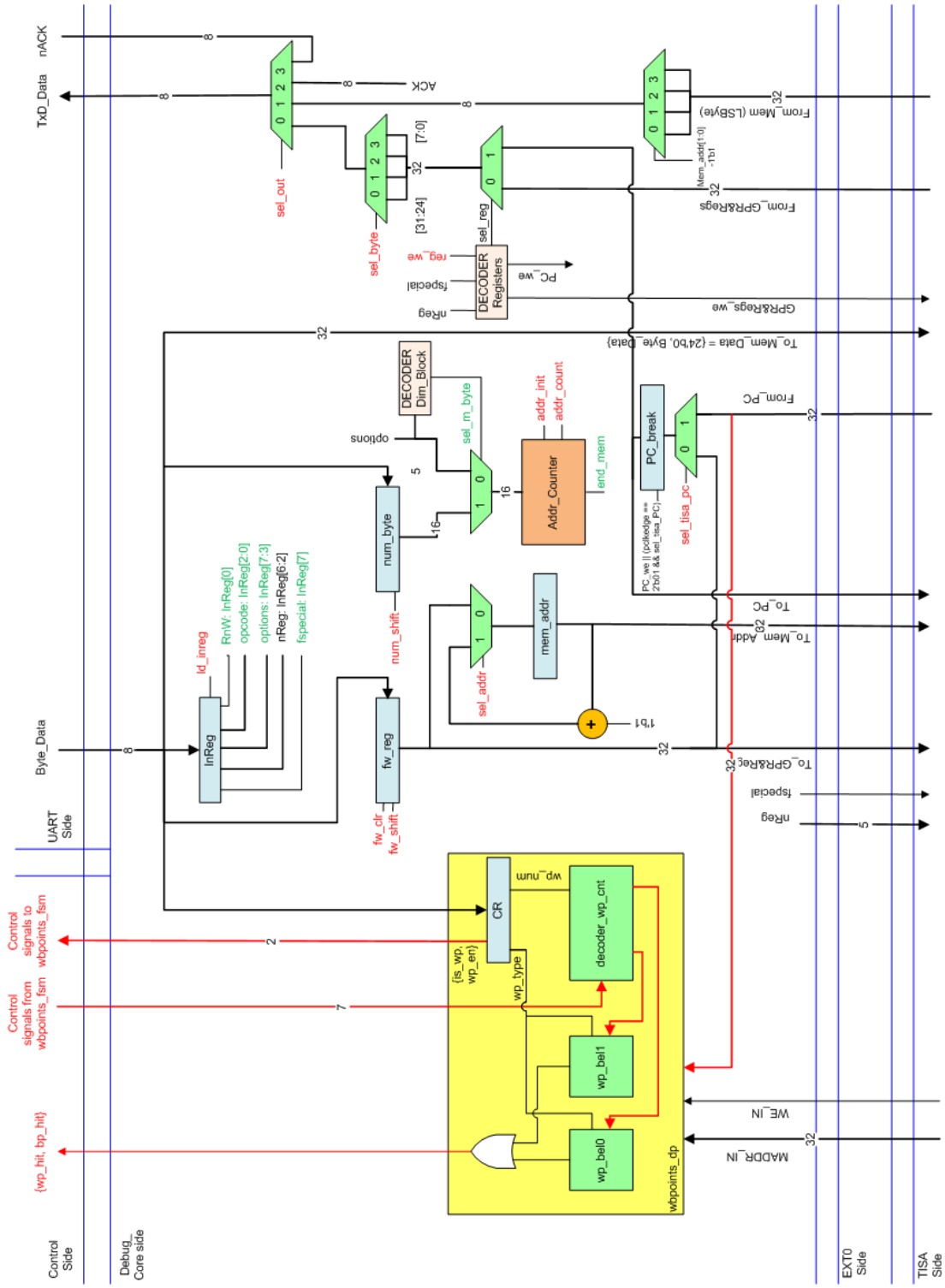


Figura 4.17: Modulo Debug_dp con il supporto ai breakpoints e ai watchpoints

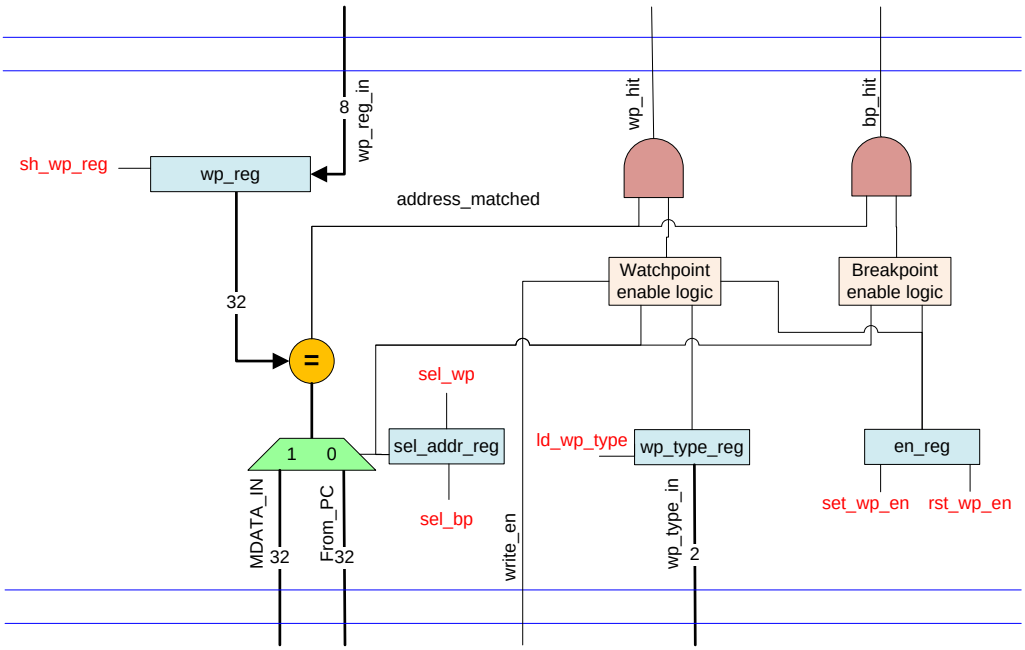
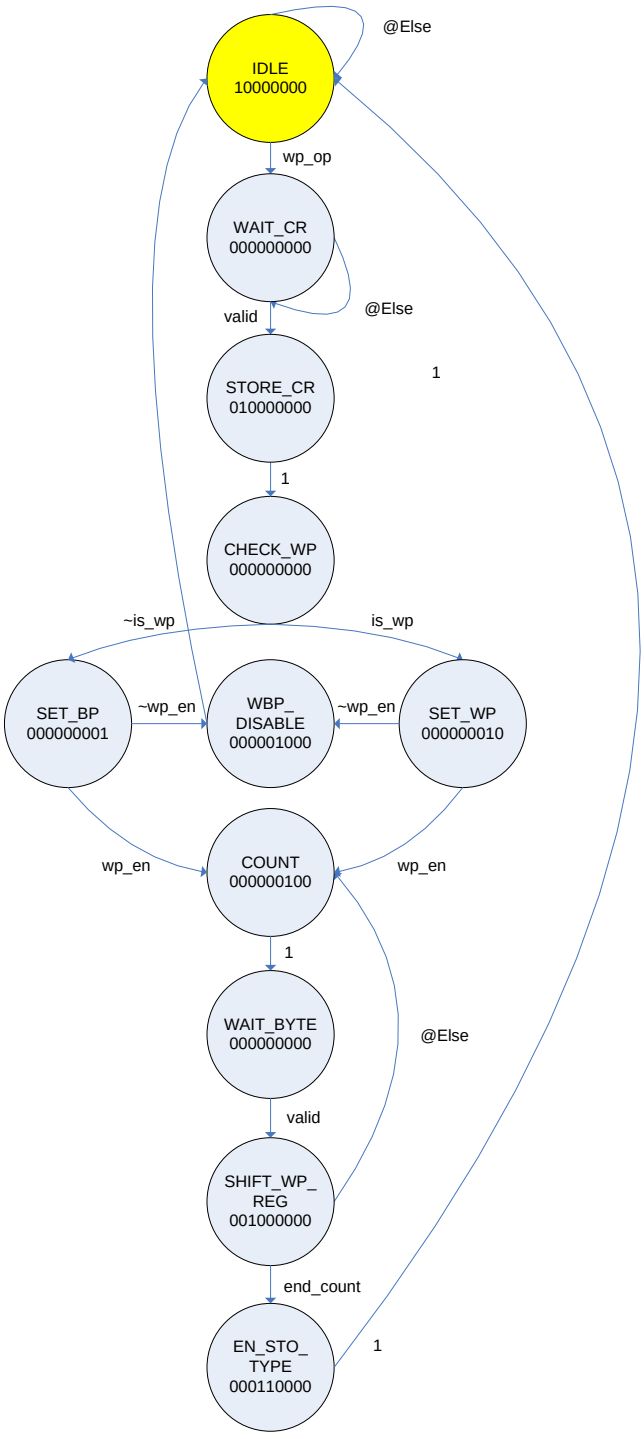


Figura 4.18: Modulo wp_bel



Moore Outputs = {wp_done, ld_CR, sh_wp_reg, ld_wp_type, set_wp_type, rst_wp_type, count_up, is_wp_on, is_wp_off}

Figura 4.19: wbpoints_fsm

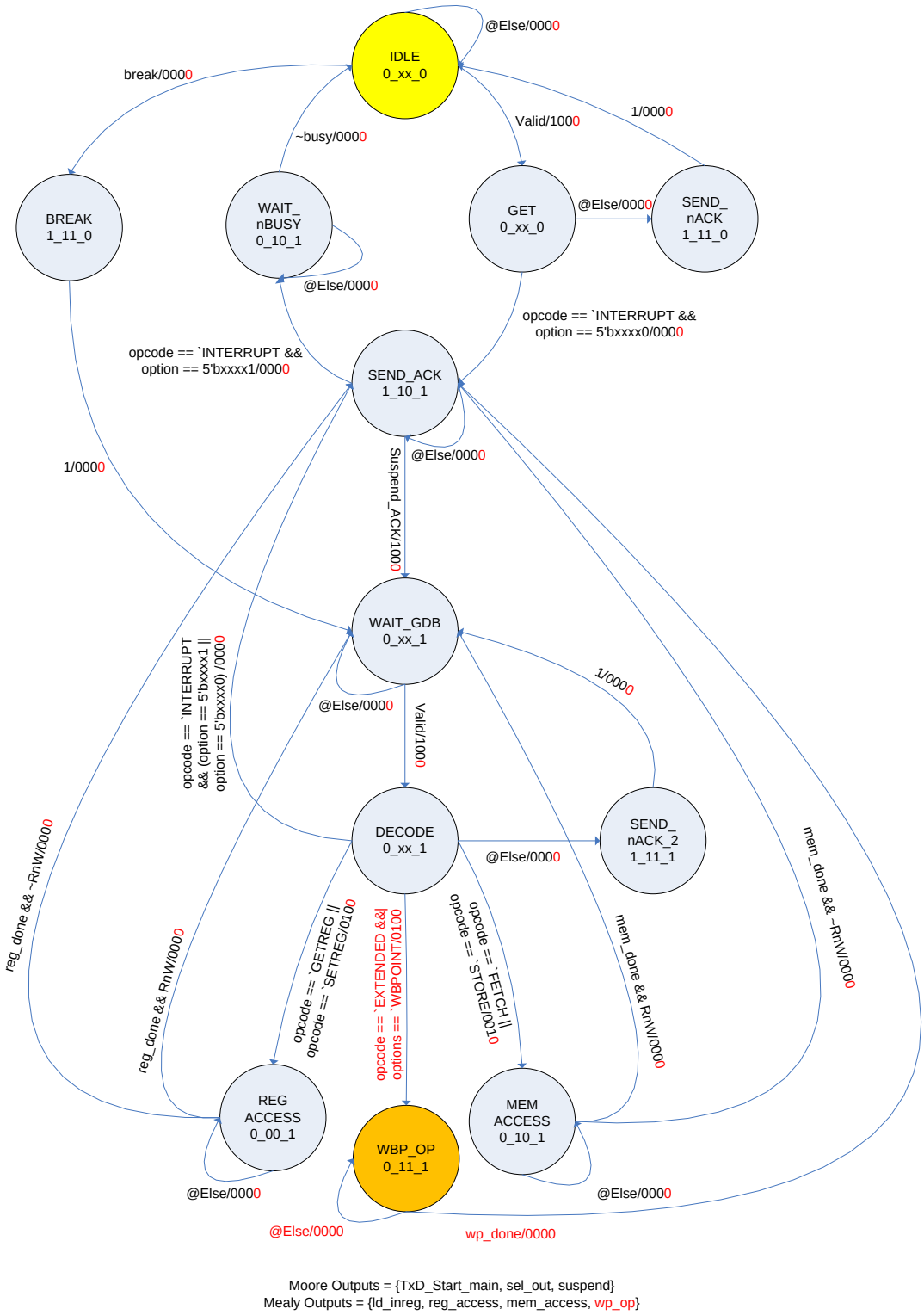


Figura 4.20: main_fsm modified for watchpoint support

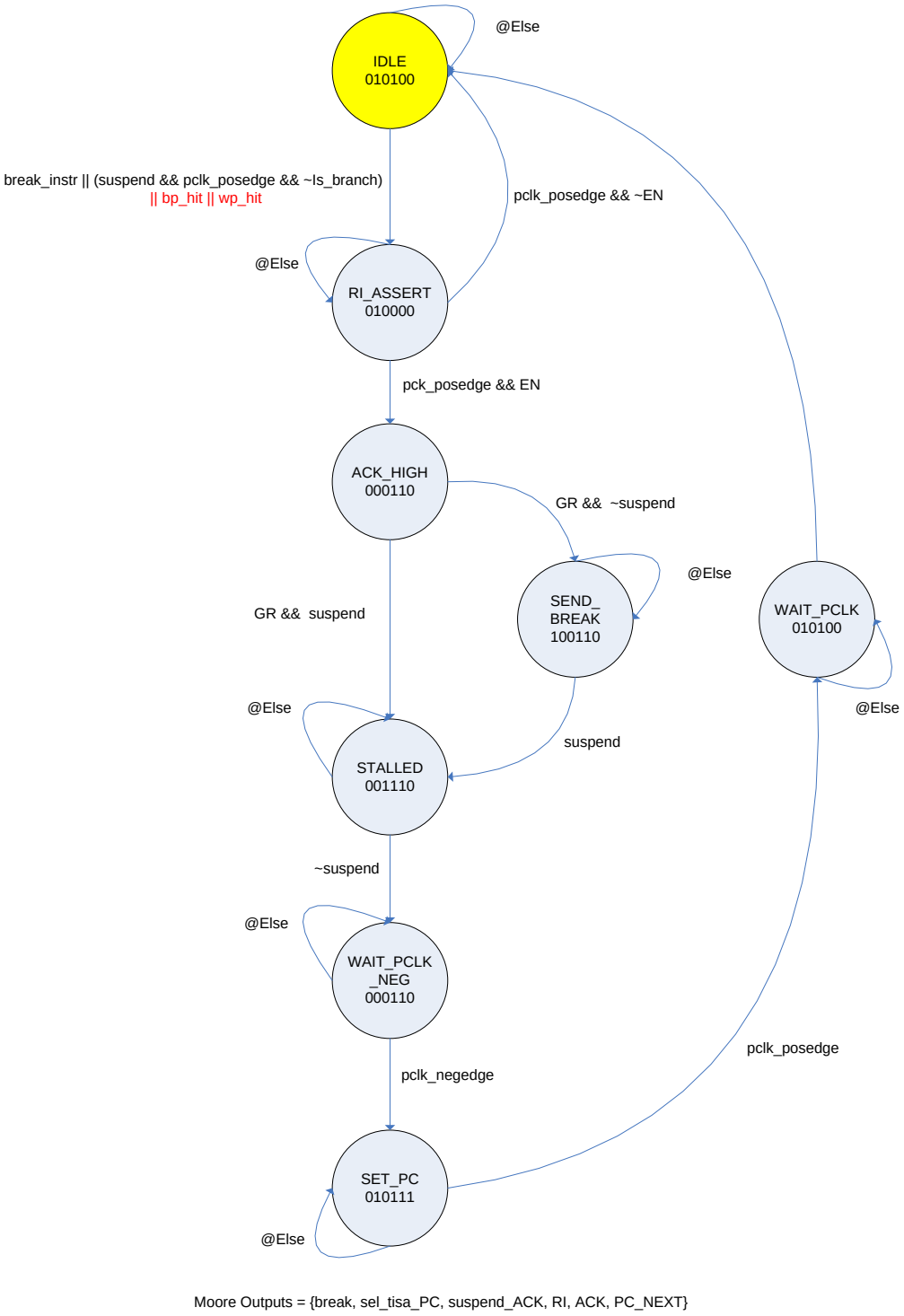


Figura 4.21: ext_debug_control_fsm modified for watchpoints support

4.6.2 Estendere le funzionalità di eBug mediante l'interazione con altre Estensioni eMIPS

Le funzionalità di eBug possono essere estese mediante l'utilizzo di altre Estensioni sviluppate per il processore eMIPS. Un esempio sono le Estensioni generate dal compilatore PSL-toVerilog (P2V). Il P2V è in grado di tradurre un insieme di asserzioni relative ad una parte di un programma software, espresse tramite un sottoinsieme del Property Specification Language (PSL), in Estensioni eMIPS che osservano l'esecuzione del programma e validano le asserzioni. Il PSL è un linguaggio che permette di definire la correttezza del comportamento del programma software in modo naturale e compatto.

A titolo d'esempio, si consideri di voler controllare che il contenuto di una variabile si mantenga sempre all'interno di uno specifico intervallo senza però dover ricompilare o modificare il comportamento temporale del programma (ad esempio mandando sullo standard output la variabile). Al momento P2V è l'unico sistema che permette di farlo creando automaticamente delle Estensioni eMIPS che controllano passivamente l'esecuzione del programma. Nel caso considerato, P2V controllerà senza alcun overhead il valore della variabile osservata segnalando il caso in cui assuma un valore non appartenente all'intervallo specificato. Ad esempio, può essere asserita una trap che viene poi gestita opportunamente dal sistema operativo.

Esistono comunque due limitazioni utilizzando questo approccio. La prima è che non è possibile osservare lo stato del processo nell'esatto momento in cui l'asserzione è violata, poiché viene immediatamente eseguito il codice dell'handler del sistema operativo. La seconda dipende dal fatto che non si è in grado di capire il motivo per cui l'asserzione è stata violata. In sostanza si può fare verification ma non debugging.

Queste due limitazioni possono essere facilmente superate con l'utilizzo coordinato di eBug. Invece che asserire una trap P2V può inserire un'istruzione *break* nel registro di pipeline dello stadio ID dato che l'interfaccia tra il TISA e le Estensioni permette questo tipo di accesso. Questo produrrebbe sempre una trap per il sistema operativo nel caso che eBug non sia presente. Se, viceversa, eBug è presente e abilitato per il processo corrente, prende il controllo del processore esattamente nel momento in cui viene violata l'asserzione e senza condizionare in alcun modo lo stato del sistema. La violazione viene riportata al debugger e l'utente può quindi indagare sulle cause dell'errato comportamento.

La collaborazione tra eBug e P2V può essere ancora più stretta. P2V è implementato in Pyton, utilizzando un interprete. Questo permette di collegare l'interprete della command line di GDB con l'interprete Pyton e creare delle Estensioni P2V mentre si sta debuggando il programma. In sostanza l'utente scrive delle asserzioni PSL relative al comportamento del programma sotto osservazione quando questo è sospeso. Viene quindi creata una nuova Estensione e inserita in uno slot riconfigurabile separato per poi ripristinare l'esecuzione del processo.
Vi è, inoltre, un interessante effetto collaterale in questo approccio dato che l'utente può creare un nuovo insieme di dichiarazioni formali, relative al comportamento del programma, come risultato naturale dell'attività di debugging. Questo permette di quantificare l'entità effettiva dei test eseguiti e di creare dei dati da usare come input per strumenti di analisi più sofisticati.

4.7 *Risultati*

In questo paragrafo le prestazioni di eBug sono misurate in due modi differenti. In entrambi i casi verrà fatto il confronto tra il sistema base e quello modificato con l'aggiunta del supporto hardware ai breakpoints e ai watchpoints. Nel primo sottoparagrafo è mostrato il risultato delle sintesi logiche tra le diverse implementazioni dell'Estensione eBug. Questa misura permette di quantificare l'impatto dell'aggiunta di nuove caratteristiche da un punto di vista hardware. In seguito è misurata la variazione, in termini di tempo di risposta, dal punto di vista dell'utente, sempre nel caso in cui vengano aggiunte le nuove facilities.

4.7.1 **Risultati delle Sintesi Logiche**

Tutte le implementazioni sono state eseguite usando una scheda di prototipazione Xilinx ML401 basata sul dispositivo Xilinx Virtex4 ed in particolare sul modello XC4VLX25-10ff68. Per eseguire le sintesi, l'implementazione e la generazione del file di configurazione della FPGA, è stato usato il tool Xilinx ISE 8.2.01i con l'aggiunta dell'overlay per la PR.
La Tabella 4.5 e la Tabella 4.6 mostrano i risultati delle sintesi logiche. La prima riga di entrambe le tabelle corrisponde all'implementazione base dell'Estensione eBug, mentre le successive righe, corrispondono rispettivamente alle implementazioni con due, quattro e otto slot disponibili per supportare i breakpoints ed i watchpoints hardware. La Tabella 4.5 riguarda i risultati ottenuti in termini di area (slices

utilizzate) e di massima frequenza di clock di funzionamento in funzione di due differenti criteri di ottimizzazione usati per le sintesi. Quando l'ottimizzazione avviene ricercando la minore occupazione d'area, la frequenza massima del dispositivo decresce in modo evidente con l'aumentare del numero degli slots senza però fornire un equivalente miglioramento in termini di slices usate. La Tabella 4.6 evidenzia quest'aspetto mostrando le percentuali di risparmio d'area e riduzione di frequenza nel caso in cui si scelga l'ottimizzazione per area.

	Area optimization		***Speed optimization***	
	Area	**f(MHz)**	**Area**	**f(MHz)**
SW WP	**273**	**112,96**	**316**	**175,04**
2 HW WP	**359**	**88,51**	**381**	**175,00**
4 HW WP	**422**	**89,70**	**451**	**174,93**
8 HW WP	**568**	**61,13**	**603**	**174,61**

Tabella 4.5: Risultati delle sintesi logiche

Si può osservare che il miglior tradeoff è ottenuto con l'ottimizzazione rispetto alla frequenza poiché si ha una piccola penalizzazione in termini di area ma una frequenza di lavoro che si mantiene pressoché costante. Questo era un risultato atteso dato che l'Estensione è stata sviluppata per essere "piccola" indipendentemente dalle ottimizzazioni ottenibili con i tool di sintesi.

	% Area Savings	***% Freq. Reduction***
SW WP	**13.6**	**35.47**
2 HW WP	**5.77**	**49.42**
4 HW WP	**6.43**	**48.72**
8 HW WP	**5.80**	**64.99**

Tabella 4.6: Trade-off tra l'ottimizzazione per Area e per Frequenza

Nell'implementazione del sistema eMIPS usata, lo spazio del dispositivo FPGA dedicato ad un'Estensione è di circa 1300 slices. Estrapolando l'andamento mostrato in Tabella 4.5, si può stimare che un'implementazione di eBug può supportare un massimo di 27 watchpoints hardware. Quando questi non vengono usati eBug occupa solo il 21% dello spazio disponibile lasciando circa l'80% a disposizione per altri usi come ad esempio i moduli generati dalle asserzioni compilate con il P2V.

4.7.2 Tempi di risposta

Come seconda misura delle prestazioni di eBug, è stato rilevato il tempo di risposta nell'utilizzo del client debugger nel caso dell'utilizzo dei watchpoints con o senza il supporto hardware. L'obiettivo era quantificare l'impatto dell'aggiunta della nuova funzionalità dal punto di vista dell'utente che vuole usare eBug per debuggare le applicazioni in esecuzione su eMIPS. Il test è stato eseguito utilizzando un semplice codice scritto in C che esegue un loop infinito all'interno del quale viene incrementata e stampata su console, una variabile *i*. Il programma è stato eseguito su eMIPS e ci si è collegati da remoto con GDB sospendendo il programma ad una iterazione arbitraria. A questo punto è stato inserito un watchpoint per la variabile *i* ed è stato misurato il tempo intercorso tra il comando *continue*, che ha ripristinato l'esecuzione del programma, e la successiva interruzione dovuta all'accesso alla variabile *i*. Le misure sono state ripetute per un certo numero di volte ed il loro valor medio è stato riportato nella Tabella 4.7. C'è stata una solo una leggera varianza nelle misure fatte, il che conferma che i valori in tabella sono ben rappresentativi dell'effettivo tempo di risposta. L'esperimento è stato condotto usando due differenti configurazioni dei sistemi host, per valutare anche l'impatto del sistema su cui girano il client debugger ed *emips2gdb*. La prima configurazione (Setup 1 nella tabella) è costituita da una singola macchina con processore dual-core Intel Centrino Core2/6600 con frequenza di clock di 2.4GHz e con Windows XP SP2 come sistema operativo. Su questa macchina sono stati eseguiti sia GDB che *emips2gdb*. La seconda configurazione include due macchiane collegate su rete LAN. La prima macchina, su cui è stato eseguito GDB, usa due processori Intel Xeon con frequenza di 2.8GHz e sistema operativo Windows Server 2003 SP2. La seconda, su cui è stato eseguito *emips2gdb*, è un PC basato su un vecchio processore Intel Pentium3 con frequenza di 800MHz e con installato Windows 2000 SP4.

	Software	**Hardware**	**Speedup**
Setup 1	**272 sec**	**1,1 sec**	**247**
Setup 2	**44 sec**	**0,4 sec**	**110**

Tabella 4.7: Prestazioni percepite dall'utente.

Dalla Tabella 4.7 è possibile osservare due cose:

1. Un netto incremento nel tempo di risposta, nel caso dell'utilizzo del supporto hardware per i watchpoint. Questa differenza, pur mostrando un valore di speedup notevole (con un fattore minimo di 110), è molto più interessante se

si considerano i valori dei dati in termini di tempo assoluto. Difficilmente, infatti, un utente verrà incoraggiato ad usare uno strumento potente come i watchpoints se deve attendere, tra un accesso e l'altro alla variabile osservata, quasi un minuto.

2. La differenza di prestazioni tra le due configurazioni, dovuta molto probabilmente alla gestione dello scheduling dei processi (soprattutto per il Setup1) da parte del sistema operativo, è evidente sopratutto nel caso dei watchpoints software. Si può quindi affermare che nel caso dell'assenza du un supporto hardware per i watchpoints, non solo l'utente è costretto a dei ritardi di risposta molto elevati, ma anche ad una loro forte variabilità in funzione del sistema host utilizzato.

Capitolo 5

Conclusioni

Questo lavoro è stato motivato dall'idea sviluppare e implementare dei tools utilizzabili dallo sviluppatore software per sfruttare il potenziale offerto dalle architetture riconfigurabili, con particolare riferimento a quelle basate su FPGA. Questi sistemi sono in grado di unire potenza computazionale e flessibilità e sono, quindi, una piattaforma molto indicata per avere, in modo semplice e veloce, un'elevata efficienza in differenti domini applicativi.
Uno dei problemi legati all'uso di questi sistemi, che ha spinto a notevoli sforzi sia la ricerca accademica che l'industria, consiste nel renderli fruibili al programmatore, evitando che debba confrontarsi con la conoscenza dei dettagli architetturali o delle tecniche di progettazione digitale.
In questo lavoro è stata presentata la concezione, lo sviluppo e l'implementazione di due strumenti per supportare il programmatore in due fasi fondamentali dello sviluppo software su sistemi riconfigurabili: la compilazione e il debugging.
Il primo permette di partizionare il codice sorgente di un algoritmo o, in generale di un'applicazione, in modo che possa essere eseguita in parte da una CPU tradizionale e in parte dalla porzione riconfigurabile del sistema rappresentata da una FPGA. Il tool identifica automaticamente i gruppi di operazioni che, implementati ed eseguiti su una FPGA come unità funzionali dedicate, introducono degli speedup significativi. Il sistema sviluppato è stato concepito considerando, ed estendendo, un algoritmo già presentato in ambito accademico.
Il sistema è stato testato con alcuni dei più comuni benchmarks della suite Mediabench (DCT, IDCT, ADPCM), producendo un significativo incremento di prestazioni rispetto all'esecuzione su microprocessore.
I risultati positivi ottenuti inducono a proseguire il lavoro svolto. Un primo passo è quello di valutare lo speedup reale, dato che nei risultati proposti è stato trascurato l'overhead dovuto al trasferimento dei dati dalla CPU alla FPGA e viceversa. Questa decisione, motivata dal fatto che, tramite l'uso di un DMA e di una gestione opportuna delle dipendenze è possibile ridurre al minimo questo contributo, tuttavia richiede un'adeguata analisi quantitativa. Un'ulteriore sviluppo del lavoro, riguarda la comparazione tra le prestazioni e la flessibilità ottenibili tra l'approccio proposto (

che per selezionare le unita funzionali estese analizza il codice sorgente) e quello proposto dai tools che, per eseguire la selezione, considerano i basick block ottenuti dall'analisi sull'eseguibile.

Il secondo tool sviluppato è eBug, un sistema hardware-software per supportare il debugging delle applicazioni eseguite sul microprocessore dinamicamente riconfigurabile eMIPS. eBug è stato sviluppato seguendo fondamentalmente tre obiettivi: riutilizzo dell'area del dispositivo, sicurezza e estendibilità. Il primo obiettivo è stato raggiunto implementando la componente hardware come Estensione del processore eMIPS. In questo modo eBug non è presente nel dispositivo se non è richiesto, consentendo il riuso dell'area per implementare altre funzionalità. Il secondo aspetto, fondamentale in contesti general purpose multi-tasking, ha riguardato la possibilità di tenere l'attività di debugging sotto il controllo del software di sistema impedendo l'interferenza tra i processi in esecuzione e quello soggetto alla fase di testing. L'ultimo obiettivo raggiunto, è stato quello di sviluppare eBug in modo estendibile per poter facilmente incrementare le facilities messe a disposizione dell'utente. A tal fine è stato mostrato il procedimento per aggiungere il supporto hardware ai watchpoints e ai breakpoints evidenziando, nel paragrafo concernente i risultati, l'aumento di occupazione d'area, la variazione in termini di frequenza massima utilizzabile e i tempi di risposta nell'utilizzo dei watchpoints.

Seguendo lo stesso procedimento è possibile estendere ulteriormente le capacità di eBug. In futuro sarebbe interessante implementare i watchpoints value-based che, invece di osservare l'accesso ad un particolare indirizzo di memoria, osservano se un particolare valore viene scritto in una qualsiasi variabile del programma. Ancora più stimolante, potrebbe essere l'uso dei watchpoints a dimensione variabile che consentono di osservare non un particolare indirizzo, ma un intervallo di indirizzi fornendo un potentissimo strumento di debugging. Anche in questo caso sarebbe relativamente semplice modificare la logica per la gestione dei watchpoints utilizzando due watchpoint slots come estremi dell'area di memoria da osservare. Questo permetterebbe di monitorare tipi di dato complessi come gli array e le strutture C o le classi del C++.

I possibili sviluppi di eBug, comunque, non riguardano solo gli aspetti legati al debugging. Sarebbe interessante, ad esempio, analizzare l'impatto sull'Estensione eBug nel caso in cui si volesse implementare la comunicazione con il sistema host mediante interfacce Ethernet o Usb.

Bibliografia

[1]. **Pittman, R., N., Lynch, N., L, Forin, A.** *eMIPS, A Dynamically Extensible Processor.* s.l. : Microsoft Research Technical Report MSR-TR-2006-143, October 2006.

[2]. **Hong Lu, Alessandro Forin.** *P2V: An Architecture for Zero-Overhead Online Verification of Software Programs.* s.l. : Workshop on Application Specific Processors, WASP 2007.

[3]. **Forin, A., Neekzad, B., Lynch, N., L.** *Giano: The Two-Headed Simulator.* s.l. : Microsoft Research Technical Report MSR-TR-2006-130, September 2006.

[4]. **Patterson, David A.** *Reduced Instruction Set Computers.* s.l. : Commun. ACM 28(1): 8-21 , 1985.

[5]. **Intel.** Intel® Streaming SIMD Extensions 4 (SSE4) Instruction Set Innovation. [Online] http://www.intel.com/technology/architecture-silicon/sse4-instructions/index.htm.

[6]. **AMD.** AMD 3DNow! Technology. [Online] http://www.amd.com/us-en/Processors/TechnicalResources/0,,30_182_861_1028,00.html.

[7]. **David Culler, J.P. Singh , Anoop Gupta.** *Parallel Computer Architecture: A Hardware/Software Approach.* s.l. : Morgan Kaufmann Publishers, 1999.

[8]. **R. Razdan, M. D. Smith,.** *High-Performance Microarchitectures with Hardware-Programmable Functional Units* . s.l. : 27th ISM, 1994.

[9]. **J. R. Hauser, J. Wawrzynek.** *Garp: A MIPS Processor with a Reconfigurable Coprocessor.* s.l. : FCM, 1997.

[10]. **Borgatti M., Lertora F., Foret B., Cali L.** *A reconfigurable system featuring dynamically extensible embedded microprocessor, FPGA, and customizable I/O.* s.l. : IEEE Journal of Solid-State Circuits , 2003.

[11]. **Tensilica.** Tensilica: Configurabe and Standard Processor Cores for Soc Design. [Online] http://www.tensilica.com/.

[12]. **T.J. Callahan, J.R. Hauser, J. Wawrzynek.** *The Garp Architecture and C Compiler.* s.l. : IEEE Computer Society Press, 2000.

[13]. **Z. Guo, B. Buyukkurt, W. Najjar, K. Vissers.** *Optimized Generation of Data-Path from C Codes for FPGAs.* s.l. : IEEE Computer Society, 2005.

[14]. **Xilinx.** Virtex4 Family overview. [Online] Xilinx Inc., 2005. http://direct.xilinx.com/bvdocs/publications/ds112.pdf.

[15]. —. Xcell Journal Online- Virtex4 View From the Top - Issue52. [Online] http://www.xilinx.com/publications/xcellonline/xcell_52/xc_v4topview52.htm.

[16]. —. Datasheet - Virtex4 Family Overview. [Online] http://www.xilinx.com/support/documentation/data_sheets/ds112.pdf.

[17]. **Journal, FPGA and Structured ASIC.** Virtex4 - Xilinx Details its Next Generation. [Online] http://www.fpgajournal.com/articles/20040608_virtex4.htm.

[18]. **Xilinx.** HDL Coding Practices to Accelerate Design Performance. [Online] http://www.xilinx.com/support/documentation/white_papers/wp231.pdf.

[19]. —. Designing with DSP48 Blocks Using Precision Synthesis. [Online] http://www.xilinx.com/publications/xcellonline/xcell_54/xc_dsp48-54.htm.

[20]. —. Xilinx Microblaze soft processor core. [Online] http://www.xilinx.com/products/ipcenter/micro_blaze.htm.

[21]. —. PowerPC Processor Solution. [Online] http://www.xilinx.com/products/silicon_solutions/fpgas/virtex/virtex4/capabilities/powerpc.htm.

[22]. **IBM.** CoreConnect Bus Architecture. [Online] http://www-01.ibm.com/chips/techlib/techlib.nsf/productfamilies/CoreConnect_Bus_Architecture.

[23]. **Xilinx.** Development System Reference Guide, Chapter 5, Partial Reconfiguration. [Online] 2005. http://toolbox.xilinx.com/docsan/xilinx8/books/docs/dev/dev.pdf.

[24]. —. Planahead. [Online] http://www.xilinx.com/ise/optional_prod/planahead.htm.

[25]. —. *Two Flow for Partial Reconfiguration: Module Based or Difference Based.* s.l. : Xilinx Inc., 2003.

[26]. —. *Using Partial Reconfiguration to Time Share Device Resources in VirtexII and VirtexII Pro.* s.l. : Xilinx Inc., 2005.

[27]. **Kubilay Atasu, Laura Pozzi, Paolo Ienne.** *Automatic Application-Specific Instrucrion Set Extensions under Microarchitectural Constraints.* Los Angeles : 40th DAC Design Automation Conference, June 2003.

[28]. The Stanford Suif Compiler Group. [Online] http://suif.stanford.edu/.

[29]. Machsuif . [Online] http://www.eecs.harvard.edu/hube/software/software.html.

[30]. **Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman.** *Compilers: principles, techniques and tools.* s.l. : Addison-Wesley, 1986.

[31]. **Keith Cooper, Linda Torczon.** *Engineering a compiler.* s.l. : Morgan kauffmann Publisher, 2003.

[32]. **Mucknick, Steven S.** *Advanced Compiler Design and Implementation.* s.l. : Morgan kauffmann Publisher, 1997.

[33]. **K. Compton, S. Hauck.** *Reconfigurable Computing: A Survey of Systems and Software.* s.l. : submitted to ACM Computing Surveys, 2000.

[34]. **Meier, K., Forin, A.** *MIPS-to-Verilog, Hardware Compilation for the eMIPS Processor.* s.l. : Microsoft Research, WA, September 2007. MSR-TR-2007-128.

[35]. Leon Processor user manual. [Online] http://www.gaisler.com/cms/.

[36]. **architecture, Sparc processor.** [Online] http://www.sparc.org/.

[37]. GDB: The GNU Project Debugger. [Online] http://www.gnu.org/software/gdb/ .

[38]. **Xilinx.** Xilinx Embedded System Tools Reference. [Online] http://www.xilinx.com/ise/embedded/edk91i_docs/est_rm.pdf.

[39]. **FPGAs, PowerPC processor in Xilinx.** [Online] http://www.xilinx.com/.

[40]. **Xilinx.** Xilinx Microblaze Debug Module MDM. [Online] http://www.xilinx.com/bvdocs/ipcenter/data_sheet/opb_mdm.pdf.

[41]. **Hennessy, J. L., Patterson, D.A.** *Computer Organization and Design: The Hardware/Software Interface.* . s.l. : Morgan Kaufmann Publishers, San Francisco, CA. , 1998.

[42]. Microsoft Giano. [Online] at http://research.microsoft.com/downloads/ and http://www.ece.umd.edu/~behnam/giano.html.

[43]. **Sutherland, S.** *The Verilog PLI Handbook, 2nd ed.* . s.l. : Kluwer Academic Publishers, Norwell, MA. , 2002.

[44]. WinDbg multipurpose debugger. [Online] http://www.microsoft.com/whdc/devtools/debugging/default.mspx.

[45]. **Kane, G., Heinrich, J.** *MIPS RISC Architecture.* s.l. : Prentice Hall, Upper Saddle River, NJ, 1992.

[46]. **Dean, J., et al.** *ProfileMe: Hardware Support for Instruction-Level Profiling on Out-Of-Order Processors.* 1997.

[47]. **Graham, S.L., P.B. Kessler and M.K. McKusick.** *gprof: a Call Graph Execution Profiler.* s.l. : SIGPLAN Symp. on Compiler Construction, pp. 120-126, 1982.

[48]. **Pittman, R., N., Forin, A.** *Microsoft eMIPS Release v1.0.* s.l. : Microsoft Research, Fall 2007.

[49]. **Sukhwani, B., Forin, A., Pittman, R. N.** *Extensible On-Chip Peripherals.* s.l. : Microsoft Research Technical Report MSR-TR-2007-120, September 2007.

[50]. **Zagha, M., B. Larson, S. Turner, and M. Itzkowitz.** *Performance Analysis Using the MIPS R10000 Performance Counters.* . s.l. : Supercomputing, November 1996.

[51]. **Zhang, X., et al.** *System Support for automatic Profiling and Optimization.* s.l. : Proceedings of the 16th Symposium on Operating Systems Principles, 1997.

[52]. **Zilles, C.B. and G.S. Sohi.** *A Programmable Co-processor for Profiling.* s.l. : International Symposium on High-Performance Computer Architectures, 2001.

[53]. **Balch, Mark.** *Complete Digital Design: A Comprehensive Guide to Digital Electronics and Computer System Architecture.* s.l. : McGraw-Hill, 2003.

[54]. **Xilinx.** System ACE. [Online] http://www.xilinx.com/products/silicon_solutions/proms/system_ace/index.htm.

[55]. —. Virtex4 User Guide. [Online] http://www.xilinx.com/support/documentation/user_guides/ug070.pdf.

[56]. **P. Athanas, H. Silverman.** *Processor Reconfiguration through Instructions-Set Metamorphosis.* s.l. : Computer Vol.26, 1993.

[57]. **N. Clark, J. Blome, M. Chu, S. Mahalke, S. Biles, K. Flautner.** *An Architecture Framework for Transparent Instruction Set Customization in Embedded Processors.* s.l. : ISCA, 2005.

[58]. **D. Lau, O. Pritchard, P. Molson.** *Automated Generation of Hardware Accelerators with Direct Memory Access from ANSI/ISO Standard C Functions.* s.l. : FCCM, 2006.

[59]. **C. Rowen, D. Maydan.** *Automated Processor Generation for System-on-Chip.* s.l. : ESSCIRC'01, 2001.