Università degli studi di Cagliari
Facoltà di Ingegneria

Corso di Laurea di 1° livello in Ingegneria Elettronica

Presidente Ing. Elio Usai

Configurazione e simulazione comportamentale del processore S1 Core e studio architetturale del processore SUN OpenSPARC T1

Laureando
Emanuele Luzzu
36023

Relatore
Luigi Raffo

Anno Accademico
2009/2010

A

Piera e Carlo,

Davide e Josto.

Ringraziamenti.

Questo lavoro non avrebbe mai potuto prendere forma senza la costante presenza del mio relatore, il Professor Luigi Raffo: a lui rivolgo un sincero ringraziamento.
Ringrazio l'Ingegner Paolo Meloni per il puntiglio e la pazienza con cui ha saputo interpretare le mie difficoltà, incoraggiandomi a dipanare i nodi più intricati e accattivanti del lavoro.
Per i preziosi consigli, che mi hanno consentito di portare avanti interessanti argomentazioni, e per il suo impegno, che ha reso il mio lavoro più sereno, ringrazio l'Ingegner Simone Secchi.

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

INTRODUZIONE

Con la presente tesi, si vogliono proporre i risultati sperimentali che hanno portato ad una parziale configurazione della procedura di boot dell'S1 Core, nella prospettiva di avere una architettura SPARC V9 *compliant* da istanziare su FPGA[1]. Si vuole inoltre fornire una descrizione operativa del *design*, dei *tool* e delle metodologie per la simulazione del processore anche con strumenti opensource. Si intende presentare un metodo per poter simulare l'S1 Core su Modelsim®, piattaforma non supportata dalle distribuzioni ufficiali nè dell'S1 Core, nè dell'OpenSPARC T1, da cui l'S1 Core è ricavato. Nell'ottica di fornire uno strumento che faciliti l'approccio ai test, si intende completare il documento con una breve descrizione dell'OpenSPARC T1, che consenta di avere, in poche pagine, una visione d'insieme sul core e sui legami tra le unità funzionali. A corredo teorico della configurazione della procedura di boot dell'S1 Core, si intende fornire una descrizione funzionale dei registri interni dello SPARC core.

I processori sono le unità funzionali su cui è basato il controllo della tecnologia moderna, dagli elettrodomestici al sistema di controllo elettronico di automobili, centrali energetiche, processi di produzione. E' stato fatto un grosso passo avanti nella conoscenza e nella progettazione dei processori nel momento in cui la SUN Microsystem, ora Oracle, ha deciso di distribuire con licenza GNU GPL v2 i codici sorgenti di una implementazione delle architetture UltraSPARC T1 e T2, completi

[1]FPGA: *Field Programmable Gate Array*. Si tratta di hardware preconfigurato composto di celle elementari la cui conformazione varia da produttore a produttore. La fortuna di questa tecnologia è di avere hardware riprogrammabile che velocizza il processo di prototipazione. Tale tecnologia sta venendo largamente impiegata nel processo di progettazione e test di dispositivi digitali, come ad esempio i processori.

di documentazione e di piattaforme di test e sintesi su FPGA Xilinx. Il mondo dell'opensource si è arricchito di processori di ultima generazione, multicore (8 core) e che implementano multithreading hardware (4 thread per core). E' data la possibilità di avere una istanza di un solo core, a singolo thread tramite una procedura chiamata *riduzione*. L'OpenSPARC, pur essendo il suo codice sorgente pubblico, sviluppa protocolli di comunicazione proprietari e, per sfruttare la piattaforma di test sull'hardware, è necessario essere forniti di tool proprietari non gratuiti. L'S1 Core nasce dalla esigenza di poter istanziare un singolo core SPARC che comunichi tramite protocolli opensource, e con la possibilità di ridurre ulteriormente il design, in vista di una integrazione On-Chip con altri sistemi. La versione rilasciata è stata testata col sistema automatizzato fornito con l'OpenSPARC, ma corredata di script che generano simulazioni e sintesi anche con Icarus Verilog[2]. Nella presente tesi si propongono test quasi tutti eseguiti con Icarus Verilog, non avendo a disposizione tools proprietari richiesti per eseguire i test forniti con l'OpenSPARC.

Il processo che porta alla istanziazione dell'S1 Core su FPGA prevede che siano state preventivamente ottenute simulazioni corrette sul piano della logica. La configurazione del core è risultata incompleta. E' stato pertanto necessario studiare in dettaglio gli script dell'S1 Core sia per ottenere simulazioni *custom*, sia perchè la documentazione è limitata. La ricerca di una configurazione ottimale del core, ha richiesto uno studio minuzioso sui registri interni dello SPARC, sulle interconnessioni delle sue unità ed una indagine approfondita sulle implicazioni imposte dall'hardware ridotto dell'S1 Core. I risultati delle simulazioni sono stati analizzati tramite lo studio diretto delle forme d'onda. Si è proceduto cercando di prelevare dal design relativamente pochi segnali, che fossero significativi per ogni tipologia di simulazione. Per raggiungere alcuni risultati, è stato necessario modificare il codice Verilog[3] dell'S1 Core ed incrementare le memorie RAM e ROM.

I risultati raggiunti verranno esposti e supportati da simulazioni.

[2]Icarus Verilog: simulatore e sintetizzatore opensource.
[3]Verilog: un linguaggio di descrizione dell'hardware (HDL).

Capitolo 2

QUADRO GENERALE

2.1 OpenSPARC T1: hardware opensource

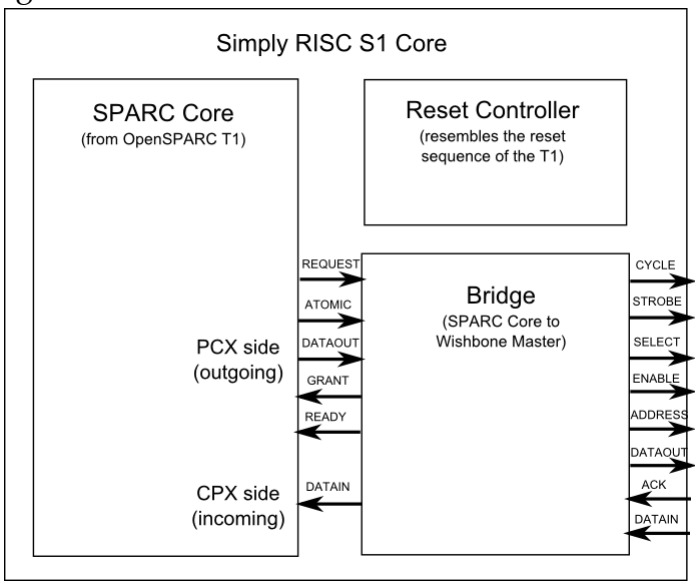
L'OpenSPARC T1 è un processore opensource. E' stato divulgato il codice sorgente, scritto in Verilog HDL, in un package che comprende anche i test, automatizzati tramite *shell script* per Solaris e Linux, le configurazioni per simulazione e sintesi e vari modi per ridurre il processore. La procedura di riduzione consiste nel disabilitare funzionalità e unità del processore completo, 8 cores e 32 hardware thread (*strands*), fino ad avere hardware minimo di un solo core a singolo thread. Inoltre sono automatizzate le sintesi per alcune FPGA Xilinx, prevalentemente Virtex-5. Nella documentazione dell'OpenSPARC T1 è presentata una procedura *step-by-step* per poter sintetizzare due processori OpenSPARC su due FPGA Xilinx e far girare Solaris sulle due FPGA, come se fossero un unico processore dual-core. Le riduzioni sono state pensate per poter usufruire di un sistema che esegua codice SPARC su aree più piccole, con conseguente risparmio di spazio e costi. Grazie a questo risparmio di area, è possibile integrarlo su SoC[1].

2.2 S1 Core – Codename Sirocco

L'S1 Core nasce dalla collaborazione di tesisti e professori dell'Università di Messina e dell' Università di Bristol all'interno di un progetto patrocinato dalla ST Microelectronics. E' figlio di una riduzione a singolo core dell' OpenSPARC T1, e ha la peculiarità di essere connesso ad un bridge con protocollo opensource, il

[1] SoC, System on Chip.

Figura 2.1: S1 Core connesso tramite Wishbone Master



Wishbone, mentre l'interfaccia PCX/CPX dell'OpenSPARC è vincolata da licenza. Esistono tre versioni dell'S1 Core, riportate in tabella 2.1. La figura 2.1 descrive l'S1 Core, evidenziando i moduli principali e i segnali di interfaccia del bridge Wishbone, sia verso lo SPARC core che con l'esterno[2]. Sono stati rilasciati anche i moduli Verilog che interfacciano il core tramite un bus che implementa il protocollo AMBA.

Tabella 2.1: Configurazioni dell'S1 Core

Versione	Descrizione	Virtex5 Area
S1 Core EE	4 threads, 16k+8k L1 caches	∼60K LUTs
S1 Core SE	1 thread, 16k+8k L1 caches	∼40K LUTs
S1 Core ME	1 thread, no L1 caches	∼37K LUTs

2.3 La Tesi

L'idea iniziale è di poter avere un singolo core SPARC che occupi area minima e che comunichi con un bus diverso dal proprio, per poterlo integrare in un SoC

[2]I dati riportati sono relativi alla distribuzione rilasciata sul sito Web dell'S1 Core. http://www.srisc.com/?s1

sintetizzabile su FPGA Virtex5 Xilinx.

La restante parte del SoC è sviluppata con strumenti software della Xilinx e simulata su Modelsim, di Mentor Graphics. E' quindi sembrato logico pianificare simulazioni e sintesi dello SPARC su identici sistemi.

Una specifica non vincolante è quella di poter avere il processore simulabile e sintetizzabile sia su sistema Linux che Windows.

2.3.1 Scelta della piattaforma

OpenSPARC T1: pro e contro. L'OpenSPARC T1 viene fornito con un insieme di script Linux e Solaris di simulazione, riduzione, sintesi e test che si appoggiano su ambienti VCS, NC-Verilog e xst, non è invece supportato Modelsim. Inoltre, se si fosse riuscito ad avere una istanza simulabile e sintetizzabile dell'OpenSPARC T1 ridotto, si sarebbe dovuto progettare l'interfaccia CPU-bus ex novo. L'interfaccia con il singolo core, tagliando fuori dal design la L2-cache, prevede di dover riprodurre anche i gestori di reset, clock, trappola (*trap*), *interrupts* ecc. Nonostante questo, l'OpenSPARC T1 è dotato di un efficiente sistema di testing. Il numero dei test preconfezionati va da 64 per la versione minima a quasi 1000 test per la versione completa. A patto di avere a disposizione un simulatore supportato, lanciando un solo comando si potrà ottenere il design di una riduzione dell'OpenSPARC T1 nonchè il risultato di tutti i test riguardanti la configurazione scelta.

S1 Core: pro e contro. L'S1 Core è un buon punto di partenza alternativo. E' fornito in un pacchetto in cui la riduzione dell'OpenSPARC T1 è già stata effettuata, che implica di poter avere una riduzione del core senza avere a disposizione i simulatori VCS e NC-Verilog; inoltre non si ha bisogno di configurare il sistema OpenSPARC per la simulazione. Per contro, non si ha a disposizione il sistema automatizzato di test fornito nel pacchetto OpenSPARC: questo implica che, a fronte di un malfunzionamento, i test sarebbero da costruire.

Seppure il bus dell'S1 Core non sia quello desiderato, poiché viene compiuta all'interno del pacchetto la riduzione del processore tramite script *bash*, si possono ripercorrere i passi fatti e modificare dove utile. Il bus è accompagnato da un gestore di reset, usato per inizializzare il processore in simulazione, e di un testbench per la simulazione dell'intero S1 Core. Il gestore di interrupt è un priority-encoder 64-6, dunque possono essere gestiti interrupt da al più 63 periferiche.

E' inoltre presente il modulo Verilog di descrizione di RAM e ROM Wishbone-compatibili. Nel pacchetto sono inclusi script per la simulazione automatizzata sia tramite VCS che Icarus Verilog, un simulatore e sintetizzatore opensource. Per la sintesi, sono predisposti script per Icarus Verilog, xst (un sintetizzatore Xilinx) e Design Compiler di Synopsys.

Il Wishbone bus dell'S1 Core non implementa la gestione di tutti i tipi di richieste che può generare lo SPARC core. Supporta infatti transizioni di pacchetti all'interfaccia CPX/PCX (*Processor-to-Cache / Cache-to-Processor Crossbar*) che possono essere solo derivanti da richieste di fetch (*IMISS*), di dati (*LOAD*) e di salvataggio di dati (*STORE*). L'S1 Core supporta singoli cicli di *read/write*. Viene inoltre dichiarato che con versioni di Icarus successive a quella usata per la simulazione, il simulatore va in *crash*. Il pacchetto include una procedura di boot da ROM, scritta in SPARC assembly, e un programma scritto in C che viene inserito in RAM. Il codice-applicazione viene cross-compilato[3] tramite uno script (compile_test). Il core è stato sintetizzato con successo, e, in una versione modificata e non rilasciata, è stato fatto girare Linux sul core istanziato su FPGA.

La scelta. Si è scelto di utilizzare l'S1 Core.

Si è tentata la simulazione del core "as is", ma non è andata a buon fine, in nessuna delle tre versioni. La simulazione è stata effettuata su Ubuntu Linux, con Icarus Verilog 0.9.2-1. Con la versione completa del S1 Core (EE version, 4thread, 16+8K L1 caches), aggiungendo un modulo Verilog mancante in fase di compilazione[4], la simulazione va a buon fine ma non da i risultati sperati. Con le due versioni minime (1 *thread*, con e senza caches L1 rispettivamente), Icarus Verilog va in *crash* durante la simulazione, sebbene la compilazione del codice Verilog vada a buon fine. Nonostante ciò si è riusciti a creare delle simulazioni funzionanti su Modelsim, per tutti e tre le versioni, sia su Ubuntu Linux 10.04 che Microsoft Windows 7. Le simulazioni della versione EE danno risultati identici su Icarus Verilog e su Modelsim, a parità di programmi inseriti in ROM e RAM. Le simulazioni delle versioni ridotte, su Modelsim, variano dalla simulazione della versione EE solo per i parametri che riguardano la gerarchia di memoria. Questo ha fatto

[3]Un *cross-compilatore* è un compilatore che, invece di compilare il codice-applicazione per il sistema su cui è installato, genera codice macchina per un altro sistema. Questo consente, per esempio, di compilare codice per macchine SPARC anche su sistemi Intel o AMD.

[4]Si tratta del modulo '*dffr*', presente nel modulo '*swrvr_clib.v*' dell'OpenSPARC T1, vedi §5.1.1.

supporre che la descrizione dell'Hardware fosse corretta, e che per ottenere una simulazione funzionante del core dovesse essere modificata la procedura di boot fornita di default, la quale presenta delle incongruenze. Si sono riscontrati tre ordini di problemi:

- Gli unici registri dell' Integer Register File che rispondono correttamente sono: un set di registri globali e un set di locali, gli altri non risultano modificabili in scrittura;

- Mentre funzionano correttamente le transizioni sul bus di richieste IMISS (istruzioni), viene generata una eccezione all'esecuzione di una istruzione di LOAD o STORE di dati da RAM. Viene restituito codice di trappola 036_{16}, che corrisponde ad una "mem address not aligned exception". Questo sia che l'istruzione richieda load/store di parole da 8, 16, 32 o 64 bit e sia che venga richiesta in ROM o in RAM. La differenza è che le istruzioni caricate dalla ROM sono sempre *non-cacheable*, e dunque fanno il *bypass* delle cache L1;

- I registri interni, che identificano lo stato del processore (priviledge ed hyperpriviledge register), non sembrano totalmente coerenti con la configurazione hardware. Molti inoltre non vengono inizializzati in fase di boot, pur essendo il loro valore indeterminato.

2.3.2 Strumenti e metodologie

La fase di test dell'S1 Core è inziata con la ricerca di cause e soluzioni dei problemi evidenziati nel precedente paragrafo. Nei capitoli successivi saranno fornite interpretazioni delle cause e presentate soluzioni che condurranno verso problematiche più articolate, radicate nei registri di stato dello SPARC core. Il processo che porta alla istanziazione dell'S1 Core su FPGA prevede che siano state preventivamente ottenute simulazioni corrette sul piano della logica.

La configurazione del core è risultata incompleta, pertanto è stato necessario studiare in dettaglio gli script dell'S1 Core sia per ottenere simulazioni "*custom*" sia perchè la documentazione è limitata. Si è poi deciso di inserire nel presente documento le conoscenze acquisite sulla configurazione del package, con la convinzione che possa essere utile a chi volesse utilizzare il design dell'S1 Core.

La soluzione di tali problemi ha richiesto di effettuare uno studio minuzioso sui registri interni del core, sulle interconnessioni delle unità ed una indagine approfondita sulle implicazioni imposte dall'hardware ridotto dell'S1 Core.

Il percorso verso la soluzione non è stato pertanto lineare, anche perchè si è lavorato con tools non ufficialmente supportati dall'OpenSPARC T1. Non avendo a disposizione un sistema di test automatizzati, si è proceduto tramite verifica diretta delle forme d'onda. Con tale metodologia sarebbe inutile visualizzare in simulazione tutti i segnali del core (ma anche tutti quelli di una sola unità) in quanto sarebbero di difficile lettura, considerando anche che in ogni simulazione vengono eseguite circa 1500 istruzioni, compresi i loop. In buona sostanza, si è proceduto cercando di prelevare dal design relativamente pochi segnali significativi per ogni tipologia di simulazione. Questo modus operandi ha il difetto di dover conoscere a priori cosa aspettarsi dalle simulazioni.

Per raggiungere alcuni risultati, come si vedrà, è stato necessario modificare il codice Verilog dell'S1 Core ed incrementare le memorie RAM e ROM.

Partendo dalle problematiche fin qui illustrate sono stati analizzati alcuni argomenti. I risultati raggiunti verranno esposti e supportati da simulazioni.

2.4 Obiettivi del documento

Gli obiettivi del presente documento sono:

- fornire una rapida guida alla documentazione ufficiale dell' OpenSPARC T1 e dell'S1 Core;
- dare una descrizione funzionale del pacchetto S1 Core, per il quale la documentazione è scarna, nell'ottica della configurazione di test;
- fornire un insieme di passi che portano ad avere il 100% dei moduli dell'S1-Core compilanti su Icarus Verilog e Modelsim;
- simulare l'S1 Core su Icarus Verilog e Modelsim;
- descrivere la procedura di boot e le problematiche di configurazione del core, fornire delle proposte di soluzione ed esporre punti irrisolti;

- riunire alcune delle nozioni sull'OpenSPARC T1 che sembrano più utili per la comprensione del funzionamento del pacchetto S1 Core e dello SPARC core in sè, spesso in riferimento alla procedura di boot.

Capitolo 3

LA DOCUMENTAZIONE DI RIFERIMENTO

Sia il pacchetto OpenSPARC T1 che l'S1 Core includono della documentazione, rispettivamente nelle directory doc e docs, raggiungibili dalla cartella principale. Vi sono inoltre validi documenti scaricabili direttamente dai rispettivi siti web.

3.1 Documentazione OpenSPARC T1

Il package OpenSPARC T1 è reperibile all'indirizzo

http://www.opensparc.net/opensparc-t1/download.html

3.1.1 Documentazione inserita nel package

Parte della documentazione relativa all'OpenSPARC T1 è situata nella directory doc, raggiungibile dal path principale del pacchetto.

OpenSPARCT1_DataSheet.pdf Descrive la struttura esterna dell'OpenSPARC T1, per macroblocchi. Le CPU sono trattate come blocchi, così come le caches L2, i blocchi di RAM, la FPU. Dà una visione generica esterna del processore, con informazioni su caratteristiche di interfaccia dei vari blocchi e bus. Viene ripreso nel capitolo 7.

OpenSPARCT1_DVGuide.pdf Spiega, dopo una breve introduzione, come utilizzare il pacchetto dell'OpenSPARC. Viene descritto come configurare il pacchetto in un sistema Solaris o Linux per poter simulare il processore, fare le riduzioni, le sintesi e i test.

OpenSPARCT1_ExternalInterface.pdf Si occupa principalmente del J-BUS. E' descritto anche il *memory-mapping* (vedi §4.3) del core. Comprende un utile glossario alla fine.

OpenSPARCT1_MegaCell.pdf In questo modulo vengono descritte tutte le memorie che sono istanziate nel design e il loro funzionamento. Viene fornito per ogni memoria il lo schematico dei segnali interni a livello di Flip-Flop e Multiplexer per dare informazioni su traduzione degli indirizzi, lettura e scrittura. Sono descritte le cache L1 istruzioni e dati, le memorie CAM, vari tipi di registri, tra cui l'IRF. La descrizione funzionale è minimale, punta alla comprensione da parte del lettore della configurazione hardware delle memorie: mira al "come è fatto", piuttosto che al "cosa fa".

OpenSPARCT1_Micro_Arch.pdf Descrive minuziosamente il funzionamento di ogni unità del processore OpenSPARC T1. Vengono descritti tutti i moduli interni ed esterni al singolo core. Il target è dare una descrizione funzionale, facendo riferimento a registri interni e valori degli stessi quando utile alla comprensione funzionale.

3.1.2 Documentazione sull'OpenSPARC T1 on-line

Altra documentazione sull'OpenSPARC T1, non inserita nel package, è liberamente scaricabile dal web.

3.1.2.1 3.1.2.1 OpenSPARC T1 main page:

http://www.opensparc.net/opensparc-t1/index.html

3.1.2.2 OpenSPARC Internals Book

http://www.opensparc.net/offers/OpenSPARC_Internals_Book.pdf

Dà una visione globale sull'OpenSPARC, sia T1 che T2. Viene descritto il processore in termini di funzionalità e features. Viene descritta l'architettura OpenSPARC e vengono marcate le differenze tra le implementazioni T1 e T2. Viene spiegato come il processore può essere configurato. Vengono descritte le varie fasi del progetto del processore, dalla scrittura del codice HDL, alla simulazione su FPGA o ASIC, alla fabbricazione del chip. Sono descritti tool e metodi per lo sviluppo di codice su Opensparc, compilazione, test e ambienti di test per il programmatore. E' presente un capitolo dedicato all'S1 Core, che spiega come è stato pensato e costruito il pacchetto dell'S1 Core. Un capitolo è dedicato al design. Il capitolo "Design e Verification Suites" spiega come configurare il pacchetto OpenSPARC per riduzioni, simulazione, test e sintesi dell'Opensparc. Vi è un elenco di risorse consultabili relative all'OpenSPARC. Vi è un utile elenco della terminologia dell'OpenSPARC, con la spiegazione dei termini specifici e l'associazione tra sigle e nomi estesi.

3.1.2.3 Microarchitecture Specifications

http://opensparc-t1.sunsource.net/specs/OpenSPARCT1_Micro_Arch.pdf

Fornisce una approfondita descrizione delle unità dell'OpenSPARC T1. Di particolare interesse nell'ambito della tesi, sono i primi 3 capitoli. Nel primo capitolo si da una descrizione generale dei moduli componenti l'OpenSPARC T1: contiene le informazioni che vengono riprese nel documento "OpenSPARC Internals Book" (3.1.2.2). Nel capitolo "SPARC Core" vengono analizzate funzionalmente tutte le unità interne al singolo core OpenSPARC. E' stato un capitolo estensivamente usato nello svolgimento dei test sull'S1 Core in quanto non solo fornisce una specifica per ogni unità del core, ma si mettono in risalto le modalità di connessione logica tra le unità e come ogni unità interviene sulla *pipeline*; viene anticipato il funzionamento di alcuni dei più importanti registri di configurazione. Il terzo capitolo tratta invece della "CPU-Cache Crossbar" (CCX). Quest'ultima, è sostanzialmente una unità di comunicazione tra ognuno degli 8 core e la cache L2 e l'unità di I/O (FPU, J-Bus). E' con la CCX che "parla" il bus Wishbone dell'S1 Core, ed è in questo capitolo che viene spiegata la conformazione dei pacchetti in ingresso e in uscita dal core. Gli altri capitoli descrivono le restanti unità dell'OpenSPARC T1, esterne alle CPU.

3.1.2.4 UltraSPARC Architecture 2005 Hyperpriviledge

http://opensparc-t1.sunsource.net/specs/UA2005-current-draft-HP-EXT.pdf

Insieme ai due manuali-utente sulle architetture UltraSPARC, http://www.comms.scitech.susx.ac.uk/fft/programming/805-0087.pdf e http://www.comms.scitech.susx.ac.uk/fft/programming/802-7220-02.pdf, fornisce le specifiche dell'ISA (Instruction Set Architecture) implementata dalle architetture UltraSPARC. Vi sono in dettaglio il funzionamento delle istruzioni e il funzionamento dettagliato dei registri di configurazione. E' stato estensivamente usato nel corso della presente tesi, in quanto si è reso necessario dover fare il debug della procedura di boot dell'S1 Core.

Può essere d'aiuto consultare anche il medesimo documento di riferimento per l'OpenSPARC T2: http://opensparc-t2.sunsource.net/specs/UST2-UASuppl-current-draft-HP-EXT.pdf

3.1.2.5 OpenSPARC T1 supplement to UltraSPARC architecture

http://opensparc-t1.sunsource.net/specs/UST1-UASuppl-current-draft-HP-EXT.pdf

Vi sono diverse 'variabili libere' nella implementazione di una architettura. Anche nel caso della architettura UltraSPARC, alcuni indirizzi o alcuni bit dei registri di configurazione è previsto che debbano essere definiti dai progettisti nel caso ve ne sia necessità in fase di implementazione. Da qui, l'utilità di un documento a corredo della "UltraSPARC architecture 2005 Hyperpriviledge", che serve a dichiarare quali 'parametri indefiniti' della architettura UltraSPARC siano stati usati e come. Questo documento è impostato in maniera identica al precedente "UltraSPARC architecture 2005 Hyperpriviledge", ma specifica quali scelte sono state adottate nell'OpenSPARC T1 per implementare le specifiche dichiarate "*architecture-dependent*" nella definizione della architettura UltraSPARC.

3.1.2.6 sun4v specifications

http://opensparc-t1.sunsource.net/specs/Hypervisor-api-current-draft.pdf

Racchiude le specifiche per la virtualizzazione dei sistemi operativi su architetture UltraSPARC. Una feature importante delle architetture UltraSPARC è di gestire via hardware 3 livelli di esecuzione del codice:

- esecuzione a livello utente;

- esecuzione di codice priviledge, dedicato ai sistemi operativi;
- esecuzione a livello hyperpriviledge, un ulteriore layer che consente di eseguire del codice al di fuori dei sistemi operativi.

L'esistenza del livello hyperpriviledge consente la virtualizzazione del processore, eseguire contemporaneamente più sistemi operativi indipendenti e monitorare le risorse affidate all'uno o all'altro. La virtualizzazione consiste nel dedicare ad ogni sistema operativo solo parte dell'hardware a disposizione e parte delle istruzioni macchina (quasi tutte, a dire il vero). Nei primi 7 capitoli vengono definite le risorse delle CPU da configurare per la virtualizzazione ed il loro stato iniziale dopo un *Power-On-Reset*, gli ASI[1] corrispondenti ad alcuni registri di stato, le convenzioni adottate dal software e dalle procedure di gestione degli errori (*traps*) nell'uso dell'Integer Register File (comunemente detti "registri" del processore). Nel seguito vengono descritte le API (*Application Programming Interface*) utilizzabili per configurare il processore in vista della virtualizzazione dei sistemi operativi.

3.1.2.7 Moduli Verilog in versione html

http://www.opensparc.net/pubs/t1/html/verilog.html

Fornisce un utile strumento per la visualizzazione e la ricerca all'interno dei moduli Verilog del progetto OpenSPARC. E' una 'immagine html' dei moduli Verilog del processore. I nomi di tutte le variabili, che siano esse segnali, moduli, parametri o quant'altro, sono dei link, e questo permette un più agevole accesso alle tante variabili presenti e alla più facile comprensione dei collegamenti tra i moduli.

3.2 Documentazione S1 Core

La documentazione dell'S1 Core è interamente inclusa nel package, nella directory docs, accessibile dal path principale. Il pacchetto è scaricabile dal sito

[1] ASI, *Address Space Identifier*. Nelle architetture UltraSPARC, ad ogni istruzione (32 bit) sono associati altri 8 bit che specificano in quale spazio degli indirizzi l'istruzione viene eseguita. Il primo bit specifica se l'istruzione deve essere eseguita in modalità utente, o in modalità privileggiata. E' una informazione talmente rilevante che i registri interi sono da 72 bit: 64 per il dato e 8 per salvare l'informazione sull'ASI associato all'istruzione che ha generato il valore.

http://www.srisc.com/?download

Una copia completa, in formato PDF, dei file della cartella docs dell'S1 Core è presente all'indirizzo

www.srisc.com/download/simplyrisc-s1-0.1.pdf

Nel file README.txt si fornisce una riga di descrizione per ogni file della cartella docs.

Capitolo 4

S1 CORE: DESCRIZIONE OPERATIVA DEL PACKAGE

4.1 Installazione

L'S1 Core richiede di essere installato su un sistema Linux. E' stato configurato sulla distribuzione Ubuntu 10.04. Si supporrà nel seguito che il pacchetto S1 Core sia stato prelevato da

http://www.srisc.com/download/s1_core.tar.gz

e scompattato nella cartella /home/user/s1_core, o equivalentemente in ~/s1_core. Si suppone inoltre che il pacchetto dell'OpenSPARC T1, reperibile all'indirizzo

http://www.opensparc.net/offers/OpenSPARCT1.1.7.tar.bz2

sia scompattato nel path ~/opensparc_t1.

Gli script presenti nel pacchetto sono per shell bash, presente di default in Ubuntu; nei codici e nei riferimenti alla shell, si supporrà che si sia in ambiente bash.

Vi sono tre dipendenze da soddisfare per eseguire correttamente importanti script presenti nel pacchetto: Icarus Verilog, il programma per shell sed e, nel caso non si abbia a disposizione una macchina UltraSPARC, un cross-compilatore.

Icarus Verilog è un compilatore, simulatore e sintetizzatore per Verilog. L'S1 Core è stato testato dagli ideatori su Icarus 0.8, mentre per questa tesi è stata usata la versione 0.9. Con la versione 0.9 è stata stato riscontrato essere funzionante solo la versione del processore EE, dunque con le caches L1 e con 4 threads. Con le due

versioni ridotte (SE, ME) il simulatore va in crash. Per installare l'ultima versione di Icarus, digitare da shell:

```
sudo apt-get install iverilog
```

Il programma sed è disponibile nei repositories ufficiali di Ubuntu, installabile direttamente con apt-get o dal gestore pacchetti Synaptic.

Il cross-compilatore è un compilatore che permette di trasformare il codice-sorgente in codice-macchina di un processore diverso da quello su cui è installato il sistema operativo. In fase di test, c'è bisogno del compilatore per macchine SPARC per generare il codice-macchina da inserire in ROM e in RAM perché venga eseguito nella simulazione. Il codice-sorgente è codice assembly UltraSPARC o codice C, viene dunque usato il cross-compilatore gcc-sparc64, disponibile sul sito http://debian.speedblue.org/.

Per visualizzare il risultato delle simulazioni di Icarus Verilog, può essere utile usare un tool visuale opensource, gtkwave, il cui download è disponibile al link

http://gtkwave.sourceforge.net/gtkwave-3.3.13.tar.gz.

Per la simulazione, si può installare Modelsim. Su Modelsim sono compilabili tutte e tre le versioni del core, oltre a quella completa, anche quella a singolo thread è quella senza caches L1.

Per la sintesi, si può installare ISE, il software Xilinx, scaricabile gratuitamente nella versione WebPack.

4.2 Cartelle e files dell'S1 Core

Viene qui data una descrizione di massima di cartelle e files inclusi nel package dell'S1 Core. Il significato delle sottocartelle e le modalità di utilizzo dei file sarà approfondito quando verrà trattata la cartella s1_core/tools/bin, contenente gli script che maneggiano files e dati del package.

4.2.1 La directory principale

Nella directory principale del pacchetto S1 Core, vi sono due files: il README.txt e lo script sourceme. Inoltre vi sono 5 cartelle.

README.txt Contiene informazioni sulla documentazione del pacchetto.

sourceme E' lo script che configura le variabili d'ambiente che saranno richiamate negli altri script. Per eseguirlo, lanciare da shell

```
usr@name$ source ~/s1_core/sourceme
```

Il comando export crea, o sovrascrive se già presente, una variabile d'ambiente. Bisogna editarlo in modo tale che le le variabili S1_ROOT e T1_ROOT contengano i path dove sono stati scompattati l'S1 Core e l'OpenSPARC T1 rispettivamente. Per esempio:

```
export S1_ROOT=~/s1_core
export T1_ROOT=~/opensparc_t1/
```

D'ora in avanti ci si riferirà al main path dell'S1 Core e dell'OpenSPARC T1 con $S1_ROOT e $T1_ROOT.

E' poi esportata la variabile $PATH, alla quale viene aggiunta la cartella $S1_ROOT/tools/bin, contenente gli eseguibili del package, che diventano così richiamabili in qualsiasi path ci si sia posizionati sulla shell senza dover specificare l'indirizzo. Alla variabile $PATH consiglio di aggiungere anche il path di installazione del cross-compilatore, il quale dovrebbe essere nominato di default con sparc64-linux-gcc, sparc64-linux-as ecc.:

```
export $PATH=.:/$PATH/:$S1_ROOT/tools/bin:/usr/sparc64/bin
```

Le variabili FILELIST_* contengono la lista dei file Verilog del progetto, che saranno poi usati per lanciare simulazioni e sintesi. Nelle tre versioni cambia infatti il numero dei files, in quanto non tutti quelli della versione EE sono necessari nelle due ridotte. Per questo, alla fine della esecuzione del sourceme viene stampato un messaggio che ricorda di aggiornare i FILELIST (tramite lo script update_filelist) quando si fa una riduzione del core (con lo script update_sparccore). Può essere utile inserire l'esecuzione di sourceme nel file ~/.bashrc, per non doverlo eseguire ogni volta che si apre una nuova shell. Per farlo, eseguire una volta sola il comando

```
usr@name$ echo "#" >> ~/.bashrc
usr@name$ echo "# S1 Core env variable conf script" >> ~/.
   bashrc
usr@name$ echo "source ~/s1_core/sourceme" >> ~/.bashrc
```

4.2.2 docs

Contiene la documentazione relativa al package e le licenze.

Di particolare rilevanza riguardo le caratteristiche dell'S1 Core sono i file SPEC.txt e TODO.txt.

SPEC.txt Descive le specifiche dell'S1 Core, e vengono qui riassunte.

L'S1 Core consta di 4 macro-moduli: Reset-controller, Interrupt-controller, SPARC Core, Wishbone Bridge.

L'ISA (Instruction Set Architecture) è la medesima dello SPARC V9 a 64-bit. Sono escluse le istruzioni dedicate ad unità non presenti nello SPARC core, come FPU (Floating Point Unit) e SPU (Stream Processing Unit). La documentazione principale di riferimento per l'ISA è inclusa nel 3.1.2.4 e nel 3.1.2.5.

L'architettura V9 è supportata dal compilatore GCC. Inoltre, anche Ubuntu Linux, di default, può essere installato su sistemi SPARC, in particolare sull'OpenSPARC T1, e anche, con delle modifiche, sul S1 Core.

Viene descritto il *memory mapping* dell'S1 Core e le differenze con in *memory mapping* dell'OpenSPARC T1, vedi §4.3.

Il Reset Controller intercetta il segnale di reset (il Power On Reset) e si preoccupa di generare un insieme di segnali richiesti per l'avvio (boot) corretto dello SPARC Core.

L'Interrupt controller intercetta gli interrupt esterni al core e genera un pacchetto CPX corretto per lo SPARC core. Questa unità non è testata.

Il Wishbone bridge si preoccupa di tradurre il protocollo di comunicazione usato dallo SPARC core (PCX/CPX protocol) in Wishbone. PCX/CPX sta per Processor-to-Cache-Crossbar e Cache-to-Processor-Crossbar, dove la Cache alla quale il nome si riferisce è quella L2 dell'OpenSPARC T1, eliminata nell'S1 Core: dati e istruzioni dell'S1 sono indirizzati direttamente in/da RAM.

Inoltre vi sono i dati del Wishbone bus, che vengono riportati in forma integrale:

- "Wishbone Master interface that follows revision B.3";
- "standard signals names identified by leading "wbm_" chars";
- "no ERR/RTY support";

- "64-bit Address Bus (with some bits unused);" vedi §4.3
- "64-bit Data Bus supporting 8, 16, 32 and 64-bit accesses";
- "data transfer ordering is Big Endian";
- "supports Single Read/Write Cycles."

TODO.txt Il file TODO.txt riporta le limitazioni ed i problemi ancora presenti nell'S1 Core relativi a design, simulazione. Vengono riportati in forma integrale, nella versione attuale dell'S1 Core:
Higher priority tasks:

- synth problem: Icarus assertion failed

Lower priority:

- support for PCX/CPX packets other than IMISS/LD/ST
- full test-suite for verification
- harness and interrupt handlers to verify INT_CTRL

Nell'ambito di questa tesi è stato riscontrato che, lanciando il tool che assembla il verilog del core, mentre la riduzione "EE" va a buon fine, Icarus fallisce le due regressioni "SE" e "ME".

Inoltre la simulazione si blocca alla richiesta di una istruzione di load/store. Il problema non sembra essere del Wishbone, ma interno allo SPARC. Questi aspetti verranno estensivamente discussi nell'analisi della procedura di boot del processore.

4.2.3 hdl

4.2.3.1 filelist.*

I files filelist.*nome_tool* contengono la lista dei file verilog da aggiungere al progetto di sintesi o di simulazione, per ogni sintetizzatore o simulatore supportato. filelist.icarus e filelist.fpga fanno entrambi capo a Icarus Verilog, uno per la simulazione e l'altro per la sintesi rispettivamente.

4.2.3.2 behav

Contiene due sottocartelle: sparc_libs e testbench.

behav/sparc_libs Vi sono due librerie originali dell'OpenSPARC. Contengono la dichiarazione, in forma parametrica, di moduli di base come and, xor, not, adder, moduli di calcolo dei bit di parità usati nel core per la verifica dei dati inseriti nelle caches. Sono moduli totalmente combinatori, nei quali la logica viene gestita prevalentemente tramite assign; in qualche modulo vi sono case in always con full-sensitivity-list. Questi due moduli saranno modificati al lancio del tool update_macrocell. Nel design Opensparc, questi due files si possono trovare sotto $T1_ROOT/lib.

behav/testbench Contiene tre files: mem_harness.v, s1_defs.h, testbench.v. Questi sono tre moduli Verilog dell'S1 Core, non presenti nell'OpenSPARC T1.

s1_defs.h Contiene tutte le #defines utili alla parametrizzazione dell'S1 Core. Richiama il file t1_defs.h, che contiene invece le defines dell'OpenSPARC. Definisce il timescale per la simulazione: negli altri moduli il timescale non è dichiarato e viene ereditato da qui. Viene dichiarata la grandezza del bus Wishbone (64-bit). Sono codificati gli stati della FSM che governa il bus. Vi è la codifica dei parametri che servono a gestire le temporizzazioni nel gestore di reset, che nell'S1 Core è customizzato e si trova in $S1_ROOT/hdl/rtl/s1_top/rst_ctrl.v.

testbench.v E' il modulo "top" usato in simulazione. Richiama le librerie. Gli assign, definiscono che quando RAM e ROM non sono indirizzati dal bus, essi mettano in uscita il primo dato, presente all'inizio della memoria. All'interno dell blocco *initial*:

- `$display` stampa a video.
- `$dumpfile` e `$dumpvars` creano un file, trace.vcd, su cui verranno scritti i risultati della simulazione, leggibili poi con gtkwave.
- vengono attivati clock e reset: il reset è comandato dal gestore di reset, ed è un reset di tipo POR (power on reset). Questa informazione è importante, in quanto lo SPARC core ha diversi tipi di reset e dopo ogni reset lo stato

dei registri del processore è differente: può dipendere anche dalla configurazione antecedente al reset (tipicamente con i reset attivabili via software). I reset sono descritti estensivamente nel documento 3.1.2.4.

- `#49000` è un tempo sufficiente per eseguire la procedura di boot e un piccolo programma, per cicli di clock da 1 nsec.

Vengono poi istanziati il modulo top dell'S1 Core, RAM e ROM. Le due memorie sono identiche, a meno dei parametri dichiarati alla fine del testbench, e sono descritte dal modulo mem_harness. Degno di nota è che gli ultimi 3 bit dell'indirizzo vengono ignorati, in quanto servono ad indirizzare il byte all'interno della double-word. Ogni double-word è da 64 bit, e servono $64/8 = 8$ combinazioni per indirizzarla. Le memorie vengono inizializzate con 64'h0100000001000000 che corrispondono alla codifica di due istruzioni di NOP (no operation). I dati da inserire nelle memorie, i "programmi", sono contenuti nei due files esadecimali ram_harness.hex e rom_harness.hex. Questi due file sono nella cartella $S1_ROOT/tests e $S1_ROOT/tests/boot, e vengono prima modificati (contengono il risultato della compilazione) ed è poi fatto il link nella cartella di progetto del simulatore (in modo da essere accessibili in simulazione) dallo script compile_test. Nei commenti che precedono le defparam per RAM e ROM, viene ricordato quali sono gli indirizzi associati alle due memorie nel memory-mapping dell'S1 Core: [$FFF0000000_{16}$: $FFF0000FFF_{16}$] per la ROM (9 bit) e [0000040000_{16} : $000004FFFF_{16}$] per la RAM (13 bit). Il *memory-mapping* completo si può trovare nel file di documentazione $S1_ROOT/doc/SPEC.txt.

mem_harness.v E' il file Verilog che descrive le memorie ROM e RAM Wishbone-compatibili.

I parametri dichiarati in questo modulo verranno modificati nel testbench.v. La memoria è indicata dal doppio array mem. Il dato in uscita dalla memoria è wbs_data_o. tmp_rd e tmp_wd tengono il dato da leggere o scrivere sulla memoria. addr_bits è un parametro che detiene il numero logico di bit necessario per indirizzare la memoria, mentre wbs_addr_i è l'indirizzo di memoria richiesto dallo SPARC. Il dato in lettura (tmp_rd) sarà dunque mem[wbs_addr_i[addr_bits+2:3]]: vengono esclusi dall'indirizzo gli ultimi 3 bit, quelli che indirizzano il byte all'interno della parola (da 64 bit); addr_bits è incrementato di 2 e non di 3 perché corrisponde alla grandezza logica. La generazione del dato in scrittura è più

complessa perché può essere richiesta dal CORE la modifica di un singolo byte all'interno della parola, e dunque bisogna preservare gli altri: wsb_sel_i è un segnale a 8 bit che detiene l'informazione su quali byte della parola vogliono essere modificati.

All'interno dell' always@(clock) viene gestito il ciclo di scrittura o lettura, entrambi durano un singolo ciclo di clock: la memoria è sincrona sia in lettura che in scruttura. Quando sono alti wbs_cycle_i e wbs_strobe_i (vengono sempre alzati contemporaneamente dal Wishbone master) la memoria è in lettura o scrittura a seconda che wsb_we_i (*write enable*) sia basso o alto. Viene sempre alzato il segnale di *acknowledge* (wsb_ack_o). In lettura viene portato tmp_rd in uscita (wsb_data_o). In scrittura, viene assegnato tmp_wd alla locazione di memoria indirizzata, mentre l'uscita resta al suo valore di default: wsb_data_o=64'bZ.

4.2.3.3 macrocell/sparc_libs

In questa cartella, inizialmente vuota, verranno posizionati i due files presenti nella cartella
$S1_ROOT/hdl/behav/sparc_libs al lancio dello script
$S1_ROOT/tools/bin/update_macrocell. Avviando lo script update_macrocell, le librerie m1_lib.v e u1_lib.v che saranno posizionate in questa cartella, avranno all'interno soli moduli di interfaccia, svuotati di assign e always: questo farà sì che il processore perda ogni funzionalità e, in fase di simulazione, non si otterranno variazioni dei segnali dopo il reset. Le due librerie della cartella
$S1_ROOT/hdl/behav/sparc_libs resteranno invece invariate.

4.2.3.4 rtl

Contiene due cartelle: s1_top e sparc_core. A parte qualche libreria, il testbench e la descrizione della memoria, in questa cartella è contenuto tutto il design del processore.

rtl/s1_top Qui sono contenuti i file Verilog che descrivono il bus Wishbone, il Reset Controller, l'Interrupt Controller, il top-module di Wishbone-bus e SPARC core e due file *_defs.h contenenti dichiarazioni di parametri.

int_ctrl.v Interrupt Controller. Codifica gli interrupt esterni allo SPARC, ricevuti nel segnale da 64 bit sys_irq_i, tramite un priority encoder e genera l'uscita a 6 bit sys_interrupt_source_o. Inoltre, durante il reset non possono essere generati interrupt.

rst_ctrl.v Reset Controller. Il gestore di reset è dotato di un timer che temporizza le varie fasi del reset: cycle_counter. Finché il segnale di reset è alto, il counter è 0 e viene inviato allo SPARC core come unico segnale sys_reset_final_o. Quando si abbassa (il testbench pilota il segnale di reset, che rimane alto per 1000 cicli di clock) vengono alzati in ogni fase due segnali e poi tenuti alti indefinitamente.
Il clock di sistema (gclk_o) non viene azionato finchè la procedura di reset non è terminata: questo fa si che il processore rimanga immobile e non cambino i valori nei registri prima che il reset sia terminato. Quando sono alti tutti i segnali pilotati, sys_reset_final_o viene abbassato e successivamente fatto ripartire il clock dello SPARC.

spc2wbm.v SPARC to Wishbone Master. E' il Wishbone Bus. Gestisce il trasferimento da e verso lo SPARC core dei pacchetti (dati o istruzioni). Gestisce anche la traduzione dei pacchetti da PCX/CPX (protocollo di comunicazione dello SPARC core) a Wishbone (protocollo di comunicazione delle unità esterne). Vengono qui richiamate solo alcune peculiarità del modulo: per una comprensione completa della FSM (Macchina a Stati Finiti), riferirsi alla documentazione ufficiale sul protocollo Wishbone B3, reperibile su www.opencores.org.
Tra i commenti, dopo la dichiarazione dei segnali, sono dichiarate quali istruzioni sono supportate dal bus, da e verso lo SPARC core:

dallo SPARC al Wishbone: "spc2wbm_type (5 bits) is one of: { LOAD_RQ, IMISS_RQ, STORE_RQ, CAS1_RQ, CAS2_RQ, SWAP_RQ, STRLOAD_RQ, STRST_RQ, STQ_RQ, INT_RQ, FWD_RQ, FWD_RPY, RSVD_RQ }"

dal Wishbone allo SPARC: "wbm2spc_type (4 bits) is one of: { LOAD_RET, INV_RET, ST_ACK, AT_ACK, INT_RET, TEST_RET, FP_RET, IFILL_RET, EVICT_REQ, ERR_RET, STRLOAD_RET, STRST_ACK, FWD_RQ_RET, FWD_RPY_RET, RSVD_RET }"

Viene poi fatta la traduzione del pacchetto da PCX/CPX a Wishbone. Nella documentazione dell'OpenSPARC T1, in 3.1.2.3, è descritto in dettaglio il protocollo PCX/CPX, qui si riporta qualche dato.

I pacchetti in uscita dallo SPARC core (PCX) sono di 128 bit, di cui i 64 bassi contengono il dato, e i restanti 64 bit contengono informazioni riguardo il packet stesso. In tabella 4.1, una breve descrizione dei pacchetti PCX.

I pacchetti in ingresso allo SPARC core (CPX) sono di 145 bit, di cui i 128 bassi contengono il dato, e i restanti 17 bit contengono informazioni riguardo il packet stesso. In tabella 4.2, una breve descrizione dei pacchetti CPX.

All'interno dell'always è descritta la macchina a stati del bus. E' inattivo nel caso di segnale di reset alto, e viene preparato il packet di wake-up (sveglia) per lo SPARC core da 145 bit: 0x17000...0010001. Nello stato di *wakeup* viene inviato il pacchetto preparato durante il reset, per poi entrare nello stato di quiete, *idle*. Viene esaminata quindi un possibile invio di un pacchetto di interrupt. Da qui in poi, le sequenze di stati assunti dal bus dipendono dalla richiesta. Viene dapprima esaminato il pacchetto PCX inviato dallo SPARC core per poi essere processato. Alcune richieste necessitano il passaggio forzato attraverso più stati per consentire l'invio (ricezione) di dati composti di più pacchetti. Vi sono inoltre dei comandi $display, che, se dichiarato il parametro "DEBUG", generano stringhe che verranno inserite nel file di log della simulazione: per le simulazioni con Icarus Verilog (tramite lo script run_icarus), il file si trova in $S1_ROOT/run/sim/icarus/sim.log.

Tabella 4.1: PCX packet generato dallo SPARC core per il Wishbone bus.

Bit	**Descrizione. PCX packet bits**
123	*Valid*. Dice se il pacchetto è valido.
122:118	*Return Type*. Dice di che tipo è il pacchetto (Load, Store, IFILL Request ecc.).
117	*Non Cacheable*. Dice se il dato richiesto deve o no bypassare le Caches L1. Tipicamente, il fetch di istruzioni dalla *BOOT PROM* sono pacchetti *Non-Cacheable*.
116:114	*CPU ID*. Dice da quale CPU avviene la richiesta.
113:112	*Thread ID*. Dice a quale thread è associata la richiesta.

Bit	Descrizione. PCX packet bits
111	*Invalidate*. Dichiara se sia una richiesta di invalidare un dato nella cache L2 perché venga poi aggiornato.
110	*Prefetch*. E' una richiesta di prefetch di una istruzione.
109	*Block Init Store*. Se alto, la store è block-init. Per dettagli, vedere §3.1.2.3, paragrafo 3.4.8.
108:107	*Replacement L1 Way*. In una load o in un pacchetto IFILL (richiesta di riempimento Instruction Cache L1) indica in quale "way" il pacchetto sarà posizionato nella cache.
106:104	*Transaction Size*. Indica di quanti bit è il dato (Byte, Half word (2 byte), Word (4 byte), Extended word (8 byte), Cache Line (16/32 byte)).
103:64	*Transaction Address*. Questi 40 bit indicano l'indirizzo associato al dato.
63:0	*Data*. E' il dato, l'informazione vera e propria contenuta nel pacchetto. Se il dato non è a 64 bit, viene replicato sui bit "vuoti".

Tabella 4.2: CPX packet generato dal Wishbone per lo SPARC core.

Bit	Descrizione. CPX packet bits
144	*Valid*. Dice se il pacchetto è valido. Viene buttato se questo bit non è a 1.
143:140	*Return type*. Questi 4 bit indicano che tipo di transizione è inviata.
139	*L2 Miss*. Indica che il dato non era presente nella cache L2.
138:137	*Error*. La transazione è ha avuto un errore. Il bit 138 indica che l'errore è non-correggibile, mentre il 137 indica che è correggibile. Ogni byte di cache è controllato da un bit di parità: una discordanza tra un byte di cache e il bit di parità genera un pacchetto con errore.
136	*Non Cacheable*. Dice se il dato richiesto deve o no fare il bypass delle Caches L1. Tipicamente, il fetch di istruzioni dalla *BOOT PROM* sono pacchetti Non Cacheable.
135:134	*Thread ID*. L'identificatore del Thread che ha richiesto la transazione.

Bit	Descrizione. CPX packet bits
133	*Way Valid*. Indica che la linea di cache che si sta caricando è una linea valida nella cache di un altro core.
132:131	*Way*. Indica in quale linea di cache risiede il dato. Alla ricezione del dato, il core deve invalidare il dato presente nella L1 cache dell'altro core.
130	*Four Byte Fill*. E' usato nei pacchetti *I-Fill return*, per indicare che il campo data contiene una sola istruzione (invece delle 2 normalmente presenti). Tipicamente, questo bit è alto durante il fetching dallo spazio di I/O, solitamente dalla *BOOT PROM*.
129	*Atomic*. Indica che sta avvenendo una transazione atomica (il dato è diviso in più pacchetti che sono inviati obbligatoriamente uno di seguito all'altro, senza intermezzi di altri pacchetti non riguardanti il dato).
128	*Prefetch*. Indica che il pacchetto sta arrivando al core in seguito ad una richiesta di *prefetch*.
127:0	*Data*. Contiene il dato. I 16 byte sono tutti riempiti per le *cacheable-load* e per gli *I-Fill return* (richiesta di riempimento della cache L1) packet. Per la configurazione di pacchetti di tipo *ack, invalidate data field, interrupt, FPU return* vedere la documentazione §3.1.2.3.

s1_top.v S1 Core Top Module. E' il modulo che istanzia i macro-moduli dell'S1 core: il core SPARC, il Wishbone Master bridge e i due controller di Reset e Interrupt. Disabilita inoltre tutti i segnali di test, fuse e scan. Essendo istanziato un solo SPARC core, gli assegna via hardware il *CPU ID* 0.

s1_defs.h Sono dichiarati i parametri usati solo nei moduli aggiuntivi dell'S1 Core (rispetto all'OpenSPARC) e il 'timescale, che viene ereditato da tutti i moduli che fanno l'include di questo file. Le tre 'define FPGA_SYN, FPGA_SYN_1THREAD e FPGA_SYN_NO_SPU sono usate nei moduli verilog dell'OpenSPARC per discernere tra i moduli da istanziare o meno nelle diverse riduzioni.

t1_defs.h Sono dichiarati i parametri usati nello SPARC core. Viene richiamato nel file

s1_defs.h, il quale viene a sua volta invocato in tutti i moduli presenti in questo path, in particolare nell file s1_top.v: questo fa si che i parametri qui dichiarati siano visibili a tutti i moduli del design.

rtl/sparc_core In questo path sono inclusi tutti i files Verilog che descrivono il core estratto dall'OpenSPARC T1.

include Vi è anche una cartella, include, che resterà vuota: viene usata come path d'appoggio per l'esecuzione dello script di riduzione (tools/bin/update_sparccore). I files che durante lo script vengono qui copiati, usati e cancellati, pur non avendo utilità funzionale per simulazioni e sintesi del core, possono rivelarsi utili nella comprensione dello SPARC, essendo la dichiarazione dei parametri dei suoi macro-moduli (IFU, LSU, TLU ecc.). Si possono recuperare dal pacchetto dell'OpenSPARC T1 dalla directory iop, che contiene tutto il design dell'OpenSPARC: $T1_ROOT/design/sys/iop/include.

SPARC files Senza addentrarsi nello specifico contenuto, si può ricavare qualche informazione dai nomi dei files (spesso corrispondenti ai moduli in essi contenuti), che può rivelarsi utile in fase di simulazione nella ricerca dei segnali all'interno del core. Il design è infatti molto vasto e la ricerca del segnale voluto non sempre risulta agevole. In generale, nei moduli che contengono la stringa *dp* si riferiscono al data-path[1] del modulo, mentre quelli con *ctl* sono in genere moduli che descrivono la logica che governa i data-path. Si riporta qui per comodità il listato di un modulo che risulta mancante in fase di building del verilog, appartenente al file swrvr_clib.v, come richiamato in tabella 4.3.

[1] *Data-path e Controllo*. Una tecnica ormai assodata nella progettazione di unità digitali è di dividere in moduli differenti l'hardware necessario a detenere informazioni (data-path) con l'hardware che ne governa il funzionamento (controllore). Il data-path può essere visto come un oggetto, mentre il controllore decide come questo oggetto deve essere usato. Questa tecnica è estensivamente usata nel design dell'OpenSPARC T1, anche perché consente di avere codice flessibile, modulare e riutilizzabile. Ad un codice con queste caratteristiche è associato il risparmio di tempo e denaro: è più raro commettere errori logici in fase di stesura del codice; il codice organizzato è più facile da mantenere e modificare; la fase di debug, tipicamente più lunga di quella di stesura, si riduce.

Tabella 4.3: Descrizione dei files dello SPARC core.

nomi	**descrizione**
bw_r_*	Descrivono memorie dello SPARC. A loro è dedicato un intero files di documentazione, vedi §3.1.1. In genere sono file di 100 righe, che possono essere agevolmente letti per estrarre informazioni cercate.
bw_r_dcd, bw_r_icd, bw_r_idct	L1 Data Cache, L1 Instruction Cache, L1 Data Instruction Cache Tag.
bw_r_frf	Registri Floating Point. Nell'S1 Core la FPU non è istanziata.
bw_r_irf, bw_r_irf_register	Integer Register File. *_irf è la descrizione di una *window* di 32 registri. *_register è invece la descrizione del singolo registro a 72 bit.
bw_r_rf*x*	16x32: ICache e DCache valid bits. 32x80: Trap stack. Usato per salvare vettori che contengono informazioni sulla gestione degli errori.
bw_r_tlb	Translation Lookaside Buffer. Memoria responsabile della traduzione degli indirizzi e la comparazione del campo TAG dell'istruzione o dato. La funzionalità è descritta nei paragrafi 2.3.9 (ITLB) e 2.4.4(DTLB) del documento §3.1.2.3.
cpx_spc*	Core-to-Processor-Crossbar - SParc-Core. Moduli di traduzione per segnali in ingresso al core.
lsu*	Load Store Unit. Nella LSU, sono definiti circa 1000 segnali. I files lsu_dc* si riferiscono alla Data Cache; lsu_q* sono i moduli che gestiscono le code, necessarie per garantire il Total Store Order; lsu_stb* è lo Store Buffer; lsu_tlb e lsu_tag si riferiscono al DTLB e ai DCache Tag.
lsu_asi_decode	Descrive il modulo che intercetta i primi 8 bit di ogni istruzione, che detengono l'ASI (Address Space Identifier). In questo modulo sono codificati e, leggendolo, si possono sapere le associazioni tra gli otto bit dell'ASI e il nome dei registri a cui essi puntano. Gli ASI per l'architettura UltraSPARC sono descritti in §3.1.2.4 e §3.1.2.5, ma molti sono *architecture-dependent* e bisogna leggere il verilog se si ha necessità di sapere alcune corrispondenze.
sparc	Descrive il *top-module* dello SPARC Core.
sparc_ffu*	Floating Point Frontend Unit.

nomi	descrizione
sparc_ifu*	Instruction Fetch Unit. Una unità fondamentale nella gestione dei cicli di istruzione. Controlla, tra l'altro, la pipe, la ICache e ICache Tag, detiene i Program Counter di esecuzione e di ogni singolo thread, gestisce l'alternarsi dei threads (ifu_thrfsm), ha parecchi "contatti" sia con la LSU che con la TLU (unità di gestione degli errori).
tlu*, sparc_tlu*	Trap Logic Unit. E' l'unità che gestisce gli errori (trappole). A questa unità è dedicata anche la gestione della MMU (Memory Management Unit). Gestisce inoltre il super-super-utente (Hypervisor), nel modulo tlu_hiperv.
swrvr_clib, swrvr_dlib	Sono due moduli di libreria. Nelle simulazioni che sono state effettuate nell'ambito di questa tesi, risulta che, per la corretta compilazione del verilog dell'S1 Core, bisogni aggiungere manualmente il modulo "dffr", che si può trovare nel design OpenSPARC su internet, cercando in §3.1.2.7, nel modulo swrvr_clib.v. E' stato riportato il listato integrale in 5.1.1

4.2.4 run

In questo path, composto inizialmente da cartelle vuote, vengono creati i risultati delle simulazioni e delle sintesi. Le cartelle sono organizzate per tipo (simulazione o sintesi) e per nome del simulatore o sintetizzatore.

4.2.5 tools

E' la sezione operativa del pacchetto dell'S1 Core. La cartella tools/bin contiene gli script necessari alla riduzione, simulazione, sintesi e compilazione del codice per il core. tools/opt contiene due tools opzionali. La cartella tools/src, contiene file di dati usati dai tools della cartella bin. I tools si possono eseguire da qualsiasi path (senza richiamarne il path completo) se si è precedentemente eseguito il comando (vedi §4.2.1)

```
usr@name$ source ~/s1_core/sourceme
```

4.2.5.1 src

Contiene dei files di appoggio per vari tools.

build_* Sono files usati durante il lancio degli script tools/bin/build_*, per compilare il Verilog dell'S1_core. Al loro interno compattano il comando 'run' del sintetizzatore/simulatore, fornendo già delle opzioni necessarie alla corretta costruzione dell'ambiente di compilazione dell'S1 Core nei diversi tools. Si può inoltre configurare per definire su quale FPGA il sintetizzatore dovrà effettuare la sintesi.

bw_r_* Sono i files verilog che descrivono le caches L1 dello SPARC Core e i rispettivi Tag, a cui è stata lasciata solo l'interfaccia: il corpo dei moduli è vuoto. Verranno sostituiti ai rispettivi moduli originali (in $S1_ROOT/hdl/rtl/sparc_core/) nel caso sia richiesto il core senza caches al lancio dello script $S1_ROOT/tools/bin/update_sparccore (con l'opzione -me).

dump2hex.c E' un programma, scritto in c, che, dato un file binario, lo traduce in un file esadecimale, in righe da 16 caratteri. Quest'ultima forma è quella utile per generare i files tests/boot/rom_harness.hex e tests/ram_harness.hex, usati poi in fase di simulazione (richiamandoli nel file mem_harness.v, vedi §4.2.3.2) per riempire ROM e RAM con i programmi voluti. Questo programma C verrà compilato (per il PC in uso, non per lo SPARC!) ed eseguito due volte all'interno dello script tools/bin/compile_test.

gtkwave.sav E' un file di salvataggio del programma gtkwave, utile per visualizzare le forme d'onda generate dalla simulazione con Icarus Verilog lanciando lo script tools/bin/run_icarus. Aprendo questo file da gtkwave, potremmo, senza andarle a cercare nel design, le forme d'onda di ingressi, uscite e qualche variabile di stato del Wishbone bus. Per il debug non bastano, ma in prima approssimazione si riesce a vedere se il bus risponde agli stimoli e se esegue correttamente la transazione richiesta (il pacchetto che viaggia sul bus ha in sè, oltre il dato, molte informazioni sulla sua natura, vedi le tabelle 4.1 e 4.2).

linker.map Serve al cross-compilatore, in fase di linking, per generare il codice-macchina a partire dal codice sorgente C del programma che verrà inserito in RAM. Sono dichiarate all'interno gli indirizzi della prima word di ROM e RAM, oltre che le rispettive grandezze fisiche. Utile notare che il primo indirizzo di ROM è $0xFFF0000020$, e che quindi le prime 4 istruzioni (2 righe) del file $S1_ROOT/tests/boot/rom_harness.hex non verranno eseguite dal processore. Viene utilizzato dallo script tools/bin/compile_test.

sourceme E' una copia del file $S1_ROOT/sourceme, vedi §4.2.1.

sparc.v E' una copia modificata del top-module di un core dell'S1 Core, presente anche in
$S1_ROOT/hdl/rtl/sparc_core: questa copia prevede una modifica nell'istanziazione della SPU (Stream Processing Unit), che non è inclusa nel design dell'S1 Core, vedi §4.2.5.3.

synopsys_dc.setup E' un file dove vengono definite delle variabili d'ambiente per il tool dc di Synopsys, usato in fase di sintesi dallo script
$S1_ROOT/tools/bin/build_dc.

4.2.5.2 opt

Nella cartella sono messi a disposizione due tools opzionali, che non sono strettamente necessari per simulare e sintetizzare il core. Possono essere invece utili in fase di debug.

disass E' un tool utile per disassemblare del codice sorgente. Per eseguire il tool, lanciare su shell bash il comando

```
usr@name$ cd $S1_ROOT/tools/opt/disass
usr@name$ ./disass
```

Il file prog.c è il codice corrispondente alla procedure di boot dell'OpenSPARC. Lanciando il tool, viene prima generato il file prog.o e poi disassemblato. L'output di default è il prompt; per scrivere il disassemblato su file, per esempio "prog.dump", basta eseguire

```
usr@name$ cd $S1_ROOT/tools/opt/disass
usr@name$ ./disass > prog.dump
```

Per eseguire il tool su un listato differente, modificare il file "disass", scrivendo la directory e il nome del file da disassemblare.

trcan Trace analizer. Serve per comparare forme d'onda. Richiede di avere un sistema OpenSPARC T1 configurato e funzionante per poter essere correttamente eseguito.

4.2.5.3 bin

In questa directory vi è una collezione di script preconfigurati. Per eseguire correttamente i programmi bash contenuti in questa directory, bisogna prima aver configurato le variabili d'ambiente in $S1_ROOT/sourceme e averlo eseguito sulla stessa shell che si intende usare (vedi §4.2.1):

```
usr@name$ source ~/s1_core/sourceme
```

Per meglio aiutare alla comprensione dei legami tra i tools, si dà qui la sequenza di comandi che dovrebbe portare ad una corretta simulazione, usando Icarus Verilog.

```
usr@name$ source ~/s1_core/sourceme
usr@name$ update_sparccore -ee
usr@name$ update_filelist
usr@name$ compile_test hello
usr@name$ build_icarus
usr@name$ run_icarus
```

Il comando source è forzatamente il primo. update_sparccore e update_filelist devono avere forzatamente questa sequenza, e vanno lanciati (entrambi) solo se si vuole cambiare la riduzione del core da usare. compile_test, build_icarus e run_icarus devono essere lanciati in questa sequenza. build e run vanno a buon fine solo se update_filelist è stato lanciato almeno una volta dopo aver eseguito update_sparccore.

build_nomeTool Questi script configurano, ciascuno per il simulatore/sintetizzatore dichiarato nel nome, le variabili d'ambiente necessarie a compilare cor-

rettamente il Verilog dell'S1 Core, che viene poi compilato. Molti di questi, si appoggiano ai file filelist.nomeTool della directory $S1_ROOT/hdl/ (vedi §4.2.3.1) per includere nel design i files necessari. I files filelist.nomeTool devono dunque essere aggiornati prima del lancio dei tools di build. Lo script che li configura è update_filelist, vedi §4.2.5.3. Inoltre questi script creano dei files nella cartella run/-sim/nome_simulatore o run/synth/nome_sintetizzatore, nella quale viene generato lo script "testbench", utilizzati dagli script run_*.

Per compilare i file Verilog dell'S1 Core usando Icarus, lanciare da shell:

```
usr@name$ build_icarus
```

clean_env L'esecuzione di questo tool elimina tutti i files di simulazione (la cartella $S1_ROOT/run viene ripulita) e ogni file *.bin derivato dalla compilazione di codice C o assembly. E' stato pensato per ripulire il pacchetto prima di una eventuale ridistribuzione del pacchetto. Per eseguirlo, digitare da shell

```
usr@name$ clean_env
```

compile_test Compila del codice-applicazione generando codice-macchina SPARC, che viene poi convertito in forma esadecimale e scritto sui due file rom_harness.hex e ram_harness.hex, dai quali verranno prelevati istruzioni e dati in fase di simulazione (vedi §4.2.3.2).

Il codice-applicazione che verrà inserito in ROM è prelevato da un file sorgente Assembly, $S1_ROOT/tests/boot/boot.s, e il codice-macchina viene scritto nel file $S1_ROOT/tests/boot/rom_harness.hex; vengono generati il file disassemblato .dump, file binario .bin, il codice oggetto .o.

Il codice-applicazione che verrà inserito in RAM è invece contenuto in un file sorgente C, $S1_ROOT/tests/hello.c, e il codice-macchina viene scritto nel file $S1_ROOT/tests/ram_harness.hex; vengono generati il file disassemblato .dump, file binario .bin, il codice oggetto .o.

Per usare lo script, eseguire su shell bash:

```
usr@name$ compile_test hello
```

dove "hello" è il nome privo di estensione del file sorgente C il cui codice sarà cross-compilato[2] (vedi §4.1), fatto il dump e inserito in ram_harness.hex; il file .C deve essere nella cartella
$S1_ROOT/tests/.

Può essere utile, in fase di test, modificare questo file per ottenere la compilazione di files diversi da quelli di default e per inserire tutto il codice-macchina in ROM (sia quello di boot che quello dell'applicazione che di default viene inserito in RAM). Si analizza qui lo script, per evidenziare come poterlo cambiare a piacere.

Le prime righe non commentate controllano che siano definite le variabili di shell dell'S1 Core (vedi §4.2.1). Il comando

```
cd $S1_ROOT/tests
```

sposta il path di lavoro nella directory tests, dove sono presenti i sorgenti da compilare e i files .hex. Viene ripulita la directory dai file derivanti da precedenti compilazioni: commentare questa riga in caso si volessero mantenere, cambiandone il nome perché nei passi successivi saranno sovrascritti comunque:

```
rm -f *.o *~ *.bin *.dump *.hex      # Make clean
```

Vengono in seguito stampati a video copyright e come usare lo script: di default si aspetta un parametro in ingresso, che corrisponde al nome del codice-applicazione scritto in C e destinato alla RAM. Viene generato un errore se il file da cross-compilare non è nella cartella test.

Viene compilato tramite gcc il programma dump2hex.c (vedi §4.2.5.1) per la macchina e sistema operativo in uso generando il file binario che servirà in seguito.

Viene cross-compilato il file assembly tests/boot/boot.s e generato il file binario e il codice oggetto; nella seconda istruzione viene generato il dump-file:

```
sparc64-linux-as -ah -am -o boot/boot.bin boot/boot.s
sparc64-linux-objdump -d -EB -w -z boot/boot.bin > boot/
   boot.dump
```

Vi è una pipe di tre comandi l'output generato da uno viene preso come input dall'altro: grep seleziona le righe di codice e la riga che inizia per ".nome_sorgente"

[2]Un *cross-compilatore* è un compilatore che, invece di compilare il codice-applicazione per il sistema su cui è installato, genera codice macchina per un altro sistema. Questo consente, per esempio, di compilare codice per macchine SPARC anche su sistemi Intel o AMD.

(piuttosto che dichiarazioni del compilatore) dal file di dump; egrep estrae solo il codice; dump2hex.bin trasforma il codice in esadecimale. Il risultato, viene poi sostituito al contenuto del file rom_harness.hex. Per inserire il codice generato alla fine del file, piuttosto che sostituire il contenuto, usare ">>" al posto di ">".

Le righe successive replicano i medesimi passi partendo dal file hello.c fino ad inserire le istruzioni in ram_harness.hex.

Per inserire il codice in ROM, modificare l'ultima linea non commentata del file:

```
grep "␣␣" $1.dump | egrep -v "file␣format" | dump2hex.bin
   >> boot/rom_harness.hex
```

Per estrarre il file assembly SPARC da un file sorgente C, lanciare il comando:

```
usr@name$ sparc64-linux-gcc -S nome_file.s nome_file.c
```

Può essere utile voler scrivere un programma in assembly da inserire in ram. Assumendo che il file assembly da compilare sia nel path $S1_ROOT/tests e si chiami hello.s, digitare i seguenti comandi:

```
usr@name$ gcc -o dump2hex.bin $S1_ROOT/tools/src/dump2hex.c
usr@name$ sparc64-linux-as -ah -am -o hello.bin hello.s
usr@name$ sparc64-linux-objdump -d -EB -w -z hello.bin >
   hello.dump
usr@name$ grep "␣␣" hello.dump | egrep -v "file␣format" |
   dump2hex.bin > ram_harness.hex
```

run_icarus e run_vcs Si preoccupano di lanciare la simulazione, lanciando lo script $S1_ROOT/run/sim/icarus/testbench o $S1_ROOT/run/sim/vcs/testbench rispettivamente. I due script sono stati creati dagli script di compilazione build_icarus e build_vcs. Viene inoltre:

- fatto il link simbolico dei file $S1_ROOT/tools/ram_harness.hex e $S1_ROOT/tools/boot/rom_harness.hex
- lanciata la simulazione
- creati, nella propria cartella di run/sim, un file di log nei quali viene salvato l'output dei comandi $display presenti nel verilog

- creati i files per le forme d'onda trace.vcd, leggibili anche con gtkwave.

Questo è l'ultimo passo per la simulazione. A simulazione completata si può analizzare il risultato.

tar_env Crea un pacchetto compresso per la ridistribuzione del core. Può essere utile lanciare prima lo script clean_env, vedi §4.2.5.3.

update_filelist Aggiorna i files $S1_ROOT/hdl/filelist.* (vedi §4.2.3.1). Contengono la lista dei nomi dei files Verilog che descrivono il design dell'S1 Core. Quando si lancia una riduzione del core (con lo script update_sparccore, vedi §4.2.5.3), i files verilog del progetto cambiano: perché simulazioni e sintesi siano eseguite sul design corretto, bisogna eseguire update_filelist subito dopo aver fatto la riduzione. E' necessario che lo script di riduzione e update_filelist vengano lanciati nella giusta sequenza:

```
usr@name$ update_sparccore -riduzione  #riduzione = 'ee' o
   'se' o 'me'
usr@name$ update_filelist
```

update_macrocell Crea una copia modificata dei file di libreria u1_lib.v e m1_lib.v nella cartella $S1_ROOT/hdl/macrocell/sparc_libs, vedi §4.2.3.3. La modifica consiste nel lasciare solo l'interfaccia dei moduli verilog dichiarati nelle due librerie, rendendole funzionalmente inutili. Importando queste librerie al posto delle originali, lo SPARC core è "inanimato", in quanto in queste librerie è presente la logica combinatoria di "piccola taglia" del core: and, xor, not ecc.. Potrebbe essere utile per testare il bus Wishbone con i moduli di Reset e l'Interrupt Controller. Se si vuole testare il core completo, non includere queste librerie modificate. Per eseguire lo script digitare su bash shell:

```
usr@name$ update_macrocell
```

update_sparccore Crea il design dell'S1 Core. Consente di creare 3 design differenti per numero di thread e la presenza o meno delle caches L1, la CPU è sempre unica. Prende un parametro in ingresso, il quale discrimina tra i 3 design possibili, vedi anche tab.2.1:

Elite Edition: il design risultante si compone di 1 SPARC core, completo di 4 thread e delle cache dati e istruzioni di primo livello.

```
usr@name$ update_sparccore -ee
```

Single-thread Edition: il design risultante si compone di 1 SPARC core che gestisce 1 thread; sono presenti le caches L1 dati e istruzioni.

```
usr@name$ update_sparccore -se
```

Memory-less Edition: il design risultante si compone di 1 SPARC core che gestisce 1 thread; le caches L1 sono assenti.

```
usr@name$ update_sparccore -me
```

Per eseguire correttamente update_sparccore è necessario avere:

- il pacchetto dell'OpenSPARC T1 scompattato; è reperibile su http://www.opensparc.net/opensparc-t1/download.html
- le variabili d'ambiente bash S1_ROOT e T1_ROOT correttamente configurate, vedi 4.2.1;
- Icarus Verilog;
- il programma testuale *sed*.

Lo script sceglierà dal design dell'OpenSPARC T1, da $T1_ROOT/design/sys/iop, i files utili alla creazione dell'S1 Core richiesto, inserendoli nella cartella $S1_ROOT/hdl/rtl/sparc_core. Vengono copiate anche due librerie da $T1_ROOT/lib. Lo script modifica alcuni files verilog di modo che la CPU SPARC sia compatibile con il bus Wishbone. Le modifiche ai files Verilog sono effettuate mediante il programma sed: per conoscere parametri e modi di utilizzo, digitare su una shell: **`sed`** `--help`. Viene usato Icarus Verilog per impostare i parametri globali che gestiscono la configurazione del core ricavato dall'OpenSPARC T1. Il design del T1 è di per sé altamente configurabile: è stato previsto infatti che l'OpenSPARC potesse essere ridotto o potessero essere escluse delle funzionalità, impostando dei parametri globali. L'uso di questi parametri è descritto estensivamente nella documentazione dell'OpenSPARC T1, vedi §3.1.1 e §3.1.2.2. Sebbene sia possibile specificarne di aggiuntivi, nei tre design di default dell'S1 Core vengono usati i seguenti:

FPGA_SYN il codice verilog è ottimizzato per una migliore compatibilità con i devices fisici;

FPGA_SYN_1THREAD il codice verilog prevede che la CPU gestisca un solo hardware thread;

FPGA_SYN_NO_SPU non viene istanziata la SPU (Stream Processing Unit).

4.2.6 tests

La directory tests è adibita a contenere i codici sorgenti dei programmi che, in fase di simulazione, saranno eseguiti dal processore. Il codice ed i file sono divisi tra *procedura di boot*, contenuti nella cartella tests/boot, e applicazione utente, contenuti in tests/. Per quanto il contenuto delle due memorie sia una sequenza di istruzioni e dati trattati dal processore all'identico modo, si sceglie di suddividere il codice tra quello corrispondente alla procedura di boot e non. Tale suddivisione è dovuta a due cause:

- Fisicamente sono presenti due memorie distinte, RAM e ROM (o boot PROM).
- La RAM è progettata per mantenere istruzioni e dati sia del (dei) sistema operativo che dei programmi-utente: è usata costantemente durante il funzionamento della macchina, la comunicazione CPU-RAM deve essere veloce ed ha uno spazio di indirizzi dedicato. L'accesso alla ROM avviene solo in fase di boot, tramite operazioni di I/O: vengono qui salvate le istruzioni ed i dati necessari alla configurazione iniziale del processore, in modo che i registri di stato siano coerenti con l'hardware effettivamente presente.

Il codice-macchina SPARC della procedura di boot è inserito di default nel file boot/rom_harness.hex. Il codice-macchina SPARC corrispondente al codice--applicazione da eseguire una volta che il core è stato configurato, è inserita nel file-immagine della RAM, tests/ram_harness.hex.

Gli altri files già presenti nella cartella o che qui saranno creati dai tools di compilazione (vedi §4.2.5.3) o dal dumper (vedi §4.2.5.1), sono files sorgenti (assembly o C) oppure risultati intermedi di compilazione e dump necessari ad ottenere le immagini delle memorie ram_harness.hex e boot/rom_harness.hex.

Di default, lanciando il tool compile_test (vedi §4.2.5.3), il codice inserito in ram_harness.hex è generato dalla cross-compilazione del file C hello.c; il codice inserito in boot/rom_harness.hex è generato dalla cross-compilazione del file assembly boot/init.s.

ram_harness.hex e rom_harness.hex saranno usate in fase di simulazione: questi due files saranno letti dal processore come se fossero le memorie fisiche.

4.3 Memory-mapping

Il *memory mapping* è una tecnica che consente al core di scambiare dati con qualsiasi dispositivo come se stesse comunicando con la memoria (tramite operazioni di load e store). Il Bus si preoccuperà di redirigere il dato alla (o dalla) memoria fisica piuttosto che a (o da) un altro dispositivo basandosi sull'indirizzo associato al dato. Per esempio, se gli indirizzi sono ad n bit, vi saranno 2^n indirizzi possibili, alcuni dei quali sono associati dal bus alla RAM, altri associati alla ROM, altri associati all'I/O ecc. Il processore potrà comunicare con qualsiasi dispositivo esterno scegliendo l'indirizzo ed effettuando una load o una store: non c'è bisogno di prevedere nell'ISA operazioni specifiche per ogni dispositivo. Il *memory map* di un processore, indica le corrispondenze tra ogni dispositivo e l'insieme di indirizzi a lui dedicato. Per l'S1 Core il memory map è dichiarato nel file $S1_ROOT/docs/SPEC.txt, vedi 4.2.2.

Nell'S1 Core e nell'OpenSPARC T1, sia il Data-Bus che l'Address-Bus, sono a 64 bit. Il bus indirizzi codifica i 64 bit come indicato in tab.4.4.

Tabella 4.4: Codifica bus indirizzi (Adress Bus).

Bit dell'indirizzo	Descrizione
63:59	Specificano lo spazio di indirizzi cui l'istruzione si riferisce.
58:40	Sono 19 bit resi nulli via hardware
39:0	Sono i 40 bit di *physical address*, provenienti dallo SPARC core.

I bit $[63 : 59]$ dell'indirizzo sono codificati in tab.4.5.

Il testbench dell'S1 Core instanzia una RAM indirizzabile con 16 bit, di cui 13 utili per indirizzare la parola all'interno della RAM e 3 per indirizzare il byte

Tabella 4.5: Codifica Adress Bus, bit $[63 : 59]$.

Bit [63:59] binario	**Bit [63:59] esadecimale**	**Regione indirizzata**
00001	0×08	Ram Bank 0
00010	0×10	Ram Bank 1
00100	0×20	Ram Bank 2
01000	0×40	Ram Bank 3
10000	0×80	I/O

all'interno della parola. Poichè il phisical address è da 40 bit, dati e istruzioni da e verso la RAM apparterranno al banco 0 e nelle comunicazioni con la RAM i primi 5 bit avranno valore 0×08. La ROM appartiene invece allo spazio di indirizzi di I/O, dunque i fetch delle istruzioni di boot avranno i primi 5 bit settati a 0×80, indipendentemente dalla sua dimensione fisica.

I primi 8 bit del phisical address dell'S1 Core e l'OpenSPARC T1 hanno diversi *memory map*, descritti nelle tabelle 4.6 e 4.7 rispettivamente. Come dichiarato nel testbench (vedi §4.2.3.2), gli indirizzi di RAM e ROM avranno range di indirizzi rispettivamente:

- 0x0800000000040000 – 0x080000000004FFFF per la RAM;
- 0x8000000FFF000000 – 0x800000FFF0000FFF per la ROM.

Gli indirizzi, sia dell'S1 Core che dell'OpenSPARC T1, hanno granularità di 3 bit: i bit $[2 : 0]$ di ogni indirizzo sono utilizzati per discernere tra gli 8 byte delle parole. Ogni parola è da 64 bit.

Tabella 4.6: Bit $[39 : 32]$ del memory map dell'S1 Core.

Indirizzi, bit $[39 : 32]$	**Componente**
$0 \times 00 - 0 \times 7F$	RAM
$0 \times 80 - 0 \times 95$	*–Reserved–*
0×96	Clock Unit
0×97	Ram Controller
0×98	Wishbone Interconnect Hardware
0×99	DMA Controller
$0 \times 9A - 0 \times 9D$	*–Reserved–*
$0 \times 9E$	General Purpose I/O
$0 \times 9F$	Interrupt Controller
$0 \times A0 - 0 \times FE$	*–Reserved–*
$0 \times FF$	Boot ROM

Tabella 4.7: Bit $[39 : 32]$ del memory map dell'OpenSPARC T1.

Indirizzi, bit $[39 : 32]$	**Componente**
$0 \times 00 - 0 \times 7F$	RAM
$0 \times 80 - 0 \times 95$	*–Reserved–*
0×96	Clock Unit
0×97	Ram Controller
0×98	Modulo di gestione dello I/O bridge
0×99	TAP Unit
$0 \times 9A - 0 \times 9D$	*–Reserved–*
$0 \times 9E$	TAP to ASI
$0 \times 9F$	I/O Bridge Interrupt Table
$0 \times A0 - 0 \times BF$	L2 Cache Control Register
$0 \times C0 - 0 \times FE$	JBUS
$0 \times FF$	Boot ROM

Capitolo 5

S1 CORE: TEST, SIMULAZIONI E PROCEDURA DI BOOT

Nel presente capitolo sarà dapprima presentato il modo per avviare una simulazione dell'S1 Core, sia con Icarus Verilog e Gtkwave che con Modelsim XE. In questa prima parte, vengono elencati i passaggi che portano ad ottenere una simulazione funzionante, fornendo una descrizione operativa del package. Mentre nel capitolo 4 si è fornita una descrizione di file e tool del package, nel presente si cercherà di esporre le modalità di utilizzo. Saranno proposte anche modifiche per aumentare la fruibilità del pacchetto od ottenere risultati differenti.

In seconda istanza viene focalizzata l'attenzione sulla procedura di boot di default dell'S1 Core.

Dopo il reset, gestito via hardware, il processore inizia il fetch di istruzioni da ROM. Durante la procedura di boot si esegue codice hyperpriviledge il cui scopo è configurare un *virtual processor* (o più) in modo che l'hardware raggiunga uno stato coerente e perchè possa essere avviato un sistema operativo[1].

Nel pacchetto di default dell'S1 Core, mentre la configurazione del core dovrebbe essere completa, alla fine del boot non viene avviato un sistema operativo ma eseguito un semplice programma che effettua STORE e LOAD sullo stesso indirizzo. La simulazione si blocca però all'esecuzione della prima istruzione del programma e non effettua le LOAD e le STORE: si è pensato ci fosse un *bug*. Per

[1] L'OpenSPARC T1 fornisce la possibilità di avviare contemporaneamente più di un sistema operativo, e gestisce via hardware la concorrenza. In realtà ad ogni sistema operativo viene assegnato uno o più core (interi), dunque sull'S1 Core è avviabile un solo sistema operativo.

questo motivo è stata compiuta un'analisi approfondita della procedura di boot e della configurazione dei registri interni dello SPARC.

La configurazione dello SPARC core è stata affrontata su due livelli: innanzitutto si è verificato che la teoria combaciasse sia con le configurazioni impostate nel boot di default sia col processore fisico; si sono cercati poi riscontri in simulazione, per valutare anche le conseguenze (*side effects*) delle configurazioni. In fase di studio e simulazione della procedura di boot, sono state riscontrate delle incongruenze e la mancata configurazione di alcune caratteristiche. Nel presente capitolo verrà affrontato lo studio della procedura di boot di default dell'S1 Core, verranno fatte considerazioni e proposte modifiche facendo riferimento alla teoria sui registri. Alla quasi totalità di registri e meccanismi trattati in questa sezione, è dedicato un corrispondente paragrafo teorico nel capitolo 7. La trattazione su configurazioni mancanti e discussione sul programma da eseguire dopo la configurazione di boot (che sia esso caricato in ROM o in RAM), vengono esposti in un successivo capitolo.

Va precisato che è stato supposto che la descrizione dell'hardware fosse corretta, essendo quella dell'S1 Core una distribuzione stabile, usata in diversi progetti.

5.1 Simulazioni di base dell'S1 Core con Icarus Verilog

Si assume qui che si abbia a disposizione un sistema Linux configurato con i programmi ed i pacchetti dichiarati nella procedura di installazione, vedi §4.1. Verrà usato il programma opensource gtkwave per visualizzare le forme d'onda.

Si assume che il pacchetto .tar dell'S1 Core sia stato scompattato nella directory home dell'utente, sotto il path ~/s1_core e che il package dell'OpenSPARC T1 sia scompattato sotto il path ~/opensparc_t1.

Le simulazioni delle tre versioni dell'S1 Core non sono andate a buon fine. Qui verranno descritte le problematiche inerenti all'avere una configurazione dei tool e dei file che consenta al simulatore di fornire gli output, che verranno posti di default nella cartella $S1_ROOT/run. Inoltre, verranno date delle indicazioni di massima sull'interpretazione di alcuni segnali del Wishbone bus direttamente messi a disposizione nel pacchetto dell'S1 Core. Si dichiarerà quali altri segnali

sono ritenuti utili per la soluzione di problemi che risultano evidenti anche ad una prima analisi delle forme d'onda.

5.1.1 S1 Core Elite Edition

Prima di tutto, si ha bisogno che le variabili d'ambiente bash siano configurate: dopo aver configurato il file ~/s1_core/sourceme, vedi §4.2.1, si faccia il source del file

```
usr@name$ cd ~/s1_core
usr@name:~/s1_core$ source sourceme
```

Ora si può lanciare lo script di riduzione del core. Si sceglie qui di lanciare la riduzione *Elite Edition*, nella quale il core gestisce 4 thread e ha entrambe le cache di primo livello. Aggiorniamo inoltre i file che servono per creare il progretto nei tools di simulazione e sintesi e detengono la lista dei file appartenenti al core.

```
usr@name:~/s1_core/tools/bin$ update_sparccore -ee
usr@name:~/s1_core/tools/bin$ update_filelist
```

Ora il design dovrebbe essere pronto. Si deve ora avere operative le immagini di ROM e RAM, con i relativi programmi: di default, lanciando lo script compile_test, vengono scritti i files $S1_ROOT/tools/boot/rom_harness.hex e $S1_ROOT/tools/ram_harness.hex. Il programma di boot, inserito in ROM è nel file $S1_ROOT/tools/boot/boot.s, mentre quello che viene inserito in RAM è $S1_ROOT/tools/hello.c. A patto di aver correttamente installato il cross-compilatore, per generare le immagini delle memorie basta lanciare lo script compile_test, che si aspetta come parametro in ingresso il nome, privo di estensione, del file contenente il codice-sorgente C, del programma che verrà inserito in RAM.

```
usr@name:~/s1_core/tools/bin$ compile_test hello   #senza
   estensione .c
```

Ora il pacchetto dovrebbe essere configurato, e dovrebbe bastare il lancio dei due tools build_icarus e run_icarus per ottenere la simulazione dell'S1 Core. Lanciando build_icarus, si ottiene un errore in fase di compilazione del verilog, a causa dell'assenza del modulo dffr:

```
usr@name:~/s1_core/tools/bin$ build_icarus
        Building testbench using Icarus simulator...
        ~/s1_core/hdl/rtl/sparc_core/sparc_ifu_thrcmpl.v
           :205: error: Unknown module type: dffr
        38 error(s) during elaboration.
        *** These modules were missing:
                        dffr referenced 1 times.
        *** Done!
```

Si tratta di un modulo di libreria. Ho trovato questo modulo, nella gerarchia dei moduli verilog dell'OpenSPARC T1.5 publicata sul web, vedi 3.1.2.7. Il modulo, nel design dell'OpenSPARC, appartiene al file swrvr_clib.v, e quindi ho deciso di aggiungerlo all'identico modulo dell'S1_Core presente in $S1_ROOT/hdl/rtl/sparc_core/swrvr_clib.v. Il modulo è riportato qui di seguito.

```
module dffr (din, clk, rst, q, se, si, so);
//synopsys template
parameter SIZE = 1;input [SIZE-1:0] din ; // data in
input clk ; // clk or scan clk
input rst ; // reset
output [SIZE-1:0] q ; // output
input se ; // scan-enable
input [SIZE-1:0] si ; // scan-input
output [SIZE-1:0] so ; // scan-output
reg [SIZE-1:0] q ;
`ifdef NO_SCAN
always @ (posedge clk)
        q[SIZE-1:0] <= ((rst) ? {SIZE{1'b0}} : din[SIZE-1:0] );
`else// Scan-Enable dominates
always @ (posedge clk)
q[SIZE-1:0] <= se ? si[SIZE-1:0] : ((rst) ? {SIZE{1'b0}} : din[
   SIZE-1:0] );
assign so[SIZE-1:0] = q[SIZE-1:0] ;
`endif
endmodule // dffr
```

A questo punto, eseguendo nuovamente il build del progetto, non si dovrebbero ottenere errori:

```
usr@name:~/s1_core/tools/bin$ build_icarus
        Building testbench using Icarus simulator...
        Done!
```

Ora che lo script build_icarus (vedi §4.2.5.3) ha inizializzato lo script S1_ROOT/run/sim/icarus/testbench necessario all'esecuzione della simulazione, si può lanciare la simulazione del core. La simulazione seguirà tempi e specifiche dichiarati nel testbench, vedi §4.2.3.2.

Per simulare l'S1_Core, lanciare:

```
usr@name:~/s1_core/tools/bin$ run_icarus
        Simulation completed!
                To see the output:
        less /home/xar/s1_core/run/sim/icarus/sim.log
                To watch the waveforms:
        gtkwave /home/xar/s1_core/run/sim/icarus/trace.vcd
```

5.1.2 S1 Core SE e ME con Icarus Verilog

Per avviare le simulazioni delle versioni *Single Thread Edition* e *Mini Edition* (vedi §4.2.5.3) valgono le medesime considerazioni fatte per la versione EE, vedi §5.1.1. A differenza della versione completa, quando si lancia il tool run_icarus, Icarus Verilog non riesce ad eseguire la simulazione a causa di un errore in una sua funzione interna. In ogni caso, i design creati con update_sparccore sembrano validi, in quanto sono stati simulati su Modelsim con risultati confrontabili con la simulazione della versione EE, sia su Icarus che su Modelsim. I due simulatori, a parità di file rom_harness.hex e ram_harness.hex, e dunque di istruzioni da eseguire, danno risultati identici.

Per costruire la versione SE ed ME dell'S1 Core, dopo aver configurato il file $S1_ROOT/sourceme, basta eseguire in sequenza i comandi seguenti:

```
usr@name$ cd ~/s1_core
usr@name:~/s1_core$ source sourceme       # per variabili
   di shell
usr@name:~/s1_core$ update_sparccore -se  # -me per la Mini
   -Edition
```

```
usr@name:~/s1_core$ update_filelist       # aggiorna la
   lista dei file del progetto
usr@name:~/s1_core$ compile_test hello  # compila la
   procedura di boot e il programma da inserire in ram
```

Anche in questi design mancherà il modulo dffr dalla libreria $S1_ROOT/hdl/rtl/sparc_core/swrvr_clib.v e bisognerà inserirlo perchè il progetto sia compilabile. Una volta aggiunta, vedi §5.1.1, si può compilare il progetto:

```
usr@name:~/s1_core$ build_icarus
```

Ora non resta che lanciare la simulazione del progetto per Icarus, il quale genera un errore, che viene riportato di seguito al comando di run:

```
usr@name:~/s1_core$ run_icarus
internal error: 11vvp_fun_not: recv_vec4_pv(1'bX, 0, 1, 4)
   not implemented
vvp: vvp_net.cc:2465:
virtual void vvp_net_fun_t::recv_vec4_pv(vvp_net_ptr_t,
   const vvp_vector4_t&, unsigned int, unsigned int,
   unsigned int, void**): Assertion '0' failed.
~/s1_core/tools/bin/run_icarus: riga 10: 15947 Aborted
```

Riguardo l'errore non è stata tentata alcuna strada per risolverlo. Poichè va a buon fine la fase di build, si suppone che le relazioni tra i moduli Verilog siano corrette, e si è tentata con successo la simulazione su Modelsim. Tutte e tre le versioni dell'S1 Core sono risultate simulabili con Modelsim.

5.1.3 Simulazione con Modelsim

Su Modelsim, tutte e tre le versioni dell'S1 Core sono completamente simulabili.

Innanzitutto bisogna costruire il design desiderato dell'S1 Core e per farlo bisogna avere un sistema Linux configurato come dichiarato nel §4.1. Qui si prenderà ad esempio la versione Mini del processore. Aprire una shell, configurare le variabili d'ambiente e lanciare la riduzione del core:

```
usr@name:~$ source ~/s1_core/sourceme
usr@name:~$ update_sparccore -me
usr@name:~$ update_filelist
```

```
usr@name:~$ compile_test hello
```

L'esecuzione di update_filelist non è necessaria per la simulazione su Modelsim, ma è utile se sullo stesso design si vogliono avviare test con altri simulatori (per esempio Icarus Verilog), vedi §4.2.5.3. Lo script compile_test inizializzerà le memorie RAM e ROM.

Ora bisogna copiare il modulo dffr nel file $S1_ROOT/hdl/rtl/sparc_core.

A questo punto il design è stato creato: non resta che aprire Modelsim, creare un progetto e importare nel progetto il design. Su Linux, si potrebbe creare la libreria di progetto tramite comandi da shell, ma si sceglie qui la via grafica, utilizzabile su qualsiasi piattaforma su cui l'ambiente grafico di Modelsim possa essere visualizzato. Per comodità, i path saranno scritti indicando con S1_ROOT la directory dove è stato estratto il pacchetto.

Cliccare su "`file -> new -> project`" per avviare un nuovo progetto; definire il nome del progetto, per esempio s1_core_me, il path, per esempio $S1_ROOT/run/sim/modelsim e il nome di libreria, per esempio work.

Si può ora scegliere se importare i file di design lasciandoli nelle loro locazioni oppure se copiarli nel directory di progetto. Scegliamo la seconda strada, immettendo tutti i files nella directory work. Diversamente, scegliendo per esempio di separare i files di libreria dal design, dovremmo poi modificare le direttive di "`include`" dei top-files per indicare i path assoluti delle librerie.

Cliccando col tasto destro nella finestra di progetto, scegliere "`add to project -> existing File...`", e nella finestra di controllo selezionare "`Folder = Top Level`" e attivare la spunta su "`Copy to project directory`" per ogni file importato. Nella tabella 5.1, si indicano i path dai quali i file sono da importare nel progetto; per singoli file viene dichiarato il path completo. Durante l'importazione, selezionare come "`Files of type: = All files (*)`".

Vi sono due file, t1_defs.h e s1_defs.h, che contengono le dichiarazioni dei parametri usati rispettivamente nei moduli dello SPARC core e nei moduli aggiuntivi dell'S1 Core: a causa della loro estensione non vengono riconosciuti come moduli Verilog da Modelsim. Ci sono due modi perchè vengano riconosciuti e la compilazione vada a buon fine:

- selezionare il file fare click su "`View -> Properties...`", dunque fare click su "`Change Type`", selezionare "`Verilog -> OK -> OK`".

Tabella 5.1: Files da importare nel progetto Modelsim.

Sono presenti file non *.v	Directory
no	S1_ROOT/hdl/behav/sparc_libs/*.*
si	S1_ROOT/hdl/rtl/s1_top/*.*
no	S1_ROOT/hdl/behav/testbench/testbench.v
no	S1_ROOT/hdl/behav/testbench/mem_harness.v
si	S1_ROOT/tests/ram_harness.hex
si	S1_ROOT/tests/boot/rom_harness.hex
no	S1_ROOT/hdl/rtl/sparc_core/*.*

- Un altro modo è quello di entrare nella cartella $S1_ROOT/hdl/rtl/s1_top/. Rinominare i due file t1_defs.h e s1_defs.h in t1_defs.v e s1_defs.v. Solo ora importarli nel progetto, sempre mettendo la spunta all'opzione di copia. Bisogna ora cambiare nei moduli "top" del progetto le referenze ai nomi dei due moduli: Modelsim può dare una mano, in quanto, compilando dirà in quali files non sono stati definiti i parametri richiesti. Nei moduli che generano eccezioni, basta modificare la linea

  ```
  `include "s1_defs.h"
  ```

  modificando l'estensione in ".v". I moduli sono in tabella:

parametri prelevati da	modulo richiedente, da modificare
t1_defs.h	s1_defs.v
s1_defs.h	spc2wbm.v
s1_defs.h	s1_top.v
s1_defs.h	testbench.v
s1_defs.h	rst_ctrl.v

Se ora si lancia la simulazione, nonostante vada a buon fine, si ottengono delle forme d'onda piatte. Questo è avvviene perché le memorie RAM e ROM sono inizializzate solo se nel modulo mem_harness è definito il parametro "DEBUG". Si può aggirare il problema in due modi convenienti a seconda delle esigenze:

- se si desidera che durante la simulazione venga stampato sulla shell di modelsim il corrispondente del file di log di Icarus, aprire il file s1_defs e

includere la stringa

```
`define DEBUG 1
```

assieme alle altre. Aprire il modulo mem_harness e, all'inizio del modulo, includere s1_defs:

```
`include "s1_defs.h"
```

o con estensione ".v" a seconda della scelta adottata in precedenza. Salvare e ricompilare i moduli.

- se invece si desidera solamente avere la RAM e ROM inizializzate, senza che vengano stampate le informazioni di debug, aprire il file mem_harness.v e cercare la stringa "`$readmemh`", intorno alla riga 80. Si può notare come il blocco initial nel quale è dichiarata sia racchiuso all'interno di un controllo `` `ifdef `` – `` `endif ``: basta commentare le due righe `` `ifdef DEBUG `` e `` `endif `` per ottenere l'effetto desiderato. Salvare e ricompilare il modulo.

Una accortezza che può risultare comoda in Modelsim è cambiare nel testbench il comando
"`$finish`" con "`$stop`".

L'ambiente di simulazione è pronto, si può lanciare la simulazione cliccando su `Simulate -> Start Simulation...` , scegliendo il modulo `work/testbench -> OK`. A questo punto scegliere dal menu `View -> Wave` per visualizzare le forme d'onda. Riguardo la scelta delle forme d'onda da visualizzare, potrebbe essere utile esaminare i segnali riportati in tabella 5.2.

In fase di testing, su Linux, può risultare conveniente utilizzare lo script compile_test per modificare le istruzioni di ROM e RAM. Si può fare in modo che rom_harness.hex e ram_harness.hex della cartella di progetto Modelsim siano dei collegamenti a file, in modo da poterli manipolare con i tools forniti con il pacchetto S1 Core. Per farlo, aperta una shell, portarsi al path del progetto Modelsim e creare il link:

```
usr@name:~$ source ~/s1_core/sourceme
usr@name:~$ cd $S1_ROOT/run/sim/modelsim
usr@name:~/s1_core/run/sim/modelsim$ ln -fs ../../../tests/
   ram_harness.hex ram_harness.hex
```

```
usr@name:~/s1_core/run/sim/modelsim$ ln -fs ../../../tests/
   boot/rom_harness.hex rom_harness.hex
```

Ora si potrà modificare il codice macchina da eseguire editando i file hello.c e boot.s, e cross-compilandoli con il tool compile_test.

Se si volessero usare files di nome diverso da rom_harness.hex e ram_harness.hex, bisognerebbe indicare il nuovo nome alla fine del testbench settando il parametro memfilename.

5.1.4 Breve analisi della simulazione di default

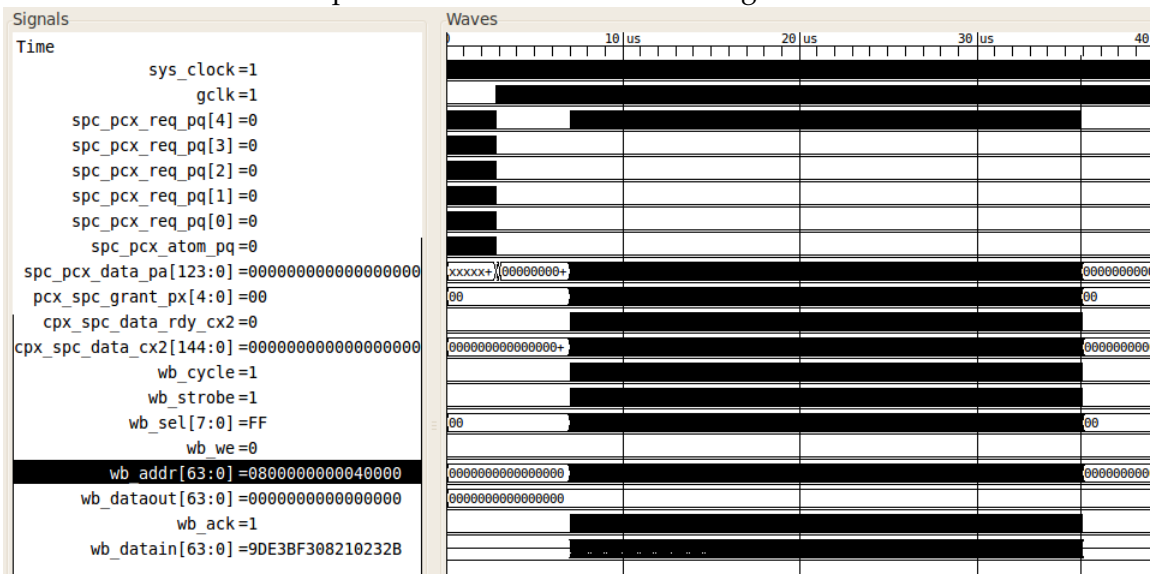
Nel file di log, vi sono informazioni su tutte le transazioni avvenute sul Wishbone-bus, pacchetto per pacchetto. Aprendolo con un editor di testi, si può notare come siano quasi tutte richieste di fetch di istruzioni da ROM, come indicato dalla stringa "Fetch for Boot is 1": altri due indizi sono il *Non Cacheable* bit alto e il tipo di pacchetto: IMISS_RQ per le richieste dello SPARC core e IFILL_RET per i pacchetti di ritorno (dalla ROM alla CPU), tipiche del fetch da ROM dello SPARC V9. L'address toglie ogni dubbio: il primo bit dell'indirizzo completo è a 1 (ovvero la regione puntata è lo spazio di I/O) e i primi 8 bit del *phisical-address* (ultimi 40 bit dell'istruzione) valgono $0xFF$. Inoltre il pacchetto di ritorno, contiene una sola istruzione, da 32 bit: gli altri bit di informazione sono a 0. Si può già dire che il fetch delle istruzioni da ROM sembra essere coerente. Gli ultimi 2 pacchetti (2 di richiesta e 2 di risposta) sono invece richieste di fetch da RAM. Si può notare anche qui che il *Non Cacheable* bit è a 1, SEL vale $0 \times FF$, dunque il pacchetto di ritorno conterrà 4 istruzioni (32 bit ciascuna, per un totale di 128bit) e il *return-packet* non sarà colmato con una stringa di '0' come per il fetch da ROM. Inoltre le linee di cache L1 sono da 32 byte (256 bit), che corrispondono a 8 istruzioni (da 32 bit): la richiesta di IMISS da RAM genera due richieste consecutive sul bus (che trasporta massimo 128 bit per volta), e solo nel secondo pacchetto di ritorno delle 2 transazioni, alle righe 30131 e 30178, "Atomic LD/ST or 2nd IFill Packet" è a 1. In ogni caso, la simulazione si ferma dopo aver fatto il fetch del primo blocco di istruzioni da RAM, mentre la ram contiene un programma con altre istruzioni. Sembra che il bus risponda bene alle richieste di *miss* dalla I-Cache, sia da ROM che da RAM.

Per avere una visione completa della simulazione, si può usare gtkwave offrendo in ingresso al programma il file trace.vcd creato in simulazione, come suggerito al termine della simulazione:

```
usr@name:~$ gtkwave $S1_ROOT/run/sim/icarus/trace.vcd
```

Bisogna ora aggiungere i segnali di cui si vuole visualizzare le forme d'onda: alcuni sono forniti già pronti nel pacchetto dell'S1 Core. Per aprirlo si può aprire il menu 'file' e scegliere 'read save file', dunque aprire il file $S1_ROOT/tools/src/gtkwave.sav. Si possono in questo modo visualizzare le forme d'onda utili a comprendere quanto avviene sul Wishbone bus. Le forme d'onda disegnate mostrano le variazioni di alcuni segnali all'interfaccia della CPU SPARC: sullo stato interno del processore, si possono solo ricavare alcune informazioni per via deduttiva. In figura 5.1, il risultato della simulazione.

Figura 5.1: Simulazione di default dell'S1 Core. Alcuni segnali sono 'pieni' perchè variano molto in fretta rispetto alla risoluzione dell'immagine.



La simulazione, come atteso dal testbench (vedi §4.2.3.2), inizia con il reset che si alza ed il clock globale (gclk) è fermo. Quando il segnale di reset si abbassa, dopo 1000ns, è il reset controller a prevalere nella simulazione fino a quando non è terminata la procedura di reset e iniziano le richieste al bus (i segnali sono 'pieni'). Si può notare subito che qualcosa non va in RAM: il programma inserito in RAM dovrebbe infatti eseguire due istruzioni di STORE e due di LOAD sullo stesso indirizzo di memoria, e dunque dovrebbero variare due volte il segnale wb_we

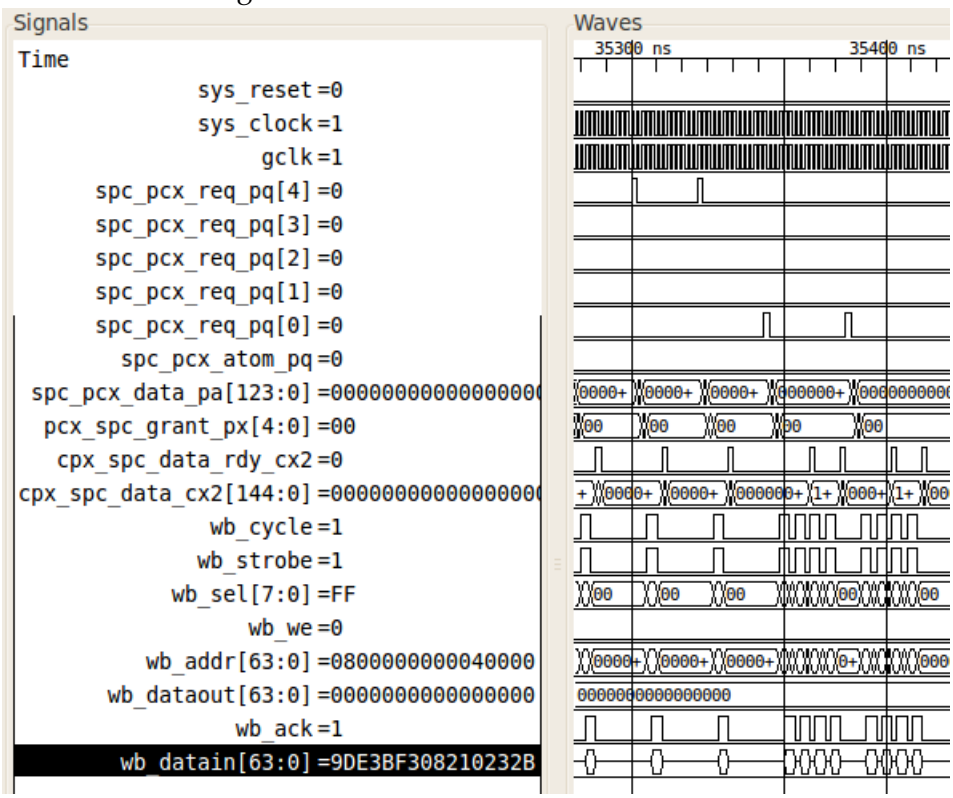
(*write enable*) e wb_dataout, mentre rimangono immutati. Il segnale di we alto, indica che sta avvenendo una scrittura in RAM (dovuta ad una STORE), mentre il segnale wb_dataout detiene il dato da immettere in memoria, proveniente dallo SPARC core.

Per visualizzare le richieste della CPU alla RAM, bisogna ingrandire le ultime variazioni dei segnali (ad esempio wb_datain, che indica quali informazioni vanno da RAM e I/O verso il core), in fig.5.2. Il *marker*, la seconda linea verticale che attraversa le forme d'onda, è posizionato sulla prima transazione dal primo blocco di RAM: i dati relativi ai segnali si riferiscono al punto dove il marker è posizionato, e si può notare come wb_addr valga 0x0800000000040000. Si nota come le variazioni sul bus siano molto più repentine di quelle del fetch da ROM (a sinistra del *marker*): mentre da ROM un pacchetto CPX contiene una sola istruzione, in risposta alle richieste di fetch da RAM viene riempito il buffer CPX con 4 istruzioni (128 bit, gli altri 17 bit sono di controllo). Come ci si aspettava dalla lettura del file di log, viene fatta la richiesta e transazione di due soli pacchetti (spc_pcx_req_pq si alza ad ogni richiesta da parte della CPU), vedi fig. 5.2. Stando alle informazioni messe a disposizione da questi segnali, non si sa quale sia l'ultima istruzione che viene eseguita (e che probabilmente genera una eccezione): per saperlo bisognerebbe avere a disposizione informazioni sullo stato del PC (Program Counter) e del TPC (Thread Program Counter). Inoltre sarebbe utile scoprire se e quale eccezione viene generata, informazione che potrebbe essere a disposizione nei registri del modulo di gestione delle trappole (*trap*).

Ingrandendo l'immagine sulla sinistra e andando a visualizzare il primo valore non nullo di wb_addr, si vede come il primo fetch non venga fatto della prima parola di ROM, bensì della nona, iniziando dal *Phisical Address* (PA) 0x800000FFF0000020[2]: questo fa si che le prime 8 istruzioni di ROM (le prime 4 righe di rom_harness.hex, che contiene 2 istruzioni per riga) non vengano eseguite. Questa è una particolarità importante del *Power On Reset* (POR) del core SPARC V9: nella costruzione della procedura di boot salvata in ROM bisogna tenerne conto, per esempio inserendo 8 istruzioni valide (devono essere valide perchè altrimenti il compilatore genererà un errore) all'inizio del codice di ROM, in tests/boot/boot.s, con la consapevolezza che non verranno eseguite; potrebbero

[2]Il Program Counter avanza di 4 in 4, perchè le istruzioni sono da 32 bit, mentre gli indirizzi hanno granularità del byte. Gli ultimi 4 bit del PC e degli indirizzi richiesti al bus nel fetch delle istruzioni saranno dunque sempre uno tra i valori 0x0, 0x4, 0x8 e 0xC.

Figura 5.2: Ingrandimento alla fine della simulazione. A sinistra della linea di divisione, sono richiesti fetch da ROM, a destra da RAM. I dati disponibili sulla colonna '*Signals*' si riferiscono ai valori dei segnali sul *marker*.



essere, per esempio, delle NOP (*No OPeration*).

5.1.5 Ulteriori segnali utili in fase di Test

Come cercato di evidenziare nell'analisi della simulazione di base dell'S1 Core EE, da RAM vengono caricate solo le prime 8 istruzioni, e per sondarne le cause è necessario reperire dal design dello SPARC core ulteriori informazioni. La CPU ha un design molto vasto e nella scrittura del Verilog viene usata una convenzione sui nomi dei segnali in modo che ogni nome di segnale, oltre l'identificativo che ne specifica l'utilità, indichi quale è l'unità che l'ha generato oppure il tipo di registro di cui porta il valore. Spesso è indicato alla fine del nome a quale stadio della *pipeline* di esecuzione si riferisce, vedi fig.7.2. Nonostante questa accortezza, i nomi dei segnali restano composti di sigle, e dunque sostanzialmente criptici. Lo stesso segnale può cambiare nome, passando da una unità all'altra, o perchè

viene usato in due o più stadi della *pipeline*. I registri spesso non sono definiti come un unico segnale, ma ogni porzione significativa assume un nome. Salterà all'occhio l'abbondanza di segnali: una unità come la LSU (Load Store Unit) ne istanzia quasi 1000.

Per questi motivi si enunciano nella tabella 5.2 i segnali che sono stati di particolare utilità in fase di test dell'S1 Core, e dove reperirli nel design.

Tabella 5.2: Alcuni segnali interni dell'OpenSPARC T1 core.

Nome	**Path**	**Descrizione**
PC		
pc_f	s1_top_0.sparc_0.ifu.fdp.pc_f	Program Counter, fetch stage.
pc_s	s1_top_0.sparc_0.ifu.fdp.pc_s	Program Counter, thread selection stage.
pc_d	s1_top_0.sparc_0.ifu.fdp.pc_d	Program Counter, decode stage.
pc_e	s1_top_0.sparc_0.ifu.fdp.pc_e	Program Counter, execution stage.
pc_m	s1_top_0.sparc_0.ifu.fdp.pc_m	Program Counter, memory stage.
pc_w	s1_top_0.sparc_0.ifu.fdp.pc_w	Program Counter, write-back stage.
TIR		
tir	s1_top_0.sparc_0.ifu.fdp. t0_inst_reg.q	Thread Instruction Register, è il valore immagazzinato nel registro.
TRAPS		
pending thread 0	s1_top_0.sparc_0.tlu.tcl. dffe_pending_thrd0.q	Registra la trappola corrente.
tl	s1_top_0.sparc_0.tlu.tcl.dffe_tl0.q	Trap Level Zero register.
htba_0	s1_top_0.sparc_0.tlu.tdp. dffe_true_htba0.q	Hyperpriviledge Trap Base Address 0, bit [63:13]
WINDOW MANAGEMENT REGS		
cwp	s1_top_0.sparc_0.exu.rml. cwp.dff_thr0.q	Current Window Pointer
cansave	s1_top_0.sparc_0.exu.rml. cansave_reg.dff_reg_thr0.q	Can Save register

Nome	Path	Descrizione
canrestore	s1_top_0.sparc_0.exu.rml. cansrestore_reg.dff_reg_thr0.q	Can Restore register
cleanwin	s1_top_0.sparc_0.exu.rml. cleanwin_reg.dff_reg_thr0.q	Clean Window
otherwin	s1_top_0.sparc_0.exu.rml. otherwin_reg.dff_reg_thr0.q	Other Window
hi_wstate	s1_top_0.sparc_0.exu.rml. hi_wstate_reg.dff_reg_thr0.q	Window State register, most significant bits
lo_wstate	s1_top_0.sparc_0.exu.rml. lo_wstate_reg.dff_reg_thr0.q	Window State register, least significant bits
IRF (INTEGER REGISTER FILE)		
registerXX XX=00-31	s1_top_0.sparc_0.exu.irf. bw_irf_core.registerXX.rd_data	Register: [0-7] Global. [8-15] Out. [16-23] Local. [24-31] Input.

5.1.6 Scelta del simulatore: considerazioni e problematiche

Esiste la versione XE di Modelsim, gratuita solo per Windows, fornita dalla Xilinx. E' stato istanziato con successo l'S1 Core su questa versione. Le simulazioni danno risultati identici che con Icarus Verilog. Non è stata però usata estensivamente nel portare avanti la presente tesi per due ragioni: una di tempo e l'altra di usabilità.

Su Modelsim XE, la compilazione dei 140 file impiega 5 minuti circa e la simulazione dura circa 1 ora. Sulla medesima macchina, Icarus Verilog impiega pochi secondi per la compilazione e 2-3 minuti circa per la simulazione. Le prestazioni ridotte di Modelsim XE, sono dovute al fatto è una versione gratuita del software commerciale.

Non sono stati trovati cross-compilatori, sia C che assembly da X86 a UltraSPARC per Windows. Avendoli, la cross-compilazione della procedura di boot e il programma da inserire in RAM sarebbe dovuta essere fatta da zero, non avendo a disposizione script o procedure già automatizzate per Windows. Il tutto comporta anche una conoscenza non banale del cross-compilatore perchè, per dirne una su tutte, il link del codice è da ri-definire, perchè l'S1 ha RAM e ROM di di-

mensioni ridotte rispetto agli standard dell'OpenSPARC T1. Detto questo, sono state comunque eseguite delle prove su Modelsim XE in ambiente Windows durante il testing della procedura di boot: dato il risultato della simulazione e deciso dove apportare le modifiche al codice, bisogna costruire le istruzioni da cambiare bit a bit, trasformarle in esadecimale e riconoscere le versioni esadecimali delle istruzioni circostanti per inserirle nel punto voluto. Questo sistema è risultato poco utile, dunque è stato abbandonato presto a favore di Icarus Verilog e Ubuntu Linux. Si sarebbe potuto, è vero, cambiare da un sistema operativo all'altro, compilando su Linux e eseguendo su Windows, ma, visti i risultati simili tra Icarus e Modelsim, e anche tra le simulazioni sulle varie versioni del core, si è deciso di usare prevalentemente Ubuntu e Icarus.

5.2 Introduzione ai test sul boot

Le istruzioni di LOAD e STORE, nella simulazione di default non vengono eseguite. In realtà non si sa quante altre, sia all'interno della procedura di boot e del programma di RAM non siano eseguite correttamente. Fiduciosi del fatto che, mentre la procedura di boot è stata assemblata a mano, il programma in RAM è il prodotto finito di una cross-compilazione, si è stati portati a pensare che fosse la procedura di boot ad avere delle carenze. Detto questo, però, il programma si ferma in RAM.

Si è deciso di ispezionare il codice assembly che genera la procedura di boot ed il programma in ram. Come si vedrà nelle simulazioni, nella procedura di boot non sempre le operazioni sui registri sono logicamente coerenti, e la simulazione si blocca alla prima istruzione di RAM, che, in codice SPARC assembly, recita:

```
40000: 9d e3 bf 30     save  %sp, -208, %sp
```

Nella descrizione e nella modifica della procedura di boot, verranno richiamate nozioni su registri e configurazioni dello SPARC core, alcune delle quali vengono riportate nel capitolo 7.

La procedura di simulazione, una volta configurato il package (vedi §4.1) ed eseguita una riduzione del core (vedi §5.1.1), consiste di tre passaggi:

- modifica dei files boot/boot.s e hello.s della cartella tests. Il file hello.s si può ottenere come descritto nel parargrafo 4.2.5.3;

- lanciare lo script di cross-compilazione compile_test;
- lanciare la simulazione con lo script run_icarus o lo script di run del simulatore in uso.

Nella maggior parte delle simulazioni, è usato Icarus Verilog in ambiente Linux, sulla versione EE dell'S1 Core, il cui design comprende le caches di primo livello e 4 thread. Nonostante questo, si lavora su un solo thread.

La ricerca dei registri di configurazione nel design, risulta molto complessa. Poichè molti di essi sono indirizzabili tramite istruzioni, è più semplice, per conoscerne il valore, lanciare una istruzione di lettura dal registro di interesse verso l'IRF (Integer Register File) ed avere in simulazione solo i segnali dell'IRF. Verrà usato frequentemente questo escamotage, in quanto il processore non ha mai mostrato incongruenze nell'esecuzione di istruzioni su registri interni o nell'esecuzione di operazioni aritmetiche ed i dati ottenuti sono congrui con la teoria. Un breve richiamo teorico sui registri che vengono usati durante la trattazione sul boot, è sviluppato nella sezione 7.5.

Quando la CPU inizia ad eseguire le istruzioni, è stato appena lanciato un Power-On-Reset. I registri si trovano in una situazione di partenza descritta nel documento "*UltraSPARC architecture 2005*" (vedi §3.1.2.4), nel capitolo riguardante i reset. Un riassunto della tabella potrebbe essere "*Unknown*": quasi tutti i registri di stato sono di valore indeterminato, e bisogna assegnargli un valore proprio durante la procedura di boot. Il codice del boot è eseguito in modalità hyperpriviledge, per configurare i registri in modo che lo stato dei Virtual Processor in esecuzione sia coerente con l'hardware prima che il SO (ad accesso priviledge) venga eseguito (da RAM).

5.3 La procedura di boot di default dell'S1 Core

La procedura di boot dell'S1 Core è situata nella directory $S1_ROOT/tests/boot. Il file boot.s è la procedura scritta in assembly. Dalla sua compilazione e dump si ricavano le istruzioni in formato esadecimale utili alla simulazione, e sono inserite nel file rom_harness.hex della stessa cartella.

Sfrondato dei commenti non indispensabili, la procedura di boot dell'S1 core si presenta come di seguito.

5.3.1 LSU Diagnostic Register

Il LSU Diagnostic Register è descritto nel paragrafo 7.5.4.

5.3.1.1 Configurazione dal file boot.s di default

```
/*Starting Boot Procedure*/

!! Set the LSU Diagnostic Register to enable
!! all ways for L1-icache and L1-dcache
!! and using the "random␣replacement" algorithm
        mov   0x10, %g1
        stxa  %l1, [%g1] (66)
```

L'istruzione di MOV, imposta il valore del registro *g1* a 10_{16}: è una istruzione compatta, equivalenta ad una OR con campo immediate: `or %g0,0x10, %g1`.

Nella STORE ALTERNATE, il contenuto del registro *l1* è scritto sul LSU Diagnostic Register, che ha *asi* 42_{16} con *virtual address* 10_{16}.

5.3.1.2 Verifica: teoria e simulazione

Sono queste le prime istruzioni della procedura di boot e le prime due che in assoluto il processore dovrebbe eseguire. Non vengono però eseguite, perchè il primo fetch da ROM è impostato sempre con *virtual address* 20_{16}. Poiché la memoria ha linee da 64 bit ed ogni istruzione è da 32 bit, allora la prima istruzione di ROM eseguita è la nona: le prime 4 linee vengono ignorate. Ci si aspetterebbe che le prime 8 istruzioni di ROM siano delle NOP (no-operation), o comunque che non siano utili alla configurazione.

Nella seconda istruzione il registro l1 viene inserito nel LSU Diagnostic Register senza essere prima inizializzato. Qui è appena terminata la procedura di reset (Power-On Reset) ed i registri interi sono tutti annullati. Dunque viene scritto 0x0 nel registro LSU Diagnostic Register: è una operazione inutile, perchè dopo un POR ha valore di default 0. Si è questo verificato in simulazione: è bastato infatti inserire una istruzione di lettura dal LSU Diagnostic Register e salvarla in un registro globale:

```
mov   0x10, %g1
ldxa  [%g1] (0x42), %g2 ! Simulazione: g2 = 0x0
```

Nel commento è dichiarato che il LSU Diagnostic Register si vuole configurato in modo che sia la D-Cache che la I-Cache abbiano un metodo di rimpiazzamento "Random Replacement", quello di funzionamento normale per le due cache. Per farlo, *l1* deve avere il valore 00_{16}, valore che il registro ha già di default.

E' dunque plausibile anche che le due istruzioni possano non essere eseguite per nulla, basandosi solo sui valori di default dei registri.

5.3.1.3 Modifiche proposte

In seguito alle considerazioni di cui sopra, sembra corretto corredare le due istruzioni in esame in modo da tener conto dell'inizio della procedura all'indirizzo 20_{16} e la comunque corretta inizializzazione a zero del registro LSU Diagnostic Register.

```
/*Starting Boot Procedure*/
! Il primo fetch e' all'indirizzo 0x20: aggiunte 8 NOP
1-4     nop;    nop;    nop;    nop;
5-8     nop;    nop;    nop;    nop;

/*  Set the LSU Diagnostic Register to enable all ways
 *  for L1-icache and L1-dcache and using the
 *  "random replacement" algorithm  */
9       clr   %l1                ! %l1 = 0
10      mov   0x10, %g1
11      stxa  %l1, [%g1] (66)    ! LSU_DIAG_REG = b00
```

5.3.2 LSU CONTROL REGISTER

Il LSU Control Register è descritto nel paragrafo 7.5.5.

5.3.2.1 Configurazione dal file boot.s di default

```
! Set the LSU Control Register to enable
! L1-icache and L1-dcache
        mov  3, %l1
        stxa  %l1, [%g0] (69)
```

Viene impostato il valore di *l1* a *3*. Viene poi copiato nel LSU Control Register, corrispondente all'asi 45_{16}, con indirizzo virtuale 0_{16}. Il registro *g0* contiene sempre zero: i tentativi di scrittura vengono ignorati senza generare eccezioni.

5.3.2.2 Verifica: teoria e simulazione

Vengono, in questa istruzione, abilitate le caches di primo livello, settando ad 1 i bit *IC* e *DC* del registro. Si ricorda qui che uno dei design forniti con l'S1 Core è privo di cache L1, e dunque in quel caso queste istruzioni non dovrebbero essere presenti.

Non vengono invece abilitate le MMU (bit 3° e 4°). Questo comporterà in simulazione che gli accessi alla RAM siano *non-cacheable*.

I bit di controllo del watchpoint sono settati a 0 di default, dunque disabilitati. Non ci sono indirizzi particolari che vogliono essere controllati.

5.3.2.3 Modifiche proposte

Si nota qui che, nonostante questa configurazione possa essere coerente con due delle 3 versioni (EE ed SE, in cui sono presenti le caches L1), nella simulazione di default queste istruzioni non sono eseguite perchè sono in ROM prima del Virtual Address 20_{16}. Dunque nella simulazione di default, sono disabilitate sia le cache L1 che le rispettive MMU.

E' dichiarato nella documentazione (vedi §3.1.2.4) che per rendere attive alle istruzioni seguenti le modifiche al bit DC bisogna successivamente lanciare una istruzione di FLUSH, DONE o RETRY prima di un accesso in memoria (LOAD/STORE). Sembra coerente rendere eseguibili queste due istruzioni, e lanciare una FLUSH subito di seguito. All'inserimento di tale istruzione, la simulazione si blocca, in quanto la gerarchia di memoria e la MMU non sono ancora configurate. Anche l'esecuzione delle istruzioni di DONE o RETRY lascia il processore in uno stato inconsistente: vengono scritti i registri STATE priviledge e hyperpriviledge con i valori salvati nei vettori-trappola, che però sono ancora indeterminati.

```
! Set the LSU Control Register to enable
! L1-icache and L1-dcache
12      mov  3, %l1
13      stxa  %l1, [%g0] (69)
```

5.3.3 HPSTATE

Il registro HPSTATE, *HyperPriviledge Status register*, è descritto nel paragrafo 7.5.1.

5.3.3.1 Configurazione dal file boot.s di default

```
!! Set hpstate.red = 0 and hpstate.enb = 1
        rdhpr %hpstate, %l1
        wrhpr %l1, 0x820, %hpstate
```

RDHPR e WRHPR sono istruzioni pensate per leggere e scrivere i registri dell' *hypervisor*, e sono utilizzabili solo se si esegue in *hyperpriviledge* (HP) *mode*. L'istruzione di read riporta nel registro *l1* il contenuto del HPSTATE, che detiene le principali configurazioni di funzionamento del Virtual Processor quando esegue in *HP mode*. La wrhpr fa la XOR tra i primi due argomenti (il secondo, in questo caso è un *immediate*) e scrive il risultato nel HPSTATE. 0x820 funziona da maschera per i bit da commutare, HPSTATE{11} e HPSTATE{5}. In sostanza *l1* detiene il vecchio valore del registro che si vuole aggiornare, il secondo argomento contiene la maschera per i bit che verranno cambiati grazie alla xor, ed il terzo argomento è di nuovo il registro che si vuole cambiare.

5.3.3.2 Verifica: teoria e simulazione

Lo stato iniziale del registro dopo il reset è riportato in tabella 5.3.

Tabella 5.3: Registro HPSTATE dopo il reset (POR) e dopo l'esecuzione dell'istruzione WRHPR.

Bit	Campo	Stato dopo POR	Stato dopo WRHPR	Nome esteso
11	enb	0	1	–
10	ibe	0	0	Instruction Breakpoint Enable
5	red	1	0	RED: Reset, Error e Debug
2	hpriv	1	1	Hyperpriviledge
0	tlz	0	0	Trap Level Zero

E' verificato in simulazione che i valori corrispondono. La modifica al HPSTATE.enb, è richiesta esplicitamente nella documentazione. Mettendo a 0 HPSTA-

TE.red si esce dal RED state, entrando in esecuzione puramente hyperpriviledge. Il bit HPSTATE.tlz non è necessario impostarlo a 1, in quanto non si ha nè la necessità di fare il descheduling del processo, essendo l'unico eseguito, nè si prospetta l'esecuzione (da RAM) di un sistema operativo; inoltre, non si ha un sistema di trappole configurato (vedi §6.4.1).

5.3.4 I-Cache e D-Cache SFSR

Il registro SFSR, *Sinchronous Fault Status Register*, è descritto nel paragrafo 7.5.6.

5.3.4.1 Configurazione dal file boot.s di default

```
!! Clear L1-icache and L1-dcache SFSR
        mov     0x18, %g1
        stxa    %g0, [%g0 + %g1] 0x50     !! IMMU SFSR=0
        stxa    %g0, [%g0 + %g1] 0x58     !! DMMU SFSR=0
```

Impostato il Virtual Address in *g1*, vengono annullati (viene inserito il valore di *g0*) i registri SFSR della I-MMU e della D-MMU. Corrispondono questi agli asi 50_{16} e 58_{16}, con VA 18_{16}.

5.3.4.2 Verifica: teoria e simulazione

I registri SFSR detengono il codice di errore corrispondente ad eccezioni che riguardano la gerarchia di memoria. Dovendo decidere il valore di default, dato che si suppone che le precedenti istruzioni non abbiano generato errore (e tale ipotesi è verificata in simulazione), sembra coerente assumere valore nullo.

5.3.5 SPARC Error Enable Register

Lo SPARC Error Enable Register è descritto nel paragrafo 7.10.

5.3.5.1 Configurazione dal file boot.s di default

```
!       SPARC Error Enable Reg.
! In file defines.h cregs_sparc_error_en_reg_r64:=3
! so the effect should be "trap on correctable error" and
! "trap on uncorrectable error"
```

```
sethi %hh(0x3),%g1
or    %g1,%hm(0x3),%g1
sllx  %g1,32,%g1
sethi %hi(0x3),%l1
or    %l1,%g1,%l1
or    %l1,%lo(0x3),%l1
stxa  %l1, [%g0] (75)   ! copy the content of the
        ! l1 register into the "SPARC Error Enable reg"
```

Le prime 6 istruzioni costruiscono per parti la costante 3_{16} sul registro *l1*, usando *g1* come appoggio. In sequenza, viengono settati i bit *[63:42]*, *[41:32]*, *[32:10]* e *[9:0]*.

L'ultima istruzione copia il contenuto di *l1* nello *SPARC Error Enable Reg*, indirizzato dall' ASI $4B_{16}$ (75 in decimale) con VA 0_{16}.

5.3.5.2 Verifica: teoria e simulazione

Il commento a corredo, suggerisce che tra le defines dell'OpenSPARC T1 la costante associata al nome del registro è settata a 3.

Ulteriore e decisa conferma, si ha dalla documentazione dell'UltraSPARC T1 (vedi §3.1.2.4), che richiede espressamente che i due bit controllabili di questo registro siano alti e vengano annullati solo in casistiche particolari. Sembra dunque coerente la scelta di mettere a 1 sia il nceen che il ceen bit.

5.3.5.3 Modifiche proposte

Le prime 6 istruzioni, sono state generate dal gcc e potrebbero essere sostituite da una OR, più comoda per inserire numeri senza segno inferiori agli 11 bit del campo *immediate*.

```
!!  SPARC Error Enable Reg.
        or    %g0, 0x3, %l1     !! l1 = 3
        stxa  %l1, [%g0] (75)   !! copy the content of the
           l1 register
                                !! into the "SPARC Error
                                   Enable reg"
```

5.3.6 HTBA

l registro HTBA, *HyperPriviledge Trap Base Address*, è descritto nel paragrafo 7.5.2.

5.3.6.1 Configurazione dal file boot.s di default

```
!! HTBA
        sethi %hh(0x80000),%g1
        or   %g1,%hm(0x80000),%g1
        sllx  %g1,32,%g1
        sethi %hi(0x80000),%l1
        or    %l1,%g1,%l1
        or    %l1,%lo(0x80000),%l1
        wrhpr %l7, %g0, %htba           !! bits 63-14
           select the trap vector
```

Le prime 6 istruzioni costruiscono per parti la costante 80000_{16} sul registro *l1*, usando *g1* come appoggio. In sequenza, viengono settati i bit *[63:42]*, *[41:32]*, *[32:10]* e *[9:0]*.

L'ultima istruzione copia il contenuto di *l7* nell'HTBA, direttamente indirizzabile tramite *%htba*, senza dover impostare l'ASI. Il registro *l7* è nullo.

5.3.6.2 Verifica: teoria e simulazione

Il codice appare incongruente perchè viene creata con le prime 6 istruzioni una costante che poi non è usata: prima del prossimo utilizzo, *l1* viene infatti riinizializzato. Il registro *l7* è nullo di default: ci si costruisce 80000_{16}, che tra l'altro è uno dei due valori che le vengono assegnati nella procedura di boot in *RED state* dell'OpenSPARC T1, e poi non la si usa. L' altra costante assegnabile all'HTBA nel file hred_reset_handler.s dell'OpenSPARC T1 è 120000_{16}.

Il registro HTBA salva una parte di un indirizzo, nei bit [47:14], mentre i primi 14 bit sono sempre letti come 0. L'indirizzo che viene salvato nella parte alta del registro è dunque 20_{16}. Nel caso venga generata una trappola in modalità hyperpriviledge, sarà scelto il *trap vector* salvato nel TTE che corrisponde in parte a questo indirizzo.

In simulazione si verifica che dopo il POR il registro ha tutti i valori indeterminati (x), meno gli ultimi 14 bit che sono sempre letti come zero, e che la sua "parte attiva" prende valore 20_{16}.

5.3.6.3 Modifiche proposte

Anzichè assegnare all'HTBA il valore del registro *l7* (=0_{16}), verrà inizializzato con *l1*.

Il valore 80000_{16}, è quello di default nella procedura di gestione del RED state nell'OpenSPARC T1. I bit più significativi (oltre il ventesimo) sono tutti a 0 dunque il VA salvato nel HTBA dovrebbe essere un indirizzo di RAM. La RAM dell'S1 Core è più piccola della RAM normalmente supportata dal T1, ed il suo spazio di indirizzi va da 40000_{16} a $4FFFF_{16}$. Questo implica che, inizializzandolo ad 80000_{16}, la generazione di una trappola in Hyperpriviledge mode farà il fetch da un indirizzo inesistente.

Non essendo però caricato in RAM un sistema operativo o comunque non sono incluse nel package dell'S1 le immagini di RAM con le procedure di errore, a meno di non costruirle "by hand", non è utile utilizzare un indirizzo valido di RAM. Semplicemente, alla generazione di una trappola, si bloccherà la simulazione.

Se invece si reindirizzasse l'HTBA verso un indirizzo valido di RAM, per esempio 48000_{16}, alla generazione di una eccezione (a patto di non modificare l'immagine di RAM in ram_harness.hex), verranno eseguite delle NOP a partire da tale indirizzo[3]: nel testbench è stata infatti inizializzata sia la RAM che la ROM con istruzioni NOP. Un esempio potrebbe essere il seguente:

```
! HTBA
mov 1, %g1              ! g1 = 1
sllx %g1, 18, %l1       ! l1 = 40000
sllx %g1, 15, %g1       ! g1 = 8000
or %l1, %g1, %l1        ! l1 = 48000
wrhpr %l1, %g0, %htba   ! bits 63-14 select the trap vector
```

5.3.7 Inizializzazione del I-TLB e del D-TLB

Questo segmento di codice è stato preso integralmente dai codici disponibili nel pacchetto dell'OpenSPARC T1. I due TLB hanno grandezza di 512 *doubleword* (da 64 bit ciascuna) nell'S1 Core come nell'OpenSPARC T1, pertanto sembra legittimo

[3]Questo è quasi vero: la costruzione degli indirizzi per la gestione delle trappole verrà meglio analizzata nelle simulazioni §6.4.1.

supporre che siano corrette. La maggior parte del tempo di simulazione è usato per eseguire i loop di inizializzazione dei TLB.

```
! I- and D-TLB initialize
! *** Instructions merged from file hboot_tlb_init.s *** !

! Init all itlb entries
mov 0x30, %g1
mov %g0, %g2
itlb_init_loop:         ! ITLB LOOP
                ! clear data and tag entries for TLB buffer
stxa %g0, [%g1] 0x50    ! IMMU TLB Tag Access reg=0
stxa %g0, [%g2] 0x55    ! IMMU TLB Data Access reg=0,
                ! g2 values from 0x000 to 0x7f8
add %g2, 8, %g2 !
cmp %g2, 0x200  ! (g2 == 512) ? (icc=1), loop
                 ! (512*8=4096=0x1000), but max VA=0x7F8
bne itlb_init_loop      ! if (g2!=512) run another loop
nop

! Init all dtlb entries
        mov 0x30, %g1
        mov %g0, %g2
dtlb_init_loop:         ! DTLB LOOP
stxa %g0, [%g1] 0x58    ! DMMU TLB Tag Access register=0
stxa %g0, [%g2] 0x5d    ! DMMU TLB Data Access register=0,
                ! g2 values from 0x000 to 0x7f8
add %g2, 8, %g2 ! g2=g2+8 bytes every time(64 bits)
cmp %g2, 0x200  ! compare g2 with 512
bne dtlb_init_loop      ! if (g2!=512) then run another
   loop
nop

! Clear itlb/dtlb valid
stxa %g0, [%g0] 0x60    ! ASI_ITLB_INVALIDATE_ALL
```

```
        ! (IMMU TLB Invalidate register)=0
mov 0x8, %g1
stxa %g0, [%g0 + %g1] 0x60      ! ASI_DTLB_INVALIDATE_ALL
        ! (DMMU TLB Invalidate register)=0
!! *** End of inserted instructions *** !!
```

Vengono dapprima inizializzati i registri globali *g1* e *g2* rispettivamente a 30_{16} e 0_{16}, e saranno usati come Virtual Address da accoppiare agli ASI per la scrittura dei dati nei TLB.

Il Tag Access Register e una linea di dati del TLB vengono impostati a 0. Viene aggiunto 8, corrispondente al numero di Byte di una *doubleword* (64 bit). Avviene dunque la comparazione tra il valore di g2, che mantiene il virtual address, con il valore di fine ciclo (512). L'istruzione CMP (*compare*), viene usata per modificare lo stato del CCR.icc.z (*Integer Condition Code register*, campo z): viene fatta la sottrazione tra i due argomenti, la quale ha come effetto secondario di settare il campo z del registro CCR.icc, che è 1 quando la ALU ha risultato nullo in una operazione a 32 bit. BNE (*Branch if Not Equal to zero*) salta alla etichetta se CCR.icc.z è diverso da zero. La istruzione di NOP dopo il salto è necessaria in quanto il BNE è un'istruzione con *delay*: l'istruzione successiva viene eseguita comunque, prescindendo dal verificarsi della condizione di salto.

Dopo che tutte le linee dei TLB sono inizializzate a 0, vengono invalidati tutti i valori salvati nei TLB, settando i valid bit a 0. La invalidazione dei TLB non viene fatta durante il reset, ed è espressamente indicato nella documentazione (vedi §3.1.2.5) che il software di gestione del reset si debba preoccupare di invalidarne i dati perchè, una volta abilitate I-MMU e D-MMU il loro comportamento sarebbe indefinito.

Il funzionamento del TLB Tag Access Register (sia per la I-TLB che la D-TLB) è richiamato nella sezione 7.5.8.

Il funzionamento del TLB Data Access Register (sia per la I-TLB che la D-TLB) è richiamato nella sezione 7.5.8.

La configurazione che viene data a questi registri è indifferente, poichè in seguito alla inizializzazione tutte le linee dei due TLB vengono rese non valide. I TLB sono mantenuti sia dal sistema operativo che aggiornati automaticamente via HW: durante il boot, però, non vengono aggiornati e in RAM non risiede un sistema operativo ma un semplice programma, duque viene supposto che la

configurazione automatica di caricamento delle pagine sul TLB sia corretta.

Nei TLB, oltre all'indirizzo fisico e virtuale della pagina, vengono salvate tutte le informazioni riguardanti la pagina: se ci sia o meno del codice eseguibile, se istruzioni e dati siano *cacheable*, la dimensione della pagina.

5.3.7.1 Modifiche proposte

Un TLB miss causa la generazione di una trappola (fast_instruction_access_MMU_miss o fast_instruction_access_MMU_miss). La gestione della trappola prevede di verificare se nel TSB e nel TTE Tag Target se vi sia già la traduzione richiesta per la pagina, se il TTE non è presente nel TSB, allora verrà fatto il JMP ad un codice di gestione della trappola (che dovrebbe essere presente in RAM o ROM) più sofisticato e più lento.

Nel seguito della procedura di boot, e nel programma caricato in RAM non vi sono scritture del TSB o scritture dirette del TLB per la traduzione delle pagine: verrà dimostrato (vedi figura 6.15) che la simulazione, in certe condizioni, si blocca alla esecuzione di LOAD e STORE da RAM proprio perchè non è presente la traduzione degli indirizzi nei TLB nè il codice di gestione del TLB miss.

La configurazione di TLB e TSB resta ancora irrisolta allo stato attuale del progetto.

Si può modificare il codice che annulla il contenuto dei TTE dei TLB in modo che venga risparmiato ad ogni ciclo l'esecuzione di una istruzione, sfruttando lo *slot di delay* che segue l'istruzione BNE. I due cicli di inizializzazione dei TLB occupano la maggior parte del tempo di simulazione e la eliminazione di una istruzione per ciclo, consente di non eseguire circa 1000 istruzioni. Può essere modificato come di seguito (viene fatto l'esempio per il solo I-TLB):

```
!! Eliminazione di una istruzione per ciclo.

! Init all itlb entries
mov 0x30, %g1
mov %g0, %g2
stxa %g0, [%g1] 0x50    ! IMMU TLB Tag Access reg=0
itlb_init_loop:         ! ITLB LOOP
stxa %g0, [%g2] 0x55    ! IMMU TLB Data Access reg=0,
                ! g2 values from 0x000 to 0x7f8
```

```
add %g2, 8, %g2 !
cmp %g2, 0x200  ! (g2 == 512) ? (icc=1), loop
bne itlb_init_loop      ! if (g2!=512) run another loop
stxa %g0, [%g1] 0x50 !Execution in delay-slot of BNE
```

5.3.8 Primary Context e Secondary Context Register

I *context register* sono pensati per permettere al software priviledge di diversificare gli spazi di indirizzi virtuali (VA). Gli indirizzi virtuali sono il modo con cui il software di livello utente comunica con la memoria: vengono poi tradotti dal software priviledge in *real address* (RA) e hyperpriviledge e viene costruito l'indirizzo fisico (PA). La tabella che segue, indica come vengono formati differenti spazi di indirizzi e i privilegi che il software deve avere per manipolarli.

Virtual Address	Virtual Address	User
Real Address	context ID :: Virtual Address	Priviledge
Phisical Address	partition ID :: context ID :: Virtual Address	Hyperpriviledge

I *context register primary* e *secondary* sono condivisi tra le MMU, mentre il Nucleus Context register è messo a 0 via hardware.

Inoltre, il numero di bit dei *context register* delle architetture UltraSPARC T1 è implementato con un campo di 13 bit (bit [12:0]), del *context register*.

5.3.8.1 Configurazione dal file boot.s di default

```
!! Initialize primary context register
        mov 0x8, %l1
        stxa %g0, [%l1] 0x21
!! Initialize secondary context register
        mov 0x10, %l1
        stxa %g0, [%l1] 0x21
```

I registri *primary* e *secondary context*, sono indirizzabili entrambi con ASI 21_{16} e VA 8_{16} e 10_{16} rispettivamente. Vengono inizializzati a 0.

Sembra coerente che il primo spazio di indirizzi virtuali sia quello con *context register* nullo. Non verrà più cambiato il valore dei *context register* per tutto l'arco

della simulazione, in quanto in RAM non è caricato un sistema operativo che gestisca differenti spazi di indirizzi virtuali. Nell'eseguire semplice programma che effettua LOAD e STORE, non sembra utile modificare questi campi.

5.3.9 LSU Control Register, secondo accesso

Il LSU Control Register è descritto nel paragrafo 7.5.5.

5.3.9.1 Configurazione dal file boot.s di default

```
!! GCC version
!! LSU_CTL_REG[3]=1 (DMMU enabled)
!! LSU_CTL_REG[2]=1 (IMMU enabled)
!! LSU_CTL_REG[1]=1 (L1-dcache enabled)
!! LSU_CTL_REG[0]=1 (L1-icache enabled)

        mov  0xF, %l1           ! all enabled
        stxa  %l1, [%g0] (69)
```

L' ASI 45_{16} con VA 0_{16} corrisponde all'LSU Control Register. Entrambe le cache rimangono attive, e vengono abilitate anche le MMU.

5.3.9.2 Modifiche proposte

L'abilitazione della cache dati e della MMU comporta, dipendentemente anche da HPSTATE.red e dall'ASI, differenti meccanismi di traduzione degli indirizzi. Per ottenere una LOAD o STORE in memoria che non generi un meccanismo di trappola, la configurazione dei registri afferenti alla I- e D-MMU sia coerente. Se si vuole sfruttare la associazione automatica degli ASI alle istruzioni di LOAD e STORE, occorre preconfigurare ad hoc dei TTE del D-TLB in riferimento alle pagine di memoria su cui si vuole caricare dati o leggerli.

Configurando l'S1 Core, si vuole evitare di generare eccezioni perchè l'eccezione va gestita tramite la configurazione di registri della TLU (Trap Logic Unit) e l'inserimento sia in memoria che nei registri interni di codice per la gestione delle trappole. E' una procedura molto complessa, pensata perchè sia gestita dal SO o da un sistema di boot automatizzato associato al SO, sia per la quantità di eccezioni che possono essere generate, sia perchè dalla gestione delle trappole dipende tutta la gestione del software.

Anche il setting delle MMU, TLB e TSB (Translation Storage Buffer) è pensato per essere gestito via SO, ma avendo l'S1 Core una RAM relativamente piccola (64 Kbyte), si potrebbe pensare di dividere la memoria in pagine, e configurare nel boot i TTE (Translation Table Entry) in modo tale da avere nel TLB le traduzioni di tutte le pagine di memoria, e non causare TLB-miss. La dimensione minima delle pagine per l'OpenSPARC T1 è di 8 Kbyte, e sarebbe sufficiente studiare come configurare al più 8 TTE dei 64 disponibili nel D-TLB per non generare TLB-miss o errori dipendenti da errate traduzioni VA$\longrightarrow$PA o RA$\longrightarrow$PA. Avere il sistema di traduzione configurato, potrebbe essere sufficiente per non generare errori dovuti alla esecuzione di codice compilato anche a livello utente e livello kernel (*user* e *priviledge code* rispettivamente), in cui la generazione degli ASI è automatica e sono richieste traduzioni di indirizzi.

La configurazione della gerarchia di memoria è una problematica che resta ancora aperta nello studio affrontato nella presente tesi. Molti test sulla procedura di boot, nella ricerca di una configurazione che consentisse di non attivare dinamiche di traduzione degli indirizzi dei dati, sono stati effettuati disattivando sia le MMU che le Caches tramite l'annullamento del LSU Control Register. Anche in questo caso, per ASI da e verso la memoria, vengono attuate traduzioni RA$\longrightarrow$PA, pur eseguendo il codice in RED state (lo stato che più di tutti è abilitato ad utilizzare direttamente indirizzi fisici, essendo tipicamente in esecuzione un Virtual Processor Hiperpriviledge).

Ancora non si è raggiunta la configurazione che consenta di effettuare LOAD e STORE da e verso la RAM che vadano a buon fine.

5.3.10 Generazione indirizzo di base per il codice in memoria

5.3.10.1 Configurazione dal file boot.s di default

```
!! GCC version
        sethi  %hi(0), %g1
        sethi  %hi(0x40000), %g2        !! Jump address to
           memory
        mov  %g1, %g1
        mov  %g2, %g2
        sllx  %g1, 0x20, %g1
```

```
        or  %g2, %g1, %g2
```

Viene generato l'indirizzo della prima istruzione di RAM e salvato nel registro globale *g2*. Il salto incondizionato che sfrutterà tale indirizzo, è la penultima (viene sempre eseguita anche l'istruzione successuva) istruzione del codice in ROM. L'indirizzo 40000_{16} è effettivamente quello del primo blocco di RAM, come indicato nella documentazione a corredo dell'S1 Core, vedi §4.3.

Nel caso si volesse eseguire tutto il codice in ROM, anche lasciando questo segmento di codice, basta commentare l'istruzione di JMP e copiare il file ram_harness.hex all'interno di rom_harness.hex, aggiungendolo alla fine.

5.3.11 Codice Hyperpriviledge con Trap Level 1

Informazioni generali sull'unità *Trap Logic Unit* ed alcuni suoi registri, sono fornite nella sezione §7.2.6. Il funzionamento del registro HPSTATE è invece descritto nel §5.3.3.

5.3.11.1 Configurazione dal file boot.s di default

```
! HTSTATE[TL=1]
rdhpr %hpstate, %g3
wrpr 1, %tl     ! current trap level = 1

sethi %hh(0x0),%g1      ! l1 = 0  (start)
or %g1,%hm(0x0),%g1
sllx %g1,32,%g1
sethi %hi(0x0),%l1
or %l1,%g1,%l1
or %l1,%lo(0x0),%l1     ! l1 = 0  (end)

wrhpr %g4, %g0, %htstate  ! reset HTSTATE reg that store
                ! hyperpriviliged state after a trap
wrpr 0, %tl     ! current trap level = 0 (No Trap)
mov 0x0, %o0    ! please don't delete since used in
                ! customized IMMU miss trap
```

Nella prima istruzione, l'HPSTATE corrente viene copiato in *g3*. Viene impostato il trap level (TL) a 1. E' dunque azzerato il registro *l1*, usando *g1* come

registro temporaneo. Il valore di *g4*, registro mai inizializzato e dunque con valore 0, viene scritto nel HTSTATE del trap level 1. Il livello di trappola viene dunque portato a 0 con una nuova scrittura in TL. Ci si assicura poi che il primo registro di output, *o0*, contenga valore nullo.

5.3.11.2 Verifica: teoria e simulazione

Il valore corrente di HPSTATE è stato modificato all'inizio della procedura di boot e vale 0x820, vedi §5.3.3. Un HPSTATE con tale valore implica che il codice sia eseguito in modalità hyperpriviledge.

Il registro *g4* ha invece valore nullo e tale valore avrà anche HTSTATE[TL=1]. Il ruolo di HTSTATE è definire quale valore dovrà avere HTSTATE quando verrà richiamata una procedura di trappola. Quando viene generata una eccezione con TL=0, verrà incrementato TL (TL=1) e HPSTATE assumerà il valore di HTSTATE[TL=1].

Il sospetto è che ci sia stato un errore, in quanto viene salvato HPSTATE in g3, e poi si usa il valore di g4, mai inizializzato, per settare HTSTATE, il che rende inutile l'aver copiato HTSTATE. Inoltre il valore nullo non si addice ad HTSTATE, in quanto nella documentazione ci si raccomanda di tenere sempre alto il valore di HPSTATE{11}, dunque ci si aspetterebbe che HTSTATE venga impostato almeno a 0x800. Meno discutibile è invece il fatto che si scelga un HTSTATE[TL=1] senza i privilegi hyperpriviledge, in quanto maxptl=3 e dunque si dovrebbe avere solo il livello priviledge per la gestione delle trappole con TL=1.

In ogni caso, assicurare modalità hyperpriviledge per il livello 1 di trappola, assicura che molte eccezioni possano essere gestite richiamando una sola procedura di trappola (senza innoltrarsi in più livelli). Non vengono infatti settati gli altri livelli di trappola (sono 6, 7 compreso lo 0). Verrà dunque assunto nel seguito che si sia commesso un errore, e che si fosse voluto inizializzare HTSTATE=HPSTATE=*g3*.

Una volta assicurato che, se dovesse venire generata una eccezione il codice potrà essere eseguito con massimi privilegi, si imposta a 0 il TL. Lo stato 0 per TL implica che non sia in corso l'esecuzione di trappole, e che il codice può continuare ad essere eseguito normalmente.

L'esecuzione di questo frammento di codice fa notare come non vengano generate eccezioni di sorta al passaggio da un trap level ad un altro, e non vi è nemmeno l'esigenza di uscire dal livello di trappola corrente chiamando le apposite

istruzioni DONE o RETRY (che ripristinano automaticamente lo stato di alcuni registri (come PC e NPC) del *virtual processor* al livello di trappola precedente).

L' esigenza di impostare *o0*=0, è dovuta al fatto che nelle librerie assembly dell'OpenSPARC, da dove è stato preso il codice, c'era scritto in un commento di impostarlo a zero, ed è stato lasciato per prudenza.

5.3.11.3 Modifiche proposte

Come precedentemente discusso, sarei propenso ad assumere che HTSTATE[TL=1] venga impostato a 804_{16}, identico al valore attuale di HPSTATE. Questo perchè, in fondo, non si hanno a disposizione le procedure di trappola in RAM, e dunque anche la generazione di una eccezione bloccherebbe il funzionamento del processore. Sarebbero anche da configurare ulteriori registri sia per questo trap-level che per i restanti, ma per i motivi appena citati, si concorda nel configurarne uno solo. Sarebbe inoltre da configurare, per il livello 1 di trappola, anche il regitro PSTATE (vedi tabella 6.2), perchè, per quanto venga ignorato il suo bit PSTATE.priv (priviledge) quando HPSTATE.hpriv (hyperpriviledge), ci sarebbe, per esempio, da disabilitare il bit PSTATE.pef (FPU enable) in quanto non è presente la FPU. Quest'ultimo punto è però indifferente se non si utilizzano istruzioni floating point.

Il setting a 0 del registro *l1*, verrà eseguito con la più semplice istruzione MOV.

```
! HTSTATE[TL=1]
rdhpr %hpstate, %g3
wrpr 1, %tl     ! current trap level = 1

mov 0x0, %l1    ! l1 = 0

wrhpr %g3, %g0, %htstate    ! reset HTSTATE reg that store
                !hyperpriviliged state after a trap
wrpr 0, %tl     ! current trap level = 0 (No Trap)
mov 0x0, %o0    ! please don't delete since used in
                ! customized IMMU miss trap
```

5.3.12 Jump in RAM

```
! Jump in RAM
        jmp %g2         ! jump to 0x40000

!! wrhpr %g0, 0x800, %hpstate
        ! ensure bit 11 of the HPSTATE register is set
nop
nop
```

Viene fatto il salto al primo indirizzo di RAM. La istruzione successiva è una NOP, necessaria perchè viene sempre eseguita l'istruzione successiva a quelle "*delayed*", come appunto il JMP.

5.3.12.1 Verifica: teoria e simulazione

Si verifica in simulazione che effettivamente il salto viene eseguito e viene iniziato il fetch da RAM. Il comportamento del codice in RAM, come ci si aspetta, cambia rispetto alle configurazioni che sono state apportate in ROM, soprattutto per quanto riguarda l'attivazione di MMU e delle cache L1, ma non solo.

Il codice inserito in RAM sarà trattato nel prossimo capitolo.

5.3.12.2 Modifiche proposte

Nessuna, solo alcune considerazioni che nascono da una istruzione lasciata commentata.

E' stata riportata anche una istruzione commentata che dà un nuovo valore ad HPSTATE. E' stata lasciata per notare che non solo si assicura che il bit 11 di HPSTATE sia alto, ma toglie anche al processore virtuale i privilegi di hyperuser. Il fatto che il processo possa non essere più hyperpriviledge, comporta anche che gli indirizzi in RAM vengano trattati diversamente e abbiano sicuramente bisogno almeno della traduzione da *Real Address* a *Phisical Address*. Come più volte dichiarato all'interno del presente capitolo, non sono configurati nella procedura di boot i registri necessari ad una corretta traduzione. Si preferisce dunque lasciare commentata la WRHPR su HPSTATE.

Inoltre, nella documentazione dell'OpenSPARC T1 (vedi §3.1.2.5) si viene scoraggiati dall'inserire una modifica ai registri di stato (sia priviledge che hyperpriviledge) nello slot di delay di un salto: nel caso si voglia rendere attiva tale istruzione, dovrebbe essere eseguita prima del JMP.

Capitolo 6

S1 CORE: TEST E CONFIGURAZIONE DEL BOOT

Nel capitolo 5, sono state evidenziate alcune incongruenze sulla procedura di boot di default dell'S1 Core. Verrà qui innanzitutto riproposta una versione delle istruzioni del boot che sembra corretta in base alle considerazioni del precedente capitolo. Messa a punto una simulazione di partenza, saranno applicate ulteriori modifiche, basate anche sulla ispezione di segnali e registri che non vengono esplicitamente configurati nel boot fornito di default. Saranno presentate ed analizzate le simulazioni generate dalle configurazioni adottate.

6.1 La simulazione di base

Viene qui definito un punto di partenza, una simulazione di riferimento. Sarà sempre supposto che vengano usate le configurazioni definite nel seguito del presente paragrafo, a meno che non sia specificato altrimenti.

6.1.1 Configurazione del package

Si suppone che nella esecuzione del codice da shell sia preventivamente stato lanciato il source del file sourceme presente nella home del package dell'S1 Core (vedi §4.2.1).

Verrà usata la versione EE del core per tre motivi:

- molte simulazioni sono state lanciate con le diverse configurazioni e la maggior parte danno risultati identici se le cache L1 vengono disabilitate (i campi dc e ic del LSU Control register sono nulli);
- non verranno usati e configurati *virtual processor* (*hardware thread*) diverso dal primo e sotto questo profilo tutte e tre le versioni sono identiche;
- Icarus Verilog non riesce a simulare le versioni SE ed ME dell'S1 Core. Si suppone dunque che si abbia un sistema configurato come descritto in 5.1.1.

Vengono apportate due piccole modifiche al tool di compilazione, sito in $S1_ROOT/tools/bin/compile_test. Il cross-compilatore gcc-4.4 permette di compilare codice non solo generiche architetture V9, ma anche per V9b ed avere a disposizione nella scrittura del codice assembly anche alcuni nomi di costanti e di registri (al posto degli asi). Nonostante tali costanti non verranno qui usate, questo passaggio è fondamentale se si vuole eseguire del codice preso direttamente dall'assembly di corredo all'OpenSPARC T1[1]. Aprire dunque il file con un editor di testo ed inserire, ogni volta che viene richiamato un tool del cross-compilatore gcc, l'opzione "-Av9b", prestando attenzione a non inserirlo nella compilazione del dumper, che è un tool da compilare per la macchina in uso.

Si aggiunga inoltre alla fine del file compile_test la stringa

```
cat ram_harness.hex >> boot/rom_harness.hex
```

per fare in modo che dopo la compilazione del codice-applicazione, il contenuto di ram_harness.hex venga copiato alla fine di boot/rom_harness.hex. Questo permetterà, commentando il JMP alla fine della procedura di boot (boot.s), di fare in modo che anche il codice-applicazione venga eseguito da ROM.

Servirà, ad un certo punto, poter scrivere un programma assembly da inserire in RAM, in modo da avere più controllo sulle istruzioni, senza dover usare un programma compilato in C. Un modo per ottenere l'esadecimale in ram_harness.hex[2],

[1] Il cross-compilatore gcc4.4-1 ancora NON supporta l'architettura UltraSPARC: compila appunto codice per V9. Sebbene tutte le istruzioni del V9 siano supportate dalle architetture UltraSPARC, non è sempre vero il contrario per i registri interni del processore: nell'usare codice parametrico per ASI e costanti di default, bisogna essere sicuri che esse effettivamente corrispondano a quelle desiderate. Anche per questo motivo non sono stati usati in simulazione e non verranno qui usati.

[2] Per non appesantire la trattazione si usano nomi di file senza il path di riferimento. Per la maggior parte dei file richiamati nel presente capitolo la loro locazione è nelle cartelle $S1_ROOT/tools/bin, $S1_ROOT/tests e $S1_ROOT/tests/boot. La gerarchia dei file del package dell'S1 Core è riportata nela capitolo 4.

corrispondente all'assembly hello.s, è descritto nel §4.2.5.3. Per creare il corrispondente file assembly di hello.c, basta lanciare

```
usr@name:$ cd $S1_ROOT/tests
usr@name:~/s1_core/tests$ sparc64-linux-gcc -S hello.c
```

Inserire dunque in compile_test, al posto delle linee che seguono il commento "Compile the C test" le seguenti:

```
sparc64-linux-as -ah -am -Av9b -o $1.bin $1.s
sparc64-linux-objdump -d -EB -w -z $1.bin > $1.dump
grep "  " $1.dump | egrep -v "file format" | dump2hex.bin >
    ram_harness.hex
```

D'ora in avanti, se non diversamente specificato, si suppone che il tool compile_test produca:

- ram_harness.hex, dalla compilazione del file assembly $S1_ROOT/tests/hello.s;

- il file rom_harness dalla compilazione del file assembly $S1_ROOT/tests/boot/boot.s cui è aggiunto sempre alla fine il risultato della compilazione di hello.s. Verrà eseguito in simulazione il codice tutto dalla ROM se il JMP alla fine di boot.s è commentato, altrimenti la procedura di boot sarà eseguita da ROM e il codice-applicazione da RAM. Di base, si suppone che il JMP sia commentato, e si faccia il fetch di tutte le istruzioni dalla sola ROM.

Si ricorda inoltre che i files ".dump" contengono la corrispondenza tra istruzione assembly, istruzione esadecimale e posizione dell'istruzione in memoria. Queste corrispondenze sono utili quando si analizzano le forme d'onda: l'istruzione assembly è facilmente comprensibile dall'umano, l'istruzione esadecimale si può leggere sul bus wishbone e nel TIR (*Thread Instruction register*), mentre il *Program Counter* (PC) fornisce l'indirizzo di memoria. Bisogna tenere a mente che l'istruzione effettivamente eseguita segue l'evoluzione del (dei) PC che si evolve lungo la *pipeline*.

6.1.2 Il file boot.s di riferimento

Si supporrà che il file $S1_ROOT/tests/boot/boot.s tenga il codice di boot. Viene qui assemblato il codice della procedura di boot di riferimento.

Si sceglie di eseguire il codice in hyperpriviledge state, con HPSTATE.red = 0. Inoltre sia le caches di primo livello che le MMU saranno inizialmente disattivate. Viene disattivato il JMP alla memoria e si suppone che il codice-applicazione venga inserito come appendice del file rom_harness.hex.

```
/* Procedura di boot di riferimento */
nop    ! 8 nop: base-address for POR is 0xFFF0000020
nop
nop
nop
nop
nop
nop
nop

mov 0, %l1              ! Clear l1
mov 0x10, %g1
stxa %l1, [%g1] 0x42    ! LSU_DIAG_REG = b00

/* Le caches saranno gestite in seguito */
!  mov  3, %l1
!  stxa %l1, [%g0] 0x45  ! MMU disattivate, caches attive

! HPSTATE.enb = 1, HPSTATE.red = 0, HPSTATE.hpriv = 1
wrhpr %g0, 0x824, %hpstate

mov 0x18, %g1
stxa %g0, [%g0 + %g1] 0x50      ! IMMU SFSR=0
stxa %g0, [%g0 + %g1] 0x58      ! DMMU SFSR=0

or %g0, 0x3, %l1        ! l1 = 3
stxa %l1, [%g0] (75)  ! SPARC_Error_Enable_reg = 3
```

```
! Set HTBA: HTBA[63:14] = 0x12
mov 1, %g1              ! g1 = 1
sllx %g1, 18, %l1       ! l1 = 40000
sllx %g1, 15, %g1       ! g1 = 8000
or %l1, %g1, %l1        ! l1 = 48000
wrhpr %l1, %g0, %htba   ! bits 63-14 select
                        ! Hpriv trap vector


! *** Instructions merged from file hboot_tlb_init.s *** !

! Init all itlb entries
mov 0x30, %g1
mov %g0, %g2
itlb_init_loop:         ! ITLB LOOP
                ! clear data and tag entries for TLB buffer
stxa %g0, [%g1] 0x50    ! IMMU TLB Tag Access reg=0
stxa %g0, [%g2] 0x55    ! IMMU TLB Data Access reg=0,
                ! g2 values from 0x000 to 0x7f8
add %g2, 8, %g2 !
cmp %g2, 0x200  ! (g2 == 512) ? (icc=1), loop
                 ! (512*8=4096=0x1000), but max VA=0x7F8
bne itlb_init_loop      ! if (g2!=512) run another loop
nop

! Init all dtlb entries
        mov 0x30, %g1
        mov %g0, %g2
dtlb_init_loop:         ! DTLB LOOP
stxa %g0, [%g1] 0x58    ! DMMU TLB Tag Access register=0
stxa %g0, [%g2] 0x5d    ! DMMU TLB Data Access register=0,
                ! g2 values from 0x000 to 0x7f8
add %g2, 8, %g2 ! g2=g2+8 bytes every time(64 bits)
```

```
cmp %g2, 0x200  ! compare g2 with 512
bne dtlb_init_loop      ! if (g2!=512) then run another
   loop
nop

! Clear itlb/dtlb valid
stxa %g0, [%g0] 0x60    ! ASI_ITLB_INVALIDATE_ALL
        ! (IMMU TLB Invalidate register)=0
mov 0x8, %g1
stxa %g0, [%g0 + %g1] 0x60      ! ASI_DTLB_INVALIDATE_ALL
        ! (DMMU TLB Invalidate register)=0

! *** End of inserted instructions *** !

mov 0x8, %l1
stxa %g0, [%l1] 0x21    ! Initialize
                        ! primary context register = 0
mov 0x10, %l1
stxa %g0, [%l1] 0x21    ! Initialize
                        ! secondary context register = 0

mov  0x0, %l1   ! MMUs disabled, caches L1 disabled.
stxa  %l1, [%g0] (69)   ! LSU_ctl_reg = bx0000

sethi  %hi(0), %g1
sethi  %hi(0x40000), %g2    ! Jump address to RAM code
mov  %g1, %g1
mov  %g2, %g2
sllx  %g1, 0x20, %g1
or  %g2, %g1, %g2       ! jump address fully initialized.

! Set HTSTATE[TL=1] = HPSTATE = 0x824 !
rdhpr %hpstate, %g3
wrpr 1, %tl             ! current trap level = 1
```

```
mov 0x0, %l1            ! l1 = 0
wrhpr %g3, %g0, %htstate   ! reset HTSTATE reg that store
                ! hyperpriviliged state after a trap
wrpr 0, %tl     ! current trap level = 0 (No Trap)
mov 0x0, %o0    ! please don't delete since used in
                ! customized IMMU miss trap

/* jump commented out: remember to add RAM code to
   rom_harness.hex*/
!       jmp %g2         ! Jump to 0x40000
        nop
        nop
/* APPLICATION CODE FOLLOWS ... */
```

6.1.3 Il file hello.s estratto da hello.c

Finchè non verranno giustificati cambiamenti, verrà usata in simulazione la compilazione del file `hello.c`. Il file viene riportato nel listato di seguito:

```
// Sample program that writes two words at a predefined
   address
int main() {
        unsigned long* address;
        address = (unsigned long*)0x0000CAC0;
        (*address) = 0xC1A0C1A0;        // First store
        address = (unsigned long*)0x0000CAC0;
        (*address) = 0x0ADDCAFE;        // Second store
        return 0; }
```

Il corrispondente codice assembly, ottenuto tramite il cross-compilatore gcc da x86 a SPARC V9b, viene riportato di seguito:

```
/* hello.s file content */
        .file   "hello.c"
        .section        ".text"
        .align 4
        .global main
```

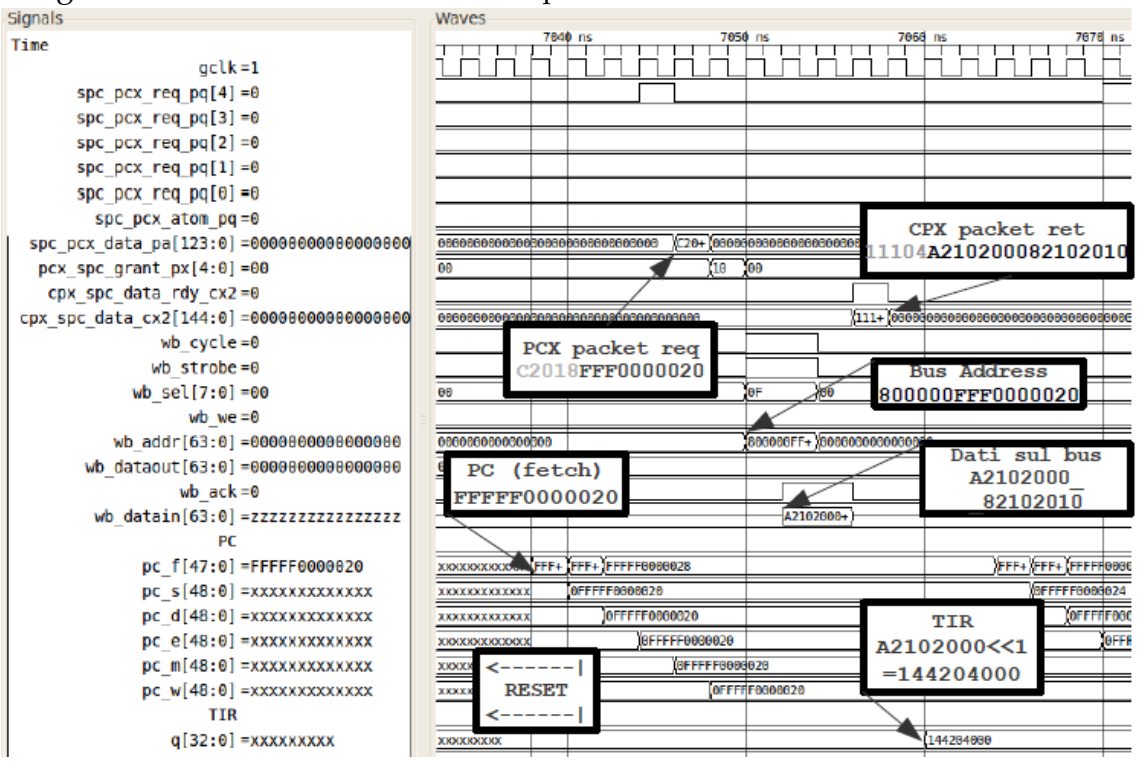
```
        .type   main, #function
        .proc   04
main:
        .register       %g2, #scratch
        save    %sp, -208, %sp
        mov     811, %g1
        sllx    %g1, 6, %g1
        stx     %g1, [%fp+2023]
        ldx     [%fp+2023], %g2         ! g2 = 0xCAC0
        sethi   %hi(3248537600), %g1
        or      %g1, 416, %g1           ! g1 = 0xC1A0C1A0
        stx     %g1, [%g2]              ! STORE
        mov     811, %g1
        sllx    %g1, 6, %g1
        stx     %g1, [%fp+2023]
        ldx     [%fp+2023], %g2         ! g2 = 0xCAC0
        sethi   %hi(182306816), %g1
        or      %g1, 766, %g1           ! g1 = 0x0ADDCAFE
        stx     %g1, [%g2]              ! STORE
        mov     0, %g1
        sra     %g1, 0, %g1
        mov     %g1, %i0
        return  %i7+8
         nop
        .size   main, .-main
        .ident  "GCC: (GNU) 4.1.1 ()"
        .section        ".note.GNU-stack"
```

I commenti sono stati aggiunti.

La prima istruzione (corrispondente ad una istruzione fisica) del file (`save %sp, -208, %sp`), non si riuscirà a farla eseguire in simulazione prima di aver configurato correttamente i registri *general purpouse* e la simulazione si bloccherà proprio all'esecuzione di quella istruzione.

Sono state evidenziate nel listato le STORE di dati corrispondenti alle STORE del programma C e sono stato espressi gli indirizzi e i dati in esadecimale.

Figura 6.1: Simulazione del fetch della prima istruzione da ROM. Dati esadecimali.



6.1.4 Analisi delle forme d'onda della simulazione di riferimento

Inserito manualmente o tramite script il contenuto di ram_harness.hex alla fine di rom_harness.hex, si può lanciare la simulazione:

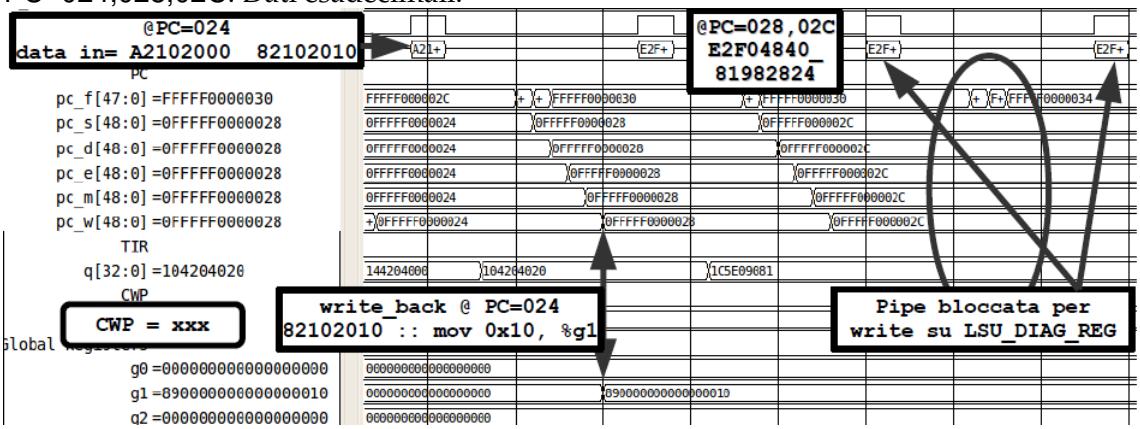
```
usr@name:~$ build_icarus
usr@name:~$ run_icarus
usr@name:~$ gtkwave $S1_ROOT/run/sim/icarus/trace.vcd
```

Verrà usato Icarus Verilog, per quanto su Modelsim i risultati non dovrebbero essere differenti. Si suppone di avere a disposizione oltre alle forme d'onda dei segnali del Wishbone bus i segnali suggeriti nel §5.1.5.

6.1.4.1 Fetch delle istruzioni: PC, TIR e transizioni sul bus

In figura 6.1 è riportata sezione della simulazione in cui il processore inizia a richiedere fetch da ROM. Dopo il POR il PC viene impostato all'indirizzo 20_{16} di ROM. Viene inviato un pacchetto di richiesta dell'istruzione mancante nella I-

Figura 6.2: Simulazione del fetch della 2°,3° e 4° istruzione da ROM, con PC=024,028,02C. Dati esadecimali.

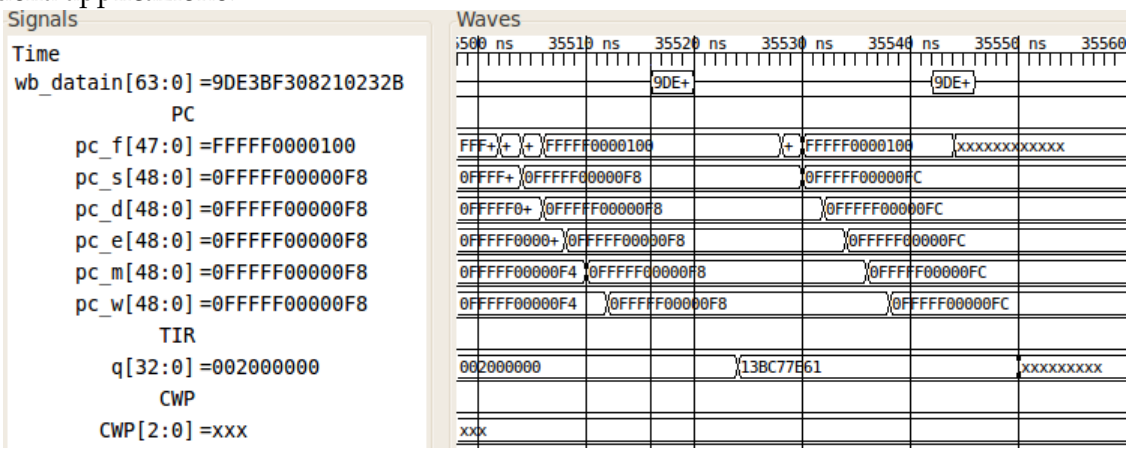


cache (*instruction miss*) al bus tramite un pacchetto PCX contenente i bit di stato e l'indirizzo. Il bus imposta l'*address*: 0x80 indica lo spazio di I/O, 0xFFF la ROM e 0x020 la linea di ROM da 2 istruzioni (64 bit). La ROM mette a disposizione sul bus il dato da 64 bit e dunque nel pacchetto dati di ritorno, oltre alla prima, vi sarà anche la seconda istruzione (se ne ha riscontro leggendo sia il file rom_harness.hex che sim.log). Viene generato il pacchetto di ritorno CPX comprensivo dei bit di controllo e dei 64 bit di dato: il dato sarà disponibile anche nel segnale wb_datain. Solo ora viene scelta una delle due istruzioni inviate dal bus e inizializzato il TIR: il TIR ha 33 bit e l'istruzione è scritta nei 32 bit più significativi.

Quando viene richiesto il fetch della seconda istruzione (rappresentato in figura 6.2), per PC = 0x0FFFFF0000024, l'address sul bus non cambia, viene prelevato lo stesso pacchetto dalla ROM e viene eseguita questa volta l'istruzione presente ai 32 bit meno significativi del dato (82102010_{16}, corrispondente a `mov 0x10, %g1`). Si può notare come il registro *g1* (penultima riga del grafico) venga scritto in corrispondenza della fine dello stage di *write back* dell'istruzione 024_{16} di ROM.

Il fatto che venga richiesta sul bus più volte la transazione del medesimo pacchetto di dati è caratteristico non solo del fetch da ROM, ma di tutte i dati che effettuano transazioni cosiddette *non-cacheable*. Il dato non viene immagazzinato in cache (*cache bypass*), non sarà disponibile all'interno del core ad una seconda richiesta e dovrà essere effettuata una nuova richiesta sul bus. L'esecuzione della pipeline si blocca attendendo che il dato (l'istruzione da eseguire) arrivi dallo spazio di I/O.

Figura 6.3: Zoom sulla fine della simulazione: viene eseguita solo la prima istruzione della applicazione.



La esecuzione della quarta istruzione, che fa il set del LSU Diagnostic Register, blocchi la pipeline per due cicli di pipeline in quanto richiede che vengano compiute più operazioni atomicamente (tipicamente su ASI interni).

6.1.4.2 Ultime istruzioni della simulazione di riferimento

E' stato evidenziato in figura 6.2 che il valore del CWP (*Current Window Pointer*), è indeterminato (e rimarrà tale per tutta la simulazione): nel seguito del capitolo verrà configurato, in quanto la sua mancata configurazione non permette di usare correttamente la struttura dei registri *general purpouse* dello SPARC.

In figura 6.3, è presentata l'ultima istruzione eseguita dal core in simulazione. Come si nota, in seguito sia il Program Counter (in fetch) che il TIR assumono valori indeterminati. L'istruzione con PC=0F8 è la prima del codice-applicazione provveniente da hello.c (si può verificare controllando il file hello.dump): anche se sembra che anche la seconda istruzione venga eseguita correttamente non è così. L'istruzione "`save %sp, -208, %sp`" tenta di adoperare i registri di stato dei registri *general purpouse*, che sono indeterminati (vedi il CWP in figura 6.2). E' questa un'operazione non concessa neanche all'esecutore hyperpriviledge[3]. La seconda istruzione tenta di impostare a 811_{10} il registro *g1*, il quale non può più

[3]L'esecuzione in hyperpriviledge mode consente di non dover avere obbligatoriamente il processore in stati coerenti, però deve essere in grado di far eseguire al *virtual processor* meccanismi automatizzati solo quando questi hanno la possibilità di essere eseguiti correttamente: è il caso della istruzione SAVE. In generale, non si possono eseguire operazioni differenti dall'inizializzazione con un registro o valore indeterminato.

essere indirizzato e la simulazione si blocca in quanto il processore solo ora "si accorge" che i registri non sono più in uno stato coerente.

6.2 Configurare i registri general purpouse

I registri *general purpouse* sono descritti nel §7.4.

6.2.1 Problematiche: scritura dei *windowed register* di I/O

I registri di stato che gestiscono i registri *gerneral purpouse*, dopo un POR, assumono valore indeterminato, come il CWP in figura 6.3. In virtù della mancata configurazione, non sono utilizzabili i registri di input e output della finestra, essendo il CWP indeterminato. Un set di registri global è invece indirizzabile e utilizzabile in quanto il registro GL viene automaticamente impostato a maxgl (3, nell'OpenSPARC) dopo un POR. E' utilizzabile anche un set di registri *local* in quanto è previsto un set per finestra e non sono condivisi tra diverse *window*. Si può modificare il boot di base (§6.1.2) e inserire all'inizio del file[4] le istruzioni che scrivono sui registri di input e output della finestra corrente "inesistente", per poi ritrovarli facilmente in simulazione; per medesime ragioni, 4 NOP sono aggiunte dopo il nuovo codice (qui sono riportate commentate).

Tenere presente, che i registri di *input* e *output* hanno valore di default 0, come i *global* ed i *local*. La scrittura sui registri local è fatta per avere riferimenti, accorgersi che il processore è in esecuzione.

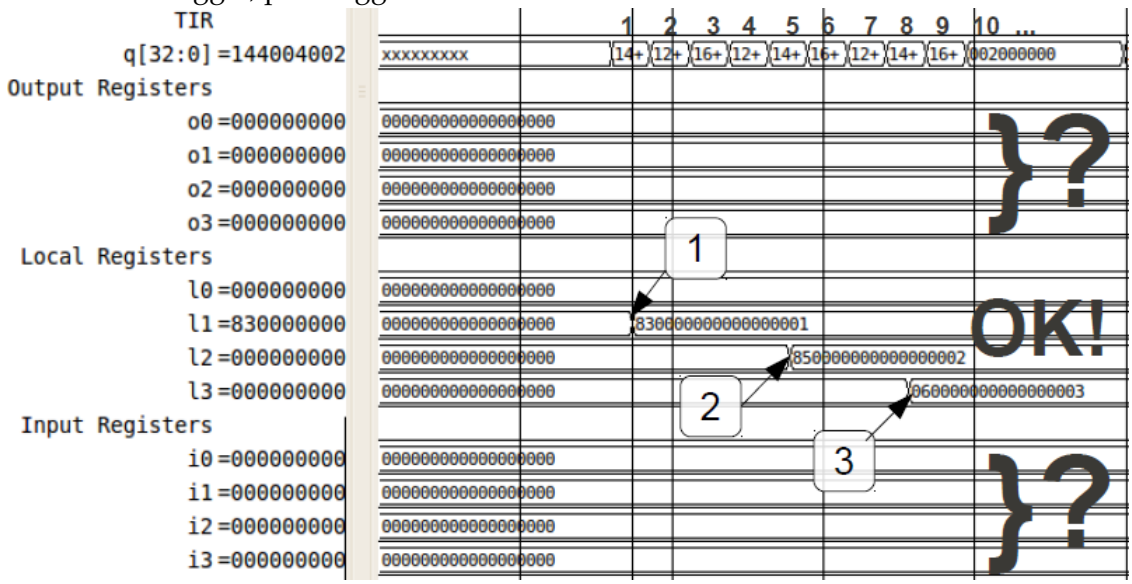
```
! @file boot.s
! modifica test registri general purpouse di I/O
! istruzioni inserite all'inizio del file, dopo le 8 nop.
1       add %g0, 0x1, %l1       !l1 = 1 !! g0 è sempre
   nullo.
2       add %g0, 0x1, %o1       !o1 = 1
3       add %g0, 0x1, %i1       !i1 = 1
4       add %g0, 0x2, %l2       !l1 = 2
5       add %g0, 0x2, %o2       !o1 = 2
```

[4]La posizione, in questo caso, non lede la generalità dei comportamenti allo studio. In fase di test, sono state tentate scritture sui *windowed register* di *input* e output anche in altre fasi della procedura di boot , sia in ROM che in RAM, con uguale risultato.

```
6       add %g0, 0x2, %i2       !i1 = 2
7       add %g0, 0x3, %l3       !l1 = 3
8       add %g0, 0x3, %o3       !o1 = 3
9       add %g0, 0x3, %i3       !i1 = 3
! nop; nop; nop; nop;   !il TIR vale 002000000 quando
                        ! è eseguita una nop.
```

In figura, i risultati della simulazione.

Figura 6.4: Prova di scrittura sui registri windowed di I/O. Mentre la scrittura sui *local* avviene correttamente, i registri di input e output non vengono scritti. In alto, il TIR è stato inserito per rendere conto degli spazi di esecuzione degli istruzioni. E' stato anche inserito il conteggio, per maggiore chiarezza.



Come messo in evidenza nella discussione al §6.1.4.2, l'esecuzione della istruzione SAVE non va a buon fine, e la simulazione si blocca.

6.2.2 Soluzione trovata: scritura dei *windowed register* di I/O

La soluzione sta in una corretta configurazione dei registri priviledge che gestiscono le finestre. Non c'è una soluzione unica, qui ne viene proposta una. Si riporta qui l'equazione che andrebbe rispettata:

$$CANSAVE + CANRESTORE + OTHERWIN = N\_REG\_WINDOWS - 2 \tag{6.1}$$

CWP=0 [vedi §7.4.3] Si sceglie una delle otto finestre disponibili, per esempio 0.

CANSAVE=6 [vedi §7.4.3] Si immagina che tutte le windows siano *valide* e *disponibili* per essere utilizzate dalla corrente applicazione. Questa supposizione, a rigore, non dovrebbe essere accettabile perchè si dovrebbero prima impostare CANSAVE=0, pulire (azzerare i dati) le finestre successive aumentando CWP, e poi tornare alla finestra CWP=0 e impostare a 6 il CANSAVE. Questo non viene fatto perchè durante il POR i registri vengono impostati di tutti a 0, come si vede dall'inizio della simulazione e si sfrutta questa caratteristica per rendere anche più agile il codice delle simulazioni, abbattendone il tempo di calcolo.
Nota: aver impostato il valore di CANSAVE a 6, implica anche aver deciso i valori di OTHERWIN e CANRESTORE per la 6.1. Inoltre, viene dichiarato che la finestra CWP=7 è quella di overlap.

CANRESTORE=0 [vedi §7.4.3] Verrà assegnato 0, perchè effettivamente il codice che si sta eseguendo è quello del primo programma, sicuramente non ci sono procedure chiamanti.

OTHERWIN=0 [vedi §7.4.3] Si sceglie di impostare a 0 OTHERWIN, in quanto non vi sono altri programmi che avrebbero potuto usufruire di qualche finestra, dunque nessuna andrà preservata.

CLEANWIN=7 [vedi §7.4.3] tutte le finestre sono pulite: contengono 0 o dati validi, come ricordato nella discussione sull'impostazione del valore di CANSAVE.

WSTATE=7? [vedi §7.4.3.1] Quando viene generata una eccezione di gestione delle finestre (*window spill* o *fill*) il contenuto di WSTATE.normal o WSTATE.other viene usato per discernere tra varie procedure di gestione (*traps*) dell'errore: faranno parte di un indirizzo di trappola. Come si è

tentato di ribadire, i codici-trappola sono gestiti via software (in larga parte dal sistema operativo) e non sono a disposizione per i test che si stanno qui proponendo: RAM e ROM contengono unicamente codici provenienti da rom_harness.hex e ram_harness.hex, oppure contengono istruzioni NOP (01000000_{16}, vedi §4.2.3.2). Per questo motivo, è indifferente il contenuto di WSTATE e chiaramente si dovrà stare attenti a non generare eccezioni di *spill* e *fill* dei *window register*. Detto questo, verrà inizializzato a 7, perchè è il valore che gli si da di default in un test dell'OpenSPARC T1, i cui codici sono aperti. Dunque WSTATE.other=000_2 e WSTATE.normal=111_2.

Si sottolinea ancora una volta che i vettori trappola ed i *trap handler* non sono configurati dunque quando viene generata una qualsiasi eccezione, questa non verrà gestita (o non verrà gestita correttamente[5]). Questo comporta il fatto di non poter usare molte delle istruzioni che gestiscono in modo automatizzato i *windowed register*, perchè sfruttano le condizioni di errore che vengono generate quando si utilizzano le finestre in casi limite (tipicamente, eseguono finchè non viene chiamata una *window fill* o *spill trap*). Nonostante questo si hanno a disposizione le istruzioni di base SAVE e RESTORE per l'interazione con le finestre: con la configurazione attuata, non può essere lanciata per prima una RESTORE perchè CANRESTORE è nullo, e verrebbe generata una *window fill exception*; potranno altresì essere eseguite fino a 6 SAVE consecutive prima che venga generata una *window spill exception*.

Tenendo conto delle considerazioni di cui sopra, si presenta di seguito un esempio di codice e la relativa simulazione. Le istruzioni sono inserite nel file boot.s subito dopo le 8 NOP iniziali. Sono dapprima configurati i registri interni e

[5]L'indirizzo di base (bit 48:13) a cui viene fatto il salto per eseguire il codice-trappola di gestione delle eccezioni è determinato dall'HTBA, se l'eccezione è generata con livello trappola > maxptl, oppure dal TBA (se l'eccezione è generata con livello trappola $\leq$ maxptl: nel caso dell'S1 Core, se l'indirizzo punta in RAM o ROM le possibilità sono:

1. viene eseguito codice di boot o codice del programma in RAM;
2. gli ultimi 13 bit della trappola indirizzano in ROM/RAM ma oltre l'ultima istruzione scritta (i programmi sono inseriti all'inizio delle due memorie) e vengono eseguite le NOP con cui sono inizializzate di default le memorie nel testbench;
3. la simulazione si blocca perchè viene tentato il fetch da indirizzi di memoria non corrispondenti ad alcun dispositivo.

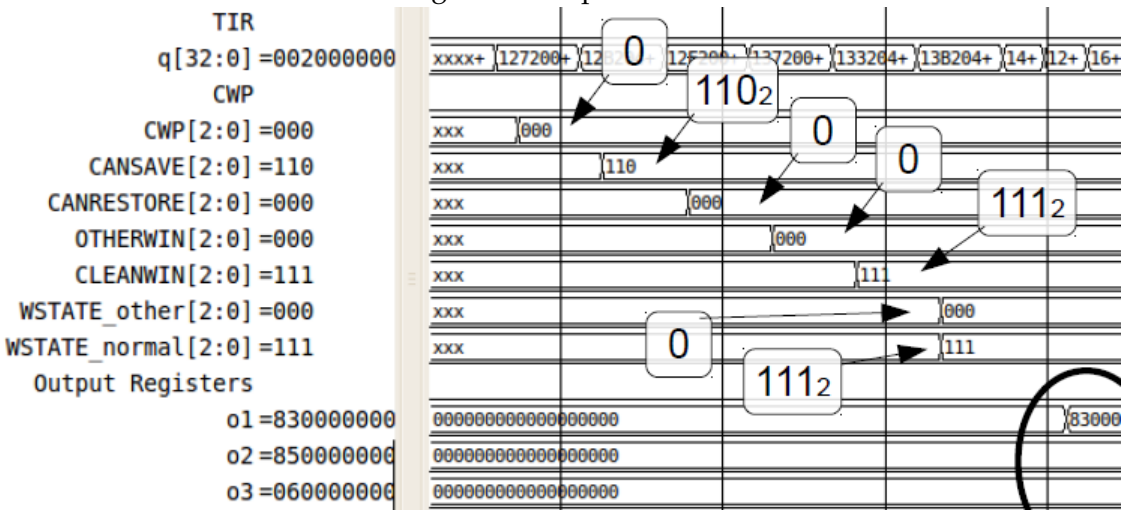
poi ri-eseguito il codice che ha fallito nella precedente simulazione. Questa volta, l'esecuzione delle istruzioni andrà a buon fine e verrà correttamente assegnato il valore desiderato ai registri di I/O. Si può sfruttare questo risultato per evidenziare come i registri vengano anche letti correttamente: saranno mantenute identiche le prime 3 istruzioni, che fanno il set dei registri *l1*, *o1* e *i1*, e nelle successive sarà aggiunto 1 ai registri appena inizializzati per calcolare il valore da immettere nel registro seguente (2, 3). Il codice sarà di chiarimento:

```
! @file boot.s
! modifica test registri general purpouse di I/O
! istruzioni inserite all'inizio del file, dopo le 8 nop.

1       wrpr %g0, %g0, %cwp             ! CWP   = 0
2       wrpr %g0, 0x6, %cansave         ! CANSAVE       = 6
3       wrpr %g0, %g0, %canrestore      ! CANRESTORE    = 0
4       wrpr %g0, %g0, %otherwin        ! OTHERWIN      = 0
5       wrpr %g0, 0x7, %cleanwin        ! CLEANWIN      = 7
6       wrpr %g0, 0x7, %wstate        ! WSTATE = (b)000_111

7       add %g0, 0x1, %l1       !l1 = 1
8       add %g0, 0x1, %o1       !o1 = 1
9       add %g0, 0x1, %i1       !i1 = 1
10      add %l1, 0x1, %l2       !l1 = 2
11      add %o1, 0x1, %o2       !o1 = 2
12      add %i1, 0x1, %i2       !i1 = 2
13      add %l2, 0x1, %l3       !l1 = 3
14      add %o2, 0x1, %o3       !o1 = 3
15      add %i2, 0x1, %i3       !i1 = 3

! nop; nop; nop; nop;   ! nop presenti in simulazione
!* resto del codice di boot *!
```

L'istruzione WRPR scrive nel terzo argomento (*destination*) il risultato della XOR dei primi due (*source_register* e *register_or_immediate*): il secondo argomento può essere un registro o un campo *immediate*, praticamente un numero fino a 13 bit (comprensivi di segno).

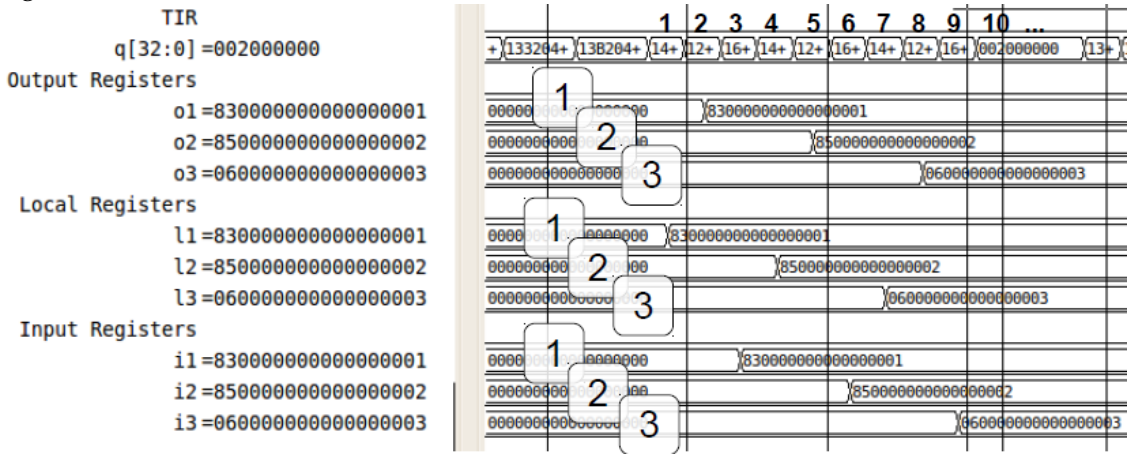
Per mostrare meglio i risultati, la simulazione è riportata in più immagini. In figura 6.5 è riportata la simulazione delle istruzioni priviledge che configurano i registri di stato dei *windowed register*. Nella stessa figura, si intravvede a destra che il registro *o1* è stato scritto.

Figura 6.5: Simulazione: Istruzioni 1-6: i registri priviledge assumono il valore scelto. Si evidenzia a destra in basso che i registri di output iniziano a cambiare.



In figura 6.6 è rappresentata la simulazione delle medesime istruzioni che nella simulazione 6.4 non danno i risultati voluti. Come si può notare, a turno i registri *windowed local*, *output* e *input* assumano i valori assegnati.

Figura 6.6: Simulazione: i windowed register di I/O vengono ora correttamente scritti. I valori presenti nelle etichette a sinistra sono presi in un istante successivo, dove tutti e 6 i registri hanno assunto il valore corretto.



6.2.2.1 Annullamento dei registri windowed, in tutte le finestre

Nella trattazione precedente, è stato detto che i registri windowed, dopo il reset, vanno a 0. Questo è vero, ma la documentazione ufficiale raccomanda, prima di dichiarare (tramite CLEANWIN) che le finestre contengono dati validi (o dati impostati oppure zero), bisognerebbe che venisse generata una trappola (attivata automaticamente eseguendo una SAVE, con CLEANWIN=0) oppure andrebbero impostati "manualmente". A causa della mancata gestione delle trappole, la seconda opzione è più congeniale, allo stato attuale, per l'S1 Core. Si riporta dunque il ciclo che imposta a 0 i registri *windowed*. Il seguente codice dovrebbe essere seguito dalla configurazione dei registri di stato dei *windowed register*:

```
! @file boot.s
! impostare a 0 i windowed register

mov %g0, %g1    ! globals
mov %g0, %g2
mov %g0, %g3 ! ... mov %g0, %g4 ecc fino a 7

mov 7, %g1  ! g1 = maxwin (si può anche ricavare dall'HVER)

reg_init_loop:
wrpr %g1, %cwp  ! il primo CWP = 7
mov %g0, %r8    ! output REG
mov %g0, %r9
mov %g0, %r10   !... fino a r15

mov %g0, %r16   ! local REG
mov %g0, %r17
mov %g0, %r18   !... fino a r23

mov %g0, %r24   ! input REG
mov %g0, %r25
mov %g0, %r26   ! ... fino a r31

brnz %g1, reg_init_loop !'Branch If Not Zero' su g1
```

```
sub %g1, 1, %g1 ! CWP-- (decremento nel delay slot di brnz)

/* Configurare qui i regitri di stato
   dei Windowed Register */
```

6.3 Problematiche: esecuzione di SAVE e RESTORE

E' necessaria l'analisi di queste due istruzioni in quanto la simulazione di default dell'S1 Core si blocca proprio all'esecuzione di una SAVE. Come discusso nel §6.2, la mancata esecuzione della SAVE è dovuta ad un codice di boot carente della configurazione dei registri di stato per la gestione delle finestre. In ultima analisi, una prima risposta al perchè non vengono eseguite le store nella simulazione 5.1, può essere appunto che l'istruzione di STORE non viene eseguita perchè fallisce l'esecuzione di una istruzione precedente (la SAVE). In realtà, questa non è la sola causa: fallirà anche quando sarà poi fisicamente eseguita, e le ragioni dell'ulteriore fallimento verranno sondate in seguito.

6.3.1 Richiami teorici su SAVE e RESTORE

Per la trattazione completa sulle istruzioni di SAVE e RESTORE, e riferirsi alla documentazione ufficiale §3.1.2.4.

Quando viene chiamata una SAVE o una RESTORE, cambiano valore alcuni registri di stato delle *window* come mostrato in tabella 6.1:

Tabella 6.1: Evoluzione dei registri di stato alla esecuzione di SAVE e RESTORE.

	CWP	CANRESTORE	CANSAVE	OTHERWIN	CLEANWIN	WSTATE
SAVE	+1	+1	-1	—	—	—
RESTORE	-1	-1	+1	—	—	—

CWP viene incrementato o decrementato modulo *N_REG_WINDOWS* -1; CAN-

SAVE e CANRESTORE si comporteranno sempre secondo l'equazione 6.1, se al momento dell'esecuzione di una delle due istruzioni tale equazione è già rispettata.

L'effetto è di cambiare l'indirizzo della finestra corrente, e dunque cambiare la "vista" sui registri *general purpouse*. La forma i SAVE e RESTORE è la seguente:

```
save %source_reg, reg_or_imm, %dest_reg          ! SAVE
restore %source_reg, reg_or_imm, %dest_reg       ! RESTORE
```

SAVE e RESTORE hanno lo stesso comportamento e la stessa forma della istruzione ADD, e accettano dunque 3 argomenti: i registri di *source* (i primi due argomenti, di cui il secondo sostituibile con un campo *immediate*) sono prelevati dalla finestra corrente, sono addizionati e la somma è inserita nel registro di destinazione (terzo argomento), il quale appartiene però alla prossima *window* (puntata da CWP+1 o CWP-1).

L'esecuzione di una operazione di addizione contestualmente a SAVE e RESTORE ha uno scopo ulteriore, rispetto alla possibilità di eseguire "due istruzioni in una", ed è la gestione dello *STACK*. Lo *STACK* è un array di dati che tipicamente occupa gli indirizzi più significativi della RAM. L'addizione serve dunque al passaggio da una finestra all'altra di riferimenti allo *STACK POINTER*. Lo *STACK POINTER* è indicato come registro "*%sp*", convenzionalmente corrispondente ad *o6*.

Dopo l'esecuzione di una SAVE, il registro *o6* (*sp*) viene indirizzato nella nuova finestra da *i6*: *i6* prende il nome di *FRAME POINTER* ed è indicato in assembly anche con "*%fp*".

Dopo l'esecuzione di una RESTORE, invece, è il registro *i6* (*fp*) che verrà usato nella nuova finestra (in realtà si sta tornando ad una finestra precedente) come *o6* (*sp*).

6.3.2 Simulazione: SAVE e RESTORE eseguono correttamente

Si descrive la costruzione ed esecuzione di codice costruito ad hoc per evidenziare che l'esecuzione delle istruzioni di SAVE e RESTORE, una volta configurati i registri priviledge di gestione delle finestre, sono eseguite correttamente a patto che non generino eccezioni *window fill* e *window spill*. La discussione nel merito del codice generato dal cross-compilatore da hello.c è rimandata ad altra sezione.

Poichè la corretta esecuzione delle istruzioni SAVE e RESTORE è imprescindibile dalla inizializzazione dei registri priviledge di gestione delle finestre, si riporta parte del codice già presentato al §6.2.2. Si aggiungano le due istruzioni di SAVE e RESTORE, rigorosamente in sequenza, altrimenti verrà generata una eccezione:

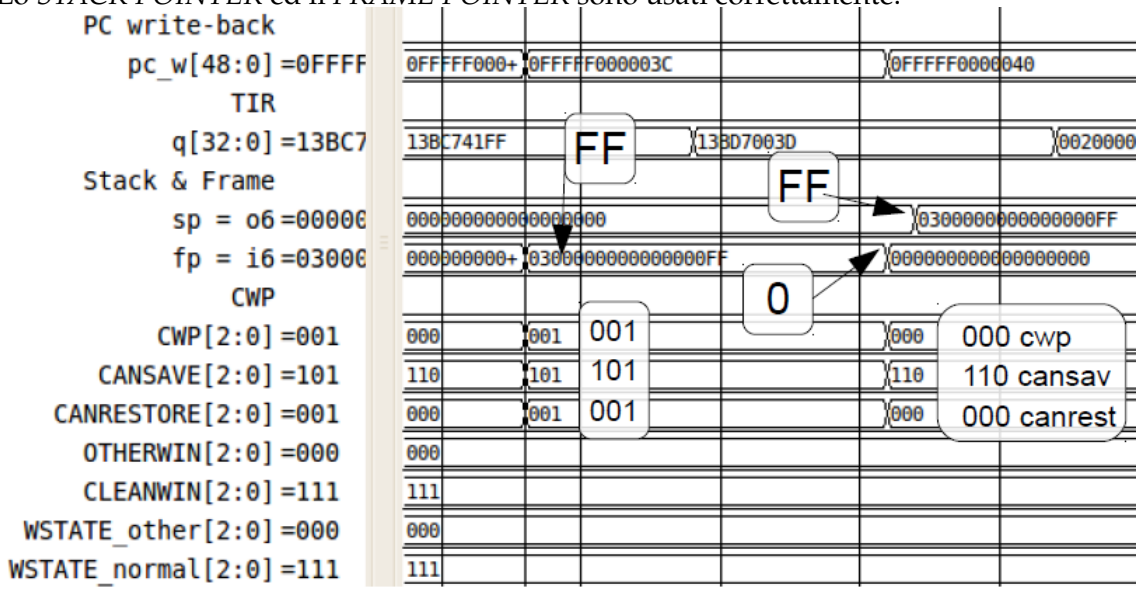
```
! @file boot.s
! modifica test registri general purpouse di I/O
! istruzioni inserite all'inizio del file, dopo le 8 nop.

1       wrpr %g0, %g0, %cwp             ! CWP   = 0
2       wrpr %g0, 0x6, %cansave         ! CANSAVE       = 6
3       wrpr %g0, %g0, %canrestore      ! CANRESTORE    = 0
4       wrpr %g0, %g0, %otherwin        ! OTHERWIN      = 0
5       wrpr %g0, 0x7, %cleanwin        ! CLEANWIN      = 7
6       wrpr %g0, 0x7, %wstate        ! WSTATE = (b)000_111

7       save %sp, 0xFF, %sp             ! SAVE
8       restore %sp, %fp, %sp           ! RESTORE

! nop; nop; nop; nop;   ! nop presenti in simulazione
!* resto del codice di boot *!
```

In figura 6.7, è riportata la simulazione di SAVE e RESTORE; nelle forme d'onda, sono stati evidenziati i valori assunti dai registri di configurazione per una migliore visualizzazione. E' riportato anche il plot del PC(wb), per far notare come i registri di stato delle finestre vengano scritti nello stage di *write-back*.

La simulazione conferma come:

- all'esecuzione delle due istruzioni cambiano CWP, CANRESTORE e CANSAVE, secondo la tabella 6.1;
- *o6* (*sp*) è inizialmente nullo e, all'esecuzione della SAVE, viene aggiunto FF_{16} e salvato nel registro *i6* (*fp*) della nuova finestra;
- *o6* (*sp*) è ancora nullo e *i6* (*fp*) vale FF_{16}: all'esecuzione di RESTORE, vengono sommati e *o6* (*sp*) della nuova finestra assume valore FF_{16}.

Figura 6.7: Simulazione: SAVE e RESTORE sono eseguite correttamente. I regostri di stato cambiano all'esecuzione della SAVE e sono ripristinati dall'esecuzione di RESTORE. Lo *STACK POINTER* ed il *FRAME POINTER* sono usati correttamente.



6.4 Eccezione nella esecuzione della prima STORE

Ora che l'istruzione SAVE può essere eseguita, il codice-applicazione provveniente dalla compilazione di hello.c può continuare ad essere eseguito. Le prime 4 istruzioni del codice-applicazione sono:

```
! @file hello.s
! prime 4 istruzioni eseguibili

save %sp, -208, %sp      ! SAVE
mov  811, %g1            ! g1 = 0x32B
sllx %g1, 6, %g1         ! g1 = CAC0 [shift left operation]
stx  %g1, [%fp+2023]  ! STORE g1(64bit) to addr 0x00001F53
!... resto del codice ...
```

Di queste istruzioni è fatto il fetch da ROM, in quanto è stata eliminata l'istruzione JMP dal alla fine del codice di boot (boot.s) e fatto il *join* dei file esadecimali: ram_harness.hex è stato copiato in coda a rom_harness.hex. Il codice di boot che viene simulato è quello descritto nel §6.1.2, cui sono stati aggiunti, dopo le 8 NOP iniziali,

1. le 15 istruzioni presentate al §6.2.2: 6 di configurazione dei registri *windowed* e 9 per il test in scrittura dei registri local, input e output;

2. le 2 istruzioni SAVE e RESTORE per il test delle stesse: i registri di stato delle finestre sono dunque nel resto della simulazione nella configurazione dichiarata in §6.2.2. La RESTORE è stata però modificata in modo che il registro o6 della finestra che verrà usata durante la simulazione torni a 0:

```
restore %sp, -0xFF, %sp
```

3. 4 NOP.

Come tra poco verrà discusso, anche se la simulazione non si blocca (nel senso che il processore continua ad eseguire istruzioni fino al tempo fornito nel testbench) non è andata a buon fine.

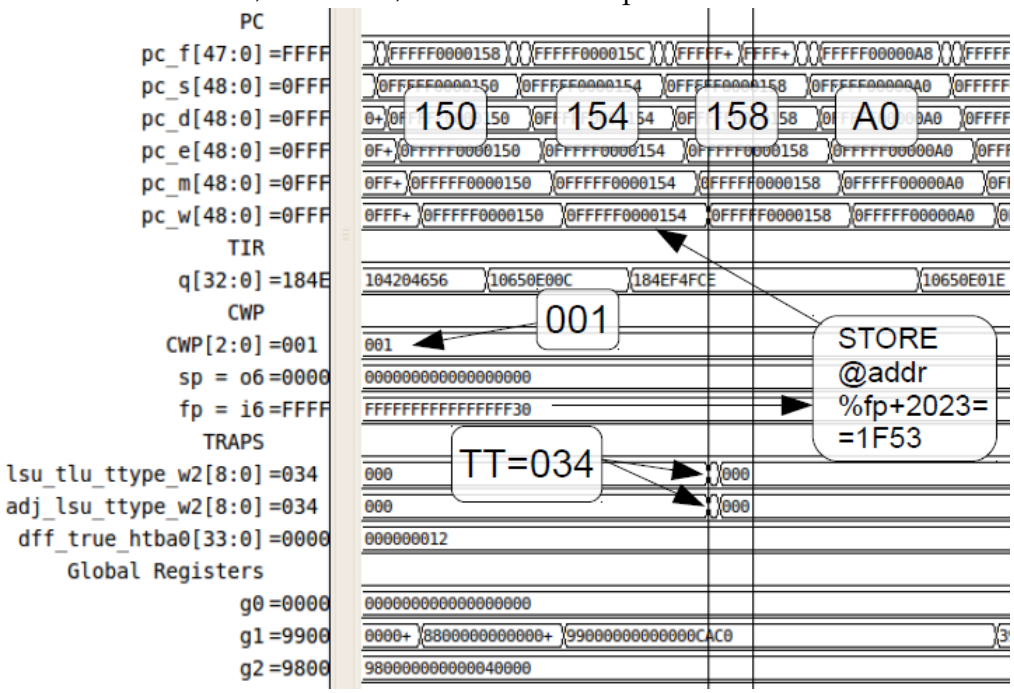
In figura 6.8, l'esecuzione delle istruzioni di cui sopra. Rintracciarle nella simulazione è semplice: dove, per la quarta volta, cambia il CWP, è appena stata eseguita la prima istruzione (cambia nella fase di *writeback* del PC). In figura il CWP è già ad 1, perchè si è tagliata l'esecuzione della SAVE, già commentata in precedenza, in favore di una migliore visualizzazione delle restanti istruzioni.

Nella figura 6.8, il CWP è appena cambiato e, come si vede, *fp* è negativo, perchè la SAVE toglie 208 ad *sp* che è nullo. Si legge al centro in basso che g1 viene poi scritto con $CAC0_{16}$. A questo punto, mentre il PC = $0FFFF0000154_{16}$ viene eseguita la STORE di una *doubleword*, all'indirizzo *fp*+2023_{10} = *fp*+$7E7_{16}$ = $1F53_{16}$. Tale indirizzo non è allineato con le parole da 64 bit: gli ultimi bit dell'indirizzo di una *doubleword* devono avere i 4 bit meno significativi 8_{16} oppure 0_{16}.

Tale indirizzo, è inoltre fuori dallo spazio di RAM dell'S1 Core, poichè la RAM ha indirizzi compresi nel *range* [40000_{16}:$4FFFF_{16}$].

Lo SPARC core risponde a tale violazione con la generazione di una eccezione *mem_address_not_aligned*. Tale eccezione attiva una *precise trap*: prima di attivare il codice-trappola vengono eseguite tutte le altre istruzioni lanciate prima della STX (PC[11:0]=154_{16}), che ha generato l'eccezione (anche quelle degli altri *strand*). Per questo prima di saltare all'indirizzo-trappola, viene eseguito un altro ciclo di pipeline (PC[11:0]=158_{16}) ma, come si vede, nel TIR non viene caricata un'altra istruzione.

Figura 6.8: Simulazione: fail della prima STORE, generazione della eccezione *mem_address_not_aligned* con TT[TL]=0 e TT=34_{16}. Dei PC, sono evidenziati i bit meno significativi. Tranne 2023, in base 10, i numeri sono espressi in codice esadecimale.



A questo punto viene chiamato il codice-trappola, tramite la costruzione dell'indirizzo di trappola corrispondente alla eccezione ed allo stato dello *strand*. L'indirizzo cui viene fatto il salto, è $A0_{16}$.

6.4.1 Gestione della eccezione in *Red state*

Nella simulazione in figura 6.8, viene scelto, alla generazione della eccezione, di far saltare il PC (e dunque l'esecuzione del thread) all'indirizzo $A0_{16}$ per i seguenti motivi:

- quando viene generata l'eccezione, avendo scelto di eseguire in *RED state*, i primi 50 bit dell'HTBA (a 64bit) sono predeterminati e tutti a 1 meno che i bit [15:14] ;

- nei bit [13:5] viene inserito TT[TL], 0 nel caso in esame, perchè settato a 0 alla fine del file boot.s;

- E' previsto un offset per ogni tipo di reset, ad iniziare con 20_{16} se è un POR[6], a maggiore priorità, aggiungendo 20_{16}, fino al SIR (*Software-Initiated Reset*) che ha priorità più bassa ed offset 80_{16}. A tutte le altre trappole generate in *RED state* è assegnato l'offset $A0_{16}$: si tratta del caso in esame.

Vista la costruzione (hardware) degli indirizzi per le trappole in *RED state*, risulta sbagliato scrivere il codice di boot, come è stato fatto, a partire dall'indirizzo 20_{16} in poi, perchè si occupano spazi di memoria (ROM) dedicati alla gestione delle trappole in *RED state*. Sarebbe invece corretto utilizzare le prime 8 istruzioni (tante sono, fino all'*address* 40_{16} di ROM, dedicato al *Watchdog Reset*) a disposizione dall'indirizzo 20_{16} per poter eseguire un salto ad un indirizzo di RAM. Saltare ad un indirizzo di RAM piuttosto che ad un altro di ROM, che sicuramente sarà oltre $A0_{16} + 20_{16} = C0_{16}$, per un motivo preciso: la ROM dell'S1 Core dispone di indirizzi nel range di *offset* [000_{16}:FFF_{16}], ovvero 12 bit.

In realtà, 12 bit non bastano per la gestione delle trappole: per quelle hyperpiviledge (HTBA) sevono 14 bit di offset nel range [0_{16}:$3FFF$] e memoria da 512 parole da 32 bit (indirizzate con granularità del byte): 16 Kbyte. Per gestire anche le trappole del software priviledge il registro di competenza per l'indirizzo base è il TBA, il quale prevede un ulteriore bit, il bit TBA[14] , a 1 quando la trappola è stata generata da codice eseguito in TL=0: servirebbe ulteriore spazio di indirizzi [0_{16}:$7FFF_{16}$]. Gli ultimi 5 bit dei vettori-trappola sono sempre nulli, pertanto i *trap-handler* sono di un minimo di 8 istruzioni ciascuno.

Naturalmente, visto che l'S1 Core è sprovvisto di *handler* per le trappole (a meno di non modellare l'immagine di RAM e ROM anche con i codici-trappola dall'OpenSPARC T1 o con l'esecuzione, dopo il boot, di un sistema operativo), si può decidere di assegnare un indirizzo di RAM come indirizzo base per TBA e HTBA. Per esempio, all'HTBA, nel codice di esempio al §6.1.2 è stato deciso di assegnare come indirizzo-base il 48000_{16}; il TBA non è stato impostato ed ha ancora valore indeterminato; il TBA viene usato come indirizzo-base quando vengono generate alcune trappole da codice di livello utente o priviledge.

Si è detto che la ROM non basterebbe per inserire i codici-trappola dello SPARC core. La RAM basterebbe appena per gestire i *trap-handlers* (8 istruzioni per trappola, necessari a generare un indirizzo e fare il *jump*): verrebbero dedicati i 3/4

[6]Proprio per questo si sono inserite 8 NOP iniziali nella procedura di boot: la simulazione dell'S1 Core inizia con l'invio, tramite *reset controller*, di un *Power On Reset* al core il quale entra subito in *RED state* con TL=maxtl e salta all'indirizzo di ROM 20_{16}.

della RAM per la sola gestione degli indirizzi per i gestori di trappola. In questo caso, si potrebbe pensare di assegnare il primo quarto di indirizzi al codice ([40000_{16}:$43FFF_{16}$]), il secondo quarto di RAM ([44000_{16}:$47FFF_{16}$]) per le trappole chiamate in hyperpriviledge mode (all'HTBA), e la seconda metà ([48000_{16}:$4FFFF_{16}$]) alla gestione da parte dell'HTBA del software hyperpriviledge. TBA e HTBA sarebbero allora così settati:

```
        !TBA 48000:4BFE0 per TL=0
        !TBA 4C000:4FFE0 per TL>0
sethi %hi(48000), %l0
wrpr %l0, %g0, %tba    !bits 63-15 select Priv trap vector
        !HTBA 44000:4BFE0
sethi %hi(44000), %l0
wrhpr %l0, %g0, %htba  !bits 63-14 select Hpriv trap vector
```

Per la sola esecuzione del codice di boot (158_{16} byte) e qualche semplice programma di test, fino a circa quattromila istruzioni, sarebbe accettabile.

Potrebbe essere utilile stabilire un compromesso, per poter dedicare più spazio ai programmi. Se si accettasse di non poter eseguire codice di livello utente, allora si potrebbe imporre uguali indirizzi per l'HTBA ed il TBA: si progetterebbe così di inserire nel terzo quarto di RAM gli handler per codice hyperpriviledge al posto di quelli per TL=0 e dedicare l'ultimo quarto di RAM al codice di gestione delle trappole priviledge (facente riferimento al TBA). Si ricorda che in ogni caso questi calcoli sarebbero utili solo "a livello organizzativo", in quanto si sta dedicando spazio bastante appena perchè ogni handler di trappola costruisca un indirizzo a 64 bit e vi faccia il JMP (8 istruzioni per eccezione). In ogni caso, basterebbe una RAM più grande per inserire il codice di gestione delle trappole, vedi §6.4.2. Se si accetta il suddetto compromesso, la gestione delle trappole occuperebbe la metà alta della RAM (la quale dispone di uno spazio di indirizzi con offset a 16 bit [40000_{16}:$4FFFF_{16}$], i 3 bit più significativi dell'*address* sono fissi) si potrebbero impostare TBA e HTBA come segue:

```
        !TBA 48000:4BFE0 per TL=0
        !TBA 4C000:4FFE0 per TL>0
sethi %hi(48000), %l0
wrpr %l0, %g0, %tba     !bits 63-15 select Priv trap vector
        !HTBA 48000:4BFE0
```

```
wrhpr %l0, %g0, %htba  !bits 63-14 select Priv trap vector
```

Nel pacchetto dell'S1 Core, nella cartella $S1_ROOT/docs/other è riportato il codice disassemblato del *reset handler* per *RED state* dell'OpenSPARC T1, salvato in
$T1_ROOT/verif/diag/assembly/include/hred_reset_handler.s, e la procedura è proprio quella di usare le prime 8 istruzioni di ROM dalla 20_{16} per fare un jump in RAM e fare poi il fetch da RAM per l'inizializzazione. Tale codice è in parte ripreso ed in parte commentato nel file boot.s ufficiale dell'S1 Core. Le prime 8 istruzioni del reset handler in *RED state* dell'OpenSPARC T1 (cross)compilato con gcc, è riportato di seguito:

```
! OpenSPARC T1 hpriv_reset_handler.s
! gcc-compiled code
fff0000020:  03 00 00 00  sethi  %hi(0), %g1
fff0000024:  05 00 01 00  sethi  %hi(0x40000), %g2
fff0000028:  82 10 60 00  mov  %g1, %g1
fff000002c:  84 10 a0 c0  or  %g2, 0xc0, %g2
fff0000030:  83 28 70 20  sllx  %g1, 0x20, %g1
fff0000034:  84 10 80 01  or  %g2, %g1, %g2
fff0000038:  81 c0 80 00  jmp  %g2   ! jump in RAM @40000
fff000003c:  01 00 00 00  nop
```

6.4.1.1 Cambiare il codice di ROM per la gestione delle trappole in RED state

I primi indirizzi di ROM, sono dedicati alla creazione degll'indirizzo per i gestori di trappola dei reset, più un indirizzo generico per le trappole che si generano in *RED state*. Anche nella piccola ROM dell'S1 Core, si può pensare di lasciare lo spazio per la gestione delle trappole in *RED state*. Basta una piccola modifica: Nelle prime 8 istruzioni eseguite dopo il POR (address 20_{16} - 36_{16}) si inserisce la costruzione l'indirizzo $A0_{16} + 8_{[istruzioni]} * 4_{[\frac{byte}{istruzione}]} = C0_{16}$ e vi si fa il jump. Il resto di istruzioni del POR *handler*, andranno inserite a partire da questo indirizzo. Senza dover fare hacking su rom_harness.hex, si inseriranno semplicemente $(C0_{16} - 40_{16})/4 = 32_{10}$ istruzioni NOP.

```
! @file boot.s
! istruzioni da inserire all'inizio del file.
```

```
nop     ! ... inserire 8 NOP per gli indirizzi [0x0:0x1C]
! costruire in 8 istruzioni l'indirizzo per il gestore del
   POR (codice di boot): indirizzi [0x20:0x3C]
sethi  %hi(0), %g1
sethi  %hi(0x0), %g2
mov  %g1, %g1
or  %g2, 0xc0, %g2
sllx  %g1, 0x20, %g1
or  %g2, %g1, %g2
jmp  %g2                ! jump in ROM @C0
nop
! scrivere 32 NOP per lasciare spazio per la gestione delle
    trappole in RED state
nop     ! ... altre 32 nop per indirizzi [0x40:0xBC]
```

6.4.2 Cambiare dimensione a RAM e ROM dell'S1 Core

Può rendersi necessario avere ROM o RAM più grandi, quantomeno in simulazione. Bastano modifiche a poche istruzioni Verilog del testbench, in $S1_ROOT/rtl/behav/testbench/testbench. Si prenda il caso che si voglia incrementare di 4 bit gli indirizzi di RAM (da 16 bit a 20 bit) e ROM. Subito dopo la dichiarazione dei wire, sono assegnati gli spazi di indirizzi per ROM e RAM, e vanno cambiati: vegono riportate di seguito segmenti del file per RAM e ROM incrementate.

```
// Decode the address and select the proper memory bank
assign wb_cycle_rom  = ((wb_addr[39:16]==24'hFFF000) ?
   wb_cycle : 0);
assign wb_strobe_rom = ((wb_addr[39:16]==24'hFFF000) ?
   wb_strobe : 0);
assign wb_cycle_ram  = ((wb_addr[39:20]==20'h00004) ?
   wb_cycle : 0);
assign wb_strobe_ram = ((wb_addr[39:20]==20'h00004) ?
   wb_strobe : 0);
```

Occorre ancora cambiare la dimensione fisica delle memorie. Entrambe sono istanze del medesimo modulo, mem_harness, il quale accetta alcuni parametri tra cui il numero di bit degli indirizzi della memoria che modella. Tali parametri sono definiti alla fine del testbench: come viene ricordato nei commenti, le linee di memoria sono da 64 bit mentre gli indirizzi dello SPARC hanno granularità del byte, dunque bisogna indicare il numero di bit di indirizzo cui è sottratto 3.

```
// ROM has Physical Address range
//      [0xFFF0000000 : 0xFFF000FFFF]
defparam rom_harness.addr_bits = 13;
// RAM has Physical Address range
//      [0x0000400000 : 0x00004FFFFF]
defparam ram_harness.addr_bits = 17;
```

Bisogna rieseguire il *build* del progetto prima di lanciare una nuova simulazione per simulare sul nuovo design.

6.4.2.1 Test su RAM e ROM incrementate

I programmi eseguiti su RAM e ROM con diverse grandezze, dovranno tenere conto di questo nella formulazione esplicita degli indirizzi. Nel caso che vengano incrementate di 16 volte, aggiungendo 4 bit di indirizzamento, nei test dovranno essere certamente cambiati gli indirizzi di RAM, in quanto i precedenti non sono più validi: istruzioni di salto incondizionato (JMP), e di set esplicito degli indirizzi pensati per puntare alla RAM di default dell'S1 Core, ora si trovano a puntare all'esterno dello spazio di indirizzi della memoria. Gli indirizzi di ROM, invece, saranno ancora validi, perchè il primo indirizzo di ROM (a parte l'estensione dei bit) è passato da $000_{16} \longrightarrow 0000_{16}$; il primo indirizzo di RAM, invece, non parte più da 40000_{16} ma da 400000_{16}.

Prendendo ad esempio il solo file boot.s, è necessario riscrivere solo alcune istruzioni, afferenti al jump in RAM e al set dll'HTBA. Sono aggiunte anche le istruzioni necessarie al set del TBA, progettato in modo che non vi sia sovrapposizione con gli indirizzi dedicati alle trappole in *hyperpriviledge mode* nel HTBA. Per una simulazione di prova, mantenendo intatto il codice di boot precedente, tali istruzioni possono essere aggiunte in coda (prima del JMP) e può essere decommentato il JMP per verificare che venga eseguito il fetch da RAM.

Figura 6.9: Simulazione dell'ultima istruzione di ROM e fetch della prima istruzione di RAM, con indirizzi di RAM a 20bit [400000_{16} : $4FFFFF_{16}$] e di ROM a 16 bit [0000_{16} : $FFFF_{16}$]. La simulazione è effettuata su Modelsim.


```
! @file boot.s: Aggiunto alla fine del file
! Set Jump address to RAM
sethi  %hi(0), %g1
sethi  %hi(0x400000), %g2  ! Jump address to RAM code
mov  %g1, %g1
mov  %g2, %g2
sllx  %g1, 0x20, %g1
or  %g2, %g1, %g2  ! 64bit jump address fully initialized.
! Set TBA & HTBA
sethi %hi(0x448000), %g4
wrpr %g4, %g0, %tba  ! bits 63-15 select Priv trap vector
sethi %hi(0x440000), %g4
wrhpr %g4, %g0, %htba  !bits 63-15 select Hpriv trap vector
! Jump to first RAM address
jmp %g2
nop
```

Una simulazione con i nuovi indirizzi, è mostrata in figura 6.9. Viene mostrato il Program Counter che richiede i fetch delle prime istruzioni di RAM.

6.5 Gerarchia di memoria e gestione degli indirizzi

Vi sono varie questioni legate al corretto indirizzamento della memoria, all'uso delle caches, all'attivazione delle MMU e alla traduzione degli indirizzi. Alcuni di questi argomenti vengono qui analizzati, ed è proposta una soluzione compatibile con il core.

6.5.1 Correzione dell'allineamento

Si è fatto notare nel §6.4 come alla tentata esecuzione della prima STX, venga generata l'eccezione *mem_addr_not_aligned* con TT=34_{16}. L'eccezione è dovuta al fatto che si tenta di accedere ad una parola da 64 bit da un indirizzo non allineato con i 64 bit: gli indirizzi allineati con doubleword, devono avere gli ultimi 3 bit a 000_2.

Effettivamente, l'indirizzo cui punta la prima STORE (e le altre, anche se non eseguite), non rispetta questa regola: è tentato l'accesso all'address $fp+2023_{10} = fp+7E7_{16} = 1F53_{16}$.

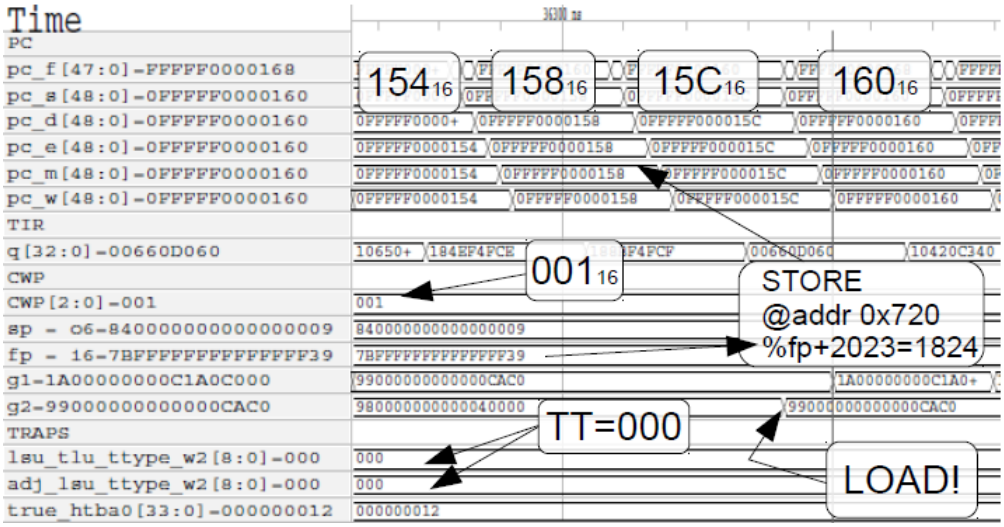
Prima di avanzare una soluzione, si fa qualche considerazione sul file hello.s.

Il gcc salva le variabili locali del programma nello *stack*: nel main è creata una variabile locale (*address*) che viene inizializzata. Il gcc risponde configurando un offset rispetto allo *STACK POINTER*, nel quale farà la store del dato all'atto dell'inizializzazione, e ne farà la load ogni volta che viene usata tale variabile. Questo è il motivo per cui nel codice generato dal gcc vi sono più store e load di quelle dichiarate nel main.

Sapendo in anticipo che il programma generato dal gcc farà prima una SAVE togliendo 208 al vecchio stack pointer (*o0*, attualmente a 0 nel boot.s) e poi vi aggiungerà 2023 (una volta diventato *FRAME POINTER* nella nuova finestra), si può progettare di settare nella procedura di boot lo *STACK POINTER* in modo che l'indirizzo finale sia allineato con dati da 64 bit. Si può aggingere nel file boot.s, per esempio appena prima del *jump*, la seguente riga:

```
! @file boot.s
mov 0x9, %sp    ! 2023 - 208 = 1815,
                        ! 1815 + 9 = 1824
                        ! 1824(decimale) = 720 (hex)
```

Figura 6.10: Simulazione: l'esecuzione della prima STX non genera eccezioni. La simulazione è effettuata con Icarus Verilog, in un'altra modalità di visualizzazione.



In figura 6.10 è riportata la simulazione equivalente a quella riportata in figura 6.8, effettuata nelle medesime condizioni, con l'aggiunta dell'istruzione `mov 0x9, %sp`. Ora non è più generata una eccezione e dunque la simulazione prosegue senza fare il fetch del codice-trappola.

Va evidenziato che 720_{16} non è un indirizzo di RAM, evita però la generazione dell'eccezione.

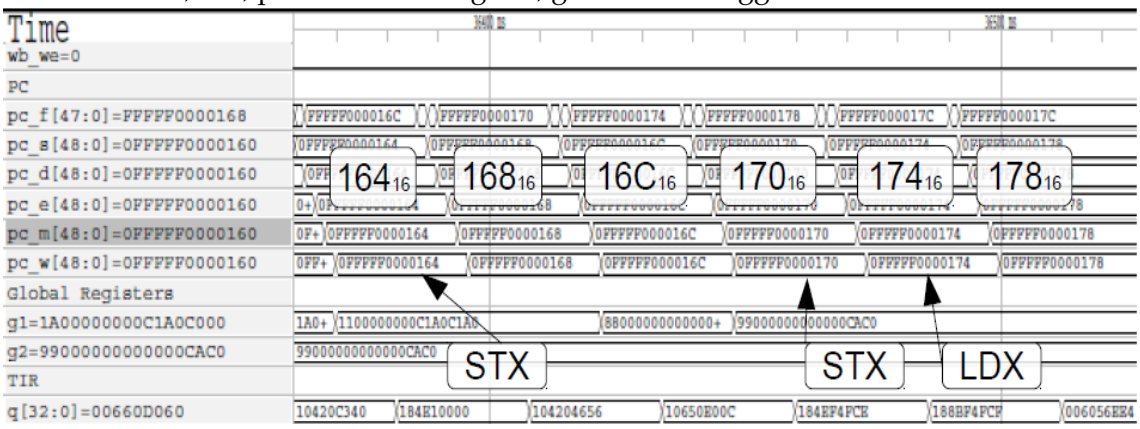
6.5.2 Una sequenza STORE-LOAD eseguita correttamente?

In figura 6.10 può apparire corretta l'esecuzione della prima sequenza STORE-LOAD di hello.s. Avendo come riferimento i registri globali, sembrano eseguite correttamente le istruzioni:

```
stx     %g1, [%fp+2023]
ldx     [%fp+2023], %g2          ! g2 = 0xCAC0
```

Infatti il registro *g2* prende il valore esatto, sembra, caricando il dato dalla memoria. La memoria invece non viene scritta e non v'è nessuna richiesta di store al bus: in figura 6.11 è inserito anche il segnale di *write enable* (wb_we), alto quando il dispositivo puntato deve essere scritto, che invece rimane sempre a 0. Neanche l'*address-bus* prende come indirizzo $CAC0_{16}$.

Figura 6.11: Simulazione: seguito della figura 6.10. Viene effettuata la nuova sequenza di store e load, che, pur venendo eseguita, g2 non viene aggiornato.



Iniziamo col dire che $CAC0_{16}$ non è un indirizzo valido di RAM. In ogni caso, il registro *g2*, almeno una volta, viene non solo aggiornato ma scritto col dato corretto. Un elemento diverso dalle altre scritture sui registri, la scrittura su *g2* ce l'ha: viene impostato il suo valore non nello stadio di *writeback*, ma in quello di *select* (del thread).

Tutte queste motivazioni insieme, hanno fatto pensare ad una bufferizzazione del dato, prima che venga mandato sul bus. In effetti lo SPARC core ha un buffer in uscita (STORE) ed uno in ingresso (LOAD) per fare in modo che il TSO (Total Store Order) possa essere rispettato. LOAD e STORE devono essere eseguite nella sequenza con cui sono state richieste. Questo non è esattamente vero, nel senso che la sequenza di invio sul bus può essere modificata tramite un algoritmo di riordinamento che lavora in modo tale da ottimizzare i tempi, mantenendo l'ordine tra load e store che lavorano sugli stessi dati in modo che mai venga utilizzato un dato non aggiornato. La profondità del buffer è di 2 (per thread). La terza STX, al PC con offset 170_{16}, non causa un flush del buffer (e conseguente richiesta sul bus dei effettuare la prima STX in figura §6.10) perchè è identica alla prima, anche il registro di destinazione è lo stesso. Inoltre il dato bufferizzato è ancora valido: non c'è un'altro core che possa cambiarlo e non è stata effettuata una diversa STORE sullo stesso indirizzo. Per questo motivo la prima LOAD imposta bene *g2* al valore $CAC0_{16}$: il dato è ancora presente nello store buffer, è valido e non c'è motivo di caricarlo da RAM, viene direttamente preso dallo store buffer (interno allo SPARC core). La seconda LOAD, poichè il TSO va rispettato, prima di essere eseguita necessita che siano effettuate le STORE verso la memoria e dunque con

delle transizioni sul bus.

Le store non vengono però eseguite a causa del fatto che gli indirizzi verso l'esterno del core concernenti i dati, vengono tradotti dalla MMU (anche se disabilitata), a differenza delle istruzioni. Il fetch delle istruzioni avviene correttamente, dagli indirizzi fisici perchè quando si è in *RED state* ed in *hyperpriviledge mode* le istruzioni non vengono mai tradotte. Vengono tradotti invece gli indirizzi riguardanti scambi di dati con l'esterno del core: bisogna avere configurati i registri *context* e *partition ID*, oltre che al D-TLB correttamente configurati per avere una traduzione corretta.

6.5.2.1 STORE-LOAD con indirizzo fisico corretto

Si vuole verificare il comportamento del core alla esecuzione delle medesime istruzioni, ma con LOAD e STORE ad indirizzi fisici tutti appartenenti allo spazio di RAM. Un modo semplice per farlo è cambiare gli indirizzi direttamente nel file C hello.c. Modificati gli indirizzi da $CAC0_{16}$ in $4CAC0_{16}$, il file conterrà le seguenti istruzioni:

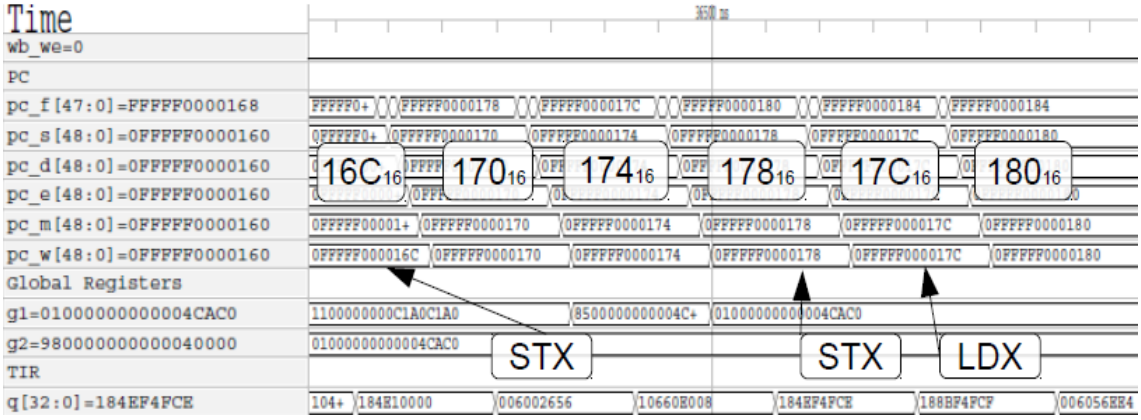
```
int main() {
        unsigned long* address;
        address = (unsigned long*)0x0004CAC0;
        (*address) = 0xC1A0C1A0;  // First store
        address = (unsigned long*)0x0004CAC0;
        (*address) = 0x0ADDCAFE;  // Second store
        return 0; }
```

Inoltre, si deve fare in modo che anche le STORE che vengono fatte sfruttando lo *STACK POINTER* siano fatte verso la RAM. Nel file boot.s, si ridefinisca lo *STACK POINTER* della finestra con CWP=0 (già impostata):

```
! Stack Pointer Align
setx 0x40209, %g4, %sp  ! è importante il 9 e che
        ! l'address finale sia oltre le istruzioni eseguite
```

La simulazione corrispondente, è riportata in figura 6.12. La traduzione in assembly del nuovo file boot.s prevede due istruzioni in più di quelle generate con il vecchio, dovute alla SETX.

Figura 6.12: Simulazione: STORE-LOAD con indirizzo fisico corretto. La simulazione si blocca con PC=178_{16}.



Come si può notare, il comportamento è identico a quello della simulazione in figura 6.11. Significa questo che tale comportamento non è dovuto all'indirizzo in sè, ma al fatto che viene tentata una traduzione che fallisce.

6.6 Unità di traduzione non configurate

Si è tentato, sia negli scorsi paragrafi che nel precedente capitolo di evidenziare che una delle mancanze della procedura di boot di default dell'S1 Core sia la non configurazione dei TLB e solo parziale delle MMU. Non si propone qui una soluzione a tali problematiche, in quanto ancora in corso di studio. Si propone invece una simulazione che richiami la traduzione dell'indirizzo da *Real Address* (RA) a *Phisical Address*, e venga effettuata una richiesta di STORE sul bus in modo da rendere esplicita la problematica.

Nella simulazione proposta, dovrà essere evitata la generazione di errori dovuti alle altre cause presentate in precedenza nel capitolo, dunque:

- vi saranno le istruzioni riguardanti i registri di stato per la gestione delle finestre, vedi §6.2.2;
- si dovrà poter eseguire la SAVE e avere lo *STACK POINTER* che punti ad un indirizzo di RAM, per esempio inserendo l'istruzione `stx 0x40209, %g4,%sp`;

In aggiunta sono 3 ulteriori impostazioni da configurare:

- il JMP verso la RAM deve essere attivo, bisogna fare il fetch da RAM, perchè durante l'esecuzione di istruzioni da ROM vi sono restrizioni sull'inserimento in cache di dati e isruzioni;

- bisogna uscire dal RED state, impostando a 0 il quinto bit di HPSTATE: in Red mode, si hanno restrizioni sulle traduzioni degli indirizzi. Basterà aggiungere subito prima della JMP l'istruzione

  ```
  wrhpr %g0, 0x804, %hpstate
  ```

  Si mantengono i privilegi hyperpriviledge per l'esecuzione perchè gli indirizzi riguardanti le istruzioni in priviledge mode verrebbero tradotti e non si riuscirebbe a fare i fetch delle istruzioni.

- si ha necessità di abilitare caches L1 e MMU sia dati che istruzioni, anche se non inizializzate: si sta ricercando una condizione di errore. Dunque il LSU Control Register dovrà essere a FF_{16}

  ```
  mov 0xF, %l1
  stxa %l1, [%g0] (0x45)
  ```

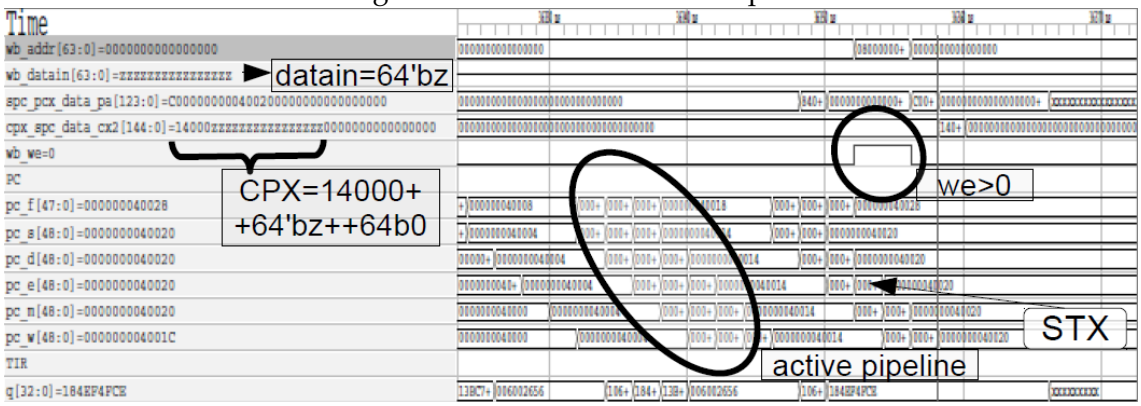
La simulazione, questa volta, si blocca alla prima store, alla quarta istruzione di RAM, ma con comportamento differente dalle precedenti simulazioni: molti segnali assumono valore indeterminato e si hanno anche effetti osservabili sul bus perchè il pacchetto PCX viene generato. Un'altro effetto osservabile è che, avendo introdotto la possibilità di usare le caches, la pipeline non si blocca più attendendo che ogni istruzione sia completamente eseguita.

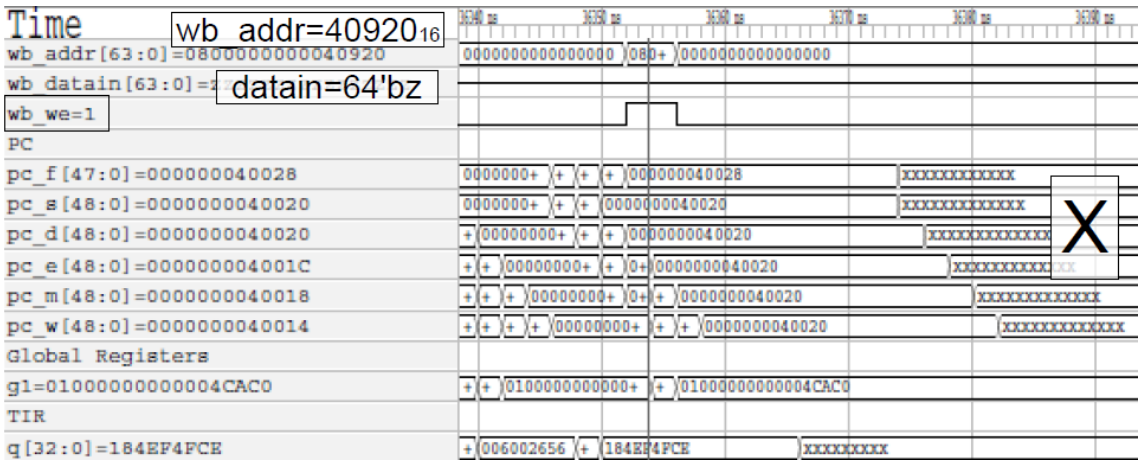
Come evidenziato in figura, è vero che viene generato un pacchetto, ma i dati di ritorno dalla RAM al core sono a *'z'* (alta impedenza).

A conferma dei dati visivi, viene riportato di seguito il contenuto del file di log ($S1_ROOT/run/sim/icarus/sim.log) riguardante le ultime richieste sul bus.

```
INFO: SPC2WBM: *** NEW REQUEST FROM SPARC CORE ***
INFO: SPC2WBM: Request to RAM Bank 0
INFO: SPC2WBM: Request is not atomic
INFO: SPC2WBM: Valid bit is 1
```

Figura 6.13: Simulazione: STORE-fail con caches e MMU attive. Con la cache istruzioni attiva, la pipeline viene eseguita senza attendere la conclusione di ogni istruzione ed i PREFETCH vanno a buon fine. Si può notare come il segnale di *write-enable* si alzi alla richiesta di STORE. Le due figure sono scale diverse della parte finale della simulazione.





```
INFO: SPC2WBM: Request of Type STORE_RQ
INFO: SPC2WBM: Non-Cacheable bit is 0
INFO: SPC2WBM: CPU-ID is 0
INFO: SPC2WBM: Thread is 0
INFO: SPC2WBM: Invalidate All is 0
INFO: SPC2WBM: Replaced L1 Way is 0
INFO: SPC2WBM: Request size is 8 Bytes
INFO: SPC2WBM: Address is 0000040920
INFO: SPC2WBM: Data is 000000000004cac0
INFO: MEMH testbench.ram_harness:
        W@36354000 ns, AD=0800000000040920 SEL=ff
                DAT=000000000004cac0
INFO: MEMH testbench.ram_harness:
        W@36356000 ns, AD=0800000000040920 SEL=ff
                DAT=000000000004cac0
INFO: WBM2SPC: *** RETURN PACKET TO SPARC CORE ***
INFO: WBM2SPC: Valid bit is 1
INFO: WBM2SPC: Return Packet of Type ST_ACK
INFO: WBM2SPC: L2 Miss is 0
INFO: WBM2SPC: Error is 0
INFO: WBM2SPC: Non-Cacheable bit is 0
INFO: WBM2SPC: Thread is 0
INFO: WBM2SPC: Way Valid is 0
INFO: WBM2SPC: Replaced L2 Way is 0
INFO: WBM2SPC: Fetch for Boot is 0
INFO: WBM2SPC: Atomic LD/ST or 2nd IFill Packet is 0
INFO: WBM2SPC: PFL is 0
INFO: WBM2SPC: Data is zzzzzzzzzzzzzzzz0000000000000000
INFO: TBENCH: Completed Simply RISC S1 Core simulation!
```

Tali risultati, per la configurazione adottata, non possono essere ottenuti con la versione mini (senza cache L1, singolo thread) dell'S1 Core per ovvi motivi. Restano validi invece per la versione SE.

Non verranno date ulteriori informazioni a riguardo, in quanto l'argomento è ancora in fase di studio e di test.

6.7 Registri Priviledge e Hyperpriviledge

Nello scorso e nei precedenti paragrafi del presente capitolo, si sono esposte diverse problematiche tese ad ottenere uno stato coerente dello SPARC core. Durante la fase di test del core, si sono tentate molte strade, prima di arrivare alle considerazioni esposte in questo e nello scorso capitolo. Sono emerse altre incongruenze e carenze nella configurazione del core.

Vi sono altri registri di stato dello SPARC core, talmente importanti che sono loro state dedicate delle istruzioni per la lettura e scrittura. Sono i registri priviledge (*non-register-window priviledge registers*) e hyperpriviledge, leggibili e scrivibili con le istruzioni RDPR, WRPR e RDHPR, WRHPR rispettivamente.

La configurazione di questi ulteriori registri non è stata presa in considerazione perchè, a causa di limitazioni sulla configurazione di trappole e gerarchia di memoria, si è costretti ad eseguire il codice in hyperpriviledge mode (o addirittura in RED state). Questo inibisce gli effetti e le configurazioni di molti registri di stato.

Alcuni di questi, come HPSTATE o HTBA, sono stati già discussi e configurati. Nel presente paragrafo saranno più o meno sommariamente discussi alcuni dei registri priviledge e hyperpriviledge del core, presentandoli anche tramite simulazioni. Vi sono altri registri, cui invece sono dedicate le istruzioni RDASR e WRASR, (tra cui l'ASI ed il CCR) ugualmente importanti ma che verranno tralasciati in quanto cambiano le informazioni contenute "*a runtime*", sono più operativi che di stato.

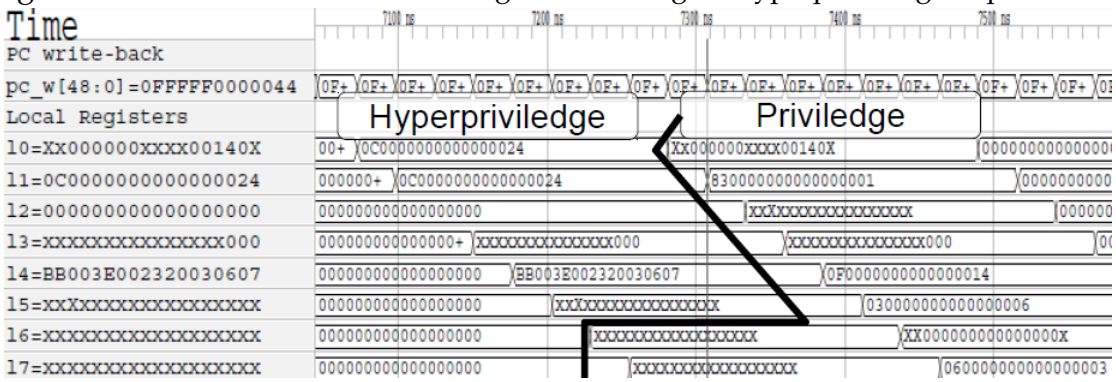
Una trattazione completa riguardo tali registri è riportata nella documentazione dell'UltraSPARC, vedi §3.1.2.4.

6.7.1 Simulazione: valori dopo il POR

Per sapere i valori dei registri priviledge e hyperpriviledge del core, il modo che si è trovato più semplice è, più che istanziarli come segnali in simulazione, farne una lettura tramite le istruzioni loro dedicate e avere in simulazione solo i registri.

Per semplicità, si eviterà di avere dipendenze da altre istruzioni e configurazioni e si useranno le risorse a disposizione subito dopo il POR (come registri *local* e *global*), inserendo le istruzioni di lettura e scrittura, subito dopo le 8 nop iniziali, nel file boot.s. I valori riportati sulla destra sono esadecimali e si assume la

Figura 6.14: Simulazione: valori dei registri Priviledge e Hyperpriviledge dopo il POR.



convenzione che i valori segnati con una 'X' maiuscola non siano tutti e quattro indeterminati.

```
! Lettura                !valore rilevato in simulazione
rdhpr %hpstate , %l0             !!0C0000000000000024
rdhpr %htstate , %l1             !!0C0000000000000024
rdhpr %hintp , %l2               !!000000000000000000
rdhpr %htba , %l3                !!xxxxxxxxxxxxxxx000
rdhpr %hver , %l4                !!BB003E002320030607
rdhpr %hstick_cmpr , %l5         !!xXxxxxxxxxxxxxxxx

rdpr %tpc , %l6                  !!xxxxxxxxxxxxxxxxxx
rdpr %tnpc , %l7                 !!xxxxxxxxxxxxxxxxxx
rdpr %tstate , %l0               !!Xx000000xxxx00140X
rdpr %tt , %l1                   !!8300000000000000001
rdpr %tick , %l2                 !!xxXxxxxxxxxxxxxxx
rdpr %tba , %l3                  !!xxxxxxxxxxxxxxx000
rdpr %pstate , %l4               !!0F0000000000000014
rdpr %tl , %l5                   !!030000000000000006
rdpr %pil , %l6                  !!XX000000000000000x
rdpr %gl , %l7                   !!060000000000000003
```

La simulazione delle istruzioni presentate, è riportata in figura 6.14.

Si nota come molti dei registri siano indeterminati o abbiano valori solo in parte configurati. Il valore valido di alcuni registri dipende dal fatto che alcuni bit vengono letti sempre con lo stesso valore (tipicamente 0), come l'HTBA, mentre

altri perchè sono parzialmente configurati.

6.7.2 PSTATE

Priviledge STATE. Svolge più o meno lo stesso ruolo, per il software priviledge, di HPSTATE per quello hyperpriviledge. I campi di PSTATE sono descritti in tabella.

Tabella 6.2: PSTATE.

Bit	Campi	Valore POR (bin)	Descrizione
12	—	0	—
11:10	—	00	—
9	cle	0	se 1, è assunto little endianess per la generazione di asi impliciti. Può ripercuotersi su TTE.ie e MMU.
8	tle	0	come cle, per 0<TL<maxptl. tle viene copiato su cle quando viene attivata una trappola, solo verso priviledge mode
7:6	mm	00	memory model
5	—	0	—
4	pef	1	enable FPU. Per essere attivata la FPU deve essere attivo anche FPRS.fef
3	am	0	address mask. Consente al software SPARC a 32 bit di essere eseguito correttamente su una macchina SPARC a 64 bit
2	priv	1	se 1, quando HPSTATE.hpriv=0 il software viene eseguito in modalità priviledge.
1	ie	0	se 1, possono essere eseguite *disrupting exception*.
0	—	0	—

PSTATE, governa il comportamento del software in priviledge mode. In generale, è attivo quando HPSTATE.hpriv è nullo. Nel caso dovesse essere eseguito del codice priviledge sull'S1 Core, a meno di non aggiungerla a posteriori, dovrebbe essere disabilitata la FPU, per esempio inserendo l'istruzione

```
wrhpr %g0, 0x4, %pstate
```

Il bus dell'S1 Core lavora in big endian, dunque sembra corretto lasciare PSTATE.cle e PSTATE.tle nulli.

Il valore di PSTATE.mm decide sul comportamento dei buffer nei riguardi di LOAD e STORE: il valore 00_2 implica che si stia adottando il TSO (*Total Store Order*) dunque le STORE risultano sempre essere eseguite nella sequenza richiesta dai programmi in esecuzione. E' la configurazione più restrittiva e tutte le architetture SPARC devono implementarla.

6.7.3 Registri priviledge e hyperpriviledge Trap-related

Di alcuni registri riguardanti le trappole, si è già discusso e non verranno ripresi.

Vi sono due registri, HTSTATE e TSTATE, dedicati a dichiarare il contenuto di HPSTATE e PSTATE per ogni livello di trappola. Il contenuto di questi due registri, quando viene generata una eccezione e si entra in un livello di trappola, verrà sostituito a PSTATE e HPSTATE: decidono con quale livello di privilegio (di default) sarà eseguito il codice di gestione delle trappole, per un determinato TL. Si può notare come tanto HTSTATE quanto TSTATE i registri abbiano, dopo il POR, gli stessi valori di HPSTATE e PSTATE, per quanto TSTATE tenga PSTATE traslato di qualche bit.

TPC e TNPC, salvano invece il contenuto del Program Counter e del Next Program Counter alla generazione di una trappola. PC e NPC salvati, si riferiscono al livello di trappola precedente, i quali, dopo aver gestito la trappola, verranno reinseriti automaticamente se alla fine del codice trappola si lancerà l'istruzione RETURN. La loro funzionalità spiega anche il perchè siano indeterminati in seguito ad un POR, valore che in questo caso si potrebbe considerare accettabile, a patto di non uscire dal gestore di reset con una RETURN. Vi sono dunque HVER.maxtl (vedi §7.5.3) istanze di TPC e TNPC.

TT, detiene il codice corrispondente al tipo di trappola attualmente in esecuzione. Anche i reset hanno un tipo di trappola associato, e all'inizio della simulazione TT incorpora il valore associato al POR: TT=1.

TL, tiene invece conto del *Trap Level*. Vi sono HVER.maxtl livelli di trappola, e dopo un POR, TL assume il suo valore massimo: TL=6. Il codice hyperpriviledge, in RED state, può eseguire con TL anche maggiore di HVER.maxtl, per quanto lo stato del core in quel momento sia incoerente. Un valore nullo di TL implica che non siano in esecuzione codici-trappola, si tratta dello stato normale di esecuzione del codice. I livelli di trappola sono divisi in due categorie: quelli con esecuzione priviledge indicati da maxptl, e quelli ad esecuzione hyperpriviledge indicati da maxtl>maxptl. Spesso nelle implementazioni si assume che maxptl=Hver.gl, che corrisponde al numero massimo di global register implementati meno 1.

GL, è il registro che gestisce i HVER.maxgl set di *global register*, con funzione equivalente al CWP per i *windowed register*. Dopo un POR assume valore massimo: nell'OpenSPARC maxgl=3, e sono gestiti 4 set di registri globali.

6.7.4 Simulazione in priviledge mode

La presente simulazione riprende quasi tutti i concetti espressi nel presente e nello scorso capitolo.

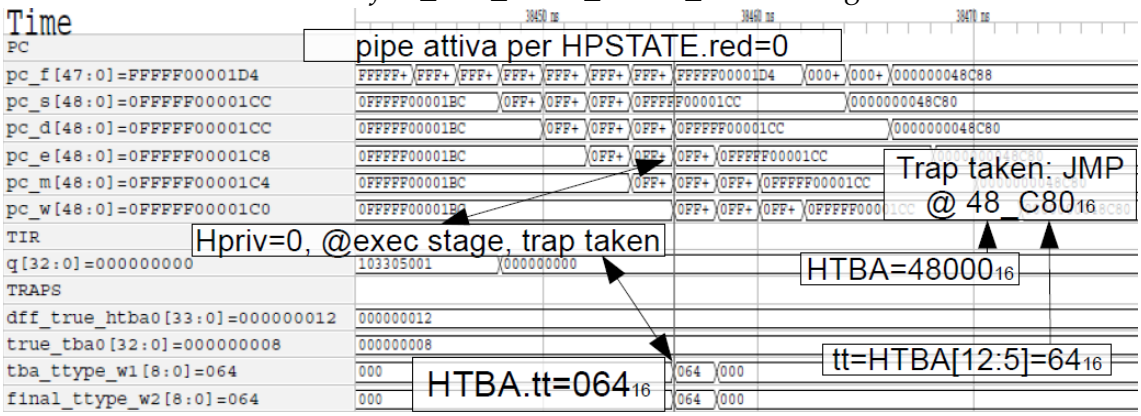
Come riprova che senza configurare i meccanismi di traduzione degli indirizzi non è eseguibile codice priviledge, si può impostare una simulazione che tenti di eseguire il salto da ROM a RAM in priviledge mode. Verrà generato un errore, e per evitare che vengano generate eccezioni differenti da quella voluta (ci si aspetta in sostanza un errore di traduzione). Per cogliere l'errore, bisogna che la procedura di boot contenga le configurazioni corrette dei *windowed register*, e in questo caso si è deciso di attivare la gerarchia di memoria: LSU Control register = F_{16}.

Mentre il tipo di trappola generato sarà sempre lo stesso, l'indirizzo cui viene fatto il salto per eseguire il codice-trappola è riferito alle impostazioni di TBA e HTBA. Si imposti il HTBA a 48000_{16} ed il TBA a 40000_{16} per mostrare che verrà letto l'HTBA anche se si è passati in priviledge mode. Come nelle simulazioni precedenti, si è impostato livello hyperpriviledge in TT[TL=1] e si imposta il TT=0. Inoltre, bisogna togliere i privilegi di hyperuser allo strand, annullando HPSTATE.hpriv. per entrare in priviledge mode: come si vedrà, appena ca-

dranno i privilegi hyperpriviledge, viene richiesta la traduzione delle istruzioni per farne il fetch, ma la MMU non ne sarà in grado e genererà l'eccezione *fast_data_access_MMU_miss* con TT=064_{16} .

```
! @file: boot.s. ultime istruzioni.
!... Configurare CWP e windowed registers ...
! configurazione Trap Type & TSTATE
rdhpr %hpstate, %g3
wrpr  1, %tl            ! current trap level = 1
mov   0x0, %l1  ! l1 = 0
wrhpr %g3, 0x20, %htstate   ! reset HTSTATE reg that store
                ! hyperpriviliged state after traps
wrpr  0, %tl            ! current trap level = 0 (No Trap)
mov   0x0, %o0  ! please don't delete since
                ! used in customized IMMU miss trap

! exit Red STATE
wrhpr %g3, 0x20, %hpstate

! Configurazione TBA e HTBA:
! SET TBA: TBA[63:15] = 0x13
sethi %hi(0x40000), %g4
wrpr  %g4, %g0, %tba    !bits 63-15 select Priv trap vector
sethi %hi(0x48000), %g4
wrhpr %g4, %g0, %htba  !bits 63-15 select Hpriv trap vector
        nop     ! @0x1bc

! test on priviledge software run
wrpr %g0, 0x4, %pstate  !@0x1C0
wrhpr %g0, 0x800, %hpstate      !@0x1C4
        nop
        nop
        nop
jmp %g2         ! Jump to 0x40000
        nop
```

Figura 6.15: Simulazione: Condizione di errore dovuta alla tentata esecuzione di codice priviledge. Appena si perdono i privilegi di hyperuser, si fallisce il fetch della prossima istruzione. Una eccezione di *fast_data_access_MMU_miss* viene generata.



In figura, è riportata la simulazione delle istruzioni che generano la trappola. Non ci si faccia trarre in inganno dal fatto che il segnale TBA_ttype detenga il valore corretto della trappola: una volta intrapresa una trappola, sia che venga letto il TBA che l'HTBA, l'offset di entrambi i registri viene aggiornato con il valore di TT[TL]. Per questo, si è deciso di evidenziare che il registro effettivamente letto fosse l'HTBA impostando il TBA all'inizio della RAM perchè non avessero indirizzi sovrapposti.

Capitolo 7

OPENSPARC T1 – CODENAME NIAGARA

Nel presente capitolo verranno date delle informazioni generali e specifiche sull'OpenSPARC T1. Senza avere la pretesa di esaustività, vengono presentate unità funzionali, registri e procedure, che sono di ausilio alla discussione della presente tesi ed alla comprensione dei precedenti capitoli. La documentazione riguardante sia l'OpenSPARC T1 che l'architettura UltraSPARC in generale, sono più che esaustive e non si ha qui la pretesa di migliorarle. Si intende invece mettere insieme alcuni concetti fondanti dell'architettura dell'OpenSPARC T1, in modo che, in poche pagine, si riesca ad avere informazioni generali su aspetti non trattati nei precedenti capitoli e più dettagliate su quelli affrontati.

Il capitolo è concettualmente diviso in due sezioni, una generale e l'altra più tecnica dove si forniscono informazioni sul funzionamento di registri e di strutture.

Nella sezione generale, si cercherà di fornire in poche frasi i concetti riguardanti il funzionamento e l'utilizzo di moduli e unità, soffermandosi su alcune particolarità che distinguono le architetture SPARC da altre, e che sono state di particolare interesse per la discussione sui test.

In una seconda parte del capitolo, si riporta il ruolo dei singoli bit di alcuni registri interni dello SPARC core: si è prestata particolare attenzione a rimarcare concetti generali e funzionamenti particolari che sono di ausilio alla comprensione delle simulazioni. Non si trovano, in linea di massima, informazioni sulle possibili eccezioni generate da un uso improprio di registri nè sulle dipendenze

dall'architettura. Si è invece cercato di evidenziare se un registro o ASI appartenga all'architettura UltraSPARC generale piuttosto che a quella dell'OpenSPARC T1.

In entrambe le trattazioni, la scelta degli argomenti è avvenuta anche in base alle difficoltà che sono state incontrate in fase di studio e di test di alcune dinamiche.

7.1 Descrizione Funzionale dell' OpenSPARC T1

Il processore Opensparc T1 è una implementazione dell'architettura UltraSPARC T1 della Sun Microsystem, nome in codice Niagara. E' composto da diversi macroblocchi, indicati nella figura7.1:

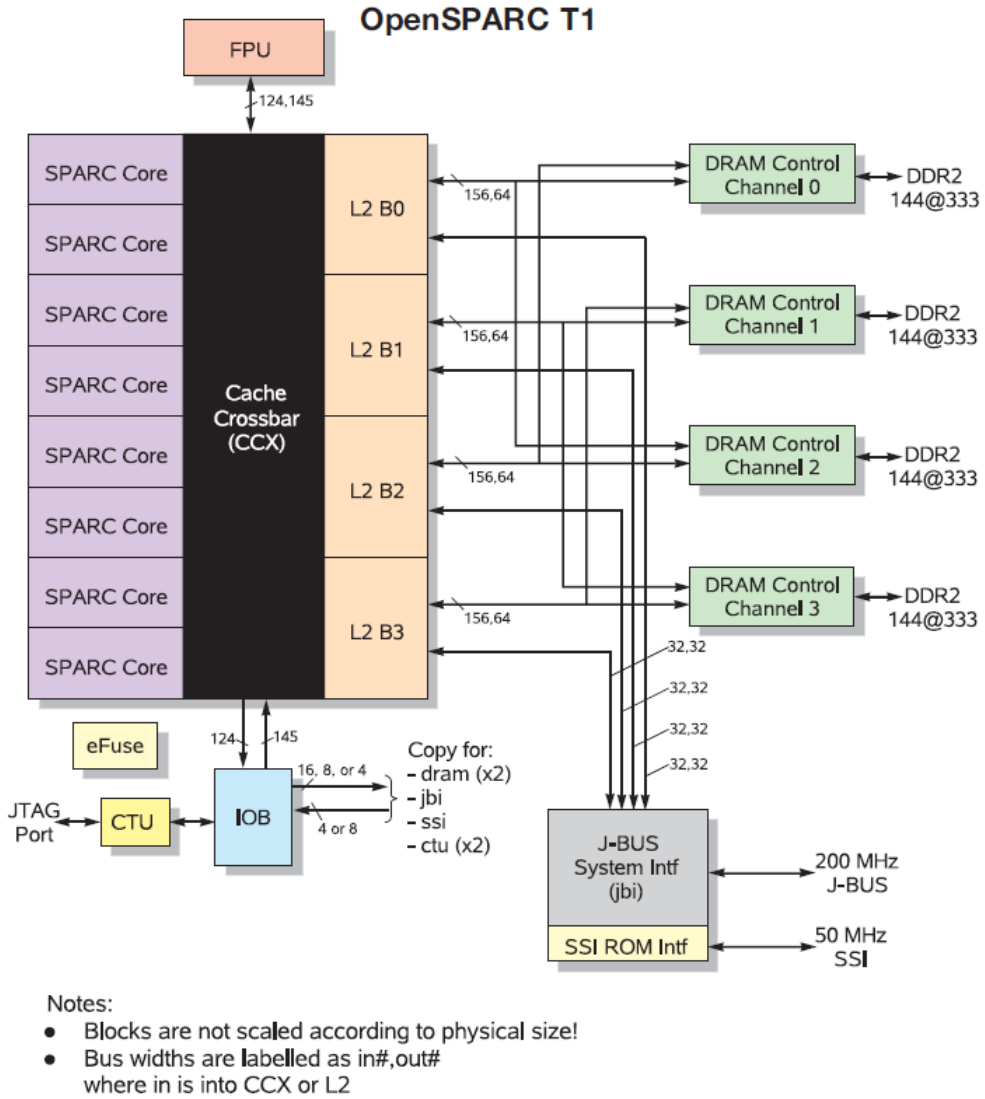
SPARC physical core: ogni CORE degli 8 presenti implementa l'architettura UltraSPARC. Sono gestiti quattro *vitrual-processors*[1] (detti anche *hardware strands* o *threads*) per CORE. Ogni thread ha dedicati sia l'Integer Register File (con 8 *window*), sia la maggior parte degli ASI, ASR e i registri priviledged. I thread condividono invece la gerarchia di memoria: instruction e data caches e i TLBs. Il software può essere eseguito in tre modalità: *Hyperpriviledge* (HW management), *Priviledge* (Operative Systems), *User* (applicazioni). Supporta il CMT (Chip Level Multithreading): le 8 CPU possono gestire fino a 32 thread contemporanei.
La pipeline consiste in 6 stages: Fetch, Switch, Decode, Execute, Memory, Writeback. Nel S-stage viene scelto il thread che andrà in esecuzione.

Floating Point Unit: gli 8 COREs dell' OpenSPARC T1 condividono la stessa FPU, alla quale vengono inviati i dati come se fosse una unità esterna al processore: viene effettuata un' operazione di I/O. Contiene 32 registri

[1] Un *virtual processor* è l'insieme di risorse (registri, prevalentemente) che vengono visti in un dato istante da un thread in esecuzione. Due thread differenti, per esempio, avranno differenti valori di Program Counter, in un dato istante. Vi sono delle risorse che tutti i 32 thread condividono, come la cache L2 e il modulo di I/O. Ogni processore esegue fino a 4 thread, dunque le cache L1 e i TLB sono condivisi solo tra i 4 thread dello stesso SPARC core. Le risorse non condivise tra tutti i 32 thread sono replicate 32mod[n° di thread che le condividono] volte: vi sono 32 thread-Program-Counter e 8 cache L1.

Figura 7.1: Diagramma a blocchi dell' OpenSPARC T1.
Fonte figura: documentazione OpenSPARC, vedi §3.1.1.

Diagramma a blocchi dell' OpenSPARC



da 32 bit (*word*), 32 da 64 bit (*doubleword*), 16 da 128 bit (*quad-word*). Può non essere presente, o essere disabilitata.

L2 Cache: ha 4 banchi identici di memoria (4 ways) identificati dai bit 7:6 del physical-address. Ogni banco è di 3 Mbyte, 12 way set associativa con politica di sostituzione pseudo-LRU (con *used-bit*); ha linee da 64 byte.

DRAM controller: ha 4 (o 2) ways e ogni banco di L2 comunica esclusivamente con uno e uno solo di DRAM. Tutti i banchi di DRAM devono essere uguali. OpenSPARC T1 usa DDR2 DIMM. Ogni banco/porta DRAM è grande 2 DIMMs (128 bit word + 16 bit Error Correction Code).

I/O Bridge (IOB): decodifica gli indirizzi nelle transazioni di dati verso l'esterno e indirizza il dato o al J-Bus o al SSI. Mantiene le informazioni necessarie agli interrupt esterni.

J-Bus Interface (JBI): connette l' OpenSPARC T1 con il sottosistema di I/O. E' un bus dati e indirizzi a 200 MHz, che comunica con parole da 128 bit.

SSI ROM Interface: gestisce la comunicazione tra FPGA e la BOOT PROM, tramite interfaccia seriale a 50 Mbit/sec.

Clock and Test Unit (CTU): contiene l'unità di generazione del clock, dei reset e la circuiteria per la comunicazione tramite JTAG. Inoltre ha l'unità CREG che consente di leggere dall' esterno, tramite JTAG, non solo le transazioni che avvengono nell' I/O unit, ma anche molti dei registri interni del processore indirizzati da ASI e qualsiasi locazione di memoria, mentre l'OpenSPARC T1 è in esecuzione.

EFuse: (electronic fuse) contiene informazioni sulla configurazione del processore, sul costruttore, sul numero di strand avviabili. Questi registri sono fusi elettronicamente nel design.

7.2 SPARC Core

Un processore OpenSPARC T1 ha 8 SPARC V9 cores, e ogni core ha diverse unità funzionali. Una breve descrizione delle caratteristiche di ogni unità, verrà esposta di seguito.

7.2.1 Instruction Fetch Unit

7.2.1.1 SPARC Core Pipeline

La pipeline è composta da 6 stages:

- Fetch — F-stage
- Thread Selection — S-stage
- Decode — D-stage
- Execute — E-stage
- Memory — M-stage
- Writeback — W-stage

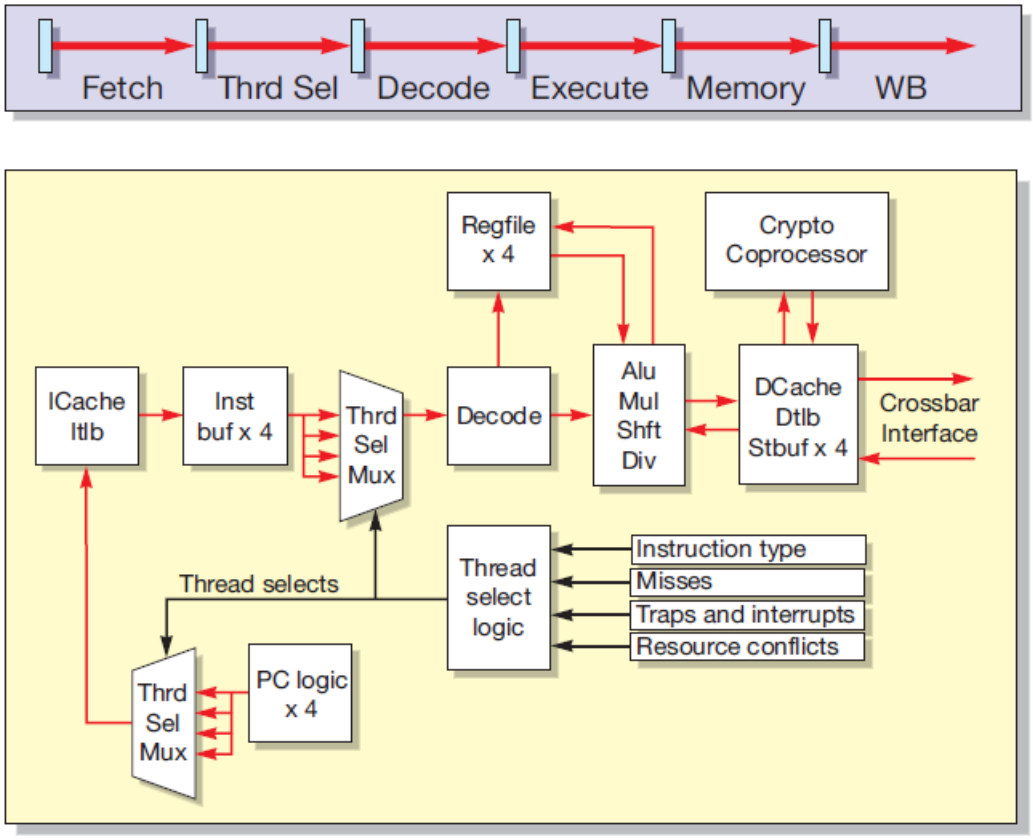
Nello stage di Fetch avviene l'accesso alla I-cache e al ITLB. Nell' S-stage viene scelto il thread da eseguire. La decodifica dell'istruzione e l'accesso all'Integer Register File (IRF) avviene durante il Decode stage. La valutazione delle condizioni di salto (branch evaluation) e la valutazione delle istruzioni aritmetiche dell'ISA[2] avviene nel passo di esecuzione. L'accesso in memoria e il writeback competono rispettivamente agli stage Memory e Writeback.

7.2.1.2 Instruction Fetch

Program Counter (PC) e *Thread Instruction Register* (TIR) sono i registri che detengono l'indirizzo della istruzione attualmente in esecuzione sul processore. *Next Program Counter* e *Thread Next Instruction Register* (NIR), tengono invece l'indirizzo della prossima istruzione. In realtà esistono 6 program counter, uno per ogni stadio della *pipeline*. Ogni istruzione attraversa in sequenza tutti gli stadi della pipeline: una volta che ha eseguito uno stadio, passa al successivo e nello stadio precedente viene eseguita l'istruzione successiva. Questo discorso vale per il fetch da RAM di istruzioni *cacheable*, ovvero la maggior parte. Vi sono casi particolari come il fetch da boot ROM o istruzioni di LOAD e STORE su registri di stato del

[2] *ISA* (Instruction Set Architecture) è l'insieme delle istruzioni eseguibili sul processore. Ogni core dell'OpenSPARC implementa l'ISA SPARC V9. Il documento di riferimento è UA2005-current-draft-HP-EXT.pdf per il V9 e UST1-UASuppl-current-draft-HP-EXT.pdf per le configurazioni Architecture-Dependent adottate nell'OpenSPARC T1.

Figura 7.2: Pipeline dell'OpenSPARC T1.
Fonte figura: documentazione OpenSPARC, vedi §3.1.1.



processore che bloccano la pipeline in modo da essere l'unica istruzione eseguita sul core: una misura di sicurezza per conservare la consistenza dello stato della CPU. Altro caso particolare è il *prefetch* di una istruzione: senza sapere quale effettivamente sarà la istruzione successiva, viene comunque richiesta in cache la presenza della istruzione dell'indirizzo successivo, in quanto è molto probabile che sia quella la prossima di cui fare il fetch.

Esistono 4 Thread Instruction Register, uno per ogni thread.

7.2.1.3 Level 1 Instruction Cache

La cache istruzioni è indirizzata fisicamente come 4-way set-associativa, ha 16 Kbyte di memoria. Ogni linea di cache è da 32 byte, corrispondenti a 8 istruzioni. Ha una singola porta dati e la I-cache viene riempita con accessi da 16 byte: servono due accessi per riempire una linea di cache, e viene garantita l'atomicità

dell'operazione di riempimento (I-cache *fill*). I dati inseriti in I-cache includono 32 bit di istruzione, 1 bit di parità e 1 bit di pre-decodifica. La memoria dei campi TAG è a singola porta. L'array usato per il salvataggio dei valid-bit è a sè stante e ha due porte, una in lettura ed una in scrittura. La politica di rimpiazzo delle linee di cache è pseudo-random. Gli accessi in lettura hanno maggiore priorità degli accessi in scrittura.

La I-cache viene riempita tramite CPX, pacchetti che arrivano dalla cache L2 attraverso la LSU. Vice versa, in seguito ad un I-cache miss implica la generazione di una apposita richiesta al bus, tramite un pacchetto PCX I-FILL. Tutti i miss di istruzioni richiesti dai 4 thread vengono gestiti dal MIL (*Missed Instruction List*).

7.2.1.4 Instruction Table Lookaside Buffer

Il I-TLB è responsabile della traduzione degli indirizzi e della comparazione dei TAG. E' sempre attivo per codice non-hyperpriviledge e sempre disattivato durante l'esecuzione di codice hyperpriviledge. L'I-TLB contiene 64 linee e supporta pagine da 8 Kbyte, 64 Kbyte, 4 Mbyte e 256 Mbyte. L'algoritmo di rimpiazzamento è *pseudo-LRU* (Last Recently Used), ed ha un efficiente meccanismo di *autodemap* che si preoccupa che non sia salvata più volte la traduzione della medesima pagina.

La *feature* dell'*autodemap* dei TLB è uno dei vanti delle architetture UltraSPARC.

Le *entry* dei TLB sono gestiti via software, prevalentemente priviledge (il SO), anche se non ne è preclusa la configurazione al codice hyperpriviledge. E' prevista anche una struttura aggiuntiva che fa da cache per le *entry* dei TLB, chiamata TSB (Translation Storage Buffer), anch'essa gestita via software, il cui utilizzo è opzionale.

7.2.1.5 Thread

Ogni SPARC core gestisce via hardware 4 thread. Alla selezione del thread è dedicato uno stage della pipe, in modo che ad ogni colpo di "SPARC core clock" possa essere selezionato un thread da eseguire.

Un thread può entrare nello stato di wait per diverse cause:

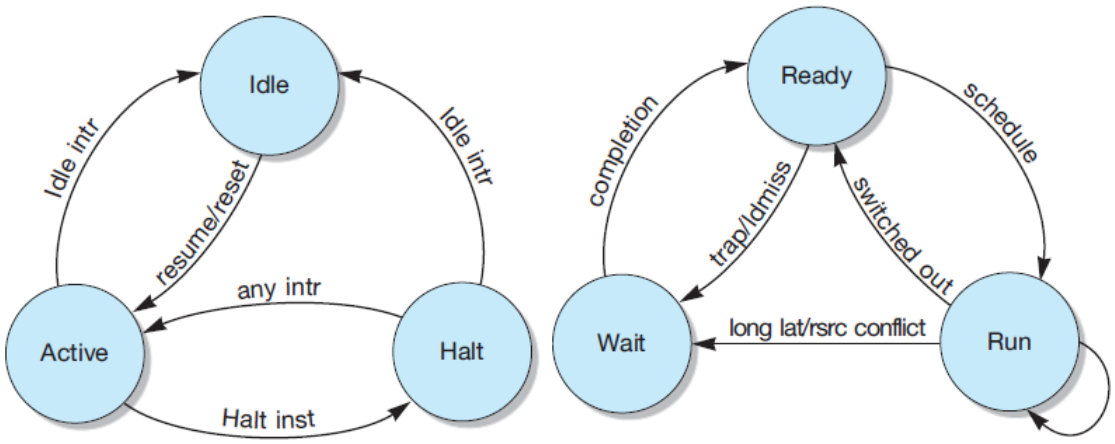
- ha richiesto l'esecuzione di una *long latency operation*, come LOAD, STORE, MUL, BRANCH e così via;

- la pipeline è in stallo (*stall*) a causa di una *long latency operation*, un miss in cache, il processo sta eseguendo una trappola, c'è un conflitto di risorse.

E' presente anche un meccanismo di *rollback*, che consente in modo efficiente di ripristinare uno stato precedente della pipe (massimo 6 stadi) nel caso una *esecuzione speculativa* non vada a buon fine. L'*esecuzione speculativa* si basa su una previsione di caso migliore, e consente di eseguire un thread anche se di norma, per l'operazione richiesta, non potrebbe prendere possesso della pipeline; nel caso la previsione non vada a buon fine viene fatto un *rollback*. Un esempio renderà il concetto più chiaro. Un thread ha richiesto una LOAD da RAM e, essendo questa una *long latency operation*, dovrebbe entrare in wait; invece viene eseguito comunque, sperando di trovare il dato già nella cache dati; se il dato è nella D-cache la previsione è andata a buon fine e si è risparmiato tempo, altrimenti si torna indietro per tutti gli stage eseguiti dal processo.

Le due immagini in figura 7.3, descrivono una i tre macrostati idle (stato iniziale), halt (bloccato) e active (tenta di eseguire codice) ed i tre stati in cui si può trovare un thread quando attivo: wait (in attesa di un risultato), ready (pronto, ma un altro thread è in esecuzione), run (sta eseguendo, sfrutta il processore).

Figura 7.3: Stati dei Thread a sinistra, lo stato di esecuzione a destra.
Fonte figura: documentazione OpenSPARC, vedi §3.1.1.



7.2.2 LSU

La *Load Store Unit* (LSU) processa le istruzioni con opcode (*operation code*) riguardante la memoria, come vari tipi di LOAD e STORE, istruzioni CAS,SWAP, FLUSH,

PREFETCH e MEMBAR. La LSU è connessa a tutte le unità funzionali del core e gestisce le comunicazioni tra SPARC core e *crossbar* (CCX). Tramite la CCX il core interagisce sia con la memoria che con il sistema di I/O tramite il trasferimento di pacchetti.

L'architettura parallela con cui la LSU è implementata, le consente di gestire contemporaneamente 4 load, 4 store, un fetch, una operazione floating point, una operazione di stream, un interrupt e l'invio di un pacchetto. Riceve dati da 13 unità. Durante un ciclo di SPARC pipe, la LSU compie operazioni su differenti stage:

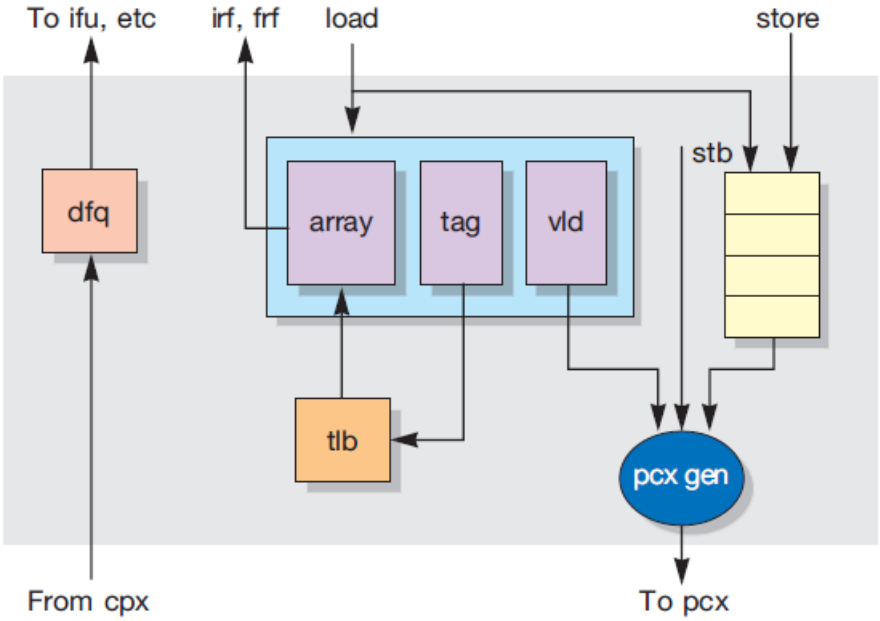
- *execution stage*: accede ai set-up sia della D-cache che al D-TLB;
- *memory stage*: accede alla D-cache, ai TAG e al D-TLB;
- *writeback stage*: accede allo *Store Buffer*, rivela se è stata richiamata una trappola, effettua l'esecuzione del *bypass* dei dati;
- *writeback-2 stage*: genera richieste PCX al bus ed effettua il writeback verso la D-cache.

Gestisce l'ordinamento dei dati (*Memory Model*), sia a livello locale del singolo core che a livello globale. Il memory model in uso nella CPU ad un dato istante è vincolato al valore di HPSTATE.mm. Tutte le architetture UltraSPARC devono implementare almeno il TSO (*Total Store Order*). In figura 7.4, vengono mostrate le connessioni di alcune unità tramite la LSU e la gestione di alcune richieste.

La LSU gestisce inoltre l'ordine dei pacchetti in uscita ed in arrivo al core. Ha due strutture per gestire l'ordinamento dei dati, una per le LOAD (Load Miss Queue) e una per le STORE (Store Buffer).

La LSU gestisce la DFQ (*Data Fill Queue*), la quale è un buffer che ordina i dati in ingresso ed in uscita da e verso gli indirizzi di I/O. Tramite una coda FIFO (*ASI Queue*), la LSU gestisce LOAD e STORE verso tutti gli ASI interni al core, anche non appartenenti alla LSU: ogni thread può generare al più una LOAD/STORE alternate (contemporanea). Un ulteriore buffer, la *Bypass Queue*, ordina le operazioni di lettura (*read*) non provvenienti dalla cache L2 e che vengono scritte in modo asincrono nell'*Integer Register File* (IRF); tra queste, sono gestite le letture full-RAW (*Read after Write*) direttamente dallo Store Buffer (dati in attesa di essere scritti in memoria) e LDXA da ASI di registri interni.

Figura 7.4: LSU: gestione dei dati.Fonte figura: documentazione OpenSPARC, vedi §3.1.1.



7.2.2.1 Level 1 Data Cache

La cache dati è indirizzata fisicamente come 4-way set-associativa, ha 8 Kbyte di memoria. Ha una singola porta per lettura e scrittura (RW port) per dati e tag e, come la I-cache, ha un array di *valid bit* a sè stante. Le LOAD sono *allocating* (vengono comunque scritte in cache), mentre le STORE sono *non-allocating* e seguono la politica *write-through* (se il dato è presente in cache, viene aggiornato, altrimenti viene aggiornato solo in memoria).

Ogni linea di D-cache è sempre presente anche nella cache L2.

La D-cache non contiene mai dati presenti nella I-cache.

La D-cache è protetta da codici di parità e, se si verifica una discrepanza nella parità, il dato viene ri-caricato dalla cache L2 e viene invalidato se l'errore si verifica nella cache L2.

7.2.2.2 Data Translation Lookaside Buffer

Il D-TLB funziona da cache per le ultime 64 TTE (Translation Table Entry), tenute in un array *fully-associative*. Il D-TLB è condiviso dai 4 thread, e le TTE non sono mai ridondanti. Fornisce supporto per le seguenti traduzioni di indirizzi da 32 bit:

- VA$\longrightarrow$ PA (traduzione da *Virtual Address* (VA) a *Phisical Address* (RA))
- VA = PA (*bypass* da indirizzo virtuale a fisico, per operazioni in *hypervisor mode*)
- RA$\longrightarrow$PA (traduzione da *Real Address* (RA) a *Physical Address* (PA), è effettuato il *bypass* della traduzione per operazioni in *supervisor mode*)

7.2.3 EXU

La *Execution Unit* (EXU) è l'unità di calcolo intero dello SPARC core. Ha 4 unità:

- la ALU, l'unità aritmetico-logica;
- lo SHIFT, che si occupa di traslare i bit di un dato, tutti contemporaneamente;
- la IMUL, l'unità per moltiplicazioni di interi, accessibile tramite apposite istruzioni, categorizzate come *long latency operation*;
- la IDIV, l'unità per divisione di interi, accessibile tramite apposite istruzioni, categorizzate come *long latency operation*.

7.2.4 Floating Point Frontend Unit

La FFU gestisce le operazioni floating point e le operazioni grafiche da inviare alla FPU (Floating Point Unit), se presente. La FPU è infatti esterna sia al core che all'intero processore, ed è condivisa tra gli 8 core dell'OpenSPARC (dell'UltraSPARC, in generale). Le operazioni floating point devono essere eseguite pertanto con accessi allo spazio di I/O, per loro natura lenti. Questa scelta è giustificata dal fatto che statisticamente le operazioni floating point sono meno dell'1% del totale.

7.2.5 Memory Management Unit

La Memory Management Unit (MMU) mantiene il contenuto dei TLB: dell'I-TLB, che risiede nella IFU, e del D-TLB, appartenente alla LSU.

L'OpenSPARC T1 fornisce supporto hardware per la virtualizzazione, e molte immagini o istanze del sistema operativo possono coesistere in esecuzione su un

Tabella 7.1: Layer di virtualizzazione dell'OpenSPARC T1.
Fonte figura: documentazione OpenSPARC, vedi §3.1.1.

Gerarchia		**Privilegi di esecuzione del codice**
Applicazioni		User mode
Istanza OS1	Istanza OS2	Priviledge mode
Hypervisor Layer		Hyperpriviledge mode
OpenSPARC T1 hardware Layer		—

microprocessore che gestisce il *multi-threading* hardware (*Chip MultiThreading*). L' *hypervisor* (HV) *layer* virtualizza la sottostante CPU (Central Processing Unit). Le istanze di immagini del SO formano le partizioni della sottostante *virtual machine*. L' hypervisor migliora la portabilità dei sistemi operativi e si assicura che condizioni di errore in un dominio non influenzino le altre partizioni. Sono possibili fino a 8 partizioni: ad ognuna è associato un codice univoco di 3 bit, il *partition ID*.

L'hypervisor usa Phisical Address (PA), il supervisor usa Real Address (RA), differenti astrazioni dei PA, mentre le applicazioni usano Virtual Address (VA) per accedere alla memoria.

7.2.6 Trap Logic Unit

Una trappola (*trap*) è un trasferimento di controllo a software che esegue codice con privilegi più elevati di quello eseguito correntemente.

La Trap Logic Unit (TLU) è l'unità dedicata alla gestione delle eccezioni e delle interruzioni. Alla generazione di una eccezione o all'invio di un pacchetto di interrupt al core, la normale esecuzione del programma può essere interrotta per gestire una situazione di errore, una richiesta di reset e così via. Esistono tre tipi di trappola: *precise*, *deferred* e *disrupting*. Ognuna di queste tipologie si comporta diversamente nei confronti del programma che sta venendo eseguito: ad esempio, quando una istruzione genera una eccezione gestita con *precise trap*, prima di eseguire il codice di gestione viene atteso che tutte le istruzioni precedenti abbiano avuto effetto.

L'OpenSPARC T1 prevede maxtl livelli di trappola: possono essere gestite maxtl eccezioni contemporanee, comprese le eccezioni generate mentre si sta ge-

stendo una precedente eccezione. Il livello di esecuzione di trappola attuale è salvato nel registro TL (Trap Level):

- $TL = 0$, indica che sta venendo eseguito normale codice;
- $1 \leq TL < maxptl$, indica che il codice di gestione della trappola è gestito tramite codice Priviledge;
- $1 \leq TL \leq maxtl$, indica che il codice di gestione della trappola è gestito tramite codice Priviledge;

I valori di maxptl e maxtl sono scritti via hardware nel registro HVER (vedi §7.5.3); deve inoltre essere sempre verificato che $0 < maxptl < maxtl$. Nell'OpenSPARC T1, maxptl = 3[3] e maxtl = 6.

Le trappole vengono gestite con una complessa struttura di registri, utili per esempio al salvataggio dello stato corrente del *virtual processor*, per tenere memoria dello stato che il processore deve avere quando mentre l'eccezione deve essere gestita, per sapere qual'è l'indirizzo di memoria dal quale inizia il codice di gestione della trappola. Tali registri sono replicati per ogni livello di trappola ammissibile, dunque maxtl volte.

I codici di gestione delle trappole sono salvati generalmente in memoria, ma è prevista una Trap Table dove sono salvate le prime 8 istruzioni per ogni trappola (32, per le più comuni). Ogni trappola ha un identificativo univoco che, quando è richiesta la sua esecuzione, viene scritto nel registro TT[TL] (Trap Type, relativo al livello di trappola). In memoria, sono presenti i codici di gestione della trappola, a partire da un indirizzo, scritto nel TBA (Trap Base Address) per le trappole priviledge, e nel HTBA per le trappole hyperpriviledge. Il tipo di trappola, fa da offset rispetto all'indirizzo di base.

Una trappola si presenta come un'inaspettata chiamata a procedura. All'ingresso nel codice di trappola, l'hardware compie una serie di passaggi:

1. salvare alcuni registri di stato del *virtual processor* (program counter, CWP, ASI, CCR, PSTATE, TT ecc) in uno stack hardware;
2. entrare in uno stato di esecuzione priviledge con un predeterminato PSTATE o hyperpriviledge con predefiniti PSTATE e HPSTATE;

[3] In una tipica implementazione, maxptl = HVER.maxpgl, dove maxpgl è il numero massimo di set da 8 registri globali.

3. eseguire il codice indicato dal *trap handler* (puntatore) presente nel *trap vector* (vettore trappola);

4. finito di eseguire il codice corrispondente alla trappola, viene lanciata una istruzione di DONE o RETRY, che ripristinano il precedente stato del *virtual processor*.

7.3 Istruzioni dell'OpenSPARC

7.3.1 Esecuzione delle istruzioni

Viene fatto il fetcDiagramma a blocchi dell' OpenSPARC T1.h dell'istruzione situata all'indirizzo di memoria indicato dal PC (Program Counter), dunque eseguita. L'esecuzione di una istruzione avrà come risultato implicito anche l'assegnazione di un nuovo valore al PC e al NPC (Next Program Counter).

Una istruzione può generare una eccezione, che viene gestita dal modulo Trap della CPU. Se l'esecuzione non genera condizioni di errore e non è una istruzione di salto (Control Transfer Instruction), il NPC viene copiato nel PC ed incrementato di 4. Ci sono due tipi di istruzioni CTI:

- *delayed*: il PC assume il valore del NPC e il NPC assume l'indirizzo indicato nell'istruzione;

- *immediate*: il PC assume l'indirizzo indicato nell'istruzione, il NPC assume il valore dell'indirizzo indicato nell'istruzione + 4.

A tutte le istruzioni ed i normali accessi ai dati sono sempre associati 8 bit di Address Space Identifier (ASI) i quali vengono uniti ai 64 bit dell'indirizzo di memoria. Le istruzioni *LOAD alternate* e *STORE alternate* possono associare automaticamente un ASI al loro indirizzo o possono usare il valore corrente del registro ASI.

7.3.2 Address Space Identifier (ASI)

Le architetture UltraSPARC associano un *address space identifier* (ASI) a tutti gli indirizzi di memoria. L'ASI è usato per:

- distinguere tra diversi spazi di indirizzi;

- fornire un identificativo unico per ogni spazio di indirizzi;
- mappare registri di controllo e diagnostica per il virtual processor.

La MMU usa indirizzi virtuali a 64 bit[4] più ASI da 8 bit per generare indirizzi di memoria, di I/O e dei registri interni. A questo spazio di indirizzi fisici (PA) si può avere accesso tramite una mappatura (traduzione) virtual-to-phisical oppure tramite il *bypass* della MMU.

Sono assegnati differenti range di ASI ai tre differenti livelli di protezione, come indicato in tab. 7.2. Assegnazione dell'ASI, per software con differenti priorità.

Tabella 7.2: Assegnazione dell'ASI, per software con differenti priorità.

Privilegi minimi di accesso	**ASI [range]**
Priviledge	00_{16}—$3F_{16}$
Hyperpriviledge	40_{16}—$7F_{16}$
User	80_{16}—FF_{16}

Vige una ulteriore categorizzazione degli ASI, rispetto al modo con cui la MMU traduce gli indirizzi ad essi associati:

- un ASI *Virtual-Translating* (il tipo più comune) fa in modo che l'indirizzo cui è associato venga trattato come *Virtual Address* e la MMU effettuerà una traduzione VA$\longrightarrow$PA;
- un ASI *Non-Translating* fa in modo che l'indirizzo cui è associato venga trattato come un *Phisical Address* e non venga tradotto dalla MMU, sarà usato così com'è. Questa tipologia di ASI è tipicamente usata per l'accesso ad registri interni;

[4]Benchè i VA siano a 64 bit, solo 48 vengono gestiti dall'OpenSPARC T1: i 16 bit [63:48] sono letti come l'estensione di segno del bit 47. Alcuni registri interni (come l'HTBA) permettono la scrittura del full VA a 64 bit e deve essere il software a preoccuparsi che vengano estesi correttamente gli ultimi 16 bit.

- un ASI *Real-Translating* fa in modo che l'indirizzo cui è associato venga trattato come un *Real Address* e la MMU effettuerà la traduzione RA⟶PA. I *Real-Translating* ASI sono tipicamente usati dal software priviledge e hyperpriviledge per accedere direttamente in memoria usando indirizzi *Real* o *Phisical* rispettivamente.

La lista completa degli ASI dell'OpenSPARC T1, compresi gli "*implementation-dependent*" della architettura UltraSPARC, è fornita nella documentazione ufficiale, vedi §3.1.2.5, al capitolo "*Address Space Identifier (ASIs)*".

7.3.3 Alcune istruzioni

7.4 Registri general purpouse, l'IRF

L'Integer Register File (IRF) contiene 640 registri da 64 bit, per un totale di 5 Kbyte di memoria. Ogni registro è protetto da codici di correzione di errore (ECC). Gli accessi in lettura e scrittura sono eseguiti in un ciclo di *SPARC core clock*[5]. Ad ogni thread sono assegnati 160 registri: 128 per le 8 *window* (finestre) e 32 globali, divisi in 4 set con 8 registri per set.

Solo 32 registri della *window* corrente sono visibili al thread in un dato istante[6], suddivisi come in tabella 7.3.

Tabella 7.3: Registri general purpouse visibili in un dato istante.

—	**globals**			**output**			**locals**			**input**		
label name	`g0`	...	`g7`	`o0`	...	`o7`	`l0`	...	`l7`	`i0`	...	`i7`
R name	R[0]	...	R[7]	R[8]	...	R[15]	R[16]	...	R[23]	R[24]	...	R[31]

Ogni volta che viene selezionato un nuovo thread viene eseguito un cambio di *window*.

Nel seguito ci si riferirà al numero di window supportate dall'architettura con *N_REG_WINDOWS* (8, nell'OpenSPARC T1), e al numero di registri globali con

[5]Un ciclo di SPARC core clock è il tempo di esecuzione di uno stage della pipeline, che, per l'OpenSPARC T1, è di 6 colpi di clock fisico.

[6]Si potrebbero definire i 32 registri visibili come "contesto", ma tale termine è usato per discriminare gli spazi di indirizzi dei diversi sistemi operativi da parte del hypervisor (Context Register) e non verrà qui usato.

MAXGL+1 (3+1, nell'OpenSPARC T1)[7].

Di seguito viene illustrato come la scelta dei 32 registri visibili è attuata separatamente per registri globali (8, dei 32 visibili) e windowed register (24, dei 32 visibili).

7.4.1 Global R registers

Nello SPARC core sono implementati *MAXGL*+1 set di 8 registri *global*. Solo un set, R[0]–R[7], è abilitato e accessibile ad ogni istante. Il set di registri globali attualmente abilitato è selezionato dal registro GL, ad accesso priviledge.

Il registro *g0* viene sempre letto come 0 e le scritture non hanno effetti visibili al software.

I set di registri accessibili dal software priviledge sono quelli con GL$\leq$maxpgl, che nell'OpenSPARC T1 vale 2. Il set di registri corrispondente a GL = 3 è invece accessibile solo al software hyperpriviledge.

Nonostante di solito le implementazioni della architettura UltraSPARC prevedano che maxpgl=maxptl[8], la gestione dei registri globali non è in alcun modo legata ai livelli di trappola; inoltre è a sè stante anche rispetto alla gestione dei registri locali e di I/O.

7.4.2 Windowed R Registers

In ogni istante è visibile un set di 24 registri chiamato "*register window*" (finestra). Sono i registri general purpouse R[8]–R[31]. Le finestre, sono *N_REG_WINDOWS*, e la finestra attualmente attiva è indicata dal valore di CWP (Current Window Pointer), il quale conta modulo *N_REG_WINDOWS*.

Per ogni finestra, e dunque per ogni valore di CWP, gli 8 registri R[16]–R[23], i *local*, sono unici per ogni *window*. Ad ogni valore di CWP corrispondono sempre gli stessi registri *local*.

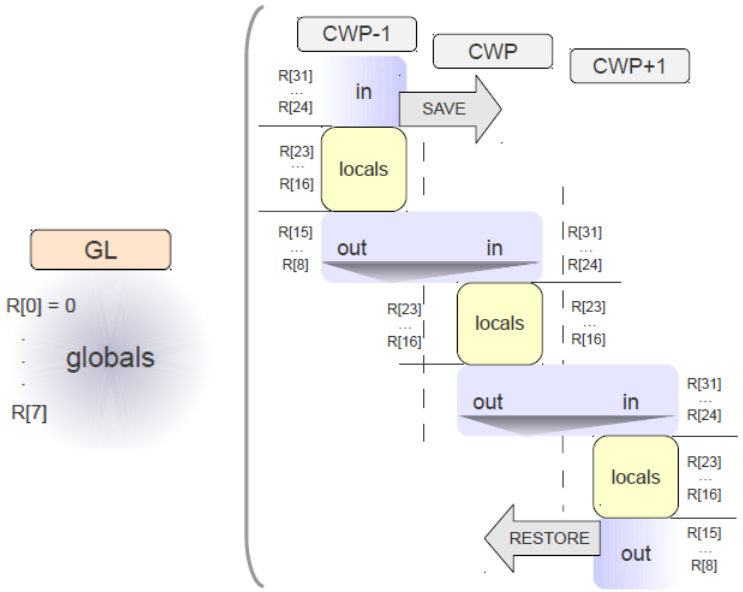
I registri che diventano R[8]–R[15] in una finestra, sono chiamati registri di *out*, mentre i registri R[24]–R[31] sono chiamati *in*. I registri *out* e *in* di una finestra sono in comune con le due finestre adiacenti (valore_attuale_CWP+1 e valore_attuale_CWP - 1, modulo *N_REG_WINDOWS*). Nello specifico, i registri

[7] *N_REG_WINDOWS* è definito dal "massimo valore + 1" assumibile dal registro CWP; *MAXGL* è un campo del registro non modificabile HVER (vedi §7.5.3).

[8] L'OpenSPARC rispetta maxpgl=maxptl.

di *input* della finestra attuale sono i registri di *output* della precedente (non una copia, ma fisicamente gli stessi), mentre i registri di *output* della finestra attuale sono i registri di *input* della successiva. L'immagine 7.5 esplica questo concetto.

Figura 7.5: Tre *register window* adiacenti. Sono evidenziati i valori di CWP e parziale funzionamento delle istruzioni SAVE e RETURN. I registri *global* anche nella rappresentazione grafica sono separati (indipendenti) dai *window register*.



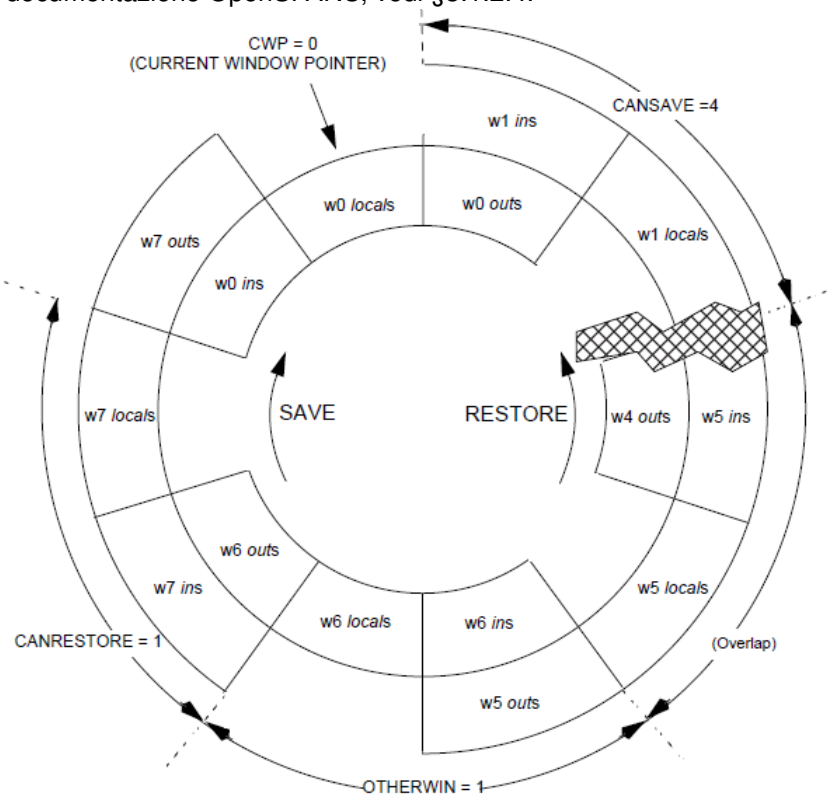
In figura sono rappresentate le istruzioni SAVE e RESTORE. Eseguendo una SAVE, si vorrebbe preservare il contesto corrente (*local* e *output* attuali): si passa dunque alla finestra successiva (con CWP=CWP+1); una SAVE è utile per esempio nelle chiamate a funzione (i registri di input della nuova finestra avranno i parametri passati dalla funzione chiamante). Eseguendo l'istruzione RESTORE si vuole tornare ad un precedente contesto: viene decrementato il CWP, ed i dati della finestra che si sta lasciando non dovrebbero più essere considerati validi[9]; RESTORE è utile nel rientro da funzioni.

Il CWP dichiara quale finestra è attiva, e conta modulo *N_REG_WINDOWS*, dunque i registri di *input* quando CWP=0 sono sovrapposti (*overlap*) ai registri di *output* della finestra CWP=*N_REG_WINDOWS*-1. Al software, appare come se il virtual processor mettesse a disposizione una pila infinita di *window*.

[9]In realtà i dati restano validi perchè i windowed register non sono condivisi tra i thread. Può però cambiare il loro valore a causa della esecuzione di codice con più elevati privilegi.

Una ulteriore rappresentazione dei registri *windowed* è un cerchio su cui sono disposte tutte e *N_REG_WINDOWS* le finestre, con la sovrapposizione l'una con l'altra dei registri di I/O (figura).

Figura 7.6: Rappresentazione grafica dei *windowed register* e dei registri di stato delle window (CWP, CANSAVE, CANRESTORE...).
Fonte figura: documentazione OpenSPARC, vedi §3.1.2.4.



Una finestra è valida quando contiene tutti i registri nulli oppure dati validi del programma in esecuzione: il numero di finestre valide è mantenuto dal registro CLEANWIN. A causa dell'overlap, solo *N_REG_WINDOWS*-1 finestre sono disponibili, una si perde a causa del fatto che dovrebbe contenere dati validi di due contesti diversi: quando sono state effettuate 7 SAVE partendo con CWP=0, si è alla finestra CWP=7 e se venissero cambiati i registri di output si sovrascriverebbero i registri di input (che hanno altri dati validi). Ogni volta che viene eseguita una SAVE, è incrementato CANRESTORE e decrementato CANSAVE; vice versa quando viene eseguita una RESTORE viene incrementato CANSAVE e decrementato CANRESTORE. CANSAVE è usato per segnalare un *overflow* (vie-

ne generata una *spill trap*), mentre CANRESTORE per segnalare la condizione di *underflow* (generando una *fill trap*).

E' possibile che qualche finestra abbia dati validi provvenienti da altri contesti (altri programmi o *software thread*, per esempio): tali finestre sono conteggiate in OTHERWIN.

La seguente equazione deve sempre essere valida:

$$CANSAVE + CANRESTORE + OTHERWIN = N\_REG\_WINDOWS - 2 \tag{7.1}$$

Le due window indicate numericamente nella equazione, sono la finestra corrente e la finestra di *overlap*.

In figura 7.6 sono rappresentate sia alcune finestre che i valori di alcuni registri di stato dei *windowed register*: CWP=0, CANSAVE=4, CANRESTORE=1, la finestra 5 è di *overlap* (viene persa). L'istruzione di SAVE sposta il CWP in senso orario, e definisce il verso dei CWP crescenti; vice versa RESTORE, che fa decrescere CWP, è in senso antiorario.

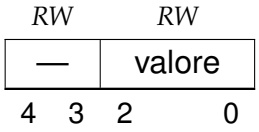
7.4.3 Registri priviledge per il management dei *windowed register*

I registri trattati in questo paragrafo sono tutti registri ad accesso priviledge. Sono previste due istruzioni per leggere e scrivere su questi registri: RDPR e WRPR.

CWP (PR 9), CANSAVE (PR 10), CANRESTORE (PR 11), CLEANWIN (PR 12) e OTHERWIN (PR 13) sono registri a 5 bit[10], di cui solo i 3 meno significativi utilizzabili nell'OpenSPARC T1. Assumono valore massimo valore pari a 7. CANSAVE, CANRESTORE e CLEANWIN non dovrebbero mai avere valori superiori a *N_REG_WINDOWS* - 2, sebbene al software hyperpriviledge sia concesso non rispettare tale regola nè l'equazione 7.1. Di seguito, una rappresentazione dei suddetti registri:

[10]Le architetture UltraSPARC supportano fino a 32 window.

Tabella 7.4: E' illustrata la forma di CWP, CANSAVE,CANRESTORE,CLEANWIN, OTHERWIN.



7.4.3.1 WSTATE (PR 14)

Il registro *Window STATE* (WSTATE), anch'esso ad accesso priviledge, specifica quali bit saranno inseriti in TT[TL]{4:2} sulle trappole causate dalla generazione di eccezioni "*window spill*" o "*window fill*".Questi bit sono usati per selezionare una tra le otto finestre. Se OTHERWIN=0 quando inizia l'esecuzione del codice-trappola causata da *window spill* o *fill*, allora vengono inseriti in TT[TL] i bit WSTATE.normal; altrimenti sono inseriti i bit WSTATE.other[11]. In tabella 7.5, è fornita una rappresentazione grafica di WSTATE, mentre in tabella 7.6è fornita una rappresentazione di TT[TL] nel caso particolare che avvenga la generazione eccezioni *window fill* o *spill*.

Tabella 7.5: Registro WSTATE (PR 14).

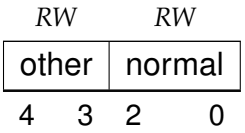


Tabella 7.6: TT[TL] per *window spill* e *fill trap*.

bit	field name	valore	
8:6	spill_or_fill	010_2 per *spill trap*	011_2 per *fill trap*
5	other	1, se otherwin>0	
4:2	wtype	WSTATE.other se other=1; WSTATE.normal altrimenti	
1:0	—	00_2	

[11] *Trap Type* è il registro da 9bit su cui viene scritto via hardware il codice corrispondente all'eccezione che viene generata. Esiste un registro TT per ogni *Trap Level* (TL).

7.5 Registri interni

7.5.1 HPSTATE

Accesso Hyperpriviledge, accessibile con le istruzioni WRHPR e RDHPR.

L'*Hyperpriviledge State register*, contiene i campi di controllo Hyperpriviledge del Virtual Processor. C'è una istanza di HPSTATE per ogni Virtual Processor.

Ci sono solo 4 bit sui 64 del registro che assumono significato.

HPSTATE.enb : HPSTATE{11} Il valore di default è 0 dopo un reset.

Non è specificato nella documentazione il significato di questo bit. In una nota del documento 3.1.2.5, viene richiamato il programmatore ad assicurarsi che questo bit abbia sempre valore 1, per abilitare non meglio specificate funzionalità del processore. Anche nella tabella dello stesso documento che riporta lo stato del processore dopo un POR (Power-On Reset) viene evidenziato che dopo il reset il bit è nullo, e che debba essere impostato a 1: *"must be set to 1 by software"*.

HPSTATE.ibe : HPSTATE{10} *Instruction Breakpoint Enable*. Il valore di default è 0.

Quando HPSTATE.ibe=1, possono essere generate le *instruction breakpoint exception*.

HPSTATE.red : HPSTATE{5} *RED state*: *Reset, Error and Debug state*.

HPSTATE.red è settato a 1 dopo un reset e quando viene generata una trappola mentre TL = HVER.maxtl – 1. Via software, si può uscire dal *RED state* in due modi:

1. eseguendo una istruzione chiarisce di DONE o RETRY, che impostano HPSTATE ad una versione precedentemente salvata, se quando è stata salvata non si era in *RED state*;

2. scrivendo 0 nel HPSTATE.red, attraverso l'istruzione WRHPR, rispettando due raccomandazioni:

   - non inserire la WRHPR nel *delay slot* di una istruzione DCTI (alcune istruzioni di *branch* eseguono comunque l'istruzione successiva, a pre-

scindere dalla condizione di controllo, come ad esempio l'istruzione JMP).

- HPSTATE.hpriv = 0 e HPSTATE.red = 1 è una combinazione che lascia il Virtual Processor in uno stato indefinito.

HPSTATE.hpriv : HPSTATE{2} *Hyperpriviledge mode*. Il valore dopo un reset è 1.

Quando HPSTATE.hpriv è 1 il Virtual Processor opera in modalità hyperpriviledge, ignorando il bit PSTATE.priv.

Se HPSTATE.hpriv è 0 il processore opera in modalità priviledge o non-priviledge, a seconda del valore di PSTATE.priv (vedi tab. 6.2).

L'istruzione WRHPR non dovrebbe mai essere inserita nel delay slot di una istruzione DCTI, quando usata per cambiare valore ad HPSTATE.hpriv.

HPSTATE.tlz : HPSTATE{0} *Trap Level Zero*. Il valore dopo un reset è 1.

Se HPSTATE.tlz è 1, ogni qual volta da si passa da uno stato con TL >0 ad uno stato con TL =0 viene generata la eccezione *trap_level_zero*. Altrimenti la generazione di tale trappola viene inibita.

Il software HP può abilitare questo bit, consente una più efficiente procedura di *descheduling* del processore virtuale.

7.5.2 HTBA

Accesso hyperpriviledge, accessibile con le istruzioni WRHPR e RDHPR.

L'*Hyperpriviledge Trap Base Address*, contiene i primi 34 bit dell'indirizzo fisico di memoria dal quale inizia il codice per la gestione di una eccezione occorsa durante l'esecuzione di codice hyperpriviledge. La porzione dell'indirizzo è usata per scegliere tra i vettori-trappola. Esiste una istanza dell'HTBA per Virtual Processor.

Nell'OpenSPARC T1 sono implementati solo i 34 bit [47:14] del registro: i 13 bit bassi sono sempre nulli, mentre i bit più significativi sono l'estensione di segno del dato, ricopiano il bit 47. Quando viene generata una trappola in hyperpriviledge mode, viene costruito un *vettore-trappola* (Trap Table Entry) che comprende: i bit [47:14] di questo registro, una copia di TT[TL] (il *trap-level*, la priorità dell'eccezione) ai bit [13:5], l'estensione di segno per i bit [63:48] e i 5 bit [4:0] sono a 0.

Una TTE può dunque contenere fino a $2^5 = 32$ byte: ogni TTE può contenere le prime 8 istruzioni del codice di gestione della trappola (*trap handler*).

7.5.3 HVER

Accesso hyperpriviledge, accessibile in sola lettura con l'istruzione RDHPR.

Hyperpriviledged Implementation VERsion.

Il registro ad accesso hyperpriviledge HVER è scritto via hardware ed è in sola lettura. Vi sono scritte delle costanti caratteristiche della versione del core.

Tabella 7.7: HVER.

Bit	Campi	Valore POR (bin)	Descrizione
63:48	manuf	0000_0000 0011_1110	Campo dedicato all'ID del produttore
47:32	impl	0000_0000 0010_0011	Sigla dell'implementazione del core: in questo caso OpenSPARC T1, implementazione dell'UltraSPARC
31:24	mask	0010_0000	Specifica l'attuale implementazione del set di maschere per i valori. Scelta dal produttore
23:19	—	0_0000	—
18:16	maxgl	011	Numero (meno 1) di registri globali supportati dall'implementazione
15:8	maxtl	0000_0110	Numero massimo di livelli di trappola del core. E' anche il valore massimo del registro TL
7:5	—	000	—
4:0	maxwin	0_0111	Numero massimo di window supportati dal core. E' il massimo valore del registro CWP

7.5.4 LSU Diagnostic Register

Accesso hyperpriviledge, ASI 42_{16}, Virtual Address 10_{16}.

Dopo un POR, assume valore 0.

Disabilita l'associatività nelle caches di primo livello. E' usato nella diagnostica e nel debug. In tabella 7.8, si riassume il comportamento.

Tabella 7.8: LSU Diagnostic Register.

Bit	Campi	R/W	Descrizione
63:2	-	RO	*Reserved*
1	dassocdis	RW	Se a 1, l'algoritmo di rimpiazzo delle linee della D-Cache non sarà LRU, ma verranno usati i bit 12:11 del VA per determinare la linea da sovrascrivere quando avviene un *miss*.
0	iassocdis	RW	Come *dassocdis*, ma riguarda la I-Cache anzichè la D-Cache.

Documento di riferimento: "UltraSPARC T2 Supplement to the UltraSPARC Architecture 2007", vedi fine §3.1.2.5.

7.5.5 LSU Control Register

Accesso hyperpriviledge, ASI 45_{16}, Virtual Address 0_{16}.

Dopo un POR, assume valore 0.

Il LSU Control Register contiene campi che controllano molte funzioni hardware legate alla gerarchia di memoria: I-Cache, D-Cache, MMU.

Tabella 7.9: LSU Control Register.

Bit	Campi	RW	Descrizione
63:41	-	R	*Reserved*
40:33	PM	R	*Reserved. (Normally Physical Address Data Watchpoint Byte Mask*)
32:25	VM	RW	*LSU.virtual_address_data_watchpoint_mask*. Il registro ASI_DMMU_VA_WATCHPOINT contiene i 64 bit del *virtual address* che si vuole controllare. Gli 8 bit di VM indicano quale tra i byte della doubleword si vuole controllare. Se VM è 0, il Watchpoint è disabilitato. Se abilitato e avviene il match, una *VA_watchpoint trap* è generata.
24	PR	R	*Reserved. (Physical Address Data Watchpoint Read Enable.)*
23	PW	R	*Reserved. (Physical Address Data Watchpoint Write Enable)*

Bit	Campi	RW	Descrizione
22:21	VR/VW	RW	*LSU.virtual_address_data_watchpoint_enable*. Se a 1, il read(VR)/write(VW) di un dato nel range di indirizzi virtuali del *Virtual Watchpoint Register* causa una *watchpoint trap*.
20:4	-	R	*Reserved.*
3	DM	RW	*LSU.enable_D-MMU*[12]. Se 0, la D-MMU è disabilitata (*write-through mode*).
2	IM	RW	*LSU.enable_I-MMU* [3]. Se 0, la I-MMU è disabilitata (*write-through mode*).
1	DC	RW	*LSU.D-Cache_enable*. Se 0, i miss sono forzati ad avere accesso alla D-cache senza poterla riempire. Dopo aver cambiato il bit, perchè l'operazione abbia effetto sulle successive istruzioni, bisogna eseguire una FLUSH, DONE o RETRY.
0	IC	RW	*LSU.I-Cache_enable*. Se 0, i miss sono forzati ad avere accesso alla I-cache senza poterla riempire.

7.5.6 SFSR

Synchronous Fault Status Register.

ASI 50_{16}, Virtual Address 18_{16} per il I-MMU SFSR;

ASI 58_{16}, Virtual Address 18_{16} per il D-MMU SFSR.

Detiene il valore delle eccezioni riguardanti la MMU (Memory Management Unit). In generale, il valore di questo registro cambia quando è generata una eccezione che riguardi l'accesso ai dati, errori di traduzione degli indirizzi e indirizzi non allineati con i dati su cui operano. Può cambiare il suo valore anche al generarsi di altre eccezioni, che hanno come effetto secondario la generazione di un errore tra quelli gestiti dalla MMU.

Ulteriori dettagli sulle eccezioni legate alla MMU, si può avere consultando i documenti richiamati ai paragrafi 3.1.2.4 e 3.1.2.5.

[12]Quando la MMU/TLB è disabilitata, il VA viene tradotto come PA. Gli accessi sono *non-cacheable* con *side-effects*.

7.5.7 SPARC Error Enable Register

Accesso Priviledge. ASI $4B_{16}$, Virtual Address 0_{16}.

Il registro consiste in due soli campi (vedi tabella 7.10), ognuno dei quali abilita la generazione di trappole riguardanti ECC (*Error Correction Code*) e parità.

La documentazione (vedi §3.1.2.5) dichiara che entrambi i campi nceen e ceen dovrebbero essere sempre attivi, salvo in qualche raro caso dovuto alla concomitanza di più errori.

Tabella 7.10: SPARC Error Enable Register.

bit	Campi	R/W	Descrizione
63:2	-	RO	*Reserved*
1	nceen	RW	Se a 1, è possibile la generazione di *Trap on Uncorrectable error*.
0	ceen	RW	Se a 1, è possibile la generazione di *Trap on Correctable error*.

7.5.8 I-/D-TLB Data-Access/Data-In Registers and I-/D-TLB Tag Access Register

Gli ASI ed i virtual address che indirizzano i sei registri, Data Access, Data In e Tag Access, della I-MMU e la D-MMU, sono listati in tabella 7.12.

Tabella 7.12: ASI e VA per I-/D-TLB Data-Access/Data-In e I-/D-TLB Tag Access Register.

Nome	I-MMU ASI	D-MMU ASI	VA{63:0}	R/W	Repliche
I-/D- TLB Tag Access Register	50_{16}	58_{16}	0_{16}	R	Strand
I-/D- Data In Register	54_{16}	$5C_{16}$	0_{16}	W	Phisical Core
I-/D- Data Access Register	55_{16}	$5D_{16}$	$0_{16}...1F8_{16}$	RW	Phisical Core

Sia il Data Access che il Data In register servono per leggere e scrivere i TLB. I due registri svologono lo stesso ruolo in contesti diversi:

- il Data Access register serve per accessi diretti ai TLB da parte del sistema operativo, a scopo di diagnostica o scrittura diretta di una TLB-entry;
- il Data In register è usato per la riscrittura automatica dei TLB e TSB, dovuta a TLB-miss e TSB-miss.

I due registri, possono essere scritti sia nella forma TTE *sun4u* che *sun4v*, controllata dal bit 10 del VA associando 0 e 1 rispettivamente. Il bit real del *TLB entry* è controllato invece dal bit 9 del VA e verrà riscritto nel medesimo bit del *TLB entry*.

La forma del Virtual Address per indirizzare il TLB Data In register è descritto in tabella 7.13.

Tabella 7.13: MMU TLB Data In, Virtual Address Format

—	sun4v	real	—	000
63 — 11	10	9	8 — 3	2 — 0

La forma del Virtual Address per indirizzare il TLB Data Access register è descritto in tabella 7.14. Il campo *TLB entry* può valere 0...63 e specifica quale dei 64 valori (linee) del TLB si vuole scrivere: questo e gli altri campi sono specificati nel VA dell'istruzione STXA usata per scrivere questo registro.

Tabella 7.14: MMU TLB Data Address, Virtual Address Format

—	sun4v	real	TLB entry	000
63 — 11	10	9	8 — 3	2 — 0

Per esempio, la scrittura di questo registro, volendo scrivere la 3° entry, con real=0 e sun4v=1, sarebbe: `stxa %g0, [0x418] 0x5d`.

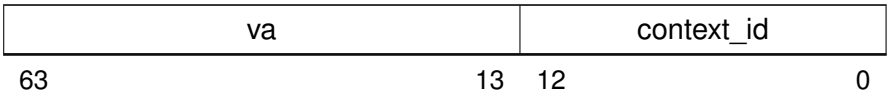
I Tag Access Register sono usati come buffer temporaneo per la scrittura dei *TLB entry tag*. Può essere scritto in due occasioni:

- è generata una eccezione dovuta ad un miss, ed il registro viene scritto automaticamente con il VA ed il context, per facilitare la formazione del TSB Tag Target Register;

- una scrittura diretta tramite l'istruzione STXA. Prima di una scrittura sul TLB Data Access Register, il sistema operativo deve scrivere il campo tag in questo registro. Anche nella generazione automatica di un valore nel TLB da parte della MMU avviene nella stessa sequenza: viene impostato il tag e poi scritto il dato.

I due campi di questo registro sono il VA, con i bit più significativi che sono l'estensione di segno del bit VA{47} e i 13 bit del *context*, che vengono letti come nulli se non è associato un contesto all'accesso. La forma del registro è illustrata in tabella 7.15.

Tabella 7.15: I/D MMU TLB Tag Access Register



Nel documento UltraSPARC Architecture 2007, una tabella chiarisce il corretto utilizzo di load e store nei due registri, e viene qui riportata in tab. 7.16.

Tabella 7.16: Uso di Tag Access, TLB Data-In e Data-Access register.

SW operation		**Effetti sui registri della MMU**		
LD/ST	**Register**	**TLB tag**	**TLB data**	**Tag Access Register**
Load	Tag Access	No effect. Contents returned	No effect.	No effect.
	Data In	—		
	Data Access	No effect.	No effect. Contents returned	No effect.
Store	Tag Access	No effect.	No effect.	Written with store data
	Data In	TLB entry determined by replacement policy written with contents of Tag Access Register	TLB entry determined by replacement policy written with store data	No effect.
	Data Access	TLB entry specified by STXA address written with contents of Tag Access Register	TLB entry specified by STXA address written with store data	No effect.

SW operation		Effetti sui registri della MMU		
LD/ST	Register	TLB tag	TLB data	Tag Access Register
TLB miss		No effect.	No effect.	Written with VA and context of access

7.6 Reset

Ad ogni CPU dell'OpenSPARC T1 possono essere inviati 5 tipi di reset:

- power-on reset (POR)
- warm reset (WMR)
- externally initiated reset (XIR)
- watchdog reset (WDR)
- software-initiated reset (SIR)

Le tipologie POR, WMR e XIR sono segnali di reset inviati dall'esterno del chip. WDR e SIR sono invece generati dal virtual processor in risposta ad alcune condizioni.

Lo stato del processore dopo un reset è descritto nel capitolo dedicato al reset, nei documenti §3.1.2.4.

Capitolo 8

CONCLUSIONI

E' stata fornita una procedura che rende simulabile l'S1 Core su due piattaforme: Icarus Verilog e Modelsim.

Percorrendo una strada nuova, si sono svolti test sperimentali sull'S1 Core con strumenti opensource come Icarus Verilog.

Nella istanziazione del design di due verioni del core, quelle in realtà più interessanti perchè di minore area, Icarus Verilog fallisce: si è riusciti a simulare i design ridotti su Modelsim, in precedenza non supportato, e si è descritta la procedura.

Nel capitolo 4 si è dimostrata l'inadeguatezza della procedura di boot dell'S1 Core, anche nell'esecuzione di un semplice programma che richieda due STORE in memoria. Si sono elaborati i codici che consentono di raggiungere stati stabili per l'uso corretto dei *windowed register*, il management delle eccezioni e dei codici-trappola. Si è indagato sui registri di stato più importanti del core e fornite configurazioni compatibili. In simulazione, si è avuto sempre un occhio di riguardo verso la configurazione delle MMU e sono state portate alla luce problematiche inerenti la traduzione degli indirizzi, un nodo parzialmente irrisolto. Si è arrivati, certo, a definire una configurazione che generasse sul bus una richiesta corretta di STORE in RAM, ma la memoria non risponde finora con dati validi. Va aggiunto che il controllo della gerarchia di memoria dello SPARC core è assegnata per buona parte al codice priviledge[1]. Nell'ottica di avere un sistema operativo che sfrutti il processore, restano da configurare solo alcuni dei registri dedicati alla gestione degli indirizzi, tipicamente quelli ad accesso hyperpriviledge.

[1] Il livello di esecuzione priviledge è dedicato ai sistemi operativi.

Indagando su situazioni-limite generate ad hoc per evidenziare le dinamiche che ogni configurazione proposta mette in essere, si è cercato di dare un senso ai diversificati comportamenti del core.

Al momento di stesura del presente documento, si è avuta coscienza che alcune parti del processore sarebbero state ancora da ottimizzare; è stata considerata inadeguata la documentazione dell'S1 Core nell'ottica di effettuare test senza disporre della piattaforma OpenSPARC T1 funzionante. E' stata pertanto incrementata la documentazione sull'S1 Core, evidenziando le relazioni tra i file e le funzionalità operative di ciascuno all'interno del package. Si sono indicate soluzioni sembrate congeniali a rendere il sistema di script più *user-friendly*. A supporto delle simulazioni, non è stato tralasciato di riportare informazioni teoriche sull'OpenSPARC T1. E' stata inoltre redata una piccola guida all'uso della vasta documentazione utilizzata nello studio del core e nell'interpretazione dei test.

Uno degli obiettivi del progetto nel quale si colloca questa tesi è di avere una istanza su FPGA dell'S1 Core, per altro, modificandone l'hardware del bus. Prima compiere questi passi, è necessario che l'hardware sia verificato tramite simulazioni che rassicurino sul comportamento logico corretto. L'S1 Core, nella sua versione originale, non è stato convincente da questo punto di vista. I risultati riportati a supporto della tesi, dimostrano che l'S1 Core è simulabile e che l'hardware risponde bene in simulazione. Finora non sono stati riscontrati problemi sulla descrizione dell'hardware.

Sicuramente altre due sezioni del core andranno configurate al meglio: il controllo dei dati da e verso le memorie e i registri di gestione del *multi-threading*, argomento quest'ultimo che impregna le logiche dello SPARC core e va ora affrontato con la configurazione dei registri dedicati.

Un ulteriore sviluppo che, dato l'uso diffuso di Modelsim, può non essere secondario, è di studiare un sistema che renda simulabile sia l'S1 Core che l'OpenSPARC T1 su tale piattaforma, tramite la creazione di script che rendano Modelsim compatibile con i test e le riduzioni dell'OpenSPARC. E' questa una problematica molto sentita dagli hardware designer interessati all'OpenSPARC e che ci si è posti durante la scelta della piattaforma di sviluppo del progetto. Un piccolo passo verso tale compatibilità è stato compiuto mostrando come si può usare Modelsim per testare l'S1 Core attraverso la creazione di un progetto per via grafica.

Naturalmente, sarebbe augurabile poter avere il sistema di test dell'Open-

SPARC eseguibile anche con Icarus Verilog: per testare con certezza relativamente assoluta l'integrazione di nuovi componenti su un design complesso come quello dello SPARC, è necessario avere test comparati, per ridurre al minimo gli inevitabili errori umani di valutazione.

Una volta configurato il boot e simulato con successo l'S1 Core, si potrà dare il via al processo di sintesi ed averne una istanza su FPGA. Essendo un processore, i campi di applicazione di tale tecnologia nell'ambito dell'elettronica moderna sono praticamente illimitati. Dato che lo SPARC core è stato estratto da una struttura già di per sè multicore, implementa caratteristiche che ne fanno un ottimo candidato ad essere parte integrante di un Network on Chip.