VERIFICA FUNZIONALE DELL'ALU DEL PROCESSORE GAISLER LEON3

PIERLUIGI PICCIAU

Laurea Magistrale in Ingegneria Informatica
Università degli Studi di Padova

11 Dicembre 2012

Pierluigi Picciau: *Verifica funzionale dell'ALU del processore Gaisler Leon3,*
11 Dicembre 2012

ABSTRACT

Questo documento è frutto del mio lavoro di tesi svolto sotto la supervisione dell'Ing. Alessandro Ogheri. Mi sono occupato della verifica funzionale dell'ALU del microprocessore Gaisler Leon3 rilasciato con licenza GPL. In un primo momento ho studiato le basi del linguaggio di descrizione dell'hardware VHDL, per poi rivolgere la mia attenzione verso le tecniche di verifica dell'hardware e verso il linguagggio di verifica *e*. Grazie alle conoscenze apprese ho sviluppato un ambiente di verifica per il processore Leon3, focalizzando l'attività di verifica sulle istruzioni che coinvolgono l'ALU (istruzioni aritmetiche, logiche e di controllo del flusso). L'ambiente sviluppato è stato infine impiegato per l'esecuzione di una serie di test.

People who are really serious about software should make their own hardware.

— Alan Kay

INDICE

ELENCO DELLE FIGURE

ELENCO DELLE TABELLE

CODICI

1 INTRODUZIONE

1.1 BACKGROUND

Il processo di progettazione dei componenti hardware si è evoluto nel tempo in risposta al veloce progredire della tecnologia e alla crescente capacità produttiva. Con l'affinamento delle metodologie di design si è presto evidenziato un gap di produttività dovuto alla fase di verifica della progettazione. A causa dell'aumentare della complessità dei circuiti elettronici, e al contempo del livello di astrazione con il quale essi vengono progettati, l'attività di verifica richiede un sempre maggior numero di risorse. Per tale motivo ad oggi risultano di estremo interesse metodologie di verifica che consentano di colmare il gap produttivo. Le nuove metodologie permettono non solo di ridurre il time-to-market, ma anche di aumentare la qualità stessa della verifica.

1.2 IL LAVORO DI TESI

Il lavoro presentato in questo documento è stato svolto sotto la supervisione dell'azienda *Ogheri Consulting GmbH*, nella persona dell'Ing. Alessandro Ogheri, il quale si occupa da diversi anni della verifica funzionale di hardware. I principali clienti di *Ogheri Consulting GmbH* sono grandi aziende europee o multinazionali che necessitano di ingegneri da affiancare al team di progetto per la fase di verifica del design.

L'obiettivo proposto per il lavoro di tesi consiste nello sviluppo in linguaggio *e* di un ambiente per la verifica funzionale dell'ALU del microprocessore Gaisler Leon3. Questo processore è disponibile con licenza open source GPL sottoforma di codice sorgente VHDL, risultando pertanto molto adatto a fini di ricerca o educativi. Il processore Leon3 è caratterizzato da una archiettura semplice, costruita sulle basi dell'architettura SPARC v8 e compatibile con essa. L'azienda che ne porta avanti lo sviluppo è *Aeroflex Gaisler AB*, con sede in Svezia.

1.3 STRUTTURA DEL DOCUMENTO

Questo documento è suddiviso in due parti. Nella prima parte si presenta al lettore lo stato dell'arte relativamente alla progettazione di hardware (Capitolo 2) e alla sua verifica (Capitolo 3).

Nella seconda parte si descrive il componente hardware sottoposto a verifica (Capitolo 4) e si analizza l'ambiente di verifica funzionale

sviluppato a tal proposito (Capitolo 5).

Nel capitolo 6 vengono descritte le modalità di realizzazione dei test compiuti con l'ambiente di verifica e vengono proposti possibili ulteriori sviluppi.

Parte I

STATO DELL'ARTE: PROGETTAZIONE E VERIFICA DELL'HARDWARE

2 PROGETTAZIONE DI LOGICHE DIGITALI

2.1 LINGUAGGI HDL

A causa dell'esplosione della complessità dei circuiti elettronici digitali, a partire dagli anni '70 i progettisti di circuiti iniziarono a sentire l'esigenza di un sistema per descrivere ad alto livello le logiche digitali. Nacquero così i primi HDL, ovvero *Hardware Description Language*[1]: tali linguaggi permettono di rappresentare la struttura spaziale e temporale dei sistemi elettronici, nonchè il loro comportamento.

I linguaggi HDL differiscono dai comuni linguaggi di programmazione software per alcuni aspetti, tra i quali si evidenziano il supporto esplicito alla programmazione concorrente e la nozione del tempo, entrambe caratteristiche fondamentali per la trattazione di sistemi hardware.

Gli HDL vengono impiegati per scrivere le specifiche dell'hardware e il codice risultante può essere eseguito, simulando così il funzionamento dell'hardware prima ancora che esso venga fisicamente prodotto. Inoltre se si limita l'utilizzo dei costrutti all'interno del sottoinsieme dei costrutti cosiddetti sintetizzabili, è possibile utilizzare un software chiamato *tool di sintesi* per derivare in modo automatico le operazioni logiche di base e produrre una rete di porte logiche elementari che implementino l'hardware modellato in HDL.

Allo stato attuale, i linguaggi HDL più diffusi e meglio supportati sono due:

- Verilog
- VHDL

2.2 VHDL

VHDL è l'acronimo di *VHSIC Hardware Description Language*, dove *VHSIC* sta per *Very High Speed Integrated Circuits*[2]. VHDL è un HDL tra i più utilizzati.

Lo sviluppo di VHDL iniziò come progetto del Dipartimento della Difesa statunitense in risposta alla necessità di un linguaggio per descrivere l'hardware che fosse dotato di alcune caratteristiche: doveva essere leggibile sia da operatori umani che da macchine; doveva obbligare lo sviluppatore a scrivere codice strutturato e comprensibile,

1 Linguaggi di descrizione dell'hardware.
2 Circuiti integrati a velocità molto elevata.

PERIODO	AVVENIMENTO
inizio '70	Discussione Iniziale
fine '70	Definizione dei requisiti
1982	Contratto di sviluppo da parte di IBM, Intermetrics e TI
1984	Versione 7.2
1986	Definizione Standard IEEE 1076
1987	Il Dipartimento della difesa adotta lo standard (IEEE.1076)
1988	Crescente supporto da parte dei produttori CAE
1991	Revisione
1993	Aggiornamento standard IEEE.1076
1999	Estensione VHDL-AMS
2006	Approvazione Draft 3.0 di VHDL-2006
2008	Approvazione VHDL 4.0 (noto come VHDL-2008)
2009	Pubblicazione VHDL 4.0 (IEEE 1076-2008)

Tabella 1: VHDL: Cronologia.

di modo che il codice stesso potesse essere utilizzato come un documento di specifiche; doveva infine gestire in modo efficace il concetto della concorrenza dovuta all'intrinseco parallelismo dell'hardware digitale.

VHDL è un linguaggio che viene costantemente esteso e revisionato: è previsto un processo di revisione ogni 5 anni. Oltre a tale processo di revisione, con il passare del tempo sono stati compiuti ulteriori sforzi per rendere standard alcune estensioni che risultano essere molto utilizzate nella pratica. Queste estensioni comprendono ad esempio dei *packages* contenenti dei tipi di dato molto comuni. Inoltre sono stati definiti come standard dei sottoinsiemi di VHDL, ad esempio quello specifico per la sintesi[3].

Una sfida impegnativa è stata intrapresa di recente con l'intento di aggiornare VHDL, introducendo elementi al linguaggio per consentire la trattazione di segnali misti analogico-digitali. Questo aggiornamento prende il nome di VHDL-AMS[4] ed è un sovrainsieme di VHDL.

2.2.1 *Livelli di astrazione*

La versatilità e l'espressività di VHDL sono per buona parte dovuti alla gestione di più livelli di astrazione. Per *astrazione* si intende l'occultamento di informazioni che risultano essere troppo dettaglia-

3 IEEE 1076.6

4 VHDL Analogue-Mixed Signal

te rispetto al livello di dettaglio nel quale ci si è posti. Una volta che si è stabilita la differenza tra informazioni essenziali e non, l'informazione che non risulta importante per il livello corrente di dettaglio viene scartata.

I livelli di astrazione di un circuito digitale sono quattro (riportati in figura 1:

1. **Livello comportamentale** (*Behaviour*). In questo livello viene tracciata una descrizione funzionale del modello. Non è presente la nozione di *clock di sistema* e le transizioni dei segnali avvengono in modo asincrono. Le descrizioni a livello comportamentale tipicamente non sono sintetizzabili, ma solo simulabili. Il livello comportamentale è adatto alla descrizione del funzionamento di sistemi ed algoritmi.

2. **Livello RTL** (*Register Transfer Level*). Il design è partizionato in una parte di logica combinatoria e una parte di elementi di memoria (registri). Questi ultimi (ad esempio flip flop e latch) sono controllati dal clock di sistema. La logica combinatoria consiste in un insieme di funzioni impiegate per calcolare il valore successivo che verrà memorizzato negli elementi di memoria. Per descrivere il livello RTL è sufficiente (e necessario) solo il 10%-20% dei costrutti VHDL. Inoltre occorre seguir un preciso stile di modellazione, con il quale è garantita la sintetizzabilità del design.

3. **Livello logico**. In questo livello il design è rappresentato come una *netlist*[5] di porte logiche elementari (*AND, OR, NOT...*) e di elementi di memoria. La netlist è generata a partire dalla descrizione RTL con l'aiuto di un tool software di sintesi, il quale utilizza una libreria di celle elementari rese disponibili dalla tecnologia target impiegata. La libreria contiene informazioni su tutte le porte logiche diponibili per una data tecnologia, corredate dai loro parametri operativi.

4. **Livello del layout**. Le differenti celle selezionate nell'ambito della tecnologia target vengono disposte nel chip e con esse anche le connessioni. Il circuito a questo livello è pronto per il processo di produzione.

VHDL è un linguaggio applicabile ai primi tre livelli superiori (livello comportamentale, livello RTL e livello logico), mentre non è adeguato per la descrizione di un layout. La progettazione del livello comportamentale e RTL tipicamente avviene con la scrittura di codice sorgente VHDL da parte di uno sviluppatore umano. I livelli inferiori (logico e di layout) invece sono raggiunti tramite operazioni (tipicamente automatiche) di sintesi.

5 L'insieme delle connessioni (*net*) elettriche di un circuito elettronico.

2.2.2 *Sintesi*

Con *sintesi* ci si riferisce al processo di transizione da un livello di astrazione superiore al livello immediatamente inferiore. I differenti tipi di sintesi sono riportati in figura 1, insieme ai livelli di astrazione di un circuito digitale.

1. **Sintesi comportamentale** (*da livello comportamentale a livello RTL*). Una sintesi di questo tipo è realizzabile solo da uno sviluppatore umano, in quanto richiede una notevole capacità creativa. La sintesi comportamentale automatica è al momento possibile solo in alcune applicazioni particolari (ad esempio il design di celle RAM, per le quali è sufficiente specificare solo il valore di alcuni parametri generici).

2. **Sintesi logica** (*da livello RTL a livello logico*). Questo tipo di sintesi è stata perfezionata nel corso degli anni, ed ora è effettuabile in modo completamente automatico tramite dei tool software. Condizione necessaria affinchè la sintesi automatica possa avvenire è che lo sviluppatore descriva il livello RTL utilizzando un determinato sottoinsieme ridotto di costrutti VHDL.

3. **Sintesi del layout** (*da livello logico a livello del layout*). Nota anche come *place&route* nel caso di design FPGA/CPLD, la sintesi del layout avviene in modo automatico grazie all'impiego di algoritmi efficienti di posizionamento, frutto della ricerca.

2.2.3 *Flusso di progettazione*

In figura 2 è riportato un modello semplificato del flusso di progettazione con VHDL (e più in generale con un HDL).

Tipicamente il design viene scritto direttamente in VHDL a livello RTL. Deve poi essere verificato utilizzato un simulatore VHDL: a tale scopo viene scritto un modello di *testbench* per l'applicazione di uno stimolo, per l'estrazione dei risultati e per la loro verifica. Poichè i testbench non verranno sintetizzati, nella loro scrittura non c'è alcuna restrizione relativamente ai costrutti VHDL da utilizzare o allo stile di modellazione da impiegare.

Il design a questo punto viene letto da un tool di sintesi, al quale deve essere specificata anche la tecnologia target da utilizzare per la realizzazione dell'hardware vero e proprio. Il design viene così sintetizzato in una *netlist* di porte logiche. Anche questo livello deve essere verificato con lo stesso testbench del livello RTL.

Successivamente viene calcolato il layout (nel caso di design ASIC) o il place&route (per design FPGA/CPLD). Le informazioni temporali vengono ricavate dai tool di layout e riscritte (in gergo *back-*

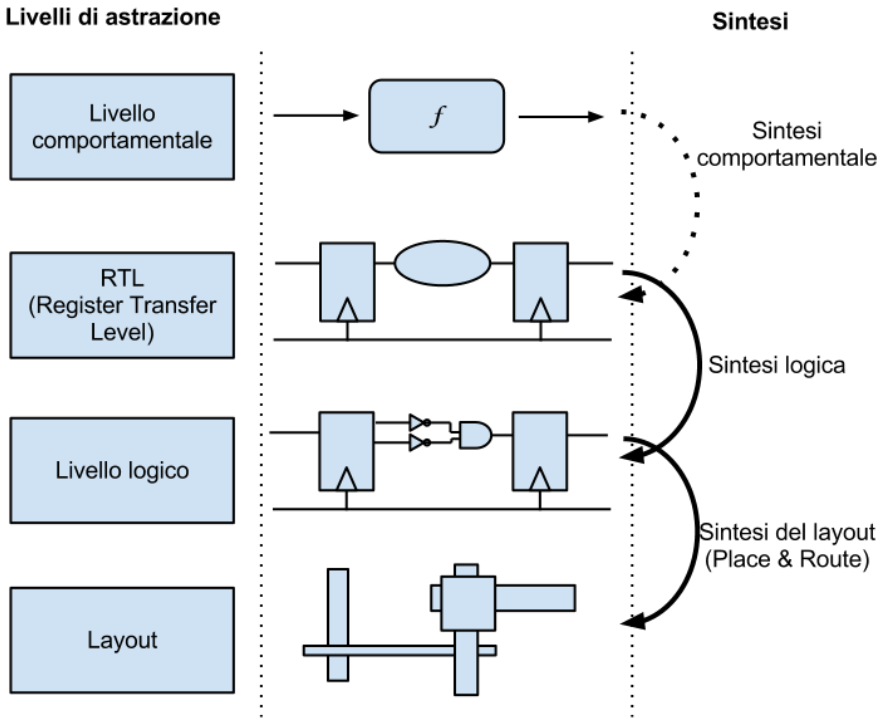

Figura 1: Livelli di astrazione e operazioni di sintesi.

annotated) nella netlist per effettuare la verifica dei vincoli temporali tramite un simulatore.

2.2.4 *Oggetti principali di VHDL*

Si presentano ora gli oggetti principali di VHDL che consentono di descrivere l'hardware in modo strutturato, mantenendo una suddivisione modulare tra i vari componenti e mantenendo separati il concetto di interfaccia dal concetto di funzionamento interno.

ENTITY. Un'entità descrive un'interfaccia la quale consiste generalmente in una lista di porte di comunicazione verso l'ambiente esterno.

Codice 1: Esempio VHDL: entity

```
1 entity HALFADDER is
2   port(A,B : in bit;
3        SUM, CARRY : out bit);
4 end entity HALFADDER;
```

L'interfaccia tra il modulo e l'ambiente è descritta all'interno la dichiarazione di *entity*, seguita da un nome descrittivo (*HALFADDER*). I segnali di input e quelli di output sono definiti nel costrutto *port*. Per ciascun segnale viene specificato un nome

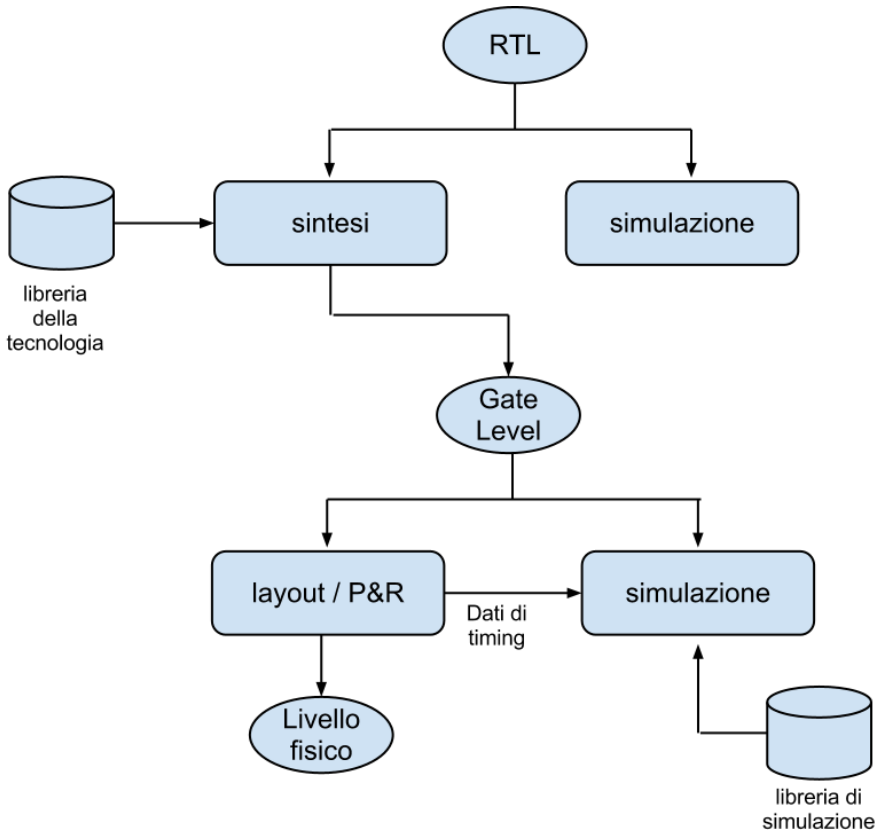

Figura 2: Flusso di progettazione.

che lo descriva, la direzione del flusso dati (*in*, *out*, *buffer*, *inout*) e il tipo di dato trasportato.

ARCHITECTURE. Un'architettura contiene la descrizione del funzionamento interno di una entità, ovvero la sua implementazione. Ciascuna architettura è strettamente collegata ad una certa entità; ogni entità può tuttavia avere molteplici architetture al suo interno. Ciascuna architettura, per una medesima entità, deve avere un nome identificativo diverso.

Codice 2: Esempio VHDL: architecture

```
1 architecture RTL of HALFADDER is
2 begin
3 SUM <= A xor B;
4 CARRY <= A and B;
5 end architecture RTL;
```

Ciascuna istruzione situata tra *begin* ed *end* viene eseguita concorrentemente alle altre: ciò significa che l'ordine in cui le istruzioni sono scritte non è importante.

CONFIGURATION. La configurazione è l'unico oggetto simulabile in VHDL, e consiste nella selezione di una coppia entità-architettura utilizzata per costruire il modello.

PROCESS. Sebbene le istruzioni VHDL siano eseguite concorrentemente, il costrutto del processo consente di ottenere una esecuzione sequenziale delle istruzioni. Il processo stesso invece, visto come unità atomica, viene eseguito concorrentemente ad altre istruzioni e/o processi (figura 3.

Codice 3: Esempio VHDL: process

```
1 A_O_X: process (A,B)
2 begin
3 Z_OR <= A or B;
4 Z_AND <= A and B;
5 Z_XOR <= A xor B;
6 end process A_O_X;
```

L'esecuzione di un processo è azionata da eventi. Tali eventi possono essere espressi implicitamente in quella che è chiamata *sensitivity list*, ovvero la lista di segnali che segue la parola *process* (nell'esempio: A e B); oppure possono essere specificati esplicitamente utilizzando l'istruzione *wait* all'interno del processo. Il simulatore VHDL invoca il codice del processo non appena il valore di almeno uno dei segnali appartenenti alla sensitivity list cambia. Di conseguenza, tutti i segnali che vengono in qualche modo letti all'interno del processo, ovvero tutti

i segnali che influenzano il suo comportamento, devono essere inseriti nella sensitivity list.

PACKAGE. I package sono collezioni di definizioni di tipi di dato, sottoprogrammi, costanti e altro. Sono particolarmente utili nei lavori di gruppo, dove ciascuno sviluppatore dovrebbe lavorare con le stesse definizioni di dati. Questo semplifica le connessioni tra moduli progettati da sviluppatori diversi. Un package è suddiviso in una intestazione e un corpo. L'intestazione contiene i prototipi di procedure o funzione, la definizione dei tipi di dati e così via. Il corpo del package invece contiene l'effettiva implementazione dei sottoprogrammi.

LIBRARY. Una libreria è una collezione di unità VHDL compilate (in codice oggetto). Tutte le strutture precedentemente analizzate, come package, entity, architecture, configuration possono essere raccolte in una libreria. Ciascuna libreria è identificata da un nome logico, il quale deve essere mappato dal tool di simulazione in un percorso fisico nel file system del computer.

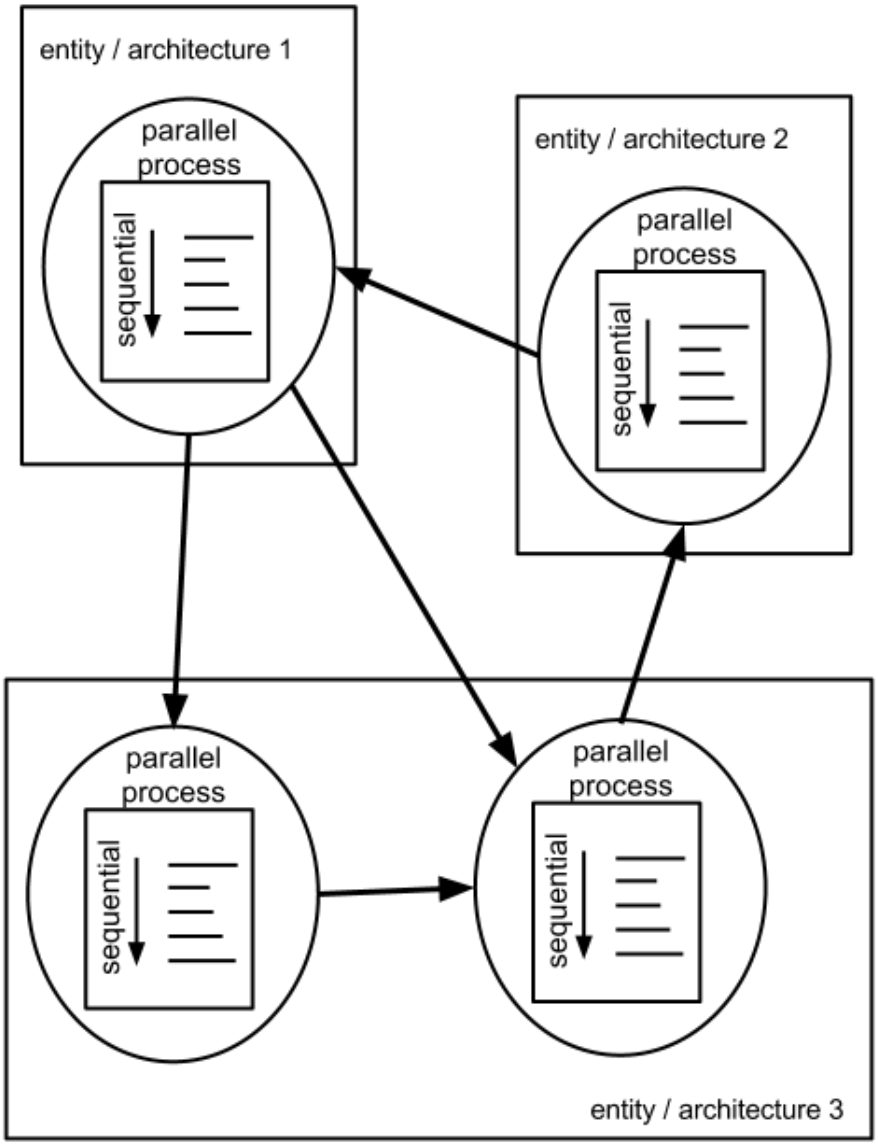

Figura 3: Esecuzione concorrente e sequenziale.

2.2.5 *Un design di esempio: timer contatore*

In questo esempio viene implementato un timer che utilizza un contatore incrementato ad ogni ciclo di clock di sistema, fino al raggiungimento di un valore prefissato (calcolato in base al ritado che si vuole ottenere e alla frequenza di clock).

Codice 4: Design di esempio VHDL: timer contatore

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity timer_transport is
5     Port ( ttgo : in  STD_ULOGIC;
```

```
6           clk : in  STD_ULOGIC;
7           reset : in  STD_ULOGIC;
8           ttr : out  STD_ULOGIC);
9  end timer_transport;
10
11 architecture RTL of timer_transport is
12
13 signal counter : integer range 0 to 16383;
14
15 begin
16    COUNT: process(reset,clk)
17    variable var : std_ulogic := '0';
18    begin
19       if (reset='1') then
20          counter <= 0;
21       elsif (clk'event and clk='1') then
22          ttr <= '0';
23          if (ttgo='0') then
24             var := '1';
25             counter <= 0;
26          end if;
27          if (var='1') then
28             if (counter /= 16383) then
29                counter <= counter + 1;
30             else
31                ttr <= '1';
32                var := '0';
33             end if;
34          end if;
35       end if;
36 end process COUNT;
37
38 end architecture RTL;
```

IMPORTAZIONE DELLE LIBRERIE. Righe 1-2. Viene importata la libreria IEEE che contiene le definizioni dei tipi di dato previsti dallo standard 1164. Tra questi, il tipo di dato principale è *STD_ULOGIC* (standard unresolved logic) che può rappresentare uno degli stati riportati in tabella 2.

DEFINIZIONE ENTITY. Righe 4-9. Viene definita l'entità *timer_transport*, caratterizzata da 4 porte a singolo bit, delle quali 3 sono di ingresso e una di uscita.

DEFINIZIONE ARCHITECTURE. Righe 11-38. Con queste istruzioni viene descritto il funzionamento interno (al quale è stato associato il nome *RTL*) dell'entità precedentemente definita. In questo esempio tutto il funzionamento consiste in un unico processo sequenziale.

DEFINIZIONE PROCESS. Righe 16-36. Il processo *COUNT* viene definito e associato ad una *sensitivity list* che comprende i segnali

reset e *clk*. Ciò significa che il processo viene eseguito ogni qualvolta vi sia una variazione nel valore portato da uno di quei due segnali. Il primo *if* che si incontra (riga 19) all'interno del processo implementa un reset asincrono, che avviene qualora la causa di avvio del processo sia un variazione a '1' del segnale *reset*. Altrimenti (riga 21) se si tratta di una variazione del segnale *clk*, su fronte di salita[6], inizia il conteggo se il segnale *ttgo* ha valore '0'. L'entrata nella modalità conteggio è contrassegnata dall'assegnazione del valore '1' alla variabile *var*. In modalità conteggio, ad ogni colpo di clock (segnale *clk*) viene incrementata la variabile *counter*, fino al raggiungimento del valore di stop (16383).

SIMBOLO	VALORE
U	Non definito (a cui non è mai stato dato un valore)
X	Sconosciuto (il cui valore non è determinabile)
0	0 logico
1	1 logico
Z	Alta impedenza
W	Segnale debole, senza un valore determinabile
L	Segnale debole, 0 logico
H	Segnale debole, 1 logico
-	Indifferente (*Don't care*)

Tabella 2: Valori di STD_ULOGIC.

2.2.6 *Pro e contro di VHDL*

Riassumendo, i vantaggi che VHDL introduce nell'ambito della progettazione di sistemi hardware sono i seguenti.

AD ALTO LIVELLO. La programmazione ad alto livello consente di arrivare più velocemente alla fase di simulazione. Questo a sua volta consente di anticipare la fase di debug. Una simulazione più veloce apre alla possibilità di provare più architetture diverse per uno stesso algoritmo, aumentando così la capacità di selezionare un'architettura ottimale.

INDIPENDENTE DALL'IMPLEMENTAZIONE. Il codice a livello RTL è indipendente dall'implementazione tecnologica utilizzata. La scelta della tecnologia target (sia essa ASIC o FPGA) può essere posticipata, essa infatti è selezionata solo in fase di sintesi. L'in-

6 transizione da '0' a '1'

dipendenza dall'implementazione minimizza i cambiamenti necessari al design in caso di passaggio a tecnologie o produttori alternativi.

FLESSIBILE. Un linguaggio standard come VHDL consente il riutilizzo di design da progetti precedenti o da fornitori commerciali di IP[7].

BASATO SUL LINGUAGGIO. L'utilizzo di un linguaggio di programmazione rende più facile la trasposizione di algoritmi. Il formato testo inoltre è più facile da gestire sia in fase di inserimento che in fase di modifica (potenzialmente da più sviluppatori in contemporanea).

La scelta di utilizzare VHDL può comportare anche dei rischi e degli svantaggi. I più rilevanti riguardano questi aspetti.

CURVA DI APPRENDIMENTO. Adottare VHDL comporta l'apprendimento di un nuovo linguaggio e di almeno due tool software (simulatore e tool di sintesi). La curva di apprendimento può essere ripida.

METODOLOGIA. Sebbene VHDL sia uno standard, non è accompagnato da una metodologia standard. La molteplicità di soluzioni da parte di produttori diversi potrebbe portare all'impiego di tool software da parte di fornitori differenti.

DESIGN ANALOGICO. VHDL è concepito per il design digitale. Il suo utilizzo per design misti o analogici non è ancora consolidato, nonostante esistano delle estensioni a tale scopo (es. VHDL-AMS).

STILE DI PROGRAMMAZIONE. Il successo del progetto dipende anche dallo stile di programmazione. La performance e l'area del design finale dipendono in larga misura dallo stile di programmazione impiegato per la descrizione del design iniziale.

PIANIFICAZIONE. VHDL richiede una fase importante di pianificazione e partizionamento del design, da effettuarsi prima dell'inizio della scrittura del codice sorgente.

7 Intellectual Property

3

LA VERIFICA DELL'HARDWARE

Il processo di sviluppo di un componente hardware (tipicamente un circuito logico integrato) consiste di due principali macroattività:

- Progettazione (*design*);
- Verifica (*verification*).

L'attività di progettazione consiste nella traduzione dei requisiti iniziali in un oggetto descritto da un linguaggio HDL. Questa attività, essendo principalmente un'attività di tipo creativo, espone al rischio che il prodotto finale presenti delle discrepanze (intenzionali o meno) dai requisiti prefissati. Si giustifica quindi l'esigenza di un'attività di verifica, volta ad individuare tali discrepanze (comunemente dette *bug*) e ad indirizzare verso una loro corretta risoluzione.

La natura fisica del prodotto (microcircuiti di silicio) e le caratteristiche del processo produttivo (lavorazioni nell'ordine dei nanometri) rendono impossibile qualsiasi forma di riparazione successiva alla produzione; a ragione di questo, l'attività di verifica di un componente hardware giunge a un pieno successo solo nei casi in cui riesca ad eliminare la totalità dei bug presenti nel design originario, prima che esso venga mandato in produzione.

3.1 SFIDE

Le fonderie di silicio continuano a ridurre le dimensioni fisiche delle strutture di silicio che possono essere realizzate in un circiuto integrato. Questo affinamento della capacità produttiva è accompagnato da un significativo miglioramento della capacità e delle performance dei circuiti. Il trend di evoluzione tecnologica dell'elettronica è stato abilmente colto nel 1965 da Gordon Moore (cofondatore di *Intel*, che l'ha racchiuso nell'ormai celebre legge empirica che porta il suo nome (figura 4) e che recita: *Il numero di transistor collocabili su di un singolo chip di silicio raddoppia ogni 18 mesi*.

A titolo di esempio, si consideri che un moderno processore Intel fabbricato con processo produttivo a 22 nm ha una frequenza di clock superiore di 4 mila volte al primo processore Intel 4004 del 1971; ciascun transistor utilizza mediamente 5 mila volte meno energia, e il prezzo di un singolo transistor è diminiuto di 50 mila volte. Intel ad oggi produce più di 5 miliardi di transistor al secondo.

La produttività della progettazione arranca alle spalle dell'aumento della densità consentito dai miglioramenti tecnologici. Il gap produttivo non può essere risolto semplicemente affidando il problema ad

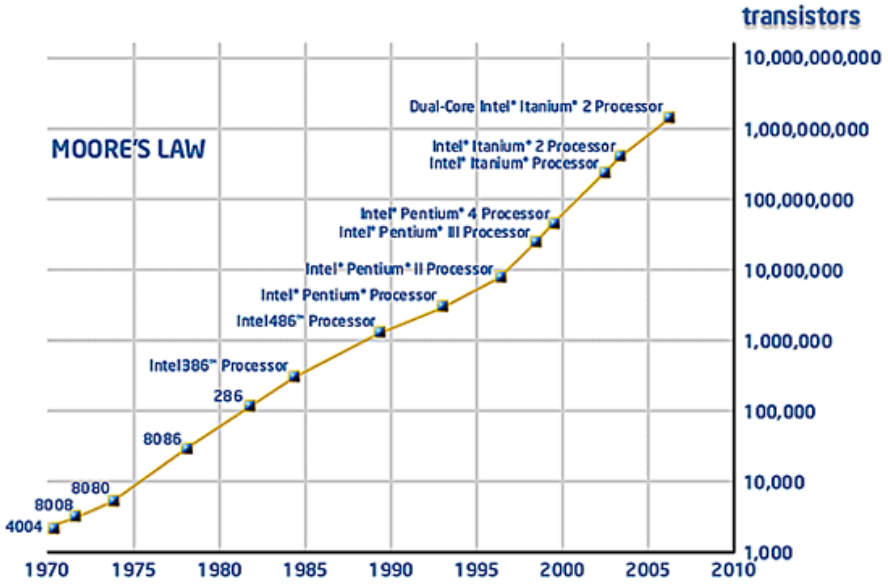

Figura 4: Legge di Moore: evoluzione nel tempo del numero dei transistor di un comune microprocessore commerciale.

un numero maggiore di ingegneri, come insegna la legge di Brooks.[1] L'industria ha risposto a questa sfida adottando strategie di progettazione orientate al riutilizzo. Tale concetto riduce non solo gli sforzi in fase di progettazione, ma riduce anche lo sforzo in fase di verifica di un nuovo design. Contemporaneamente all'esplosione della complessità di progettazione, si osserva nel tempo una drammatica riduzione nei requisiti di time-to-market (TTM) per i dispositivi elettronici. Questa tendenza è ulteriormente accentuata da un processo di migrazione relativo ai settori di applicazione della tecnologia elettronica: a partire da ambiti militari e industriali si va ora verso prodotti di tipo consumer, caratterizzati da un ciclo di vita sensibilmente più corto.

Si è stimato che tra il 40% e il 70% degli sforzi totali di sviluppo siano dovuti alla fase di verifica, e questo ha portato la community degli sviluppatori a lavorare per colmare il gap di produttività anche nella fase di verifica del design.

I miglioramenti nella produttivià sono stati resi possibili negli anni grazie all'introduzione di:

- nuove metodologie di verifica più adatte alla verifica di sistemi con crescente complessità;
- componenti di verifica riutilizzabili;
- linguaggi dedicati per la verifica dell'hardware, in grado di facilitare l'adozione dei nuovi paradigmi di verifica.

1 La legge di Brooks è un principio valido nell'ambito dell'ingegneria informatica che dice: Aggiungere forza lavoro ad un progetto software in ritardo, lo farà ritardare ancora di più. In altri termini, *Nine women can't make a baby in one month.*

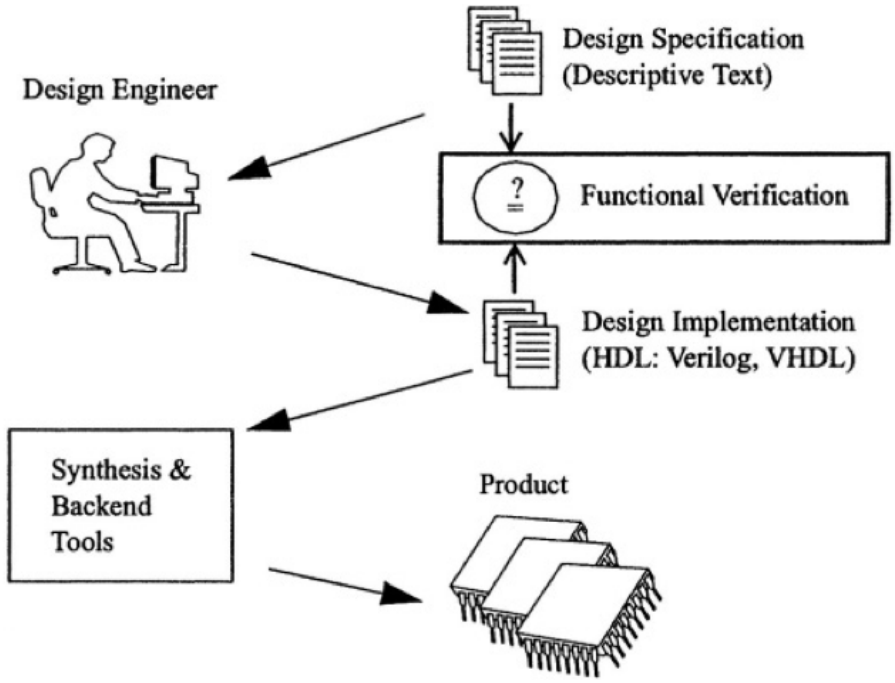

Figura 5: Verifica funzionale.

3.2 METODOLOGIE

L'attività di progettazione ha lo scopo di creare una implementazione hardware attraverso l'interpretazione delle specifiche di design. Questa implementazione iniziale è tradotta in una implementazione finale attraversando i livelli di astrazione dal più alto al più basso. Il passo più delicato e più soggetto ad errori è la scrittura del primo livello, ovvero la traduzione manuale delle specifiche in una implementazione. I passi successivi invece sono caratterizzati da un elevato automatismo e di conseguenza più raramente sono causa di errori.

L'obiettivo primario della verifica funzionale è verificare che l'implementazione iniziale sia funzionalmente equivalente alle specifiche di progettazione (figura 5).

L'attività fondamentale di verifica si basa quindi sull'immettere un flusso di informazioni nel dispositivo, e sul controllare il flusso di informazioni in uscita dal dispositivo. La verifica di questo tipo, basata sulla simulazione consiste nel portare il dispositivo in uno specifico stato e quindi controllare che la risposta sia corretta. In caso contrario si è rilevato un bug.

Il team di verifica, analizzando le specifiche del dispositivo e identificando le sue caratteristiche, stabilisce gli stati obiettivo da visitare. Questa scelta però non dà garanzia di visitare tutti gli stati che presentano bug: alcuni di essi potrebbero nascondersi in stati al di fuori degli obiettivi. Una verifica efficace dovrebbe non essere limitata agli obiettivi noti.

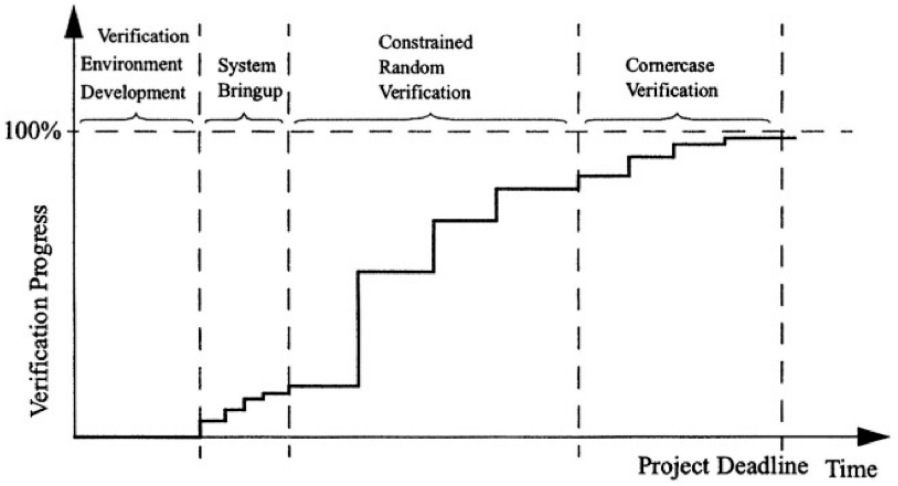

Figura 6: Progresso della verifica con directed test based verification.

3.2.1 *Directed testing: l'approccio tradizionale*

Nell'approccio tradizionale, che è di fatto un approccio a forza bruta, il verification engineer scrive dei test diretti per ciascun elemento presente nel piano di test. Questo approccio non presenta le caratteristiche auspicabili di automazione nè di completezza. Il directed-testing non presenta automazione perchè occorre del lavoro per determinare e visitare gli stati del dispositivo, e del lavoro per verificare se l'obiettivo sia stato raggiunto. Questa metologia inoltre non risulta completa poichè si ha una scarsa copertura degli scenari che non sono stati precedentemente definiti come obiettivo.

Un vantaggio (probabilmente uno dei pochi) di una verifica basata sul directed-testing è che il progetto avanza con proporzione lineare rispetto al tempo speso: i progressi avvengono un piccolo passo alla volta, come evidenziato in figura 6.

3.2.2 *Random testing: un migliore approccio*

La randomizzazione è una tecnica potente, in grado di migliorare drasticamente la qualità della verifica. Con la metodologia del random-testing si introduce un miglioramento per quanto riguarda l'aspetto di completezza. Si può osservare infatti che, premessa una quantità di tempo sufficiente, tutti gli scenari da verificare possono essere raggiunti generando sequenze random di transazioni; ovviamente occorre porre dei vincoli alla generazione random in modo da generare solo un sottoinsieme di transazioni valide. Utilizzando questa metodologia non è più necessario individuare, implementare e verificare ciascuno scenario. Inoltre la generazione random consente di raggiungere molti più scenari, e non solo quelli fissati a priori, migliorando la completezza della verifica.

3.2.3 *Coverage-driven verification: il migliore approccio*

Il random-testing porta a un significativo avanzamento nella completezza e nella produttività dell'attività di verifica. Nonostante ciò, i potenziali miglioramenti promessi da questo approccio possono essere difficili da concretizzare senza una chiara strategia per misurare il progresso della verifica. Con l'approccio coverage-driven si migliora la metodologia random-testing introducendo la nozione di coverage[2]. Il coverage assicura che tutti gli obiettivi siano raggiunti, fornendo inoltre informazioni sull'efficacia di ciascuna simulazione.

La generazione casuale dell'input riduce lo sforzo manuale, e in questo modo gli scenari più facili da raggiungere vengono verificati automaticamente; invece il controllo del coverage consente di rilevare gli scenari più difficili che richiedono un'attenzione manuale. Questo consente di non sprecare tempo in qualcosa che può fare l'automazione.

Le fasi di vita di un progetto di verifica che adotta la strategia coverage-driven sono le seguenti:

1. **Sviluppo del piano di verifica.** Il piano di verifica viene sviluppato in base alle specifiche del DUT e ai feedback del team di progettazione. L'obiettivo consiste nel produrre un piano di verifica completo, in quanto l'attività successiva dipenderà da esso.

2. **Implementazione dell'infrastruttura di verifica.** Si implementa l'infrastruttura necessaria per supportare il piano di verifica

3. **Attivazione dell'ambiente di verifica.** Vengono eseguite alcune istanze di simulazione per assicurarsi del corretto funzionamento dell'ambiente di verifica.

4. **Verifica random con vincoli.** Si mantiene il minimo numero indispensabile di vincoli, con l'obiettivo di coprire il maggior numero possibile di scenari. I risultati dal coverage collection sono utilizzati per controllare l'efficacia di ciascuna simulazione e per modificare i vincoli per guidare la generazione di scenari mancanti.

5. **Verifica dei corner case.** Vengono controllati i corner-case (*casi limite*) che richiedono modifiche specifiche all'infrastruttura di verifica, spesso con approccio simile ai test diretti.

2 copertura

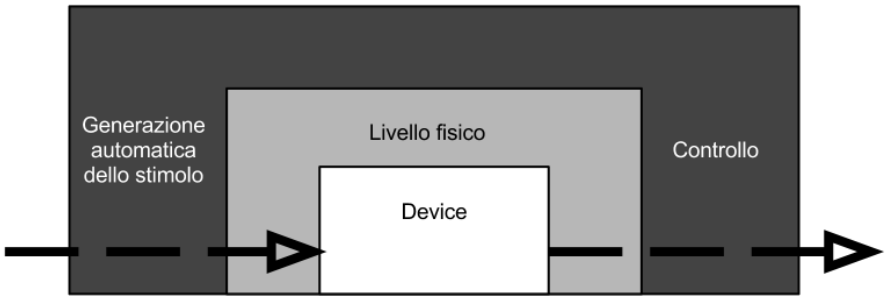

Figura 7: Ambiente per la verifica coverage-driven.

3.3 IL LINGUAGGIO E

Il linguaggio di programmazione *e* appartiene alla famiglia dei linguaggi HVL, *hardware verification languages*. Lo sviluppo di *e* fu cominciato nel 1992 in Israele da Yoav Hollander, per essere utilizzato con il software *Specman*. In seguito l'azienda di Hollander fu acquisita da *Cadence Design Systems*, la quale si occupa attualmente del linguaggio e dei tool ad esso associati.

3.3.1 *Peculiarità di e*

Le principali caratteristiche di *e* sono:

- generazion casuale (e vincolata) dello stimolo;
- definizione e raccolta di una metrica per la copertura funzionale (coverage);
- espressioni di linguaggio temporale per scrivere asserzioni;
- programmazione aspect-oriented;
- neutralità rispetto all'HVL utilizzato per il modello del dispositivo da verificare;
- alta riusabilità del codice.

Il linguaggio *e* utilizza un approccio aspect-oriented, il quale è una estensione della programmazione orientata agli oggetti. Questo approccio è una funzionalità chiave per consentire al programmatore di aggiungere facilmente nuove funzionalità a del codice già scritto, in modo non invasivo.

3.3.2 *Nozioni di base*

Il codice eseguibile di *e* deve essere racchiuso tra i due marcatori <′ e ′>. Tutto ciò che appare al di fuori di tali marcatori viene interpretato come commento.

e dispone di due tipi diversi di classi

STRUCT Le struct sono classi dinamiche, e tipicamente vengono utilizzate per modellare dati, ad esempio gli input e gli output del DUT.[3]

UNIT Le unit sono classi statiche, e trovano impiego nella creazione dei componenti permanenti della struttura del testbench.

Una classe, sia essa unit o struct, può contenere dei campi (*fields*), metodi (*methods*), porte (*ports*), e vincoli (*constraints*). I campi possono essere numerici, stringhe o anche riferimenti ad altre classi.

Ad esempio, il payload di un pacchetto contenente 8 bit di dati, un bit di parità e una lista di 4 bit di flag, può essere modellato nel seguente modo:

Codice 5: Esempio e: struct (1)

```
<'
struct data_payload {
        data : uint(bits:8);
        parity : bit;
        flag_bits[4]: list of bit;
};
'>
```

La keyword struct definisce l'oggetto chiamato *data_payload*. Una struct può contenere anche una struct precedentemente dichiarata; ad esempio:

Codice 6: Esempio e: struct (2)

```
<'
struct data_packet {
        header : uint(bits:4);
        payload : data_payload;
};
'>
```

All'interno di una struct possono anche essere definiti metodi. Ad esempio:

Codice 7: Esempio e: struct(3)

```
<'
struct data_payload {
        data : uint(bits:8);
        parity : bit;

        set_parity(p:bit) is {
                parity = p;
```

3 *Device Under Test*. Si tratta del dispositivo da verificare.

```
8         }
9 };
10 '>
```

3.3.3 *Regole di generazione*

In *e* il valore di ogni campo è generato in modo casuale di default. Per controllare la generazione casuale si utilizzano dei vincoli. I vincoli sono membri della classe, esattamente come gli altri campi, e vengono espressi in modo dichiarativo e non procedurale: non è importante l'ordine nel quale i vincoli (purchè si tratti di vincoli *hard*) sono scritti. La programmazione dichiarativa si basa sull'esistenza di un insieme di variabili e un insieme di regole (fisse) associate alle variabili. Il compilatore (o l'ambiente runtime) ha la responsabilità di usare regole di inferenza per instaurare relazioni tra le variabili e utilizzare le dichiarazione per eseguire operazioni durante l'esecuzione del programma. Lo stile di programmazione dichiarativo è utilizzato per la risoluzione di problemi di soddisfacimento vincoli. In *e* per esprimere tali vincoli viene utilizzata la keyword keep seguita da una espressione che deve poter essere valutata con risultato booleano TRUE o FALSE. Il risolutore di vincoli assegnerà ai campi un valore che garantisca un risultato TRUE a tale espressione. In *e* i vincoli sono processati dal compilatore e rivalutati in runtime ogni volta che viene generato il data object a cui sono applicati.

Codice 8: Esempio e: vincoli

```
1 <'
2 struct sbt_packet_s {
3         ...
4         keep (port_num > 0);
5         keep (addr != 128);
6 };
7 '>
```

3.4 *e* COME LINGUAGGIO DI VERIFICA

Il linguaggio *e* è stato progettato da zero per l'attività di verifica dell'hardware. Per questo motivo possiede delle caratteristiche che rispondono alle precise esigenze dell'attività di verifica. Le principali sono riassunte nella tabella 3.

3.4.1 *Cadence Specman*

Il linguaggio *e* viene distribuito da *Cadence* insieme al tool software *Specman*.

attività	esigenze	strumenti di *e*
Astrazione dalla simulazione	Nozione di concorrenza	Concorrenza e controllo dei thread
Generazione dello stimolo	Generazione casuale vincolata	Generazione casuale vincolata
	Variazione dello stimolo per requisiti specifici di verifica	Meccanismi di estensione
Pilotare lo stimolo	Interfacciarsi con il simulatore HDL	Interfaccia HDL, e-ports
	Associare verification objects con le istanze del modulo da simulare	Units
	Sincronizzarsi con il simulatore HDL	Eventi, espressioni temporali
	Passare dall'astrazione dei dati all'astrazione fisica	Packing
	Applicare lo stimolo	Metodi, TCM
Controllo dei risultati	Raccogliere i dati dal simulatore HDL	Interfaccia HDL, e-ports
	Passare dall'astrazione fisica all'astrazione dei dati	Unpacking
	Controllo del protocollo temporale	Espressioni temporali
Coverage	Coverage	Coverage

Tabella 3: Esigenze dell'attività di verifica e strumenti di *e*.

Specman è un tool per l'esecuzione, la compilazione e il debugging di ambienti di verifica scritti in linguaggio *e*. Poichè *Specman* non comprende un simulatore HDL, occorre interfacciarlo con un simulatore esterno compatibile. Di frequente tale simulatore è *NC-Sim*, prodotto dalla stessa *Cadence*. L'impiego dei due prodotti software della stessa casa garantisce un ottimo livello di integrazione e performance.

Specman mette a disposizione un ambiente di esecuzione runtime che include un risolutore di vincoli, impiegato per la generazione casuale, e una infrastruttura di comunicazione con il simulatore HDL. Lo sviluppatore può concentrarsi solo sul lavoro di scrittura dei componenti che variano a seconda del design da sottoporre a verifica.

Come da figura 9, lo sviluppatore dovrà definire in linguaggio *e* alcuni moduli di base per ciascun progetto di verifica, e dei vincoli aggiuntivi a seconda del tipo test da effettuare. Il vantaggio più evidente di questo approccio consiste nella possibilità di riutilizzare la stessa architettura per effettuare test di tipo diverso.

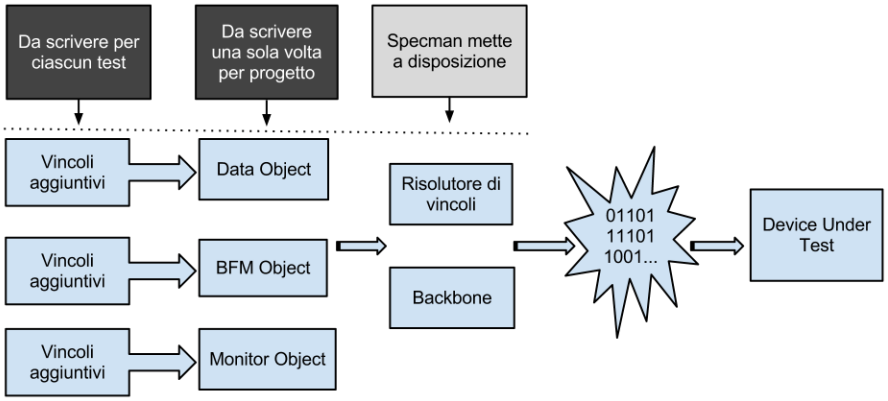

Figura 8: Specman.

3.4.2 *Struttura dell'ambiente di verifica*

Un ambiente di verifica scritto in linguaggio *e* si compone di alcuni oggetti fondamentali, riportati anche in figura 9. Tutti gli oggetti seguenti vengon scritti dallo sviluppatore per ogni progetto di verifica. In termini di linguaggio *e*, tutti questi componenti risultano essere delle *unit*.

DUT DUT è l'acronimo di *Device Under Test*. Si tratta del modello HDL del componente da sottoporre a verifica.

ENVIRONMENT L'enviroment rappresenta il livello superiore della gerarchia. Al suo interno vengono collocati e connessi gli oggetti necessari al funzionamento dell'ambiente di verifica.

AGENT Un agent rappresenta un'entità autonoma in grado di interagire con il DUT e con altri agent. Negli ambienti di verifica più semplici si ha un singolo agent.

DRIVER Il driver è il componente che genera le transazioni da sottoporre al DUT.

BFM Un acronimo per *Bus Functional Model*. Il BFM ha il compito di tradurre le informazioni generate ad alto livello dal driver in informazioni comprensibili dal DUT. Si tratta di una implementazione del protocollo di comunicazione del DUT.

MONITOR L'oggetto monitor ha il compito di raccogliere dati senze modificare il comportamento del DUT. Tipicamente un ambiente di verifica standard dispone di due tipi di monitor: input monitor e output monitor. I primi hanno il compito di raccogliere il flusso di dati nella direzione dal BFM al DUT. I secondi invece raccolgono i dati in uscita dal DUT, ovvero le risposte del dispositivo alle sollecitazioni.

CHECKER Il checker è quell'oggetto che ha il compito di valutare la correttezza del comportamento del dispositivo, in riferimento ai dati di input immessi.

SIGNAL MAP La signal map è un componente condiviso tra gli altri oggetti e viene utilizzata per associare gli identificatori logici dei segnali (in linguaggio *e*) ai segnali del modello HDL.

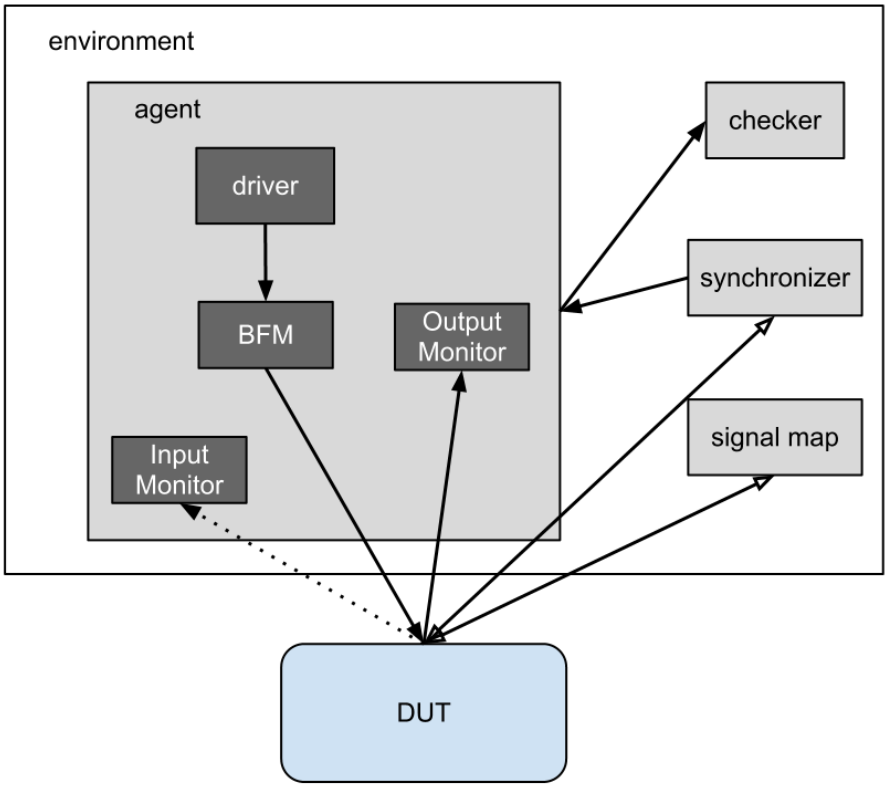

Figura 9: Struttura dell'ambiente di verifica.

3.4.3 *Interazione con il simulatore HDL*

Per accedere ai segnali del DUT nel simulatore in un programma in linguaggio *e* si utilizza il costrutto *port*. Una porta va dichiarata all'interno di una *unit* e può essere utilizzata per accedere ai valori della rete RTL tramite l'operatore *$*.

Un esempio di dichiarazione di una porta è il seguente:

Codice 9: Esempio e: port

```
1 <'
2 unit sbt_monitor {
3     ...
4     grant_p : in simple_port of bit is instance;
5     keep grant_p.hdl_path() == "gnt";
6     ...
7     out(grant_p$);
8 };
9 '>
```

Il vincolo posto sul metodo *hdl_path()* consente di specificare a quale segnale è associata tale porta (nell'esempio si tratta del segnale che nel modello HDL è identificato dal nome *gnt*).

Nella riga 7 si vede l'utilizzo dell'operatore *$* per visualizzare a schermo il valore del segnale tramite il metodo *out*.

Parte II

PROGETTAZIONE E IMPLEMENTAZIONE DI UN AMBIENTE DI VERIFICA PER L'ALU DEL MICROPROCESSORE LEON3

4

L'ARCHITETTURA SPARC E IL PROCESSORE LEON3

4.1 ARCHITETTURA SPARC

L'acronimo SPARC significa *Scalable Processor ARChitecture*. SPARC è un'architettura per microprocessori derivata da una impostazione di tipo big-endian[1] e RISC.[2]

SPARC iniziò ad essere sviluppato da *Sun Microsystems* (recentemente acquisita da *Oracle Corporation*) e venne introdotto nel mercato a metà del 1987. Nel 1989 venne istituita la *SPARC International, Inc.*, una organizzazione dedita alla promozione dell'architettura SPARC, alla gestione del trademark e ai test di conformità. L'architettura SPARC viene concessa in licenza a numerosi produttori di hardware.

La versione dell'architettura SPARC alla quale si fa riferimento in questo documento è la 8, rilasciata nel 1990.

4.1.1 *Caratteristiche SPARC*

SPARC include le seguenti caratteristiche principali:

- **Spazio di indirizzi lineare a 32 bit.**
- **Ridotto numero di formati di istruzione e semplicità degli stessi.** Tutte le istruzioni sono ampie 32 bit e sono allineate a 32 bit in memoria. Ci sono solo tre tipi di formati di istruzione, caratterizzati da un posizionamento uniforme dei campi opcode e indirizzi di registro. Le uniche istruzioni che possono accedere alla memoria e all'I/O sono istruzioni di tipo load e store.
- **Pochi modi di indirizzamento.** Un indirizzo di memoria è dato da *registro* + *registro* oppure da *registro* + *immediato*.
- **Accesso ai registri triadico.** La maggior parte delle istruzioni opera su due operandi registro (oppure un registro e una costante), e memorizza il risultato in un terzo registro.

1 La memorizzazione inizia dal byte più significativo per finire col meno significativo.

2 L'acronimo RISC (dall'inglese *reduced instruction set computer*) indica una filosofia di progettazione di architetture per microprocessori che predilige lo sviluppo di un'architettura semplice e lineare. Questa semplicità di progettazione permette di realizzare microprocessori in grado di eseguire il set di istruzioni in tempi minori rispetto a una classica architettura CISC.

- **Un register file ampio e a finestre.** In ciascun istante, un programma vede 8 registri globali più una finestra di 24 registri. I registri della finestra possono essere descritti come una cache per gli argomenti della procedura, i valori locali e gli indirizzi di ritorno.
- **Registro floating-point separato.** Tale registro è configurabile via software in 32 registri a precizione singola (32 bit), oppure 16 registri a precisione doppia (64 bit) oppure ancora in 8 registri a precisione quadrupla (128 bit).
- **Trasferimenti di controllo ritardati.** Il processore carica sempre l'istruzione successiva dopo una istruzione di trasferimento di controllo ritardato. L'effettiva esecuzione dell'istruzione dipende dal valore dell'apposito bit di annullamento posto nell'istruzione di trasferimento di controllo.
- **Trap handler veloci.** Le trap[3] sono vettorizzate attraverso una tabella e provocano l'allocazione di una nuova finestra nel register file.
- **Istruzioni in versione *tagged*.** Le istruzioni add/subtract in versione *tagged* assuono che i due bit meno significativi degli operandi siano bit di tag.
- **Sincronizzazione multi-processore nelle istruzioni.** Una istruzione esegue una operazione atomica *read-then-set*; un'altra istruzione esegue una operazione atomica *exchange-register-with-memory*.
- **Coprocessore.** L'architettura definisce in modo chiaro un set di istruzioni per un coprocessore, in aggiunta al set di istruzioni a virgola mobile.

Un processore SPARC è formato da tre componenti logiche distinte:

- un'unità per aritmetica intera, *integer unit*, (IU);
- un'unità per aritmetica a virgola mobile, *floating-point unit*, (FPU);
- un coprocessore (CP), opzionale.

Ciascuna unità dispone al suo interno di un proprio set di registri.

Questa organizzazione consente una massima concorrenza tra l'esecuzione di istruzioni intere, a virgola mobile e del coprocessore. Tutti i registri (con l'unica possibile eccezione del coprocessore) sono a 32 bit. Gli operandi delle istruzioni sono tipicamente registri singoli, coppie di registri o quadruple di registri.

3 interruzioni software

4.1.2 *La Integer Unit*

La IU contiene i registri *general-purpose*, d'ora innanzi indicati con la lettera *r*, e controlla il funzionamento generale del processore. La IU esegue le istruzioni aritmetiche su interi e calcola gli indirizzi di memoria per i *load* e *store*. Memorizza inoltre il program counter (PC) e controlla l'esecuzione delle istruzioni da parte della FPU e del CP.

Un'implementazione della IU può contenere da 40 a 520 registri general-purpose a 32 bit. Questi sono organizzati in 8 registri *global*, più uno stack circolare di finestre, ciascuna da 16 registri. Le finestre possono essere da 2 a 32. Il numero di finestre può variare a seconda dell'implementazione e di conseguenza anche il numero di registri totale.

In ciascun istante, una istruzione può accedere agli 8 registri globali e ad una finestra. I 24 registri presenti in tale finestra sono così suddivisi:

- 8 registri *in*;
- 8 registri *local*;
- 8 registri *out*.

I registri di in e out di una certa finestra si sovrappongono ai registri della finestra che la precede e della finestra che la segue, come si può vedere in figura 10.

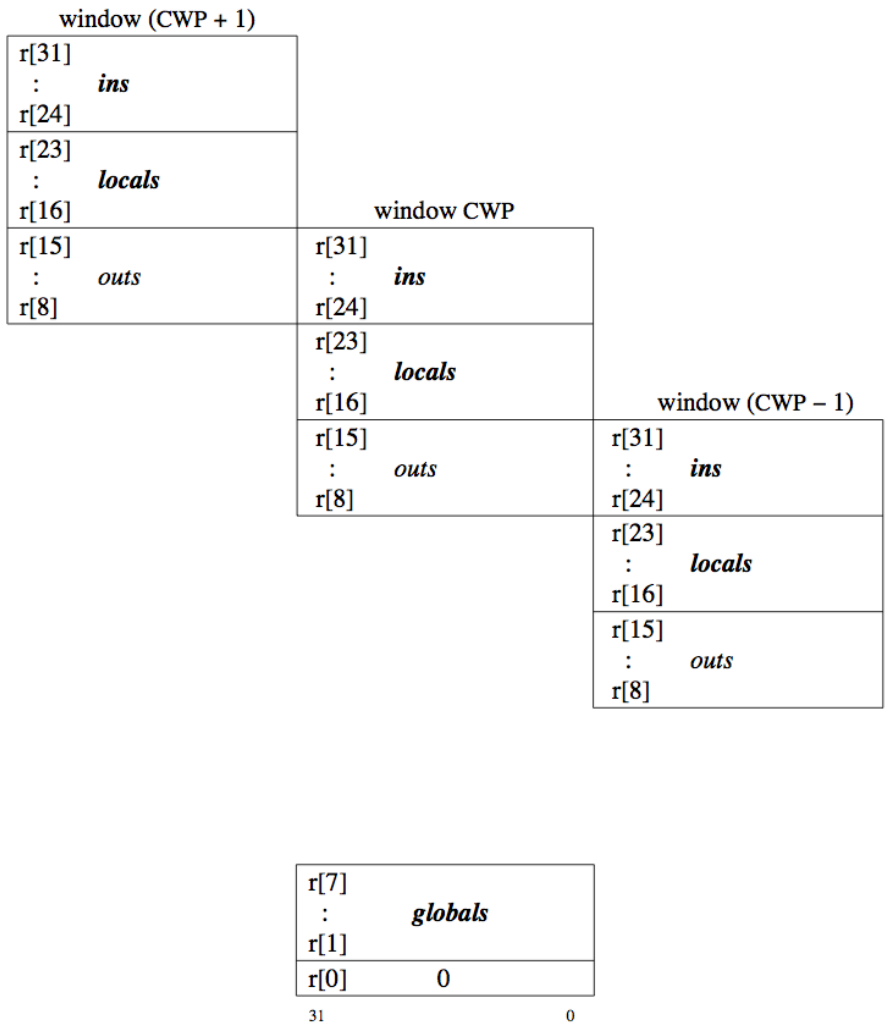

Figura 10: Sovrapposizione dei registri a finestra.

La finestra corrente in uso è specificata dal CWP (*current window pointer*), che è un campo nel registro di stato del processore (PSR). L'overflow e l'underflow delle finestre sono rilevati attraverso il registro WIM (*window invalid mask*).

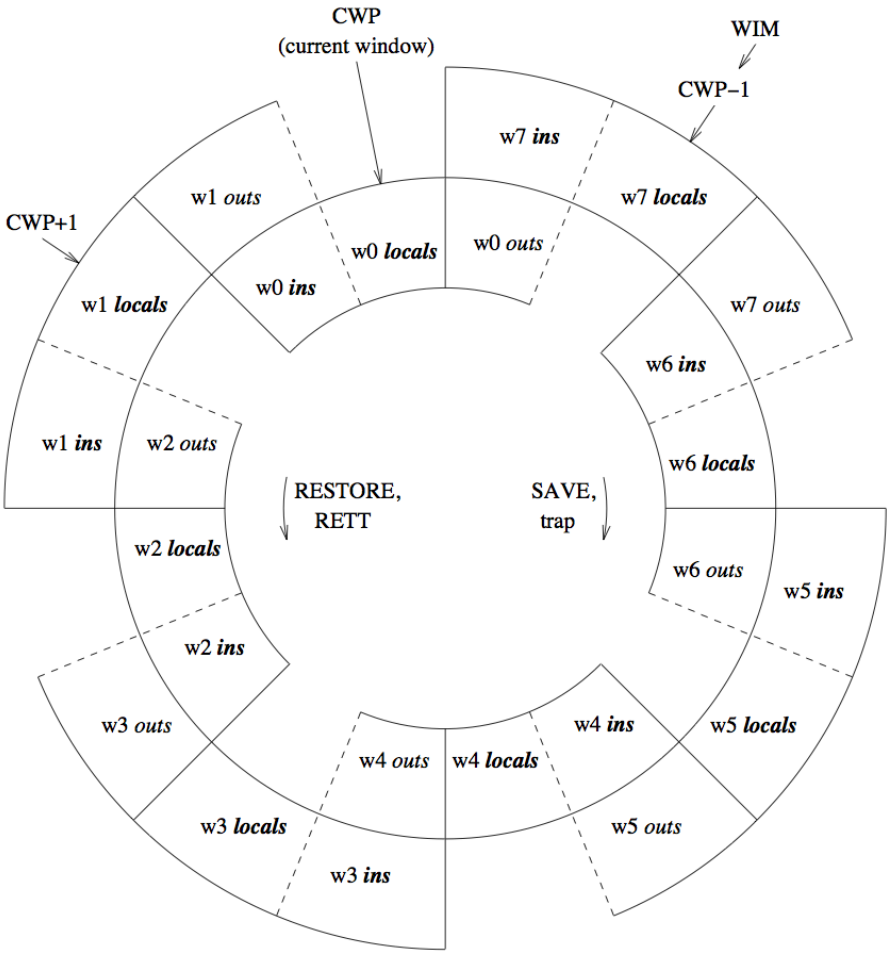

Figura 11: Organizzazione e sovrapposizione dei registri a finestra circolare.

4.1.3 *Formato delle istruzioni*

Le istruzioni sono codificate in tre formati da 32 bit e possono essere suddivise in sei categorie generali.

Un'istruzione è letta dalla memoria all'indirizzo dato dal program counter (PC). Viene quindi eseguita, oppure ignorata se l'istruzione precedente era un branch con annullamento. Una istruzione può anche generare una trap in caso vengano rilevate condizioni eccezionali, causate dall'istruzione stessa (*precise trap*), un'istruzione precedente (*deferred trap*), un interrupt esterno (*interrupting trap*), o una richiesta esterna di reset.

Accanto al registro PC è presente un registro contenente il program counter successivo (nPC). Questi due registri consentono un modello di esecuzione di tipo delayed-branch.

Se un'istruzione non genera una trap, il program counter successivo viene copiato nel PC e il program counter successivo viene incrementato di 4. Se invece l'istruzione è un'istruzione di trasferimento del controllo il processore scrive l'indirizzo di target nel nuovo

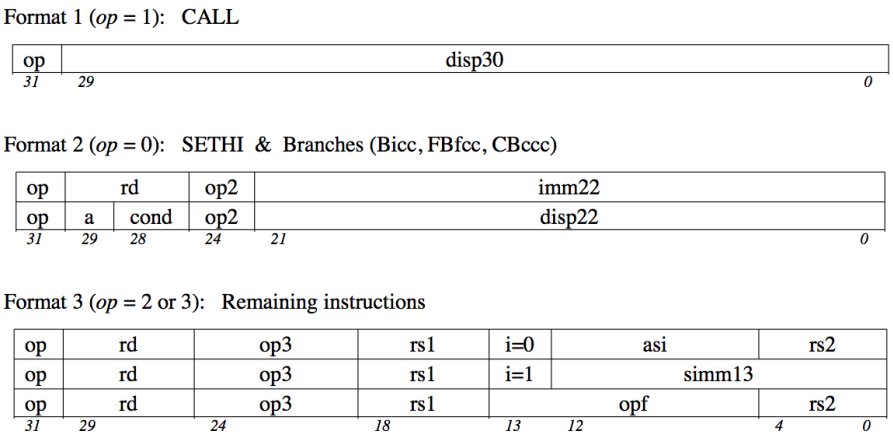

Figura 12: Formato delle istruzioni.

FORMATO	OP	DESCRIZIONE
1	1	CALL
2	0	Bicc, FBfcc, CBccc, SETHI
3	3	istruzioni di memoria
3	2	aritmetiche, logiche, shift e restanti

Tabella 4: Codifica per op.

program counter.

I campi delle istruzioni sono così specificati:

OP / OP2 Questi campi da 2 bit e 3 bit rispettivamente codificano i 3 principali formati di istruzione e le istruzioni di formato 2.

RD Questo campo a 5 bit rappresenta l'indirizzo del registro di destinazione (o sorgente). Il registro può essere *r*, della FPU o del coprocessore.

A Il bit *a* in una istruzione di branch annulla l'esecuzione dell'istruzione seguente se il salto è condizionale e non viene effettuato, oppure se è non condizionale e viene eseguito.

COND Questo campo da 4 bit seleziona la condizione sulla quale effettuare il test per un'istruzione di branch.

IMM22 Questo campo a 22 bit rappresenta una costante che verrà posta nella porzione superiore di un registro dall'istruzione SETHI.

DISP22 / DISP30 *disp22* rappresenta l'offset tra il PC di arrivo e il PC corrente in un branch (i valori vanno estesi in segno e sono allineati alla word). *disp30* è analogo per le call.

FORMATO	ISTRUZIONI
0	UNIMP
1	non implementata
2	Bicc
3	non implementata
4	SETHI
5	non implementata
6	FBfcc
7	CBccc

Tabella 5: Codifica per op2.

OP3 Questo campo di 6 bit, insieme ad 1 bit di op, codifica le istruzioni di formato 3.

I Il bit *i* seleziona il secondo operando ALU per le istruzioni di aritmetica intera e le istruzioni load/store. Se *i* = 0 l'operando è *r[rs2]*. Se *i* = *1* l'operando è *simm13*, esteso in segno da 13 a 32 bit.

ASI Questo campo da 8 bit è l'identificatore dello spazio di indirizzi utilizzato da istruzioni load/store.

RS1 Questo campo da 5 bit è l'indirizzo del registro *r*, della FPU o del coprocessore, relativo al primo operando sorgente.

RS2 Questo campo a 5 bit è l'indirizzo del registro *r*, della FPU o del coprocessore, relativo al secondo operando sorgente.

SIMM13 Questo campo da 13 bit, una volta esteso a 32 bit mantenendo il segno, è un valore immediato usato come secondo operando ALU per una operazione aritmetica o load/store quando *i* = 1.

OPF Questo campo da 9 bit codifica un'operazione a virgola mobile oppure un'operazione del coprocessore.

4.1.4 *Registri di stato*

- *PSR.* Processor State Register. Registro a 32 bit che contiene diversi campi di controllo del processore, oltre a un insieme di informazioni sullo stato. Tra i campi importanti si ricordano quello relativo al *CWP (current window pointer)* e quello relativo ai bit degli *integer condition codes*:
  - *n* indica se il risultato dell'operazione ALU è negativo;

  - z indica se il risultato dell'operazione ALU è zero;
  - v indica se l'operazione ALU ha causato overflow;
  - c indica se l'operazione ALU ha eseguito un riporto.

- *WIM.* Window Invalid Mask. Registro a 32 bit che contiene l'informazione relativa alle finestre dei registri di sistema. Un bit attivo nella posizione n del registro WIM indica che la finestra puntata da $CWP = n$ non è valida.
- *TBR.* Trap Base Register. Registro a 32 bit che controlla gli indirizzi di branch relativi all'occorenza di una trap.
- *Y.* Multiply/Divide Register. Registro a 32 bit che contiene la word più significativa di una moltiplicazione intera a doppia precisione.

4.1.5 *Istruzioni*

Si propone ora un elenco delle istruzioni selezionate ed esaminate per l'attività di verifica funzionale.

4.1.5.1 *SETHI*

opcode	***op***	***op2***	***operation***
SETHI	00	100	Set High-Order 22 bits

Format (2):

Figura 13: Formato dell'istruzione SETHI.

L'istruzione SETHI azzera i 10 bit meno significativi di *r[rd]* e rimpiazza i suoi 22 bit più significativi con il valore del campo *imm22*.

L'istruzione SETHI non modifica i bit *icc*.

Un'istruzione SETHI con $rd = 0$ e $imm22 = 0$ è utilizzata e definita come istruzione NOP.

4.1.5.2 *NOP*

opcode	*op*	*op2*	*operation*
NOP	00	100	No Operation

Format (2):

Figura 14: Formato dell'istruzione NOP.

L'istruzione NOP non modifica alcuno stato visibile della CPU, eccetto i registri PC e nPC.

Si noti che l'istruzione NOP è un caso speciale di istruzione SETHI con *imm22* = *0* e *rd* = *0*.

4.1.5.3 *Istruzioni logiche*

opcode	*op3*	*operation*
AND	000001	And
ANDcc	010001	And and modify icc
ANDN	000101	And Not
ANDNcc	010101	And Not and modify icc
OR	000010	Inclusive Or
ORcc	010010	Inclusive Or and modify icc
ORN	000110	Inclusive Or Not
ORNcc	010110	Inclusive Or Not and modify icc
XOR	000011	Exclusive Or
XORcc	010011	Exclusive Or and modify icc
XNOR	000111	Exclusive Nor
XNORcc	010111	Exclusive Nor and modify icc

Format (3):

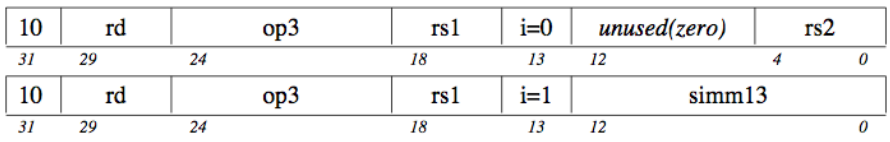

Figura 15: Formato delle istruzioni logiche.

Queste istruzioni implementano le operazioni logiche sui bit. Esse calcolano *r[rs1]* **operation** *r[rs2]* se il campo *[*i] è zero, altrimenti calcolano *r[rs1]* **operation** *sign_ext[simm13]*. Il risultato viene scritto in *r[rd]*.

Le istruzioni ANDcc, ANDNcc, ORcc, ORNcc, XORcc, e XNORcc modificano i bit *icc*.

Le istruzioni ANDN, ANDNcc, ORN, e ORNcc calcolano la negazione del secondo operando prima di applicare l'operazione principale (AND oppure OR).

4.1.5.4 *Istruzioni di shift*

opcode	*op3*	*operation*
SLL	100101	Shift Left Logical
SRL	100110	Shift Right Logical
SRA	100111	Shift Right Arithmetic

Format (3):

10	rd	op3	rs1	i=0	*unused(zero)*	rs2
31	*29*	*24*	*18*	*13*	*12*	*4* *0*
10	rd	op3	rs1	i=1	*unused(zero)*	shcnt
31	*29*	*24*	*18*	*13*	*12*	*4* *0*

Figura 16: Formato delle istruzioni di shift.

Il contatore delle posizioni di shift per queste istruzioni corrisponde ai 5 bit meno significativi di *r[rs2]* se il campo *i* è zero, oppure il valore del campo *shcnt* se *i* vale uno.

Quando *i* è 0, i 27 bit più significativi del valore in *r[rs2]* vengono ignorati. Quando i è 1, i bit dal 5 al 12 dell'istruzione di shift sono riservati e dovrebbero avere valore zero.

SLL trasla a sinistra il valore di *r[rs1]* di un numero di posizioni pari al contatore di posizioni.

SRL e SRA traslano a destra il valore di *r[rs1]* di un numero di posizioni pari al contatore di posizioni.

SLL e SRL sostituiscono le posizioni liberate con zeri, mentre SRA riempie le posizioni vuote con il bit più significativo di *r[rs1]* mantenendo così il segno. Se il contatore di posizioni è zero non avviene alcuno shift.

Tutte queste istruzioni scrivono il risultato in *r[rd]*.

Queste istruzioni non modificano i bit *icc*.

4.1.5.5 *ADD*

opcode	*op3*	*operation*
ADD	000000	Add
ADDcc	010000	Add and modify icc
ADDX	001000	Add with Carry
ADDXcc	011000	Add with Carry and modify icc

Format (3):

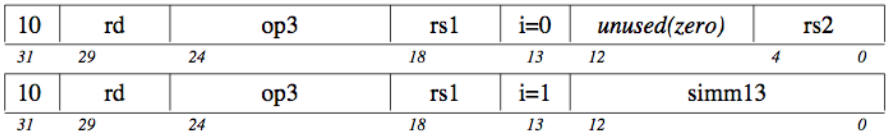

Figura 17: Formato delle istruzioni di ADD.

ADD e ADDcc calcolano *r[rs1] + r[rs2]* se il campo *i* è zero, oppure *r[rs1] + sign_ext(simm13)* se il campo *i* vale uno. La somma viene scritta in *r[rd]*.

ADDX e ADDXcc (ADD eXtended) in aggiunta sommano anche il bit del riporto (*c*), ovvero calcolano *r[rs1] + r[rs2] + c* oppure *r[rs1] + sign_ext(simm13) + c*.

ADDcc e ADDXcc modificano i bit *icc*. Nell'addizione si verifica overflow se entrambi gli operandi hanno lo stesso segno e il segno della somma è differente.

4.1.5.6 *SUB*

opcode	*op3*	*operation*
SUB	000100	Subtract
SUBcc	010100	Subtract and modify icc
SUBX	001100	Subtract with Carry
SUBXcc	011100	Subtract with Carry and modify icc

Format (3):

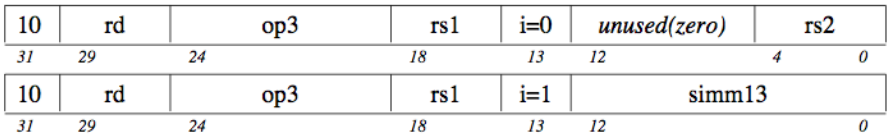

Figura 18: Formato delle istruzioni di SUB.

Queste istruzioni calcolano *r[rs1] - r[rs2]* se il campo *i* vale zero, oppure *r[rs1] - sign_ext(simm13)* se il campo *i* vale uno. Il risultato viene scritto in *r[rd]*.

SUBX e SUBXcc (SUBtract eXtended) in aggiunta sottraggono anche il bit del riporto (*c*), ovvero calcolano *r[rs1] - r[rs2]* oppure *r[rs1] - sign_ext(simm13) - c*. Il risultato viene scritto in *r[rd]*.

SUBcc e SUBXcc modificano i bit *icc*. Nella sottrazione si verifica overflow se gli operandi hanno segni diversi e il segno della differenza è diverso dal segno di *r[rs1]*.

4.1.5.7 *SAVE e RESTORE*

opcode	*op3*	*operation*
SAVE	111100	Save caller's window
RESTORE	111101	Restore caller's window

Format (3):

10	rd	op3	rs1	i=0	*unused(zero)*	rs2
31	*29*	*24*	*18*	*13*	*12* *4*	*0*

10	rd	op3	rs1	i=1	simm13
31	*29*	*24*	*18*	*13*	*12* *0*

Figura 19: Formato delle istruzioni SAVE e RESTORE.

L'istruzione SAVE sottrae una unità dal CWP (in modulo NWINDOWS) e confronta questo valore(new_CWP) con il contenuto del registro Window Invalid Mask (WIM). Se il bit del registro WIM corrispondente a new_CWP è 1, allora viene generata una trap di tipo *window_overflow*. Se invece il bit è 0, non viene generata alcuna trap e il valore new_CWP viene scritto in CWP. Questo fa si che la finestra corrente diventi la finestra CWP-1, salvando così la finestra del chiamante.

L'istruzione RESTORE aggiunge una unità al CWP (in modulo NWINDOWS) e confronta questo valore(new_CWP) con il contenuto del registro Window Invalid Mask (WIM). Se il bit del registro WIM corrispondente a new_CWP è 1, allora viene generata una trap di tipo *window_underflow*. Se invece il bit è 0, non viene generata alcuna trap e il valore new_CWP viene scritto in CWP. Questo fa si che la finestra corrente diventi la finestra CWP+1, ripristinando così la finestra del chiamante.

Inoltre, se e solo se non viene generata una trap, le istruzioni SAVE e RESTORE si comportano come normali istruzioni ADD, eccetto per il fatto che gli operandi (*r[rs1]* e *r[rs1]*) vengono letti dalla vecchia finestra (ovvero dalla finestra indirizzata dal CWP originario), mentre la somma è scritta in *r[rd]* della nuova finestra (ovvero la finestra inidirizzata dal new_CWP).

L'aritmetica su CWP viene sempre effettuata in modulo NWINDOWS.

4.1.5.8 *Branch su integer condition code*

opcode	*cond*	*operation*	*icc test*
BA	1000	Branch Always	1
BN	0000	Branch Never	0
BNE	1001	Branch on Not Equal	**not** Z
BE	0001	Branch on Equal	Z
BG	1010	Branch on Greater	**not** (Z or (N **xor** V))
BLE	0010	Branch on Less or Equal	Z **or** (N **xor** V)
BGE	1011	Branch on Greater or Equal	**not** (N **xor** V)
BL	0011	Branch on Less	N **xor** V
BGU	1100	Branch on Greater Unsigned	**not** (C **or** Z)
BLEU	0100	Branch on Less or Equal Unsigned	(C **or** Z)
BCC	1101	Branch on Carry Clear (Greater than or Equal, Unsigned)	**not** C
BCS	0101	Branch on Carry Set (Less than, Unsigned)	C
BPOS	1110	Branch on Positive	**not** N
BNEG	0110	Branch on Negative	N
BVC	1111	Branch on Overflow Clear	**not** V
BVS	0111	Branch on Overflow Set	V

Format (2):

Figura 20: Formato delle istruzioni di branch.

- **Branch non condizionali (BA, BN).** Se il campo *annul* è 0, l'istruzione BN (Branch Never) si comporta come una NOP. Se il campo *annul* è 1, l'istruzione seguente (chiamata *delay instruction*) è annullata (non viene eseguita). In nessun caso avviene un trasferimento di controllo.

- **Branch condizionali.** Le istruzioni di branch condizionali (tutte, eccetto BA e BN) valutano il valore dei bit *icc* secondo il contenuto del campo *cond* presente nell'istruzione. Questa valutazione produce un risultato di tipo vero o falso. Se vero, il branch viene intrapreso, cioè l'istruzione provoca un trasferimento di controllo ritardato verso l'indirizzo *PC + (4 x sign_ext(disp22))*. Se falso, il branch non viene intrapreso.

  Se un branch condizionale viene effettuato, l'istruzione di delay è sempre eseguita, a prescindere dal valore del campo *annul*. Se un branch condizionale non viene effettuato e *annul* vale 1, l'istruzione di delay viene annullata.

Nota: il campo *annul* ha effetti differenti a seconda che il branch sia condizionale o meno.

4.1.5.9 *CALL*

opcode	*op*	*operation*
CALL	01	Call and Link

Format (1):

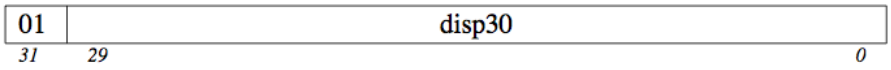

Figura 21: Formato dell'istruzione CALL.

L'istruzione CALL provoca un trasferimento di controllo non condizionale e ritardato verso l'indirizzo *PC + (4 x disp30)*. Poichè la dimensione del salto *disp30* è codificata a 30 bit, l'indirizzo di arrivo può essere arbitrariamente distante.

L'istruzione CALL inoltre scrive il valore di *PC*, che contiene l'indirizzo dell'istruzione CALL, nel registro *r[15]*.

4.1.5.10 *JMPL*

opcode	*op3*	*operation*
JMPL	111000	Jump and Link

Format (3):

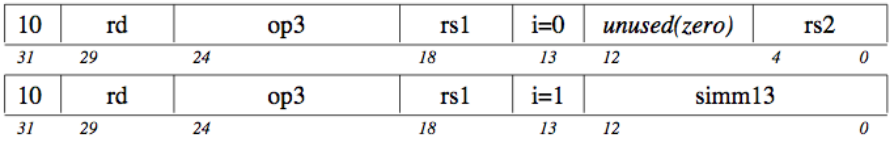

Figura 22: Formato dell'istruzione JMPL.

L'istruzione JMPL provoca un trasferimento di controllo non condizionale all'indirizzo dato da *r[rs1] + r[rs2]* se il campo *i* è zero, oppure *r[rs1] + sign_ext(simm13)* se *i* vale uno.

L'istruzione JMPL copia il valore di PC, che contiene l'indirizzo dell'istruzione JMPL, nel registro *r[rd]*.

Se uno dei due bit meno significativi (o entrambi) dell'indirizzo a cui saltare è diverso da zero viene generata una trap di tipo *mem_address_not_aligned*.

4.2 IL PROCESSORE LEON3

Leon3 è un processore a 32 bit conforme all'architettura SPARC v8. Specificatamente progettato per applicazioni embedded, Leon3 è ca-

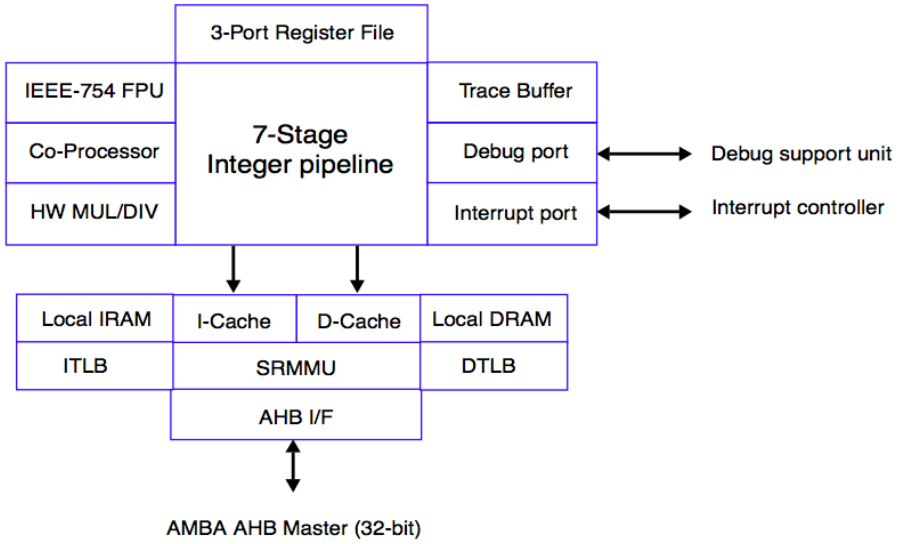

Figura 23: Struttura del processore Leon3.

ratterizzato da alte prestazioni unite ad una bassa complessità e ad un basso consumo energetico.

Il core Leon3 (figura 23) ha le seguenti caratteristiche principali: pipeline a 7 stadi con architettura Harvard,[4] memorie cache separate per dati e istruzioni, moltiplicatore e divisore hardware, supporto per il debug, estensioni multi-processore.

La IU del processore Leon3 implementa l'unità intera prevista dall'architettura SPARC v8. L'implementazione è orientata verso alte performance e bassa complessità. La IU è dotata delle seguenti caratteristiche:

- pipeline da 7 stadi;
- interfaccia separata per data cache e cache istruzioni;
- supporto per un numero variabile di finestre di registri (da 2 a 32);
- moltiplicatore hardware con molitplicatore opzionale 16x16 bit (MAC) e accumulatore 40 bit;
- divisore a radice 2;
- branch prediction statica;
- gestione delle trap vettorizzata a singolo vettore.

4 L'architettura Harvard è un tipo di architettura hardware per computer in cui vi è separazione tra la memoria contenente i dati e quella contenente le istruzioni.

4.2.1 *Pipeline*

L'IU del processore Leon3 utilizza una singola pipeline per l'esecuzione delle istruzioni (figura 25), dotata di 7 stadi:

1. **FE (Instruction Fetch)** Se la cache delle istruzioni è abilitata, l'istruzione è prelevata dalla cache. In caso contrario l'operazione di prelievo è inoltrata al bus AHB. L'istruzione è valida alla fine di questo stadio e viene memorizzata all'interno della IU.
2. **DE (Decode)** L'istruzione è decodificata e gli indirizzi target di CALL e Branch sono generati.
3. **RA (Register Access)** Gli operandi sono letti dal register file oppure da bypass interni.
4. **EX (Execute)** Le operazioni ALU, logiche e di shift vengono eseguite. Per le operazioni sulla memoria (esempio LOAD) e per JMPL/RETT viene generato l'indirizzo.
5. **ME (Memory)** La cache dei dati viene letta o scritta.
6. **XC (Exception)** Le trap e gli interrupt vengono risolti. Per le letture da cache, i dati vengono allineati come appropriato.
7. **WR (Write)** Il risultato dell'operazione ALU, logica o di shift viene scritto nel register file.

4.2.1.1 *Branch prediction*

Il processore Leon3 supporta un meccanismo di branch prediction statico, che può essere opzionalmente abilitato. Il branch prediction riduce la penalizzazione nella quale si incorre per salti preceduti da una istruzione che modifica i bit *icc*. Il predittore utilizza una strategia *branch-always* e inizia a fare il fetch dell'istruzione utilizzando l'indirizzo di salto. Se la predizione è corretta vengono risparmiati 1 o 2 cicli di clock rispetto alla soluzione senza branch prediction. Se la predizione si rivela sbagliata non si incorre in ulteriori penalizzazioni rispetto alla soluzione senza branch prediction.

In generale il branch prediction migliora le performance di un 10%-20% nelle applicazioni di controllo.

4.2.1.2 *Reset del processore*

Il processore subisce un reset attivando per almeno 4 cicli di clock il pin di RESET. Di default, l'esecuzione avverrà a partire dall'indirizzo di memoria 0.

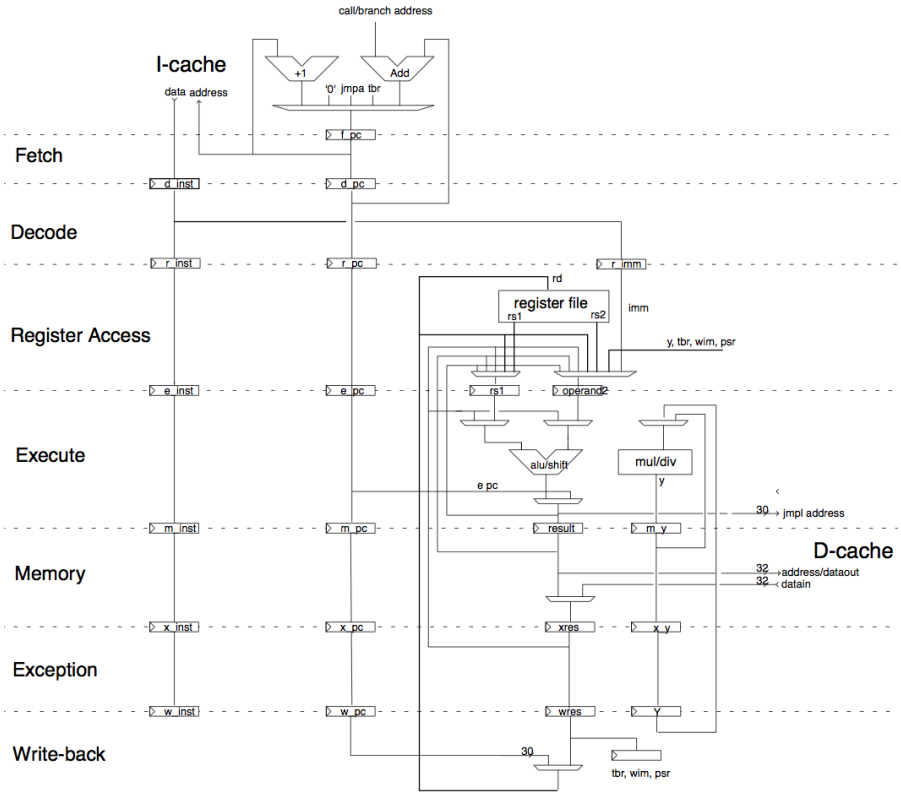

Figura 24: Flusso dei dati nella Integer Unit.

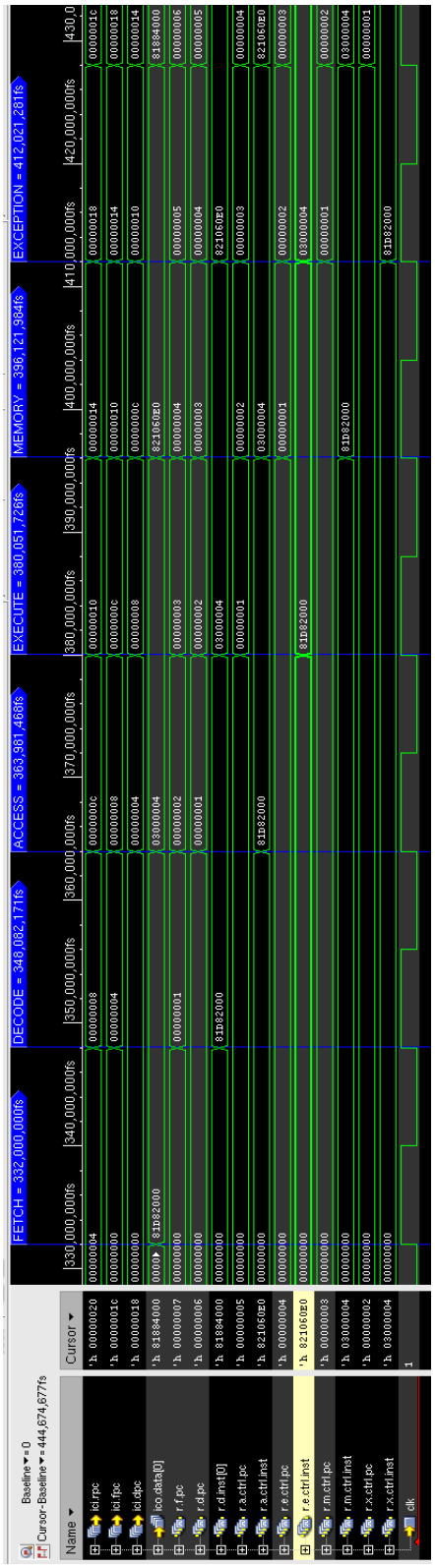

Figura 25: Waveform: attraversamento della pipeline.

5 VERIFICATION ENVIRONMENT PER LEON3

5.1 INTRODUZIONE

In questo capitolo viene presentato l'ambiente di verifica sviluppato in linguaggio *e* per l'ALU del processore Gaisler Leon3. L'unità ALU nel processore Leon3 coincide con il componente che è denominato *Integer Unit*, il quale è stato descritto in dettaglio nel capitolo 4. La descrizione del componente verrà integrata in questo capitolo, riportando i segnali di principale importanza per l'attività di verifica svolta.

5.1.1 *Funzionalità da verificare*

L'*Integer Unit* del processore *Leon3* svolge le principali operazioni aritmetico-logiche, oltre a coordinare tutti gli altri componenti (cache istruzioni, cache dati, FPU, coprocessore, ecc...).

Per questo progetto si è deciso di focalizzare l'attività di verifica sul processo di esecuzione delle istruzioni aritmetiche e logiche, trascurando o trattando in minima parte tutti gli aspetti non direttamente collegati ad esso (come ad esempio gli accessi alla memoria centrale, la gestione delle interruzioni, le operazioni a virgola mobile).

L'obbiettivo di verifica che viene posto è pertanto quello di valutare la correttezza dell'operato del processore relativamente a:

- fetch delle istruzioni;
- decodifica delle istruzioni;
- accesso ai registri;
- bit di stato (integer condition codes);
- memorizzazione del risultato nei registri;
- gestione del puntatore CWP dei registri a finestra;
- valutazione delle condizioni di branch.

Nella fattispecie, le istruzioni supportate dall'ambiente sviluppato e sottoposte quindi a verifica sono le seguenti:

- AND
- ANDCC
- ANDN

- ANDNCC
- OR
- ORCC
- ORN
- ORNCC
- XOR
- XORCC
- XNOR
- XNORCC
- ADD
- ADDCC
- SUB
- SUBCC
- ADDX
- SUBX
- ADDXCC
- SUBXCC
- SLL
- SRL
- SRA
- SAVE
- RESTORE
- BICC
- CALL

La descrizione dettagliata di queste funzioni si trova nel capitolo 4.

5.2 COMPONENTI DEL DUT

Il processore Leon3 è distribuito all'interno di un pacchetto denominato *GRLIB*, il quale contiene diverse varianti del core Leon3 insieme ad altri componenti di supporto, quali ad esempio memorie RAM, memorie ROM, controller vari, porte I/O general purpose. Il *device under test* dell'environment presentato corrisponde alla Integer Unit del core Leon3, ed è descritta in linguaggio VHDL nel file principale *iu3.vhd*, all'interno di *GRLIB*.

Per sottoporre a verifica il componente si è creato un modello VHDL di testbench, istanziando il componente *iu3* e connettendo alle sue porte alcuni segnali, in modo da poter interfacciare il componente con l'ambiente di verifica.

5.2.1 *Segnali di interfaccia di iu3*

SEGNALI DI CONTROLLO Questi segnali sono utilizzati per controllare il comportamento generale dell'ALU. Tra di essi c'è il segnale del clock (*clk*), il segnale per il reset (*rstn*) e il segnale per sospendere l'esecuzione dell'ALU (*holdn*).

CACHE DELLE ISTRUZIONI Il gruppo dei segnali utilizzati per comunicare con la cache delle istruzioni. L'identificatore *ico* rappresenta l'insieme dei segnali in uscita dalla cache e in ingresso all'ALU, mentre *ici* è l'insieme dei segnali che effettuano il percorso inverso. Di particolare importanza in riferimento all'ambiente di verifica:

- *ico.data[0]* Segnale a 32 bit che trasporta l'istruzione in uscita dalla cache.
- *ici.rpc* Segnale a 32 bit che trasporta il valore di PC da leggere al prossimo clock, in ingresso alla cache.
- *ici.fpc* Segnale a 32 bit che trasporta il valore di PC che si sta leggendo nel clock presente, in ingresso alla cache.
- *ici.inull* Segnale a 1 bit che vale 0 (zero) se l'istruzione non viene annullata, in ingresso alla cache.

CACHE DEI DATI Il gruppo dei segnali utilizzati per comunicare con la cache dei dati. L'identificatore *dco* rappresenta l'insieme dei segnali in uscita dalla cache e in ingresso all'ALU, mentre *dci* è l'insieme dei segnali che effettuano il percorso inverso.

REGISTER FILE Il gruppo dei segnali utilizzati per comunicare con i registri dell'ALU. L'identificatore *rfo* rappresenta l'insieme dei segnali in uscita dal register file e in ingresso all'ALU, mentre *rfi* è l'insieme dei segnali che effettuano il percorso inverso.

MOLTIPLICATORE E DIVISORE Il gruppo dei segnali utilizzati per comunicare con il modulo moltiplicatore esterno. L'identificatore *mulo* rappresenta l'insieme dei segnali in uscita dalla cache e in ingresso all'ALU, mentre *muli* è l'insieme dei segnali che effettuano il percorso inverso.

UNITÀ A VIRGOLA MOBILE Il gruppo dei segnali utilizzati per comunicare con l'unità a virgola mobile esterna. L'identificatore *fpo* rappresenta l'insieme dei segnali in uscita dalla FPU e in ingresso all'ALU, mentre *fpi* è l'insieme dei segnali che effettuano il percorso inverso.

COPROCESSORE Il gruppo dei segnali utilizzati per comunicare con la cache delle istruzioni. L'identificatore *cpo* è l'insieme dei segnali in uscita dal CP e in ingresso all'ALU, mentre *cpi* è l'insieme dei segnali che effettuano il percorso inverso.

INTERRUPT Il gruppo dei segnali utilizzati per comunicare con il controller degli interrupt. L'identificatore *irqi* è l'insieme dei segnali in uscita dal controller degli interrupt e in ingresso all'ALU, mentre *irqo* è l'insieme dei segnali che effettuano il percorso inverso.

SUPPORTO AL DEBUG Il gruppo dei segnali utilizzati per comunicare con l'unità di supporto al debug. L'identificatore *dbgi* è l'insieme dei segnali in uscita dal'unità e in ingresso all'ALU, mentre *dbgo* è l'insieme dei segnali che effettuano il percorso inverso.

TRACE BUFFER Il gruppo dei segnali utilizzati per comunicare con il trace buffer. L'identificatore *tbo* è l'insieme dei segnali in uscita dalla cache e in ingresso all'ALU, mentre *tbi* è l'insieme dei segnali che effettuano il percorso inverso.

5.2.2 *Configurazione di iu3*

Il componente *iu3* è stato progettato per essere ampiamente configurabile. Attraverso l'utilizzo del costrutto VHDL *generic*, in fase di istanziazione del componente si possono specificare i valori dei parametri di configurazione.

Per l'ambiente di verifica sviluppato si è deciso di ridurre al minimo la complessità dell'integer unit, disabilitando le funzionalità non indispensabili ai fini degli obiettivi di verifica fissati. La configurazione completa è riportata in tabella 6.

5.3 FILE REGISTER

Sebbene da un punto di vista logico i registri appartengano al componente ALU, nel design del processore Leon3 questi sono descritti

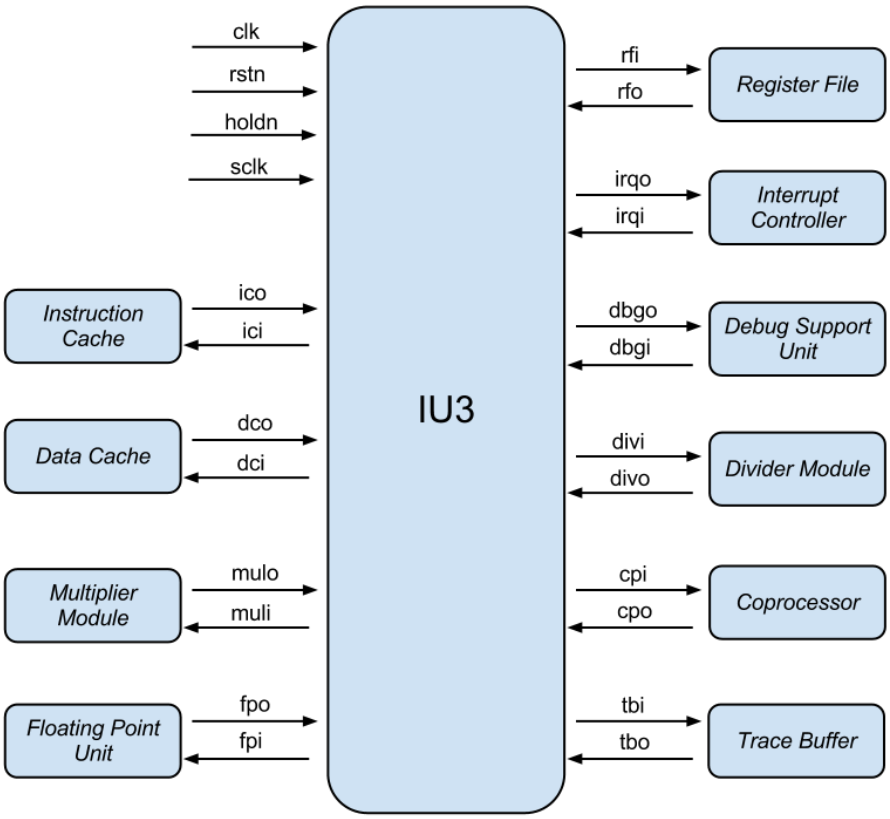

Figura 26: Segnali di interfaccia di iu3.

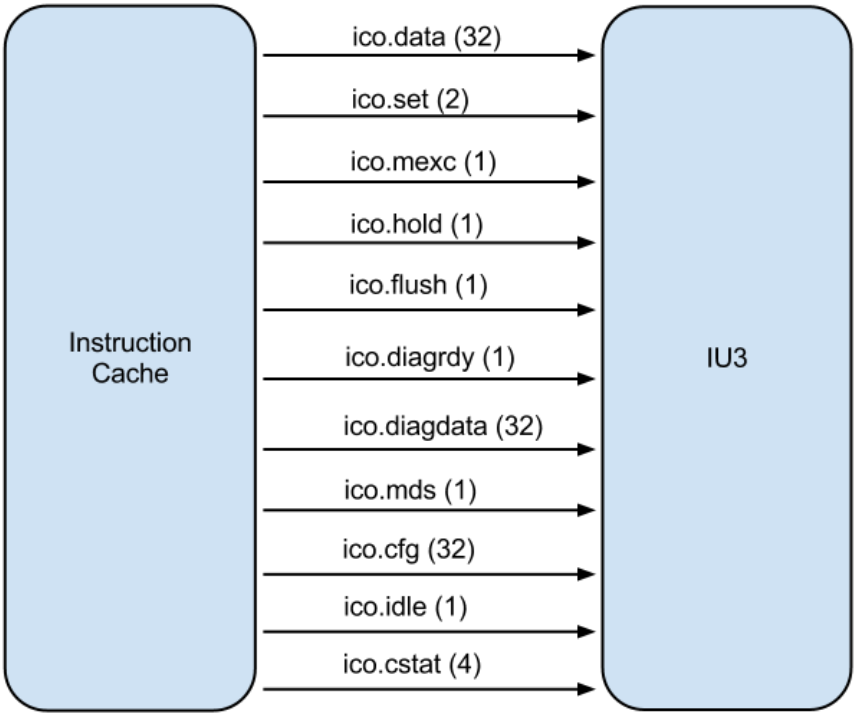

Figura 27: Segnali da cache istruzioni verso IU.

NOME	VALORE	DESCRIZIONE
nwin	8	Numero di finestre di registri
isets	1	Cache istruzioni, numero di set
dsets	1	Cache dati, numero di set
fpu	0	Disabilita modulo FPU
v8	0	Disabilita supporto alle istruzioni MUL e DIV
cp	0	Disabilita coprocessore
mac	0	Disabilita supporto alle istruzioni SMAC e UMAC
dsu	0	Disattiva interfaccia verso la Debug Support Unit
nwp	0	Numero di watchpoint
pclow	2	Bit meno significativi del PC, da non generare. PC[1:0] sono posti sempre a zero e non vengono calcolati, generando così sempre indirizzi multipli di 4
notag	0	Disabilita istruzioni tagged-arithmetic e CASA
lddel	2	Pipeline load delay (2 cicli)
disas	1	Stampa le istruzioni disassemblate nella console del simulatore VHDL
tbuf	0	Disabilita instruction trace
pwd	0	Disabilita supporto power-down
svt	0	Disabilita single-vector trapping
smp	0	Disabilita SMP
fabtech	0	Tecnologia di destinazione
clk2x	0	Disabilita supporto double clocking
bp	0	Disabilita branch prediction

Tabella 6: Parametri di configurazione di iu3 (generic).

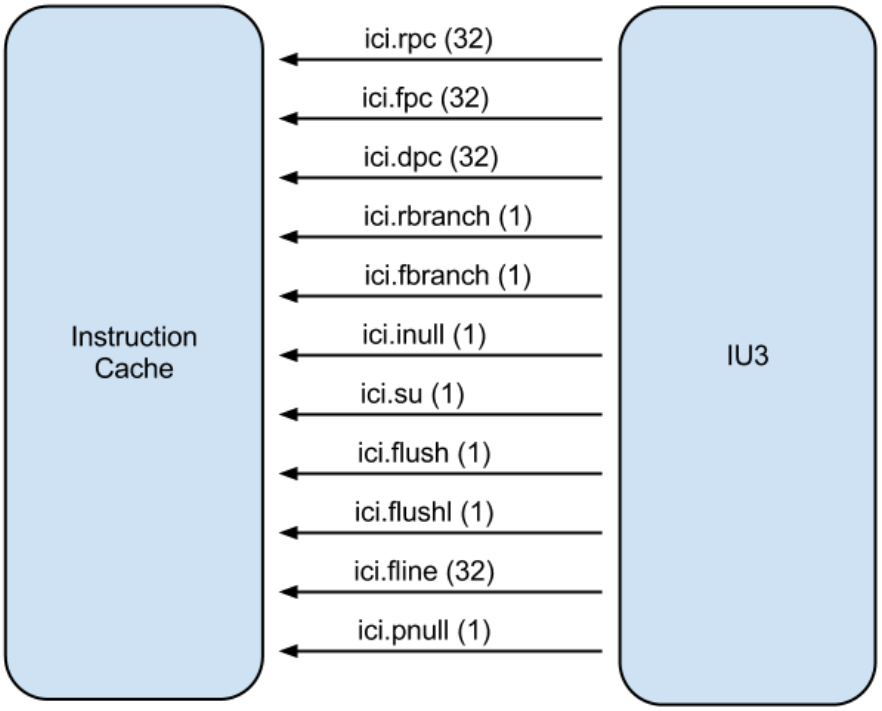

Figura 28: Segnali da IU a cache istruzioni.

in un componente separato, a vantaggio di una maggiore modularità e della configurabilità del numero di registri. Per gli obiettivi della verifica è stato pertanto necessario istanziare nel modello testbench anche il componente relativo ai registri, che in *GRLIB* è denominato *regfile_3p*.

5.3.1 *Segnali di interfaccia di regfile_3p*

CLOCK I segnali *rclk* e *wclk* sono, rispettivamente, i segnali di ingresso del clock per la lettura e la scrittura. In genere sono entrambi connessi al clock di sistema.

SEGNALI DI LETTURA I seguenti:

- *raddr1* e *raddr2*. Sono i segnali di ingresso a 8 bit che rappresentano gli indirizzi di lettura dei due registri sorgenti.
- *re1* e *re2*. Sono i segnali di ingresso a 1 bit che indica la presenza di un dato valido sul rispettivo segnale di indirizzo.
- *rdata1* e *rdata2*. Sono i segnali di uscita a 32 bit che trasportano il dato memorizzato nel registro selezionato.

SEGNALI DI SCRITTURA I seguenti:

- *waddr*. Il segnali di ingresso a 8 bit che rappresenta l'indirizzi di scrittura del registro di destinazione.

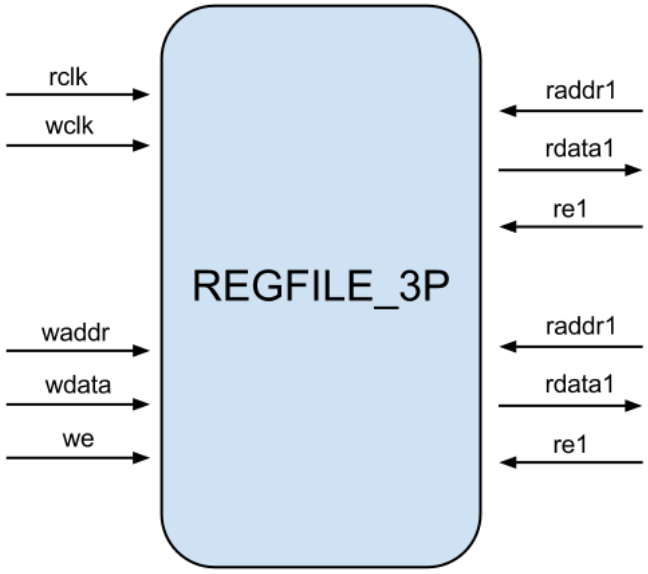

Figura 29: Segnali di interfaccia di regfile_3p.

- *we*. Il segnale di ingresso a 1 bit che indica la presenza di un dato valido sul segnale di indirizzo.
- *wdata*. Il segnale di ingresso a 32 bit che trasporta il dato da scrivere nel registro selezionato.

5.3.2 *Configurazione di regfile_3p*

Anche il componente *regfile_3p* è stato realizzato in modo da essere configurabile per quanto riguarda la capienza (numero di registri) e l'ampiezza (dimensione dei registri).

I valori scelti per il componente istanziato nell'ambiente di verifica sviluppato sono riportati nella tabella 7.

NOME	VALORE	DESCRIZIONE
tech	0	Tecnologia di destinazione
abits	8	Bit di indirizzo (calcolato come $\log_2(\text{NWINDOWS}+1)+4$)
dbits	32	Bit dei dati
numregs	136	Numero di registri (calcolato come $\text{NWINDOWS}*16+8$)

Tabella 7: Parametri di configurazione di regfile_3p (generic).

5.4 TIPI DI DATO

I primi componenti costruiti per l'ambiente di verifica sono state le strutture rappresentanti i tipi di dato principali che sono oggetto della verifica. Trattandosi di un processore, tali dati sono le istruzioni.

Si è scelto di partizionare i dati relativi all'esecuzione di una istruzione secondo questo criterio logico:

- **dati di input:**
  - instruction word (i 32 bit dell'istruzione);
  - valore PC dell'istruzione;
  - valore contenuto nei due registri sorgente.
- **dati di output:**
  - indirizzi di lettura e scrittura nel file register;
  - bit di stato (*condition codes*);
  - il *CWP* (current window pointer).

5.4.1 *Istruzione: l3iu_inst_s*

La definizione del dato di base di input è contenuta nella struttura *l3iu_struct_s*.

I campi di questa struttura corrispondono con i campi che caratterizzano le istruzioni, ampiamente descritti nella sezione 4.1.3. La struttura *l3iu_inst_s* è stata sviluppata in modo gerarchico, per rispettare la corretta suddivisione delle istruzioni in formati. Così facendo, il processo di generazione di un'istruzione casuale si sviluppa lungo un percorso del grafo ad albero riportato nella figura 30. I possibili percorsi portano a tipologie diverse di istruzioni valide.

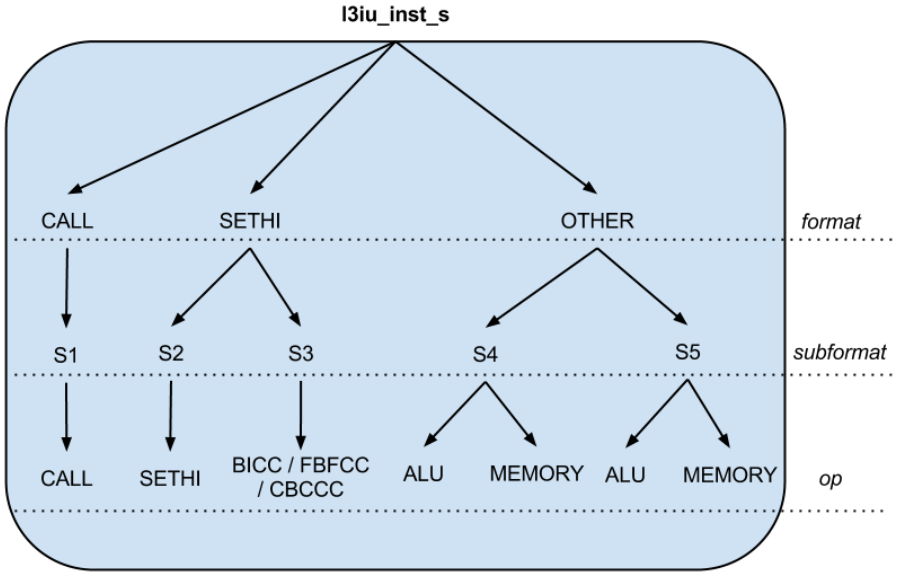

Figura 30: Struttura l3iu_inst_s.

5.4.2 *Risultato: l3iu_result_s*

La definizione del dato di base di output è contenuta in *l3iu_inst_s* che è una *struct*. In essa sono contenute le informazioni prodotte dall'ALU per effetto dell'esecuzione di una istruzione.

I campi di questa struttura sono i seguenti:

- **rf1_addr.** Indirizzo di lettura del registro sorgente 1
- **rf2_addr.** Indirizzo di lettura del registro sorgente 2
- **rs1_ren.** Enable del registro sorgente 1
- **rs2_ren.** Enable del registro sorgente 2
- **rd_addr.** Indirizzo di scrittura del registro destinazione
- **rd_data.** Valore da scrivere nel registro destinazione
- **icc_z.** Bit z (integer condition codes)
- **icc_n.** Bit n (integer condition codes)
- **icc_c.** Bit c (integer condition codes)
- **icc_v.** Bit v (integer condition codes)
- **cwp_e.** *CWP* successivo all'esecuzione dell'istruzione

5.5 REGISTER FILE

La presenza di un elemento di memoria qual è il componente del *register file* ha richiesto la modellazione di un'unità analoga in codice *e*, in grado di lavorare come copia del register file del DUT.

Le operazioni che prevedono letture o scritture nel register file del DUT vengono eseguite anche nel register file *mirror*; questo consente la verifica istante per istante della consistenza del register file.

L'unità *l3iu_regfile_u* contiene la definizione del register file mirror, comprendente la struttura dati per l'effettiva memorizzazione del contenuto dei registri, e due funzioni per la lettura e scrittura dei dati:

- **read(addr : byte) : int**. Restituisce i 32 bit contenuti nel registro di indirizzo *addr*.
- **write(data : int, addr : byte).** Memorizza i 32 bit *data* nel registro di indirizzo *addr*, eventualmente sovrascrivendo i dati già contenuti.

In caso di un indirizzo *addr* non valido, il problema viene segnalato e la simulazione viene interrotta.

Gli indirizzi dei registri presenti nelle istruzioni sono indirizzi relativi alla finestra corrente, indicata dal valore memorizzato in CWP. Quando l'ALU deve accedervi nel register file, l'indirizzo fisico del registro è calcolato secondo i valori riportati in tabella 8.

FINESTRA	REGISTRI LOGICI	INDIRIZZI FISICI
-	0-7 (global)	128-135
0	8-15 (out)	0-7
0	16-23 (local)	8-15
0	24-31 (in)	16-23
1	8-15 (out)	16-23
1	16-23 (local)	24-31
1	24-31 (in)	32-39
2	8-15 (out)	32-39
2	16-23 (local)	40-47
2	24-31 (in)	48-55
3	8-15 (out)	48-55
3	16-23 (local)	56-63
3	24-31 (in)	64-71
4	8-15 (out)	64-71
4	16-23 (local)	72-79
4	24-31 (in)	80-87
5	8-15 (out)	80-87
5	16-23 (local)	88-95
5	24-31 (in)	96-103
6	8-15 (out)	96-103
6	16-23 (local)	104-111
6	24-31 (in)	112-119
7	8-15 (out)	112-119
7	16-23 (local)	120-127
7	24-31 (in)	0-7

Tabella 8: Mappa degli indirizzi nel register file.

5.6 ARCHITETTURA DELL'AMBIENTE DI VERIFICA

5.6.1 *Environment, Agent, Syncronizer e Signal Map*

L'ambiente di verifica si sviluppa a partire dall'unità *l3iu_env_u*. Questa al suo interno si occupa di istanziare i componenti dell'ambiente di verifica: l'agente, il sincronizzatore, la mappa dei segnali e il checker. Inoltre vengono stabiliti i collegamenti per far comunicare tra

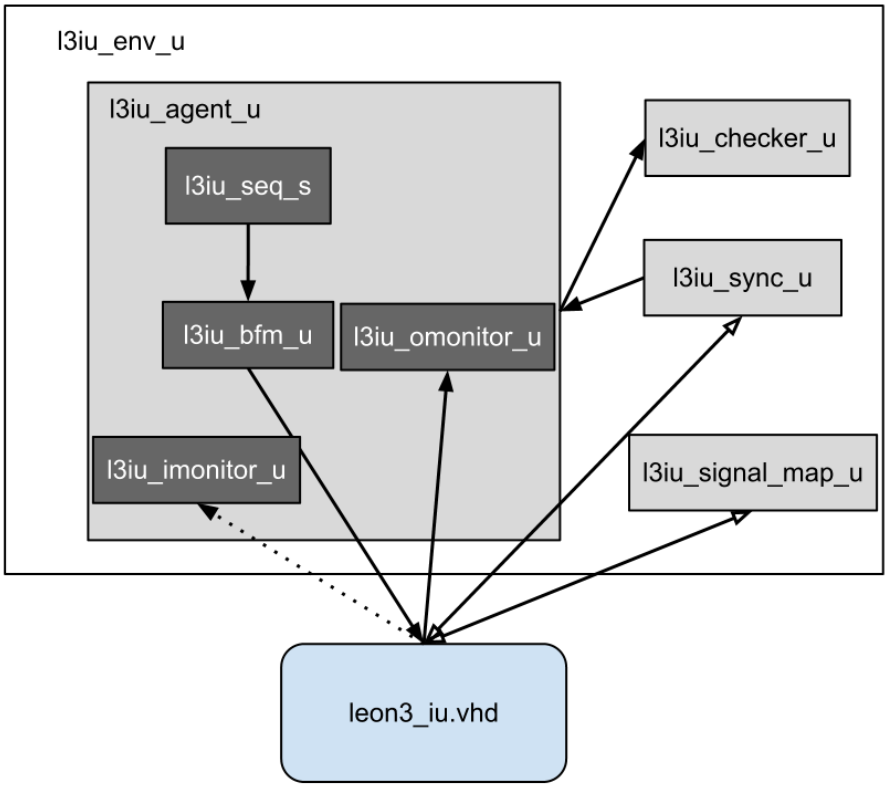

Figura 31: Architettura dell'ambiente di verifica.

loro i vari componenti.

L'interfacciamento con i segnali del DUT viene gestito in modo centralizzato dal modulo Signal Map, descritto nell'unità *l3iu_signal_map_u*. Questa contiene le porte elencate nella tabella 9.

NOME E	SEGNALE VHDL	DESCRIZIONE
clk_p	*clk_s*	Clock di sistema
rst_p	*rstn_s*	Reset
i_data_p	*ico_s.data[0]*	Output dell'istruzione dalla cache
i_inull_p	*ici_s.inull*	Annullamento dell'istruzione
i_rpc_p	*ici_s.rpc*	Input nella cache dell'indirizzo della prossima istruzione da leggere
i_fpc_p	*ici_s.fpc*	Input nella cache dell'indirizzo dell'istruzione in fetch
r1_data_p	*rfo_s.data1*	Valore del contenuto del registro sorgente 1
r2_data_p	*rfo_s.data2*	Valore del contenuto del registro sorgente 2
r1_ren_p	*rfi_s.ren1*	Read enable del registro sorgente 1
r2_ren_p	*rfi_s.ren2*	Read enable del registro sorgente 2
r1_addr_p	*rfi_s.raddr1*	Indirizzo del registro sorgente 1
r2_addr_p	*rfi_s.raddr2*	Indirizzo del registro sorgente 2
rd_waddr_p	*rfi_s.waddr*	Indirizzo del registro destinazione
rd_data_p	*rfi_s.wdata*	Valore da memorizzare nel registro di destinazione
icc_p	*iu_instance.r.m.icc*	Integer condition codes
cwp_e_p	*iu_instance.r.e.cwp*	CWP corrente

Tabella 9: Porte principali definite in l3iu_signal_map_u.

La sincronizzazione tra i vari componenti dell'ambiente di verifica e il DUT stesso viene facilitata dalla presenza di una unità sincronizzatrice, *l3iu_sync_u*, la quale emette gli eventi riportati in tabella 10.

NOME E	DESCRIZIONE
clock_e	Evento clock di sistema, emesso sul fronte di salita del segnale *clk_s*
reset_e	Evento di reset del processore, emesso se il segnale di reset è rimasto basso per più di 4 cicli di clock.

Tabella 10: Eventi definiti in l3iu_sync_u.

5.6.2 *Driver*

Il driver è il componente che produce i dati di input da sottoporre al DUT. In questo ambiente di verifica si è sfruttata una funzionalità del linguaggio *e* utile per semplificare e standardizzare la produzione di sequenze di input: il costrutto *sequence*.

Nella sua versione più semplice, con *sequence* viene prodotto automaticamente un driver in grado di generare elementi del tipo specificato. Inoltre è possibile specificare, sottoforma di vincolo, la lunghezza della sequenza di elementi da generare.

Il driver *l3iu_driver_u* viene definito nella struttura *l3iu_seq_s*.

5.6.3 *BFM*

Il BFM è costituito da una *unit* chiamata *l3iu_bfm_u*. Come già descritto nel capitolo 3, il BFM è quel componente di un ambiente di verifica che implementa il protocollo fisico del DUT, consentendo di comunicare con esso e di inviare gli stimoli di input. Il BFM sviluppato opera principalmente sui segnali di ingresso del processore relativi alla cache delle istruzioni, sostituendosi ad essa in quanto a funzionalità.

Il funzionamento del BFM è suddiviso in più fasi:

1. **Attesa del reset.** In questa fase il BFM resta in attesa della condizione di reset del processore, la quale viene segnalata tramite evento generato dal monitor di input.

2. **Sequenza di init.** Il BFM invia al processore una serie di istruzioni per inizializzare il processo di verifica.

3. **Ciclo principale.** All'interno del ciclo principale il BFM preleva una istruzione dal driver, controlla la validità dell'istruzione in riferimento allo stato corrente e la invia al processore. Il ciclo principale viene eseguito finchè non si esauricono le istruzioni prodotte dal driver.

4. **Finalizzazione.** Una volta terminate le istruzioni della sequenza, il BFM immette nel processore alcune istruzioni NOP per consentire il completamento dell'attraversamento della pipeline da parte delle ultime istruzioni del test.

5.6.3.1 *Sequenza di init*

La sequenza di init eseguita dal BFM si occupa della corretta impostazione dei registri di stato del processore e dell'azzeramento di registri ALU.

I registri di stato che vengono inizializzati sono i seguenti:

- *PSR.* Processor State Register.
- *WIM.* Window Invalid Mask.
- *TBR.* Trab Base Register.
- *Y.* Multiply/Divide Register.

Successivamente nel register file vengono azzerati i registri globali e si procede poi all'azzeramento dei registri contenuti in ciascuna delle 8 finestre. A causa della struttura circolare del registro a finestre, è sufficiente azzerare solamente il gruppo dei registri local e uno a scelta tra i due gruppi di registri in e out. Il gruppo di registri non azzerato in modo esplicito si azzererà ugualmente a causa della sovrapposizione dei registri tra finestre adiacenti.

Le istruzioni di init sono riportati nelle tabelle 11, 12 e 13. Le istruzioni relative all'azzeramento dei registri della finestra vengono eseguite all'interno di un ciclo *for* che itera per 8 volte (valore pari al numero delle finestre presenti) l'azzeramento di una singola finestra. Ogni ciclo si conclude con l'istruzione di cambio finestra (*RESTORE*).

ISTRUZIONE	NOTAZIONE ASSEMBLY	DESCRIZIONE
0x81d82000	flush [%g0+0x0000]	L'istruzione di flush sincronizza il fetch delle istruzioni con le operazioni di accesso alla memoria. Sebbene non sia richiesta esplicitamente dallo scenario di verifica, l'istruzione è stata inserita per garantire compatibilità con modifiche future.
0x03000004	sethi %hi(0x00001000),%g1	Carica nella parte alta del registro globale 1 il numero 0x00001000 e azzera i bit restanti della parte bassa.
0x821060e0	or %g1,0x00E0,%g1	Memorizza nel registro globale 1 il risultato dell'OR tra i bit del registro globale 1 e il numero 0x00E0. Questa istruzione e la precedente hanno come effetto la memorizzazione del numero 0x00001000E0 nel registro globale 1.
0x81884000	wr %g1,%g0,%psr	Scrive nel registro *psr* il risultato dell'OR tra i bit del registro globale 1 e il registro globale 0 (tutti zeri). Questa istruzione ha come effetto la copiatura del contenuto del registro globale 1 nel psr.
0x81900000	wr %g0,%g0,%wim	Scrive nel registro *wim* il risultato dell'OR tra i bit del registro globale 0 e il registro globale 0 (tutti zeri). Questa istruzione ha come effetto l'azzeramento del contenuto del registro wim.
0x81980000	wr %g0,%g0,%tbr	Scrive nel registro *tbr* il risultato dell'OR tra i bit del registro globale 0 e il registro globale 0 (tutti zeri). Questa istruzione ha come effetto l'azzeramento del contenuto del registro tbr.
0x81800000	wr %g0,%g0,%y	Scrive nel registro *y* il risultato dell'OR tra i bit del registro globale 0 e il registro globale 0 (tutti zeri). Questa istruzione ha come effetto l'azzeramento del contenuto del registro y.

Tabella 11: Istruzioni della sequenza di init.

ISTRUZIONE	ASSEMBLY	DESCRIZIONE
0x80100000	or %g0,%g0,%g0	Azzeramento registro globale 0
0x82100000	or %g0,%g0,%g1	Azzeramento registro globale 1
0x84100000	or %g0,%g0,%g2	Azzeramento registro globale 2
0x86100000	or %g0,%g0,%g3	Azzeramento registro globale 3
0x88100000	or %g0,%g0,%g4	Azzeramento registro globale 4
0x8a100000	or %g0,%g0,%g5	Azzeramento registro globale 5
0x8c100000	or %g0,%g0,%g6	Azzeramento registro globale 6
0x8e100000	or %g0,%g0,%g7	Azzeramento registro globale 7

Tabella 12: Istruzioni della sequenza di init: azzeramento dei registri globali.

La scrittura del valore 0x00001000E0 nel registro di stato del processore ha questo effetto:

- i bit dal 23 al 20 sono azzerati, ovvero vengono azzerati gli *integer condition codes*;
- il bit 7 porta il processore in *supervisor mode*;
- il bit 5 abilita le trap;
- i bit dal 4 allo 0 sono azzerati, azzerando così il CWP.

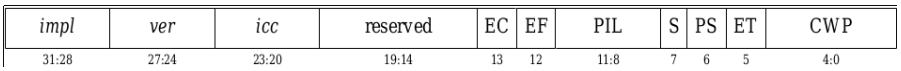

Figura 32: Registro PSR (Processor Status Register).

5.6.3.2 *Controllo della validità delle transizioni*

Esistono delle sequenze di istruzioni non valide secondo le specifiche SPARC. Poichè il driver non effettua tale controllo, il BFM prima di inviare l'istruzione al processore verifica se all'istante presente quella funzione è valida. Le regole più importanti consistono in:

- Vietato inserire istruzioni di trasferimento del controllo all'interno del delay slot di una istruzione di trasferimento del controllo.
- Le istruzioni SAVE e RESTORE possono generare una trap di overflow o underflow se ci si trova rispettivamente nella finestra di indice più basso e in quella di indice più alto.

Il BFM è stato progettato per evitare entrambe le situazioni, scartando l'istruzione fornita dal driver qualora si verificasse una violazione della validità, e passando direttamente all'istruzione successiva. L'istruzione scartata non contribuisce alla determinazione del coverage.

ISTRUZIONE	ASSEMBLY	DESCRIZIONE
0xa0100000	or %g0,%g0,%l0	Azzeramento registro locale 0
0xa2100000	or %g0,%g0,%l1	Azzeramento registro locale 1
0xa4100000	or %g0,%g0,%l2	Azzeramento registro locale 2
0xa6100000	or %g0,%g0,%l3	Azzeramento registro locale 3
0xa8100000	or %g0,%g0,%l4	Azzeramento registro locale 4
0xaa100000	or %g0,%g0,%l5	Azzeramento registro locale 5
0xac100000	or %g0,%g0,%l6	Azzeramento registro locale 6
0xae100000	or %g0,%g0,%l7	Azzeramento registro locale 7
0x90100000	or %g0,%g0,%o0	Azzeramento registro out 0
0x92100000	or %g0,%g0,%o1	Azzeramento registro out 1
0x94100000	or %g0,%g0,%o2	Azzeramento registro out 2
0x96100000	or %g0,%g0,%o3	Azzeramento registro out 3
0x98100000	or %g0,%g0,%o4	Azzeramento registro out 4
0x9a100000	or %g0,%g0,%o5	Azzeramento registro out 5
0x9c100000	or %g0,%g0,%o6	Azzeramento registro out 6
0x9e100000	or %g0,%g0,%o7	Azzeramento registro out 7

Tabella 13: Istruzioni della sequenza di init: azzeramento dei registri local, in, out.

5.6.3.3 *Invio di un'istruzione*

L'invio di una istruzione avviene in un thread separato per ciascuna istruzione da inviare.

5.6.4 *Input Monitor*

Il componente monitor di input si occupa del rilevamento dei dati che vengono immessi nel DUT. In questo caso il monitor di input rileva le istruzioni che il BFM man mano invia alla CPU.

Per ogni istruzione rilevata, l'input monitor ricostruisce il tipo di dato corrispondente (*l3iu_inst_s*) e lo invia al checker tramite il costrutto *e* di *method port*.

Il rilevamento di ogni istruzione avviene in un thread separato per ogni istruzione, il quale si estende nel tempo per coprire l'attraversamento della pipeline da parte dell'istruzione, consentendo così la raccolta delle informazioni di interesse. Le fasi e le informazioni raccolte sono le seguenti:

1. **FE (Instruction Fetch)** I 32 bit dell'istruzione vengono prelevati e tramite funzione di *unpacking* viene ricostruira la corrispondente struttura *l3iu_inst_s* con i campi correttamente impostati.
2. **DE (Decode)** -
3. **RA (Register Access)** I valori in uscita dal register file (corrispondenti al contenuto dei registri sorgenti) vengono aggiunti alla struttura *l3iu_inst_s* relativa all'istruzione corrente.
4. **EX (Execute)** -
5. **ME (Memory)** -
6. **XC (Exception)** -
7. **WR (Write)** -

5.6.5 *Output Monitor*

Il componente monitor di output si occupa del rilevamento dei dati che provengono in uscita dal DUT e di alcuni dati relativi alle informazioni di stato interne ad esso. In questo caso il monitor di output rileva i risultati delle istruzioni che il BFM man mano invia alla CPU, e gli effetti della loro esecuzione.

Per ogni istruzione eseguita, l'output monitor ricostruisce il tipo di dato corrispondente (*l3iu_res_s*) e lo invia al checker tramite il costrutto *e* di *method port*.

Il rilevamento delle informazioni avviene in un thread separato per ogni istruzione, il quale si estende nel tempo per coprire l'attraversamento della pipeline da parte dell'istruzione, consentendo così

la raccolta delle informazioni di interesse. Le fasi e le informazioni raccolte sono le seguenti:

1. **FE (Instruction Fetch)** -
2. **DE (Decode)** Vengono prelevati i valori diretti al register file relativi agli indirizzi di lettura dei registri sorgente.
3. **RA (Register Access)** -
4. **EX (Execute)** -
5. **ME (Memory)** Si leggono i valori correnti di CWP e degli *integer condition codes*. Tali valori possono essere stati modificati per effetto dell'esecuzione dell'istruzione corrente.
6. **XC (Exception)** -
7. **WR (Write)** Vengono prelevati i valori diretti al register file e relativi all'indirizzo del registro di destinazione e del contenuto da scriverci.

STADIO	INPUT MONITOR	OUTPUT MONITOR
Fetch	Istruzione	-
Decode	-	Indirizzi registri sorgenti
Register Access	Contenuto dei registri sorgenti	
Execute	-	-
Memory	-	CWP e integer condition codes
Exception	-	-
Write	-	Indirizzo e dati del registro di destinazione

Tabella 14: Input monitor e output monitor a confronto.

5.6.6 *Checker*

Il componente checker (implementato nell'unit *l3iu_checker_u*) si occupa del'attività principale di verifica.

Il checker è dotato di una struttura dati di tipo coda per memorizzare le istruzioni che riceve dall'input monitor. Ogni volta che un'istruzione finisce di essere eseguita, l'output monitor invia al checker la struttura dati contenente i risultati dell'elaborazione (*l3iu_result_s*); il checker procede allora estraendo dalla coda delle istruzioni la prima istruzione non ancora processata.

Poichè l'esecuzione delle istruzioni da parte del processore avviene in ordine di arrivo, la struttura dati impiegata di tipo *FIFO* garantisce che ad ogni ricezione di un risultato si estragga l'istruzione corrispondente.

Il checker gestisce gli scenari relativi al branch, come la presenza dello *slot delay* in seguito di un'istruzione di branch e del flag *annul* per l'annullamento dell'istruzione stessa. Quando l'istruzione corrente è una istruzione di delay, il checker la processa o meno a seconda dello stato di *annul* presente nell'istruzione di branch.

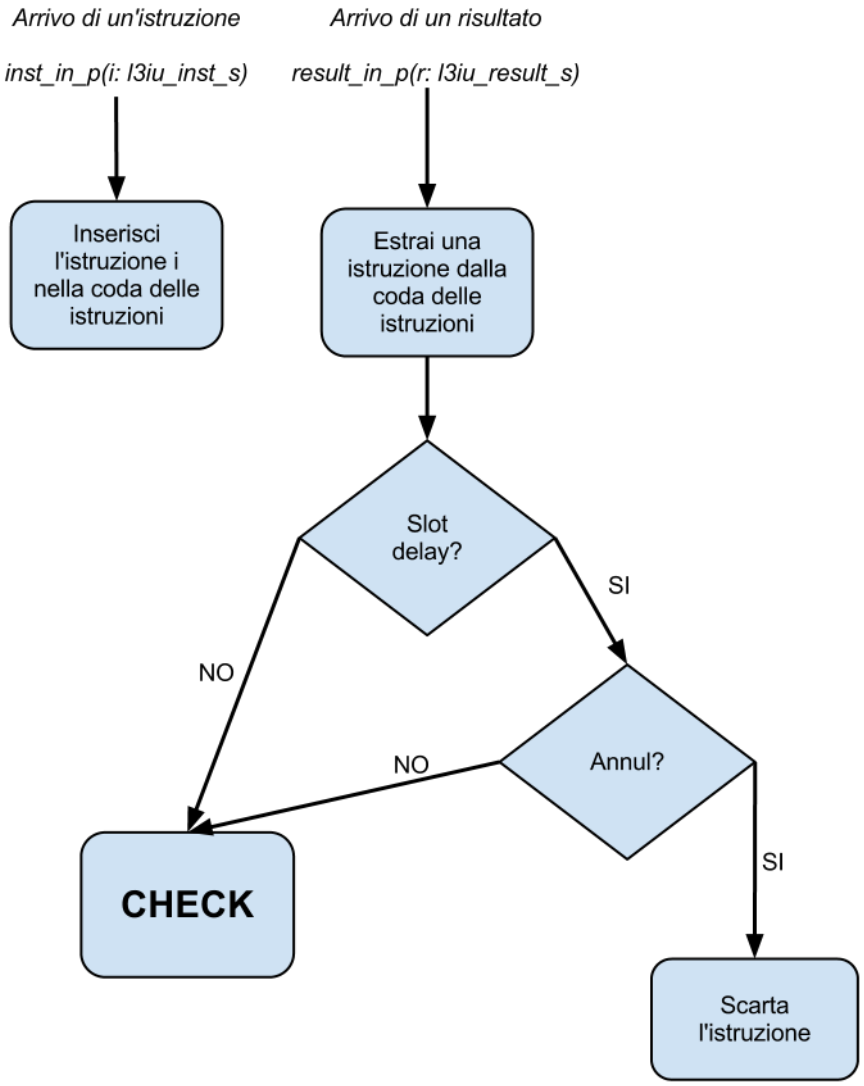

Figura 33: Funzionamento del checker.

In presenza di un'istruzione da processare (ovvero di un'istruzione che risulta effettivamente eseguita dalla CPU), il checker effettua questi passi:

1. **Decodifica dell'istruzione.** L'istruzione, descritta nella struttura *l3iu_inst_s* corrente viene decodificata, individuando gli *op-*

code, gli indirizzi di eventuali registri ed eventuali valori immediati.

2. **Esecuzione dell'istruzione.** L'istruzione viene eseguita.

3. **Controllo.** Si confronta il risultato dell'esecuzione attuale da parte del checker con il risultato contenuto nella struttura *l3iu_res_s* attuale. Gli elementi controllati sono riportati nella tabella 15.

4. **Salvataggio dello stato.** Le informazioni attuali relative a CWP, PC, e *integer condition codes* vengono salvate nelle variabili di stato. Il nuovo contenuto del registro di destinazione, se presente, viene memorizzato aggiornando la copia mirror del register file.

ATTRIBUTO	DESCRIZIONE
PC	Il program counter dell'istruzione deve corrispondere al program counter vecchio + 4, o al program counter di destinazione di un branch/call.
Source Register 1 Address	L'indirizzo del registro sorgente 1 selezionato dall'istruzione deve corrispondere all'indirizzo di lettura 1 selezionato nel register file.
Source Register 1 Data	I dati letti dal registro sorgente 1 devono corrispondere ai dati memorizzati nel medesimo registro del register file mirror.
Source Register 2 Address	L'indirizzo del registro sorgente 2 selezionato dall'istruzione deve corrispondere all'indirizzo di lettura 2 selezionato nel register file.
Source Register 2 Data	I dati letti dal registro sorgente 2 devono corrispondere ai dati memorizzati nel medesimo registro del register file mirror.
Risultato	L'operazione effettuata deve produrre il risultato corretto.
Destination Register Address	L'indirizzo del registro di destinazione selezionato dall'istruzione deve corrispondere all'indirizzo di scrittura selezionato nel register file.
Integer condition codes	I bit di stato relativi al risultato dell'operazione, se l'istruzione lo prevede, devono essere impostati correttamente.
Condizione di branch	Per le istruzioni di branch condizionato, la condizione di salto deve essere computata correttamente.

Tabella 15: Checker: elementi sottoposti a controllo.

5.7 COVERAGE

Per valutare quali e quante istruzioni vengono eseguite in ciascun test, si è scelto di utilizzare le istruzioni stesse come metrica della copertura. Questa impostazione permette di controllare l'esito del processo di generazione casuale, verificando in particolare che tutte le istruzioni che si intendono provare siano state effettivamente generate. Inoltre questo approccio consente di osservare se vi siano alcune istruzioni che, a causa delle modalità di generazione, vengano generate con minor frequenza; è possibile quindi correggere questa tendenza impostando degli appositi vincoli nel file di test chiedendo il rispetto di una certa proporzione nell'assegnazione del valore a certi campi.

L'elenco completo degli *item* soggetti a coverage è riportato nella tabella 16.

ITEM	DESCRIZIONE
op	Campo di 2 bit di ciascuna istruzione. Identifica la tipologia dell'istruzione.
op2	Campo di 3 bit delle istruzioni della famiglia con formato SETHI. Identifica una specifica istruzione.
S4op3_alu	Campo di 6 bit delle istruzioni di famiglia ALU con sottoformato S4. Identifica una specifica istruzione.
S4op3_memory	Campo di 6 bit delle istruzioni di famiglia MEMORY con sottoformato S4. Identifica una specifica istruzione.
S5op3_alu	Campo di 6 bit delle istruzioni di famiglia ALU con sottoformato S5. Identifica una specifica istruzione.
S5op3_memory	Campo di 6 bit delle istruzioni di famiglia MEMORY con sottoformato S5. Identifica una specifica istruzione.

Tabella 16: Item del coverage delle istruzioni.

6 CONCLUSIONI

6.1 TEST EFFETUATI

L'ambiente di verifica sviluppato e descritto nel capitolo 5 è stato impiegato per l'esecuzione di una serie di test. Durante la fase di sviluppo dell'ambiente sono stati effettuati dei test comprendenti singole sequenze di istruzioni di lunghezza variabile (da 100 a 1000 istruzioni), generate a partire da un insieme ridotto di tipi di istruzioni. Questi test hanno fatto da guida durante la fase di sviluppo dell'ambiente di verifica, contribuendo in modo decisivo nel debugging del codice *e*. Con l'aumentare dei progressi nell'implementazione dell'ambiente è stato possibile estendere il numero di istruzioni supportate, fino al raggiungimento degli obiettivi di verifica prefissati.

Il collaudo finale dell'ambiente è avvenuto tramite l'esecuzione automatica di alcune batterie di test, dove ciascuna batteria comprende l'esecuzione di alcune sequenze di test di 5000 istruzioni. Il numero di sequenze per ogni batteria è stato fatto variare tra 10 e 50.

6.1.1 *Test singoli*

Un esempio di file di test per l'esecuzione di un test singolo è quello riportato nel listato 10. Il vincolo espresso tra le righe 6-10 serve a definire in termini percentuali la quantità di istruzioni da generare per tipo, rispetto al totale: nell'esempio si tratta del 10% di istruzioni SETHI, 10% di istruzioni CALL, 80% di istruzioni OTHER. I valori sono stati fissati in questo modo considerato che la tipologia OTHER è la più grande econtiene al suo interno più dell'80% delle istruzioni totali.

Durante la simulazione viene prodotto un log che, oltre a segnalare la presenza di eventuali errori, descrive istante per istante i controlli eseguiti. Un esempio di log è visibile in figura 34.

Codice 10: l3iu_test_main.e

```
<'
import l3iu_top;

extend l3iu_inst_s {

  keep soft format == select {
    10: SETHI;
    10: CALL;
    80: OTHER;
    };

  keep op in [SETHI, ALU, CALL];

  keep subformat in [S1, S2, S3, S4, S5];

  keep op3_alu in [AND, ANDCC, ANDN, ANDNCC, OR, ORCC,
                   ORN, ORNCC, XOR, XORCC, XNOR, XNORCC,
                   ADD, ADDCC,
                   SUB, SUBCC,
                   ADDX, SUBX, ADDXCC, SUBXCC,
                   SLL, SRL, SRA,
                   SAVE, RESTORE
                ];

  keep op2 in [SETHI, BICC];

  keep rs1>=0;
  keep rs1<32;

  keep rs2>=0;
  keep rs2<32;

  keep rd>=0;
  keep rd<32;
};

extend MAIN l3iu_seq_s {
  keep count==1000;
};
'>
```

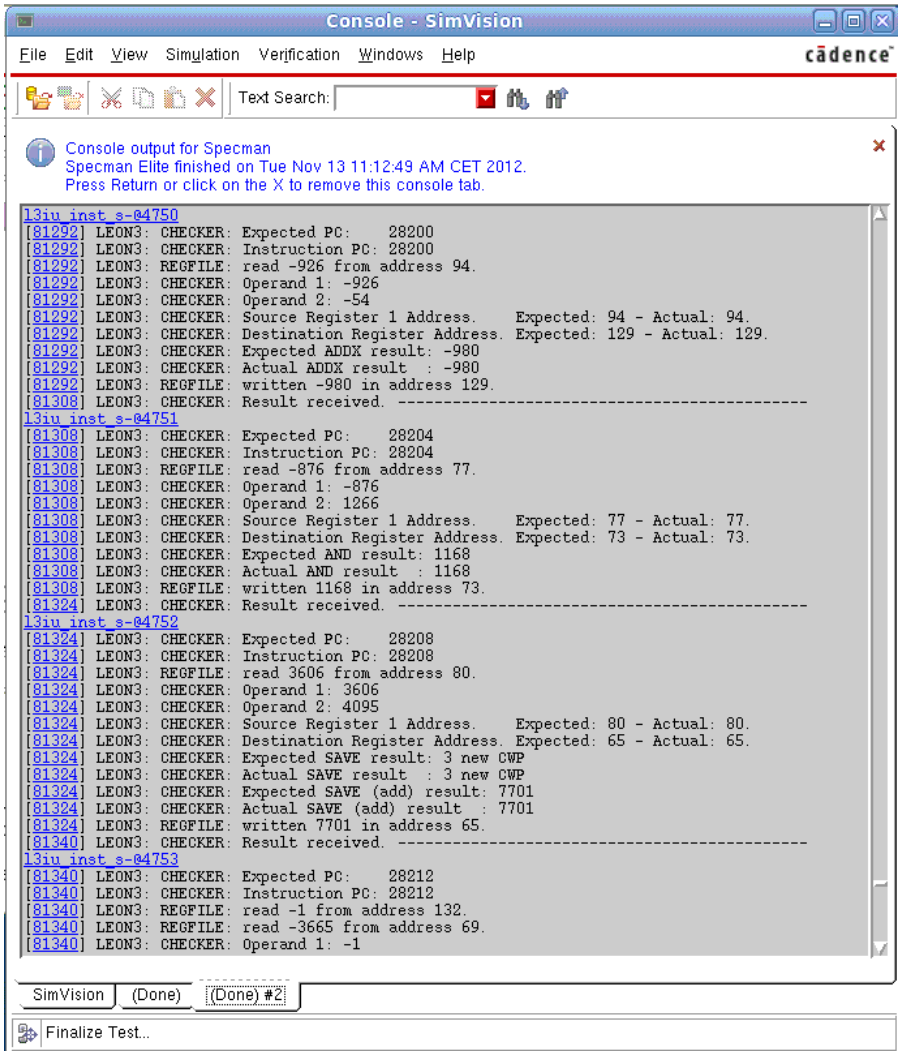

Figura 34: Test: log dello svolgimento. Nel log appaiono le istruzioni eseguite e il confronto tra i valori attesi e quelli rilevati. Una discrepanza tra i due tipi di valore denota un errore nell'esecuzione del DUT, oppure (cosa molto probabile in fase di sviluppo iniziale) un bug dell'ambiente di verifica.

6.1.2 *Batch test con Incisive Enterprise Manager*

Incisive Enterprise Manager è un tool software messo a disposizione da *Cadence* per facilitare l'esecuzione di test estensivi e come supporto ai test di regressione.

Il programma consente di pianificare ed eseguire ripetutamente un'istanza di test, variando ad ogni esecuzione il seed del generatore casuale. In questo modo è possibile osservare il comportamento di un certo test in presenza di evoluzioni diverse. Alla fine dell'esecuzione si ottiene il report dell'esito di ciascuna istanza (superata/fallita). Inoltre è possibile effetuare la fusione (*merge*) dei dati di coverage ottenuti in ciascuna istanza e visionare così un rapporto sintetico del

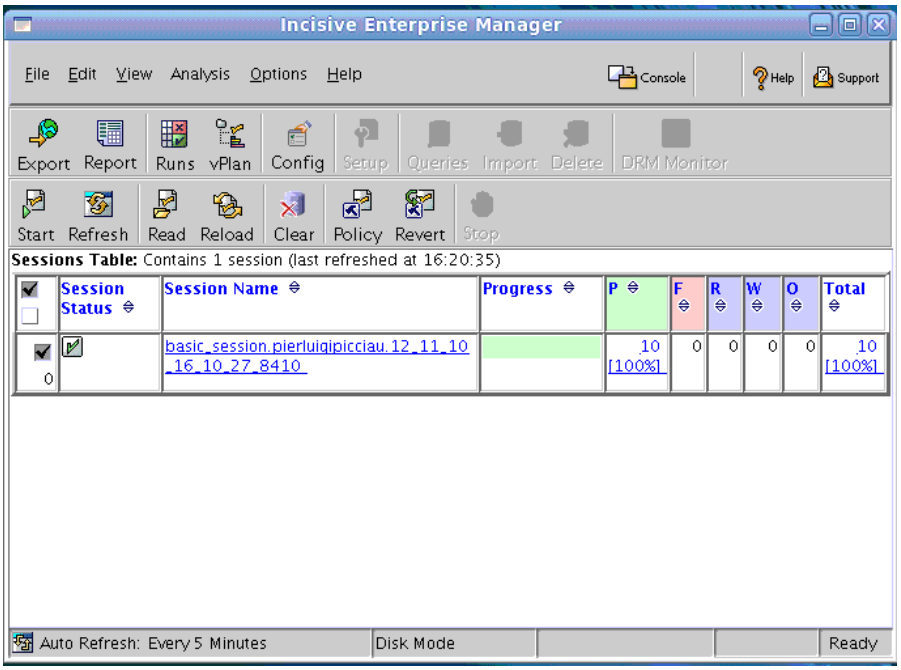

Figura 35: Incisive Enterprise Manager: Sessions Table. Questa tabella riporta le sessioni attive, evidenziandone il progresso (istanze eseguite, istanze completate e istanze fallite).

coverage comprensivo di tutti i test.

Nel rapporto di esecuzione prodotto da *Incisive Enterprise Manager* è visualizzato il seed utilizzato da ciascuna istanza; questo, all'occorrenza, consente di rieseguire quell'istanza separatamente dalle altre, facilitanto così il debugging da parte dello sviluppatore.

6.2 COVERAGE

Il rapporto del coverage permette di misurare la completezza del processo di verifica. Gli elementi sottoposti a coverage sono stati descritti nella sezione 5.7.

A seguire, le figure 37, 38, 39 e 40 rappresentano, a titolo di esempio, le informazioni sul coverage raccolte durante l'esecuzione di una istanza di test di circa 10 mila istruzioni.

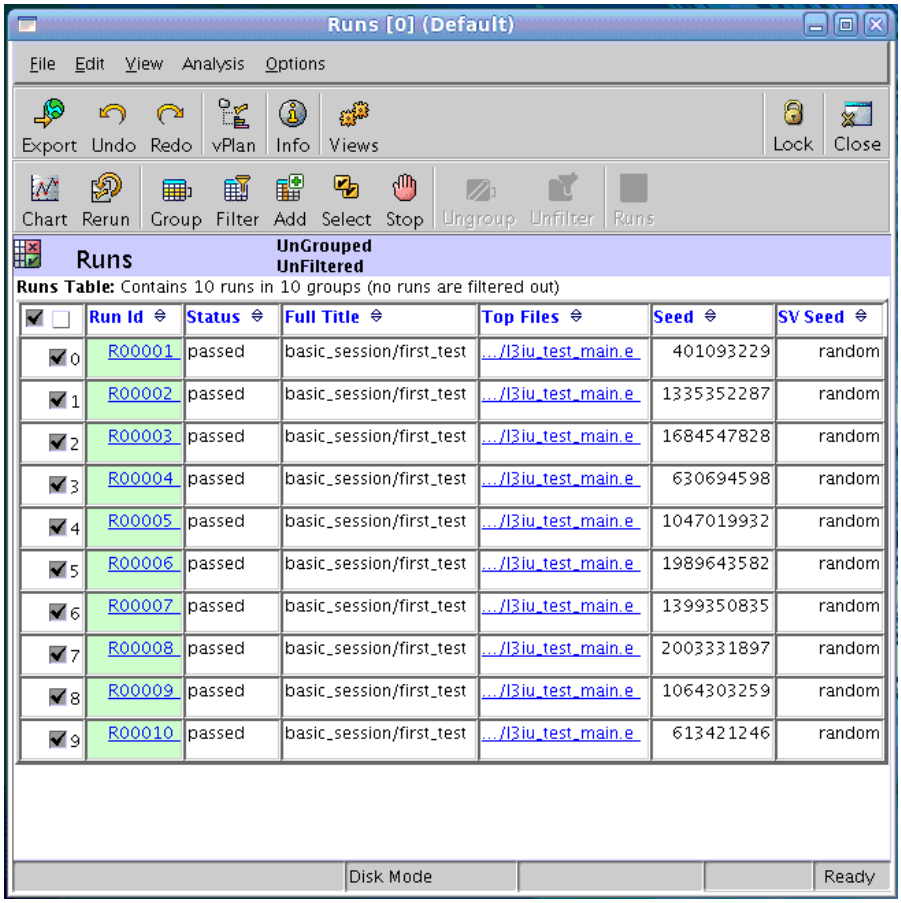

Figura 36: Incisive Enterprise Manager: Runs Table. Questa tabella riporta le istanze di esecuzione contenute in una medesima sessione. Ciascuna istanza è corredata delle informazioni su id, stato dell'esecuzione e seed utilizzato dal generatore casuale.

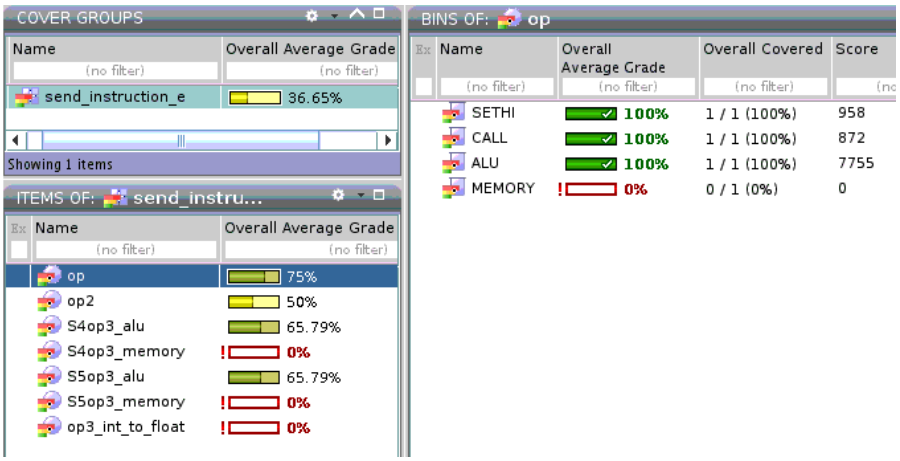

Figura 37: Dati di coverage per il campo *op*, rappresentante la famiglia principale dell'istruzione. L'insieme relativo alle istruzioni di tipo MEMORY risulta vuoto, proprio perchè tali istruzioni sono state escluse dagli obiettivi di verifica prefissati. Durante questa istanza di test risultano essere state eseguite 958 istruzioni di famiglia SETHI, 872 istruzioni di famiglia CALL e 7765 istruzioni ALU.

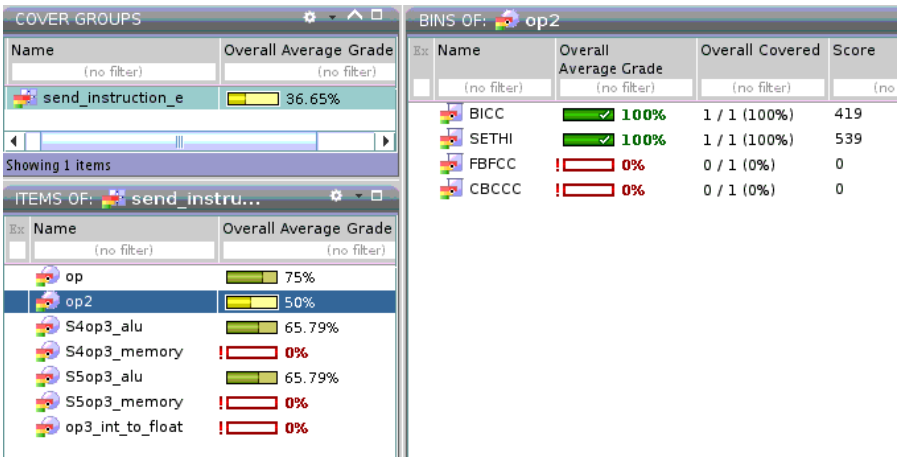

Figura 38: Dati di coverage per il campo *op2*, rappresentante i tipi di istruzione contenuti nella famiglia SETHI. Gli insiemi relativi alle istruzioni di tipo FBFCC e CBCCC risultano vuoti, proprio perchè tali istruzioni riguardano FPU e coprocessore e sono state escluse dagli obiettivi di verifica prefissati. Durante questa istanza di test risultano essere state eseguite 419 istruzioni BICC e 539 istruzioni SETHI.

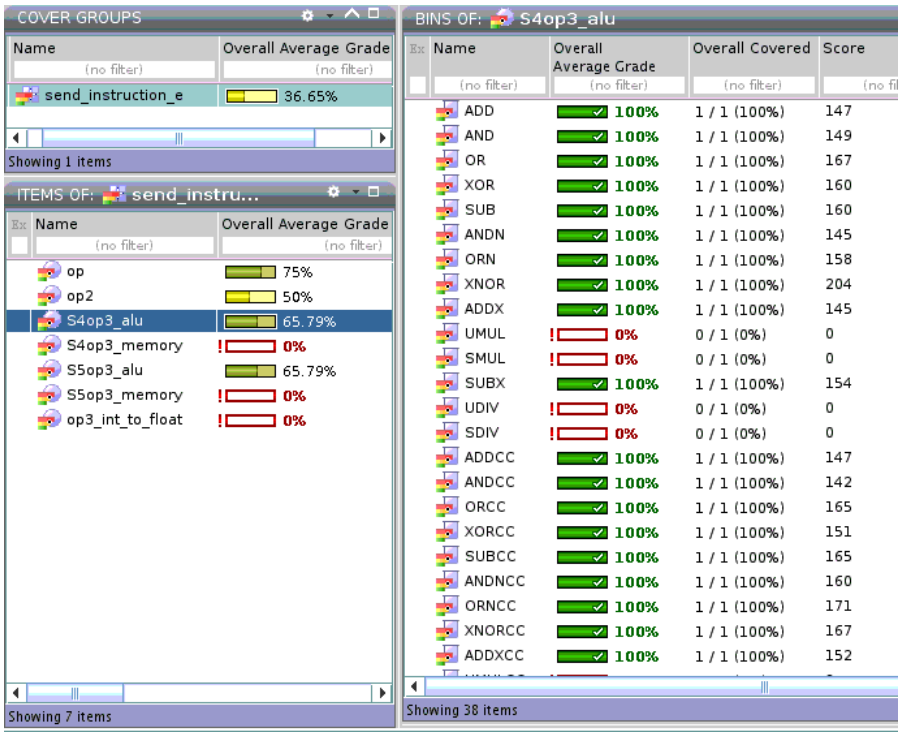

Figura 39: Dati di coverage per il campo *op3_alu* delle istruzioni di formato *S4*, ovvero le istruzioni di tipo ALU con due registri sorgente. Alcune istruzioni (ad esempio UMUL, SMUL, UDIV, e SDIV) non sono state verificate, proprio perchè tali istruzioni riguardano il modulo moltiplicatore e divisore, ed erano state escluse dagli obiettivi di verifica prefissati.

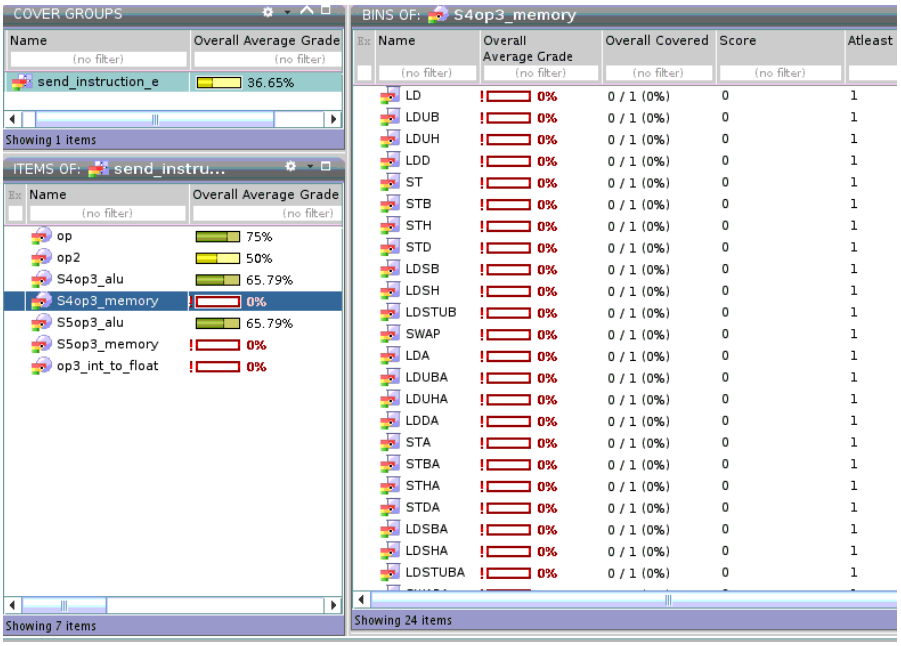

Figura 40: Dati di coverage per il campo *op3_memory* delle istruzioni di formato *S4*, ovvero le istruzioni di accesso alla memoria. Queste istruzioni erano state escluse dagli obiettivi di verifica prefissati e infatti non risultano essere processate istruzioni di questo tipo.

6.3 SVILUPPI FUTURI

L'ambiente di verifica sviluppato e presentato in questo documento, sebbene risponda ai requisiti posti in fase iniziale, si presta a possibili miglioramenti, tra i quali:

- l'aumento del numero di istruzioni ALU supportate, soprattutto per quanto riguarda la *tagged arithmetic*, il modulo moltiplicatore e divisore, le istruzioni FPU;
- il supporto alle istruzioni di accesso alla memoria centrale;
- la simulazione delle dinamiche di *cache miss* in fase di fetch di un'istruzione;
- il supporto alla branch prediction.

BIBLIOGRAFIA

[1] *VHDL Application Workshop*. Esperan, 2000.

[2] *Specman Elite Basics for Verification Environment Developers*. Cadence, 2006.

[3] *Specman Elite Basics for Verification Environment Users*. Cadence, 2006.

[4] Henry Chang, Larry Cooke, Merrill Hunt, Grant Martin, Andrew McNelly, and Lee Told. *Surviving the SOC Revolution*. Kluwer Academic Publishers, San Jose, CA, USA, 1st edition, 1999.

[5] Ulrich Heinkel, Martin Padeffke, Werner Haas, Thomas Buerner, Herbert Braisz, Thomas Gentner, and Alexander Grassman. *The VHDL Reference*. John Wiley & Sons, Chicester, West Sussex, England, 1st edition, 2000.

[6] Prakash Rashinkar, Peter Paterson, and Leena Singh. *System-on-a-chip Verification*. Kluwer Academic Publishers, Norwell, MA, USA, 1st edition, 2001.