POLITECNICO DI TORINO

Corso di Laurea Magistrale in Ingegneria Informatica

Tesi di Laurea Magistrale

Tecniche di sviluppo di programmi di testing on-line per system-on-chip automotive

Relatori:
Paolo Bernardi
Ernesto Sanchez Sanchez
Riccardo Cantoro

Candidato:
Simone Regis

Ottobre 2017

Indice

Elenco delle tabelle

Elenco delle figure

Introduzione

Attualmente in ambito automotive, visto l'aumento dei dispositivi elettronici a bordo utilizzati per rendere più sicura la guida (e.g. ABS, ESP, EBD) o per facilitarla (e.g. sistema start and stop, sospensioni elettroniche, park assist), l'elettronica all'interno dei veicoli richiede unità sempre più specifiche, e queste a loro volta richiedono microcontrollori affidabili. Inoltre gli algoritmi utilizzati per soddisfare queste nuove esigenze sono diventati progressivamente più complessi, richiedendo quindi una potenza computazionale maggiore. In quest'ottica quindi i produttori di microcontrollori hanno implementato nuovi device, quali unità specifiche per l'accelerazione di alcune funzionalità e processori multicore.

In accordo con gli standard ISO 26262 e ASIL, è necessario sviluppare strategie per il testing on-line da impiegarsi in ambienti safety-critical e mission-critical che richiedono un sistema totalmente affidabile. Questi requisiti si traducono in una serie di processi di controllo del sistema lungo tutto il suo ciclo di vita. Questi includono l'analisi del rischio, la verifica e la validazione del progetto ma anche le varie operazioni durante e alla fine della produzione e dell'assemblaggio. Sempre più spesso vanno però applicate operazioni di test addizionali durante il ciclo di vita del prodotto, come un testing on-line periodico e un rilevamento simultaneo degli errori. I requisiti di affidabilità possono essere quindi soddisfatti trovando un compromesso tra il livello di fault detection o fault tolerance e il costo ammissibile.

Capitolo 1

SBST (Software-based self-test)

Questo capitolo introduce al mondo del testing di system-on-chip con particolare attenzione ad ambienti safety-critical come quello automotive o aerospace. Vengono introdotte le principali tecniche utilizzate in ambito industriale concentrandosi sull'approccio SBST (Software-based self-test) adottato. Vengono quindi introdotti i principali modelli a cui si fa riferimento per rilevare tutti i possibili guasti elettronici che si potrebbero verificare a livello di chip/processore con l'obiettivo di garantire una copertura dei fault, i.e. guasti, quanto più grande possibile.

1.1 Introduzione al SBST

Nel campo di applicazione dei sistemi integrati basati su microprocessore, l'approccio SBST (Software-based self-test) è stato affrontato in modi diversi all'interno della comunità di ricerca scientifica [1] [2] [3] e costituisce una valida soluzione sulla base dei requisiti richiesti dagli standard [13]. Le tecniche SBST consistono sostanzialmente nel lasciar eseguire alla CPU una sequenza di istruzioni per poter eccitare e propagare alle uscite i guasti che affliggono il sistema [4]. Comparato a soluzioni di testing più tradizionali basate sull'hardware, come può essere ad esempio la tecnica BIST (Built-in self-test), presenta molti vantaggi: la possibilità di testing [5] e di diagnosi [6] autonomo sia per quanto riguarda il processore che per i periferici, l'operabilità in normal mode, la facilità di implementazione, l'assenza di necessità di modifiche a livello di design hardware e il consenso all'esecuzione dei test alla frequenza nominale del circuito. In ogni caso le metodologie SBST sono ancora limitate nel loro campo di applicabilità a livello industriale date le difficoltà nella scrittura di programmi di test efficienti ed efficaci, e le difficoltà nell'inventare nuove metodologie utili per il testing. In situazioni che riguardano i test manifatturieri sono ancora spesso utilizzate soluzioni BIST per la loro peculiarità di raggiungere

alti valori di copertura in breve tempo, seguendo un flusso di sviluppo ridotto per quanto riguarda applicazioni on-line.

Le soluzioni SBST sono invece da preferire quando si è alla ricerca di uno strumento di controllo periodico per monitorare lo stato di salute del sistema senza interferire con il comportamento normale dello stesso [7]. Una soluzione standard adottata in ambito industriale risulta essere l'applicazione periodica di test a partire da una libreria Core self-test (CST), composta da un insieme di programmi di test SBST. Chiaramente questi programmi richiedono spazio in memoria flash e, in accordo con lo scheduler, necessitano tempo per essere eseguiti in modo concorrente con l'applicazione primaria. In questa situazione, il processore è periodicamente forzato ad eseguire codice per il self-test in grado di rilevare la possibile presenza di guasti permanenti all'interno del core del processore o nei periferici connessi a questo [8]. Queste procedure sono progettate per attivare possibili guasti, quindi comprimere e salvare i risultati del self-test in locazioni di memoria disponibili, oppure far scattare un segnale nel caso in cui il test non sia terminato correttamente.

1.2 Descrizione di fault

In questa sezione verranno approfondite le nozioni base necessarie alla comprensione delle procedure di testing in relazione, in modo particolare, al concetto di fault, ossia di guasto, che consiste in una condizione anormale o in un difetto elettronico che può condurre ad un fallimento del sistema stesso.

Risulta di essenziale conoscenza il concetto di osservabilità, su cui si basa l'approccio SBST. L'obiettivo di questo metodo, infatti, è quello di riuscire a propagare l'effetto del guasto, che si può verificare in un determinato punto del circuito, fino alle uscite, rendendolo di fatto osservabile. Per osservabilità quindi si intende quella proprietà secondo cui una variazione del valore logico di un determinato segnale si riesce a propagare fino alle uscite, alterando l'output dell'unità in questione. Il guasto non deve essere quindi mascherato da altri segnali o da porte logiche.

1.2.1 Fault, errore e failure

All'interno di un contesto in cui l'affidabilità assume un ruolo centrale, è necessario fare una distinzione tra tre concetti ampiamente utilizzati, ossia quelli di fault, errore e failure. Si noti come questi aspetti siano fortemente legati da una relazione di causa-effetto, come si può notare in Figura 1.1.

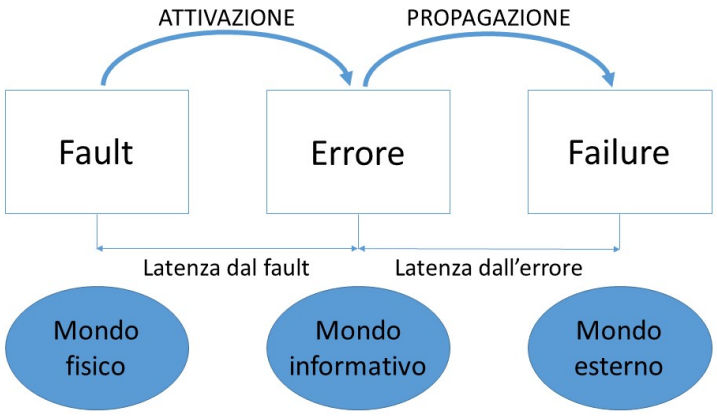

Figura 1.1: Relazione tra fault, errore e failure

Per fault si intende un difetto fisico o un'imperfezione che si verifica all'interno di una unità hardware. Esso si verifica all'interno di un componente che costituisce un sistema, e quindi proprio all'interno del mondo fisico. L'attivazione del fault comporta lo scatenarsi di un errore, ossia una deviazione rispetto alla correttezza e all'accuratezza che il sistema dovrebbe avere nel suo complesso. L'errore costituisce quindi la manifestazione del fault e si mostra al livello delle informazioni, come possono essere ad esempio dei dati su word in uscita da un modulo. Infine l'inadempimento di alcune azioni previste, dovute a informazioni errate, comporta un malfunzionamento del sistema. Questo quindi si ripercuote sul mondo esterno rendendo l'utente del sistema a conoscenza dell'errore. Quest'ultimo passo è quello che definisce un'anomalia del comportamento; risulta quindi di grande interesse andare ad indagare a fondo sull'esistenza di fault all'interno del sistema, soprattutto in relazione a situazioni critiche che potrebbero verificarsi, cercando di evitarle o quantomeno di smorzarne gli effetti.

1.2.2 Modelli di fault

I fault [14] sono assegnati a delle classi che corrispondono al loro stato corrente di rilevazione, e viene utilizzato un codice a due caratteri per specificare la classe stessa. Le classi sono definite gerarchicamente, ma solo quelle di basso livello possono essere assegnate a un fault, mentre quelle di alto livello servono per riportare i risultati o per fare osservazioni a un certo livello di astrazione. Di seguito sono riportate tutte le classi esistenti della gerarchia e vengono analizzate le principali utili per la trattazione in corso.

DT (Detected)

Questa classe comprende quei fault che sono identificati come "fortemente" rilevati. Questo garantisce una differenza osservabile tra il valore atteso e il valore risultante per effetto del fault. Il rilevamento può avvenire per simulazione o attraverso un'analisi delle implicazioni.

- **DR** - *Detected Robustly*: fault marcati detected utilizzando criteri WNR (weak non-robust), ROB (robust) o HFR (hazard-free robust). Durante l'ATPG almeno un pattern che ha consentito il rilevamento viene mantenuto.
- **DS** - *Detected by Simulation*: fault marcati detected dopo un'esplicita simulazione dei pattern. Durante l'ATPG almeno un pattern che ha consentito il rilevamento viene mantenuto.
- **DI** - *Detected by Implication*: fault marcati detected dopo un'analisi delle implicazioni a livello circuitale. Questo viene fatto attraverso una prova formale per cui si riescono a marcare detected alcuni fault, pur non riuscendolo a verificare in simulazione.
- **D2** - *Detected clock fault with loadable nonscan cell faulty value of 0 and 1.*
- **TP** - *Transition partially detected*: fault marcati detected con un ritardo che eccede il ritardo minimo specificato dall'opzione -max_delta_per_fault. Un fault di questo tipo può continuare ad essere simulato con l'intento di ottenere un test migliore per il guasto in esame.

PT (Possibly Detected)

Questa classe comprende quei fault di cui non è chiaro il fatto se siano stati rilevati o meno. Un comportamento usuale è quello di ritenerne detected il 50%, dato che statisticamente ciò può considerarsi vero, ma è possibile forzare il valore a qualsiasi altra percentuale.

- **AP** - *ATPG Untestable Possibly Detected*: un'analisi ha determinato che il fault non può essere rilevato con i vincoli ATPG correnti e la risposta di un sistema guasto sarebbe simulare il comportamento con una X[1] piuttosto che con uno 0 o un 1.
- **NP** - *Not analyzed, Possibly Detected*: identici ai fault AP eccetto per il fatto che o l'analisi non è stata completata o non si può provare che il fault verrà sempre simulato con una X. E' possibile che un pattern differente consenta di rilevare il fault e marcarlo quindi DS.
- **P0** - *Detected clock fault and loadable nonscan cell faulty value is 0.*
- **P1** - *Detected clock fault and loadable nonscan cell faulty value is 1.*

UD (Undetectable)

Questa classe comprende quei fault che non possono essere rilevati (né "fortemente" né "possibilmente") sotto alcuna condizione. Nel calcolo del test coverage (vedi Sezione 1.2.3) non sono considerati dato che non hanno un'implicazione logica sul comportamento del circuito e non possono causare failure.

- **UU** - *Undetectable Unused*: fault localizzati su circuiteria che non ha un collegamento con un punto esternamente osservabile.
- **UO** - *Undetectable Unosservable*: simili ai fault UU con la differenza che questi sono collocati su porte logiche inutilizzate con fanout[2]. I fault su porte inutilizzate senza fanout[3] sono invece marcati UU.

[1] Significa "unknown", valore quindi sconosciuto.
[2] Collegate in uscita ad altre porte logiche inutilizzate.
[3] Collegate in uscita ad altre porte logiche che però non sono inutilizzate.

- **UT** - *Undetectable Tied*: fault localizzato su un pin legato al medesimo valore di quello nel caso di guasto.

- **UB** - *Undetectable Blocked*: fault localizzato su circuiteria che è bloccata dal propagare i segnali sino ad un punto osservabile per colpa di specifiche interconnessioni logiche.

- **UR** - *Undetectable Redundant*: fault appartenenti a porzioni di logica irraggiungibili, solitamente aggiunti per motivazioni di equiripartizione della potenza.

AU (ATPG Untestable)

Questa classe comprende quei fault che non possono né essere rilevati "fortemente" né si può provare che siano ridondanti. Nel calcolo del test coverage sono considerati alla stregua di quelli non testati perché potenzialmente potrebbero causare failure.

- **AN** - *ATPG Untestable Not-Detected*: fault che non possono essere "possibilmente" rilevati ed è stata compiuta un'analisi per provare che non possono né essere rilevati con le condizioni ATPG correnti né hanno superato il *redundancy check*.

- **AX** - *ATPG Untestable Timing Exceptions*.

ND (Not Detected)

Questa classe comprende quei fault per cui una generazione di test non è ancora stata in grado di creare dei pattern per controllarli o osservarli.

- **NC** - *Not Controlled*: non si è ancora stati in grado di trovare un pattern per controllare il luogo del guasto per portarlo in uno stato necessario per il rilevamento.

- **NO** - *Not Observed*: nonostante il luogo del guasto sia controllabile, non è ancora stato trovato un pattern per osservare il fault.

1.2.3 Calcolo di fault coverage

Per quanto riguarda i principi secondo cui vengono calcolati i valori di copertura dei guasti [14], si può far riferimento sostanzialmente a due metodi, denominati rispettivamente *fault coverage* e *test coverage*. Nel caso della fault coverage la formula utilizzata è la seguente:

$$fault\_coverage = \frac{\#DTfaults + (\#PTfaults \cdot PTcredit)}{\#totalFaults}$$

Mentre per la test coverage si utilizza:

$$test\_coverage = \frac{\#DTfaults + (\#PTfaults \cdot PTcredit)}{\#totalFaults - \#UDfaults - (\#AU/ANfaults \cdot AU/ANcredit)}$$

I valori costanti definiti all'interno delle formule possono assumere dei valori personalizzati, ma l'approccio di default prevede valori pari a: $PTcredit = 0.5$, $AU/ANcredit = 0$. Con queste assunzioni si presentano quindi le seguenti conseguenze:

- vengono considerati rilevati la metà dei guasti che sono nello stato *Possibly Detected*, andando ragionevolmente ad includere una porzione di tali fault;
- il contributo dei fault *ATPG Untestable Not-Detected* viene di fatto eliminato e quindi, non andando a sottrarsi al numero totale di fault, ci si ritrova in una situazione più simile tra le due formule, dove questi fault vengono considerati non rilevati piuttosto che da non tenere in conto.

In questa trattazione, quando si farà accenno ai valori di copertura ottenuti, si farà sempre riferimento alla fault coverage. Come si può notare, risulta essere quella che fornisce un valore di copertura sempre minore o uguale alla test coverage, andando di fatto a costituire il caso peggiore di copertura trai due.

1.3 Processo di classificazione dei fault

La classificazione dei fault risulta essenziale all'interno di una procedura SBST [9]. Essa consta della misurazione dell'abilità nella rilevazione dei guasti che potrebbero influenzare negativamente il comportamento dell'intero SoC, o parte di questo. Il processo non è banale, come può dimostrare lo schema in Figura 1.2, e può risultare lento o non accurato se alcuni fattori importanti vengono meno. La procedura di classificazione è tradizionalmente divisa in tre fasi consecutive: la preparazione del programma di test, la simulazione dello stesso e la successiva fault-simulazione.

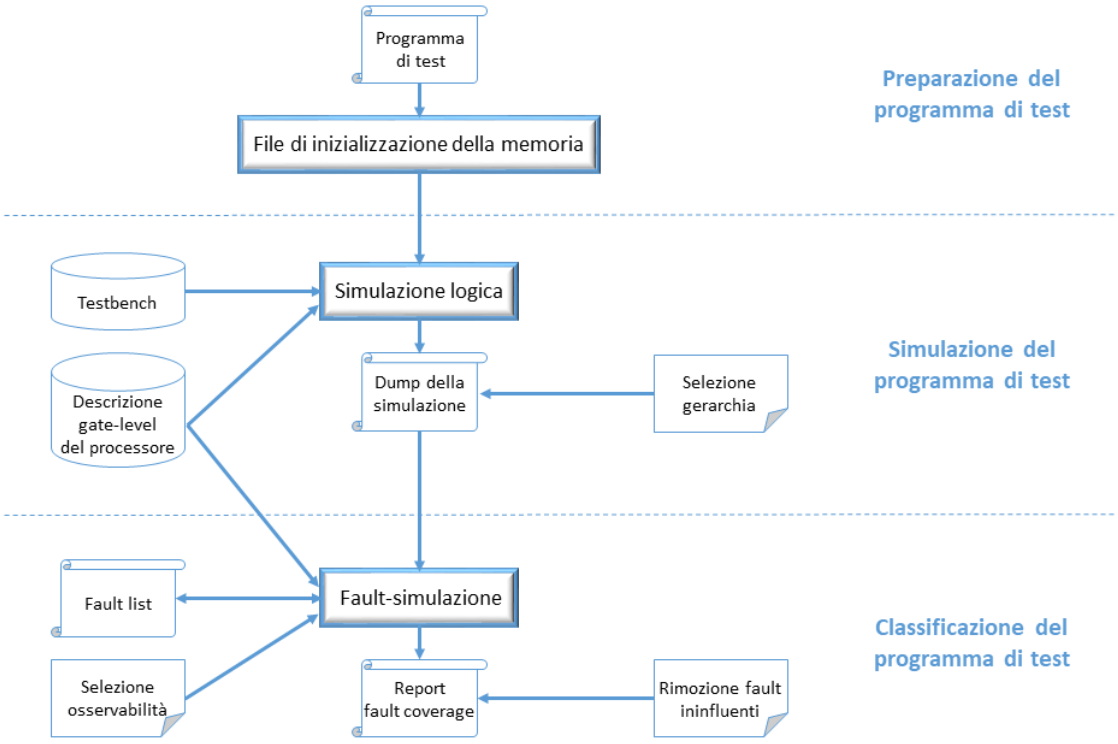

Figura 1.2: Flusso di classificazione dei fault per la generazione di un report della fault coverage

La prima fase è la preparazione del programma di test che consiste nello studio delle funzionalità della porzione di hardware che si intende testare, e nell'effettiva implementazione e traduzione in un programma di test. Quest'ultimo dovrà essere

poi compilato e convertito in un formato compatibile con gli ambienti di simulazione e di fault-simulazione. Nel caso di studio si tratta di un programma scritto in linguaggio assembler.

La seconda fase si individua nella simulazione del circuito, avendo cura di emulare il comportamento desiderato del dispositivo. E' necessario tenere in conto la conformità con le condizioni operative del sistema stesso (tra cui si può trovare la configurazione al tempo di reset), e questo si può ottenere mediante un opportuno testbench. Il risultato del passo di simulazione è il file *dump*[4] che verrà usato successivamente per fornire gli input allo step di fault-simulazione; questo registra il valore di ogni segnale nel tempo, sia quelli di input che quelli di output. A seconda del SoC bisogna selezionare un opportuno livello gerarchico a cui effettuare la simulazione, in modo che si risparmi tempo al passo successivo.

L'ultima fase consiste nel processo di fault-simulazione durante cui viene generata e processata una fault list che si basa su una descrizione a livello gate del SoC sotto esame, al livello gerarchico del sistema selezionato. A seconda del tool EDA[5] può essere necessaria una manipolazione del file *dump* fornito dal processo di simulazione, in modo tale da garantire la compatibilità con il fault-simulatore. Per ottenere valori di fault coverage che rispecchino la realtà, è necessario selezionare opportunamente i segnali che devono essere osservati, in modo tale da poter distinguere il corretto comportamento da quello viziato da guasti. Questo è un punto cruciale nell'intero flusso dato che una selezione errata può portare a misurazioni squilibrate. Quello che viene fatto a questo passo è quindi confrontare l'output gold, ossia in comportamento atteso dei segnali opportunamente selezionati, con l'output ottenuto a partire dall'iniezione puntuale di ogni fault presente all'interno della fault list. In questo modo si è in grado di capire se quel particolare guasto viene rilevato attraverso un comportamento anomalo.

[4]Nel caso di studio viene generato un file EVCD (Extended Value Change Dump).
[5]Electronic Design Automation.

Alla fine del processo di classificazione dei fault viene prodotto un report riguardante la copertura dei guasti rilevata. Per mostrare il valore di fault coverage effettivo bisogna però manipolare questo report rimuovendo quei fault che non possono essere rilevati attraverso una procedura SBST. Per esempio, i segnali appartenenti alle scan chain, utilizzate per i test manifatturieri, possono comportare dei guasti, ma questi non si tradurranno mai in un malfunzionamento del SoC sul campo.

1.4 Struttura programma di test

L'inclusione di routine SBST all'interno dell'ambito di lavoro del chip è una questione critica. Bisogna quindi tenere in conto le problematiche legate all'integrazione:

- la cooperazione con altri moduli software (e.g. il sistema operativo);
- il *context switching* e il monitoraggio dei risultati;
- la robustezza in caso di guasto, problema strettamente legato alla gestione degli interrupt.

Considerando la cooperazione con altri moduli, i programmi di test devono essere costruiti includendo funzionalità chiave che permettano al test stesso di essere lanciato, monitorato ed eventualmente interrotto da processi a più alta priorità. Il programma di test deve essere strutturato attentamente in modo da configurare le aree di memoria e le risorse dei periferici per scopi di testing. Prima di tutto, una soluzione praticabile e consigliata per lo sviluppo di sistemi operativi e moduli software, risiede nell'adozione, per ambienti di impiego automotive, dello standard EABI (Embedded-application binary interface) [12]. Questo definisce delle convenzioni standard per il formato dei file, il tipo dei dati, l'utilizzo dei registri, l'organizzazione dello *stack frame* ed il passaggio di parametri a funzione. Per questa ragione ogni programma di test include un prologo EABI e un epilogo EABI, posti rispettivamente all'inizio e alla fine del programma di test, in modo tale da salvare e ripristinare lo stato della normale esecuzione. Il frame EABI viene creato non appena il programma di test comincia, quindi qualsiasi scheduler è in grado di lanciare l'esecuzione del test in qualsiasi momento.

Inoltre si identifica nella *signature*, i.e. firma, quello strumento per cui si riesce a capire se il programma di test è stato in grado di rilevare un guasto o meno. Questa *signature* è rappresentata dalla concatenazione dei risultati delle istruzioni che si considerano rilevanti, per mezzo di un registro accumulatore a cui vengono successivamente sommati i valori ottenuti. In questo modo si verifica il valore del registro alla fine di una simulazione gold, ossia in assenza di guasti, e si definisce quindi il contenuto della firma. Questo risultato verrà riottenuto ogni qualvolta il programma di test non rilevasse alcun guasto, mentre invece non sarebbe coincidente nel caso di almeno un rilevamento. Prima dell'epilogo EABI, quindi, viene eseguito un controllo della firma, per comprendere di che tipo di comportamento si è in presenza, e comportarsi di conseguenza. In caso di rilevazione di fault quello che viene fatto è segnalare, attraverso un opportuno segnale, il comportamento erroneo del sistema. In questa situazione il comportamento standard è quello di portare il sistema in uno stato di sicurezza, come può essere uno stato di *halt*, oppure effettuare un reset a livello di chip e rieseguire il controllo delle funzionalità.

Ovviamente, oltre alle sezioni descritte in precedenza, è presente il corpo del programma di test. Questo deve essere redatto dal programmatore, selezionando, con l'opportuna strategia, le istruzioni e i pattern da applicare in modo tale da rendere visibile all'uscita la maggior parte dei comportamenti anomali che il modulo sotto esame può presentare. Questo non è chiaramente banale e la trattazione in questione ne è la prova.

Si possono sostanzialmente identificare tre tipologie di programma di test [8], classificabili in:

- *run-time*: possono essere interrotti da richieste del sistema, solitamente utilizzati per coprire moduli computazionali come quelli aritmetici;
- *non-exceptive*: non possono essere interrotti dato che richiedono la manipolazione di registri special purpose, solitamente utilizzati per testare il Register File;
- *critical*: lanciano intenzionalmente un'interruzione per testare moduli critici come quello delle eccezioni appunto.

I test run-time, come dice il nome, vengono eseguiti a run-time, quindi durante l'arco di tempo di utilizzo del chip nel suo ambiente di lavoro standard. Le altre due tipologie di test vengono invece eseguite a boot-time, ossia prima dell'avvio del sistema operativo che abilita le funzioni operative del chip. Questo perché questi test vertono su aspetti critici del processore, agendo su registri e moduli che a run-time non possono invece venire alterati.

1.5 Tecnica ATPG

La dimensione e la complessità dei circuiti reali rende molto complessa la generazione manuale di pattern. Sono state quindi inventate tecniche automatiche per l'identificazione di buoni test vector, tra queste si classificano i tool ATPG (Automatic Test Pattern Generation) quale è per esempio TetraMAX. Questi strumenti operano solitamente a livello gate ed hanno un'architettura composta dai macroblocchi individuabili in Figura 1.3.

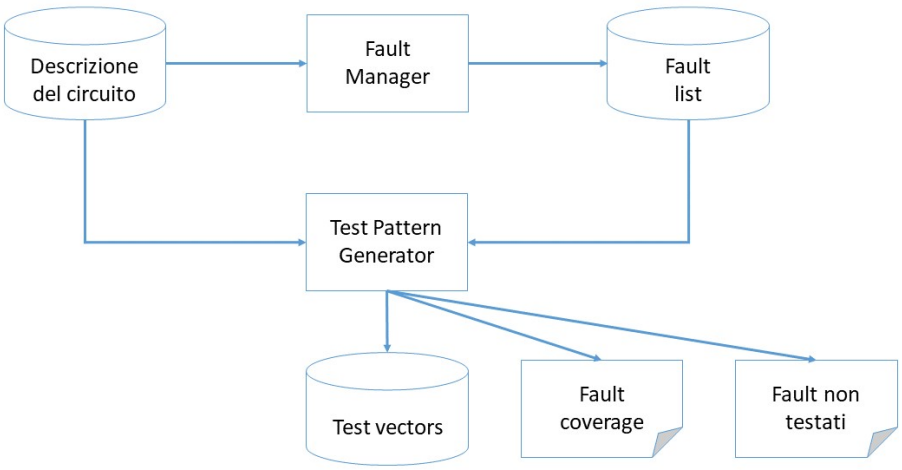

Figura 1.3: Architettura generale tool ATPG

Il Fault Manager è incaricato della generazione della fault list operando il collasso dei fault e l'identificazione di quelli che risultano non testabili. La fault list generata avrà quindi la caratteristica di contenere l'elenco di tutti quei possibili fault che potrebbero interessare il sistema, su cui poi la tecnica deve agire cercando di trovare

i pattern opportuni. Il Test Pattern Generator è fondamentalmente composto da due moduli: un modulo ATPG e un modulo di fault-simulazione. Quest'ultimo si occupa di iniettare nel circuito i guasti e verificare la copertura dei fault a fronte di specifici pattern applicati agli ingressi. L'output del tool ATPG risulta quindi essere composto da un set di test vector, il valore di fault coverage e l'elenco dei fault non testati.

Cercando di analizzare accuratamente la generazione dei test vector, si possono individuare i seguenti passi:

1. Generazione completa della fault list: svolta dal Fault Manager e riguardante, in teoria, anche una porzione di hardware, un singolo modulo ad esempio.

2. Collasso della fault list: svolto dal Fault Manager, prevede l'identificazione preliminare dei fault non testabili e consente la riduzione del costo degli step successivi utilizzando regole di equivalenza e di dominanza (meno frequenti).

3. Analisi di testabilità: produce informazioni statistiche circa la testabilità di ogni fault, rendendole disponibili ai passi successivi in modo da garantire, per esempio, l'ordinamento delle fault list ed una valutazione di controllabilità e di osservabilità dei punti interni.

4. Generazione del primo set di test vector.

5. Fault-simulazione: processo che consente l'identificazione dei fault detected eliminandoli dalla fault list, per proseguire in modo più veloce ai passi successivi.

6. Generazione dei test vector: è l'operazione più importante e più onerosa. Eseguita ciclicamente fino al raggiungimento di determinate condizioni di stop (e.g. tempo, copertura, ...).

7. Compattazione test vector: consente di ridurre, a volte considerevolmente, la durata e la dimensione dei test, senza però andare ad inficiare la fault coverage.

Per quanto riguarda la generazione dei test vector (passo 6), si possono citare le seguenti operazioni che la caratterizzano:

- selezione di un fault target;
- lancio della procedura ATPG per l'individuazione dei test vector relativi al fault target;
- esecuzione del Fault Simulator per operare il *fault dropping*[6].

Alla fine del processo ATPG, i fault contenuti nella fault list vengono quindi distribuiti trai seguenti macrogruppi: *detected*, *not detected*, *potentially detected* e *untestable*. Questa distinzione è però già stata analizzata a fondo nella Sezione 1.2.2.

1.5.1 TetraMAX

Per la trattazione in esame si farà riferimento, come già avvenuto in Sezione 1.2.2 per quanto riguarda i modelli di fault, alla logica e alle convenzioni adottate da TetraMAX [14], tool ATPG sviluppato da Synopsys. Questo programma genera automaticamente test pattern provvedendo un'unica soluzione ottimizzata per una vasta gamma di metodologie di test. Questo consente ai progettisti RTL di creare, velocemente e in modo efficiente, pattern di test per i design anche trai più complessi. Tra le caratteristiche principali possiamo individuare:

- supporto dei modelli di fault e creazione di pattern che tengano in conto le problematiche legate alla potenza;
- garanzia di rapidi incrementi di rendimento grazie all'isolamento delle funzionalità delle posizioni dove è presente un difetto;
- padronanza completa della regola di controllo di scansione e di compressione;
- integrazione di un simulatore dei guasti per la classificazione dei modelli di test strutturali;

[6] Eliminazione dalla fault list dei fault detected a questo passo.

- supporto di processori multicore per esecuzioni più veloci;

- interfaccia grafica integrata, comprensiva di una visualizzazione gerarchica dei moduli e di un visualizzatore di forme d'onda.

1.6 μGP per algoritmi evolutivi

μGP (MicroGP) [15] è un ottimizzatore versatile in grado di superare l'euristica umana convenzionale trovando la soluzione ottima di problemi complessi. Dato un compito, i.e. task, prima propone una serie di soluzioni casuali per poi raffinarle e migliorarle iterativamente. Il suo algoritmo euristico utilizza il risultato delle valutazioni combinato ad alcune informazioni strutturali interne per poter esplorare, in modo efficiente, lo spazio di ricerca, e per fornire la soluzione ottimale. L'applicazione originale di μGP è stata la creazione di programmi di assemblaggio per testare diversi microprocessori (da lì la lettera "μ" nel suo nome) e successivamente però ne è stato esteso l'impiego ad una vasta gamma di problemi tra cui: la creazione di programmi di test per la convalida pre e post silicio, la progettazione di reti bayesiane, la creazione di funzioni matematiche con rappresentazione ad albero, l'ottimizzazione intera e combinatoria, l'ottimizzazione dei parametri a valore reale ed anche la creazione di *corewar warrior*.

μGP sfrutta un algoritmo evolutivo (da lì l'acronimo "GP" di *genetic programming*) in cui la popolazione viene considerata in ogni fase del processo di ricerca e nuovi individui vengono generati attraverso meccanismi che emulano sia la riproduzione sessuata che asessuata. Le nuove soluzioni ereditano tratti distintivi da quelle esistenti e possono coesistere le buone caratteristiche individuate nei genitori. Le soluzioni migliori hanno una maggiore probabilità di riprodursi e riuscire a vincere la lotta simulata per l'esistenza.

L'utilizzo del tool che verrà fatto è strettamente legato alla creazione di programmi di test per microprocessori. Se ne riporta quindi, in Figura 1.4, lo schema di funzionamento legato alle finalità appena individuate con l'obiettivo di garantire valori di fault coverage elevati.

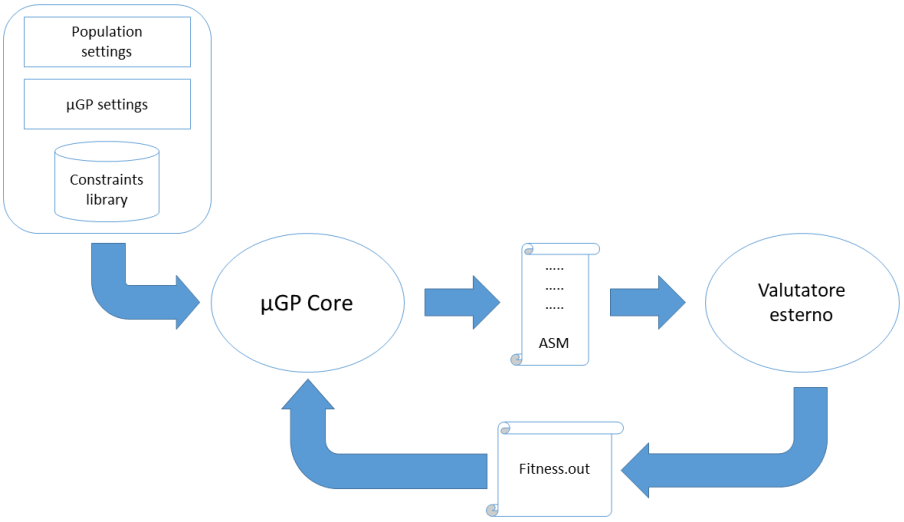

Figura 1.4: Schema di funzionamento del tool μGP

μGP riceve in input tre file che servono a determinare l'evoluzione dell'algoritmo. Definendo in anticipo come μGP dovrà evolvere, i.e. secondo quali parametri, si lascia quindi completa libertà al core del tool di esplorare lo spazio delle soluzioni, con la conseguente generazione di più individui figli, programmi assembler in questo caso. Questi ultimi andranno poi valutati da un valutatore esterno. Questo perché, a seconda dell'obiettivo che si sta perseguendo, bisognerà fornire valori di fitness, i.e. bontà, ad hoc. Quindi si dovrà anche andare a definire un valutatore che fornirà in output un file in cui saranno contenute proprio queste informazioni. Il valutatore deve eseguire le seguenti operazioni (in questo caso di studio le seguenti, ma in generale va adattato all'utilizzo che se ne vuol fare):

- compilazione del programma assembler;
- simulazione a livello gate con conseguente generazione del file EVCD;
- fault simulazione partendo da fault list definita in anticipo;
- produzione del file *fitness.out* riportando, per ogni riga, il nome dell'individuo e il rispettivo valore di fitness.

Per poter garantire l'esecuzione dell'algoritmo evolutivo si deve quindi procedere con la modifica dei vari file di configurazione, tutti aventi struttura XML per una più facile comprensione e redazione. Di seguito sono elencati i principali contenuti di ognuno di questi.

Il file ***μGP settings*** contiene le configurazioni generali per l'evoluzione tra cui il seed per la replicazione univoca dell'algoritmo e la lista dei file di *population settings*.

Il file ***Population settings*** contiene informazioni riguardanti gli individui che vengono utilizzati e generati dal processo evolutivo, tra cui:

- *mu*, dimensione della popolazione all'inizio e alla fine di ogni generazione;
- *nu*, dimensione della popolazione iniziale creata in modo casuale;
- *inertia*, utilizzata per regolare il meccanismo di auto-adattamento di μGP essendo che rappresenta la resistenza del sistema alla spinta verso l'assunzione di valori nuovi (range di valori tra 0 e 1);
- *lambda*, ossia il numero di operatori genetici applicati ad ogni passo;
- *sigma*, utilizzata per regolare la forza degli operatori genetici, ottenendo quindi variazioni più nette per valori alti;
- *fitness parameter*, numero di valori di fitness che il core deve ricevere in input per poter valutare ogni individuo, ossia solamente il valore di fault coverage in questo caso;
- parametri per la selezione degli individui genitori, questo utilizzando il metodo di default *tournamentWithFitnessHole* che può essere classificato come classico meccanismo di *tournament selection* con la possibilità di comparare individui anche tenendo conto della loro entropia e non solamente del valore di fitness (range di valori di *fitnessHole* tra 0 e 1, tenendo conto che con valori prossimi allo zero si utilizza con più probabilità la fitness). Si definisce inoltre un valore di *tau* medio, massimo e minimo, rappresentante il numero di genitori che vengono selezionati per l'evoluzione.

La ***Constraints library*** consta di una libreria contenente i vincoli principali per la struttura di ogni individuo. Grazie alla grande flessibilità di impostazioni si è in grado di rappresentare quasi tutti i tipi di individuo, dai più banali ai più complessi. E' presente un'intelaiatura per la definizione del singolo individuo, la quale può suddividersi in:

- definizione dei tipi, ossia dove vengono dichiarati i tipi delle variabili e delle costanti che poi verranno utilizzate nel corpo del programma vero e proprio;
- prologo, ciò che va in testa al programma;
- epilogo, ciò che va in fondo al programma;
- sezioni, che si compongono a loro volta di un prologo, un epilogo e di una o più sottosezioni;
- sottosezioni (all'interno delle sezioni), composte anche loro da un prologo e un epilogo, ma anche da macro;
- macro, ossia definizioni di blocchi di codice che constano di un'espressione, i.e. codice con alcuni *placeholder* che vengono rimpiazzati durante la generazione, e della definizione dei parametri, i.e. una porzione dove si fa riferimento al tipo globale definito in precedenza in modo tale da poterlo riusare localmente all'interno dell'espressione (nota: si può anche impostare un numero minimo e massimo di ripetizioni della macro stessa, variando tutte le volte il singolo blocco sulla base delle sostituzioni che vengono fatte dei *placeholder*).

1.7 Approccio manuale

Questa tipologia di approccio al problema risulta utile quando le tecniche automatiche presentano i loro limiti. In queste situazioni l'unica soluzione perseguibile risulta quella di compiere un'analisi puntuale dell'UUT (Unit Under Test), studiando il circuito a entrambi i livelli di astrazione possibile, ossia RTL-level e gate-level. In quest'ultima situazione è possibile andare ad analizzare direttamente la descrizione VHDL/Verilog fornita dal produttore, e quindi cercare di trarre informazioni

dal comportamento funzionale, dal nome assegnato ad alcuni segnali e da eventuali commenti dei progettisti hardware.

Secondariamente, ma non per importanza, ci si basa sull'analisi dei risultati di fault coverage ottenuti, cercando di impostare strategie sempre migliori. Questo è fatto fondamentalmente in simulazione, ossia analizzando il valore dei segnali nel tempo, dipendenti dallo stato del processore e dal flusso di esecuzione. Inoltre, per avere informazioni aggiuntive circa le modalità secondo cui si stanno variando i valori dei singoli segnali, esistono tool che registrano il risultato di una simulazione proponendo statistiche su questi ad un livello macroscopico che consenta, attraverso un'osservazione accurata da parte del programmatore, di rilevare alcuni tipi di comportamento.

In questa sezione verranno illustrate a grandi linee le funzionalità di due tool che sono stati utilizzati nel corso delle procedure di testing manuale.

1.7.1 NCSim

NCSim è una suite di strumenti sviluppata da Cadence Design Systems ed opera nel campo della progettazione e della verifica di ASIC, SoCs e FPGAs. NCSim è il nome del motore di simulazione del core, anche se il nome della generica suite è Incisive. I programmi utilizzati in questo contesto risultano quindi essere principalmente:

- NCSim: motore di simulazione unificato per Verilog, VHDL e SystemC. Consente il caricamento di immagini istantanee generate da NC Elaborator. Questo tool può essere eseguito sia in modalità grafica che da riga di comando batch.
- Sim Vision: Un visualizzatore grafico autonomo di forma d'onda e tracer di netlist, ossia in grado di tenere traccia del valore assunto da tutti i segnali nel corso del tempo.

In generale, essendo un tool particolarmente ampio e complesso, ne viene qui riportata solo una delle utilità più sfruttate per lo studio in esame. L'impiego di NCSim ha consentito la visualizzazione a schermo dei vari segnali presenti all'interno

del core. Se ne può avere un esempio in Figura 1.5. In questo modo si è in grado di visualizzare sia la logica di controllo che il datapath per riuscire a capire quando determinate istruzioni vengono eseguite e dove. Inoltre, cosa più importante, capire i valori assunti dai segnali in uscita ai moduli in esame, per poterne verificare in una prima fase il comportamento, e in seconda battuta la corretta e completa esecuzione del programma di test realizzato.

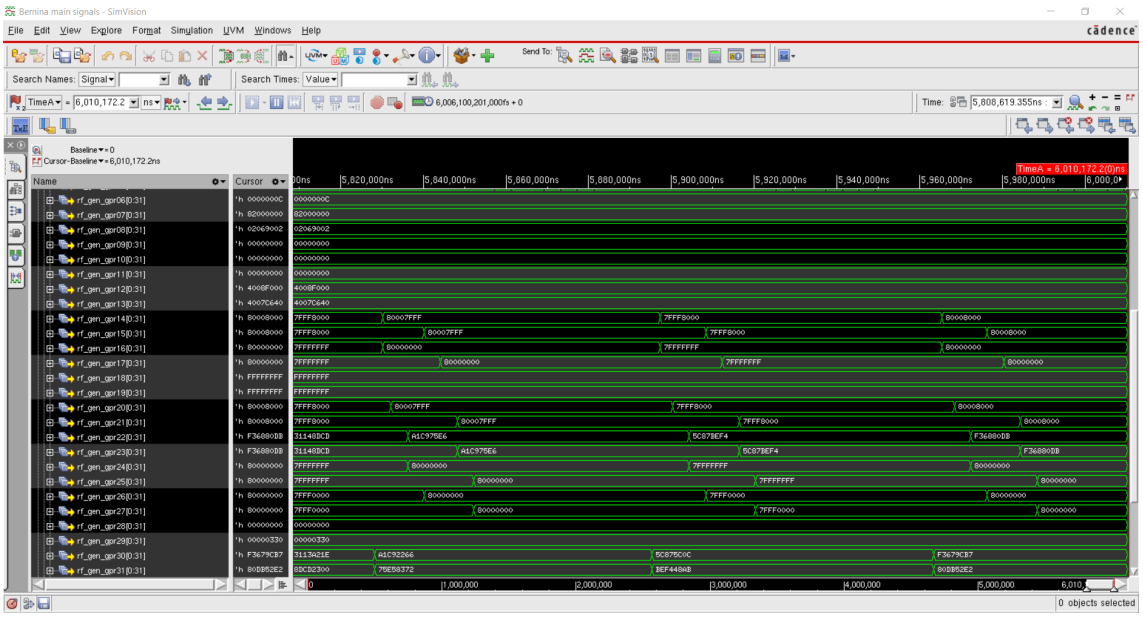

Figura 1.5: Schermata di Sim Vision per la visualizzazione di forme d'onda

Altre funzioni di NCSim prevedono, come già detto, la visualizzazione dell'effettivo codice Verilog, ma anche l'osservazione RTL-level e gate-level dei vari moduli presenti nel chip. Quest'ultima caratteristica consente di visionare proprio i singoli segnali, consentendo quindi di seguire il flusso logico e le interconnessioni tra le varie unità.

1.7.2 IMC

IMC (Integrated Metrics Center) è un tool di analisi progettato da Cadence per caricare e analizzare le coperture relative ad alcuni parametri RTL-level. Provvede una ricca interfaccia grafica (come si può vedere in Figura 1.6) per i vasti valori di copertura che si possono visualizzare, dalla copertura di codice a quella funzionale. Con IMC viene fornito un ambiente unificato per l'analisi che, opportunamente combinato con un database contenente i valori di copertura sufficientemente potente, garantisce interoperabilità tra i vari tool funzionali di verifica di Cadence. IMC consente di convergere ai valori di copertura dei fault desiderati in maniera più veloce rispetto ai report tradizionali HTML o di puro testo. Consente un'analisi real-time con un'interfaccia grafica intuitiva e suddivisa in viste per ogni tipo di specifica copertura supportata. Supporta tutte le metriche standard RTL garantendone il funzionamento sia fondato su singola esecuzione che su più esecuzioni unite insieme.

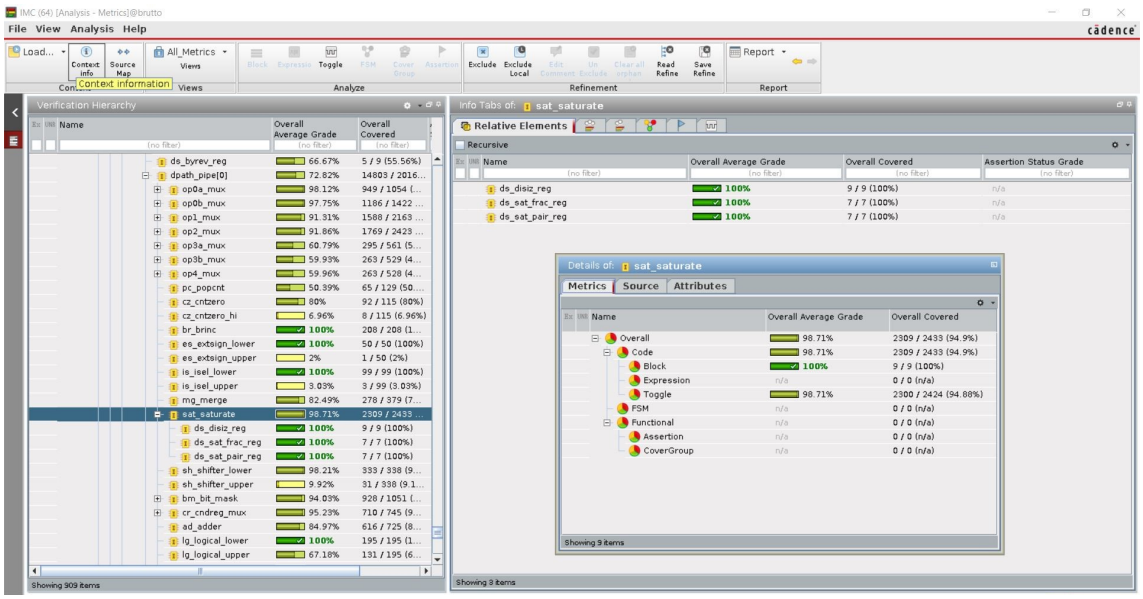

Figura 1.6: Schermata di IMC con le informazioni relative ad una generica esecuzione

I principali tipi di informazione estrapolabili da IMC sono i seguenti:

- copertura di blocchi di codice, verificando che il programma eseguito passi in ogni ramo decisionale possibile con più iterazioni, in modo tale da consentire una copertura totale del codice scritto;
- copertura delle espressioni, ossia che si assumano, ad iterazioni differenti, tutte le combinazioni degli input che consentano il verificarsi di determinate condizioni controllate nel codice;
- *toggle activity*, consentendo di capire se tutti i segnali interessati dall'esecuzione del codice commutino il loro valore con transizioni 0→1 e 1→0, in modo che si possano effettivamente verificare delle variazioni dei segnali stessi sulla base dell'esecuzione dei programmi di test;
- valori di copertura di gruppo, relativi a risultati ottenuti da più esecuzioni, fornendo informazioni su come determinati risultati siano stati raggiunti da programmi di test differenti.

Capitolo 2

Caso di studio

Questo capitolo introduce l'architettura caso di studio partendo da un punto di vista più macroscopico, analizzando quindi singolarmente le unità funzionali oggetto di testing.

2.1 Overview SoC Bernina (SPC58NN84)

Il system-on-chip a cui si riferisce la trattazione è l'SPC58NN84 (Bernina), progettato da STMicroelectronics. La famiglia di chip SPC58X risponde alle necessità, sempre crescenti negli ultimi anni, da parte del settore automotive, legandosi in maniera molto forte alla sicurezza e all'affidabilità del prodotto. Le principali caratteristiche di questa famiglia di system-on-chip possono essere individuate in:

- architettura del processore multicore;
- core *dual issue*, ossia che ad ogni ciclo di clock il processore è in grado di spostare due istruzioni da uno stadio della pipeline ad un altro. Questo comportamento dipende esclusivamente dal processore, ma sostanzialmente significa che questo è in grado di sostenere l'esecuzione di due istruzioni per ogni ciclo di clock;
- supporto di istruzioni di tipo Variable Length Encoding (VLE) [16] e di tipo Lightweight Signal Processing (LSP) [17];
- presenza di due core principali affiancati entrambi da Checker Core, con l'obiettivo di replicare l'operato svolto dai primari verificandone l'operato. Questo è un comportamento molto importante in ambienti safety-critical, in cui il risultato delle operazioni deve essere affidabile;

- presenza di un Hardware Security Module (HSM) per garantire integrità funzionale delle Electronics Control Units (ECUs), rilevamento delle intrusioni e protezione contro attacchi maligni provenienti dalla rete, e.g. Ethernet o interfaccia CAN;
- utilizzo di specifiche unità per l'accelerazione di operazioni di signal processing per operazioni DSP (Digital Signal Processing). Questo attraverso istruzioni LSP APU, progettate a tale scopo;
- frequenza di lavoro del microcontrollore da 4MHz a 180MHz;
- design in tecnologia a 40nm;
- embedded flash da 1MB, 2MB, 4MB o 6MB, con correzione ECC (Error Correction Code);
- embedded RAM (condivisa e locale per ogni core) da 256KB o 512KB, con correzione ECC (Error Correction Code);
- embedded instruction cache e embedded data cache da 1KB, 2KB o 4KB;
- modulo Nexus per debug attraverso porta JTAG.

La famiglia SPC58X è basata sul paradigma RISC (Reduced Instruction Set Computer) con architettura di processore PowerPC. Da un punto di vista macroscopico si possono individuare una coppia di *crossbar switch*[1], come mostrato in Figura 2.1. Ogni crossbar switch è associato ad un core, ad una memoria RAM locale e a differenti periferici. Oltre alla memoria RAM locale è presente anche una memoria condivisa accessibile da ogni core. Allo stesso modo anche ogni periferico può essere acceduto da ogni core, pagando però un tempo di accesso maggiore se non connesso direttamente al proprio crossbar switch.

[1] I *crossbar switch*, anche detti *matrix switch*, sono costituiti da una collezione di interruttori organizzati in una configurazione a matrice. Tra le linee multiple di input e di output si viene a formare un incrocio di linee interconnesse tra cui può essere stabilita una connessione chiudendo un interruttore, i.e. uno *switch*, il quale è presente ad ogni intersezione. Sostanzialmente la funzione svolta è quella di un normale bus.

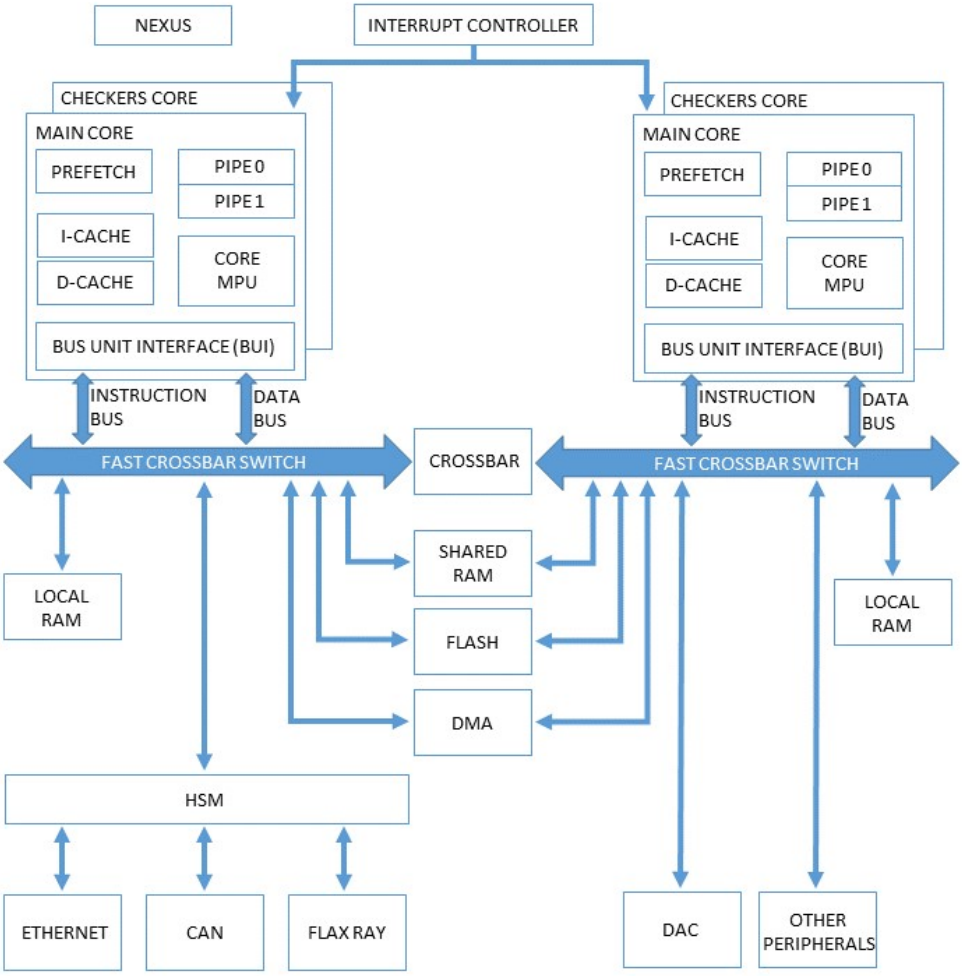

Figura 2.1: SoC automotive della famiglia SPC58X di STMicroelectronics

2.2 Overview core e200z4256n3

I core presenti all'interno del chip Bernina sono del tipo e200z4256n3, un genere di microprocessore della famiglia e200z che implementa una versione a basso costo dell'architettura PowerISA 2.06. Il core è dual issue[2] e supporta le istruzioni VLE (Variable Length Encoding) ISA attraverso i registri general purpose a 32 bit, garantendo una maggiore densità di codice. Le istruzioni base PowerISA 2.06 a 32 bit a lunghezza fissa non sono direttamente supportate; lo sono invece quelle LSP (Lightweight Signal Processing) APU che consentono il supporto a operazioni real time di tipo SIMD (Single Instruction Multiple Data). Tutte le istruzioni aritmetiche operano su dati provenienti dai registri general purpose, ed è supportato inoltre l'accoppiamento dei registri in modo tale da supportare le istruzioni a 64 bit definite dallo standard LSP APU, le quale forniscono risultati vettoriali e scalari. E' consentita una grande varietà di tipologie di manipolazione di dati, moltiplicazione, accumulazione e prodotto scalare, tutte basate su un accesso specializzato in memoria tale da garantire fino a 16 moltiplicazioni con accumulazione per ciclo, lavorando con registri a 16 bit.

Il core e200z4256n3 integra inoltre:

- una unità Embedded Floating-point (EFPU2) APU che consente operazioni floating point real time a singola precisione[3], sempre tramite il supporto dei registri general purpose;
- una coppia di unità di esecuzioni intere;
- una unità di controllo per i salti condizionali;
- una unità adibita al fetch delle istruzioni;
- una unità load/store basata su architettura completamente a pipeline, con 2 cicli di latenza in fase di load e con supporto ad accessi non allineati;

[2] Vedi definizione alla Sezione 2.1.

[3] Operazioni a virgola mobile basate su 32 bit; la rappresentazione prevede l'impiego di un bit di segno e i restanti bit da dividersi per la parte relativa alla caratteristica e alla mantissa.

- un register file *multi-ported*[4] capace di sostenere sei letture e tre scritture per ciclo di clock;
- una Instruction Cache da 16KB e una Data Cache da 8KB, entrambe con architettura 2-way set associative[5];
- una Memory Protection Unit che consente la protezione di diverse aree di memoria adibite al salvataggio di istruzioni e dati;
- bus di sistema istruzioni e dati a 64 bit e basati su protocollo Dual AHB (Advanced High-performance Bus)[6].

Infine sono presenti delle accortezze anche dal lato della gestione energetica come un design low power basato su tre differenti modalità di risparmio energetico: *halt*, *stop* e *wait*.

Facendo invece un riassunto di quelle che sono le principali caratteristiche architetturali, possiamo trovare:

- la pipeline delle istruzioni presenta cinque stadi, di cui due per l'esecuzione;
- la maggior parte delle operazioni aritmetiche e logiche sono eseguite in un singolo ciclo di clock con l'eccezione della moltiplicazione, che è implementata con un vettore hardware con struttura a pipeline, e della divisione;
- le operazioni di salto condizionale eseguono in parallelo il calcolo dell'indirizzo destinazione del salto e la decode dell'istruzione; ciò si traduce in un tempo di esecuzione di due cicli di clock per salti correttamente predetti, e in un tempo di un ciclo di clock per salti non eseguiti;
- le istruzioni di load e store sono provvedute per byte, halfword e word; inoltre queste vengono messe in pipeline per consentire un throughput di un singolo ciclo.

[4]Unità di memoria con porte multiple in lettura e scrittura.
[5]Una particolare posizione di memoria può essere caricata in cache in due distinte posizioni nella cache di livello 1.
[6]L'interfaccia dual master minimizza l'overhead di performance sul bus di sistema.

Si riporta, inoltre, nella Figura 2.2 sottostante, un diagramma a blocchi semplificato dell'architettura del core.

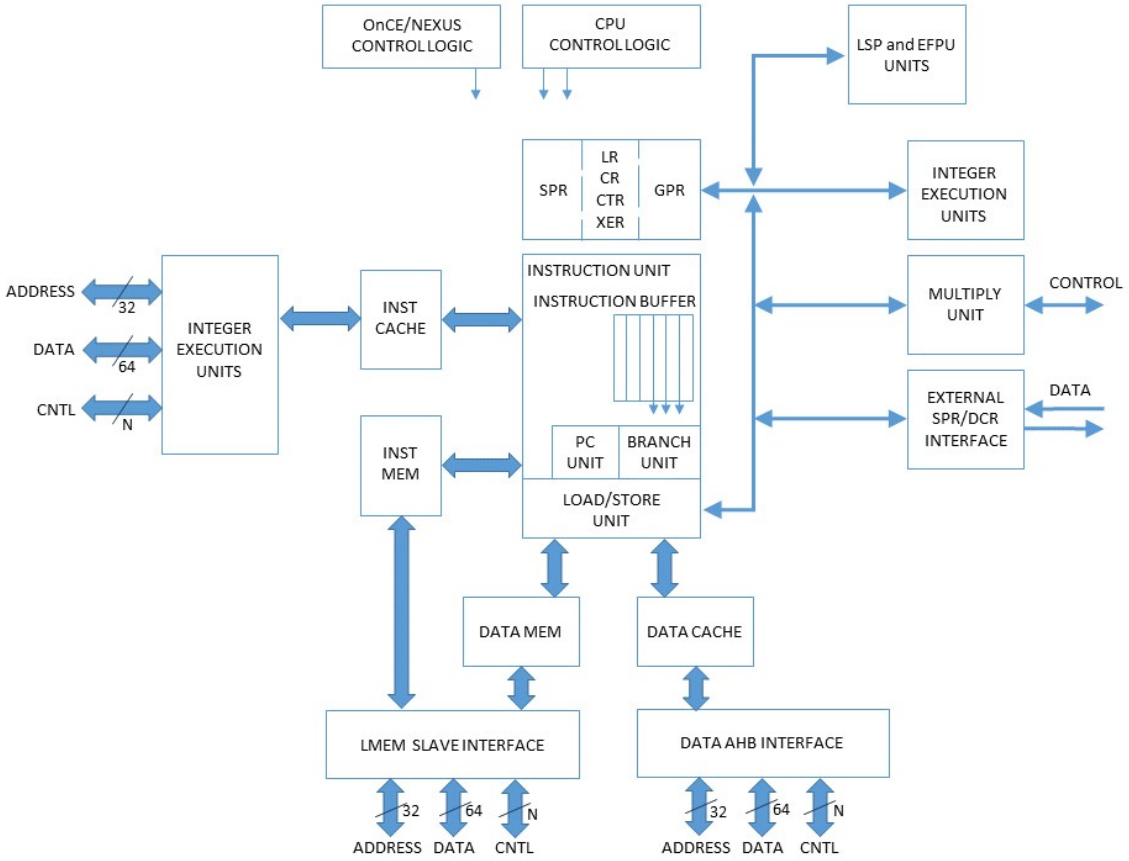

Figura 2.2: Diagramma a blocchi core e200z4256n3

Il modello dei registri su cui si appoggia l'architettura PowerISA 2.06 è quasi totalmente supportato dal processore in esame. I registri possono sostanzialmente essere divisi in due tipologie:

- registri livello utente: registri accessibili da tutto il software, sia che esso abbia privilegio utente o supervisor. Sono inclusi i seguenti:
  - ◊ General Purpose Registers: i 32 registri a 32 bit (GPR0-GPR31) costituiscono i registri sorgente e destinazione per ogni tipo di operazione e forniscono i dati per la generazione degli indirizzi;

  - ◊ Condition Register (CR): registro a 32 bit costituito da otto campi da 4 bit che riflettono i risultati di certe operazioni aritmetiche e provvede un meccanismo per testarne la validità e compiere salti sulla base del risultato di tale test;
  - ◊ Integer Exception Register (XER): registro che tiene traccia di overflow e di eventuali carry[7];
  - ◊ Link Register (LR): registro che fornisce l'indirizzo destinazione per istruzioni di salto condizionale che prevedono il suo impiego;
  - ◊ Count Register (CTR): registro che contiene un contatore, tipicamente utilizzato decrementandolo, utilizzato per la realizzazione di cicli che possono concludersi con un salto, e quindi con la riesecuzione del loop stesso, o con la prosecuzione lineare all'interno del codice.

- registri livello supervisor: registri accessibili esclusivamente da codice di livello supervisor. Sono inclusi i seguenti registri:
  - ◊ registri di controllo del processore: numerosi registri tra cui si può evidenziare il Machine State Register (MSR) che definisce lo stato corrente del processore, consentendo l'abilitazione e la disabilitazione di alcune funzionalità dello stesso;
  - ◊ Storage Control Register: registro che fornisce l'identificativo del processo o del task in esecuzione;
  - ◊ Thread Management Configuration Register: registro read-only che indica la configurazione di threading supportata;
  - ◊ Interrupt Registers: registri per la gestione puntuale delle procedure di gestione degli interrupt;
  - ◊ Debug Facility Register: registri per la corretta gestione delle procedure di debug.

E' possibile osservare, in Figura 2.3, uno schema comprensivo dei registri utilizzabili in modalità supervisor.

[7] Riporti di operazioni aritmetiche.

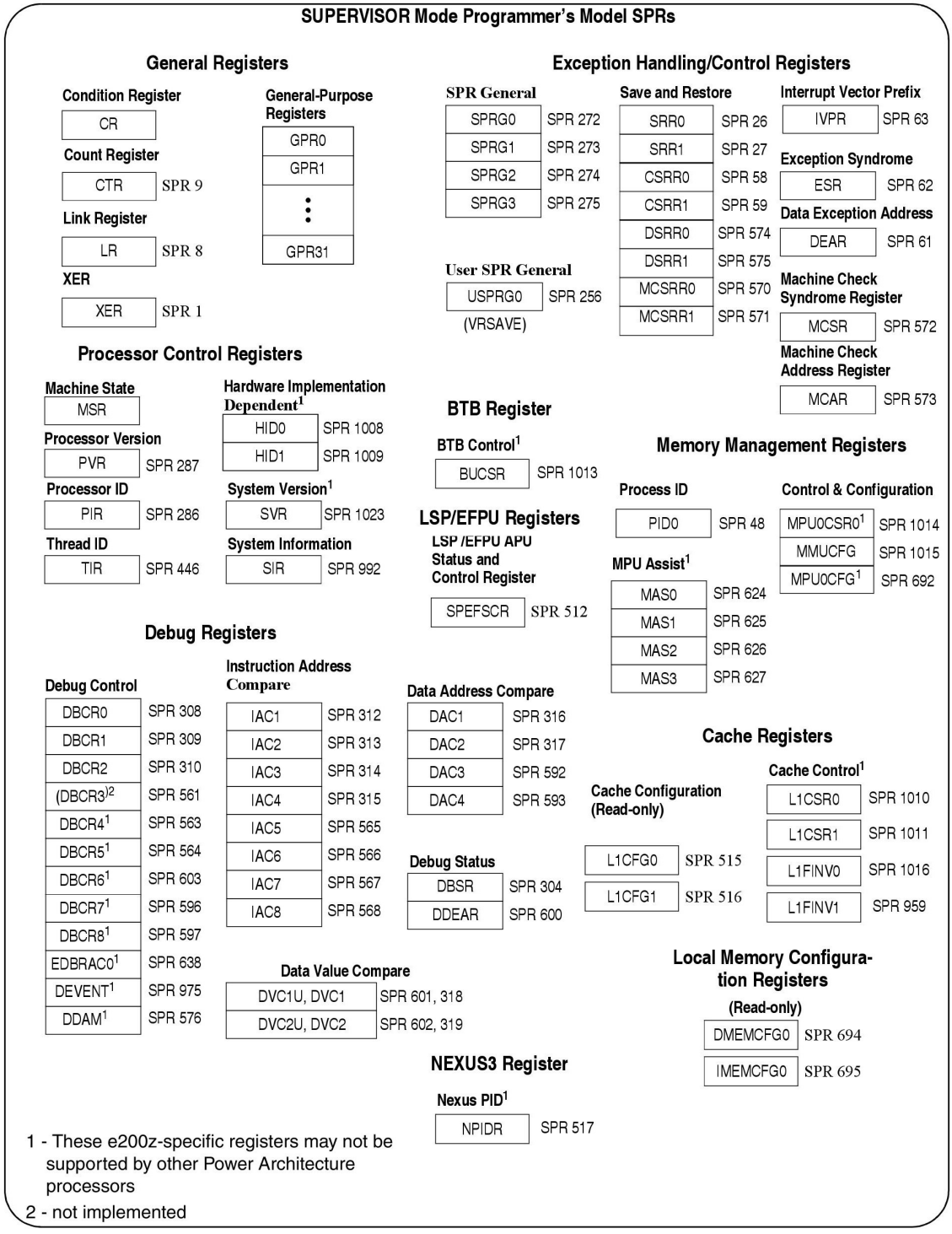

Figura 2.3: e200z4256n3: modello registri special purpose in modalità supervisor

2.3 Moduli caso di studio

All'interno del core del chip sono stati analizzati differenti moduli con l'obiettivo di innalzare la fault coverage, i.e. la copertura dei guasti, a valori il più prossimi possibile ad una copertura totale, chiaramente irraggiungibile. La copertura target per un generico modulo è fissata ragionevolmente ad un 80%, con differenze di trattamento a seconda della specifica unità in esame.

I valori di partenza da cui parte l'analisi sono prossimi allo 0%, con una minima copertura già garantita da altri programmi di test relativi ad altri moduli. Questo comportamento è da considerarsi assolutamente nella norma data la condivisione dei bus tra moduli differenti e l'esecuzione di alcune istruzioni (o parti di istruzioni) da parte di moduli che apparentemente non sono legati all'operazione richiesta. Questo è comune per particolari esigenze del processore stesso, che può necessitare di risultati aggiuntivi per ottimizzazioni future, oppure a fronte di una parallelizzazione dell'attività richiesta.

In ogni caso verranno presi in esame solamente i valori di fault coverage ottenuti mediante la sola esecuzione dei programmi di test sviluppati ad hoc per i vari moduli. Questo sia per sottolineare i risultati ottenuti che per avere una visione incrementale dei valori raggiunti, indipendente dallo sviluppo parallelo di altri programmi di test che potrebbero falsare, in qualche modo, la presentazione dei risultati. Gli esiti complessivi verranno presentati solo in fase conclusiva per evidenziare se siano stati raggiunti gli obiettivi prefissati, in modo tale da garantire una copertura complessiva adeguata.

In questa sezione verranno descritti da un punto di vista architetturale i vari moduli che sono stati testati, sottolineandone la logica di funzionamento e la loro collocazione all'interno del core, di cui, a livello macroscopico, si possono distinguere fondamentalmente due parti:

- la logica di controllo, ossia quell'insieme di segnali che vengono generati dal processore per comandare i vari blocchi, fornendo al momento esatto gli opportuni segnali di controllo;

- il datapath, ossia quei blocchi comandati dalla logica di controllo che svolgono azioni sulla base di quelli che sono i segnali in ingresso, costituendo, di fatto, il vero cuore operazionale del processore.

Essendo in presenza di un processore dual issue, si possono individuare, a livello di core, due differenti pipe che, grazie alla duplicazione di alcuni moduli, garantiscono una certa parallelizzazione delle operazioni. Tra le varie unità che verranno analizzate in seguito i moduli brinc e saturate sono presenti in entrambe le pipe con due entità quasi identiche, mentre il modulo circular buffer risulta comune a entrambe le pipe. Nel caso dei primi due chiaramente questo si riduce ad avere due differenti valori di copertura dei guasti, con annesse anche alcune problematiche legate al riuscire a forzare l'esecuzione delle istruzioni su entrambe le pipe.

2.3.1 Brinc

Il modulo brinc è un modulo facente parte del datapath e svolge operazioni di incremento *bit-masked*[8] dopo l'esecuzione di un'operazione di inversione dei bit[9] [10]. La sua funzione è fondamentalmente quella di velocizzare operazioni che accedono a buffer e strutture dati che necessitano di inversione di bit, tipico approccio seguito da algoritmi di DSP (Digital Signal Processing) o di FFT (Fast Fourier Transform). Per esempio, a un buffer di otto elementi salvato in memoria con ordinamento (0,1,2,3,4,5,6,7) si fa tipicamente accesso con ordine *bit-reversed* (0,4,2,6,1,5,3,7). Proprio per la loro peculiarità di accelerare accessi in memoria, le operazioni di brinc sono tipicamente seguite da istruzioni di load o di store, in modo tale che il risultato dell'istruzione stessa possa essere usato come valore indice per il successivo accesso in memoria, ottenendo, in questo modo, il dato seguente (considerando l'ordine di accesso *bit-reversed*) [17]. C'è comunque da dire che, in molti casi, gli algoritmi di FFT sono implementati all'interno del processore da numerosi registri general purpose, rendendo di fatto non necessario l'utilizzo di operazioni di load dalla memoria.

[8] Si sfrutta una maschera di bit per definire quali sono i bit soggetti all'operazione da svolgere.
[9] Es. da 0xFFF1 a 0x8FFF.

2.3.2 Circular buffer

Il modulo circular buffer è un modulo facente parte del datapath. Il suo scopo è quello di supportare la modalità di indirizzamento circolare, appoggiandosi ad una opportuna struttura dati quale è un buffer circolare, il quale garantisce tempi di accesso ottimizzati a strutture tipo vettori. Questa operazione si basa fondamentalmente su operazioni di incremento o decremento di un dato indice. Quando l'indice raggiunge il valore massimo all'interno del vettore, i.e. punta all'ultimo elemento del vettore stesso, un successivo incremento riporta il valore dell'indice a 0, i.e. punta al primo elemento del vettore. Stesso ragionamento a parti invertite per il decremento dell'indice partendo dal primo elemento, azione che lo sposta sull'ultimo. In Figura 2.4 è riportato un esempio di comportamento nel caso in cui si parta dal secondo elemento del vettore, i.e. indice 1, e si decrementi il valore dello stesso di 3 unità; il risultato è che, considerando un buffer di 8 elementi, si ottenga un puntatore al settimo elemento del vettore, i.e. indice 6.

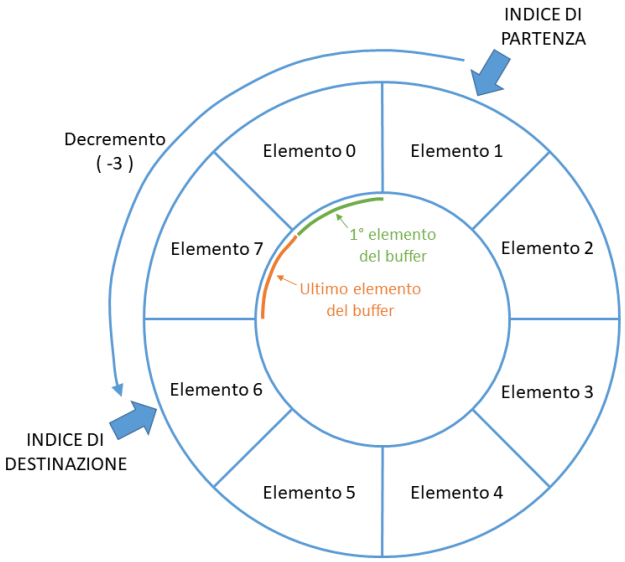

Figura 2.4: Esempio: buffer circolare

La modalità di indirizzamento circolare necessita che gli elementi del vettore siano allineati su indirizzi doubleword con la dimensione del buffer che possa andare da 8B a 8KB. Per esempio, l'istruzione ***zcircinc*** [17] è adibita proprio allo svolgimento dell'incremento per quanto riguarda un buffer circolare. Per compiere l'operazione sono necessari: l'indice all'interno del buffer, delle informazioni di controllo e l'offset (con segno) da sommare all'indice corrente.

2.3.3 Saturate

Il modulo saturate è un modulo aritmetico facente parte del datapath che svolge particolari istruzioni operanti su uno o due operandi. L'obiettivo generale è quello di ottenere la saturazione entro una certa soglia di un valore in ingresso o, più complesso, del risultato di un'operazione che opera su valori arbitrariamente con e senza segno, tenendo indifferentemente in considerazione ogni tipo di dato, quindi doubleword, word, halfword e byte.

Per saturazione si intende quell'operazione matematica per cui un numero viene limitato all'interno di un intervallo compreso tra un valore minimo e un valore massimo. Se questo è maggiore del massimo viene impostato al valore massimo, se minore del minimo, viceversa, è impostato al valore minimo.

Le operazioni di saturate [17] vanno ad aggiornare i flag di overflow (OV) e di summary overflow (SO) all'interno del registro XER (Integer Exception Register), adibito alla segnalazione di carry e overflow per operazioni di tipo intero. Mentre il bit di overflow viene settato per ogni operazione aritmetica, riferendosi quindi solamente all'avvenuto scatenamento di overflow da parte dell'ultima istruzione, il bit di summary overflow è impostato a 1 nel caso in cui si verifichi overflow la prima volta, e rimane poi invariato se questo non si verifica più; porta quindi con sé, nel tempo, l'informazione dell'avvenuto verificarsi dell'evento. In particolari situazioni questo può essere utile per sapere se si è verificato overflow all'interno di una certa sezione di codice, ricordandosi però di settare a 0 il valore del flag SO all'inizio di tale parte. Solo il programmatore può fissare il valore a 0 in maniera puntuale, i.e. se il valore venisse settato una volta a 1 dopo un'altra istruzione rimarrebbe invariato nel tempo.

Si possono inoltre distinguere diverse tipologie di istruzioni supportate dal processore in esame che sfruttano l'operato del saturate:

- operazioni a singolo operando in ingresso il cui valore è saturato sulla base di:
  - ◊ SS[0:1]: il risultato deve essere su byte (SS=00), halfword (SS=01) o word (SS=10);
  - ◊ IU[0]: il valore in ingresso deve considerarsi signed (IU=0) o unsigned (IU=1);
  - ◊ OU[0]: il valore in uscita deve essere signed (OU=0) o unsigned (OU=1);
  - ◊ SA[0]: se SA=1 va saturato il valore assoluto dell'operando in ingresso.

  L'impiego è quello di consentire la saturazione ai valori massimi possibili per i bit considerati validi in uscita, partendo da numeri variabili in ingresso. I segnali sopraccitati sono integrati nell'opcode, andando a costituire, quindi, istruzioni differenti;

- operazioni di saturazione e di arrotondamento a due operandi in ingresso, i quali vengono combinati a seconda dell'istruzione con una somma, una sottrazione, ecc. L'output viene saturato poi entro certe soglie;

- operazioni di *pack* e *unpack*, ossia l'elaborazione che viene fatta per consentire l'invio di dati non contigui; questi vengono "impacchettati" prima dell'invio e "spacchettati" dopo la ricezione, garantendo l'integrità dell'informazione. Queste funzioni si appoggiano anche al modulo di saturate a cui fanno svolgere alcune operazioni.

Capitolo 3

Tecniche di sviluppo di programmi di test

Questo capitolo analizza le tecniche di sviluppo di programmi di collaudo adottate per il testing dei moduli caso di studio. La trattazione predilige il riepilogo degli eventi in ordine temporale lineare, in modo da sottolineare le problematiche incontrate e le soluzioni adottate.

3.1 Tecnica ATPG per brinc e circular buffer

In questa sezione verrà analizzata la strategia comune adottata per testare i moduli brinc e circular buffer. Il metodo prevede di utilizzare i pattern generati da un tool ATPG (Automatic Test Pattern Generation) per la creazione di programmi di test, come descritto in Figura 3.1.

Inizialmente, viene letta la netlist[1] e viene costruito il modello dell'UUT (Unit Under Test), quindi viene generata la fault list associata a tale modulo. Successivamente viene selezionata un'istruzione eseguita dal modulo in questione e, partendo dal codice operativo della stessa, ne vengono forzati i segnali in ingresso corrispondenti, in modo tale da imitare il normale comportamento del processore. Come si può notare in Figura 3.2, vengono invece lasciati liberi i segnali associati agli operandi, in modo tale che il tool ATPG possa variarli trovando, a seconda dei parametri impostati, una soluzione ragionevolmente buona. Dopo la generazione, il tool ATPG fornisce un risultato che indica il numero di pattern applicati e il successivo incremento della copertura dei fault che si ottiene con la loro applicazione. Se la percentuale di copertura non è sufficientemente buona (non si ritengono risultati

[1] Insieme delle connessioni (net) elettriche di un circuito elettronico.

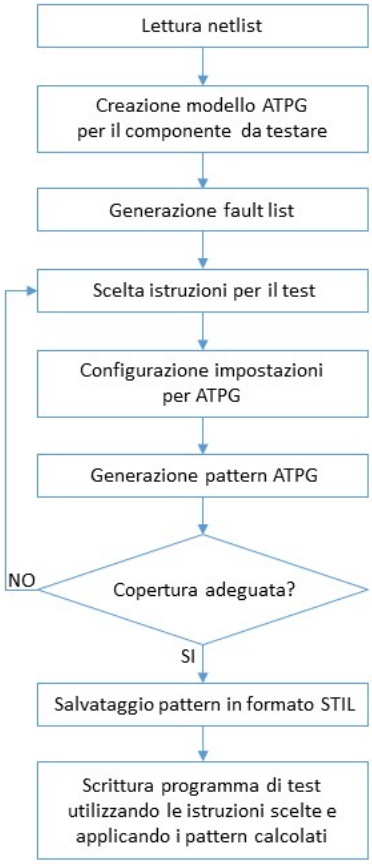

Figura 3.1: Passi per la generazione di un programma di test basato su ATPG

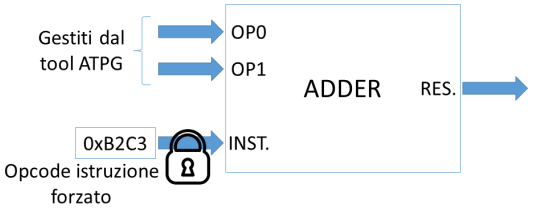

Figura 3.2: Esempio: valore segnali per applicazione tecnica ATPG a modulo adder

soddisfacenti valori al di sotto di un 70%–75%) o sono richiesti troppi pattern per il raggiungimento di un valore alto, allora bisogna procedere con la variazione delle condizioni iniziali, tra cui si può annoverare la scelta della specifica istruzione su cui il test ATPG si può effettuare. Infine, quando si raggiungono i risultati sperati (se chiaramente questi vengono raggiunti), si può passare alla scrittura manuale di un programma di test che esegua la/le istruzione/i selezionate in precedenza, applicando agli ingressi i pattern generati dal tool. Si può avere un'idea del programma risultante, di cui si riporta uno pseudocodice per semplicità, in Figura 3.3.

```
ADDER_TEST:

PROLOGO CODICE EABI

MOVE    RI, 0   ;Index Register
MOVE    R31, 7  ;registro contenente il numero di coppie di pattern

loop:   ;Caricamento operandi nei registri R0 e R1
        LOAD    R0, INDIRIZZO_SEZIONE_DATI + offset_op0
        LOAD    R1, INDIRIZZO_SEZIONE_DATI + offset_op1

        ;Applicazione operandi al modulo adder
        ADD     R2, R0, R1      ;R2 = registro destinazione

        ADD     R2 ALLA FIRMA (SIGNATURE)

        AGGIORNAMENTO offset_op0
        AGGIORNAMENTO offset_op1

        INC RI
        COMPARE RI A R31
        BRANCH_NOT_EQUAL loop   ;se non sono stati usati tutti i pattern salta

CHECK_SIGNATURE ;controllo valore firma

EPILOGO CODICE EABI

RETURN



;Sezione dati con 7 coppie di pattern per op0 e op1
SEZIONE_DATI:
.short  0x4BC2  ;op0 pattern1
.short  0xCC39  ;op1 pattern1
.short  0x19A6  ;op0 pattern2
.short  0xD362  ;op1 pattern2
.short  0xC331  ;op0 pattern3
.short  0x5891  ;op1 pattern3
 ...     ...         ...
```

Figura 3.3: Esempio: programma di test per applicazione tecnica ATPG a modulo adder

Dopo la redazione del programma di test va valutata la fault coverage e quindi, sulla base del risultato, decidere se mantenere il programma di test appena scritto in presenza di un risultato soddisfacente. Se il programma realizzato costituisce il primo della serie, le coperture date dalla tecnica ATPG verranno rispettate (in realtà questo non è totalmente vero dato che solitamente si applica un sottoinsieme di tutti i pattern forniti dal tool), altrimenti alcuni fault coperti da quest'ultimo potrebbero già essere stati marcati detected da altri programmi di test, non garantendo

quindi un incremento sensibile del valore della copertura. Questo è ovviamente un problema, quindi, in linea generale, per i programmi successivi al primo, si richiede un valore raggiunto dalla tecnica ATPG (proprio in fase valutativa) molto più alto rispetto al normale; si possono considerare buoni valori di copertura superiori al 90%.

Applicazione a brinc e circular buffer

Proprio seguendo le linee guida sopra riportate, sono stati realizzati in modo incrementale programmi di test relativi ai moduli brinc e circular buffer.

Per quanto riguarda il modulo brinc, l'esperimento con tecnica ATPG è stato effettuato sull'unica istruzione operante su questo modulo, ossia ***zbrminc*** [17]. Questa istruzione esegue le operazioni già descritte in Sezione 2.3.1. Ad una prima applicazione sono stati raggiunti valori di copertura del 73,16% su pipe0 e del 75,02% su pipe1, semplicemente facendo eseguire la medesima istruzione ai moduli facenti parte delle due differenti pipe, applicando lo stesso valore degli input. Ad un certo punto dello sviluppo dei test per il modulo saturate, come verrà descritto meglio nella Sezione 3.2.2, è stata fatta una scoperta che va ad incidere sui pattern applicati. Questa modifica non va ad influire però sulla tecnica in sé, ma specificamente sui singoli pattern, invertendo il valore logico di alcuni bit di questi. Per questo motivo non viene approfondita in questa sezione la motivazione alla base dell'alterazione compiuta sui pattern in ingresso al modulo. Questa viene rimandata, come già anticipato, alla Sezione 3.2.2.

Con questa modifica apportata si sono riusciti a garantire valori di fault coverage del 91,71% su pipe0 e del 93,16% su pipe1, attraverso la realizzazione di due differenti programmi di test.

Per quanto riguarda il modulo circular buffer, invece, è stata applicata la tecnica ATPG all'istruzione ***zcircinc*** [17]. Questa opera su due differenti registri dove sono codificate tutte le informazioni utili per l'incremento e il decremento all'interno di una struttura a buffer circolare. Si possono quindi individuare le informazioni relative all'indirizzo attualmente puntato, alla lunghezza del buffer e alla dimensione

di ogni singolo blocco; dopodiché, specificando il valore dell'incremento (o del decremento), si ottiene, all'interno del registro destinazione, il contenuto dell'indirizzo aggiornato seguendo i parametri specificati. Grazie all'implementazione di due differenti programmi di test, si sono riusciti a garantire valori di copertura dei fault relativi a questo modulo del 73,26%.

3.2 Il caso saturate

In questa sezione viene descritta l'analisi dello specifico modulo saturate, il quale assume una particolare rilevanza nella sperimentazione di differenti tecniche di testing, sia per quanto riguarda le specifiche difficoltà incontrate, sia per quanto concerne i metodi adottati per affrontarle.

3.2.1 Primo approccio con ATPG

Come nei casi precedentemente esposti, il primo tentativo è stato quello di adottare la tecnica ATPG, la quale si era dimostrata vincente in termini di tempo speso nel metterla in pratica e di copertura ottenuta. Sono stati effettuati diversi tentativi per poter individuare quelle istruzioni che consentissero una maggior copertura dei fault, interessando magari porzioni di hardware maggiori; l'analisi è quindi partita prendendo in considerazione le istruzioni logicamente più complesse. I risultati non sono stati buoni come in precedenza, e questo perché la complessità del modulo in questo caso era maggiore, e la varietà di istruzioni che lo interessavano anche. Per questo motivo, anche per non inficiare troppo il tempo di esecuzione dei singoli programmi, sono stati raccolti con la tecnica ATPG alcuni pattern ritenuti migliori perché in media riuscivano a dare buoni risultati su differenti istruzioni accuratamente selezionate. Nel caso di studio sono stati generati quindi due differenti programmi di test operanti su istruzioni diverse: un set di istruzioni a due operandi in ingresso e un set di istruzioni a singolo operando in ingresso. In definitiva si è deciso di ciclare sulle seguenti istruzioni:

- *SATURATE1.ppc*: 12 coppie di pattern applicate a ***zvpkswshs***, ***zsatsdsw***, ***zsatuduw***, ***zvpkshgwshfrs***, ***zvpkswshfrs***, ***zpkswgswfrs***, ***zpkswgshfrs*** [17];
- *SATURATE2.ppc*: 20 pattern applicati a ***satsbs***, ***satubs***, ***satsbu***, ***satubu***, ***satshs***, ***satuhs***, ***satshu***, ***satsws***, ***satuws***, ***satswu***, ***satuwu*** [17].

L'applicazione di questi programmi di test ha consentito di raggiungere una copertura del 60,87% su pipe0 e del 60,61% su pipe1. Questo non solo grazie alla mera applicazione dei pattern ATPG; infatti, a questi, sono stati aggiunti alcuni particolari pattern "a scacchiera" che hanno consentito il rilevamento di altri fault. Se ne può avere un esempio, anche per capire il perché siano chiamati in questo modo, in Tabella 3.1.

Valore (HEX)	**Valore (BIN)**
0xAAAA	1010 1010 1010 1010
0x5555	0101 0101 0101 0101
0xCCCC	1100 1100 1100 1100
0x3333	0011 0011 0011 0011

Tabella 3.1: Esempi di pattern a scacchiera

3.2.2 I limiti della tecnica ATPG

Dopo il raggiungimento di valori di copertura ragionevoli, i.e. intorno al 60%, l'attesa era quella di veder salire ulteriormente la percentuale con la successiva applicazione del medesimo metodo ad altre istruzioni. Sono state quindi selezionate le istruzioni ***zvpkswshfrs***, ***zvpkswshs***, ***zvpkswuhs***, ***zsatsdsw***, ***zsatsduw***, ***zvsatshuh***, ***zsatswsh***, ***zvslhss***, ***zvslhus*** e altre [17]. Recuperato l'*Instance name* di uno dei due moduli di saturate gemelli presenti all'interno delle due differenti pipe, sono stati impostati all'interno del tool TetraMAX i bit corrispondenti ai codici operativi delle istruzioni selezionate ed altre impostazioni; infine è stata lanciata una *Basic scan* con alto fattore di aggregazione cercando quindi di generare un numero ragionevolmente esiguo di pattern che comunque consentano un buon incremento del

valore di copertura (le informazioni relative ai pattern sono contenute in un file STIL generato da TetraMAX).

Già nella fase di pattern generation vengono riscontrati però comportamenti poco confortanti dal punto di vista del raggiungimento di buone coperture, considerando che il tool non è stato in grado di andare oltre al 70% – 75%. Questo si è tradotto quindi in programmi che marcavano detected pochi fault in più rispetto ai precedenti, con incrementi nell'ordine dell'1% – 3%, corrispondenti a circa 100 – 300 fault complessivi. Il piccolo incremento è da associare ad una sovrapposizione dei fault marcati detected dai precedenti programmi rispetto a quelli generati a questo passo, sovrapposizione molto grande appunto, la quale definisce un grosso insieme di fault easy-to-cover. Inoltre non tutti i pattern generati vengono sfruttati per la scrittura dei programmi di test, selezionando solo quelli che consentono, da soli, all'individuazione di più fault, mantenendo quindi l'occupazione in memoria del programma stesso contenuta. In questo caso sono state fatte differenti prove sfruttando dai 10 ai 30 pattern automaticamente generati.

Anche provando a cambiare la selezione delle istruzioni (e quindi gli opcode) utilizzate per la generazione dei pattern, i risultati sono stati i medesimi. Un'ulteriore prova è stata quella di ridurre il fattore di aggregazione per l'individuazione dei pattern portandolo al minimo configurabile, ma i risultati sono stati di poco superiori a quelli precedenti, i.e. circa un 1% di ulteriore incremento.

A questo punto era necessaria un'analisi più approfondita che è cominciata sfruttando le potenzialità di IMC (Integrated Metrics Center), alla ricerca di porzioni di hardware non testate. L'obiettivo è stato quello di riuscire ad individuarle, associandole alla presenza di basse attività in determinate parti del modulo saturate. Questo comportamento non è però stato verificato dato che la *toggle activity*, i.e. l'attività complessiva per cui si hanno delle transizioni di stato $0 \rightarrow 1$ o $1 \rightarrow 0$ dei segnali presenti, si è rivelata essere molto alta, i.e. 94.88% ottenuta con un rapporto di 2300/2424, con una qualità individuata dal tool pari al 98.71%. Questi valori non consentono di supporre un inutilizzo ma semmai un mascheramento di alcuni fault a valle degli stessi, rendendoli di fatto non osservabili alle uscite.

Un'osservazione a livello gate o RTL sembrava l'unica soluzione per poter individuare una particolare disposizione delle porte o dei coni logici che bloccasse di fatto la propagazione dei guasti. Un'analisi di questo tipo non ha però dato i frutti sperati, ma ha consentito di rilevare un comportamento anomalo tale per cui, in alcuni casi, la fase di sintesi ottimizza il circuito aggiungendo porte logiche addizionali all'input e all'output del modulo[2] per questioni di bilanciamento e rinfrescamento dei segnali. Questo ha comportato una differente valutazione delle tecniche ATPG utilizzate in precedenza dato che il tool TetraMAX opera a livello gate, generando quindi i pattern ad hoc per un'applicazione diretta agli ingressi del modulo; ma vista la presenza di alcune porte NOT agli ingressi, i valori venivano filtrati e modificati. Questa condotta ha del particolare nel fatto che a livello RTL si possa effettivamente notare come il valore del segnale sia effettivamente quello applicato, mentre a livello gate venga introdotta questa discordanza.

Un esempio di questo comportamento e della tecnica utilizzata per il testing è fornito dalla Figura 3.4 in cui, per semplicità, si assume l'impiego del modulo adder. Nel caso in questione, attraverso il programma di test in Figura 3.5, vengono applicati i pattern 0x1234 e 0x5678 ottenuti tramite tecnica ATPG. La presenza però di buffer e inverter altera i valori in ingresso al modulo adder ottenendo, in questo esempio, 0x54A7 e 0x016A. Il risultato atteso di 0x68AC non viene quindi ottenuto, l'output del modulo risulta essere 0x5611 salvo passare attraverso ulteriori buffer e inverter. Il risultato finale risulta quindi essere 0x6E24, ben diverso dal valore 0x68AC atteso.

[2]Questo comportamento è stato riscontrato solamente a livello gate perché l'ottimizzazione subentra solamente a questo livello di dettaglio. E questo è stato il problema principale della scoperta tardiva, dato che la maggior parte delle simulazioni precedenti sono state effettuate a livello RTL, sia data la velocità maggiore nell'avere un risultato sia perché effettivamente l'obiettivo è quello di verificare l'operato logico del modulo in esame.

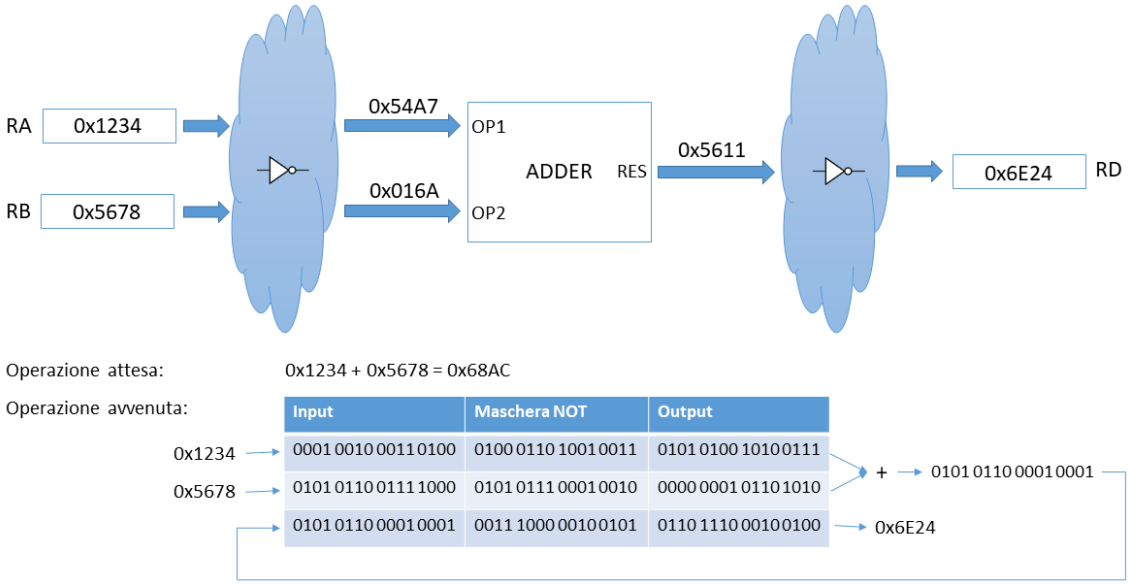

Figura 3.4: Esempio di comportamento del modulo adder con presenza di buffer e inverter agli ingressi e alle uscite

```
MOV     RA, 0x1234
MOV     RB, 0x5678
ADD     RD, RA, RB      ;RD = registro accumulatore per la firma
```

Figura 3.5: Esempio di programma di test per modulo adder

Dato che i pattern sono calcolati senza tenere conto di questa circuiteria, è necessario trovare l'esatta maschera di porte NOT applicata agli ingressi e sfruttarla per creare una funzione inversa che consenta di trovare i valori da mettere nei registri tali per cui, dopo il passaggio dei segnali attraverso le porte NOT, si ottengano, agli operandi in ingresso al modulo interessato, esattamente i valori calcolati con ATPG. Questo rende valido l'utilizzo di tale tecnica di cui in precedenza, quindi, si faceva uso improprio. Revisionando quindi i pattern utilizzati all'interno dei programmi di test, sono stati ricalcolati i valori di copertura che si sono però assestati intorno al 65%, a fronte di molti pattern applicati, circa 40.

L'osservazione di questo comportamento, sebbene abbia consentito, come descritto nella Sezione 3.1, l'innalzamento del livello di copertura del modulo brinc, nel caso del saturate non ha fornito risultati validi.

3.2.3 La tecnica manuale "walking bit"

L'ormai totale e chiara presa di coscienza che fosse necessario un approccio manuale si era palesata. Dopo alcuni tentativi infruttuosi, ha sortito buoni risultati quella che definisco d'ora in poi tecnica "walking bit", la quale trova fondamento nella natura stessa del modello di fault dello stuck-at. La strategia infatti prevede l'utilizzo di pattern in cui tutti i bit sono 0 eccetto uno solo che vale 1; questo bit poi si "muove" (da lì "cammina") andando ad assumere tutte le posizioni possibili del pattern in input. In questo modo, nel caso di uno stuck-at alla posizione dell'input dove è presente il bit a 1, si avrebbe il passaggio da un numero diverso da 0 a un numero pari a 0, con una conseguente alterazione del comportamento. Si è riscontrato che questa tecnica facilita altresì la propagazione dei fault fino alle uscite, anche per fault presenti proprio all'interno del modulo (non solamente agli ingressi quindi) dove viene tendenzialmente alterata la logica e quindi anche le porzioni circuitali interessate. In modo analogo al "walking bit 1" si può applicare anche la strategia duale "walking bit 0" dove ogni bit viene invertito, sarà quindi uno 0 a muoversi e tutti gli altri bit saranno a 1.

Nel caso del modulo saturate si sono presi accorgimenti maggiori data la varietà di operazioni che il modulo consente e la loro complessità logica, dove spesso, ragionando sul codice Verilog, sono svolti molti controlli su singoli bit per scegliere il comportamento da adottare. L'idea generale è stata quella di far commutare il comportamento del modulo sulla base di uno stuck-at agli ingressi, individuando due possibili comportamenti principali delle uscite, ossia saturando il valore dell'ingresso (o del risultato di un'operazione sugli ingressi) al valore massimo/minimo oppure no. Questo comportamento è stato ricreato applicando la tecnica del "walking bit" all'intera word o a porzioni di questa in modo tale che, con uno stuck-at alla posizione giusta, si passasse da una situazione di non saturazione a una di saturazione. E'

riportato di seguito un semplice esempio per la comprensione della strategia adottata e, successivamente, un caso di applicazione reale.

Ipotizzando un generico cast da halfword a byte, si riporta in Tabella 3.2 l'applicazione della tecnica "walking bit 1". Si può notare come vengano creati dei pattern tali per cui, all'interno del byte più significativo, sia presente un solo 1. Questo comporta saturazione dato che il numero in input non risulta rappresentabile su un singolo byte. Nella pratica quindi si sposta il bit partendo dalla posizione 7 fino alla posizione 0, creando di fatto 8 pattern differenti. Nel caso in cui si presenti uno stuck-at 0 corrispondente alla posizione del bit posto a 1 si ricadrebbe nel comportamento complementare, ossia si avrebbe agli input un numero correttamente castabile a byte. In questo caso in uscita si otterrebbe quindi il contenuto del byte meno significativo del numero in input.

Valore in ingresso	Output (gold)	Fault	Output (fault)
0000 0001 0000 0000	0000 0000 0111 1111	S@0 bit 7	0000 0000 0000 0001
0000 0010 0000 0000	0000 0000 0111 1111	S@0 bit 6	0000 0000 0000 0001
...	...	...	...
0100 0000 0000 0000	0000 0000 0111 1111	S@0 bit 1	0000 0000 0000 0001
1000 0000 0000 0000	0000 0000 1000 0000	S@0 bit 0	0000 0000 0000 0001

Tabella 3.2: Pattern applicati con tecnica "walking bit 1" nel caso banale di cast da halfword a byte

Analogamente, come riportato in Tabella 3.3, si riporta l'applicazione duale, ossia quella del "walking bit 0". Si noti come il risultato "gold" preveda, come in precedenza, la saturazione del valore agli ingressi, mentre nel caso della presenza dello stuck-at il comportamento sia l'inverso.

Valore in ingresso	Output (gold)	Fault	Output (fault)
1111 1110 1111 1111	0000 0000 1000 0000	S@1 bit 7	0000 0000 0000 0001
1111 1101 1111 1111	0000 0000 1000 0000	S@1 bit 6	0000 0000 0000 0001
...	...	...	...
1011 1111 1111 1111	0000 0000 1000 0000	S@1 bit 1	0000 0000 0000 0001
0111 1111 1111 1111	0000 0000 0111 1111	S@1 bit 0	0000 0000 0000 0001

Tabella 3.3: Pattern applicati con tecnica "walking bit 0" nel caso banale di cast da halfword a byte

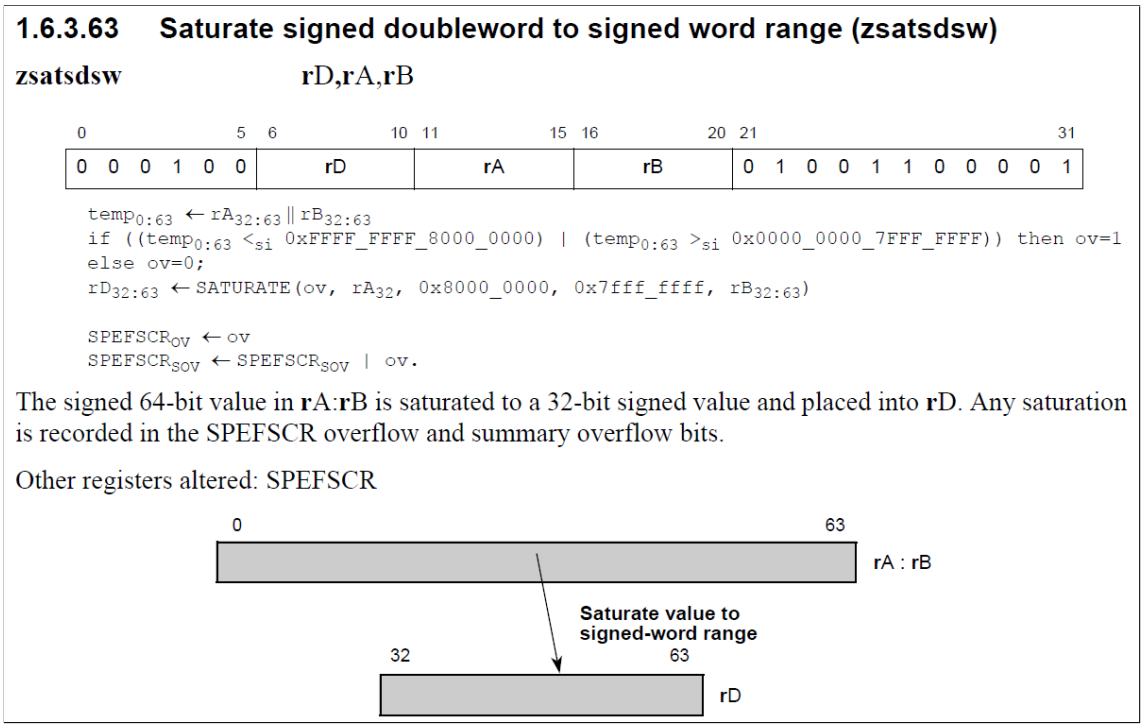

Figura 3.6: Istruzione ***zsatsdsw***: estratto del manuale LSP [17]

Una volta compreso su un esempio banale il meccanismo di base, si procede con l'analisi di un'istruzione su cui è stata effettivamente applicata la tecnica "walking bit". Come si può vedere in Figura 3.6, nel caso dell'istruzione ***zsatsdsw*** [17], il valore su doubleword con segno viene saturato ad un valore su word con segno. Si può notare come, se il valore dell'input è inferiore a 0xFFFF_FFFF_8000_0000 o

superiore a 0x0000_0000_7FFF_FFFF, non può essere rappresentato su word a 32 bit. L'approccio seguito è stato quello dell'applicazione della tecnica "walking bit 1" alla word alta dell'input, come si può notare in Tabella 3.4. In un processore esente da fault il comportamento è quello di saturare al valore massimo con segno rappresentabile su 32 bit, ossia 0x7FFF_FFFF. Nel caso invece che sia presente un fault a quel determinato bit dell'ingresso viene riportata in uscita la word bassa, fissata in questo caso a 0x0000_0001. Si noti come invece l'ultimo pattern dia come risultato gold[3] 0x8000_0000 dato che si tratta di un valore in ingresso negativo a differenza degli altri casi.

Valore in ingresso	Output (gold)	Fault	Output (fault)
$_0$0000 ... 0001_{31} $_{32}$0000 ... 0001_{63}	7FFF FFFF $_{HEX}$	S@0 bit 15	0000 0001 $_{HEX}$
$_0$0000 ... 0010_{31} $_{32}$0000 ... 0001_{63}	7FFF FFFF $_{HEX}$	S@0 bit 14	0000 0001 $_{HEX}$
$_0$0000 ... 0100_{31} $_{32}$0000 ... 0001_{63}	7FFF FFFF $_{HEX}$	S@0 bit 13	0000 0001 $_{HEX}$
...	...	...	...
$_0$0010 ... 0000_{31} $_{32}$0000 ... 0001_{63}	7FFF FFFF $_{HEX}$	S@0 bit 2	0000 0001 $_{HEX}$
$_0$0100 ... 0000_{31} $_{32}$0000 ... 0001_{63}	7FFF FFFF $_{HEX}$	S@0 bit 1	0000 0001 $_{HEX}$
$_0$1000 ... 0000_{31} $_{32}$0000 ... 0001_{63}	8000 0000 $_{HEX}$	S@0 bit 0	0000 0001 $_{HEX}$

Tabella 3.4: Pattern applicati con tecnica "walking bit 1" a istruzione ***zsatsdsw***

E' stato seguito anche l'approccio duale con la tecnica "walking bit 0" dove, in un processore esente da fault, il comportamento è quello di andare in saturazione saturando al valore minimo rappresentabile su 32 bit: 0x8000_0000. In un processore con fault presenti invece viene riportata la word bassa, fissata a 0xFFFF_FFFF. Si riporta lo schema dei pattern applicati in Tabella 3.5. Si noti, come nel caso precedente, il comportamento differente dell'ultimo pattern applicato.

[3]Si intende il risultato atteso nel caso in cui il processore sia esente da fault, in buona sostanza il risultato valido.

Valore in ingresso	Output (gold)	Fault	Output (fault)
$_{0}1111 \ldots 1111_{31}\ _{32}1111 \ldots 1111_{63}$	8000 0000 $_{HEX}$	S@1 bit 15	FFFF FFFF $_{HEX}$
$_{0}1111 \ldots 1101_{31}\ _{32}1111 \ldots 1111_{63}$	8000 0000 $_{HEX}$	S@1 bit 14	FFFF FFFF $_{HEX}$
$_{0}1111 \ldots 1011_{31}\ _{32}1111 \ldots 1111_{63}$	8000 0000 $_{HEX}$	S@1 bit 13	FFFF FFFF $_{HEX}$
...	...	...	...
$_{0}1101 \ldots 1111_{31}\ _{32}1111 \ldots 1111_{63}$	8000 0000 $_{HEX}$	S@1 bit 2	FFFF FFFF $_{HEX}$
$_{0}1011 \ldots 1111_{31}\ _{32}1111 \ldots 1111_{63}$	8000 0000 $_{HEX}$	S@1 bit 1	FFFF FFFF $_{HEX}$
$_{0}0111 \ldots 1111_{31}\ _{32}1111 \ldots 1111_{63}$	7FFF FFFF $_{HEX}$	S@1 bit 0	FFFF FFFF $_{HEX}$

Tabella 3.5: Pattern applicati con tecnica “walking bit 0” a istruzione ***zsatsdsw***

Considerazioni molto simili sono state fatte nel caso di altre istruzioni quali ***zsatsduw***, ***zvpkswshfrs***, ***satsbs***, ***satshs***, ***satsws***, ***satsbu***, ***satshu*** e ***satswu*** [17]. Le uniche differenze sono state legate alla specifica funzione dell’istruzione, necessitando di uno studio ad hoc visto il diverso impatto sull’operato dell’unità.

Dopo l’utilizzo di questa tecnica, l’incremento di copertura è stato significativo ed ha portato ad ipotizzare la duplicazione di hardware all’interno del modulo, anche per lo svolgimento di operazioni affini. Questo si è potuto evincere dal fatto che l’incremento di copertura apportato dall’impiego successivo della ***zsatsdsw*** e poi della ***zsatsduw*** è stato assai simile. Stupisce infatti come, nel secondo caso, si possa paragonare l’aumento al primo, nonostante le scelte fatte rispetto ai vari bit siano praticamente uguali. Questa considerazione è risultata vera anche per successive applicazioni ad altre istruzioni.

La copertura raggiunta grazie all’impiego della tecnica manuale “walking bit” è stato pari al 75,70% su pipe0 e al 75,50% su pipe1, attraverso due differenti programmi di test. Non sono stati qui riportati gli specifici casi di applicazione per evitare la ridondanza, l’approccio rimane comunque il medesimo sopra descritto. La tecnica rimane comunque valida anche in generale per molte altre istruzioni svolte dal modulo saturate ed è possibile estenderne l’utilizzo a tutti i moduli aritmetici del processore, sia quelli che hanno un comportamento abbastanza semplice e lineare, ma soprattutto a quelli che svolgono operazioni complesse, con comportamenti

differenti a seconda di specifici segnali in ingresso, magari interessati da molte istruzioni diverse. A questo punto sono state raggiunte sì coperture già molto buone, ma soprattutto si è riuscito a capire l'approccio base per il testing del modulo che verrà sfruttato successivamente per spingersi ancora oltre.

3.2.4 Approccio con μGP

Dopo aver trovato una buona tecnica di testing si necessitava di estendere ad altre istruzioni questo approccio. Si è però individuata una difficoltà intrinseca nel problema, ossia che l'approccio seguito utilizzando la tecnica "walking bit" fosse completamente manuale e quindi, per la creazione di un singolo programma di test operante su un'istruzione, si sarebbe dovuto spendere molto tempo, a livello di istruzione, per comprenderne a fondo l'impiego e per trovare il modo di effettuare gli shift dei bit delle word. Nel complesso rimanendo comunque un approccio prono ad errori, essendo che tutto questo comportamento deve essere revisionato in simulazione, sia nel caso gold che in quello con guasti.

L'idea a questo punto è stata quella di affidarsi ad un algoritmo evolutivo che non provasse effettivamente tutte le maschere che differiscono per un solo bit, ma che cercasse automaticamente la strada migliore, magari con qualche pattern casuale e, in qualche modo, riuscisse ad alzare la copertura del modulo. A questo proposito si è fatto uso di μGP impostando i valori per il procedere dell'algoritmo, e poi, mediante tuning, si è cercato di affinarli per raggiungere valori di fault coverage sempre maggiori. Sono stati quindi preparati alcuni algoritmi evolutivi che, sempre meglio, sono riusciti ad avvicinarsi ad alti valori di copertura. Ogni algoritmo evolutivo eseguito da μGP ha comportato la creazione di molti individui figli, che, mano a mano, andavano a costituire la popolazione di partenza per una nuova generazione. Il risultato è stato quello, per ogni esecuzione, di avere un set di individui con la loro rispettiva valutazione in termini di fault coverage, estrapolando da questi il migliore, ossia quello associato ad una percentuale maggiore. Alla fine, per avere il giusto compromesso tra dimensione dei programmi di test, tempo di esecuzione degli stessi e valore di copertura raggiunto, si sono mantenuti due degli otto approcci evolutivi compiuti (e quindi due degli otto individui migliori). Si è riusciti quindi a

raggiungere una copertura dell'85,19% su pipe0 e dell'85,66% su pipe1 per mezzo di due programmi di test.

I primi algoritmi sono stati impostati in modo tale da lasciar il più libero possibile μGP, quindi consentendogli di scegliere le istruzioni e i valori dei pattern da una serie fornita, rispettando i vincoli imposti dalle singole istruzioni circa il numero di operandi e i valori che questi potessero assumere. L'implementazione prevedeva la ripetizione di un certo numero di blocchi di codice con la stessa struttura e, in ognuno di questi, veniva copiato un valore in dei registri che avrebbero costituito le sorgenti per l'operazione che sarebbe stata scelta euristicamente. I risultati non sono però stati buoni data la troppa libertà, sia sulla scelta delle istruzioni che sugli operandi. Questi ultimi, nelle prime esecuzioni, sono stati completamente liberi di assumere tutti i valori possibili su 32 bit, quindi tra 0 (0x0000_0000) e 4294967295 (0xFFFF_FFFF).

Si è andati quindi nella direzioni di svolgere più esecuzioni agendo, per ognuna di queste, su istruzioni differenti di volta in volta. Questo modus operandi avrebbe consentito di lasciar concentrare μGP sulla scelta degli operandi e meno sui fattori combinati. Inoltre, per quanto riguarda i possibili pattern in ingresso, ne è stata ridotta la scelta a valori binari più oculati, sempre nel pieno stile della tecnica "walking bit" adottata in precedenza. Quello che ne è risultato, dopo diverse prove, sono stati due programmi di test aventi entrambi la stessa struttura, di cui si riporta in Figura 3.7 la macro definita all'interno del file che rappresenta la *constraints library*.

```
<macro id="istruz_macro">
<expression>
        e_lis           r29, (<param ref="op1" />)@h
        e_or2i          r29, (<param ref="op1" />)@l
        e_lis           r28, (<param ref="op2" />)@h
        e_or2i          r28, (<param ref="op2" />)@l
        e_lis           r27, (<param ref="op3" />)@h
        e_or2i          r27, (<param ref="op3" />)@l
        e_lis           r26, (<param ref="op4" />)@h
        e_or2i          r26, (<param ref="op4" />)@l

        <param ref="gen_op_1" /> r31, r29, r28
        <param ref="gen_op_2" /> r30, r27, r26
        adde            r4, r4, r31      <!-- r4 = accumulatore per la firma -->
        adde            r4, r4, r30

</expression>
<parameters>
        <item xsi:type="definedType" ref="istr3ops" name="gen_op_1" />
        <item xsi:type="definedType" ref="istr3ops" name="gen_op_2" />
        <item xsi:type="definedType" ref="integerNumber" name="op1" />
        <item xsi:type="definedType" ref="integerNumber" name="op2" />
        <item xsi:type="definedType" ref="integerNumber" name="op3" />
        <item xsi:type="definedType" ref="integerNumber" name="op4" />
</parameters>
</macro>
```

Figura 3.7: Macro definita per generazione evolutiva di programmi di test con μGP

Come invece si può osservare in Figura 3.8, la definizione dei tipi utilizzati all'interno della macro differisce solamente per le istruzioni utilizzate mentre la scelta degli operandi rimane pressoché identica.

Da notare, sempre in Figura 3.8, l'utilizzo di pattern intelligenti, come già precedentemente fatto nel corso dell'approccio manuale. I risultati, dopo queste esecuzioni, sono stati molto buoni permettendo il raggiungimento di valori di fault coverage pari all'85,19% su pipe0 e dell'85,66% su pipe1. L'obiettivo di estendere l'approccio manuale a istruzioni differenti rispetto a quelle analizzate manualmente ha dato i suoi frutti, consentendo di non analizzare mano a mano le istruzioni definendo pattern ad hoc. Sicuramente più preciso e puntuale l'approccio umano, ma anche meno rapido; in questo modo invece μGP si prende in carico la scelta dei pattern corretti per determinate istruzioni cercando di trovare quelli che funzionano meglio, anche se non necessariamente tutti.

```
<item xsi:type="constant"
name="istr3ops">
        <value>zvpkswshs </value>
        <value>zvpkswuhs </value>
        <value>zvpkuwuhs </value>
        <value>zvpkswshs </value>
</item>
<item name="integerNumber"
type="constant">
        <value>0x00000105</value>
        <value>0x00000205</value>
        <value>0x00000405</value>
        <value>0x00000805</value>
          ...     ....      ...
        <value>0x10000005</value>
        <value>0x20000005</value>
        <value>0x40000005</value>
        <value>0x80000005</value>

        <value>0xFFFFFEF5</value>
        <value>0xFFFFFDF5</value>
        <value>0xFFFFFBF5</value>
        <value>0xFFFFF7F5</value>
          ...     ....      ...
        <value>0xEFFFFFF5</value>
        <value>0xDFFFFFF5</value>
        <value>0xBFFFFFF5</value>
        <value>0x7FFFFFF5</value>
</item>
```

```
<item xsi:type="constant"
name="istr3ops">
        <value>zvpkshgwshfrs </value>
        <value>zpkswgswfrs   </value>
</item>
<item name="integerNumber"
type="constant">
        <value>0x00000105</value>
        <value>0x00000205</value>
        <value>0x00000405</value>
        <value>0x00000805</value>
          ...     ....      ...
        <value>0x10000005</value>
        <value>0x20000005</value>
        <value>0x40000005</value>
        <value>0x80000005</value>

        <value>0xFFFFFEF5</value>
        <value>0xFFFFFDF5</value>
        <value>0xFFFFFBF5</value>
        <value>0xFFFFF7F5</value>
          ...     ....      ...
        <value>0xEFFFFFF5</value>
        <value>0xDFFFFFF5</value>
        <value>0xBFFFFFF5</value>
        <value>0x7FFFFFF5</value>
</item>
```

Figura 3.8: Definizione dei tipi per generazione evolutiva di programmi di test con μGP (per due programmi differenti)

Per quanto riguarda l'evoluzione in sé, con l'iterazione dei tentativi sono stati variati i parametri relativi alla dimensione della popolazione. Inizialmente era stata impostata una popolazione iniziale (nu) di 40 individui col mantenimento, ad ogni passo, di una popolazione (mu) di 30 individui, agendo poi su questi per le successive generazioni. Si è passato poi, nella versione definitiva, a considerare come popolazione iniziale 10 individui (nu) e poi mantenendo 20 individui (mu) ad ogni passo dell'iterazione. Questo ha dato risultati migliori cercando sin da subito di trovare una buona strada, riducendo di fatto la generazione iniziale. Questa scelta è stata inoltre dovuta ai tempi di classificazione di ogni singolo individuo, infatti i processi di simulazione e fault-simulazione erano abbastanza esosi in termini di tempo, richiedendo diversi minuti per essere eseguiti; e questo era necessario per ogni singolo individuo per poter fornire il valore di fitness, i.e. bontà, in questo caso la fault coverage.

Per quanto riguarda invece gli altri parametri, affinati tutti attraverso tuning, sono stati utilizzati i seguenti valori:

- *inertia*: 0.9 (valore di default);
- *lambda*: 6, essendosi concentrati su tipi di operatori genetici utili allo scopo, e quindi in numero ridotto rispetto a tutti quelli previsti da μGP;
- *sigma*: 0.9, spingendo quindi per un forte cambiamento forzando gli operatori di mutazione ad agire in maniera abbastanza forte;
- parametri di selezione degli individui genitori: *tournamentWithFitnessHole* con: $tau = 2$ – $tauMin = 1{,}5$ – $tauMax = 4$ – $fitnessHole = 0.1$, cercando di fatto di sfruttare solamente i genitori migliori per le successive generazioni.

3.3 Accorgimenti per sincronizzazione pipe

Come già detto, essendo in presenza di un processore dual issue, sono presenti due pipe per consentire una parallelizzazione del carico di lavoro, ma non c'è una duplicazione totale dei vari moduli. Per quando concerne il testing sono state riscontrate difficoltà nel riuscire a far salire la copertura in modo parallelo su entrambe le pipe a causa di particolari comportamenti. Le soluzioni adottate sono dipese dal caso specifico e l'avvicendarsi dei segnali è sempre stato verificato mediante simulazione. A tal proposito si fa notare come il comportamento sia stato verificato guardando effettivamente i segnali di controllo rappresentanti l'opcode in ingresso ai due moduli gemelli sotto esame, questo ogni volta in relazione all'istruzione applicata.

Come si può vedere in Figura 3.9, il meccanismo principale per sopperire a questa problematica è quello di replicare, una di seguito all'altra, la medesima istruzione in modo tale che, essendo la prima eseguita su pipe0, il processore esegua su pipe1 la seconda considerando il modulo precedente occupato. Il risultato in alcuni casi è stato quello di veder sì eseguita la prima istruzione su pipe0, ma la seconda, pur venendo messi agli ingressi i valori corretti degli input, veniva stallata, non eseguendo di fatto alcuna operazione; l'istruzione in questione veniva quindi dirottata nuovamente su pipe0 dove effettivamente veniva svolta. Questa predilizione particolare

per la prima pipe, i.e. pipe0, rispetto all'altra ha portato a valori di copertura dell'UUT su pipe1 molto più bassi rispetto a quelli su pipe0. Sono stati quindi adottati accorgimenti per far sì che l'istruzione desiderata venisse effettivamente eseguita da entrambe le unità. Come si può notare in Figura 3.10, è stata inserita un'istruzione di divisione, quale è per esempio ***divw*** [17], che si è verificato venir sempre eseguita su pipe0. In questo modo, data anche la durata di tale istruzione, quella immediatamente successiva viene forzata su pipe1 sbloccando di fatto l'asincronismo che si era notato in precedenza; da questo punto in poi, infatti, tutte le istruzioni del blocco saranno correttamente eseguite. In figura è anche inserita un'istruzione **NOP** per mostrare come sia rieseguito il riallineamento delle pipe in modo tale che l'istruzione successiva vada, come di norma, su pipe0.

```
ISTRUZIONE1     R1, R2, R3      ;pipe0
ISTRUZIONE1     R4, R5, R6      ;pipe1
ISTRUZIONE2     R7, R8, R9      ;pipe0
ISTRUZIONE2     R10, R11, R12   ;pipe1
ISTRUZIONE3     R13, R14, R15   ;pipe0
ISTRUZIONE3     R16, R17, R18   ;pipe1

;R31 = registro dove viene mantenuta e
;progressivamente aggiornata la firma
ADDE    R31, R1, R4             ;pipe0
ADDE    R31, R7, R10            ;pipe1
ADDE    R31, R13, R16           ;pipe0
```

Figura 3.9: Approccio base per sincronizzazione pipe

```
;divisione per eseguire forzatamente
;la prossima istruzione su pipe1
DIVW    R28, R29, R30           ;pipe0

ISTRUZIONE1     R1, R2, R3      ;pipe1
ISTRUZIONE1     R4, R5, R6      ;pipe0
ISTRUZIONE2     R7, R8, R9      ;pipe1
ISTRUZIONE2     R10, R11, R12   ;pipe0
ISTRUZIONE3     R13, R14, R15   ;pipe1
ISTRUZIONE3     R16, R17, R18   ;pipe0

;NOP per riallineare le pipe
;come nel caso precedente
NOP                             ;pipe1

;R31 = registro dove viene mantenuta e
;progressivamente aggiornata la firma
ADDE    R31, R1, R4             ;pipe0
ADDE    R31, R7, R10            ;pipe1
ADDE    R31, R13, R16           ;pipe0
```

Figura 3.10: Accorgimenti particolari per sincronizzazione pipe

Capitolo 4

Tool di analisi dei risultati

In questa sezione verrà descritto il tool di analisi di fault list creato appositamente per esaminare puntualmente i guasti, non avvalendosi più solamente di un valore di copertura percentuale, ma andando a vedere proprio il singolo fault come viene coperto da differenti programmi di test. Verranno descritte le motivazioni che ne hanno portato alla creazione, le estensioni all'utilizzo che sono state introdotte, la struttura del tool in sé e infine una vera e propria applicazione sul caso di studio.

4.1 Motivazioni

Le motivazioni di spinta sono state legate alla necessità di avere un programma che desse informazioni diverse dalla mera conoscenza di un valore di copertura (che tendenzialmente si costruisce in maniera incrementale). Questo è dovuto ai diversi programmi di test che sono stati necessari per far salire la copertura del modulo saturate. A questo punto diventava importante sapere se nuovi programmi di test andassero a coprire nuovi fault o grandi insiemi di fault già coperti, ma in maniera migliore. Questo comportamento può essere altresì verificato mandando in esecuzione i nuovi programmi da soli verificandone la copertura, ma, un po' per le diverse versioni di programmi generati da μGP, un po' per avere un tool che riuscisse a fare tutto in automatico (anche in altre situazioni), si è presa in considerazione l'idea di creare un analizzatore di fault list. Questo, a partire da fault list separate, insistenti sugli stessi fault, avrebbe dovuto capire in che relazione fossero i diversi programmi di test; se per esempio ci fossero sovrapposizioni e quanto marcate fossero, se si fosse nella situazione in cui qualche programma passato potesse non essere considerato perché portatore di un contributo strettamente proprio esiguo. In ottica generale però è utile, nella scrittura di programmi di test manuali, avere informazioni circa

i fault che si possono considerare easy-to-cover, ossia quelli che sono facilmente coperti da ogni programma di test e che quindi, ragionevolmente, si possono evitare di prendere in considerazione per successivi programmi. Per quanto riguarda altri guasti, che invece sono marcati detected solamente da un programma di test, potrebbe essere interessante estrapolarli selettivamente per farne un'analisi più puntuale.

4.2 Funzionalità

Il tool realizzato è utilizzabile sia da riga di comando, in modo tale che il suo operato possa venire sfruttato all'interno di qualche script, sia tramite interfaccia grafica, chiaramente più intuitiva. Si andrà quindi ad analizzare le funzionalità offerte dal tool "Polito Fault Lists Analyzer" partendo da una descrizione della GUI (Graphical User Interface) realizzata, della quale si può avere un'anteprima in Figura 4.1.

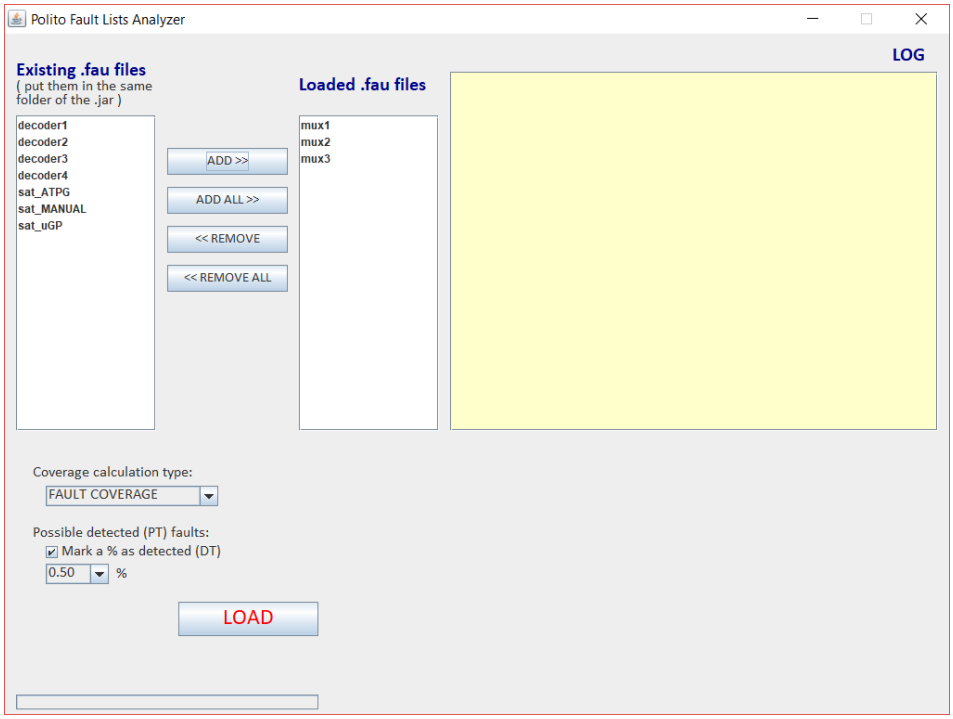

Figura 4.1: Schermata di avvio del tool "Polito Fault Lists Analyzer"

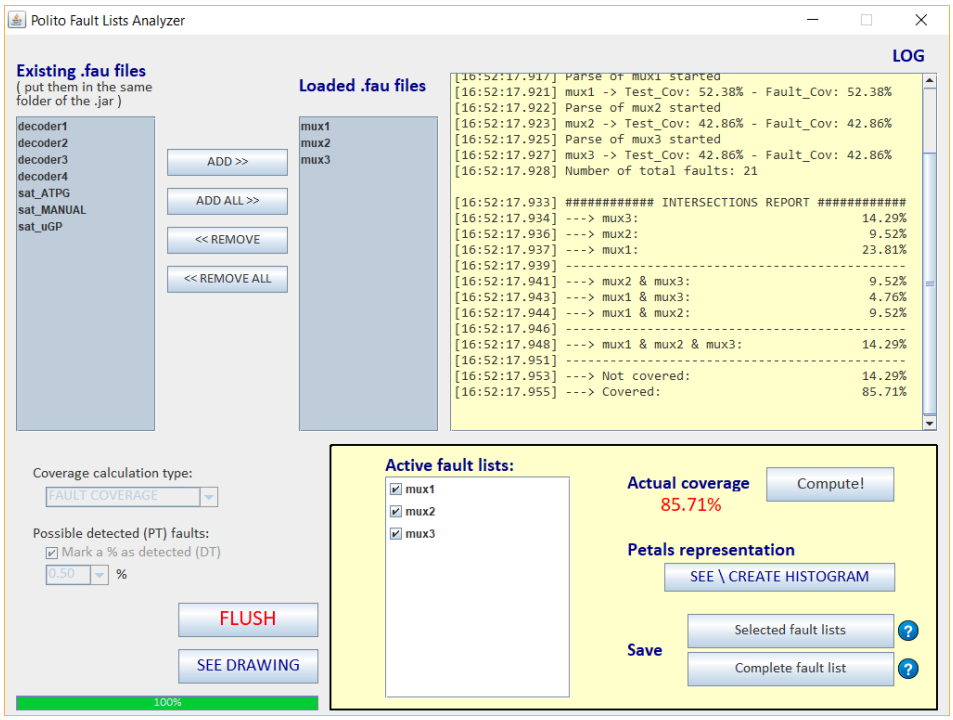

Figura 4.2: "Polito Fault Lists Analyzer": schermata a caricamento avvenuto

Dopo aver selezionato e spostato nell'apposita colonna le fault list di interesse, se ne può procedere al caricamento attraverso il bottone "LOAD". Prima di procedere con questa operazione, si possono impostare i due settaggi seguenti:

- tipo di calcolo della copertura, a scelta tra "Fault coverage" e "Test coverage";
- percentuale di fault PT (*Possibly Detected*) da marcare DT (*Detected*), con valore di default pari al 50%.

Dopodiché il tool elabora le fault list in ingresso creandosi tutte le strutture dati utili, mostrando inoltre nel "LOG" (il cui contenuto è replicato nel file di output *results.log*, comodo per essere letto da script) i valori di copertura totali delle singole fault list e fornendo poi un report con tutti i valori di copertura divisi per ogni possibile sottoinsieme. Partendo dalla rappresentazione dei singoli petali[1] si passa alle

[1]Nome assegnato ai sottoinsiemi propri data la loro caratteristica di rappresentare le parti più esterne degli insiemi in una rappresentazione grafica.

intersezioni di soli due insiemi, e così via, fornendo infine la copertura totale data dalla sovrapposizione delle varie fault list. Un esempio di come queste informazioni vengono mostrate si può trovare alla pagina precedente in Figura 4.2.

Per comprendere meglio cosa rappresentano le varie percentuali riportate nella schermata di log, si può vederne una rappresentazione grafica in Figura 4.3. Questa schermata è accessibile cliccando sul bottone "SEE DRAWING" e i valori riportati all'interno della rappresentazione insiemistica sono esattamente gli stessi che sono riportati in forma testuale nella schermata precedente.

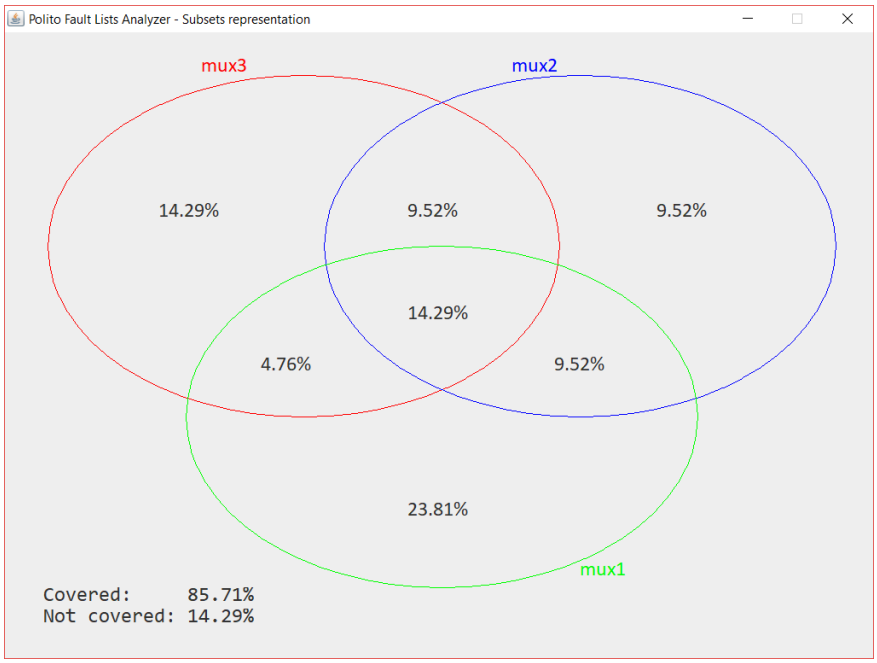

Figura 4.3: "Polito Fault Lists Analyzer": rappresentazione insiemistica delle percentuali di copertura di differenti fault list

Importante sottolineare il fatto che, per esempio, la percentuale 4.76% che si può notare in Figura 4.3 tra l'insieme "mux1" e "mux3", identifichi esclusivamente quella regione per cui i fault sono coperti da entrambe le fault list, ma non dalla fault list "mux2". Questo ragionamento si estende ad ogni singolo sottoinsieme, in modo tale da avere indicazioni il più puntuali possibili.

Come si può osservare in Figura 4.2, dopo il caricamento delle fault list desiderate, vengono disabilitati tutti i comandi di pre-loading[2] e compare una sezione dove si possono compiere operazioni addizionali post-caricamento basate tutte sulle fault list "attive" selezionabili per mezzo di checkbox. Le azioni percorribili constano di:

- possibilità di calcolare la copertura totale garantita da tutte le fault list "attive";
- ottenere una rappresentazione a istogramma dei soli petali della raffigurazione insiemistica, in modo tale da avere una chiara idea di quali programmi di test potrebbero essere rimossi non andando a inficiare troppo sulla copertura, garantendo quindi un miglioramento in termini di tempi di esecuzione e occupazione in memoria. Se ne può vedere un esempio in Figura 4.4. Inoltre l'immagine, oltre ad essere visualizzata, è anche esportata in formato PNG nella sottodirectory "/generated";
- esportare nell'usuale formato FAU le singole fault list definite da un singolo sottoinsieme[3], marcando quindi detected i fault coperti da quel sottoinsieme e non-detected gli altri. Oppure si può esportare la fault list complessiva dove sono marcati detected i fault che sono detected da almeno una fault list e non-detected tutti gli altri; questo caso è l'unico dove si tiene conto dell'unione delle fault list.

[2]Per ritornare alla configurazione iniziale è sufficiente la pressione del bottone "FLUSH".

[3]Come riportato nell'help associato a tale funzionalità, si fa riferimento, come già detto, solamente a quei fault strettamente contenuti da tutte le fault list attive, ma non coperti dalle fault list non attive.

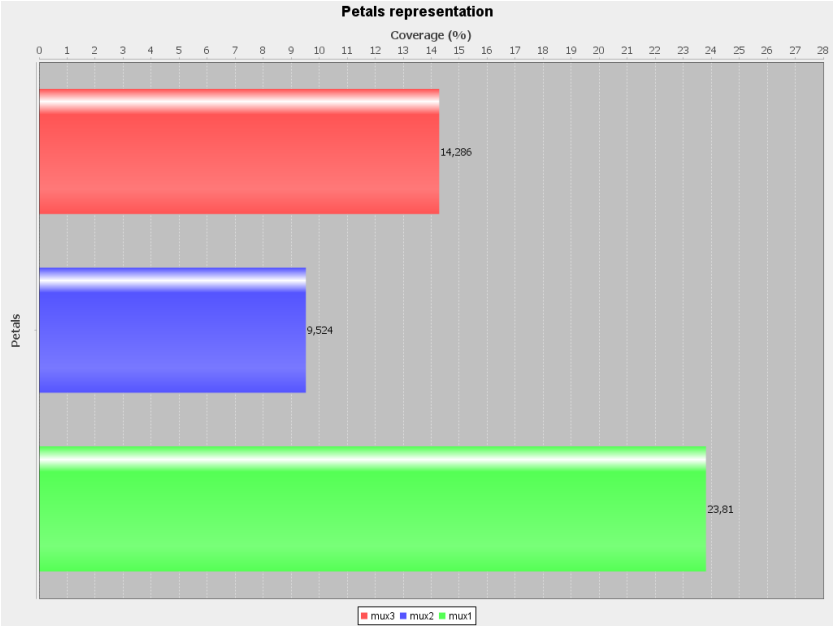

Figura 4.4: "Polito Fault Lists Analyzer": rappresentazione a istogramma della copertura ottenuta dai singoli petali

Si riporta infine, per completezza, il metodo di utilizzo del programma da riga di comando, in Figura 4.5.

```
Usage:  -gui              launch the program with a graphical interface
                          (other parameters will be ignored)
        -files            1:n name of input .fau files (include file extension)
        [-calctype]       [TEST_COV|FAULT_COV] - default: TEST_COV
        [-ptcredit]       [0:1] - default: 0.5
```

Figura 4.5: "Polito Fault Lists Analyzer": metodo di utilizzo da riga di comando

4.3 Modalità realizzative

Il tool è stato realizzato basandosi sul linguaggio Java nella sua declinazione Java 8, in modo da consentirne l'esecuzione su calcolatori con differenti sistemi operativi installati. A parte queste peculiarità offerte dalla Java Virtual Machine (JVM), è stato adottato anche per la necessità di creare delle interfacce grafiche, effettuare disegni "a mano" e avere un supporto nella creazione di grafici da parte di librerie esterne pre-esistenti.

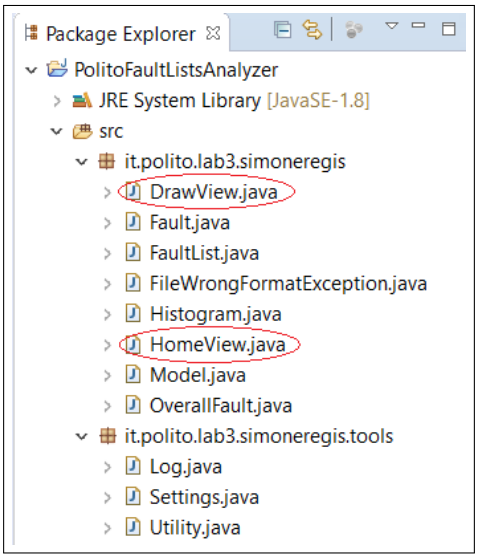

Figura 4.6: "Polito Fault Lists Analyzer": elenco delle classi

Come si può immediatamente osservare in Figura 4.6, sono state realizzate due differenti classi per l'interfaccia grafica, ognuna associata ad una determinata schermata della GUI (schermata Home e schermata di rappresentazione insiemistica); entrambe create con l'ausilio di Windows Builder. E' presente poi la classe Model che è quella adibita ad ogni effettivo compito richiesto dall'utente (per mezzo della GUI o da riga di comando), di cui si approfondirà in seguito. Importanti sono le classi FaultList, Fault e OverallFault di cui gli oggetti istanza delle stesse rappresentano rispettivamente una singola fault list in ingresso, un singolo fault all'interno

di una fault list e un singolo fault comune a differenti fault list. Quest'ultimo contiene anche variabili in grado di tenere traccia di come questo era stato marcato dai differenti file in ingresso.

Interessante è approfondire il comportamento del tool al caricamento delle differenti fault list (per altro comportamento di default per il lancio da riga di comando). Dopo tutti i controlli necessari, tra cui la presenza dei file in input e della cartella "/generated", vengono lette, riga per riga, tutte le fault list costruendo gli oggetti FaultList. Dopodiché devono essere messe insieme le informazioni relative allo stesso fault presente in oggetti diversi, e questo viene fatto con un'operazione con complessità $\Theta(nmk)$ con:

- n: numero di fault all'interno di ogni fault list
- m: numero di fault list
- k: costante

Questo perché l'algoritmo utilizzato prevede una lettura completa della prima fault list con complessità $\Theta(n)$, poi, per ogni altra fault list, una scansione lineare con ricerca in una struttura dati organizzata con una tabella di hash, con complessità quindi $\Theta((m-1)nk)$ (con k pari al tempo di accesso costante alla struttura di hash). Ne risulta quindi una complessità pari a:

$$\Theta(n + (m-1)nk) = \Theta(n((m-1)k+1)) = \Theta(n(mk-k+1)) = \Theta(nmk).$$

Era stato considerato in precedenza anche un altro metodo basato su un preordinamento delle fault list, il quale presentava però complessità maggiore. Il costo di ordinamento di ogni fault list, con i migliori algoritmi di ordinamento, è $\Theta(n \cdot log(n))$. A questo bisogna aggiungere poi la scansione lineare di ogni fault list ottenendo quindi:

$$\Theta(mn \cdot log(n) + mn) = \Theta(mn(1 + log(n))) = \Theta(mn \cdot log(n)).$$

Dopo la creazione di tutti gli oggetti di tipo OverallFault vengono create tutte le combinazioni di sottoinsiemi individuabili da maschere binarie. Ogni fault viene

quindi collocato nel giusto sottoinsieme e viene elaborato un report prima testuale e poi grafico.

4.4 Applicazione a modulo saturate

Date le forti motivazioni legate al caso del saturate che hanno portato alla realizzazione del tool, ne è conseguita l'applicazione alle tre differenti tipologie di programmi realizzate in precedenza: quelli realizzati con tecnica ATPG, quelli con approccio manuale e quelli autogenerati da μGP sulla base di linee guida impostate. Sono stati quindi eseguiti separatamente i programmi di test relativi a questi diversi approcci e caricati poi all'interno di "Polito Fault Lists Analyzer". I risultati ottenuti sono facilmente osservabili in Figura 4.7, dove ritroviamo la copertura totale raggiunta e altre informazioni a contorno, e in Figura 4.8 con la rappresentazione dei singoli petali.

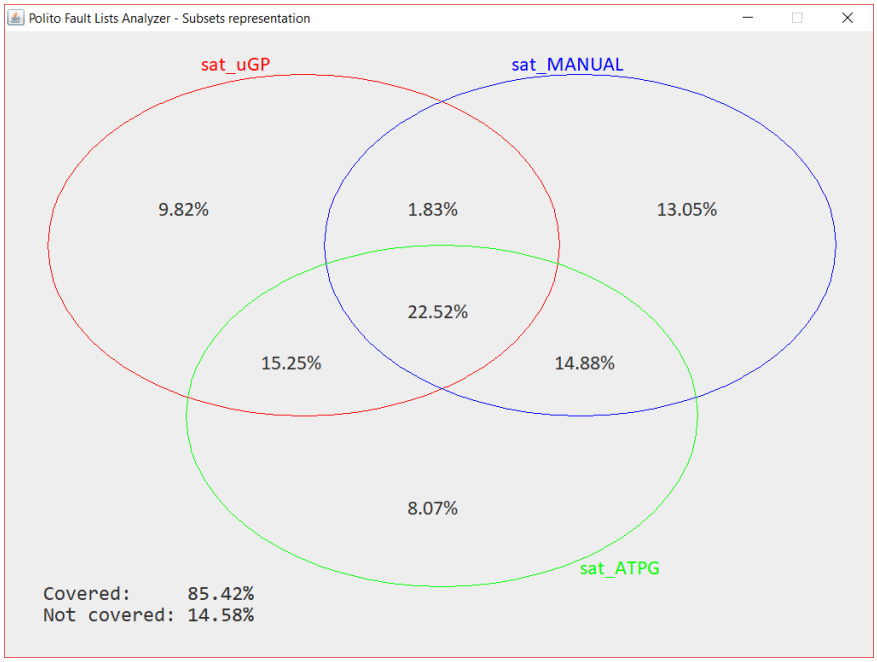

Figura 4.7: "Polito Fault Lists Analyzer": risultati insiemistici per applicazione a modulo saturate

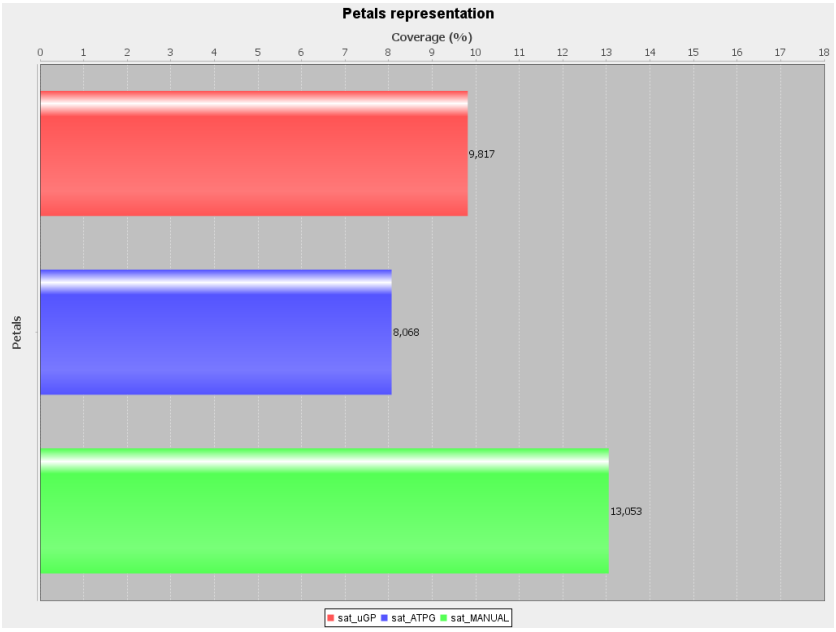

Figura 4.8: "Polito Fault Lists Analyzer": rappresentazione petali per applicazione a modulo saturate

Si può intanto notare come l'intersezione dei tre insiemi costituisca un buon 22.52%, percentuale di fault easy-to-cover abbastanza alta, indice del fatto che il modulo sia facile da testare su alcune porzioni di hardware, ragionevolmente rappresentate da coni logici in ingresso e in uscita. Notiamo poi petali abbastanza ben equilibrati, significativo del fatto che la direzione per il testing intrapresa sia stata corretta, andando via via a coprire sempre parti di hardware non ancora testate. Le percentuali coperte da questi sono: 8.07% per l'approccio ATPG, 13.05% per quello manuale e 9.82% per la tecnica basata su μGP. Riguardo a questo, c'è da dire che, l'approccio utilizzato è stato quello incrementale, quindi magari ci sarebbero state configurazioni migliori con bilanciamenti migliori, ma le scelte fatte a un certo passo sono state ovviamente condizionate da decisioni precedenti; i risultati ottenuti possono comunque ritenersi soddisfacenti.

Una nota anche all'1.83% di sovrapposizione tra i soli sat_MANUAL e sat_uGP, il che suggerisce una buona impostazione dell'algoritmo evolutivo a partire dai programmi manuali scritti precedentemente, non dando vita a una grossa sovrapposizione (a parte quei fault easy-to-cover). Questa osservazione avvalora inoltre l'ipotesi che effettivamente istruzioni diverse utilizzino, all'interno del modulo, parti di hardware differenti, magari alternandole su suggerimento dell'opcode. Questa ipotesi non è chiaramente dimostrabile non potendo ragionevolmente verificare tale comportamento a livello gate, questo vista anche l'azione dell'ottimizzatore e l'innumerevole quantitativo di porte logiche e di segnali.

Si sottolinea inoltre come questi risultati, per alcuni aspetti, possano anche essere ottenuti mediante l'utilizzo di TetraMAX, svolgendo svariate operazioni. Questo approccio non è da considerarsi applicabile in tempi ragionevoli, ma, per questioni di validazione del tool, è stato impiegato su alcuni casi, anche complessi. Si sono ottenuti buoni risultati, ossia uguali a quelli forniti dal tool "Polito Fault Lists Analyzer", verificando la veridicità delle percentuali fornite.

Capitolo 5

Conclusioni

L'obiettivo del raggiungimento del valore di copertura target dell'80% complessivo sull'intero core è stato soddisfatto ampiamente, grazie anche ai risultati ottenuti sui moduli presi in esame. Le percentuali di fault coverage sono risultate adeguate, nella maggior parte dei casi anche al di sopra del valore target. Per quanto riguarda i casi in cui questo non è stato raggiunto, si vuole ricordare l'obiettivo generale che è quello del raggiungimento del valore target sull'intero core. Questo significa sì che ogni modulo vada testato cercando di ottenere questo risultato, senza dimenticare però il numero di fault che interessano ogni singola unità; infatti è di maggiore interesse un modulo complesso (con molti fault possibili) per l'apporto significativo che dà alla percentuale complessiva.

Oltre a ciò è stato possibile automatizzare l'analisi dei risultati forniti dai diversi programmi di test sotto forma di fault list. Sono state infatti analizzate diverse caratteristiche d'insieme dovute all'impiego di più programmi di test operanti sugli stessi fault. Questo fornendo, grazie al tool di analisi delle fault list, informazioni macroscopiche relative alla copertura dei guasti, come ad esempio indicazioni sui fault easy-to-cover o le rappresentazioni dei singoli petali che consentono di dare una chiara indicazione della bontà del singolo programma di test.

Una buona parte dei contenuti riportati in questa trattazione hanno inoltre contribuito alla redazione di una pubblicazione scientifica [11], presentata in occasione del "25th IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)", nell'ottobre del 2017.

5.1 Risultati ottenuti

I risultati ottenuti vengono qui divisi in due parti: i primi relativi ai programmi di test scritti per riuscire a garantire i valori di copertura target, i secondi in relazione alla redazione del tool di analisi di fault list "Polito Fault Lists Analyzer".

Per quando riguarda i programmi di test si riportano in Tabella 5.1 i risultati ottenuti, accorpati per tecnica di implementazione comune, in modo da avere risultati significativi e completi. Si sottolinea come i seguenti tempi di simulazione derivino da un'impostazione della frequenza di lavoro del processore a 16MHz.

Programma di test	**Tempo simulazione**	**Tempo fault-simulazione**	**Occupazione in flash**
brinc	$\sim 2698\mu$s	$\sim$ 20'	1,3 KB
circular_buffer	$\sim 2104\mu$s	$\sim$ 30'	896 B
saturate_ATPG	$\sim 10374\mu$s	$\sim$ 40'	972 B
saturate_MANUAL	$\sim 23094\mu$s	$\sim$ 1h	2,9 KB
saturate_μGP	$\sim 12526\mu$s	$\sim$ 40'	8,6 KB

Tabella 5.1: Informazioni relative ai programmi di test realizzati

Per quanto riguarda invece i valori di fault coverage, si possono osservare in Tabella 5.2 i valori di copertura ottenuti. Questi sono forniti per entrambe le pipe del processore dual issue e, in relazione ai risultati del modulo saturate, sono riportate tra parentesi tonde le percentuali cumulative, evidenziando l'aumento del valore di copertura col procedere dell'approccio incrementale.

Programma di test	**Pipe**	**Numero di fault**	**Fault coverage**
brinc	0	724	91,71%
	1	614	93,16%
circular_buffer	entrambe[1]	4738	73,26%
saturate_ATPG	0	8347	60,87%
	1	8057	60,61%
saturate_MANUAL	0	8347	52,65% (75,70%)
	1	8057	51,94% (75,50%)
saturate_μGP	0	8347	49,14% (85,19%)
	1	8057	49,73% (85,66%)

Tabella 5.2: Risultati di fault coverage relativi ai programmi di test realizzati

Per quanto riguarda invece il tool di analisi, si possono riassumere le seguenti caratteristiche:

- linguaggio utilizzato: Java 8
- linee di codice: $\sim$ 2300
- dimensione su disco del file eseguibile: 2,8 MB
- tempo di elaborazione (esempi):
  - ◊ caso saturate (3 fault list da 16.410 fault): $\sim$ 2s
  - ◊ caso Bernina (intero chip) (18 fault list da 729.522 fault): $\sim$ 1h

[1] Modulo comune ad entrambe le pipe.

5.2 Sviluppi futuri

Viste le difficoltà incontrate, potrebbe essere interessante, riguardo al modulo saturate, riuscire a raggiungere valori di copertura ancora maggiori o ottimizzare il codice in modo tale da garantire i medesimi risultati a fronte di un'occupazione in memoria flash inferiore e tempi di esecuzione più brevi.

In linea più generale si potrebbe estendere l'approccio garantito dalla tecnica manuale "walking bit" ad altri moduli aritmetici del processore, cercando di adattarla allo specifico caso di studio. Questo potrebbe essere fatto in prima istanza manualmente, ma poi si potrebbe estenderne ed automatizzarne l'impiego grazie all'utilizzo di μGP, impostando le istruzioni corrette e gli opportuni valori dei pattern.

Per quanto riguarda invece il tool "Polito Fault Lists Analyzer", come qualsiasi programma, potrebbe essere migliorato, partendo dall'interfaccia grafica, ma soprattutto a livello di funzionalità, estendendo le informazioni riportate basandosi comunque su una struttura dati completa e ben formata, in grado di consentire qualsiasi tipo di operazione sulle caratteristiche dei fault in possesso.

In seconda battuta si potrebbero adoperare strutture dati differenti per il salvataggio delle informazioni, a seconda dell'utilizzo dei dati che si intende fare. Questo chiaramente in un'ottica di ottimizzazione, soprattutto per quanto concerne l'impiego su molte fault list composte da un elevato numero di fault.

Bibliografia

[1] S. M. Thatte, J. A. Abraham, *Test Generation for Microprocessors*, IEEE Trans. Computers, Volume C-29, Issue 6, pp. 429-441, Gennaio 1980

[2] A. Paschalis, D. Gizopoulos, N. Kranitis, M. Psarakis, Y. Zorian, *Deterministic software-based self-testing of embedded processor cores*, Design, Automation and Test in Europe (DATE), pp. 92-96, 2001

[3] C. H. P. Wen, Li. C. Wang, K.-T. Cheng, *Simulation-based functional test generation for embedded processors*, IEEE Trans. Computers, Volume 55, Issue 11, pp. 1335-1343, Novembre 2006

[4] M. A. Skitsas, C. A. Nicopoulos, M. K. Michael, *DaemonGuard: OS-assisted selective software-based Self-Testing for multi-core systems*, IEEE Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), pp. 45-51, Ottobre 2013

[5] M. Psarakis, D. Gizopoulos, E. Sánchez, M. Sonza Reorda, *Microprocessor Software-Based Self-Testing*, IEEE Design & Test of Computers, 2010, Volume 27, Issue 3, pp. 4-19, 22 Gennaio 2012

[6] F. Reimann, M. Glass, A. Cook, L. Rodríguez Gomez, J. Teich, D. Ull, H. J. Wunderlich, U. Abelein, P. Engelke, *Advanced diagnosis: SBST and BIST integration in automotive E/E architectures*, ACM/EDAC/IEEE Design Automation Conference (DAC), pp. 1-6, Giugno 2014

[7] A. Paschalis, D. Gizoupoulos, *Effective Software-Based Self-Test Strategies for On-Line Periodic Testing of Embedded Processors*, Design, Automation and Test in Europe Conference (DATE) and Exhibition, 16-20 Febbraio 2004

[8] P. Bernardi, R. Cantoro, S. De Luca, E. Sánchez, A. Sansonetti, *Development Flow for On-Line Core Self-Test of Automotive Microcontrollers*, IEEE Transactions on Computers, 2016, Volume 65, Issue 3, pp. 744-754

[9] P. Bernardi, M. Grosso, E. Sánchez, O. Ballan, *Fault grading of software-based self-test procedures for dependable automotive applications*, Design, Automation & Test in Europe Conference & Exhibition (DATE), pp. 1-2, Marzo 2011

[10] W. C. Moyer, *Method for implementing a bit-reversed increment in a data processing system*, Patent grant US 20090327332 A1, 30 Giugno 2008

[11] P. Bernardi, S. De Luca, D. Piumatti, S. Regis, E. Sanchez, A. Sansonetti, *On the In-field Testing of Spare Modules in Automotive Microprocessors*, 25th IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC), Ottobre 2017.

[12] S. Sobek, K. Burke, *PowerPC Embedded Application Binary Interface (EABI): 32-Bit Implementation*, disponibile all'indirizzo http://www.freescale.com

[13] ISO/DIS26262, Road vehicles - functional safety, 2009

[14] TetraMAX ATPG Reference Manual, disponibile all'indirizzo https://www.synopsys.com

[15] μGP wiki, disponibile all'indirizzo http://ugp3.sourceforge.net

[16] Variable-Length Encoding VLE extension programming interface manual, disponibile all'indirizzo http://www.nxp.com

[17] Lightweigth Signal Processing APU Reference Manual, disponibile all'indirizzo http://www.freescale.com