POLITECNICO DI TORINO

Corso di Laurea Magistrale in Ingegneria Informatica

Tesi di Laurea Magistrale

Sviluppo di programmi di collaudo per test on-line di processori embedded usati nel settore automotive

Relatori:
Ernesto SANCHEZ
Davide PIUMATTI

Candidato:
Cozmin POGONEA

Dicembre 2019

Indice

1 Introduzione **6**

1.1 Sicurezza 8

1.2 Obiettivo 9

2 Background **10**

2.1 Software-Base Self-Test 10

2.2 Fault models 11

2.2.1 Stuck-at model 11

2.2.2 Single Stuck-at model 11

2.2.3 Categorie e classi di fault 14

2.3 Calcolo della fault coverage 18

2.4 Tool utilizzati 19

2.4.1 NCSIM 19

2.4.2 TetraMAX 21

2.4.3 IMC 23

2.4.4 FLAT 25

2.5 Processo di classificazione dei guasti 28

2.5.1 Preparazione del programma di test 29

2.5.2 Simulazione 29
2.5.3 Fault simulation 30
2.6 Costruzione STL 32

3 Approccio proposto **35**
3.1 Vincoli da soddisfare 35
3.2 Struttura programmi di test 38
3.3 Tecniche per la scrittura di programmi di test non intrusivi . . 42
3.3.1 Unità aritmetica 43
3.3.2 Unità di load e store 44
3.3.3 Register File 46
3.3.4 Unità di Branch Prediction 48
3.3.5 Unità di decodifica 50

4 Caso di studio **52**
4.1 Descrizione microcontroller 52
4.2 Descrizione core e200z4d 55
4.2.1 Set di istruzioni 57
4.2.2 Modello dei registri 58

5 Risultati sperimentali **63**
5.1 Risultati prodotti dalla suite di test intrusiva 63
5.2 Risultati prodotti dalla suite di test non intrusiva 66

6 Conclusione **69**

Bibliografia **71**

Elenco delle figure

2.1 AND gate SSA faults 13
2.2 Schermata di Sim Vision per la visualizzazione di forme d'onda 20
2.3 Moduli che costituiscono il cpu_dpath 22
2.4 Moduli che costituiscono il cpu_control 22
2.5 Schermata di IMC 24
2.6 Classificazione difficoltà individuazione guasti 26
2.7 Struttura interna FLAT 27
2.8 Processo di classificazione dei guasti 28
2.9 Processi di fault simulation 31
2.10 Confronto di due programmi di test 33

3.1 Struttura programmi di test 38
3.2 Struttura stack frame 41
3.3 ALU 43
3.4 Tecnica pattern a scacchiera 45
3.5 Strategia testing Register File 46
3.6 Interazioni di una BTB con le diversi fasi di una pipeline . . . 48
3.7 Strategia di test dell'unità di decodifica 51

4.1 Struttura cpu 54
4.2 Diagramma a blocchi core 56
4.3 Registri livello utente 60
4.4 Registri livello supervisor 62

Elenco delle tabelle

3.1 Uso registri PowerPC EABI 40

5.1 Composizione della suite di test intrusiva 64

5.2 Risultati suite di test intrusiva 65

5.3 Composizione della suite di test non intrusiva 67

5.4 Risultati suite di test non intrusiva 68

Chapter 1

Introduzione

La diffusione dei dispositivi elettronici in ambito automotive sta aumentando con grande velocità. I produttori di autoveicoli richiedono dispositivi sempre più veloci, meno costosi, che consumano meno e con un alto grado di affidabilità.

Questi dispositivi elettronici sono comunemente etichettati come critici per la sicurezza. Una delle tecniche comunemente adottate per aumentare la sicurezza è il test sul campo, che può avvenire sia tramite l'inserimento di hardware apposito con il compito di monitorare lo stato di salute dei vari elementi del dispositivo (es. core del processore, memoria) durante la fase di accensione o spegnimento, sia attraverso l'uso di moduli ECC (error correcting code) che testano il sistema in modo concorrente durante il suo normale funzionamento. Il metodo di testing basato sull'uso di hardware aggiuntivo può garantire un'alta percentuale di copertura dei guasti, ma risulta anche invasivo e costoso. Dall'altro canto il test sul campo è supportato anche da soluzioni software o Software-Based Self-Test (SBST). In questo caso il core

del processore è tenuto ad eseguire programmi di test accuratamente progettati e raccolti in una Software Test Library (STL), sia durante la fase di accensione e spegnimento, ma anche periodicamente durante il suo funzionamento. Questa metodologia di test non richiede alcun hardware aggiuntivo, ma comporta una grande difficoltà nello scrivere programmi di test efficienti.

Al giorno d'oggi i principali produttori di semiconduttori utilizzano STL composte da test sia intrusivi che non. I primi (chiamati anche test critici) sono eseguiti durante le fasi di accensione e spegnimento del dispositivo per non creare conflitti con il Sistema Operativo (SO). Infatti questi tipi di programmi di test hanno bisogno di privilegi supervisor in quanto utilizzano registri speciali, scatenano interruzioni etc.... I secondi (chiamati anche test a run-time) sono eseguiti dal SO come una normale applicazione che non richiede alcun privilegio speciale.

La strada seguita in quest'opera è quella del testing non intrusivo. Questo ha permesso di testare il dispositivo durante tutte le sue fasi di funzionamento ma ha anche comportato alcune limitazioni imposte dai vincoli che questa soluzione presenta.

1.1 Sicurezza

Data la crescente criticità dei compiti che i chip sono tenuti a svolgere, gli standard di sicurezza sono molto stringenti.

In ambito automotive lo standard che definisce la sicurezza funzionale per i veicoli stradali è l'ISO 26262. Da questo ne deriva anche l'Automotive Safety Integrity Level (ASIL) che è uno scheda di classificazione del rischio derivante dal Safety Integrity Level usato nel IEC 61508. L'ASIL viene stabilito eseguendo un'analisi dei rischi di un potenziale pericolo osservando la gravità, l'esposizione e la controllabilità dello scenario operativo del veicolo.

L'obiettivo di sicurezza per quel rischio a sua volta comporta i requisiti di ASIL. Ci sono quattro ASIL identificati dallo standard: ASIL A, ASIL B, ASIL C, ASIL D. ASIL D detta i requisiti di integrità più alti sul prodotto e ASIL A il più basso.

1.2 Obiettivo

L'obiettivo di questo lavoro è quello di presentare le varie difficoltà che si incontrano quando si produce una libreria di test usata in ambienti critici per la sicurezza. Una gran parte di queste difficoltà è dovuta ai numerosi vincoli a cui si deve sottostare usando metodi di test non intrusivi in real-time. I vincoli però sono necessari per rendere tutto il più trasparente possibile al SO. Infatti quest'ultimo deve eseguire in parallelo sia codice utente sia i programmi di test. Andremo a parlare dei vincoli ma anche della struttura dei programmi di test che vanno a formare la nostra libreria, mettendo in evidenza i vantaggi e gli svantaggi rispetto ad un approccio intrusivo non real-time.

Chapter 2

Background

2.1 Software-Base Self-Test

Il Software-based self-testing (SBST) è un approccio di test non intrusivo in cui il core del processore è chiamato ad eseguire appositi programmi di test durante le fasi di accensione/spegnimento, ma anche periodicamente durante il suo normale funzionamento. Questa tecnica permette di non utilizzare hardware aggiuntivo e allo stesso tempo permette di eseguire i test at-speed (alla stessa frequenza di funzionamento del core).
Questi programmi di test (run-time tests) vengono allocati nella memoria Flash per poi essere eseguiti dal SO come una semplice applicazione che non necessità di nessun privilegio speciale. Essendo eseguiti a run-time i test hanno alcune limitazioni tra cui la necessità di non modificare la memoria RAM per evitare di sovrascriverne il contenuto. Inoltre per interagire correttamente con il codice del sistema operativo i testi devono soddisfare l'Embedded-Application Binary Interface (EABI).

Ogni programma di test produce una cosiddetta firma (signature) che viene poi confrontata con un risultato atteso presente nello stesso programma di test. In caso di non uguaglianza il processore viene considerato difettoso.

2.2 Fault models

In un circuito elettrico i guasti fisici si manifestano sotto forma di guasti elettrici che vengono poi interpretati come guasti (faults) a livello logico. I guasti fisici, essendo troppo numerosi e spesso non analizzabili, sono raggruppati in pochi o addirittura in un solo tipo di guasto. Questo rende possibile un'analisi del problema che altrimenti non sarebbe affrontabile. Il principale guasto che viene usato nei circuiti digitali è il famigerato Stuck-at Fault.

2.2.1 Stuck-at model

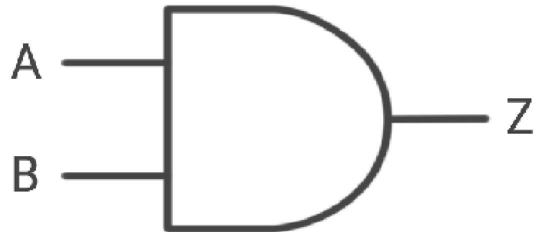
Nello Stuck-at model uno o più linee logiche sono bloccate o a zero o a uno. Questi tipi di fault prendono il nome rispettivamente di Stuck-at zero (SA0) e di Stuck-at one (SA1). Il modello che andremo a considerare è il Single Stuck-at (SSA) in cui soltanto una linea del circuito è bloccata a zero o a uno.

2.2.2 Single Stuck-at model

Prendendo come riferimento la Fig. 2.1, le proprietà che definiscono uno single Stuck-at fault sono:

- soltanto una linea nel circuito è difettosa

- la linea difettosa è permanentemente bloccata a 0 o a uno 1
- la linea difettosa può essere sia input che output di un gate



Inputs	FF	Faulty Response					
AB	Response	A/0	B/0	Z/0	A/1	B/1	Z/1
00	0	0	0	0	0	0	1
01	0	0	0	0	1	0	1
10	0	0	0	0	0	1	1
11	1	0	0	0	1	1	1

Figure 2.1: AND gate SSA faults

Inoltre dati i guasti F1 F2 e i rispettivi test T1 T2, si definiscono le seguenti proprietà:

1. Equivalenza

    - F1 è equivalente a F2 se T1 = T2
    - Qualsiasi test che rileva F1 rileva anche F2 e vice versa

2. Dominanza

    - F2 domina F1 se T2 $\subseteq$ T1
    - Un test che rileva F2 allora rileva anche F1
    - la relazione non è simmetrica

Usando tali proprietà possiamo collassare/eliminare rispettivamente tutti i guasti equivalenti e dominanti (Fault Collapsing).

2.2.3 Categorie e classi di fault

In base allo stato in cui si trovano, i fault sono assegnato a delle classi che a loro volta sono organizzate in categorie. Un simbolo di due caratteri viene utilizzato come nome abbreviato per entrambe le classi e categorie. Esistono 5 categorie di fault di livello superiore contenenti un totale di 17 classi di livello inferiore:

1. **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 attraverso simulazione o per un'analisi delle implicazioni.

   - **DR** (Detected robustly): fault rilevati utilizzando criteri WNR (weak non-robust), ROB (robust) o HFR (hazard-free robust)
   - **DS** (Detected by simulation): fault rilevati dopo un'esplicita simulazione dei pattern
   - **DI** (Detected by implication): fault non sono rilevati grazie a pattern specifici bensì grazie a un'analisi delle shifting scan chains
   - **D2** (Detected clock fault with loadable nonscan cell faulty value of 0 and 1)

2. **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 (che prende il significato di sconosciuto) piuttosto che un 1 o uno 0.

- **NP** (Not analyzed-possibly detected): identici ai fault AP eccetto per il fatto che o l'analisi non è stata completata o non si puòprovare 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)

3. **UD**: Undetectable

   Questa classe comprende quei fault che non possono essere rilevati (nè "fortemente" nè "possibilmente") sotto alcuna condizione. Nel calcolo del test coverage 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 unobservable): simili ai fault UU con la differenza che questi sono collocati su porte logiche inutilizzate con fanout. I fault su porte inutilizzate senza fanout sono invece marcati UU.
   - **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.

4. **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)

5. **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 é ancora stato trovato un pattern per osservare il fault.

2.3 Calcolo della fault coverage

Per quanto riguardo il calcolo della copertura dei guasti, esistono sostanzialmente due principali metodi chiamati rispettivamente fault coverage e test coverage. Nel caso della fault coverage la formula utilizzata è la seguente:

$$fault_coverage = \frac{DTfaults + (PTfaults * PTcredit)}{TOTALfautls}$$

Invece per il test coverage si utilizza:

$$test_coverage = \frac{DTfaults + (PTfaults * PTcredit)}{TOTALfaults - UDfaults - \frac{AU}{AN}faults * \frac{AU}{AN}credit}$$

I valori costanti definiti all'interno delle formule possono chiaramente assumere dei valori personalizzati, ma l'approccio di default prevede valori pari a: PTcredit = 0.5, $\frac{AU}{AN}$credit = 0. Con queste assunzioni si hanno 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 piu simile tra le due formule, dove questi fault vengono considerati non rilevati piuttosto che da non tenere in conto.

D'ora in avanti quando si farà accenno ai valori di copertura ottenuti, si intenderà sempre il valore della fault coverage. Come si può notare, il valore della fault coverage risulta essere sempre minore o uguale a quello della test coverage, andando di fatto a costituire il caso peggiore di copertura tra i due.

2.4 Tool utilizzati

2.4.1 NCSIM

NCSIM è una suite di strumenti sviluppata da Cadence Design Systems e si occupa della progettazione e della verifica di ASIC, SoC e FPGA. In realtà la suite prende il nome di Incisive, mentre NCSIM è il motore di simulazione principale. Gli strumenti che sono stati utilizzati in questo contesto sono:

- 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 sia da riga di comando batch.
- Sim Vision: visualizzatore grafico autonomo di forma d'onda e tracer di netlist.

Grazie a NCSim è stato possibile osservare a schermo i vari segnali presenti all'interno del core(Fig. 2.2), determinando con assoluta precisione il comportamento dei programmi di test. In particolare è stato possibile osservare il numero di clock necessari a completare i test, quando venivano eseguite le istruzioni, gli input e gli output prodotti. In questo modo abbiamo potuto

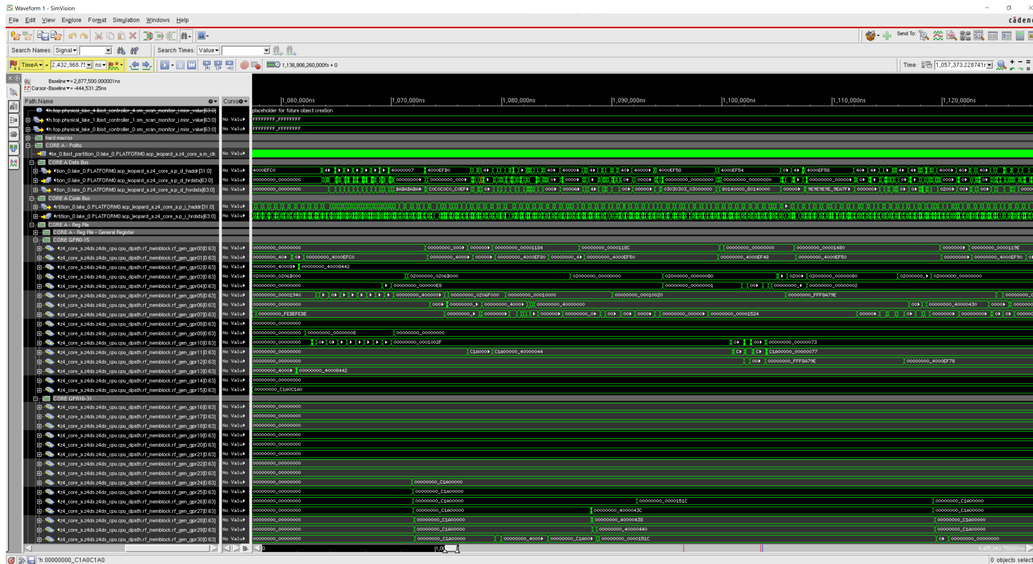


Figure 2.2: Schermata di Sim Vision per la visualizzazione di forme d'onda

determinare che i test soddisfacessero i vari vincoli e si comportassero nella maniera voluta.

NCSim oltre a consentire la visualizzazione del codice Verilog, consente anche l'osservazione a livello RTL e gate dei vari moduli presenti all'interno del core. Questo ci ha permesso di capire come sono fatti internamente i vari moduli e come testarli nella maniera più efficace.

2.4.2 TetraMAX

TetraMAX è un 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;
- controllo completo della regola di controllo della scansione e della compressione;
- integrazione di un simulatore dei guasti per la classificazione dei modelli di test strutturali;
- supporto per processori multi core per esecuzioni più veloci;
- interfaccia grafica integrata comprensiva di una visualizzazione gerarchica dei moduli e di un visualizzatore di forme d'onda.

Nelle figure 2.3 e 2.4 viene riportata la copertura dei guasti dei vari moduli presenti nel core.

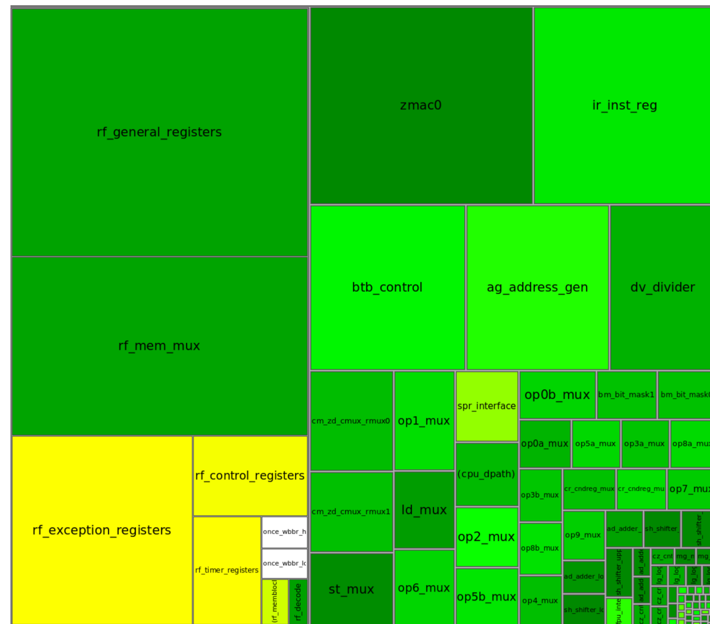


Figure 2.3: Moduli che costituiscono il cpu_dpath

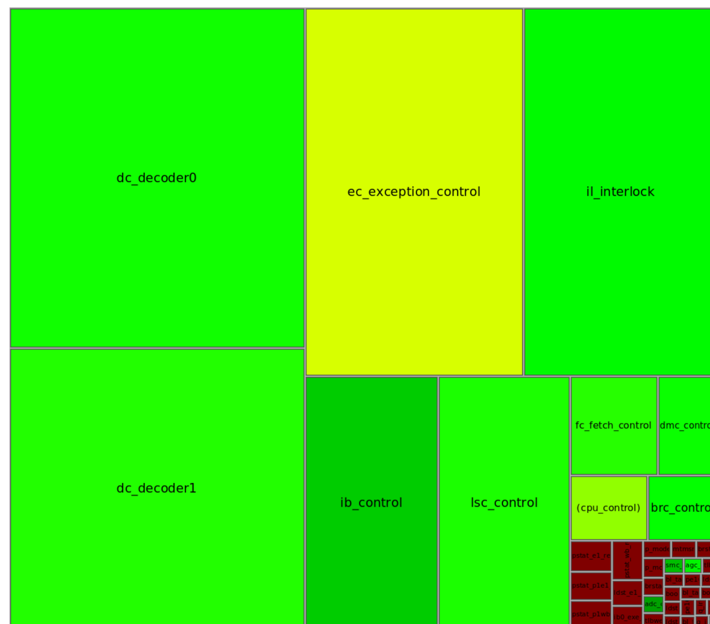


Figure 2.4: Moduli che costituiscono il cpu_control

2.4.3 IMC

IMC (Integrated Metrics Center) è un strumento di analisi della copertura a livello RTL progettato dalla Cadence per caricare e analizzare tale copertura per molteplici parametri. Fornisce una ricca interfaccia grafica per una grande quantità di valori di copertura, da quella del codice a quella funzionale. Con questo strumento la Cadence mette a disposizione un ambiente semplice e unificato di analisi e visualizzazione, che combinato con un database di copertura garantisce interoperabilità tra i vari strumenti funzionali di verifica della 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 di copertura supportata. Supporta tutte le metriche standard RTL garantendo il funzionamento sia fondato su singola esecuzione che su più esecuzioni unite.

Di seguito vengono riportate le principali informazioni estrapolabili da IMC:

- 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 veri

care 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 circa come determinati risultati sono stati raggiunti da programmi di test differenti.

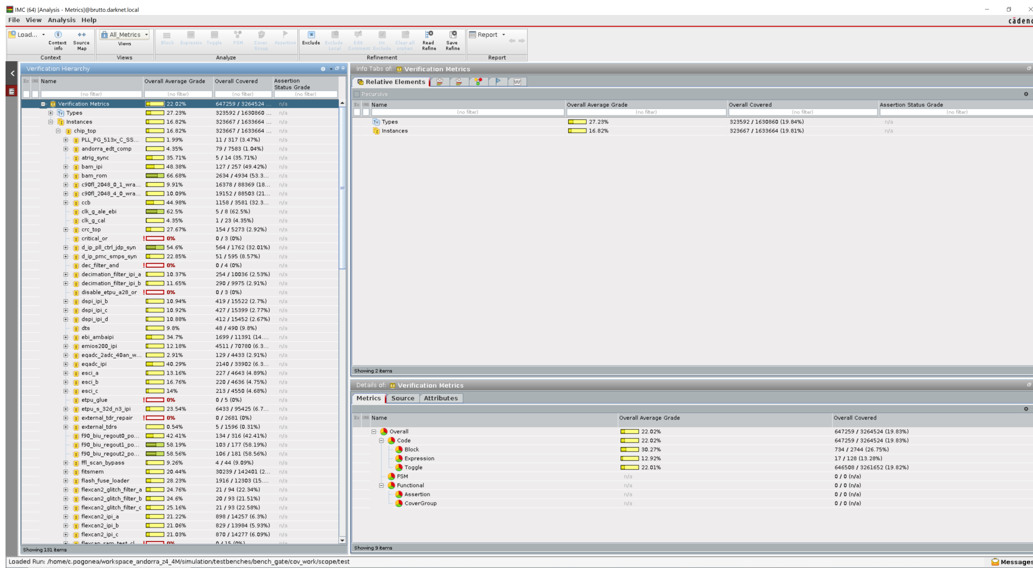


Figure 2.5: Schermata di IMC

2.4.4 FLAT

FLAT (Fault-List Analysis Tool)[7] è uno strumento sviluppato dal Politecnico di Torino con l'obiettivo di supportare la fase di sviluppo di Software Test Libraries (STL) eseguendo operazioni di fault-list sui risultati preliminari. In particolare il tool ha le seguenti funzionalità:

1. Dato uno specifico guasto, lo strumento è in grado di mostrare quali test sono stati in grado di individuare tale guasto.

2. Riporta il contributo che ogni programma di test dà alla copertura totale; questa analisi può essere fatta a diversi livelli, a partire da tutto il core fino ad arrivare alla singola unità (per esempio una singola unità aritmetica o logica).

3. Permette di aggiungere o togliere dei programmi di test dalla STL in base alla Fault Coverage ottenuta.

4. Dà la possibilità di valutare quali test sono necessari per valutare una soluzione ottimale di Fault Coverage della STL; a questo scopo il tool fornisce anche degli indici di qualità

5. Mostra l'effetto a cascata dei programmi di test [3]. In altre parole questo permette di vedere gli effetti che un particolare programma di test, sviluppato per una specifica unita, ha su tutto il core del processore.

6. E' possibile anche fare una classificazione della difficoltà di individuare vari guasti del processore (2.6). In questa classificazione troviamo 4 categorie:

(a) **Very Hard to Detect (VH2D)**: guasto individuato da un solo programma di test.

(b) **Hard to Detect (H2D)**: guasto individuato da due programmi di test.

(c) **Easy to Detect (E2D)**: guasto individuato da più di due programmi di test.

(d) **Very Easy to Detect (VE2D)**: guasto individuato da tutti i programmi di test.

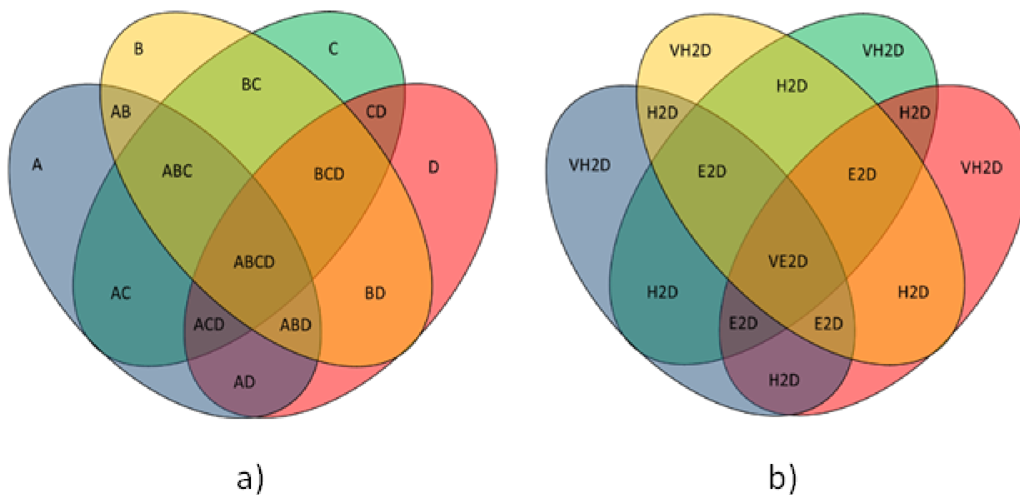


Figure 2.6: Classificazione difficoltà individuazione guasti

La struttura interna di FLAT si può osservare nella Fig. 2.7. All'interno è presente un Database di tipo PostgreSQL che contiene una tabella per ogni fault list da analizzare. I dati vengono estrapolati dal Database usando un generatore automatico di query che in base agli input del utente

genera tali query. FLAT può essere usato sia da linea di comando, sia da interfaccia grafica ma può anche essere richiamato da un altro programma usando un'apposita API. FLAT è basato sulla versione 3.6 di Python mentre l'interfaccia grafica è stata creata usando la versione gratuita di QT.

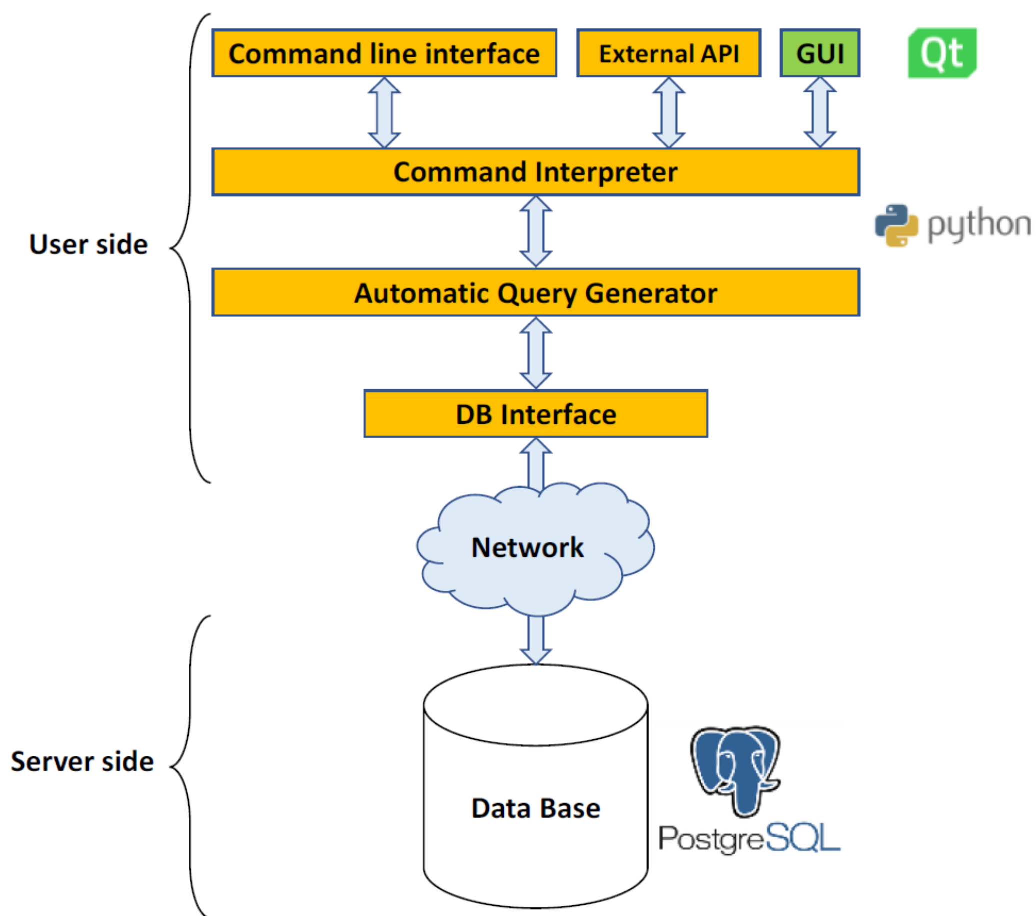


Figure 2.7: Struttura interna FLAT

2.5 Processo di classificazione dei guasti

La classificazione dei guasti risulta essenziale all'interno di una procedura SBST e consta nella misurazione dell'abilità della stessa di rilevare i guasti che potrebbero influenzare negativamente il comportamento dell'intero chip o parte di questo. Il processo non è banale come può dimostrare lo schema in Fig. 2.8, 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 simulation.

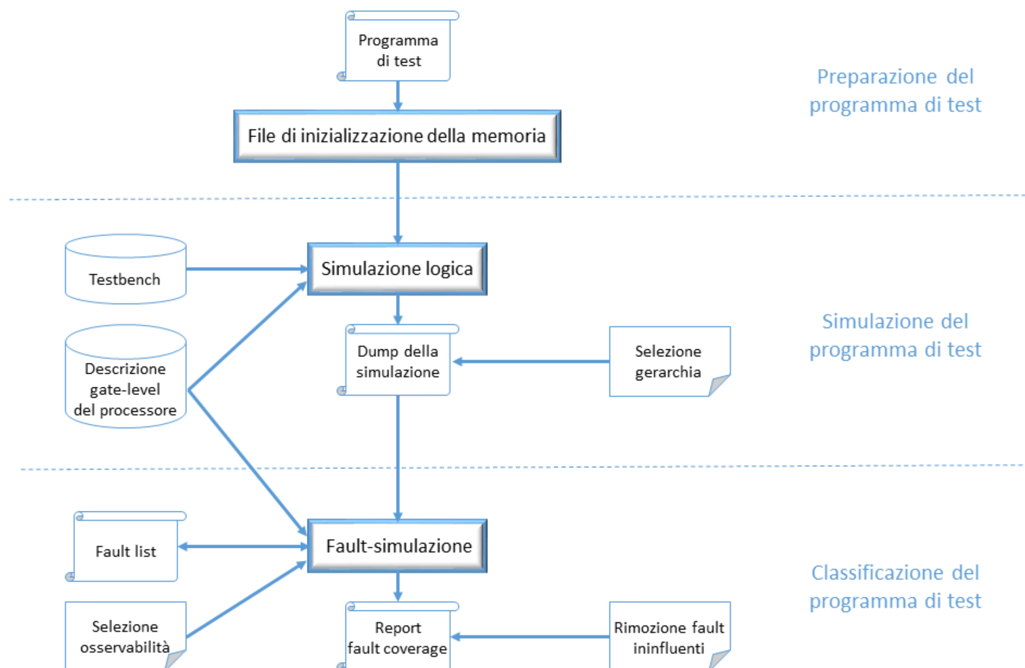


Figure 2.8: Processo di classificazione dei guasti

2.5.1 Preparazione del programma di test

La prima fase del processo di classificazione è formato dalla preparazione del programma di test. Questa viene fatta studiando le funzionalità del modulo per il quale si intende sviluppare tale test per poi passare alla parte di scrittura (che in questo caso viene fatta in linguaggio assembler). In questo contesto per alcuni moduli (es. il modulo aritmetico) sono stati usati in input pattern generati con tecniche ATPG per poter accelerare lo sviluppo della suite di test. Una volta scritto il programma di test, questo verrà compilato e convertito in un formato compatibile con gli ambienti di simulazione e simulazione dei guasti.

2.5.2 Simulazione

La seconda fase si individua nella simulazione del circuito avendo cura di emulare il comportamento desiderato del dispositivo. E' necessario tenere in conto anche dell'ambiente in cui opera il processore, in particolare essendo i test eseguiti a run-time insieme ad altri programmi, questi non devono interferire con il normale funzionamento del sistema. Il risultato del passo di simulazione è il file dump 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 eettuare la simulazione, in modo che si risparmi tempo al passo successivo.

2.5.3 Fault simulation

Uno dei costi maggiori nella generazione di una STL è dato dalla potenza computazionale richiesta dalla fault simulation. Prendendo come esempio un processore con circa 200k stuck-at faults, per fault simulare un programma di test della durata di 1ms, usando un processore quad-core a 2GHz, servirebbero circa 3 giorni. Tale costo diventa insostenibile se il processo è iterativo e bisogna produrre numeri programmi di test prima di ottenere una buona copertura dei guasti. L'approccio da noi seguito è quello illustrato in [3]. Questo ci ha permesso di ridurre notevolmente il tempo di sviluppo basandoci sui seguenti principi:

- **modularità:** un processore embedded è suddivisibile in numerosi moduli che possono essere considerati separatamente. Questo significa che tale processo di suddivisione è applicabile anche alla fault list.

- **parallelizzazione:** affrontando i moduli separatamente è possibile usare il processo di fault simulation su più workstation contemporaneamente.

- **side-effect:** sviluppando test specifici per le varie unità del processore, è probabile che si avrà effetto collaterale che consiste nell'individuazione di guasti anche negli altri moduli che non sono direttamente il target dei test specifici.

La strategia prevede l'esecuzione dei passaggi sotto elencati fino alla considerazione di tutti i moduli:

1. **generazione**, possibilmente in parallelo, di programmi di test per un

set di moduli interni al processore, scelti con attenzione, fino al raggiungimento di una copertura soddisfacente.

2. **esecuzione di una sincronizzazione** considerando tutti i moduli e i test precedentemente generati. Per sincronizzazione si intente l'esecuzione del processo di fault simulation non soltanto sul singolo modulo per il quale il test è stato scritto, ma su tutto il core.

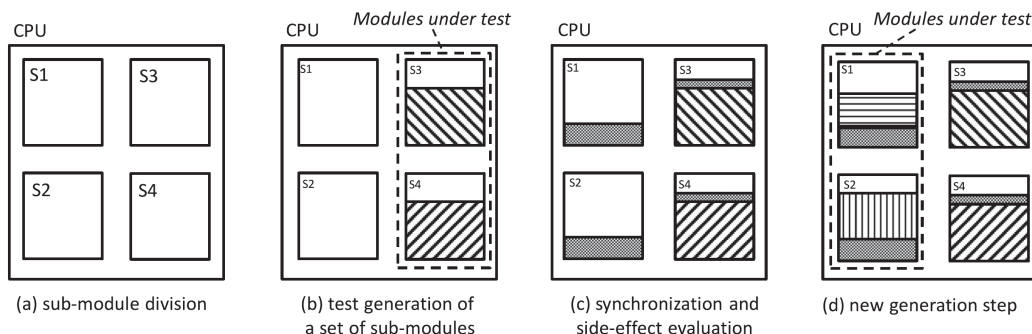


Figure 2.9: Processi di fault simulation

Prendendo come esempio la Fig. 2.9, si può vedere come tale processo sia suddiviso in quattro passaggi. Al punto (a) il processore viene suddivisi in numerosi sotto moduli (S1, S2, S3, S4). Al punto (b) invece vengono scelti due moduli in parallelo (S3, S4), e vengono sviluppati programmi di test specifici fino al raggiungimento di un livello di copertura soddisfacente. Si può quindi iniziare il processo di sincronizzazione (c) che produce un effetto collaterale positivo sia su S1 e S2, ma anche su S3 e S4. Questo succede perché i programmi di test scritti per S3 producono un effetto su S4 e viceversa.

Al punto (d) invece si riparte con il processo di generazione di nuovi programmi di test. Vale la pena specificare che i passaggi precedenti sono

serviti ad alleggerire la fault list di partenza dei moduli S1 e S2 prima di iniziare con il loro processo di generazione.

Il principale vantaggio che tale metodo offre è la velocità di sviluppo della suite di test perché:

- le fault list da prendere in considerazione sono in modo significativo più piccole della fault list completa, portando ad una veloce fault simulation che di solito è nell'ordine dei minuti;
- dopo ogni sincronizzazione, il numero di guasti attivi nei sotto moduli è ridotto, portando di nuovo ad un incremento della velocità.

2.6 Costruzione STL

Basandoci sulle informazioni fornite dal tool FLAT, in particolare gli indici di qualità, è stato possibile selezionare i programmi da scartare e quelli da inserire nella STL. Tali indici sono applicati nella situazione mostrata nella Fig. 2.10, dove un nuovo programma di test TN è integrato con uno precedente TP.

A + B rappresenta la percentuale di copertura di guasti del test già presente nella STL, mentre C rappresenta l'incremento della copertura apportato dal nuovo test TN. Sulla base di questo FLAT offre quattro diversi indici, due legali alla fault coverage, uno legato all'uso di memoria e infine uno legato al tempo tempo di esecuzione. Tali indici hanno un valore compreso tra 0 e 1, dove è indica il peggior risultato possibile mentre 1 indica la situazione ottimale. Gli indici in questione sono:

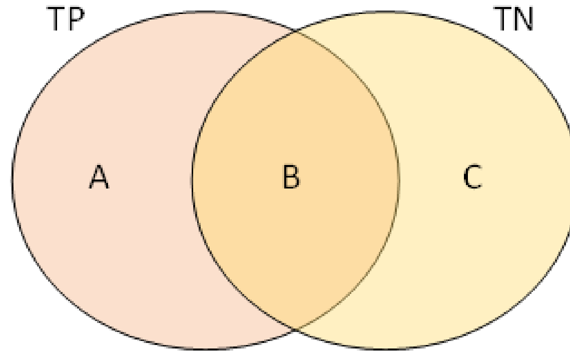


Figure 2.10: Confronto di due programmi di test

1. **Coverage Index of New Faults (INF):** questo indice considera i nuovi guasti coperti esclusivamente dal nuovo test TN (regione C rappresentata nella 2.10). Quindi in questo caso consideriamo

$$INF = C$$

2. **Index of test program Overlap (IO):** questo indice considera i guasti individuati dai due programmi di test TN e TP. Nella situazione ideale i guasti individuati da TN dovrebbero differire interamente da quelli TP.

$$IO = \begin{cases} 1 - \frac{A+C}{A+B+C} & \text{B} < \text{A} \\ 0 & \text{B} = \text{A} \end{cases}$$

3. **Memory Index (IM):** questo indica considera l'occupazione di memoria. Data una dimensione massima che la STL può occupare, la dimen-

sione di ogni programma di test è paragonata alla dimensione massima:

$$IM = 1 - \frac{MEMORIA_FLASH_TN}{DIMENSIONE_MASSIMA_STL - DIMENSIONE_ALTRI_TEST}$$

4. **Time Executed Index (IE):** Questo indice considera il tempo richiesto dal processore per eseguire il nuovo test TN. Dato un tempo massimo di eseguzione della STL, ogni programma di test è paragonato al tempo massimo di esecuzione:

$$IM = 1 - \frac{DURATA_TN}{DURATA_MAX_STL - DURATA_ALTRI_TEST}$$

Questo indice indice non è stato usato in quanto può essere usato solamente per test eseguiti a boot-time.

Usando i primi 3 indici sopra elencati è stato possibile scartare con precisione i test che apportavano un basso incremento di copertura in relazione alla memoria occupata e agli altri test.

Chapter 3

Approccio proposto

In questo capitolo verranno illustrati i vincoli (constraints) che sono stati soddisfatti durante la scrittura dei programmi di test, insieme ai vari moduli del processore che sono stati testati.

3.1 Vincoli da soddisfare

Data la natura dei test (run-time), ci si è trovati ad affrontare numerosi vincoli che di seguito vado ad elencare:

1. **Nessuna modifica ai Special-Purpose Registers (SPRs)**
   Dato che gli SPRs sono usati per configurare il core, la gestione di questi registri è riservata al SO.

2. **Nessuna modifica della memoria RAM**
   Dato che la memoria RAM è gestita dal sistema operativo come memoria di sistema, ogni modifica può comportare un'alterazione delle operazioni del codice applicativo.

3. **Tempo massimo di esecuzione**

   Le applicazioni automotive spesso sono eseguite in real time. Questo significa che l'intervallo di tempo riservato ad ogni programma di test è limitato. L'intervallo di tempo rappresenta il tempo massimo d'esecuzione che in questo caso è di 255 colpi di clock.

4. **Nessuna interruzione né eccezione**

   L'unità dedicata alle eccezioni è configurata per gestire diverse fonti di interruzione. Una o più System-Interrupt Service Routine (S-ISR) sono scritte per soddisfare una richiesta specifica. Un programma di test per le eccezioni fa uso di interruzioni e eccezioni scatenate di proposito e gestite da una Test-Interrupt Service Routine (TISR) come alternativa alla S-ISR. Dal momento che le ISR fanno parte del SO, questa modifica non è consentita.

5. **I test devono essere interrompibili**

   Ogni test deve essere interrompibile da una richiesta di interrupt. Questo implica che:

   - i registri dedicati al context switch non devono essere modificati
   - l'esecuzione del codice utente deve rimanere inalterata

   Per far si che questi requisiti siano soddisfatti si scrivono programmi di test comformi allo standard EABI.

6. **Nessun utilizzo di periferiche**

   I programmi di test non intrusivi non possono usare né configurare periferiche, in quanto questo altererebbe il comportamento generale del SoC.

7. **Configurazione User Mode**

   Il codice applicativo viene normalmente eseguito in User Mode (cioè con limitato accesso alle risorse del sistema). Dato che la STL è eseguita in modo concorrente insieme alle altre applicazioni software, anche i test devono essere eseguiti in User Mode.

3.2 Struttura programmi di test

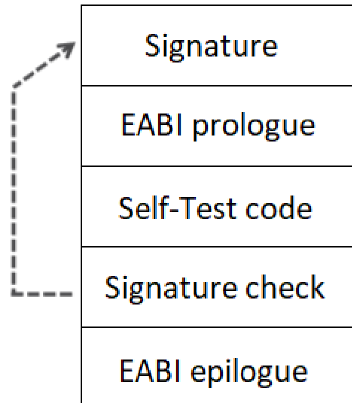


Figure 3.1: Struttura programmi di test

Come si può vedere nella Fig. 3.1 i programmi di test sono costituiti da diverse parti:

1. Nella parte 'SIGNATURE' è presente un'intestazione che riporta oltre al nome del test anche un suo id e una firma. La firma viene calcolata in base alle operazioni e agli input che un programma di test deve eseguire. Il test deve produrre sempre la stessa firma altrimenti il sistema presenta qualche guasto.

2. Nella parte 'EABI prologue' avviene il salvataggio dello stato del dispositivo.

3. La parte 'Self-Test Code' contiene invece le operazioni che il test deve eseguire e il risultato viene salvato in uno dei registri. Molti dei pattern

usati sono stati ottenuti impiegando tecniche ATPG mentre altri sono stati studiati ad-hoc.

4. 'Signature check' esegue la funzione di confronto tra il risultato prodotto dal test e quello atteso (la firma definita nella prima parte del test). Il risultato di questo confronto viene scritto in un registro scelto dall'utente.

5. Nella parte 'EABI epilogue' avviene il ripristino dello stato del dispositivo.

In particolare nelle due fasi 'EABI prologue/epilogue' viene rispettivamente creato/distrutto lo stack frame (Fig. 3.2) del programma di test. Lo stack è composto da numerosi stack frame e lo Stack Pointer (SP) punta sempre all'indirizzo più piccolo (la funzione in esecuzione) in quanto è organizzato partendo dall'indirizzo più alto in memoria. Ogni funzione che chiama un'altra funziona o altera lo stato di registri nonvolatile deve necessariamente creare uno stack frame. Questo operazione si fa decrementando lo SP di un valore pari allo spazio richiesto da tale funzione. La distruzione avviene invece aumentando lo SP di un valore pari alla dimensione dello stack frame di quella funzione.

Si è cercato di usare ove possibile solamente i registri definiti come 'Volatile' dal PowerPC EABI (Tabella 3.1), in modo da poter evitare le operazioni di salvataggio e ripristino di tali registri in quanto comporterebbero un utilizzo maggiore di colpi di clock.

Register	**Type**	**Used for**
R0	Volatile	Language Specific
R1	Dedicated	Stack Pointer (SP)
R2	Dedicated	Read-only small data area anchor
R3-R4	Volatile	Parameter passing / return values
R5-R10	Volatile	Parameter passing
R11-R12	Volatile	
R13	Dedicated	Read-write small data area anchor
R14-R31	Nonvolatile	
F0	Volatile	Language Specific
F1	Volatile	Parameter passing / return values
F2-F8	Volatile	Parameter passing
F9-F13	Volatile	
F14-F31	Nonvolatile	
Fields CR2-CR4	Nonvolatile	
Other CR fields	Volatile	
Other Registers	Volatile	

Table 3.1: Uso registri PowerPC EABI

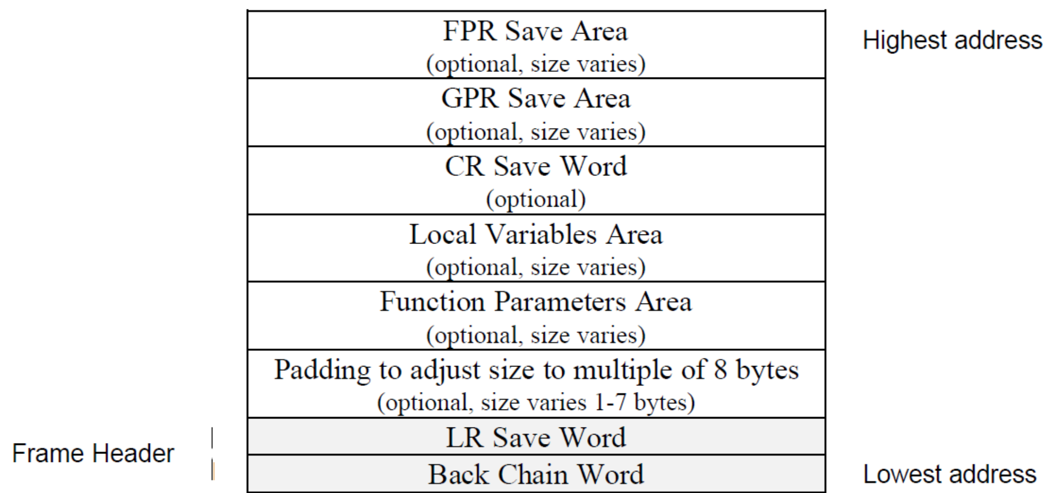


Figure 3.2: Struttura stack frame

3.3 Tecniche per la scrittura di programmi di test non intrusivi

In questa sezione viene proposte strategie per trasformare alcuni test intrusivi in test non intrusivi. Secondo [3], le unità all'interno di un microcontrollore possono in 5 categorie:

1. **Unità aritmetiche**
   dedicate all'esecuzione di specifiche operazioni aritmetiche e logiche.

2. **Unità speciali**
   dedicate alla gestione delle memorie, delle eccezioni e delle operazioni di load e store

3. **Unità Register File**
   contenente tutti i registri General Purpose (GPR) e Special Purpose (SPR)

4. **Unità di indirizzamento**
   che includono il calcolo del Effective Address, la gestione del Program Counter (PC) e l'unità di Branch Prediction (BPU)

5. **Moduli di controllo**
   dedicati alla gestione della pipeline, dei forwarding paths e la decodifica degli opcode delle istruzioni

Nelle sotto sezioni seguenti si descrivono nel dettaglio le tecniche e gli algoritmi usati per creare programmi di test non intrusivi per alcune delle unità o subunità sopra menzionate.

3.3.1 Unità aritmetica

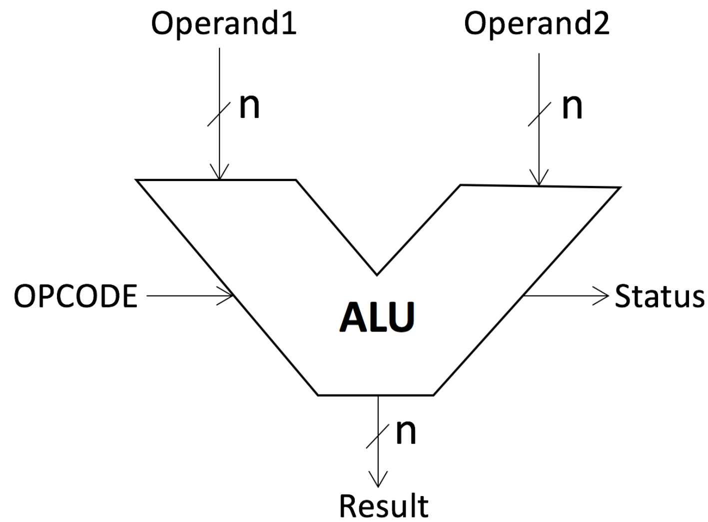


Figure 3.3: ALU

L'unità aritmetica (ALU) ha come input due operandi e un opcode. In base al opcode ricevuto verrà eseguita una certa operazione aritmetica o logica sugli operandi che produrrà un risultato e uno stato (es. overflow, zero etc..).

Quando si ha a che fare con questi moduli, i produttori di test spesso utilizzano tools di tipo Automatic Test Pattern Generation (ATPG). Per generare test non intrusivi, i pattern generati da questi strumenti sono stati suddivisi e usati come input da più programmi di test. Questo è stato necessario per poter soddisfare i vincoli di tempo.

3.3.2 Unità di load e store

L'unità di load e store è responsabile di tutte quelle operazioni di load e store da e verso la memoria. Generalmente viene testa facendo uso di programmi di test che utilizzano specifici indirizzi della memoria RAM. Una tecnica efficace consiste nel fare operazioni di load e store con pattern a scacchiera (come mostrato nella Fig. 3.4) in uno specifico indirizzo di memoria, salvando poi i risultati dei test sotto forma di firma. Nella prima fase dell'algoritmo viene eseguita un'operazione di store-word del pattern 0101...0, seguito da 4 operazioni d load-byte per poi finire con 2 operazioni di load-half-word. Questa sequenza operazioni viene poi nuovamente eseguita usando il pattern complementare 1010...0. La versione non intrusiva dell'algoritmo sopra descritto può essere creata in questo modo:

1. suddividdendo le operazioni in molteplici sotto-test
2. eseguendo le operazioni di di load e store esclusivamente nello stack frame

1.a) ST W 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1

1.b) LD B 0 1 0 1 0 1 0 1

1.c) LD B 0 1 0 1 0 1 0 1

1.d) LD B 0 1 0 1 0 1 0 1

1.e) LD B 0 1 0 1 0 1 0 1

1.f) LD HW 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1

1.g) LD HW 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1

Repeat from 1.a to 1.g with complementary test pattern

2.a) ST B 0 1 0 1 0 1 0 1

2.b) ST B 0 1 0 1 0 1 0 1

2.c) ST B 0 1 0 1 0 1 0 1

2.d) ST B 0 1 0 1 0 1 0 1

2.e) LD W 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1

Repeat from 2.a to 2.e with complementary test pattern

3.a) ST HW 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1

3.b) ST HW 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1

3.c) LD W 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1

Repeat from 3.a to 3.c with complementary test pattern

Figure 3.4: Tecnica pattern a scacchiera

3.3.3 Register File

Il Register File è l'insieme di tutti i registri del core. Questo comprende quindi sia i registri speciali (SPRs) che quelli general purpose (GPRs). Un modo efficace per testare il register file viene descritto in [4]. Tale approccio divide i registri in gruppi, secondo la distanza di Hamming tra le codifiche dei registri.

(Step 1)

```
pattnA equ 0x55555555
pattnB equ 0xAAAAAAAA
```

(Steps 2a-2b, first-half)

Instruction on *Pipeline A*	Instruction on *Pipeline B*	*Write* input
`move r1, #pattnA`	`move r4, #pattnA`	0101..01
`move r2, #pattnA`	`move r7, #pattnA`	0101..01
`move r0, #pattnB`	`move r5, #pattnB`	1010..10
`move r3, #pattnB`	`move r6, #pattnB`	1010..10

(Steps 2c-2d, first-half)

Instruction on *Pipeline A*	Instruction on *Pipeline B*	*Read1* output	*Read2* output
add r1,**r1**,**r1**	add r4,**r4**,**r4**	0101..01	0101..01
add r1,**r2**,r1	add r4,**r7**,r4	0101..01	--
add r1,r1,**r2**	add r4,r4,**r7**	--	0101..01
add r0,**r0**,**r0**	add r5,**r5**,**r5**	1010..10	1010..10
add r0,**r3**,r0	add r5,**r6**,r5	1010..10	--
add r0,r0,**r3**	add r5,r5,**r6**	--	1010..10
store r1	store r4		
store r0	store r5		

(Steps 2a-2b, second-half) Repeat (Steps 2a-2b, first-half) with inverted pipelines*

(Steps 2c-2d, second-half) Repeat (Steps 2c-2d, first-half) with inverted pipelines*

(Steps 3a-3b, first-half) Repeat (Steps 2a-2b, first-half) with inverted patterns**

(Steps 3c-3d, first-half) Repeat (Steps 2c-2d, first-half)

(Steps 3a-3b, second-half) Repeat (Steps 2a-2b, second-half) with inverted patterns**

(Steps 3c-3d, second-half) Repeat (Steps 2c-2d, second-half).

* Instructions on Pipeline *A* are executed on *Pipeline B* and vice-versa.

** Instructions using `#pattnA` use `#pattnB` and vice-versa.

Figure 3.5: Strategia testing Register File

Prendendo in considerazione una versione semplificata (Fig.3.5) dove sono presenti solamente 8 registri, l'algoritmo può essere suddiviso in 3 passaggi:

1. gli 8 registri sono divisi in 4 gruppi (Group A1: r1, r2, Group A2: r4, r7 Group B1: r0, r3, Group B2: r5, r6) e vengono scelti come input due pattern pattnA e pattnB che assumono rispettivamente i valori 0x55555555 e 0xAAAAAAAA;

2. in questa fase vi è la propagazione degli pattern nei vari gruppi di registri secondo la seguente logica:
   Pipeline A) Pattern A in Group A1, Pattern B in Group B1
   Pipeline B) Pattern A in Group A2, Pattern B in Group B2
   Dal momento che ogni pipeline copre solamente una metà dei registri, questa operazione viene svolta invertendo i gruppi sulle due pipeline:
   Pipeline A) Pattern A in Group A2, Pattern B in Group B2
   Pipeline B) Pattern A in Group A1, Pattern B in Group B1

3. nell'ultima fase dell'algoritmo vengono semplicemente invertiti i pattern tra i vari gruppi come elencato di seguito:
   Pipeline A) Pattern B in Group A1, Pattern A in Group B1
   Pipeline B) Pattern B in Group A2, Pattern A in Group B2
   Pipeline A) Pattern B in Group A2, Pattern A in Group B2
   Pipeline B) Pattern B in Group A1, Pattern A in Group B1

Nella versione non intrusiva di questo algoritmo di test soltanto i GPRs possono essere testati. Inoltre l'intero algoritmo può essere suddiviso in vari test più piccoli per poter soddisfare tutti i vincoli.

3.3.4 Unità di Branch Prediction

La BPU (Branch Prediction Unit) è una componente della CPU che cerca di prevedere l'esito di un'operazione su cui si basa l'accettazione di una istruzione di salto condizionato, evitando rallentamenti che possono essere molto evidenti in una architettura con pipeline. Il comportamento di una BTB e le sue interazioni con le diversi fasi di una pipeline sono rappresentati nella Fig. 3.6.

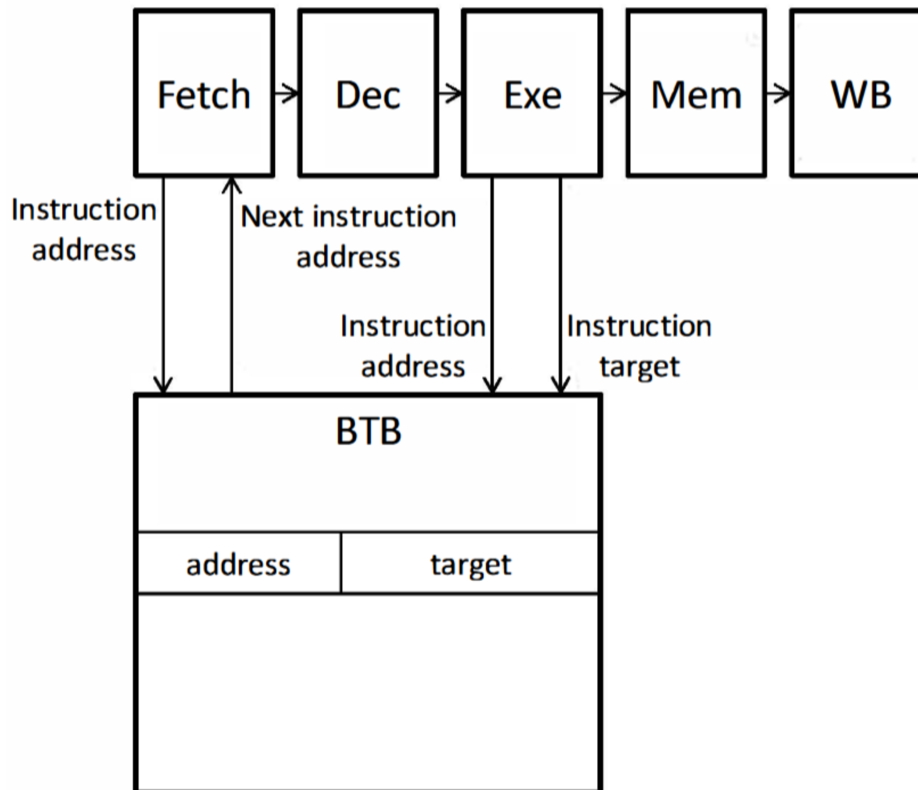


Figure 3.6: Interazioni di una BTB con le diversi fasi di una pipeline

Per fare un esempio molto pratico, la CPU si comporta come un turista che non sa quale strada prendere ad un bivio; mentre consulta la cartina, imbocca una strada sperando nell'intuito. Se indovina, evita di fermarsi al bivio e perdere tempo; se non indovina, non deve tornare al bivio, ma è come se ripartisse istantaneamente dal bivio. L'importanza di questa operazione è evidente soprattutto per i microprocessori moderni, superscalari e con lunghe pipeline, che per ogni errore di previsione devono sprecare molti cicli di clock di lavoro prezioso.

La BPU può essere efficacemente testata usando l'algoritmo March Test proposto in [1]. Tale algoritmo individua istruzioni di salto presenti a vari indirizzi di memoria, per poi testarne tutti gli stack-at faults associati a ciascun bit della tabella. Un soluzione efficiente è quella di scrivere dinamicamente istruzioni di salto a indirizzi di memoria prefissati, per fare in modo che quell'indirizzo vada a finire nella BTB. Il numero di indirizzi che si vuole prendere in considerazione deve essere uguale al numero di righe presenti nella BTB. Dal momento che questo algoritmo viola i vincoli di tempo, una versione non intrusiva può essere creata suddividendo il tutto in vari sotto test. Il numero di salti che possono essere eseguiti durante questi test è fortemente di tempo dai vincoli di tempo presenti. Nel peggiore dei casi si avranno a disposizione soltanto un paio di salti che sono sufficienti a caricare soltanto una entry nella BTB. L'effetto collaterale di questa suddivisione è un comportamento non deterministico. Inoltre, dato che la BTB non viene inizializzata prima di eseguire il test e dato che la tecnica usata come politica di sostituzione delle entry è la Least-Frequently Used (LFU), ogni test deve essere eseguito più volte per coprire tutte le entry. Per di più, dato che il

candidato per la sostituzione non è noto in anticipo a runtime, il problema deve essere affrontato statisticamente usando il teorema probabilistico della distribuzione geometrica.

Per le distrubuzione geometriche, data una variabile aleatoria X, il valore atteso E(X) si calcola usando la formula E(X) = 1/p, dove x è il numero di prove necessarie per ottenere il primo successo (nel nostro caso equivale al numero di volte che un programma deve essere eseguita prima di avere una hit diversa da quella precedente), e p è la probabilità di successo di una hit diversa. Il numero medio di volte che il test deve essere eseguito prima di poter coprire tutte le entries della BTB è dato dalla 3.1, dove n sono il numero di entries.

$$\sum_{i=1}^{n} E(X_i) = 1 + \frac{n}{(n-1)} + ... + \frac{n}{(n-(n-1))} = n \sum_{i=1}^{n} \frac{1}{i} \tag{3.1}$$

Se consideriamo una BTB con 8 entry, la probabilità di successo relativa alla prima entry è 1, alla seconda 7/8, alla terza 6/8 e cosi via. Secondo la formula ogni test deve essere eseguito 22 volte per coprire tutte le entries.

Il secondo problema riguarda l'allocazione dei test nella RAM. Dato che i programmi di test non possono essere liberamente allocati nella RAM in quando devono coesistere insieme al codice applicativo dell'utente, l'unica soluzione percorribile è quella di replicare i test numerose volte nella memoria.

3.3.5 Unità di decodifica

L'unità di decodifica si occupa di ricevere le istruzioni in ingresso e di attivare le opportune unità interne del processore per eseguire l'istruzione caricata.

Questa unità può essere testata usando l'approccio proposto in [2], che fa uso del manuale utente per determinare se un'istruzione è legale oppure no. In particolare questa procedura identifica le istruzioni che differiscono tra di loro soltanto per un bit (distanza di Hamming di 1-bit). In questo modo un cambiamento nell'istruzione da testare produrrà una firma errata.

Consideriamo ora le due istruzioni presenti nella Fig. 3.7. Esse differiscono tra di loro soltanto per il ventinovesimo bit. Eseguendo queste istruzioni una dopo l'altra, se è presente uno stack-at fault al bit 29, una delle istruzioni si trasformerà nell'altra.

Opcode bit test	First legal instruction		Second legal instruction	
	opcode	Mnemonic	opcode	mnemonic
29	7C000214	add r0 r0 r0	3C000214	lis r0, 0x124

Figure 3.7: Strategia di test dell'unità di decodifica

Questo approccio permette anche di trovare coppie di istruzioni di cui una è legale mentre l'altra non lo è. Nel caso di un approccio di test non intrusivo questa tecnica non può essere seguita in quanto un'istruzione illegale scatena un'interrupt. E' stato quindi necessario scrivere molteplici programmi di test che soddisfacessero i requisiti di tempo.

Chapter 4

Caso di studio

4.1 Descrizione microcontroller

Il microcontrollore utilizzato è basato sull'Architettura Power (Power Architecture) e integra diverse soluzione per applicazioni automotive tra cui il controllo del servosterzo, il controllo del telaio e più in generale qualsiasi applicazione critica che richiede standard di sicurezza elevati.

Il dispositivo è basato su un'architettura di sicurezza dual-core che soddisfa gli standard di sicurezza ISO26262 ASILD e IEC61508 SIL3. Per minimizzare l'utilizzo di software e moduli aggiuntivi necessari a soddisfare gli standard sopra elencati, è provvista la ridondanza on-chip per i componenti critici del microcontrollore tra cui:

- il core della CPU
- l'eDMA controller
- l'interrupt controller

- il sistema crossbar bus
- la memory protection unit
- il flash-memory controller e gli SRAM controllers
- il peripheral bus bridge
- i timer di sistema
- il watchdog timer

Inoltre codice ECC è disponibile sia per la memoria SRAM sia per la memoria flash.

I core presenti nel microcontrollore fanno parte della famiglia e200z4 Power Architecture. La dual-issue pipeline di questi dispositivi offre un'alta efficienza che consente alte prestazioni con una minima dissipazione di potenza.

Il microcontrollore è stato sviluppato con memoria flash ad alte prestazioni a 90nm che offre una sostanziale riduzione di costi per feauture e un significativo miglioramento delle prestazioni

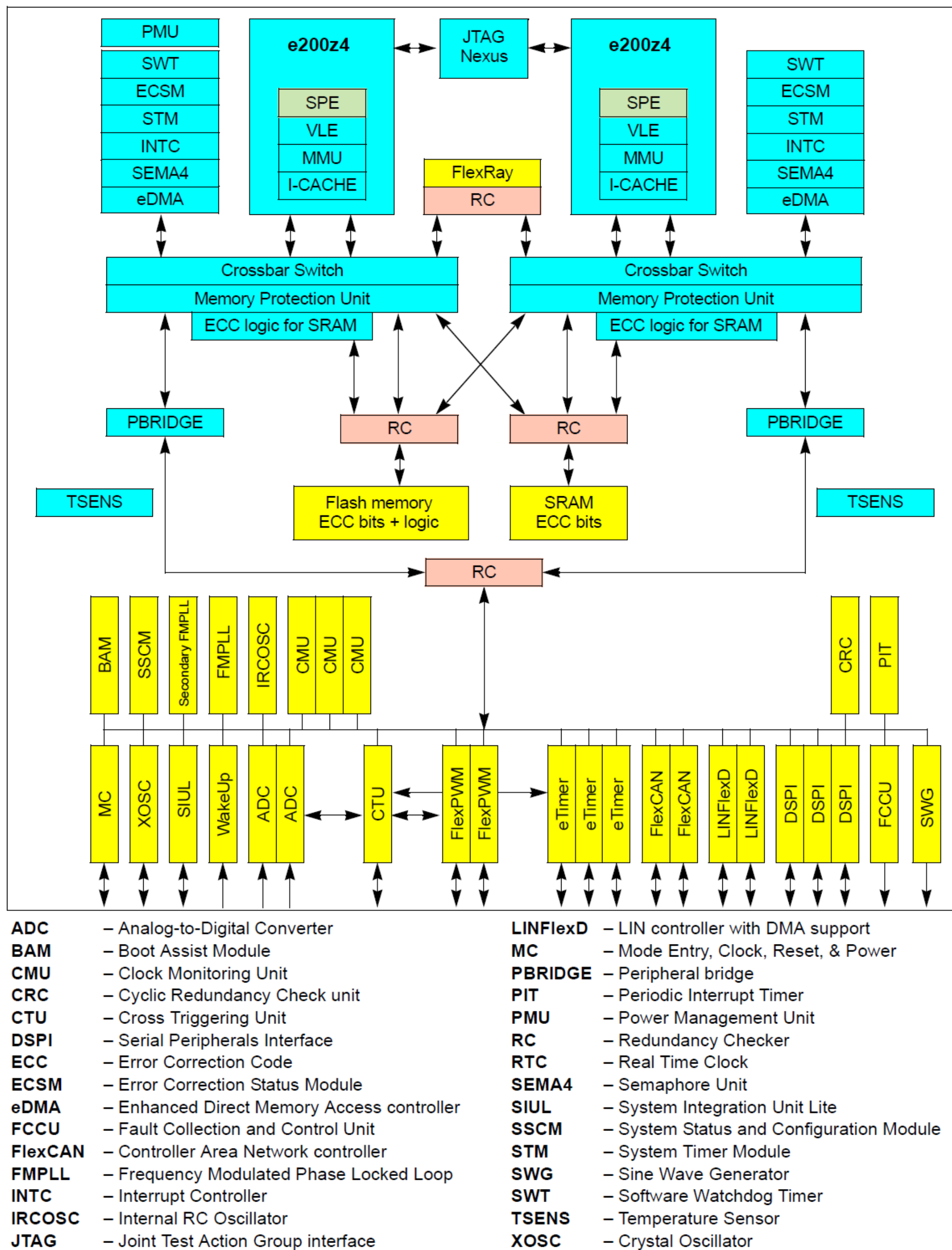


Figure 4.1: Struttura cpu

4.2 Descrizione core e200z4d

La famiglia di processori e200z4d è un insieme di CPU cores che implementano la versione a basso costo della tecnologia Power Architecture.
Il core integra:

- una pipeline 32-bit in-order dual-issue a cinque stadi con porte multiple sia in lettura che in scrittura;
- una Branch processing unit (BPU) con un branch target buffer (BTB) a 8 entry
- una unità adibita al fetch delle istruzioni
- un general-purpose register file a 64-bit
- una unità load/store
- una coppia di unità di esecuzioni intere
- una unità Embedded floating-point (FPU) che supporta operazioni scalari e vettoriali SIMD floating-point a singola precisione usando il register file a 64-bit
- due bus di sistema avanzati ad alte prestazioni (AHB) a 64-bit

Parlando invece delle principali caratteristiche delle unità di esecuzione, possiamo trovare:

- l'unità che gestisce le istruzioni presenta un path a 64-bit che permette di fare il fetch di due istruzioni power ISA a 32-bit o di quattro istruzioni VLE a 16-bit per ciclo di clock

- la maggior parte delle operazioni aritmetiche e logiche sono eseguite in un singolo ciclo di clock ad eccezione della divisione e della moltiplicazione

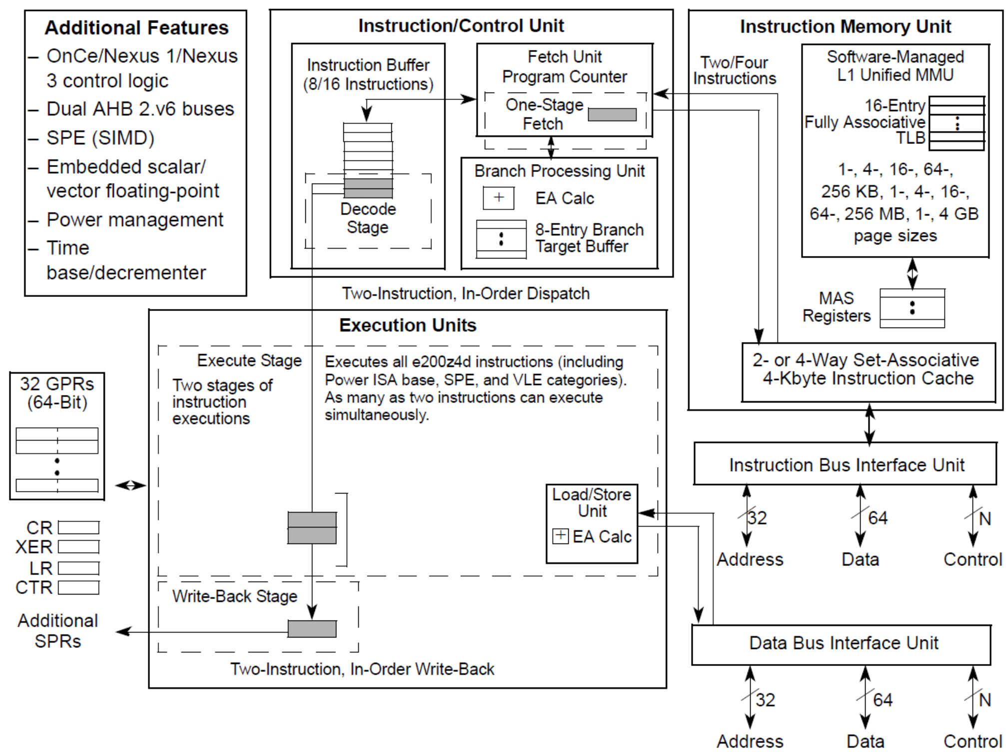


Figure 4.2: Diagramma a blocchi core

4.2.1 Set di istruzioni

Il core è conforme con la Power Architecture instruction set architecture (ISA), anche se le istruzioni Power Architecture ISA floating-point non sono implementate direttamente in hardware, ma vengono emulate attraverso il software. I set di istruzioni che il core mette a disposizione sono:

- istruzioni floating-point che forniscono supporto alle operazioni real-time single-precision usando i GPRs.

- istruzioni signal processing extension (SPE) che fornisono supporto alle operazioni SIMD (Single Instruction Multiple Data) fixed-point e single-precision usando sempre i GRPs. Per poter supportare questo categoria di istruzioni i GRPs sono stati estesi a 64 bit.

- oltre alle istruzioni Power Architecture ISA embedded di base il core implementa anche le istruzioni variable-length encoding (VLE) che garantiscono una maggiore densità di codice.

4.2.2 Modello dei registri

Il core integra la maggior parte dei registry definiti dall'architettura Power ISA ad eccezione dei registri floating-point FPR0–FPR31 e FPSCR. Infatti il core non supporta l'architettura hardware floating-point POWER ISA. In generale i registri possono essere divisi in due grandi categoria:

1. registri livello utente che sono accessibili da qualsiasi software sia con privilegi a livello utente sia con privileggi a livello supervisor. Tra i questi registri troviamo:

   - trentadue registri GPR (GPR0–GPR31) a 64-bit usati come sorgente o destinazioni per le varie operazioni aritmetiche. Le istruzioni Power ISA a 32 bit sfruttano solamente i 32 bit bassi dei GPR mentre le istruzioni SPE usano l'intero registro.
   - un Condition Register (CR) a 32-bit che è costituito da 8 cambi da 4-bit ciascuno (CR0-CR7). Questi vengono usati per salvare i risultati di alcune operazioni aritmetiche e forniscono un meccanismo per il testing e per il branching.

   I rimanenti registri a livello utente sono registri di tipo Special Purpose Register (SPR). Questi sono:

   - l'Integer exception Register (XER) che serve per tenere traccia del carry del e del overflow durante un'operazione aritmetica
   - il Link Register (LR) che fornisce l'indirizzo destinazione per istruzioni di salto condizionale

- il Count Register (CTR) che contiene un contatore utilizzato per i cicli che viene decrementato
- il Time base upper (TBU) e il time base lower (TBL), due timer a 32-bit che sono in modalità solo lettura
- quattro registri Special Purpose (SPRG4-SPRG7) accessibili in modalità solo lettura
- l'architettura Power ISA definische anche un registro speciale US-PRG0 accessibile sia in modalità lettura che scrittura

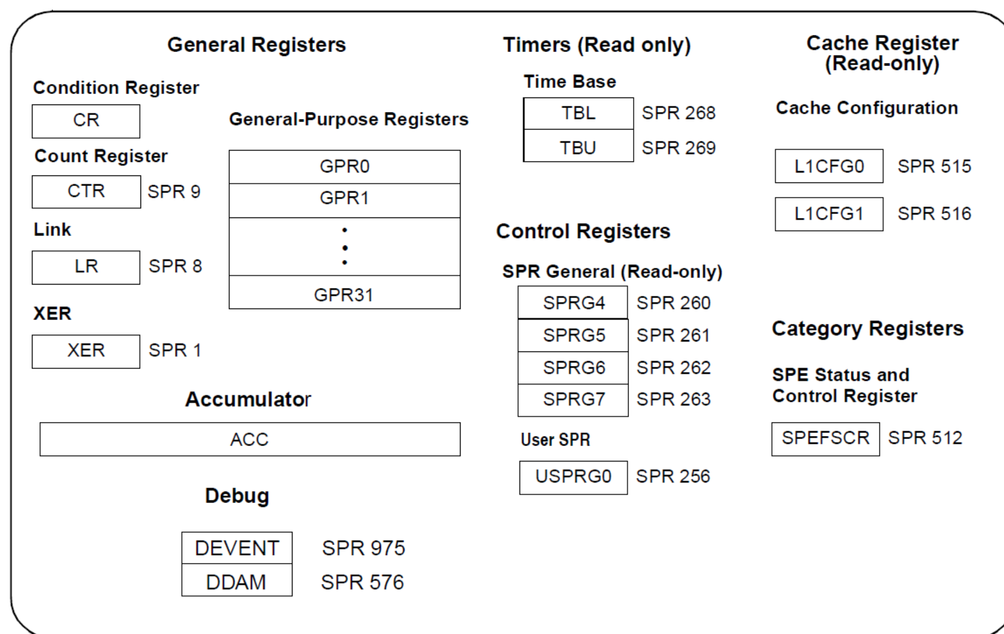


Figure 4.3: Registri livello utente

2. registri livello supervisor accessibili esclusivamente da codice di livello supervisor. Questi registri sono raggruppati in cinque categorie principali:

   - **Processor Control registers**

     Tra questi registri è presente il Machine state register (MSR) che contiene lo stato in cui si trova il processore. In caso di interrupt il contenuto di questo registro viene salvato in uno dei registri machine state save/restore (SRR1,CSRR1, DSRR1, MCSRR1).

   - **Storage Control register**

     Questo registro contiene l'indicazione del processo o del task attualmente in esecuzione.

   - **Interrupt Registers**

     Registri usati per la gestione delle procedure di interrupt.

   - **Debug facility registers**

     Registri usati per il debug.

   - **Timer Registers**

     Contengono registri come il Decrementer register (DEC) usato per scatenare un'eccezione programmata al termine di un delay preimpostato o il Time base register (TB) usato per salvare l'ora del giorno e per altre operazioni che necessitano dell'utilizzo di timers.

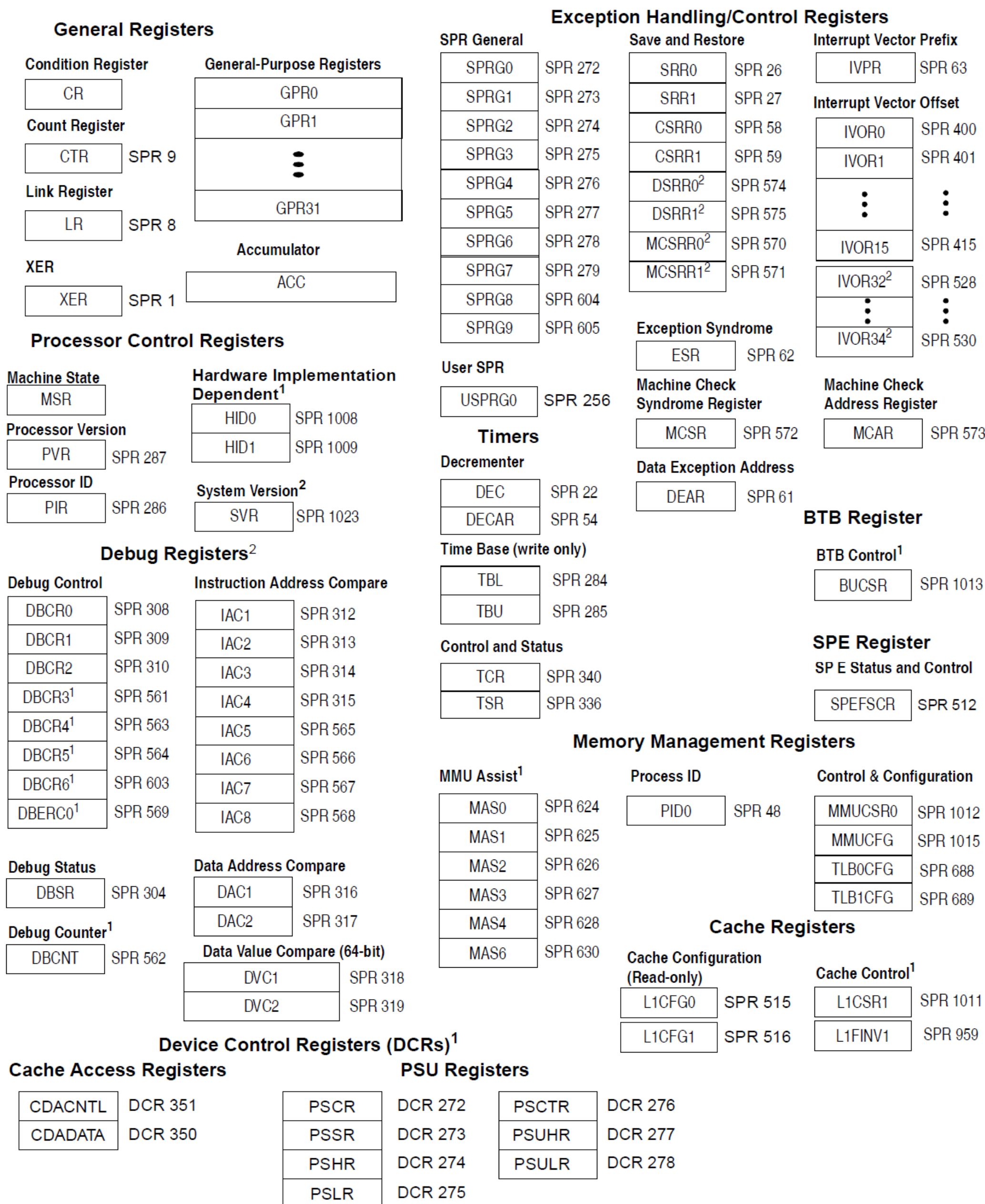


Figure 4.4: Registri livello supervisor

Chapter 5

Risultati sperimentali

In questo capitolo verranno illustrati i risultati sperimentati prodotti dai programmi di test a run-time e messi a confronto con i programmi di test eseguiti solamente al boot-time. Questo risultati si possono anche consultare in [6].

5.1 Risultati prodotti dalla suite di test intrusiva

Questi tipi di programmi di test sono eseguiti durante le fasi di accensione e spegnimento del sistema. In questo modo non si viene a creare nessun conflitto con il SO e i programmi di test possono eseguiti in supervisor mode rendendo possibile testare in maniera più approfondita tutte le unità del sistema. Una tra queste è il Register File (contiene numerosi registri speciali non accessibili in user mode) che con i suoi 240k guasti incide in maniera importante nel confronto dei risultati prodotti dai programmi intrusivi e

CPU Module	test programs	Duration [cc.]	Size [KB]
Arithmetic adder	3	988	1.1
Divider	4	4,168	1.4
Logic operation	11	5,625	2.7
Multiplier	2	960	0.9
Shifter	3	820	1.5
Exception	1	8,400	1.8
BTB	3	2,916	2.4
Register file	1	1,800	2.1
Fetch unit	1	30,900	1.5
Forward unit	1	3,400	4
Decoder unit	1	1,500	2.7
Load / Store unit	2	750	1.3
TOTAL	**33**	**62,227**	**23.4**

Table 5.1: Composizione della suite di test intrusiva

quelli non intrusivi.

Nella tabella 5.1 possiamo vedere tutti i programmi di test relativi alle varie unità insieme alla loro durata (misurata in tempi di clock) e al loro utilizzo di memoria. Se guardiamo la media del tempo di esecuzione di ogni singolo programma di test (escludendo quello della fetch unit che modifica in modo importante la media), questa risulta essere pari a 979 colpi di clock. Questo risultato ci fa capire che questi programmi sono lunghi se messi a confronto con quelli non intrusivi che hanno il vincolo di poter durare al massimo 255 colpi di clock.

Analizzando invece la tabella 5.2 si nota che il risultato prodotto in ter-

CPU Module	faults	FC Intrusive [%]
Arithmetic adder	10,268	88.39%
Divider	20,625	82.37%
Logic operation	19,266	83.83%
Multiplier	71,541	96.58%
Shifter	12,548	96.64%
Exception	13,695	37.36%
BTB	29,677	68.41%
Register file	264,022	91.59%
Address generator	28,419	61.37%
Fetch unit	58,244	62.73%
Forward unit	103,506	67.26%
Decoder unit	91,523	56.84%
Load / Store unit	12,420	89.71%
Control unit	50,442	56.93%
TOTAL	**797,562**	**75.31%**

Table 5.2: Risultati suite di test intrusiva

mini di fault coverage della suite di test intrusiva è stata del 75.31%.

5.2 Risultati prodotti dalla suite di test non intrusiva

Questi tipi di programmi di test vengono invece eseguiti a run-time, in parallelo al SO. Devono essere quindi eseguiti in user mode e devono soddisfare anche tutti i requisiti descritti nel Capitolo 3.

Come si può notare dalla tabella 5.3, il numero di programmi di test che contiene la suite di test non intrusiva è pari a 105. Questo numero è notevolmente superiore a quello della suite intrusiva, ma si può anche notare come la durata sia di molto inferiore (23.766cc contro 62.227cc). Questa è una diretta conseguenza del fatto che gran parte del codice presente nel programmi di test intrusivi scatena delle interruzioni e quindi non può essere replicato nei programmi non intrusivi. Un altro dato importante è quello riguardante la durata d'esecuzione totale dei programmi di test relativi al moltiplicatore e all'unità aritmetica. Nel caso dei test non intrusivi questo dato risulta essere superiore perché vi è un maggiore overhead nel salvare e ripristinare i vari registri usati.

Si può inoltre osservare come non si sono potuti scrivere programmi di test mirati per alcune unità come la Forward unit, la Fetch unit e l'Exception (questo è dovuto al fatto che scrivere programmi di test per queste unità comporta violare i vincoli).

La tabella 5.4 riporta inoltre i risultati finali ottenuti dalla suite non intrusiva. La fault coverage è di 67.12%, un risultato del 8.19% inferiore alla suite intrusiva. Questi valori (fault coverage test intrusivi e non intrusivi) sono stati calcolati eliminando circa 1.63% dei guasti totali etichettati come

CPU Module	**test programs**	**Duration [cc.]**	**Size [KB]**
Arithmetic adder	6	1,309	1.3
Divider	21	4,763	2.1
Logic operation	28	6,163	3.6
Multiplier	5	1,055	1.2
Shifter	4	714	1.7
Exception	0	0	0
BTB	28	6,975	8.5
Register file	4	927	1.7
Fetch unit	0	0	0
Forward unit	0	0	0
Decoder unit	5	970	1.4
Load / Store unit	4	890	1.5
TOTAL	**105**	**23,766**	**23**

Table 5.3: Composizione della suite di test non intrusiva

CPU Module	**faults**	**FC Non-intrusive [%]**
Arithmetic adder	10,268	87.50%
Divider	20,625	80.46%
Logic operation	19,266	82.23%
Multiplier	71,541	96.46%
Shifter	12,548	95.68%
Exception	13,695	7.97%
BTB	29,677	53.64%
Register file	264,022	68.22%
Address generator	28,419	43.56%
Fetch unit	58,244	59.26%
Forward unit	103,506	66.69%
Decoder unit	91,523	45.40%
Load / Store unit	12,420	89.53%
Control unit	50,442	55.81
TOTAL	**797,562**	**67.12%**

Table 5.4: Risultati suite di test non intrusiva

"untestable", secondo quanto riportato in [5]. La gran parte della differenza che c'è tra le due STL è dovuta al fatto che numerosi guasti marcati come "detected" dai test intrusivi non sono proprio eccitabili da quelli non intrusivi a causa dei vari vincoli. Dati le numerose limitazioni e la possibilità di testing a run-time, questo è da considerarsi un ottimo risultato anche in ottica di un successivo sviluppo.

Chapter 6

Conclusione

L'obiettivo primario di quest'opera è stato quello di presentare sia le difficoltà sia i risultati dello sviluppo di una STL non intrusiva operante in ambienti critici per la sicurezza. Tale approccio pur portando ad una fault coverage (67.12%) minore rispetto ad un approccio intrusivo (75.31%), ha permesso di testare il dispositivo durante tutte le sue fasi di funzionamento (accensione, spegnimento, run-time).

Tutto questo ha dato la possibilità di monitorare in tempo reale lo stato di salute del processore, aspetto che risulta di vitale importanza in un ambiente come quello automotive. Infatti i test contenenti nella STL possono essere eseguiti periodicamente a run-time dal SO a patto di soddisfare alcuni vincoli (tempo massimo di esecuzione, nessuna interruzione o eccezione, nessuna alterazione della memoria RAM, nessuna alterazione dei SPR, possibilità di interrompere il test etc...). I vincoli prima elencati non hanno però permesso di aumentare ulteriormente la fault coverage globale in quanto alcuni moduli, in particolare il Register File che con i suoi 264,022 su 797,562 guasti totali

costituisce il 33.1% di tutto il chip, non sono ulteriormente testabili.

La principale differenza tra i risultati prodotti dalla suite di test intrusiva e quella non intrusiva sta proprio nell'impossibilità di testare in profondità tutti i moduli del device. Prendendo sempre come esempio il Register File (che presenta molti moduli non accessibili a run-time) si vede come la suite intrusiva produce una copertura sul modulo pari a 91.59%, mentre la suite non intrusiva produce un risultato pari a 68.22%.

L'approccio seguito è stato quindi quello di sacrificare circa l'8.19% di copertura totale di guasti a favore della possibilità di un monitoraggio continuo che è sicuramente un elemento di fondamentale importanza in ambienti critici per la sicurezza.

In futuro si potrebbe rendere più consapevole il SO dei test non intrusivi. Facendo ciò si avrà la possibilità di rendere i vincoli meno stringenti, potendo testare le unità più in profondità e aumentando la fault coverage.

Bibliografia

[1] D. Changdao, M. Graziano, E. Sanchez, M. S. Reorda, M. Zamboni, and N. Zhifan, "On the functional test of the btb logic in pipelined and superscalar processors," *14th Latin American Test Workshop - LATW*, 2013.

[2] P. Bernardi, R. Cantoro, L. Ciganda, E. Sanchez, M. S. Reorda, S. D. Luca, R. Meregalli, and A. Sansonetti, "On the in-field functional testing of decode units in pipelined risc processors," *IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, 2014.

[3] P. Bernardi, R. Cantoro, S. D. Luca, E. Sánchez, and A. Sansonetti, "Development flow for on-line core self-test of automotive microcontrollers," *EEE Transactions on Computers*, 2016.

[4] P. Bernardi, R. Cantoro, S. D. Luca, E. Sanchez, A. Sansonetti, and G. Squillero, "Software-based self-test techniques for dual-issue embedded processors," *IEEE Transactions on Emerging Topics in Computing*, 2017.

[5] P. Bernardi, M. Bonazza, E. Sanchez, M. S. Reorda, and O. Ballan, "On-line functionally untestable fault identification in embedded processor cores," *2013 Design, Automation Test in Europe Conference Exhibition (DATE), Grenoble, France*, 2019.

[6] P. Bernardi, R. Cantoro, A. Floridia, D. Piumatti, C. Pogonea, A. Ruospo, E. Sanchez, S. D. Luca, and A. Sansonetti, "Non-intrusive self-test library for automotive critical applications: Constraints and solutions," *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2019.

[7] P.Bernardi, D. Piumatti, and E. Sanchez, "Facilitating fault-simulation comprehension through a fault-lists analysis tool," *10th IEEE Latin American Symposium on Circuits & Systems (LASCAS)*, 2019.