Università degli Studi di Padova

Facoltà di Ingegneria

Corso di Laurea Magistrale in Ingeneria Elettronica

Tesi di laurea

Una suite di test per valutare gli effetti della radiazione cosmica su microprocessore PowerPC 7448

Relatore: Prof. Alessandro Paccagnella

Correlatore: Ing. Simone Gerardin

Laureando: Martino Facchin

ver.α – 4 dicembre 2011

Autore: Martino Facchin

Indice

Capitolo 1

Introduzione

1.1 Prospettive del telecontrollo

Fin dagli albori della conquista spaziale i satelliti hanno svolto un ruolo importantissimo nel migliorare le condizioni di vita sulla Terra. Il mondo delle telecomunicazioni si basa ormai estensivamente sull'uso di satelliti, mentre altri settori stanno cominciando ad interessarsi alle potenzialità dello spazio considerando i possibili vantaggi derivanti.

Il settore del telerilevamento si è già avvantaggiato in maniera estensiva della tecnologia satellitare, ma per affrontare le sfide future è necessaria un'innovazione a livello di componenti e di paradigmi.

É stato infatti pronosticato che, nei prossimi decenni, le caratteristiche principali dei sistemi di telecontrollo satellitari dovranno essere l'indipendenza e la capacità di usare poca banda per le trasmissioni. Al giorno d'oggi infatti sono proprio questi i fattori che impediscono di disporre dell'informazione acquisita in modalità realtime o quasi realtime.

I canali di comunicazione esistenti saranno sempre più inadeguati all'aumentare della quantità di dati da scaricare a Terra, fino a rendere il processo praticamente inutile (le operazioni di download e successiva elaborazione potrebbero variare da alcune ore a diverse decine di ore). Si rende quindi necessario uno sviluppo sostanziale delle capacità elaborative e "decisionali" del satellite stesso.

Per questo motivo la direzione intrapresa è quella di montare direttamente sul satellite dei microprocessori avanzati, in grado di svolgere una miriade di operazioni che vanno dalla compressione di immagini e video all'elaborazione dei dati

grezzi acquisiti in modo da poter agire autonomamente (ad esempio eseguendo uno zoom sull'area di interesse prima che sia la base a terra a comunicarlo).

1.2 Proposte Tecnologiche

Un progetto recente si inserisce in questo contesto, proponendo sia un'architettura di elaborazione di bordo, modulare e scalabile, che la relativa componentistica per i radar di prossima generazione in grado di soddisfare le esigenze delle nuove applicazioni di telerilevamento. L'obiettivo di questa proposta è quello di fornire le direttrici di sviluppo tecnologico affinchè siano supportate le applicazioni future con un orizzonte temporale di 10 - 15 anni.

1.2.1 Enti interessati

Fanno parte del progetto diverse aziende ed università, ognuna impegnata a titolo diverso (progettazione elettronica, software e testing) nella realizzazione della piattaforma di prova.

L'Università di Padova, grazie alle competenze del gruppo di ricerca RREACT, si occupa della parte di progetto relativa alla caratterizzazione degli effetti della radiazione ionizzante sull'elettronica che sarà montata a bordo del satellite, in particolar modo sul microprocessore.

1.3 Obiettivo della tesi

Questa tesi tratterà la realizzazione dei software di test necessari al fine di eseguire le sessioni di irraggiamento in maniera semplice e allo stesso tempo completa, spaziando da prove statiche a test dinamici.

Per problemi di tempistiche la tesi non contiene i risultati degli irraggiamenti, per cui saranno stimati i risultati previsti in base alla letteratura disponibile.

Capitolo 2

Le Radiazioni Ionizzanti

Introduzione

Il seguente capitolo tratta l'origine e la natura delle radiazioni ionizzanti, i loro effetti sull'elettronica attuale, i metodi per migliorare l'affidabilità dei dispositivi e le metodologie per eseguire test specifici su di essi. L'obiettivo è ovviamente mostrare la natura del "nemico" che la scheda si troverà ad affrontare una volta in orbita, per cui saranno analizzate in dettaglio le radiazioni spaziali e i loro effetti sull'elettronica digitale realizzata con processi produttivi decisamente scalati.

2.1 Origine e caratteristiche della radiazione

Per *radiazioni ionizzanti* si intendono tutte le particelle (o fotoni) il cui passaggio nella materia provoca la ionizzazione della stessa (ovvero la creazione di coppie elettrone-lacuna nel materiale). Le radiazioni hanno origine sia in ambiente spaziale che terrestre e si dividono principalmente in due categorie: direttamente ionizzanti (ioni pesanti, protoni, elettroni, raggi γ e raggi X) o indirettamente ionizzanti (neutroni). Queste particelle devono essere estremamente energetiche per poter ionizzare altri atomi, per cui le principali fonti terrestri sono i contaminanti radioattivi nei chip ed i neutroni atmosferici, mentre quelle spaziali possono avere nature molto differenti.
Possiamo trovare:

- Particelle intrappolate nelle fasce di Van Allen
- Particelle provenienti dal Sole

- Raggi Cosmici

Le particelle presenti nelle fasce di Van Allen (una cintura naturale creata dalla presenza del campo magnetico terreste - figura 2.1) sono prevalentemente protoni ed elettroni, la cui presenza è dovuta principalmente all'intrappolamento del vento solare. L'atmosfera limita l'estensione di queste fasce tra 500 e 30000 km. Dal momento che la concentrazione di particelle potenzialmente nocive è elevatissima (si possono incontrare protoni con energia pari a 400MeV, capaci di penetrare 143mm di piombo [1]) è altamente sconsigliato il posizionamento di satelliti in questa zona.

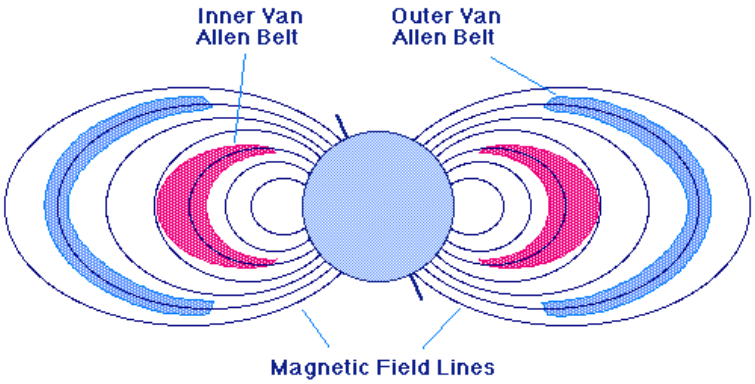


Figura 2.1: Posizione delle Fasce di Van Allen

Il fenomeno delle tempeste solari è particolarmente noto per portare malfunzionamenti all'elettronica in orbita come a quella a Terra, come ad esempio nel 2003 in occasione del più grande *solar flare* mai registrato. Anche i raggi cosmici sono insidiosi in quanto generati da eventi catastrofici che avvengono nello spazio esterno (come ad esempio l'esplosione di supernovae). Pur essendo questi eventi imprevedibili il flusso finale è costante; rappresentano un problema soprattutto per i satelliti geostazionari (posizionati a circa 36000 km), dal momento che a contatto con l'atmosfera le particelle cosmiche primarie interagiscono con altri atomi dando origine ad una cascata di particelle secondarie individualmente meno energetiche.

2.1.1 Unità di misura

Per poter valutare l'entità della radiazione sono state adottati dei termini e delle grandezze specifiche che tengono contro dell'energia depositata e della massa delle

particelle in funzione dell'area attraversata o del tempo trascorso. le principali unità di misura sono:

- Flusso (ϕ): numero di particelle per unità di area nell'unità di tempo
  Unità di misura: $Particelle/(cm^2 \cdot s)$
- Fluenza (Φ): numero di particelle per unità di area (integrale nel tempo del flusso)
  Unità di misura: $Particelle/cm^2$
- Dose (D): energia della radiazione per unità di massa
  Unità di misura: $J/Kg = Gray$ o $erg/g = rad = 0.01Gray$

La radiazione può provocare ionizzazione, displacement degli atomi e agitazione termica. Nel caso sia completamente ionizzante, ovvero tutta l'energia depositata porti esclusivamente alla creazione di coppie elettrone-lacuna, la Dose si definisce:

- Total Ionizing Dose (TID): energia ionizzante/massa
  Definizione: $LET \cdot \Phi$

L'appena citata LET (Linear Energy Transfer) rappresenta la quantità totale di energia ionizzante ceduta per unità di spazio (MeV/cm), normalizzata rispetto alla densità del materiale colpito (mg/cm^3). La sua unità di misura è quindi $MeV \cdot cm^2/mg$. In funzione della LET si possono confrontare fonti radiative con energie e spessori di penetrazione differenti, rendendo possibile l'esecuzione dei test in centri di ricerca dotati di diversi metodi di generazione del fascio.

Ultimo parametro ma non per importanza è la

- Sezione d'urto (cross section, σ)
  Definizione: numero di eventi di interesse/fluenza
  Unità di misura: cm^2

Gli eventi di interesse sono solitamente bitflips [1] o altri tipi di SEEs (Single Event Effects, letteralmente Effetti da evento singolo). Il plot della cross section rilevata a differenti valori di LET aiuta ad esprimere graficamente la suscettività del dispositivo, a capirne la massima LET sopportabile senza errori (LET_th) ed il valore a cui la cross section satura a fronte di un aumento della LET (σ_sat) (Figura 2.2).

[1] bitflip: cambiamento dello stato logico di una cella di memoria dovuto a cause esterne

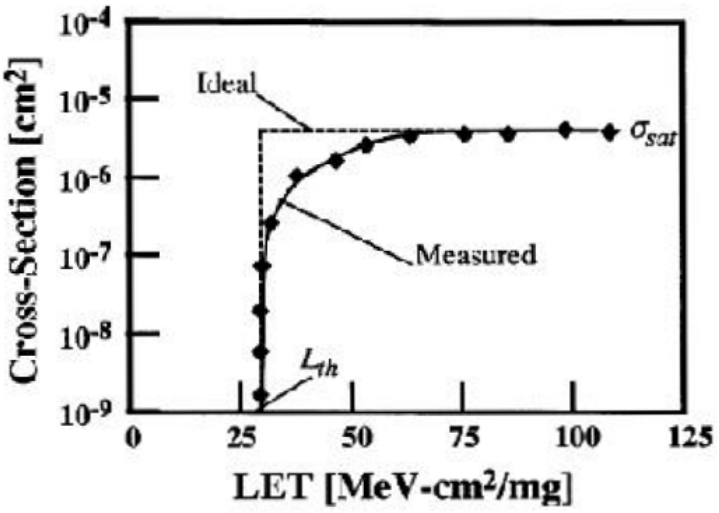


Figura 2.2: Esempio di $\sigma(LET)$ - si possono notare LET_{th} e σ_{sat}

2.2 Guasti provocati dalle radiazioni

Il progresso delle potenzialità computazionali dell'elettronica di oggigiorno è dovuto in massima parte alla realizzazione dei dispositivi con tecnologie sempre più scalate, ed ha come dirette conseguenze l'aumento della densità di transistor (e quindi di funzioni logiche realizzabili nella stessa area di silicio) e la riduzione del loro consumo energetico (grazie all'effetto secondario di abbassamento della tensione di alimentazione). Di contro la miniaturizzazione estrema ha portato a galla dei problemi prima sconosciuti, dal manifestarsi di effetti quantistici fino alla comparsa di perturbazioni del funzionamento dovute a fenomeni ambientali prima ininfluenti.

I fenomeni principali sono quelli dovuti alla lunga esposizione alla radiazione e quelli transitori, generati dall'impatto di una singola particella dotata di carica contro un'area sensibile del transistor.

Tra i primi è necessario citare i cosiddetti effetti di tipo TID (Total Ionizing Dose), che riguardano l'intrappolamento di lacune nell'ossido di gate che alla lunga portano alla creazione di difetti all'interfaccia, provocando la modifica del campo elettrico e quindi delle soglie di accensione/spegnimento del transistor. A causa dello scaling dei processi produttivi questo effetto si sta spostando dall'ossido di gate (la diminuizione dello spessore dell'ossido riduce la probabilità di

intrappolare una lacuna) agli isolamenti laterali.

2.2.1 Single Event Effects

Tutti gli effetti transitori vengono invece chiamati Single event effects (SEEs) per descrivere l'avvenuta interazione tra una singola particella ed il dispositivo. Le particelle elettricamente non neutre possono provocare infatti spostamenti di carica che perturbano momentaneamente il funzionamento del dispositivo, e nei casi siano ad alta LET gli effetti possono essere macroscopici.
Nella suddivisone standard si parla di:

- Soft errors: Fenomeni non distruttivi ma che causano perdite d'informazione non trascurabili
  - Single Event Upset (SEU)
  - Multiple Bit Upset (MBU)
  - Single Event Transient (SET)
  - Single Event Functional Interrupt (SEFI)
  - Single Event Latch-up (SEL) (potenzialmente distruttivo)
- Hard errors: Fenomeni distruttivi che arrecano danni permanenti alla struttura
  - Single Event Gate Rupture (SEGR)
  - Single Event Burnout (SEB)
  - Stuck Bits

SEUs

Affinché il bit rappresentato dal valore logico "alto" o "basso" in una cella di memoria commuti in seguito all'interazione tra la radiazione ed il device under test è necessario che la collisione avvenga in una determinata area sensibile del dispositivo. Ad esempio, considerando la cella di memoria SRAM in figura 2.3 un punto critico è la giunzione drain-bulk. Se la particella incidente genera portatori liberi in questa zona è possibile che l'impulso di corrente transitorio abbassi la tensione che tiene fisso il bistabile, portando alla commutazione anche l'altro nodo e quindi invertendo il valore logico precedente. Solo se la carica

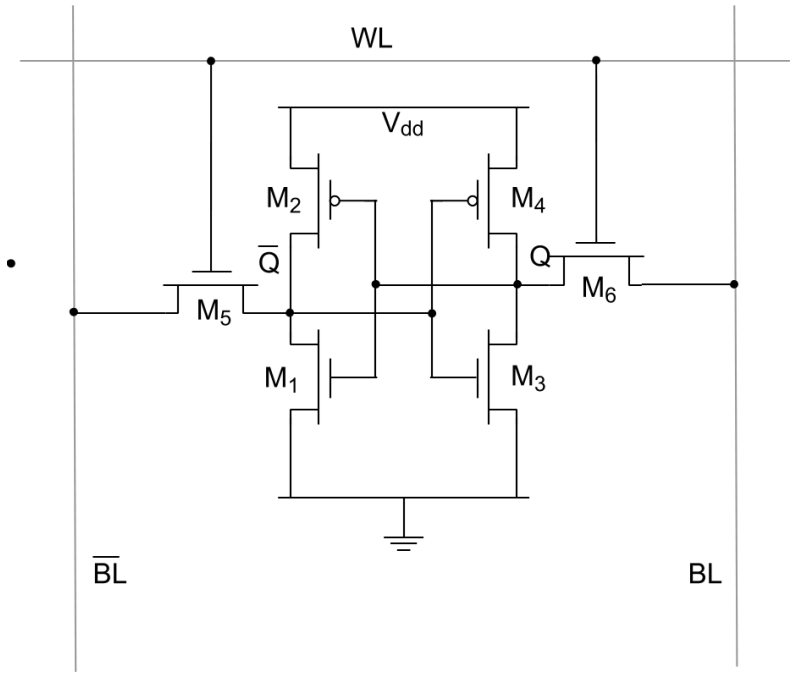


Figura 2.3: Cella SRAM a 6 transistor

raccolta è superiore ad una certa soglia viene innescata la commutazione, per cui la particella deve possedere una LET sufficientemente alta.

Quello considerato è un esempio limite dal momento che i circuiti sono formati da una miriade di nodi interconnessi, che influenzano la capacità totale del nodo "vittima". Quindi la quantità di carica critica dipende da una serie di parametri circuitali quali la geometria e la tensione di alimentazione. Dal momento che un effetto secondario dello scaling è la riduzione della tensione di alimentazione si può ipotizzare che dispositivi più scalati siano anche più sensibili ai SEU in quanto diminuisce la carica critica. In realtà la sensibilità è condizionata da molti altri fattori, per cui in molti dispositivi recenti questa correlazione si nota molto meno che in passato.

Un'osservazione molto importante può essere compiuta sulla frequenza dei SEU: infatti in molti dei dispositivi testati (processori, memorie, FPGA e circuiti elementari) la quantità di commutazioni 0→1 è sensibilmente diversa dalla quantità di 1→0. Questa asimmetria è dovuta alla geometria e alla porte logiche costituenti del circuito. Considerando infatti le configurazioni di strutture elementari quali uno shift register (i cui flip flop sono formati da coppie NOR-NOT)

ci accorgiamo che le due commutazioni non sono equiprobabili ma manifestano una netta predisposizione ad una delle due transizioni a causa della differenza di capacità tra i suoi nodi. Questo non è vero in circuiti simmetrici nei quali si verifica pedissequamente l'equiprobabilità dei due eventi.

2.2.2 SEFI

Nei processori più avanzati si verifica con una casistica "interessante" anche il fenomeno dei SEFI (Single Effect Functional Interrupt). Se vengono colpite aree del dispositivo deputate al controllo del flusso di esecuzione può accadere che questo si venga a modificare irreparabilmente. Un esempio tipico può essere un bitflip nel Link register (che contiene l'indirizzo dell'istruzione da eseguire quando termina quella in corso) o nel Program counter (che controlla il flusso del processo). La soluzione più semplice è il reset del dispositivo; sfortunatamente questo comporta la perdita dei risultati della routine in corso e l'incongruenza dei dati elaborati (non si può sapere se è occorso un SEFI finché non si vengono a generare situazioni inconsistenti).

La quantità di SEFI (sotto forma di hangs) aumenta in maniera sostanziale col diminuire della tensione di alimentazione (figura 2.4 da [2])

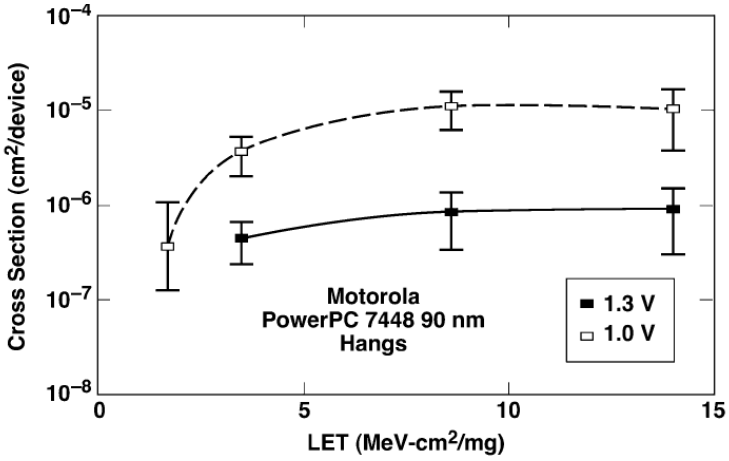


Figura 2.4: Confronto tra le cross sections relative agli hangs nel processore PowerPC 7448 alimentato a 1.3V e a 1V

2.2.3 SEL

Si definisce Single Event Latch-up l'accensione, a causa dell'interazione con una particella, dell'SCR parassita presente nella maggior parte dei circuiti digitali, soprattutto in quelli realizzati in tecnologia MOS. In una coppia pmos-nmos sono presenti due SCR parassiti (normalmente polarizzati in inversa). Se però uno di essi dovesse attivarsi la struttura parassita creerebbe un feedback positivo capace di accendere anche l'altro e di aumentare il flusso di corrente in maniera indiscriminata. Se non vengono prese contromisure (quali lo spegnimento del dispositivo ed il conseguente ripristino dell'operatività) il fenomeno può portare in breve tempo alla fusione del dispositivo (burnout).

2.3 Radiation Hardening

Per combattere i fenomeni presentati sono state sviluppate innumerevoli contromisure, sia fisiche che logiche. Queste tecnologie, chiamate Radiation hardening, sono state sviluppate nell'ottica dell'utilizzo spaziale, dal momento che rappresenta l'ambiente più pericoloso nel quale si possano subire danni (i dispositivi operano in maniera indipendente e le possibilità di riparazione sono quasi inesistenti). Le più utilizzate sono le seguenti:

2.3.1 Tecnologia SOI

Con l'acronimo SOI (Silicon on insulator) si intende il processo produttivo per circuiti CMOS nel quale il dispositivo viene realizzato su un sottile strato di silicio sovrastante uno strato di ossido ($Si0_2$) invece che direttamente sul substrato bulk; questa tecnologia, ad esclusione degli svantaggi dovuti al costo e alla presenza del body flottante, comporta un grande miglioramento dell'immunità alla radiazione. Infatti il volume soggetto all'insorgenza di SEEs in caso di impatto diminuisce notevolmente (conseguentemente alla riduzione dello spessore dell'area attiva). Anche l'eventualità che avvenga un latch-up viene diminuita di vari ordini di grandezza grazie all'isolamento tra p-well ed n-well. Non migliora invece la TID in quanto aumenta il volume di ossido per dispositivo. Anche la tecnologia SOS (Silicon on sapphire) viene utilizzata spesso in ambito rad-hard.

2.3.2 Utilizzo di memorie differenti

Le DRAM (dynamic RAM) recenti si sono dimostrate essere molto suscettibili ai SEFI in virtù della grande complessità ed alta integrazione dei loro circuiti di controllo. Sono in ogni caso preferite alle SRAM in ambienti ostili in quanto meno sensibili ai SEE. Sono anche in sviluppo nuove tecnologie come l'MRAM (Magnetoresistive RAM) che per loro natura sono totalmente immuni alla radiazione ionizzante.

2.3.3 Ridondanza

Il metodo più semplice per liberarsi dei bitflip è ovviamente quello di eseguire più di una volta gli stessi calcoli. Questo può avvenire su un unico device (al costo della perdita di velocità di elaborazione) o su più unità replicate che lo eseguono contemporaneamente (con ridotta perdita di velocità ma con costi estremamente maggiori). La ridondanza più comune è la tripla, con tre unità (solitamente microprocessori) che elaborano gli stessi dati, li confrontano e scelgono quello definitivo in base al principio del majority voting. La ridondanza tripla può venire in grande aiuto quando una delle unità dovesse smettere di funzionare: in quel caso le altre due (essendo identiche ed indipendenti) potrebbero continuare la loro attività in maniera trasparente.

2.3.4 Algoritmi di correzione d'errore

Il mondo delle telecomunicazioni insegna che gli errori possono anche essere corretti via software se si dispone di abbastanza informazione sul dato di partenza. Esistono svariati algoritmi che permettono il recupero di un segnale digitale danneggiato (o almeno la conoscenza della sua fallacia). Il più semplice di questi è il

- Controllo di parità

Il suo funzionamento è estremamente semplice: appena il pacchetto di dati viene generato, si contano gli 1 presenti. Se il numero è pari il bit di parità viene messo a 0, altrimenti a 1. Alla ricezione il pacchetto può arrivare intatto o danneggiato. Per scoprirlo basta contare il numero di 1 presenti, che dev'essere pari (infatti se il dato vero e proprio aveva un numero dispari di 1, il bit di parità posto a 1 rende il numero di 1 totali pari). Se invece è avvenuto un bitflip singolo (sulla parola o sul bit) il numero di 1 sarà dispari, per cui si scarta il dato (o se ne chiede il

rinvio). Il codice di parità non ci permette di sapere quale sia il bit danneggiato, per cui la sua utilità si ferma al riconoscimento del dato errato, senza permetterne il recupero. Oltretutto nel caso avvengano due o più bitflip sulla stessa parola la probabilità che il controllo di parità segnali il guasto è del 50%; d'altronde queste casistiche sono estremamente più rare rispetto agli errori singoli. D'altra parte il costo in termini di area occupata e complessità di un circuito che calcola il bit di parità è ridicolo per cui le memorie veloci, nelle quali sarebbe troppo oneroso utilizzare algoritmi più complicati, utilizzano spesso questo meccanismo per garantire la coerenza dei dati.
Il problema della correzione dell'errore occorso è stato affrontato in tempi abbastanza remoti dell'era dell'Informazione; il primo ad occuparsene fu Richard Hamming, che negli anni 50 implementò il primo algoritmo di correzione d'errore (ECC) noto come

- Codice Hamming (7-4)

Questo codice fu creato con l'obiettivo di correggere gli errori che affliggevano le trasmissioni minimizzando il rapporto risorse computazionali/tempo sprecato. L'idea di fondo è che un canale deve essere veramente disturbato affinchè cambino 2 bit, per cui è sufficiente riconoscere queste rare eventualità, concentrandosi sulla correzione dei singoli bitflip. Nella sua accezione (12,8) il codice si compone di 8 bit di dato e 4 di parità. La creazione del dato da inviare avviene in questo modo (nel caso di parità pari): [3]

1. Si decide il numero di bit di parità (p) come $log_2 lunghezzaword + 1$ (nel nostro caso $8 = 2^3$ quindi p=4)

2. Si inseriscono i bit di parità (al momento posti a zero) nelle posizioni 2^0 , 2^1, 2^2, 2^3

3. Si completano i rimanenti spazi coi bit della word di partenza (nell'esempio 10101101)

0	0	1	0	0	1	0	0	1	1	0	1	Code
1	2	3	4	5	6	7	8	9	10	11	12	digit
p	p		p				p					parity

Tabella 2.1: situazione iniziale: tutti i bit di parità posti a 0

4. Per decidere il valore il bit da parità 1 partiamo dalla posizione 1 e sommiamo tutti i bit dispari; se la somma è un numero pari il valore della parità è zero.

5. Per decidere il valore dei bit 2 si procede con la stessa modalità "a salti" considerando però due bit alla volta; nel nostro caso sommiamo i valori dei bit 2 e 3, 6 e 7, 10 e 11 e ne calcoliamo la parità scrivendola nel bit 2.

0	**1**	1	0	0	1	0	0	1	1	0	1	Code
1	2	3	4	5	6	7	8	9	10	11	12	digit
p	p		p				p					parity

Tabella 2.2: scrittura dei primi 2 bit di parità

6. Allo stesso modo si procede con gli altri due bit di parità, rispettivamente usando blocchi di 4 e 8 bit e partendo sempre dal bit stesso (nel primo caso si sommano i bit 4,5,6,7,12, nel secondo i bit 8,9,10,11,12)

0	**1**	1	**0**	0	1	0	**1**	1	1	0	1	Code
1	2	3	4	5	6	7	8	9	10	11	12	digit
p	p		p				p					parity

Tabella 2.3: parola completa

Una volta ricevuta la parola finale bisogna assicurarsi della sua correttezza: il procedimento è lo stesso della codifica, ovvero si ricalcolano i bit di parità sulla parola ricevuta. Si esegue poi lo xor tra i bit ricevuti e quelli ricalcolati. A questo punto si hanno 3 casistiche:

- Non sono avvenuti bitflip: in questo caso i bit di parità sono gli stessi e lo xor dà come risultato 0000

- É avvenuto un bitflip: in questo caso il risultato dello xor è diverso da 0. Per correggere il bit modificato si sommano le posizioni dei bit di parità che hanno cambiato valore; il risultato è la posizione del bit errato. Ad esempio spedendo la word realizzata precedentemente (011001011101):

  - Il dato arriva come 011001011100: i bit di parità in posizione 4 e 8 cambieranno di valore; il bit da correggere è il 12=4+8

  - Il dato arriva come 001001011101: in questo caso il bitflip è sul bit di parità; sarà quindi lui l'unico a cambiare in quanto 2+0=2 (il dato è esatto)

- Se invece avvengono bitflip multipli si possono scoprire ma non correggere: infatti se il dato viene ricevuto come 010001011100 lo xor darà come risultato 1111; si riesce quindi a determinare che sono avvenuti bitflip multipli ma non la loro posizione (infatti 1+2+4+8=15, che può essere interpretato in maniera non univoca come somma di due valori - nel modo corretto 12+3 come nell'errato 10+5)

I codici di correzione d'errore possono essere inseriti nei circuiti di controllo delle memorie ed il check può venire eseguito ad intervalli regolari (leggendo i dati, confrontando l'ECC ed eventualmente correggendo i dati fallaci). Il meccanismo, come abbiamo appena visto per una parola di 8bit (nelle memorie sono solitamente 32 o 64), è molto più complicato del semplice refresh delle memoria DRAM per cui viene inserito solo in sistemi fault-tolerant nei quali la velocità non è un fattore dominante.

2.4 Strutture di test

Per condurre i test sulla Terra sono necessari impianti di accelerazione di particelle (ciclotroni) coi quali è possibile creare e controllare il fascio che poi inciderà sul device. La composizione del fascio e le energie in gioco sono fortemente dipendenti dallo scopo principale per il quale è adibita la facility.

Ad esempio il Laboratorio nazionale di Legnaro, nel quale il gruppo di ricerca RREACT esegue la maggior parte dei test, ha a disposizione 4 acceleratori (AN2000, CN, TANDEM-XTU e ALPI) che permettono di spaziare un ampio range di energie e tipi di fascio (principalmente neutroni, protoni e ioni pesanti)

Capitolo 3

Analisi Preliminari

3.1 Scelte Hardware

Come analizzato nell'introduzione, la disponibilità di un modulo di elaborazione dati ad elevate prestazioni rappresenta il punto chiave nello sviluppo dei sistemi radar di nuova generazione. Tale processore deve rispondere a requisiti di capacità computazionale e di flessibilità tali da poter essere utilizzato per diverse applicazioni all'interno del sistema.

Per questi motivi è stato selezionato un microprocessore ad elevate prestazioni di tipo COTS (Commercial Off The Shelf) piuttosto che su un chip custom. In particolare il processore scelto è un PowerPC di generazione G4, quali il 7447A o il 7448 prodotti da Freescale.

La scelta non è casuale, ma si basa sui risultati di precedenti studi e tiene conto dei seguenti fattori:

1. La flessibilità di un processore General Purpose dalle elevate prestazioni:
   (a) Consumo in Full Power Mode
      - Massimo: 28.4 W @ 1.6GHz (Tj = 105°C)
      - Tipico: 20.0 W @ 1.6GHz (Tj = 65°C)
   (b) 3000 Dhrystone e 2.1 MIPS @ 1.3 GHz
   (c) Architettura superscalare
   (d) Caches on chip di grandi dimensioni (L1 da 32kB instruction e 32kB data, L2 da 512kB nel 7447A e 1Mb nel 7448)

   (e) Unità vettoriali intere e floating point a doppia precisione (basate sul set di istruzioni AltiVec)

2. L'ampia diffusione nel settore embedded che garantisce l'affidabilità del prodotto e la disponibilità di tools di sviluppo consolidati

3. Le caratteristiche intrinseche che rendono questi processori adatti ad applicazioni spaziali, quali:

   (a) L'intrinseca resistenza all'accumulo di cariche data dalla ridotta sezione di canale (130nm per il 7447A, 90nm per il 7448)

   (b) L'uso di tecnologia SOI (Silicon On Insulator), per sua natura resistente alla radiazione

   (c) I check di parità sul bus di sistema e sulla cache

   (d) La presenza della tecnologia ECC (Error Correction Code) sui dati dell'L2 (solo nel 7448)

4. I risultati positivi (estremamente pochi SEFI, errori di calcolo ottenuti con un software campione nell'ordine dei $5 \cdot 10^{-3}$ device/day) dimostrati dai test precedentemente effettuati. Risultati ugualmente confortanti si trovano anche in letteratura ([2] e [4])

3.2 L'architettura PowerPC

3.2.1 Storia

PowerPC (acronimo di Performance Optimization With Enhanced RISC - Performance Computing, a volte abbreviato in PPC) è un'architettura di microprocessori RISC creata nel 1991 dall'alleanza Apple-IBM-Motorola. Inizialmente progettata per il mercato consumer, ha avuto un notevole successo nei mercati embedded ed high performance. L'architettura PowerPC è in larga parte basata sulla precedente POWER, creata da IBM, con la quale mantiene elevati livelli di compatibilità, tanto da rendere molto spesso possibile l'esecuzione di programmi compilati per una piattaforma anche sull'altra.

La generazione G4 dei processori PowerPC è stata sviluppata da Motorola in stretta collaborazione con Apple alla fine degli anni 90. La decisione di includere

il set di istruzioni vettoriali AltiVec fu una scelta contrastata, al punto da allontanare IBM dal progetto, ma portò la generazione G4 a diventare estremamente popolare nei mercati embedded, media processing ed avionico.

Dopo la scissione della divisione hardware di Motorola (che prese il nome di Freescale) lo sviluppo di soluzioni G4 fu ripensato in chiave embedded, rinominando il core in e600 e utilizzando come base il 7448.

Dal 2006 si è visto un riavvicinamento delle architetture POWER e PowerPC che ha portato alla definizione di Power Architecture, un set di istruzioni condiviso che condensa le caratteristiche di entrambe, relegando le differenze non più all'architettura ma alle caratteristiche incluse in base alla categoria (Base, Server, Embedded).

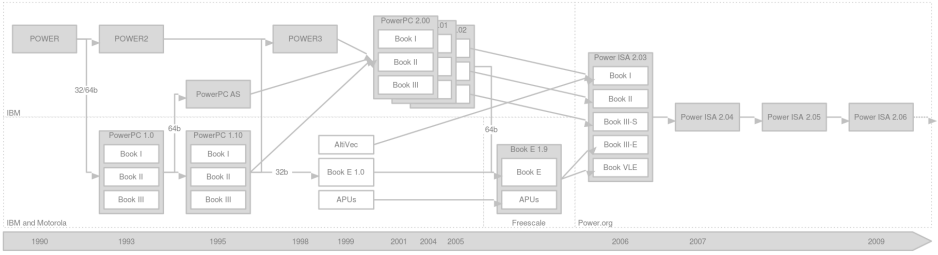


Figura 3.1: Evoluzione dell'architutura PowerPC

Esempi recenti di chip basati su Power Architecture sono:

- Xenon, Broadway e Cell (processori per console utilizzati rispettivamente da Xbox 360, Nintendo Wii e Playstation 3 e prodotti da IBM)
- POWER7 (processore per supercomputer ideato da IBM)
- QorIQ (evoluzione della piattaforma PowerQUICC di Freescale, indirizzata al mercato embedded)

3.2.2 Definizione dell'architettura

L'architettura PowerPC, in quanto aderente ai concetti RISC (Reduced Instruction Set Computer) [5] , è caratterizzata dai seguenti obiettivi:

- Semplificazione della progettazione dell'hardware e del compilatore
- Massimizzazione delle prestazioni

- Minimizzazione dei costi

Per raggiungere questi risultati le architetture di tipo RISC applicano dei concetti fondamentali:

- Le istruzioni sono poche, tutte di uguale lunghezza (32 bit nel caso dell'architettura PowerPC) ed eseguite in tempo costante dall'hardware

- Tutte le operazioni si svolgono tra registri o al massimo tra registri e operatori immediati (costanti)

- Esistono solo due istruzioni che trattano direttamente con la memoria (load from memory e store to memory)

Per poter far in modo che un'architettura di questo tipo possa rivaleggiare con una CISC (che consente tutti i tipi di operandi e lunghezza variabile delle istruzioni al costo di avere molto più silicio occupato ma una velocità maggiore) sono necessari un gran numero di registri e delle tecnologie che permettano di ricavare dei vantaggi dalla semplicità della fase di decode delle istruzioni.

3.2.3 Registri

Nelle ultime versioni delle specifiche (Power ISA 2.06) è obbligatoria la presenza di 32 General Purpose Registers (a 32 o 64 bit), 64 Vector Scalar Registers (32 Floating Point a 64 bit e 32 Vector a 128 bit), 8 Condition Registers e un gran numero di registri Special Purpose (circa 100 nei processori Freescale).

Una delle caratteristiche più particolari dell'architettura Power è l'assenza di registri indispensabili come lo stack pointer; questi vengono mappati in registri General Purpose seguendo delle convenzioni dette ABI (Application Binary Interfeces) con il solo scopo di garantire l'interoperabilità tra programmi scritti da persone diverse.

L'ABI del ppc32 segue le specifiche SVR4 (System V R4) le cui convenzioni sono mostrate nella tabella 3.2.

Alcuni registri speciali, come ad esempio il link register, non possono essere scritti direttamente ma solo attraverso istruzioni specifiche la cui coerenza sarà controllata dal compilatore.

Register	Classification	Notes
r0	local	commonly used to hold the old link register when building the stack frame
r1	dedicated	stack pointer
r2	dedicated	table of contents pointer
r3	local	commonly used as the return value of a function, and also the first argument in
r4-r10	local	commonly used to send in arguments 2 through 8 into a function
r11-r12	local	volatile (caller-save): must be saved before calling a subroutine and restored after returning
r13-r31	global	non volatile (callee-save)
lr	dedicated	link register; cannot be used as a general register. Use mflr (move from link register) or mtlr (move to link register) to get at, e.g., mtlr r0
cr	dedicated	condition register

Tabella 3.2: Registri General Purpose

3.2.4 Superscalarità, Pipelining e Branch Prediction

Tre caratteristiche molto importanti presenti nei microprocessori recenti sono la superscalarità, la presenza di pipeline e di uno o più branch predictor. Queste tre tecnologie sono indispensabili per garantire un impiego ottimale del processore, impegnato altrimenti più ad aspettare che ad eseguire calcoli.

Un processore è superscalare quando più unità funzionali sono replicate e possono funzionare in parallelo. Questo concetto è nato su architetture RISC a causa della semplicità e modesta richiesta di area delle sue ALU; in tempi più recenti, grazie allo scaling tecnologico, si è potuto applicare ad architetture CISC come l'x86. Affinché queste unità possano eseguire più di un'operazione all'interno di un ciclo di clock è necessario rifornirle continuamente di dati; vengono in aiuto in questo caso pipelining e branch prediction. Il primo definisce una tecnologia per accodare le istruzioni ed eseguirle in diversi stadi contemporaneamente. Ad esempio, in un processore privo di pipeline, l'unità che svolge il fetch dell'istruzione viene utilizzata solo una volta fino alla fine dell'esecuzione dell'istruzione stessa, comportando una perdita di rendimento. Se invece viene implementata una pipeline (letteralmente: "condotto") la stessa unità esegue la sua funzione una volta ogni ciclo di clock (Figura 3.2). L'architettura più *pipeline-friendly* è

ancora una volta quella RISC, dal momento che tutte le istruzioni hanno durata temporale fissa.

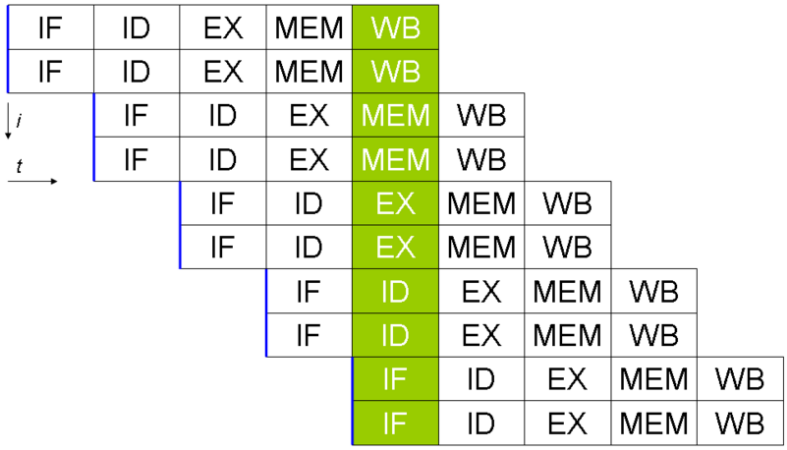


Figura 3.2: Pipelines in un processore superscalare

L'unico problema delle pipeline è dato dalle istruzioni di salto condizionato: non si può infatti sapere quale sarà il risultato della condizione del salto fino alla sua uscita dalla pipeline, continuando solo a quel punto (con la pipeline vuota) l'esecuzione. Per risolvere l'inconveniente sono stati inventati molteplici algoritmi di branch prediction, implementati poi direttamente in hardware. Questa tecnologia si occupa di prevedere il risultato della condizione di salto prima che questa sia verificata, in modo da continuare a riempire la pipeline. Sono stati sviluppati branch predictor che forniscono la risposta giusta più del 95% delle volte! Nelle tecnologie più avanzate si può sfruttare la superscalarità per prevedere N differenti scenari, scegliendo poi quello effettivamente preso con la minima penalità possibile (come se il salto non ci fosse mai stato).

É anche possibile far funzionare più predictor in parallelo, alcuni veloci ma poco precisi insieme ad altri più lenti ma anche più precisi. Se la speculazione veloce viene confermata non abbiamo perso tempo, mentre se viene smentita dell'altro predictor basta inserire una bolla nella pipeline (penalità di un ciclo di clock) per poi eseguire il salto.

3.2.5 Cache

I meccanismi appena presentati sarebbero inutili se le istruzioni e i dati dovessero essere recuperati direttamente dalla RAM (con latenze dell'ordine di almeno un

centinaio di cicli di clock). É quindi necessario che la memoria sia divisa in gerarchie, in cui ogni livello risponda a diversi requisiti di velocità, latenza e capacità.

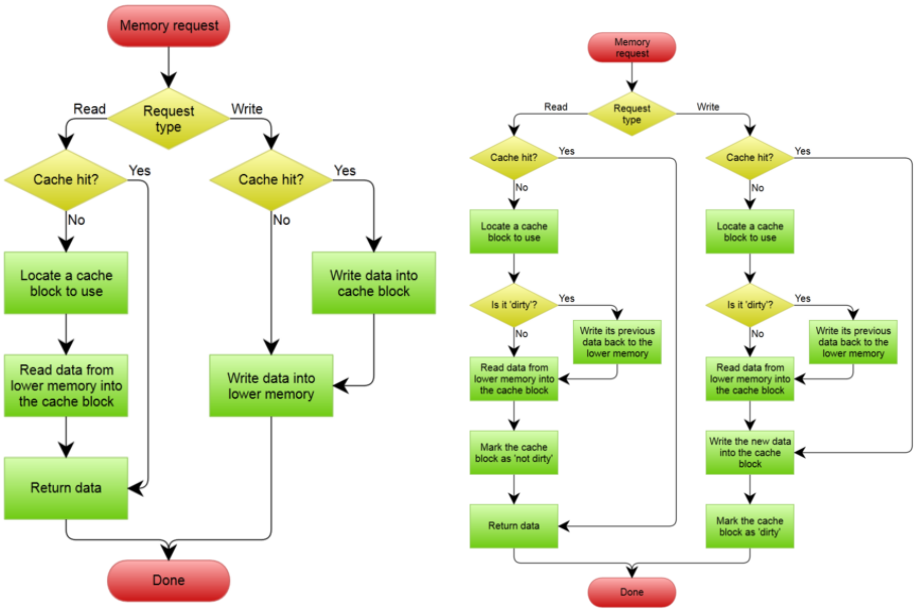
Le memorie più vicine al processore (spesso estremamente vicine in quanto incluse nel die) sono chiamate cache e contengono le istruzioni e i dati che serviranno più probabilmente nei cicli successivi.

La cache può essere divisa tra dati e istruzioni (architettura Harward) oppure unificata. Il dato residente in cache è identificato da un tag che indica il suo indirizzo in memoria e che consente di capire se il valore richiesto può essere recuperato dalla cache stessa (in questo caso si ha un hit) o se è necessario leggere la memoria centrale (miss). Considerando la ridotta capienza delle cache, necessaria a garantirne un'elevata velocità, i dati in essa contenuta dovranno essere spesso rimpiazzati da altri; vari algoritmi di replacement offrono una soluzione poco costosa (in termini di perdita di velocità) al problema. Un dato modificato in cache viene riscritto in RAM utilizzando una delle due policies principali: si parla di write-through se la scrittura viene fatta simultaneamente in cache e in memoria, di write-back se il dato modificato viene segnato con un bit *dirty* e copiato in RAM solo quando è candidato ad essere rimpiazzato (è curioso notare come la seconda soluzione, che sembra migliore dal punto di vista delle tempistiche, richieda due accessi alla memoria principale per completare un replacement: uno per scrivere il dato dirty ed un altro per recuperare il sostituto).

Tutto il meccanismo di caching si basa sul trade-off tra ricerca della velocità e necessità di minimizzare i miss; per questo motivo l'organizzazione delle cache dipende fortemente dalle caratteristiche "di contorno" del processore. Sono state sviluppati tre paradigmi di organizzazione:

- Direct mapping: ogni entry può essere allocata in un solo indirizzo della cache (ottenuto di solito attraverso l'operazione *indirizzo-fisico mod grandezza-cache*). Questa tecnica è la più veloce dal momento che, alla richiesta del dato, basta controllare una location in cache per scoprire se è presente o no; la frequenza di miss è però molto alta, soprattutto se la cache è piccola. Viene perciò utilizzata se la penalità di miss non è così elevata e la velocità di esecuzione è fondamentale.

- Full Associativity: al contrario del direct mapping, un dato può essere allocato in una qualunque posizione in cache. La frequenza di miss diminuisce



(a) Write Through (b) Write Back

Figura 3.3: Confronto tra strategie Write Through e Write Back

spaventosamente, ma aumenta la lentezza dal momento che per sapere se il dato è presente bisogna eseguire una ricerca attraverso tutti i tag.

- N-way set Associativity: tenendo presente il motto latino *in medio stat virtus*, spesso la soluzione migliore sta nel mezzo: in questo tipo di organizzazione un dato può essere mappato in N locazioni diverse, riducendo i miss rispetto al direct mapping ed aumentando la velocità rispetto alla full associativity in quanto la ricerca coinvolgerà solo gli N indirizzi possibili. Solitamente N va da 2 a 16 ma i valori estremi non vengono quasi mai utilizzati a causa del loro comportamento, troppo simile alle controparti "pure" (direct mapping per N=2 e full associativity per N=16) al costo di un'elevata complessità circuitale.

Dal momento che la modifica della memoria non è un'esclusiva del processore ma può essere eseguita da qualunque dispositivo dotato di DMA (Direct Memory Access) è necessario garantire la coerenza dei dati, attraverso complicate e costose unità hardware oppure rendendo *non cacheable* l'area di memoria soggetta a DMA.

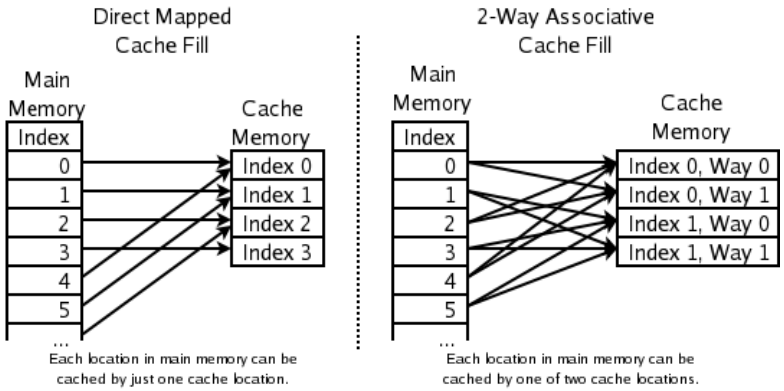


Figura 3.4: Differenza tra direct mapping e 2 way set associativity

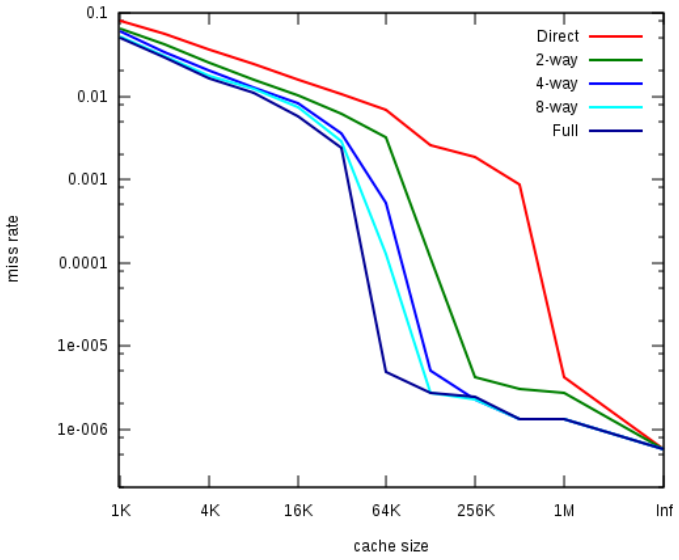


Figura 3.5: Miss Rate vs Cache Size, working set $\sim$ 32kB

3.2.6 Assembly PowerPC

L'assembly PPC possiede una discreta quantità di caratteristiche originali che meritano una rapida panoramica. Innanzitutto i registri non hanno un nome ma solamente un numero identificativo, che è lo stesso per tutte e tre le tipologie di dati (ad esempio 4 può rappresentare: il valore 4, il registro gpr4, il registro fpr4 o il registro vr4 a seconda del tipo di istruzione e della sua posizione tra

gli operandi). Ovviamente il codice risulta molto difficile da decifrare (trovare la differenza tra *add 3,3,4* e *addi 3,3,4* può non essere uno scherzo) quindi l'assembler accetta (e anzi consiglia vivamente) l'utilizzo dei nomi convenzionali che poi lui tradurrà in assembly puro.

Altra caratteristica interessante è l'ambiguità del valore 0: se infatti nei casi precedenti l'ambiguità era dovuta solo al modo di rappresentazione, nel caso del valore 0 le cose si complicano. Vediamo ad esempio l'istruzione load immediate: *li 3,1* carica il valore 1 nel registro r3, ma in realtà è solo un'istruzione mnemonica, in quanto viene tradotta come *addi 3,0,1*. Lo zero dovrebbe rappresentare il registro gpr0, ma in questo caso assume il valore immediato di 0 (il comando diventa "aggiungi uno al valore zero e copia il risultato in r3"). Le specifiche PowerPC ammettono questa irregolarità in contesti sempre ben definiti come quello appena presentato.

Un'altra caratteristica importantissima è legata alla lunghezza fissa delle istruzioni: dal momento che ognuna di esse è rappresentata da una word è impossibile caricare in un registro un valore immediato della stessa lunghezza (che può rappresentare un indirizzo assoluto in memoria o un numero). Infatti la struttura dell'istruzione addi è la seguente:

Opcode	src register	dest register	immediate value
6 bits	5 bits	5 bits	16 bits

Per eseguite quest'operazione sono stati introdotti gli operatori @ha e @l (rispettivamente parte alta del valore con estensione di segno e parte bassa) che si occupano del caricamento corretto in due istruzioni (ad esempio tramite la sequenza *lis 4,num@ha ; addi 4,4,num@l*). Può sembrare stravagante ma sui PowerPC a 64 bit il load immediate di un valore richiede 5 istruzioni! Vengono infatti usati altri due operatori (@highest e @higher) e successivamente al caricamento (in 2 istruzioni) dei 32 bit alti la word viene ruotata (un'altra operazione) e viene completato il caricamento con altre due istruzioni. Questi dolori, dovuti alla lunghezza fissa delle istruzioni, sono ripagati ampiamente dalla semplicità dell'implementazione fisica ed architetturale.

3.3 Piattaforma di Sviluppo

3.3.1 Il processore

Il processore Freescale 7448 è l'ultimo rappresentante puro della generazione G4 e il riferimento per l'e600 (cfr. 3.2.1). Le caratteristiche tecniche sono molto interessanti, soprattutto se rapportate al suo basso consumo. Il processore possiede molte delle caratteristiche presentate nel precedente capitolo; le caratteristiche complete sono

Architettura superscalare:

- Capacità di recuparare fino a quattro istruzioni dall'instruction cache contemporaneamente
- Possono esserci fino a 16 istruzioni in esecuzione simultaneamente
- La maggior parte delle istruzioni richiede un solo ciclo di clock
- Pipeline a 7 stadi
- Undici unità di esecuzione indipendenti
  - 4 unità intere (IUs) che condividono i 32 registri GPR
  - Tre IU identici (IU1a, IU1b, and IU1c) possono eseguire tutte le istruzioni intere a parte moltiplicazione, divisione e lettura/scrittura di registri special purpose
  - IU2 esegue varie istruzioni, inclusi confronti, moltiplicazione, divisione e lettura/scrittura di registri special purpose
  - FPU a 5 stadi e 32 registri floating point a 64 bit
  - Pienamente compatibile con lo standard IEEE 754-1985 per operazioni di singola e doppia precisione
  - Supporta la modalità non-IEEE per operazioni time-critical
  - Un'unità vector permute (VPU)
  - L'unità vettoriale intera 1 (VIU1) esegue le istruzioni AltiVec a bassa latenza (ad esempio vaddsbs, vaddshs e vaddsws)
  - L'unità vettoriale intera 2 (VIU2) esegue le istruzioni AltiVec ad alta latenza (come ad esempio vmhaddshs, vmhraddshs e vmladduhm)

  - Un'unità vector floating point (VFPU)
  - Tre cicli di latenza di caricamento per GPR e AltiVec (byte, half word, word, vector) con throughput di un ciclo
  - Quattro cicli di latenza di caricamento per FPR con throughput di un ciclo
  - Nessun ritardo extra introdotto per l'accesso disallineato nei limiti di una doppia word.
  - Supporta sia la modalità big che little endian

Branch prediction:

- La Branch processing unit (BPU) può fornire predizioni statiche e dinamiche
- É presente una branch target instruction cache (BTIC) con 128 entry (32-set, four-way set-associative), praticamente una cache delle istruzioni di branch incontrate. Se l'istruzione target è nella BTIC, il fetch nella coda delle istruzioni avviene un ciclo prima che se venisse recuperata dalla cache. Tipicamente un fetch che esegue un hit nella BTIC fornisce le prime quattro istruzioni del flusso di esecuzione
- Una branch history table (BHT) da 2048 entry con 2 bit per entry capace di fornire 4 livelli di predizione: non preso, spesso non preso, preso e spesso preso
- Fino a tre branch speculativi

Cache:

- Cache L1 on-chip con architettura Harvard (instruzioni e dati separati), 32Kbyte ciascuna , eight-way set-associative (Figura 3.7)
- Algoritmo di replacement Pseudo least-recently-used (PLRU)
- Programmabile come write-back o write-through distintamente per pagina o per blocco.
- L'instruction cache può fornire quattro istruzioni per ciclo di clock; la data cache quattro word.

- Le cache possono essere disabilitate e bloccate in software
- Supporto al check di parità
- Cache unificata L2 on-chip, 1-Mbyte, eight-way set-associative
- Supporto al check di parità sui tag
- Possibilità di scegliere tra ECC e parità sui dati
- Particolari registri consentono l'iniezione di errori per eseguire test su software di error recovery

Supporto a sistemi multiprocessore:

- Protocolli MESI realizzati in hardware per garantire la coerenza della cache

Power Management:

- Il Dynamic frequency switching (DFS) permette di dimezzare o portare ad un quarto la frequenza del processore via software
- Tre modalità di risparmio energetico:
  - Nap: Viene fermato il fetch delle istruzioni. Rimangono in funzione solo il clock per la time base e la logica JTAG
  - Sleep: Tutte le funzioni interne sono disabilitate a parte il PLL, bloccato e funzionante
  - Deep sleep: Anche il PLL viene disabilitato

Un discorso a parte dev'essere fatto sul packaging. Trattandosi, come in tutti i processori recenti, di un ball grid array (nello specifico un *high coefficient of thermal expansion ceramic BGA* a 360 contatti) il package è "inverso" [6], e quindi l'area attiva si trova a contatto coi pin. Irraggiando dal top del package (quindi dal retro del wafer) è necessario quindi correggere la LET per tener conto della perdita di energia subita dal fascio nell'attraversamento del silicio. Tramite assottigliamento della cover sarà possibile ottenere uno spessore di attraversamento prima di raggiungere la zona sensibile di circa 70μm.

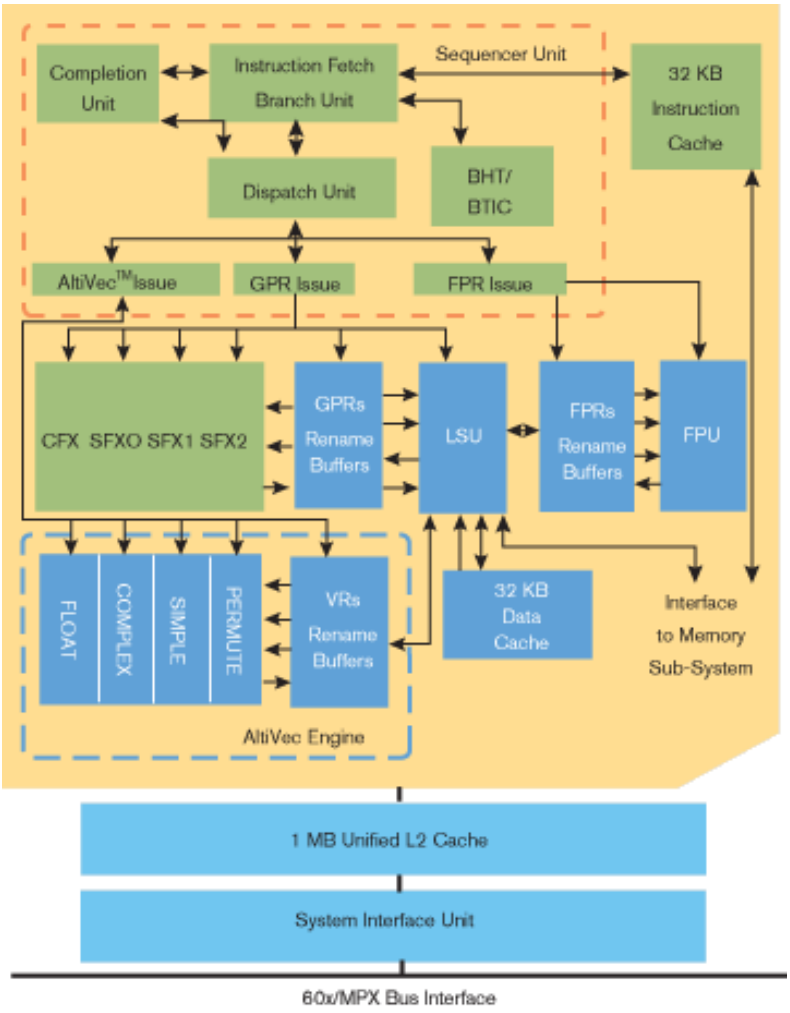


Figura 3.6: Schema a blocchi del processore

3.3.2 La scheda di sviluppo

Dal momento che la scheda finale non era ancora disponibile al momento della scrittura dei software di test, la piattaforma utilizzata è stata l'HPC II (High Performance Computing Platform II) di Freescale. Questa scheda rappresenta il server reference design di Freescale per il 7448 ed è equipaggiata, oltre al già citato processore, di un integrato (Tundra PC109) che svolge la funzione di Host Bridge e di Memory Controller. Quasi sicuramente sarà sostituito, nella scheda finale, da un FPGA basato su un design custom, dal momento che la complessità degli Host Bridges commerciali è decisamente elevata e non sono disponibili dati

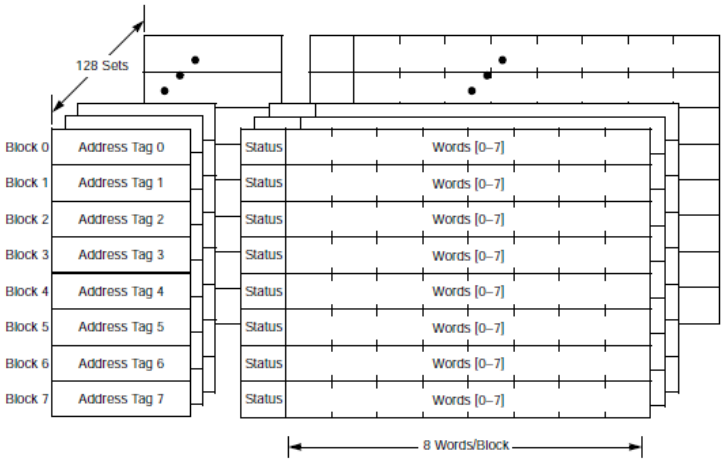


Figura 3.7: Struttura della cache L1 nel processore 7448

di comportamento in ambiente radiativo.

Dal punto di vista delle memorie sono disponibili 16MB di memoria flash e 512MB di RAM. Il software precaricato include due bootloader (U-boot e DINK32), residenti in flash e avviabili separatamente grazie ad un microswitch, ed una distribuzione Linux embedded.

3.3.3 L'ambiente di sviluppo

L'ambiente di sviluppo utilizzato per la tesi è stato CodeWarrior di Freescale. Questo potente IDE permette, tra le altre cose, di visualizzare lo stato dei registri, della RAM ma soprattutto delle caches, semplificando in maniera sostanziale il lavoro richiesto per eseguire il dump di queste ultime. I modi di interfacciarsi con la scheda sono molteplici ed includono seriale RS-232, Ethernet e JTAG/COP. É poi molto interessante la possibilità di scriptare le azioni da compiere, sia in maniera simile ad una macro sia tramite una shell programmabile in Tcl. L'IDE offre anche la possibilità di avviare la scheda in modalità bareboard, usando dei semplici file di testo per inizializzare rapidamente il processore e i registri. Si possono quindi avviare programmi in modalità standalone riducendo in maniera sostanziale l'overhead di informazioni dovute alla presenza di un sistema operativo sottostante.

Capitolo 4

Test Statici

Introduzione

Il seguente capitolo tratta innanzitutto le idee realizzative iniziali, confrontandole con le problematiche principali che si devono prevedere prima di iniziare lo sviluppo dei programmi di test, quindi la scelta della soluzione migliore ed infine l'implementazione finale. Come base di partenza sono state prese le indicazioni (un pò fumose, dal momento che non era certo l'obiettivo dei loro papers) di [2, 7, 8].

4.1 Idee realizzative e Problematiche

Considerando la dotazione hardware e software della scheda di sviluppo (e sperabilmente anche di quella definitiva) le idee di implementazione principali si possono dividere in 3 categorie:

- Test basati sull'esecuzione di un programma all'interno di un sistema operativo completo
- Test basati sull'esecuzione di un programma all'interno di un sistema operativo semplice (non multithreaded) o lanciati da un bootloader
- Test pensati per essere eseguiti senza substrati software (modalità bareboard)

I test possono essere ulteriormente divisi a seconda del grado di automatizzazione nell'esecuzione delle routine, e si va da modalità completamente manuali ad altre completamente automatizzate.

Analizzerò ora le possibili alternative per quasi tutte queste casistiche, mostrandone i pro e i contro, per giungere quindi alla soluzione scelta per essere implementata.

4.1.1 Test all'interno di un S.O.

La caratteristica principale di tutti i Sistemi Operativi recenti (Realtime e non) è la concorrenza. Con questo nome si intende la capacità di eseguire, grazie ad un algoritmo di scheduling, più programmi (task) "contemporaneamente". Ovviamente la sensazione di contemporaneità è data dal rapido susseguirsi di context switches che provocano l'interruzione di un programma per eseguirne un altro.

Nei sistemi non-preemptive la successione di queste interruzioni è data dai programmi stessi, che rilasciano volutamente la CPU per terminazione o mettendosi in coda di dati di I/O. Al contrario, un sistema preemptive consente a task di interromperne altre, in base al livello di priorità assegnato ad ognuno da uno scheduler. Le aree protette del sistema e le parti di codice critico possono essere dichiarate non interrompibili, e solitamente corrispondono a chiamate al kernel (a priorità massima). [9]

Considerando lo scenario prospettato nel capitolo introduttivo, i satelliti di nuova generazione dovranno necessariamente essere dotati di un Sistema Operativo, possibilmente realtime (a causa dei vincoli di temporizzazione stringenti). Sembra quindi che effettuare i test in ambiente radiativo facendo eseguire un programma sopra un S.O. possa essere una buona idea. Tra gli indubbi vantaggi troviamo:

- La possibilità di eseguire test completamente automatizzati, nei quali diversi programmi concorrono all'interno del dispositivo testato con mansioni diversificate, che possono essere l'esecuzione delle routine di prova, il controllo della loro corretta esecuzione, l'analisi e correzione degli errori.
- La possibilità di interfacciarsi con computer esterni (per il download dei risultati o per il controllo del flusso di esecuzione) tramite mezzi molto "semplici" ed economici, quali Ethernet o dispositivi di storage USB.
- Rapidità nella scrittura del codice e facile deployment.
- Se il Sistema Operativo è Linux, Freescale fornisce un programma di debugging molto potente chiamato AppTrk (Application Target Resident Kernel)

che gira sulla scheda ed intercetta tutte le chiamate al kernel in maniera trasparente.

Tuttavia i contro sono molti e non tutti aggirabili:

- Innanzitutto bisogna garantire che il codice di test non venga interrotto durante l'irraggiamento, problema quasi impossibile da evitare se il programma viene eseguito in modalità utente (si può risolvere scrivendo un modulo del kernel, che però vanifica la portabilità dell'applicativo); anche usando AppTrk l'esecuzione del codice non avviene in maniera esclusiva.

- Il codice eseguito potrebbe non essere quello desiderato, a causa della natura distruttiva del test. Questo potrebbe portare la macchina in stati di esecuzione indesiderati che potrebbero compromettere completamente i risultati, e l'eventuale automatizzazione "spinta" porterebbe a scoprire il danno troppo tardi.

- Qualunque tipo di interrupt può portare ad un flush della cache e alla perdita dei SEU occorsi (solo se questa è configurata in modalità write-back), eventualità decisamente apprezzabile in ambiente di lavoro ma decisamente inaccettabile durante un test statico.

4.1.2 Test all'interno di un bootloader (o S.O. minimale)

Una soluzione ai problemi appena posti può essere quella di scegliere come base un sistema operativo minimale, nella fattispecie un bootloader. La mancanza di supporto al multithreading è un pro non indifferente, consentendo l'esecuzione esclusiva del nostro programma (una volta disattivati gli interrupt hardware). Sembrerebbe un ottimo metodo ma presenta anch'esso due inconvenienti:

- La portabilità è estremamente ridotta se il codice viene inserito direttamente nel bootloader (che rappresenta la procedura ottimale nel caso si voglia eseguirlo attraverso il JTAG.)

- Nel caso si preferisca invece lanciare il codice in modalità standalone all'interno al bootloader, l'output del debugger software (basato su gdb) è risultato non affidabile; anche la connessione alla scheda via JTAG "a caldo" comporta dei problemi di non facile soluzione.

4.1.3 Test eseguiti con programmi standalone

Nei paragrafi precedenti non si è particolarmente tenuto contro di quella che è forse la problematica principale che si presenta durante i test: nell'eventualità di un hang del processore infatti si viene, in entrambi i casi, a perdere molto tempo (per accorgersene e ripristinare la situazione iniziale, perdendo oltretutto i dati raccolti). L'unico modo veramente sicuro per evitare questa spiacevole situazione è l'utilizzo di un debugger JTAG.

Per quanto scomodo, ingombrante, contenente elettronica attiva (quindi sensibile anch'essa alle radiazioni) e necessariamente collegato ad un computer di controllo, è l'unico modo veramente sicuro per non incappare nei problemi sopraccitati; a meno che la radiazione non interferisca anche con il modulo on-chip debug all'interno del processore sarà sempre possibile eseguire il dump dei registri e della cache.

Oltretutto una delle caratteristiche principali dell'integrazione tra l'IDE (CodeWarrior) e il debugger JTAG è quella di consentire al programmatore una rapida "messa in funzione" della scheda tramite dei file di testo contenenti tutte le direttive necessarie all'inizializzazione (valore da assegnare ai registri di controllo etc.).

4.2 Scelta della soluzione migliore

Quest'ultima soluzione è indubbiamente la migliore, consentendo la portabilità dell'applicativo su tutte le schede dotate della stessa famiglia di processori senza la necessità di modificare pesantemente il codice , al solo costo di cambiare le direttive contenute nel file di inizializzazione seguendo i datasheet del nuovo hardware sottostante (ad esempio del Bridge chip)

Un altro fattore che porta a questa scelta è l'indisponibilità della scheda finale e quindi l'impossibilità di eseguire su di essa test in laboratorio per valutare l'affidabilità della soluzione.
Le modalità di implementazione possono essere due:

- L'avvio di un programma dummy che non esegue niente (o un loop infinito sull'istruzione NOP[1]) mentre attraverso i file di configurazione e la console di CodeWarrior si inizializzano i registri e la cache con i valori prescelti.

[1] no-operation, in assembly ppc rappresentata dall'opcode 0x60000000 (ori r0, r0, 0)

Viene poi interrotto il ciclo ed effettuato il dump delle memorie di nostro interesse.

- L'avvio in modalità debug di un programma che esegue in software le stesse inizializzazioni, entrando poi in un loop infinito. I file di testo si occupano solo del setup della scheda; il download dei dati avviene allo stesso modo.

Sebbene la prima di queste alternative sia molto interessante si scontra con un problema pratico di impossibile soluzione: il debugger JTAG fornitoci (Freescale USB TAP) si interfaccia al computer host attraverso l'USB; l'overhead dovuto a questo tipo di connessione sommato al peculiare metodo di lettura/scrittura della cache (ipotesi confermata da un ingegnere Freescale) non consente di scriverne il contenuto in tempi accettabili (la scrittura della cache L2, di dimensione 1Mbyte, impiega circa 20 minuti con il clock del debugger impostato a 20MHz, un'ora col clock a 4MHz). Anche la lettura è molto lenta, risultando comunque circa due volte più veloce della scrittura. Oltretutto l'inizializzazione dei registri floating point è affetta da un bug che ne impedisce il corretto caricamento (i valori presenti in questi registri si comportano in maniera simil-randomica)

Per questi motivi la soluzione scelta è stata la seconda, pur necessitando di più codice. A causa dei problemi di lentezza nel download si è scelto di non scaricare l'intera cache L2 ma solo una sua porzione per poi interpolare il risultato, dal momento che si presume che la cross section in funzione del numero di bitflips sia uguale tra L1D ed L2 (vedi [4]).

4.3 Implementazione

L'implementazione si basa su del codice C con l'utilizzo di assembly nelle parti che regolano il caricamento dei registri. Il codice integrale è riportato con commenti nella sezione Codice.

Per riempire la cache con il valore desiderato (nel nostro caso il numero esadecimale 0x55555555, corrispondente in binario al pattern a scacchiera 010101...) è stato usato un hack che prende spunto dal modo di funzionamento della stessa cache; le principali operazioni sono state tre:

- Attivazione della cache configurando l'L2 (di norma unificata) come solo dati (bit DO del registro di controllo L2CR = 1)

- Creazione di un grande array di interi inizializzato col valore del pattern.

- Qualche migliaio di scambi di dati tra le celle dello stesso array in modo da portare il loro contenuto in cache

- Scrittura della memoria appena superiore al top dello stack con lo stesso pattern (per qualche motivo, anche se l'array è allineato, quest'area di memoria entra in L1D ed essendo presenti in essa dati non inizializzati è meglio sovrascriverla in modo da non falsare i test)

Si è scelto, a causa della soluzione adottata, di non toccare la cache L1I e di confrontare il dump post-irraggiamento con il contenuto ottenuto da una sessione di prova svolta in laboratorio. Questa cache risulterebbe praticamente vuota a causa della risicata quantità di istruzioni presenti nel codice; è stato quindi incluso anche un file C (non un header) contenente svariate funzioni utili (inutilizzate) che vengono comunque caricate in cache in base al principio di spazialità.

Per caricare i registri con lo stesso pattern si sono usate le istruzioni assembly relative ad ognuno dei set accessibili all'utente. Per i registri General Purpose si è preferito utilizzare l'istruzione li (load immediate) tramite una macro (come riportato nel capitolo 3.2 la caratteristica dell'architettura PowerPC di avere istruzioni di lunghezza fissa 32bit impedisce il caricamento immediato di una word in un registro; bisogna quindi utilizzare più istruzioni). Per i registri floating point e vettoriali si è invece scelto di effettuare un load dalla memoria (che era stata precedentemente riempita per effettuare il caricamento delle cache). Questa operazione richiede una sola istruzione per i registri floating point e due istruzioni AltiVec per i registri vettoriali.

Finita la parte di inizializzazione, il programma entra in un loop infinito su NOP dal quale può uscire solo se due registri differiscono tra loro; considerando remota quest'eventualità, si suppone che il ciclo continui fino ad un'interruzione esplicita da parte dell'operatore (ad esempio alla conclusione dell'irraggiamento). Il check è interamente escludibile tramite define, rendendo possibile il loop infinito incondizionato. Nel caso non ci sia il define ed avvenga un cambiamento nel valore di un registro specifico (che può essere anche effettuato via JTAG) si esce dal ciclo: l'evento provoca il flush delle cache in RAM e termina il processo. L'idea di fondo parte dal fatto che che il debugger è molto più efficiente nell'effettuare il dump della RAM piuttosto che della cache.

Uno script in Tcl (incluso anch'esso nella sezione Codice) controlla tutti questi eventi, fornendo un'"interfaccia" semplificata ma allo stesso tempo non limitante all'operatore. Dopo aver effettuato lo stop del processo si occupa di avviare il

dump di caches e registri (nel caso della cache L2 viene scaricato solo un subset di celle per rientrare nei vincoli temporali imposti dal test, ovvero al massimo 5 minuti). Il passo successivo è l'analisi in tempo reale degli errori occorsi: viene lanciato un altro programma (anch'esso presente in Codice) che formatta, converte e scansiona il report cercando eventuali bitflip. Oltre a mostrare il numero di eventi occorsi il programma riporta anche l'eventuale asimmetria tra le transizioni $0 \rightarrow 1$ e $1 \rightarrow 0$.

Capitolo 5

Test Dinamici

Introduzione

Il seguente capitolo tratta la realizzazione di test dinamici e semi-dinamici per la valutazione dei soft errors in condizioni operative. Dal momento che in questo tipo di test è importante osservare il comportamento di un'applicazione reale (meglio ancora se si tratta di qualche routine definitiva) il target è stato posto più sul metodo di realizzazione che sull'effettivo funzionamento del programma di test. Le idee sono mutuate in gran parte da [10] e da [7] con alcuni spunti personali dovuti alle specifiche della scheda finale.

5.1 Analisi del problema

I test di irraggiamento statici sono estremamente utili per capire cosa accadrà realmente nel momento in cui il processore si troverà in ambiente spaziale. Offrono anche un valore numerico facilmente comprensibile (la cross section per LET) che permette di confrontare la resistenza in ambiente ostile di dispositivi anche molto diversi tra loro.

La condizione in cui avviene l'irraggiamento è però decisamente pessimistica: il processore è fermo (o comunque non esegue nulla) e tutte le componenti sensibili contribuiscono alla possibilità di "vedere" un errore.

La realtà è estremamente diversa; analizzando un programma reale mediamente complesso in esecuzione sul microprocessore ci accorgiamo che:

- Solo un subset dei registri viene utilizzato (di solito i compilatori tendono ad ottimizzare così il tempo necessario al salvataggio del contesto nell'even-

tualità di un context switch; se il codice non è in assembly vengono seguite strettamente le ABI del processore)

- Neanche tutti i registri utilizzati sono ugualmente sensibili ai bitflip: se uno di questi dovesse essere, per la maggior parte del tempo, un registro target (sul quale vengono effettuate solo scritture) l'eventualità di scoprire un bitflip occorsogli sarebbe estremamente bassa in quanto il dato in esso contenuto verrebbe "ripulito" molto spesso

- Nei test statici i controlli di parità ed ECC sulle caches sono disabilitati in quanto il processore è fermo; nei test dinamici sono invece funzionanti, proteggendo in maniera completa dai single bitflips e in maniera parziale (solo per l'L2) dai multiple bitflips.

- Se la cache dovesse essere impostata in modalità write-through si viene ad eliminare anche la remota possibilità che un dato bypassi i controlli sulle cache e venga portato in RAM; infatti vengono scritti solo i valori appena modificati dal software, e non quelli frutto di soft errors.

Seguendo logicamente queste considerazioni si capisce come non si possono prevedere valori importanti come il numero di errori o il numero di hangs in un dato periodo operativo solo tramite test statici. Una volta eseguiti questi per avere un'idea sull'affidabilità del componente è quindi necessario predisporre dei test più significativi, che daranno un valore attendibile dei guasti che potranno occorrere in situazione operativa.

Essendo molto più *application specific*, questi test devono necessariamente utilizzare del codice che effettivamente girerà sulla scheda definitiva. In mancanza di questo i fattori da tenere in considerazione durante la scrittura sono molti:

- Tentare di utilizzare il più alto numero possibile di unità funzionali, in particolar modo stressando le ALU relative a tutti i set di registri.

- Creare dei cicli la cui esecuzione sia mediamente lunga utilizzando però poco codice di facile decifrabilità

- Produrre una firma riconoscibile alla fine di ogni iterazione, ovvero lavorare sempre sugli stessi dati (magari salvati in una porzione protetta della RAM)

Lo scopo di questi test è principalmente l'acquisizione di un dato univoco che possa condurre ad una scelta oculata riguardo la contromisura più adeguata da

adottare contro i SEE. Esistono infatti una valanga di soluzioni hardware e software pensate per garantire che il processo sia privo di errori o che questi siano recuperabili, ma ognuna di queste richiede un costo più o meno esplicito in termini di potenza elaborativa. Grazie ai test dinamici si può capire quale possa essere la soglia di errore accettabile, differenziata per unità funzionali e quindi per importanza all'interno dell'elaborazione complessiva. Ad esempio un programma che dovesse eseguire solo calcoli con interi potrebbe benissimo rinunciare a molti dei sistemi di sicurezza necessari per garantire l'immunità agli errori dei registri vettoriali (i quali occupando un'area quadrupla saranno più soggetti all'insorgenza di bitflips).

5.2 Esempi di test

In letteratura esistono principalmente due test "standardizzati", uno riguardante i registri GPR e l'altro gli FPR.

Il primo dei due si basa sulla scrittura di un valore in un registro (solitamente il pattern a scacchiera 0x55555555), la moltiplicazione per il contenuto del successivo (0x2) ed il confronto col contenuto di un terzo (0xAAAAAAAA). Se il confronto va a buon fine il ciclo riprende utilizzando i registri immediatamente successivi, altrimenti viene riportato l'errore salvando il contenuto del risultato. Da questa informazione si può tentare di ricavare l'errore occorso. Questo metodo, per quanto elegante, ha il problema di sottostimare gli errori derivanti dai bitflip avvenuti nell'instruction cache, dal momento che questa è praticamente vuota.

Per quanto riguarda la virgola mobile i programmi standard sono di solito benchmark matematici quali il calcolo dell'i-esima cifra di π, l'esecuzione di un algoritmo per calcolare l'FFT su un certo numero di campioni o il calcolo del frattale di Mandelbrot per un numero adeguato di iterazioni. Tutti questi algoritmi sono estremamente CPU-intensive ed è possibile decidere in maniera abbastanza precisa la durata temporale di ogni iterazione.

Dal momento che il programma individuato per il test vero e proprio sarà un'FFT su N campioni mi sono concentrato sulla creazione di altri due test, uno dei quali impieghi i registri vettoriali attraverso il set di istruzioni AltiVec.

5.3 Test GPR - Sudoku

Questo test implementa un algoritmo di risoluzione di Sudoku che non sfrutta alcuna ottimizzazione al fine di occupare il più possibile le cache. Il completamento richiede 50 iterazioni del loop (dal momento che lo schema da risolvere è sempre lo stesso) e, se al termine di queste non si giunge ad una soluzione, viene stampato il risultato errato la cui conoscenza potrà tornare utile per analizzare il problema occorso. La istruzioni assembly più presenti sono confronti, addizioni, sottrazioni e divisioni tra interi e movimenti da e verso la memoria (nel nostro caso la cache ad esclusione dei compulsory miss durante la prima iterazione). Viene quindi analizzato dal test il comportamento delle unità intere (UIs) e della branch target instruction cache (BTIC). Il codice è riportato nel capitolo 7.4.

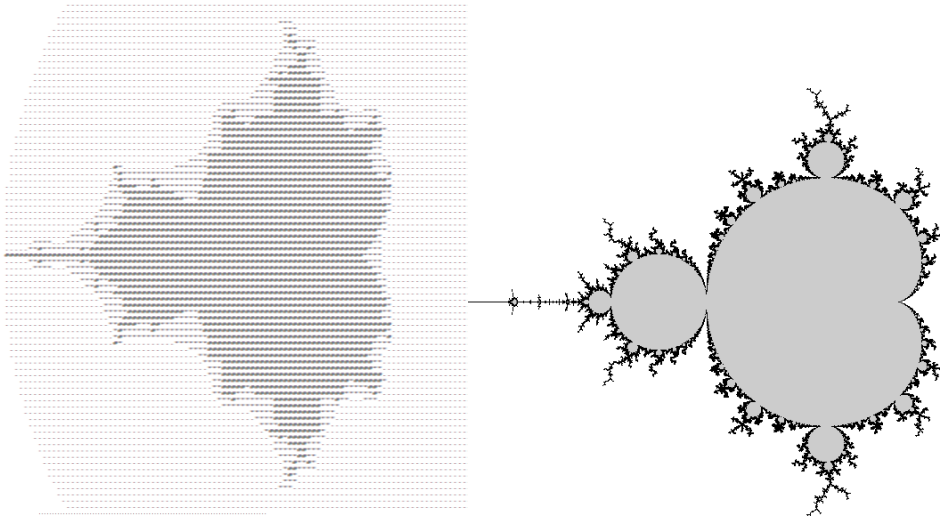
5.4 Test AltiVec - Mandelbrot

Più interessante dal punto di vista dell'applicazione finale è il test sulle istruzioni AltiVec. Queste sono infatti il vero valore aggiunto del processore, ed il boost prestazionale ottenibile sfruttandole per calcoli intensivi è veramente elevato. I calcoli maggiormente vettorizzabili sono quelli che coinvolgono un array di valori e la ripetizione della medesima operazione su ogni elemento. Uno degli algoritmi più CPU-intensive a possedere queste caratteristiche è il calcolo di frattali.

La definizione di frattale è: l'insieme dei punti per i quali la successione definita non diverge. Una caratteristica fondamentale è l'appartenenza dei punti all'insieme dei numeri complessi, fattore che rende la successione "fluttante" anche se l'operazione da applicare è estremamente semplice. Il primo a scoprire le importantissime proprietà (non solo geometriche) di questi oggetti matematici fu Benoit Mandelbrot, che li mostrò al mondo per la prima volta nel suo *Gli Oggetti Frattali* nel 1975. Il primo frattale da lui proposto fu il cosiddetto "frattale di Mandelbrot", la cui definizione sconcerta per semplicità: infatti la serie necessaria per generarlo è $z_{n+1} = z_n^2 + c$ con z_0 e c complessi. [11] L'algoritmo implementato è la traduzione in codice AltiVec di una versione in C utilizzata per effettuare benchmark [12]. L'unica modifica apportata è la vettorizzazione delle operazioni, che abilita l'algoritmo ad agire su quattro "pixel" alla volta.

Il risultato è un'area di memoria contenente degli interi che rappresentano il grado di appartenenza al frattale secondo la convenzione PBF (Portable Bitmap Format), facilmente confrontabile (*in loco* o dopo averne eseguito il dump) con

il risultato corretto. Gli obiettivi del benchmark sono stessare tutte le unità vettoriali (a parte la vector permute) ed avere una grande quantità di cache occupata (la generazione di una figura 512x512 pixel riempie interamente la cache L2).



(a) Risultato del test - ascii (b) Frattale in alta definizione

Figura 5.1: Frattale di Mandelbrot

5.5 Peculiarità della scheda di test

La scheda finale possiede delle caratteristiche peculiari che meritano di essere testate; nella fattispecie sono presenti due watchdog (uno dei quali "smart"), un algoritmo di data comparison, la possibilità di proteggere da scrittura segmenti della DDR e un cosiddetto "scrub reset". Queste funzionalità fanno quasi tutte parte di una periferica denominata Health Control, realizzata su FPGA e responsabile del controllo della corretta esecuzione del flusso.

5.5.1 Smart Watchdog

Ad un watchdog tradizionale (contatore a 32bit, clock 10MHz, resettato dalla CPU se questa non va in stallo) viene affiancato uno smart watchdog, decisamente più sofisticato e destinato al controllo fine del flusso di esecuzione. Può monitorare un elevato numero di eventi controllando anche che le tempistiche ven-

gano rispettate. Non sarà testato all'interno della nostra procedura dal momento che non sarà mai in esecuzione più di un programma contemporaneamente.

5.5.2 Data Comparison

Si presenta come un set di registri che permette di confrontare dati, ad esempio due ripetizioni dello stesso calcolo. I modi di funzionamento sono due: comparazione tra numeri e majority voting. Nel primo caso si confrontano i due risultati e, se non corrispondono, viene generato un interrupt che segnala alla CPU il malfunzionamento. Nel secondo caso, invece, si procede al confronto tra numeri e se questi risultano diversi si esegue un'ulteriore volta il calcolo per procedere poi per majority voting. Dal momento che queste tecnologie devono essere implementate con un elevato livello di compatibilità con il software in esecuzione non sarà probabilmente possibile eseguire il test di questa funzionalità in ambiente radiativo.

5.5.3 Scrub Reset

Con questo nome si intende descrivere una procedura capace di resettare la CPU ad intervalli periodici quando questa non stia eseguendo operazioni (per eliminare la necessità di salvare dei dati). Lo scopo della funzionalità è di "pulire" eventuali SEU interni che alla lunga potrebbero accumularsi causando il blocco della CPU. Questa è la caratteristica più interessante da testare, dal momento che sarà possibile escluderla e controllare la sua effettiva utilità tenendo costante il flusso applicato alla scheda nelle due condizioni d'uso. É anche l'unica che per essere testata necessita di test in ambiente radiativo dal momento che dovrebbe proteggere da effetti non simulabili attraverso iniezione di guasti.

Capitolo 6

Conclusioni

Il problema affrontato in questa tesi "vive" sul confine tra elettronica ed informatica. Per l'elettronico puro le questioni riguardanti il software sono spesso viste come un inutile fastidio, un'ulteriore sovrastruttura che lo separa dalla vera comprensione fisica dei fenomeni che avvengono nei dispositivi. Per un informatico puro, invece, il processore è una scatola nera nella quale i suoi programmi vengono eseguiti alla perfezione; per lui l'hardware è un substrato infallibile sul quale fare cieco affidamento. Ovviamente i due estremi non hanno molto senso di esistere (sono più degli stereotipi) ma accade comunque spesso che i due mondi debbano avvicinarsi.

Durante lo svolgimento di questo lavoro di tesi ho notato più volte che sarebbe bastata un pò più di conoscenza dell'"altro" campo (qualunque esso fosse in quel momento) per evitare errori stupidi o semplicemente per migliorare il lavoro.

Per questo motivo avrei desiderato testare sul campo i miei programmi ma sfortunatamente l'indisponibilità della scheda definitiva ha fatto slittare i tempi; con questa a disposizione sarebbe stato molto interessante implementare dei test con iniezione di guasti e vedere (anche in laboratorio) cosa ci si sarebbe potuto aspettare dagli irraggiamenti. In ogni caso l'obiettivo del lavoro è stato raggiunto, soprattutto per quanto riguarda la configurabilità del codice e la sua riutilizzabilità.

Le tematiche affrontate (che a prima vista possono sembrare molto di nicchia) hanno una grande validità in molti settori in decisa crescita quali l'integrazione hardware-software nei dispositivi portatili (che ha permesso la nascita del mercato delle App) e l'affidabilità degli stessi (basti pensare che Intel produce ancora i suoi chip su substrati bulk e che l'imminente step tecnologico è 22nm, circa 4

volte più scalato del processore analizzato). Il principale beneficiario di queste ricerche è quindi il mercato *embedded*, a causa della crescente necessità di autonomia (quindi consumi e tensione di alimentazione ridotti) e di integrazione dei dispositivi portatili che li rende particolarmente sensibili alla radiazione a livello terrestre.

Il lavoro svolto è stato quindi assai utile per acquisire metodi e conoscenze in prospettiva lavorativa, senza sottovalutare però l'aspetto teorico.

Capitolo 7

Codice

7.1 Test statico

```
//////////////////////////////////////////////////////
/////// Programma per il test statico:      ///////////
/////// riempie registri e cache con valori   ///////////
/////// definiti dal define #patt e quindi  ///////////
/////// esegue un loop infinito          ///////////
//////////////////////////////////////////////////////

#include <stdlib.h>
#include <stdio.h>
#include "init_mmu_cache.h"
#include "init_mmu_cache.c"

#define patt 0x55555555   //pattern col quale riempire cache e registri (010101010etc)
//#define dim 2097152
#define dim 800000      //dimensione array

#define infinite_loop       //undefine or comment if you want to be able to exit the
    loop

//macro that fill registers with desired data (could be smarter)

#define load_r(rI) li rI, 0; oris rI, rI, patt@h; ori rI, rI, patt@l
#define load_v(vI)  lvewx vI, r0, r3; vspltw  vI,vI,0

asm void cache_on();
asm void system_call();
asm void write_reg();
void create_array();
void touch_and_go(int a[], int i);
void empty_cache_and_write_back(int a[]);
asm void loop_and_post();
asm void fill_float_and_vectors();
asm void mmu_cache_init();
asm void fill_gaps();

//function for filling modified memory at the end of stack (only because it goes in
    cache)

asm void fill_gaps()
{
  nofralloc
  #define memloc 0x00cd2b80
  li r3, 0; oris r3, r3, memloc@h; ori r3, r3, memloc@l
```

```
  stw r4, 0(r3);
  stw r4, 4(r3);
  stw r4, 8(r3);
  stw r4, 12(r3);
  stw r4, 16(r3);
  stw r4, 20(r3);
  stw r4, 24(r3);
  stw r4, 28(r3);
  stw r4, 32(r3);
  stw r4, 36(r3);
  stw r4, 40(r3);
  stw r4, 44(r3);
  stw r4, 48(r3);
  stw r4, 52(r3);
  stw r4, 56(r3);
  stw r4, 60(r3);
  blr
}

//useless function, only for testing (demonstrates write-back capability)

void empty_cache_and_write_back(int a[])
{
  int i;
  for (i=0;i<dim;i++)
  {
    a[i]=0x00000000;
  }

  //FlushDataCache();
}

//fill all registers with desired data

asm void write_reg()
{
  load_r(r0)
  load_r(r1)
  load_r(r2)
  //load_r(r3)
  load_r(r4)
  load_r(r5)
  load_r(r6)
  load_r(r7)
  load_r(r8)
  load_r(r9)
  load_r(r10)
  load_r(r11)
  load_r(r12)
  load_r(r13)
  load_r(r14)
  load_r(r15)
  load_r(r16)
  load_r(r17)
  load_r(r18)
  load_r(r19)
  load_r(r20)
  load_r(r21)
  load_r(r22)
  load_r(r23)
  load_r(r24)
  load_r(r25)
  load_r(r26)
  load_r(r27)
  load_r(r28)
  load_r(r29)
  load_r(r30)
  load_r(r31)
```

```
  blr
}

//activates caches and mmu

asm void cache_on()
{
//  bl init_caches
  bl mmu_cache_init
}


//initializes floating point and vector registers

asm void fill_float_and_vectors()
{
  nofralloc
  lfd f0, 0(r3)
  lfd f1, 0(r3)
  lfd f2, 0(r3)
  lfd f3, 0(r3)
  lfd f4, 0(r3)
  lfd f5, 0(r3)
  lfd f6, 0(r3)
  lfd f7, 0(r3)
  lfd f8, 0(r3)
  lfd f9, 0(r3)
  lfd f10, 0(r3)
  lfd f11, 0(r3)
  lfd f12, 0(r3)
  lfd f13, 0(r3)
  lfd f14, 0(r3)
  lfd f15, 0(r3)
  lfd f16, 0(r3)
  lfd f17, 0(r3)
  lfd f18, 0(r3)
  lfd f19, 0(r3)
  lfd f20, 0(r3)
  lfd f21, 0(r3)
  lfd f22, 0(r3)
  lfd f23, 0(r3)
  lfd f24, 0(r3)
  lfd f25, 0(r3)
  lfd f26, 0(r3)
  lfd f27, 0(r3)
  lfd f28, 0(r3)
  lfd f29, 0(r3)
  lfd f30, 0(r3)
  lfd f31, 0(r3)

  addi r3, r3, 16

  load_v(v0)
  load_v(v1)
  load_v(v2)
  load_v(v3)
  load_v(v4)
  load_v(v5)
  load_v(v6)
  load_v(v7)
  load_v(v8)
  load_v(v9)
  load_v(v10)
  load_v(v11)
  load_v(v12)
  load_v(v13)
  load_v(v14)
  load_v(v15)
  load_v(v16)
```

```
  load_v(v17)
  load_v(v18)
  load_v(v19)
  load_v(v20)
  load_v(v21)
  load_v(v22)
  load_v(v23)
  load_v(v24)
  load_v(v25)
  load_v(v26)
  load_v(v27)
  load_v(v28)
  load_v(v29)
  load_v(v30)
  load_v(v31)

  blr
}


//readdes and overwrites the array values many times in order to fill the caches

void touch_and_go(int a[], int i)
{
  int k;
  int s;
  for (s=0;s<i;s++)
  {
    for (k=i*1000;k<dim-(i*1000);k++)
    {
      a[k]=a[k+s*i];
    }
  }
}


//BIG array creation

void create_array(int dati[])
{
  int i;

  for (i=0;i<dim;i++)
  {
    dati[i]=patt;
  }

  touch_and_go(dati,20);
  touch_and_go(dati,30);
  touch_and_go(dati,40);
  touch_and_go(dati,50);
  touch_and_go(dati,100);
  touch_and_go(dati,300);
  touch_and_go(dati,350);
}


//loop function and optional post loop code (invalidates and flushes L2)

asm void loop_and_post()
{
  nofralloc
  loop:
  #ifndef infinite_loop
  cmpw    r3, r4
  #endif
  ori    r0, r0, 0      //NOP while r3==r4 (only if infinite_loop not defined)
  beq loop              //if infinite_loop defined loops forever
```

```
  oris     r3, r3, L2GI_EN@h      //executes if r3!=r4; dumps cache to ram
  ori      r3, r3, L2GI_EN@l
  mtspr    l2cr, r3
  isync
  post_proc:          //invalidate L2
  mfspr r4, l2cr
  addis r3, r0, L2GI_EN@h
  ori    r3, r3, L2GI_EN@l
  and    r3, r4, r3
  cmpwi     r3,0x0000
  bne       post_proc

  bl loop
}


void main()
{
  int dati[dim];                                  //array goes in stack
  //int *dati = (int *) malloc(sizeof(int) * dim);       //array goes in heap
  cache_on();
  create_array(dati);
  //  printf("Beginning Test, can't write it or it goes in cache :( \r\n");
  write_reg();
  fill_gaps();
  fill_float_and_vectors();
  asm (load_r(r3));
  loop_and_post();
}
```

7.2 Automazione e controllo CodeWarrior

```
#############################################################
####### Script di automazione per Codewarrior       #######
####### gestice nell'ordine: inizializzazione,      #######
####### avvio dell'applicazione, dump ed esecuzione #######
####### del programma di analisi dati               #######
#############################################################


#inizializzazione: abilita i comandi nascosti, disabilita l'output paginato e passa
     alla cartelle corretta

proc init {} {
puts "Execute only once!!!!!"
cmdwin::setvisible on cmdwin::ca
cmdwin::setvisible on cmdwin::caln
config page off
cd C:/Documents and Settings/RREACTguest/Documenti/Registers

}

#avvio applicazione: lancia il progetto, esegue gli step in modalitÃ  interattiva

proc run {} {
debug Registers.mcp
puts "\n\nPress return key to start the infinite loop"
gets stdin
finish

puts "\n\nPress return key to stop the infinite loop"
gets stdin
stop

puts "\n\nPress return key to get the dump (takes almost 3 minutes)"
gets stdin
```

```
getlog
}

#esegue il dump delle caches e dei registri e stampa il tempo d'esecuzione

proc getlog {} {
set timez [clock seconds]
log s $timez
puts "\n\n"
puts [clock format [clock seconds] -format "%Y-%m-%d %H:%M:%S"]
puts "\n\nPrinting L1 Data cache\n\n"
cmdwin::caln::get 1:0 1024
puts "\n\nPrinting L1 Instruction cache\n\n"
cmdwin::caln::get 0:0 1024
puts "\n\nPrinting L2 Unified cache (set to data only) (64Kb subset) \n\n"
cmdwin::caln::get 2:0 2048
puts "\n\nPrinting Registers\n\n"
# executed after everything else because sometimes it goes wrong and false cache
    readings
reg all
set times [clock seconds]
set elaps [expr "$times - $timez"]
set string1 "\n\nExecution took "
set string2 " seconds\n\n"
puts [append b $string1 $elaps $string2]
log off all
analize $timez
}

#lancia l'analisi dei dati

proc analize {timez} {

puts "\n\nError checking routine\n\n"
cd Bin
#esegue il backup del log in una cartella specifica
exec mv ../$timez ../Logs/$timez
#spezza il log nelle parti giuste riformattandole (grazie a sed)
exec sed -n {14,1037p} ../Logs/$timez | cut -c 25- > ../Temp/l1d_new
exec sed -n {1046,2069p} ../Logs/$timez | cut -c 24- > ../Temp/l1i_new
exec sed -n {2078,4125p} ../Logs/$timez | cut -c 25- > ../Temp/l2d_new
exec sed -n {4132,4204p} ../Logs/$timez > ../Temp/reg_new
exec diff > ../Results/$timez
puts [exec cat ../Results/$timez]

cd ../
puts "\nPress return key to kill the debug session"
gets stdin

################################################
##### TODO: selective kill, rerun, pre log? ######
################################################

kill
}
```

7.3 Routine di controllo errori

```
//////////////////////////////////////////////////////////
/////// Software per analizzare i dump raccolti ///////////
/////// Gira sul computer host e riceve in     ///////////
/////// ingresso 4 files (L1i, L1d, L2(d), reg) ///////////
//////////////////////////////////////////////////////////////

#include <stdio.h>
#include <stdlib.h>
```

```
//macro conversione da decimale a binario; C non lo gestisce come dato stampabile

#define BYTETOBINARYPATTERN "%d%d%d%d"
#define BYTETOBINARY(byte)  \
  (byte & 0x08 ? 1 : 0), \
  (byte & 0x04 ? 1 : 0), \
  (byte & 0x02 ? 1 : 0), \
  (byte & 0x01 ? 1 : 0)



int main(int argc, char* argv[])
{
  int ztoi=0;   //zeros to ones
  int itoz=0;   //ones to zeros
  printf("\nAnalizing L1 Data\n\n");
  confr("../Refs/l1d_ref", "../Temp/l1d_new",ztoi,itoz,0);
  printf("\nAnalizing L1 Instruction\n\n");
  confr("../Refs/l1i_ref", "../Temp/l1i_new",ztoi,itoz,0);
  printf("\nAnalizing L2 Data\n\n");
  confr("../Refs/l2d_ref", "../Temp/l2d_new",ztoi,itoz,0);
  printf("\nAnalizing Registers\n\n");
  confr("../Refs/reg_ref", "../Temp/reg_new",ztoi,itoz,1);
  return 0;
}

//metodo "sporco" per capire se un numero è zero

int sum_all(char* s){

  int i=atoi(s);
  return i;

}

//metodo per la creazione del report; se reg==1 si stanno analizzando i registri (
     codifica diversa del file sorgente)
//chiamato da confr;

int report(FILE* solution, FILE* useful,int ztoi,int itoz,int reg)
{
  int i=0;
  int sbf=0;    //single bitflips
  int mbf=0;    //multiple bitflips

  if (reg==0) {                         //cache routine
  solution = fopen("../Temp/solution", "r");
  useful = fopen("../Temp/useful", "r");
  while (!feof(solution))
  {


    char s[33];           //la lunghezza per le cache può essere 33
    char g[33];
    fgets(s, 33, solution);   //recupera una stringa
    fgets(g, 33, useful);

    int k=(i/2)%8;          //calcola il numero di colonna (utile nel report)
    int u;
    if (sum_all(s)!=0) {      //se il numero recuperato non è 00000000 vuol dire che è
         avvenuto almeno un bitflip
      printf("Bitflips found at row %d, column %d -> originally: %s\n",(i-k)/16+1,k+1,
           g);
      //printf("i: %d\n",i);
      int k=0;            //trova il numero di bitflips e la loro "natura" (0->1 o
           1->0)
      for (u=0;u<33;u++) {
          if (s[u]=='1') {if (g[u]=='0') {ztoi++;} else {itoz++;}
```

```
                    if (k==0) {sbf++;}
                    if (k==1) {sbf--;mbf++;}
                    k++;
                    }

          }
        };

    i++;

  }
  }
  else {                                //modalità registri
  solution = fopen("../Temp/solution", "r");
  useful = fopen("../Temp/useful", "r");
  while (!feof(solution))
  {


    char s[150];       //128bit registri vettoriali + 10simboli (nome registro)
    char g[150];
    fgets(s, 150, solution);
    fgets(g, 150, useful);

    int j=(i/2)%8;
    int u;


      int k=0;              //sembra arabo ma deriva dalla formattazione dell'output
           dalla funzione confr
      for (u=0;u<150;u++) {
          if (s[u]=='1' && s[u+3]!='$' && i!=126 && i!=128) {
                    if (g[u]=='0') {ztoi++;} else {itoz++;}
                    if (k==0) {sbf++;printf("Bitflips found: %s\n",s);}
                    if (k==1) {sbf--;mbf++;}
                    k++;

                    }

          }


    i++;

  }



  }


  fclose(solution);
  printf("Number zeros to ones: %d\n", ztoi);
  printf("Number ones to zeros: %d\n", itoz);
  printf("Words with single bitflip: %d\n", sbf);
  printf("Words with multiple bitflips: %d\n", mbf);

}


//metodo per confrontare i dati acquisiti con quelli reference

int confr(char* refx, char* newx,int ztoi,int itoz,int reg)
{
  FILE* new;
  FILE* ref;
  FILE* solution;
  FILE* useful;
```

```
  ref = fopen(refx, "r" );
  new = fopen(newx, "r" );
  solution = fopen("../Temp/solution", "w");
  useful = fopen("../Temp/useful", "w");


  int a,b,c,l,m;

  int previous=0;
  int reg_done=0;
  if (reg==0) {        //in caso di cache
  while (!feof(new))
    {
      a=fgetc(new)-48;       //converte reference e dati in valore decimale (da ascii)
          ->
      b=fgetc(ref)-48;
      if (a>=10 && a<16) {a=a-39;}
      if (b>=10 && b<16) {b=b-39;}
      c=a ^ b;              //effettua lo xor tra reference e nuovo valore


      //stampa su file e formattazione (per poi darlo in pasto a report)

      if (a<0 && a!=-12)  { if (previous!=0) {fprintf(solution,"\n");previous=0;
           fprintf(useful,"\n");}} else
      //printf("%d %d\n",a,b);
      {fprintf(solution, BYTETOBINARYPATTERN, BYTETOBINARY(c));previous=1;
      fprintf(useful, BYTETOBINARYPATTERN, BYTETOBINARY(b));
      //printf("%d %d\n",a,b);
      }
    }
  }
  else {          //modalità registri (la complessità dipende dalla formattazione del
       file di partenza (codewarrior))
  while (!feof(new))
    {
      l=fgetc(new);
      m=fgetc(ref);
      a=l-48;
      b=m-48;
      if (a>=10 && a<16) {a=a-39;}
      if (b>=10 && b<16) {b=b-39;}
      c=a ^ b;
      if (reg_done==0) {
        if (m!=' ')
        {
        fprintf(solution,"%c", m);
        fprintf(useful,"%c", m);}
      if (a==-12) {reg_done=1;fprintf(solution,"$", m); fprintf(useful,"$", m);}}
      else {
      if (a<0)  { if (previous!=0) {fprintf(solution,"\n");previous=0;reg_done=0;
           fprintf(useful,"\n");}} else
      //printf("%d %d\n",a,b);
      {fprintf(solution, BYTETOBINARYPATTERN, BYTETOBINARY(c));previous=1;
      fprintf(useful, BYTETOBINARYPATTERN, BYTETOBINARY(b));
      reg_done=1;
      //printf("%d %d\n",a,b);
      }
      }
    }


  }
  fclose(solution);
  fclose(useful);
  report(solution,useful,ztoi,itoz,reg);  //lancia report passandogli il file ottenuto
        dallo xor e quello del file acquisito
}
```

7.4 Test dinamico GPR - Sudoku

```
////////////////////////////////////////////////////////
/////// Risolve un sudoku predefinito         ////////
/////// Utile come test statico sull'unitÃ  intera  ////////
////////////////////////////////////////////////////////


#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include "init_mmu_cache.h"
#include "init_mmu_cache.c"
#include "vBigDSP.h"
#include <__ppc_eabi_init.h>


typedef struct cell{
    int numbers[10];
    bool done;
    int how_many;
  int def_number;
}t_cell;

typedef struct table{
  struct cell game[9][9];
}t_table;




void init_cell(int number, int row, int column, t_table *G)
{
     for (int i=0;i<10;i++) {G->game[row-1][column-1].numbers[i] = 0;}
     G->game[row-1][column-1].numbers[number]=number;
     G->game[row-1][column-1].def_number=number;
     G->game[row-1][column-1].done=true;
     G->game[row-1][column-1].how_many=0;
}

void init_all(t_table *G)
{
  int row;
  int column;
  int i=0;
  for (row=1;row<10;row++)
    {
    for (column=1;column<10;column++) {
  for (int i=0;i<10;i++) {G->game[row-1][column-1].numbers[i] = i;}
  G->game[row-1][column-1].done = false;
  G->game[row-1][column-1].how_many = 9;}
    }
}

int find_magic_num(int row, int column)
{
  int a = (row-1)*9+(column-1);
  int b = (a%9)/3;
  int c = a/27;
  return (b*3)+c;
}


void dis_square(int number, int rowx, int columnx, t_table *G)
{

  int mn=find_magic_num(rowx,columnx);
  //printf("%d \n",mn);
```

```
  for (int row=1;row<10;row++)
  {
  for (int column=1;column<10;column++) {
  if (find_magic_num(row,column)==mn && G->game[row-1][column-1].numbers[number]!=0 &&
        G->game[row-1][column-1].how_many!=1){
  //printf("2\n");
  G->game[row-1][column-1].numbers[number]=0;
  G->game[row-1][column-1].how_many--;}}
    }
}

//usa il metodo di risoluzione standard

void disable_lines_and_square(int number, int row, int column, t_table* G)
{


  for (int i=1;i<10;i++){
  if (G->game[i-1][column-1].numbers[number]==number) {
  if (G->game[i-1][column-1].how_many!=1) {
  //printf("0\n");
  G->game[i-1][column-1].numbers[number]=0;
  G->game[i-1][column-1].how_many--;}}}

  for (int i=1;i<10;i++){
  if (G->game[row-1][i-1].numbers[number]==number) {
  if (G->game[row-1][i-1].how_many!=1) {
  //printf("1\n");
  G->game[row-1][i-1].numbers[number]=0;
  G->game[row-1][i-1].how_many--;}}}

  dis_square(number,row,column,G);


}

//stampa il risultato

void printGraph (t_table *G)
{

  printf("\nSudoku solver\n\n");
  for (int row=1;row<10;row++){
    for (int column=1;column<10;column++)
    {
      printf("%d ", G->game[row-1][column-1].def_number);
      if (column==9) printf("\n");
      if (column%3==0 && column!=9) printf("  ");
      if (row%3==0 && column==9) printf("\n");
    }
}

}

//se tutti i numeri dell'array numbers sono 0 a parte uno ritorna 1

int is_found(int row, int column, t_table* G)
{
  int k=0;
  int a;
    for (int i=0;i<10;i++)
    {if (G->game[row-1][column-1].numbers[i]!=0)
      {a=G->game[row-1][column-1].numbers[i];k++;}}
    //if (row==6 && column==9) {printf("%d,%d\n",a,k);};
    if (k!=1) {a=0;};
    return a;
}

//se tutti i numeri dell'array numbers sono 0 a parte uno, conferma quel numero
```

```
void confirm_number(int row, int column, t_table *G)
{
  for (int i=0;i<10;i++)
    {if (G->game[row-1][column-1].numbers[i]!=0)
      {G->game[row-1][column-1].def_number=G->game[row-1][column-1].numbers[i];G->game
          [row-1][column-1].done=true;G->game[row-1][column-1].how_many=1;}
    }
  //printf("number found %d %d %d\n",G->game[row-1][column-1].def_number,row,column);
}

//esegue un giro completo, trova i numeri "definitivi" e continua l'esecuzione

void iterate(t_table *G)
{

  for (int row=1;row<10;row++)
  {
  for (int column=1;column<10;column++) {
  if (is_found(row,column,G)!=0) {


    confirm_number(row,column,G);
    disable_lines_and_square(is_found(row,column,G), row, column, G);

      }
  }
}
}




void main()

{
  //init_alloc();
  mmu_cache_init();
  t_table Gx;
  t_table *G = &Gx;
  init_all(G);
  init_cell(1,1,4,G);
  init_cell(2,2,5,G);
  init_cell(3,2,6,G);
  init_cell(6,2,8,G);
  init_cell(4,2,9,G);
  init_cell(8,3,1,G);
  init_cell(4,3,3,G);
  init_cell(9,3,5,G);
  init_cell(2,4,1,G);
  init_cell(8,4,3,G);
  init_cell(9,4,4,G);
  init_cell(5,4,5,G);
  init_cell(1,4,7,G);
  init_cell(3,4,8,G);
  init_cell(6,4,9,G);
  init_cell(6,5,1,G);
  init_cell(2,5,4,G);
  init_cell(4,5,6,G);
  init_cell(5,5,8,G);
  init_cell(7,5,9,G);
  init_cell(9,6,2,G);
  init_cell(4,6,8,G);
  init_cell(5,7,1,G);
  init_cell(6,7,2,G);
  init_cell(7,7,3,G);
  init_cell(9,7,6,G);
  init_cell(1,7,8,G);
```

```
  init_cell(2,7,9,G);
  init_cell(1,8,2,G);
  init_cell(4,9,4,G);
  init_cell(6,9,6,G);
  init_cell(3,9,9,G);
  for (int i=0;i<100;i++) {iterate(G);}

  printGraph(G);

//  main_vect();    //lancia programma di test (vBigDSP) che esegue l'FFT

  return;
}
```

7.5 Test dinamico AltiVec - Mandelbrot

```
////////////////////////////////////////////////////////
/////// Calcolo del frattale di Mandelbrot    ///////////
/////// con n=2: sfrutta Altivec e salva  ///////////
/////// i risultati in una location di memoria  ///////////
/////// che puo' cambiare ogni loop      ///////////
////////////////////////////////////////////////////////

  #include <stdio.h>
  #include <stdlib.h>
  #include <stdbool.h>
  #include "init_mmu_cache.c"



  #define ITER_BLACK 0xFFFFFFFF


  unsigned int* memUnaligned_;
  // Buffer pointer aligned
  unsigned int* mem_;

unsigned int maxi_ = 40;
double ax_ = -2.0f;
double ay_ = -1.5f;
double ex_ =  1.0f;
double ey_ =  1.5f;
double sx_, sy_;
double four = 4.0f;
int y;
int npixel;

unsigned int* rowBuffer;

int w_; // Window width
int h_; // Window height

bool doublebuf_ = false;
double ax0_, ay0_, ex0_, ey0_;



  void main() {

  //__init_hardware();
  //__init_user();

  mmu_cache_init();   //inizializza la cache nelle stesse modalitÃ  previste dai test
       statici

  w_ = 131;          //pixel da calcolare, asse x
  h_ = 131;          //pixel da calcolare, asse y
```

```
maxi_ = 50000;
ax_ = -2.0f;
ay_ = -1.5f;
ex_ =  1.0f;
ey_ =  1.5f;
sx_ = (ex_ - ax_) / ((double) w_);
sy_ = (ey_ - ay_) / ((double) h_);
four = 4.0f;


int wx = ((w_+15)&0xfffff0);
int hx = ((h_+15)&0xfffff0);
npixel = wx*hx;
memUnaligned_ = (unsigned int*) malloc((npixel+16) * sizeof(unsigned int));    //
     allocazione della memoria
mem_ = (unsigned int*) ((((unsigned long)memUnaligned_) + 15) & (~0xf));       //
     allineamento (non serve se si definisce
                                                          //dalle impostazioni del
                                                              compilatore,
                                                         //necessario per altivec
printf("Run started, please be patient");

if (memUnaligned_ == NULL)  {
  printf("ERROR: Failed to allocate pixel buffer\n");
}




for (y=0; y<h_; y++) {

int row = y;                  //una riga alla volta
unsigned int maxi = maxi_;
rowBuffer = &mem_[row*w_];



// C row calculation routine
// Calc vars
//double _cx = ax_;
float _cx = (float)ax_;
float _cy = (float)(ay_ + sy_*((float)row));
float _sx = (float)sy_;


float __attribute__ ((aligned(16))) cxs[] = {(float)_cx, (float)_cx, (float)_cx, (
     float)_cx};
float __attribute__ ((aligned(16))) cys[] = {(float)_cy, (float)_cy, (float)_cy, (
     float)_cy};
float __attribute__ ((aligned(16))) coeffs[] = {0, (float)_sx, 2*((float)_sx), 3* ((
     float)_sx)};
float __attribute__ ((aligned(16))) qsx[] = {4*((float)sx_), 4*((float)sx_), 4* ((
     float)sx_), 4*((float)sx_)};
float __attribute__ ((aligned(16))) zeros[] = {0.0,0.0,0.0,0.0};
float __attribute__ ((aligned(16))) four[] = {4.0,4.0,4.0,4.0};
unsigned int __attribute__ ((aligned(16))) ones[] = {1,1,1,1};
unsigned int __attribute__ ((aligned(16))) iter_black[] = {ITER_BLACK, ITER_BLACK,
     ITER_BLACK, ITER_BLACK};
unsigned int __attribute__ ((aligned(16))) iter_max[] = {maxi, maxi, maxi, maxi};



asm volatile (
              ".align 8                              \n"
              "        vxor v0, v0, v0                 \n"        //caricamento dei
                   registri
              "        lvx v1, 0, %[cxs]               \n" // cx
              "        lvx v2, 0, %[cys]               \n" // cy
```

```
                "          lvx v3, 0, %[coeffs]          \n"
                "          vaddfp v1, v1, v3             \n"
                "          lvx v4, 0, %[four]            \n"
                "          lvx v6, 0, %[qsx]             \n"
                "          lvx v7, 0, %[iter_black]      \n"
                "          lvx v15, 0, %[iter_max]       \n"
                "                                        \n"
                "NxtPixA:                                 \n"
                "          mtctr %[maxi]                 \n"
                "          vor v8, v1, v1                \n" // zx
                "          vor v9, v2, v2                \n" // zy
                "          vxor v20, v20, v20            \n" // Quad iters counter.
                "          lvx v21, 0, %[ones]           \n" // Quad iters incrementers.
                "NxtIA:                                   \n"
                "          vmaddfp v10, v8, v8, v0       \n" // zx2
                "          vmaddfp v11, v9, v9, v0       \n" // zy2
                "          vmaddfp v9, v8, v9, v0        \n" // zy_ = zx * zy
                "          vaddfp v12, v10, v11          \n" // modulo2 = zx2 + zy2
                "          vsubfp v8, v10, v11           \n" // zx = zx2 - zy2
                "          vaddfp v9, v9, v9             \n" // zy = 2 * zx * zy
                "          vcmpgtfp. v13, v4, v12         \n" // Set iter mask (if all
                    elements are in bailout then CR6[2] is set and the quad is done).
                "          beq- cr6, DonePixA            \n"
                "          vand v21, v21, v13            \n" // Mask the incrementers.
                "          vadduwm v20, v20, v21         \n" // Inc quad iters.
                "          vaddfp v8, v8, v1             \n" // zx = zx + cx
                "          vaddfp v9, v9, v2             \n" // zy = zy + cy
                "          bdnz+ NxtIA                   \n"
                "                                        \n"
                "DonePixA:                                \n"
                "          vcmpequw v13, v20, v15        \n"
                "          vandc v20, v20, v13           \n"
                "          vand  v21, v7, v13            \n"
                "          vor v20, v20, v21             \n"
                "          stvx v20, 0, %[rowBuffer]     \n"
                "          vaddfp v1, v1, v6             \n" // Step to next quad pixels.
                "          add %[rowBuffer], %[rowBuffer], %[sixteen]\n"
                "          sub. %[x], %[x], %[one]       \n"
                "          bne+ NxtPixA                  \n"
                : /* output */     //"=&r" (oldval), "=&r" (tmp), "=m" (*once_control)
                : /* input */
                [cxs]         "r" (cxs),
                [cys]         "r" (cys),
                [coeffs]      "r" (coeffs),
                [qsx]         "r" (qsx),
                [zeros]       "r" (zeros),
                [ones]        "r" (ones),
                [one]         "r" (1),
                [four]        "r" (four),
                [rowBuffer]   "r" (rowBuffer),
                [x]           "r" (w_/4),
                [maxi]        "r" (maxi),
                [iter_black]  "r" (iter_black),
                [iter_max]    "r" (iter_max),
                [sixteen]     "r" (16)
                : /* clobbered */ "cr0", "cr6", "v1", "v2", "v3", "v4", "v5", "v6", "
                    v8", "v9", "v10", "v11", "v12", "v13", "v15", "v20", "v21"
                );


    }



// Stampa sulla console una versione del frattale in ASCII art (forse Ã¨ meglio non
    metterla nella tesi :) )
```

```
/*  for (y=0;y<h_;y++)
  {
    for (int q=0;q<w_;q++)
    {
      if (mem_[y*w_+q] >2)
      {
        if (mem_[y*w_+q] >10)
        {
          if (mem_[y*w_+q] >100)
        {
          printf("#");
        }

        else printf("*");

        }

        else   printf(".");
      }
      else printf(" ");
    }
    printf("\n");
  }

*/
  while (1)
  {

  }
}
```

Bibliografia

[1] W. Hess, *The Radiation Belt and Magnetosphere*, 1968.

[2] F. Irom, F. Farmanesh, and C. Kouba, "Single-event upset and scaling trends in new generation of the commercial SOI PowerPC microprocessors," *IEEE Transactions on Nuclear Science*, vol. 53, no. 6, pp. 3563–3568, 2006.

[3] "History Of Hamming Codes." [Online]. Available: http://biobio.loc.edu/chu/web/Courses/Cosi460/hamming _ codes.htm

[4] D. Lambert, X. Vega, D. Alexandrescu, and B. Azais, "Radiation Effects in the PowerPC7448 Microprocessor," in *RADECS 2010*. IEEE.

[5] J. L. Hennessy and D. A. Patterson, *Computer Architecture - A Quantitative Approach (4. ed.)*. Morgan Kaufmann, 2007.

[6] J. M. Rabaey, A. Chandrakasan, and B. Nikolic, *Circuiti Integrati Digitali - L'ottica del progettista*. Pearson Prentice Hall, 2005.

[7] P. Peronnard, R. Ecoffet, M. Pignol, D. Bellin, and R. Velazco, "Predicting the SEU error rate through fault injection for a complex microprocessor," in *Industrial Electronics, 2008. ISIE 2008. IEEE International Symposium on*, 30 2008-july 2 2008, pp. 2288 –2292.

[8] S. Guertin and F. Irom, "Recent Results for PowerPC Processor and Bridge Chip Testing," in *Radiation Effects Data Workshop (REDW), 2010 IEEE*. IEEE, pp. 8–8.

[9] G. Clemente, F. Filira, and M. Moro, *Sistemi Operativi - Architettura e Programmazione concorrente*. Edizioni Libreria Progetto Padova, 2011.

[10] R. Koga, W. A. Kolasinski, M. T. Marra, and W. A. Hanna, "Techniques of Microprocessor Testing and SEU-Rate Prediction," *Nuclear Science, IEEE Transactions on*, vol. 32, no. 6, pp. 4219 –4224, dec. 1985.

[11] E. W. Weisstein, "Mandelbrot set." [Online]. Available: http://mathworld.wolfram.com/MandelbrotSet.html

[12] G. Buchholz, "Computer language benchmarks game." [Online]. Available: http://shootout.alioth.debian.org/u32/performance.php?test=mandelbrot

[13] L. Lamport, *LaTeX: A Document Preparation System*, 2nd ed. Addison Wesley Professional, June 1994, iSBN: 0-201-52983-1.

[14] *MPC7448 RISC microprocessor hardware specifications.* Freescale, MPC7448EC Rev.4, 3/2007.

[15] "POWER to the People: A History of Chipmaking at IBM." [Online]. Available: http://www.devx.com/ibm/Article/20944

[16] "Introduction to assembly on the PowerPC." [Online]. Available: http://www.ibm.com/developerworks/library/l-ppc/

[17] "gcc PowerPC Assembly Quick Reference (Cheat Sheet)." [Online]. Available: http://www.cs.uaf.edu/2005/fall/cs301/support/ppc/index.html

[18] "Assembler Instructions with C Expression Operands." [Online]. Available: http://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html

[19] "Assembly Language Programming." [Online]. Available: http://users.ece.utexas.edu/ ∼ valvano/embed/chap12/chap12.htm

[20] D. Evans, "Balance of Power: Introducing PowerPC Assembly Language." [Online]. Available: http://www.mactech.com/articles/develop/issue _ 21/21balance.html

[21] hbr, "Assembly Primer Series - For SPU, PPC and ARM." [Online]. Available: http://brnz.org/hbr/?page _ id=737

[22] J. Rentzsch, "Save your code from meltdown using PowerPC atomic instructions." [Online]. Available: http://www.ibm.com/developerworks/library/pa-atom/