POLITECNICO DI MILANO
Dipartimento di Elettronica e Informazione
V FACOLTÀ DI INGEGNERIA - CORSO DI LAUREA IN INGEGNERIA INFORMATICA

Simple Power Analysis a livello di Instruction Set Architecture: modelli, metodi e strumenti

Tesi di Laurea di
Patrick BELLASI
Matr. 640559

Relatore: Chiar.mo Prof. **William FORNACIARI**
Correlatore: Prof. **Carlo BRANDOLESE**

Anno accademico 2004-2005

Ai miei genitori

Indice

Introduzione

"Il mio nome è Linus,
e sono il vostro Dio!"
Linus Torvalds

La sempre maggiore diffusione di sistemi portatili, come cellulari, laptop e palmari, unita all'introduzione di nuove tecnologie per la gestione delle informazioni personali e l'autenticazione degli utenti hanno contribuito alla nascita di nuove esigenze e problematiche. Due di queste esigenze sono il contenimento dei consumi energetici, al fine di prolungare l'autonomia dei dispositivi aumentandone la portabilità, e la disponibilità di sistemi di autenticazione e sicurezza sufficientemente robusti. Per queste ragioni molti ricercatori hanno convogliato i loro sforzi in due direzioni:

- la definizione di nuovi modelli, il più possibile accurati ed efficienti, per la descrizione e l'analisi del comportamento energetico di processori, istruzioni e memorie
- l'introduzione di schemi di cifratura più robusti, motivati dalla crescente potenza di calcolo disponibile nei moderni sistemi di elaborazione

Questo lavoro di Tesi si inserisce in un contesto che sta in mezzo a questi due campi di ricerca cercando di sfruttare i risultati raggiunti nel primo al fine di supportare gli studi del secondo. Più precisamente l'obbiettivo che si pone è quello di proporre un nuovo approccio per la valutazione delle vulnerabilità dei sistemi basato su alcuni interessanti risultati raggiunti nel campo della previsione dei consumi energetici dei sistemi di elaborazione.

Inquadramento della problematica. La sicurezza dell'informazione è una tematica piuttosto delicata soprattutto quando è necessario proteggere dati strettamente personali. Gli schemi di cifratura basano generalmente la loro sicurezza sul fatto che, nonostante l'elevata potenza di calcolo oggigiorno disponibile, servirebbero comunque tempi di elaborazione proibitivi per risalire alla chiave di cifratura operando "per tentativi".

Negli ultimi anni sono però state introdotte nuove tecniche di criptoanalisi che, sfruttando le cosiddette "side-channel information", consentono di ridurre drasticamente i tempi di cracking di una chiave di cifratura nel caso in cui l'implementazione dell'algoritmo crittografico non sia stata opportunamente studiata. Queste tecniche si basano ad esempio sulla possibilità di estrarre delle informazioni dall'osservazione del tracciato di potenza assorbita dal processore mentre quest'ultimo esegue le operazioni di cifratura. I dati raccolti possono essere successivamente rielaborati al fine di evidenziarne eventuali correlazioni con il valore di alcune variabili dell'algoritmo eseguito. In questo modo si riesce generalmente a ridurre drasticamente lo spazio dei possibili valori per un eventuale successivo attacco a forza bruta.

Il problema che sorge quindi è quello di riuscire a produrre, e prima ancora possibilmente a progettare, dispositivi di sicurezza che siano sufficientemente robusti rispetto a questa nuova tipologia di attacchi.

L'approccio attuale. Tecniche come la *Simple Power Analysis* (SPA) e la *Differential Power Analysis* (DPA) sono note ormai da anni come strumenti sia per il cracking di sistemi esistenti che per lo studio della robustezza di quelli nuovi. Attualmente però l'applicazione di queste tecniche presenta diversi svantaggi, principalmente: la necessità di disporre di sofisticate e costose attrezzature e la possibilità di applicare tali tecniche solamente su sistemi già realizzati. Quest'ultimo aspetto, in particolare, è piuttosto limitante in quanto non consente di verificare la robustezza di un sistema già in fase di progetto. In questo caso infatti è necessaria quanto meno una prototipizzazione del dispositivo sul quale effetture le rilevazioni necessarie all'analisi.

Nuove prospettive. Al fine di superare queste limitazioni, facendo dell'analisi di potenza uno strumento utilizzabile già nelle primissime fasi della progettazione di un nuovo dispositivo, una possibile soluzione è rappresentata dalla possibilità di generare i tracciati di potenza mediante *simulazione*. In questo caso bisogna però considerare il fatto che esistono diversi *livelli di astrazione* ai quali operare la simulazione di una architettura. Un livello troppo basso, pur garantendo le migliori previsioni di assorbimento, richiede tipicamente lunghissimi tempi di simulazione che ne rendono di fatto proibitivo l'impiego ai fini pratici dell'analisi di potenza. Operando invece ad un livello troppo alto il rischio che si corre è quello di non riuscire ad ottenere le informazioni necessarie all'analisi.

Una nuova metodologia. L'approccio proposto da questo lavoro di Tesi fa ricorso ad un *simulatore comportamentale* sviluppato a supporto di una nuova metodologia per la caratterizzazione in potenza degli instruction set. Precisamente il simulatore si occupa di determinare le tempistiche esatte di permanenza di ogni istruzione di un programma all'interno del processore. Il modello per il quale è stato sviluppato mira a determinare il CPI[1] come prodotto fra il livello reale di parallelismo, che l'architettura riesce a sfruttare, ed il tempo nominale d'esecuzione sommato al ritardo dovuto alle interazioni intra-istruzione.

Quello che interessa ai nostri scopi è invece un tracciato dell'assorbimento d'energia del processore, per ogni ciclo di clock, mentre esegue un algoritmo. A tal fine abbiamo proposto un *nuovo modello* per la caratterizzazione energetica dei singoli componenti della pipeline di un generico processore. Tale modello, partendo da una caratterizzazione statica dell'instruction set, stima l'assorbimento energetico medio di ogni singola istruzione in funzione delle unità della pipeline che mantiene impegnate durante un ciclo di clock. Una volta nota questa stima è possibile tracciare gli assorbimenti di un qualunque algoritmo, eseguito sull'architettura considerata, ottenendo in uscita un tracciato degli assorbimenti energetici previsti per ogni singolo ciclo di clock.

Una *metodologia* guida nella corretta applicazione del nuovo modello:

1. *tuning del modello* - in questa fase si passa dalla caratterizzazione statica a livello di instruction set alla caratterizzazione delle singole componenti della pipeline. Affinché ciò sia possibile è necessario produrre una traccia d'esecuzione per il tuning con cui generare le informazioni necessarie alla determinazione dei parametri del nuovo modello. Questa operazione dev'essere eseguita un'unica volta per ciascuna nuova architettura che si intende supportare.

[1] *Clock Cycle per Instruction*, si rimanda all'Appendice B per maggiori dettagli.

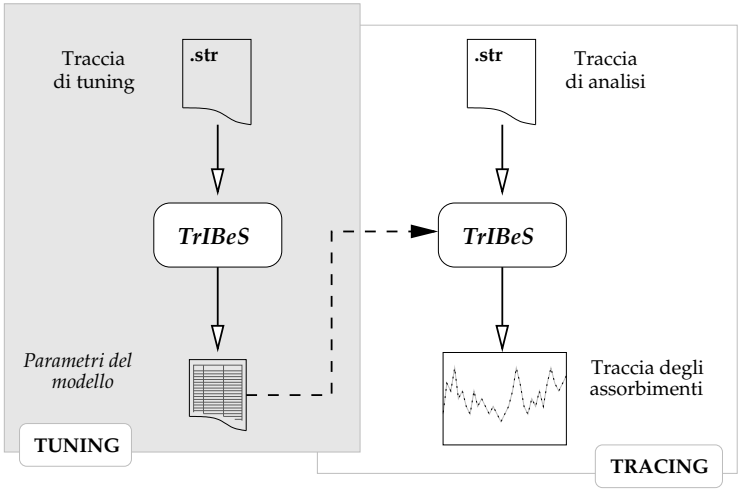


Figura 1: Schema generale per l'utilizzo del nuovo modello di tracciamento dei consumi.

2. *analisi dell'algoritmo* - una volta calibrato il modello nella prima fase, qualunque programma può essere simulato al fine di generare la traccia degli assorbimenti energetici.

In Figura 1 viene mostrato il flusso completo descritto dalla metodologia appena illustrata.

Il lavoro di Tesi si è quindi concretizzato anzitutto nella realizzazione di una estensione del simulatore TrIBeS al fine di supportare le diverse fasi della metodologia indicata. Successivamente è stato introdotto il completo supporto per una nuova architettura: l'$ARM7TDMI$, usata nel resto della Tesi come architettura di riferimento. Infine la metodologia indicata sopra è stata applicata alla nuova architettura al duplice fine di validare il modello e dimostrare la sua usabilità nell'ambito della *Simple Power Analysis*.

La Tesi è stata organizzata nei seguenti capitoli:

Sicurezza dei sistemi di cifratura
Presenta una breve introduzione alla sicurezza delle informazioni. L'attenzione si focalizza su due aspetti principali: i sistemi di cifratura e le metodologie adottate al fine di ottenere informazioni segrete in modo fraudolento.

Metodologie per la stima degli assorbimenti
Illustra il modello per la stima degli assorbimenti da cui prende spunto questo stesso lavoro di Tesi. In particolare viene ripresa e commentata la

metodologia di stima basata sul concetto di funzionalità e la sua utilità viene inquadrata nel contesto dei sistemi per l'ottimizzazione dei consumi di potenza.

Modello per la generazione di tracce di potenza assorbita
Dopo avere inquadrato nei due capitoli precedenti la tematica di interesse e gli strumenti di cui disponiamo inizialmente, questo capitolo introduce la proposta di un nuovo modello per il tracciamento della potenza assorbita per singolo ciclo di clock. Il modello mostrato si basa sull'idea di utilizzare un'opportuna estensione del simulatore comportamentale TrIBeS al fine di generare i dati necessari all'analisi di vulnerabilità.

Simulatore software per l'analisi degli attacchi in potenza
Sono presentate in questo capitolo le modifiche apportate al simulatore comportamentale TrIBeS, al fine di generare un tracciato della potenza assorbita per singolo clock cycle, implementando il modello descritto nel capitolo precedente.

Risultati sperimentali
Quanto visto, a livello teorico-implementativo, nei capitoli precedenti viene messo in pratica in quest'ultimo. L'obbiettivo principale è quello di verificare la validità della soluzione proposta e mostrarne le possibilità d'impiego.

Di quelli elencati, gli ultimi tre sono i capitoli che rappresentano il contenuto originale della Tesi. A conclusione del lavoro è riportato poi un breve commento sui risultati ottenuti e sui possibili sviluppo futuri in questo campo.

CAPITOLO 1

Sicurezza dei sistemi di cifratura

"*Non ci sono sicurezze su questa terra,*
solo opportunità"
Generale Douglas MacArthur

QUESTO capitolo presenta una breve introduzione alla sicurezza delle informazioni. L'attenzione si focalizza su due aspetti principali: i sistemi di cifratura e le metodologie disponibili al fine di ottenere informazioni, eventualmente segrete, in modo fraudolento.

Una breve introduzione richiama il concetto di cifratura delle informazioni. Cosa significa e quali sono gli obbiettivi viene spiegato con un classico esempio applicativo.

Successivamente vengono presentate diverse implementazioni di sistemi crittografici, cercando di delineare per ciascuno i principali aspetti positivi e negativi.

Infine verrà definito il concetto di *"Side Channel Informations"* e discussi i diversi tentativi noti in letteratura allo scopo di sfruttare tali informazioni.

Figura 1.1: Esempio di comunicazione a rischio fra Alice e Bob.

1.1 Introduzione

Da sempre l'uomo ha avuto la necessità di comunicare. Lo scambio di informazioni a volte avviene in contesti nei quali è necessario che gli unici fruitori del messaggio siano il mittente ed il suo destinatario. Altre volte è necessario garantire l'identità delle parti coinvolte; o ancora che il messaggio stesso passi dal mittente al destinatario senza subire variazioni.

1.1.1 Il problema

L'avvento delle nuove tecnologie ha reso sempre più comune l'uso di una forma di comunicazione "digitale". Le informazioni vengono scambiate sempre più frequentemente usando dispositivi elettronici o sistemi informatici: il massiccio uso della posta elettronica ne è un esempio.

Questi sistemi, per loro natura, prevedono spesso che il messaggio inviato possa essere letto da più persone oltre al legittimo destinatario. In taluni contesti questa situazione può essere indesiderata. Classico è l'esempio mostrato in figura Figura 1.1.
Alice vuole inviare un messaggio *confidenziale* a Bob, alcune delle cose indesiderabili che possono capitare sono:

- ❏ Evil intercetta e legge il messaggio
- ❏ Evil modifica il contenuto del messaggio prima che giunga a Bob

Figura 1.2: Esempio di comunicazione sicura fra Alice e Bob.

Un'altra problematica frequente in queste situazioni è garantire l'identità del mittente. Come può Bob esser sicuro che in messaggio inviato sia stato scritto effettivamente da Alice?

1.1.2 La soluzione

Alle problematiche illustrate precedentemente si cerca una soluzione facendo ricorso all'uso della cifratura delle informazioni[1].

In questo schema di comunicazione, come mostrato in figura Figura 1.2, Alice e Bob utilizzano una *chiave di cifratura*. La chiave consente loro di comunicare scambiandosi messaggi in una forma non comprensibile per chi è estraneo alla comunicazione. Quando Alice deve inviare un messaggio (detto *plaintext*) a Bob, usa la sua chiave segreta per *cifrare* il messaggio. Il messaggio così "offuscato" (detto *chipertext*) può ora essere inviato tranquillamente al suo destinatario, sicuri che chiunque eventualmente lo intercetti non sarà in grado di comprenderne i contenuti. Quando Bob riceve il messaggio lo riporta in una forma intellegibile usando la propria chiave di decifratura.

Gli *schemi di cifratura*[2] possono suddividersi grossolanamente in due categorie:

[1] Un buon testo di riferimento sull'argomento è [1]

[2] Per schema intendiamo la procedura algoritmica che consente di passare dal *plaintext* al *chipertext* e viceversa

- *simmetrici* - la chiave usata per la decifratura è uguale a quella utilizzata per la cifratura.
- *asimmetrici* - la chiave usata per la decifratura è diversa da quella usata per la cifratura. In questi schemi per ogni chiave di cifratura esiste una ed una sola corrispettiva chiave di decifratura[3].

Diciamo *algoritmo di cifratura* quel procedimento che ci consente di ottenere il *chipertext* a partire dal *plaintext* data una chiave. Stabilito uno schema di cifratura esistono diversi possibili algoritmi di cifratura utilizzabili.
Tutti gli algoritmi di cifratura si basano però su un fondamentale principio:

> *La sicurezza di un algoritmo di cifratura si deve basare esclusivamente sulla* segretezza della sua chiave *e non sulla segretezza del procedimento utilizzato per produrre il chipertext a partire dal plaintext usando una data chiave.*

1.2 Sistemi di cifratura

L'uso della cifratura delle informazioni è sempre più presente nella nostra vita. La crittografia è alla base di molte delle tecnologie che usiamo quotidianamente. Il commercio elettronico e la posta elettronica sono due delle tecnologie in cui più facilmente traspare l'uso di questi sistemi. Meno evidente può risultare l'uso che ne facciamo nei dispositivi portatili quali , cellulari, smartphone, ricetrasmettitori ed altri sistemi embedded in generale. La televisione digitale ricorre da anni all'impiego di smart card per la decodifica dei canali a pagamento. L'uso di documenti digitali[4], che contengono diverse informazioni sull'individuo, sarà ben presto una realtà così come lo è da anni quello di bancomat e carte di credito.

[3] Nel caso ciò si dimostri non essere vero si dice che si è trovata una *collisione* e lo schema di cifratura non viene più considerato sicuro.

[4] Come la carta d'identità digitale

	Economicità	Prestazioni	Sicurezza	Costo
Hardware	✗	✓	✓	✗
Software	✓	✓	✗	✓
Shared Load	✓	✓/✗	✓	✓

Figura 1.3: *Confronto fra le diverse implementazioni.*

1.2.1 Implementazioni

I sistemi di cifratura sono realizzati mediante diverse tecnologie in funzione del contesto in cui devono essere applicati. Una prima classificazione potrebbe essere:

- *hardware* - la cifratura è demandata ad una logica più o meno complessa implementata direttamente in hardware.
- *software* - la cifratura è effettuata eseguendo un algoritmo su un processore di tipo general-purpose.
- *shared load* - un insieme di operazioni comuni a diversi sistemi di cifratura sono implementate in hardware. Successivamente si specializza il sistema scrivendo il software per la particolare applicazione di interesse.

Tradizionalmente, anche se caratterizzata da costi elevati, l'implementazione hardware è stata la più diffusa in quanto garantisce alte prestazioni. L'avvento di *Internet* e delle nuove tecnologie che gli ruotano attorno, come i protocolli , e , ha portato ad implementazioni software nelle quali si riescono a contenere drasticamente i costi, anche se a discapito delle prestazioni.

Una nuova categoria di prodotti, cosiddetti *shared load*, è emersa negli ultimi due decenni. Hardware e software collaborano al fine di creare un efficiente e quanto più economico sistema di cifratura. Le smart card crittografiche sono un esempio di questa tecnologia. Disponibili da tempo, a costi che si aggirano intorno a qualche decina di Euro, integrano la logica necessaria per la cifratura a chiave pubblica e rappresentano una valida soluzione per la memorizzazione e gestione delle informazioni. Le loro prestazioni non sono paragonabili a quelle di una CPU ma questo non è in generale un problema. In un sistema *shared load*: il protocollo di cifratura è implementato in software nel processore host, mentre le operazioni critiche, che coinvolgono l'uso della chiave privata dell'utente, sono demandate alla smart card. In Tabella 1.3 sono riassunte queste considerazioni.

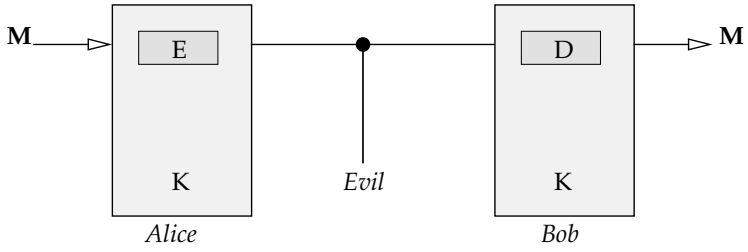


Figura 1.4: Modello crittografico tradizionale.

1.3 Side channel information

Gli algoritmi di cifratura, tradizionalmente, sono sempre stati progettati ed analizzati su basi teoriche, trattandoli come "semplici" oggetti matematici. Il modello sul quale era basata l'analisi di sicurezza di uno schema di cifratura era quello a *scatola nera*.

Come mostrato in Figura 1.4, un algoritmo interno mappa ogni messaggio $\boldsymbol{M}$ in ingresso nel corrispondente *chipertext*, in modo parametrico rispetto ad una chiave segreta $\boldsymbol{K}$.

Sfortunatamente, la descrizione di uno *schema di cifratura*[5] come ideale oggetto matematico si scontra con una serie di problematiche: quando viene implementato nel mondo reale il *chiper* interagisce con l'ambiente esterno e da esso viene influenzato.

1.3.1 Nuovo modello crittografico

Nel semplice modello presentato precedentemente, Evil ha la possibilità di ottenere informazioni addizionali sul *chiper* monitorando informazioni come la potenza assorbita, le radiazioni elettromagnetiche o il tempo di esecuzione. Evil può anche tentare di influenzare il comportamento del chiper agendo su variabili come la temperatura, la tensione di alimentazione o ancora il segnale di clock. Un migliore modello per lo studio della sicurezza di una sistema di cifratura deve tener conto di questi fattori così come mostrato in Figura 1.5.

La relativa semplicità con la quale le informazioni aggiuntive possono essere acquisite, e soprattutto la possibilità di individuare delle correlazioni con i bit della chiave segreta, hanno consentito la nascita di una nuova e rivoluzionaria generazione di attacchi, detti: ***side channel attacks***.

[5] di seguito *chiper*

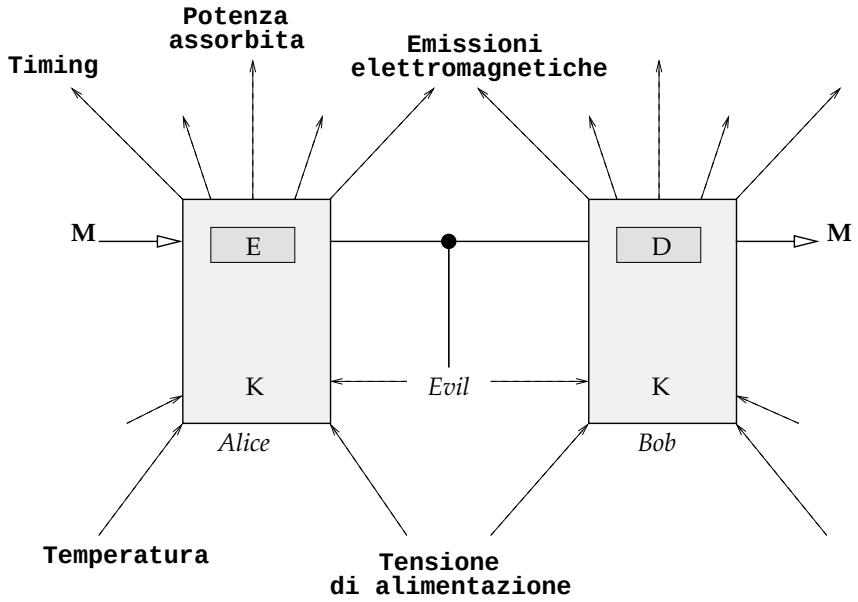


Figura 1.5: Modello crittografico migliorato.

1.4 Attacchi Side Channel

Riportiamo di seguito una breve descrizione dei diversi metodi di *side-channel criptoanalisi* noti in letteratura. L'obiettivo è quello di mettere in evidenza come le informazioni collaterali illustrate nella Sez. 1.3 possano essere sfruttate al fine di "bucare" un sistema crittografico.

Questa tipologia di attacchi è stata sviluppata nella seconda metà degli anni '90. Da allora è stata impiegata con successo in numerosi tentativi di forzare diversi algoritmi di cifratura, sia a chiave pubblica che simmetrici. Un buon riassunto sulla tipologia degli attacchi side channel si trova in [2] e [3]. Articoli più dettagliati verranno citati nelle sezioni seguenti.

1.4.1 Timing analysis (TA)

Il tempo di esecuzione è da sempre una delle cifre di merito nella progettazione dei sistemi di calcolo. Nell'ambito dei sistemi crittografici, la ricerca della implementazione più veloce possibile è sempre stato uno degli obbiettivi di progetto. In questa visione tuttavia si trascura il fatto che il tempo di esecuzione può essere un veicolo di informazioni preziose in merito a cosa l'algoritmo sta facendo. Questa idea fu introdotta per la prima volta da Kocker [4] nel 1995. I crittografi compresero per la prima volta che molti degli algoritmi di cifratura allora in

circolazione erano vulnerabili a questa nuova tipologia di attacchi; il New York Times fece addirittura una notizia in prima pagina [5].

L'idea alla base dell'attacco è che un sistema crittografico può impiegare una quantità di tempo diversa nella elaborazione di diversi input. Ciò può derivare da diversi fattori:

- ottimizzazioni hardware al fine di evitare operazioni non necessarie
- salti o istruzioni condizionali che possono portare a diversi percorsi di esecuzione
- caching della memoria
- diverso tempo di esecuzione delle istruzioni macchina (come addizioni o moltiplicazioni)

In definitiva: informazioni in merito al *plaintext* possono essere desunte osservando i tempi di esecuzione delle istruzioni del sistema crittografico attaccato. Sebbene potrebbe sembrare che poche informazioni utili possano trasparire da questo canale (per esempio la distanza di Hamming[6] della chiave), è stato dimostrato [4] che esistono attacchi in grado di recuperare per intero una chiave segreta di cifratura.

1.4.2 Fault analysis (FA)

La progettazione di sistemi "robusti", che difficilmente si comportano in modo imprevisto e che eventualmente siano in grado di porre rimedio al verificarsi di situazioni inaspettate, da sempre impegna i progettisti. Tuttavia, piccoli difetti sono spesso considerati come accettabili e quindi tollerati.

Anche in questi casi ci si è accorti come un errore, apparentemente innocente, durante il funzionamento di un sistema di cifratura, può avere un forte impatto sulla sicurezza dello stesso.

Questa tipologia di attacchi è stata inizialmente proposta da Boneh in [6], e successivamente sviluppata da Biham in [7] ad altri ancora in [8] e [9]. L'attacco si basa sulla osservazione degli errori che si verificano nel corso di successive esecuzioni dell'algoritmo di crittografia in esame. Gli errori vengono maliziosamente provocati dall'attaccante in diversi modi; ad esempi iniettando dei picchi nella tensione di alimentazione, producendo irregolarità della frequenza di clock o ancora facendo ricorso a radiazioni di varia natura.

Un banale esempio di *fault analysis* è il seguente. Supponiamo di disporre di un sistema di cifratura in grado di inviare sia testo in chiaro che cifrato, in

[6] Il numero di bit a 1 nella rappresentazione binaria della chiave

funzione del valore di un particolare bit in un registro. Se questo bit venisse accidentalmente modificato il resto della trasmissione risulterebbe in chiaro.

Ancor più realistico è l'esempio seguente. Consideriamo una smart card che contenga una implementazione del . L'ambiente in cui opera la carta può essere influenzato dalla persona che la possiede. Supponiamo che l'attaccante sia in grado di produrre un errore su un singolo bit di R_{15} (la "semichiave" destra all'inizio del sedicesimo round)[7]. L'attaccante osserva quindi due operazioni di cifratura e conserva i risultati. La cifratura con la chiave corretta produce il chipertext C, mentre quella prodotta introducendo l'errore in un singolo bit della chiave genererà il chipertext $\hat{C}$. Come viene mostrato in [7] è possibile risalire da questi chipertext all'intera round key K_{16}, partendo da considerazioni basate sul fatto che i due chipertext sono stati generati a partire da un unico bit diverso nella R_{16}. Tale operazione richiede un piccolo numero di coppie $\langle C, \hat{C} \rangle$. Ottenuta la round key basta poi un classico attacco di tipo broute force, di complessità media $O(2^7)$, per risalire all'intera chiave di cifratura. Questo metodo di analisi in particolare è noto con il nome di *differential fault analysis* ed è stato proposto in [7] per il cracking effettivo di una implementazione del DES. Il lettore interessato può trovare approfondimenti in [7] e [3]; in [10] si ha invece la descrizione ufficiale del DES.

1.4.3 Electromagnetic emissions Analysis (EMA)

Da tempo si studia la possibilità di sfruttare le informazioni che derivano dalle emissioni elettromagnetiche, ma solo di recente sono stati studiati attacchi side channel che le sfruttino effettivamente. Nel 2002, ricercatori della IBM hanno pubblicato un articolo [11] in cui vengono dimostrati vari modi con i quali sfruttare le emissioni elettromagnetiche al fine di attaccare una smart card. Questo risultato faceva seguito al lavoro di Gandolfi et al. [12] e di Quisquater [13].

Le emissioni elettromagnetiche possono essere di varia natura:

- *emissioni dirette* - causate dal flusso di corrente nei circuiti.
- *emissioni indirette* - causate da accoppiamenti elettrici e elettromagnetici con componenti in prossimità del circuito. Possono manifestarsi come modulazioni di un segnale portante.

Gli autori di [11] mostrano con un esperimento come sia possibile risalire alle informazioni contenute in una smart card usando le emissioni dirette.

Alle analisi side channel viste nella breve carrellata precedente, si aggiunge la *PowerAnalysis*. Essendo questo tipo di analisi alla base di uno degli obiettivi che

[7] Si veda la Figura 1.11 e [10] per i dettagli implementativi del DES

ci si era posti con questo lavoro di tesi, le dedichiamo un maggiore dettagli nella sezione che segue.

1.5 Power analysis

I dispositivi elettronici in generale, e quindi anche i microprocessori, i microcontrollori e le smart card crittografiche, sono implementati usando reti logiche a semiconduttori. Quando un transistor[8] cambia stato è possibile osservare una *switching current* fluire fra l'alimentazione ed il riferimento di massa. Ciò significa che ogni volta che un processore compie una operazione, una data quantità di potenza viene assorbita. La quantità di potenza consumata dipende dai bits dell'OPCODE[9] dell'istruzione, dal valore degli operandi ed ancora dai bit manipolati nei registri, dal bus indirizzi e dati, dalla memoria, ...

La potenza assorbita da una smart card crittografica o da un microprocessore può essere misurata con relativa semplicità inserendo un piccolo resistore (e.g. 47Ω) in serie al riferimento di massa. Misurando le variazioni di tensione ai capi del resistore, ed usando la semplice relazione $I = \frac{V}{R}$, siamo in grado di determinare il flusso di corrente che attraversa il resistore e quindi anche la potenza consumata: $P = \frac{V^2}{R}$.

Campionando e memorizzando la tensione ai capi del resistore con uno oscilloscopio digitale abbiamo una serie di informazioni side-channel che vengono impiegate nell'ambito della *power analysis* al fine di "bucare" i sistemi crittografici. Queste tecniche sono state introdotto inizialmente da P. Kocher, J. Jaffe e B. Jun in [14]. Altri lavori interessanti sul tema si trovano in:

- Akkar et al. analizzano ulteriormente e formalizzano i modelli di power analysis in [15]
- Messerges et al. in [16] e Mayer-Sommer in [17] esaminano gli attacchi power analysis alle smart-card
- Chari et al. in [18], Messerges in [19], Shamir in [20] e Lash in [21] illustrano le applicazione della power analysis come strumento per attaccare l'

[8] Elemento alla base delle singole porte logiche

[9] *codice operativo*, la rappresentazione binaria in linguaggio macchina di ogni operazione

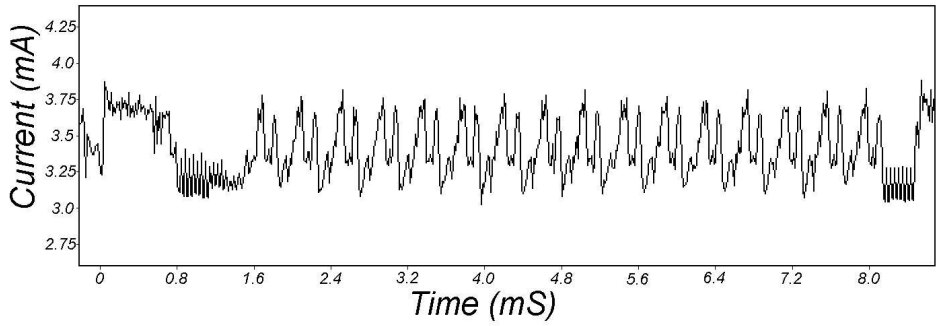


Figura 1.6: Andamento della corrente durante l'esecuzione del DES su una tipica smart card [14].

1.5.1 Simple power analysis

Con il termine *Simple Power Analysis*[10] ci si riferisce alla tecnica basata sulla diretta interpretazione delle tracce di consumo di potenza acquisite durante l'esecuzione di un algoritmo crittografico.

In assenza di adeguate contromisure il ricorso alla SPA può consentire l'identificazione delle operazioni eseguite dall'algoritmo. In alcuni casi può addirittura portare alla identificazione dei bit della chiave di cifratura.

In generale l'uso della sola SPA consente di mettere in evidenza certe caratteristiche macroscopiche dell'algoritmo eseguito. Ad esempio, se un algoritmo include un ciclo che viene eseguito n volte, siamo in grado di vedere nella traccia di potenza assorbita un pattern che si ripete per n volte. Allo stesso modo, se un algoritmo è caratterizzato da salti condizionali che portano a due insiemi di istruzioni con diverso assorbimento, l'esame della traccia di potenza ci consente di sapere quale ramo d'esecuzione è stato preso e quindi di estrarre informazioni sul valore della condizione.

La Figura 1.6 mostra una tipica traccia delle potenza dissipata da una smart card durante l'esecuzione del DES. In particolare, in assenza di particolari precauzioni a livello hardware o software, possiamo notare come le 16 iterazioni che caratterizzano l'algoritmo siano ben distinguibili.

Nella figura Figura 1.7 viene presentato un dettaglio della seconda e terza iterazione del DES relative alla stessa traccia della Figura 1.6. Maggiori dettagli sulle operazioni eseguite sono a tal punto visibili: i due registri di 28bit C e D vengono ruotati di un bit nella seconda iterazione e di due nella terza. Le frecce riportate in figura evidenziano queste operazioni e come siano facilmente visualizzabili nella traccia della corrente assorbita.

[10] di seguito SPA

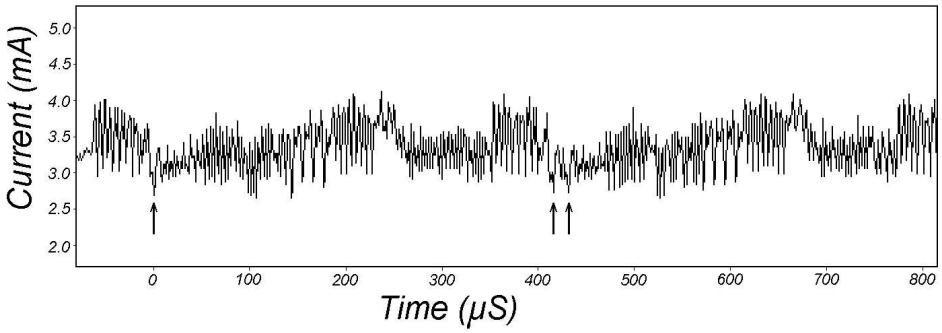


Figura 1.7: Dettaglio del consumo di potenza nella seconda e terza iterazione del DES [14].

Infine, in Figura 1.8 vengono mostrate, con ancor maggiore risoluzione, due tracce relative ad uno stesso blocco di codice del DES. Entrambe le tracce corrispondono a 7 cicli di clock e mostrano una evidente divergenza in corrispondenza del sesto ciclo. La variazione è dovuta alle presenza nel codice di una istruzione di salto condizionale. Nella prima traccia il salto viene preso mentre nella seconda ciò non avviene. Quest'ultimo esempio mostra chiaramente come, se la condizione di salto dipendesse unicamente da un bit nella chiave, la semplice analisi della traccia relativa all'assorbimento consentirebbe di svelare il valore di tale bit.

Come gli esempi precedenti mostrano, la SPA consente di *mettere in evidenza la dipendenza del flusso di controllo dal valore degli operandi*. Ciò si è dimostrato essere sufficiente per minare la sicurezza di diversi algoritmi di cifratura. Le operazioni che principalmente risultano vulnerabili ad attacchi di questo tipo sono:

- *confronti* - l'uguaglianza fra due valori compare spesso come condizione nelle istruzioni di salto.
- *moltiplicazione modulo n* - i circuiti che realizzano la moltiplicazione modulo n, soprattutto quelli implementati nelle architetture più semplici come le smart card, sono generalmente fonte di informazioni collaterali. Principalmente si rivelano vulnerabili ad attacchi basati su TA o SPA.
  Questa operazione è fondamentale in diversi degli attuali sistemi di crittografia a chiave pubblica, nei relativi algoritmi di *firma digital* e *key agreement*.
- *esponenziale modulo n* - in molti degli algoritmi a chiave pubblica (e.g. RAS, ElGamal), algoritmi di firma digitale (e.g. RAS, ElGamal, DSA) e protocolli di key agreement (e.g. Diffie-Hellman), la chiave privata *key* è utilizzata come

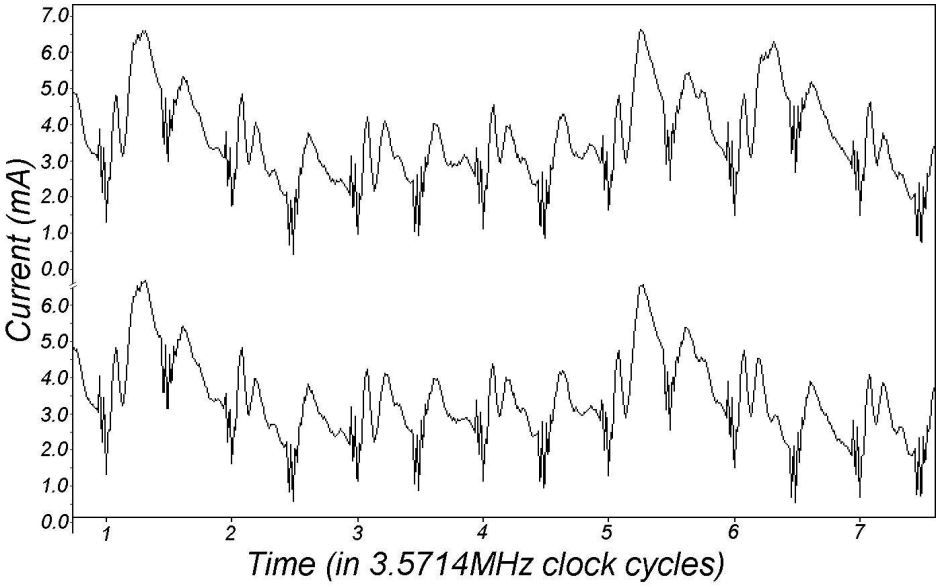


Figura 1.8: Dettaglio del consumo di potenza in due porzioni del DES. Entrambe si riferiscono a 7 cicli di clock [14].

esponente in operazioni del tipo: $c = p^{key} \ mod \ n$.
Un modo semplice per implementare tale operazione è riportato in Figura 1.9. L'esponente viene analizzato dal bit più significativo a quello meno significativo. Nel corso di ogni iterazione viene eseguita una operazione di elevamento al quadrato (*square*) più un'eventuale operazione di moltiplicazione (*multiply*).
Quest'ultima operazione, in particolare, viene eseguita *solo* nel caso in cui l'esponente abbia valore '1'. Ovviamente, nel caso in cui le due operazioni di moltiplicazione ed elevamento al quadrato siano caratterizzate da un diverso consumo di potenza, l'esponente può essere dedotto da una semplice analisi della traccia.

- *moltiplicazione scalare* - gli algoritmi di cifratura basati sulle curve ellittiche usano, in luogo dell'esponenziale modulo n, l'operazione di moltiplicazione di un punto y della curva per lo scalare *key*.
  L'algortimo considerato in tal caso viene detto *double-and-add* ed è mostrato in Figura 1.10. Anche in questo caso, l'informazione che rischia di esser compromessa da una semplice analisi della traccia di potenza, è lo scalare *key* che in taluni algoritmi[11] rappresenta proprio la chiave privata di

[11] come nel caso dell'ECDSA: versione su curve ellittiche del classico DSA. Si veda [22] per i

Ingressi: la base y; l'esponente $key = key_{len}key_{len-1}...key_1$ (si suppone che key_{len}, il bit più significativo di key, sia pari ad '1'; il modulo n
Uscite: $c = c^{key} \ mod \ n$

$c \leftarrow y$;
for *i che va da* $(len - 1)$ *a* 1 **do**
  $c \leftarrow c^{key} \ mod \ n$;
  if $x_i = 1$ **then**
    $c \leftarrow c * y \ mod \ n$;
  end
end

Figura 1.9: *Esempio di procedura "square-and-multiply".*

Ingressi: un punto y sulla *curva ellittica*; lo scalare $key = key_{len}key_{len-1}...key_1$ (si suppone che key_{len}, il bit più significativo di key, sia pari ad '1'; il modulo n
Uscite: il punto della curva $c = key \ y$

$c \leftarrow y$;
for *i che va da* $(len - 1)$ *a* 1 **do**
  $c \leftarrow 2 \ c$;
  if $x_i = 1$ **then**
    $c \leftarrow c + y$;
  end
end

Figura 1.10: *Esempio di procedura "double-and-add".*

cifratura.

In conclusione possiamo affermare che la simple power analysis può essere impiegata nella pratica al fine di evidenziare macro-caratteristiche degli algoritmi di cifratura e dei sistemi che la implementano. Queste caratteristiche possono essere utilizzate a fini di *reverse engineering* di implementazioni chiuse o ancora per identificare aree di interesse di algoritmi non noti, a supporto di analisi successive.

Tuttavia, si dimostra che la simple power analysis da sola non è sufficiente in generale per bucare la implementazioni commerciali degli algoritmi crittografici. Queste implementazioni sono infatti solitamente ottimizzate al fine di mascherare le caratteristiche degli algoritmi implementati.

dettagli.

1.5.2 Differential power analysis

Le istruzioni di un processore si distinguono per il loro consumo di potenza. Con la SPA siamo in grado di rilevare solo le variazioni di potenza dovute alla esecuzione di istruzioni caratterizzate da un diverso profilo di assorbimento. In una traccia di potenza ci sono tuttavia anche delle variazioni più contenute e difficili da individuare. Queste micro variazioni sono dovute ad esempio alla *dipendenza dai dati* delle istruzioni eseguite.

Un modello più raffinato del costo di una istruzione, in termini di potenza assorbita, potrebbe esser così espresso:

$$P_{assorb} = P_{instr} + P_{data\_dep}$$

In questo nuovo modello, ad un costo base dell'istruzione che è relativo alla sua tipologia, aggiungiamo un contributo di potenza che dipende unicamente dal valore dei suoi operandi. Questi piccoli contributi sono difficilmente identificabili, all'interno di una singola traccia di potenza, a causa degli errori di misura e del rumore che gli si sovrappone.

La *Differential Power Analysis*[12] consente di superare queste limitazioni. Raccogliendo un numero sufficientemente ampio di tracce di potenza, ed applicando opportune tecniche statistiche, è possibile sperare di riuscire a recuperare gran parte di queste informazioni. Il particolar tipo di analisi statistica richiesta dipende dall'algoritmo in esame. In generale però, quello che si cerca di fare è di amplificare le componenti del segnale effettivamente correlate col valore di uno specifico operando.

L'obiettivo principale di un attacco DPA è generalmente quello di recuperare la chiave segreta di una algoritmo di cifratura. Affinché l'attacco possa avere successo dev'essere verificata la seguente *ipotesi fondamentale*:

> *L'algoritmo di cifratura deve presentare una* variabile intermedia *v_h il cui valore sia esprimibile in funzione di un sottoinsieme K_D dei bit della chiave e del* plaintext *(o del* chipertext*)*

Un primo esempio di attacco DPA, dietro la verifica dell'ipotesi vista, è stato proposto da P. Kocher, J. Jaffe e B.Jun in [23] e [14].
Chiamando K la chiave di cifratura e v_h la particolare variabile soddisfacente all'ipotesi fondamentale, l'attacco si concretizza nei seguenti passi:

1. vengono acquisite N tracce del consumo do potenza:

[12] di seguito DPA

$$S_i[j] \qquad \text{con } 1 \leq i \leq N$$

relative ad altrettante esecuzioni dell'algoritmo di cifratura [13]. In aggiunta si memorizza, per ciascuna traccia, il corrispondente valore del chipertext c_i (plaintext p_i).

2. scelto un bit b di v_h, si costruisce la *funzione di selezione*:

$$D(\cdot) \longrightarrow \{0, 1\}$$

che consente di calcolare il valore di b a partire dal sottoinsieme K_D, di bit della chiave, e da c_i (p_i).

3. si ipotizza per i k bit di K_D uno dei possibili 2^k valori. A questo punto, mediante la $D(\cdot)$, è possibile determinare per ciascun c_i (p_i) il valore teorico di b e conseguentemente *partizionare* le N tracce $S_i[j]$ acquisite nei due insiemi:

$$S_0 = \{S_i[j] \mid D(\cdot) = 0\}$$
$$S_1 = \{S_i[j] \mid D(\cdot) = 1\}$$

4. per ciascuno dei due insiemi S_0 ed S_1 si determina la media delle tracce che ne fanno parte, rispettivamente:

$$A_0[j] = \left(\frac{1}{|S_0|}\right) \sum_{S_i[j] \in S_0} S_i[j]$$

$$A_1[j] = \left(\frac{1}{|S_1|}\right) \sum_{S_i[j] \in S_1} S_i[j]$$

dove abbiamo indicato con $|S_0|$ e $|S_1|$ la cardinalità di S_0 ed S_1 rispettivamente, tali che: $|S_0| + |S_1| = N$.

5. sottraendo fra loro i due segnali medi prima calcolati si ottiene infine la cosidetta *traccia differenziale* del consumo di potenza:

$$\Delta_D[j] = A_0[j] - A_1[j]$$

[13] è sufficiente che le tracce catturino il consumo nella regione dell'algoritmo in cui viene manipolata la v_h.

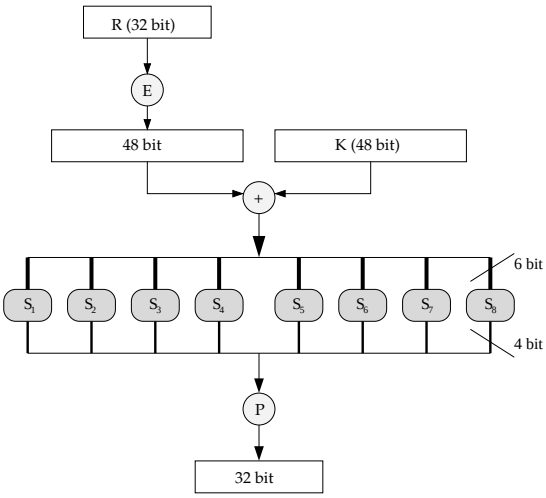


Figura 1.11: La funzione Feistel $f(R_{i-1}, K_i)$ del DES [10].

Ora, se l'ipotesi fatta su K_D è corretta, il valore calcolato mediante la funzione di selezione $D(\cdot)$ coincide effettivamente, per tutti i chipertext c_i (plaintext p_i), con il valore reale del bit b nella v_h. La funzione $D(\cdot)$ è cioè *correlata* con il valore di b. In questo caso la differenza nel termine P_{data_dep} nei due casi in cui b valga '1' oppure '0' tende a manifestarsi nella traccia differenziale $\Delta_D[j]$ con dei picchi.

Se invece, il valore ipotizzato per K_D è sbagliato, la $D(\cdot)$ risulta di fatto *scorrelata* rispetto a b. In tal caso, la partizione delle tracce $S_i[j]$ nei due insiemi S_0 ed S_1 è da ritenersi del tutto casuale determinando quindi un andamento di $A_0[j]$ presumibilmente molto simile a quello di $A_1[j]$. La traccia differenziale $\Delta_D[j]$ sarà quindi ovunque prossima a zero e l'attacco dovrà essere ripetuto, a partire dal punto 3, con un nuovo valore di K_D

Attacco differential power analysis al DES

Per fissare le idee con un esempio concreto vediamo come implementare l'attacco descritto nel caso del DES[14]. In particolare analizziamo l'ultima delle sedici iterazione dell'algoritmo dettagliata in Figura 1.11.

La funzione $f(R_{i-1}, K_i)$ riceve in ingresso rispettivamente:

- 32 bit del blocco R_{15}, risultato delle precedenti elaborazioni sulla metà sinistra del plaintext

[14] si veda [10] per una completa descrizione dell'algoritmo

- ❑ 48 bit della sottochiave K_{16}

Ricordando che R_{15} coincide con L_{16}, possiamo ricavarlo direttamente dai bit del chipertext applicando su di loro la trasformazione inversa alla IP^{-1}, ovvero IP^{15}. Considerando quindi una S-box: ciascuno dei suoi 4 bit in uscita è funzione solo del chipertext c e dei bit della sottochiave K_{16} in ingresso alla stessa S-box.
Costruendo dunque un totale di 8 funzioni di selezione, una per ogni S-box, è possibile realizzare un attacco al DES basato su DPA che determina i 48 bit della sottochiave in un massimo di 8×2^6 tentativi. Ottenuta la K_{16} si risale ai corrispondenti 48 bit della chiave segreta K percorrendo a ritroso l'algoritmo di *key schedule* e si completano i 56 bit di K con un classico attacco di *forza bruta* sui rimanenti 8 bit.

In definitiva la DPA consente di risalire alla chiave segreta del DES con un numero massimo di tentativi pari a $8 \times 2^6 + 2^8 = 3 \times 2^8$ contro i ben 2^{56} tentativi necessari usando semplicemente un attacco di forza bruta.

La figura Figura 1.12 mostra i risultati di una attacco al DES ottenuti a partire da una insieme di $N = 1000$ tracce di assorbimento. La prima delle quattro tracce rappresenta la potenza media dissipata dall'algoritmo durante le operazioni di cifratura. Le successive sono invece tracce differenziali; tra queste solo la prima è stata ricavata usando l'ipotesi corretta sul valore dei 6 bit della sottochiave K_{16}. Nelle altre possiamo verificare che, essendo ottenute partendo da ipotesi errate sul valore dei bit di K_{16}, non presentano alcun picco significativo, proprio come ci aspettavamo.

[15] IP ed IP^{-1} sono entrambe note, inoltre $C = IP^{-1}(R_{16}, L_{16})$

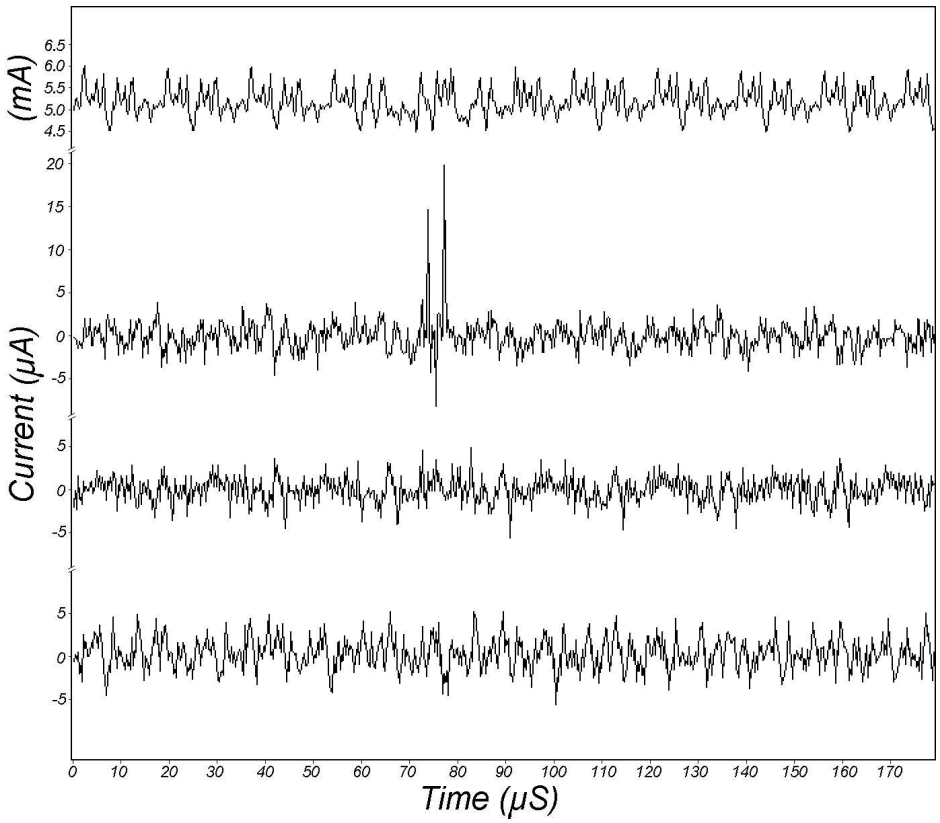


Figura 1.12: Risultati di una attacco basato su DPA nei confronti del DES eseguito su una tipica smart card [14].

CAPITOLO 2

Metodologie per la stima degli assorbimenti

"Mi si consenta di cominciare con fatti - fatti
nudi e crudi, verificati per mezzo di libri e cifre -
sui quali non posson sussistere dubbi"
Jonathan Harker

QUESTO capitolo presenta il modello per la stima degli assorbimenti da cui prende spunto lo stesso lavoro di Tesi. Dopo una introduzione al problema della riduzione dei consumi, ed una rapida carrellata delle soluzioni proposte a diversi livelli architetturali, si definiscono le ragioni che hanno spinto verso lo studio di approcci di più alto livello.

Nella Sez. 2.2 viene introdotto il concetto si stima a livello software e dettagliato un approccio di stima impiegato in parte anche per questo lavoro. La metodologia di stima basata sul concetto di funzionalità viene presentata nella Sez. 2.3 e la sua utilità viene inquadrata nel contesto dei sistemi per l'ottimizzazione dei consumi di potenza.

Infine nell'ultima sezione viengono descritti con sufficiente dettaglio i componenti del framework, creato a supporto dell'analisi comportamentale, e come interagiscono fra loro. Per meglio comprenderne il funzionamento un'appendice alla fine dell'opera commenta un caso d'uso sviluppato a supporto di questo lavoro di Tesi.

2.1 Ottimizzazione delle architetture

Il successo sul mercato di un dispositivo portatile è dettato sempre più anche dalla autonomia delle sue batterie. Il consumo energetico è divenuto quindi uno degli aspetti critici nella progettazione dei sistemi, specialmente nell'ambito dei sistemi embedded e conseguentemente dei processori che ne costituiscono parte essenziale. Nel corso degli anni sono state quindi sviluppate diverse tecniche, finalizzate ad ottimizzare i consumi oltre che migliorare le prestazioni, operanti a diversi livelli di astrazione: logico, architetturale e di sistema operativo[1].

2.1.1 Livello logico

Fino a qualche anno addietro, le tecniche di ottimizzazione dei consumi erano di natura sostanzialmente hardware. Le tecnologie che si basano sull'uso del silicio, per la produzione dei sistemi hardware, si sono sempre più perfezionate. La riduzione dello spessore del substrato semiconduttivo, ad esempio, ha via via consentito non solo di aumentare le prestazioni raggiungibili ma anche di contenere le dissipazioni.

La sola propagazione del segnale di clock è pari al 30% del consumo di un processore. Fra le principali soluzioni studiate per il contenimento dei consumi a questo livello troviamo:

- *clock gating* - disattiva quei rami dell'albero di propagazione del clock verso i dispositivi non utilizzati
- *half-frequency clocks* - prevede l'uso di entrambi i fronti del segnale di clock per sincronizzare gli eventi
- *half-swing clocks* - ovvero l'impiego di clock che usano solo metà della tensione di alimentazione V
- *logiche asincrone* - sono logiche in cui i componenti non necessitano di un segnale di sincronizzazione. Tali logiche necessitano di un aumentano del numero di segnali e di connessioni, sono dunque più complesse da progettare e testare. Un compromesso è rappresentato dai sistemi "ibridi": globalmente asincroni ma localmente sincroni.

[1] Una interessante dissertazione sulla tematica la si trova nell'articolo di Mudge [24]

2.1.2 Livello architetturale

Le ricerche nel campo delle architetture dei calcolatori si focalizzano principalmente sul miglioramento delle prestazioni. Tuttavia i principali temi di ricerca, ovvero l'aumento del *parallelismo* e l'*esecuzione speculativa*, hanno generalmente effetti positivi anche sul fronte della riduzione dei consumi. Il parallelismo consente di eseguire più attività contemporaneamente e spesso si traduce in significativi risparmi energetici. L'esecuzione speculativa, che consente alle istruzioni di proseguire anche se non sono ancora state risolte le dipendenze fra loro, può invece in alcuni casi produrre uno spreco di potenza. Tuttavia non sempre è così, la *branch prediction* è una delle più note tecniche di speculation e se la predizione è accurata aumenta di molto il rapporto $MIPS/W$. Altre soluzioni operate a questo livello sono:

- *caching delle memorie* - la memoria è infatti un'altra delle principali fonti di consumo energetico. L'impiego di una struttura gerarchica per le memorie consente di ridurre la frequenza degli accessi alla memoria principale. I vantaggi che si ottengono non sono solo in termini di prestazioni ma anche di risparmio energetico, infatti è possibile di disattivare i livelli non usati.
- *codifica dei bus* - con la conseguente riduzione della potenza assorbita.
- *pipelining* - consente di ottimizzare l'uso delle risorse del processore aumentando notevolmente il throughput[2]. Questa tecnica tuttavia non è molto significativa dal punto di vista dei consumi in quanto è spesso accompagnata da anche da un aumento delle frequenze di clock, che limitano la possibilità di diminuire le tensioni di alimentazione, e quindi produce un aumento dei consumi.

2.1.3 Livello di sistema operativo

La riduzione della tensione di alimentazione offre significativi benefici dal punto di vista del risparmio energetico. Affiancando questa considerazione all'osservazione che spesso un processore non deve lavorare al massimo delle sue potenzialità ne deriva l'idea che abbassando la frequenza del clock è possibile anche ridurre la tensione di alimentazione e conseguentemente contenere i consumi. Le tecniche che implementano questo principio sono note come *voltage scaling* e generalmente trovano un supporto direttamente a livello di sistema operativo.

Gli approcci visti fin'ora hanno consentito di raggiungere notevoli risultati sia

[2]Numero di istruzioni completate a parità di cicli di clock. Solitamente misurato in MIPS (*Million instructions per second*)

dal punto di vista delle prestazioni dei sistemi di elaborazione che da quello dei consumi dei dispositivi. Bisogna però osservare che l'aumento della complessità dei sistemi è stato motivato anche dalla richiesta del mercato di prodotti sempre più sofisticati anche dal punto di vista delle *funzionalità* disponibili.

La realizzazione di funzionalità complesse risulta spesso più conveniente, sia da un punto di vista economico che tecnologico, se implementa via software. Per questo motivo, negli ultimi anni, abbiamo assistito ad un considerevole aumento della componente software. Non è infrequente il caso di dispositivi che utilizzano processori general puorpose specializzati per le loro funzioni grazie a software applicativi sviluppati appositamente. Questa tendenza ha inevitabilmente portato a rendere più incisiva l'influenza che lo strato applicativo ha, tanto nelle prestazioni quanto nei consumi, al punto di divenire una delle parti critiche nella attività di progettazione. L'ottimizzazione del software è quindi divenuto un nuovo argomento di studio.

La realizzazione di sistemi embedded ottimizzati richiede però l'impiego di nuove *metodologie di progettazione* che tengano conto contemporaneamente di diversi aspetti: hardware, architetturali ed anche software. Si parla in questi casi di . Al fine di ottimizzare i consumi, in particolare, riveste una certa importanza la possibilità di avere informazioni attendibili sul presunto consumo di un dispositivo quando eseguirà un certo applicativo, già nelle primissime fasi del suo progetto. Il codesign può trarre vantaggio da queste informazioni, pilotando scelte strategiche come l'implementazione in software piuttosto che in hardware di talune componenti del dispositivo.

2.2 Stima della potenza assorbita

Le considerazioni del paragrafo precedente, in merito alla sempre maggiore incidenza dello strato software sui consumi energetici di un dispositivo, ci fanno capire come sia importante riuscire a predire il comportamento energetico di un sistema già dalle primissime fasi del suo progetto. A questo proposito molti studi sono stati fatti e troviamo in letteratura diversi approcci al problema.

Le principali metodologie disponibili per la stima degli assorbimenti si applicano a diversi livelli di astrazione, a partire dall'hardware fino a quello software-architetturale, con diversi vantaggi e svantaggi. Brevemente possiamo così riassumerle:

- *Stime a livello di transisitor* - si basano su una complessa e dettagliata rappresentazione del microprocessore come rete di transistors. Queste tecniche sono in grado di fornire delle stime di consumo molto accurate al punto da poterle sostituire alla misura effettuata operando direttamente sul core

del microprocessore, operazione questa spesso impraticabile. D'altra parte però, essendo basate su modelli fisici a *tempo continuo*, richiedono tempi di simulazioni molto lunghi che le rendono impraticabili al fine di ottenere delle caratterizzazioni di consumo ad alto livello, ad esempio per un intero programma.

- *Stima a livello di porte logiche* - la simulazione del sistema avviene a *tempo discreto* e ciò consente di ridurre notevolmente la complessità rispetto alla tecnica precedente, senza tuttavia compromettere significativamente l'accuratezza dei risultati. Tuttavia i tempi richiesti rendono anche questo tipo di analisi impraticabile al fine di caratterizzare sistemi complessi come un intero microprocessore.

- *Stima a livello di trasferimento tra-registri* - il sistema viene modellato come una rete di blocchi logici interconnessi. L'ipotesi alla base di queste metodologie è che le proprietà energetiche di un blocco possono essere analizzate studiandolo in modo isolato dal resto del sistema. Le informazioni disponibili per ciascun blocco vengono poi combinate fra loro attraverso una caratterizzazione delle interconnessioni e l'individuazione di una descrizione probabilistica d'alto livello dell'intero sistema. Un sottoinsieme del VHDL, detto RTL-VHDL o Verilog, è il linguaggio di descrizione usato tipicamente a questo livello di astrazione.

- *Stima a livello architetturale* - si tratta della più astratta rappresentazione che si possa avere di un sistema complesso come un microprocessore. Il sistema viene scomposto in *unità funzionali* viste come "black-box[3]" delle quali non si conosce la struttura interna ma solo il comportamento esterno. Tale comportamento può eventualmente essere individuato usando opportuni simulatori software ad-hoc per ciascun blocco funzionale. La modellazione e simulazione in questo caso può avvenire usando anche linguaggi di programmazione d'alto livello come il C++ nelle sue estensioni come SystemC o Hardware C.

I diversi approcci visti si differenziano fra loro fondamentalmente per due aspetti: i tempi necessari alla simulazione e l'accuratezza delle previsioni prodotte. In Figura 2.1 viene mostrato l'andamento di questi fattori nei diversi casi, le frecce indicano la direzione in cui si ha un aumento delle cifre di merito. Come possiamo osservare: lavorando a bassissimo livello si ottengono le stime più accurate, ma la complessità dei modelli richiede lunghi periodi di simulazione.

[3]Unità della quale si conosce solo l'interfaccia esterna e non si ha visione dei dettagli implemetativi interni.

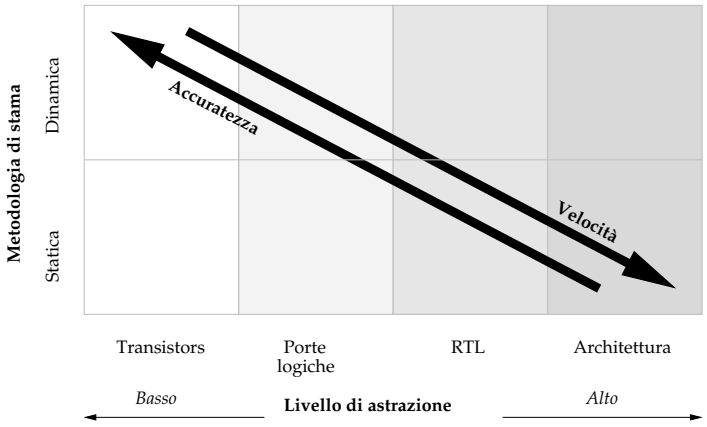


Figura 2.1: Semplificata classificazione dei diversi approcci alla stima dei consumi di potenza.

Se invece ci si accontenta di minori dettagli è possibile impiegare modelli più performanti.

2.2.1 Stima di potenza a livello software

Le principali limitazioni dei metodi di stima presentati nella sezione precedente sono legate alla mancanza di informazioni sui dettagli implementativi dell'architettura in esame ed ai medio-lunghi tempi di simulazione. Questi fattori rendono tali stima *difficilmente praticabili*. Bisogna poi anche osservare che, in talune applicazioni, una stima molto accurata e con un bassissimo errore percentuale non sempre è richiesta, soprattutto se per ottenerla è necessario formulare un modello piuttosto complesso[4].

Per rispondere a queste problematiche sono stati proposti, a partire dalla metà degli anni '90, i cosidetti "*Instruction-level power simulator*[5]". L'idea alla base è quella di individuare una *caratterizzazione in potenza* di ciascuna delle istruzioni che costituiscono l' del processore in esame. Successivamente, la stima degli assorbimenti di potenza di un dato programma può essere ottenuta semplicemente analizzando il codice effettivamente eseguito e sommando i contributi di consumo di ciascuna istruzione.

Interessanti riferimenti bibliografici sono il lavoro originale di Tiwari *et. al.*

[4] Ad esempio nella DPA ha più significato l'andamento della potenza assorbita piuttosto che il suo valore assoluto

[5] Simulatori di potenza assorbita a livello di istruzioni

[25], che rappresenta il primo tentativo di formulare un modello degli assorbimenti di potenza operante a livello software, ed i lavori da questo conseguiti come in [26] e [27]. In [28] Sarta *et. al.* propongono un raffinamento degli approcci precedenti introducendo nel loro modello, per la prima volta, una forma di dipendenza della potenza assorbita dal valore degli operandi delle istruzioni eseguite. La dipendenza dai dati della potenza assorbita, non solo raffina ulteriormente la precisione di questa classe di simulatori senza inficiare in modo significativo sui tempi di elaborazione, ma è anche particolarmente interessante dal punto di vista della DPA. In questo contesto infatti, come si è spiegato nel Par. 1.5.2 a pagina 20, è di fondamentale interesse riuscir a mettere in evidenza le correlazioni fra la potenza assorbita ed il valore di alcuni operandi.

Questa metodologia di stima è stata impiegata per la caratterizzazione del set di istruzioni del microprocessore $ARM7TDMI$, utilizzato come riferimento in questo lavoro di tesi, di seguito ne diamo quindi una descrizione più dettagliata. Come detto, alla base del funzionamento di un instruction-level power simulator vi è la caratterizzazione dell'insieme di istruzioni dell'architettura considerata. A seconda dell'accuratezza del metodo di caratterizzazione impiegato possiamo distinguere due tipologie di analisi: statica o dinamica.

Analisi statica del costo delle istruzioni[6]

La corrente assorbita dal microprocessore viene misurata mentre esegue una lunga sequenza di istruzioni uguali. L'assorbimento medio determinato viene poi considerato come rappresentativo per una qualunque istruzione di quel tipo. Questa operazione dovrà essere ripetuta per ciascuna delle istruzioni del processore al fine di caratterizzarlo completamente. In Figura 2.2 viene mostrato un esempio di codice assembly, utilizzabile per le misure, in cui un loop esegue 5000 volte una sequenza di N istanze della medesima istruzione *instr*.

L'obbiettivo di queste simulazioni è quello di costruire una tabella in cui ad ogni istruzione viene associato il numero medio di cicli macchina necessari a completarla e la relativa corrente[7] assorbita.

I costi così determinati però non tengono conto di alcuni fattori di assorbimento, per questo vengono detti *base cost*[8], e questo tipo di analisi è detto statico.

[6] Questo in particolare è l'approccio usato nell'ambito di questo lavoro di tesi
[7] E di conseguenza la potenza, usando la nota formula $P = V \cdot I$
[8] Costi base

```
        mov   5000, cx
loop:   instr
        ...
        ...                              (N volte)
        ...
        instr
        add   cx, -1, cx
        cmp   cx, 0
        bg    loop
```

Figura 2.2: *Ciclo assembly per la misura della corrente media assorbita.*

Analisi dinamica del costo delle istruzioni

Il comportamento di un microprocessore, ed in generale quello di ogni sistema digitale, dipende, oltre che dai valori in ingresso, anche dal suo stato. Conseguentemente è ragionevole aspettarsi che l'assorbimento del processore, mentre esegue un programma, dipenda anche dalla particolare sequenza di istruzioni eseguite. Questo tipo di dipendenza può essere messo in luce solo ricorrendo ad un'analisi di tipo dinamico ottenuta cioè caratterizzando l'insieme di istruzioni mediante l'uso di un generico programma che faccia manifestare tutti gli aspetti dinamici relativi alla sua esecuzione.

Sperimentalmente si è osservato che il consumo stimato per l'esecuzione di una sequenza di istruzioni è sempre inferiore a quello effettivamente misurato. Questa differenza si aggira intorno al 2–4% del base cost. Si tratta quindi di un piccolo contributo, imputabile al fatto che il modello ignora lo stato del processore ("*circuit state overhead*").

Più importanti sono invece i contributi di consumo dovuti a e ; questi tendono ad aumentare, rispetto al valore nominale riportato sui manuali, il numero di cicli di clock necessari ad eseguire una istruzione. Esperimenti condotti al fine di isolare questi effetti hanno mostrato che la loro incidenza si aggira intorno al 75% del base cost medio dell'intero instruction set, ovvero è paragonabile al consumo di una istruzione `nop`. Tenendo conto di ciò, un modello più accurato è stato proposto in [25]. Secondo tale approccio si stimano i costi di un *pipeline stall* (BC_{stall}) e di un *cache miss* (BC_{miss}) in modo da poterli impiegare per raffinare il modello iniziale. Questi costi, detti "*pipeline stall base cost*" e "*cache miss base cost*" , sono determinati usando del codice ad-hoc in cui si verifica un numero predeterminato di stalli nel flusso di esecuzione.

In Figura 2.3 è mostrato un esempio di codice in cui, se il numero n di vol-

```
mov   1, ax
mov   2, bx
add   ax,bx,cx
mov   1,ax
mov   2,bx
add   ax,xb,cx
```

Figura 2.3: *Codice assembly per la misurazione del costo di uno stallo.*

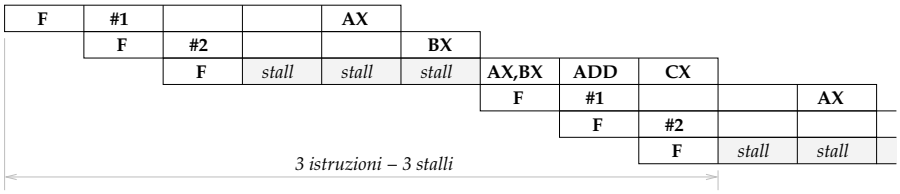


Figura 2.4: Stato della pipeline durante l'esecuzione del codice per la misura del base cost di uno stallo.

te che viene ripetuta la tripletta di istruzioni di interesse è sufficientemente alto, il contributo di potenza assorbito dal codice diverso da quello di stallo è trascurabile.

Tale codice, eseguito su un processore con pipeline a cinque stadi, produrrebbe una situazione simile a quella mostrata in Figura 2.4. Il numero di stalli prodotti è facilmente prevedibile e conseguentemente l'overhead di assorbimento imputabile a ciascuno di essi risulta essere dato da:

$$BC_{stall} = \frac{E_{act} - \sum_{i=1}^{3n} BC_i}{3n}$$

dove BC_i sono i base cost delle diverse istruzioni mentre E_{act} è l'energia assorbita effettivamente misurata. Il base cost di un cache miss si può calcolare operando in modo analogo.

Rielaborando poi questi costi aggiuntivi con delle informazioni statistiche sul profilo del codice eseguito in termini di numero medio di cache miss e pipeline stall si individuano i valori di *stall penalty* (SP) e *cache miss penalty* (MP) che vanno a sommarsi ai costi base raffinando il modello di previsione degli assorbimenti secondo tale relazione:

$$E_{tot} = \sum_{i=1}^{N} BC_i + SP + MP$$

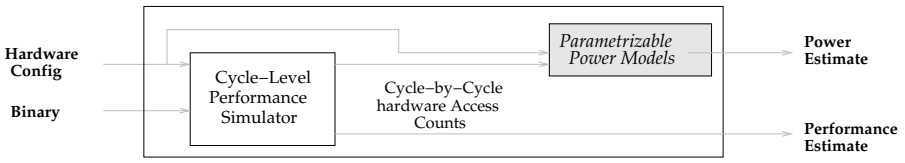


Figura 2.5: Struttura generale del simulatore di potenza Wattch.

dove N è il numero totale di istruzioni del codice in esame.

I risultati fin qui discussi sono incoraggianti in quanto mostrano che un modello di previsione a livello software è realizzabile. Tuttavia una approccio come quello descritto è, per costruzione, troppo dipendente dall'architettura in esame e manca di generalità rispetto al processore. Lo stesso setup richiesto per effettuare gli esperimenti presenta alcune problematiche, legate principalmente alle alte frequenze cui operano i processori di ultima generazione e quindi al costo delle attrezzature necessarie per la raccolta dei dati. Per queste ragioni tale metodologia, seppur realmente applicabile, viene considerata difficilmente perseguibile ai fini pratici del co-design.

2.3 Stima della potenza basata sulle funzionalità

Come descritto nelle sezioni precedenti, lo studio dei consumi dal punto di vista del software ha portato sostanzialmente a due approcci che lavorano a livelli diversi: architetturale e di instruction set.

Le soluzioni che operano a livello architetturale si basano generalmente sul tentativo di caratterizzare in potenza i principali blocchi dell'architettura a microprocessore considerata. Successivamente tali informazioni vengono utilizzate da un simulatore architetturale, più o meno generico, al fine di produrre i dati sui consumi dei programmi eseguiti. Una interessante implementazione di questa metodologia è discussa in [29] dove si presenta *Wattch*, un framework per l'analisi e l'ottimizzazione della dissipazione di potenza, che si presenta come estensione del simulatore architetturale *SimpleScalar*[9]. In Figura 2.5 viene mostrato lo schema generale di questa soluzione. Questi modelli generalmente non sono impiegabili nei casi in cui non siano noti i dettagli architetturali del processore d'interesse ed inoltre richiedono un certo sforzo per mappare nel modello

[9]Maggiori dettagli sul simulatore di architetture SimpleScalar si possono ottenere sul sito del progetto `http://www.simplescalar.com` o in [30].

la particolare architettura considerata. Inoltre, basando il loro funzionamento su una simulazione a livello di architettura, la produzione dei valori di consumo per un particolare programma richiede generalmente lunghi periodi di elaborazione.

Per cercar di superare queste problematiche sono stati introdotti i modelli basati sulla misura del consumo dell'instruction set. Questi modelli, già descritti nel precedente paragrafo, soffrono tuttavia di limitazioni legate principalmente alla scarsa generalizzabilità dei risultati ottenuti ed alla non praticità dell'ambiente di misurazione.

A fronte di queste considerazioni un nuovo approccio al problema della stima degli assorbimenti è stato proposto e sviluppato. Questa nuova metodologia può essere inquadrata sostanzialmente nella classe dei modelli che lavorano a livello di instruction set cercando però di superare le limitazioni tipiche di queste implementazioni. Ciò che ne è risultato è un modello sufficientemente generale ed indipendente da uno specifico microprocessore, che consente tuttavia di caratterizzare in potenza un instruction set con un sufficiente grado di accuratezza. Per ottenere questi risultati, evitando al contempo i lunghi tempi di elaborazione necessari a simulare accuratamente una seppur generica architettura, la metodologia proposta astrae dal livello architetturale e si focalizza sul concetto di *funzionalità* coinvolte nell'esecuzione di una istruzione.

L'accuratezza delle stime prodotte da questo modello è stata dimostrata formalmente in modo matematico, inoltre, l'analisi delle proprietà statistiche del modello conferma due interessanti possibilità di generalizzazione:

- *Intra-processore* - un modello costruito usando un significativo sottoinsieme di istruzioni consente l'estrapolazione della caratterizzazione energetica dell'intero instruction set.
- *Inter-processore* - un modello costruito usando un insieme di processori, anche solo parzialmente caratterizzati, consente l'estensione dei risultati ad una nuova architettura.

2.3.1 Il concetto di funzionalità

Una completa definizione del modello dal punto di vista matematico e statistico viene presentata nel lavoro di Brandolese *et. al.* [31]. Sostanzialmente la metodologia si basa sul concetto di *funzionalità* intesa come un insieme di attività orientate al raggiungimento di un certo risultato. Queste attività possono coinvolgere una parte o anche tutte le *unità funzionali* individuabili nella struttura di un *generico processore*. É bene notare anzi, a questo proposito, che il concetto di funzionalità definito dal modello è indipendente da quello di unità funziona-

li effetivamente individuabili in una particolare architettura[10]. Ogni istruzione eseguita viene poi vista come una successione di attività computazionali ciascuna delle quali utilizza una delle funzionalità individuate[11]. In un modello del genere, la potenza consumata da ogni istruzione può essere vista come la somma pesata dei contributi di ciascuna unità funzionale.

Una fase di tuning basata su un limitato insieme di dati sperimentali consente di associare ad ogni unità funzionale un'informazione sulla corrente media assorbita per ciclo di clock. La potenza assorbita da una istruzione dipendente ovviamente dal numero di cicli di clock necessari a completarla. In una prima stesura del modello, descritta in [31], il numero di cicli di clock necessari al completamento di una istruzione veniva considerato coincidente con il valore nominale riportato sui manuali del microprocessore considerato (). La tipologia di analisi effettuata era quindi di natura statica e non consentiva di tener conto dei ritardi, dovuti ad esempio a conflitti di pipeline o all'accesso alle memorie, che invece contribuiscono in modo significativo a determinare l'effettivo numero di cicli di clock necessari a completare l'istruzione. Come conseguenza il modello tende a sottostimare il reale assorbimento, con una percentuale di errore che è ancora più significativa nel caso delle architetture superscalari caratterizzate da un maggior grado di parallelismo.

In [32] viene proposta una estensione del modello al fine di individuare una descrizione formale e sufficientemente generale degli *overhead* dovuti alle interazioni intra-istruzione all'interno della pipeline. Ricordando però la Figura 2.1 possiamo osservare che l'analisi dinamica, essendo basata su simulazione, richiede generalmente dei tempi di elaborazione superiori a quelli necessari ad una più semplice analisi statica del codice. Tuttavia la strategia proposta da questi lavori è particolarmente interessante in quanto:

- si basa su una *caratterizzazione dinamica*, che consente di costruire un modello più aderente alla realtà, da effettuarsi una sola volta per una data architettura
- produce una accurata *descrizione statica* dei consumi, per un dato instruction set, che può essere vantaggiosamente dagli strumenti di analisi statica

In altre parole, questa metodologia consente di raggiungere il duplice obbiettivo di disporre di strumenti di stima efficienti quanto quelli statici ed al contempo accurati quanto quelli dinamici.

[10] Precisamente per funzionalità si intende una collezione di attività disguiunte in tempo o spazio dalle attività che costituiscono ciascuna altra funzionalità.

[11] Dette anche *unità funzionali*

2.3.2 Il progetto POET

Il modello brevemente descritto nel paragrafo precedente fa parte del più ampio progetto POET (***P**ower **O**ptimisation for **E**mbedded sys**T**ems*): un completo framework per l'analisi e l'ottimizzazione di software per sistemi embedded in grado di stimare l'assorbimento energetico di un programma scritto in C, eseguito su una determinata architettura, ed eventualmente proporre ottimizzazioni che ne migliorino l'efficienza[12]. Internamente il suo funzionamento si basa su tre "flussi", o livelli di astrazione:

- *flusso assembly:* rende disponibile un framework scritto in C++ per la modellazione di generiche architetture a microprocessore in termini di *unità funzionali*. Un modello così definito consente di simulare, da un punto di vista comportamentale, l'esecuzione di un programma binario su tale architettura sintetizzando informazioni come la corrente media assorbita dalle unità funzionali, l'incidenza degli stalli ed il grado di parallelismo a livello di istruzione, il tutto per classe di istruzioni.

- *flusso sorgente:* fornisce la stima del consumo e del tempo di esecuzione di ogni elemento sintattico di un sorgente C basandosi sulle stime generate dal flusso assembly. La possibilità di lavorare direttamente a livello sorgente consente di ridurre di diversi ordini di grandezza i tempi necessari alla stima dei consumi rispetto a quelli richiesti dai simulatori che operano a livello di istruzione. Operando direttamente a livello sorgente e più immediato individuare le regioni critiche di codice su cui intervenire a fini di ottimizzazione. Inoltre, grazie all'introduzione di modelli di guadagno in tempo/potenza di diverse trasformazioni, è più facile e veloce l'esplorazione dello spazio di ottimizzazione nelle sezioni critiche di codice.

- *flusso libreria:* consente di tener conto nelle simulazioni anche degli assorbimenti dovuti all'uso di librerie di terze parti, per le quali non si dispone del codice sorgente, usando delle stime statistiche sul consumo di ciascuna chiamata a funzione di libreria.

In questa visione il nuovo approccio alla stima di potenza e prestazioni è rappresentato dal flusso assembly e nel seguito ci concentreremo nella descrizione della sua implementazione. Ai fini di questo lavoro di tesi è stato infatti necessario intervenire a questo livello operando opportune modifiche ed estensioni con l'obbiettivo di introdurre un adeguato supporto al tracciamento della potenza assorbita per singolo ciclo di clock.

[12] Questa funzionalità è attualmente implementata solo in modo parziale.

2.4 Simulazione comportamentale: il flusso assembly

Senza addentrarci troppo nei dettagli teorici del modello[13] ricordiamo che esso si basa sulla conoscenza di due cifre di merito: l'overhead ed il coefficiente di parallelismo.

L'*overhead* è un parametro statistico che indica la frequenza media degli stalli durante l'esecuzione di una istruzione in un generico codice. Il *coefficiente di parallelismo* fornisce invece una stima del grado di parallelismo che una istruzione riesce a sfruttare, tipicamente una frazione di quello teoricamente reso disponibile dal processore reale. Conoscendo questi due parametri, in base al modello, è possibile risalire ad una stima più accurata del numero di cicli di clock necessari all'effettivo completamento di una istruzione assembly mediante la relazione:

$$CPI_{est} = p(s) \cdot [n(s) + oh(s)]$$

dove $n(s)$ e $oh(s)$ sono rispettivamente il numero di cicli di clock nominali necessari all'esecuzione[14] ed il numero medio di stalli per l'istruzione s. Il termine $p(s)$ tiene invece conto del coefficiente di parallelismo. Il modello energetico definito in [32] può così essere applicato considerando il nuovo termine CPI_{est}, che tiene conto delle interazioni fra istruzioni, consentendo di ottenere una stima più realistica della potenza media assorbita dall'istruzione.

2.4.1 Analisi comportamentale

Coefficienti di parallelismo ed overhead possono essere determinati, per ciascuna istruzione, mediante una *analisi statistica* dell'esecuzione di un programma reale in una data architettura. Un tale studio consente infatti di mettere in evidenza le interazioni dinamiche fra le istruzioni; così, ad esempio, gli overhead da stallo possono essere determinati semplicemente mediando i contributi delle singole istruzioni che ricadono entro una finestra temporale significativa[15].

Per determinare i parametri di interesse è quindi necessario impiegare un accurato simulatore delle tempistiche d'esecuzione di una data architettura. In particolare, per ciascuna istruzione assembly eseguita γ_k, tre valori sono di interesse:

- ❏ $t_{in}(\gamma_k)$ il time-stamp[16] in cui l'istruzione comincia ad essere eseguita

[13] Si rimanda a [31] e [32] per una descrizione formale del modello.

[14] Ovvero il valore in CPI riportato sui manuali dell'instruction set considerato.

[15] L'ampiezza di questa finestra, ovvero del numero di istruzioni consecutive da considerare per l'analisi, corrisponde alla massima latenza di una istruzione all'interno della pipeline.

[16] Espresso come il numero di cicli di clock trascorsi dall'inizio della simulazione.

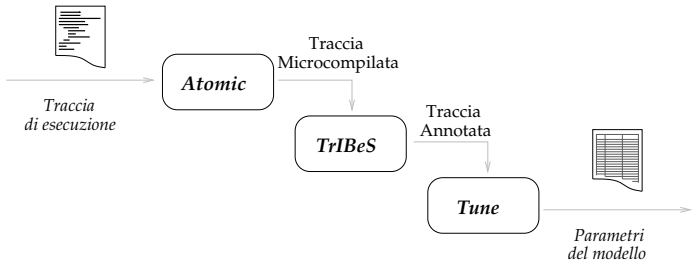


Figura 2.6: Struttura generale del flusso assembly per l'analisi comportamentale.

- ❏ $t_{out}(\gamma_k)$ il time-stamp in cui l'istruzione termina la sua esecuzione
- ❏ $n(\gamma_k)$ il valore (nominale) di CPI dell'istruzione

Note queste quantità, l'overhead dell'istruzione $oh(\gamma_k)$ può essere facilmente calcolato usando la relazione:

$$oh(\gamma_k) = t_{out}(\gamma_k) - t_{in}(\gamma_k) - n(\gamma_k)$$

Mentre $n(\gamma_k)$ può facilmente essere rintracciato nelle specifiche del processore considerato, per conoscere i valori t_{in} e t_{out} è necessario ricorrere ad un *simulatore comportamentale* ovvero ad una simulazione in cui si tenga conto delle sole proprietà tempistiche delle singole istruzioni, trascurando quelle funzionali o dipendenti dai dati.

A tale scopo è stato sviluppato un completo framework costituito fondamentalmente da tre tools: Atomic, TrIBeS e Tune. Questi strumenti, che nel loro complesso passano sotto il nome di *ATT Tools*, rappresentano quello che è noto come flusso assembly all'interno del progetto POET.

La struttura generale del flusso assembly, con le relazioni fra i vari tools, viene mostrata in figura Figura 2.6, di seguito dettaglieremo ciascuno dei suo componenti. Anticipiamo però subito che una delle caratteristiche della implementazione è quella di garantire il funzionamento in modo indipendente da una specifica architettura a microprocessore. Il flusso prevede quindi una fase iniziale di traduzione in cui il codice specifico di una architettura viene mappato su quello di una "macchina comportamentale" molto semplice per poi essere da quest'ultima processato al fine di raccogliere le informazioni necessarie sulle tempistiche.

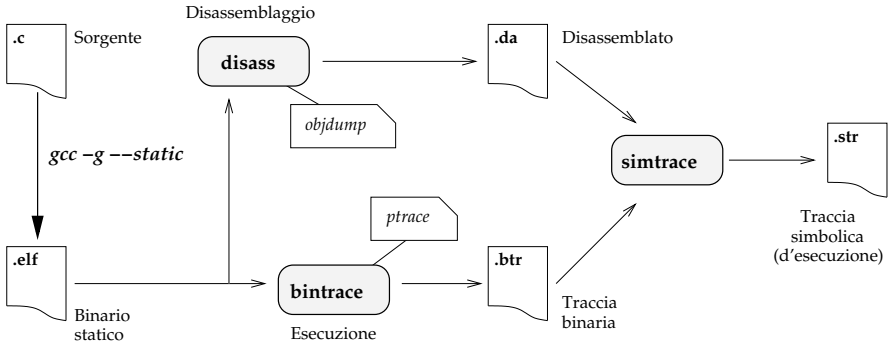


Figura 2.7: Preprocessing per la generazione di una traccia d'esecuzione.

2.4.2 Preprocessing: generazione della traccia d'esecuzione

Nella Figura 2.6 vediamo che l'elaborazione ha inizio a partire da una *traccia d'esecuzione* . Questa traccia rappresenta l'esatta successione di istruzioni assembly che vengono eseguite per una data istanza di un programma. Uno stesso programma, operante su dati d'ingresso diversi, potrà quindi generare tracce d'esecuzione diverse. E' importante osservare che in una traccia d'esecuzione il flusso di controllo è completamente conosciuto, ad esempio per ogni istruzione di salto condizionato siamo in grado di sapere se quest'ultimo è stato eseguito oppure no.

La Figura 2.7 schematizza il processo necessario alla creazione di una traccia d'esecuzione. Le fasi presenti, con le rispettive operazioni, possono essere così riassunte:

compilazione il generico sorgente C viene compilato[17] avendo cura di abilitare la generazione delle informazioni di debug e di "collegarlo" staticamente con le librerie usate.

disassemblaggio il binario[18] prodotto viene disassemblato con *disass*. Questo tool è semplicemente un wrapper per *objdump* cui aggiunge una serie di utili trasformazioni[19] al fine di omogeneizzare il codice prodotto in modo da risultare più indipendente dall'architettura.

bintrace attraverso questo tool si lancia il binario prodotto nella prima fase passandogli anche gli eventuali parametri di input. Sostanzialmente bintrace

[17] Il compilatore di riferimento è *gcc*, si veda `man gcc` per i dettagli di funzionamento
[18] Operando in ambiente GNU/Linux il formato binario usato è quasi sempre *elf*
[19] Come il renaming di alcuni registri o la riformattazione degli indirizzi di spiazzamento

```
...
2 8048267  N   xor     ecx<0x000a7b>,ecx<0x000a7b>
2 8048269  N   test    eax<0x030fb2>,ecx<0x000000>
2 804826b  N   je      0x804827b
5 804826d  N   mov     0x80a4388,eax
2 8048272  N   test    eax<80a4388>,eax<80a4388>
2 8048274  T   jn      0x804827b
3 804827b  N   mov     0x20(ebp),eax<80a4388>
...
```

Figura 2.8: *Esempio di traccia simbolica prodotta nella fase di preprocessing.*

esegue il binario d'interesse come un processo figlio in modo da poterlo controllare mediante *ptrace*[20]. Come output viene prodotta la successione dei valori assunti dal registro PC del processore, durante l'esecuzione del processo figlio, ed il valore dei sui registri GPR[21]. Quanto prodotto da questa fase è già una traccia d'esecuzione, detta *traccia binaria*, tuttavia è necessaria una ulteriore elaborazione al fine di renderla adatta all'elaborazione da parte del flusso ATT.

symtrace nell'ultimo passaggio traduce la traccia binaria in una *traccia simbolica*. Per ciascun valore del PC presente nella traccia binaria viene individuata la corrispettiva istruzione assembly nel disassemblato e riportata in uscita avendo cura di affiancare a ciascun registro il corrispettivo valore[22]. La sintassi della traccia prodotta contiene anche informazioni in merito ai salti, in particolare un flag consente di sapere se una certa istruzione di salto è stata onorata: una *T* indica un *Taken Branch* mentre una *N* un *Not Taken Branch*[23].

Un esempio di traccia d'esecuzione adatta ad essere processata dal flusso assembly è riportata in Figura 2.8, come si vede si tratta semplicemente di codice assembly opportunamente *annotato* con informazione sul valore dei registri e sull'esito dei salti.

[20] System call per il controllo di processi, si veda `man ptrace`.

[21] I registri del processore a disposizione delle applicazioni

[22] Il contenuto del registro si riferisce al valore in esso presente *prima* dell'esecuzione dell'istruzione

[23] "Taken branch" e "Not Taken Branch" indicano rispettivamente un salto nel flusso di controllo eseguito oppure no

2.4.3 Microcompilazione: Atomic

La traccia d'esecuzione, con l'elenco delle istruzioni assembly eseguite, viene convertita nel microcodice del simulatore comportamentale TrIBeS. Questa traduzione consente al simulatore di operare in modo indipendente da una particolare architettura ed instruction set, rendendolo di fatto una sorta di macchina virtuale.

Atomic (*Architecture-specific Trace Oriented Micro Instruction Compiler*) si occupa di questa traduzione ed è l'unico componente degli ATT tools ad essere dipendente da una particolare architettura. Il microcompilatore è scritto in C, per ciascuna architettura supportata[24], ed è costituito da un parser scritto in *bison* ed uno scanner ottenuto usando *flex*[25].

Il microcodice descrive le operazioni che devono essere compiute da TrIBeS al fine di simulare l'esecuzione di una certa operazione sull'architettura descritta. Ogni operazione assembly viene quindi scomposta in un certo insieme di *microistruzioni* direttamente interpretabili dal simulatore. Il microcodice prodotto può essere testuale o binario, quest'ultimo formato consente di velocizzare la simulazione.

Una microistruzione è composta da un *codice operativo* ed una serie di parametri alcuni dei quali possono essere opzionali, ecco un esempio del formato per ciascun tipo di microistruzione definibile:

`require` ⟨*clock_cycle*⟩ ⟨*fu_name*⟩
: definisce il numero di cicli di clock che l'istruzione spende nell'unità funzionale indicata, opzionalmente è possibile specificare a quale successiva unità funzionale dev'essere inviata l'istruzione, una volta completata, nel caso ci sia la possibilità di una scelta multipla

`read` ⟨*register_number*⟩ ⟨*register_file*⟩
: richiede la lettura dal registro del register file indicato

`write` ⟨*register_number*⟩ ⟨*register_file*⟩
: richiede la scrittura nel registro del register file indicato

`load` ⟨*address*⟩
: richiede l'accesso alla risorsa memoria per la lettura dell'indirizzo indicato

`store` ⟨*address*⟩
: richiede l'accesso alla risorsa memoria per la scrittura nell'indirizzo indicato

[24] In questo lavoro di tesi si è completato il microcompilatore per l'ARM7TDMI
[25] Un buon riferimento facilmente reperibile su questi due strumenti è [**?**]

Istruzione assembly	Microcodice TrIBeS		
`mul r0, r1, r2`	`read`	`0`	`regfile-int`
	`read`	`1`	`regfile-int`
	`require`	`16`	`alu-int`
	`write`	`2`	`regfile-int`
`ld [r0+r1], r2`	`read`	`0`	`regfile-int`
	`read`	`1`	`regfile-int`
	`load`	`1`	`<address>`
	`write`	`2`	`regfile-int`
`add f0, f1, f2`	`read`	`0`	`regfile-fp`
	`read`	`1`	`regfile-fp`
	`require`	`5`	`alu-fp`
	`write`	`2`	`regfile-fp`

Figura 2.9: *Esempi di microcodice per lo SPARCv8.*

`branch` ⟨*target*⟩ ⟨*direction*⟩ ⟨*type*⟩
richiede la verifica della correttezza della previsione di un salto presso la risorsa di branch prediction

`use` ⟨*resource_type*⟩ ⟨*resource*⟩
richiede l'accesso alla risorsa specificata

La Tabella 2.9 mostra alcuni esempi di istruzioni assembly e relativo microcodice riferite all'instruction set dell'architettura SPARCv8. Possiamo osservare che la moltiplicazione viene tradotta in quattro microistruzioni, tre delle quali sono usate per accedere al banco dei registri per gli interi, due volte in lettura ed una in scrittura. La rimanente microistruzione indica che l'esecuzione della moltiplicazione richiede 16 cicli di clock per essere eseguita dalla ALU per gli interi. Il secondo esempio si riferisce ad una operazione di caricamento da memoria, il campo <address> indica che esiste un parametro opzionale che contiene in tal caso l'indirizzo da caricare. Nell'ultima istruzione viene richiesta una operazione in virgola mobile, è da notare l'uso della ALU e del banco dei registri appropriati.

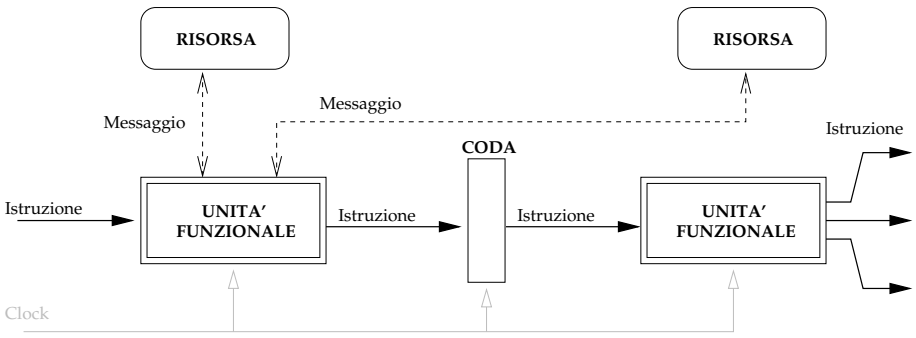


Figura 2.10: Architettura di TrIBeS.

2.4.4 Simulazione comportamentale: TrIBeS

Abbiamo visto nel Par. 2.4.1 che, al fine di determinare la caratterizzazione temporale del modello, è necessario conoscere l'esatto istante di inizio e fine dell'esecuzione di ogni istruzione della traccia. A tal fine è necessario simulare il flusso delle istruzioni all'interno dell'architettura considerata con la precisione del singolo ciclo di clock. Quello che serve è quindi un simulatore comportamentale in grado cioè di tracciare le interdipendenze fra istruzioni. A tale scopo è stato sviluppato *TrIBeS* (*Trace-based Instruction Behavioral Simulator*): un framework completo per lo sviluppo di simulatori comportamentali.

Ad alto livello TrIBeS può esser visto come un insieme di oggetti indipendenti ma interagenti e coordinati da un motore di sincronizzazione, in figura Figura 2.10 è mostrata l'architettura del simulatore le cui componenti sono:

unità funzionali :

sono l'equivalente delle unità funzionali del processore, ciascuna di esse infatti si mappa direttamente su uno stadio della pipeline[26]. Ricevuta in ingresso una istruzione, la conservano per il numero di cicli di clock necessari[27] a completare l'elaborazione richiesta e successivamente le inoltrano in una delle code di uscita dell'unità.

risorse :

una elaborazione può implicare l'uso di *risorse*, in tal caso le unità funzionali provvedono a verificare la dispobibilità della stessa a fornire il servizio richiesto inviandogli un "messaggio di richiesta". Nel caso la risorsa non

[26] Notare che il concetto di funzionalità, così come definito nel Par. 2.3.1, è diverso da quello di unità funzionale.

[27] Indicati da una microistruzione di **require**, 1 se non presente

```
ordinaUnitàFunzionali();
foreach unità funzionale do
 | execute_cycle();
end
foreach coda do
 | update();
end
foreach risorsa do
 | update();
end
```

Figura 2.11: *Simulazione di un ciclo di clock.*

sia disponibile l'uso viene negato e l'istruzione viene mantenuta all'interno dell'unità funzionale, al successivo ciclo di clock si ritenta l'accesso alla risorsa.

code :

creano le connessioni fra le unità funzionali che quindi non comunicano direttamente fra loro. Bensì ogni unità funzionale ha esattamente una *coda d'ingresso* ed una o più *code d'uscita* e comunicano fra loro inoltrando le istruzioni eseguite sulla loro coda d'uscita che rappresenta quella d'ingresso l'unità funzionale di destinazione.

In questa architettura TrIBeS si comporta come uno schedulatore di istruzioni le quali entrano nel simulatore da una speciale coda d'ingresso (*input queue*), prelevandole direttamente dalla traccia simbolica, che prende nota del ciclo di clock in cui ciò avviene: t_{in}. Le istruzioni "attraversano" poi il simulatore venendo accodate per l'esecuzione, all'interno delle unità funzionali, compatibilmente con i vincoli imposti dalle risorse. Quando tutte le microistruzioni che le costituiscono sono state processate, le istruzioni vengono inserite in una speciale coda d'uscita (*output queue*) che le scrive nel file d'uscita annotando anche l'istante in cui ciò avviene, ovvero t_{out}.

Le varie componenti vengono sincronizzate al fine di cooperare correttamente secondo il semplice algoritmo riportato in Figura 2.11 che viene ripetuto per ogni ciclo di clock fintanto che ci sono ancora istruzioni all'interno del simulatore. Anzitutto si effettua un riordinamento logico delle unità funzionali in base alla "età" delle istruzioni che stanno elaborando dando precedenza a quelle contenenti istruzioni con t_{in} minore. Seguendo l'ordine così stabilito si esegue un ciclo di elaborazione per ciascuna unità funzionale. Successivamente le code vengono aggiornate in modo tale da prelevare le istruzioni completate dalle unità funzio-

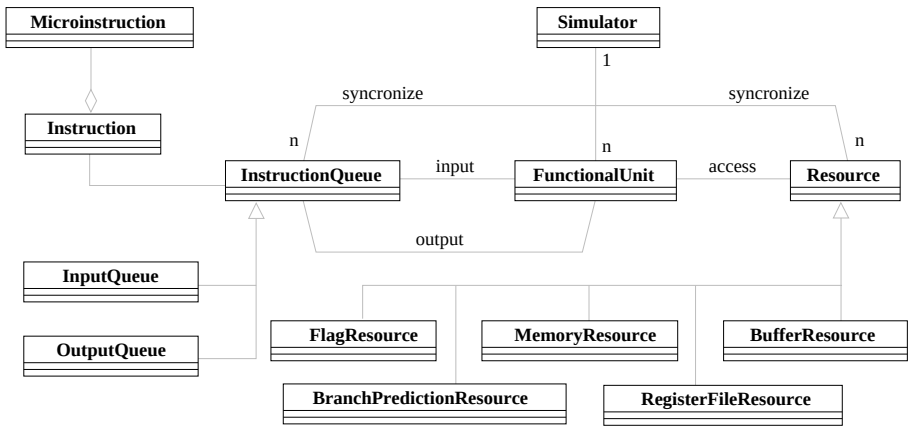


Figura 2.12: Diagramma delle classi di TrIBeS.

nali in cui si trovano e spostarle in quelle in cui proseguiranno l'esecuzione a partire dal ciclo di clock successivo. Infine si aggiorna lo stato delle risorse.

Operando in tal modo, TrIBeS produce in uscita tutte le informazioni necessarie al calcolo del coefficiente di parallelismo e dell'overhead per ciascuna istruzione γ_k della traccia d'esecuzione Γ considerata.

Uso del simulatore

L'uso del simulatore si concretizza nella descrizione dell'architettura d'interesse in termini di risorse ed unità funzionali, connettendo poi queste ultime per mezzo di code. Il progettista è supportato in ciò dall'SDK, fornito a corredo del simulatore TrIBeS, che implementa una gerarchia di classi base C++ a partire dalle quali specializzare a livello comportamentale le necessarie unità funzionali e risorse.

La Figura 2.12 mostra, facendo uso della notazione UML, la relazione che sussiste fra le diverse classi base disponibili. `Simulator` è la classe statica che implementa il motore di simulazione in modo simile a quanto riportato in Figura 2.11. La classe `Instruction` invece rappresenta la generica istruzione che fluisce all'interno del simulatore, si avrà una istanza di tale classe per ciascuna istruzione assembly γ_k della traccia eseguita. Ciascun oggetto `Instruction` è costituito da una aggregazione di `Microinstruction` , le microistruzioni, che sostanzialmente sono delle semplici struct inizializzate dalla coda d'ingresso mediante le informazioni lette dal file della traccia simbolica. `FunctionalUnit` , `Resource` e `InstructionQueue` sono le classi che modellano i componenti del simulatore atti a descrivere una architettura, ovvero rispettivamente: le unità

funzionali, le risorse e le code. Come mostra il diagramma vengono specializzate da `InstructionQueue` le due code speciali di ingresso ed uscita per la lettura della traccia d'esecuzione[28] e la produzione della traccia annotata. La classe `Resource` è invece una classe astratta da cui è possibile derivare diversi tipi di risorse, in figura ne sono riportati alcuni esempi.

Al fine di implementare il supporto al simulatore per una nuova architettura basterà derivare e specializzare le classi necessarie a partire da quelle indicate in figura. Ciascuna funzionalità individuata dovrà essere mappata in una classe derivata da `FunctionalUnit` e conterrà il codice necessario a gestire le richieste d'accesso alle risorse. L'uso delle risorse consente invece di modellare qualunque tipo di interazione inter-istruzione come stalli o cache miss. Ad esempio un tipico hazard RAW[29] può essere visto semplicemente come il blocco di un registro da parte di una istruzione e il tentativo di accesso in lettura allo stesso da parte di una istruzione successiva. Quando una risorsa nega l'accesso, l'unità funzionale che ha avanzato la richiesta stalla l'istruzione corrispondente per un ciclo di clock e ritenta l'accesso nel ciclo successivo. Lo stallo viene così propagato all'unità funzionale precedente attraverso la coda d'ingresso che, divenendo piena, non potrà più accogliere istruzioni. Questo semplice meccanismo è sufficientemente potente da consentire la modellizzazione di qualunque tipo di stallo in modo semplice e distribuito.

In Appendice A, a carattere esemplificativo, viene descritta l'implementazione del supporto al microprocessore *ARM7TDMI* che è stato usato come riferimento per questo lavoro di Tesi.

2.4.5 Analisi dei dati: Tune

L'ultimo passaggio della metodologia è la fase di tuning in cui le tempistiche d'esecuzione, prodotte da TrIBeScon la simulazione comportamentale, vengono analizzate al fine di individuare i parametri statistici del modello. Questa elaborazione dei dati di simulazione viene operata da Tune un tool che, in un singolo passaggio sulla traccia generata da TrIBeS, determina la frequenza delle singole classi di istruzione e corrispondentemente la relativa latenza e coefficiente di parallelismo.

Per concludere riportiamo alcune cifre di merito che consentono di stimare le prestazioni dell'implementazione data del modello. I test compiuti su una macchina dual Pentium III 966Mhz con 512MB di RAM operante in ambiente GNU\Linux RedHat 7.2. Delle singole fasi sono state misurate le prestazioni

[28] Ovvero la traccia simbolica prodotta nella fase di preprocessing
[29] *Read After Write*

Fase	**Tool**	**Prestazioni**
Preprocessing	bintrace	1.9 Minstr/s
Microcompilazione	atomic	70 Kinstr/s
Simulazione	tribes	4 Kinstr/s
Tuning	tune	90 Kinstr/s
Stima	annotate	140 Kinstr/s

Figura 2.13: *Prestazioni dei tools sviluppati a supporto del modello.*

riportate in Tabella 2.13. Il flusso di simulazione coinvolge tutte le fasi indicate e di conseguenza, eseguendole in cascata, si ottiene un throughput massimo di circa $4 Kistruzioni/s$. D'altra parte però, l'applicazione del modello per la stima dei consumi su una architettura già caratterizzata richiede solo la fase di "annotazione" che, come mostrano i dati, è molto più efficiente.

CAPITOLO 3

Modello per la generazione di tracce di potenza assorbita

"E' approccio comune quello di prendere un metodo e provarlo.
Se è fallimentare bisogna ammetterlo francamente e provarne un altro.
Quel che è importante è avere sempre una strada da provare."
Franklin D. Roosvelt

DOPO avere inquadrato nei due capitoli precedenti la tematica di interesse e gli strumenti di cui disponiamo, introduciamo in questo capitolo la proposta di un nuovo modello per il tracciamento della potenza assorbita ogni ciclo di clock. Il modello proposto si basa sull'idea di utilizzare il simulatore comportamentale TrIBeS al fine di generare i dati necessari all'analisi di vulnerabilità.

Un breve richiamo alle tecniche comunemente usate per le analisi di interesse consente di mettere in evidenza i difetti di queste e quindi giustificare la convenienza di un nuovo approccio.

Viene poi introdotto il concetto di stima comportamentale dei consumi. Ad una semplice soluzione del problema utilizzando questo approccio fa seguito una serie di considerazioni sul funzionamento del simulatore: vengono indagate le cause di stallo della pipeline ed introdotto il concetto di stato delle unità funzionali.

Sulla base delle considerazioni fatte presentiamo infine la proposta per un modello di stima comportamentale descrivendolo in dettaglio.

3.1 Introduzione

Nel Cap. 1 abbiamo visto come siano disponibili tecniche per risalire ad informazioni sensibili, come la chiave di una algoritmo di cifratura, semplicemente osservando la potenza assorbita dal processore. Questi attacchi si basano sulle "*Side Channel Information*" ed in particolare sulla possibilità di sfruttare la correlazione che sussiste fra una istruzione eseguita e l'energia da essa consumata. Senza particolari accorgimenti a livello hardware, o opportune riscritture della porzione di algoritmo che esegue le operazioni critiche, i sistemi di cifratura potrebbero risultare vulnerabili a queste tipologie di attacchi. Di conseguenza ha interesse individuare nuove tecniche che consentano di studiare il grado di esposizione di un dispositivo già nelle primissime fasi della sua progettazione.

Nel capitolo successivo abbiamo visto invece come si siano sviluppate diversi sistemi per l'analisi e l'ottimizzazione dell'esecuzione del software. Queste metodologie erano inizialmente finalizzate a supportare gli studi sull'aumento delle performance di un sistema di elaborazione. Successivamente, quando le problematiche legate al risparmio energetico[1] sono diventate più stringenti, nuove metodologie per la stima della potenza assorbita da un dispositivo sono state proposte a diversi livelli di astrazione. L'aumento della componente software all'interno dei sistemi ha fatto sì che la stima della potenza assorbita, direttamente imputabile alla struttura di un programma eseguito su un certo processore, abbia acquisito sempre maggiore interesse. La ricerca si è quindi focalizzata nell'individuazione di metodologie che fossero nel contempo: sufficientemente precise[2], generalizzabili[3] ed efficienti in termini di tempo necessario alla produzione dei risultati.

In questo contesto di necessità trasversali è stato sviluppato il simulatore TrIBeS, descritto nella parte finale del capitolo precedente. Si tratta di un simulatore comportamentale che risponde in modo innovativo al problema della stima degli assorbimenti. Basandosi su un ben definito modello matematico-statistico questo simulatore è in grado di individuare una caratterizzazione statica degli assorbimenti di un instruction set a partire da una analisi dinamica operata su alcuni benchmark.

L'obbiettivo di questo lavoro di tesi è quello di studiare la possibilità di impiegare un approccio innovativo per la valutazione della vulnerabilità di sistemi

[1] Soprattutto nel caso di dispositivi portatili alimentati a batterie
[2] Quanto basta per poter confrontare diverse scelte di progetto
[3] Ovvero con la massima indipendenza da architetture specifiche

di cifratura agli attacchi basati sull'analisi degli assorbimenti. L'idea è quella di semplificare l'attività di verifica dal punto di vista operativo cercando di utilizzare per l'analisi una traccia di potenza generata per simulazione. L'introduzione in un simulatore del supporto all'analisi di potenza assorbita renderebbe inoltre possibile effettuare questi studi già nella fase di design del dispositivo stesso.

3.1.1 Nuovo approccio all'analisi di vulnerabilità

Attualmente gli approcci all'analisi di vulnerabilità side-channel si basano sulla misura fisica operata direttamente sul core del processore mentre esegue le operazioni di cifratura. Un tale modo di operare, che in seguito chiameremo *metodo classico* , ha diversi svantaggi:

- richiede quantomeno la *realizzazione fisica* di un prototipo del dispositivo, sul quale operare le misure, con i conseguenti costi aggiuntivi nel caso si debba poi modificare lo stesso per risolvere le problematiche riscontrate
- l'attrezzatura richiesta per la rilevazione della potenza assorbita è piuttosto sofisticata e costosa ed il setup dev'essere eseguito con cura
- non sempre è applicabile; ad esempio nel caso si voglia verificare un dispositivo prodotto da terzi potrebbe non essere possibile rilevare le tensioni senza rischiare di danneggiare il dispositivo stesso

In Figura 3.1 è schematizzata una tipica installazione necessaria alla rilevazione fisica delle misure di tensione, notiamo in particolare l'impiego di un oscilloscopio digitale che dovrà avere una sufficiente risoluzione[4].

Quello che serve ai fini delle analisi di vulnerabilità in oggetto è sostanzialmente un tracciato dell'*andamento* della potenza assorbita. Ovvero non è necessario disporre del valore esatto della potenza che viene assorbita in ogni istante, quanto piuttosto è necessario un tracciato che mostri chiaramente le sue variazioni, anche se eventualmente scalate di un fattore costante.

Le considerazioni fin qui presentate ci spingono a pensare ad un approccio alternativo. In particolare potremmo pensare di generare i dati necessari all'analisi per simulazione. L'impiego di un simulatore dell'architettura di interesse avrebbe diversi vantaggi:

- generare un tracciato della potenza assorbita, con un sufficiente dettaglio in termini di valore assoluto, conservando le caratteristiche dell'andamento

[4] In termini di campioni acquisiti al secondo, per avere una buona visione del singolo ciclo di clock è necessario che la banda dell'oscilloscopio sia almeno doppia della frequenza del processore analizzato

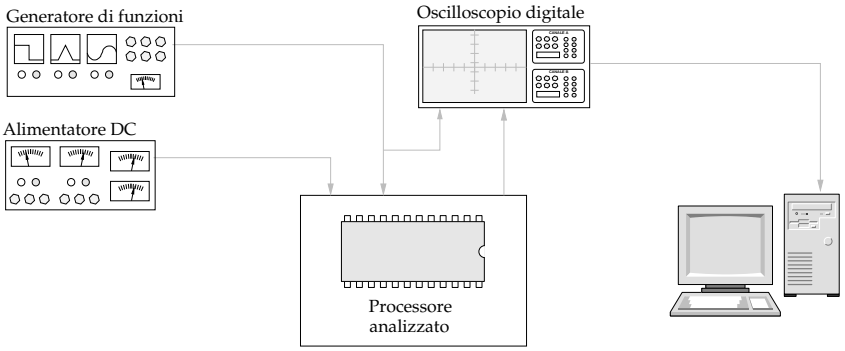


Figura 3.1: Tipica installazione per il rilevamento fisico della potenza assorbita.

- ❏ applicabile già nella fase di progettazione, con il vantaggio di poter utilizzare le informazioni che fornisce a supporto delle scelte di progetto prima di andare in produzione
- ❏ esser più economico non richiedendo l'acquisto di sofisticate attrezzature[5] e contribuendo ad evitare lo sviluppo di prototipi

Scegliendo di provare a percorrere questa strada cerchiamo di mettere in evidenza alcuni aspetti di cui tener conto al fine di inquadrare bene la soluzione da proporre. Esistono infatti diverse possibilità per la simulazione software del comportamento di un dispositivo, alcune di queste sono state discusse nella Sez. 2.2. Quello che a noi interessa fondamentalmente è riuscire ad avere uno strumento di simulazione competitivo, rispetto al metodo classico, per quanto riguarda i tempi necessari alla raccolta dati pur mantenendo un sufficiente grado di precisione sul tracciamento dell'andamento di potenza assorbita.

Queste prime considerazioni ci fanno scartare fin da subito tutti i metodi di simulazione che operano ad un livello di astrazione troppo bassa[6]. Come abbiamo visto infatti in questi casi i tempi di simulazione richiesti rendono proibitivo il tracciamento di un intero programma.

[5]Se non per la verifica della correttezza della stima degli assorbimenti da parte del simulatore stesso

[6]Ovvero stime a transistor level, gate level e RT level. Si rimanda alla Sez. 2.2 per una descrizione sommaria di questi approcci.

3.1.2 Risoluzione della simulazione

Concentrandoci quindi sui simulatori architetturali, o a livelli di astrazione ancor superiore, il dettaglio maggiore che possiamo raggiungere è quello del *singolo clock cycle*. In altre parole, quello che tali simulatori ci possono fornire è una previsione della potenza assorbita per ogni ciclo di clock. Questo dettaglio è inferiore rispetto a quello raggiungibile con il metodo classico, dove si può avere una risoluzione di alcune decine di campioni rilevati per singolo clock cycle[7], tuttavia non dovrebbe essere una eccessiva limitazione per i nostri scopi. Ricordiamo infatti che, sul tracciato di potenza assorbita, nel caso della SPA è necessario riuscire a mettere in evidenza il flusso di controllo, mentre per la DPA la dipendenza dai dati. In entrambi i casi possiamo ragionevolmente supporre che tutti gli effetti di interesse siano evidenti analizzando i singoli cicli di clock.

3.1.3 Possibilità di analisi

Le possibilità di analisi che ci sono offerte da un tracciato di potenza generato per simulazione dipendono sostanzialmente dal livello di astrazione al quale opera il simulatore. In funzione del tipo di analisi e delle sue caratteristiche possiamo intuire che alcuni livelli di astrazione non sono adeguati.

La dipendenza dai dati della traccia generata è una delle principali limitazioni. Questa caratteristica, fondamentale nel caso della DPA, non può essere espressa utilizzando ad esempio un simulatore comportamentale come TrIBeS in cui non viene realizzata una simulazione funzionale che tenga conto dei risultati delle operazioni e del valore che assumono i registri. Viceversa, anche con un tale simulatore, sarebbe comunque possibile effettuare SPA. In questo caso infatti la dipendenza dai dati richiesta è quella che si manifesta in una variazione del flusso di controllo che, comportando l'esecuzione di operazioni diverse con diversi profili di assorbimento, bene si manifesta nella traccia generata per simulazione comportamentale.

Tali osservazioni ci fanno subito capire che se decidiamo di ricorrere ad un semplice simulatore comportamentale dobbiamo rinunciare alla possibilità di effettuare DPA[8], in caso contrario sarà necessario ricorrere ad un simulatore di più basso livello.

Le considerazioni fin qui fatte vengono sintetizzate in Tabella 3.2, in que-

[7] Usando oscilloscopi digitali sufficientemente precisi ma, oltre certe frequenze di clock, anche parecchio costosi.

[8] Quantomeno ciò non sarà possibile fintanto che non si introduca nel simulatore una sufficiente forma di dipendenza dai dati.

	Metodo Classico	Simulazione architetturale	Simulazione comportamentale
Costo	✗	✓	✓
Complessità	✗	✗	✓
Generalità	✗	✗	✓
Velocità simulazione	✓	✗	✓
Supporto al design	✗	✓	✓
Risoluzione sub-clock cycle	✓	✗	✗
SPA	✓	✓	✓
DPA	✓	✓	✗

Figura 3.2: *Tabella comparativa delle caratteristiche dei metodi per l'analisi di potenza assorbita.*

sta tabella comparativa vediamo come l'ipotesi di ricorrere ad un simulatore comportamentale per le analisi di potenza assorbita abbia diversi vantaggi.

3.2 Stima comportamentale dei consumi

Nel paragrafo precedente abbiamo visto quanti vantaggi possa presentare la possibilità di effettuare una analisi di vulnerabilità agli attacchi in potenza già in fase di progettazione. Ora vogliamo studiare la possibilità di implementare un tale supporto all'interno di un simulatore comportamentale, in particolare riferendoci a TrIBeS.

Come spiegato nel Par. 2.4.4, TrIBeS è un simulatore comportamentale basato sul concetto di funzionalità. Una generica architettura a microprocessore viene descritta in termini di risorse ed unità funzionali, queste ultime interconnesse da code a formare una pipeline funzionale. Ogni istruzione assembly di una traccia di esecuzione viene quindi tradotta in un insieme di microistruzioni, ciascuna di esse contiene informazioni in merito all'uso di una certa unità funzionale oppure

alla necessità di accedere ad una delle risorse disponibili. Durante la simulazione le istruzioni fluiscono all'interno della pipeline "consumando" le microistruzioni presso le unità funzionali che visitano.

TrIBeS è stato progettato per raccogliere le informazioni necessarie al modello di caratterizzazione statica dei costi di un instruction set. I dati che produce sono quindi di natura temporale, ovvero il numero di cicli di clock effettivamente necessari ad eseguire ogni istruzione. Il simulatore esegue sostanzialmente una analisi dinamica del codice che consente di tener conto sia dei ritardi imputabili alle interazioni intra-istruzione che di quelli derivanti dall'accesso a risorse.

Quello che serve allo scopo dell'analisi di potenza sono invece delle informazioni sul consumo per singolo ciclo di clock. Nel resto del capitolo cercheremo di capire se e come è possibile ottenere tali informazioni dal simulatore comportamentale TrIBeS.

3.2.1 Analisi del problema

Una soluzione banale del problema potrebbe esser quella di sommare semplicemente i consumi medi delle istruzioni che sono presenti ad ogni ciclo di clock all'interno della pipeline funzionale. Un tale approccio viene schematizzato in Figura 3.3: il motore di simulazione, oltre a sincronizzare le unità funzionali, raccoglie le informazioni sulle istruzioni presenti in pipeline e, conoscendo i consumi attesi da ciascuna di esse[9], sintetizza il consumo complessivo del sistema per ogni ciclo di clock.

Un tale approccio tuttavia potrebbe rivelarsi troppo grossolano per almeno un paio di ragioni:

- non consente di discriminare il consumo di una istruzione in funzione dello stato della pipeline e dello stadio in cui essa si trova
- non tiene conto in alcun modo del valore dei parametri dell'istruzione e quindi della dipendenza dai dati della potenza assorbita

Con riferimento alla figura possiamo vedere infatti che nel caso si verifichi, ad esempio, uno stallo in $FU3$ avremo per due cicli di clock la stessa lettura di potenza assorbita: $P_{t_1} = p_1 + p_2 + p_3 = P_{t_2}$. Al contrario potremmo aspettarci invece che le istruzioni presenti nelle unità funzionali precedenti a quella che determina lo stallo siano caratterizzate da un consumo diverso, probabilmente inferiore. Questa intuizione deriva dall'osservazione che la circuiteria rappresentata dalle unità funzionali bloccate non sta attivamente elaborando delle informazioni e di conseguenza è ragionevole supporre che vari pure il suo consumo.

[9] Questi consumi sono noti a partire ad esempio da una caratterizzazione dell'instruction set eseguite con lo stesso TrIBeS.

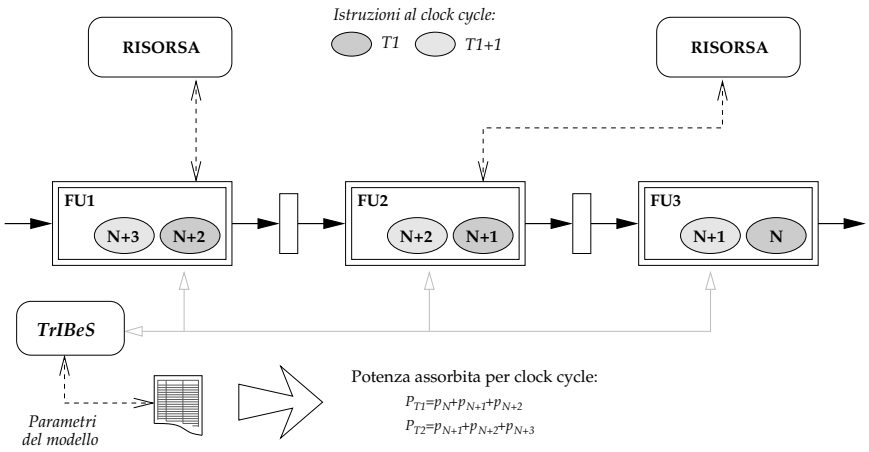


Figura 3.3: Soluzione banale: somma dei consumi medi di ogni istruzione in pipeline per ciascun ciclo di clock.

Un'altra osservazione che sembra ragionevole è che una istruzione sia caratterizzata da un consumo di potenza diverso in funzione della unità funzionale in cui si trova. Sappiamo infatti che le unità funzionali sono rappresentative di un insieme di attività svolte dal processore al fine di ottenere un certo risultato. Ognuna di queste attività coinvolge parte della circuiteria del microprocessore e solo questa contribuisce a determinare il consumo di potenza dell'istruzione nel ciclo di clock. Quando l'istruzione avanza nella pipeline impegna probabilmente circuiterie diverse ed il suo consumo risulterà quindi verosimilmente diverso.

Per quanto riguarda la dipendenza dai dati possiamo dire che TrIBeS, essendo un simulatore comportamentale, allo stato attuale non consente di supportare questo tipo di analisi. Tuttavia è possibile ipotizzare alcune estensioni funzionali che consentano in una certa misura di tracciare la dipendenza dai dati ma che probabilmente non sono implementabili nel semplice modello proposto prima[10].

Le osservazioni sin qui fatte ci spingono a cercare di formulare un modello di consumo per ciclo di clock che sia funzione dell'istruzione, dello stadio della pipeline in cui essa si trova e dello stato dell'intera pipeline. Lo stato della pipeline in particolare ci consente di sapere quali componenti sono attive ed impegnate effettivamente in una elaborazione, queste sono le componenti che ci aspettiamo abbiamo associato un maggiore consumo. Lo stato di ciascun componente della pipeline è determinato principalmente da come procede il flusso di esecuzione. Gli stalli sono i principali eventi che modificano tale flusso, rivediamo quindi di

[10]Una proposta di estensione funzionale del simulatore viene descritta brevemente nel capitolo conclusivo fra i possibili sviluppi futuri di questo lavoro.

seguito quali sono le principali cause.

Cause di stallo della pipeline

Nel simulatore comportamentale TrIBeS le cause di stallo della pipeline sono sostanzialmente di duplice natura:

- ❏ elaborazione multiciclo presso una unità funzionale
- ❏ attesa per l'accesso a risorsa

Il primo caso si verifica quando una istruzione γ_n presenta una microistruzione di tipo `require <c> <fu-k>`[11]. In tal caso, quando γ_n raggiunge l'unità funzionale `<fu-k>` vi rimane per il numero di cicli di clock `<c>`. Ciò consente di simulare il tempo necessario all'elaborazione della istruzione da parte dell'unità funzionale. Nel frattempo, nelle unità funzionali a monte di `<fu-k>`, si ha l'esecuzione delle istruzioni successive a γ_n. Nel caso in cui l'istruzione γ_{n+1} immediatamente successiva a γ_n richieda un numero di cicli di clock, presso l'unità funzionale `<fu-j>` che la sta elaborando, inferiore a quello di quest'ultima si ha uno stallo. In tal caso infatti, completata la sua parte di elaborazione, `<fu-j>` cercherà di inoltrare l'istruzione verso l'unità successiva `<fu-k>` che però è ancora impegnata da γ_n. L'unità funzionale `<fu-k>` è in stallo e questa condizione si propaga all'indietro verso tutte le unità funzionali a monte che hanno già ultimato la loro parte di elaborazione sulla istruzione che hanno in consegna.

Nel secondo caso lo stallo viene invece provocato da una qualunque delle istruzioni che prevedono l'accesso ad una risorsa[12]. Dalla descrizione del funzionamento del simulatore sappiamo infatti che, quando una istruzione deve accedere ad una risorsa, l'unità funzionale nella quale si trova verifica la disponibilità della risorsa. Se quest'ultima non fosse disponibile l'unita funzionale aspetta un ulteriore ciclo di clock per poi ritentare la richiesta. Potrebbe però anche accadere che la risorsa necessiti di un certo numero di cicli per rendere disponibile il servizio richiesto. Anche in questo caso l'unità funzionale si troverebbe bloccata ad aspettare la risorsa e lo stallo si potrebbe propagare alle unità funzionali a monte come un classico stallo di pipeline. È bene osservare che in quest'ultimo caso possiamo distinguere due tipologie di stallo: quello dell'unità funzionale che attende una risorsa e quello di tutte le unità funzionali a monte che sono bloccate a livello di pipeline.

[11] Si rimanda al 2.4.3 per una descrizione dettagliata delle tipologie di microistruzione
[12] Ad esempio il register file o la memoria

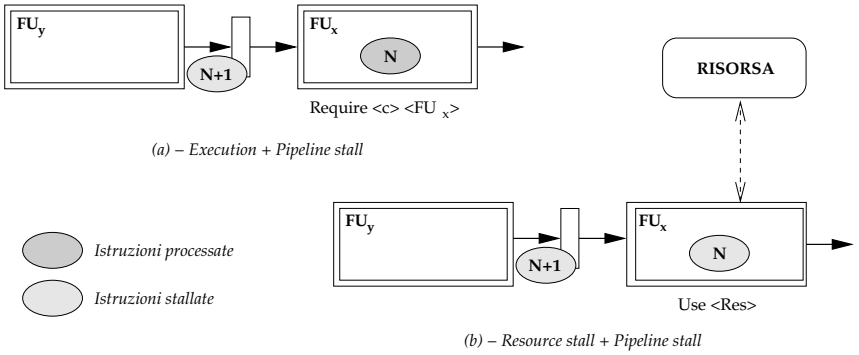


Figura 3.4: Possibili stati operativi di una unità funzionale in un dato ciclo di clock.

Stato delle unità funzionali

L'analisi delle possibili cause di stallo all'interno del simulatore TrIBeS ci consente di definire il concetto di *stato delle unità funzionali*. Ovvero:

> *Per stato di una unità funzionale intendiamo la condizione operativa in cui si trova una unità funzionale della pipeline in un determinato ciclo di clock*

Tale condizione operativa può quindi cambiare ad ogni ciclo di clock ed i possibili stati sono riassunti in figura Figura 3.4, ovvero:

- *execution* l'unità funzionale simula l'effettiva elaborazione di una istruzione. Questo è ad esempio il caso di FU_x in Figura 3.4-(a) in cui una istruzione di `require` impegna l'unità per un certo numero di cicli di clock
- *pipeline stalled* l'unità funzionale ha completato l'esecuzione della istruzione corrente ma non riesce ad inoltrarla lungo la pipeline verso l'unità successiva in quanto quest'ultima è impegnata per qualche ragione. Questo caso è quello di FU_y in entrambi i casi della Figura 3.4
- *resource stalled* l'unità funzionale aspetta la disponibilità di una risorsa oppure che questa completi il servizio richiesto dalla istruzione che sta processando. L'unità FU_x in Figura 3.4-(b) è un esempio di tale situazione.

3.3 Modello per la stima comportamentale

Alla luce delle considerazioni fatte precedentemente il modello di stima del consumo per ciclo di clock che vogliamo non è sufficiente che sia in grado di distinguere fra istruzioni diverse. Riteniamo invece significativo, ai fini di una stima più accurata, riuscire a tener conto anche dell'unità funzionale che sta elaborando l'istruzione, dello stato in cui questa si trova, ed ancora se l'istruzione ha richiesto l'uso di una certa risorsa.

Dato che il consumo è determinato dall'uso delle componenti fisiche del processore nel tempo, sembrerebbe logico anche cercare di individuare un modello che ci consenta di determinare il consumo come somma di contributi associati alle diverse componenti utilizzate da ciascuna istruzione. Un approccio di tal genere sarebbe molto simile a quello di un simulatore architetturale, in cui processore di interesse viene descritto nelle sue varie componenti ciascuna delle quali, oltre ad implementare una logica funzionale, è a conoscenza della propria caratterizzazione in potenza. In questi casi il motore di simulazione si limita a interrogare i singoli blocchi funzionali ed a sommare i contributi d'assorbimento da essi dichiarati ad ogni singolo ciclo di clock.

Ragionando in questi termini possiamo cercare di inquadrare una soluzione simile applicabile a TrIBeS che, pur non essendo un vero e proprio simulatore architetturale, è comunque basato sulla rappresentazione del processore come insieme di "blocchi comportamentali": le risorse e le unità funzionali. L'idea è quella di riuscire ad individuare una caratterizzazione in potenza dei blocchi alla base del simulatore per poter poi usare un approccio simile a quello descritto poco sopra.

La soluzione che proporremo si basa sulle due seguenti assunzioni:

> *Ogni unità funzionale è caratterizzata da un diverso assorbimento in funzione del suo stato e dell'istruzione che sta attualmente processando*

> *Ogni risorsa è caratterizzata da un diverso assorbimento in funzione dell'uso che se ne vuole fare, ovvero del tipo di istruzione che ne richiede un servizio*

Queste assunzioni sembrano ragionevoli e si giustificano sulla base delle considerazioni fatte in merito alle tipologie di stallo. Abbiamo infatti visto che gli stalli portano ad una definizione di stato delle unità funzionali. A sua volta lo stato rappresenta una condizione operativa e quindi anche una particolare modalità di utilizzo dell'hardware del microprocessore. Siccome il consumo deriva

dal modo in cui vengono usate le componenti fisiche del processore e le unità funzionali così come le risorse sono un'astrazione di queste[13], possiamo sicuramente affermare che il consumo, per ciascuna istruzione, sia dipendente dalla unità funzionale in cui si trova, dal suo stato ed eventualmente da quale risorsa si aspetta un servizio.

Questa discussione ci ha quindi portato ad affinare un nuovo modello di consumo dell'istruzione. Per distinguerlo da quello che non tiene conto dei nuovi parametri lo chiameremo *modello di consumo comportamentale* e si esprime in tali termini:

$$E_{\gamma_k,t} = f(FU_{\gamma_k,t},\ stato(FU_{\gamma_k,t}),\ risorse(\gamma_k, t)) \tag{3.1}$$

La relazione dice che:

> *l'energia $E_{\gamma_k,t}$ associata da una istruzione γ_k al ciclo di clock t è funzione dei seguenti parametri:*
>
> - *della unità funzionale in cui si trova γ_k al ciclo di clock t*
> - *dello stato in cui si trova tale unità funzionale*
> - *delle risorse che γ_k sta utilizzando al ciclo di clock t*

Sostituendo alla tabella dei costi per istruzione usata nella soluzione banale di Figura 3.3 questo più raffinato modello dei costi, dovremmo riuscire ad ottenere una più accurata stima dei consumi per singolo ciclo di clock. Tale soluzione ci consentirà di affacciarci al problema della analisi delle vulnerabilità previo traccia generata per simulazione mantenendo al contempo un sufficiente livello di generalizzabilità della soluzione per eventuali sviluppi futuri[14].

[13] In realtà questo non sempre è vero per le risorse usate a volte per modellare dei comportamenti del processore.

[14] In particolare, come vedremo, per l'introduzione di un supporto alla dipendenza dai dati del simulatore comportamentale TrIBeS in modo da rendere possibile anche la DPA.

3.3.1 Ridistribuzione dei costi

Dato un instruction set ed una particolare architettura, con i metodi visti nel capitolo precedente, siamo in grado determinare una caratterizzazione statica per il consumo di potenza di ciascuna istruzione. Quello di cui disponiamo è quindi una tabella che, per ciascuna istruzione[15], ci indica il valore medio di potenza assorbita per ogni ciclo di clock in cui l'istruzione risulti essere all'interno della pipeline. Questo valore non tiene conto di alcuni fattori come: dove si trova l'istruzione all'interno della pipeline e qual'è lo stato dell'unità funzionale che la ha in consegna.

Quello che abbiamo proposto nel paragrafo precedente è un modello di costo di una istruzione più raffinato. Infatti il contributo di assorbimento di una istruzione non è più di natura statica bensì può variare ad ogni ciclo di clock ed è definito da un certo insieme di parametri.

Dobbiamo ora trovare un modo per determinare questa più dettagliata caratterizzazione dell'insieme di istruzioni. In particolare sarebbe interessante poter usare come dati di partenza proprio quelli della caratterizzazione statica che siamo già in grado di conoscere. L'idea quindi è quella di partire dai dati "grossolani" della classificazione statica e rielaborarli per ottenere una visione a grana più fine che tenga conto dei parametri del nuovo modello, diremo *ribaltamento dei costi* questo processo.

> *Per* ribaltamento dei costi *intendiamo quel procedimento che ci consente di passare dalla caratterizzazione statica ad una a grana pi`u fine che esprima il consumo di ogni istruzione in funzione dei parametri del nuovo modello di consumo comportamentale.*

Non ci resta ora che da definire un metodo per il effettuare il ribaltamento dei costi. Tuttavia dobbiamo subito osservare che non abbiamo alcun modo per rilevare direttamente il consumo di una istruzione in una data unità funzionale, che si trovi in un certo stato oppure che stia utilizzando una certa risorsa. Sappiamo che il consumo è legato all'uso delle componenti hardware del processore ma è anche vero che non sempre possiamo mappare esattamente una certa unità funzionale o risorsa su un preciso sottosistema fisico del processore in esame.

Per risolvere il nostro problema potremmo quindi pensare di far ricorso ad un metodo di analisi statistica: osserviamo il comportamento di una certo numero di istruzione all'interno della pipeline e da questo cerchiamo di desumere la caratterizzazione energetica in funzione dei parametri comportamentali. Per

[15] Più precisamente per ciascuna classe di istruzione, dove ad ogni classe appartengono tutte le istruzioni che hanno lo stesso profilo energetico.

Figura 3.5: Dati sulle frequenze raccolti con l'analisi comportamentale.

la precisione il metodo di ribaltamento che proponiamo si sviluppa in due fasi, vediamole in dettaglio.

Analisi comportamentale

Prendiamo una traccia di esecuzione, sufficientemente grande e rappresentativa di tutte le istruzioni del processore in esame, e simuliamola usando TrIBeS. Durante la simulazione raccogliamo, per ciascuna istruzione, le informazioni sul comportamento della stessa all'interno della pipeline. Quello che possiamo ottenere sono dei conteggi, in particolare per ciascuna istruzione processata conosceremo:

- ❏ quanti cicli di clock l'istruzione ha speso all'interno di ciascuna unità funzionale in funzione dello stato della stessa
- ❏ quanti cicli di clock l'istruzione ha speso all'interno delle diverse unità funzionali in attesa di una risorsa o di ricevere un servizio da essa

I dati così ottenuti costituiscono una tabella in cui abbiamo una riga per ciascuna istruzione γ_k eseguita. Le colonne rappresentano invece i dati sui conteggi, con riferimento alla Figura 3.5 il loro significato è:

classe indica la tipologia di istruzione; l'instruction set viene infatti analizzato[16] e le sue istruzioni classificate raggruppando quelle che hanno un comportamento energetico simile, al fine di semplificare le analisi successive

unità funzionale per ciascuna unità funzionale vengono riportati tre valori corrispondenti al numero di cicli di clock spesi dall'istruzione al suo interno quando essa si trovava rispettivamente nello stato di *execute*, *pipeline stall* e *resource stall*

[16]La classificazione può essere effettuata manualmente, come abbiamo fatto noi, oppure per mezzo di opportuni strumenti di analisi e classificazione delle istruzioni

risorsa per ciascuna risorsa viene indicato il numero di cicli di clock per i quali l'istruzione ha dovuto attendere un servizio[17]

Ridistribuzione ai minimi quadrati

Dai dati sulle frequenze ottenuti nella prima fase dobbiamo ora sintetizzare il consumo dell'istruzione in funzione dei parametri del modello comportamentale. Ricordiamo che quello che vogliamo ottenere è una misura del consumo di una data istruzione in funzione dell'unità funzionale, del suo stato e delle risorse utilizzate. Quello di cui disponiamo è invece il dato sul consumo medio per ciclo di clock della istruzione.

Non sappiamo se sussiste una relazione fra i parametri del modello ed i dati di frequenza determinati, tuttavia possiamo supporre che ci sia una qualche correlazione fra l'energia associabile ad un blocco del simulatore ed il numero di cicli di clock che una istruzione l'ha mantenuto impegnato. L'idea quindi è quella di ripartizionare il costo necessario alla completa esecuzione di una istruzione sulle diverse unità funzionali e risorse in modo proporzionale a quanto l'istruzione le ha impegnate.

Definendo:

$$\Psi = \{FU_i\}, \quad i \in [1; x]$$
$$\Phi = \{Res_l\}, \quad l \in [1; y]$$
$$\Omega = \{exec,\ pstall,\ rstall\}$$

possiamo impostare un sistema, avente una equazione per ciascuna istruzione simulata $\gamma_k \in \Gamma$, di questo tipo:

$$\sum_{\substack{i\in[1;x]\\ l\in[1;y]}} c_{kil}F_{kil} \ + \ \sum_{j\in\Omega} c_{kj}R_{kj} \ = \ c_{\gamma_k}P_{\gamma_k} \qquad (k = 1 \ldots |\Gamma|) \tag{3.2}$$

in cui:

[17] Non distinguiamo se la risorsa era impegnata a servire questa istruzione oppure un'altra, tuttavia tale estensione è facilmente inseribile nel modello.

c_{kil} è il numero di cicli di clock che l'istruzione γ_k si è trovata nella $i-esima$ unità funzionale mentre quest'ultima era nello stato l

F_{kil} è la potenza incognita associabile alla $i-esima$ unità funzionale quando si trova nello stato l in corrispondenza dell'istruzione γ_k

c_{kj} è il numero di ciclo di clock per i quali l'istruzione γ_k ha dovuto attendere la risorsa $j-esima$

R_{kj} è la potenza incognita associabile alla $j-esima$ risorsa quando l'istruzione γ_k è in sua attesa

Da notare che nell'equazione: i termini $c_{kil}F_{kil}$ sono tanti quanti gli elementi dell'insieme $\{\Psi \times \Omega\}$, ovvero ci sono 3 termini per ogni unità funzionale, mentre i termini $c_{kj}R_{kj}$ sono solo uno per ciascuna risorsa in Φ. Infine:

c_{γ_k} è il numero complessivo di cicli di clock che sono stati necessari all'esecuzione dell'istruzione γ_k, da quando è entrata in pipeline fino a che ne è uscita

P_{γ_k} è la potenza media per ciclo di clock assorbita dalla istruzione γ_k

Del sistema dato le uniche incognite sono i termini F_{kil} ed R_{kj} che rappresentano appunto le potenze utilizzate dal modello comportamentale dei costi. I valori di c_{kil} e c_{kj} vengono determinati per simulazione nella prima fase, così come c_{γ_k}[18]. I termini P_{γ_k} rappresentano invece i nostri dati di partenza.

Un sistema così impostato è evidentemente sovradimensionato, avremo infatti una equazione per ciascuna istruzione eseguita. Per risolverlo possiamo fare ricorso al metodo dei minimi quadrati[19]. Diciamo:

$\underline{y}$ il vettore dei termini noti, è un vettore colonna costituito dai termini $c_{\gamma_k}P_{\gamma_k}$

$\underline{x}$ il vettore delle incognite, è un vettore colonna costituito da una delle possibili permutazioni degli elementi dell'insieme $(\Psi \times \Omega) \cup \Phi$

$\underline{\underline{A}}$ la matrice dei coefficienti, di dimensione $(3|\Psi| + |\Phi|) \times (|\Gamma|)$, costituita dai termini c_{kil} e c_{kj} corrispondenti alle rispettive incognite nel vettore $\underline{x}$

Possiamo quindi riscrivere il nostro problema in questa forma:

$$\underline{y} = \underline{\underline{A}}\,\underline{x}$$

[18]Tale valore viene già fornito di TrIBeS nella implementazione precedente alla estensione introdotta con questo lavoro di tesi.

[19]Eventualmente *vincolati* in modo da evitare di ottenere dei valori di potenza assorbita negativa.

Risolvere tale problema ai minimi quadrati significa trovare quei valori di $\underline{\hat{x}}$ tale che:

$$\min_{\underline{\hat{x}}} \|\underline{x} - \underline{\hat{x}}\|^2 \;=\; \min_{\underline{\hat{x}}} \|\underline{y} - \underline{\underline{A}}\,\underline{\hat{x}}\|^2$$

Sviluppando la forma quadratica:

$$\phi(\underline{\hat{x}}) = \underline{\hat{x}}^T\,\underline{\underline{A}}^T\,\underline{\underline{A}}\,\underline{\hat{x}} - 2\underline{y}^T\,\underline{\underline{A}}\,\underline{\hat{x}} + \underline{y}^T\,\underline{y}$$

ed annullando la derivata prima otteniamo la soluzione cercata:

$$\frac{\partial\phi(\underline{\hat{x}})}{\partial\underline{\hat{x}}} = 0 \qquad \longrightarrow \qquad 2(\underline{\underline{A}}^T\,\underline{\underline{A}})\underline{\hat{x}} - 2\underline{\underline{A}}^T\underline{y} = 0$$

Ovvero[20]:

$$\underline{\hat{x}} = (\underline{\underline{A}}^T\,\underline{\underline{A}})^{-1}\underline{\underline{A}}^T\underline{y} = pinv(\underline{\underline{A}})\underline{y}$$

Il vettore $\underline{\hat{x}}$, soluzione ai minimi quadrati del sistema di Eq. (3.2), ci fornisce la caratterizzazione del modello comportamentale di consumo. Tali valori ci consentono di applicare la Eq. (3.1) per determinare il costo della istruzione ad ogni ciclo di clock in funzione dei parametri del modello.

Le equazioni del sistema impostato sono, al più, tante quante le istruzioni della traccia eseguita. In realtà alcune di queste equazioni non saranno fra loro linearmente indipendenti e potranno essere semplificate eliminando le ripetizioni. Affinché la caratterizzazione ottenuta con il metodo di ribaltamento dei costi spiegato[21] sia sufficientemente accurata è necessario che la traccia soddisfi i seguenti requisiti:

1. sia rappresentativa di tutte le istruzioni dell'instruction set considerato[22]
2. presenti il maggior numero possibile di permutazioni di istruzioni diverse, in modo tale da garantire la migliore copertura possibile di tutti i casi di interazione fra istruzioni all'interno della pipeline

In ogni caso la traccia deve essere comunque sufficientemente grande, in termini di istruzioni eseguite, al fine di rendere significativa l'applicazione del metodo dei minimi quadrati. Tipici valori sono quelli di tracce con almeno qualche milione di istruzioni assembly, fino ad un massimo di alcune decine di milioni, che corrispondono all'esecuzione di programmi reali.

[20] Dove $pinv(\underline{\underline{A}})$ è la matrice pseudoinversa di Moore-Penrose della matrice $\underline{\underline{A}}$

[21] Applicato sul sistema già semplificato dalle equazioni non linearmente indipendenti

[22] Nel caso ottimale dovrebbe contenere almeno una istruzione per ognuna di quelle disponibili.

Analisi "class based" vs "analisi classless"

Risolvendo direttamente ai minimi quadrati il problema di Eq. (3.2) otteniamo una caratterizzazione energetica dei blocchi funzionali del simulatore, e delle risorse, che sarà indipendente dalla particolare istruzione eseguita. Per tale ragione diciamo questo tipo di soluzione *analisi classless*.

Possiamo già immaginare che tale tipo di analisi commetterà un errore tanto maggiore nella previsione dei consumi di una istruzione, quanto più la traccia eseguita non soddisfi appieno alla prima delle caratteristiche evidenziate precedentemente. Se infatti una particolare istruzione non compare nella traccia usata nella fase di caratterizzazione di una architettura il modello potrebbe non tenere conto di un suo particolare comportamento energetico.

Una alternativa a questo approccio, che consenta una caratterizzazione a grana più fine, è quella che diciamo *analisi class based*. In questo caso il problema originario viene scomposto in un insieme di sottoproblemi, simili nella struttura, uno per ciascuna classe di istruzione.

> *Una classe di istruzioni è un insieme delle istruzioni di un instruction set che sono caratterizzate da un comportamento funzionale/energetico simile.*

L'instruction set viene quindi precedentemente analizzato al fine di individuare tutte le possibili classi di istruzioni[23]. Successivamente, i dati sulle frequenze prodotti dalla prima fase vengono scomposti raggruppando fra loro tutte e sole le righe della tabella prodotta che corrispondono ad istruzioni della medesima classe. Per ciascuna classe si imposta e risolve un problema identico a quello dell'analisi classless.

Questo approccio consente di stimare in modo più accurato il comportamento energetico tipico di ogni istruzione, o meglio della sua classe, presso le diverse unità funzionali e risorse. Tuttavia rende obbligatorio il soddisfacimento del primo requisito sulla traccia. Infatti, nel caso in cui un particolare classe di istruzioni non fosse presente nella traccia usata in fase di caratterizzazione, il modello risulterebbe mancante del corrispondente costo.

Entrambi gli approcci verranno considerati e confrontati nel capitolo sulla valutazione del modello, rimandiamo quindi ad allora ulteriori considerazioni sulla opportunità dell'impiego dell'uno piuttosto che dell'altro.

Riassumendo, il modello proposto ambisce ad individuare una sorta di "caratterizzazione architetturale" a partire da una a livello di instruction set. Quella

[23] Questa operazione di classificazione può essere effettuata manualmente, come nel nostro caso, oppure facendo ricorso a qualche tipo di analisi automatica.

che otteniamo non è propriamente una caratterizzazione architetturale in quanto si riferisce al blocchi di un simulatore comportamentale, TrIBeS nel nostro caso, e non ai reali blocchi funzionali del processore fisico[24]. L'aspetto più interessante è quello di poter disporre di una nuova metodologia per il tracciamento dei consumi per ciclo di clock[25]. Questa nuova metodologia unirebbe un approccio architetturale, strutturalmente più dettagliato, ad una migliore velocità di simulazione, in quanto effettuata a livello comportamentale. Nel prossimo capitolo dettaglieremo una estensione del simulatore TrIBeS a supporto di questo nuovo modello.

[24]Tuttavia spesso la corrispondenza fra queste due entità è particolarmente marcata.
[25]Ovviamente nel caso il modello si dimostri effettivamente sufficientemente corretto

CAPITOLO 4

Simulatore software per l'analisi degli attacchi in potenza

"Se l'unico strumento che hai è un martello: comportati come se tutto il resto fosse un chiodo"
Abraham Maslow

PRESENTIAMO in questo capitolo le modifiche apportate al simulatore comportamentale TrIBeS, al fine di generare un tracciato della potenza assorbita per singolo clock cycle, implementando il modello descritto nel capitolo precendente.

Inizialmente riassumiamo gli obbiettivi dell'estensione e ne definiamo poi i requisiti. Successivamente analizziamo in dettaglio ciascuna delle due fasi proposte dal modello, cominciando con alcune considerazioni sullo stato del simulatore e deducendo poi la soluzione che meglio sembra adattarsi compatibilmente ai requisiti definiti.

Infine riportiamo uno schema generale per l'uso dell'estensione e dettagliamo le componenti esterne al simulatore necessarie per l'analisi dei dati ai fini dell'operazione di ribaltamento dei costi.

4.1 Introduzione

Il modello per il tracciamento della potenza assorbita per singolo ciclo di clock, formalizzato nel capitolo precedente, è stato implementato come estensione del simulatore comportamentale TrIBeS. Questo simulatore, introdotto a supporto di una metodologia per la stima della potenza assorbita da un programma, opera già a livello di ciclo di clock. Quello che serve in più al nostro scopo è:

- ❏ la generazione dei dati sulle frequenze ovvero il numero di cicli di clock che ciascuna istruzione spende nelle diverse unità funzionali[1] e quello necessario per gli accessi alle risorse, secondo lo schema riportato in Figura 3.5 a pagina 62
- ❏ il tracciamento del consumo dell'intera pipeline, per ciascun ciclo di clock, in funzione dei parametri del nostro modello, ovvero secondo la relazione:

$$P_t = \sum_{\substack{FU_x \in pipeline \\ Res_y \in pipeline}} p(\gamma_k, FU_x, stato(FU_x) \mid \gamma_k \, in \, FU_x) + p(\gamma_k, Res_y \mid \gamma_k \, usa \, Res_y)$$

L'estensione che proporremo deve soddisfare alcuni requisiti:

- ✓ *compatibilità all'indietro:* il nuovo codice deve integrarsi in quello esistente senza modificarne il funzionamento attuale. In particolare dev'essere prevista una modalità di esecuzione apposita per l'attivazione del nuovo codice
- ✓ *prestazioni:* il nuovo codice dev'essere quanto più possibile ottimizzato per consentire sia una veloce generazione dei dati di frequenza e, soprattutto, del tracciato di potenza assorbita[2].
- ✓ *generalizzabilità:* la soluzione dev'essere quanto più possibile indipendente dall'architettura, così come lo è TrIBeS stesso, ed al contempo essere predisposta ad eventuali estensioni future[3]
- ✓ *semplicità:* in fase di definizione di una nuova architettura, il codice dell'estensione dev'essere il più ridotto possibile e quello presente sufficientemente semplice[4]

[1] Distinguendo lo stato delle stesse

[2] Le prestazioni sono più importanti nella fase di tracciamento dato che è quella più comunemente usata, mentre la prima fase è necessari solo per il tuning di una nuova architettura

[3] Ad esempio per il supporto della dipendenza dai dati ai fini di una analisi funzionale necessaria nel caso di DPA

[4] In modo da poterlo eventualmente gestire con un futuro tool per la configurazione automatica di una nuova architettura.

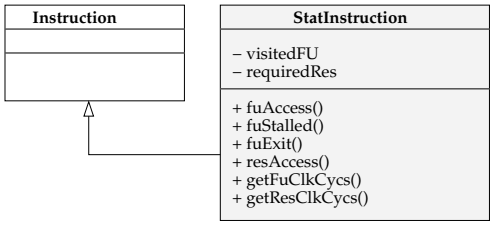


Figura 4.1: Diagramma UML che descrive l'estensione della classe Instruction.

Tenendo conto di questi requisiti, andiamo a dettagliare nei successivi paragrafi la soluzione proposta a supporto di ciascuna delle due fasi.

4.2 Estensione TrIBeS per la generazione delle statistiche

L'estensione proposta si base su alcune considerazioni. Le istruzioni attraversano le unità funzionali della pipeline necessarie alla loro esecuzione, fermandosi in esse il tempo necessario alla loro elaborazione più quello dovuto ad eventuali stalli. Le stesse istruzioni contengono le informazioni[5] necessarie per l'accesso alle risorse da parte di alcune delle unità funzionali che visitano.

L'idea quindi è la seguente: facciamo in modo che una istruzione, durante il suo viaggio all'interno della pipeline, oltre a consumare microistruzioni di cui si compone raccolga anche i dati di frequenza richiesti.

La soluzione che ne deriva, come mostra il class diagram UML in Figura 4.1 consiste quindi in una semplice estensione della classe **`Instruction`** preesistente che prevede come:

membri alcune strutture dati necessarie alla conservazione delle informazioni sui cicli di clock spesi nelle unità funzionali attraversate o attendendo servizi dalle risorse utilizzate

metodi le funzioni di interfaccia per la manipolazione delle nuove strutture dati nonché per l'estrazione delle informazioni raccolte

[5] In termini di microistruzioni

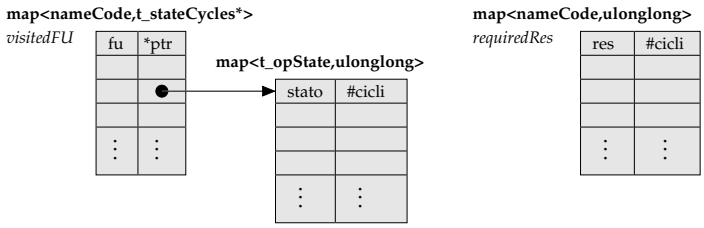


Figura 4.2: Strutture dati utilizzate per la raccolta dei dati sulle frequenze.

4.2.1 Strutture dati

Al fine di ottimizzare la soluzione si è fatto un largo uso delle strutture dati generiche fornite dalla [6] del C++. In particolare, come mostrato in Figura 4.2, sono state usate delle liste a puntatori implementate usando le **`STL::map<T>`**. Precisamente:

`StatInstruction::visitedFU`
: contiene una entry per ciascuna unità funzionale in cui l'istruzione è passata, distinguendo lo stato in cui si trovava

`StatInstruction::requiredRes`
: consente di sapere quali risorse ha usato l'istruzione e per quanti cicli di clock ha atteso un servizio da ciascuna

Queste strutture sono a visibilità privata, per il loro utilizzo vengono quindi forniti un insieme di metodi il cui utilizzo andiamo a descrivere in seguito.

4.2.2 Metodi

I metodi forniti dalla classe **`StatInstruction`** devono essere richiamati, all'interno del codice specifico di una architettura, quando si verificano particolari eventi. Il loro utilizzo tuttavia, come richiesto da uno dei requisiti, è abbastanza semplice in quanto localizzato sempre in alcuni punti specifici dei metodi **`execSingleCycle`**, **`execMultiCycle`** ed **`execInstruction`**, tutti definiti nella classe che si deriva da **`FunctionalUnit`** per implementare una particolare unità funzionale.

Con il supporto della Figura 4.3 vediamo in dettaglio come l'estensione si integra nel codice specifico di una architettura.

[6] Standard Template Library.
Una buona documentazione la si trova su `http://www.sgi.com/tech/stl/`

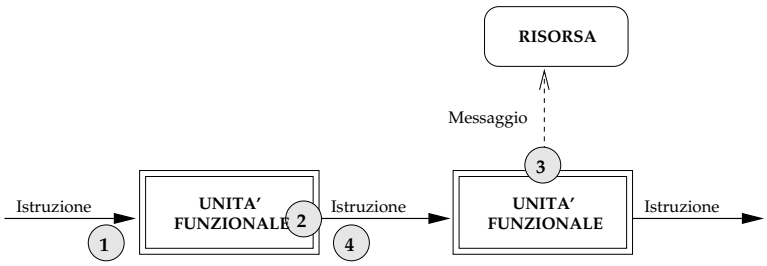


Figura 4.3: Punti d'innesto dell'estensione della classe **`Instruction`** nel codice specifico di una architettura.

① *accesso ad una unità funzionale*
in corrispondenza di **`FunctionalUnit::execSingleCycle`** si ha il fetch di una istruzione dalla coda d'ingresso dell'unità funzionale. A questo punto l'invocazione del metodo **`FunctionalUnit::fuAccess`** memorizza il ciclo di clock **`t`** di ingresso della istruzione nella unità funzionale **`FU`**.

② *pipeline stall*
ci si accorge di uno stallo della pipeline quando, terminata l'esecuzione di una istruzione, l'unità funzionale cerca di inserirla nella sua coda d'uscita e questa operazione fallisce. Ciò può accadere in due punti precisi, in corrispondenza di **`execSingleCycle`** o **`execMultiCycle`**. In questi casi l'invocazione di **`FunctionalUnit::fuStall`** consente di conteggiare correttamente i cicli spesi a causa di un pipeline stall all'interno di ogni unità funzionale.

③ *resource stall*
l'unità funzionale stalla in attesa di una risorsa quando quest'ultima non è disponibile oppure richiede più cicli di clock per fornire il servizio richiesto. Nel simulatore ci accorgiamo di ciò quando si tenta l'accesso alla risorsa e quindi solo in **`execInstruction`**. In corrispondenza del codice di accesso ad una risorsa, l'uso di **`FunctionalUnit::resAccess`** consente di conteggiare correttamente sia i cicli di clock necessari alla risorsa per servire l'istruzione che quelli per un eventuale stallo dell'unità funzionale in attesa del servizio richiesto.

④ *uscita da una unità funzionale*
quando una istruzione ha terminato la sua esecuzione e l'untià funzionale riesce ad inserirla nella coda d'uscita l'uso di **`FunctionalUnit::fuExit`**

consente di calcolare il giusto numero di cicli di clock spesi entro l'unità funzionale per la pura elaborazione[7]. Ciò avviene negli stessi punti del codice indicati per ②.

I metodi fin qui esposti sono stati inseriti in altrettante macro, dal nome equivalente, dichiarate nell'header della nuova classe **`Tracker`**[8]. In tal modo si riesce a mantenere più pulito il codice dipendente dall'architettura ed a guadagnare in prestazioni. Le macro si preoccupano infatti di disabilitare il codice dell'estensione quando la simulazione non lo richiede. In Figura 4.4 viene mostrato un esempio [9] di utilizzo delle macro dell'estensione, come si vedo il loro uso risulta molto semplice senza "sporcare" troppo il codice originale.

Quando l'istruzione raggiunge infine la coda d'uscita, come schematizzato in Figura 4.3, questa provvede automaticamente ad inserire nel file prodotto i valori delle frequenze secondo lo schema indicato in Figura 3.5 a pagina 62. A tale fine è stata sufficiente una piccola modifica al codice di **`OutputQueue::insert`** in cui vengono scandite le liste **`visitedFU`** e **`requiredRes`** associate all'istruzione corrente e stampati i dati presenti usando i metodi: **`getFuClkCycs`** e **`getResClkCycs`** e entrambi di **`StatInstruction`**. L'ordine con il quale vengono prodotti i dati delle frequenze coincide con quello che si è usato per dichiarare le unità funzionale e le risorse. In particolare, per i componenti che intendiamo tracciare, nella *procedura di configurazione*[10] del simulatore si devono usare due funzioni specifiche in luogo di **`Utility::declareName`**. Il metodo **`Utility::declareFU`** va usato per le unità funzionali mentre per le risorse **`Utility::declareRes`**.

4.3 Estensione TrIBeS per il tracciamento della potenza assorbita

L'estensione del simulatore per il tracciamento della potenza assorbita ogni ciclo di clock ha richiesto un intervento più significativo sulla struttura delle classi base dell'SDK fornito inizialmente da TrIBeS. Tuttavia, l'uso dei concetti tipici della programmazione ad oggetti ha consentito comunque di elaborare una soluzione sufficientemente elegante nel rispetto dei requisiti definiti ad inizio capitolo.

[7] I valori in caso di pstall e rstall sono calcolati automaticamente nei punti precedenti, durante la permanenza dell'istruzione dentro l'unità funzionale

[8] La vedremo in dettaglio nel prossimo paragrafo dato che viene usata principalmente per il tracciamento della potenza nella seconda fase

[9] L'esempio rappresenta l'implementazione dei **`FunctionalUnit::execSingleCycle`** nel caso dell'unità funzionale `FetchUnit` per l'architettura *ARM7TDMI* implementata agli scopi di questo lavoro di tesi.

[10] La descrizione della routine di configurazione viene rimandata al Par. 4.3.1 a pagina 77.

```
  void FetchUnit::execSingleCycle( ulonglong clock ) {
    while( prefetchBuffer.size() < 2 ) {
      Instruction* tmpInstr = new Instruction();
      if( (queueIn->extract( &tmpInstr ) == true ) ) {
        prefetchBuffer.push_front( tmpInstr );
      } else {
        delete tmpInstr;
        break;
      }
    }
    int i = 0;
    while( (i < queueInternalSize) &&
      (prefetchBuffer.size() > 0) ) {
        queueInternal[i] = prefetchBuffer.back();
        prefetchBuffer.pop_back();
① ☞     FU_ACCESS ();
        queueInternal[i]->setCounter( 0 );
        queueInternal[i]->setTin( clock );
        if ( execInstruction( i ) ) {
          if ( queueOut[queueOutDefault]
              ->insert(queueInternal[i]) ) {
④ ☞         FU_EXIT ();
            queueInternal[i] = NULL;
          } else {
② ☞         FU_STALL ();
          }
        }
        i++;
    }
  }
```

Figura 4.4: *Esempio dell'uso delle macro dell'estensione per la raccolta dati nel codice dipendente dall'architettura.*

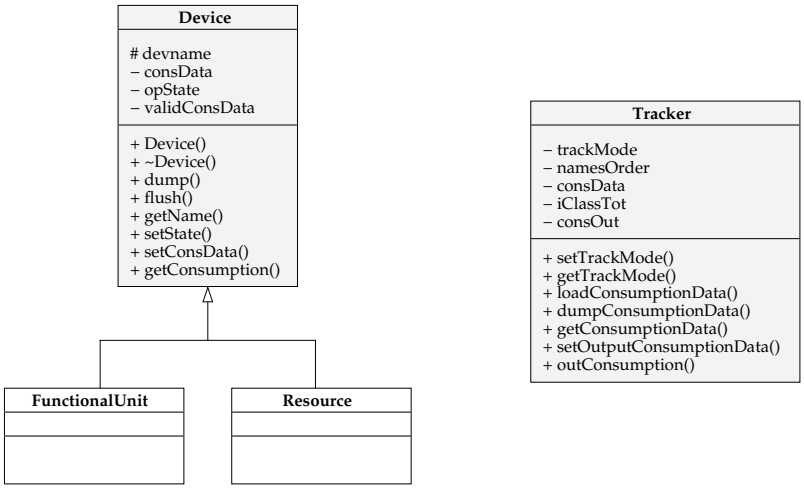


Figura 4.5: Diagramma UML che descrive le classi usate per la fase di tracciamento della potenza assorbita.

La soluzione proposta è scaturita dalle seguenti considerazioni. All'interno della pipeline ci sono delle componenti che consumano potenza in funzione di alcuni parametri come l'istruzione che le usa o lo stato della pipeline. Precisamente queste componenti sono le unità funzionali e le risorse mentre le code, nel modello formulato nel Cap. 2, non assorbono energia. Queste componenti devono essere quindi dotate di una caratterizzazione di potenza in funzione dei parametri del modello. Notiamo inoltre che le risorse vengono contattate dalle unità funzionali che stanno processando l'istruzione che ne fanno richiesta.

L'estensione proposta si concretizza nell'aggiunta di due classi come mostrato in Figura 4.5. Vediamole in dettaglio:

`Device`
: questa classe definisce il concetto di componente del simulatore caratterizzato da un assorbimento di potenza, in funzione dei parametri del modello. Di seguito chiameremo *device* questi componenti. Da tale classe quindi, in ragione delle considerazioni fatte sopra, vengono derivate le classi **`FunctionalUnit`** e **`Resource`**. In questo modo riusciamo ad includere elegantemente il codice per il tracciamento della potenza assorbita nelle classi opportune.

`Tracker`
: è fondamentalmente una classe (statica) di utilità fornita a supporto del tracciamento degli assorbimenti. Mantiene le informazioni sulle caratteristiche d'assorbimento di ogni device del simulatore, raccolte all'interno

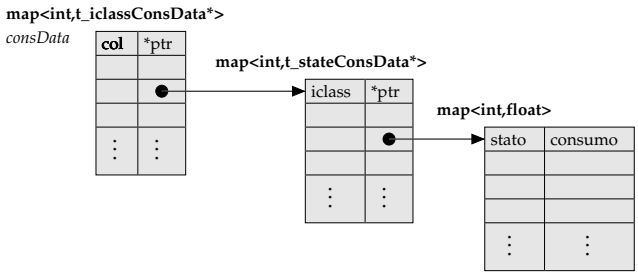


Figura 4.6: Strutture dati utilizzate per la conservazione della caratterizzazione in potenza assorbita dei "device" del simulatore.

della struttura dati **`Tracker::consData`** costituita, ancora una volta, con una opportuna combinazione di liste dinamiche di classe **`STL::map<T>`** organizzate come mostra la Figura 4.6.

In aggiunta, una piccola modifica a **`FunctionalUnit::execCycle`** è stata fatta facendolo passare da:

```
virtual void execCycle (ulonglong clock);
```

a:

```
virtual float execCycle (ulonglong clock)=0;
```

L'idea alla base del funzionamento del codice per il tracciamento del consumo di potenza è incentrata intorno alla possibilità di rilevare il consumo come somma di contributi associati alle sole unità funzionali. Il funzionamento del simulatore prevede infatti, per ciascun ciclo di clock, l'esecuzione di un passo di elaborazione su ciascuna delle unità funzionali. Le unità funzionali devono essere aggiornate per quanto riguarda il loro stato, al fine di fornire il corretto valore di potenza assorbita ad ogni ciclo. Inoltre, sappiamo che le risorse vengono "interrogate" dalle unità funzionali se l'istruzione che stanno elaborando lo richiede.

Possiamo quindi pensare che, per ciascun ciclo di elaborazione, siano le stesse unità funzionali a ritornare il proprio consumo, in funzione dello stato e dell'istruzione corrente, già sommato a quello della risorsa che eventualmente stanno utilizzando. Lo schema di funzionamento del simulatore visto in Figura 2.11 a pagina 45 verrà quindi modificato in quello mostrato in Figura 4.7.

Vediamo ora dove è necessario inserire il nuovo codice, nella parte dipendente dall'architettura, al fine di generare il tracciato degli assorbimenti di potenza.

```
ordinaUnitàFunzionali();
foreach unità funzionale do
 | consumo += execute_cycle();
end
foreach coda do
 | update();
end
foreach risorsa do
 | update();
end
stampa(consumo)
```

Figura 4.7: *Simulazione di un ciclo di clock con tracciamento della potenza assorbita.*

4.3.1 Configurazione di una architettura

La routine di configurazione di una architettura, rappresentata dalla funzione **`ConfigureArchitecture`**[11], dev'essere aggiornata in tre punti:

1. *dichiarazione dei device*
   una delle prime operazioni da effettuare in fase di configurazione è la dichiarazione di unità funzionali, risorse e code in modo da generare un hash numerico utilizzato come indice delle stesse all'interno del simulatore. A tale scopo l'SDK di TrIBeS fornisce il metodo **`declareName`**. L'estensione fornisce due metodi, che sono sostanzialmente un wrapper a quest'ultimo, da usare per la dichiarazione delle unità funzionali (**`declareFU`**) e delle risorse (**`declareRes`**) che intendiamo mettere sotto il controllo del sistema di tracciamento dei consumi. Questi metodi si occuperanno inoltre di inizializzare le strutture dati necessarie al tracciamento.

2. *caricamento delle caratterizzazioni di consumo*
   la caratterizzazione in consumo delle istruzioni dell'instruction set deve essere caricata da file mediante **`Tracker::loadConsumptionData`**. Il metodo **`Tracker::setOutputConsumptionData`** consente invece di definire il file nel quale dovrà essere salvata la traccia dei consumi prodotta durante la simulazione.

3. *definizione dei consumi dei device*
   a ciascun *device* dovrà quindi essere assegnata la propria caratterizzazio-

[11] Tale funzione dev'essere esportata dalla libreria che implementa il supporto per una particolare architettura. Tipicamente viene definita all'interno di un file C con nome `Config<architettura>.cpp`.

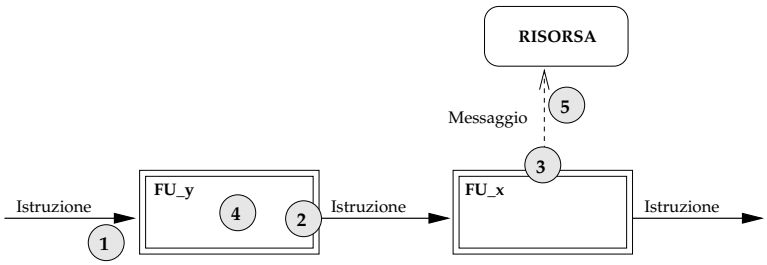


Figura 4.8: Punti d'innesto dell'estensione per il tracciamento dei consumi nel codice specifico di una architettura.

ne di consumo mediante il metodo **`device::setConsData`**. Quest'ultimo sostanzialmente passa al device il corretto riferimento all'interno della **`Tracker::consData`** che a sua volta può essere ottenuto dal metodo statico **`Tracker::getConsumptionData`**.

4.3.2 Definizione comportamentale di una architettura

Una volta configurata una architettura secondo le direttive indicate precedentemente i device sono in grado di fornirci i consumi ad ogni ciclo di clock in funzione dei parametri del modello, ovvero: l'istruzione che stanno elaborando[12] e lo stato in cui si trovano[13].

Nella definizione del comportamento di ogni componente è tuttavia necessario inserire del codice che consenta sia di aggiornarne lo stato che di ritornare i contributi di consumo richiesti. Ancora una volta l'approccio che abbiamo seguito è quello di definire delle macro che incapsulano alcuni metodi pensati allo scopo. Con l'aiuto della figura Figura 4.8 vediamo in dettaglio come l'estensione si integra nel codice specifico di una architettura.

Tre sono i punti in cui è necessario intervenire per tenere traccia del cambiamento di stato di una unità funzionale[14]:

① *accesso ad una unità funzionale*
in corrispondenza di **`FunctionalUnit::execSingleCycle`** si ha il fetching di una istruzione dalla coda d'ingresso dell'unità funzionale. Quan-

[12]Nel caso delle risorse si intende l'istruzione che ne richiede un servizio.

[13]Ricordiamo che il concetto di stato è definito solo per le unità funzionali e non per le risorse, tuttavia l'estensione proposta è generalizzabile in tal senso.

[14]Ricordiamo ancora che lo stato è definito solo per le unità funzionali ed ha tre possibili valori, chiamati nel codice *working*, *waitFU* e *waitRes*, a seconda che l'unità funzionale stia rispettivamente: elaborando una istruzione, in stallo di pipeline oppure in attesa di un servizio da parte di una risorsa.

do una istruzione entra in una unità funzionale quest'ultima, almeno per un ciclo di clock, la elabora. Dovremo tenere conto di questo stato quando ritorniamo il consumo per il ciclo corrente. A questo punto quindi l'invocazione del metodo **`FunctionalUnit::setState( OS_working )`** riporta su "working" lo stato dell'unità funzionale. La macro che racchiude questa chiamata è **`FU_ACCESS`**.

② *pipeline stall*
ci si accorge di uno stall della pipeline quando, terminata l'esecuzione di una istruzione, l'unità funzionale cerca di inserirla nella sua coda d'uscita e questa operazione fallisce. Ciò può accadere in due punti precisi, in corrispondenza di **`execSingleCycle`** o **`execMultiCycle`**. In questi casi l'invocazione di **`FunctionalUnit::setState( OS_waitFU )`** consente di impostare correttamente lo stato per la rilevazione dei consumi nel ciclo di clock successivo. Questa funzione è richiamabile con la macro **`FU_STALL`**.

③ *resource stall*
l'unità funzionale stalla in attesa di una risorsa quando quest'ultima non è disponibile oppure richiede più cicli di clock per fornire il servizio richiesto. Nel simulatore ci accorgiamo di ciò quando si tenta l'accesso alla risorsa e quindi solo in **`execInstruction`**. In corrispondenza del codice di accesso ad una risorsa, l'uso di **`FunctionalUnit::setState( OS_waitRes )`** consente di aggiornare lo stato dell'unità funzionale coerentemente al modello. La macro di wrapping è **`RES_STALL`**.

La determinazione del consumo effettivo in ogni ciclo di clock[15] si ha invece sommando il contributo dell'unità funzionale con quello delle eventuali risorse che utilizza. I punti di interesse nel codice sono due:

④ *esecuzione di una istruzione*
in corrispondenza del metodo **`FunctionalUnit::execInstruction`** si ha la simulazione dell'effettiva esecuzione di una istruzione con il consumo delle microistruzioni e l'eventuale accesso alle risorse richieste. Quando tale funzione viene eseguita lo stato dell'unità funzionale per il ciclo corrente è già definito e possiamo quindi sommare il suo contributo mediante l'invocazione di **`FunctionalUnit::getConsumption`**. La macro che si occupa di ciò è **`FU_TRACKING`**.

⑤ *accesso ad una risorsa*
sempre nell'ambito della **`FunctionalUnit::execInstruction`** si ha la possibilità di usare le risorse cui l'unità funzionale da accesso. In questi

[15] Che corrisponde ad una iterazione del ciclo di Figura 4.7.

casi, oltre a consumare la relativa microistruzione, ci si deve preoccupare di sommare al consumo dell'unità funzionale quello della risorsa cui si sta per accedere. In corrispondenza di ogni punto d'accesso alle risorse sarà quindi necessario invocare il metodo **`Resource::getConsumption`** "wrappato" dalla macro **`RES_TRACKING(resource)`**.

In Figura 4.9 mostriamo un esempio[16] di uso delle macro dell'estensione. Ancora una volta possiamo apprezzare come siano stati soddisfatti i requisiti richiesti nella soluzione trovata, specialmente quello di semplicità.

4.4 Schema generale della soluzione

Abbiamo discusso nei paragrafi precedenti l'estensione dell'SDK di TrIBeS al fine di supportare il nuovo modello per il tracciamento della potenza assorbita per clock cycle. In Figura 4.10 viene mostrato lo schema generale per l'utilizzo del nuovo modello, come vediamo esistono fondamentalmente due fasi:

① *tuning*

in questa prima fase il simulatore comportamentale viene usato al fine generare i dati sui tempi di residenza/uso di unità funzionali e risorse da parte di ciascuna istruzione della traccia d'ingresso `tune.str`. I dati così prodotti, assieme alla caratterizzazione dell'instruction set dell'architettura simulata, vengono elaborati dal modello per il ribaltamento dei costi che abbiamo deciso di utilizzare[17]. Come risultato otteniamo una caratterizzazione, in funzione dei parametri del modello, delle unità funzionali e delle risorse presenti nel simulatore. Per attivare questa fase basta[18] avviare TrIBeS con l'opzione `-s`.

② *tracing*

supponendo di disporre della caratterizzazione ottenuta nella prima fase, possiamo procedere al tracciamento della potenza assorbita da un programma semplicemente fornendogli la caratterizzazione assieme alla trac-

[16] L'esempio rappresenta l'implementazione di **`FunctionalUnit::execInstruction`** nel caso dell'unità funzionale `FetchUnit` per l'architettura *ARM7TDMI* implementata agli scopi di questo lavoro di tesi.

[17] Nel nostro caso facciamo una stima ai minimi quadrati dei parametri del modello come descritto nel Par. 3.3.1

[18] Supponendo di aver correttamente integrato le macro illustrate nel Par. 4.2.2 all'interno del codice di definizione del comportamento dell'architettura usata per la simulazione.

```
bool FetchUnit::execInstruction( int pos ) {
  int  my_name   = queueIn->getName();
  int  mCodeSize = queueInternal[pos]
           ->getMicroCodeSize();
④ ☞ FU_TRACKING ();
  for( int i = 0; i < mCodeSize; i++ ) {
    MicroInstruction* mInstr =
      queueInternal[pos]->getMicroCode( my_name );
    int opcode = GET_OPCODE(mInstr->code);
    int param  = GET_PARAM(mInstr->code);
    switch( opcode ) {
      case LOAD_OPCODE:
⑤ ☞     RES_TRACKING (memory);
        if( memory->read( param, 1,
                  queueInternal[pos]->getId() )
            == false ) {
③ ☞       RES_STALL ();
          return false;
        }
      break;
      case FORWARD:
        delete mInstr;
        return true;
      default:
        return true;
    }
    queueInternal[pos]->removeMicroInstruction( mInstr );
  }
  return true;
}
```

Figura 4.9: *Esempio dell'uso delle macro per il tracciamento dei consumi dell'estensione nel codice dipendente dall'architettura.*

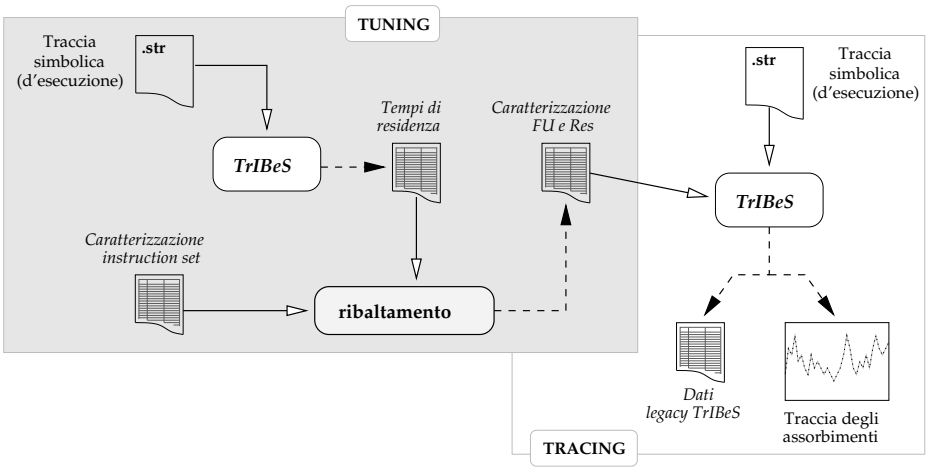


Figura 4.10: Schema generale per l'utilizzo del nuovo modello di tracciamento dei consumi.

cia da simulare. Quel che viene prodotto[19] è un file contenente, per ciascun ciclo di clock, il consumo stimato dell'intera pipeline[20].

Bisogna osservare che la prima fase dev'essere eseguita una sola volta per ciascuna diversa architettura. Ottenuta la caratterizzazione delle unità funzionali e delle risorse, al fine del tracciamento della potenza assorbita per ciclo di clock da una generico programma basta infatti applicare direttamente la seconda fase. Questo è piuttosto vantaggioso dato che la fase computazionalmente più pesante è proprio la prima, in particolare per quanto riguarda l'analisi matematica dei dati con il ribaltamento dei costi.

Prestazioni dell'estensione

Al fine di giudicare l'impatto del nuovo codice nelle prestazioni del simulatore riportiamo in Tabella 4.11 una comparazione delle prestazioni del vecchio codice del simulatore con quello della nuova versione. In particolare riportiamo le prestazioni del nuovo codice usando il simulatore in "legacy mode", ovvero senza raccolta dei tempi di residenza o tracciamento della potenza[21], nel modo di tuning ed in quello tracing.

[19] Oltre al classico file di output di TrIBeS con le informazioni necessarie al modello di caratterizzazione di un instruction set per cui il simulatore è stato inizialmente introdotto.

[20] Ci preoccuperemo nel prossimo capitolo di determinare la qualità della previsione ottenuta oltre che di valutare la sua usabilità nell'ambito dell'analisi agli attacchi in potenza.

[21] In questa modalità il simulatore genera gli stessi dati prodotti dalla versione precedente alle modifiche introdotte

Versione simulatore	Prestazioni	Variazione %
TrIBeS 3.0	4 Kins/s	–
TrIBeS 3.4 in legacy mode	4 Kins/s	0%
TrIBeS 3.4 in tuning	3.6 Kins/s	−10%
TrIBeS 3.4 in tracing	3.4 Kins/s	−15%

Figura 4.11: *Comparazione delle prestazioni nelle diverse configurazioni.*

I test sono stati condotti utilizzando un doppio Pentium*III* a 966*MHz* con 1*GB* di memoria di lavoro, gestito da Linux Slackware. Sulla implementazionedell'architettura $ARM7TDMI$[22] è stata simulata l'esecuzione di una implementazione del DES che corrispondeva ad una traccia di 5'743'806 istruzioni assembler. Le prestazioni della simulazione nelle diverse versioni/configurazioni di TrIBeS sono riportate in ella.

4.4.1 Ribaltamento dei costi

A supporto della elaborazione dei dati sui tempi di residenza, prodotti nella prima fase di tuning, sono stati sviluppati una serie di tools che implementano il metodo di stima ai minimi quadrati. Questi strumenti sono rappresentati per lo più da script BASH che elaborano le informazioni secondo le due modalità discusse nel Par. 3.3.1 a pagina 66, ovvero: l'analisi *"class based"* e quella *"classless"*.

In Figura 4.12 viene presentato **`getConsumption_classbased.sh`**.
Tale script, presa in ingresso la caratterizzazione dell'instruction set e la traccia di tuning generata da TrIBeS, provvede a risolvere tanti sistemi ai minimi quadrati, uno per ciascuna classe di istruzioni definita, producendo una caratterizzazione delle unità funzionali e delle risorse funzione della classe di istruzioni. La risoluzione effettiva del problema ai minimi quadrati è affidata a *GNU Octave*[23] generando dinamicamente un *m-file* per ciascuna classe di istruzioni da analizzare.

In Figura 4.13 viene invece mostrato **`getConsumption_classless.sh`**.
Gli input sono gli stessi del primo script ma ora cambiano le modalità di elaborazione dei dati generati da TrIBeS, in particolare viene impostato, all'interno di

[22] Sviluppata agli scopi di questa tesi e discussa in Appendice A.

[23] Tools e linguaggio d'alto livello per l'elaborazione numerica dei dati. Si veda il sito `http://www.octave.org/` per maggiori dettagli.

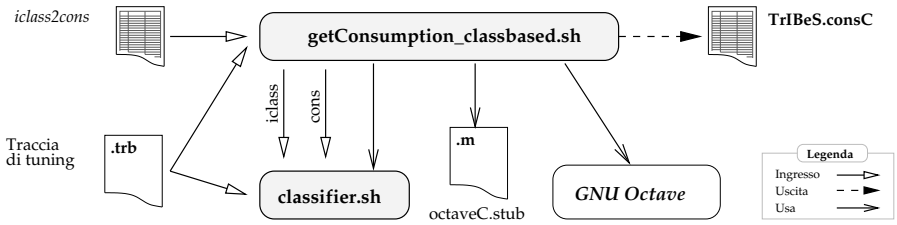


Figura 4.12: Struttura dei tools di analisi "class based".

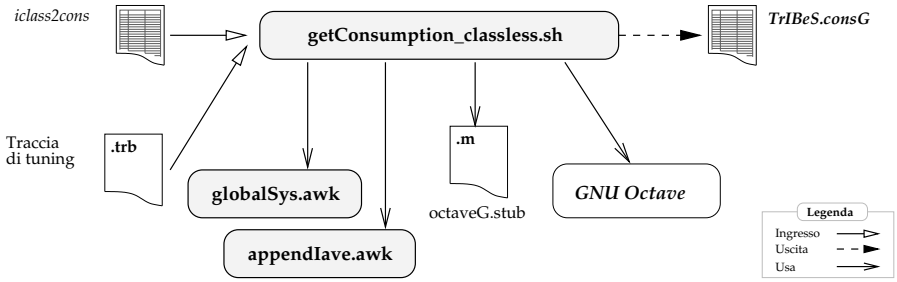


Figura 4.13: Struttura dei tools di analisi "classless".

un *m-file*, un unico problema ai minimi quadrati risolto poi sempre con l'uso di *GNU Octave*.

Le figure servono a rappresentare graficamente le diverse relazioni fra gli script utilizzati e i files su cui operano, per il dettaglio del funzionamento degli stessi rinviamo poi alla documentazioni che è stata inclusa in testa ad ognuno.

CAPITOLO 5

Risultati sperimentali

"Gli errori sono inevitabili;
il vantaggio è che sbagliando si impara"
Massima popolare

QUANTO visto, a livello teorico-implementativo, nei capitoli precedenti viene messo in pratica in quest'ultimo. L'obbiettivo principale è quello di verificare la validità della soluzione proposta e mostrarne le possibilità d'impiego.

Dapprima viene quindi effettuata la calibrazione del modello, creando una opportuna traccia di tuning da usare al fine di individuare i valori dei parametri. Successivamente si studia la bontà della caratterizzazione individuata mettendo in evidenza le differenze che sussistono fra i diversi possibili approcci per la caratterizzazione, ovvero quello class based e quello classless.

Seguono una serie di esempi didattici finalizzati sia a mostrare le possibilità di analisi della traccia di potenza che a valutare ulteriormente le differenze che sussistono fra le tracce generate con diverse caratterizzazioni.

Infine, nell'ultimo paragrafo, vengono presentati un paio di esempi concreti di SPA prima con l'analisi delle macrocaratteristiche di una implementazione del DES e successivamente con un piccolo ma significativo esempio di attacco ad una implementazione debole di un sistema di cifratura.

5.1 Tuning del modello

Al fine di validare il modello sviluppato è stato necessario anzitutto procurarsi una caratterizzazione delle unità funzionali e delle risorse in funzione dei parametri del modello stesso. Questa caratterizzazione, come spiegato nel Par. 3.3.1, si ottiene mediante un processo di *ridistribuzione dei costi* medi statici a partire da una serie di informazioni sulla permanenza delle istruzioni nelle singole unità funzionali e di utilizzo delle risorse. A loro volta, le informazioni necessarie al processo di ridistribuzione dei costi vengono generate da una *traccia di tuning* . Sostanzialmente si tratta di una traccia TrIBeS con alcune caratteristiche particolari:

- ❏ un numero di microistruzioni sufficiente a coprire tutte le possibili classi di istruzioni presenti nell'instruction set analizzato
- ❏ il maggior numero possibile di sequenze di microistruzioni diverse al fine di analizzare tutte le possibili interazioni fra istruzioni

Il modo migliore per avvicinarsi il più possibile ad entrambi i requisiti è quello di considerare una traccia contenente un elevato numero di istruzioni generata a partire dal tracciamento di algoritmi appartenenti a domini applicativi diversi. Abbiamo quindi provveduto a selezionare un certo numero di algoritmi, precisamente la scelta è ricaduta su:

bsort ordinamento[1] di un vettore di 100 numeri usando l'algoritmo bubblesort

qsort ordinamento[1] di un vettore di 100 numero usando l'algoritmo quicksort

rle calcola la "redundancy length encoding" di stringhe da 128 bit generate casualmente

crc16 calcola il codice ciclico di ridondanza su stringhe generate casualmente

md5 determina il digest di stringhe da 500 caratteri generate in modo casuale usando l'algoritmo md5

fft esegue la trasformata di Fourier

aes esegue il test di Monte Carlo[2] su una implementazione dell'AES

[1] L'operazione viene ripetuta più volte caricando ogni volta con diversi valori generati casualmente il vettore da ordinare.

[2] Il test consiste fondamentalmente nell'applicare più volte la cifratura di stringhe diverse per confrontarne poi i risultati.

Algoritmo	Dimensione traccia
bsort	12363156
qsort	2798716
rle	1065162
crc16	1261506
md5	2337
fft	19692183
aes	3794334
Totale:	40977394

Figura 5.1: *Numero di istruzioni nelle singole tracce usate per costruire la traccia di tuning.*

In Tabella 5.1 è mostrata la dimensione delle singole tracce generate. Tali tracce sono poi state unite a formare un'unica grande traccia, da usare per la generazione dei dati di tuning, che vediamo essere costituita da più di quaranta milioni di microistruzioni.

Pur essendo la traccia di tuning molto grande, non è stato possibile coprire tutte le istruzioni e classi di istruzioni disponibili per l'architettura scelta[3]. In Figura 5.2 viene studiata la copertura dal punto di vista del tipo di istruzioni. L'$ARM7TDMI$ dispone di[4] 59 diverse istruzioni, come mostra la figura[5] tuttavia non tutte le istruzioni sono rappresentate all'interno della traccia e la distribuzione di frequenza di quelle presenti non è molto uniforme. Le istruzioni maggiormente presenti sono quelle di tipo 1, che nelle nostra implementazione corrispondono alle istruzioni `MOV`[6], e nel complesso non siamo riusciti a far comparire all'interno della traccia di tuning meno del 30% delle possibili istruzioni.

Raggruppando le istruzioni per classi ed effettuando lo stesso studio i risultati non sono molto diversi. Le diverse classi di istruzioni definite per l'$ARM7TDMI$ sono 47, il numero è così elevato poiché per quasi ciascuna istruzione disponiamo di una caratterizzazione statica del consumo medio per ciclo di clock. Possiamo

[3]Ricordiamo essere l'$ARM7TDMI$. Si rimanda alla Appendice A per i dettagli sulla sua implementazione.

[4]Operando in ARM mode.

[5]In realtà, pur avendo adottato una scala logaritmica, alcune istruzioni avevano una frequenza così bassa da non essere visualizzabili all'interno del grafico.

[6]L'associazione fra istruzione assembly ed indice numerico viene operata da Atomic al fine di ottimizzare le elaborazioni successive.

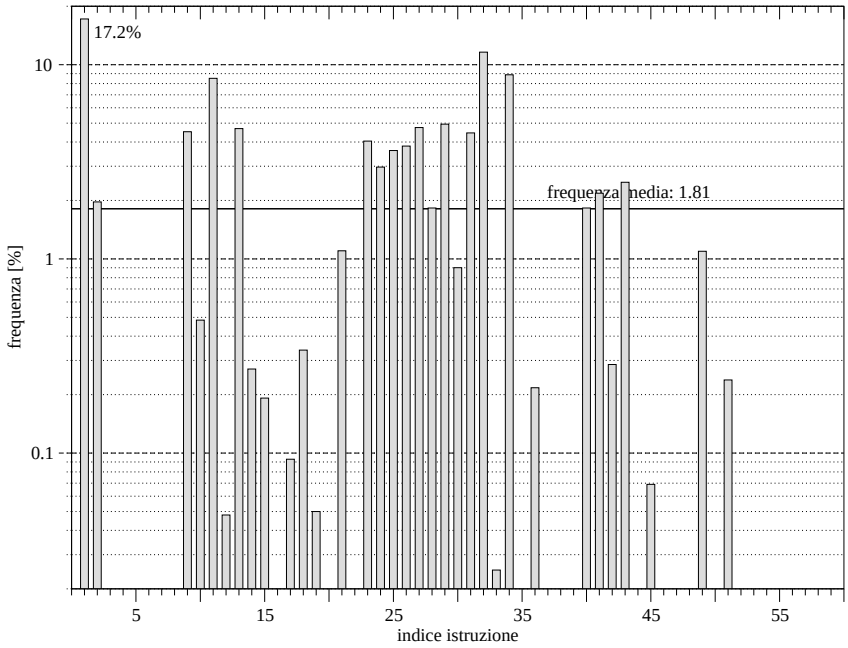


Figura 5.2: Percentuale di presenza delle istruzioni nella traccia di tuning.

osservare tuttavia dalla Figura 5.3 una copertura più estesa, intorno all'80%, ed uniforme. Il massimo si attesta, con la medesima percentuale, sulla classe 11 che corrisponde proprio alle istruzioni di `MOV`.

5.1.1 Ridistribuzione dei costi

La traccia di tuning generata, seguendo il procedimento descritto nella Sez. 4.4, è stata simulata con TrIBeS. I dati sulle frequenze così ricavati sono poi stati elaborati con degli script come quelli descritti in Figura 4.12 e Figura 4.13 a pagina 84, ottenendo dei sistemi di equazioni da risolvere ai minimi quadrati. La Figura 5.4 mostra il numero di equazioni diverse che sono state ottenute per ciascuna classe di istruzioni. In totale le equazioni distinte sono 523, il loro numero è quindi adatto per impostare un problema ai minimi quadrati di tipo *classless*.

Avendo 11 variabili da stimare[7], e volendo risolvere il problema ai minimi quadrati anche nel caso *class based*, sono necessarie almeno 11 equazioni distinte per ciascuna classe di istruzioni che vogliamo caratterizzare. Come mostrato in

[7]Questo numero deriva dalla particolare architettura che stiamo considerando ed in particolare da come abbiamo implementato il simulatore TrIBeS proprio per l'$ARM7TDMI$. Precisamente rappresenta la somma del numero di unità funzionali, in ciascuno dei tre possibili stati, più quello delle risorse.

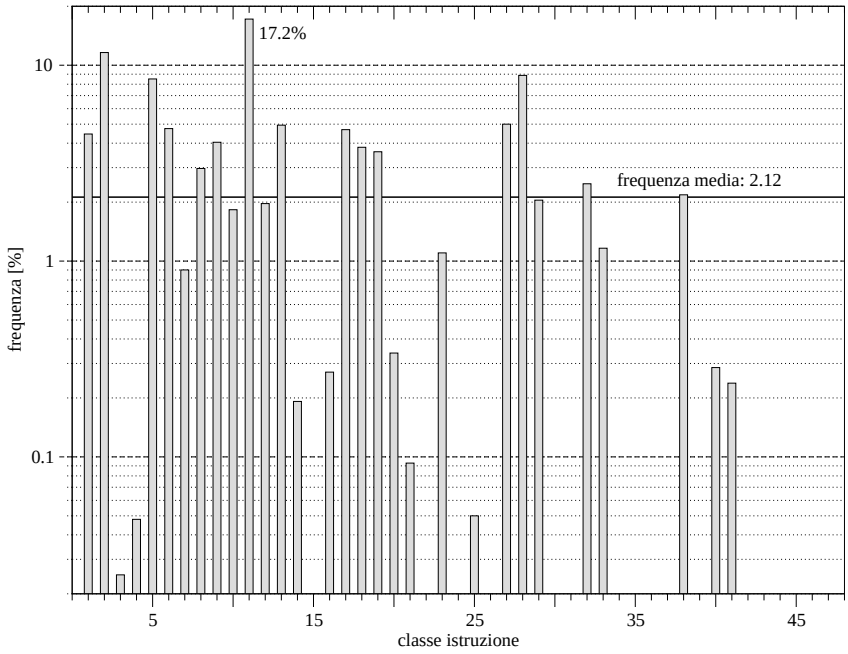


Figura 5.3: Percentuale di presenza delle classi di istruzioni nella traccia di tuning.

figura quindi non tutte le classi di istruzioni sono adatte ad essere caratterizzate ai minimi quadrati. Per quanto riguarda l'approccio class based procediamo quindi in questo modo, risolviamo il problema ai minimi quadrati per le classi in cui risulta ben posto ed utilizziamo invece la caratterizzazione classless per le altre classi.

Alla fine di queste elaborazioni abbiamo ottenuto la caratterizzazione energetica, secondo i parametri del nostro modello, per le unità funzionali e le risorse utilizzate dal simulatore. Tali risultati vengono mostrati in Tabella 5.5 nella quale, per ciascuna classe di istruzione viene riportato il consumo statico, ovvero il nostro dato di partenza, e la caratterizzazione ottenuta con il processo di ridistribuzione dei costi. Alla caratterizzazione class based[8] si riferiscono le prime 44 righe[9], l'ultima riga rappresenta invece la caratterizzazione classless che risulta, per definizione, indipendente dalla particolare classe di istruzione.

Gli stessi dati sono stati raccolti all'interno dei due file chiamati *tribes.consC* e *tribes.consG*[10], e verranno utilizzati per la generazione delle tracce nella fase

[8]Per le classi per la quali non è stato possibile risolvere il problema ai minimi quadrati si è considerata la caratterizzazione classless.

[9]Le classi dalla 44 alla 59 sono state collassate in un'unica riga dato che hanno lo stesso valore.

[10]Rispettivamente per la caratterizzazione class based e per quella classless.

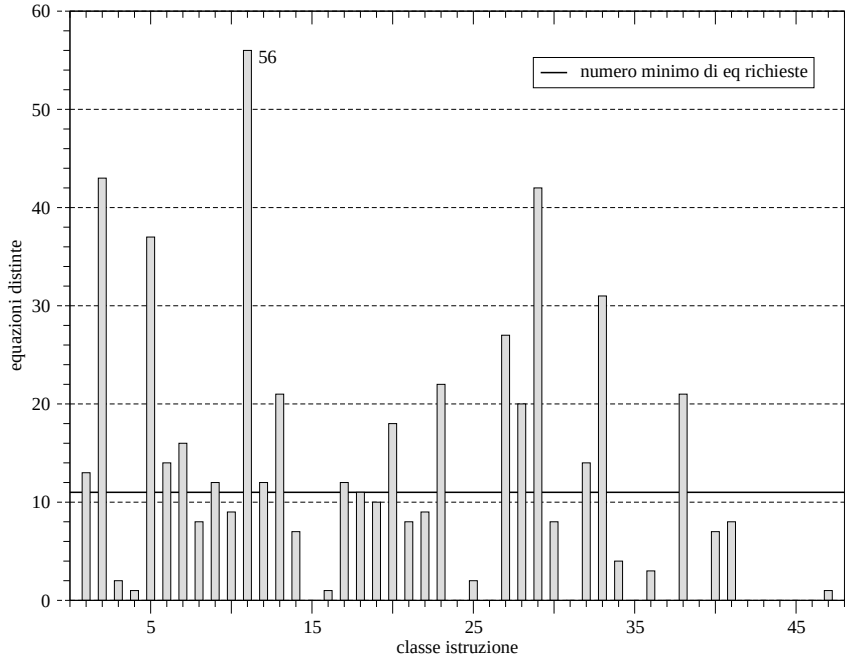


Figura 5.4: Numero di equazioni distinte per ciascuna classe di istruzioni.

successiva della nostra analisi, ovvero la convalidazione della caratterizzazione ottenuta.

5.2 Validazione del modello

Al fine di validare la caratterizzazione energetica ottenuta nella fase di tuning abbiamo effettuato uno studio mirato a verificare la bontà della previsione di energia consumata da un programma utilizzando la caratterizzazione individuata. Precisamente, abbiamo valutato lo scostamento fra la previsione di consumo ottenuta utilizzando il metodo statico classico, ovvero come somma del consumo delle singole istruzioni eseguite, e quella fornita dal simulatore TrIBeS modificato, definita dalla somma del consumo per singolo ciclo di clock.

Nella Figura 5.6 viene riportato il consumo stimato per l'esecuzione di ciascuno degli algoritmi utilizzati nella stessa fase di tuning. Il consumo è stato normalizzato rispetto a quello determinato in modo classico riportando quindi nella figura le energie stimate in percentuale. Da tale raffronto emergono due considerazioni:

- la caratterizzazione classless tende sempre a *sovrastimare* il consumo energetico

	Istr#	IS cons	Fetch			Decode			Execute			Thumb	Memory
			Ex	Ps	Rs	Ex	Ps	Rs	Ex	Ps	Rs		
Caratterizzazione class less	1	11.210	6.726	2.242	11.210	6.726	0.000	11.210	11.210	0.000	0.000	0.000	8.968
	2	12.070	7.242	2.414	12.070	7.242	0.000	12.070	12.070	0.000	0.000	0.000	9.656
	3	13.300	8.258	8.386	10.996	8.258	0.000	10.681	15.443	9.940	0.000	0.000	1.994
	4	10.880	8.258	8.386	10.996	8.258	0.000	10.681	15.443	9.940	0.000	0.000	1.994
	5	10.700	6.420	2.140	10.700	6.420	0.000	10.700	10.700	0.000	0.000	0.000	8.560
	6	12.060	8.258	8.386	10.996	8.258	0.000	10.681	15.443	9.940	0.000	0.000	1.994
	7	9.100	8.258	8.386	10.996	8.258	0.000	10.681	15.443	9.940	0.000	0.000	1.994
	8	8.550	8.258	8.386	10.996	8.258	0.000	10.681	15.443	9.940	0.000	0.000	1.994
	9	8.830	5.298	1.766	8.830	5.298	0.000	8.830	8.830	0.000	0.000	0.000	7.064
	10	11.610	8.293	1.659	11.610	8.293	0.000	11.610	8.293	0.000	0.000	0.000	9.951
	11	10.270	6.162	2.054	10.270	6.162	0.000	10.270	10.270	0.000	0.000	0.000	8.216
	12	13.220	9.443	1.889	13.220	9.443	0.000	13.220	9.443	0.000	0.000	0.000	11.331
	13	12.100	7.260	2.420	12.100	7.260	0.000	12.100	12.100	0.000	0.000	0.000	9.680
	14	13.920	8.258	8.386	10.996	8.258	0.000	10.681	15.443	9.940	0.000	0.000	1.994
	15	13.900	8.258	8.386	10.996	8.258	0.000	10.681	15.443	9.940	0.000	0.000	1.994
	16	11.330	8.258	8.386	10.996	8.258	0.000	10.681	15.443	9.940	0.000	0.000	1.994
	17	11.400	6.840	2.280	11.400	6.840	0.000	11.400	11.400	0.000	0.000	0.000	9.120
	18	9.010	8.258	8.386	10.996	8.258	0.000	10.681	15.443	9.940	0.000	0.000	1.994
	19	9.200	8.258	8.386	10.996	8.258	0.000	10.681	15.443	9.940	0.000	0.000	1.994
	20	14.450	8.258	8.386	10.996	8.258	0.000	10.681	15.443	9.940	0.000	0.000	1.994
	21	13.370	8.258	8.386	10.996	8.258	0.000	10.681	15.443	9.940	0.000	0.000	1.994
	22	15.050	8.258	8.386	10.996	8.258	0.000	10.681	15.443	9.940	0.000	0.000	1.994
	23	14.230	8.258	8.386	10.996	8.258	0.000	10.681	15.443	9.940	0.000	0.000	1.994
	24	14.110	8.258	8.386	10.996	8.258	0.000	10.681	15.443	9.940	0.000	0.000	1.994
	25	14.780	8.258	8.386	10.996	8.258	0.000	10.681	15.443	9.940	0.000	0.000	1.994
	26	10.100	8.258	8.386	10.996	8.258	0.000	10.681	15.443	9.940	0.000	0.000	1.994
	27	10.490	8.258	8.386	10.996	8.258	0.000	10.681	15.443	9.940	0.000	0.000	1.994
	28	11.240	2.248	2.248	11.240	2.248	0.000	11.240	11.240	2.248	0.000	0.000	8.992

Continua a pagina seguente...

Figura 5.5: *Caratterizzazioni energetiche ottenute per ridistribuzione dei costi.*

Tabella 5.5 *Caratterizzazioni energetiche ottenute per ridistribuzione dei costi (continua dalla pagina precedente).*

	Istr#	IS cons	Fetch			Decode			Execute			Thumb	Memory
			Ex	Ps	Rs	Ex	Ps	Rs	Ex	Ps	Rs		
continuazione...	29	11.030	8.258	8.386	10.996	8.258	0.000	10.681	15.443	9.940	0.000	0.000	1.994
	30	11.670	8.258	8.386	10.996	8.258	0.000	10.681	15.443	9.940	0.000	0.000	1.994
	31	11.220	8.258	8.386	10.996	8.258	0.000	10.681	15.443	9.940	0.000	0.000	1.994
	32	11.560	11.560	11.560	11.560	11.560	0.000	11.560	11.560	11.560	0.000	0.000	0.000
	33	11.890	8.258	8.386	10.996	8.258	0.000	10.681	15.443	9.940	0.000	0.000	1.994
	34	11.880	8.258	8.386	10.996	8.258	0.000	10.681	15.443	9.940	0.000	0.000	1.994
	35	11.110	8.258	8.386	10.996	8.258	0.000	10.681	15.443	9.940	0.000	0.000	1.994
	36	11.090	8.258	8.386	10.996	8.258	0.000	10.681	15.443	9.940	0.000	0.000	1.994
	37	11.770	8.258	8.386	10.996	8.258	0.000	10.681	15.443	9.940	0.000	0.000	1.994
	38	11.630	8.258	8.386	10.996	8.258	0.000	10.681	15.443	9.940	0.000	0.000	1.994
	39	11.210	8.258	8.386	10.996	8.258	0.000	10.681	15.443	9.940	0.000	0.000	1.994
	40	12.070	2.414	2.414	12.070	2.414	0.000	12.070	12.070	2.414	0.000	0.000	9.656
	41	11.890	4.573	0.915	11.890	4.573	0.000	11.890	4.573	0.915	0.000	0.000	10.975
	42	11.200	8.258	8.386	10.996	8.258	0.000	10.681	15.443	9.940	0.000	0.000	1.994
	43	11.330	8.258	8.386	10.996	8.258	0.000	10.681	15.443	9.940	0.000	0.000	1.994
	44-59	0.000	8.258	8.386	10.996	8.258	0.000	10.681	15.443	9.940	0.000	0.000	1.994
	1-59	—	8.258	8.386	10.996	8.258	0.000	10.681	15.443	9.940	0.000	0.000	1.994

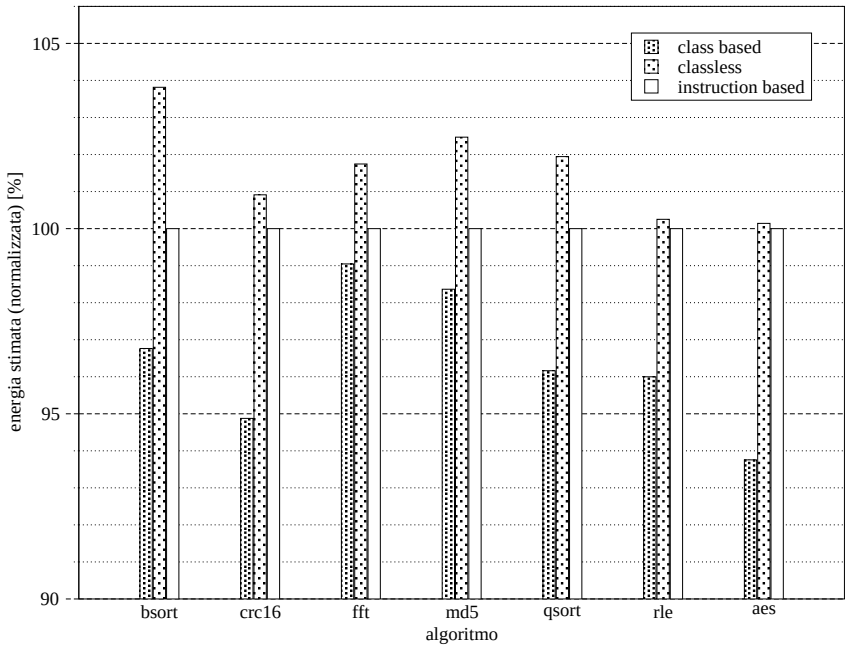


Figura 5.6: Energia stimata normalizzata per metodo di tuning.

- la caratterizzazione class based tende invece a *sottostimare* il consumo energetico

L'errore commesso è in entrambi i casi abbastanza contenuto mantenendosi entro il 5% nel caso di caratterizzazione classless, ed appena superiore per quella class based. Le colonnine indicate con*instruction based* si riferiscono alla previsione ottenuta sommando i contributi di energia per istruzione. Tale somma è stata però ottenuta utilizzando lo stesso TrIBeS e fornendogli una caratterizzazione che associava un contributo costante alle unità funzionali, pari per ciascuna istruzione al costo della stessa, e nullo alle risorse. Così facendo ci si aspettava che, ad ogni ciclo di clock, TrIBeS sommasse semplicemente il costo statico di ciascuna istruzione presente in pipeline. Il grafico mostra che la previsione è stata verificata; questa è un'ulteriore prova della correttezza dell'estensione implementata.

Del resto però, ai fini della SPA, quello che maggiormente conta non è tanto un buon livello di previsione sul consumo complessivo dell'algoritmo quanto piuttosto una sufficiente accuratezza della previsione anche su piccole sequenze di istruzioni. Più interessante è quindi lo studio seguente. Abbiamo tracciato alcuni algoritmi diversi da quelli utilizzati per il tuning del modello, fra cui una implementazione del DES, e studiato l'andamento dell'errore di previsione al variare del numero di istruzioni considerate. Precisamente abbiamo considerato

sequenze di un certo numero di istruzioni consecutive e per ciascuna sequenza confrontato il costo stimato secondo il metodo classico è quello fornito da TrIBeS. Il risultato di tale analisi è riportato in Figura 5.7. I simboli riportano il valore esatti di una rilevazione mentre la linea è una interpolazione polinomiale dei punti campione. Come possiamo vedere sono state studiate le sequenze di $10n$, $20n$ e $50n$ istruzioni, dove n è una potenza di 10. L'andamento riportato in figura è abbastanza buono. La convergenza dell'errore percentuale a valori intorno al $3 - 4\%$ conferma l'analisi precedente ed evidenzia la stabilità del nuovo modello introdotto. Su piccole sequenze di istruzioni l'errore commesso è ovviamente maggiore ma si mantiene entro limiti di accettabilità compresi fra il 10% ed il 20% nel caso di solo 10 istruzioni per rientrare poi, già a partire da sole 30 istruzioni entro la soglia del 5% di scostamento.

Non siamo ancora riusciti a spiegarci chiaramente l'esistenza del minimo nel caso di caratterizzazione class based, l'unica possibile interpretazione per ora è che il modello orientato alle classi sia in grado di descrivere meglio l'andamento dei consumi già per poche istruzioni a differenza di quello classless. In quest'ultimo caso infatti la caratterizzazione è stata determinata considerando il contributo di tutte le istruzioni. Conseguentemente è ragionevole aspettarsi una previsione migliore con l'aumento del numero di istruzioni considerate. Viceversa, il modello class based dovrebbe essere in grado di descrivere meglio il comportamento anche di poche istruzioni per poi convergere verso una accuratezza di descrizione simile a quella classless all'aumento del numero di istruzioni. Questi sono esattamente i comportamenti che traspaiono dal grafico e che ci mostrano come il modello class based sia, coerentemente con quanto ci aspettavamo, generalmente migliore rispetto a quello classless.

5.3 Possibilità di analisi

Per valutare le qualità del tracciato di energia stimata prodotto da TrIBeS agli scopi della SPA, confrontando ulteriormente le differenze fra le due tipologie di analisi, abbiamo considerato una serie di esempi didattici di analisi del profilo di potenza assorbita.

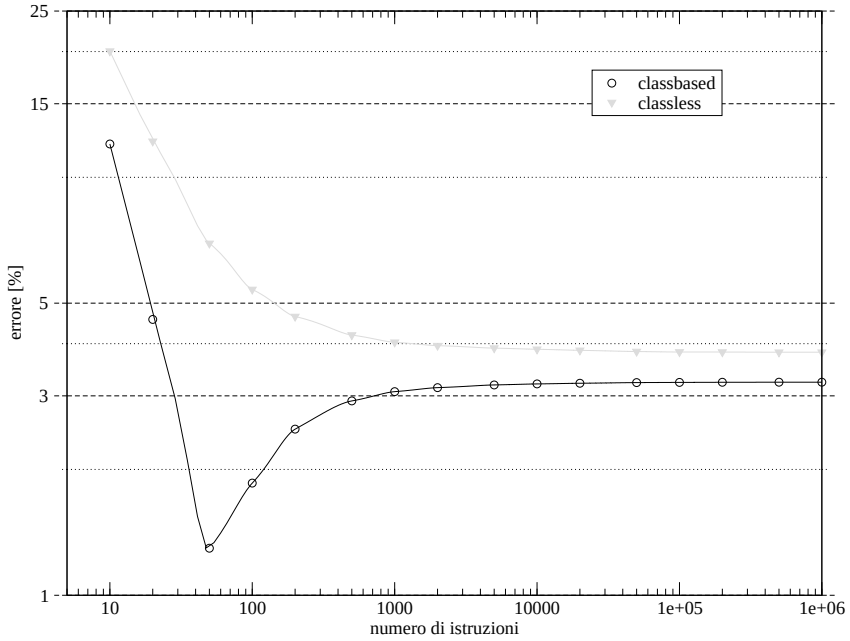


Figura 5.7: Andamento percentuale dell'errore di stima in funzione del numero di istruzioni consecutive considerate.

5.3.1 Differenze nel flusso di controllo

Cominciamo con un semplice esempio di analisi del flusso di controllo. Il codice considerato è mostrato in Figura 5.8 e consiste sostanzialmente in un costrutto **`if-then-else`**, all'interno di un ciclo eseguito due volte, la prima volta viene preso il ramo **`else`** calcolando una somma mentre la seconda volta si esegue la moltiplicazione. Le due istruzioni eseguite sono diverse, anche come classe energetica, e di conseguenza ci aspettiamo un tracciato di energia assorbita con profilo diverso nelle due iterazioni del ciclo.

Il profilo dell'energia assorbita, in funzione della caratterizzazione utilizzata, viene mostrato nelle Figura 5.9a, Figura 5.9b e Figura 5.9c, rispettivamente nel caso di caratterizzazione statica, classless e class based. I marker[11] evidenziano a coppie degli intervalli di cicli di clock corrispondenti ciascuno ad una delle due iterazione del ciclo. ***for***[12].

Com'è possibile vedere nelle figure le due esecuzioni del ciclo producono un diverso tracciato di energia, proprio come ci aspettavamo. Possiamo ulterior-

[11] Di seguito diremo marker le linee tratteggiate verticali utilizzate al fine di contrassegnare particolari cicli di clock o definire intervalli di questi.

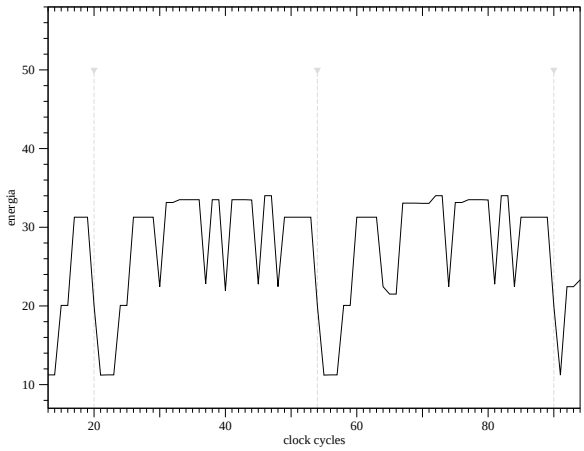
[12] Più precisamente stanno ad indicare il ciclo di clock in corrispondenza del quale si ha l'ingresso in pipeline dell'istruzione assembler che testa la condizione dell'***if***

```
        for(i=0; i<20; i++){
          if (i) {
② ☞             i*=2;
          } else {
① ☞             i+=10;
          }
        }
```

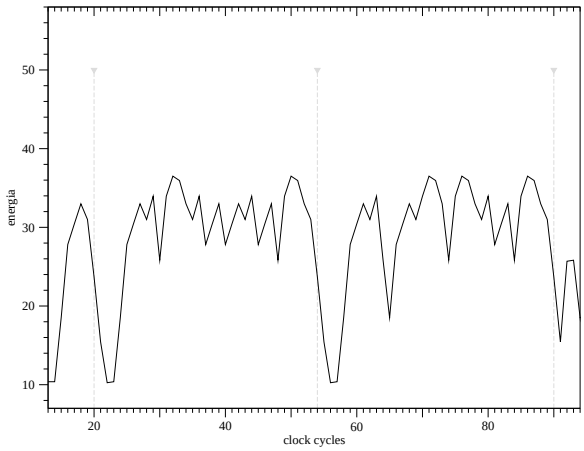
Figura 5.8: *Codice d'esempio per lo studio della traccia di un flusso di controllo.*

mente osservare che il tracciato risulta essere sempre più frastagliato, mostrando componenti a più alta frequenza, quando passiamo dalla caratterizzazione statica a quella classless fino a quella class based. Il sospetto che ci viene è quindi quello che ci siano maggiori differenze nell'ultimo caso rispetto ai precedenti. Al fine di confermare questa intuizione abbiamo fatto uno studio dei tracciati inteso a mettere in evidenza le differenze che sussistono fra un ciclo di clock della prima iterazione con quello corrispondente nella seconda, per un totale di 36 cicli di clock. I risultati sono mostrati in Figura 5.10a e Figura 5.10b. Nei grafici viene riportata, per ciascun ciclo di clock, una colonna di altezza proporzionale alla differenza di energia fra il ciclo di clock nella prima iterazione ed il corrispondente ciclo della seconda. Le colonne "vuote" si riferiscono all'analisi della traccia ottenuta da caratterizzazione statica e sono state sovrapposte alle rispettive colonne "piene" della caratterizzazione classless e class based rispettivamente nella parte (a) ed in quella (b) della figura.

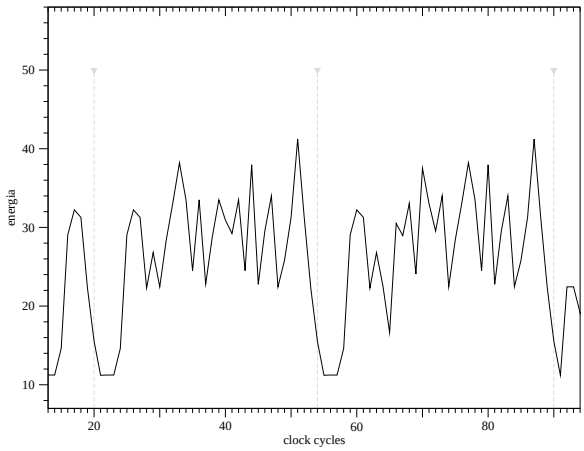
Come possiamo osservare l'intuizione è esatta e la differenza media fra le energie dei due cicli, indicata nelle legenda, è molto maggiore nel caso in cui abbiamo utilizzato la caratterizzazione class based (circa 15%) che non in quello corrispondente alle caratterizzazioni classless (circa 10%) e statica (circa 9%). Per concludere, è corretto far notare che, operando per ispezione la differenza dei tracciati è più facilmente identificabile in Figura 5.9a che non nelle altre. Infatti in questo caso una traccia meno perturbata rende subito evidenti all'occhio umano le macro-differenze esistenti. Tuttavia è pure vero che, pensando ad una ipotetica elaborazione ed analisi automatizzata delle tracce, è comunque vantaggioso poter disporre di una più dettagliata traccia di previsione dell'assorbimento energetico.



(a) caratterizzazione statica (instruction based)

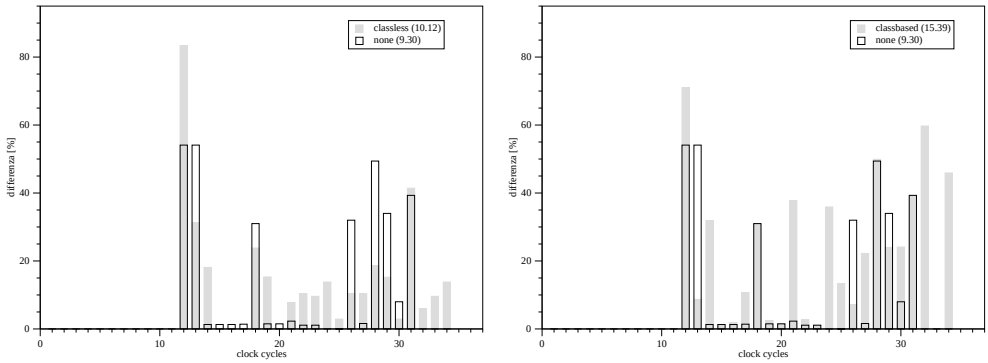


(b) caratterizzazione classless



(c) caratterizzazione class based

Figura 5.9: Studio del flusso di controllo: esempi di esecuzione di istruzioni diverse, confronto fra le caratterizzazioni.



(a) caratterizzazione statica vs classless (b) caratterizzazione statica vs class based

Figura 5.10: Differenze fra blocchi di codice: confronto in percentuale considerando modelli diversi.

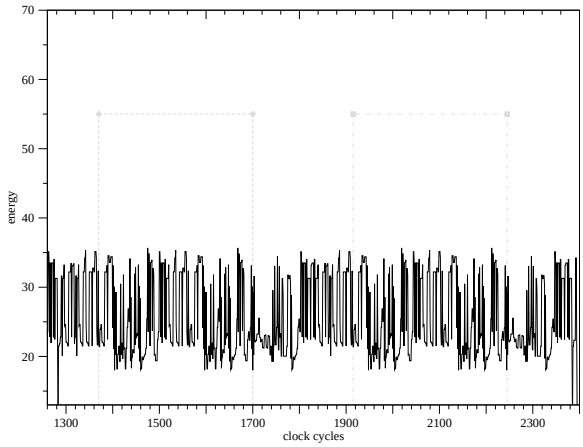
5.3.2 Equivalenze nei cicli

In modo duale a quanto fatto nel paragrafo precedente abbiamo poi valutato la possibilità di individuare dei pattern periodici all'interno di una traccia d'assorbimento. Ci aspettiamo un tale comportamento quando vengono eseguite le istruzioni del corpo di un ciclo.

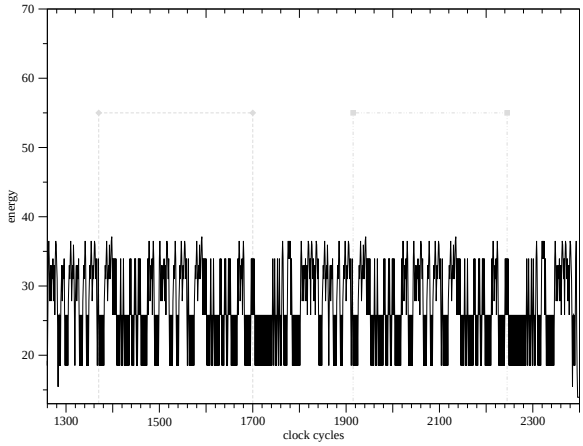
Il semplice codice utilizzato per l'esperimento è mostrato in Figura 5.12, si tratta della moltiplicazioni di una matrice $[2x2]$ per un vettore colonna $[1x2]$.

I risultati del tracciamento con i diversi modelli di caratterizzazione vengono messi a confronto nella serie di grafici in Figura 5.11. I marker riportati mettono in evidenza i cicli di clock nei quali viene eseguito il corpo del ciclo. A differenza del caso precedente ora l'analisi si concentra su un intervallo di cicli molto più esteso, ancora una volta possiamo osservare come l'informazione presente nella traccia generata con classificazione class based sia più dettagliata delle altre due. Effettuando uno studio sulle differenze, ciclo di clock per ciclo di clock, si nota che la traccia si ripete esattamente uguale in tutti i casi[13].

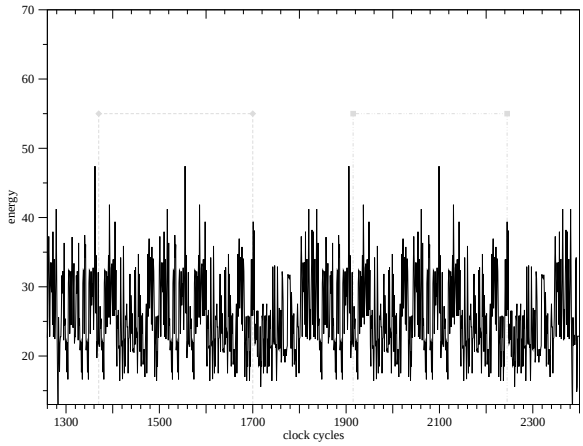
[13] Il grafico non è stato riportato dato che avrebbe sempre valori nulli per ogni ciclo di clock.



(a) caratterizzazione statica (instruction based)



(b) caratterizzazione classless



(c) caratterizzazione class based

Figura 5.11: Studio di un ciclo: esempi di doppia iterazione sul corpo di un ciclo.

```
#define SIZE 2;
 ...
//multipling matrix
for (j=0; j<SIZE; j++) {
  Y[j] = 0;
  for (l=0; l<SIZE; l++)
    Y[j] += A[j][l]*X[l];
}
```

Figura 5.12: *Codice d'esempio per lo studio della traccia di un ciclo.*

5.4 Esempi reali di SPA

Concludiamo questo lavoro di Tesi con alcuni esempi pratici di *Simple Power Analysis* al fine di mostrare l'adeguatezza dello strumento realizzato ad essere utilizzato a supporto di questo tipo di analisi.

5.4.1 Analisi di una implementazione del DES

Per questa prova abbiamo impiegato una implementazione in C del DES, compilata in microcodice e tracciata con TrIBeS utilizzando la caratterizzazione class based.

Il tracciato dell'energia assorbita stimata, mostrato in Figura 5.13, è costituito da un elevato numero di campioni, risulta per questo piuttosto fitto. Tuttavia è comunque adatto ad osservare alcune macro-caratteristiche dell'implementazione. La traccia mette in evidenza infatti diverse fasi dell'algoritmo come lo *scheduling delle chiavi*, l'*Initial Permutation* (IP) e la sequenza successiva dei DES *rounds*. Vediamo di identificare queste porzioni nella traccia facendo riferimento al codice specifico.

Il programma eseguito è abbastanza semplice: prende in ingresso una chiave di codifica ed un plaintext. Anzitutto vengono determinate, a partire dalla chiave di codifica, tutte le sotto-chiavi che serviranno nei diversi round dell'algoritmo. Questa operazione è eseguita nella funzione:

```
des_set_key( &ctx, encKey );
```

durante la quale in un ciclo ripetuto 16 volte sono calcolate le chiavi per ciascuno dei sedici round della fase di cifratura. I cicli sono ben evidenti nel tracciato di potenza in corrispondenza della regione evidenziata dalla prima coppia di marker.

Nella stessa funzione indicata precedentemente, l'implementazione del `des` considerata, calcola pure le sotto-chiavi di decodifica. Ciò avviene in un bana-

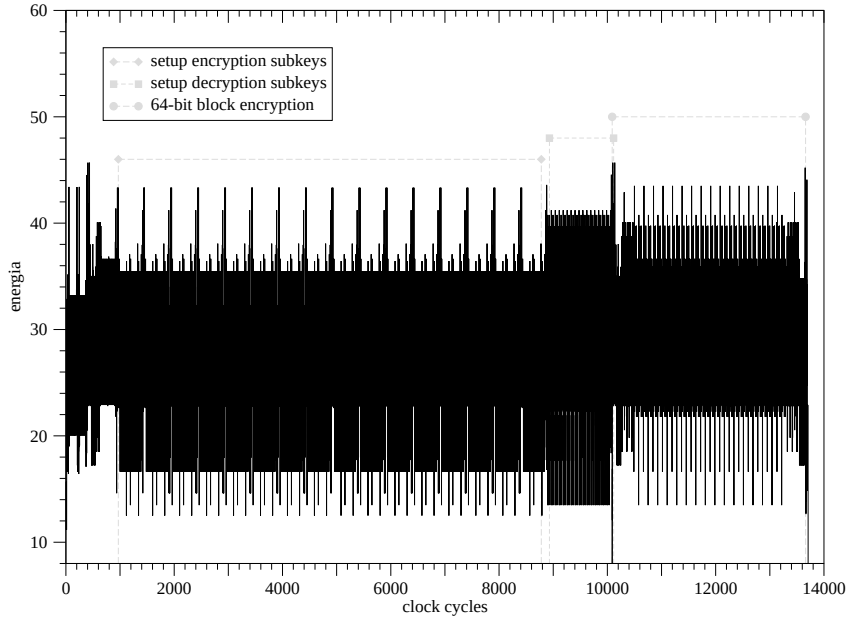


Figura 5.13: Tracciato dell'eneriga assorbita, stimata con caratterizzazione class based, per una implementazione di riferimento del DES.

le ciclo ripetuto 32 volte ed evidenziato nel tracciato dalla seconda coppia di marker.

Infine si ha l'esecuzione della cifratura vera e propria che avviene nell'intervallo di cicli di clock definito dalla terza coppia di marker. In Figura 5.14 abbiamo una visione più dettagliata della sola fase di cifratura. Ben distinguibili sono la permutazione iniziale e finale contrassegnate dalla prima e terza coppia di marker interni. I sedici rounds del processo di cifratura sono altrettanto ben visibili e contrassegnati da marker.

Possiamo fare a questo punto un confronto fra il nostro risultato, ottenuto per simulazione, e quello classico ottenuto da Kocher in [14] per rilevazione diretta sul processore, già mostrato in Figura 1.6 a pagina 16. Dal confronto possiamo vedere che anche l'andamento dell'energia prevista per simulazione è molto simile a quello che si potrebbe ottenere in un caso reale. Non solo abbiamo regioni diverse ad assorbimento diverso ma, come evidenziava Kocher nel suo articolo, anche nel nostro caso i diversi rounds dell'algoritmo sono ben distinguibili nella traccia.

Per finire in Figura 5.15 viene mostrato in dettaglio il tracciato di due rounds consecutivi del processo di cifratura. Ancora una volta possiamo osservare come il tracciato si ripeta identico in ciascuna iterazione. Questo è un limite dell'im-

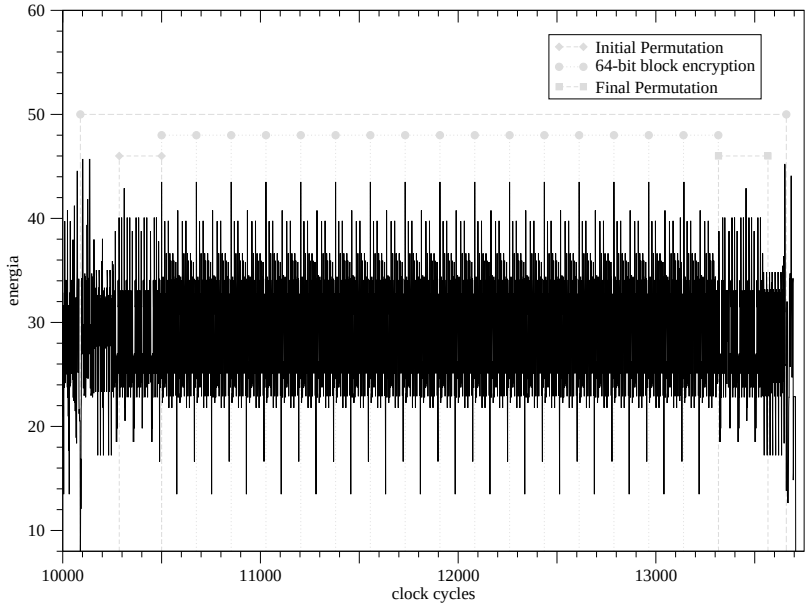


Figura 5.14: Dettaglio dell'energia assorbita, dall'implementazione del DES considerata, nella sola fase di cifratura. Sono ben evidenti i sedici rounds dell'algoritmo.

plementazione attuale di TrIBeS che, essendo un simulatore comportamentale, non considera gli aspetti funzionali del codice simulato. La conseguenza di ciò è che non riusciamo ad avere una certa forma di dipendenza dai dati dell'energia assorbita prevista. Nella realtà, come mostrato nella traccia cui ci riferivamo precedentemente, l'esecuzione del corpo di un ciclo più volte non genera mai la stessa traccia e questo perché l'energia assorbita dipende anche dal particolare valore dei dati che si stanno elaborando. Con il nostro simulatore non riusciamo a tracciare queste variazioni e di conseguenza il simulatore non è ancora adatto ad essere impiegato nell'ambito della più potente DPA. Tuttavia i risultati che si possono ottenere anlizzando una traccia con la SPA non sono insignificanti. Riuscire a metter in evidenza le macro-caratteristiche di una implementazione consente ad esempio di isolare delle regioni critiche in cui applicare ulteriori studi.

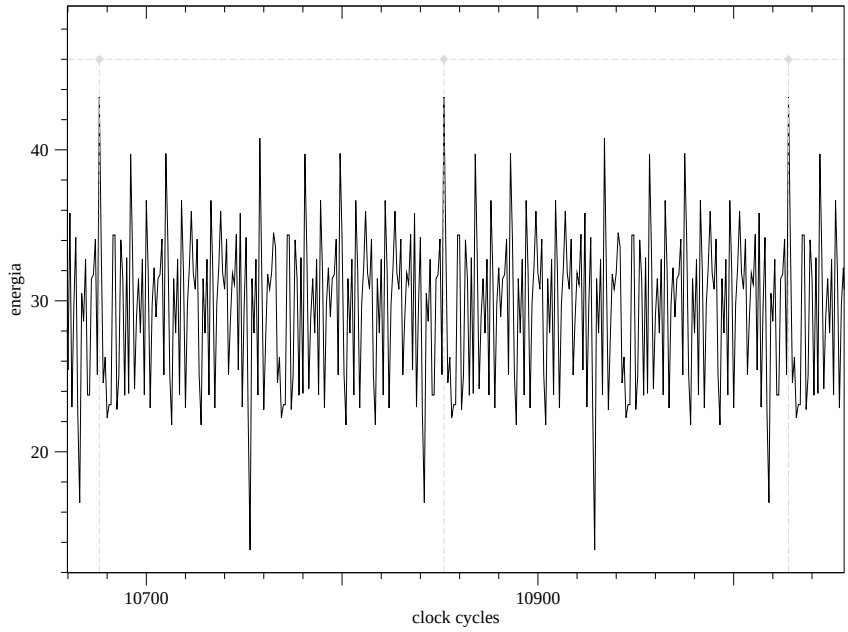


Figura 5.15: Dettaglio dell'energia assorbita, dall'implementazione del DES considerata, in due rounds consecutivi.

5.4.2 Studio di una implementazione debole

Presentiamo ora un esempio pratico di come si possa impiegare la SPA al fine di verificare la robustezza di certe implementazioni di algoritmi di cifratura. Il caso che consideriamo è quello del calcolo dell'esponenziale modulo n di un intero, questa funzione è usata ad esempio in alcune fasi di algoritmi di cifratura noti ed attualmente utilizzati come DSA ed ElGamal.

Una semplice funzione che consente di ottenere l'esponenziale modulo n di un intero y è la *"square-and-multiply"* la cui implementazione abbiamo già riportato in Figura 1.9 a pagina 19. Abbiamo implementato lo pseudocodice nel semplice programma C mostrato in Figura 5.16. Come possiamo vedere la chiave di cifratura ad 8 bit, definita direttamente all'interno del codice nella variabile `e`, ha valore:

```
0xb4 == 10110100
```

La linea di codice indicata da ① calcola il quadrato modulo n ad ogni iterazione del ciclo sui bit della chiave, mentre la linea contrassegnata con ② esegue la moltiplicazione.

La debolezza dell'implementazione è dovuta proprio alla dipendenza dell'esecuzione del codice in ② dal valore dei bit della chiave. In questo caso abbiamo

quindi un chiaro esempio di dipendenza del flusso di controllo dai dati, l'ipotesi per l'applicabilità della SPA è verificata ed una tale implementazione è vulnerabile ad attacchi di tipo SPA. A dimostrazione di ciò abbiamo compilato in microcodice l'esempio e l'abbiamo poi tracciato con TrIBeS usando la solita caratterizzazione statica. Il tracciato dell'energia assorbita prevista che abbiamo ottenuto è mostrato in Figura 5.17.

Nella figura sono state posizionate due tipologie di marker diversi in corrispondenza di tutti i controlli che precedono le istruzioni di *square* e di *multiply* al fine di porre in evidenza le differenze della traccia quando viene eseguita una istruzione piuttosto che l'altra. Possiamo osservare che il numero di cicli di clock che separa una istruzione di *multiply* da quella successiva di *square* non è sempre lo stesso. Precisamente quando il bit della chiave che si sta analizzando in un determinato ciclo è nullo l'operazione di *multiply* non viene eseguita e di conseguenza i cicli di clock che intercorrono per giungere di nuovo al test sulla variabile di controllo del ciclo **`while`** sono in numero inferiore.

In sostanza, osservando la traccia possiamo ricostruire esattamente il valore della chiave di cifratura, infatti:

- ogni volta che ci sono "pochi" clock cycles fra i un marker square ed il successivo multiply il bit corrispondente della chiave vale 0
- ogni volta che ci sono "tanti" clock cycles fra i un marker square ed il successivo multiply il bit corrsipondente della chiave vale 1
- il primo bit[14] vale sempre 1
- davanti al MSB bit ad 1 ci sono tanti bit a zero quanti necessari a completare la lunghezza della chiave[15]

Seguendo queste semplici regole è facile rileggere dalla traccia il valore in binario della chiave di cifratura usata.

[14] Ovvero il bit più significativo (MSB).
[15] Nel nostro caso dovremo avere sempre 8 bit in totale.

```
#include <stdlib.h>

#ifndef uint8
#define uint8 unsigned char
#endif

uint8 e=0xb4; /*Binary: 10110100*/
uint8 n=0x85;

uint8 plain=0x6; /*Playntext: 110*/
uint8 chiper;

int main () {
  char keybits;

  /* finding MSB set to 1 */
  keybits=8;
  while ( !(e & 0x80) && keybits>0 ) {
    e<<=1; keybits--;
  }

  if ( keybits>0 ) {
    chiper=plain;
    e<<=1; keybits--;

    while ( keybits>0 ) {
      /* square */
①☞    chiper=(chiper*chiper)%n;
      if ( e & 0x80 ) {
        /* multiplay */
②☞      chiper=(chiper*plain)%n;
      }
      e<<=1; keybits--;
    }
  }
  return 0;
}
```

Figura 5.16: *Implementazione in C della funzione di* square-and-multiply.

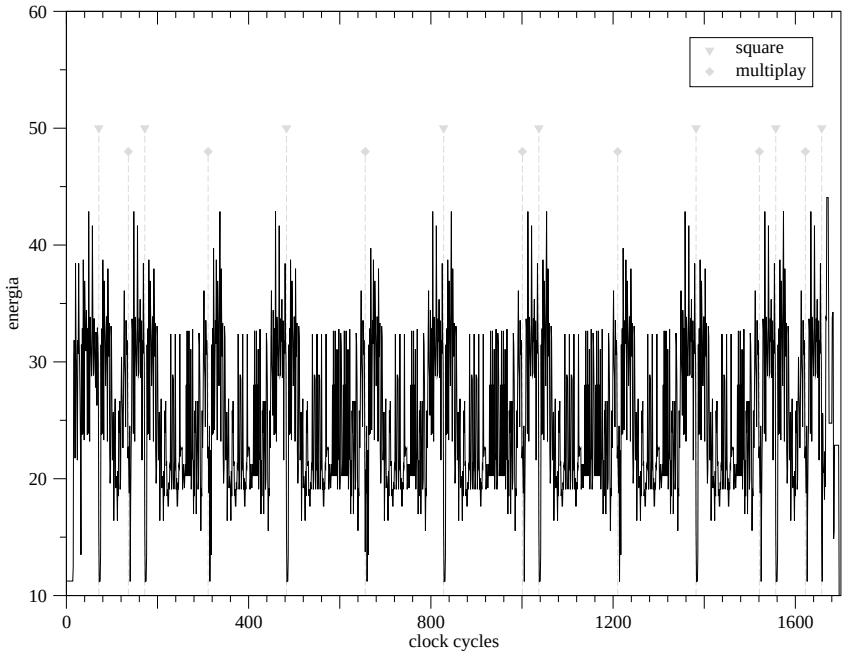


Figura 5.17: Esempio di attacco SPA in cui si mostra la vulnerabilità di una algoritmo di *square-and-multiply*.

Conclusioni e sviluppi futuri

"Le possibilità per un futuro grandioso diverranno certezze solamente quando noi stessi ci assumeremo le responsabilita per il nostro futuro"

Gifford Pinchot

OBBIETTIVO principale di questo lavoro di Tesi è stato quello di affrontare le problematiche legate all'analisi di potenza da una diversa prospettiva. L'impiego di tecniche di simulazione a livello si *Instruction Set Architecture*, al fine di ottenere i dati necessari all'analisi, in sostituzione alle più costose e complesse tecniche previste dal metodo classico, con la rilevazione diretta dal core dei valori di energia, comporta diversi vantaggi. La possibilità di ricorrere a questo tipo di analisi già in fase di progettazione rappresenta indubbiamente il principale punto a favore dell'approccio proposto. In questo modo diventa possibile, ad esempio, studiare la robustezza di un sistema prima ancora di realizzarlo ed eventualmente modificare alcune scelte progettuali.

Analisi dei risultati

Nell'applicazione della metodologia la fase più onerosa è quella di tuning di una nuova architettura, finalizzata alla caratterizzazione energetica dei componenti della pipeline. In questa fase il passaggio cruciale consiste nell'individuazione di una traccia di tuning significativa e rappresentativa dell'intero instruction set.

Una volta ottenuta al caratterizzazione per una data architettura il sistema è immediatamente utilizzabile per il tracciamento e l'analisi di sicurezza un qualunque altro programma.

I risultati ottenuti sono in linea con le aspettative. Siamo riusciti a dimostrare che un approccio software-oriented all'analisi di potenza non solo è fattibile ma anche relativamente facile da implementare rispetto al metodo classico. I tracciati di potenza generati mostrano un sufficiente livello di dettaglio ai fini della *Simple Power Analysis* e possono essere ottenuti con uno sforzo relativamente contenuto.

Sviluppi futuri

La mancanza di una adeguata strumentazione non ci ha consentito di effettuare una validazione dei risultati ottenuti rispetto al processore reale. Ciò che è stato possibile fare è stato quindi solo una validazione rispetto ai dati d'ingresso finalizzata a verificare che la stima energetica complessiva, prodotta dalla nuova versione del simulatore, non fosse molto discordante rispetto a quella indicata dalla caratterizzazione statica dell'instruction set. Questo ci ha consentito di affermare che il simulatore è sufficientemente preciso nella stima dei consumi ma non di verificarne la precisione assoluta sul singolo ciclo di clock.

La precisione del singolo ciclo di clock non è tanto importante ai fini dell'analisi di potenza, dove invece è molto più interessante una certa fedeltà al caso reale dell'andamento degli assorbimenti. Tuttavia un studio della precisione del modello in termini assoluti rispetto al caso reale potrebbe essere comunque interessante ed aprire le porte ulteriori possibili applicazioni. La possibilità di tracciare, abbastanza velocemente e semplicemente, i consumi di un generico programma, con una sufficiente accuratezza, potrebbe infatti trovare applicazione ad esempio nel campo dell'ottimizzazione dei consumi.

Cosa ancora più interessante è l'ulteriore estensione del modello al fine di includere la dipendenza dai dati del tracciato prodotto. Essendo TrIBeS un simulatore comportamentale era, ed è tuttora, estraneo agli aspetti funzionali. Tuttavia potrebbe esser non eccessivamente onerosa l'introduzione di una seppur semplice forma di tracciamento dell'energia legata al valore degli operandi. Ad esempio, semplicemente associando un costo all'attività di switching, potrebbe essere abbastanza immediata l'introduzione di pochi parametri al modello che consentano di tener conto nella traccia di energia assorbita dei valori che vengono manipolati nei registri. Benché semplice, una tale forma di dipendenza dai

dati potrebbe tuttavia essere sufficiente a manifestare nella traccia prodotta quelle minime variazioni su cui applicare tecniche di analisi di potenza più evolute come la *Differential Power Analysis*.

APPENDICE A

Supporto Atomic e TrIBeS per l'*ARM7TDMI*

"Sappiamo come fare, quello che ci serve ora è qualcosa con cui giocare"
Hague

PRESENTIAMO in questa appendice, a titolo di esempio, il lavoro svolto per implementare il supporto in TrIBeS ed Atomic al processore *ARM7TDMI*. Su questa stessa architettura è stata effettuata la validazione del modello per il tracciamento degli assorbimenti per clock cycle proposto in questa Tesi.

Anzitutto viene descritta l'architettura del processore. Senza scendere troppo in dettaglio sono tuttavia messi in evidenza tutti gli aspetti particolari e le cose di cui si dovrà tener conto al fine di implementare una corretta siumulazione dell'architettura.

A seguire, due paragrafi: il primo si occupa dell'implementazione della libreria TrIBeS per la simulazione comportamentale del processore, ed il secondo, risalendo all'indietro il processo di elaborazione, tratta della libreria Atomic e dettaglia come si realizza il compilatore in microcodice.

Il paragrafo conlusivo spiega i tools sviluppati per la generazione della traccia d'esecuzione nella fase di preprocessing. La soluzione adottata a questo livello si differenzia infatti da quella tradizionalmente usata e quindi vengono discusse le ragioni di un diverso approccio ed il dettaglio della sua implementazione.

A.1 Architettura del processore

La famiglia ARM è composta da processore general-purpose a 32 bit, disponibili in diverse versioni architetturali:

ARMv4T implementata nell'arm7, processore con a 3 stadi

ARMv5T con l'arm9 e l'arm10, processori con pipeline a 5 e 6 stadi rispettivamente

ARMv6 con l'arm11, l'ultimo nato della famiglia, è un processore con 8 stadi di pipeline

Nella nostra discussione ci concentreremo sull'architettura ARMv4T, progettata esplicitamente per consentire implementazioni molto semplici ma al contempo altamente performanti. La semplicità dell'architettura si traduce anche in bassi consumi di potenza rendendola particolarmente adatta per l'adozione all'interno di sistemi embedded e portatili in generale.

I processori di questa presentano le caratteristiche tipicamente di una architettura , come un elevato numero di registri organizzati in *register files*, accesso alle memoria di tipo *load/store*, pochi e semplici modi di indirizzamento ed ancora istruzioni macchina di lunghezza fissa. Alcune caratteristiche aggiuntive sono:

- ❏ controllo sulla ALU[1] e sullo shifter in ogni operazione di elaborazione dati
- ❏ istruzioni per caricare e salvare più registri contemporaneamente
- ❏ esecuzione condizionata di tutte le istruzioni

Il processore specifico di cui ci occuperemo è l'$ARM7TDMI$, una completa implementazione dell'architettura ARMv4T. Quella a seguire non è tuttavia una completa descrizione del processore, che potete invece trovare in [33] e [34], quanto piuttosto un riassunto di ciò che è necessario conoscere ai nostri fini.

[1] *Unità aritmetico logica*

A.1.1 Specifiche architetturali

Il core del *ARM7TDMI* è una tipica implementazione dell'architettura di Von Neumann con singolo bus a 32 bit sia per i dati che per le istruzioni. Le uniche operazioni che possono accedere direttamente alle memoria sono quelle di **`load`** e **`store`**. I tipi di dato supportati sono: Byte (8 bit), Word (32 bit) e l'Halfword (da 16 bit).

Un interessante aspetto dell'*ARM7TDMI* è quello di essere *bi-endianness*, risultando quindi compatibile con entrambe le forme di usate dalle memorie. La configurazione di default del processore usa il modo *little-endian* per la rappresentazione dei dati. Non esistono istruzioni macchina per cambiare il modo di *endianess*, viceversa è presente un input hardware che consente di configurarlo in funzione del sistema di memoria utilizzato.

A.1.2 Istruzioni macchina

Un'altra caratteristica distintiva delle architetture ARM è la disponibilità di un doppio instruction set:

- *ARM*: un instruction set a 32 bit
- *THUMB*: un instruction set a 16 bit

Questa particolare caratteristica consente di combinare i benefici delle architetture a 16 bit, come un'alta densità del codice[2], con quelli tipici delle architetture a 32 bit, caratterizzate tipicamente da migliori prestazioni nella manipolazione di dati a 32 bit e dalla possibilità di gestire facilmente più ampi spazi di indirizzamento.

L'instruction set THUMB è sostanzialmente un sottoinsieme di quello ARM a 32 bit, ogni istruzione a 16 bit viene mappata sulla corrispondente istruzione a 32 bit ed hanno lo stesso effetto all'interno del processore. Infatti, durante l'esecuzione in THUMB mode, le istruzioni a 16 bit vengono trasparentemente "decompresse" nelle equivalente a 32 bit durante la fase di *Decode*. In aggiunta la istruzioni a 16 bit hanno completo accesso a tutti i register files del processore, questo, aggiunto alla possibilità di cambiare dallo stato ARM a quello THUMB durante l'esecuzione stessa[3], conferisce all'architettura un elevato grado di flessibilità e di possibilità di cooperazione fra i due insiemi di istruzioni.

[2]Ovvero un maggior numero di istruzioni macchina a parità di dimensione del codice. Questa caratteristica è particolarmente interessante nel caso dei sistemi embedded.

[3]Usando l'istruzione **BX**.

A.1.3 Modalità operative

I modi operativi previsti sono sette:

- *User* (usr): La normale modalità di esecuzione
  FIQ (fiq): a supporto del trasferimento dati
- *IRQ* (irq): usato in generale per il servizio agli interrupt
- *Supervisor* (scv): modalità protetta riservata al sistema operativo
- *Abort mode* (sbt): ci si entra dopo che il prefetch di una istruzione o di un dato non è andato a buon fine
- *System* (sys): modo privilegiato per il sistema operativo
- *Undefined* (und): ci si entra quando viene chiesta l'esecuzione di una istruzione sconsciuta

Eccezion fatta per il modo usr, gli altri sei modo sono privilegiati e disponibili solo per la programmazione di sistema. Queste modalità protette hanno accesso a dei registri loro riservati ed hanno completo controllo su tutte le risorse del sistema.

A.1.4 Esecuzione condizionale

Una peculiarità dei processori ARM è la possibilità di condizionare l'esecuzione di qualunque istruzione. Ogni istruzione contiene infatti un campo condizione, se alcuni flags del registro CPSR non rispettano la condizione esplicitata dall'istruzione, l'istruzione stessa viene convertita in una NOP e quindi il suo avanzamento nella pipeline non produce alcun effetto. L'esecuzione condizionale viene usata ad esempio per fare oppure no un salto condizionato.

A.1.5 Struttura della pipeline

L'$ARM7TDMI$ ha una pipeline piuttosto semplice a tre soli stadi, la Figura A.1 ne è una rappresentazione.

Lo stadio di fetch semplicemente recupera una istruzione da memoria e la carica all'interno dell'*Instruction Register*. Nello stadio successivo di Decode l'istruzione viene decodificata assieme ai registri che utilizza, se l'istruzione è di tipo ARM viene prima decompressa. Infine, nell'ultimo stadio, vengono eseguite le seguenti operazioni:

1. gli operandi sono letti dai registri
2. le operazioni relative all'ALU o all'unità di shifting eventualmente richieste

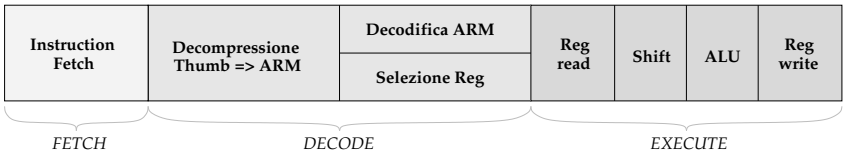


Figura A.1: Pipeline a tre stadi dell'$ARM7TDMI$

3. i risultati vengono scritti negli opportuni registri

Possiamo subito osservare che gli stadi non sono ben bilanciati, come invece vorrebbe il paradigma a pipeline. La maggior parte della computazione avviene infatti nello stadio di esecuzione.

Data Hazard Grazie alla semplicità della pipeline non si possono verificare dei *data hazard* in quanto tutte le operazioni di lettura e scrittura dei registri avvengono solo in fase di esecuzione. Conseguentemente non è richiesta l'implementazione di alcun meccanismo di forwarding o di controllo dinamico (waw, war, raw) rendendo molto semplice la progettazione del datapath e dei compilatori.

Control Hazard I potenziali hazard sul flusso di controllo vengono risolti, seppure in modo non propriamente performante, usando l'esecuzione condizionata; una unità di *branch prediction* infatti non è presente al fine di mantenere basso il costo del processore.

Passano due cicli di clock da che una istruzione di salto condizionale viene inserita in pipeline a quando la sua destinazione e decisa, nella unità d'esecuzione. Le due istruzioni assembly successive a quella di salto vengono comunque immesse nella pipeline al fine di mantenerla piena. Quando il salto condizionato viene valutato, se dev'essere preso diviene una normale istruzione di salto non condizionato e le due istruzioni successive annullata dalla esecuzione condizionale, altrimenti viene ignorato ed il tutto si risolve come se si fosse trattato di un NOP. In conclusione quindi, nel caso di *branch taken* si "consumano" 2 cicli di clock in aggiunta ai tre necessari affinché venga risolto l'indirizzo del salto, mentre nel caso di *branch not taken* è come se la pipeline venisse attraversata da una istruzione nulla.

Structural Hazard Le istruzioni ed i dati condividono il bus, tuttavia la particolare struttura della pipeline limita la possibilità che si possano verificare degli hazard strutturali. In una architettura load/store sono infatti solo queste le istruzioni che possono accedere direttamente alla memoria.

Una load richiede sempre tre cicli di clock, nel primo viene calcolato l'indirizzo sorgente ed al contempo si carica una nuova istruzione in pipeline, nel secondo si accede alle memoria per il recupero delle informazioni ed infine, nel terzo ciclo di clock[4], il valore recuperato viene effettivamente salvato nel registro di destinazione e l'istruzione esce dalla pipeline. Una istruzione di store funziona in modo analogo ad eccezzione dell'ultimo ciclo di clock che non è più presente.

In definitiva una istruzione di load richiede sempre almeno $2 + 3$ cicli mentre una di store esattamente $2 + 2$ cicli.

Alla luce delle considerazioni fin qui fatte possiamo concludere che l'architettura dell'$ARM7TDMI$ non si presta ad ottenere un elevato *grado di parallelismo*, inoltre, a seguito della lunga permanenza delle istruzioni nell'ultimo stadio, la pipeline non si presta ad operare con cicli di clock troppo brevi.

Prestazioni migliori vengono raggiunte con l'ARM9, caratterizzato da un core con pipeline a 5 stadi in cui l'unico stadio di esecuzione dell'$ARM7TDMI$ viene scomposto in 3 parti. Suddividendo l'esecuzione di una istruzione in cinque cicli di clock la massima frequenza di clock raggiungibile è facilmente raddoppiabile. Per contro l'ARM9 ha un design più complesso del core e quindi è anche caratterizzato da maggiorni consumi.

A.1.6 I registri

Nel complesso sono disponibile 37 registri, dei quali: 31 sono *registrid'uso generale* a 32 bit ed i rimanenti sei sono *registri di stato*. Quando l'esecuzione avviene nello sato ARM è possibile accedere a 16 registri d'uso generale e due resitri di sato visibili in ogniuna delle modalità operative. Nei modi privilegiati sono disponibile altri registri in *modalità banked* ovvero rimappati su alcuni registri "normali" a seconda della modalità operativa.

I rgistri da *r0* a *r13* sono d'uso generale, ma convenzionalmente *r13* viene usato come *Stack Pointer* (SP). I registri *r14* e *r15* hanno invece una funzione particolare:

r14 viene detto *Link Register* (LR) ed assolve a due funzioni particolari:

[4]Nel caso in cui non ci siano maggiori latenza della memoria, ad esempio in seguito ad un cache miss.

1. contiene l'indirizzo di ritorno da una routine[5]. L'invocazione di un sottoprogramma avviene mediante l'invocazione dell'istruzione *Branch and Link* che provvede a salvare l'indirizzo di ritorno nel registro BL. Per ritornare da una routine basta quindi ricopiare il registro LR nel PC.
2. quando viene sollevata un'eccezzione contiene l'indirizzo di ritorno dal gestore dell'eccezzione

r15 rappresenta il *Program Counter* (PC), può essere letto e scritto dalle istruzioni ma su di esso sono definite alcune restrizioni d'accesso

I registri di stato sono due: *Currente Program Status Register* (CPSR) e *Saved Program Status Register* (SPSR). Il primo contiene i bit di stato per il programma corrente mente il secondo, accessibile solo nei modi privilegiati, contiene le informazioni sull'eccezzione che ha causato l'entrata nello stato privilegiato.

Nello stato THUMB è accessibile solo un sottoinsieme dei registri dello stato ARM, precisamente i registri da *r0* a *r7* più i registri speciali visti precedentemente.

A.2 Libreria TrIBeS per l'$ARM7TDMI$

Nella sezione precedente abbiamo dato uno sguardo generale all'architettura dell'$ARM7TDMI$, ora faremo vedere come sia possibile costruire un simulatore comportamentale per questa architettura, usando l'SDK fornito con TrIBeS. L'esempio che riportiamo è abbastanza interessante poiché, essendo riferito ad una architettura reale piuttosto semplice, consente di vedere con relativa semplicità tutti i concetti alla base della strutturazione di un simulatore per TrIBeS e del suo funzionamento, senza tuttavia precludere informazioni che potrebbero essere necessarie ad implementare il supporto per architetture più complesse.

Come sappiamo dal Cap. 2, TrIBeS è un framework costituito da diverse classi molte delle quali astratte. Le classi rendono disponibile un ambiente per la costruzione di simulatori comportamentali; quello che lo sviluppatore deve fare è estendere alcune delle classi base, ridefinendo i loro metodi, al fine di descrivere il funzionamento delle unità funzionali e delle risorse dell'architettura simulata.

[5]Procedura o funzione

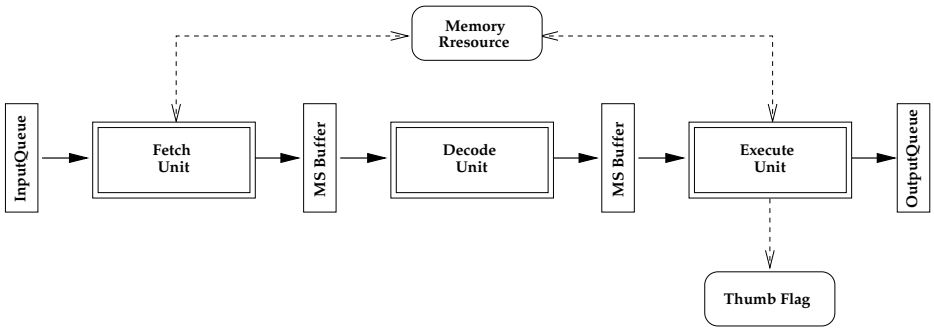


Figura A.2: Architettura della pipeline simulata per l'$ARM7TDMI$

A.2.1 Struttura generale

La struttura del simulatore che sviluppato riflette esattamente quella dell'architettura fisica della pipeline di un $ARM7TDMI$ mostrata in Figura A.1. Ogni stadio della pipeline, come invece mostra la Figura A.2, è stato mappato in una corrispondente *unità funzionale* del simulatore. Il simulatore è quindi costituito da tre unità funzionali: *fetch unit*, *decode unit* ed *execution unit*.

Due unità funzionali vengono unite da una *instruction queue* che si comporta come un buffer di istruzioni e modella i registri fisicamente presenti fra due stadi consecutivi di una pipeline. Queste code non sono programmate come dei registri master-slave, una unità funzionale immette l'istruzione che ha terminato di elaborare nella sua coda d'uscita e nel ciclo di clock successivo, l'unità funzionale a valle la prenderà in consegna leggendola dalla stessa coda.

InputQueue ed *OutputQueue* sono delle code speciali, la prima legge le istruzioni da simulare direttamente dal buffer d'ingresso[6], l'OutputQueue in modo duale scrive l'istruzione sul file d'uscita[7] non appena la riceve.

La memoria è una risorsa, piuttosto complessa e strutturata[8], il cui scopo è quello di simulare l'impatto di una intera gerarchia di memoria sulle prestazioni del codice eseguito. Lo sviluppatore ha la possibilità di definire la risorsa memoria in modo da riflettere la struttura di quella fisicamente presente, specificando alcuni parametri di configurazione per ciascun livello della gerarchia nonché il tipo di cache adottate e la loro politica.

[6]Non appena riceve la richiesta di acquisire una nuova istruzione.

[7]Quello che riporta su file la coda d'uscita in realtà sono tutta una serie di informazioni relative ad esempio alla latenza dell'istruzione.

[8]Per implementarla abbiamo fatto uso delle classi sviluppata da Carlo Pinciroli come estensione di TrIBeS al fine di simulare generiche gerarchia di memoria.

```
while no prefetchBuffer.pieno() do
    i ⟵ InputQueue.getInstruction();
    stallaPerCicli(leggiMemoria());
    prefetchBuffer.inserisci(i);
end
```

Figura A.3: *Algoritmo di simulazione per la FetchUnit.*

A.2.2 L'unità funzionale: *FetchUnit*

Il suo compito è quello di simulare il recupero delle istruzioni dalla memoria per caricarle all'interno del *prefetch buffer*, a partire dal quale verranno poi inoltrate nello stadio di decodifica.

Durante la simulazione le istruzioni vengono estratte dalla *ImputQueue* e caricate in una coda interna all'unità funzionale. I possibili ritardi dovuti all'accesso alla memoria vengono emulati accedendo alla risorsa memoria. Quest'ultima abilità l'inoltro della istruzione verso l'unità di decodifica solo quando sono passati il numero di cicli di clock necessari a recuperare e fornire il dato richiesto. Sostanzialmente quindi l'unità funziona, ricevuta la nuova istruzione continua a fare polling presso la risorsa memoria finché questa non le comunica che è possibile far proseguire l'esecuzione. Con riferimento al modello per il tracciamento dei consumi proposto nel 3.2.1, durante questo periodo la FetchUnit si trova nello stato di *resource stalled*. In pseudocodice l'implementazione del metodo **`execInstruction`** della *FetchUnit* viene riportata in Figura A.3. Notiamo per completezza che il *prefetch buffer* può non essere presente in talune implementazioni dell'*ARM7TDMI*, conseguentemente la scelta è stata quella di implementarlo comunque ma in modo parametrico rispetto alla sua dimensione. Ovviamente un buffer di dimensione unitaria non ha particolari effetti sulla simulazione e può essere usato per modellare le architetture che ne sono prive.

A.2.3 L'unità funzionale: *DecodeUnit*

Come abbiamo già discusso, nell'*ARM7TDMI* i data hazard non si possono verificare grazie alla struttura intrinseca della pipeline, infatti le istruzioni leggono i registri solo nell'ultimo stadio della pipeline ed a quel punto tutti gli operandi sono stati già eventualmente scritti. Conseguentemente la *DecodeUnit* viene implementata banalmente come uno stadio vuoto che non consuma alcuna microistruzione ma che semplicemente conserva ogni istruzione per un ciclo di clock e poi la fa passare allo stadio successivo non appena quest'ultimo è disponibile.

A.2.4 L'unità funzionale: *ExecuteUnit*

La maggior parte dell'elaborazione all'interno di un $ARM7TDMI$ avviene nello stadio di esecuzione modellato dall'`ExecuteUnit` nel simulatore TrIBeS.

Molte istruzioni richiedono un numero prefissato di cicli di clock nello stadio di esecuzione, ciò viene emulato usando microistruzioni `require`. Quando una di queste microistruzioni viene letta dall'*ExecutionUnit* si controlla l'unità di destinazione e se coincide con la stessa *ExecutionUnit* allora l'unità viene "stallata" per il numero di cicli richiesti. Praticamente, all'interno dell'unità, viene settato un contatore pari al valore della `require` decrementato poi ad ogni ciclo finché non si annulla abilitando l'istruzione ad uscire dall'*ExecutionUnit*. Sempre in riferimento al modello per il tracciamento dei consumi proposto nel 3.2.1, durante questo periodo l'unità si trova nello stato di *execution*.

L'unità di esecuzione è in grado di tener conto anche delle latenze dovute ad accesso alla memoria. Le istruzioni assembler di **`Load`** e **`Store`** vengono tradotte rispettivamente in microistruzioni `load` e `store` del simulatore. Quando una di queste microistruzioni viene riconosciuta dall'*ExecutionUnit* quest'ultima fa una richiesta alla risorsa memoria per l'indirizzo specificato. La risorsa memoria è così in grado di determinare il numero di cicli di clock che saranno necessari affinché il dato sia disponibile. Ad ogni ciclo del simulatore l'*ExecutionUnit* fa polling verso la risorsa memoria finché questa non gli dirà che il dato è pronto e quindi l'istruzione è abilitata a proseguire.

Una particolare caratteristica dell'architettura ARM è la capacità di cambiare l'instruction set utilizzato durante l'esecuzione stessa, passando dal modo ARM a quello THUMB o viceversa. Questi cambiamenti avvengono in corrispondenza delle istruzioni assembler **`BX`** che vengono tradotte per il simulatore in microistruzioni *use* riferite alla risorsa *ThumbFlag*.

Riportiamo in Figura A.4 lo pseudocodice del metodo **`execInstruction`** dell'unità di esecuzione *FetchUnit*.

A.2.5 Esempio di compilazione in microistruzioni

Affinché TrIBeS possa simulare correttamente l'esecuzione delle istruzioni all'interno della pipeline, come sappiamo, è necessario tradurre ogni istruzione assembler in un conveniente insieme di microistruzioni del simulatore. A modo di esempio vediamo come dev'essere tradotta in microcodice una istruzione di **`Load`** al fine di ottenerne una corretta simulazione.

Anzitutto l'istruzione[9] dev'essere letta da memoria nella fase di fetch. Una microistruzione chiede alla *FetchUnit* di caricare da memoria la word che contie-

[9]Questo vale per qualunque istruzione eseguita.

```
riempiCodaInterna();
foreach istruzione i in codaInterna do
    foreach micro-istruzione m in i do
        if FUrichiesta(m) == EX:Execute then
            switch tipoMicroistruzione(m) do
                case require
                | stallaPerCicli(cicliRichiesti(m));
                end
                case load
                | stallaPerCicli(leggiMemoria(<indirizzo>));
                end
                case store
                | stallaPerCicli(scriviMemoria(<indirizzo>));
                end
                case use
                | thumbFlag ⟵ (thumbFlag +1) mod 2 ;
                end
            end
        end
    end
end
```

Figura A.4: *Algoritmo di simulazione per l'ExecutionUnit.*

ne l'istruzione:

```
FE:Fetch load <indirizzo>
```

Successivamente, nella nostra architettura, l'istruzione attraversa l'unità di decodifica senza fare nulla in particolare per poi essere ulteriormente "elaborata" nell'unità di esecuzione. La maggior parte dell'esecuzione avviene proprio nell'*ExecutionUnit*: nel primo ciclo di clock l'indirizzo di memoria del dato da caricare viene calcolato, nel ciclo successivo il dato viene richiesto alla memoria ed infine, nel terzo ciclo di clock nel caso non ci siano latenze da parte della gerarchia di memoria[10], il dato richiesto è disponibile per essere memorizzato nel registro di destinazione.

É importate fare in modo che l'*EsecutionUnit* si comporti esattamente come la pipeline reale e quindi in particolare, che l'accesso alla risorsa memoria avvenga

[10]Ciò avviene in corrispondenza di un *cache hit*. Nel caso di *cache miss* la risorsa memoria determinerà il ritardo in cicli di clock necessario al reperimento del dato.

```
FE:Fetch    load <indirizzo>
EX:Execute  require 1 EX:Execute
EX:Execute  store <indirizzo>
```

Figura A.5: *Esempio completo di microcodice per l'istruzione* **`Store`**

solo in corrispondenza del secondo ciclo di clock. Per ottenere questo risultato serve aggiungere alla precedente una opportuna sequenza di microistruzioni[11]:

```
EX:Execute require 1 EX:Execute
EX:Execute load <indirizzo>
EX:Execute require 1 EX:Execute
```

Il simulatore esegue le microistruzioni nell'esatto ordine in cui vengono riportate, di conseguenza l'*EsecutionUnit* trovandosi prima una microistruzione `require` e quindi stalla per un ciclo di clock. Nel secondo ciclo di clock trova una `require` e quindi si ha l'accesso alla memoria. Infine, quando la memoria segnalerà la disponibilità del dato richiesto, l'unità stalla per un ulteriore ciclo di clock simulando correttamente la fase di *write back* del dato nel registro di destinazione.

Una istruzione di **`Store`** viene tradotta in modo simile alla precedente, ad eccezzione dell'ultimo ciclo in quanto non è necessaria una *write back*. Il microcodice completo per questo tipo di istruzione è mostrato in Figura A.5.

Una considerazione finale va fatta in merito alle istruzioni di salto condizionale. Come abbiamo fatto notare l'*ARM7TDMI* non dispone di particolari soluzione per la predizione dei salti ma sfrutta invece il meccanismo dell'esecuzione condizionale. Ciò comporta diverse latenza per le istruzioni di salto a seconda che il salto divenga un *taken branch* oppure un *not taken branch*. L'uso corretto delle istruzioni `require` è sufficiente a modellare correttamente il comportamento del processore. Un *taken branch* viene emulato stallando l'*ExecutionUnit* per tre cicli di clock, con una `EX:Execute require 3`, mentre un *not taken branch*, che sappiamo equivalente ad una istruzione di salto convertita poi in NOP, è modellato usando semplice una microistruzione `EX:Execute require 1`.

[11] Per chiarezza ricordiamo che la prima parte di una microistruzione indica l'unità funzionale cui è destinata mentre la parte terminale indica in quale unità funzionale l'istruzione dovrà proseguire la sua elaborazione

```
1 0000dc94  –  ldr    r3<0x000000>, [r11<0x1ffff4>, #-16]
1 0000dc98  –  cmp    r3<0x000000>, #1
1 0000dc9c  N  bgt    0000df70
1 0000dca0  –  stmdb  sp<0x1ffbcc>!, r0<0x015cc0>, r1<0x1ffff8>
```

Figura A.6: *Esempio di una porzione di traccia d'esecuzione*

A.3 Libreria Atomic per l'*ARM7TDMI*

Come descritto nel 2.4.3 a pagina 42, lo scopo di Atomic è quello di elaborare una *traccia d'esecuzione* convertendo ciascuna istruzione assembler in un insieme di microistruzioni. Le microistruzioni utilizzate dovranno essere adatte a simulare correttamente con TrIBeS il comportamento della istruzione cui si riferiscono.

Fondamentalmente Atomic è un compilatore costituito da un insieme di macro generali, da utilizzare al fine di produrre microistruzioni compatibili con il formato di TrIBeS, con in aggiunta delle funzioni specifiche per ogni architettura. Il compilatore per una particolare architettura viene scritto in C usando come generatori di scanner e parser rispettivamente *flex* e *bison*[12].

Di seguito descriviamo in dettaglio come è stata implementata la libreria Atomic per l'*ARM7TDMI*. Grazie al fatto che la pipeline del processore considerato è piuttosto semplice non si sono riscontrate particolari difficoltà nel definire la struttura generale della libreria. Tuttavia alcune caratteristiche particolari dell'architettura, come la possibilità di cambiare l'instruction set durante l'esecuzione, hanno richiesto un certo impegno per gestire correttamente la microcompilazione.

A.3.1 Struttura della traccia d'esecuzione

Come riferimento riportiamo in Figura A.7 uno stralcio di traccia d'esecuzione pronta per essere processata con Atomic[13]. Possiamo osservare che le istruzioni seguono tutto il medesimo schema:

```
<INDIRIZZO> [TN-] OPCODE[COND][FLAG] [PARAMETRI]
```

<INDIRIZZO> rappresenta l'indirizzo dell'istruzione in memoria, mentre il carattere a seguire indica se l'istruzione di salto cui è associato è un *taken branch*

[12]Come riferimento per questi strumenti consigliamo [35], facilmente reperibile in *Internet*, in aggiunta alle esaustive pagine di manuale di ciascuno dei due tools.

[13]Vedremo successivamente come una traccia così formattata viene ottenuta per la nostra architettura.

'T' oppure un *not taken branch* 'N', nel caso l'istruzione non sia un salto viene usato il simbolo '–'.

A partire da `OPCODE`[14], segue l'istruzione assembler vera e propria per capire la quale occorre ricordare che l'istrucion set dell'arm7 è fortemente basato sul paradicma dell'esecuzione condizionale e consente di esplicitare delle "post-azioni" da eseguire come effetto collaterale dell'istruzione stessa. Tutte queste opzioni vengono specificate semplicemente mediante dei suffissi che seguono il codice operativo.

Ogni istruzione ARM può essere condizionata e viene effettivamente eseguita solamente se la particolare condizione indicata da `COND`[15] risulta essere verificata[16]. Nel caso in cui la condizione non valga è necessario che Atomic stesso si preoccupi di riconvertirla in una NOP generando l'opportuno microcodice per l'*ExecutionUnit*. Il campo `FLAG` invece dettaglia meglio il particolare tipo di istruzione considerata.

Considerando poi i `PARAMETRI` dobbiamo osservare che l'architettura supporta diversi modi di indirizzamento[17], spesso un parametro è costiuito da un indirizzo base più uno spiazzamento. Affinchè l'emulazione dell'accesso alla memoria operato da TrIBeS sia efficace è necessario che Atomic calcoli direttamente l'indirizzo finale da usare come parametro della corrispondente microistruzione.

Notiamo infine che, nella traccia d'esecuzione, ogni registro viene "annotato" con il valore in esso contenuto immediatemenre prima dell'esecuzione dell'istruzione in cui compare. Questi valori saranno usati dal compilatore Atomic al fine di generare correttamente gli indirizzi in memoria ad esempio per le operaizoni di load/store.

A.3.2 Struttura delle microistruzioni

La pipeline di simulazione per l'*ARM7TDMI*, come abbiamo visto nel paragrafo precedente, è costituita da tre soli stadi. Conseguentemente ogni istruzione generalmente dev'essere tradotta in almeno tre microistruzioni `require`, una per ciascuno stadio. Negli stadi di *fetch* e *decode* tuttavia tutte le istruzioni richiedono un solo ciclo di clock e quindi la rispettiva microistruzione può essere omessa[18].

Nella *FetchUnit* viene simulato l'accesso a memoria con l'ingresso di ogni istruzione in pipeline, una microistruzione di `load` è quindi necessaria per indi-

[14]Che rappresenta il codice mnemonico dell'istruzione assembler.

[15]Si veda [33] p.1-20 per i dettagli.

[16]Abbiamo già fatto notare come questo approccio venga effetivamente utilizzato ad esempio per gestire i salti, trasformando in semplici NOP quelli non presi.

[17]Si veda [33] p.1-16 per i dettagli.

[18]TrIBeS infatti si preoccupa già di fare attraversare ad ogni istruzione ciascuno statio della pipeline.

```
INSTRUCTION_START <iclass> <latency>
     FE:Fetch load <indirizzo>
     EX:Execute require 1
INSTRUCTION_END
```

Figura A.7: *Struttura base del microcodice di ogni istruzione*

```
INSTRUCTION_START 49 7
     FE:Fetch load 0xdca0
     EX:Execute require 1
     EX:Execute store 0x1ffbbc
     EX:Execute store 0x1ffbc0
     EX:Execute require 4 EX:Execute
INSTRUCTION_END
```

Figura A.8: *Struttura base del microcodice di ogni istruzione*

care l'indirizzo dell'istruzione caricata e che verrà poi passato alla risorsa memoria. Il resto delle microistruzioni sarà invece realtivo all'*ExecutionUnit* e cambierà in funzione della particolare istruzione assembler considerata.

La struttura base del microcodice di ogni istruzione è mostrata in Figura A.7. Le stringhe `INSTRUCTION_START` ed `INSTRUCTION_END` delimitano le microistruzioni realtive ad ogni istruzione assembler. I due parametri <*iclass*> e <*latency*> indicano invece rispettivamente la classe di istruzione e la sua latenza nominale.

Dalla struttura base si differenziano le istruzioni più complesse che prevedono una maggiore elaborazione in fase di esecuzione e che per questo vengono modellate con un maggior numero di microistruzioni per l'*ExecutionUnit*. Questo è il caso ad esempio delle operazioni di **`Load`** e **`Store`** che ammettono anche la possibilità di lavorare su più registri contemporaneamente. In questo caso basterà semplicemente generare una microistruzione di load/store per ciascuno degli indirizzi richiesti. A titolo di esempio riportiamo in Figura A.8 la traduzione in microcodice dell'ultima delle istruzione assembler riportata nella traccia d'esecuzione dell'esempio precedente.

A.3.3 Gestione del doppio instruction set

Una delle caratteristiche dell'*ARM7TDMI* è non solo il supporto ad un doppio instruction set, a 32 e 16 bit, ma pure la capacità di passare da uno all'altro durante l'esecuzione. All'interno di una stessa traccia d'esecuzione possiamo quindi trovarci istruzioni di entrambi gli insiemi; il passaggio da uno all'altro è governato dalla particolare istruzione assembler **BX** (*Branch and eXchange instruction set*):

```
BX [COND]<registro>
```

Questo particolare comportamento rende più complessa la creazione del compilatore Atomic per questa architettura. La soluzione pensata prevede l'utilizzo di due grammatiche diverse, da usare in funzione dello stato, facendo poi in modo che il compilatore passi dall'una all'altra all'occorrenza di una istruzione **BX**. Tale istruzione è condizionata al valore di un bit che indica l'instruction set da utilizzare, Atomic dovrà valutare la condizione ed eventualmente cambiare il parser da utilizzare per le istruzioni a seguire. La scrittura dello scanner *flex* è stata invece più semplice, infatti è bastato scrivere un solo analizzatore che si occupa di riconoscere gli elementi sintattici di entrambe le grammatiche ritornando il corrispettivo token semantico al parsificatore *bison*.

La Figura A.9 mostra la strutturazione generale della libreria Atomic creata a supporto dell'*ARM7TDMI*. Come mostrato la libreria ha un unico entry-point rappresentato dalla funzione **myparse()**. Questa funzione si preoccupa di controllare lo stato del compilatore ed invocare di volta in volta il corretto scanner. Gli analizzatori sintattici sono infatti due, generati con due diverse grammatiche in *bison*. La tecnica utilizzata è quella del *double parsing*: sostanzialmente si fa in modo che ogniuno dei parser creati esporti funzioni equivalenti con nomi diversi grazie alla direttiva **%name-prefix**.

La **myparse** quindi non è altro che una funzione wrapper verso il corretto parser in funzione dello stato, il suo codice è riportato in Figura A.10. Ciascuna delle funzioni di parsing richiama poi lo stesso scanner rappresentato dalla funzione interna alla libreria definito con la direttiva:

```
define YY_DECL int mylex
```

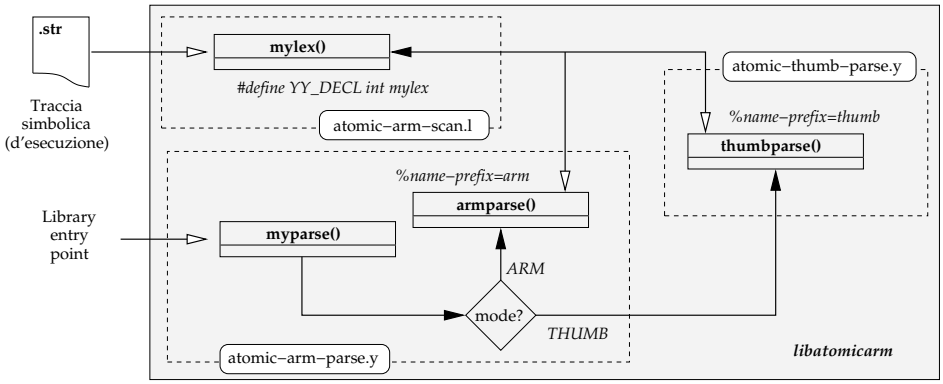


Figura A.9: Struttura della libreria Atomic per l'$ARM7TDMI$

A.3.4 Generazione della traccia d'esecuzione

Generalmente le tracce d'esecuzione vengono generate con i tools già discussi, quando abbiamo illustrato il flusso assembly nel Par. 2.4.2 a pagina 40, parlando di preprocessing della traccia d'esecuzione. Nel caso dell'$ARM7TDMI$ tuttavia si è deciso di non utilizzare quei tools, le ragioni di tale scelta sono diverse e le vedremo successivamente.

L'approccio che si è utilizzato è stato quello di generare la traccia d'esecuzione stessa per simulazione: utilizzando GDB. Il debugger GNU supporta infatti l'esecuzione di programmi compilati per instruction set diversi da quello dell'architetture sulla quale viene fatto girare. Questa caratterisitca, inserita nel debugger al fine di rendere possibile il controllo di programmi compilati per architetture diverse da quella utilizzata dallo sviluppatore, sfrutta la stessa syscall *ptrace* utilizzata anche dal tool *bintrace*. Per abilitare tale funzionalità è necessario procurarsi i sorgenti di GDB[19] e *crosscompilarlo* definendo le variabili HOST=*i386* e TARGHET=*arm*. In questo modo viene generato un debugger in grado di tracciare binari in formato `arm-none-elf`. Bisogna osservare infatti che questa soluzione consente il tracciamento di programmi *standalone* che girano nativamente su arm senza il supporto di un sistema operativo, poichè GDB in simulation mode non supporta il tracing delle chiamate di sistema.

Il principale vantaggio di questa soluzione è la velocità di produzione della traccia. Infatti l'uso di un simulatore ci consente di lavorare su una qualunque architettura *i386* con una maggiore potenza di calcolo. Così facendo ed utilizzando in particolare per la simulazione la stessa piattaforma dual Pentium*III* già

[19]Disponibili su WEB nella pagina del progetto:
`http://www.gnu.org/software/gdb/gdb.html`

```
int myparse() begin
    /* scanning reult                                          */
    int exit_code = 0 ;
    /* Starting scanning in ARM mode.  At armparse
       return cheking if a mode change occurred                */
    exit_code = armparse;
    while ! exit_code and thumbMode do
        /* mode change occurred...  doing THUMB parsing */
        exit_code = thumbparse();
        if exit_code then
            break ;
        end
        exit_code = armparse();
    end
    return exit_code ;
end
```

Figura A.10: *Codice della funzione per il* double parsing

descritta a pag 82, siamo riusciti a ridurre i tempi per la generazione della traccia di un ordine di grandezza circa rispetto a quelli necessari per l'elaborazione sulla board con *ARM7TDMI* a 200*Mhz* in nostro possesso[20]. Altro non trascurabile vantaggio è la possibilità di poter lavorare indipendentemente dalla disponibilità di una piattaforma hardware. Ciò rende eventualmente possibile l'adattamento della soluzione anche ad altre architetture non fisicamente disponibili, semplicemente sfruttando il supporto che GDB fornisce, come Instruction Set simulator, per un esteso numero di piattaforme.

In Figura A.11 viene mostrato lo schema generale dei tools usati per la generazione della traccia d'esecuzione. Il binario in formato `arm-none-elf`, compilato abilitando l'inclusione dei simboli di debug e "linkato" staticamente[21], viene passato allo shell script **`strace_arm.sh`** assieme ad un file di comandi per GDB. L'idea infatti è quella di sfruttare la possibilità di eseguire il debugger in batch mode, passandogli un elendo di comandi da eseguire al fine di produrre i dati voluti senza necessità di interazione con l'operatore[22].

Quello che viene chiesto a GDB sostanzialmente è di produrre in uscita l'i-

[20] Precisamente la evaluation board Altera Excalibur EPXA1DB.

[21] Usando rispettivamente le opzioni *-g* e *–static* del compilatore.

[22] Questa è un'altra delle tante possibilità offerte dal versatilissimo debugger GDB, utilizzata principalmente quando si ha necessità di debuggare programmi che devono essere eseguiti senza interruzioni.

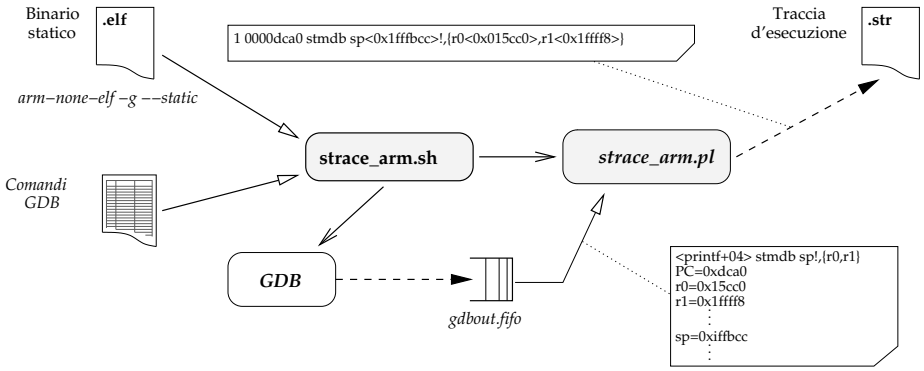


Figura A.11: Schema dei tools per la generazione della traccia d'esecuzione usando GDB come Instruction set simulator.

struzione assembler che sta per eseguire, completa di parametri, l'indirizzo in memoria di tale istruzione ed il valore di tutti iregistri; e questo lo deve fare per ogni nuova istruzione eseguita[23].

L'uscita di GDB, per ottimizzazione, viene inserita in una pipe creata dallo script bach che invoca pure un semplice script PERL "attaccandolo" all'altro capo della stessa pipe. Il codice di **`strace_amr.pl`** altro non è che un "filtro" per la corretta formattazione della traccia d'esecuzione. Prendendo i dati che GDB gli passa crea, per ciascuna istruzione assembly eseguite, la corripondente istruzione della traccia d'esecuzione avente fra l'altro i registri annotati con il valore del loro contentuto prima dell'esecuzione dell'istruzione.

[23]Un tale comportamento lo si ottiene semplicemente definendo un *watchpoint* sul registro PC ed associandogli una serie di comandi da eseguire, l'ultimo di questi è un *continue* che fa proseguire il debugging.

APPENDICE B

Glossario dei termini

Vengono di seguito raccolte le definizioni di alcuni dei termini e delle abbreviazioni utilizzate in questo lavoro. Tutte le definizioni sono state prese dal progetto di enciclopedia libera:

WIKIPEDIA
`http://www.wikipedia.org/`

AES: *Advanced Encryption Standard*.
In Crittografia, l'Advanced Encryption Standard (AES), conosciuto anche come Rijndael, è un algoritmo di cifratura a blocchi utilizzato come standard dal governo degli Stati Uniti d'America. Data la sua sicurezza e le sue specifiche pubbliche si presume che in un prossimo futuro venga utilizzato in tutto il mondo come è successo al suo predecessore, il Data Encryption Standard (DES). È stato adottato dalla National Institute of Standards and Technology (NIST) e dalla US FIPS PUB 197 nel novembre del 2001 dopo 5 anni di studi e standardizzazioni.

L'algoritmo è stato sviluppato da due crittografi Belgi, Joan Daemen e Vincent Rijmen, che lo hanno presentato al processo di selezione per l'AES con il nome di Rijndael, nome derivato dai nomi degli inventori.

Rijndael è un'evoluzione del primo algoritmo sviluppato da Daemen a Rijmen, Square. Square era stato sviluppato per SHARK.

A differenza del DES, Rijndael non usa una rete di Feistel bensì una rete a sostituzione e permutazione. La velocità è una sua caratteristica, sia quanto viene eseguito in software che quando lo è in hardware, è relativamente semplice da implementare, e richiede poca memoria.

Questo nuovo standard di cifratura sta sostituendo i precedenti standard e la sua diffusione continua ad aumentare.

Architetture superscalari: *Superscalar architecture*.
Un'architettura superscalare implementa una forma di parallelismo a livello di CPU permettendo al sistema complessivo di elaborare le istruzioni molto più velocemente di come sarebbe in grado di fare altrimenti a parità di frequenza di clock.

I processori più semplici sono scalari. Un processore scalare elabora una singolo dato alla volta. In un processore vettoriale, al contrario, una singola istruzione opera contemporaneamente su un insieme di datai. La differenza è analoga alla differenza fa l'aritmetica scalare e vettoriale.

Un processore superscalare è una sorta di miscuglio fra i due. Ogni istruzione processa un solo dato, ma ci sono molteplici unità di processo così che istruzioni multiple possano effettivamente elaborare i propri dati contemporaneamente. Un processore superscalare solitamente sostiene un tasso di esecuzione maggiore di un'istruzione per ciclo macchina. Questo è essenzialmente lo scopo dell'architettura.

Il semplice fatto di processare istruzioni multiple allo stesso tempo non rende un'architettura superscalare. Un semplice pipelining, dove una CPU può caricare una istruzione mentre esegue operazioni aritmetiche per la precedente e memorizza i risultati di quella prima ancora (eseguendo 3 istruzioni contemporaneamente) non è un processo superscalare.

Sostanzialmente tutte le CPU general purpose che si sono sviluppate dal 1998 sono superscalari.

Caching delle memorie: *Memory caching*.
La cache è un insieme di dati che duplicano i valori originari, memorizzati altrove o computati in precedenza, quando i dati originali sono costosi da recuperare[1] o da computare. Una volta che i dati sono immagazzinati nel cache, gli utilizzi futuri possono essere fatti accedendo alla copia piuttosto che accedendo o computando nuovamente i dati originali, così il tempo di accesso medio risulta inferiore.

Quando il client della cache (CPU, web browser, sistema operativo) vorrebbe accedere a dei dati, prima verifica se sono presenti in cache. Quando il dato necessario è presente in cache viene utilizzato in luogo di quello originario. Questo fatto è conosciuto come *cache hit*. Per verificare rapidamente la presenza di un dato in cache vengono utilizzati dei tag. Un tag è un identificatore associabile univocamente ad un certo dato. I tag vengono organizzati e memorizzati all'interno della cache stessa in modo da risultare velocemente recuperabili. Così, per

[1] Solitamente in termini di tempo di accesso.

esempio, un web browser potrebbe verificare la sua cache locale su disco per vedere se ha una copia dei contenuti di una pagina web remota prima di scaricarla dalla rete. In questo esempio, l'URL è il tag, e il contenuto della pagina web è il dato. La percentuale di accessi che risultano in un cache hits è conosciuta come l'*hit rate* della memoria cache.

Nella situazione opposta, quando cioè la cache consultata non contiene i dati desiderati, è conosciuta con il nome di *cache miss*. Il dato originario dev'essere recuperato da quello che viene detto back-storage durante la gestione del miss e solitamente viene pure inserita nel cache, in modo che sia presente per un eventuale prossimo accesso. Se la cache ha una capacità di immagazzinaggio limitata, può darsi che debba eliminare qualche altra entry per fare spazio. L'euristica utilizzata per selezionare la entry da eliminare è detta *replacement policy*. Una popolare replacement policy, chiamata Last Recently Used (LRU), rimpiazza la entry utilizzata meno recentemente.

Cache miss: *Vedere "Caching delle memorie".*

CPI: *Clock cycles Per Instruction*[2].

Definisce il numero (nominale) di cicli di clock necessari all'esecuzione completa di ogni istruzione di un programma ed è definito, in prima istanza, dalla relazione:

$$CPI = \frac{CPU_{clock\ cycles\ for\ a\ program}}{Instruction\ Count}$$

DSA: *Digital Signature Algorithm*.

Il Digital Signature Algorithm (DSA) è lo standard attuale del governo federale degli USA per le firme digitali. Venne proposto dal National Institute of Standards and Technology (NIST) nell'agosto del 1991 per essere utilizzato come Digital Signature Standard (DSS), secondo le specifiche FIPS 186, ed adottato poi ufficialmente nel 1993. Una revisione minore venne pubblicata nel 1996 come FIPS 186-1, e lo standard è stato sviluppato ulteriormente nel 2000 come FIPS 186-2.

DSA è coperto dal brevetto USA 5,231,668, registrato il 26 luglio 1991 e attribuito a David W. Kravitz, un ex impiegato della NSA.

Hazard:

In una architettura informatica, un hazard è un potenziale problema che si può verificare in un processore pipelined. Ci sono tipicamente tre tipi di hazard: data hazard, control hazard e structural hazard.

Le istruzioni in un processore pipelined vengono eseguite in differenti stadi, così che ad ogni istante di tempo più istruzioni possono essere elaborate con-

[2]Si veda [36], Par. 1.6 –"Quantitative Principles of Computer Design"

temporaneamente. Un hazard si verifica quando, in seguito a questa esecuzione sovrapposta, due o più istruzioni vanno in conflitto.

data hazard si verificano quando i dati non vengono modificati nell'ordine corretto. Ci sono tre situazioni in cui può accadere:

1. Read after Write (RAW): la lettura di un dato avviene prima di una scrittura quando invece doveva seguirla. Potrebbe accadere che una istruzione non abbia ancora finito di scrivere in memoria, quando l'istruzione successiva tenta di leggere gli stessi dati

2. Write after Read (WAR): la lettura di un dato avviene dopo la sua scrittura quando invece doveva precederla. Potrebbe accadere che una istruzione non abbia ancora letto un dato ma questo viene modificato da un'altra istruzione.

3. Write after Write (WAW): due scritture successive nella stessa locazione di memoria possono terminare in ordine inverso lasciando quindi un valore scorretto in memoria.

A **structural hazard** si verifica quando una parte dell'hardware del processore è necessaria per due o più istruzioni contemporaneamente. Uno structural hazard può verificarsi, ad esempio, quando un programma deve eseguire una istruzione di salto seguita da una istruzione computazionale. Poiché sono eseguite in parallelo, e poiché un salto tipicamente è lento (necessitando di effettuare in confronto, la modifica del PC e la scrittura di alcuni registri), è possibile che in talune architetture l'istruzione computazionale e quella di salto necessiteranno entrambe della ALU contemporaneamente.

control hazard si verifica quando al processore viene chiesto ad esempio di effettuare un salto condizionato, sulla base del valore di un certo parametro, ad una locazione di memoria diversa da quella immediatamente successiva alla sua. In tal caso, il processore non può sapere in anticipo se dovrà o meno processare l'istruzione immediatamente successiva a quella di salto. Questo può comportare l'esecuzione da parte del processore di una sequenza di azioni non volute. Ci sono diverse tecniche sia per prevenire il verificarsi di questi hazard che per gestirli nel momento in cui si concretizzano:

Forwarding consiste nell'inoltrare i risultati delle elaborazioni all'indietro nella pipeline verso gli stadi a monte.

Ad esempio supponiamo di voler scrivere il valore 3 nel registro $r1$, che già contiene 6, e successivamente aggiungerci 7 e memorizzare infine il risultato nel registro $r2$. Una corretta esecuzione porterebbe $r2$ a contenere il

valore 10. Tuttavia, siccome la prima istruzione non è ancora stata eseguita completamente nel momento in cui inizia l'esecuzione della seconda, alla fine avremo in $r2$ il valore 13. Al fine di prevenire queste anomalie si riporta verso gli stadi precedenti della pipeline il risultato della prima operazione (3). Questo stadio della pipeline ha ora due valori in input: il nuovo valore inoltrato dallo stadio a valle (3) e quello vecchio (6). Di conseguenza questo approccio richiede l'introduzione di una opportuna logica di controllo per determinare quale dei due valori utilizzare.

Bubbling the Pipeline questa tecnica, nota anche come *pipeline break* o *pipeline stall*, consente di evitare sia i branch hazard che quelli di controllo.

Quando una nuova istruzione entra in pipeline una opportuna logica di controllo verifica se la sua esecuzione potrebbe essere a rischio di hazard. Se il rischi sussiste la stessa logica di controllo inserisce un opportuno numero di NOPs. Così facendo, prima che l'istruzione a rischio di hazard venga eseguita quella precedente avrà il tempo necessario per essere completata evitando di fatto l'insorgere dell'hazard.

Nel caso in cui il numero di NOPs inserite sia uguale al numero di stadi della pipeline, il processore sarà stato "svuotato" da tutte le istruzioni; in questo caso si parla di "*flushing the pipeline*". Ogni forma di stallo introduce dei delay prima che il processore possa riprendere l'esecuzione.

IPSec: *IP SECurity*.
standard per la sicurezza delle comunicazioni su Internet Protocol, previa cifratura ed autenticazione di tutti i pacchetti scambiati, fornisce sicurezza a livello di rete.

IPSec è in realtà una collezione di protocolli per la sicurezza del flusso di pacchetti e per lo scambio di chiavi necessarie alla creazione del canale di comunicazione sicuro. La sicurezza del flusso di pacchetti è delegata a due protocolli: Encapsulating Security Payload (ESP) fornisce autenticazione, confidenzialità dei dati ed integrità dell'informazione; Authentication Header (AH) fornisce autenticazione ed integrità dei messaggi ma non la confidenzialità. Attualmente solo un protocollo per lo scambio della chiavi è stato definito: IKE.

PDA: *Personal Digital Assistant*.
si tratta di dispositivi portatili pensati inizialmente come agende elettroniche personali ma che attualmente sono divenuti dispositivi molto versatili. Tipicamente, un PDA base include diversi programmi come orologio, datario, rubrica indirizzi, gestione degli appuntamenti, memo e calcolatrice. Uno dei vantaggi principali dei PDA è la loro capacità di sincronizzare i loro dati con quelli di una computer.

Pipeline stall: *Vedi "Hazard"*.

Pipelining:
la pipeline è una soluzione architetturale utilizzata nei microprocessori per aumentarne le prestazioni migliorando il throughput[3].

Ogni istruzione può essere scomposta in attività più elementari ciascuna delle quali rappresenta un task all'interno del microprocessore. Le attività svolte da una CPU sono sincronizzate da un segnale di clock. Ogni attività elementare in cui si può scomporre un'istruzione richiede almeno un ciclo di clock per esser completata. Ciascuna di queste attività viene eseguita effettivamente in una diversa componente hardware del processore.

I primi processori, non pipelined, potevano eseguire una sola attività ad ogni ciclo di clock. I passi eseguiti, nell'ordine corretto, erano ad esempio:

1. Leggi la prossima istruzione
2. Leggi gli operandi, se necessario
3. Esegui l'istruzione
4. Scrivi i risultati in memoria

Un tale approccio, seppur semplice, implica un inevitabile spreco dovuto al fatto che mentre ad esempio si sta eseguendo la somma di due numeri, la circuiteria che carica i dati da memoria rimane inattiva in attesa che l'addizzione venga completata.

Il pipelining aumenta le prestazioni riducendo il tempo d'inattività di ogni componente hardware del microprocessore. Le architetture a pipeline includono della circuiteria specializzata che esaminando le istruzioni da eseguire le suddividono in sotto-istruzioni. Più sotto-istruzioni di istruzioni diverse possono essere eseguite simultaneamente da diverse componenti hardware del processore. L'unità di controllo della pipeline detta *"pipeline controller"* assicura che tutto avvenga in modo da non modificare il risultato finale.

RISC: *Reduced Instruction Set Computer*
I processori RISC hanno una unità di controllo cablata molto semplice e riservano invece molto spazio per i registri interni: una CPU RISC ha di solito da un minimo di un centinaio ad alcune migliaia di registri interni generici, organizzati in un file di registri. Il tipico set di istruzioni RISC è molto piccolo, circa sessanta o settanta istruzioni molto elementari (logiche, aritmetiche e istruzioni di trasferimento memoria-registro e registro-registro): hanno tutte lo stesso formato e la stessa lunghezza, e tutte o quasi vengono eseguite in un solo ciclo di clock. Sono presenti solo un numero ristretto di metodi di indirizzamento. Il fatto di avere

[3]Numero di istruzioni completate nell'untià di tempo

un formato unico di istruzione permette di strutturare l'unità di controllo come una pipeline, cioè una catena di montaggio a più stadi: questa innovazione ha il grosso vantaggio di ridurre il critical path interno alla CPU e consente ai RISC di raggiungere frequenze di clock più alte rispetto agli analoghi CISC.

Nel caso di context-switch o di chiamata a subroutine o comunque di uso dello stack i RISC invece di accedere alla memoria di sistema usano un meccanismo chiamato register renaming, che consiste nel rinominare i registri in modo da usare per la nuova esecuzione una diversa zona del file di registri, senza dover accedere alla memoria ogni volta.

La complessità nei RISC si sposta dall'hardware al software: un compilatore che genera codice per CPU RISC deve affrontare un duro lavoro per generare codice compatto ed efficiente, che in ogni caso sarà più grande ed occuperà più memoria dell'equivalente per CISC.

SSL: *Secure Socket Layer*.
assieme al suo successore *Transport Layer Security* (TLS), è un protocollo per le comunicazioni sicure in Internet. Ci sono solo poche differenze fra SSL 3.0 e TSL 1.0 tuttavia le due implementazioni non sono intercambiabili.

SSL fornisce un endpoint di autenticazione e riservatezza della comunicazione in Internet utilizzando la cifratura. In una tipica applicazione solo il server è autenticato[4] mentre il client rimane non autenticato; la mutua autenticazione richiede l'installazione delle chiavi di cifratura PKI direttamente nei client. Il protocollo consente alle applicazioni cliente/server di comunicare in modo tale da prevenire attacchi tipo *eavesdropping*, *tampering* e *message forgery*.

SSL implica un certo numero di fasi al fine di stabilire una connessione sicura:

- ❏ negoziazione dell'algoritmo di cifratura
- ❏ scambio delle chiavi pubbliche ed autenticazione previo certificati
- ❏ cifratura simmetrica del traffico basata su chiper

Nella corso della prima fase il client ed il server l'algoritmo crittografico da utilizzare. Le implementazioni correnti supportano i seguenti:

- ❏ per la cifratura a chiave pubblica: RSA, Diffie-Hellman, DSA or Fortezza
- ❏ per la cifratura simmetrica: RC2, RC4, IDEA, DES, Triple DES or AES
- ❏ per l'hasing: MD5 or SHA

SET: *Secure Electronic Transaction*.
è un protocollo standard per transazioni sicure con carta di credito effettuate su

[4]Ovvero la sua identità e riconosciuta ed assicurata

reti insicure, principalmente Internet, sviluppato nel 1996 da VISA e MasterCard, con il contributo di altri come: GTE, IBM, Microsoft e Netscape.

SET fa uso di tecniche di cifratura, come i certificati digitali e la cifratura a chiave pubblica, al fine di consentire alla parti coinvolte nella comunicazione di autenticarsi reciprocamente e scambiare informazioni in modo sicuro.

Il protocollo è stato fortemente pubblicizzato alla fine degli anni 90 come lo standard approvato dai maggiori gestori di carte di credito, tuttavia non è riuscito a divenire una standard de facto. Le ragioni principali sono la necessità di installare del software sui client (un così detto eWallet), il costo e la complessità per i fornitori di servizi soprattuto se paragonati alla semplicità ed economicità di soluzioni alternative adeguate ed equivalenti basate su SSL.

Speculative execution: *Speculative execution*
comporta l'esecuzione preventiva di codice i cui risultati potrebbero non servire. Si tratta di una ottimizzazione vantaggiosa solamente nel caso in cui l'esecuzione anticipata del codice comporti un consumo o un tempo di elaborazione inferiori ed il tempo complessivamente risparmiato sul lungo periodo è superiore a quello consumato per esecuzioni non necessarie.

I moderni processori a pipeline usano l'esecuzione speculativa per ridurre il costo delle istruzioni di salto condizionato. Quando una di queste istruzioni viene incontrata la logica di controllo del processore cerca di indovinare quale sarà il più probabile esito del confronto (*branch prediction*) e comincia immediatamente ad eseguire le istruzioni da questa posizione. Se la previsione dovesse poi rivelarsi errata tutti i dati prodotti in seguito al branch dovranno essere scartati. Tuttavia l'esecuzione effettuata non rappresenta comunque uno spreco in quanto, altrimenti, le unità impegnate sarebbero rimaste inutilizzate.

STL: *Standard Template Library*.
libreria software parte del C++ che rende disponibili strutture dati ed algoritmi generici.

La STL è un valido aiuto per i programmatori C++ rendendo disponibili le implementazioni di classi utilizzate comunemente, come ad esempio gli array associativi. Tali classi possono essere utilizzate sia con i tipi di dato predefiniti che con quelli definiti dall'utente e forniscono anche un nutrito insieme di utili operazioni come la copia e l'assegnamento.

La STL raggiunge questo risultato mediante l'uso dei template. Sebbene questo approccio sia molto potente il codice che ne risulta, piuttosto complesso, e sempre stato un problema per i compilatori che spesso falliscono nella compilazione.

La C++ Standard Library è definita dall'ISO/IEC 14882 ed è stata la prima libreria di algoritmi e strutture dati generiche ad esser definita seguendo quat-

tro principi fondamentali: programmazione generica, astrazione senza perdita di efficienza, modello computazionale alla Von Neumann e semantica ben definita.

Ringraziamenti

Desidero ringraziare il centro CEFRIEL nella persona del *Prof. Fornaciari* per la preziosa opportunità offertami di concludere gli studi con un lavoro applicativo interessate e stimolante.

Allo stesso modo ringrazio il *Prof. Brandolese* per la disponibilità dimostratami nel seguire e supervisionare costantemente il mio lavoro, per la preziosa collaborazione e gli indispensabili consigli che non mi ha mai negato.

Un grazie va anche ai compagni di laboratorio: *Roberto Farina*, *Luca Giuliani* e *Luca Pizzamiglio* per la sempre pronta disponibilità quando ho avuto bisogno di un loro consiglio, ed ancora *Giovanni Beltrame* e *Daniele Scarpazza* per le ripetute consulenze che mi hanno offerto anche quando si trovavano lontani.

Infine ringrazio tutte le persone che mi sono sempre state vicine regalandomi un sorriso, *i miei genitori* e *gli amici tutti* in particolare.

Patrick Bellasi
Luglio 2005

Software & Tecnologie

"Unix is user-friendly. It's just very selective about who its friends are"
anonymous

NELLO svolgimento di questo lavoro di Tesi sono sempre stato supportato dalla disponibilità di validi strumenti software. Da molto tempo sostengo e promuovo l'utilizzo di soluzioni OpenSource, da me ritenuti i migliori strumenti disponibili gratuitamente ed in grado di assolvere pressoché ad ogni esigenza. Questo lavoro di Tesi ne è una dimostrazione, ritengo quindi doveroso ricordare brevemente i più significativi al fine di contribuite ancora una volta a diffonderne la conoscenza, ma soprattutto come modo personale per dire ancora una volta un sentito *"Grazie"* a tutti coloro che come me credono nelle potenzialità di un approccio aperto all'informatica e lavorano instancabilmente per regalarci strumenti sempre migliori.

Ambiente operativo:

Sistema Operativo	**Gentoo GNU/Linux**
Desktop	**KDE**
Shell	**Bash**

Ambiente di sviluppo:

Compilatore	**GNU/gcc**
Disassemblatore	**GNU/gdb**
Documentazione	**Doxygen**

Editor	**GNU/Emacs**
IDE	**Kdevelop**
Parser	**Bison**
Scanner	**Flex**
UML	**Umbrello**
Utility	**GNU/autotools**
Version control	**CVS**

Elaborazione dati:

Calcolo matematico	**octave**
Plotting	**Xmgrace**

Publishing:

Grafica raster	**GIMP**
Grafica vettoriale	**Xfig**
Typesetting	**TEX** e **LATEX 2ε**
Editing	**kile**
Bibliografia	**Tellico**

Presentazione:

Office	**OpenOffice**
Slide	**LATEX 2ε** e **prosper**
PDF viewer	**kpdf**

Figure, Tabelle e Listati[1]

[1] Per ciascun oggetto vengono usati stili diversi nella descrizione: le figure sono scritte normalmente, le tabelle sono in *slanted* e i listati in *italic*.

Bibliografia

[1] A. Zaccagnini A. Languasco. *Introduzione Alla Crittografia*. Ulrico Hoepli Editore, 2004.

[2] D. Wagner C. Hall J. Kelsey, B. Schneier. Side channel cryptanalysis of product ciphers. *Lecture Notes in Computer Science*, 1485:97–110, 1998.

[3] J. A. Muir. Techniques of side channel cryptanalysis. Master's thesis, 2001.

[4] P. Kocher. Cryptanalysis of diffie-hellman, rsa, dss, and other systems using timing attacks. In *Lecture Notes in Computer Science*, 1996.

[5] J. Markoff. Secure digital transactions just got a littel less secure. *New York Times*, 11 1995.

[6] R. J. Lipton D. Boneh, R. A. DeMillo. On the importance of checking cryptographic protocols for faults. *Lecture Notes in Computer*, 1233:37–51, 1997.

[7] A. Shamir E. Biham. Differential fault analysis of secret key cryptosystems. *Lecture Notes in Computer Science*, 1294:513–??, 1997.

[8] M. Kuhn R. Anderson. Low cost attacks on tamper resistant devices. In *IWSP: International Workshop On Security Protocols, LNCS*, 1997.

[9] Y. Han A. B. Jeng A. D. Narasimhalu T. H. Ngair F. Bao, R. H. Deng. Breaking public key cryptosystems on tamper resistant devices in the presence of transient faults. In *Security Protocols Workshop*, pages 115–124, 1997.

[10] FIPS46-3. *Data Encryption Standard (DES)*. National Institute of Standards and Technology, 10 1999.

[11] J. R. Rao P. Rohatgi D. Agrawal, B. Archambeault. Em side-channel(s), the. In *CHES '02: Revised Papers From the 4th International Workshop On Cryptographic Hardware and Embedded Systems*, pages 29–45. Springer-Verlag, 2002.

[12] F. Olivier K. Gandolfi, C. Mourtel. Electromagnetic analysis: Concrete results. In *CHES '01: Proceedings of the Third International Workshop On Cryptographic Hardware and Embedded Systems*, pages 251–261. Springer-Verlag, 2001.

[13] D. Samyde J. J. Quisquater. Electromagnetic analysis (ema): Measures and counter-measures for smart cards. In *E-SMART '01: Proceedings of the International Conference On Research in Smart Cards*, pages 200–210. Springer-Verlag, 2001.

[14] B. Jun P. Kocher, J. Jaffe. Differential power analysis. *Lecture Notes in Computer Science*, 1666:388–397, 1999.

[15] P. Dischamp D. Moyart M. L. Akkar, R. Bevan. Power analysis, what is now possible... *Lecture Notes in Computer Science*, 1965, 2000.

[16] R. H. Sloan T. S. Messerges, E. A. Dabbish. Examining smart card security under the threat of power analysis attacks. In *IEEE Transactions On Computers*, volume 51, pages 541–55, 2002 5.

[17] R. Mayer-Sommer. Smartly analyzing the simplicity and the power of simple power analysis on smartcards. *Lecture Notes in Computer Science*, 1965:78–??, 2001.

[18] J. R. Rao P. Rohatgi S. Chari, C. Jutla. A cautionary note regarding evaluation of aes candidates on smart-cards. In *Second Advanced Encryption Standard (AES) Candidate Conference*, Rome, Italy, 1999.

[19] T. Messerges. Securing aes finalists against power analysis attacks. *Lecture Notes in Computer Science*, 1978:150–164, 2001.

[20] A. Shamir E. Biham. Power analysis of the key scheduling of the aes candidates. In *Second AES Conference*, 1999.

[21] T. Lash. A study of power analysis and the advances encryption standard. Master's thesis, 2002.

[22] FIPS186-2. *Digital Signature Standard (DSS)*. National Institute of Standards and Technology, 1 2000.

[23] B. Jun P. Kocher, J. Jaffe. *Introduction to Differential Power Analysis*. CryptographyResearch, 1998.

[24] T. Mudge. Power: A first-class architectural design constraint. *IEEE Computer*, 34(4), 4 2001.

[25] A. Wolfe V. Tiwari, S. Malik. Power analysis of embedded software: a first step towards software power minimization. *IEEE Transactions On Very Large Scale Integration (VLSI) Systems*, 2(4):437–445, 1994.

[26] A. Wolfe M. Lee V. Tiwari, S. Malik. Instruction-level power analysis and optimization of software. *VLSI Signal Processing*, (13):223–238, 1996.

[27] M. Lee V. Tiwari. Power analysis of a 32-bit embedded microcontroller. *VLSI Design Journal*, 1998.

[28] G. Ascia D. Sarta, D. Trifone. A data dependent approach to instruction level power estimation. In *IEEE Alessandro Volta Memorial Workshop On Low-Power Design*, 3 1999.

[29] M. Martonosi D. Brooks, V. Tiwari. Wattch: a framework for architectural-level power analysis and optimizations. In *ISCA*, pages 83–94, 2000.

[30] T. M. Austin D. C. Burger. Simplescalar tool set, version 2.0, the. Technical Report CS-TR-1997-1342, 1997.

[31] F. Salice D. Sciuto C. Brandolese, W. Fornaciari. An instruction-level functionally-based energy estimation model for 32-bits microprocessors. In *Design Automation Conference*, pages 346–351, 2000.

[32] W. Fornaciari F. Salice D. Sciuto V. Trianni G. Beltrame, C. Brandolese. An assembly-level execution-time model for pipelined architectures. In *ICCAD*, pages 195–200, 2001.

[33] Arm7tdmi technical reference manual (rev. 4). Technical report, 2001.

[34] Arm architecture reference manual. Technical report, 2000.

[35] A. Aaby. *Compiler Construction Using Flex and Bison*. Aabyan@wwc.Edu, 1998.

[36] J. L. Hennessy D. A. Patterson. *Computer Architecture: a Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1990. ISBN: 1-55880-069-8.

Indice analitico

L'utopia è come l'orizzonte:
cammino due passi, e si allontana di due passi.
Cammino dieci passi, e si allontana di dieci passi.

L'orizzonte è irraggiungibile.
E allora, a cosa serve l'utopia?
A questo: serve per continuare a camminare.

Eduardo Galeano

POLITECNICO DI MILANO

Dipartimento di Elettronica e Informazione
Piazza Leonardo da Vinci, 32 – 20133 MILANO