Processori ARM

Riccardo Mancini

***Abstract*— Questa tesina è un approfondimento dell'ormai diffusa architettura ARM, architettura dei processori che oggigiorno troviamo nella maggior parte dei sistemi embedded. Nella prima parte verrà descritta ed illustrata la composizione e l'organizzazione di questi processori. Poi si vedrà come sono strutturare le istruzioni e come queste vengono manipolate efficientemente attraverso la tecnica di pipelining. Dopodiché verrà delineato l'importanza del coprocessore e come questo può alleggerire il carico al processore principale. Infine verranno spese alcune parole su Jazelle, un miglioramento aggiuntivo introdotto dopo la continua diffusione di questa architettura.**

I. INTRODUZIONE

L'architettura ARM (precedentemente Advanced RISC Machine, prima ancora Acorn RISC Machine), in elettronica e informatica, indica una famiglia di microprocessori RISC a 32 bit e 64 bit sviluppata da ARM Holdings e utilizzata in molti sistemi embedded.
Grazie alle sue caratteristiche di basso consumo elettrico, rapportato alle prestazioni, l'architettura ARM domina il settore dei dispositivi mobili dove il risparmio energetico delle batterie è fondamentale.

II. ARCHITETTURA E ORGANIZZAZIONE

A. *RISC*

I SoC ARM sono di tipo RISC, cioè presentano un numero consistente di registri e poggiano su un set di istruzioni limitato che permettono la realizzazione di un'architettura semplice e lineare.
Le architetture RISC vengono inoltre definite *load-store* perché permettono di accedere alla memoria unicamente tramite delle istruzioni specifiche. Esse provvedono a leggere e scrivere i dati nei registri del microprocessore, mentre tutte le altre istruzioni manipolano i dati contenuti all'interno del microprocessore.

Nei processori CISC, che prevedono un set esteso di istruzioni con metodi di indirizzamento complessi, tutte le istruzioni possono accedere ai registri o alla memoria in modo indifferente.

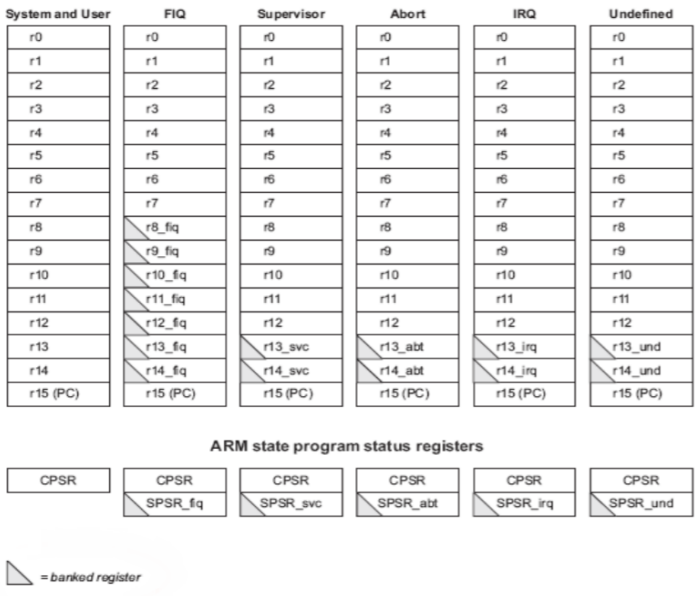
Con il paradigma RISC si usano insiemi ottimizzati di istruzioni che permettono di eseguire le operazioni con un ridotto consumo energetico rispetto all'approccio CISC ma, ovviamente, con una minore dinamicità. Le istruzioni, infatti, riducono le operazioni da espletare in attività più semplici possibile cosicché la maggior parte delle stesse vengano eseguite in egual tempo.
Esse hanno poi la stessa lunghezza in bit e usano metodi di indirizzamento non complessi aumentando la velocità di esecuzione.

B. *Registri*

Prima di illustrare i registri utilizzati nei processori ARM è necessario spendere due parole sulle modalità operative che vengono adottate. Ne troviamo ben 7, che sono:

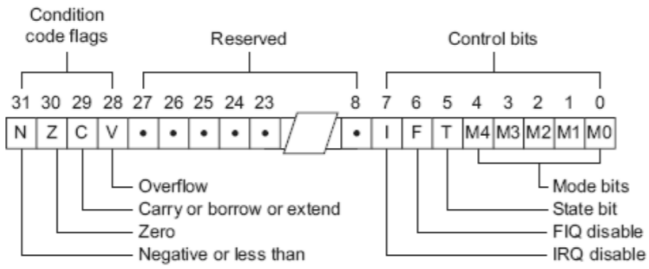
- *User*, modalità non privilegiata sotto cui girano la maggior parte dei task;
- *System*, modalità privilegiata che usa gli stessi registri dell'user mode (introdotta dalla versione v4);
- *FIQ (fast interrupt)*, modalità in cui ci si entra quando viene sollevato un interrupt ad alta priorità;
- *IRQ (interrupt)*, modalità in cui ci si entra quando viene sollevato un interrupt a bassa priorità;
- *Abort*, usato per gestire violazioni all'accesso di memoria;
- *Undefined*, usato per gestire istruzioni indefinite.
- *Supervisor*, modalità in cui si entra al reset e quando un'istruzione di software interrupt (SWI) viene eseguita.

Ogni modo operativo ha accesso a un differente sottoinsieme di registri.
Escluso lo User, i restanti modi sono privilegiati, ed alcune istruzioni possono essere eseguite solo da quest'ultime.



La CPU dispone complessivamente di 37 registri organizzati come segue:

- 16 registri da 32 bit denominati R0-R15
  - R0-R12 sono registri di uso generale;
  - R13 viene usualmente usato come Stack Pointer (SP), ma l'architettura non forza il suo impiego in tal senso;
  - R14 ha la funzione (architetturale) di subroutine Link Register (LR). In esso la logica salva l'indirizzo di ritorno (ovvero il contenuto del registro R15) quando viene eseguita l'istruzione BL (Branch and Link);
  - R15 ha la funzione architetturale di Program Counter (*PC*);
- Un registro di stato CPSR (Current Program Status Register) che contiene le informazioni riguardanti le più recenti operazioni eseguite dalla ALU, permette di attivare/disattivare le interruzioni e permette di settare la modalità operativa del processore.
  Questo registro è costituito da:
  - 4 bit esprimono altrettante condizioni (Negative, Zero, Carry e oVerflow);
  - il bit T distingue tra due (super)modalità di funzionamento: quella usuale, denominata ARM di cui si sta parlando e la modalità Thumb (di cui si parla al paragrafo *A* di *"Formato istruzioni"*).
  - i bit I and F abilitano le interruzioni normali (I) and veloci (F)
  - i bit M4-M0 identificano il modo di funzionamento.



- 20 registri "duplicati" (*banked*) (segnati con un triangolo grigio nella figura) sono privati dello specifico modo.
  SPSR, R13 e R14 sono registri duplicati per tutti i modi *exception (*intese sia interruzioni che situazioni di errore).
  I registri SPSR (Saved Program Status Register) sono usati per salvare copia del CPSR prima di passare al corrispondente modo *exception* (ovvero la parola di stato che si aveva prima di passare al servizio della corrispondente eccezione).
  I registri R13 e R14 duplicati consentono di passare al servizio di una eccezione senza doversi preoccupare di salvare i registri R13 (SP) e R14 (LR) del contesto interrotto in quanto il contesto dell'eccezione servita ha i propri R13 e R14.

C. Gestione delle eccezioni

Al verificarsi di un'eccezione il *workflow* che viene eseguito è il seguente: prima di tutto viene effettuata una copia del CPSR nello SPSR_m, cioè nell'SPSR del modo corrispondente al tipo di eccezione; poi vengono modificati i bit di CPSR, in modo da riflettere il nuovo modo, disabilitando poi le interruzioni; contemporaneamente viene memorizzato l'indirizzo di ritorno in LR_m, ovvero il contenuto del PC (R15) nel registro R14, così da copiare in PC l'indirizzo del vettore di interruzione.
Il vettore di interruzione conterrà un'istruzione di salto in modo da passare al corrispondente exception handler.
Per tornare al programma interrotto, l'exception handler ripristina CPSR da SPSR_m e ripristina il PC da LR_m.

(<u>Esempio</u>
L'eccezione di Reset ha l'effetto di: (1) passare al modo Supervisor; (2) disabilitare le interruzioni normali e le veloci (portando ad uno i due bit I e F di CPSR); (3) azzerare il bit T di CPSR (equivale a riportare la macchina in stato ARM, affermando che non è più in stato Thumb, qualora ci fosse stata); (4) forzare il PC a eseguire l'istruzione in 0x00. Dopo il reset, il contenuto di tutti i registri eccetto PC e CPSR è indeterminato.)

Nella seguente tabella vengono descritti le differenti eccezioni.

Reset	Il pin di reset va alto
Istruzione indefinita	Il processore non riconosce l'istruzione in esecuzione
Software interrupt (SWI)	Istruzione di SWI chiamata dal programma
Interruzione del prefetch	Il processore ha cercato di eseguire un'istruzione non fetchata perché l'indirizzo è illegale
Interruzione dati	Avviene quando un'istruzione di trasferimento dati cerca di caricare o immagazzinare un dato ad un indirizzo illegale
IRQ	Pin di IRQ esterno che va alto
FIQ	Pin di FIQ esterno che va alto

Exception	Mode	Address
Reset	Supervisor	0x00000000
Undefined instruction	Undefined	0x00000004
Software interrupt (SWI)	Supervisor	0x00000008
Prefetch Abort (instruction fetch memory abort)	Abort	0x0000000C
Data Abort (data access memory abort)	Abort	0x00000010
IRQ (interrupt)	IRQ	0x00000018
FIQ (fast interrupt)	FIQ	0x0000001C

Inoltre queste eccezioni hanno una priorità, grazie alla quale è possibile scegliere quale eccezione gestire prima nel caso arrivassero insieme.

Priority	Exception
Highest 1	Reset
2	Data Abort
3	FIQ
4	IRQ
5	Prefetch Abort
Lowest 6	Undefined instruction SWI

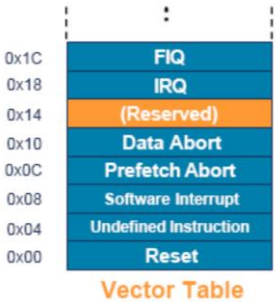
D. Interruzioni

La maggior parte dei processori ARM supporta sia IRQ che FIQ come input interrupts. FIQ viene definito come "fast" interrupt, mentre IRQ come "normal" interrupt. Entrambe le modalità di esecuzione hanno dei livelli di priorità, e la modalità FIQ ha un livello più alto rispetto a IRQ, per tale motivo se stiamo gestendo una interruzione con FIQ, essa non potrà essere sospesa da un'altra interruzione arrivata sul canale IRQ.

Ma non è la sola differenza, ne esistono altre e sono molto più sostanziali, rispetto alla precedente, sia a livello di architettura che a quello di gestione.

La prima sostanziale differenza è che la modalità FIQ ha dei registri in più rispetto alla modalità IRQ (come si nota dall'immagine nel paragrafo "*Registri"*), grazie ai quali si riduce l'overhead necessario alla copia dei valori di backup del processo precedente all'arrivo dell'interruzione. Per tale motivo, meno valori vengono spostati sullo stack e meno valori dovranno essere ripristinati nei registri alla fine della gestione dell'interruzione.

Il secondo motivo per cui FIQ è più veloce rispetto a IRQ è dovuto alla posizione della gestione degli interrupts all'interno del vettore delle interruzioni, il codice di gestione degli interrups FIQ è situato alla fine di tale vettore (0x1C)



Questo fa in modo che le istruzioni possono essere eseguite direttamente da tale locazione (dato che lo stack cresce per l'alto), senza effettuare ulteriori jump ad indirizzi di memoria (cosa non vera per le interruzioni IRQ).

Ad oggi, non molti S.O. supportano tale gestione delle interrupts con FIQ perchè può essere effettuato solo scrivendo in codice assembly.

Andiamo ad analizzare invece cosa succede nel caso di un interrupt esterno di tipo IRQ, la CPU:
1) copia CPSR nello SPSR_irq;
2) modifica appropriatamente i bit di CPSR in modo da indicare il modo "irq" e da disabilitare il sistema di interruzione (ponendo a 1 il bit I);
3) memorizza l'indirizzo di ritorno in LR_irq (R14_irq);
4) copia in PC l'indirizzo del vettore di interruzione, dove deve trovarsi il salto al gestore delle interruzioni.

Le interruzioni possono essere gestite in modo *"una alla volta"* oppure in modo *nidificato*.
Nel primo caso, il gestore delle interruzioni non riabilita il sistema di interruzione fino al termine, in modo che ulteriori, eventuali richieste di interruzione abbiano effetto. I passi della routine di servizio dell'interruzione sono:

1. Vengono salvati i registri che verranno toccati dalla routine;
2. Viene identificata la specifica interruzione e viene effettuato il salto al ramo corrispondente di codice;
3. Conclusa l'elaborazione, vengono ripristinati i registri del programma interrotto e il CPSR;
4. A questo punto, poiché il LR contiene l'indirizzo di ritorno, basterebbe scriverlo nel PC.

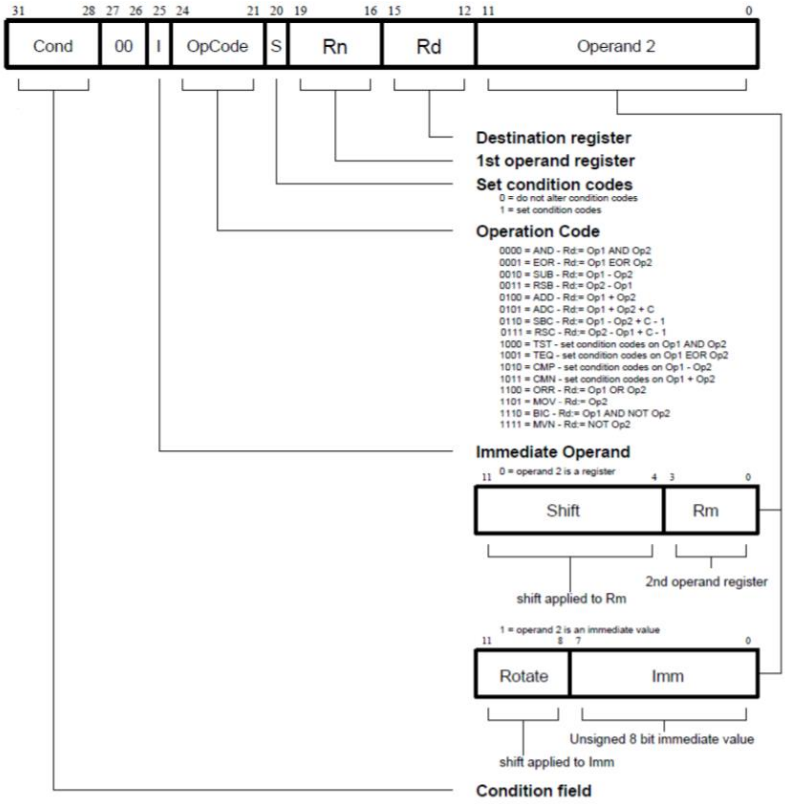
(*Nota: Le operazioni 3 e 4 sono indivisibili perché nel momento in si ripristina il CPSR, viene a sua volta riabilitato il sistema di interruzione, e se a questo punto ci fosse una richiesta di interruzione non servita, questa si manifesterebbe immediatamente, eliminando quindi l'indirizzo di ritorno che si trova in R14_irq).*

Le interruzioni nidificate comportano ulteriori difficoltà. Oltre ai registri manipolati dalla routine, dovrà essere salvato anche SPSR_irq, in quanto se il gestore delle interruzioni viene richiamato da un'altra interruzione, l'SPSR_irq deve essere al sicuro in quanto il registro in questione viene riscritto dalla nuova interruzione. Successivamente:

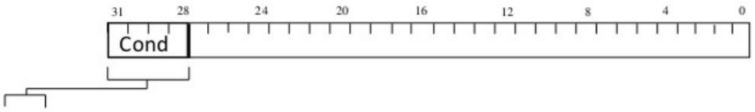
- viene azzerata la richiesta di interruzione (IRQ_i) che si sta servendo; (comporta ovviamente la lettura/scrittura sul sottosistema di i/o)
- viene riabilitato il sistema di interruzione;
- viene quindi eseguita l'elaborazione richiesta dall'interruzione (IRQ_i) che si sta servendo;
- a conclusione della elaborazione richiesta dalla specifica interruzione viene disabilitato il sistema di interruzione, riportandosi a una condizione analoga al caso dell'interruzioni gestite "una alla volta";
- la gestione dell'interruzione si chiude con il ripristino dei registri e del CPSR, in modo indivisibile (come prima).

III. FORMATO ISTRUZIONI

Il repertorio ARM prevede istruzioni su 32 bit, che manipolano dati su 32 bit.
La manipolazione dei dati avviene solo nei registri, la macchina è tipicamente *Load* (valori della memoria copiati nei registri) e *Store* (valori dei registri copiati in memoria).



Una caratteristica che contraddistingue il repertorio ARM è l'esecuzione condizionale. Come si vede i primi 4 bit di ogni istruzione contengono una condizione. L'istruzione viene eseguita solo se la condizione coincide con quella data dai 4 bit di condizione della parola di stato (registro di stato CPSR), indicati come Condition Code Flags. Un valore (dei 16 possibili, precisamente "1110") indica invece che l'istruzione viene eseguita in modo non condizionato.



0000 = EQ - Z set (equal)
0001 = NE - Z clear (not equal)
0010 = HS / CS - C set (unsigned higher or same)
0011 = LO / CC - C clear (unsigned lower)
0100 = MI -N set (negative)
0101 = PL - N clear (positive or zero)
0110 = VS - V set (overflow)
0111 = VC - V clear (no overflow)
1000 = HI - C set and Z clear (unsigned higher)
1001 = LS - C clear or Z (set unsigned lower or same)
1010 = GE - N set and V set, or N clear and V clear (>or =) Greater or equal
1011 = LT - N set and V clear, or N clear and V s (<) lower
1100 = GT - Z clear, and either N set and V set, or N clear and V set (>) Greater
1101 = LE - Z set, or N set and V clear,or N clear and V set (<, or =) Lower or equal
1110 = AL - always
1111 = NV - reserved.

A. *Thumb*

Gli ultimi processori ARM sono dotati di un set di istruzioni a 16 bit chiamato Thumb.
Il codice Thumb è sostanzialmente più leggero, ma è dotato di meno funzionalità. Per esempio solo i salti possono essere condizionati e alcuni opcode non possono essere utilizzati da tutte le istruzioni.

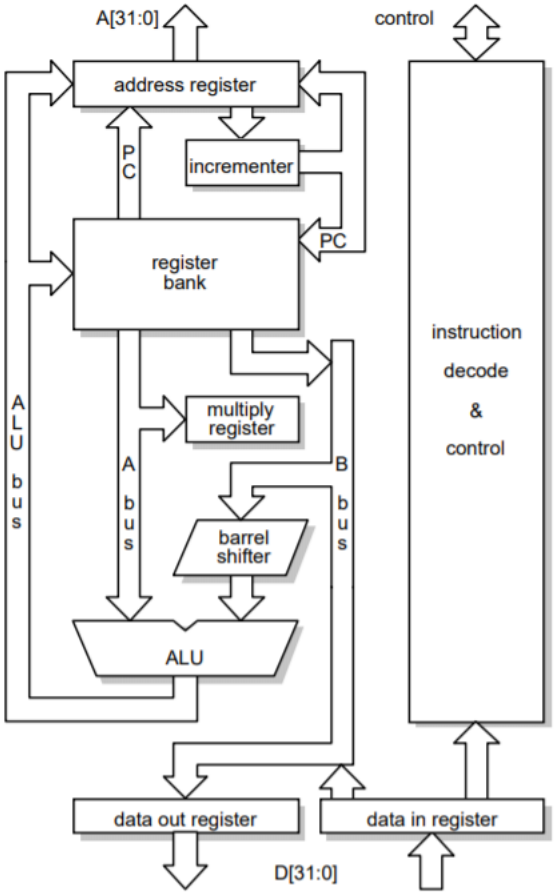
Nonostante queste limitazioni, Thumb fornisce prestazioni migliori del set di istruzioni completo nel caso di sistemi dotati di limitata larghezza di banda. Molti sistemi embedded sono dotati di un bus verso la memoria limitato e, sebbene il processore possa indirizzare a 32 bit, spesso si utilizzano indirizzamenti a 16 bit o simili.
Inoltre vengono utilizzati solo i registri R0-R7, ma i registri sorgente e destinazione sono identici.

In queste situazioni conviene creare codice Thumb per la maggior parte del programma e ottimizzare le parti di codice che richiedono molta potenza di calcolo utilizzando il set di istruzioni completo.
Il primo processore dotato di Thumb è stato l'ARM7TDMI.

IV. PIPELINE

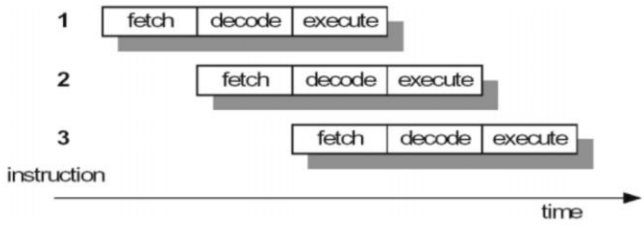
Nei processori RISC ogni istruzione è organizzata in modo che le azioni vengano svolte seguendo sempre una medesima successione, eventualmente lasciando un "buco" (NOP), se quel particolare passo non serve.
Questo fa si che si possa evitare di aspettare che un'istruzione sia completamente eseguita, prima di inizia la successiva. Mentre un'istruzione è in corso di esecuzione se ne potrebbe iniziare un'altra. Questo approccio si dice organizzazione a pipeline (o parallelismo a livello di istruzioni).

In particolare facciamo riferimento alla pipeline del modello più usato nel mercato, il modello ARM7TDMI.
La pipeline in questione è a 3 stadi (Fetch-Decode-Execute). In poche parole, durante la fase di Fetch, l'istruzione viene recuperata dalla memoria e messa nella pipeline (nell'IR); poi attraverso la fase di Decode l'istruzione appena presa viene decodificata; infine durante l'Execute l'istruzione contenuta nell'IR viene manipolata direttamente dall'ALU, oppure, nel caso invece l'istruzione non prevedesse la manipolazione, ma solo la scrittura di una cella o la sua lettura, l'ALU non viene interpellata.
La seguente figura ne mostra l'architettura.



Un'osservazione da fare è che sulla porta a destra della ALU c'è il *barrel shifter**, che rende possibile la manipolazione del secondo operando dell'operazione facendogli eseguire l'operazione stessa senza passare attraverso un'istruzione esplicita, che potrebbe modificare il contenuto del registro.

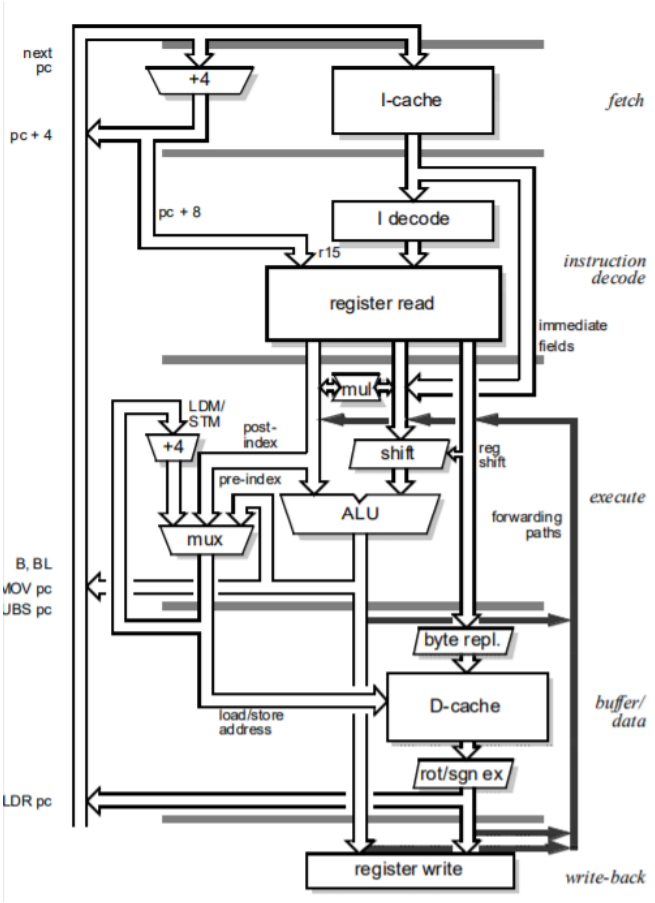
Il funzionamento della pipeline è illustrato nella seguente figura**,** dalla quale è possibile notare che in ogni momento 3 diverse istruzioni possono occupare ognuno dei tre stadi. Questo è possibile grazie all'hardware di ogni stadio che deve poter operare in maniera indipendente (l'organizzazione a pipeline deve essere organizzata all'atto della realizzazione hardware del dispositivo).



Quando il processore esegue istruzioni di processamento dati, la latenza è pari a 3 cicli di clock e il troughtput è di 1 istruzione al ciclo.
Nota: Per le istruzioni che necessitano di un accesso alla memoria (load o store) occorrono 4 cicli essendo la memoria unica per dati e istruzioni non è possibile leggere l'istruzione successiva mentre si leggono i dati.

Un barrel shifter* è un circuito digitale che può shiftare una word di un numero specificato di bit in un solo ciclo di clock. Può essere implementato come sequenza di multiplexer, con l'output di un mux connesso come ingresso del successivo in base alla distanza dello shift. Solitamente viene realizzato come cascade di mux 2x1 in parallelo. Ci sono 2 opzioni da scegliere: il tipo di shift (logico a destra, logico a sinistra, aritmetico a destra, rotazione a destra) e la distanza dello shift (di quanti bit deve essere traslato il dato).

Come si vede si tratta di una pipeline estremamente semplice.
Nelle versioni successive, ad esempio nella ARM9TDMI, la pipeline è cresciuta diventando a 5 stadi (Fetch-Decode-Execute-Memory-Write back). Questo è diventato possibile anche al cambio di architettura, passando dalla Von Neumann, all'architettura Harvard in cui vi è separazione tra la memoria contenente i dati e quella contenente le istruzioni.



Rispetto alla pipeline a 3 stadi si può notare che si è spostato lo step di lettura del registro dall'Execute al Decode, dove in quest'ultimo sono state messe 3 porte di lettura per gli operandi, in modo che la maggior parte delle istruzioni possano leggere gli operandi in un ciclo.
Inoltre si è diviso l'Execute in: ALU, Memory access e Write back. Infatti, se richiesto viene effettuato l'accesso alla memoria dati; in caso contrario, il risultato della ALU è semplicemente bufferizzato per un ciclo. Durante il write back, il risultato generato dall'istruzione è scritto nel registro, incluso qualsiasi dato caricato dalla memoria.
Il risultato è una pipeline meglio bilanciata, con latenze minimizzate tra stage, che possono girare a clock maggiori.

V. COPROCESSORI

L'architettura ARM prevede l'impiego di coprocessori con l'obiettivo di permettere l'estensione della capacità elaborativa delle CPU ARM rispetto a specifiche aree applicative.

Per definizione, un coprocessore è un dispositivo che non preleva le istruzioni dalla memoria, ma che osserva il flusso delle istruzioni prelevate dalla CPU e interviene quando l'istruzione appartiene al proprio repertorio. A livello generale il funzionamento è:
a) il repertorio di istruzioni di COPR è disgiunto da quello di CPU;
b) la sola CPU è responsabile del prelievo di istruzioni (fetch) e della loro sequenzializzazione;
c) dalle informazioni di stato disponibili sul bus il coprocessore riconosce i cicli fetch ed è in grado di catturare all'occorrenza l'istruzione che sta passando sul bus dati;
d) il coprocessore che riconosce che sta transitando sul bus un'istruzione del suo repertorio la preleva e la esegue;
e) in presenza di un'istruzione del repertorio del coprocessore, la CPU attende che il coprocessore abbia finito. Successivamente, procede con il fetch della prossima istruzione del programma.

Questo appena descritto è un comportamento ideale, in realtà bisogna tenere in considerazione che le istruzioni possono avere differente durata, dunque è necessario introdurre un meccanismo di sincronizzazione tra CPU e COPR.
Inoltre c'è anche da dire che la CPU può funzionare in pipeline, perciò la COPR, per poter eseguire istruzioni in modo sincronizzato, deve riprodurre al suo interno l'avanzamento della pipeline, riproducendo la coda di istruzioni presente nella CPU o ricevendo da quest'ultima l'informazione di stato della coda.
Per le istruzioni che operano solo i registri interni del coprocessore, questo dispone di tutti i dati e non ha che da effettuare l'operazione.
Ma per le operazioni che fanno accesso alla memoria è ancora la CPU che effettua l'indirizzamento del dato, ovvero comanda la memoria alla lettura/scrittura della parte del dato che sta all'indirizzo contenuto nell'istruzione. Se il dato fosse più grande dell'entità indirizzata dalla CPU risulta necessario effettuare ulteriori eventuali cicli di lettura o scrittura in modo da leggere o scrivere la parte di dato che segue quella indirizzata dalla CPU. Tale compito è affidato al coprocessore, che a tale scopo entra in controllo del bus ed effettua i restanti cicli di lettura scrittura richiesti. A questo punto verrebbe richiesto un meccanismo di arbitraggio per l'accesso al bus.

E' importante precisare che un coprocessore dovrebbe poter essere impiegato in modo trasparente per il programmatore, nel senso che il programmatore assembler deve poter scrivere lo stesso codice, indipendentemente dal fatto che esso sia eseguito su macchina dotata o non dotata di coprocessore.
Se il coprocessore non è presente, si genera un'eccezione per "codice di operazione ignoto", alla quale può essere associato un gestore che, in base alle informazioni di stato è

in grado di individuare la specifica istruzione che ha determinato l'eccezione, e chiamare una routine che ne simula il comportamento.

VI. MIGLIORAMENTI AGGIUNTIVI (JAZELLE)

Col passare del tempo e complice il dilagare di dispositivi portatili che sono diventati sempre più importanti nella vita di tutti i giorni, si è affacciata la necessità di eseguire applicazioni Java.

Impegnata in prima linea coi dispositivi mobile (se sono stati venduti più di 10 miliardi di microprocessori di questa famiglia è anche grazie all'esplosione di questo settore), ARM s'è trovata, quindi, di fronte al problema di come realizzare delle JVM (Java Virtual Machine) che girassero in maniera più efficiente sulle sue CPU, rendendo questo possibile attraverso Jazelle.

Avendo già sperimentato con successo la possibilità di eseguire opcode di ISA diverse da quella nativa (ad esempio con Thumb), ha pensato bene di introdurre un'altra modalità di esecuzione che prendesse a modello l'ISA "virtuale" delle JVM, che eseguono codice Java sotto forma di bytecode.

L'ISA eseguita dalla JVM è, però, estremamente diversa dall'architettura ARM, da Thumb, e in generale da qualunque altra ISA general purpose già sviluppata. Infatti la strada percorsa è stata ben diversa, la casa madre ha deciso di realizzare un "accelerazione" che mettesse a disposizione un ambiente di esecuzione che faciliti l'esecuzione dei bytecode Java.

In questa modalità il funzionamento del processore cambia radicalmente sia nell'esecuzione delle istruzioni che nell'uso dei registri, i quali subiscono una specializzazione:

- Da R0 a R3 vengono utilizzati per conservare i 4 elementi in cima allo stack Java (le JVM sono delle virtual machine di tipo "stack");
- R4 memorizza il puntatore a all'oggetto corrente su cui sta lavorando;
- R5 punta all'handler per la gestione delle istruzioni non eseguite direttamente / nativamente
- R6 rappresenta lo stack utilizzato dalla JVM, utilizzato per memorizzare risultati intermedi e valori di ritorno dalle chiamate a funzione;
- R7 contiene l'indirizzo base delle variabili globali;
- R8 punta alle costanti utilizzate dal codice in esecuzione;
- da R9 a R11 sono a disposizione della virtual machine;
- R12 è liberamente utilizzabile;
- R13, come per la modalità Thumb, conserva il puntatore allo stack (non Java: è lo stack proprio dell'applicazione in esecuzione; in questo caso la JVM)
- R14, come R12, ma viene usato anche per contenere l'indirizzo di ritorno da una chiamata a funzione (similmente alle modalità ARM e Thumb: è il registro di "Link")
- R15, infine, punta al bytecode da eseguire (è il PC).

In linea generale il *workflow* di Jazelle che porta all'esecuzione di bytecode è il seguente;
R15, come abbiamo visto, rappresenta il Program Counter della JVM, per cui viene utilizzato per leggere il bytecode da eseguire. In realtà l'ARM preleva una word, cioè 4 byte alla volta, e di questi provvede a scartare quelli inutili (quelli che, eventualmente, precedono il byte che interessa eseguire) e mantenere internamente tutti gli altri (quindi, oltre al byte da eseguire, anche gli eventuali successivi).
(Nota: Operando in questo modo esegue una sorta di caching che riduce fino a 4 volte il numero di accessi alla memoria per il fetch delle istruzioni e dei loro dati. Questo perché l'ISA delle JVM prevede degli opcode a lunghezza variabile, dove il primo byte rappresenta il "codice identificativo" del bytecode da eseguire, a cui possono seguire altri byte utili all'istruzione).

Una volta prelevato il bytecode e gli eventuali dati che gli servono, Jazelle ha due possibilità: eseguirlo direttamente, oppure richiamare l'handler puntato da R5 a cui demandarne l'interpretazione e la successiva esecuzione.
(Jazelle, infatti, non è in grado di eseguire in hardware tutte le quasi 240 istruzioni dell'ISA della JVM, ma soltanto 140 circa di esse vengono lette ed eseguite immediatamente. Le quasi 100 rimanenti richiedono la solita emulazione software prevista da una normale JVM (non dotata di JIT)).
Lasciare quasi 100 istruzioni su 240 non emulate potrebbe far decadere considerevolmente le prestazioni, ma in realtà si tratta di bytecode raramente utilizzati. ARM stima, infatti, che meno del 5% del tempo d'esecuzione venga impiegato per farle girare, mentre quasi tutto il tempo la JVM esegue le 140 che sono, pertanto, le più comuni e sulle quali ha giustamente concentrato i suoi sforzi.
Un approccio ibrido, dunque, ma che si rivela azzeccato perché richiede una quantità relativamente minore di transistor per l'implementazione, riducendo considerevolmente il consumo e mantenendo comunque delle prestazioni più elevate rispetto a un interprete interamente software.

Con questa implementazione ARM si è dimostrata particolarmente attenta alle esigenze del mercato, aggiungendo un altro motivo che contribuisce sicuramente alla diffusione di quest'architettura facendola preferire ad altre.

REFERENCES

[1] http://www.unife.it/ing/informazione/calcolatori-elettronici/dispense/6-Architettura%20ARM.pdf
[2] https://developer.arm.com/documentation/ddi0210/c/Coprocessor-Interface?lang=en
[3] http://www.ee.ncu.edu.tw/~jfli/soc/lecture/ARM9.pdf
[4] https://studylibit.com/doc/6284142/--ams-tesi-di-laurea
[5] https://www.vitadastudente.it/2013/01/14/arm-interruzioni-fiq-irq/
[6] https://stlab.dinfo.unifi.it/bucci/Teaching/Magistrale/Approfondimenti_ARM.pdf
[7] https://stlab.dinfo.unifi.it/bucci/Teaching/Magistrale/Arm.pdf
[8] https://pessina.mib.infn.it/Corsi_del_III_anno/I%20microcontrollori%20Parte%20B%20(5).pdf
[9] https://slideplayer.it/slide/533927/
[10] https://www.appuntidigitali.it/4887/jazelle-laccelerazione-java-secondo-arm/