

# Appunti Architettura

Brendon Mendicino

December 3, 2022

# Contents

<b>1</b>	<b>Introduzione</b>	<b>3</b>
<b>2</b>	<b>Pipeline</b>	<b>3</b>
<b>3</b>	<b>Pipeline Hazards</b>	<b>4</b>
<b>4</b>	<b>Floating Point</b>	<b>4</b>
<b>5</b>	<b>Exceptions</b>	<b>5</b>
<b>6</b>	<b>Cache Memories</b>	<b>6</b>
<b>7</b>	<b>Branch Prediction</b>	<b>9</b>
<b>8</b>	<b>Schedulazione Dinamica</b>	<b>13</b>
<b>9</b>	<b>Pipelining</b>	<b>14</b>
9.1	Static Scheduling . . . . .	14
9.2	Dynamic Scheduling . . . . .	14
<b>10</b>	<b>ARM</b>	<b>16</b>
10.1	Register . . . . .	17
10.2	Instruction Set . . . . .	17
10.3	Stack . . . . .	20
<b>11</b>	<b>Application Binary Interface</b>	<b>23</b>
11.1	Supervisor Calls (SVC) . . . . .	23
<b>12</b>	<b>ASM+C</b>	<b>26</b>
<b>13</b>	<b>Scheda</b>	<b>27</b>
13.1	Progetto Interrupt Controller . . . . .	28
13.2	System Timing . . . . .	28

# 1 Introduzione

## 2 Pipeline

Per misurare le prestazioni di una pipeline si usa il **throughput**. Il throughput è il numero di istruzioni che escono dalla pipeline in un intervallo di tempo.

Il datapath è composto da:

- instruction fetch: si prende dalla memoria la prossima istruzione putnatata dal PC e si incrementa quest'ultimo di 4;
- decode: si decodifica l'istruzione, attivando il datapath in modo adeguato, a prescindere dal tipo di operazione carico i due registri rs ed rt, ed il campo immediato, anche nel caso l'istruzione non fosse immediata, risparmio del tempo aumentando leggermente il consumo di potenza;
  - $A \leftarrow \text{Reg}[\text{rs}]$ ;
  - $B \leftarrow \text{Reg}[\text{rt}]$ ;
  - $\text{Imm} \leftarrow (IR_{16...31})$ ;
- execution/effective address cycle:
  - memory reference:  $\text{ALUoutput} \leftarrow A + \text{Imm}$ ;
  - register-register:  $\text{ALUoutput} \leftarrow A \text{ op } B$ ;
  - register-immediate:  $\text{ALUoutput} \leftarrow A \text{ op } \text{Imm}$ ;
  - brach:  $\text{ALUoutput} \leftarrow \text{NPC} + \text{Imm}$ ;  $\text{Cond} \leftarrow (A \text{ op } 0)$ ;
- memory access/branch completion cycle:
  - $\text{LMD} \leftarrow \text{Mem}[\text{ALUoutput}]$ ; or  $\text{Mem}[\text{ALUoutput}] \leftarrow B$ ;
  - branch: if (cond)  $\text{PC} \leftarrow \text{ALUoutput}$  else  $\text{PC} \leftarrow \text{NPC}$ ;
- write-back cycle:
  - register-register:  $\text{Reg}[IR_{16...20}] \leftarrow \text{ALUoutput}$ ;
  - ...

L'assunzione molto forte sarà che tutti i dati e le istruzioni saranno sempre nelle memoria cache, quindi si avrà un delay di un solo colpo di clock. Il registre file potrà sia essere letto che scritto, ci sarà dunque bisogno di soddisfare queste richieste in un solo colpo di clock, la scrittura avviene nella prima parte del colpo di clock mentre la lettura avviene nella seconda parte del colpo di clock.

Si aggiungono dei registri (detti pipeline register),

Aggiungere i registri della pipeline aggiunge un overhead, inoltre il clock del processore comporta un rallentamento, causato dallo skew.

### 3 Pipeline Hazards

Sono dei casi in cui l'istruzione non viene eseguita in modo corretto:

- structural hazards: dipende dalla memoria,
- data hazards: dipende da come i dati vengono scritti e letti;
- control hazards: dipende dai branch;

**Stall** Un modo di gestire gli hazards è di mandare la CPU in stallo.

Risolvere gli hazard strutturali comporta un costo, in termini di nuovo hardware e di migliorare quello esistente. Un processore con hazard strutturali avrà un clock più veloce ma problemi di accesso alla memoria, un processore senza hazard strutturali avrà un clock più lento ma nessuna limitazione di accesso alla memoria.

**Data hazards** Generati dalle dipendenze dei dati generati all'interno della pipeline. Esempio:

```
1 add r1, r2, r3
2 sub r4, r1, r5
3 and r6, r1, r7
4 or  r8, r1, r9
5 xor r10, r1, r11
```

Il registro r1 viene inizializzato nella prima istruzione e poi utilizzato nel resto del codice, ma l'operazione di scrittura in si trova alla fine, infatti l'istruzione successiva (sub) dovrebbe aspettare che r1 sia scritto prima che possa essere letto, se tuttavia proviamo a leggere r1 il risultato sarà non deterministico.

Per risolvere questo problema si hanno due soluzioni:

- mandare in stallo il processore;
- implementiamo un forwarding che permette di non attendere la scrittura del registro, ma di leggere direttamente il valore dei registri della pipeline;

### 4 Floating Point

Le operazioni floating point sono molto complesse, se si cerca di implementarle in un solo colpo di clock allora il processore diventa troppo complicato dal punto di vista logico, oppure un'altra soluzione potrebbe essere quello di rallentare il clock, per far entrare tutte le operazioni in un singolo colpo, ma entrambe le soluzioni non sono fattibili, allora si prende un approccio di suddivisione della pipeline in unità. Per supportare le operazioni di floating point in pipeline, si è optato per una separazione dalla execute in diverse unità:

- integer unit;
- fp/integer multiply;
- fp adder;
- fp/integer divider;

Questa ramificazione della pipeline va a convergere nella sezione di MEM.

Si dovrà definire la latenza, ovvero il numero di colpi di clock che una unità usa per avere un risultato, ed un intervallo di inizializzazione, ovvero il numero di colpi di clock che la seconda istruzione dovrà attendere per entrare nella sezione desiderata (come somma o divisione). Un esempio:

- add: lat: 1, int: 1;
- mult: lat: 8, int: 1;
- fadd: lat: 4, int: 1;
- div: lat: 24, int: 24;

Solitamente la divisione ha la latenza identica all'intervallo di inizializzazione. Solitamente su un unità è **pipelined** allora ha un colpo di clock come intervallo di inizializzazione, se l'unità non è pipelined allora il suo intervallo di inizializzazione è uguale alla latenza.

Un altro problema dato dagli hazard strutturali è il fatto che: più istruzioni non possono accedere in contemporanea alla fase di MEM o di WB, solitamente il criterio è FIFO, oppure si potrebbe dare maggiore priorità alle istruzioni con il maggiore numero di clock.

## 5 Exceptions

Le eccezioni sono classificate in:

- **sincrone** e **asincrone**;
- **user requested** (l'utente potrebbe creare un'eccezione) o **coerced** (data da fattori esterni);
- **maskable** o **non-maskable**: alcune eccezioni non sono mascherabili ovvero forzare l'hardware a non rispondere all'exception;
- **within instructions** o **between instructions**: within all'interno delle istruzioni (mette un'istruzione fa DI, ID, ...) oppure tra due istruzioni (ld, add);

Le macchine moderne sono chiamate **restartable machines** ovvero fanno ripartire il processore dallo stato in cui si trovava prima dell'eccezione. Quando una exception arriva il processore deve stoppare il IF, fermare la scrittura della pipeline e riuscire a tornare dalla procedura che ha chiamato l'exception.

### Example 5.1 – Interrupt protocol in 80x86

Quando la CPU rileva un interrupt, legge il tipo di interrupt dal bus, salva lo stato del processore

ARM: la CPU salva lo status, il PC ed il Processore Status Register nello stack, aggiorna i flag e salta al valore dell'exception

Le eccezioni possono essere gestite in modo **preciso** o in modo **impreciso**:

- preciso: quando avviene un'interruzione, tutte le istruzioni prima che arrivi l'istruzione devono essere completate, tutte quelle dopo devono essere rimandate, gestire questo tipo di istruzioni è molto oneroso;
- impreciso:

Nel MIPS le possibili exception sono:

- IF: page fault, accesso a memoria protetta;
- ID: opcode illegale;
- EX: exception aritmetiche;
- MEM: stesse della IF;
- WB: nessuna;

Se due exception arrivano nello stesso momento: si può creare un flag di status associato ad ogni istruzione, guardando se l'istruzione può causare un'eccezione, quando l'istruzione termina, l'exception viene scatenata, così si crea un coda evitando exception simultanee.

## 6 Chache Memories

Le cache velocizzano l'accesso alla memoria secondaria che è il collo di bottiglia dell'intero sistema. Si è creata una gerarchia di memorie molto più veloci quanto più vicine si trovano al processore.

- registri: 500 bytes, 500ps;
- L1: 64 KB, 2ns;

- L2: 256 KB, 10-20ns;
- Memoria primaria: 512 MB, 50-100ns;
- Memoria secondaria flash: 8 GB, 50 $\mu$ s;

La cache funzionano grazie ai principi di località:

- temporale: in un tempo  $t_0 + \Delta t$  dal momento in cui ho letto un elemento, è probabile che il dato venga o l'istruzione venga riusato;
- spaziale: in un spazio  $x + \Delta x$  vicino all'elemento letto, è probabile che gli elementi vicini vengano letti;

#### Theorem 6.1 – Cache Performance

- $h$ : cache ratio;
- $C$ : cache access time;
- $M$ : memory access time;

Media del tempo di accesso:

$$t_{ave} = h * C + (1 - h) * M$$

Valori soliti per  $h$  sono 0.9.

**Organizzazione della cache** Solitamente la cache è formata da una parte di controllo contenente il **cache controller** che controlla se accedendo alla cache è stato fatto un hit o un miss e in caso recupera la porzione di memoria ed una parte di dati, che contiene le **cache line** fatte da:

- validity bit: il bit ci dice se la riga è valida o meno;
- tag: identifica il blocco di memoria presente nella riga;
- data array: contiene i dati presi dalla memoria;

A partire dall'indirizzo la cache si calcola un nuovo indirizzo di accesso alle righe, formato da:

- tag: identifica il blocco di memoria (bit dell'indirizzo - bit index - bit offset);
- index: riga della cache;
- offset: byte offset all'interno della riga;

Per regolare l'accesso alla cache il controllore identifica la riga attraverso l'index e comparando i due tag decide se è un hit o un miss, se è un hit ed il validity bit è a 1, allora attraverso l'offset viene prelevato il dato.

La cache si trovano tra il bus ed il processore, per evitare conflitti di utilizzo con periferiche esterne o DMA.

Le cache moderne più vicine al processore sono separate in **Instruction-Cache** e **Data-cache**, per evitare la lettura contemporanea di istruzioni e dati (structural hazard).

**Mappatura** I tipi di mappatura (**associativity models**) sono:

- **direct mapped**: la posizione nella riga è uguale a:  $\#block\_memory \bmod \#cache\_block$ ;
- **set associative**: la cache viene partizionata in set di righe (i blocchi hanno tutti la stessa lunghezza, tipicamente 2 o 4), la posizione è determinata da:  $\#memory\_block \bmod \#sets$ , quando un altro blocco viene assegnato ad un set viene rimossa la riga meno utilizzata;
- **fully associative**: ogni blocco di memoria può essere salvata in qualsiasi riga della cache, questo ha come malus la perdita del campo index e l'indirizzo del blocco viene salvato per intero;

**Algoritmi di rimpiazzamento** Gli algoritmi usati per decidere quale riga rimpiazzare sono:

- LRU (last recently used): il più usato;
- FIFO: il meno caro in termini di prestazioni;
- LFU (least frequently used): teorico, il più efficace;
- random: semplice ed efficace;

**Update della Memoria** Quando un'operazione di scrittura è fatta sulla cache deve anche essere propagata sulla memoria, le due possibili soluzioni a questo problema sono:

- **write back**: per ogni riga della cache è introdotto un flag detto **dirty bit**, che indica quando i dati all'interno sono cambiati;
- **write through**: ogni volta che la CPU effettua un'operazione di scrittura, i dati vengono scritti sia in cache che in memoria;



## 7 Branch Prediction

Per effettuare delle predizioni esistono due tipi di approcci: prediction statici e prediction dinamici. Un esempio prediction statico è prendere tutti i salti come presi, oppure facciamo un filtro su quali tipi di branch prendere come presi (prendere i salti all'indietro come sempre presi).

Per avere delle predizioni più accurate è usare un brach prediction dinamico (speculazione). I metodi di predizione sono:

- branch history table;

**Branch History Table** Il BHT ha una memoria in cui sono contenute le informazioni relative ai salti. Si accede a questa tabella quando nella fase di fetch si prende un salto, si guarda l'informazione relativa e al salto e si legge la predizione del salto.

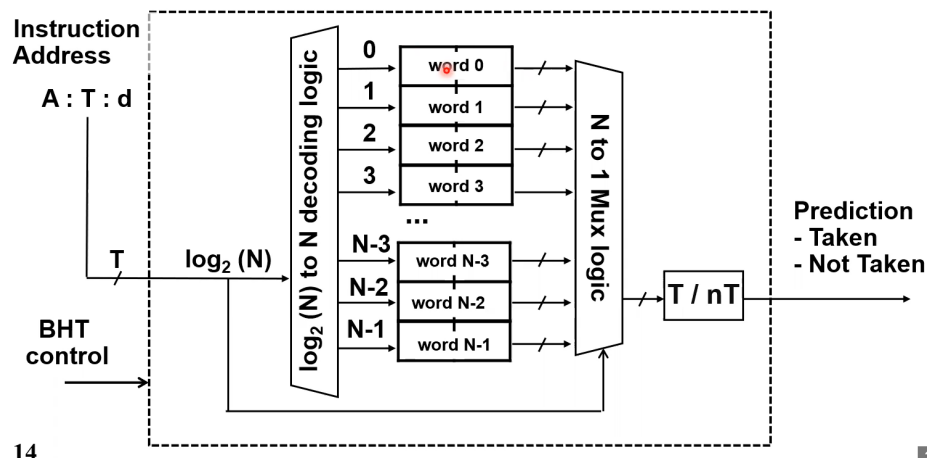


Figure 1: Bht Implementation

Solitamente il valore che indica se il branch deve essere preso o meno solitamente è rappresentato da due bit.

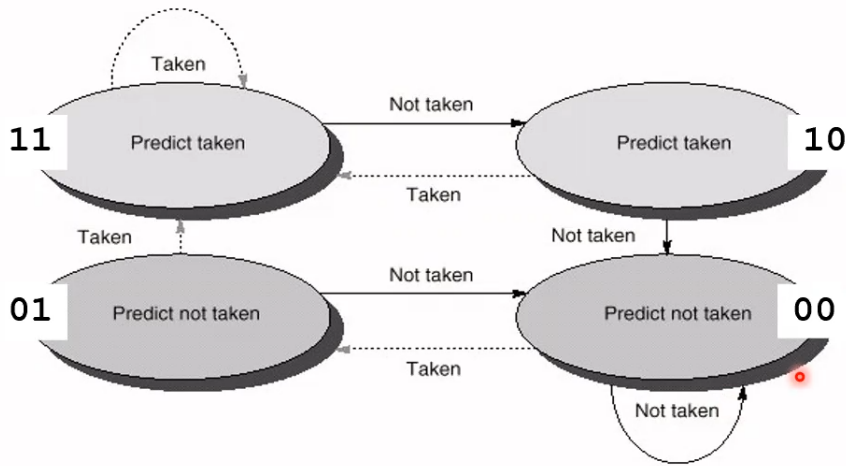


Figure 2: Two Bits Prediction Scheme

Questi tipi di predittori sono stati evoluti, usando dei contatori (**n-bit saturating counter**), quando un salto viene preso il contatore aumenta altrimenti diminuisce, se il bit più significativo è settato ad 1 allora il branch viene preso, questo tipo di contatore di saturazione è molto facile da implementare.

Altri tipi di predittori sono i **(m, n) predictors** guardano agli m-salti precedenti, date le  $2^m$  possibili salti ognuno ha un predittore da n bit.

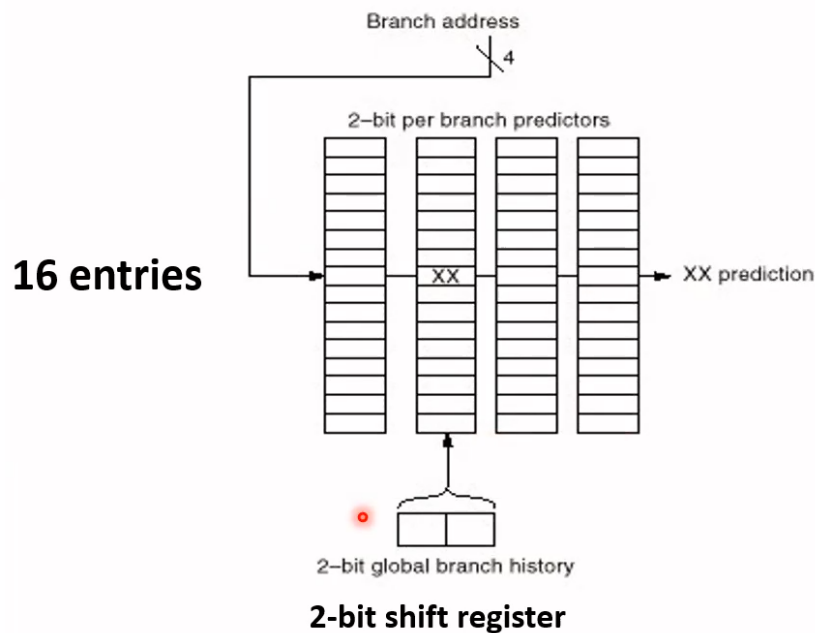


Figure 3: (2,2) Predictor

Questi tipi di predittori non danno informazioni su dove saltare.

Un altro predittore è il **Branch Target Buffer** (usato dal MIPS), questo tipo di predittore ha a disposizione dei registri in cui sono contenuti gli indirizzi di salto, questa struttura contiene l'indirizzo del salto attuale, viene interrogata durante la fase di fetch e sarà poi disponibile al secondo colpo di clock, nel colpo di clock successivo l'indirizzo viene letto, se l'istruzione è un branch e l'istruzione corrente si trova nella tabella, allora il salto deve essere preso, per ogni linea si deve salvare 30bit per indirizzo (vengono tagliati gli ultimi 2 essendo degli indirizzi). Il problema con questo predittore è che è molto costoso, infatti si deve avere un numero di entry non piccolo che sono onerose in termini di memoria.

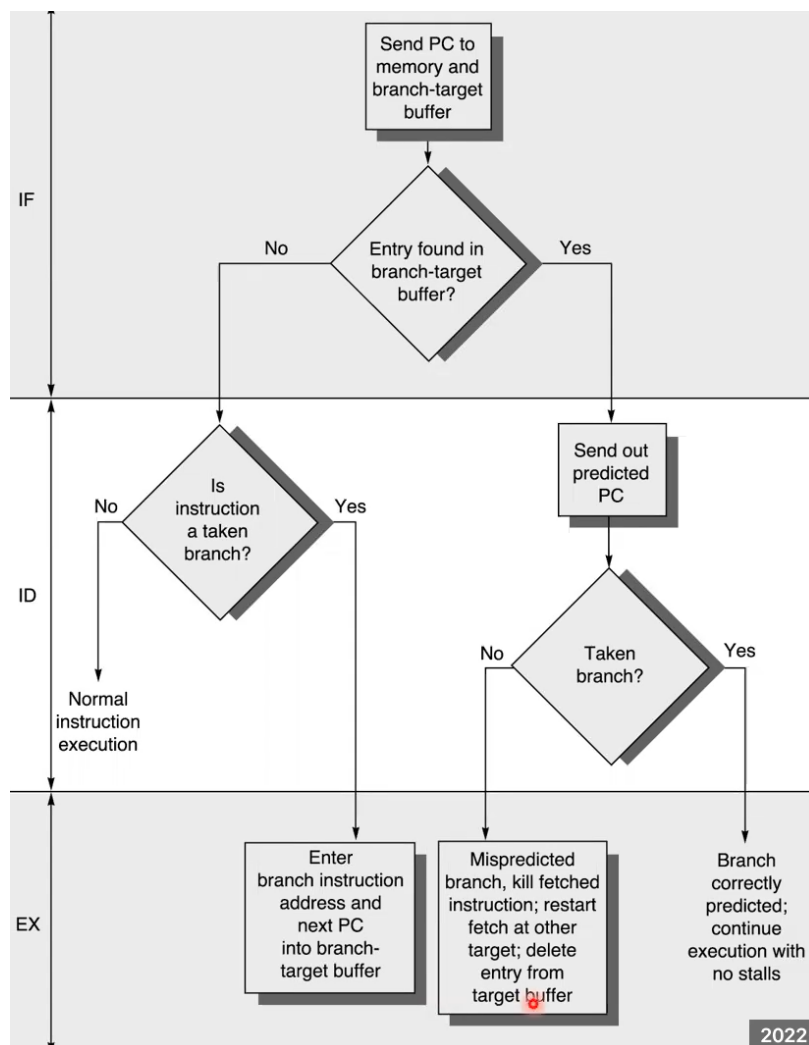


Figure 4: Comportamento Branch Target Buffer

Esistono anche due tipi di predittori chiamati **gselect** e **gshare**.

### Example 7.1 – BHT

## 8 Schedulazione Dinamica

## 9 Pipelining

Esistono delle tecniche per portare le CPI sotto il numero 1, questo è possibile farlo attraverso il fetch di più istruzioni. Esistono due tipi di processori che possono farlo:

- **superscalari**: hanno uno scheduling statico o dinamico;
- **very long instruction word (VLIW)**;

### 9.1 Static Scheduling

Per implementare un processore superscalare viene creato un **issue packet**, dove viene fatto il fetch di due (o più) istruzioni contemporaneamente se in modo statico: una è load, store, branch o operazioni ALU e l'altra è una qualsiasi operazione FP, queste due istruzioni vengono chiamate issue packet.

Instruction type		Pipe stages					
Integer instruction	IF	ID	EX	MEM	WB		
FP instruction	IF	ID	EX	EX	EX	WB	
Integer instruction		IF	ID	EX	MEM	WB	
FP instruction		IF	ID	EX	EX	EX	WB
Integer instruction			IF	ID	EX	MEM	WB
FP instruction			IF	ID	EX	EX	EX
Integer instruction				IF	ID	EX	MEM
FP instruction				IF	ID	EX	EX

Figure 5: Issue Packet Example

In un caso ideale si eseguiranno 0.5 istruzioni per colpo di clock. L'issue packet conterrà sempre una sola istruzione di branch. In questo caso l'unità FP sarà pipelined o indipendente, in qualche modo è possibile ottenere degli hazard, come: fare un'istruzione di load e subito dopo un'istruzione di write, oppure dei possibili RAW (read after write).

Nei sistemi moderni si utilizza una strategia statica in alcuni processori embedded con MIPS.

### 9.2 Dynamic Scheduling

Si può ottenere una schedulazione dinamica. Si ha un Common Data Bus (sistema di forwarding) comune, quindi viene duplicato. Supponiamo di avere le seguenti istruzioni:

```

1 loop:
2     ls r2, 0(r1)
3     daddiu r2, r2, 1
4     sd r2, 0(r1)
5     daddiu r1, r1, 4
6     bne r2, r3, loop

```

Supponiamo che:

- non ci sia speculazione:

Iteration number	Instructions	Issues at clock cycle number	Executes at clock cycle number	Memory access at clock cycle number	Write CDB at clock cycle number	Comment
1	LD R2,0(R1)	1	2	3	4	First issue
1	DADDIU R2,R2,#1	1	5		6	Wait for LW
1	SD R2,0(R1)	2	3	7		Wait for DADDIU
1	DADDIU R1,R1,#4	2	3		4	Execute directly
1	BNE R2,R3,LOOP	3	7			Wait for DADDIU
2	LD R2,0(R1)	4	8	9	10	Wait for BNE
2	DADDIU R2,R2,#1	4	11		12	Wait for LW
2	SD R2,0(R1)	5	9	13		Wait for DADDIU
2	DADDIU R1,R1,#4	5	8		9	Wait for BNE
2	BNE R2,R3,LOOP	6	13			Wait for DADDIU
3	LD R2,0(R1)	7	14	15	16	Wait for BNE
3	DADDIU R2,R2,#1	7	17		18	Wait for LW
3	SD R2,0(R1)	8	15	19		Wait for DADDIU
3	DADDIU R1,R1,#4	8	14		15	Wait for BNE
3	BNZ R2,R3,LOOP	9	19			Wait for DADDIU

Figure 6: Dynamic Scheduling Senza Speculazione

- con speculazione:

Iteration number	Instructions	Issues at clock number	Executes at clock number	Read access at clock number	Write CDB at clock number	Commits at clock number	Comment
1	LD R2,0(R1)	1	2	3	4	5	First issue
1	DADDIU R2,R2,#1	1	5		6	7	Wait for LW
1	SD R2,0(R1)	2	3			7	Wait for DADDIU
1	DADDIU R1,R1,#4	2	3		4	8	Commit in order
1	BNE R2,R3,LOOP	3	7			8	Wait for DADDIU
2	LD R2,0(R1)	4	5	6	7	9	No execute delay
2	DADDIU R2,R2,#1	4	8		9	10	Wait for LW
2	SD R2,0(R1)	5	6			10	Wait for DADDIU
2	DADDIU R1,R1,#4	5	6		7	11	Commit in order
2	BNE R2,R3,LOOP	6	10			11	Wait for DADDIU
3	LD R2,0(R1)	7	8	9	10	12	Earliest possible
3	DADDIU R2,R2,#1	7	11		12	13	Wait for LW
3	SD R2,0(R1)	8	9			13	Wait for DADDIU
3	DADDIU R1,R1,#4	8	9		10	14	Executes earlier
3	BNE R2,R3,LOOP	9	13			14	Wait for DADDIU

Figure 7: Dynamic Scheduling Con Speculazione

## 10 ARM

Nei lab verrà usato il **Coretex M3** che fa parte di ARM9. ARM sviluppa architetture per tre categorie:

- architetture embedded (SoC: system on a chip);
- sistem operativi;
- compilazione - supporto - debug tools

Uno dei moduli è il **Memory BIST**: questi parti del SoC servono per il collaudo del sistema, infatti durante la produzione sul silicio le memoria potrebbero avere dei problemi fisici (quando si scrive in un registro il valore salvato non è corretto o potrebbe intaccare quello dei registri vicini), infatti questi moduli non interagiscono direttamente con la logica del processore.

In un dispositivo ARM compliant la prima parte di codice che viene eseguita è il bootloader, per poi far partire il sistema operativo. Guadando la toolchain, si parte dal codice assembly ARM, e dal codice C/C++, questi verranno compilati da un CROSS COMPILER, il risultato finale sarà un codice compilato con un ISA per ARM. Il risultato sarà un eseguibile, con questo eseguibile sarà possibile usarlo in un simulatore di una scheda oppure caricarlo direttamente sulla scheda.

ARM cortex-M3 contiene 16 registri, tra cui il 15 è il PC. Il **barrel shifter** si avrà la possibilità di creare valori immediati fino a 32 bit, solo se il valore contiene in qualche modo una replicazione all'interno di se stesso, inoltre ci permette di fare uno shift automatico del secondo registro su cui stiamo operando salvando un'istruzione.

Esistono alcuni casi particolari:

- istruzioni **registro-registro**: dati i due registri Rn ed Rm, il primo entra direttamente nell'ALU, il secondo può essere modificato, al termine dell'operazione il risultato viene riportato in un registro;
- ...

I vari moduli sono:

- nested vectored interrupt controller;
- wake up interrupt controller interface: permette di mandare il dispositivo in sleep;
- memoria: interface code, memory protection unit;
- SRAM e interfacce periferiche: possibilità di parlare con i timer;
- debug access port: permette di effettuare il debug anche attraverso il codice;



- ITM trace, ETM trace, data watchpoint, flash patch: moduli che permettono di fare il debug;

La pipeline è formata da: ...

Le istruzioni di branch comportano una perdita di due cicli. Quando si legge dalla memoria si perde un colpo di clock.

## 10.1 Register

Non tutti i registri sono general purpose, infatti:

- 15: PC;
- 14: link register;
- 13: stack pointer: l'sp ha due versioni;

L'istruzione set utilizzato sarà il **thumb 2**, che prende le caratteristiche positive del thumb (a 16 bit) e dell'ARM, in thumb 2 esistono istruzioni a 16 e 32 bit, in questo modo i programmi sono più piccoli e le prestazioni sono simili all'ARM.

...

Ogni istruzione può essere eseguita in modo condizionato. L'architettura load e store si può utilizzare un formato di tre operandi.

## 10.2 Instruction Set

In questa architettura il PC è immagazzinato in r15, questo registro è modificabile, modificare il PC è importante quando ad esempio si ritorna da un subroutine, il registro r14 link register salva il valore di memoria di ritorno che andrà copiato nel PC. Il registro r13 è lo stack pointer, che permette di avere il valore dell'ultimo oggetto inserito, questo registro ha un valore iniziale, al termine del programma il valore dello stack pointer verrà ricaricato con il suo valore iniziale che si trova all'interno della **interrupt vector table**, la posizione dello stack precedente si trova nell'index 0. Esiste anche il program status register è diviso in 3 registri, a seconda dell'operazione che si sta facendo si potrebbe accedere solo ad una parte del registro, vi sono dei flag delle operazioni aritmetiche, questo registro è diviso in:

- application program status register (apsr);
- execution program status register (epsr);
- interrupt program status register (ipsr);

I flag sono:

- Z zero;

- N negative;
- C carry;
- V overflow;

L'esecuzione condizionata delle istruzioni non vale solo per i salti, ma per ogni istruzione, nei 32 bit delle istruzioni ARM (comprese nella versione thumb 2), si hanno 4 bit che indicano il tipo di condizione che determina se l'istruzione viene eseguita oppure no, questo viene fatto leggendo i flag o secondo certe condizioni. Se si vuole che un'istruzione modifichi un flag dobbiamo chiderlo esplicitamente, aggiungendo un **S** alla fine dell'istruzione.

v5.36 di Keil

Quando si organizza il codice si dovranno avere delle sezioni di codice. Nella memoria di codice readonly (quello che va in ROM) si possono definire delle costanti. Per salvare delle costanti in memoria esistono delle direttive per definire un tipo di dato (le direttive iniziano con **DC\*\***).

```

1 my_matrix    DCD    1, 2, 3, 4
2              DCD    3, 4, 5, 6
3              DCD    7, 8, 9, 1
4              DCD    8, 9, 6, 3
5
6 my_const     DCD    10

```

Come ad esempio il salvataggio di una matrice e di una costante con dati.

Un'altra direttiva molto importante è quella di **LTORG**, ovvero dei **literal pool**, che permette al compilatore di accedere direttamente ad alcune costanti che non sono accessibili direttamente, ad esempio se un valore immediato è troppo grande (quando si fa un'operazione come `LDR r1, =0x12345678`) viene salvato in questa zona dalla quale si può accedere.

Una volta fatte delle operazioni si possono voler salvare le variabili in memoria RAM, per fare questo si utilizza direttiva **AREA**, questa direttiva prende dei parametri:

- *—nome della sezione—*;
- **DATA / CODE**: definisce se la zona è di codice o di dati;
- **READONLY / READWRITE**: definisce se si può leggere o scrivere;
- **align=x**: definisce l'allineamento dei dati ...;

Per caricare un registro in memoria si utilizza un valore **pre-indicizzamento**: `load/store Rd, [Rt, <offset>]{!}`, se si usa **!** il valore del registro viene incrementato prima di essere letto e quindi viene salvato se non si mette il **!** il valore del registro viene incrementato dopo esser stato letto. L'offset può essere anche un

valore salvato in un registro. Oppure usando si può usare un valore **pre-indicizzato**:  
 load/store Rd, [Rt], <offset>

La direttiva **EQU** ci permette di definire delle macro, ovvero associare un valore ad un nome, esiste anche la **RN** che ci permette di definire un alias per un registro.

Esempio di moltiplicazione di matrice, ogni elemento va moltiplicato per una costante e poi salvato:

```

1  AREA MY_DATA, DATA, READWRITE, align=3
2  NEW_MATRIX SPACE 5 * 4 * 4
3
4
5  AREA |.text|, CODE, READONLY
6
7
8  ROW EQU 5
9  COL EQU 4
10
11 ELEMENTS EQU ROW * COL
12
13 matrix RN 0
14 new_matrix RN 1
15 i RN 2
16 j RN 3
17 const RN 4
18
19
20 ; Reset Handler
21
22 Reset_Handler PROC
23     EXPORT Reset_Handler [WEAK]
24
25
26     LDR matrix, =MATRIX
27     LDR new_matrix, =NEW_MATRIX
28     LDR const, =CONSTANT
29     LDR const, [const]
30     MOV i, #0
31     MOV j, #0
32
33
34 ciclo_riga MOV j, #0
35
36 ciclo_colonna MOV r6, #COL
37     MUL r6, i, r6
38     ADD r6, r6, j
39     LDR r7, [matrix, r6, LSL #2]
40
41     MUL r7, r7, const
42     STR r7, [new_matrix, r6, LSL #2]
43

```

```

44     ADD j, j, #1
45     CMP j, #COL
46     BNE ciclo_colonna
47
48     ADD i, i, #1
49     CMP i, #ROW
50     BNE ciclo_riga
51
52     ENDP
53
54 MATRIX      DCD  1, 2, 3, 4
55             DCD  3, 4, 5, 6
56             DCD  7, 8, 9, 1
57             DCD  8, 9, 6, 3
58
59 CONSTANT    DCD 10
60
61     LTORG

```

## 10.3 Stack

Nel thumb2 lo stack è discendente, infatti quando si fa un push di un dato il valore dello stack decresce. Esistono anche gli stack ascendenti, entrambi si differenziano per essere full o empty, che differenziano il modo in cui lo SP si muove.

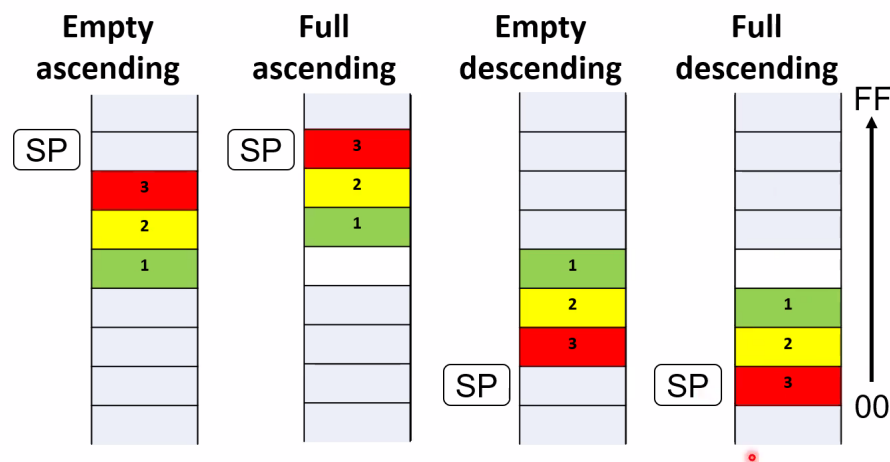


Figure 8: Tipi Di Stack

Per inserire dei dati nello stack si utilizzano.

```

1 LDM {xx} / STM {xx} <Rn>!, <regList>

```

Dovremmo fornire la lista dei registri, ad esempio `r0-r4, r10, LR`, il compilatore ordina tutto e poi vengono salvati. Il modo in cui avviene è salvano il registro più

basso nella posizione più bassa dello stack. Lo SP non si può aggiungere nella lista, mentre il PC ed il LR sono mutuamente esclusivi.

I modi di indirizzamento sono IA (increment after, di default), DB (decrement before). Esistono delle istruzioni che implementano la PUSH e la POP nel caso di full descending o empty ascending. Nel nostro caso: `PUSH <regList> = STMDB SP!, <regList>; POP <regList> = LDMIA SP!, <regList>.`

Tutto questo serve per implementare delle **subroutine**, esistono delle istruzioni che mi permettono di saltare e linkare (`BL <label>`, `BLX <Rn>`). Per determinare che un segmento di codice è una subroutine si utilizzano le keyword `PROC/FUNCTION`, `ENDP/ENDFUNC`. Esiste il problema del passaggio dei parametri, esistono 3 approcci:

- dai registri;
- by reference;
- dallo stack;

Esistono comunque degli standard di chiamata, il motivo è che potremmo chiamare delle subroutine da codice C.

Esempi di codice che fa la sottrazione ed il valore assoluto passando i 3 tipi di argomenti:

```

1 Reset_Handler
2   PROC
3
4   ;registri
5   mov r0, #42
6   mov r1, #37
7
8   BL sub1
9
10
11  ;by reference
12  mov r0, #42
13  mov r1, #37
14  LDR r3, =mySpace
15  STAMIA r3, {r0, r1}
16  BL sub2
17  LDR r2, [r3]
18
19
20  ;stack
21  mov r0, #42
22  mov r1, #37
23
24  PUSH {r0, r1, r2} ;r2 = valore di ritorno
25  BL sub3
26  POP {r0, r1, r2}
27

```

```
28     ENDP
29
30 sub1
31     PROC
32     PUSH {lr}
33
34     CMP r0, r1
35     SUBGE r2, r0, r1
36     SUBL0 r2, r1, r0
37
38     POP {pc}
39
40     ENDP
41
42 sub2
43     PROC
44     PUSH {r2, r4, r5, LR}
45     LDMIA r3, {r4, r5}
46     CMP r4, r5
47     SUBHS r2, r4, r5
48     SUBL0 r2, r5, r4
49     STR r2, [r3]
50
51     POP {r2, r4, r5, PC}
52     ENDP
53
54 sub 3
55     PROC
56     PUSH {r4, r5, r6, lr}
57     LDR r4, [sp, #16]
58     LDR r5, [sp, #20]
59
60     CMP r4, r5
61     SUBGE r6, r4, r5
62     SUBL0 r6, r5, r4
63
64     STR r6, [sp, #24]
65
66     POP {r4, r5, r6, pc}
67     ENDP
```

## 11 Application Binary Interface

Un ABI sono un insieme di standard per definire la comunione tra due moduli di programmi binari, nel nostro caso quando in C deve essere chiamata una procedura assembler, vengono definiti come dovranno essere passati i parametri. L'interesse sarà relativo alla chiamate a procedure e alla gestione delle eccezioni. Un esempio di ABI per i nomi dei registri:

Register	Synonym	Special	Role in the procedure call standard
r15		PC	The Program Counter.
r14		LR	The Link Register.
r13		SP	The Stack Pointer.
r12		IP	The Intra-Procedure-call scratch register.
r11	v8		Variable-register 8.
r10	v7		Variable-register 7.
r9		v6 SB TR	Platform register. The meaning of this register is defined by the platform standard.
r8	v5		Variable-register 5.
r7	v4		Variable register 4.
r6	v3		Variable register 3.
r5	v2		Variable register 2.
r4	v1		Variable register 1.
r3	a4		Argument / scratch register 4.
r2	a3		Argument / scratch register 3.
r1	a2		Argument / result / scratch register 2.
r0	a1		Argument / result / scratch register 1.

Figure 9: Register Abi

### 11.1 Supervisor Calls (SVC)

Sono i Software Interrupt. Si dividono in Exception per le interruzioni software e Interrupt che sono le interruzioni hardware. Nell'architettura ARM si possono per sollevare un interrupt si utilizza **SVC #id**, dove l'id è l'identificativo dell'interrupt.

La **Interrupt Vector Table** è una lista che specifica l'handler delle procedura che gestisce l'eccezione, fa eccezione solo la prima riga che è il valore iniziale dello stack pointer. Ogni riga ha uno spazio di 4 byte.

Exception Type	Index	Vector Address
(Top of Stack)	0	0x00000000
Reset	1	0x00000004
NMI	2	0x00000008
Hard fault	3	0x0000000C
Memory management fault	4	0x00000010
Bus fault	5	0x00000014
Usage fault	6	0x00000018
SVcall	11	0x0000002C
Debug monitor	12	0x00000030
PendSV	14	0x00000038
SysTick	15	0x0000003C
Interrupts	≥16	≥0x00000040

Figure 10: Ivt

Ogni eccezione ha diverse priorità, dopo l'hard fault è possibile programmarle. Le eccezioni hanno uno stato:

- inattiva;
- attiva: un'interruzione che sta venendo servita sul processore ma non è completata;
- pending: l'eccezione sta aspettando di essere schedata sul processore;

Quando arriva un'eccezione su un processore ARM, il processore deve saltare al suo handler, ma prima di fare ciò devono essere salvati il registro dei flag, pc, lr, r12, r3-r0, questo processo viene fatto in automatico.

La sintassi delle svc è: `{label} SVC immediate`, il valore dell'immediato è a 8 bit, infatti l'SVC è una istruzione del thumb a 16 bit. Dopo l'esecuzione di una SVC, oltre ad i registri salvati nello stack, il LR viene caricato con un valore (diverso da quello di ritorno) detto **EXC\_RETURN**.



EXC_RETURN	Description
0xFFFFFFF1	Return to Handler mode. Exception return gets state from the main stack. Execution uses MSP after return.
0xFFFFFFF9	Return to Thread mode. Exception Return get state from the main stack. Execution uses MSP after return.
0xFFFFFFF0	Return to Thread mode. Exception return gets state from the process stack. Execution uses PSP after return.
All other values	Reserved.

Figure 11: Exception Return

Per leggere il valore dell'immediato si prende il valore del PC salvato nello stack, essendo che il valore viene compilato insieme all'istruzione, si può recuperare il valore leggendo l'istruzione precedente (perchè lo SP è incrementato) e facendo un BIC (bit clear) ed uno shift.

Si può utilizzare la MSR sono quando si è in handler mode,  $MSR\{cond\} \text{ spec\_reg}, Rn$ .

Le modalità di utilizzo sono:

- **thread mode**: dopo un reset o dopo un'eccezione;
- **handler mode**: quando arriva un'eccezione;

I livelli di accesso sono:

- **user level**: accesso limitato;
- **privileged level**: accesso a tutte le risorse;

Handler mode è sempre a livello privilegiato.

A seconda del valore del registro di controllo si avrà lo PSP (process stack pointer) normale o un MSP (master stack pointer) quando ci si trova in handler mode. È importante ricordare che l'esecuzione è sempre in handler mode quando viene chiamata un'eccezione, bisogna quindi definire delle sezioni di per lo stack, ovvero un sezione per il PSP ed un per il MSP.

```

1 ; stack segment
2 StackSize EQU 0x00000200
3 AREA STACK, NOINIT, READWRITE, ALIGN=3
4 SPACE StackSize/2
5 StackProcess SPACE StackSize/2
6 __initial_sp

```

Quando si vuole passare a thread mode ed impostare il valore del PSP, si deve:

```

1 MOV R0, #3 ; user mode
2 MSR CONTROL, R0
3 LDR SP, =StackProcess
4
5 SVC 0x10

```

L'handler andrà gestito nel seguente modo:

```

1 PUSH {r0-r12, lr}
2
3 TST lr, #2_1000
4 MSREQ r1, PSP ; thread mode
5 LDREQ r0, [r1, #(14 + 6)*4]
6
7 MSRNE r1, MSP ; handler mode
8 LDRNE r0, [r1, #(6)*4]
9
10 BIC r0, 0xff000000
11 LSR r0, #16
12
13 ...
14
15 POP {r0-r12, lr}
16 BX lr

```

## 12 ASM+C

Per usare una funzione scritta in C da ASM si usa:

```

1 Reset PROC
2 EXPORT Reset
3 IMPORT __main
4 LDR r0, =__main
5 BX r0

```

Per usare una funzione scritta in ASM da C si usa:

```

1 exter int ARM_func(int, int, int);
2
3 int main(void)
4 {
5     int i = 0xFF, j = 2, k = 3;
6     volatile int r = 0;

```

```

7
8     r = ASM_func(i, j, k);
9     return 0;
10 }

```

La funzione `ASM_func` è dichiarata come:

```

1     AREA asm_functions, CODE, READONLY
2     EXPORT ASM_func
3
4 ASM_func
5     ; save current SP for a faster access
6     ; to parameters in the stack
7     MOV r12, sp
8     PUSH sp, {r4-r9,r10,r11,lr}
9
10    LDR r4, [r12]
11    LDR r5, [r12, #4]
12    ; prepare return value
13    MOV r0, r5
14
15    PUSH sp, {r4-r9,r10,r11,lr}
16    END

```

Se davanti ogni variabile non viene aggiunta la keyword `volatile`, il compilatore (in modalità ottimizzata) potrebbe decidere di non usare un registro perchè pensa che sia inutile.

## 13 Scheda

Per configurare gli **switch** (bottoni) in modo far scatenare un eccezione si ha bisogno di di settare i bit della porta a 01, per abilitare l'eccezione `EINT0`, si fa:

```

1 void buttonInit(void) {
2     // Porta relativa all'abilitazione dell'eccezione
3     // dello switch
4     LPC_PINCON->PINSEL4 |= (1 << 20);
5     // Registro della direzione dell'input
6     LPC_GPIO2->FIODIR  &= ~(1 << 10);
7
8     // ...
9
10    // Interrupt sensitivo al fronte di salita
11    LPC_SC->EXTMODE = 0x7;
12
13    // Abilitazione della interruzioni e delle priorit\ 'a
14    NVIC_EnableIRQ(EINT0_IRQn);
15    NVIC_SetPriority(EINT2_IRNqn, 1);
16
17    // ...
18 }

```

Un esempio di handler sarà:

```

1 void EINT0_IRQHandler (void)
2 {
3     LED_On(0);
4     /* clear pending interrupt */
5     LPC_SC->EXTINT &= (1 << 0);
6 }

```

## 13.1 Progetto Interrupt Controller

Gli eventi che si scatenano possono essere periodici o asincroni. I tipi di interruzione possono essere:

Si possono gestire le interruzioni in modo:

- **polling**: gestito dal software, ovvero si usa un ciclo infinito per controllare periodicamente se dei registri relativi alle periferiche vengono modificati, in caso di modifica vengono gestite di conseguenza;
- **interrupt**: si devono configurare i periferici in modo che quando avviene un cambiamento, viene scatenata un'interruzione, la risposta a questi eventi saranno gestite dalle loro priorità, per gestirle si usa l'Interrupt Vector Table;

Quando si vuole configurare un sistema con le interruzioni vanno inizializzate diverse istruzioni, come la configurazione dei periferici o dei valori iniziali da assegnare a dei registri, decidere la priorità ed i periferici che possono interrompere, in oltre va fatto un **acknowledge** che un interruzione è avvenuta, questo consiste nel scrivere un valore in un registro per non far risultare più l'interruzione come pending.

L'**interrupt controller**, che si trova all'interno del microcontrollore (solitamente), gestisce i pedini delle interruzioni e decide le priorità dei vari tipi. Il cortex può gestire fino a 35 possibili interrupt.

## 13.2 System Timing

Alcuni sistemi (come il cortex), ci permette di utilizzare i temporizzatori, o catturare l'intervallo di tempo tra due eventi.

Se si vuole che i timerentino per un certo periodo allora sfruttiamo la formula:  $time[s] = \frac{count}{clockPeriod[s]}$  (la costante che il simulatore vuole in input è il *count*), ogni volta che passa questo periodo viene scatenata un eccezione.

I timer a disposizione nel sistema sono 4, 2 sono attivi di default. Per assegnare un timer count, si incorre nel problema che il valore potrebbe essere più grande della grandezza di un registro, per risolvere questo problema l'lp1768 funziona con un **prescaler**, ovvero usa un valore a 64 bit che richiede due registri. I **match register** sono dei registri particolari che matchano il timer counter, ogni qual volta vengono

matchati si possono fare varie operazioni, come lanciare un eccezione, modificare il valore dei match registre o del timer, o controllare dei piedini.

La libreria dei match register ci facilita la configurazoine, offrendo:

- generazione di interrupt;
- reset il timer counter;
- stoppare il timer counter;

Per inizializzare i timer:

```
1 int main()  
2 {  
3  
4  
5 }
```

*La libreria dei timer (che fa un po schifo a detta del professore) possiede anche un wizard di configurazione.*