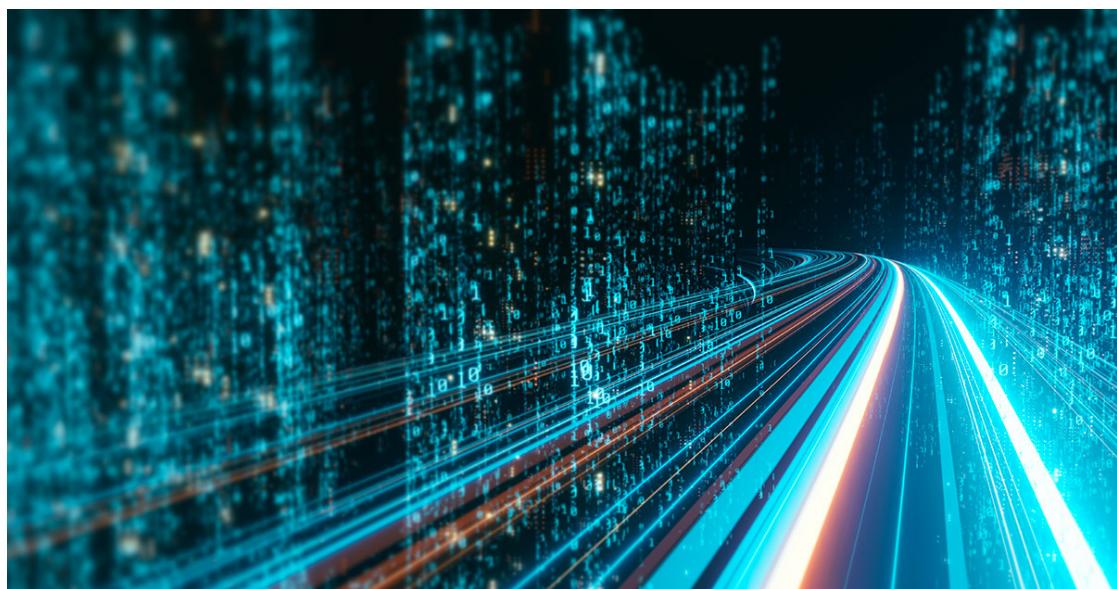


---

Enrico Lacchin

# Architetture dei sistemi digitali

*Appunti*



Materia: Architetture dei sistemi digitali

Docente: Alberto Carini

Link moodle: <https://moodle2.units.it/course/view.php?id=9113>



## Indice

<b>1 Computer Abstractions and Technology</b>	<b>1</b>
1.1 Introduzione . . . . .	1
1.2 Classi tradizionali di Computer . . . . .	1
1.2.1 L'era PostPC . . . . .	2
1.3 Definizioni . . . . .	2
1.4 Comprendere le performance di un programma . . . . .	2
1.5 8 Grandi idee . . . . .	3
1.6 Al di sotto del programma . . . . .	5
1.7 Da un linguaggio ad alto livello al linguaggio hardware . . . . .	6
1.8 Componenti di un computer . . . . .	7
1.8.1 Processore (CPU) . . . . .	7
1.8.2 Memoria . . . . .	8
1.9 Astrazioni . . . . .	8
1.10 Tecnologie per costruire processori e memorie . . . . .	9
1.11 Performance . . . . .	10
1.11.1 Tempo di risposta e Produttività . . . . .	10
1.11.2 Performance relative . . . . .	11
1.11.3 Misurare le Performance . . . . .	11
1.11.4 CPU Clocking . . . . .	11
1.11.5 Performance delle istruzioni . . . . .	12
1.11.6 CPU Performance - Equazione classica . . . . .	12
1.11.7 CPI nel dettaglio . . . . .	12
1.11.8 Riassunto . . . . .	13
1.12 The power wall . . . . .	13
1.12.1 Dissipazione di potenza in CMOS . . . . .	14
1.12.2 Problemi moderni nella dissipazione dell'energia . . . . .	14
1.13 Multiprocessori . . . . .	15
1.14 SPEC CPU Benchmark . . . . .	15
1.15 SPEC Power Benchmark . . . . .	15
1.16 Errore: Bassa potenza al minimo . . . . .	16
1.17 Insidia: Legge di Amdahl . . . . .	17
1.18 Fallacia: Computers at low utilization use little power . . . . .	17
1.19 Insidia: MIPS come metrica delle performance . . . . .	17
<b>2 Instructions: Language of the Computer</b>	<b>18</b>
2.1 Introduzione . . . . .	18
2.2 Operazioni dell'Hardware dei Computer . . . . .	18
2.2.1 Compilare un codice C in LEGv8 . . . . .	18
2.3 Operandi dell'Hardware dei Computer . . . . .	19
2.3.1 Compilare un codice C utilizzando i registri . . . . .	19
2.4 Operandi di Memoria . . . . .	19
2.5 Costanti e Operandi Immediati . . . . .	21



---

2.6 Chiarimenti . . . . .	21
2.7 Rappresentare le istruzioni nel computer . . . . .	21
2.8 LEGv8 istruzioni in formato R . . . . .	22
2.8.1 Buoni compromessi . . . . .	22
2.9 LEGv8 istruzioni in formato D . . . . .	23
2.10 LEGv8 istruzioni in formato I . . . . .	23
2.11 Memorizzazione dei programmi nei Computer . . . . .	23
2.12 Operazioni logiche . . . . .	24
2.13 Differenze tra ARMv8 e LEGv8 . . . . .	25
2.14 Istruzioni per prendere decisioni . . . . .	25
2.15 Compilazione degli <i>if – else</i> . . . . .	26
2.16 Compilazione dei cicli <i>while</i> . . . . .	27
2.17 Blocchi base . . . . .	27
2.18 Altre operazioni condizionali . . . . .	28
2.18.1 Esempio . . . . .	29
2.19 Supporto delle Procedure nei Computer . . . . .	29
2.19.1 Supporto del LEGv8 alle procedure . . . . .	29
2.19.2 Usando più registri . . . . .	30
2.19.3 Compilazione di una procedura che non chiama un'altra procedura	30
2.19.4 Utilizzo dei registri . . . . .	31
2.20 Procedure Non-Leaf . . . . .	31
2.20.1 Esempio . . . . .	32
2.20.2 Conservato / Non conservato tra le chiamate . . . . .	33
2.21 Gestione delle variabili C . . . . .	33
2.22 Struttura delle procedure . . . . .	33
2.23 Layout della Memoria . . . . .	34
2.24 Sommario delle convenzioni dei registri . . . . .	34
2.25 Operazioni Byte/Halfword . . . . .	35
2.26 Esempio "String Copy" . . . . .	35
2.27 Operandi Immediati ampi . . . . .	36
2.28 Indirizzi nei salti . . . . .	36
2.29 Sintesi della modalità di indirizzamento LEGv8 . . . . .	37
2.30 Sintesi della codifica LEGv8 . . . . .	38
2.31 Parallelismo e Istruzioni: Sincronizzazione . . . . .	38
2.31.1 Sincronizzazione in LEGv8 . . . . .	39
2.32 Tradurre e avviare un programma . . . . .	39
2.32.1 Compilatore . . . . .	40
2.32.2 Assemblatore . . . . .	40
2.32.3 Linker (anche chiamato linker editor) . . . . .	41
2.32.4 Loader . . . . .	43
2.32.5 Librerie collegate dinamicamente (DLL) . . . . .	43
2.32.6 Esempio di ordinamento in C . . . . .	44
2.33 Array vs Puntatori . . . . .	47

---



---

2.34 Istruzioni ARMv7 (32-bit) . . . . .	48
2.35 Il resto del set di istruzioni ARMv8 . . . . .	48
2.35.1 ARMv8 Istruzioni logiche aritmetiche intere . . . . .	49
2.35.2 ARMv8 Istruzioni per il trasferimento dati . . . . .	52
2.36 ARMv8 Istruzioni di Branch . . . . .	54
2.37 Errori . . . . .	55
2.38 Insidie . . . . .	55
<b>3 Aritmetic in LEGv8</b> . . . . .	<b>56</b>
3.1 Moltiplicazione in LEGv8 . . . . .	56
3.2 Divisione in LEGv8 . . . . .	56
3.3 Floating Point . . . . .	57
3.3.1 Floating Point Standard IEEE Std 754-1985 . . . . .	57
3.3.2 Range in singola precisione . . . . .	57
3.3.3 Range in doppia precisione . . . . .	58
3.3.4 Infiniti e NaNs . . . . .	58
3.3.5 Numeri denormalizzati . . . . .	58
3.3.6 Riassunti IEEE Std 754-1985 . . . . .	59
3.4 Overflow & Underflow . . . . .	59
3.4.1 Managing overflows and underflows . . . . .	59
3.5 Istruzioni Floating-Point in LEGv8 . . . . .	59
3.5.1 Floating-point in linguaggio Assembly . . . . .	60
3.5.2 Floating-point in linguaggio macchina . . . . .	61
3.6 Aritmetica accurata . . . . .	61
3.7 The BIG picture . . . . .	61
3.8 Parallelismo delle subword . . . . .	62
3.9 ARMv8 SIMD . . . . .	62
3.10 ARMv8 Operazioni aritmetiche intere e floating-point . . . . .	63
3.11 Istruzioni core LEGv8 . . . . .	65
3.12 Insidie ed Errori . . . . .	66
<b>4 The Processor</b> . . . . .	<b>67</b>
4.1 Introduzione . . . . .	67
4.1.1 Organizzazione di Von Neumann . . . . .	67
4.1.2 Organizzazione di Harward . . . . .	67
4.2 Un'implementazione di base di LEGv8 . . . . .	67
4.3 Esecuzione delle istruzioni . . . . .	68
4.4 CPU Overview . . . . .	68
4.4.1 Multiplexer . . . . .	69
4.4.2 Controllore . . . . .	69
4.5 Convenzioni di progettazione logica . . . . .	69
4.6 Metodologia di clocking . . . . .	70
4.7 Il percorso dati semplice . . . . .	70
4.7.1 Il percorso dati semplice con controllore . . . . .	71

---



---

4.8	Controllo della ALU . . . . .	71
4.9	Formato istruzioni . . . . .	72
4.9.1	ADD X1, X2, X3 . . . . .	73
4.9.2	LDUR X1, [X2, offset] . . . . .	73
4.9.3	CBZ X1, offset . . . . .	74
4.10	Perché un'implementazione a ciclo singolo non viene utilizzata oggi? . . . . .	75
4.11	Strategie di controllo multiciclo . . . . .	76
4.12	Controllori microprogrammati . . . . .	77
4.13	Controllore cablato . . . . .	78
4.14	Organizzazione Multi ciclo vs Singolo ciclo . . . . .	79
4.15	Pipeline . . . . .	79
4.15.1	Prestazioni a Singolo ciclo vs Pipeline . . . . .	80
4.16	Accelerazione della Pipeline . . . . .	81
4.17	Progettazione di set di istruzioni per la pipeline . . . . .	81
4.17.1	Pericoli della pipeline . . . . .	82
4.17.2	Pericoli strutturali . . . . .	82
4.17.3	Pericoli di dati . . . . .	82
4.17.4	Pericoli di controllo . . . . .	83
4.17.5	Riepilogo della pipeline: il quadro generale . . . . .	86
4.18	Pipeline Datapath in LEGv8 . . . . .	86
4.18.1	Registri di pipeline . . . . .	86
4.18.2	Operazioni della pipeline . . . . .	87
4.18.3	IF per Load, Store, . . . . .	87
4.18.4	ID per Load, Store, . . . . .	87
4.18.5	EX per Load . . . . .	88
4.18.6	MEM per Load . . . . .	88
4.18.7	WB per Load . . . . .	88
4.18.8	Datapath corretto per il Load . . . . .	89
4.18.9	EX per Store . . . . .	89
4.18.10	MEM per Store . . . . .	90
4.18.11	WB per Store . . . . .	90
4.19	Diagramma della pipeline Multi ciclo . . . . .	90
4.20	Diagramma della pipeline a Singolo ciclo . . . . .	91
4.21	Controllo pipeline (semplificato) . . . . .	91
4.21.1	Controllo pipeline . . . . .	91
4.22	Pericoli di dati nelle istruzioni ALU . . . . .	92
4.22.1	Dipendenze & Forwarding . . . . .	92
4.22.2	Rilevare la necessità di forward . . . . .	92
4.22.3	Percorsi di Forwarding . . . . .	93
4.22.4	Condizioni di Forwarding . . . . .	93
4.22.5	Condizioni di rilevamento . . . . .	94
4.22.6	Doppio rischio di dati . . . . .	94
4.22.7	Datapath con Forwarding . . . . .	95



4.22.8 Rilevamento dei pericoli per l'uso del load . . . . .	95
4.22.9 Come mettere in stallo la pipeline . . . . .	95
4.22.10 Pericolo di utilizzo del carico . . . . .	96
4.22.11 Percorso dati con rilevamento dei pericoli . . . . .	96
4.23 Stalli e Performance . . . . .	96
4.24 Pericoli nei branch . . . . .	96
4.24.1 Ridurre il ritardo del branch . . . . .	97
4.24.2 Esempio . . . . .	97
4.25 Previsione dinamica del branch . . . . .	98
4.25.1 Predictor a 1 bit: Shortcoming . . . . .	98
4.25.2 Predictor a 2 bit . . . . .	99
4.25.3 Tampone target di branch e altri predittori di branch . . . . .	99
4.26 Exceptions and Interrupts . . . . .	99
4.26.1 Gestione delle eccezioni . . . . .	100
4.26.2 Un meccanismo alternativo . . . . .	100
4.26.3 Azioni del gestore . . . . .	100
4.27 Exceptions in una Pipeline . . . . .	101
4.27.1 Proprietà . . . . .	101
4.27.2 Esempio . . . . .	102
4.27.3 Eccezioni multiple . . . . .	102
4.27.4 Eccezioni imprecise . . . . .	103
4.28 Interfaccia HW/SW . . . . .	103
4.29 Parallelismo tramite istruzioni . . . . .	104
4.29.1 Multiple issue . . . . .	104
4.30 Il concetto di speculazione . . . . .	104
4.30.1 Speculazione Compiler / Hardware . . . . .	105
4.31 Static Multiple Issue . . . . .	105
4.31.1 LEGv8 con Static Dual Issue . . . . .	105
4.31.2 Esempio di pianificazione . . . . .	106
4.31.3 Svolgimento del ciclo per pipeline a più emissioni . . . . .	107
4.32 Dynamic Multiple Issue . . . . .	108
4.32.1 Dynamic Pipeline Scheduling . . . . .	108
4.33 Rinominazione dei registri . . . . .	109
4.34 Esecuzione Out-of-order . . . . .	109
4.35 Speculazione . . . . .	110
4.35.1 Perché fare una pianificazione dinamica? . . . . .	110
4.36 Funzionano i multiple issue? . . . . .	110
4.37 Efficienza energetica e pipeline avanzate . . . . .	110
4.38 ARM Cortex-A53 vs Intel Core i7 920 . . . . .	111
4.38.1 ARM Cortex-A53 pipeline . . . . .	111
4.38.2 ARM Cortex-A53 performance . . . . .	113
4.38.3 Intel Core i7 920 pipeline . . . . .	113
4.38.4 Intel Core i7 920 performance . . . . .	115



<b>5 Memorie e Registri</b>	<b>116</b>
5.0.1 Load/Store . . . . .	116
5.0.2 Idea: due memorie . . . . .	117
5.0.3 Idea migliore . . . . .	118
5.0.4 Principi di località . . . . .	118
5.0.5 Più di un livello: gerarchie della memoria . . . . .	118
5.0.6 Cache . . . . .	119
5.1 Tecnologia delle Memorie . . . . .	119
5.1.1 Static Random Access Memory (SRAM) . . . . .	119
5.1.2 Dynamic Random Access Memory (DRAM) . . . . .	119
5.1.3 Memorie flash . . . . .	120
5.1.4 Disk memory . . . . .	120
5.2 Caches . . . . .	121
5.2.1 Cos'è una cache . . . . .	121
5.2.2 Usare una cache . . . . .	121
5.2.3 Cache mappata direttamente . . . . .	122
5.2.4 Guardare nella cache . . . . .	123
5.2.5 Esercizio - Cache size [#39] . . . . .	123
5.2.6 Algoritmo per leggere $x$ . . . . .	123
5.2.7 Miss rate e Miss penalty . . . . .	124
5.2.8 Approfondiamo la località spaziale . . . . .	124
5.2.9 Mappatura diretta con $s_b > 1$ . . . . .	124
5.2.10 Algoritmo per leggere $x$ ( $n_b \geq 0$ ) . . . . .	125
5.2.11 Esercizio - Misses and Hits con $s_b = 4$ [#49] . . . . .	125
5.2.12 Miss rate/penalty e dimensione dei blocchi . . . . .	127
5.2.13 Miss penalty . . . . .	127
5.2.14 Scrittura . . . . .	128
5.2.15 Esercizio - Actual CPI [#59] . . . . .	129
5.2.16 Una cache reale: processore FastMATH . . . . .	131
5.2.17 Diminuire il miss rate . . . . .	131
5.2.18 Associatività . . . . .	131
5.2.19 Trovare / mettere una $x$ con associatività $> 1$ . . . . .	132
5.2.20 Set di blocchi . . . . .	132
5.2.21 $x$ to $y$ . . . . .	132
5.2.22 Algoritmo per leggere $x$ . . . . .	132
5.2.23 Scegliere un set di blocchi (LRU) . . . . .	133
5.2.24 Scegliere un set di blocchi (random) . . . . .	133
5.2.25 Esercizio - Misses and hits con associatività [#70] . . . . .	133
5.3 Memorie Virtuali . . . . .	134



# 1 Computer Abstractions and Technology

## 1.1 Introduzione

Negli ultimi decenni, ci sono stati numerosi nuovi computer la cui introduzione sembrava rivoluzionare l'industria informatica; queste rivoluzioni sono state interrotte solo perché qualcun altro ha costruito un computer ancora migliore.

Questa corsa all'innovazione ha portato a progressi senza precedenti dall'inizio dell'informatica alla fine degli anni '40.

*Se l'industria dei trasporti fosse stata al passo con quella dei computer, ad esempio, oggi potremmo viaggiare da New York a Londra in un secondo con un centesimo.* Ogni volta che il costo dell'informatica aumenta di un altro fattore 10, le opportunità per i computer si moltiplicano. Le applicazioni che erano economicamente irrealizzabili diventano improvvisamente pratiche. Nel recente passato, le seguenti applicazioni erano "fantascienza informatica":

- Computer nelle automobili
- Cellulari
- Sequenziamento del genoma umano
- World Wide Web
- Motori di ricerca

## 1.2 Classi tradizionali di Computer

I computer sono utilizzati in tre classi di applicazioni dissimili:

**Personal computer (PC)** Un computer progettato per l'uso da parte di un individuo, generalmente dotato di un display grafico, una tastiera e un mouse.

**Server** Un computer utilizzato per eseguire programmi più grandi per più utenti, spesso contemporaneamente, e in genere si accede solo tramite una rete. Hanno alta capacità, prestazioni, affidabilità. Gamma da piccolo server alle dimensioni di un edificio. **Supercomputer:** Una classe di computer con le prestazioni e i costi più elevati; sono configurati come server e in genere costano da decine a centinaia di milioni di dollari. Allo stato attuale consistono in decine di migliaia di processori con molti terabyte di memoria e rappresentano il vertice della capacità computazionale. Sono solitamente usati per complessi calcoli scientifici e ingegneristici.



**Embedded Computer** : Un computer all'interno di un altro dispositivo utilizzato per eseguire uno predeterminato applicazione o raccolta di software. Applicazioni embedded spesso hanno requisiti unici che combinano minime performance con stringenti limitazioni su costi e potenza. Spesso i computer embedded hanno una tolleranza bassa a fallire, poiché i risultati possono essere devastanti (come il crash del computer di bordo di un aereo). Per esempio: *navigatore dell'auto, altimetro dell'aereo, smartwatch etc.*

### 1.2.1 L'era PostPC

**Personal mobile devices (PMDs)** sono piccoli dispositivi wireless per la connessione a Internet; si basano sulle batterie per l'alimentazione e il software viene installato scaricando le app. Esempi convenzionali sono smartphone e tablet.

**Cloud Computing** si riferisce a grandi raccolte di server (in giganteschi datacenter noti come Warehouse Scale Computers (WSC)) che forniscono servizi su Internet; alcuni provider affittano un numero di server variabile in modo dinamico come utilità.

**Software as a Service (SaaS)** fornisce software e dati come servizio su Internet, di solito tramite un programma leggero come un browser che viene eseguito su dispositivi client locali, invece del codice binario che deve essere installato, e viene eseguito interamente su quel dispositivo. Gli esempi includono la ricerca sul web e i social network.

## 1.3 Definizioni

Decimal term	Abbreviation	Value	Binary term	Abbreviation	Value	% Larger
kilobyte	KB	$10^3$	kibibyte	KiB	$2^{10}$	2%
megabyte	MB	$10^6$	mebibyte	MiB	$2^{20}$	5%
gigabyte	GB	$10^9$	gibibyte	GiB	$2^{30}$	7%
terabyte	TB	$10^{12}$	tebibyte	TiB	$2^{40}$	10%
petabyte	PB	$10^{15}$	pebibyte	PiB	$2^{50}$	13%
exabyte	EB	$10^{18}$	exbibyte	EiB	$2^{60}$	15%
zettabyte	ZB	$10^{21}$	zebibyte	ZiB	$2^{70}$	18%
yottabyte	YB	$10^{24}$	yobibyte	YiB	$2^{80}$	21%

## 1.4 Comprendere le performance di un programma

Le performance di un programma dipendono da una combinazione dell'efficacia degli algoritmi usati nel programma, dei software utilizzati per creare e tradurre il programma in linguaggio macchina, e del computer nell'eseguire queste istruzioni.

- **Algorithm:** determina sia il numero di istruzioni del codice e il numero di operazioni I/O da eseguire.
- **Linguaggio di programmazione, compilatore e architettura:** determina il numero di istruzioni macchina per ogni istruzione del codice.
- **Processore e memoria:** determina quanto velocemente le istruzioni possono essere eseguite.
- **Sistemi di I/O:** determinano quanto velocemente le operazioni di I/O possono essere eseguite.

## 1.5 8 Grandi idee

### 1. Progettare seguendo la legge di Moore

- La legge di Moore afferma che le risorse del circuito integrato raddoppiano ogni 18-24 mesi.
- Poiché i progetti di computer possono richiedere anni, le risorse disponibili per chip possono facilmente raddoppiare o raddoppiare quadruplicare tra l'inizio e la fine del progetto.
- Gli ingegneri informatici devono anticipare dove sarà la tecnologia al termine del progetto piuttosto che progettare per dove inizia.



## 2. Utilizzare l'astratto per semplificare la progettazione

- Una delle principali tecniche di produttività per hardware e software consiste nell'utilizzare le astrazioni per caratterizzare il design a diversi livelli di rappresentazione; i dettagli di livello inferiore sono nascosti per offrire un modello più semplice ai livelli superiori.

## 3. Rendere il caso comune veloce

- Rendere veloce il caso comune tenderà a migliorare le prestazioni meglio rispetto all'ottimizzazione del caso raro. Spesso il caso più comune è anche quello più semplice e quindi anche quello più facile da migliorare.

## 4. Performance tramite parallelismo

- Fin dagli albori dell'informatica, gli ingegneri informatici hanno offerto progetti che ottengono di più prestazioni mediante operazioni di calcolo in parallelo.

## 5. Performance tramite pipeline

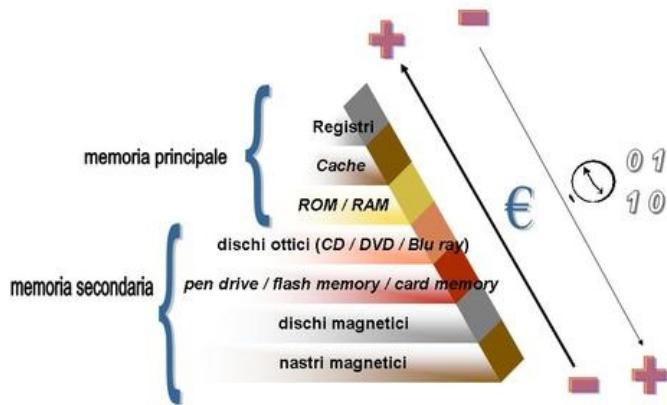
- Un particolare modello di parallelismo.
- Divide le operazioni in piccole fasi, ad es. fetch, decode, execute. Mentre l'istruzione  $i$  viene eseguita,  $i + 1$  viene decodificata,  $i + 2$  viene recuperata.

## 6. Performance tramite predizioni

- Può essere in media più veloce indovinare il caso in cui ci si trova per iniziare a lavorarci subito, piuttosto che aspettare fino a quando non lo si sa con certezza, a patto che il meccanismo per rimediare da una previsione errata non sia troppo costoso e che la previsione sia relativamente accurata.

## 7. Gerarchie di memoria

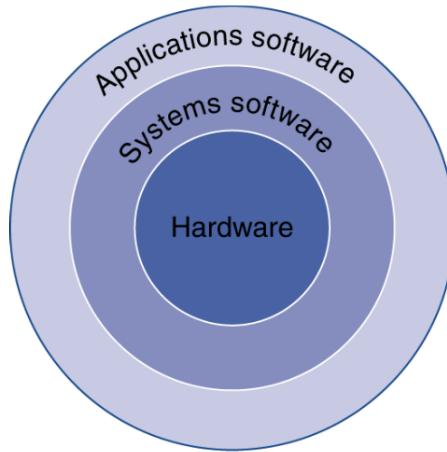
- I programmatori vogliono che la memoria sia veloce, grande ed economica, poiché la velocità della memoria spesso modella prestazioni, la capacità limita la dimensione dei problemi che possono essere risolti e il costo della memoria oggi è spesso la maggior parte del costo del computer.
- Possiamo affrontare queste richieste contrastanti con una gerarchia di memoria, con la memoria più veloce, più piccola e più costosa per bit in cima alla gerarchia e la più lenta, più grande e più economica per bit in fondo.
- Le cache danno l'illusione che la memoria principale sia veloce quasi quanto la cima della gerarchia e grande ed economico quasi quanto il fondo della gerarchia.



#### 8. Affidabilità tramite ridondanza

- I computer non solo devono essere veloci; devono essere affidabili.
- Poiché qualsiasi dispositivo fisico può guastarsi, rendiamo i sistemi affidabili includendo componenti ridondanti che possono subentrare quando si verifica un errore e per aiutare a rilevare i guasti.

### 1.6 Al di sotto del programma



Per passare da una complessa applicazione alle istruzioni primitive ci sono diversi livelli di software che interpretano e traducono operazioni di alto livello in semplici istruzioni macchina. Questi strati di software sono organizzati in modo gerarchico con le applicazioni come anello più esterno e diversi software di sistema posti tra l'hardware fisico e il programma applicativo.



- **Application Software:** tutte le milioni di righe di codice del programma scritte in un linguaggio di programmazione ad alto livello (C, java, etc.).
- **System Software:** Software che fornisce servizi comunemente utili, inclusi sistemi operativi, compiler, loaders e assemblers. Ci sono diversi tipi di programmi di sistema ma i principali sono il compilatore e il sistema operativo.  
*Compilatore:* converte il codice ad alto livello (HLL) in codice macchina.  
*Sistema Operativo:* Programma di supervisione che gestisce le risorse di un computer a vantaggio dei programmi che vengono eseguiti su quel computer:
  - Gestione input/output
  - Gestione della memoria e dell'archiviazione
  - Pianificazione delle attività e condivisione delle risorse
- **Hardware:** Tutte le componenti materiali, processore, memoria, sistemi di I/O

## 1.7 Da un linguaggio ad alto livello al linguaggio hardware

Un **istruzione** è un comando che gli hardware dei computer capiscono e al quale obbediscono. Le istruzioni sono collezioni di bits e possono essere pensate come numeri.

Un **assembler** è un programma che traduce una versione simbolica di istruzioni (assembly language) nella versione in codice binario (machine language).

L'**assembly language** è una rappresentazione simbolica delle istruzioni macchina.

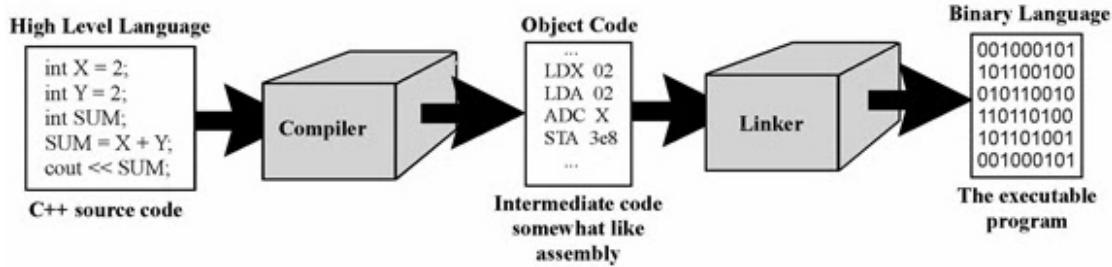
Il **machine language** è la rappresentazione binaria delle istruzioni macchina.

Un **linguaggio ad alto livello** è il livello di astrazione più vicino al linguaggio "quotidiano". Fornisce produttività e portabilità:

- permettono al programmatore di pensare in un linguaggio molto più naturale
- la velocità nel programmare aumenta perché un'idea la si esprime in meno linee di codice
- i programmi diventano indipendenti dalla macchina in cui verranno utilizzati, perché i compilatori e gli assembler possono tradurre un linguaggio di alto livello nelle istruzioni macchina di qualsiasi computer

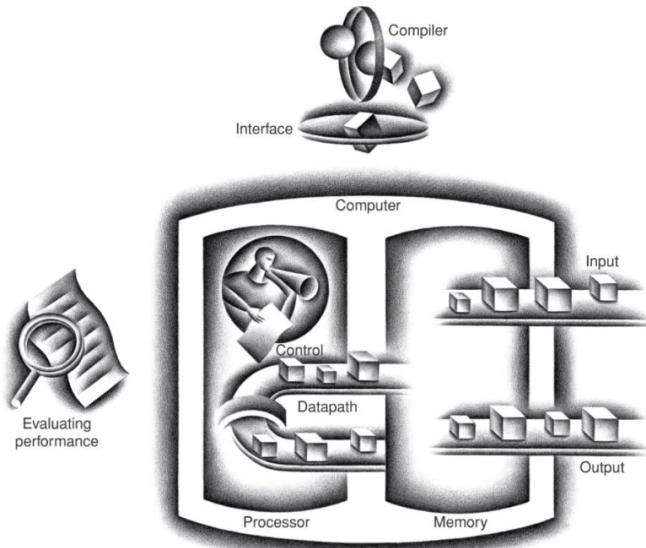
Il **linguaggio assembly** è una rappresentazione testuale delle istruzioni.

La **rappresentazione hardware** è composta solamente da cifre binarie (bit) e le istruzioni e i dati sono codificati.



## 1.8 Componenti di un computer

I 5 componenti classici di un computer sono *input*, *output*, *memoria*, *datapath* e *control*, con gli ultimi due a volte combinati e chiamati processore o central processing unit (CPU). Questa organizzazione è indipendente dalla tecnologia hardware: puoi inserire ogni pezzo di ogni computer, passato e presente, in una di queste cinque categorie.



La central processing unit (CPU) viene anche chiamato processore. È la parte attiva del computer, che contiene il datapath e il control e che svolge operazioni aritmetiche, segnala l'attivazione dei dispositivi I/O e così via.

### 1.8.1 Processore (CPU)

**Datapath**: la componente del processore che svolge le operazioni aritmetiche.

**Control**: la componente del processore che comanda il datapath, la memoria e i dispositivi di I/O in accordo alle istruzioni date dal programma in esecuzione.



### 1.8.2 Memoria

La memoria è l'area di archiviazione in cui vengono conservati i programmi quando sono in esecuzione e che contiene i dati necessari per i programmi in esecuzione.

Esistono diversi tipi di memoria:

- **Dynamic Random Access Memory (DRAM)**: memoria costruita come circuito integrato; fornisce un accesso casuale a qualsiasi posizione. I tempi di accesso sono 50 nanosecondi circa
- **Static Random Access Memory (SRAM)**: memoria costruita come un circuito integrato, ma più veloce e meno densa della DRAM.
- **Memoria cache**: una memoria piccola e veloce che funge da buffer per una memoria più lenta e più grande. Tipicamente SRAM.

SRAM e DRAM sono memorie volatili: vengono utilizzate per contenere dati e programmi mentre sono in esecuzione; ma abbiamo bisogno di memoria non volatile utilizzata per archiviare programmi e dati tra le esecuzioni.

**Memoria primaria e secondaria** Andremo a distinguere la memoria in *main memory* (o memoria primaria) e *secondary memory*.

- **Main memory**: memoria utilizzata per contenere i programmi mentre sono in esecuzione; in genere, nei computer odierni, è costituita da DRAM.
- **Secondary memory**: memoria non volatile utilizzata per memorizzare programmi e dati tra un'esecuzione e l'altra; in genere è costituita da memorie flash nei PMDs (Personal Mobile Devices), SSDs (Solid State Disks) e dischi magnetici (Hard Disk) nei server.

**Disco magnetico**: detto anche Hard Disk. Una forma di memoria secondaria non volatile composta da piatti rotanti rivestiti con un materiale di registrazione magnetico. Poiché sono dispositivi meccanici rotanti, i tempi di accesso sono compresi tra 5 e 20 millisecondi.

**Memoria flash**: una memoria a semiconduttore non volatile. È più economico e più lento della DRAM ma più costoso per bit e più veloce dei dischi magnetici. I tempi di accesso sono di circa 5-50 microsecondi.

### 1.9 Astrazioni

Una delle astrazioni più importanti è l'interfaccia tra l'hardware e il software di livello più basso: l'[architettura del set di istruzioni \(ISA - Instruction set architecture\)](#), o semplicemente architettura, di un computer. L'architettura del set di istruzioni include tutto ciò che i programmatori devono sapere per far funzionare correttamente un programma



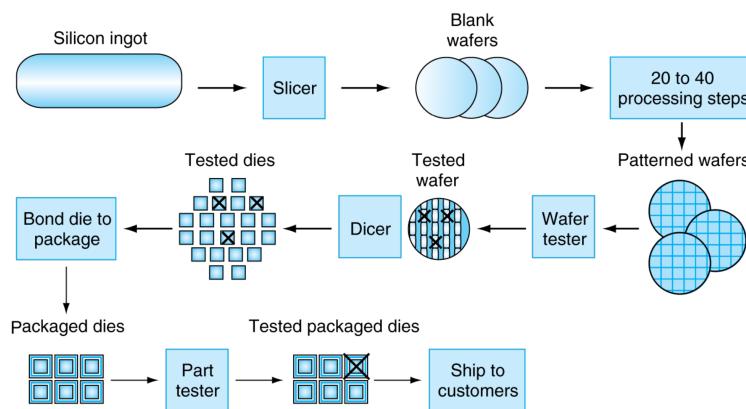
in linguaggio macchina binario, incluse istruzioni, dispositivi I/O e così via. In genere, il sistema operativo incapsula i dettagli dell'esecuzione dell'I/O, dell'allocazione della memoria e di altre funzioni di sistema di basso livello.

**Interfaccia binaria dell'applicazione (ABI - Application binary interface)**: definisce l'interfaccia tra il sistema operativo e le proprie applicazioni a livello di linguaggio macchina. La parte utente del set di istruzioni più l' interfaccia del sistema operativo utilizzate dai programmatori dell'applicazione (ISA + interfaccia software di sistema). Definisce uno standard per la portabilità binaria tra computer.

Si noti che distinguiamo l'architettura del set di istruzioni da un'implementazione dell'architettura: un'implementazione è un hardware che obbedisce all'astrazione dell'architettura. Questa interfaccia astratta consente a molte implementazioni di costi e prestazioni variabili di eseguire software identico.

## 1.10 Tecnologie per costruire processori e memorie

La manifattura di un **circuito integrato** (o **chip**) inizia con il siliceo, una sostanza che non conduce bene l'elettricità essendo un semiconduttore. Con particolari processi chimici è possibile aggiungere materiali al siliceo che permettono a piccole porzioni di trasformarsi o in eccellenti conduttori, o in eccellenti isolanti o in aree che possono comportarsi sia da conduttori che da isolanti sotto specifiche condizioni. Il processo inizia da un lingotto di cristallo di silicio che viene finemente tagliato in **wafers**. I wafer subiscono poi una serie di processi chimici che creano i transistor, porzioni conduttori e isolanti. Piccole imperfezioni nel wafer possono compromettere il suo funzionamento e per questo un unico wafer è composto da chip indipendenti (detti **dies**) che vengono separati e testati, in modo che i chip difettati vengano scartati.



**Yield (rendimento)**: la percentuale di piastre buone rispetto al numero totale di chip (dies) sul wafer.

$$Costo\ chip = \frac{Costo\ wafer}{Chip\ sul\ wafer \times Yield}$$

$$Chip\ sul\ wafer \approx \frac{Area\ wafer}{Area\ chip}$$

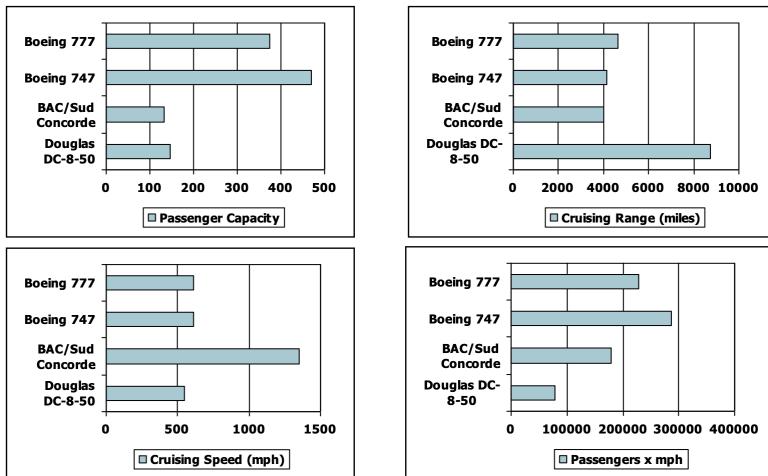


$$Yield = \frac{1}{\left[1 + \left(Difetti per area \times \frac{Area chip}{2}\right)\right]^2}$$

La seconda equazione è un'approssimazione perché non tiene conto di dover sottrarre l'area vicino al bordo circolare del wafer che taglia parte di alcuni chip. Non c'è una relazione lineare con area e tasso di difetto. Il costo e l'area del wafer sono fissi, il tasso di difetto è determinato dal processo di fabbricazione e l'area del singolo chip è determinata dall'architettura e dalla progettazione dei circuiti.

## 1.11 Performance

Quando diciamo che un computer ha prestazioni migliori di un altro, cosa intendiamo? Quale dei seguenti aeroplani ha le migliori prestazioni? Prima bisogna definire cosa si intende per prestazione, infatti si possono misurare in modo diverso le performance.



### 1.11.1 Tempo di risposta e Produttività

Se stai eseguendo un programma su due diversi computer desktop, diresti che quello più veloce è il computer desktop che esegue il lavoro per primo. Se stai eseguendo un data center con diversi server che eseguono lavori inviati da molti utenti, diresti che il computer più veloce è stato quello che ha completato il maggior numero di lavori durante la giornata.

Il **Response time** anche chiamato **execution time** è il tempo totale richiesto dal computer per completare un'attività, inclusi gli accessi al disco e alla memoria, le attività di I/O, l'effetto del sovraccarico del sistema operativo, il tempo di esecuzione della CPU e così via.

Il **throughput** anche detto **bandwidth** è un'altra misura delle prestazioni: è il numero di attività completate per unità di tempo.



In che modo vengono influenzati l'execution time e il throughput? Sostituendo il processore con una versione più veloce? Aggiungendo più processori?

### 1.11.2 Performance relative

Definiamo la performance come:

$$\text{Performance}_X = \frac{1}{\text{Execution time}_X}$$

Con due computer  $\text{Performance}_X > \text{Performance}_Y$  che equivale a scrivere  $\text{Execution time}_Y > \text{Execution time}_X$

Posso dire che  $X$  è  $n$  volte più veloce di  $Y$  se

$$\frac{\text{Performance}_X}{\text{Performance}_Y} = \frac{\text{Execution time}_Y}{\text{Execution time}_X} = n$$

### 1.11.3 Misurare le Performance

Il tempo è la misura delle prestazioni del computer: il computer che esegue la stessa quantità di lavoro nel minor tempo è il più veloce. Tuttavia, il tempo può essere definito in modi diversi, a seconda di ciò che contiamo. La più semplice definizione di tempo è chiamata **wall clock**, **response time** o **elapsed time**. Con questi termini definiamo il tempo totale per completare un'attività, inclusi accessi al disco, accessi alla memoria, attività di input/output (I/O), sovraccarico del sistema operativo: tutto.

Tuttavia i computer spesso sono condivisi e un processore potrebbe lavorare su diversi programmi simultaneamente. In questi casi il sistema potrebbe provare a ottimizzare il throughput piuttosto che provare a minimizzare il tempo trascorso su un dato programma. Il **CPU execution time** anche detto **CPU time** è il tempo effettivo impiegato dalla CPU per l'elaborazione di un'attività specifica. Non include il tempo trascorso in attesa di I/O o nell'esecuzione di altri programmi. Il tempo della CPU può essere ulteriormente suddiviso in

- **user CPU time**: il tempo CPU impiegato nel programma stesso.
- **system CPU time**: il tempo della CPU impiegato nel sistema operativo per eseguire attività per conto del programma.

Programmi diversi sono influenzati in modo diverso dalle prestazioni della CPU e del sistema.

### 1.11.4 CPU Clocking

Tutti i computer sono costruiti utilizzando un clock che determina quando si verificano eventi nell'hardware. Il periodo di clock (**clock period**) è il tempo per un ciclo di clock completo (ad es. 250 picosecondi o 250 ps). La frequenza di clock (**clock rate**) è l'inverso

del periodo di clock (ad es. 4 gigahertz o 4 GHz). Una semplice formula mette in relazione i cicli di clock e il tempo di ciclo di clock con il tempo della CPU:

$$CPU \text{ execution time} = CPU \text{ clock cycles} \times Clock \text{ cycle time}$$

$$CPU \text{ execution time} = \frac{CPU \text{ clock cycles}}{Clock \text{ rate}}$$

Questa formula chiarisce che il progettista hardware può migliorare le prestazioni riducendo il numero di cicli di clock richiesti per un programma o la durata del ciclo di clock. Il progettista hardware deve spesso rinunciare a una maggiore frequenza di clock per ridurre i cicli necessari a un programma.

### 1.11.5 Performance delle istruzioni

Le precedenti equazioni delle prestazioni non includevano alcun riferimento al numero di istruzioni necessarie per il programma. Tuttavia visto che il computer deve eseguire le istruzioni generate dal compilatore, il tempo di esecuzione deve dipendere dal numero di istruzioni in un programma. Un modo per pensare al tempo di esecuzione è che sia uguale al numero di istruzioni eseguite moltiplicato per il tempo medio per istruzione.

$$CPU \text{ clock cycles} = Instructions \text{ for a program} \times Avg. \text{ clock cycles per instruction}$$

**Clock cycles per instruction (CPI)**: numero medio di cicli di clock per istruzione per un programma o un frammento di programma. Poiché istruzioni diverse possono richiedere tempi diversi a seconda di ciò che fanno, CPI è una media di tutte le istruzioni eseguite nel programma.

### 1.11.6 CPU Performance - Equazione classica

Ora possiamo scrivere l'equazione base delle performance in termini di numero di istruzioni, CPI e tempo del ciclo di clock ottenendo:

$$CPU \text{ time} = Instruction \text{ count} \times CPI \times Clock \text{ cycletime}$$

o in termini di frequenza

$$CPU \text{ time} = \frac{Instruction \text{ count} \times CPI}{Clock \text{ rate}}$$

### 1.11.7 CPI nel dettaglio

Se classi di istruzioni diverse richiedono un numero di cicli diverso abbiamo che

$$Clock \text{ cycles} = \sum_{i=1}^n (CPI_i \times Instruction \text{ Count}_i)$$

Media pesata del CPI:

$$CPI = \frac{\text{Clock Cycles}}{\text{Instruction Count}} = \sum_{i=1}^n \left( CPI_i \times \frac{\text{Instruction Count}_i}{\text{Instruction Count}} \right)$$

dove definiamo  $\frac{\text{Instruction Count}_i}{\text{Instruction Count}}$  come frequenza relativa.

### 1.11.8 Riassunto

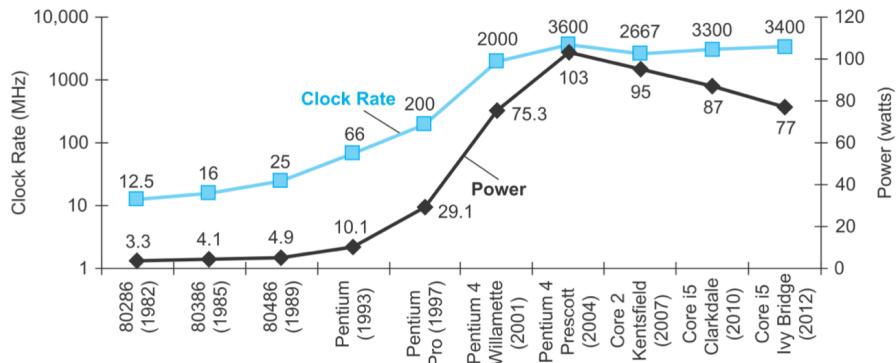
$$Time = Seconds/Program = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}}$$

Le performance di un programma dipendono da:

- **Algoritmo:** determina il numero di istruzioni del programma sorgente eseguite e quindi il numero di istruzioni del processore eseguite. Può anche influenzare il CPI favorendo istruzioni più lente o più veloci.
- **Linguaggio di programmazione:** influisce sul conteggio delle istruzioni, poiché le istruzioni da un certo linguaggio vengono tradotte in istruzioni macchina, che determinano il conteggio delle istruzioni. Può anche influenzare il CPI a causa delle sue caratteristiche. Per esempio un forte supporto per l'astrazione dei dati richiede chiamate indirette, che utilizzano istruzioni CPI più elevate.
- **Compilatore:** influisce sia sul conteggio delle istruzioni che sui cicli medi per istruzione, poiché determina la traduzione delle istruzioni dal linguaggio di partenza a istruzioni per il computer.
- **ISA:** influisce su tutti e tre gli aspetti delle prestazioni della CPU, poiché influisce sulle istruzioni necessarie per una funzione, il numero di cicli per ciascuna istruzione e la frequenza di clock complessiva del processore.

## 1.12 The power wall

La tabella mostra l'aumentare della frequenza di clock e potenza di microprocessori Intel nell'arco di trent'anni. La ragione per cui crescono assieme è che sono correlati, e il motivo per il loro rallentamento è dovuto al raggiungimento del limite di potenza per raffreddare i microprocessori.



### 1.12.1 Dissipazione di potenza in CMOS

Nei CMOS (Complementary Metal-Oxide Semiconductor) la fonte primaria di consumo di energia è la cosiddetta energia dinamica, ovvero l'energia che viene consumata quando i gate cambiano stato da 0 a 1 e viceversa.

Per la transizione  $0 \rightarrow 1 \rightarrow 0$  o  $1 \rightarrow 0 \rightarrow 1$ :

$$Energy \propto Capacitive\ load \times Voltage^2$$

Per una singola transizione  $0 \rightarrow 1$  oppure  $1 \rightarrow 0$

$$Energy \propto \frac{1}{2} \times Capacitive\ load \times Voltage^2$$

La potenza richiesta per porta è semplicemente il prodotto dell'energia di una transizione e la frequenza delle transizioni, che dipende dalla frequenza di clock:

$$Power \propto \frac{1}{2} \times Capacitive\ load \times Voltage^2 \times Frequency\ switched$$

Come possono le frequenze di clock crescere di un fattore 1000 mentre la potenza è aumentata solo di un fattore 30? Energia e potenza sono state ridotte abbassando la tensione, che si verificava ad ogni nuova generazione di tecnologia, passando da 5V fino a valori inferiori a 1V.

### 1.12.2 Problemi moderni nella dissipazione dell'energia

Il problema moderno è che un ulteriore abbassamento della tensione sembra rendere i transistor poco affidabili per via delle correnti parassite (leakage current). Sebbene l'energia dinamica sia la principale fonte di consumo di energia nei CMOS, il consumo di energia statica si verifica a causa della corrente di dispersione che scorre anche quando un transistor è spento. Nei server, le perdite sono in genere responsabili del 40% del consumo energetico.

Aumentando il numero di transistor si aumenta la dissipazione di potenza, anche se i transistor sono sempre spenti. Una varietà di tecniche di progettazione e innovazioni tecnologiche vengono implementate per controllare le perdite, ma è difficile abbassare ulteriormente la tensione.



## 1.13 Multiprocessori

I microprocessori multicore posseggono più di un processore per chip e offrono vantaggi maggiori sul numero di operazioni per unità di tempo (throughput) che sui tempi di risposta. La tecnica della **pipeline** permette di velocizzare l'esecuzione dei programmi facendo eseguire più operazioni in contemporanea, un esempio di *instruction-level parallelism* dove la natura parallela dell'hardware viene astratta, così che il programmatore può pensare all'hardware come se eseguisse le operazioni in modo sequenziale.

È però richiesta una programmazione esplicitamente parallela e più complessa. La velocità in un hardware con parallelismo dipende da come si divide il carico di lavori nei vari processori (cores) e che le operazioni ulteriori dovute alla pianificazione e alla coordinazione per il parallelismo non sprechi la capacità di calcolo aggiuntiva.

Per i programmatori è complicato:

- Massimizzare le performance
- Bilanciare il carico dei processori
- Ottimizzare la comunicazione e la sincronizzazione

## 1.14 SPEC CPU Benchmark

Le prestazioni dei processori vengono misurate utilizzando programmi di benchmark che rappresentano presumibilmente tipici carichi di lavoro effettivi.

SPEC (System Performance Evaluation Cooperative) è uno sforzo finanziato e supportato da numerosi fornitori di computer per creare serie standard di benchmark per i moderni sistemi informatici. Sviluppa benchmark per CPU, I/O, Web, ...

SPEC CPU2006 è costituito da una serie di 12 benchmark interi (CINT2006) e 17 a virgola mobile (CFP2006). Va a vedere:

- Tempo trascorso per eseguire una selezione di programmi
- I/O trascurabile, quindi si concentra sulle prestazioni della CPU
- Normalizza rispetto alla macchina di riferimento
- Riassumere come media geometrica dei rapporti di prestazione

$$\sqrt[n]{\prod_{i=1}^n \text{Execution time ratio}_i}$$

## 1.15 SPEC Power Benchmark

Segnala il consumo energetico dei server a diversi livelli di carico di lavoro, suddiviso in incrementi del 10%, in un periodo di tempo.

SPECpower iniziava con un altro benchmark SPEC per le applicazioni aziendali Java



che sforzava i processori, cache, memoria principale, Java virtual machine, compilatore, garbage collector e parti del sistema operativo.

Le prestazioni sono misurate in throughput, come numero di operazioni al secondo. I risultati dei singoli test sono poi fusi in un unico numero chiamato "overall ssj\_ops per watt":

$$\text{overall ssj\_ops per watt} = \frac{\sum_{i=0}^{10} ssj\_ops_i}{\sum_{i=0}^{10} power_i}$$

Dove  $ssj\_ops_i$  è la performance ad ogni incremento del 10% e  $power_i$  è la potenza consumata ad ogni livello di prestazione.

## 1.16 Errore: Bassa potenza al minimo

Target Load %	Performance (ssj_ops)	Average Power (watts)
100%	865,618	258
90%	786,688	242
80%	698,051	224
70%	607,826	204
60%	521,391	185
50%	436,757	170
40%	345,919	157
30%	262,071	146
20%	176,061	135
10%	86,784	121
0%	0	80
Overall Sum	4,787,166	1922
$\sum ssj\_ops / \sum power =$		2490

Guardando il benchmark dello Xeon X5650 notiamo che:

- Al 100% di carico consuma 258 W
- Al 50% di carico consuma 170 W (66%)
- Al 10% di carico consuma 121 W (47%)

Dovremmo progettare hardware per ottenere una "capacità di calcolo proporzionale all'energia". Se i server futuri utilizzassero, ad esempio, il 10% della potenza di picco al 10% del carico di lavoro, potremmo ridurre la bolletta dell'elettricità di datacenter ed emissioni di  $CO_2$ .

### 1.17 Insidia: Legge di Amdahl

Un'insidia comune: migliorare un aspetto di un computer e aspettarsi un miglioramento proporzionale delle prestazioni complessive.

La legge di Amdahl afferma che il tempo di esecuzione del programma dopo aver approntato il miglioramento è

$$T_{improved} = \frac{T_{affected}}{\text{improvement factor}} + T_{uneffected}$$

### 1.18 Fallacia: Computers at low utilization use little power

L'efficienza a bassi carichi di utilizzo è importante. Ad esempio i server WSC della Google lavorano tra il 10% e il 50% il più del tempo e al 100% per meno dell'1% del tempo.

### 1.19 Insidia: MIPS come metrica delle performance

Un comune errore è valutare le performance utilizzando solo uno o due dei parametri (periodo di clock, numero di istruzioni e CPI). L'analisi con 2 dei tre fattori può essere valida solo in limitati contesti.

Un'alternativa all'uso del tempo sono i **MIPS**: Millions of Instructions Per Second. Dato un programma:

$$\text{MIPS} = \frac{\text{Instruction count}}{\text{Execution time} \times 10^6} = \frac{\text{Instruction count}}{\frac{\text{Instruction count} \times \text{CPI}}{\text{Clock rate}} \times 10^6} = \frac{\text{Clock rate}}{\text{CPI} \times 10^6}$$

Non tiene conto delle differenze negli ISA tra computer e delle differenze di complessità tra le istruzioni. Sostituendo il tempo di esecuzione si osserva la relazione tra MIPS e CPI, quest'ultimo varia tra i programmi su una determinata CPU e quindi lo stesso vale per il MIPS.



## 2 Instructions: Language of the Computer

### 2.1 Introduzione

Per comandare l'hardware di un computer, devi parlarne la sua lingua. Le parole della lingua di un computer sono chiamate istruzioni e il suo vocabolario è chiamato set di istruzioni. I linguaggi informatici sono abbastanza simili, più simili ai dialetti regionali che alle lingue indipendenti.

Il set di istruzioni scelto è ARMv8, introdotto nel 2011. Useremo un sottoinsieme di ARMv8, chiamato LEGv8 ("Lessen Extrinsic Garruity"). Questa somiglianza dei set di istruzioni si verifica perché:

- Tutti i computer sono costruiti sulla base di principi di base simili
- Ci sono alcune operazioni di base che tutti i computer devono fornire
- I progettisti di computer hanno un obiettivo comune: il linguaggio dovrebbe facilitare la costruzione del file hardware e il compilatore massimizzando le prestazioni e riducendo al minimo i costi e l'energia.

### 2.2 Operazioni dell'Hardware dei Computer

Ogni computer deve essere in grado di eseguire operazioni aritmetiche. La notazione LEGv8 per sommare le due variabili b e c e per mettere la loro somma in a è

ADD a, b, c

Tutte le operazioni aritmetiche hanno questa forma. Supponiamo di voler inserire la somma di quattro variabili b, c, d ed e nella variabile a:

```
ADD a, b, c // La somma di b e c viene messa in a
ADD a, a, d // La somma di b, c e d è ora in a
ADD a, a, e // La somma di b, c, d ed e è ora in a
```

Richiedere che ogni istruzione abbia esattamente tre operandi è conforme alla filosofia di mantenere l'hardware semplice: hardware per un numero variabile di operandi è più complicato che per un numero fisso.

**Primo principio di progettazione:** La semplicità favorisce la regolarità

#### 2.2.1 Compilare un codice C in LEGv8

Codice C:

$f = (g+h) - (i+j);$

Cosa andrà a produrre il compilatore?



```
ADD t0, g, h // Variabile temporanea t0 che contiene g + h
ADD t1, i, j // Variabile temporanea t1 che contiene i + j
SUB f, t0, t1 // f data da t0 - t1, che equivale a dire (g + h) - (i + j)
```

## 2.3 Operandi dell'Hardware dei Computer

Le istruzioni aritmetiche utilizzano operandi di registro. LEGv8 ha un file di registro a  $32 \times 64\text{-bit}$  utilizzati per i dati a cui si accede di frequente.

- 64-bit data vengono chiamati "doubleword".  $31 \times 64\text{-bit}$  per uso generico utilizzano registri "X" da X0 a X30
- 32-bit data vengono chiamati "word".  $31 \times 32\text{-bit}$  per uso generico utilizzano registri "W" da W0 a W30

Il motivo per il limite dei registri può essere trovato nel

**Secondo principio di progettazione:** Più piccolo è più veloce

Un numero molto elevato di registri può aumentare il tempo di ciclo dell'orologio semplicemente perché i segnali elettronici impiegano più tempo quando devono viaggiare più lontano. I programmatori devono bilanciare il desiderio dei programmi di avere più registri con il desiderio del programmatore di mantenere i cicli di clock veloci. Un altro motivo per non utilizzare più di 32 è il numero di bit richiesti nel formato dell'istruzione.

### 2.3.1 Compilare un codice C utilizzando i registri

Codice C:

$$f = (g+h) - (i+j);$$

Le variabili  $f$ ,  $g$ ,  $h$ ,  $i$ ,  $j$  sono assegnate ai registri X19, X20, X21, X22, X23 rispettivamente.

Cosa andrà a produrre il compilatore LEGv8?

```
ADD X9, X20, X21 // Il registro X9 contiene g + h
ADD X10, X22, X23 // Il registro X10 contiene i + j
SUB X19, X9, X10 // f data da X9 - X10, che equivale a dire (g+h) - (i+j)
```

## 2.4 Operandi di Memoria

L'istruzione di trasferimento dati che copia i dati dalla memoria a un registro è chiamata caricamento. Il formato dell'istruzione di carico è il nome dell'operazione **LDUR**, registro di carico, seguito da il registro da caricare, quindi un registro di base e un offset, una costante utilizzata per accedere alla memoria.

Assumiamo che A è un array di 100 elementi doubleword e il compilatore ha associato le variabili g e h ai registri X20 e X21. L'indirizzo di base di A è in X22.

Compila questa dichiarazione C:



$$g = h + A[8]$$

LDUR X9, [X22, #8] // Il registro temporaneo X9 carica dalla memoria A[8]  
ADD X20, X21, X9 // g = h + A[8]

**Ma c'è un errore!** Il compilatore alloca strutture dati come array e strutture a posizioni in memoria. Il compilatore può quindi inserire l'indirizzo di partenza corretto nelle istruzioni per il trasferimento dei dati. Praticamente tutte le architetture oggi si rivolgono a singoli byte. L'indirizzo di una doubleword corrisponde all'indirizzo di uno degli 8 byte all'interno della doubleword, e gli indirizzi delle doubleword sequenziali differiscono di 8. Quindi,

LDUR X9, [X22, #64] // 8 x 8 = 64

Il computer si dividono in quelli che usano l'indirizzo del byte più a sinistra o "big end" come indirizzo doubleword rispetto a quelli che usano un byte più a destra o "little end". Il LEGv8 può funzionare come big-endian o little-endian. LEGv8 non richiede che le parole siano allineate in memoria, ad eccezione delle istruzioni e dello stack.

L'istruzione complementare al carico è tradizionalmente chiamata store; copia i dati da un registro alla memoria. Il formato di un store è simile a quello di un carico: il nome dell'operazione, **STUR**, il registro dello store, seguito dal registro da memorizzare, dal registro di base e dall'offset per selezionare l'elemento array.

Assumiamo che la variabile h sia associata al registro X21 e che l'indirizzo di base dell'array A sia in X22.

Compila questa dichiarazione C:

$$A[12] = h + A[8]$$

LDUR X9, [X22, #8] // Il registro temporaneo X9 carica dalla memoria A[8]  
ADD X9, X21, X9 // Il registro temporaneo X9 contiene h+A[8]  
STUR X9, [X22, #96] // Store h+A[8] in A[12]

Molti programmi hanno più variabili di quelle che i computer hanno di registri. I registri sono più veloci da accedere rispetto alla memoria e il funzionamento dei dati di memoria richiede carichi e archivi e più istruzioni da eseguire. Il compilatore deve usare il più possibile i registri per le variabili, riversandosi in memoria meno frequentemente. L'accesso ai registri consuma anche molta meno energia rispetto all'accesso alla memoria.

Per ottenere le massime prestazioni e risparmiare energia, un'architettura del set di istruzioni deve avere abbastanza registri e i compilatori devono usare i registri in modo efficiente. L'ottimizzazione del registro è importante!

Supponendo i dati a 64 bit, i registri erano circa 200 volte più veloci (0,25 ns contro 50 ns) e 10.000 volte più efficienti dal punto di vista energetico (0,1 contro 1000 picoJoule) rispetto alla DRAM nel 2015.

Queste grandi differenze hanno portato a cache, che riducono le prestazioni e le penalità energetiche dell'andare nella memoria.



## 2.5 Costanti e Operandi Immediati

Molte volte un programma utilizzerà una costante in un'operazione. Usando le istruzioni che abbiamo visto finora, dovremmo caricare una costante dalla memoria per usarla.

Ad esempio, per aggiungere la costante 4 al registro X22:

```
LDUR X9, [X20, AddrConstant4] // X9 = costant 4
ADD X22, X22, X9 // X22 = X22 + X9 (X9 == 4)
```

Un'alternativa è offrire versioni delle istruzioni aritmetiche in cui un operando è una costante, come **ADDI**, Add Immediate

```
ADDI X22, X22, #4 // X22 = X22 + 4
```

Gli operandi costanti si verificano frequentemente e, includendo le costanti all'interno delle istruzioni aritmetiche, le operazioni sono molto più veloci e consumano meno energia che se le costanti fossero caricate dalla memoria.

## 2.6 Chiarimenti

Sebbene i registri LEGv8 siano larghi 64 bit, il set completo di istruzioni ARMv8 ha due stati di esecuzione: AArch32, in cui i registri sono larghi 32 bit, e AArch64, che ha un registro largo 64 bit. La migrazione dai computer degli indirizzi a 32 bit ai computer degli indirizzi a 64 bit ha lasciato agli scrittori del compilatore una scelta della dimensione dei tipi di dati in C. Chiaramente, i puntatori dovrebbero essere a 64 bit, ma per quanto riguarda gli interi?

Operating System	pointers	int	long int	long long int
Microsoft Windows	64 bits	32 bits	32 bits	64 bits
Linux, Most Unix	64 bits	32 bits	64 bits	64 bits

Useremo long long int per le parole a 64 bit, size\_t per gli indici agli array (garantisce che siano della giusta dimensione, non importa quanto grande sia l'array).

Nel set completo di istruzioni ARMv8, il registro 31 è XZR nella maggior parte delle istruzioni, ma il punto di stack (SP) negli altri. Per evitare confusione, nel registro LEGv8 31 è sempre XZR e SP è sempre registro 28. Il set completo di istruzioni ARMv8 non utilizza l'ADDI mnemonico; usa solo ADD e consente all'assembler di scegliere l'opcode corretto.

## 2.7 Rappresentare le istruzioni nel computer

Le istruzioni sono codificate in codice binario, chiamato codice macchina.  
Mostreremo la versione LEGv8 dell'istruzione:

```
ADD X9, X20, X21
```



Il decimale e la rappresentazione binaria sono

1112	21	0	20	9
10001011000	10101	000000	10100	01001

Questo layout dell'istruzione è chiamato formato di istruzione. Ci sono cinque campi. Tutte le istruzioni LEGv8 sono lunghe 32 bit.

## 2.8 LEGv8 istruzioni in formato R

opcode	Rm	shamt	Rn	Rd
11 bits	5 bits	6 bits	5 bits	5 bits

- **opcode:** Codice operazione
- **Rm:** Secondo registro sorgente
- **shamt:** Shift Amount
- **Rn:** Primo registro sorgente
- **Rd:** Registro di destinazione

### 2.8.1 Buoni compromessi

Un problema si verifica quando un'istruzione ha bisogno di campi più lunghi di quelli del formato R. Ad esempio, l'istruzione del registro di carico deve specificare due registri e una costante. Se l'indirizzo dovesse utilizzare uno dei campi a 5 bit nel formato R, la costante più grande all'interno del carico l'istruzione del registro sarebbe limitata a solo  $2^{5-1}$  o 31. Questa costante viene utilizzata per selezionare elementi da array o strutture dati e spesso deve essere molto più grande di 31.

Abbiamo un conflitto tra il desiderio di mantenere tutte le istruzioni della stessa lunghezza e il desiderio di avere un unico formato di istruzioni.

**Terzo principio di progettazione:** Un buon design necessita di buoni compromessi

Il compromesso scelto dai progettisti LEGv8 è quello di mantenere tutte le istruzioni della stessa lunghezza, richiedendo così formati di istruzioni distinti per diversi tipi di istruzioni.

- Diversi formati complicano la decodifica, ma consentono istruzioni a 32 bit in modo uniforme
- Mantieni i formati il più simili possibile



## 2.9 LEGv8 istruzioni in formato D

opcode	address	op2	Rn	Rt
11 bits	9 bits	2 bits	5 bits	5 bits

Questo formato viene utilizzato dalle operazioni di load/store

- **address:** Offset costante dal contenuto del registro di base ( $\pm 256$  byte, cioè  $\pm 32$  doubleword)
- **Rn:** Registro base
- **Rt:** Registro di destinazione (load) o di risorsa (store)

LDUR X9, [X22, #64]

Opcode = 1986, Rn = 22, address = 64, Rt = 9

## 2.10 LEGv8 istruzioni in formato I

opcode	immediate	Rn	Rd
10 bits	12 bits	5 bits	5 bits

Questo formato viene utilizzato per le istruzioni immediate: Questo formato viene utilizzato dalle operazioni di load/store

- **Rn:** Registro sorgente
- **Rd:** Registro destinazione

N.B.: I campi immediate sono zero-extended

## 2.11 Memorizzazione dei programmi nei Computer

I computer odierni sono costruiti seguendo due principi chiave:

1. Le istruzioni vengono rappresentate come numeri
2. I programmi sono memorizzati in memoria per essere letti o scritti, come i dati.

Questi principi portano al concetto di programma memorizzato. La memoria può contenere il codice sorgente di un programma di editor, il corrispondente codice macchina compilato, il testo che il programma compilato sta usando e anche il compilatore che ha generato il codice della macchina. I programmi possono funzionare su programmi.

I programmi vengono spesso spediti come file di numeri binari. I computer possono ereditare software già pronto a condizione che siano compatibili con un set di istruzioni esistente.

Tale "compatibilità binaria" spesso porta l'industria ad allinearsi attorno a un piccolo numero di architetture di set di istruzioni.



## 2.12 Operazioni logiche

Logical operations	C operators	Java operators	LEGv8 instructions
Shift Left	«	«	LSL
Shift Right	»	»>	LSR
Bit-by-bit AND	&	&	AND, ANDI
Bit-by-bit OR			OR, ORI
Bit-by-bit NOT	~	~	EOR, EORI

La prima classe di tali operazioni si chiama **SHIFT**. Spostano tutti i bit in una doubleword a sinistra (spostamento logico a sinistra LSL) o a destra (spostamento logico a destra LSR), riempiendo i bit svuotati con 0.

00000000 00000000 00000000 00000000 00000000 00000000 00000000 00001001<sub>two</sub> = 9<sub>ten</sub>

LSL X11,X19,#4 // reg X11 = reg X19 << 4 bits

00000000 00000000 00000000 00000000 00000000 00000000 10010000<sub>two</sub> = 144<sub>ten</sub>

opcode	Rm	shamt	Rn	Rd
1691	0	4	19	11

Un'altra operazione utile è l'**AND**. È un'operazione bit per bit che lascia un 1 nel risultato solo se entrambi i bit degli operandi sono 1.

AND X9,X10,X11 // reg X9 = reg X10 & reg X11  
X11 00000000 00000000 00000000 00000000 00000000 00001101 11000000<sub>two</sub>  
X10 00000000 00000000 00000000 00000000 00000000 00111100 00000000<sub>two</sub>  
X9 00000000 00000000 00000000 00000000 00000000 00001100 00000000<sub>two</sub>

**OR** è un'operazione bit-by-bit che pone un 1 nel risultato se uno dei bit operandi è un 1.

X9 00000000 00000000 00000000 00000000 00000000 00111101 11000000<sub>two</sub>

**NOT** prende un operando e mette un 1 nel risultato se un bit di operando è uno 0 e viceversa. In linea con il formato a tre operandi, i progettisti di ARMv8 hanno deciso di includere le istruzioni **EOR** (OR esclusivo) invece di NOT. Poiché l'EOR crea uno 0 quando i bit sono gli stessi e un 1 se sono diversi

EOR X9,X10,X12 // NOT operation  
X10 00000000 00000000 00000000 00000000 00000000 00001101 11000000  
X12 11111111 11111111 11111111 11111111 11111111 11111111 11111111  
X9 11111111 11111111 11111111 11111111 11111111 11111111 11110010 00111111



Le costanti sono utili nelle operazioni logiche e nelle operazioni aritmetiche, quindi LEGv8 fornisce anche le istruzioni AND immediate (ANDI), OR immediate (ORRI) ed EOR immediate (EORI).

## 2.13 Differenze tra ARMv8 e LEGv8

- I campi immediati per ANDI, ORRI ed EORI del set completo di istruzioni ARMv8 non sono semplici immediati a 12 bit. ARMv8 ha l'insolita caratteristica di utilizzare un algoritmo complesso per la codifica di valori immediati. Qualche piccola costante (e.g. 1, 2, 3, 4 e 6) sono valide, mentre altre (e.g. 0, 5) non lo sono. LEGv8 utilizza semplicemente normali immediate a 12 bit come si trova in AD-DI. Questa differenza significa che EORI X1, X1, #5 è legale per LEGv8 ma non ARMv8.
- A differenza di quasi tutte le altre architetture informatiche, ARMv8 consente di spostare un registro come parte di un'istruzione aritmetica o logica. Poiché questa combinazione è insolita nelle architetture dei computer e non frequentemente generata dai compilatori, LEGv8 tratta gli shift come istruzioni separate. L'opcode utilizzato è quello di UBFM (unsignedbitfield move), ma la codifica dei campi Rm e shamt è stata semplificata.

## 2.14 Istruzioni per prendere decisioni

Il processo decisionale è comunemente rappresentato nei linguaggi di programmazione utilizzando l'istruzione if, a volte combinata con istruzioni ed etichette go to.

LEGv8 include due istruzioni decisionali, simili a una dichiarazione if con un go to.

**CBZ register, L1**

Significa andare all'istruzione etichettata L1 se il valore nel registro è uguale a zero.

**CBZ** sta per confrontare e ramificare se zero.

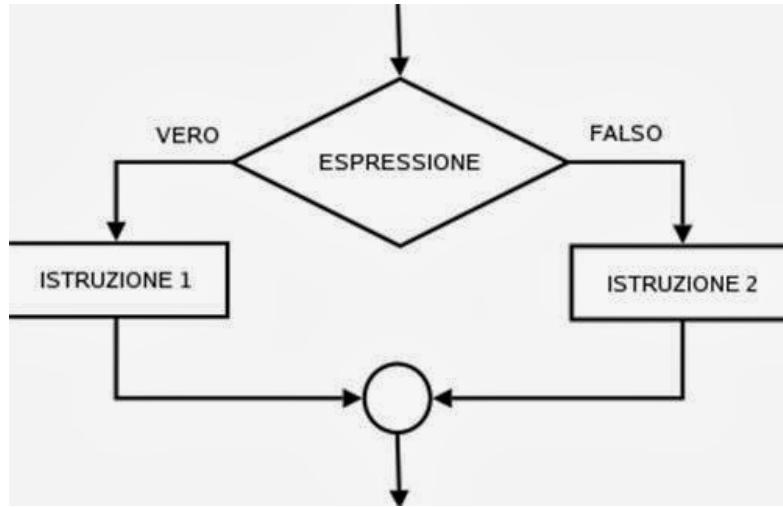
**CBNZ register, L1**

Significa andare all'istruzione etichettata L1 se il valore nel registro non è uguale a zero.

**CBNZ** sta per confronto e ramo se non zero.



## 2.15 Compilazione degli *if – else*



f, g, h, i e j sono variabili che corrispondono ai cinque registri da X19 a X23.  
Qual è il codice LEGv8 compilato per questa istruzione if in C?

```
if (i == j)
    f = g + h;
else
    f = g - h;
```

in LEGv8:

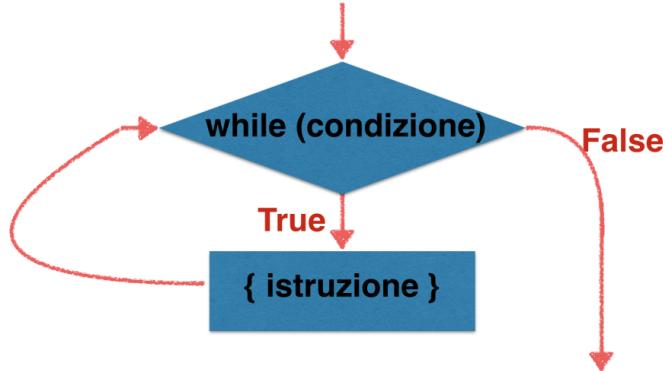
```
SUB X9, X22, X23 // X9 = i - j
CBNZ X9, Else // go to Else se i è diverso da j
ADD X19, X20, X21 // f = g + h
B Exit // go to Exit
```

```
Else:
    SUB X19, X20, X21 // f = g - h
```

Exit:



## 2.16 Compilazione dei cicli *while*



Ecco un ciclo tradizionale in C:

```
while (save[i] == k)
    i += 1;
```

Supponiamo che *i* e *k* corrispondano ai registri X22 e X24 e la base del salvataggio dell'array sia in X25.

Qual è il codice di assemblaggio LEGv8 corrispondente a questo codice C?

Loop:

```
LSL X10, X22, #3 // Temp reg X10 = i * 8
ADD X10, X10, X25 // X10 = indirizzo di save[i]
LDUR X9, [X10, #0] // Temp reg X9 = save[i]
SUB X11, X9, X24 // X11 = save[i] - k
CBNZ X11, Exit // go to Exit se save[i] è diverso da k
ADDI X22, X22, #1 // i = i+1
B Loop // go to Loop
```

Exit:

## 2.17 Blocchi base

Un blocco di base è una sequenza di istruzioni con:

- Nessun salto (branch) (tranne eventualmente alla fine)
- Nessun branch target o branch label (tranne eventualmente all'inizio).

Un compilatore identifica i blocchi di base per l'ottimizzazione. Un processore avanzato può accelerare l'esecuzione di blocchi di base.

## 2.18 Altre operazioni condizionali

L'insieme dei confronti è: minore di ( $<$ ), minore uguale di ( $\leq$ ), maggiore di ( $>$ ), maggiore uguale di ( $\geq$ ), uguale ( $=$ ) e diverso ( $\neq$ ).

I confronti devono anche riguardare la dicotomia tra numeri con segno e senza segno. I codici di condizione o le flag vengono utilizzati per gestire tutti questi casi:

- Negativo (N): il risultato settato dalla condizione ha un 1 nel bit più significativo
- Zero (Z): il risultato settato dalla condizione è 0
- Overflow (V): il risultato settato dalla condizione è in overflow
- Carry (C): Il risultato settato dalla condizione ha avuto un carry out del bit più significativo o un borrow nel bit più significativo

Sono impostati da un numero limitato di operazioni quando il codice delle condizioni è attivo: ADD, ADDI, AND, ANDI, SUB e SUBI.

Nel linguaggio assembly LEGv8, bisogna aggiungere una S alla fine di una di queste istruzioni se vuoi impostare una flag per un confronto: ADDS, ADDIS, ANDS, ANDIS, SUBS e SUBIS.

I salti condizionali (scritti come B.cond) usano combinazioni dei codici delle condizioni. Per salti condizionati da un confronto si utilizzano i seguenti codici:

- B.EQ (equal)
- B.NE (not equal)
- B.LT (less than, signed), B.LO (less than, unsigned)
- B.LE (less than or equal, signed), B.LS (less than or same, unsigned)
- B.GT (greater than, signed), B.HI (greater than, unsigned)
- B.GE (greater than or equal, signed), B.HS (greater than or same, unsigned)

	Signed numbers		Unsigned numbers	
Comparison	Instruction	CC Test	Instruction	CC Test
=	B.EQ	Z=1	B.EQ	Z=1
$\neq$	B.NE	Z=0	B.NE	Z=0
$<$	B.LT	N!=V	B.LO	C=0
$\leq$	B.LE	$\sim(Z=0 \& N=V)$	B.LS	$\sim(Z=0 \& C=1)$
$>$	B.GT	(Z=0 & N=V)	B.HI	(Z=0 & C=1)
$\geq$	B.GE	N=V	B.HS	C=1



### 2.18.1 Esempio

Codice C:

```
if(a > b)
    a += 1;
```

a in X22, b in X23

Il corrispondente codice Assembly LEGv8:

```
SUBS X9, X22, X23 // uso subtract per fare la comparazione
B.LE Exit
ADDI X22, X22, #1
```

Exit: ...

## 2.19 Supporto delle Procedure nei Computer

**Procedura:** Una subroutine memorizzata che esegue un'attività specifica in base ai parametri con cui viene fornita. Le procedure sono un modo per implementare l'astrazione nel software.

Nell'esecuzione di una procedura, il programma deve seguire questi sei passaggi:

1. Metti i parametri in un luogo in cui la procedura possa accedervi.
2. Trasferisci il controllo alla procedura.
3. Acquisisci le risorse di archiviazione necessarie per la procedura.
4. Esegui il compito desiderato.
5. Metti il valore del risultato in un luogo in cui il programma chiamante può accedervi.
6. Restituire il controllo al punto di origine, poiché una procedura può essere chiamata da più punti in un programma.

### 2.19.1 Supporto del LEGv8 alle procedure

Il software LEGv8 segue la seguente convenzione per la procedura che chiama l'assegnazione dei suoi 32 registri:

- X0 - X7: Otto registri di parametri in cui passare parametri o restituire valori.
- LR (X30): un registro dell'indirizzo di ritorno per tornare al punto di origine.

Il linguaggio assembly LEGv8 include un'istruzione solo per le procedure: **branch-and-link (BL)**



#### BL ProcedureAddress

Salta in un indirizzo e contemporaneamente salva l'indirizzo delle seguenti istruzioni, cioè l'indirizzo di ritorno nel registro LR (X30).

Per supportare il ritorno da una procedura, le architetture come LEGv8 usano l'istruzione del branch register (BR) che significa un salto incondizionato all'indirizzo specificato in un registro:

BL LR

Implicita nell'idea del programma memorizzato è la necessità di avere un registro per tenere l'indirizzo dell'istruzione corrente in esecuzione, il contatore del programma.

#### 2.19.2 Usando più registri

Supponiamo che un compilatore abbia bisogno di più registri per una procedura rispetto agli otto registri degli argomenti. Tutti i registri necessari al chiamante devono essere ripristinati ai valori che contenevano prima dell'invocazione della procedura.

La struttura dati ideale per versare i registri è uno stack, una coda last-in-first-out. Uno stack ha bisogno di un puntatore, lo **stack pointer**, all'indirizzo allocato più di recentemente nello stack. Il puntatore dello stack (**SP**), che è solo uno dei 32 registri, viene regolato da una doubleword per ciascun registro che viene salvato o ripristinato.

**N.B.** SP è in X28 in LEGv8, ma in X31 in ARMv8.

Posizionare i dati sullo stack è chiamato **push** e la rimozione dei dati dallo stack è chiamata **pop**.

Storicamente lo stack "cresce" dagli indirizzi più alti a quelli più bassi. Questa convenzione significa che si immettono i valori sullo stack sottraendo dallo stack pointer. L'aggiunta allo stack pointer riduce lo stack, pulendo così i valori dello stack.

#### 2.19.3 Compilazione di una procedura che non chiama un'altra procedura

```
long long int leaf_example (long long int g, long long int h, long long int i,  
long long int j)  
{  
    long long int f;  
  
    f = (g + h) - (i + j);  
    return f;  
}
```

Le variabili di parametro g, h, i e j corrispondono ai registri degli argomenti X0, X1, X2 e X3, f corrisponde a X19.

In Assembly:



leaf\_example:

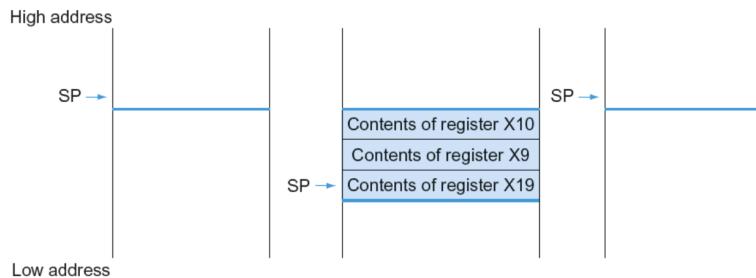
```
SUBI SP, SP, #24 // crea lo spazio nello stack per 3 oggetti
STUR X10, [SP,#16] // salva il registro X10 per utilizzarlo in seguito
STUR X9, [SP,#8] // salva il registro X9 per utilizzarlo in seguito
STUR X19, [SP,#0] // salva il registro X19 per utilizzarlo in seguito

ADD X9, X0, X1 // il registro X9 ora contiene g + h
ADD X10, X2, X3 // il registro X10 ora contiene i + j
SUB X19, X9, X10 // f = X9 - X10, che equivale a (g + h) - (i + j)

ADD X0, X19, XZR // ritorna f (X0 = X19 + 0)

LDUR X19, [SP,#0] // pulisce il registro X19 chiamato in precedenza
LDUR X9, [SP,#8] // pulisce il registro X9 chiamato in precedenza
LDUR X10, [SP,#16] // pulisce il registro X10 chiamato in precedenza
ADDI SP, SP, #24 // ripristina lo stack

BR LR // torna alla routine principale
```



#### 2.19.4 Utilizzo dei registri

Abbiamo usato registri temporanei e abbiamo assunto che i loro vecchi valori dovessero essere salvati e ripristinati. Per evitare di salvare e ripristinare un registro il cui valore non viene mai utilizzato, il software LEGv8 separa 19 dei registri in due gruppi:

- X9 - X17: registri temporanei che non sono conservati dal chiamante su un chiamata alla procedura.
- X19 - X28: registri salvati che devono essere conservati durante una chiamata alla procedura (se utilizzato, il chiamante salva e li ripristina).

### 2.20 Procedure Non-Leaf

Le procedure Non-Leaf sono procedure che chiamano altre procedure. Per le chiamate nidificate, il chiamante deve salvare sullo stack:

- Il suo indirizzo di ritorno LR
- Qualsiasi argomento (X0 - X7) e temporanei (X9 - X17) necessari dopo la chiamata

Ripristinare lo stack dopo la chiamata

### 2.20.1 Esempio

Codice C:

```
int fact (int n)
{
    if (n < 1)
        return f;
    else
        return n * fact(n-1);
}
```

Argomento n in X0, risultato in X1 (**insolito, dovrebbe essere in X0**)  
Codice Assembly:

```
fact:
    SUBI SP, SP, #16 // crea lo spazio nello stack per 2 oggetti
    STUR LR, [SP,#8] // salva l'indirizzo di ritorno
    STUR X0, [SP,#0] // salva l'argomento n

    SUBIS XZR, X0, #1 // test for n < 1
    B.GE L1 // if n >= 1, go to L1

    ADDI X1, XZR, #1 // ritorna 1
    ADDI SP, SP, #16 // poppa 2 oggetti dallo stack
    BR LR // salta al chiamante

L1:
    SUBI X0, X0, #1 // n >= : l'argomento diventa (n - 1)
    BL fact // chiama fact con (n - 1)

    LDUR X0, [SP, #0] // reimposta l'argomento n
    LDUR LR, [SP, #8] // ripulisce il registro di ritorno
    ADDI SP, SP, #16 // ripristina lo stack
    MUL X1, X0, X1 // ritorna n * fact(n - 1)
    BR LR // salta al chiamante
```



## 2.20.2 Conservato / Non conservato tra le chiamate

Preserved	Not preserved
Saved registers: X19-X27	Temporary registers: X9-X15
Stack pointer register: X28 (SP)	Argument/Result registers: X0-X7
Frame pointer register: X29 (FP)	
Link Register (return address): X30 (LR)	
Stack above the stack pointer	Stack below the stack pointer

## 2.21 Gestione delle variabili C

Una variabile C è generalmente una posizione nello storage e la sua interpretazione dipende sia dal suo tipo che dalla sua classe di archiviazione. I tipi di esempio includono numeri interi e caratteri. Il linguaggio C ha due classi di archiviazione: automatica e statica. Le variabili automatiche sono locali a una procedura e vengono eliminate quando la procedura esce. Esistono invece variabili statiche tra le uscite e le chiamate alle procedure. Vengono dichiarate al di fuori di tutte le procedure o utilizzando la parola chiave static.

Per semplificare l'accesso ai dati statici, alcuni compilatori LEGv8 riservano un registro, chiamato **global pointer**, o GP, ad esempio X27.

Le variabili automatiche vengono salvate nei registri o nello stack.

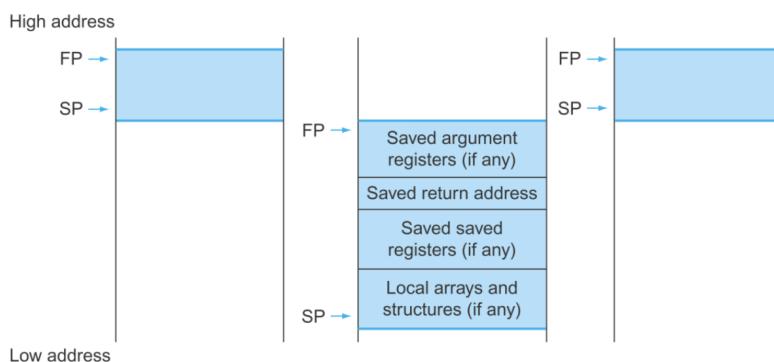
Le variabili statiche vengono salvate nel segmento dei dati statici.

La memoria allocata dinamicamente viene posizionata nell'heap.

## 2.22 Struttura delle procedure

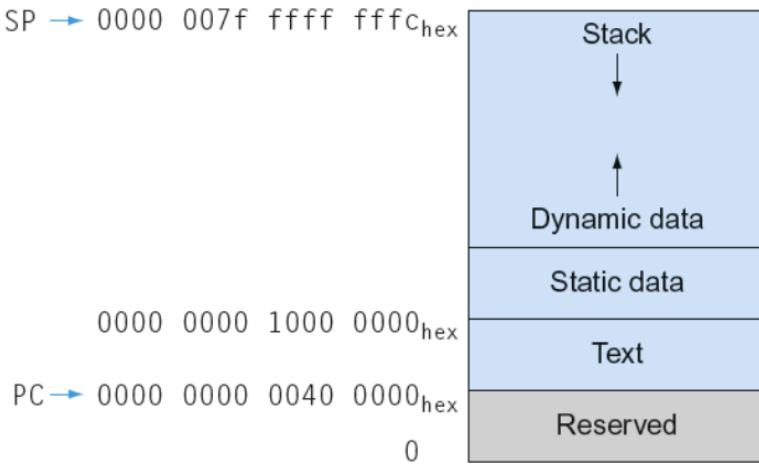
Il segmento dello stack che contiene il registro di salvataggio di una procedura e delle variabili locali viene chiamato **procedure frame** o **activation record**.

Alcuni compilatori ARMv8 usano un **frame pointer** (FP) per puntare alla prima douleword del frame di una procedura. Un puntatore dello stack potrebbe cambiare durante la procedura. Un frame pointer offre un registro di base stabile all'interno di una procedura per i riferimenti alla memoria locale.





## 2.23 Layout della Memoria



- **Text:** Codice del programma
- **Static data:** Variabili globali (e.g. variabili statiche in C, array costanti e stringhe)
- **Dynamic data:** Heap (e.g. malloc in C, new in Java)
- **Stack:** Storage automatico

## 2.24 Sommario delle convenzioni dei registri

Name	Register number	Usage	Preserved on call?
X0-X7	0-7	Arguments/Results	no
X8	8	Indirect result location register	no
X9-X15	9-15	Temporaries	no
X16 (IP0)	16	May be used by linker as a scratch register; other times used as temporary register	no
X17 (IP1)	17	May be used by linker as a scratch register; other times used as temporary register	no
X18	18	Platform register for platform independent code; otherwise a temporary register	no
X19-X27	19-27	Saved	yes
X28 (SP)	28	Stack Pointer	yes
X29 (FP)	29	Frame Pointer	yes
X30 (LR)	30	Link Register (return address)	yes
XZR	31	The constant value 0	n.a.



## 2.25 Operazioni Byte/Halfword

LEGv8 byte/halfword Load/Store

- Load byte: LDURB Rt, [Rn, offset]
- Store byte: STURB Rt, [Rn, offset]
- Load halfword: LDURH Rt, [Rn, offset]
- Store halfword: STURH Rt, [Rn, offset]

LEGv8 word load/store

- Load signed word: LDURSW Rt, [Rn, offset]
- Store word: STURW Rt, [Rn, offset]

## 2.26 Esempio "String Copy"

Codice C:

```
void strcpy (char x[ ], char y[ ])
{
    size_t i;
    i = 0;

    while ( x[i] = y[i] != '\0')
        i += 1;
}
```

Gli indirizzi di base per gli array x e y si trovano in X0 e X1, mentre i si trova in X19.  
Codice LEGv8:

```
strcpy:
    SUBI SP, SP, 8 // push X19
    STUR X19, [SP, #0]
    ADD X19, XZR, XZR // i = 0

L1:
    ADD X10, X19, X1 // X10 = indirizzo di y[i]
    LDURB X11, [X10,#0] // X11 = y[i]
    ADD X12, X19, X0 // X12 = indirizzo di x[i]
    STURB X11, [X12,#0] // y[i] = x[i]
    CBZ X11, L2 // if y[i] == 0 esci
    ADDI X19, X19, #1 // i = i + 1
    B L1 //prossima iterazione
```



L2:

```
LDUR X19, [SP, #0] // restore delle variabili
ADDI SP, SP, 8 // pop di 1 elemento dallo stack
BR LR // ritorna alla funzione chiamante
```

**Nota Bene:** I software per ARMv8 richiedono che lo stack si allinei gli indirizzi delle quadword (16 byte) per avere delle performance migliori. Questa convenzione significa che una variabile char allocata sullo stack occupa 16 byte, anche se ne ha bisogno di meno. Tuttavia, una variabile stringa C o una matrice di byte imballerà 16 byte per quadword.

LEGv8 mantiene tutto a 64 bit rispetto a fornire istruzioni di indirizzo sia a 32 bit che a 64 bit come in ARMv8, il che significa che deve includere STURW (memorizzazione di parole) come istruzione anche se non è specificato in ARMv8 nel linguaggio assembly. ARMv8 utilizza solo STUR con un nome di registro W (registro a 32 bit) invece del nome del registro X (registro a 64 bit).

## 2.27 Operandi Immediati ampi

La maggior parte delle costanti sono piccole e l'immediato a 12 bit è sufficiente.

Per la costante occasionale a 32 bit

- **MOVZ:** muoviti ampio con zeri
- **MOVK:** muoviti ampio con keep

Può impostare qualsiasi 16 bit di una costante in un registro. Il campo a 16 bit da caricare viene specificato aggiungendo LSL e quindi il numero 0, 16, 32 o 48.

The machine language version of MOVZ X9, 255, LSL 16:  

110100101	01	0000 0000 1111 1111	01001
-----------	----	---------------------	-------

 ← **IW format**

Contents of register X9 after executing MOVZ X9, 255, LSL 16:  

0000 0000 0000 0000	0000 0000 0000 0000	0000 0000 1111 1111	0000 0000 0000 0000
---------------------	---------------------	---------------------	---------------------

The machine language version of MOVK X9, 255, LSL 0:  

111100101	00	0000 0000 1111 1111	01001
-----------	----	---------------------	-------

Given value of X9 above, new contents of X9 after executing MOVK X9, 255, LSL 0:  

0000 0000 0000 0000	0000 0000 0000 0000	0000 0000 1111 1111	0000 0000 1111 1111
---------------------	---------------------	---------------------	---------------------

## 2.28 Indirizzi nei salti

Tipo B: B 1000 // go to location 100000 (base 10)

5	100000 <sub>ten</sub>
6 bits	26 bits

Tipo CB: CBNZ X19, Exit // go to Exit if X19 != 0



181	Exit	19
8 bits	19 bits	5 bits

Entrambi gli indirizzi sono PC-relative. Indirizzo = PC + offset (dalle istruzioni)  
Questa forma di indirizzamento dei branch si chiama PC-relative addressing.  
Poiché tutte le istruzioni LEGv8 sono lunghe 4 byte, LEGv8 allunga la distanza del ramo  
avendo PC-relative addressing si riferisce al numero di parole all'istruzione successiva  
invece che al numero di byte.

- Il campo a 19 bit può diramarsi di  $\pm 1$  MB dal PC corrente
- Il campo a 26 bit può diramarsi  $\pm 128$  MB dal PC corrente

La maggior parte dei salti condizionali si trova in una posizione vicina, ma occasionalmente si salta più lontano, più lontano di quanto possa essere rappresentato nei 19 bit dell'istruzione del salto condizionale. In questi casi l'assembler viene in soccorso!

Dato

CBZ X19, L1

Può sostituire il salto condizionale a indirizzi brevi con

CBNZ X19, L2  
B L1

L2:

## 2.29 Sintesi della modalità di indirizzamento LEGv8

Le modalità di indirizzamento delle istruzioni LEGv8 sono le seguenti:

1. Indirizzamento immediato, dove l'operando è una costante all'interno dell'istruzione stessa.



2. Indirizzamento del registro, dove l'operando è un registro.



3. Indirizzamento di base o spostamento, dove l'operando si trova nella posizione di memoria il cui indirizzo è la somma di un registro e di una costante nell'istruzione, ad es. LDUR, STUR





4. Indirizzamento relativo al PC, dove l'indirizzo della filiale è la somma del PC e una costante nel istruzione.



## 2.30 Sintesi della codifica LEGv8

Nome	Campi						
Grandezza campi		6 to 11 bits	5 to 10 bits	5 o 4 bits	2 bits	5 bits	5 bits
Formato R	R	opcode	Rm	shamt		Rn	Rd
Formato I	I	opcode		immediate		Rn	Rd
Formato D	D	opcode	address	op2	Rn	Rt	
Formato B	B	opcode	address				
Formato CB	CB	opcode	address			Rt	
Formato IW	IW	opcode	immediate			Rd	

## 2.31 Parallelismo e Istruzioni: Sincronizzazione

Due processori che condividono un'area di memoria

- P1 scrive, poi P2 legge
- Data race se P1 e P2 non si sincronizzano. Il risultato dipende dall'ordine di accesso

È richiesto il supporto hardware

- Funzionamento atomico della memoria di lettura/scrittura
- Nessun altro accesso alla posizione consentito tra la lettura e la scrittura

Potrebbe essere una singola istruzione. Ad esempio, scambio atomico o una coppia atomica di istruzioni.

Supponiamo di voler costruire un semplice blocco in cui il valore 0 viene utilizzato per indicare che il lucchetto è libero e 1 viene utilizzato per indicare che il lucchetto non è disponibile. Un processore tenta di impostare il blocco facendo uno scambio di 1, che si trova in un registro, con l'indirizzo di memoria corrispondente al blocco. Il valore restituito dall'istruzione di scambio è 1 se qualche altro processore aveva già rivendicato l'accesso e 0 altrimenti. In quest'ultimo caso, anche il valore viene modificato in 1, impedendo a qualsiasi scambio concorrente in un altro processore di recuperare anche uno 0.



### 2.31.1 Sincronizzazione in LEGv8

LEGv8 include un load speciale e un store speciale chiamato:

- Registro esclusivo del carico (LDXR)
- Registro esclusivo di store (STXR)

Queste istruzioni vengono utilizzate in sequenza.

Se il contenuto della posizione di memoria specificata da LDXR viene modificato prima dello STXR si verifica allo stesso indirizzo, quindi l'STXR fallisce e non scrive il valore in memoria.

L'STXR è definito sia per memorizzare il valore di un registro in memoria che per modificare il valore di un altro registro a uno 0 se ha successo e a un 1 se fallisce. STXR specifica tre registri: uno per tenere l'indirizzo, uno per indicare il fallimento o il successo e uno per tenere il valore da memorizzare in memoria.

#### Esempio 1: Atomic swap (to test/set lock variable)

again:

```
LDXR X10, [X20,#0] // Load esclusivo
STXR X23, X9, [X20] // Store esclusivo
CBNZ X9, again // Salta se store fallisce
ADD X23, XZR, X10 // metti il valore caricato in x23
```

#### Esempio 2: Lock

```
ADDI X11, XZR, #1 // Copia il valore
```

again:

```
LDXR X10, [X20,#0] // Load esclusivo per leggere lock
CBNZ X10, again // Controlla se è già a 0
STXR X11, X9, [X20] // Prova a salvare il nuovo valore
BNEZ X9, again // Salta se store fallisce
ADD X23, XZR, X10 // metti il valore caricato in x23
```

unlock:

```
STUR XZR, [X20,#0] // Libera lock scrivendoci 0
```

## 2.32 Tradurre e avviare un programma

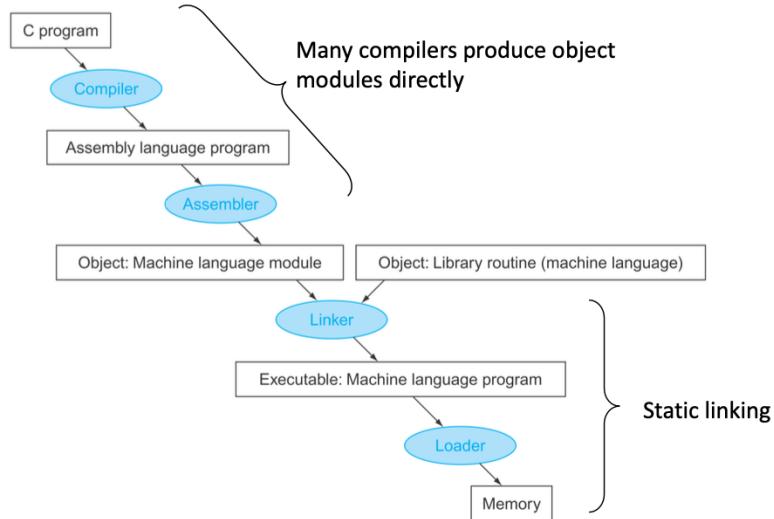
Per identificare il tipo di file, UNIX segue una convenzione di suffissi per i file:

- I file sorgente C si chiamano nome\_file.c
- I file assembly sono nome\_file.s
- I file oggetto si chiamano nome\_file.o



- Le routine di libreria collegate staticamente sono nome\_file.a
- Le routine della libreria collegate dinamicamente sono nome\_file.so
- I file eseguibili sono chiamati nome\_file.out

N.B.: MS-DOS utilizza i suffissi .c, .asm, .obj, .lib, .dll e .exe con lo stesso effetto.



### 2.32.1 Compilatore

Il compilatore trasforma il programma C in un programma di linguaggio assembly. Nel 1975, molti sistemi operativi e assemblatori sono stati scritti in linguaggio assembly perché le memorie erano piccole e i compilatori erano inefficienti. Oggi i compilatori possono produrre programmi di linguaggio assembly quasi quanto prodotto da un programmatore esperto di linguaggio assembly, e talvolta anche migliore per programmi di grandi dimensioni.

### 2.32.2 Assemblatore

L'assemblatore traduce il programma del linguaggio assembly in linguaggio macchina. Crea un file oggetto, che è una combinazione di istruzioni sul linguaggio macchina, dati e informazioni necessarie per inserire correttamente le istruzioni in memoria. L'assemblatore può anche trattare le variazioni comuni delle istruzioni del linguaggio della macchina, ad es. pseudoistruzioni. Ad esempio, LEGv8 accetta

`MOV X9, X10 // Il registro X9 da il registro X10`

- Convertito a: `ORR X9, XZR, X10 // Il registro X9 da 0 o il registro X10`

`CMP X9, X10 // Compara X9 a X10 e setta le condizioni`

- Convertito a: SUBS XZR, X9, X10 // usa X9 - X10 per settare le condizioni AND X9, X10, #15 // Il registro X9 da X10 e #15
- Convertito a: ANDI X9, X10, #15 // Il registro X9 da X10 e #15

Per produrre la versione binaria di ogni istruzione nel programma del linguaggio assembly, l'assemblatore deve determinare gli indirizzi corrispondenti a tutte le etichette.

Gli assemblatori tengono traccia delle etichette utilizzate nei rami e delle istruzioni per il trasferimento dei dati in una tabella dei simboli. La tabella contiene coppie di simboli e indirizzi.

Il file oggetto per i sistemi UNIX contiene in genere sei pezzi distinti:

- L'intestazione del file oggetto (object file header) descrive le dimensioni e la posizione degli altri pezzi del file oggetto.
- Il segmento di testo (text segment) contiene il codice del linguaggio macchina.
- Il segmento di dati statici (static data segment) contiene dati allocati per tutta la vita del programma.
- Le informazioni di trasferimento (relocation information) identificano istruzioni e parole di dati che dipendono dagli indirizzi assoluti quando il programma viene caricato in memoria.
- La tabella dei simboli (symbolic table) contiene le etichette rimanenti che non sono definite, come quelle esterne riferimenti.
- Le informazioni di debug (debugging information) contengono una descrizione concisa di come sono stati compilati i moduli in modo che un debugger possa associare le istruzioni della macchina ai file sorgente C

### 2.32.3 Linker (anche chiamato linker editor)

Il linker è un programma di sistema che combina programmi di linguaggio macchina assemblati in modo indipendente e risolve tutte le etichette non definite in un file eseguibile. Ci sono tre passaggi per il linker:

1. Posiziona simbolicamente codice e moduli di dati in memoria.
2. Determina gli indirizzi dei dati e le etichette delle istruzioni.
3. Patcha sia i riferimenti interni che esterni.

Il linker utilizza le informazioni di trasferimento e la tabella dei simboli in ciascun modulo oggetto per risolvere tutte le etichette non definite (cioè nelle istruzioni dei salti e negli indirizzi dei dati). Se tutti i riferimenti esterni vengono risolti, il linker determina successivamente le posizioni di memoria che ogni modulo occuperà. Quando il linker inserisce



un modulo in memoria, tutti i riferimenti assoluti, cioè gli indirizzi di memoria che non sono relativi a un registro, devono essere spostati per riflettere la sua vera posizione. Il linker produce un file eseguibile che può essere eseguito su un computer. Il file eseguibile è un programma funzionale nel formato di un file oggetto che non contiene riferimenti irrisolti, può contenere tabelle di simboli e informazioni sul debug. Le informazioni sul trasferimento possono essere incluse per il caricatore.

## Esempio

Header del file oggetto			
	Nome	Procedura A	
	Text Size	$100_{hex}$	
	Data Size	$20_{hex}$	
Text Segment	Indirizzo	Istruzione	
	0	LDUR X0, [X27, #0]	
	4	BL 0	
	...	...	
Data Segment	0	(X)	
	...	...	
Informazioni di trasferimento	Indirizzo	Tipo di istruzione	Dipendenza
	0	LDUR	X
	4	BL	B
Tabella simboli	Simbolo	Indirizzo	
	X	-	
	B	-	

	Nome	Procedura B	
	Text Size	$200_{hex}$	
	Data Size	$30_{hex}$	
Text Segment	Indirizzo	Istruzione	
	0	STUR X1, [X27, #0]	
	4	BL 0	
	...	...	
Data Segment	0	(Y)	
	...	...	
Informazioni di trasferimento	Indirizzo	Tipo di istruzione	Dipendenza
	0	STUR	Y
	4	BL	A
Tabella simboli	Simbolo	Indirizzo	
	Y	-	
	A	-	



File header eseguibile		
	Text Size	$300_{hex}$
	Data Size	$50_{hex}$
Text Segment	Indirizzo	Istruzione
	$0000\ 0000\ 0040\ 0000_{hex}$	LDUR X0, [X27, $\#0_{hex}$ ]
	$0000\ 0000\ 0040\ 0004_{hex}$	BL $000\ 00FC_{hex}$
	...	...
	$0000\ 0000\ 0040\ 0100_{hex}$	STUR X1, [X27, $\#20_{hex}$ ]
	$0000\ 0000\ 0040\ 0104_{hex}$	BL $3FF\ FEFC_{hex}$
	...	...
Data Segment	Indirizzo	
	$0000\ 0000\ 1000\ 0000_{hex}$	(X)
	...	...
	$0000\ 0000\ 1000\ 0020_{hex}$	(Y)
	...	...

#### 2.32.4 Loader

Il loader è un programma di sistema che posiziona un programma oggetto nella memoria principale in modo che sia pronto per l'esecuzione.

Il loader segue questi passaggi nei sistemi UNIX:

1. Legge l'intestazione del file eseguibile per determinare le dimensioni dei segmenti di testo e dati.
2. Crea uno spazio di indirizzi abbastanza grande per il testo e i dati.
3. Copia le istruzioni e i dati dal file eseguibile in memoria.
4. Copia i parametri (se presenti) al programma principale sullo stack.
5. Inizializza i registri del processore e imposta il puntatore dello stack nella prima posizione libera.
6. Salta a una routine di avvio che copia i parametri nei registri degli argomenti e nelle chiama la routine principale del programma. Quando la routine principale ritorna, la routine di avvio termina il programma con una chiamata di uscita.

#### 2.32.5 Librerie collegate dinamicamente (DLL)

Abbiamo descritto il tradizionale approccio statico al collegamento delle librerie prima che il programma venga eseguito.

Ha alcuni svantaggi:

- Le routine della libreria diventano parte del codice eseguibile. Se viene rilasciata una nuova versione della libreria che corregge bug o supporta nuovi dispositivi

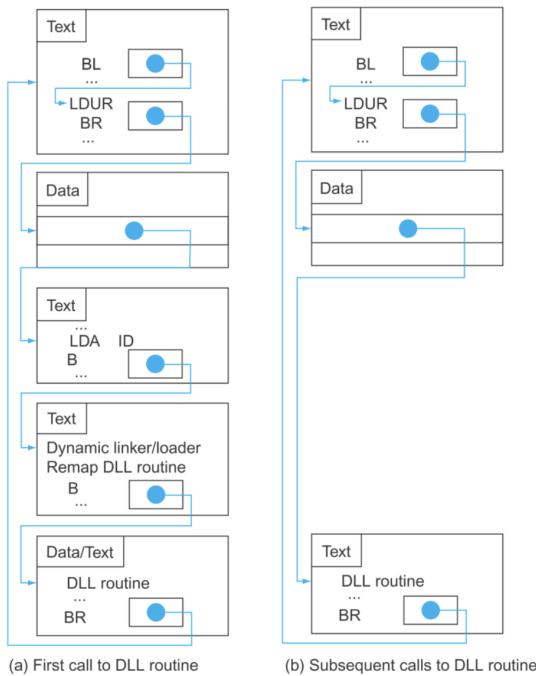


hardware, il programma collegato staticamente continua a utilizzare la vecchia versione.

- Carica tutte le routine nella libreria che vengono chiamate in qualsiasi punto dell'eseguibile, anche se tali chiamate non vengono eseguite. La libreria può essere grande rispetto al programma.

Questi svantaggi portano a librerie collegate dinamicamente (DLLs), in cui le routine di libreria non sono collegate e caricate fino all'esecuzione del programma. Sia il programma che le routine della libreria conservano informazioni aggiuntive sulla posizione delle procedure non locali e sui loro nomi.

Nella versione originale delle DLL, il loader esegue un linker dinamico, utilizzando le informazioni aggiuntive nel file per trovare le librerie appropriate e aggiornare tutti i riferimenti esterni. Ma collega ancora tutte le routine della libreria che potrebbero essere chiamate. In un approccio più efficiente ogni routine è collegata solo dopo essere stata chiamata.



### 2.32.6 Esempio di ordinamento in C

Procedura swap (leaf)

```
void swap(long long int v[ ], long long int k)
{
    long long int temp;
    temp = v[k];
```



```
v[k] = v[k + 1];
v[k + 1] = temp;
}
```

v in X0, k in X1, temp in X9

In Assembly:

```
swap:
    LSL X10, X1, #3 // X10 = k * 8
    ADD X10, X0, X10 // X10 = indirizzo di v[k]
    LDUR X9, [X10, #0] // X9 = v[k]
    LDUR X11, [X10, #8] // X11 = v[k+1]
    STUR X11, [X10, #0] // v[k] = X11
    STUR X9, [X10, #8] // v[k+1] = X9
    BL LR // Ritorna alla routine chiamante
```

Procedura swap, non-leaf chiama swap

```
void swap(long long int v[ ], size_t n)
{
    size_t i, j;
    for( i = 0; i < n; i += 1)
    {
        for (j = i - 1; j >= 0 && v[j] > v[j+1]; j -= 1)
        {
            swap(v, j);
        }
    }
}
```

v in X0, n in X1, i in X19, j in X20

Scheletro del loop esterno:

```
MOV X19,XZR // i = 0

for1stst:
    CMP X19, X1 // Compara X19 a X1 (i to n)
    B.GE exit1 // Vai a exit1 se X19 >= X1

    (body of outer for - loop)

    ADDI X19,X19,#1 // i += 1
    B for1stst // Vai al test del loop esterno

exit1:
```



Scheletro del loop interno:

```
SUBI X10, X19, #1 // j = i - 1

for2tst:
    CMP X20, XZR // Compara X20 a 0
    B.LT exit2 // Vai a exit2 se X20 < 0
    LSL X10, X20, #3 // Reg X10 = j * 8
    ADD X11, X0, X10 // Reg X11 = v + (j * 8)
    LDUR X12, [X11, #0] // Reg X12 = v[j]
    LDUR X13, [X11, #8] // Reg X13 = v[j+1]
    CMP X12, X13 // Compara X12 a X13
    B.LE exit2 // Vai a exit2 se X12 <= X13
    MOV X0, X21 // Primo parametro di swap è v
    MOV X1, X20 // Secondo parametro di swap è j
    BL swap // Chiama la funzione swap
    MOV X1, X22 // Necessario per la comparazione nel primo loop
    SUBI X20, X20, #1 // j -= 1
    B for2tst // Vai al test del loop interno

exit2:
```

Preserva i registri salvati:

```
SUBI SP, SP #40 // Crea lo spazio sullo stack pointer per 5 registri
STUR LR, [SP, #32] // Salva LR sullo stack
STUR X22, [SP, #24] // Salva X22 sullo stack
STUR X21, [SP, #16] // Salva X21 sullo stack
STUR X20, [SP, #8] // Salva X20 sullo stack
STUR X19, [SP, #0] // Salva X19 sullo stack
MOV X21, X0 // Copia il parametro X0 in X21
MOV X22, X1 // Copia il parametro X1 in X22
```

Ripristino dei registri salvati:

```
exit1:
    LDUR X19, [SP, #0] // Ripristino di X19 dallo stack
    LDUR X20, [SP, #8] // Ripristino di X20 dallo stack
    LDUR X21, [SP, #16] // Ripristino di X21 dallo stack
    LDUR X22, [SP, #24] // Ripristino di X22 dallo stack
    LDUR X30, [SP, #32] // Ripristino LR dallo stack
    ADDI SP, SP, #40 // Ripristino stack pointer
```



## 2.33 Array vs Puntatori

L'indicizzazione degli array comporta:

- Moltiplicare l'indice per dimensione dell'elemento
- Aggiunta all'indirizzo di base dell'array

I puntatori corrispondono direttamente agli indirizzi di memoria:

- Può evitare la complessità dell'indicizzazione

<pre>clear1(int array[], int size) { int i; for(i = 0; i &lt; size; i += 1)     array[i] = 0; }</pre>	<pre>clear2(int *array, int size) { int *p; for(p=&amp;array[0]; p&lt;&amp;array[size]; p++)     *p = 0; }</pre>
<pre>MOV X9, XZR // i = 0 loop1: LSL X10, X9, #3 // X10 = i * 8 ADD X11, X0, X10 // X11 = indirizzo // di array[i] STUR XZR, [X11, #0] // array[i] = 0 ADDI X9, X9, #1 // i += 1 CMP X9, X1 // Compara i a size B.LT loop1 // Se i &lt; size vai a loop1</pre>	<pre>MOV X9, X0 // p=indirizzo di array[0] LSL X10, X1, #3 // X10 = size * 8 ADD X11, X0, X10 // X11 = indirizzo // di array[size]  loop2: STUR XZR, [X9, #0] // Memory[p] = 0 ADDI X9, X9, #8 // p += 8 CMP X9,X11 // Compara p a &amp;array[size] B.LT loop2 // Se minore vai a loop2</pre>

La versione array richiede che lo spostamento sia all'interno del ciclo

- Parte del calcolo dell'indice per incrementare i
- Puntatore incrementante c.f.

Il compilatore può ottenere lo stesso effetto dell'uso manuale dei puntatori

- Eliminazione della variabile di induzione
- Meglio rendere il programma più chiaro e sicuro



## 2.34 Istruzioni ARMv7 (32-bit)

In piedi originariamente per la Acorn RISC Machine, in seguito cambiata in Advanced RISC Machine.

ARMv1 è uscito nel 1985 con indirizzi a 32 bit. Molte versioni del set di istruzioni ARM a 32 bit sono uscite nel corso degli anni, culminando con ARMv7 nel 2005. ARMv8 con indirizzi a 64 bit è stato rivelato nel 2013, con grandi differenze.

Somiglianze tra ARMv7 e ARMv8:

- Tutte le istruzioni sono larghe 32 bit per entrambe le architetture.
- L'unico modo per accedere alla memoria è tramite istruzioni di caricamento e archiviazione su entrambe le architetture.

Ma ecco alcune delle differenze:

- ARMv7 e i precedenti set di istruzioni ARM hanno solo 15 registri generici.
- Nessun registro è cablato a 0, quindi ARMv7 e i suoi predecessori hanno bisogno di istruzioni extra per eseguire alcune operazioni che ARMv8 può fare con XZR.
- Il 16esimo registro mancante in ARMv7 e nei suoi predecessori è il program counter (PC).
- Le modalità di indirizzamento ARMv7 non funzionano per tutte le dimensioni dei dati.
- ARMv7 ha istruzioni Load Multiple e Store Multiple. ARMv8 no.
- Piuttosto che il campo immediato è semplicemente una costante, è essenzialmente un input per una funzione che produce una costante. Gli otto campi immediati a bit minimi sono estesi a zero a valore a 32 bit e poi ruotato a destra il numero di bit specificato nei primi quattro bit del campo moltiplicato per due.
- A differenza di ARMv8, i primi set di istruzioni ARM hanno omesso un'istruzione divide.

## 2.35 Il resto del set di istruzioni ARMv8

Classe	Loads / Stores		Operazioni		Branches		Totale	
	AL	ML	AL	ML	AL	ML	AL	ML
Intero	49	145	74	105	-	-	123	250
Floating Point & Int Mul / Div	0	18	63	156	-	-	63	174
SIMD / Vector	16	166	229	371	-	-	245	537
Sistema / Speciali	11	55			-	-	63	95
	-	-	-	-	23	14	23	14
Totale	76	384	418	672	23	14	517	1070



Molte istruzioni di assemblaggio vengono tradotte in diverse istruzioni della macchina (ad esempio opcode) in base ai dati su cui operano.

ARMv8 include entrambe le versioni a 32 bit e 64 bit delle istruzioni all'interno della stessa architettura.

In linguaggio assembly, i programmati utilizzano registri nominati W0, W1, ... al posto di X0, X1, ... per specificare le operazioni a 32 bit.

ADD X9, X21, X9

ADD W9, W21, W9

Dalla panoramica del set di istruzioni ARMv8:

- La maggior parte delle istruzioni intere nel set di istruzioni A64 ha due forme, che operano su valori a 32 o 64 bit all'interno del file di registro generico a 64 bit.
- Quando viene selezionato un modulo di istruzioni a 32 bit, vale quanto segue:
  - I 32 bit superiori dei registri sorgente vengono ignorati;
  - I 32 bit superiori del registro di destinazione sono impostati su ZERO;
  - Gli spostamenti/rotati a destra iniettano al bit 31, invece del bit 63;
  - I flag delle condizioni, dove impostati dall'istruzione, vengono calcolati dai 32 bit inferiori.

### 2.35.1 ARMv8 Istruzioni logiche aritmetiche intere

Grassetto significa che l'istruzione è anche in LEGv8, corsivo significa che è una pseudoistruzione, e grassetto corsivo significa che è una pseudoistruzione che è anche in LEGv8.

Tipo	Simbolo	Istruzione
Op. aritmetiche	<b>ADD</b>	Addizione
	<b>ADDS</b>	Addizione e set del flag
	<b>SUB</b>	Sottrazione
	<b>SUBS</b>	Sottrazione e set del flag
	<b>CMP</b>	Comparazione
	<i>CMN</i>	Comparazione negativa
	<i>NEG</i>	Negazione
	<i>NEGS</i>	Negazione e set del flag
Op. aritmetiche immediate	<b>ADDI</b>	Addizione immediata
	<b>ADDIS</b>	Addizione immediata e set del flag
	<b>SUBI</b>	Sottrazione immediata
	<b>SUBIS</b>	Sottrazione immediata e set del flag
	<b>CMPI</b>	Comparazione immediata
	<i>CMNI</i>	Comparazione negativa immediata



Continua dalla pagina precedente

Tipo	Simbolo	Istruzione
Aritmetica estesa	ADD	Agiungi registro esteso
	ADDS	Agiungi registro esteso e setta il flag
	SUB	Sottrae registro esteso
	SUBS	Sottrae registro esteso e setta il flag
	CMP	Compara registro esteso
	CMN	Compara registro esteso negativo
Aritmetica con carry	ADC	Addizione con carry
	ADCS	Addizione con carry e set del flag
	SBC	Sottrazione con carry
	SBCS	Sottrazione con carry e set del flag
	NGC	Negazione con carry
	NGCS	Negazione con carry e set del flag
Op. logiche	AND	AND bit a bit
	ANDS	AND bit a bit e set del flag
	ORR	OR (inclusivo) bit a bit
	EOR	OR (esclusivo) bit a bit
	BIC	Pulisce bit a bit
	BICS	Puslice bit a bit e setta il flag
	ORN	OR NOT (inclusivo) bit a bit
	EON	OR NOT (esclusivo) bit a bit
	MVN	NOT bit a bit
	TST	Test bits
Op. logiche immediate	ANDI	AND bit a bit immediato
	ANDIS	AND bit a bit e set del flag immediato
	ORRI	OR (inclusivo) bit a bit immediato
	EORI	OR (esclusivo) bit a bit immediato
	TSTI	Test bits immediato
Shift di registri	LSL	Shift logico a sx immediato
	LSR	Shift logico a dx immediato
	ASR	Shift aritmetico a dx immediato
	ROR	Rotazione a dx immediata
	LSRV	Shift logico a dx del registro
	LSLV	Shift logico a sx del registro
	ASRV	Shift aritmetico a dx del registro
	RORV	Rotazione a dx del registro
Spostamento di registri	MOVZ	Sposta registro con zero
	MOVK	Sposta registro con keep
	MOVN	Sposta registro con NOT
	MOV	Sposta registro
	BFM	Spost. campo di bit
	SBFM	Spost. campo di bit (segno)

**Continua dalla pagina precedente**

Tipo	Simbolo	Istruzione
Inserim./Estraz. campo di bit	UBFM	Spost. del campo di bit (senza segno)
	BFI	Inserim. campo di bit (segno)
	BFXIL	Estraz. campo di bit e ins. in basso
	SBFIZ	Campo bit con segno inserito in 0
	SBFX	Estraz. campo di bit (segno)
	UBFIZ	Campo bit no segno inserito in 0
	UBFX	Estraz. campo di bit (no segno)
	EXTR	Estraz. di registro da una coppia
Estensione del segno	SXTB	Byte di estensione del segno
	SXTH	Halfword di estensione del segno
	SXTW	Word di estensione del segno
	UXTB	Byte di estensione senza segno
	UXTH	Halfword di estensione senza segno
Operazioni di bit	CLS	Conta i bit del segno iniziale
	CLZ	Conta i bit con 0 iniziale
	RBIT	Inverte l'ordine dei bit
	REV	Inverte i byte nel registro
	REV16	Inverte i byte nelle halfword
	REV32	Inverte i byte nelle word

Il secondo registro di tutte le operazioni di elaborazione aritmetica e logica ha la possibilità di essere spostato prima di essere utilizzato. Le opzioni di spostamento sono spostamento a sinistra logico, spostamento a destra logico, spostamento aritmetico destro e rotazione a destra. Sebbene l'assembler abbia istruzioni esplicite con questi nomi (LSL, LSR, SRA e ROR), queste sono in realtà solo pseudoistruzioni.

Per supportare l'aritmetica sui tipi di dati più ristretti, ci sono istruzioni che ti consentono di mescolare le dimensioni dei dati del secondo operando per segno che lo estende o zero estendendolo fino all'intera larghezza. Le istruzioni del registro esteso funzionano con byte, halfword o word.

Per supportare le operazioni di aggiunta e sottrazione su operandi più grandi di una doubleword, ARM include istruzioni per aggiungere o sottrarre il trasporto da un'operazione precedente. ASR fa lo spostamento aritmetico a destra, che replica il bit del segno durante lo spostamento, e ROR ruota i bit a destra; cioè, i bit spostati a destra vengono inseriti a sinistra.

Ci sono versioni di tutte le istruzioni di turno che determinano l'importo da spostare in base a un valore in un registro piuttosto che come immediato all'interno dell'istruzione. Per manipolare i campi di bit, il set completo di istruzioni ARMv8 include istruzioni che possono estrarre un campo bit da un registro e inserirlo in un altro.



### 2.35.2 ARMv8 Istruzioni per il trasferimento dati

Non abbiamo visto tutte le modalità di indirizzamento disponibili, solo l'offset immediato non scalato. Eccone altri cinque:

1. Base più un offset immediato senza segno a 12 bit scalato.
2. Base più un offset del registro a 64 bit, opzionalmente scalato.
3. Base più un offset del registro esteso a 32 bit, opzionalmente scalato.

Le opzioni di ridimensionamento delle prime tre modalità di indirizzamento moltiplicano o ridimensionano l'indirizzo nel campo immediato o nel registro per la dimensione dei dati trasferiti in byte.

Quindi, se X11 contiene  $100.000_{ten}$

**LDR X10, [X11, #16] // Modalità di scala**

Caricherà la doppia parola (8 byte) all'indirizzo  $100.128_{ten}$  ( $100.000 + 8*16$ ) nel registro X10.

L'indirizzo della seconda modalità di indirizzamento è semplicemente la somma di due registri, con l'opzione di spostamento del secondo operando di 1, 2 o 3 bit.

Se X11 contiene  $100.000_{ten}$  e X12 contiene  $1.000_{ten}$

**LDR X10, [X11, X12 LSL #3] // Base + registro scalato**

Caricherà il doppio all'indirizzo  $108.000_{ten}$  ( $100.000 + 2\ 3 *1000$ ) nel registro X10.

La terza modalità di indirizzamento utilizza semplicemente un registro a 32 bit (ad esempio, W12) invece di un registro a 64 bit

4. Pre-indicizzato da un offset immediato firmato a 9 bit non scalato.
5. Post-indicizzato da un offset immediato firmato a 9 bit non scalato.

Queste ultime due modalità di indirizzamento cambiano il registro di base come parte del calcolo dell'indirizzo.

Quindi, se X11 contiene  $100.000_{ten}$

**LDR X10, [X11, #16]! // modalità di pre-indicizzazione dell'indirizzo**

Carica la doubleword all'indirizzo  $100.016_{ten}$  nel registro X10 e cambierà X11 in  $100.016_{ten}$ .

**LDR X10, [X11], #16 // modalità di post-indicizzazione dell'indirizzo**

Carica la doubleword all'indirizzo  $100.000_{ten}$  nel registro X10 e cambierà X11 in  $100.016_{ten}$ . Tra le altre cose, per accelerare i trasferimenti di dati, ARMv8 include tre istruzioni load pair e store pair (LDP, LDPSW, STP), che trasferiscono due doubleword alla volta.



Tipos	Simbolo	Istruzione
Unscaled	<b>LDUR</b>	Carica registro
	<b>LDURB</b>	Carica byte
	<b>LDURSB</b>	Carica byte (con segno)
	<b>LDURH</b>	Carica halfword
	<b>LDURSH</b>	Carica halfword (con segno)
	<b>LDURSW</b>	Carica word (con segno)
	<b>STUR</b>	Store registro
	<b>STURB</b>	Store byte
	<b>STURH</b>	Store halfword
	<b>STURW</b>	Store word
Scaled Extended Pre & Post Indexed	<b>LDA</b>	Carica indirizzo
	<b>LDR</b>	Carica registro
	<b>LDRB</b>	Carica byte
	<b>LDRSB</b>	Carica byte (con segno)
	<b>LDRH</b>	Carica halfword
	<b>LDRSH</b>	Carica halfword (con segno)
	<b>LDRSW</b>	Carica word (con segno)
	<b>STR</b>	Store registro
	<b>STRB</b>	Store byte
	<b>STRH</b>	Store halfword
PC Pair Exclusive Exclusive Acquire/Release	<b>LDXR</b>	Load esclusivo registro
	<b>LDXR B</b>	Load esclusivo byte
	<b>LDXR H</b>	Load esclusivo halfword
	<b>LDXP</b>	Load esclusivo pair
	<b>STXR</b>	Store esclusivo registro
	<b>STXRB</b>	Store esclusivo byte
	<b>STXRH</b>	Store esclusivo halfword
	<b>STXP</b>	Store esclusivo pair
	<b>LDAXR</b>	Load-aquire registri esclusivo
	<b>LDAXRB</b>	Load-aquire byte esclusivo
	<b>LDAXRH</b>	Load-aquire halfword esclusivo
	<b>LDAXP</b>	Load-aquire pair esclusivo
	<b>STLXR</b>	Store-release registri esclusivo
	<b>STLXRB</b>	Store-release byte esclusivo
	<b>STLXRH</b>	Store-release halfword esclusivo
	<b>STLXP</b>	Store-release pair esclusivo
	<b>LDP</b>	Load pair
	<b>LDPSW</b>	Load pair (con segno)
	<b>STP</b>	Store pair
	<b>ADRP</b>	Calcola l'indirizzo di una pagina da 4 KB
	<b>ADR</b>	Calcola l'indirizzo della label



## 2.36 ARMv8 Istruzioni di Branch

Tipo	Simbolo	Istruzione
Salti condizionali	<b>B.cond</b>	Salto condizionale
	<b>CBNZ</b>	Compara e salta se non zero
	<b>CBZ</b>	Compara e salta se zero
	<b>TBNZ</b>	Testa e salta se non zero
	<b>TBZ</b>	Testa e salta se zero
Salti non condizionali	<b>B</b>	Salto senza condizioni
	<b>BL</b>	Salta a un link
	<b>BLR</b>	Salta a un link a un registro
	<b>BR</b>	Salta a un registro
	<b>RET</b>	Ritorna alla subroutine
Selezioni condizionali	<b>CSEL</b>	Selezione condizionale
	<b>CSINC</b>	Selezione condizionale dell'incremento
	<b>CSINV</b>	Selezione condizionale dell'inverso
	<b>CSNEG</b>	Selezione condizionale del negato
	<b>CSET</b>	Set condizionale
	<b>CSETM</b>	Set mask condizionale
	<b>CINC</b>	Incremento condizionale
	<b>CINV</b>	Inversione condizionale
	<b>CNEG</b>	Negazione condizionale
Comparazioni condizionali	<b>CCMP</b>	Compare tra registri condizionale
	<b>CCMPI</b>	Compare immediata condizionale
	<b>CCMN</b>	Compare tra registri negativi condizionale
	<b>CCMNI</b>	Compare immediata negativa condizionale

Ci sono altri due salti incondizionati:

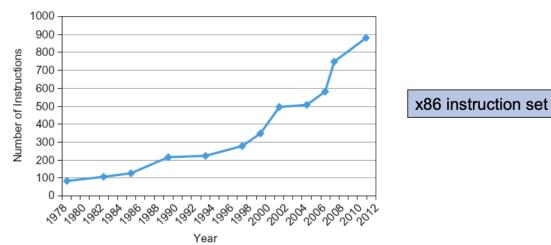
- Il primo è una variazione di ramo e link che utilizza un registro per l'indirizzo della filiale (BALR).
- Il secondo è il ritorno dalla subroutine (RET), che suona molto simile al registro delle filiali (BR);

Il motivo per cui ARMv8 ha diversi opcode per la stessa operazione è che i predittori di salti hardware possano sapere se è davvero di ritorno da una subroutine (RET), che è facile da prevedere, o di essere utilizzato in una tabella di rami (BR), che è molto più difficile da prevedere.

Ci sono istruzioni che memorizzano un valore in un registro basato sui codici delle condizioni. L'idea alla base delle istruzioni di selezione delle condizioni è quella di sostituire i salti condizionali, che possono causare problemi all'esecuzione delle pipeline se non possono essere predetti.

## 2.37 Errori

- Istruzioni potenti  $\Rightarrow$  Performance migliori  
Meno istruzioni richieste ma le istruzioni complesse sono difficili da implementare.  
Può rallentare tutte le istruzioni, comprese quelle semplici. I compilatori sono bravi a creare codice veloce da semplici istruzioni
- Usa il codice di assemblaggio per prestazioni elevate  
I compilatori moderni sono più bravi a trattare con i processori moderni. Più righe di codice  $\Rightarrow$  più errori e meno produttività
- Compatibilità all'indietro  $\Rightarrow$  il set istruzioni non cambia  
Ma accumulano più istruzioni



## 2.38 Insidie

- Le parole sequenziali non sono agli indirizzi sequenziali: Incremento di 4, non di 1!
- Mantenere un puntatore a una variabile automatica dopo i ritorni della procedura  
Ad esempio, passando il puntatore indietro tramite un argomento. Il puntatore diventa invalido quando lo stack viene svuotato

### 3 Aritmetic in LEGv8

#### 3.1 Moltiplicazione in LEGv8

Per produrre un prodotto a 128 bit correttamente firmato o non firmato, LEGv8 ha tre istruzioni:

- Moltiplicazione (**MUL**)
- Moltiplicazione con segno alto (**SMULH**)
- Moltiplicazione senza segno alto (**UMULH**).

Per ottenere il prodotto intero a 64 bit, il programmatore utilizza MUL. Per ottenere i 64 bit superiori del prodotto a 128 bit, il programmatore utilizza SMULH o UMULH, a seconda dei tipi di moltiplicatore e moltiplicatore.

Le istruzioni di moltiplicazione LEGv8 non impostano il codice delle condizioni di overflow, quindi spetta al software verificare se il prodotto è troppo grande per adattarsi a 64 bit. Non c'è overflow se i 64 bit superiori sono 0 per UMULH o il segno replicato dei 64 bit inferiori per SMULH.

#### 3.2 Divisione in LEGv8

Per gestire sia numeri interi firmati che numeri interi non firmati, LEGv8 ha due istruzioni:

- Divisione con segno (**SDIV**)
- Divisione senza segno (**UDIV**)

Il supporto hardware comune per moltiplicare e dividere consente a LEGv8 di fornire una singola coppia di registri a 64 bit che vengono utilizzati sia per moltiplicare che per dividere. Le istruzioni LEGv8 divide ignorano l'overflow: il software deve determinare se il quoziente è troppo grande. Oltre all'overflow, la divisione può anche comportare un calcolo improprio: divisione per 0. *Il software LEGv8 deve controllare il divisore per scoprire la divisione per 0 e l'overflow.*

Moltiplicazione	MUL X1, X2, X3	X1 = X2 * X3
Moltiplicazione con segno alto	SMULH X1, X2, X3	X1 = X2 * X3
Moltiplicazione senza segno alto	UMULH X1, X2, X3	X1 = X2 * X3
Divisione con segno	SDIV X1, X2, X3	X1 = X2 / X3
Divisione senza segno	UDIV X1, X2, X3	X1 = X2 / X3

### 3.3 Floating Point

Rappresentanza per i numeri non interi, compresi numeri molto piccoli e molto grandi.  
Un esempio di notazione floating point è la notazione scientifica

- Normalizzato:  $-2,34 \cdot 10^{56}$
- Non normalizzato:  $+0,002 \cdot 10^{-4}$  oppure  $+987,02 \cdot 10^9$

In binario:

$$\pm 1.xxxxxxx_2 \cdot 2^{yyy}$$

Dei tipi floating point in C sono le variabili float e double

#### 3.3.1 Floating Point Standard IEEE Std 754-1985

Due rappresentazioni: precisione singola (32 bit) e doppia precisione (64 bit)

singola: 8 bits	singola: 23 bits	
double: 11 bits	double 52 bits	
S	Esponente	Frazione

$$x = (-1)^S \times (1 + Frazione) \times 2^{Esponente-Bias}$$

S: bit di segno (0  $\Rightarrow$  non negativo, 1  $\Rightarrow$  negativo)

Normalizzazione della parte significativa:  $1.0 \leq |parte\ significativa| < 2.0$  Ha sempre un 1 bit pre-punto binario iniziale, quindi non c'è bisogno di rappresentarlo esplicitamente (bit nascosto).

Esponente: rappresentazione in eccesso: esponente effettivo + Bias

- Assicura che l'esponente sia senza segno
- Singola precisione: Bias = 127; Doppia precisione: Bias = 1023

#### 3.3.2 Range in singola precisione

Esponenti 00000000 e 11111111 riservati Valore più piccolo:

- Esponente: 00000001  
 $\Rightarrow$  Esponente effettivo =  $1 - 127 = -126$
- Frazione: 000...00  $\Rightarrow$  significa = 1,0
- $\pm 1.0 \times 2^{-126} \approx \pm 1.2 \times 10^{-38}$

Valore più grande:

- Esponente: 11111110  
 $\Rightarrow$  Esponente effettivo =  $254 - 127 = +127$
- Frazione: 111...11  $\Rightarrow$  significa  $\approx 2,0$
- $\pm 2.0 \times 2^{127} \approx \pm 3.4 \times 10^{+38}$

### 3.3.3 Range in doppia precisione

Esponenti 00000000 e 11111111 riservati Valore più piccolo:

- Esponente: 0000000001  
⇒ Esponente effettivo =  $1 - 1023 = -1022$
- Frazione: 000...00 ⇒ significa = 1,0
- $\pm 1.0 \times 2^{-1022} \approx \pm 2.2 \times 10^{-308}$

Valore più grande:

- Esponente: 1111111110  
⇒ Esponente effettivo =  $2046 - 1023 = +1023$
- Frazione: 111...11 ⇒ significa  $\approx 2,0$
- $\pm 2.0 \times 2^{1023} \approx \pm 1.8 \times 10^{308}$

### 3.3.4 Infiniti e NaNs

Esponente = 111...1 e Frazione = 000...0

- $\pm$  Infinito
- Può essere utilizzato nei calcoli successivi, evitando la necessità di controllo dell'overflow

Esponente = 111...1 e Frazione  $\neq$  000...0

- Non un numero (NaN)
- Indica un risultato illegale o indefinito
- Ad esempio,  $\frac{0}{0}$
- Può essere utilizzato nei calcoli successivi

### 3.3.5 Numeri denormalizzati

Esponente = 000...0 ⇒ Il bit nascosto è 0

$$x = (-1)^S \times (0 + \text{Frazione}) \times 2^{-Bias}$$

Più piccoli dei numeri normali. Consente un sottoflusso graduale, con precisione decrescente. Denormalizzazione con frazione = 000...0

$$x = (-1)^S \times (0 + 0) \times 2^{-Bias} = \pm 0.0$$

Notiamo che abbiamo due rappresentazione dello 0



### 3.3.6 Riassunti IEEE Std 754-1985

Precisione singola		Precisione doppia		Oggetto rappresentato
Esponente	Frazione	Esponente	Frazione	
0	0	0	0	0
0	Non zero	0	Non zero	$\pm$ Numero denormalizzato
1 - 254	Qualsiasi	1 - 2046	Qualsiasi	$\pm$ Numero floating point
255	0	2047	0	$\pm$ Infinito
255	Non zero	2047	Non zero	NaN (Non a Number)

## 3.4 Overflow & Underflow

Per quanto riguarda le operazioni intere, l'operazione aritmetica in virgola mobile può originare "trabocchi". Overflow qui significa che l'esponente è troppo grande per essere rappresentato nel campo esponente.

Anche il virgola mobile offre un nuovo tipo di evento eccezionale: la frazione diversa da zero che stiamo calcolando potrebbe diventare così piccola che non può essere rappresentata. Chiamiamo questo evento underflow: si verifica quando l'esponente negativo è troppo grande per adattarsi al campo dell'esponente.

### 3.4.1 Managing overflows and underflows

Cosa dovrebbe accadere su un overflow o underflow per far sapere all'utente che si è verificato un problema?

LEGv8 può sollevare un'eccezione, chiamata anche interrupt su molti computer. Un'eccezione o un'interruzione è essenzialmente una chiamata di procedura non programmata. L'indirizzo dell'istruzione in overflow viene salvato in un registro, e il computer salta a un indirizzo predefinito per invocare la routine appropriata per quell'eccezione.

In alcune situazioni il programma può continuare dopo l'esecuzione del codice correttivo.

## 3.5 Istruzioni Floating-Point in LEGv8

LEGv8 supporta i formati IEEE 754 a precisione singola e doppia precisione con queste istruzioni:

- Aggiunta in virgola mobile, singola (**FADDS**) e addizione, doppia (**FADDD**)
- Sottrazione in virgola mobile, singola (**FSUBS**) e sottrazione, doppia (**FSUBD**)
- Moltiplicazione in virgola mobile, singola (**FMULS**) e moltiplicazione, doppia (**FMULD**)
- Divisione in virgola mobile, singola (**FDIVS**) e divisione, doppia (**FDIVD**)
- Confronto in virgola mobile, singola (**FCMPS**) e confronto, doppia (**FCMPD**)



Registri separati in virgola mobile:

- vengono chiamati S0, S1, S2, ... per precisione singola e D0, D1, D2, . . . per doppia precisione.
- I registri in precisione singola sono solo la metà inferiore dei registri a doppia precisione.

Le istruzioni FP funzionano solo sui registri FP. I programmi generalmente non fanno operazioni intere sui dati FP, o viceversa. Più registri con un impatto minimo sulle dimensioni del codice.

Istruzioni per il load e lo store FP:

- **LDURS, LDURD**
- **STURS, STURD**

### 3.5.1 Floating-point in linguaggio Assembly

Categoria	Istruzione	Esempio	Significato
Aritmetica	ADD singola	FADDS S2, S4, S6	$S2 = S4 + S6$
	SUB singola	FSUBS S2, S4, S6	$S2 = S4 - S6$
	MUL singola	FMULS S2, S4, S6	$S2 = S4 * S6$
	DIV singola	FDIVS S2, S4, S6	$S2 = S4 / S6$
	ADD doppia	FADDD D2, D4, D6	$D2 = D4 + D6$
	SUB doppia	FSUBD D2, D4, D6	$D2 = D4 - D6$
	MUL doppia	FMULD D2, D4, D6	$D2 = D4 * D6$
	DIV doppia	FDIVD D2, D4, D6	$D2 = D4 / D6$
Branch	CMP singola	FCMPS S4, S6	Test S4 vs S6
	CMP doppia	FCMPD D4, D6	Test D4 vs D6
Dati	Load singolo	LDURS S1, [X23, 100]	$S1 = \text{Memory}[X23 + 100]$
	Load doppio	LDURD D1, [X23, 100]	$D1 = \text{Memory}[X23 + 100]$
	Store singolo	STURS S1, [X23, 100]	$\text{Memory}[X23 + 100] = S1$
	Store doppio	STURD D1, [X23, 100]	$\text{Memory}[X23 + 100] = D1$

### 3.5.2 Floating-point in linguaggio macchina

Nome	Formato	Esempi				
FADDS	R	241	6	10	4	2
FSUBS	R	241	6	14	4	2
FMULS	R	241	6	2	4	2
FDIVS	R	241	6	6	4	2
FADDD	V	243	6	10	4	2
FSUBD	R	243	6	14	4	2
FMULD	R	243	6	2	4	2
FDIVD	R	243	6	6	4	2
FCMPS	R	241	6	8	4	0
FCMPD	R	243	6	8	4	0
LDURS	D	1506	100	0	4	2
LDURD	D	2018	100	0	4	2
STURS	D	1504	100	0	4	2
STURD	D	2016	100	0	4	2
Dimensione campo		11 bits	5 o 9 bits	6 o 2 bits	5 bits	5 bits

## 3.6 Aritmetica accurata

Lo standard IEEE 754 specifica un controllo aggiuntivo dell’arrotondamento:

- Bit extra di precisione (guardia, rotonda, appiccicosa)
- **Guard** and **around**: Il primo dei due bit extra tenuti a destra durante i calcoli intermedi dei numeri in virgola mobile; utilizzato per migliorare la precisione di arrotondamento.
- **sticky bit**: Un po’ utilizzato nell’arrotondamento oltre alla guardia e al round che viene impostato ogni volta che ci sono bit diversi a zero a destra della punta rotonda.

Scelta delle modalità di arrotondamento. Consente al programmatore di mettere a punto il comportamento numerico di un calcolo

Non tutte le unità FP implementano tutte le opzioni. La maggior parte dei linguaggi di programmazione e delle librerie FP utilizza solo le impostazioni predefinite

## 3.7 The BIG picture

I modelli di bit non hanno alcun significato intrinseco. Possono rappresentare numeri interi con segno, numeri interi senza segno, numeri in virgola mobile, istruzioni, caratteri, stringhe, e così via. Ciò che è rappresentato dipende dalle istruzioni che operano sui bit nella parola.



La principale differenza tra numeri di computer e numeri nel mondo reale è che i numeri di computer hanno dimensioni limitate e quindi precisione limitata; è possibile calcolare un numero troppo grande o troppo piccolo per essere rappresentato in una parola informatica.

I programmatori devono ricordare questi limiti e scrivere i programmi di conseguenza.

### 3.8 Parallelismo delle subword

Molti sistemi grafici utilizzano 8 bit per rappresentare ciascuno dei tre colori primari. I campioni audio sono spesso rappresentati con 16 bit.

Le architetture hanno riconosciuto che molte applicazioni grafiche e audio eseguirebbero la stessa operazione sui vettori di questi dati. Pertanto, le applicazioni grafiche e audio possono trarre vantaggio dall'esecuzione di operazioni simultanee su vettori brevi. Partitionando le catene di trasporto all'interno di un vettore a 128 bit, un processore potrebbe usare il parallelismo per eseguire operazioni simultanee su vettori più corti:

- Sedici aggiunte a 8 bit
- Otto aggiunte a 16 bit
- Quattro aggiunte a 32 bit

Il parallelismo della subword è anche chiamato parallelismo a livello di dati, parallelismo vettoriale o istruzione singola, dati multipli (SIMD).

### 3.9 ARMv8 SIMD

L'ARMv8 ha aggiunto 32 registri a 128 bits (V0, V1, ..., V31) e oltre 500 istruzioni in linguaggio macchina per supportare il parallelismo delle subword.

Supporta tutti i tipi di dati delle subword che puoi immaginare:

- Numeri interi con segno e senza segno a 8 bit, 16 bit, 32 bit, 64 bit e 128 bit
- Numeri in virgola mobile a 32 e 64 bit

L'assembler ARMv8 utilizza suffissi diversi per i registri SIMD per rappresentare larghezze diverse. I suffissi sono B (byte) per gli operandi a 8 bit, H (half) per gli operandi a 16 bit, S (single) per gli operandi a 32 bit, D (double) per gli operandi a 64 bit e Q (quad) per gli operandi a 128 bit.

Il programmatore specifica anche il numero di operazioni di subword per quella larghezza di dati con un numero prima del nome del registro.

**Esempio:**

- Addizione 16 interi a 8 bit

ADD V1.16B, V2.16B, V3.16B



- Addizione 4 numeri FP a 32 bit

FADD V1.4S, V2.4S, V3.4S

Tipo	Nome	Dimensione (bits)					Precisione FP	
		8	16	32	64	128		
Addizione /Sottrazione	ADD	✓	✓	✓	✓	✓		
	FADD						✓	✓
	SUB	✓	✓	✓	✓	✓		
	FSUB						✓	✓
Moltiplicazione	UMUL	✓	✓	✓	✓	✓		
	SMUL	✓	✓	✓	✓	✓		
	FMUL						✓	✓
Comparazione	CMEQ	✓	✓	✓	✓	✓		
	FCMEQ						✓	✓
Min / Max	UMIN	✓	✓	✓	✓	✓		
	SMIN	✓	✓	✓	✓	✓		
	FMIN						✓	✓
	UMAX	✓	✓	✓	✓	✓		
	SMAX	✓	✓	✓	✓	✓		
	FMAX						✓	✓
Shift	SHL	✓	✓	✓	✓	✓		
	USHR	✓	✓	✓	✓	✓		
	SSHR	✓	✓	✓	✓	✓		
Logical	AND	✓	✓	✓	✓	✓		
	ORR	✓	✓	✓	✓	✓		
	EOR	✓	✓	✓	✓	✓		
Dati	LDR	✓	✓	✓	✓	✓	✓	✓
	STR	✓	✓	✓	✓	✓	✓	✓

### 3.10 ARMv8 Operazioni aritmetiche intere e floating-point

Tipo	Simbolo	Istruzione
Multiply & Divide	<b>MUL</b>	Moltiplica
	<b>SMULH</b>	Moltiplica alto (con segno)
	<b>UMULH</b>	Moltiplica alto (senza segno)
	<b>SDIV</b>	Dividi (con segno)
	<b>UDIV</b>	Dividi (senza segno)
	<b>SMULL</b>	Moltiplica long (con segno)
	<b>UMULL</b>	Moltiplica long (senza segno)
	<b>MNEG</b>	Moltiplica il negato
	<b>UMNEGL</b>	Moltiplica il negato long (senza segno)
	<b>SMNEGL</b>	Moltiplica il negato long (con segno)



Continua dalla pagina precedente

Tipo	Simbolo	Istruzione
Operandi FP (2 ingressi)	<b>FADDS</b>	Addizione prec. singola FP
	<b>FSUBS</b>	Sottrazione prec. singola FP
	<b>FMULS</b>	Moltiplicazione prec. singola FP
	<b>FDIVS</b>	Divisione prec. singola FP
	<b>FADDD</b>	Addizione prec. doppia FP
	<b>FSUBD</b>	Sottrazione prec. doppia FP
	<b>FNMUL</b>	Moltiplicazione negata scalare FP
	<b>FMULD</b>	Moltiplicazione prec. doppia FP
	<b>FDIVD</b>	Divisione prec. doppia FP
	<b>FCMPS</b>	Compare prec. singola FP
	<b>FCMPD</b>	Compare prec. doppia FP
	<b>FCMPE</b>	Compare "signaling" FP
	<b>FCCMP</b>	Compare condizionale FP
	<b>FCCMPE</b>	Compare condizionale "signaling" FP
Operandi FP (1 ingresso)	<b>FABS</b>	Valore assoluto FP
	<b>FNEG</b>	Negazione FP
	<b>FSQRT</b>	Radice quadrata FP
Min/Max FP	<b>FMAX</b>	Massimo FP
	<b>FMIN</b>	Minimo FP
	<b>FMAXNM</b>	Massimo FP (NaN = - Inf)
	<b>FMINNM</b>	Minimo FP (NaN = + Inf)
Mul-Add Intero	<b>MADD</b>	Multiply - Add
	<b>MSUB</b>	Multiply - Subtract
	<b>SMADDL</b>	Multiply - Add long (con segno)
	<b>SMSUBL</b>	Multiply - Subtract long (con segno)
	<b>UMADDL</b>	Multiply - Add long (senza segno)
	<b>UMSUBL</b>	Multiply - Subtract long (senza segno)
Mul-Add FP	<b>FMADD</b>	Multiply - Add FP
	<b>FMSUB</b>	Multiply - Subtract FP
	<b>FNMADD</b>	Multiply - Add negato FP
	<b>FNMSUB</b>	Multiply - Subtract negato FP
Move FP	<b>MOV</b>	Move da/a intero o registro FP
	<b>FMOVI</b>	Move immediate FP
Selezione FP	<b>FCSEL</b>	Selezione condizionale FP



### 3.11 Istruzioni core LEGv8

Istruzione LEGv8	Nome	Formato
Addizione	ADD	R
Sottrazione	SUB	R
Addizione immediata	ADDI	I
Sottrazione immediata	SUBI	I
Addizione e set flag	ADDS	R
Sottrazione e set flag	SUBS	R
Addizione immediata e set flag	ADDIS	I
Sottrazione immediata e set flag	SUBIS	I
Load registro	LDUR	D
Store registro	STUR	D
Load word (con segno)	LDURSW	D
Store word	STURW	D
Load half	LDURH	D
Store half	STURH	D
Load byte	LDURB	D
Store byte	STURB	D
Load registri esclusivi	LDXR	D
Store registri esclusivi	STXR	D
Move con zero	MOVZ	IM
Move con keep	MOVK	IM
AND	AND	R
OR inclusivo	ORR	R
OR esclusivo	EOR	R
AND immediato	ANDI	I
OR inclusivo immediato	ORRI	I
OR esclusivo immediato	EORI	I
Shift logico a sinistra	LSL	R
Shift logico a destra	LSR	R
Compare e Branch se zero	CBZ	CB
Compare e Branch se non zero	CBNZ	CB
Branch condizionale	B.cond	CB
Brach	B	B
Branch a registro	BR	R
Branch con link	BL	B
Moltiplicazione	MUL	R
Moltiplicazione alta (con segno)	SMULH	R
Moltiplicazione alta (senza segno)	UMULH	R
Divisione (con segno)	SDIV	R
Divisione (senza segno)	UDIV	R



Continua dalla pagina precedente

Istruzione LEGv8	Nome	Formato
Addizione prec. singola FP	FADDS	R
Sottrazione prec. singola FP	FSUBS	R
Moltiplicazione prec. singola FP	FMULS	R
Divisione prec. singola FP	FDIVS	R
Addizione prec. doppia FP	FADDD	R
Sottrazione prec. doppia FP	FSUBD	R
Moltiplicazione prec. doppia FP	FMULD	R
Divisione prec. doppia FP	FDIVD	R
Compare prec. singola FP	FCMPS	R
Compare prec. doppia FP	FCMPD	R
Load prec. singola FP	LDURS	D
Load prec. doppia FP	LDURD	D
Store prec. singola FP	STURS	D
Store prec. doppia FP	STURD	D

### 3.12 Insidie ed Errori

- Proprio come un'istruzione di shift sinistro può sostituire un numero intero moltiplicato per una potenza di 2, uno shift destro è lo stesso di una divisione intera di una potenza di 2.

Lo shift destro divide per  $2^i$  solo per numeri interi senza segno.

Per numeri interi con segno, ad esempio  $-\frac{5}{4}$ .

Con shift logico:  $11111011_2 >>> 2 = 0011110_2 = +62$

Con shift aritmetico: replica il bit del segno  $\rightarrow 11111011_2 >> 2 = 11111110_2 = -2$

- Le strategie di esecuzione parallela che funzionano per i tipi di dati interi funzionano anche per i tipi di dati in virgola mobile.

- I programmi paralleli possono interlasciare le operazioni in ordini imprevisti
- Le ipotesi di associatività possono fallire
- Necessità di convalidare i programmi paralleli con vari gradi di parallelismo
- I programmatori che scrivono codice parallelo con numeri in virgola mobile devono verificare se i risultati sono credibili, anche se non danno la stessa risposta del codice sequenziale.

**Errore:** L'addizione floating point NON è associativa

		$(x + y) + z$	$x + (y + z)$
$x$	$-1,50 * 10^{38}$		$-1,50 * 10^{38}$
$y$	$1,50 * 10^{38}$	$0,00 * 10^0$	
$z$	1,0	1,0	$1,50 * 10^{38}$
		$1,00 * 10^0$	$0,00 * 10^0$

## 4 The Processor

### 4.1 Introduzione

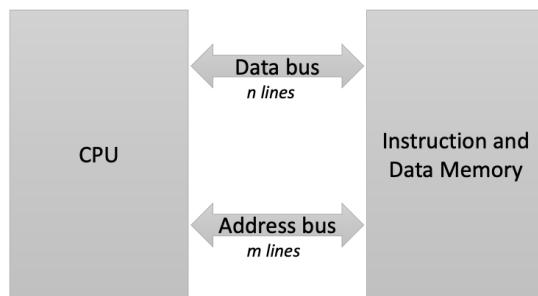
Abbiamo visto che le prestazioni di un computer sono determinate da tre fattori chiave:

- Conteggio delle istruzioni
- Tempo del ciclo di clock
- Cicli di clock per istruzione (CPI)

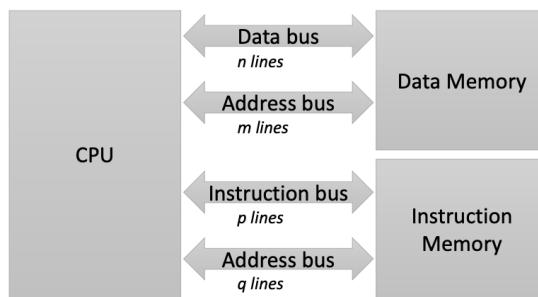
Il compilatore e l'architettura del set di istruzioni determinano il conteggio delle istruzioni richiesto per un dato programma. L'implementazione del processore determina sia il tempo del ciclo di clock che il numero di cicli di clock per istruzione.

Possiamo avere diverse organizzazioni del processore: Harward o Von Neumann  
E diverse strategie di implementazione: singolo ciclo, multi ciclo, pipeline.

#### 4.1.1 Organizzazione di Von Neumann



#### 4.1.2 Organizzazione di Harward



### 4.2 Un'implementazione di base di LEGv8

Esamineremo un'implementazione che include un sottoinsieme del set di istruzioni di base LEGv8: Le istruzioni di riferimento della memoria caricano il registro non scalato

(LDUR) e memorizzano il registro non scalato (STUR). Le istruzioni aritmetiche-logiche ADD, SUB, AND e ORR. Le istruzioni di confronto e branch se zero (CBZ) e branch (B).

Queste operazioni illustrano i principi chiave utilizzati nella creazione di un percorso dati e nella progettazione del controllo. Avremo l'opportunità di vedere: come l'architettura del set di istruzioni determina molti aspetti dell'implementazione e come la scelta delle varie strategie di implementazione influisce sulla frequenza di clock e sul CPI.

### 4.3 Esecuzione delle istruzioni

Molto di ciò che deve essere fatto per implementare queste istruzioni è lo stesso, indipendente dall'esatta classe di istruzione. Per ogni istruzione, i primi due passaggi sono identici:

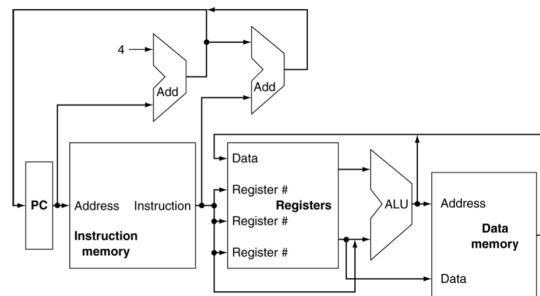
1. Invia il program counter (PC) alla memoria che contiene il codice e recupera l'istruzione da quella memoria.
2. Leggi uno o due registri, usando i campi dell'istruzione per selezionare i registri da leggere.

Dopo questi passaggi, le azioni necessarie per completare l'istruzione dipendono dalla classe di istruzione.

A seconda della classe di istruzione

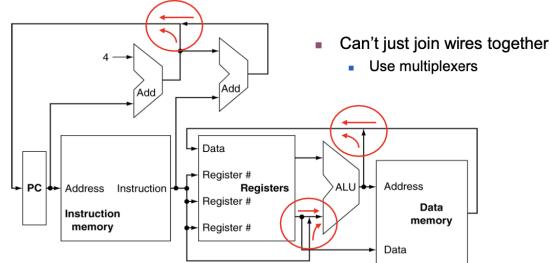
- Usa l'ALU per calcolare
  - Risultato aritmetico
  - Indirizzo di memoria per load/store
  - Confronto con zero nel branch
- Accedi alla memoria dati per il load/store
- $PC \leftarrow \text{target address or } PC + 4$

### 4.4 CPU Overview

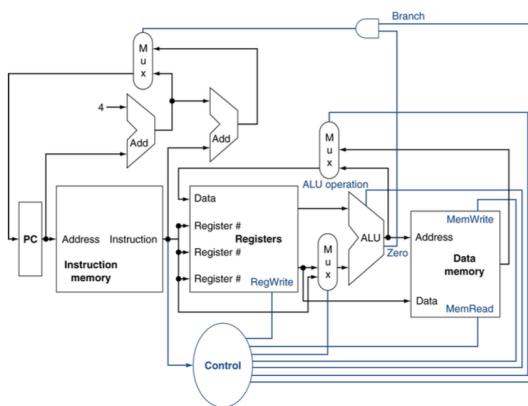




#### 4.4.1 Multiplexer



#### 4.4.2 Controllore



### 4.5 Convenzioni di progettazione logica

Gli elementi del percorso dati nell'implementazione LEGv8 sono costituiti da due diversi tipi di elementi logici:

- Elementi combinati
  - Operazioni sui dati
  - L'output è una funzione dell'input
- Elementi di stato (sequenziali)
  - Memorizza informazioni (e.g. registri e memorie)
  - Chiamiamo questi elementi elementi di stato perché, se togliessimo la spina di alimentazione sul computer, potremmo riavviarlo con precisione caricando gli elementi di stato con i valori che contenevano prima di staccare la spina.
  - Un elemento di stato ha almeno due ingressi e un'uscita:
    - \* Inserimento dati, clock.

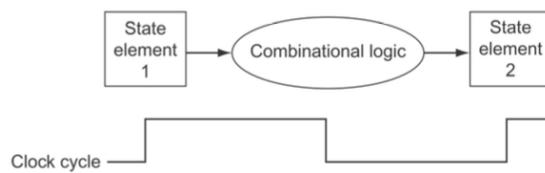


\* L'output è il valore che è stato scritto in un ciclo di clock precedente.

- Il clock viene utilizzato per determinare quando l'elemento di stato deve essere scritto.

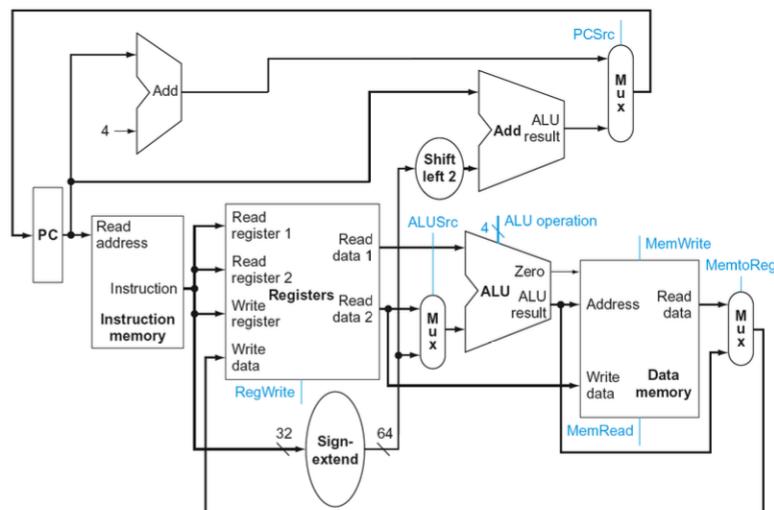
## 4.6 Metodologia di clocking

Una metodologia di clock definisce quando i segnali possono essere letti e quando possono essere scritti. Assumeremo una metodologia di clock attivata dall'edge. Tutti i valori memorizzati in un elemento logico sequenziale vengono aggiornati solo sul bordo del clock.



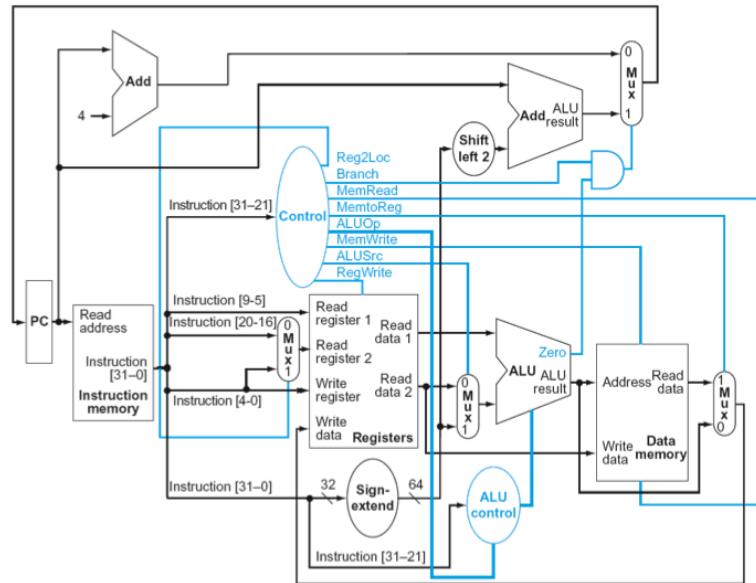
Tutti i segnali devono propagarsi dall'elemento di stato 1, attraverso la logica combinata, e allo stato dell'elemento 2 nel tempo di un ciclo di clock. Il tempo necessario affinché i segnali raggiungano l'elemento di stato 2 definisce la lunghezza del ciclo di clock. Se un elemento di stato non viene aggiornato su ogni clock, è necessario un segnale di controllo di scrittura esplicito. L'elemento di stato viene modificato solo quando viene affermato il segnale di controllo e si verifica un bordo del clock.

## 4.7 Il percorso dati semplice





#### 4.7.1 Il percorso dati semplice con controllore



## 4.8 Controllo della ALU

La ALU viene usata per:

- Load/Store: Funzione = add
- Branch: Funzione = compare
- R-Type: Funzione dipende dall'opcode

Assumiamo un ALUOp a 2-bit, dagli opcode sappiamo che:

- 00 - loads e stores
- 01 - Passaggio dell'input per il CBZ
- 10 - determina dall'opcode i campi per le operazioni di tipo R

ALU Control lines	Funzione
0000	AND
0001	OR
0010	ADD
0110	SUB
0111	Input per il branch
1100	NOR



Tipo	ALUOp	Istruzione	Campi Opcode	Azione ALU	Control input
LDUR	00	Load register	XXXXXXX	add	0010
STUR	00	Store register	XXXXXXX	add	0010
CBZ	01	CBZ	XXXXXXX	pass input b	0111
Tipo R	10	ADD	10001011000	addizione	0010
Tipo R	10	SUB	11001011000	sottrazione	0110
Tipo R	10	AND	10001010000	AND	0000
Tipo R	10	ORR	10101010000	OR	0001

## 4.9 Formato istruzioni

Vedremo l'implementazione di queste 3 istruzioni:

ADD X1, X2, X3  
LDUR X1, [X2, offset]  
CBZ X1, offset

Campo

opcode	Rm	shamt	Rn	Rd
31:21	20:16	15:10	9:5	4:0

Posizione bit

Istruzione di tipo R

Campo

1986 o 1984	indirizzo	0	Rn	Rt
31:21	20:12	11:10	9:5	4:0

Posizione bit

Istruzione di load/store

Campo

180	indirizzo	Rt
31:24	23:5	4:0

Posizione bit

Istruzione di Conditional branch

Ci sono diverse osservazioni importanti su questo formato di istruzioni su cui faremo affidamento:

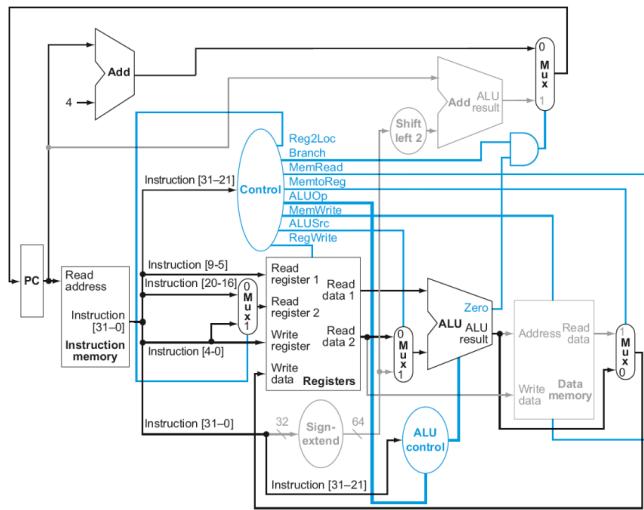
- Il campo opcode è largo tra 6 e 11 bit e si trova nei bit da 31:26 a 31:21.
- Il primo operando del registro è sempre in posizioni di bit 9:5 (Rn) sia per le istruzioni di tipo R che per il registro di base per le istruzioni di carico e conservazione.
- L'altro registro operando si trova in uno dei due posti. È in posizioni di bit 20:16 (Rm) per istruzioni di tipo R ed è in posizioni di bit 4:0 (Rt) che il registro sia scritto da un load. Anche questo è il campo che specifica il registro da testare per zero per confrontare e saltare se zero.
- Un altro operando può anche essere un offset a 19 bit per confrontare e saltare se zero o un offset a 9 bit per load e store.
- Il registro di destinazione per le istruzioni di tipo R (Rd) e per i load (Rt) è in posizioni di bit 4:0.



#### 4.9.1 ADD X1, X2, X3

Anche se tutto avviene in un ciclo di clock, possiamo pensare a quattro passaggi per eseguire l'istruzione, questi passaggi sono ordinati dal flusso di informazioni:

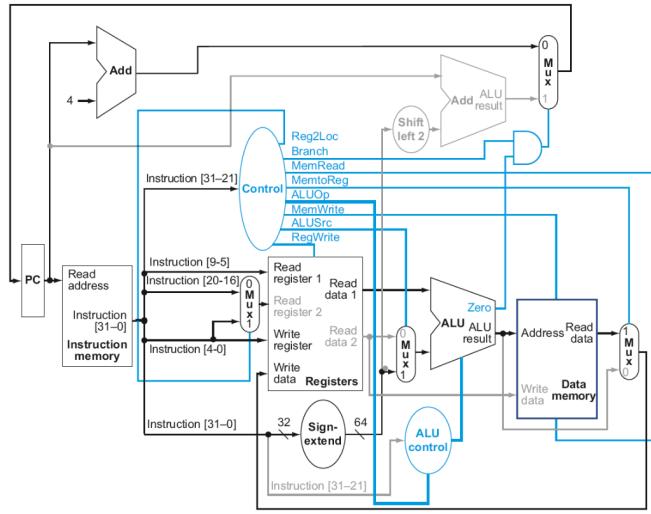
1. L'istruzione viene recuperata e il PC viene incrementato.
2. Due registri, X2 e X3, vengono letti dal file del registro; inoltre, l'unità di controllo principale calcola l'impostazione delle linee di controllo durante questo passaggio.
3. L'ALU opera sui dati letti dal file di registro, utilizzando parti dell'opcode per generare la funzione ALU.
4. Il risultato dell'ALU è scritto nel registro di destinazione (X1) nel file del registro.



#### 4.9.2 LDUR X1, [X2, offset]

Possiamo pensare a un'istruzione di carico come operativa in cinque passaggi:

1. Un'istruzione viene recuperata dalla memoria dell'istruzione e il PC viene incrementato.
2. Un valore di registro (X2) viene letto dal file del registro.
3. L'ALU calcola la somma del valore letto dal file del registro e i 9 bit estesi dal segno dell'istruzione (offset).
4. La somma dell'ALU viene utilizzata come indirizzo per la memoria dati.
5. I dati dell'unità di memoria vengono scritti nel file di registro (X1).

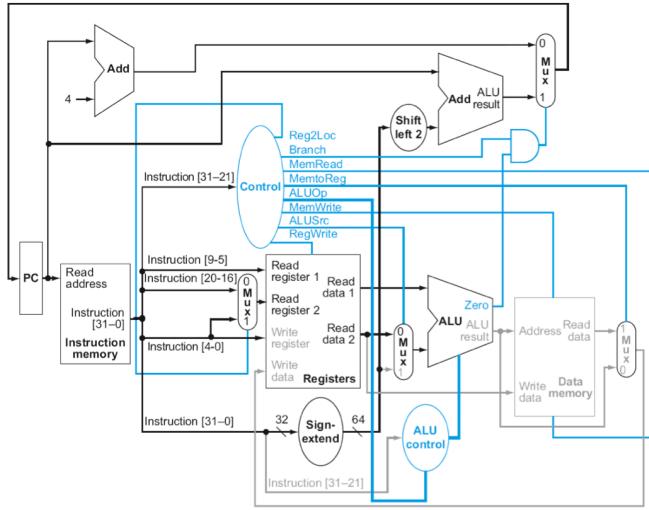


#### 4.9.3 CBZ X1, offset

Funziona in modo molto simile a un'istruzione in formato R, ma l'output ALU viene utilizzato per determinare se il PC è scritto con  $PC + 4$  o l'indirizzo di destinazione del ramo.

Possiamo pensare a quattro passaggi nell'esecuzione:

1. Un'istruzione viene recuperata dalla memoria dell'istruzione e il PC viene incrementato.
2. Il registro X1 viene letto dal file del registro usando i bit 4:0 dell'istruzione (Rt).
3. L'ALU passa il valore dei dati letto dal file del registro. Il valore del PC viene aggiunto al segno-esteso, 19 bit dell'istruzione (offset) vengono spostati a sinistra di due; il risultato è il ramo indirizzo target.
4. Le informazioni sullo stato zero dell'ALU vengono utilizzate per decidere quale risultato memorizzare nel PC.

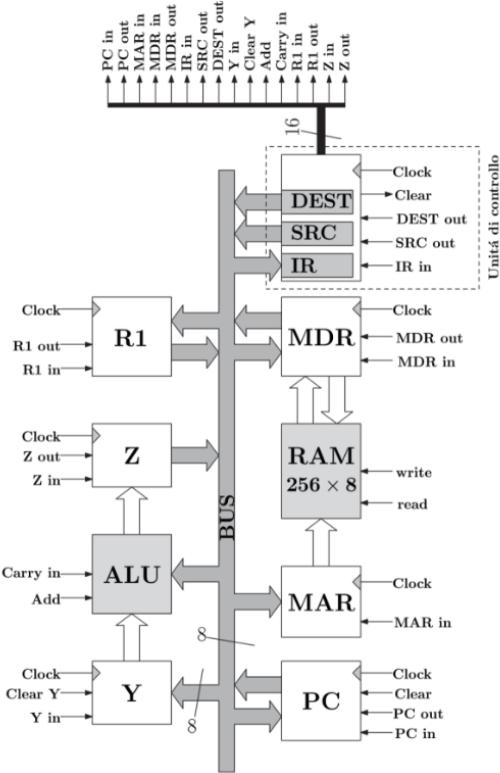


4.10 Perché un'implementazione a ciclo singolo non viene utilizzata oggi?

Anche se il design a ciclo singolo funzionerà correttamente, è troppo inefficiente per essere utilizzato nei design moderni. Si noti che il ciclo di clock deve avere la stessa lunghezza per ogni istruzione in questo progetto a ciclo singolo. Il percorso più lungo possibile nel processore determina il ciclo di clock; questo percorso è molto probabilmente un'istruzione di load, che utilizza cinque unità funzionali in serie: la memoria di istruzioni, il file di registro, l'ALU, la memoria dati e il file di registro. Sebbene il CPI sia 1, è probabile che le prestazioni complessive di un'implementazione a ciclo singolo siano scarse, dal momento che il ciclo di clock è troppo lungo. Storicamente, i primi computer con set di istruzioni molto semplici utilizzavano questa tecnica di implementazione. Tuttavia, se provassimo ad implementare l'unità in virgola mobile o un set di istruzioni con istruzioni più complesse, questo design a ciclo singolo non funzionerebbe affatto bene.



## 4.11 Strategie di controllo multiciclo



Consideriamo l'ipotetico processore della figura. L'unità di controllo decodifica ed esegue le istruzioni e aggiorna il contatore del programma (PC), recuperando la prossima istruzione. L'unità di controllo è qui composta da tre registri:

- IR ha l'OpCode dell'istruzione
- SRC può contenere un parametro incluso nell'IW o un indirizzo di memoria, spesso espresso in termini relativi come incremento del PC.
- DEST può contenere un indirizzo o il puntatore a un registro dove scrivere il risultato.

Dall'analisi dell'OpCode, l'unità di controllo deve fornire la sequenza di segnali di controllo necessari per

1. Recupera e aggiorna il PC
2. Esecuzione dell'istruzione attuale

Storicamente, sono stati seguiti due possibili approcci per l'unità di controllo:

- Approccio microprogrammato

- Approccio cablato
- I microcontrollori con vecchie architetture sono microprogrammati. Quelli più recenti, in particolare quelli RISC, sono cabati.

Due possibili strategie di clocking:

- Controllo multiciclo: Recupera, decodifica, esegui eseguito con più periodi di clock.
- Controllo a ciclo singolo: Recupera, decodifica, esegui eseguito in un unico periodo.

La strategia di controllo a ciclo singolo viene impiegata solo nei controlli cablati.

La strategia di controllo multiciclo viene utilizzata sia in tutti i controlli microprogrammati che in alcuni controlli cablati.

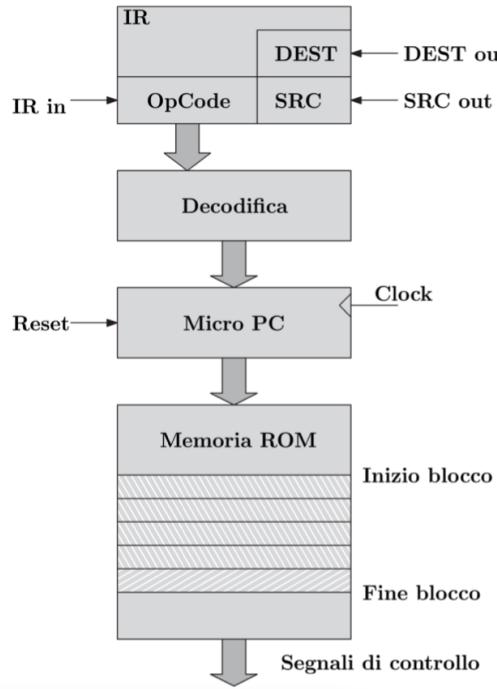
## 4.12 Controllori microprogrammati

È implementato con un'unità di controllo che replica la struttura di una semplice CPU con una memoria, un PC, un ALU, chiamato motore di microcodici. Ogni macro-istruzione corrisponde a un microcodice, composto da alcune parole. I microcodici sono memorizzati in una memoria ROM.

Due possibilità:

- Microprogrammazione orizzontale: l'unità di controllo esegue il codice rigorosamente in sequenza, avviando dall'indirizzo indicato dall'OpCode.
- Microprogrammazione verticale: sono possibili salti e consentono di ripetere segmenti di microcodice, cioè l'introduzione di micro-subroutine.

Il microcodice è composto da parole, i cui bit affermano/negano direttamente segnali di controllo specifici. La lunghezza delle parole del microcodice dipende dal numero di segnali di controllo.

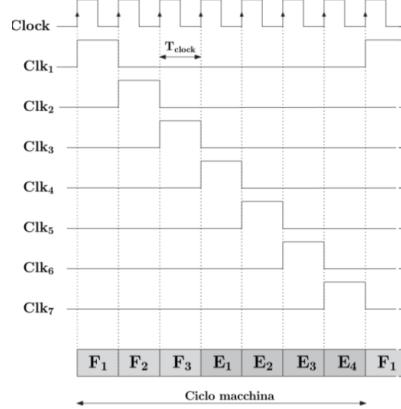
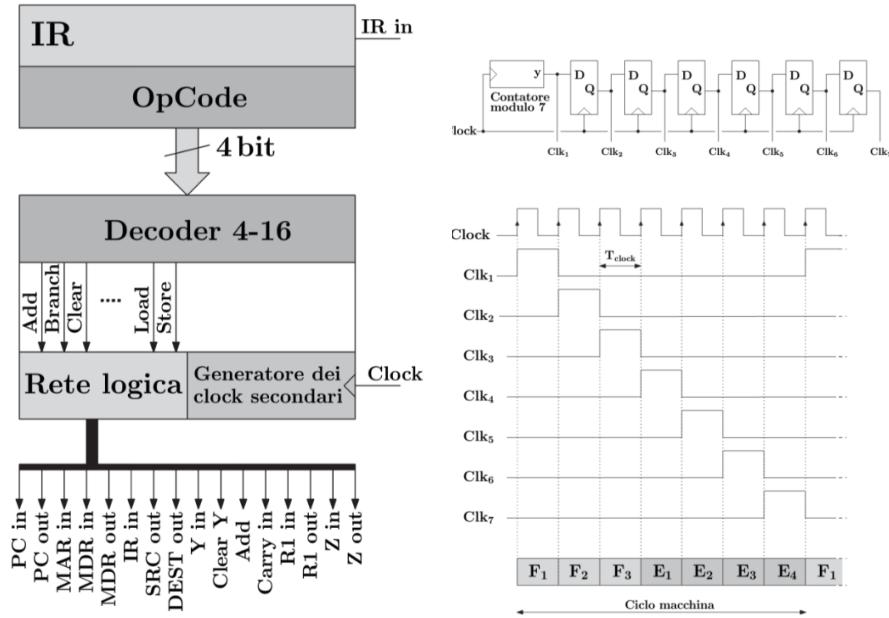


### 4.13 Controllore cablato

Il motore di microcodice viene sostituito da un circuito logico combinatorio che genera direttamente i segnali di controllo dall'OpCode dell'istruzione corrente, gestendo anche le temporizzazioni. Include un generatore di clock secondario, il cui scopo è la distribuzione temporale delle attivazioni del segnale di controllo.

Supponiamo che una singola istruzione venga eseguita in 7 periodi. Il generatore di clock secondario è un divisore di clock per 7 che genera 1 impulso ogni 7 periodi di clock, e alimenta un registro dei turni di 6 FlipFlops. Il decodificatore attiva una linea di uscita per ogni OpCode. La rete combinatoria, composta da porte AND e OR, alimenta le linee di controllo sulla base dell'OpCode e lo stato di clock secondario (da 1 a 7).

La soluzione fornisce una risposta molto rapida, con poca occupazione nell'area del silicio, ma manca di flessibilità, e potrebbe essere necessario cicli nop per gestire istruzioni più brevi.



## 4.14 Organizzazione Multi ciclo vs Singolo ciclo

Solo la disponibilità di più risorse consente una temporizzazione a ciclo singolo. Richiede almeno che la fase di recupero venga eseguita contemporaneamente per decodificare ed eseguire. Impone i seguenti requisiti di sistema:

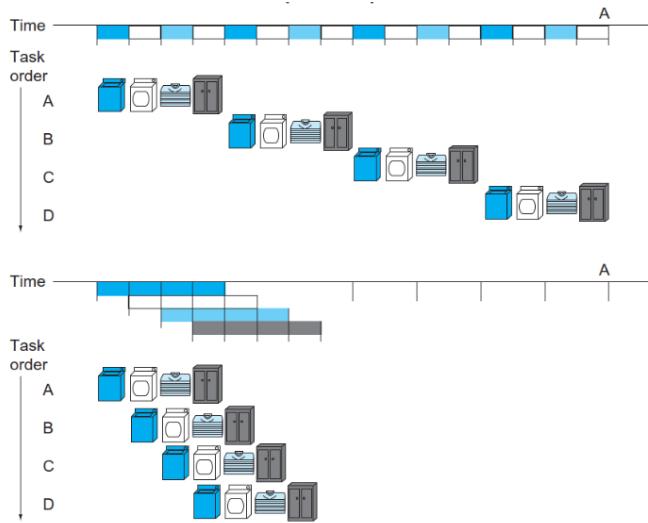
- Memoria dati separata e memoria di istruzioni
- ALU separata per l'incremento del PC
- Incremento flessibile del PC per la gestione dei salti senza l'intervento principale di ALU

A meno che il set di istruzioni non sia molto semplice, l'organizzazione a ciclo singolo è spesso inefficiente. È l'istruzione più onerosa che determina il periodo di clock. Al contrario, nell'organizzazione multiciclo, è l'unità funzionale più lenta (ALU o memoria) che determina il periodo minimo. È possibile combinare i vantaggi di entrambi, utilizzando un'organizzazione di pipeline.

## 4.15 Pipeline

La pipeline è una tecnica di implementazione in cui più istruzioni sono sovrapposte nell'esecuzione. Oggi, la pipelining è quasi universale.

**Facciamo una piccola analogia:** Esecuzione sovrapposta del bucato. Il parallelismo migliora le prestazioni



In questo caso abbiamo:

- 4 Carichi → Accelerazione =  $\frac{16}{7} = 2.3$
- Non-stop → Accelerazione =  $\frac{4n}{1n+3} \simeq 4 = \text{numero di fasi}$

Il paradosso della pipelining è che il tempo per l'elaborazione di un singolo carico di lavanderia non è più breve per il pipelining. Ma più carichi vengono elaborati all'ora. La pipeline migliora la produttività del nostro sistema di lavanderia.

#### 4.15.1 Prestazioni a Singolo ciclo vs Pipeline

Le istruzioni LEGv8 classicamente fanno cinque passaggi:

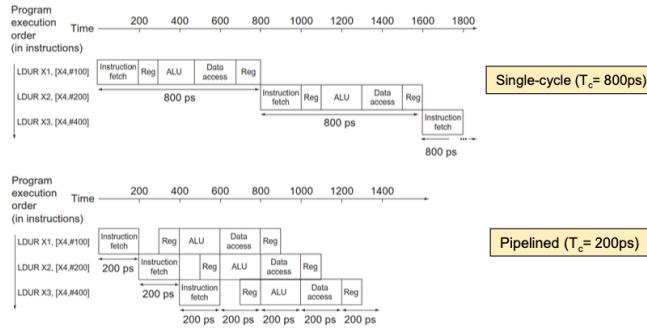
1. Recupera le istruzioni a memoria.
2. Leggi i registri e decodifica le istruzioni.
3. Esegui l'operazione o calcola un indirizzo.
4. Accedi a un operando nella memoria dati (se necessario).
5. Scrivi il risultato in un registro (se necessario).

Quindi, la pipeline del LEGv8 che consideriamo ha cinque fasi.

Limitiamo la nostra attenzione a sette istruzioni: registro di carico (LDUR), registro del negozio (STUR), aggiunta (ADD), sottrazione (SUB), AND (AND), OR (ORR) e confronto e diramazione a zero (CBZ). Supponiamo che il tempo per le fasi sia 100ps per la lettura o la scrittura del registro 200ps per altre fasi.



Classe di istruzioni	Fetch	Lettura registro	Operazione ALU	Accesso dati	Scrittura registro	Tempo Totale
Load (LDUR)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store (STUR)	200 ps	100 ps	200 ps	200 ps		700 ps
Formato R (ADD, SUB, AND, ORR)	200 ps	100 ps	200 ps		100 ps	600 ps
Branch (CBZ)	200 ps	100 ps	200 ps			500 ps



## 4.16 Accelerazione della Pipeline

Se le fasi sono perfettamente bilanciate, allora

$$Tempo tra le istruzioni_{pipeline} = \frac{Tempo tra le istruzioni_{Non pipeline}}{Numero di fasi della pipeline}$$

In condizioni ideali e con un gran numero di istruzioni, l'accelerazione della pipeline è approssimativamente uguale al numero di stadi di pipeline. Se le fasi non sono bilanciate, la velocità è inferiore. Inoltre, vedremo che la pipeline comporta un po' di spese generali. In realtà, nel nostro caso il tempo di esecuzione totale per le tre istruzioni è di 1400 ps contro 2400 ps. Ma se aggiungiamo 1.000.000 di istruzioni:  $\frac{800.000.400\text{ ps}}{200.001.400\text{ ps}} \simeq \frac{800}{200} \simeq 4$  La pipeline migliora le prestazioni aumentando la produttività delle istruzioni, in contrasto con la diminuzione del tempo di esecuzione di una singola istruzione.

## 4.17 Progettazione di set di istruzioni per la pipeline

LEGv8 è stato progettato per l'esecuzione in pipeline:

- Tutte le istruzioni sono a 32 bit: Più facile da recuperare e decodificare in un ciclo, c.f. x86: istruzioni da 1 a 17 byte
- Pochi e regolari formati di istruzioni: Può decodificare e leggere i registri in un unico passaggio
- Gli operandi di memoria appaiono solo nei carichi o nei negozi: Possiamo usare la fase di esecuzione per calcolare l'indirizzo di memoria e quindi accedere alla memoria nella fase successiva.



#### 4.17.1 Pericoli della pipeline

Ci sono situazioni nel pipelining in cui l'istruzione successiva non può essere eseguita nel seguente ciclo di clock. Questi eventi sono chiamati pericoli e ce ne sono tre diversi tipi:

- **Pericoli di struttura:** Quando un'istruzione pianificata non può essere eseguita nel ciclo di clock corretto perché l'hardware non supporta la combinazione di istruzioni impostate per l'esecuzione.
- **Pericolo di dati:** Quando un'istruzione pianificata non può essere eseguita nel ciclo di clock corretto perché i dati che sono necessario per eseguire le istruzioni non sono ancora disponibili.
- **Pericolo di controllo** (chiamato anche pericolo di branch): Quando l'istruzione corretta non può essere eseguita nel ciclo di clock della pipeline corretto perché l'istruzione che è stata recuperata non è quella necessaria; cioè, il flusso degli indirizzi di istruzioni non è quello che la pipeline si aspettava.

#### 4.17.2 Pericoli strutturali

Quando un'istruzione pianificata non può essere eseguita nel ciclo di clock corretto perché l'hardware non supporta la combinazione di istruzioni impostate per l'esecuzione. Il set di istruzioni LEGv8 è stato progettato per essere utilizzato in pipeline, rendendo abbastanza facile per i progettisti evitare rischi strutturali durante la progettazione di una pipeline. Supponiamo, tuttavia, di avere una sola memoria invece di due:

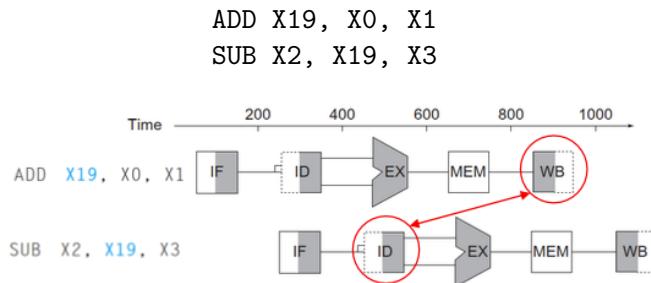
- Il load/store richiede l'accesso ai dati
- Il recupero delle istruzioni dovrebbe bloccarsi per quel ciclo (creerebbe una "bolla" della pipeline)

Quindi, i percorsi dati della pipeline richiedono istruzioni/memorie di dati separate (O cache di istruzioni/dati separate)

#### 4.17.3 Pericoli di dati

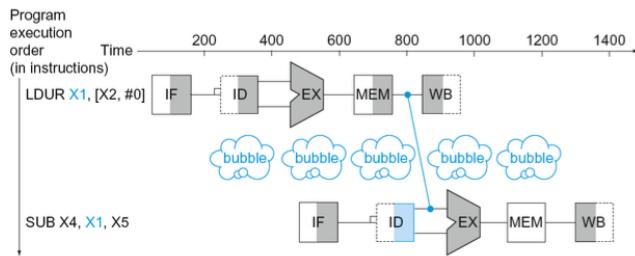
I rischi di dati derivano dalla dipendenza di un'istruzione da una precedente che è ancora in lavorazione.

Ad esempio:





**Pericolo di dati per l'uso del Load** L'inoltro non può impedire lo stallo all'interno della pipeline. Supponiamo che la prima istruzione sia stata un carico di X1 invece di un'aggiunta. I dati desiderati sarebbero disponibili solo dopo la quarta fase della prima istruzione nella dipendenza, che è troppo tardi per l'ingresso del terzo stadio di SUB.

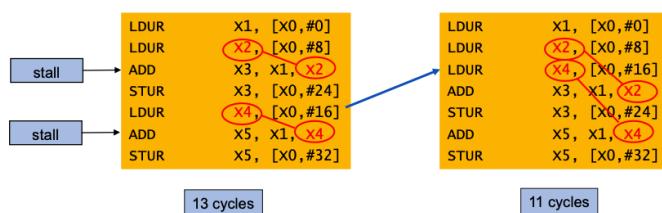


Il pericolo di dati per l'uso del load è una forma specifica di pericolo di dati in cui i dati caricati da un'istruzione di carico non sono ancora diventati disponibili quando sono necessari da un'altra istruzione. Lo **stallo della pipeline** viene chiamato anche **bolla**. Uno stallo viene avviato per risolvere un pericolo.

**Riordinare il codice per evitare stalli della pipeline** Possiamo prevenire il pericolo di utilizzo dei dati di carico riordinando il codice per evitare l'uso del risultato del carico nelle istruzioni successive.

Codice C per

$$\begin{aligned} A &= B + E; \\ C &= B + F; \end{aligned}$$



#### 4.17.4 Pericoli di controllo

Pericolo di controllo chiamato anche pericolo di branch. Quando l'istruzione corretta non può essere eseguita nel corretto ciclo di clock della pipeline perché l'istruzione che è stata recuperata non è quella necessaria; cioè, il flusso degli indirizzi di istruzioni non è ciò che la pipeline si aspettava.

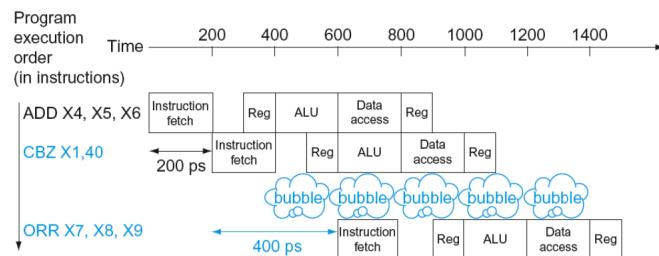
Nell'istruzione del branch condizionale, dobbiamo iniziare a recuperare l'istruzione seguendo il branch nel seguente ciclo di clock. Tuttavia, la pipeline non può assolutamente sapere quale dovrebbe essere la prossima istruzione, dal momento che ha appena ricevuto l'istruzione del branch dalla memoria!

Nella pipeline LEGv8:



- Dobbiamo confrontare i registri e calcolare l'obiettivo all'inizio della pipeline.
- Supponiamo di aver inserito abbastanza hardware extra in modo da poter testare un registro, calcolare l'indirizzo della filiale e aggiornare il PC durante la seconda fase della pipeline (fase ID).

**Stallo sui Branch** Una possibile soluzione è quella di bloccarsi immediatamente dopo aver recuperato il branch, aspettando che la pipeline determini l'esito del branch.



**Performance:** Stimare l'impatto sui cicli di clock per istruzione (CPI) dello stallo sui branch. Supponiamo che tutte le altre istruzioni abbiano un CPI di 1.

I branch condizionali rappresentano il 17% delle istruzioni eseguite in SPECint2006. Poiché le altre istruzioni (83%) hanno un CPI di 1 e i branch condizionali hanno preso un ciclo di clock in più per lo stallo, quindi vedremo un CPI di

$$CPI : 0.83 * 1 + 0.17 * 2 = 1.17$$

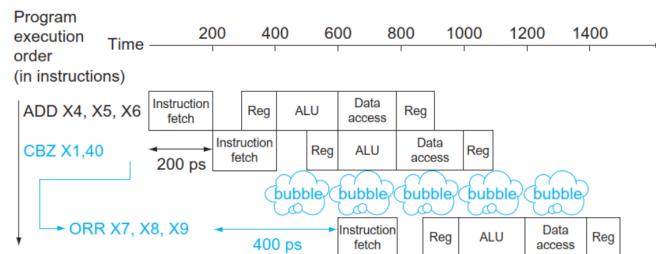
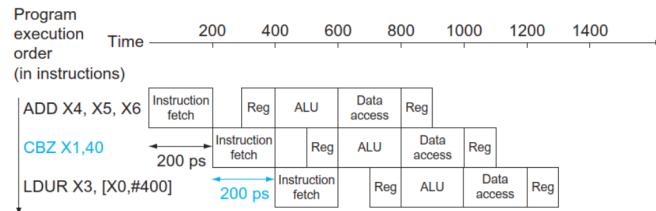
Quindi un rallentamento di 1,17 rispetto al caso ideale.

**Predizione dei branch** Se non riusciamo a risolvere il branch nella seconda fase, come spesso accade per le pipeline più lunghe, allora vedremmo un rallentamento ancora più grande che se ci bloccassimo sui branch condizionali. Il costo di questa opzione è troppo alto per la maggior parte dei computer da utilizzare e motiva una seconda soluzione al rischio di controllo: prevedere l'esito del ramo.

Questa opzione non rallenta la pipeline quando prevede correttamente.

Quando sbagli: devi rifare il load che non è stato fatto mentre si cercava di prevedere la decisione (con la creazione di una bolla).

Un semplice approccio è prevedere sempre che i branch condizionali non saranno affrontati. Quando hai ragione, la pipeline procede a tutta velocità. Solo quando vengono prelevati rami condizionali, la pipeline si blocca.



**Predizione dei branch più realistica** Una versione più sofisticata della previsione dei branch avrebbe alcuni branch condizionali previsti come presi e alcuni come non presi.

- **Previsione del branch statico:**

- Basato sul tipico comportamento del branch
- Esempio: branch loop e if-statement: Prevedi i branch presi in precedenza, prevedi i branch in avanti non presi

- **Previsione dinamica dei branch**

- L'hardware misura il comportamento effettivo del branch. Ad esempio, registrare la storia recente di ciascun branch
- Supponiamo che il comportamento futuro continui la tendenza. Quando sbaglio, fermati durante il recupero e aggiorna la cronologia

**Terzo approccio: branch ritardato** C'è un terzo approccio al pericolo di controllo, chiamato branch ritardato. Utilizzato in MIPS, TI C54 e molti altri processori. Il branch ritardato esegue sempre l'istruzione sequenziale successiva, con il branch che si svolge dopo quel ritardo nell'istruzione. È nascosto al programmatore del linguaggio assembly MIPS perché l'assembler può organizzare automaticamente le istruzioni per ottenere il comportamento del branch desiderato dal programmatore.

Il software MIPS inserirà un'istruzione immediatamente dopo l'istruzione del branch ritardata che non è influenzata dal branch.

Un branch preso cambia l'indirizzo dell'istruzione che segue l'istruzione sicura.



#### 4.17.5 Riepilogo della pipeline: il quadro generale

La pipeline aumenta il numero di istruzioni di esecuzione simultanea e la velocità con cui le istruzioni vengono avviate e completate.

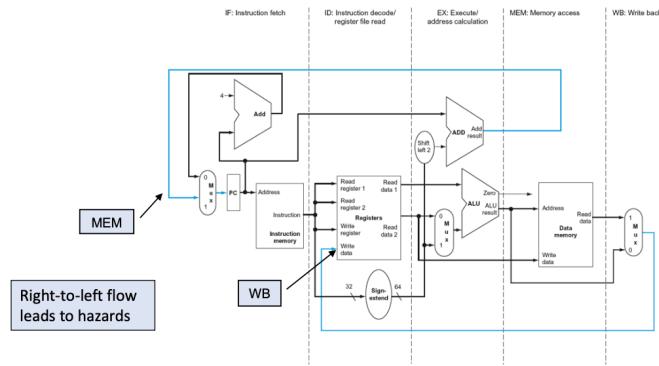
La pipeline non riduce il tempo necessario per completare un'istruzione individuale, cioè la latenza. Ad esempio, la pipeline a cinque stadi richiede ancora cinque cicli di clock per completare l'istruzione.

La pipeline migliora il throughput delle istruzioni piuttosto che il tempo di esecuzione delle singole istruzioni.

I set di istruzioni possono rendere la vita più difficile o più semplice per i progettisti di pipeline, che devono già far fronte con rischi strutturali, di controllo e di dati.

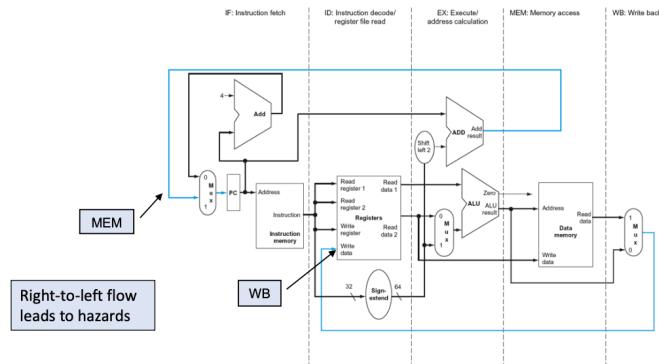
La previsione e l'inoltro dei branch aiutano a rendere veloce un computer pur ottenendo le risposte giuste.

### 4.18 Pipeline Datapath in LEGv8



#### 4.18.1 Registri di pipeline

Ha bisogno di registri tra le fasi per contenere le informazioni prodotte nel ciclo precedente



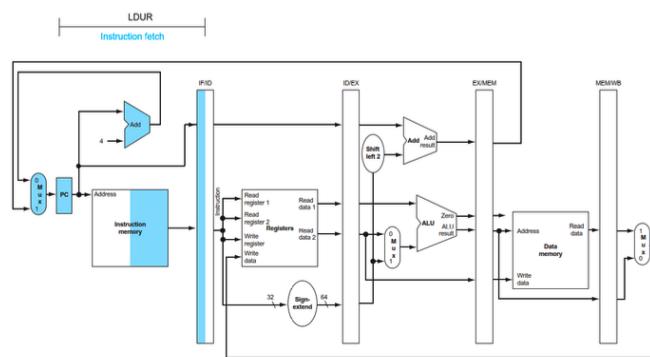


#### 4.18.2 Operazioni della pipeline

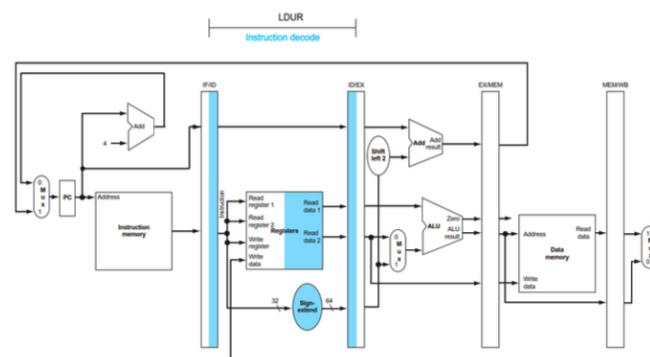
Flusso ciclo per ciclo di istruzioni attraverso il percorso dati in pipeline

- Diagramma della pipeline "A ciclo singolo"
  - Mostra l'utilizzo della pipeline in un unico ciclo
  - Evidenzia le risorse utilizzate
- c.f. diagramma "ciclo multi-clock"  
Grafico del funzionamento nel tempo

#### 4.18.3 IF per Load, Store, ...

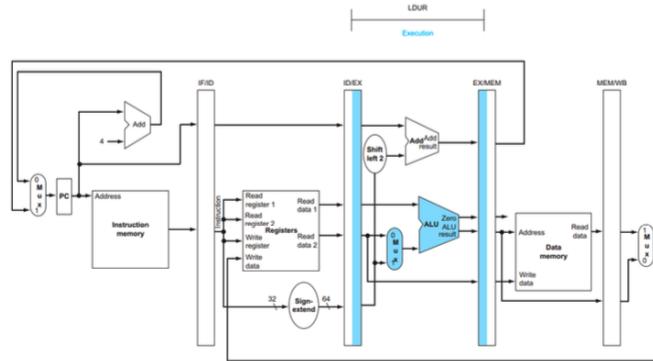


#### 4.18.4 ID per Load, Store, ...

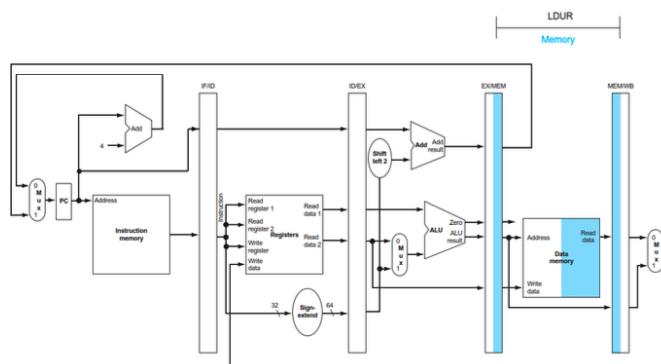




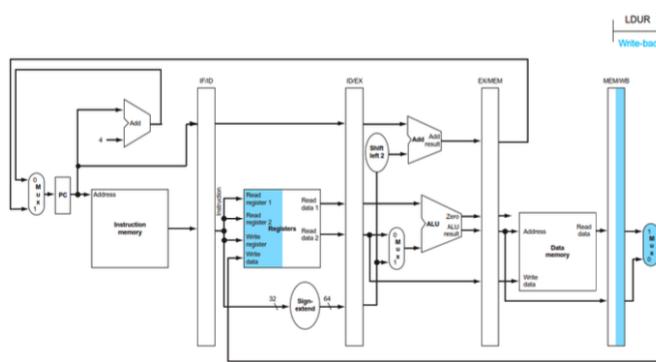
#### 4.18.5 EX per Load

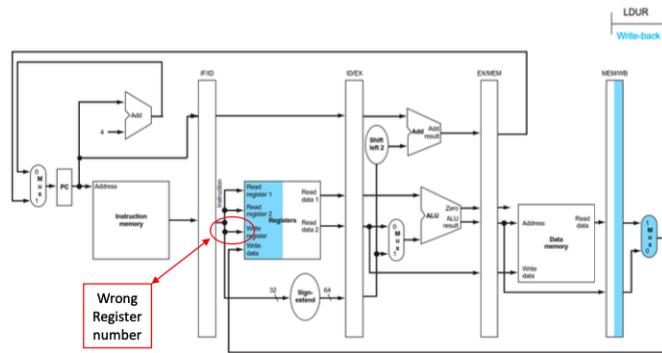


#### 4.18.6 MEM per Load

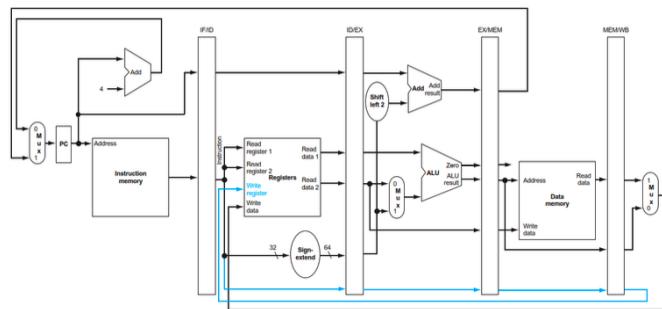


#### 4.18.7 WB per Load

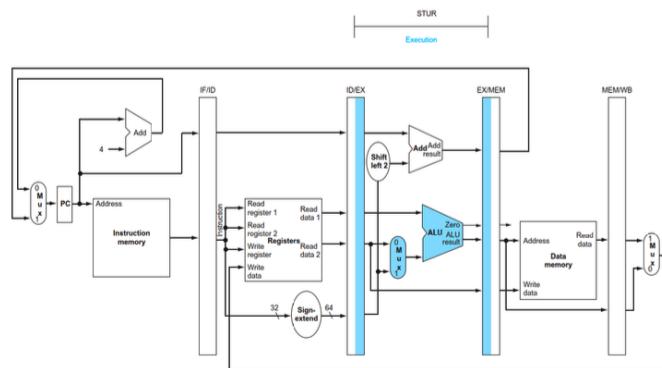




#### 4.18.8 Datapath corretto per il Load

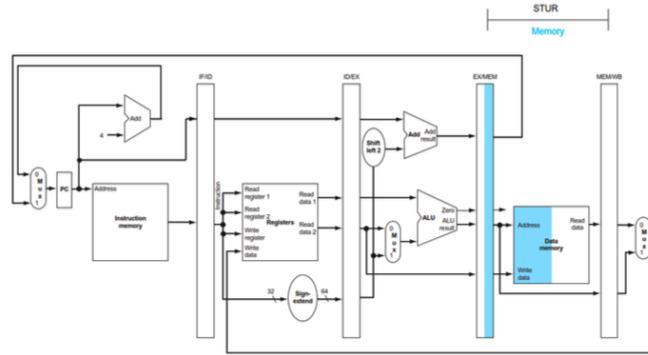


#### 4.18.9 EX per Store

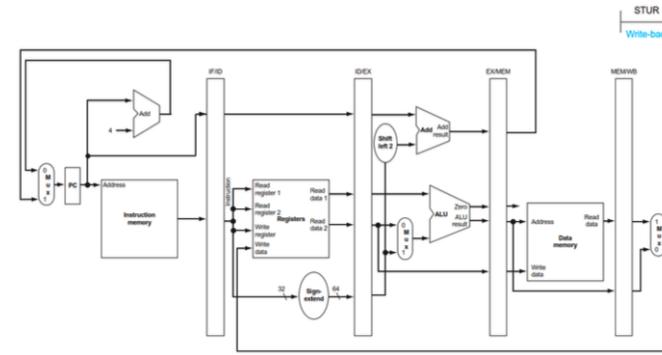




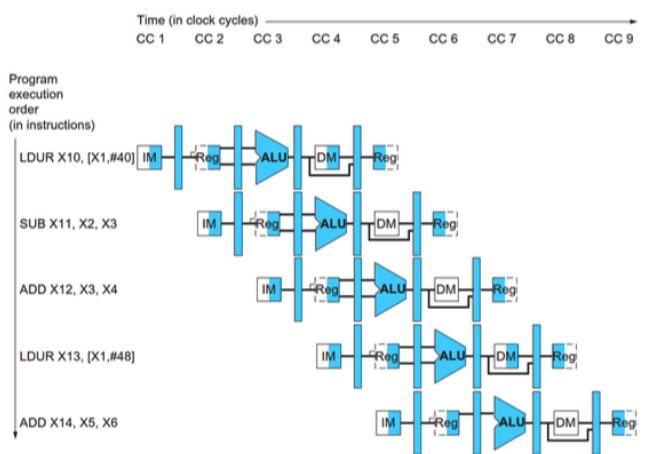
#### 4.18.10 MEM per Store



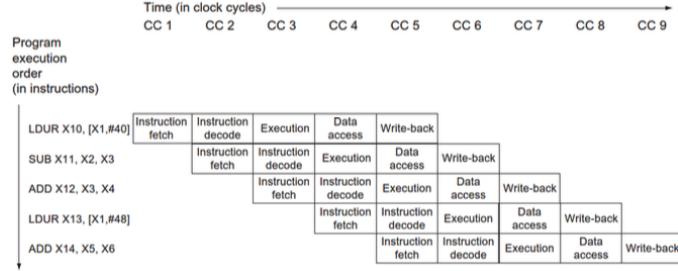
#### 4.18.11 WB per Store



### 4.19 Diagramma della pipeline Multi ciclo

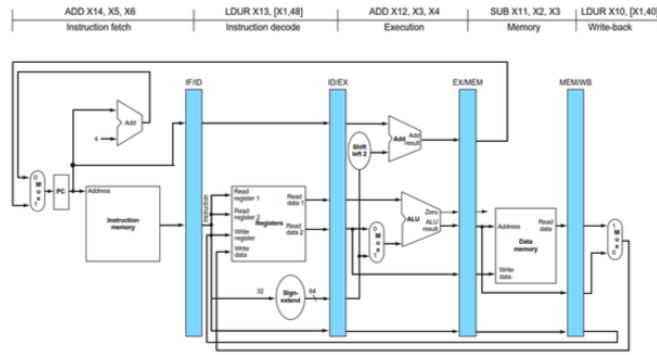


Forma tradizionale:

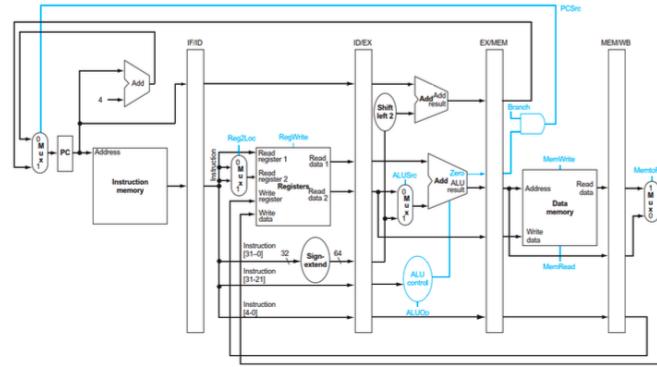


## 4.20 Diagramma della pipeline a Singolo ciclo

Stato della pipeline in un dato ciclo:

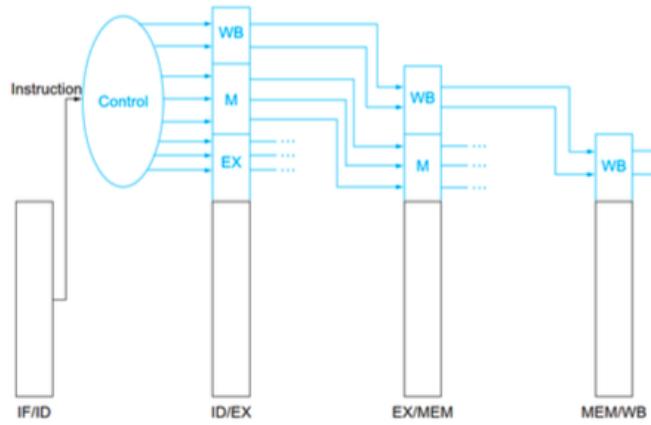


## 4.21 Controllo pipeline (semplificato)



### 4.21.1 Controllo pipeline

Segnali di controllo derivati dall'istruzione. Come nell'implementazione a ciclo singolo



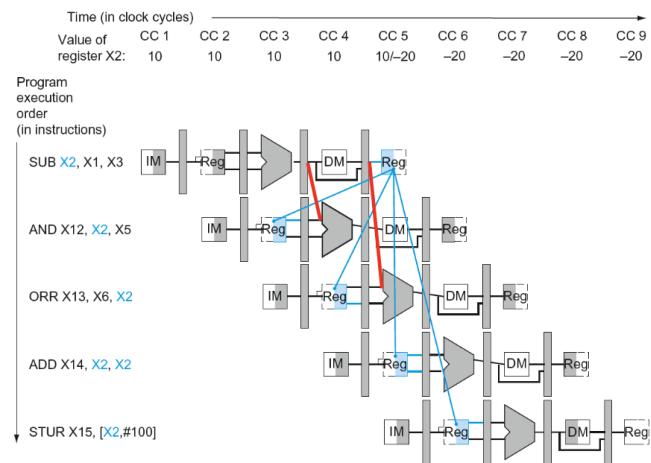
## 4.22 Pericoli di dati nelle istruzioni ALU

Consideriamo la sequenza:

```
SUB X2, X1, X3 // X2 scritto da SUB
AND X12, X2, X5 // Il primo operando (X2) dipende dal SUB
OR X13, X6, X2 // Il secondo operando (X2) dipende dal SUB
ADD X14, X2, X2 // Entrambi gli operandi dipendono dal SUB
STUR X15, [X2, #100] // La Base dipende dal SUB
```

Possiamo risolvere i pericoli con il forwarding. Come rileviamo quando fare il forward?

### 4.22.1 Dipendenze & Forwarding



### 4.22.2 Rilevare la necessità di forward

Passa i numeri del registro lungo la pipeline. Ad esempio, ID/EX.RegisterRs = numero di registro per Rs situato nel registro della pipeline ID/EX



I numeri di registro degli operandi ALU nella fase EX sono forniti da ID/EX.RegisterRn1, ID/EX.RegisterRm2

Pericoli di dati quando

1.  $EX/MEM.RegisterRd = ID/EX.RegisterRn1$  } Forward dal registro  
 $EX/MEM.RegisterRd = ID/EX.RegisterRm2$  } EX/MEM della pipeline
2.  $MEM/WB.RegisterRd = ID/EX.RegisterRn1$  } Forward dal registro  
 $MEM/WB.RegisterRd = ID/EX.RegisterRm2$  } MEM/WB della pipeline

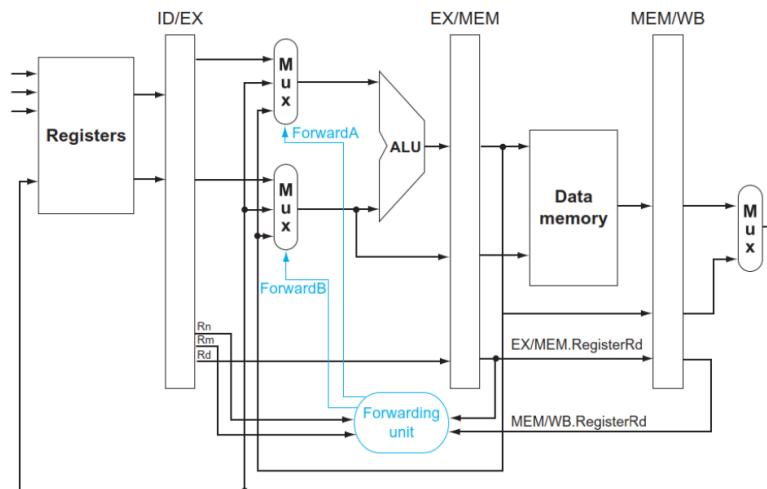
Ma solo se l'istruzione di forward scriverà a un registro!

$EX/MEM.RegWrite, MEM/WB.RegWrite$

E solo se Rd per quell'istruzione non è XZR

$EX/MEM.RegisterRd \neq 31, MEM/WB.RegisterRd \neq 31$

#### 4.22.3 Percorsi di Forwarding



#### 4.22.4 Condizioni di Forwarding

Controllo MUX	Fonte	Spiegazione
ForwardA = 00	ID/EX	Il primo operando della ALU arriva dal file registri
ForwardA = 10	EX/MEM	Il primo operando viene inoltrato dal risultato ALU precedente
ForwardA = 01	MEM/WB	Il primo operando viene inoltrato dalla memoria dati o da un risultato ALU precedente
ForwardB = 00	ID/EX	Il secondo operando della ALU arriva dal file registri
ForwardB = 10	EX/MEM	Il secondo operando viene inoltrato dal risultato ALU precedente
ForwardB = 01	MEM/WB	Il secondo operando viene inoltrato dalla memoria dati o da un risultato ALU precedente

#### 4.22.5 Condizioni di rilevamento

1. Pericolo EX

```
if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd != 31)
and (EX/MEM.RegisterRd = ID/EX.RegisterRn1)) ForwardA = 10

if (EX/MEM.RegWrite
and (EX/MEM.RegisterRd != 31)
and (EX/MEM.RegisterRd = ID/EX.RegisterRm2)) ForwardB = 10
```

2. Pericolo MEM:

```
if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd != 31)
and (MEM/WB.RegisterRd = ID/EX.RegisterRn1)) ForwardA = 01

if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd != 31)
and (MEM/WB.RegisterRd = ID/EX.RegisterRm2)) ForwardB = 01
```

#### 4.22.6 Doppio rischio di dati

Consideriamo la sequenza:

```
ADD X1, X1, X2
ADD X1, X1, X3
ADD X1, X1, X4
```

Entrambi i pericoli si verificano. Vogliamo utilizzare i dati più recenti

Rivedere le condizioni di pericolo MEM. Solo forward se la condizione di pericolo EX non è vera.

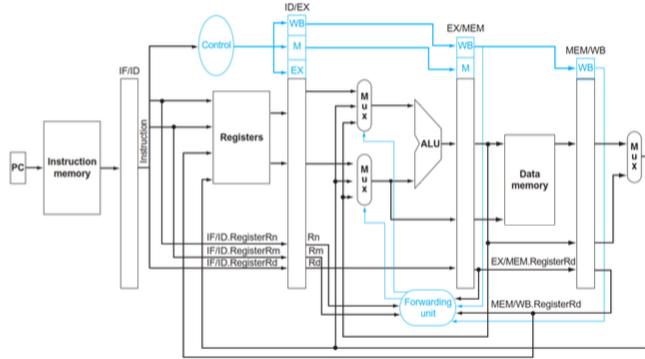
Revisione pericolo MEM:

```
if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd != 31)
and not(EX/MEM.RegWrite and (EX/MEM.RegisterRd != 31)
       and(EX/MEM.RegisterRd != ID/EX.RegisterRn1))
and (MEM/WB.RegisterRd = ID/EX.RegisterRn1)) ForwardA = 01

if (MEM/WB.RegWrite
and (MEM/WB.RegisterRd != 31)
and not(EX/MEM.RegWrite and (EX/MEM.RegisterRd != 31)
       and(EX/MEM.RegisterRd != ID/EX.RegisterRm2))
and (MEM/WB.RegisterRd = ID/EX.RegisterRm2)) ForwardB = 01
```



#### 4.22.7 Datapath con Forwarding



#### 4.22.8 Rilevamento dei pericoli per l'uso del load

Un caso in cui il forwarding non può salvare il giorno è quando un'istruzione tenta di leggere un registro seguendo un'istruzione di load che scrive lo stesso registro. In tal caso è necessario bloccare la pipeline.

Controlliamo questa condizione quando l'istruzione di utilizzo viene decodificata in fase ID. I numeri di registro degli operandi ALU nella fase ID sono forniti da

$$\text{IF/ID.RegisterRn1, IF/ID.RegisterRm2}$$

Abbiamo un pericolo di carico-uso quando

$$\text{ID/EX.MemRead and } ((\text{ID/EX.RegisterRd} = \text{IF/ID.RegisterRn1}) \text{ or }$$

Se rilevato, dobbiamo bloccarci e inserire la bolla.

#### 4.22.9 Come mettere in stallo la pipeline

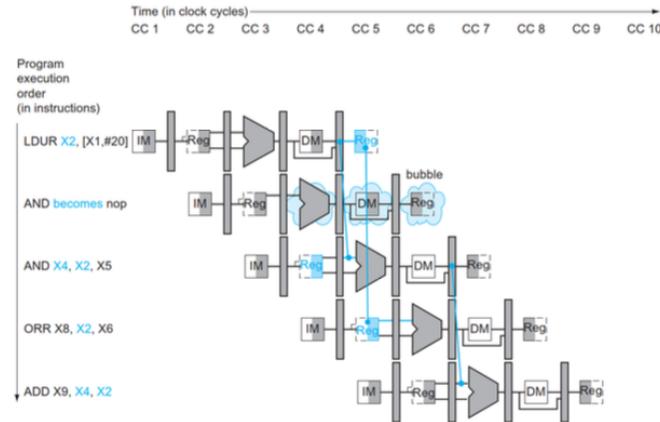
Forza i valori di controllo nel registro ID/EX a 0. EX, MEM e WB do nop (nessuna operazione).

Impedire l'aggiornamento del registro PC e IF/ID. L'istruzione decodificata viene decodificata di nuovo, Le istruzioni seguenti vengono recuperate di nuovo

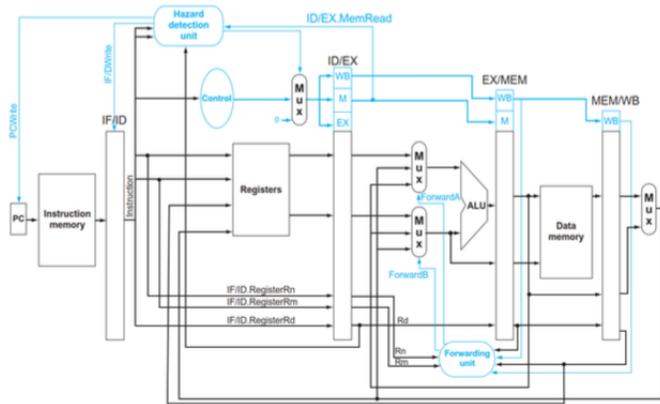
Lo stallo a ciclo singolo consente a MEM di leggere i dati per LDUR. Può successivamente inoltrare alla fase EX



#### 4.22.10 Pericolo di utilizzo del carico



#### 4.22.11 Percorso dati con rilevamento dei pericoli



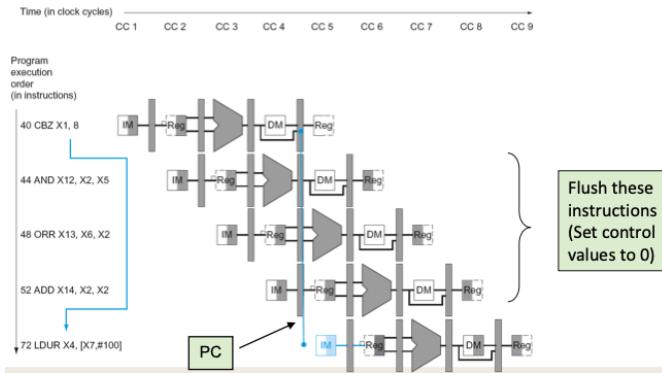
### 4.23 Stalli e Performance

Sebbene il compilatore si basi generalmente sull'hardware per risolvere i pericoli e quindi garantire una corretta esecuzione, il compilatore deve comprendere la pipeline per ottenere le migliori prestazioni. Altrimenti, stalli imprevisti ridurranno le prestazioni del codice compilato.

Gli stalli riducono le prestazioni, ma sono necessari per ottenere risultati corretti. Il compilatore può organizzare il codice per evitare pericoli e stalli, richiede la conoscenza della struttura della pipeline.

### 4.24 Pericoli nei branch

Se l'esito del branch è determinato in MEM



#### 4.24.1 Ridurre il ritardo del branch

Se spostiamo l'esecuzione condizionale del branch all'inizio della pipeline, è necessario svuotare meno istruzioni. Spostare la decisione del branch richiede che si verifichino due azioni prima:

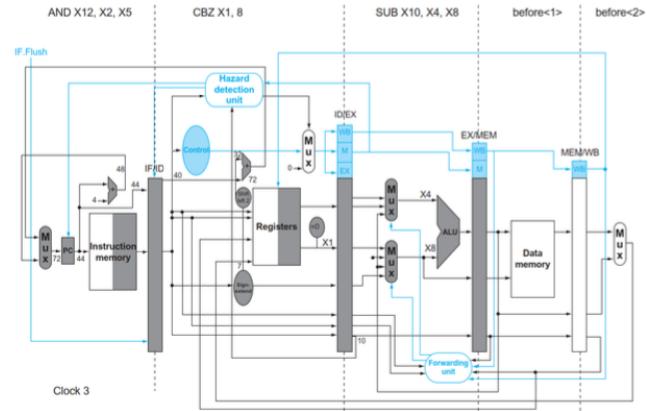
- Calcolare l'indirizzo di destinazione del branch
- Calutare la decisione del branch

Spostare il test di derivazione nella fase ID implica un ulteriore hardware di inoltro e rilevamento dei pericoli. Durante l'ID, dobbiamo decodificare l'istruzione, decidere se è necessario un bypass per l'unità di test zero e completare il test zero in modo che se l'istruzione è un branch, possiamo impostare il PC sull'indirizzo di destinazione del ramo. Gli operandi sorgente bypassati di un ramo possono provenire dai fermi della pipeline ALU/MEM o MEM/WB.

Poiché il valore in un confronto di rami è necessario durante l'ID ma può essere prodotto più tardi nel tempo, è possibile che si verifichi un rischio di dati e sarà necessario uno stall. Se un'istruzione ALU immediatamente precedente a un branch produce l'operando per il test nel branch condizionale, sarà necessario uno stall. Se un carico viene immediatamente seguito da un branch condizionale che dipende dal risultato del carico, saranno necessari due cicli di stallo.

#### 4.24.2 Esempio

```
36: SUB X10, X4, X8
40: CBZ X1, X3, 8
44: AND X12, X2, X5
48: ORR X13, X2, X6
52: ADD X14, X4, X2
56: SUB X15, X6, X7
...
72: LDUR X4, [X7,#50]
```



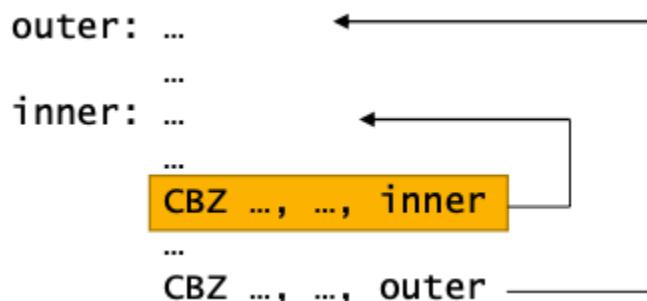
## 4.25 Previsione dinamica del branch

Nelle pipeline più profonde e superscalari, la sanzione dei branch è più significativa.  
Usando la previsione dinamica

- Buffer di previsione dei branch (aka tabella della cronologia dei branch)
- Indicizzato da indirizzi di istruzione di branch recenti
- Risultato degli store (preso/non preso)
- Per eseguire un branch
  - Tabella di controllo, aspettato lo stesso risultato
  - Inizia a recuperare dalla caduta o dal bersaglio
  - Se sbaglio, pulire la pipeline e invertire la previsione

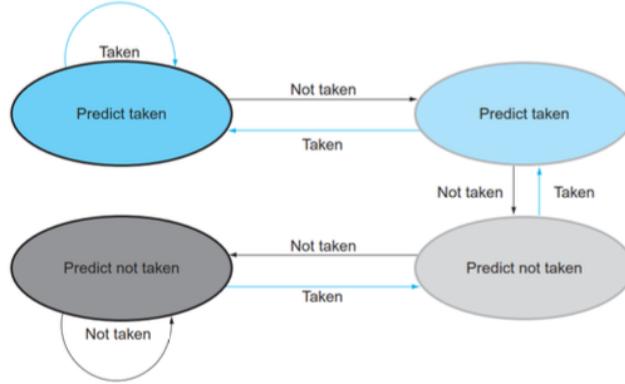
### 4.25.1 Predictor a 1 bit: Shortcoming

Branch ad anello interno mal previsti due volte!



#### 4.25.2 Predictor a 2 bit

Cambia solo la previsione su due previsioni errate successive



#### 4.25.3 Tampone target di branch e altri predittori di branch

Anche con il predittore, è ancora necessario calcolare l'indirizzo target. Penalità di 1 ciclo per un branch preso. **Branch Target buffer**:

- Cache degli indirizzi di destinazione
- Indicizzato dal PC quando le istruzioni vengono recuperate
- Se il colpo e l'istruzione è prevista dal branch, può recuperare immediatamente il target

**Correlating predictor:** Un predittore di branch che combina il comportamento locale di un particolare branch e le informazioni globali sul comportamento di un numero recente di branch eseguiti. **Tournament branch predictor:** Un predittore di branch con più previsioni per ogni branch e un meccanismo di selezione che sceglie quale predittore abilitare per un dato branch.

### 4.26 Exceptions and Interrupts

Eventi inaspettati che richiedono un cambiamento nel flusso di controllo, diversi ISA usano i termini in modo diverso

**Eccezione:** Sorge all'interno della CPU (ad esempio, opcode indefinito, overflow, syscall)

**Interrupt:** Da un controller I/O esterno

Affrontarli senza sacrificare le prestazioni è difficile

Type of event	From where?	ARMv8 Terminology
System reset	External	Exception
I/O device request	External	Interrupt
Invoke the operating system from user program	Internal	Exception
Floating-point arithmetic overflow or underflow	Internal	Exception
Using an undefined instruction	Internal	Exception
Hardware malfunctions	Either	Exception or Interrupt

#### 4.26.1 Gestione delle eccezioni

Salva il PC dalle istruzioni offensive (o interrotte). In LEGv8: Registro Link Eccettivo (ELR) A 64 Bit

Salva l'indicazione del problema. In LEGv8: Registro della sindrome da eccezione (ESR) a 32 bit, ad esempio:

- 8 che rappresentano un'istruzione indefinita
- 10 che rappresentano l'overflow o l'underflow aritmetico
- 12 che rappresentano un malfunzionamento hardware

#### 4.26.2 Un meccanismo alternativo

Interruzioni vettoriali: L'indirizzo del gestore a cui saltiamo è determinato dalla causa Indirizzo vettoriale di eccezione da aggiungere a un registro di base della tabella vettoriale:

- Motivo sconosciuto:  $00\ 0000_{two}$
- Trabocco:  $10\ 1100_{two}$
- ...:  $11\ 1111_{two}$

Le istruzioni affrontano anche l'interruzione, o passa al gestore reale

#### 4.26.3 Azioni del gestore

Leggi la causa e trasferisci al gestore pertinente, determinare l'azione richiesta. Se riavviabile:

- Intraprendere azioni correttive
- usa ELR per tornare al programma

Altrimenti

- Termina il programma
- Segnala un errore usando ELR, causa, ...

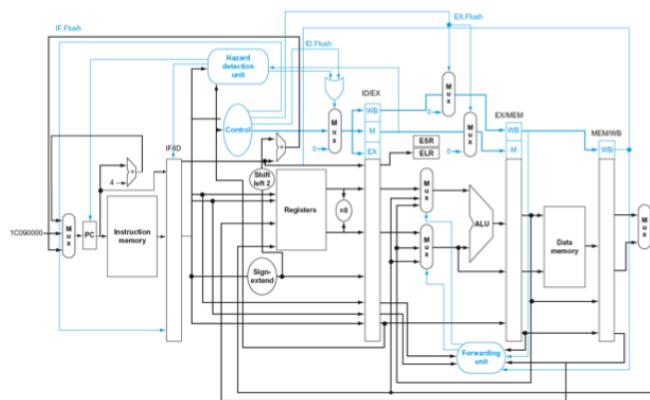
## 4.27 Exceptions in una Pipeline

Un'altra forma di pericolo di controllo. Considera l'overflow o la multifunzione hardware al punto add in EX

ADD X1, X2, X1

- Impedire che X1 venga picchiato
- Istruzioni precedenti complete
- Aggiungi di fila le istruzioni successive
- Imposta i valori del registro ESR ed ELR
- Controllo del trasferimento al gestore

Simile al branch mal predetto, usa gran parte dello stesso hardware



### 4.27.1 Proprietà

Eccezioni riavviablei

- La pipeline può pulire le istruzioni
- Il gestore esegue, quindi torna all'istruzione
- Refetched ed eseguito da zero

PC salvato nel registro ELR

- Identifica l'istruzione che causa
- In realtà PC + 4 viene salvato
- Il gestore deve regolare



#### 4.27.2 Esempio

Eccezione di ADD in

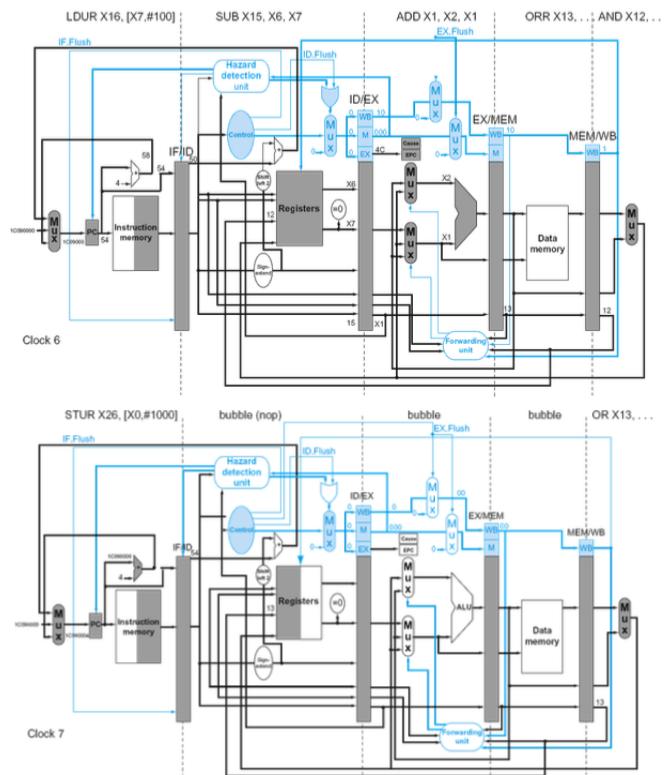
```
40 SUB X11, X2, X4
44 AND X12, X2, X5
48 ORR X13, X2, X6
--> 4C ADD X1, X2, X1 <--
50 SUB X15, X6, X7
54 LDUR X16, [X7,#100]
...

```

Handler

```
80000180 STUR X26, [X0,#1000]
80000184 STUR X27, [X0,#1008]
...

```



#### 4.27.3 Eccezioni multiple

La pipeline si sovrappone a più istruzioni. Potrebbe avere più eccezioni contemporaneamente

Approccio semplice: affronta l'eccezione dalle prime istruzioni



- Istruzioni successive di fila
- Eccezioni “precise”

In pipeline complesse

- Istruzioni multiple emesse per ciclo
- Completamento fuori ordine
- Mantenere eccezioni precise è difficile!

#### 4.27.4 Eccezioni imprecise

Basta fermare la pipeline e salvare lo stato comprese le cause di eccezione  
Lascia che il gestore lavori

- Quali istruzioni avevano eccezioni
- Quale completare o pulire, può richiedere il completamento "manuale"

Semplifica l'hardware, ma il software del gestore più complesso  
Non fattibile per pipeline complesse fuori ordine a più problemi

### 4.28 Interfaccia HW/SW

L'hardware e il sistema operativo devono funzionare insieme in modo che le eccezioni si comportino come ci si aspetterebbe.

Il contratto hardware normalmente è necessario che:

- Fermare le istruzioni offensive a metà strada
- Lascia che tutte le istruzioni precedenti siano complete
- Pulire tutte le seguenti istruzioni
- Impostare un registro per mostrare la causa dell'eccezione
- Salvare l'indirizzo dell'istruzione incriminata e poi diramare a un indirizzo prestabilito

Il contratto del sistema operativo è quello di esaminare la causa dell'eccezione e agire in modo appropriato

- Per un'istruzione indefinita o un guasto hardware, il sistema operativo normalmente uccide il programma e restituisce un indicatore del motivo.
- Per una richiesta di dispositivo I/O o una chiamata di assistenza al sistema operativo, il sistema operativo salva lo stato del programma, esegue l'attività desiderata e, ad un certo punto in futuro, ripristina il programma per continuare l'esecuzione.



## 4.29 Parallelismo tramite istruzioni

Il Pipelining sfrutta il potenziale parallelismo tra le istruzioni. Questo parallelismo è chiamato parallelismo a livello di istruzione (ILP). Esistono due metodi principali per aumentare la potenziale quantità di parallelismo a livello di istruzione:

1. Aumenta la profondità della pipeline per sovrapporre più istruzioni. Le prestazioni sono potenzialmente maggiori poiché il ciclo di clock può essere più breve.
2. Replica i componenti interni del computer in modo che possa avviare più istruzioni in ogni fase della pipeline. Il nome generale di questa tecnica è **multiple issue**. Consente alla velocità di esecuzione delle istruzioni di superare la frequenza di clock o all'IPC di essere inferiore a 1. Ad esempio, 4GHz 4-way multiple-issue: 16 BIPS, picco CPI = 0,25, picco IPC = 4. Ma le dipendenze lo riducono nella pratica

### 4.29.1 Multiple issue

Ci sono due modi principali per implementare un processore a più issue, con la differenza principale che è la divisione del lavoro tra il compilatore e l'hardware:

- **Static multiple issue:** Un approccio all'implementazione di un processore a più problemi in cui vengono prese molte decisioni dal compilatore prima dell'esecuzione.
- **Dynamic multiple issue:** Un approccio all'implementazione di un processore a più problemi in cui vengono prese molte decisioni durante l'esecuzione da parte del processore.

Due responsabilità primarie e distinte devono essere affrontate in una pipeline multi-problema:

1. Istruzioni per l'imballaggio negli slot di rilascio. Come fa il processore a determinare quante istruzioni e quali istruzioni possono essere emesse in un dato ciclo di clock?
2. Affrontare i dati e controllare i rischi

## 4.30 Il concetto di speculazione

La speculazione è un approccio che consente al compilatore o al processore di "indovinare" cosa fare con un'istruzione, per avviare l'esecuzione di altre istruzioni che possono dipendere dall'istruzione ipotizzata. Esempi

- Speculare sull'esito di un branch. Torna indietro se il percorso intrapreso è diverso
- Specula sul load. Torna indietro se la posizione viene aggiornata



La difficoltà con la speculazione è che potrebbe essere sbagliata. Qualsiasi meccanismo di speculazione deve includere sia un metodo per verificare se l'ipotesi era corretta sia un metodo per srotolare o annullare gli effetti delle istruzioni eseguite in modo speculativo. La speculazione può essere eseguita nel compilatore o dall'hardware.

#### 4.30.1 Speculazione Compiler / Hardware

Il compilatore può riordinare le istruzioni (e.g. trasloco e istruzioni attraverso un branch o un load attraverso uno store). Può includere istruzioni di "riparazione" per recuperare da un'ipotesi errata.

L'hardware può guardare avanti per le istruzioni da eseguire. Buffera i risultati fino a quando non determina che sono effettivamente necessari. Svuota i buffer in caso di speculazione errata ed eseguire nuovamente la sequenza di istruzioni corretta.

### 4.31 Static Multiple Issue

I processori static multiple issue utilizzano il compilatore per fornire assistenza con le istruzioni di packaging e la gestione dei rischi. In un processore di emissione statico, si può pensare all'insieme di istruzioni emesse in un dato ciclo di clock, il pacchetto di emissione, come a una grande istruzione con più operazioni. Poiché un processore statico a più emissioni di solito limita la combinazione di istruzioni che può essere avviata in un dato ciclo di clock, è utile pensare al pacchetto di emissione come a una singola istruzione che consente diverse operazioni in determinati campi predefiniti.

Questa visione ha portato al nome originale di questo approccio: **Very Long Instruction Word** (VLIW).

Il compilatore deve rimuovere alcuni/tutti i pericoli

- Riordinare le istruzioni in pacchetti di emissione
- Nessuna dipendenza all'interno di un pacchetto
- Possibili alcune dipendenze tra i pacchetti (Varia tra gli ISA; il compilatore deve sapere!)
- Pad con nop se necessario

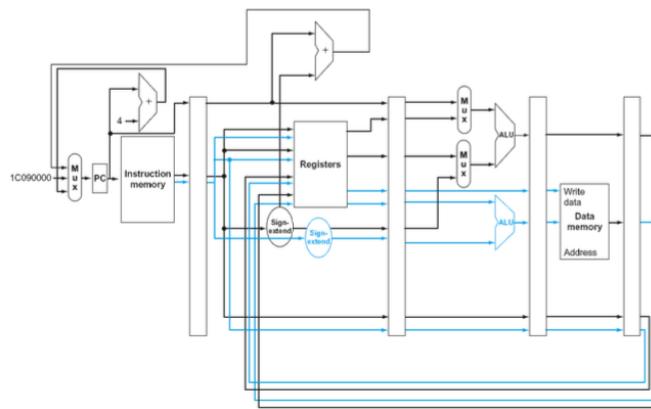
#### 4.31.1 LEGv8 con Static Dual Issue

Pacchetti a due numeri

- Un'istruzione ALU/branch
- Un'istruzione di caricamento/memorizzazione
- Allineamento a 64 bit. ALU/branch, quindi load/store. Completare un'istruzione non utilizzata con nop



Indirizzo	Tipo di istruzione	Stage della Pipeline						
n	ALU / Branch	IF	ID	EX	MEM	WB		
n + 4	Load / Store	IF	ID	EX	MEM	WB		
n + 8	ALU / Branch		IF	ID	EX	MEM	WB	
n + 12	Load / Store		IF	ID	EX	MEM	WB	
n + 16	ALU / Branch			IF	ID	EX	MEM	WB
n + 20	Load / Store			IF	ID	EX	MEM	WB



Chiaramente, questo processore a due problemi può migliorare le prestazioni fino a un fattore due!

Per farlo, tuttavia, è necessario sovrapporre il doppio delle istruzioni nell'esecuzione, e questa sovrapposizione aggiuntiva aumenta la perdita di prestazioni relativa dai dati e dai rischi di controllo.

Per esempio: Ora non possiamo usare il risultato ALU nel load/store nello stesso pacchetto

```
ADD X0, X0, X1
LDUR X2, [X0,#0]
```

Dobbiamo dividere in due pacchetti (di fatto uno stallo).

Nella nostra pipeline a cinque stadi, i load hanno una latenza di utilizzo di un ciclo di clock, che impedisce a un'istruzione di utilizzare il risultato senza andare in stallo.

Nella pipeline a due numeri e cinque stadi, il risultato di un'istruzione di load non può essere utilizzato nel ciclo di clock successivo. Ciò significa che le due istruzioni successive non possono utilizzare il risultato del caricamento senza andare in stallo.

Per sfruttare efficacemente il parallelismo, sono necessari una pianificazione più aggressiva e un compilatore più ambizioso.

#### 4.31.2 Esempio di pianificazione

Come verrebbe pianificato questo ciclo su una pipeline statica a due problemi per LEGv8?



```

Loop: LDUR X0, [X20, #0] // X0 = array di elementi
      ADD X0, X0, X21 // Add scalare in X21
      STUR X0, [X20, #0] // Store del risultato
      SUBI X20, X20, #8 // Decremento il puntatore
      CMP X20, X22 // Compare to loop limit
      B.GT Loop // Branch se X20 > X22
  
```

	Istruzioni ALU o Branch	Istruzioni di trasferimento dati	Clock Cycle
Loop:		LDUR X0, [X20, #0]	1
	SUBI X20, X20, #8		2
	ADD X0, X0, X21		3
	CMP X20, X22		4
	B.GT Loop	STUR X0, [X20, #8]	5

$$IPC = \frac{6}{5} = 1.2 \text{ (c.f. peak } IPC = 2\text{)}$$

#### 4.31.3 Svolgimento del ciclo per pipeline a più emissioni

Un'importante tecnica del compilatore per ottenere maggiori prestazioni dai cicli è lo srotolamento del ciclo, in cui vengono create più copie del corpo del ciclo. Dopo lo srotolamento, sono disponibili più ILP sovrapponendo le istruzioni di diverse iterazioni. Durante il processo di srotolamento, il compilatore introduce generalmente registri aggiuntivi. L'obiettivo di questo processo, chiamato ridenominazione dei registri, è eliminare le dipendenze che non sono dipendenze da dati veri, le cosiddette antidipendenze o dipendenze di nome.

L'antidipendenza, detta anche dipendenza dal nome, è un ordinamento imposto dal riutilizzo di un nome, tipicamente a register, piuttosto che da una vera dipendenza che porta un valore tra due istruzioni.

Svolgimento del ciclo con fattore 4 del ciclo precedente:

	Istruzioni ALU o Branch	Istruzioni di trasferimento dati	Clock Cycle
Loop:	SUBI X20, X20, #32	LDUR X0, [X20, #0]	1
		LDUR X1, [X20, #24]	2
	ADD X0, X0, X21	LDUR X2, [X20, #16]	3
	ADD X1, X1, X21	LDUR X3, [X20, #8]	4
	ADD X2, X2, X21	STUR X0, [X20, #32]	5
	ADD X3, X3, X21	STUR X1, [X20, #24]	6
	CMP X20, X22	STUR X2, [X20, #16]	7
	B.GT Loop	STUR X3, [X20, #8]	8

$$IPC = \frac{15}{8} = 1.875 \text{ (vicino al 2, ma costano i registri e la dimensione del file)}$$



## 4.32 Dynamic Multiple Issue

I processori dynamic multiple issue sono noti anche come processori superscalari. Nei processori superscalari più semplici, le istruzioni vengono emesse in ordine e il processore decide se possono essere emesse in un dato ciclo di clock zero, una o più istruzioni. Il raggiungimento di buone prestazioni su un tale processore richiede comunque che il compilatore tenti di pianificare istruzioni per separare le dipendenze e quindi migliorare il tasso di emissione delle istruzioni. Grande differenza con VLIW: ora il codice, programmato o meno, è garantito dal hardware per la corretta esecuzione. Il codice compilato verrà sempre eseguito correttamente indipendentemente dal tasso di emissione o dalla struttura della pipeline nel processore.

### 4.32.1 Dynamic Pipeline Scheduling

Molti processori superscalari estendono il framework a più problemi per includere la pianificazione dinamica della pipeline. L'hardware consente alla CPU di eseguire le istruzioni fuori ordine per evitare stalli, ma salva il risultato nei registri nell'ordine.

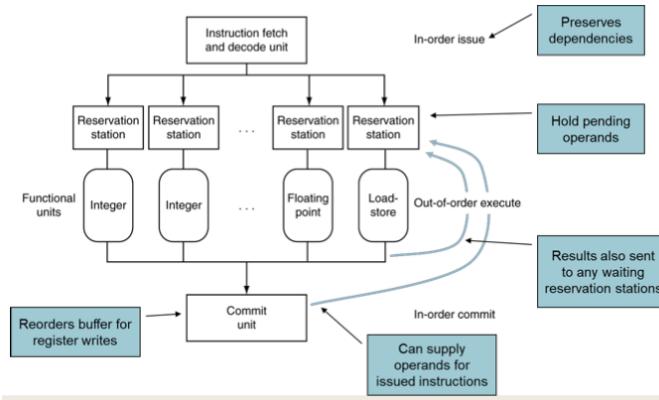
Esempio:

```
LDUR X0, [X21, #20]
ADD X1, X0, X2
SUB X23, X23, X3
ANDI X5, X23, #20
```

Anche se l'istruzione SUB è pronta per l'esecuzione, deve attendere prima il completamento di LDUR e ADD, operazione che potrebbe richiedere molti cicli di clock se la memoria è lenta. La pianificazione dinamica delle pipeline consente di evitare tali rischi. La pianificazione dinamica della pipeline sceglie quali istruzioni eseguire successivamente, eventualmente riordinandole per evitare stalli. In tali processori, la pipeline è suddivisa in tre unità principali:

- un'unità di raccolta ed emissione delle istruzioni
- più unità funzionali (una dozzina o più in design di fascia alta nel 2015)
- un'unità di commit

La prima unità recupera le istruzioni, le decodifica e invia ciascuna istruzione a un'unità funzionale corrispondente per l'esecuzione. Ciascuna unità funzionale dispone di buffer, detti stazioni di prenotazione, che contengono gli operandi e l'operazione. Non appena il buffer contiene tutti gli operandi l'unità funzionale è pronta per l'esecuzione, il risultato è calcolato. Quando il risultato è completato, viene inviato a tutte le stazioni di prenotazione in attesa e all'unità di commit, che memorizza il risultato nel buffer finché non è sicuro inserirlo nel file di registro o in memoria. Il suo buffer è spesso chiamato buffer di riordino e viene utilizzato anche per l'inoltro di operandi.



### 4.33 Rinominazione dei registri

Le stazioni di prenotazione e il buffer di riordino forniscono efficacemente la ridenominazione dei registri. Su emissione di istruzioni alla stazione di prenotazione

- Se l'operando è disponibile nel file di registro o nel buffer di riordino
  - Copiato nella stazione di prenotazione
  - Non più richiesto nel registro; può essere sovrascritto
- Se l'operando non è ancora disponibile
  - Sarà fornito alla stazione di prenotazione da un'unità funzionale
  - L'aggiornamento del registro potrebbe non essere necessario

### 4.34 Esecuzione Out-of-order

Concettualmente, puoi pensare a una pipeline pianificata dinamicamente come all'analisi della struttura del flusso di dati di un programma. Il processore esegue quindi le istruzioni in un ordine che preserva l'ordine del flusso di dati del programma. Questo stile di esecuzione è chiamato esecuzione out of order, poiché le istruzioni possono essere eseguite in un ordine diverso da quello in cui sono state recuperate.

Per fare in modo che i programmi si comportino come se fossero in esecuzione su una semplice pipeline in ordine: L'unità di recupero e decodifica delle istruzioni è necessaria per emettere istruzioni in ordine che consente di tracciare le dipendenze e l'unità di commit è richiesta per scrivere i risultati nei registri e nella memoria nell'ordine di recupero del programma.

Questa modalità conservativa è chiamata commit in-order.



## 4.35 Speculazione

La pianificazione dinamica viene spesso estesa includendo la speculazione basata sull'hardware, in particolare per i branch. Poiché le istruzioni sono state salvate in ordine, sappiamo se il branch è stato predetto correttamente prima che la previsione del percorso di quasi istruzione venivano inviate.

Una pipeline speculativa e programmata dinamicamente può anche supportare la speculazione su indirizzi di load, consentire il riordino del load-store e utilizzare l'unità di commit per evitare speculazioni errate.

### 4.35.1 Perché fare una pianificazione dinamica?

Perché non lasciare che il compilatore programmi il codice?

- Non tutti gli stalli sono prevedibili (ad esempio i cache misses)
- Non posso sempre programmare intorno ai branch. L'esito del branch è determinato dinamicamente
- Diverse implementazioni di un ISA hanno latenze e pericoli diversi

## 4.36 Funzionano i multiple issue?

Sì, ma non tanto quanto vorremmo.

Sia il pipelining che l'esecuzione di più problemi aumentano il throughput delle istruzioni di picco e tentano di sfruttare il parallelismo a livello di istruzione (ILP).

Le dipendenze da dati e controllo nei programmi, tuttavia, offrono un limite massimo alle prestazioni sostenute perché il processore a volte deve aspettare che venga risolta una dipendenza.

Gli approcci software-centrati allo sfruttamento di ILP si basano sulla capacità del compilatore di trovare e ridurre gli effetti di tali dipendenze, mentre gli approcci hardware-centrati si basano su estensioni alla pipeline e meccanismi di emissione.

La speculazione, eseguita dal compilatore o dall'hardware, può aumentare la quantità di ILP che può essere sfruttata tramite previsione, anche se è necessario prestare attenzione poiché speculare in modo errato rischia di ridurre le prestazioni.

## 4.37 Efficienza energetica e pipeline avanzate

Lo svantaggio del crescente sfruttamento del parallelismo a livello di istruzione attraverso problemi multipli dinamici e speculazioni è la potenziale inefficienza energetica. L'attuale convinzione è che mentre i processori più semplici non sono veloci come i loro sofisticati fratelli, offrono prestazioni migliori per Joule, in modo da poter offrire più prestazioni per chip quando i progetti sono vincolati più dall'energia che dal numero di transistor.

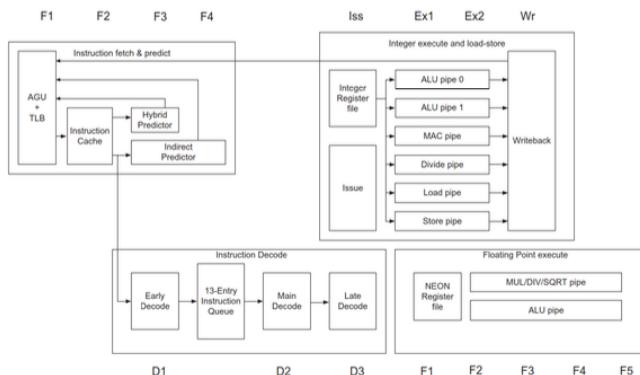


Microprocessore	Anno	Clock Rate	Stage della pipeline	Issue width	Out of order / Speculation	Cores / Chip	Power
Intel 486	1989	25 MHz	5	1	No	1	5 W
Intel Pentium	1993	66 MHz	5	2	No	1	10 W
Intel Pentium Pro	1997	200 MHz	10	3	Yes	1	29 W
Intel Pentium 4 Willamette	2001	2000 MHz	22	3	Yes	1	75 W
Intel Pentium 4 Prescott	2004	3600 MHz	31	3	Yes	1	103 W
Intel Core	2006	2930 MHz	14	4	Yes	2	75 W
Intel Core i5 Nehalem	2010	3300 MHz	14	4	Yes	2 - 4	87 W
Intel Core i5 Ivy Bridge	2012	3400 MHz	14	4	Yes	8	77 W

## 4.38 ARM Cortex-A53 vs Intel Core i7 920

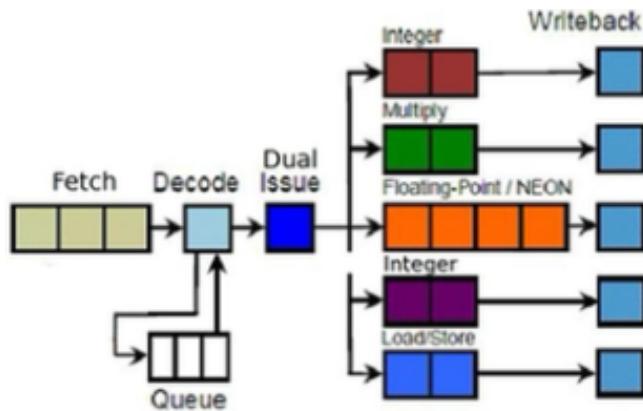
Processore	ARM Cortes-A53	Intel Core i7 920
Market	Personal Mobile Device	Server, Cloud
Thermal design power	100 milliWatts (1 core @ 1GHz)	130 Watts
Clock Rate	1,5 GHz	2,66 GHz
Cores / Chip	4 (configurable)	4
Floating point?	Yes	Yes
Multiple issue?	Dynamic	Dynamic
Peak istruction / clock cycle	2	4
Pipeline stages	8	14
Pipeline schedule	Static in-order	Dynamic Out-of-order with speculation
Branch prediction	Hybrid	2 - level
1st level caches/core	16 - 64 KiB I, 16 - 64 KiB D	32 KiB I, 32 KiB D
2nd level cache/core	128 - 2048 KiB (shared)	256 KiB (per core)
3rd level cache (shared)	(platform dependent)	2 - 8 MiB

### 4.38.1 ARM Cortex-A53 pipeline



### feature chiave del microprocessore Cortex-A53

Il Cortex-A53 core ha una dual-issue in-order front end con 5 pipeline che costituiscono il back end

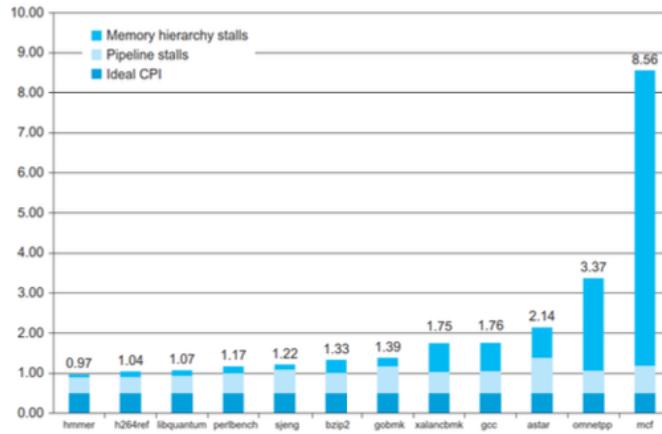


La pipeline per l'elaborazione di numeri interi ha 8 fasi di pipeline, l'elaborazione NEON e FP ha due fasi aggiuntive.

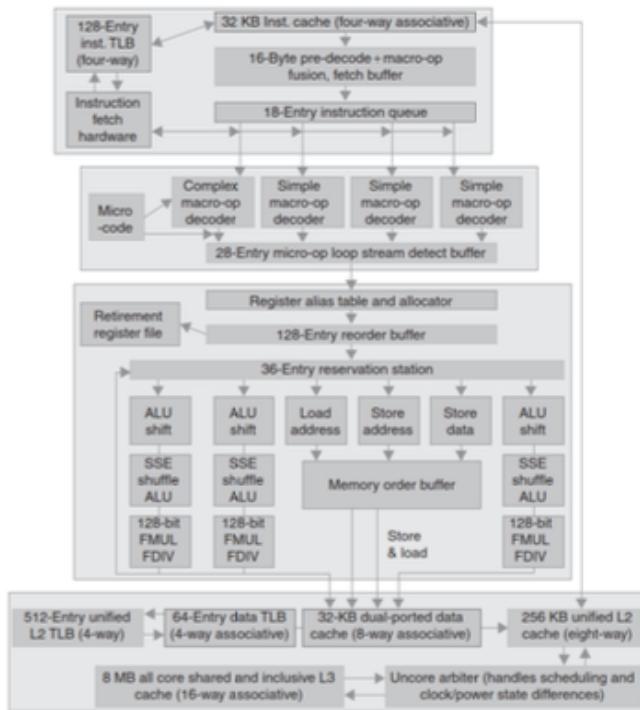
Le prime tre fasi recuperano due istruzioni alla volta e cercano di mantenere piena una coda di istruzioni di 13 ingressi. Utilizza un predittore di branch condizionali ibridi a 6k bit, un predittore di rami indiretti a 256 ingressi e uno stack di indirizzi di ritorno a 8 voci per prevedere i rendimenti delle funzioni future. La previsione dei branch indiretti richiede un'ulteriore fase di pipeline. Quando la previsione del branch è sbagliata, svuota la pipeline, con conseguente penalità di errata previsione di 8 cicli di clock. Le fasi di decodifica della pipeline determinano se ci sono dipendenze tra una coppia di istruzioni, che costringerebbero l'esecuzione sequenziale e in quale pipeline delle fasi di esecuzione inviare le istruzioni. La sezione di esecuzione delle istruzioni occupa principalmente tre fasi della pipeline e fornisce una pipeline per le istruzioni di load, una pipeline per le istruzioni di store, due pipeline per le operazioni aritmetiche intere e pipeline separate per le operazioni di moltiplicazione e divisione intera. Entrambe le istruzioni della coppia possono essere emesse alle tubazioni di load o di store. Le fasi di esecuzione hanno un forwarding completo tra le pipeline. Le operazioni floating point e SIMD aggiungono altre due fasi della pipeline alla sezione di esecuzione delle istruzioni e presentano una pipeline per operazioni di moltiplicazione/divisione/radice quadrata e una pipeline per altre operazioni aritmetiche.



#### 4.38.2 ARM Cortex-A53 performance



#### 4.38.3 Intel Core i7 920 pipeline



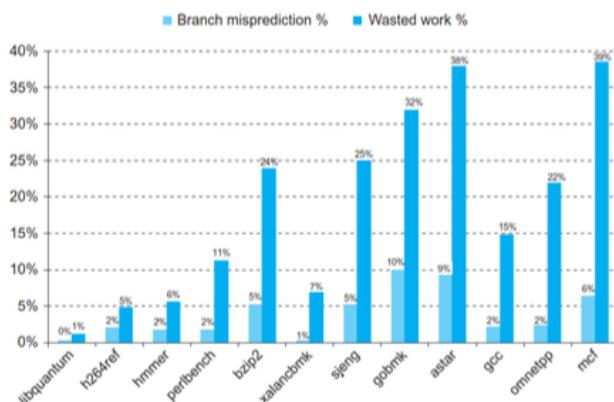
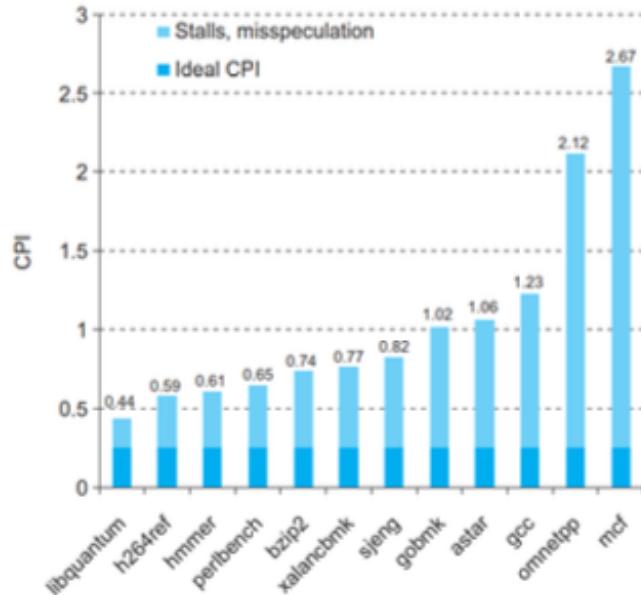
1. Recupero delle istruzioni: il processore utilizza un buffer target di branch multilevello per raggiungere un equilibrio tra velocità e precisione di previsione. C'è anche uno stack di indirizzi di ritorno per accelerare il ritorno della funzione. Le previsioni errate causano una penalità di circa 15 cicli. Utilizzando l'indirizzo previsto, l'unità di recupero delle istruzioni recupera 16 byte dalla cache delle istruzioni.



2. I 16 byte sono inseriti nel buffer di istruzioni di predecodifica: lo stadio di predecodifica trasforma i 16 byte in singole istruzioni x86. Questa predecodifica non è banale poiché la lunghezza di un'istruzione x86 può variare da 1 a 15 byte e il predecodificatore deve guardare attraverso un certo numero di byte prima di conoscere la lunghezza dell'istruzione. Le singole istruzioni x86 vengono inserite nella coda di istruzioni da 18 ingressi.
3. Decodifica micro-op: le istruzioni x86 individuali vengono tradotte in micro-operazioni (micro-op). Tre dei decodificatori gestiscono istruzioni x86 che si traducono direttamente in un'unica micro-op. Per le istruzioni x86 che hanno una semantica più complessa, c'è un motore di microcodice che viene utilizzato per produrre la sequenza micro-op; può produrre fino a quattro micro-op ogni ciclo e continua fino a quando non viene generata la sequenza micro-op necessaria. Le micro-op sono posizionate secondo l'ordine delle istruzioni x86 nel buffer micro-op da 28 ingressi.
4. Il buffer micro-op esegue il rilevamento del flusso di loop: se c'è una piccola sequenza di istruzioni (meno di 28 istruzioni o 256 byte di lunghezza) che comprende un loop, il rilevatore di flussi di loop troverà il loop ed emetterà direttamente le micro-op dal buffer, eliminando la necessità di attivare le fasi di recupero delle istruzioni e decodifica delle istruzioni.
5. Eseguire il problema di istruzioni di base: ricercare la posizione del registro nelle tabelle del registro, rinominare i registri, assegnare una voce del buffer di riordino e recuperare eventuali risultati dai registri o dal buffer di riordino prima di inviare le micro-op alle stazioni di prenotazione.
6. L'i7 utilizza una stazione di prenotazione centralizzata di 36 elementi condivisa da sei unità funzionali. Fino a sei micro-op possono essere inviate alle unità funzionali ogni ciclo di clock.
7. Le singole unità funzionali eseguono micro-op e quindi i risultati vengono rispediti a qualsiasi stazione di prenotazione in attesa e all'unità di pensionamento del registro, dove aggiorneranno lo stato del registro, una volta che si sa che l'istruzione non è più speculativa. La voce corrispondente all'istruzione nel buffer di riordino è contrassegnata come completa.
8. Quando una o più istruzioni in testa al buffer di riordino sono state contrassegnate come complete, vengono eseguite le scritture in sospeso nell'unità di pensionamento del registro e le istruzioni vengono rimosse dal buffer di riordino.



#### 4.38.4 Intel Core i7 920 performance





## 5 Memorie e Registri

Il processore opera sui dati attraverso istruzioni:

- Le istruzioni operano sui registri
- Quando necessario, caricare (leggere) i dati dalla memoria ai registri
- Quando necessario, memorizzare (scrivere) i dati dai registri alla memoria

I load e gli store richiedono tempo! Nel complesso, quanto tempo? Dipende dal numero di operazioni e dipende dalla durata della singola operazione

### 5.0.1 Load/Store

#### Quanti? (Numero)

C:

```
float fahreneit2celsius(float f){  
    return ((5.0 / 9.0) * (f - 32.0));  
}
```

LEGv8:

```
LDURS S16, [X27, const5]      // S16 = 5.0  
LDURS S18, [X27, const9]      // S18 = 9.0  
FDIVS S16, S16, S18          // S16 = 5.0 / 9.0  
LDURS S18, [X27, const32]    // S18 = 32.0  
FSUBS S18, S12, S18          // S18 = f - 32.0  
FMULS S0, S16, S18          // S0 = (5.0 / 9.0) * (fahr - 32.0)  
BR LR                         // return
```

3 loads (LDURS), possono diventare 2 con un'ottimizzazione del compilatore

#### Caso più complesso

```
// Matrice 32 x 32  
void matrixMul (double c[][] , double a[][] , double b[][]){  
    int i, j, k;  
    for (i=0; i<32; i++){  
        for (j=0; j<32; j++){  
            for (k=0; k<32; k++){  
                c[i][j] = c[i][j] + a[i][k] * b[k][j];  
            }  
        }  
    }  
}
```

Molti più load e store!

**Quanto tempo? (durata)** Dipende dalla tecnologia di memoria:

Tipo	Tempo di accesso [ns]	Costo [\$/GB]
SRAM	0,5 - 2,5	500 - 1000
DRAM	50 - 70	10 - 20
Flash	5.000 - 50.000	0,75 - 1
Dischi magnetici	5.000.000 - 20.000.000	0,05 - 0,1

Richiamo: 1 ciclo della CPU richiede  $\approx 1$  ns

**Obiettivo** : elaborare più dati, più velocemente, più economico

Idealmente, vogliamo ridurre al minimo il tempo complessivo trascorso ad accedere alla memoria. Possiamo lavorare su:

1. Numero di accessi (dipende principalmente dal codice)
2. Durata di ogni accesso

Come?

1. Scrivi codice migliore (ottimizzato)
2. Usa una memoria più veloce?

→ tempo di accesso/costo trade-off!

### 5.0.2 Idea: due memorie

Supponiamo di avere:

- Memoria ravvicinata  $M_1$ : accesso piccolo e veloce
- Grande memoria  $M_2$ : grande accesso lento

Ogni volta che cerchi l'elemento dati  $x$ :

- Se c'è in  $M_1$ , prendilo da la
- altrimenti prendilo in  $M_2$  e mettilo in  $M_1$  (libera dello spazio se necessario)

Utile se accedi presto di nuovo  $x$

### 5.0.3 Idea migliore

Utile se accedi presto di nuovo  $x$  o qualcosa di vicino a  $x$ :

- Se c'è in  $M_1$ , prendilo da lì
- altrimenti prendi ( $\dots, x-2, x-1, x, x+1, x+2, \dots$ ) da  $M_2$  e mettilo in  $M_1$  (libera dello spazio se necessario)

Utile se accedi presto di nuovo  $x$  o qualcosa di vicino a  $x$ !

### 5.0.4 Principi di località

Utile se accedi presto di nuovo  $x$  o qualcosa di vicino a  $x$ !

**Località temporale:** se si accede a una posizione dati  $x$  a  $t$ , probabilmente vi si accederà di nuovo presto (ad alcuni  $t + \delta t$ )

**Località spaziale:** se si accede a una posizione dati  $x$  a  $t$ , è probabile che le posizioni dei dati vicine a  $x$  (alcune  $x + \delta x$ ) saranno nuovamente accedute presto

- a, • b, • n (=3), • aSum, • bSum, • i

```
float findAvgDiff (float[] a,
float[] b, int n)
{
    float aSum = 0;
    float bSum = 0;
    int i;
    for (i = 0; i<n; i++)
        aSum = aSum + a[i];
    for (i = 0; i<n; i++)
        bSum = bSum + b[i];
    return (aSum - bSum) / n;
}
```

Accesses (simplified):



### 5.0.5 Più di un livello: gerarchie della memoria

Invece di solo due memorie  $M_1$  (veloce e piccola) e  $M_2$  (grande e lenta), una gerarchia di memorie:

- CPU
- $M_1$ : vicina alla CPU, la più veloce, la più piccola



- $M_2$ : un pò più distante, un pò più lenta, un pò più grande
- $M_3$ : un pò più distante, un pò più lenta, un pò più grande
- ...
- $M_k$ : lontana, lenta, grande

Pro	Contro
<ul style="list-style-type: none"><li>- Dimensione di <math>M_k</math> (la più grande)</li><li>- (Quasi) tempo di accesso di <math>M_1</math> (il più vicino)</li><li>- Il costo complessivo è molto inferiore a una M con le dimensioni di <math>M_k</math> e il tempo di accesso di <math>M_1</math> (idealità)</li></ul>	<ul style="list-style-type: none"><li>- Necessità di implementare l'algoritmo di accesso all'interno della CPU/sistema</li></ul>

### 5.0.6 Cache

La gerarchia della memoria è un modello, uno schema di soluzione per un problema comune. Altri casi:

- Copia locale dei dati di rete
- Memorizzazione di complessi risultati di calcolo
- Memoria fisica

Nome comune per la memoria più vicina: **cache**

## 5.1 Tecnologia delle Memorie

### 5.1.1 Static Random Access Memory (SRAM)

**Proprietà** : Tempo di accesso fissato (lettura/scrittura): I tempi di lettura e scrittura possono essere diversi

**Tecnologia** :

- 6-8 transistor per ogni bit (per ridurre al minimo i disturbi durante le letture): grande footprint sul silicio
- Non c'è bisogno di ricaricare: il tempo di accesso è molto vicino al tempo di ciclo
- Basso consumo energetico in standby
- "Attualmente" le SRAM non sono più chip separati, ma sono integrate nei processori

### 5.1.2 Dynamic Random Access Memory (DRAM)

**Proprietà** : Accesso più rapido a lotti di dati (le file sono bufferizzate)



### Tecnologia :

- 1 transistor + 1 condensatore per bit: è necessario aggiornare (leggere e scrivere) un ingombro inferiore rispetto alla SRAM.
- Nessun accesso mentre si ricarica: I bit sono organizzati in righe, le righe vengono ricaricate contemporaneamente, le righe sono bufferizzate

### DRAM "Moderne"

- DRAM sincrona (SDRAM): un clock facilita il trasferimento a raffica
- SDRAM a doppia velocità di dati (DDR): trasferimento sia sul bordo in salita che in discesa. DDR4-3200 significa clock a 1600 MHz. Organizzato in banche: letture/scritture fatte contemporaneamente su diverse banche
- Spesso imballato fisicamente in due moduli di memoria in linea (DIMM). Un DIMM di solito contiene 4-16 chip DRAM. Un tipico DIMM può trasferire 25600 MB/s (PC25600)

### DRAM over time

Anno	Dimensione Chip [bit]	Costo \$/GB	Tempo di accesso nuovo [ns]	Tempo di accesso esistente [ns]
1980	64 K	1.500.000	250	150
1983	256 K	500.000	185	100
1985	1 M	200.000	135	40
1989	4 M	50.000	110	40
1992	16 M	15.000	90	30
1996	64 M	10.000	60	12
1998	128 M	4.000	60	10
2000	256 M	1.000	55	7
2004	512 M	250	50	5
2007	1 G	50	45	1.25
2010	2G	30	40	1
2012	4G	10	35	0.8

### 5.1.3 Memorie flash

**Tecnologia** : Basato sulla memoria di sola lettura programmabile cancellabile elettricamente (EEPROM)

Le scritture possono indossare hardware dietro i bit. Un controller si occupa di distribuire l'utilizzo in modo uniforme

Le unità a stato solido (SSD) utilizzano la memoria flash

### 5.1.4 Disk memory

**Tecnologia** : Con parti in movimento (differentemente di SRAM, DRAM, flash)

Le unità disco rigido (HDD) sono dispositivi di archiviazione basati sulla memoria del disco



### Terminologia :

Parti meccaniche:

- Piatti (platters)
- Teste (heads)

Parti logiche:

- Traccia (track) (1000 di settori): cerchio concentrico sul piatto
- Settore (sector) (512-4096 byte): porzione della traccia
- Cilindro (cylinder): set di settori allineati verticalmente

Fasi di Read/Write:

- Ricerca (seek): la testa si muove verso la la traccia ( $\approx 3 - 13 \text{ ms}$ )
- Latenza rotazionale o ritardo ( $\approx 6 \text{ ms} @ 5400 \text{ giri/min}$ )
- Dati effettivi in lettura/scrittura

## 5.2 Caches

### 5.2.1 Cos'è una cache

**Cache:** un nascondiglio dove mettere qualcosa per un uso futuro.

Il primo utilizzo del termine nei computer indicava specificamente la piccola memoria tra il processore e la memoria principale. Al giorno d'oggi, la cache indica qualsiasi spazio di archiviazione che sfrutta la località (es. cache del browser).

### 5.2.2 Usare una cache

Consideriamo solo la lettura per ora

Ogni volta che il processore vuole leggere una locazione  $x$  di una memoria:

- Verifica se  $x$  è nella cache ( $x$  è l'indirizzo del dato)
- Se non viene trovato nella cache viene letta la memoria principale e viene memorizzato sulla cache

Come fare a

1. Sapere se  $x$  è nella cache?
2. Se c'è nella cache, come facciamo a sapere dov'è?

## Premessa Contenuto della memoria vs indirizzi della memoria

000	13
001	45
010	e4
011	14
100	0f
101	76
110	1a
111	ff

Memoria a 8 byte:

- Gli indirizzi vanno da  $000_2 = 0$  a  $111_2 = 7 \rightarrow$  indirizzo 3 bits
- Per ogni indirizzo c'è un byte:  $13_{16} = 00010011_2$
- $[x]$  è il dato contenuto all'indirizzo  $x$ :  $[011] = 14_{16} = 00010100_2$

In generale, ci sono n indirizzi di bit (e.g.  $n = 64$ )

### 5.2.3 Cache mappata direttamente

La cache è formata da  $s_c = 2^{n_c}$  triplete [bit di validità, tag, blocco]

- Il **blocco** contiene  $s_b = 2^{n_b}$  bytes.  
Il blocco  $k$ -esimo conterrà porzioni della memoria principale di  $s_b$  bytes partendo dagli indirizzi  $x$  s.t.  $x \% s_c = k$
- Il **tag** indica quale  $x$  tra i molti è effettivamente il blocco cercato.  
Ci possono essere  $2^{n_m - n_c - n_b}$  valori, con una dimensione della memoria principale pari a  $s_m = 2^{n_m}$
- Il **bit di validità** indica se il blocco è da considerare valido. Inizialmente viene settato a 0

### Esempio

Cache:

00	1	01	00010000
01	1	10	10000010
10	0	11	10000010
11	0	11	10010101

Memoria principale:

$n_m = 4$ ,  $s_m = 16$  bytes = 128 bits

0000	00000001	1000	10000001
0001	00000010	1001	10000010
0010	00000100	1010	10000100
0011	00001000	1011	10001000
0100	00010000	1100	10010000
0101	00100000	1101	10100000
0110	01000000	1110	11000000
0111	10000000	1111	11000001

Dimensione tag:

$$n_m - n_c - n_b = 4 - 2 - 0 = 2 \text{ bits}$$

Dimensione triplete:

$$1 + n_m - n_c - n_b + 8s_b = 1 + 2 + 8 = 11 \text{ bits}$$

Dimensione cache:

$$s_c(1 + n_m - n_c - n_b + 8s_b) = 4 \cdot 11 = 44 \text{ bits}$$



#### 5.2.4 Guardare nella cache

Assumiamo  $s_b = 1$ . Data una cache con  $s_c = 2^{n_c}$  blocchi, cercare  $x$  significa guardare il  $k - esimo$  blocco con  $x \% s_c = k$ .

In numeri binari  $x \% s_c = x \% 2^{n_c}$  sono "gli  $n_c$  bits di  $x$  meno significativi".

#### 5.2.5 Esercizio - Cache size [#39]

1. Per una main memory di 4 GB e una cache con 1024 blocchi di 4 word ciascuno
  - Calcolare il tag size in bits
  - Calcolare il rapporto tra "actual data stored in the cache" e "overall cache size"
2. Ripetere i calcoli fatti per una main memory di 4 GB e una cache con 1024 blocchi di 16 word ciascuno

1 word sono 4 bytes

**Soluzione:**

1. Dati:  $n_m = 32$ ,  $n_c = 10$ ,  $n_b = 4$ 
  - Tag Size:  $n_m - n_b - n_c = 32 - 10 - 4 = 18$
  - Rapporto:  $\frac{16 \cdot 8}{(16 \cdot 8 + 18)} = 87,67\%$
2. Dati:  $n_m = 32$ ,  $n_c = 10$ ,  $n_b = 6$ 
  - Tag Size:  $n_m - n_b - n_c = 32 - 10 - 6 = 16$
  - Rapporto:  $\frac{16 \cdot 8}{(16 \cdot 8 + 16)} = 88,89\%$

#### 5.2.6 Algoritmo per leggere $x$

Data  $x$  di  $n_m$  bits (assumiamo  $s_b = 1$ )

1. Prendi  $y = x_{[n_m - n_c, n_m]}$  come  $n_c$  bits meno significativi di  $x$
2. Leggi la tripletta  $t = [y]$  dalla cache all'indirizzo  $y$ 
  - $t_{val} = t_{[0, 1]}$  è il primo bit di  $t$
  - $t_{tag} = t_{[1, 1 + (n_m - n_c)]}$  sono i bits di  $t$  dal secondo al  $2 + (n_m - n_c) - esimo$
  - $t_{block} = t_{[1 + (n_m - n_c), 1 + (n_m - n_c) + 8]}$  è il blocco (di un byte)
3. Se  $t_{val} \neq 1$ , vai al punto 6
4. Se  $t_{tag} \neq$  al  $n_m - n_c$  bit più significativi di  $x$ , vai al punto 6
5. Ritorna  $t_{block}$  (**hit**)
6. Leggi  $x$  dalla memoria principale (**miss**)



### 5.2.7 Miss rate e Miss penalty

Data una sequenza di  $n$  indirizzi  $x_1, x_2, \dots, x_n$

- Il **miss rate** è il rapporto tra gli accessi risultati un miss e il totale di  $n$ . L'**hit rate** è definito come  $1 - \text{miss rate}$
- Il **miss penalty** è il tempo impiegato per leggere un blocco dalla memoria principale e metterlo nella cache. Durante questo tempo il processore non fa niente.

### 5.2.8 Approfondiamo la località spaziale

*Località spaziale: se si accede a una posizione dati  $x$  a  $t$ , è probabile che le posizioni dei dati vicine a  $x$  (alcune  $x + \delta x$ ) saranno nuovamente accedute presto.*

Abbiamo bisogno non solo di  $x$  ma anche di  $x + \delta x$  nella cache → bisogna usare blocchi più grandi! (cioè,  $s_b > 1$ ). **N.B.:** Anche il miss penalty aumenta: in caso di errore, maggiore trasferimento di dati dalla memoria principale

### 5.2.9 Mappatura diretta con $s_b > 1$

Assumiamo  $n_m = 6$ ,  $n_c = 2$ ,  $n_b = 2 \rightarrow s_b = 4$

- Il blocco a  $y = 00$  contiene:
  - bytes di  $x = 00\underline{0000}$  a  $00\underline{0011}$
  - bytes di  $x = 01\underline{0000}$  a  $01\underline{0011}$
  - bytes di  $x = 10\underline{0000}$  a  $10\underline{0011}$
  - bytes di  $x = 11\underline{0000}$  a  $11\underline{0011}$
- Il blocco a  $y = 01$  contiene:
  - bytes di  $x = 00\underline{0100}$  a  $00\underline{0111}$
  - bytes di  $x = 01\underline{0100}$  a  $01\underline{0111}$
  - bytes di  $x = 10\underline{0100}$  a  $10\underline{0111}$
  - bytes di  $x = 11\underline{0100}$  a  $11\underline{0111}$
- Il blocco a  $y = 10$  contiene:
  - bytes di  $x = 00\underline{1000}$  a  $00\underline{1011}$
  - bytes di  $x = 01\underline{1000}$  a  $01\underline{1011}$
  - bytes di  $x = 10\underline{1000}$  a  $10\underline{1011}$
  - bytes di  $x = 11\underline{1000}$  a  $11\underline{1011}$



- Il blocco a  $y = 11$  contiene:
  - bytes di  $x = 00\underline{1}100$  a  $00\underline{1}111$
  - bytes di  $x = 01\underline{1}100$  a  $01\underline{1}111$
  - bytes di  $x = 10\underline{1}100$  a  $10\underline{1}111$
  - bytes di  $x = 11\underline{1}100$  a  $11\underline{1}111$

### 5.2.10 Algoritmo per leggere $x$ ( $n_b \geq 0$ )

1.  $y = x_{[n_m - n_c - n_b, n_m - n_b[}$
2.  $t = [y]$ 
  - $t_{val} = t_{[0, 1[}$
  - $t_{tag} = t_{[1, 1 + (n_m - n_c - n_b)[}$
  - $t_{block} = t_{[1 + (n_m - n_c - n_b), 1 + (n_m - n_c - n_b) + 8 * 2^{n_b}[}$
3. Se  $t_{val} \neq 1$ , vai al punto 6
4. Se  $t_{tag} \neq x_{[0, n_m - n_c - n_b[}$ , vai al punto 6
5. Ritorna  $t_{block}[8z, 8z+8[$  con  $z = x_{[n_m - n_b, n_m]}$  (**hit**)
6. Leggi  $x_0, \dots, x_{s_b - 1}$  dalla memoria principale (**miss**)
  - $x_0 = x_{[0, n_m - n_b[} \rightarrow 0 \dots 0$  ( $n_b$  0s)
  - $x_k = x_{[0, n_m - n_b[} \rightarrow k_2$  ( $k$  come numero binario con  $n_b$  bits)
  - $x_{s_b - 1} = x_{[0, n_m - n_b[} \rightarrow 1 \dots 1$  ( $n_b$  1s)
7. Impostare a 1 il bit di validità di  $y$
8. Ritorna  $[x]$

### 5.2.11 Esercizio - Misses and Hits con $s_b = 4$ [<#49]

Da una memoria principale di 256 byte, dove  $x = [x]$ , una cache inizialmente vuota con 4 blocchi di 1 word ciascuno e le letture 10, 11, 13, 20, 21, 22, 10, 20, 21 (indirizzi decimali  $x$ )

- Mostrare il contenuto della cache dopo la terza richiesta
- Calcolare l'hit rate



**Suggerimento:** utilizzare la notazione decimale per il contenuto del blocco.

**Soluzione:**  $n_m = 8$ ,  $n_c = 2$ ,  $n_b = 2$

Passaggio 0

Cache	Validity	TAG	Blocco
00	0	0000	0
01	0	0000	0
10	0	0000	0
11	0	0000	0

Passaggio 1-2

Cache	Validity	TAG	Blocco
00	0	0000	0
01	0	0000	0
10	1	0000	8-9-10-11
11	0	0000	0

Passaggio 3

Cache	Validity	TAG	Blocco
00	0	0000	0
01	0	0000	0
10	1	0000	8-9-10-11
11	1	0000	12-13-14-15

Passaggio 4

Cache	Validity	TAG	Blocco
00	0	0000	0
01	1	0001	20-21-22-23
10	1	0000	8-9-10-11
11	1	0000	12-13-14-15

Binary	Decimal
00000000	0
...	...
00001010	10
00001011	11
...	...
00001101	13
...	...
00010100	20
00010101	21
00010110	22
...	...
11111111	255

10	MISS
11	HIT
13	MISS
20	MISS
21	HIT
22	HIT
10	HIT
20	HIT
21	HIT

**Passaggio 1-2:**

- $t_{val}$  0 → 1
- $t_{tag}$  0000
- Indirizzo di cache 10

**Passaggio 3:**

- $t_{val}$  0 → 1
- $t_{tag}$  0000
- Indirizzo di cache 11

**Passaggio 4:**

- $t_{val}$  0 → 1
- $t_{tag}$  0001
- Indirizzo di cache 01

Nel consegue che l'hit rate è pari al 66,6% e il miss rate è pari al 33,3%



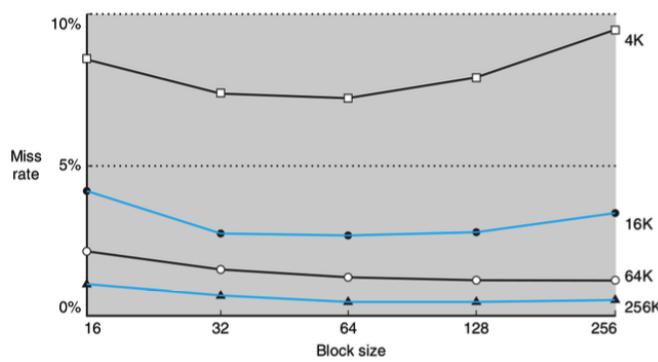
### 5.2.12 Miss rate/penalty e dimensione dei blocchi

Maggiore è la dimensione del blocco

- Minore è il miss rate (per una migliore località spaziale)
- Maggiore sarà il miss penalty
- Minore saranno il numero di blocchi (per la stessa dimensione della cache), quindi maggiore è il miss rate

Dipende dalla sequenza specifica degli accessi alla memoria

#### In pratica



### 5.2.13 Miss penalty

Maggiore è la dimensione del blocco maggiore sarà il miss penalty.

Possibili ottimizzazioni per ridurre il penalty:

- **early restart**: ricomincia quando hai letto  $[x]$ , non quando è stato letto l'intero blocco
- **request word first**: prima leggi  $[x]$ , successivamente early restart ed infine leggi l'intero blocco.

#### Quantifichiamo coi numeri :

Assumiamo:

- $s_b = 4$
- 1 ciclo per istruzione (CPI)
- 1 ciclo per leggere 1 byte dalla cache
- 100 cicli per leggere 4 bytes dalla memoria principale

Cicli per la lettura:

- Hit: 1 ciclo
- Miss:  $4 \cdot 100 + 1 = 401$  cicli
- Miss con early restart: da  $1 \cdot 100 + 1 = 101$  a  $4 \cdot 100 + 1 = 401 \rightarrow$  avg. 251 cicli
- Miss con requested word first:  $1 \cdot 100 + 1 = 101$  cicli

### Impatto sui CPI

	1%	5%	10%	50%
Miss	5	21	41	201
Miss with early restart	4	14	26	126
Miss with requested word first	2	6	11	51

#### 5.2.14 Scrittura

Se una scrittura agisce solo sulla cache, il contenuto della memoria principale ad una data  $x$  e della cache alla corrispondente  $y$  possono differire: sono incoerenti.  
Come evitare le incongruenze?

1. write-through
2. write-back

#### write-through :

A ogni scrittura

- scrivi in  $y$  nella cache
- scrivi in  $x$  nella memoria principale

Pro	Contro
Strategia semplice, entrambi semplici sia concettualmente che nell'implementazione	La cache non da nessun vantaggio nella scrittura

**Impatto sui CPI** Assumiamo:

- $s_b = 1$
- 1 ciclo per istruzione (CPI)
- 1 ciclo per leggere 1 byte dalla cache
- 100 cicli per leggere/scrivere 1 bytes dalla memoria principale

CPI effettivo basato sul tasso di errore (asse y) e sul rapporto lettura-scrittura (asse x):

	20	10	5	1
1%	7	11	18	51
5%	10	15	22	53
10%	15	19	26	55
50%	53	55	59	75

**write-back :**

A ogni scrittura: scrivo in  $y$  nella cache

A ogni miss di  $y$  (lettura e scrittura):

- Prima scrivo il blocco  $y$  nel proprio  $x$
- Infine, carico da  $x$  in  $y$

Pro	Contro
Più complessa, più difficile da implementare più componenti logiche, maggior ingombro, costi maggiori	Minore impatto sull'effettivo CPI a causa del minor numero di accessi alla memoria principale

### 5.2.15 Esercizio - Actual CPI [#59]

Si consideri un processore con un CPI di 2 e una miss penalty di 100 cicli sia in lettura che in scrittura; si consideri un programma con un rapporto lettura-scrittura di 3 a 1 che risulta in un miss rate complessivo del 4%:

1. Qual è il cambiamento nell'attuale CPI dimezzando il miss rate?
2. Qual è il cambiamento nell'attuale CPI dimezzando il miss penalty?

**N.B.:** Tre informazioni mancanti: cicli di accesso alla cache, policy di scrittura, percentuale di istruzioni lettura / scrittura nel programma.

**Soluzione:**

<i>Dati:</i>		Write back    Write through			Diff from $CPI_s$
CPI	2	$CPI_s$	10,08	30	
MISS Penalty	100				
R - W	3 a 1				
Overall miss rate	4%				
1.	Miss rate	2%	$CPI_{e1}$	6,04	4,04    1,5
2.	Miss penalty	50	$CPI_{e2}$	6,08	4    14

Calcolo il  $CPI_s$  (CPI di partenza) per entrambe le policy:

- Write back:

$$2 \cdot (MP + CPI) + CPI \cdot (100\% - OMR)$$

$$\text{nel nostro caso } 2 \cdot (100 + 2) + 2 \cdot (100\% - 4\%) = 10,08$$

- Write trought:

$$\frac{\# Read \cdot [(MP + CPI) \cdot OMR + CPI \cdot (100\% - OMR)] + \# Write \cdot (MP + CPI)}{4}$$

$$\text{nel nostro caso } \frac{3 \cdot [(100+2) \cdot 4\% + 2 \cdot (100\% - 4\%)] + (100+2)}{4} = 30$$

Svolgo gli stessi calcoli cambiando il Miss rate e il Miss penalty in base ai punti 1 e 2:

1. Overall miss rate (OMR) = 2%

- Write back:  $2 \cdot (100 + 2) + 2 \cdot (100\% - 2\%) = 6,04$
- Write trought:  $\frac{3 \cdot [(100+2) \cdot 4\% + 2 \cdot (100\% - 4\%)] + (100+2)}{4} = 28,5$

2. Miss Penalty (MP) = 50

- Write back:  $2 \cdot (50 + 2) + 2 \cdot (100\% - 2\%) = 6,08$
- Write trought:  $\frac{3 \cdot [(50+2) \cdot 4\% + 2 \cdot (100\% - 4\%)] + (50+2)}{4} = 16$

Faccio la differenza tra  $CPI_s$  e i vari CPI ottenuti in base alle policy

1. Con Miss rate al 2%

- $10,08 - 6,04 = 4,04$
- $30 - 28,5 = 1,5$

2. Con Miss penalty a 50

- $10,08 - 6,08 = 4$
- $30 - 16 = 14$



### 5.2.16 Una cache reale: processore FastMATH

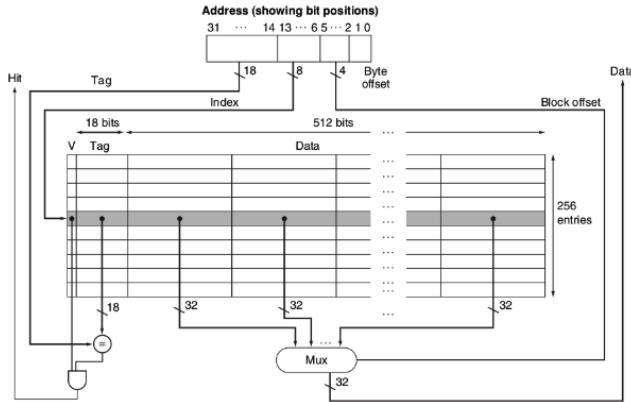


FIGURE 5.12 The 16 KIB caches in the Intrinsicity FastMATH each contain 256 blocks with 16 words per block. Note that the address size for this computer is just 32 bits. The tag field is 18 bits wide and the index field is 8 bits wide, while a 4-bit field (bits 5–2) is used to index the block and select the word from the block using a 16-to-1 multiplexor. In practice, to eliminate the multiplexor, caches use a separate large RAM for the data and a smaller RAM for the tags, with the block offset supplying the extra address bits for the large data RAM. In this case, the large RAM is 32 bits wide and must have 16 times as many words as blocks in the cache.

- 2x cache: data, instructions
- 16 KB
- $n_b = 4$
- write through e write back con buffer di scrittura

Miss rate: instructions 0,4%, data 11,4%, combinati 3,2%

### 5.2.17 Diminuire il miss rate

Prima opzione: aumentare la dimensione dei blocchi. Ha qualche contro.

Un'altra opzione: ogni  $x$  può contenere più  $y$  e non solo una → aumentare il grado di associatività

### 5.2.18 Associatività

- Ogni  $x$  lo troviamo in una sola  $y$ : associatività 1-way → NO ASSOCIATIVITY
- Ogni  $x$  lo troviamo in  $n$   $y$ : associatività n-way → di solito  $n = 2^k$
- Ogni  $x$  lo troviamo tutti gli  $y$ : associatività completa →  $k = n_c$

Con  $k = 1$ : many-to-one da  $x$  a  $y$

Con  $k > 1$ : many-to-many da  $x$  a  $y$



### 5.2.19 Trovare / mettere una $x$ con associatività $> 1$

Dove devo guardare nella cache per trovare una  $x$

- Con 1-way: in esattamente 1  $y$
- Con n-way: in  $n$   $y$

Dove devo mettere una  $x$  nella cache?

- Con 1-way: in esattamente 1  $y$
- Con n-way: in esattamente una  $y$  scelta tra  $n$ , come? tramite LRU (least recently used) o randomicamente.

### 5.2.20 Set di blocchi

Con l'associatività, i blocchi  $s_c$  sono organizzati in **set di blocchi**, ognuno formato da  $s_s = 2^{n_s}$  blocchi continui. In pratica  $s_s \in 1, 2, 4, s_c$

Un dato  $x$  può andare in un blocco del  $k$ -esimo set con  $x \% \frac{s_c}{s_s} = k$  (al posto del  $k$ -esimo blocco).  $\frac{s_c}{s_s}$  è il numero dei set di blocchi, nella memoria prende  $n_c - n_s$  bits.

Il tag indica quale  $x$  è in una data  $y$

### 5.2.21 $x$ to $y$

Assumiamo:  $n_s = 2, n_c = 4$

$k$  è l'indice del blocco di  $y$ :

- Il set  $k = 00$  contiene i blocchi da  $y = \underline{0000}$  a  $\underline{0011}$
- Il set  $k = 01$  contiene i blocchi da  $y = \underline{0100}$  a  $\underline{0111}$
- Il set  $k = 10$  contiene i blocchi da  $y = \underline{1000}$  a  $\underline{1011}$
- Il set  $k = 11$  contiene i blocchi da  $y = \underline{1100}$  a  $\underline{1111}$

Da  $x$  a  $k$ :  $x \% \frac{s_c}{s_s} = k$ , che è  $k = x_{[n_m - (n_c - n_s) - n_b, n_m - n_b]}$

Quindi  $x \rightarrow Y = \{y : y_{[0, n_c - n_s]} = x_{[n_m - (n_c - n_s) - n_b, n_m - n_b]}\}$

### 5.2.22 Algoritmo per leggere $x$

1.  $Y = \{y : y_{[0, n_c - n_s]} = x_{[n_m - (n_c - n_s) - n_b, n_m - n_b]}\}$
2. Per ogni  $y \in Y$ 
  - (a)  $t = [y] = (t_{val}, t_{tag}, t_{block})$
  - (b) Se  $t_{val} = 1$  e  $t_{tag} = x_{[0, n_m - (n_c - n_s) - n_b]}$
  - (c) Ritorna  $t_{block}_{[8z, 8z+8]}$  con  $z = x_{[n_m - n_b, n_m]}$  (**hit**)



3. Leggi  $x_0, \dots, x_{s_b-1}$  dalla memoria principale (**miss**)
4. Scegli  $k$  (con  $n_s$  bits)
5. Metti 1  $x_{[0,n_m-(n_c-n_s)-n_b[} [x_0] \dots [s_{s_b-1}]$  a  $y = x_{[n_m,n_m-(n_c-n_s)-n_b[} k_2$
6. Ritorna  $[x]$

### 5.2.23 Scegliere un set di blocchi (LRU)

Per ogni blocco nel set, dobbiamo sapere quando è stato utilizzato l'ultima volta: può essere fatto con un bit  $n_a$  di recency (un tag di utilizzo recente)

- 00 per l'ultimo usato
- 01 per il secondo usato di recente
- 10 per il terzo usato di recente
- 11 per il quasi usato di recente

Ogni volta che si accede a  $y$ , se il suo tag di recency è  $\neq 00$ , le variabili di recency di tutti i blocchi nel set devono essere aggiornate:

- Quella di questa  $y$  diventano 00
- Tutte le altre incrementano di 1 (11 rimane 11)

### 5.2.24 Scegliere un set di blocchi (random)

Solo casuale!

- Non sono necessari tag recenti
- Nessun aggiornamento sui risultati

Molto meno complesso di LRU:

- In pratica, LRU viene utilizzato solo con  $n_s = 1o2$
- L'impatto sul miss rate è basso:  $\approx 10\%$  trascurabile con cache grandi

### 5.2.25 Esercizio - Misses and hits con associatività [#70]

Per una memoria principale di 256 byte, dove  $x = [x]$ , una cache associativa inizialmente vuota con 4 blocchi di 1 word ciascuno legge 10, 11, 13, 20, 21, 22, 10, 20, 21 (indirizzi esadecimali di  $x$ ):

- Calcolare la percentuale di miglioramento nell'hit rate con  $s_s = 2$  rispetto al caso di non associatività
- Calcolare la dimensione della cache complessiva con  $s_s = 2$  e LRU

**Soluzione:**

### 5.3 Memorie Virtuali