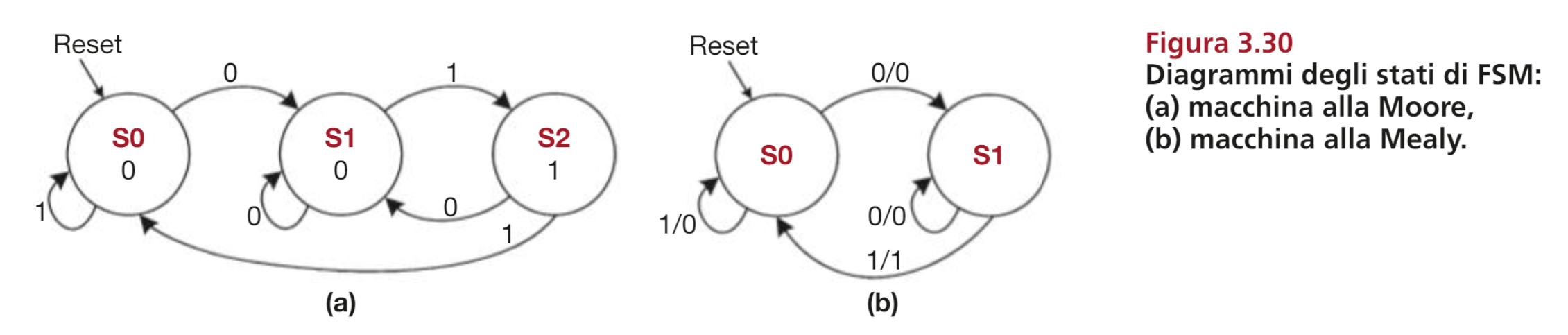
# riassunti AE 2021

Risposte e riassunti degli argomenti di *Architettura degli elaboratori 2020/2021*

## MAcchine di mealy e di moore

L’uscita delle macchine di Moore è caratterizzata solo dallo stato del sistema (registro), mentre le macchine di Mealy anche dal valore dell’ingresso.

Entrambe sono formate da un registro e da due reti combinatorie, una che calcola l’uscita **Z** e una che calcola il prossimo stato **Z**.



La grande differenza tra le due è che le macchine di Mealy sono più veloci di un ciclo di clock.

## I principi del Risc e Di ARM

Innanzitutto, ***RISC*** significa *Reduced Instruction Set Computer*, il che significa che un processore di tipo RISC ha un set di operazioni macchina ridotto su cui è possibile eseguire dei programmi. **ARM** è di tipo RISC, **x86** è di tipo ***CISC*** (*Complex …)*.  
Un set di istruzioni ridotto non significa però meno potente.

I principi di ARM sono i seguenti 4:

1. **La regolarità favorisce la semplicità:** Istruzioni caratterizzate da un numero costante di operandi ( 2 sorgenti e 1 destinazione) sono più facili da programmare e risultano più semplici da generare in hardware.
2. **Rendere veloci le cose frequenti:** Il set di istruzioni e tenuto volontariamente ristretto per permettere una decodifica ed esecuzione veloce.  
   In un sistema **RISC** con 64 operazioni possibili uso per codificare e identificare un istruzioni, in un sistema **CISC** da 256 operazioni ne uso per codificare tutte le istruzioni, e 2 bit in più significa meno informazioni nel resto della word da passare al processore (32 bit di spazio).
3. **Più è piccolo, più è veloce:** ARM usa un set di solo 16 registri globali, e questo set così piccolo è voluto per permettere un accesso molto veloce alla memoria. Se fossero di più avrei un overhead maggiore ogni volta che devo andare a fare un’istruzione (minimo 2 accessi ai registri per ogni operazione, uno per la destinazione, uno per il primo sorgente).

**Un buon progetto richiede i giusti compromessi:** Per avere un’architettura performante e competitiva bisogna anche fare dei compromessi. Le 3 più comuni in ARM sono le operazioni di **LDR/STR** con post/pre Incremento, **PUSH/POP**, e la differenziazione dei formati di traduzione delle istruzioni.

## gestione dei registri in arm

I 16 registri di ARM vengono gestiti in questo modo:

* **R0-R3:** Vengono usati per memorizzare delle variabili temporanee. Hanno un ruolo speciale perché sono 4 registri che posso usare dentro le chiamate di funzione in ***assembly*** come parametri in entrata, poi **R0** viene usato come ritorno di funzione.
* **R4-R12:** Vengono usati per memorizzare delle variabili.
* **R13-R15**: Sono chiamati anche in modo particolare:
  + **R13/SP**: Registro che fa riferimento allo ***Stack pointer.***
  + **R14/LR:** Registro che fa riferimento al ***Link-register***.
  + **R15/PC:** Registro che fa riferimento al ***Program counter***.

Per convenzione **R0-R3** sono considerati registri *“Caller-Saved”*, cioè è il chiamante della funzione che deve salvare il contenuto di quei registri se l’informazione contenuta in essi è rilevante anche dopo la chiamata.  
I registri **R4-R12** invece sono considerati *“callee-saved”*, cioè è la funzione chiamata che deve salvare tutti i registri che userà nello stack all’inizio della funzione e rimetterli in memoria alla fine.

## cpsr

Il ***CPSR*** è il **registro di stato corrente del programma.**È un registro da 32 bit, dove i 4 bit più significativi identificano i 4 **flag di condizione:**

**N** (Negative), **Z** (Zero), **C** (Carry), **V** (oVerflow)

E gli ultimi **5** bit rappresentano i **bit di modo** e rappresentano con quale tipologia di esecuzione il programma viene eseguito (modalità utente, kernel, ecc.…)

## Formati di traduzione delle istruzioni

Il formato di traduzione dei un istruzione è l’ordine logico con cui un compilatore trasforma un istruzione **assembly** in una **word a 32 bit**, dove ogni bit (o insieme di bit) assume un’informazione diversa.

ARM sceglie di avere 3 formati principali di istruzioni: *elaborazione dati, accesso alla memoria e salti.*

### Elaborazione dati

L’istruzione a 32 bit è formata da 6 campi: *cond, op, funct, RnDest, RnSrg, Src2*.  
È gestita nel seguente modo (tra parentesi i bit di riferimento):

* **[31:28] cond:** I primi 4 bit fanno riferimento alla *cond* fa riferimento alle operazioni condizionate. Se settato allora il processore deve guardare che la condizione citata sia soddisfatta.
* **[27:26] op:** Questi 2 bit danno al processore tutte le informazioni necessarie per capire il resto della word. In questo caso la sezione ***op*** ha il valore 00 e identifica un’operazione di elaborazione dati.
* **[25:20] funct:** Questi 6 bit fanno riferimento a *funct* e identificano l’operazione da svolgere. Si spezza nuovamente in 3 sotto-parti:
  + **[25] :** Bit ***I***, se settato a 1 allora il terzo operando è un immediato
  + **[24:21]** :Bit di *cmd*, la loro codifica identifica l’operazione da fare. (Es. 0010 => SUB)
  + **[20] :** Bit di *S,* se settato a 1 allora si ascolta per eventuali bit di condizione. (Se l’operazione fa un carry, vale 0, ecc.…)
* **[19:16] RnSrg:** Registro di sorgente. Bastano 4 bit perché ho al massimo 16 registri.
* **[15:12] RnDest:** Registro di destinazione numero 1. Deve sempre essere un registro.
* **[11:0] Src2:** Registro o immediato.
  + **Se il bit *I==1*:**Allora ho un immediato da 8 bit (che viene ruotato di un determinato valore per fare un valore da 32 bit, ma è poco importante).
  + **Se il bit *I==0*:**
    - **Un registro**
    - **Un registro traslato un numero di volte contenuto in un altro registro.**

### Accesso alla memoria

L’istruzione di accesso alla memoria ha una codifica quasi uguale alla elaborazione dati, con gli stessi 6 campi:

* **[31:28] cond:** Rimane Inalterato.
* **[27:26] op:** in questo caso vale 01.
* **[25:20] funct:** Questi 6 bit cambiano radicalmente da prima e diventano i seguenti:
  + ***I***: Bit dell’immediato. Mi dice se *Src2* è da trattare come un immediato.
  + ***P***: Bit di *Pre-Index.* Mi dice come devo gestire l’indice nell’accesso in memoria. (Pre-incremento, Post-Incremento, non fare nulla, ecc.…)
  + ***U:*** Bit di *Add.* Mi dice se devo aggiungere o sottrarre *Src2* dalla posizione di *RnSrg*.
  + ***W:*** Bit di *WriteBack*. Mi dice se devo andare a sovrascrivere il nuovo valore anche nella memoria dati.
  + ***L e B:*** Bit che specificano il tipo di operazione. *L==1* fa riferimento alla Load, *L==0* alla store. Se *W==1* allora si avrà una *LDRB o una STRB.*
* **[19:16] RnSrg:** Registro base da cui recuperare le informazioni in memoria.
* **[15:12] RnDest:** Registro base da cui recuperare le informazioni in memoria.
* **[11:0] Src2:** Registro o immediato.
  + **Se il bit *I==1*:** Allora ho un immediato da 12 bit.
  + **Se il bit *I==0*:** Allora ho un registro.

### Istruzioni di salto

Questa formattazione è drasticamente diversa rispetto alle altre due. La word da 32 bit è divisa in soli 4 campi: *cond, op, funct (da soli 2 bit) e un immediato da 24 bit.***Cond** è sempre la stessa, e **op** == 01.   
Il campo **funct** è di soli 2 bit, di cui il più significativo è sempre 1. Se quello meno significativo è uguale a 1 allora abbiamo una **BranchAndLink**, altrimenti una **Branch** normale.  
L’***immediato*** da 24 bit contiene il numero di istruzione a cui saltare.

## Istruzioni thumb

Le istruzioni ***Thumb*** sono lunghe 16 bit e svolgono le stesse cose delle operazioni a 32 bit (con qualche limitazione e compromesso). Le operazioni ***Thumb*** vengono usate per aumentare la densità del codice (con una sola word da 32bit posso fare due operazioni).

I compromessi da tenere in mente quando si usano le operazioni thumb sono i seguenti:

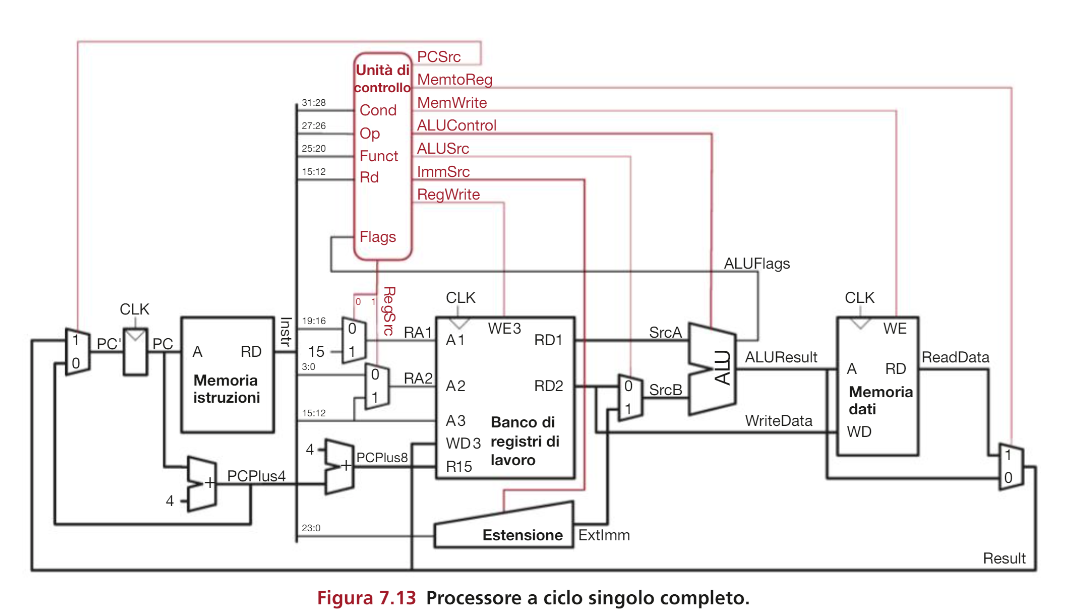
* **Abbiamo solo 8 registri:** Solo i Registri da **R0-R7** sono disponibili (3 bit per determinare il registro invece che 4)
* **Un registro viene usato sia come sorgente che come destinazione.**
* **Gli immediati sono più corti** (da 5 a 8 bit)
* **Non dispongono della esecuzione condizionata**
* **Il flag di stato viene modificato sempre.**

Le istruzioni Thumb sono utilissime perché riducono le dimensioni e il costo della memoria, ma perché necessitano solo di un bus a 16 bit per la memoria, riducendo così la potenza consumata per il fetch delle istruzioni.

## Microarchitettura a ciclo singolo

La microarchitettura a ciclo singolo è concettualmente la più semplice tra le microarchitetture che abbiamo visto, ma nasconde alcune difficoltà di realizzazione (Es: memoria dati e istruzioni separate, di difficile realizzazione sul silicio).

In un unico ciclo di clock completa un’istruzione, perciò non ha bisogno di nessun stato non architetturale (a.k.a. non ha bisogno di registri intermedi o particolari blocchi oltre a delle ALU, memorie dati, istruzioni, registri e PC).

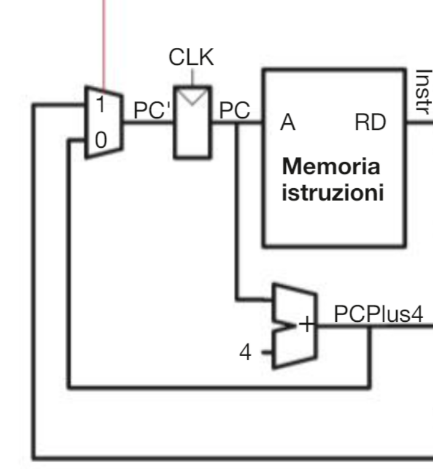




Il processore è il seguente. Si possono vedere i due path principali: il **path dati** e il **path di controllo** (in rosso).

Il path di dati si comporta in questo modo:

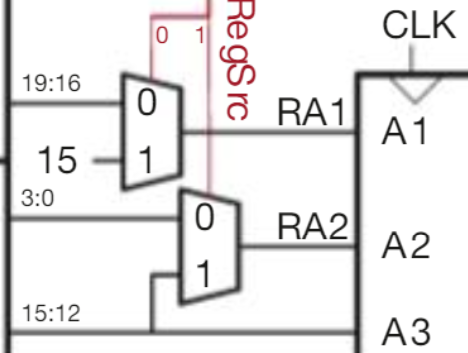
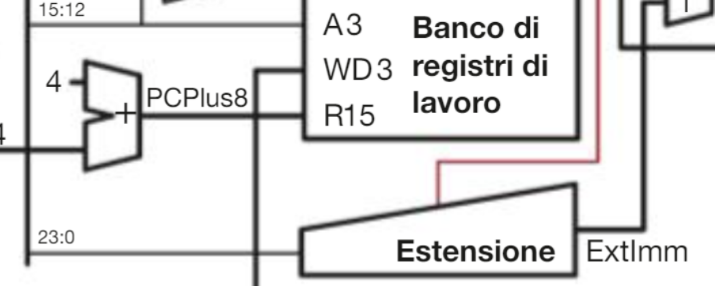
### Zona del Program counter

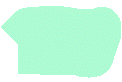
La zona del **PC** è la più semplice. Un solo registro che gestisce il valore del Program Counter del Processore. Il valore del PC cambia ad ogni ciclo, e dipende solo da una cosa: la ***PCSrc***. La ***PCSrc*** è un bit della unità di controllo che mi gestisce la sorgente del PC.   
Se è settata a 0 allora il nuovo valore del PC’ è PC+4 (incrementato dalla ALU In basso a destra), se invece è settato a 1 allora il nuovo valore del PC è il risultato della ALU blu, dopo un’istruzione di Branch.

Dopo aver recuperato il valore del PC dal suo registro mando questo valore alla memoria istruzioni, che recupera la nuova parola da 32 bit che è l’istruzione successiva.

### Accesso ai registri

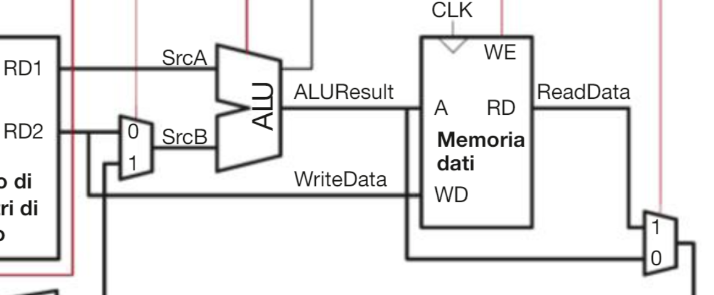
Recuperata la word a 32 bit allora inizio la fase decisionale. I primi 12 bit vanno nella zona di controllo, che capisce che operazione sto svolgendo e decide come si deve comportare di conseguenza settando i flag nel modo giusto.

 Il primo multiplexer (di colore verde) sceglie il primo registro sorgente. Se è una operazione di Branch allora ***RegSRC==1***e il registro sarà il **PC**, altrimenti è il registro dichiarato nella word in posizione **19:16.**   
Il secondo multiplexer prende due parti dell’istruzione.   
Se ho una operazione di accesso di memoria (op==01) allora il secondo registro (**15:12**) fa riferimento alla base da cui devo partire a contare, altrimenti (op==00) vado a prendere il registro identificato da **3:0**. Se ho un immediato mi importa poco, dopo l’accesso in memoria ho un multiplexer che mi gestisce questa scelta.



Più sotto ancora vediamo una **ALU** che aggiorna il valore del **PC** nel banco dei registri. Nel caso dell’immediato ho un blocco che mi estende il tutto a 32 bit (stessa grandezza delle informazioni contenute nei registri).

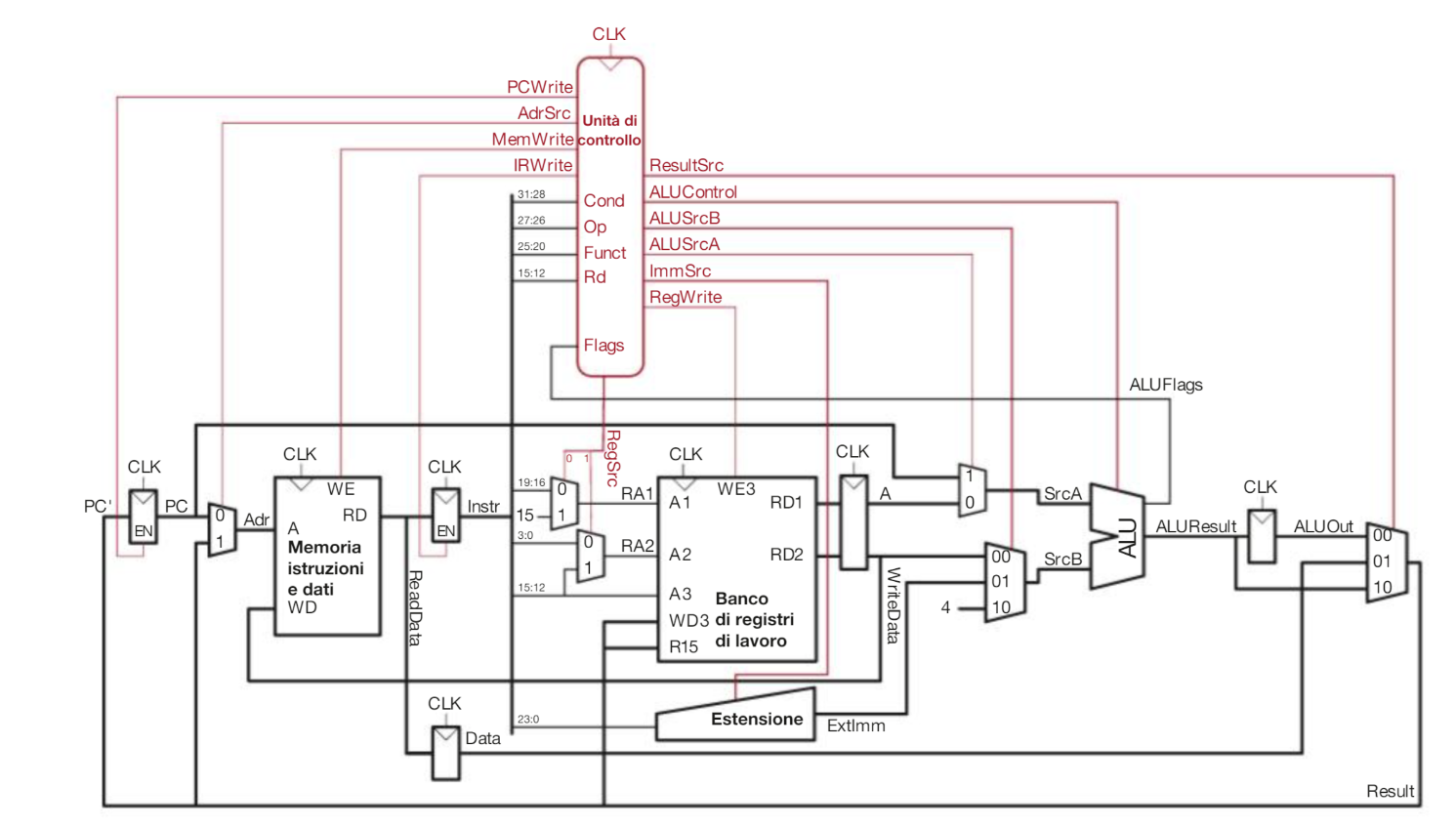
### ALU E WRITE BACK

L’ultima parte è quella della ALU e del Write Back.  
La **SrcA** è il contenuto del primo registro, la **SrcB** invece viene scelto tramite un multiplexer tra il contenuto del secondo registro e l’immediato. Il contenuto del secondo registro viene mandato anche alla memoria dati, nel caso di una operazione di LDR o STR. La **ALU setta sempre i flag NZCV**. Il risultato viene passato poi alla memoria, che se necessario recupera/aggiorna il valore in posizione WD. Se era una LDR allora il risultato viene poi passato indietro e viene aggiornato anche nel Banco dei registri/PC.

### Debolezze del Processore a ciclo singolo

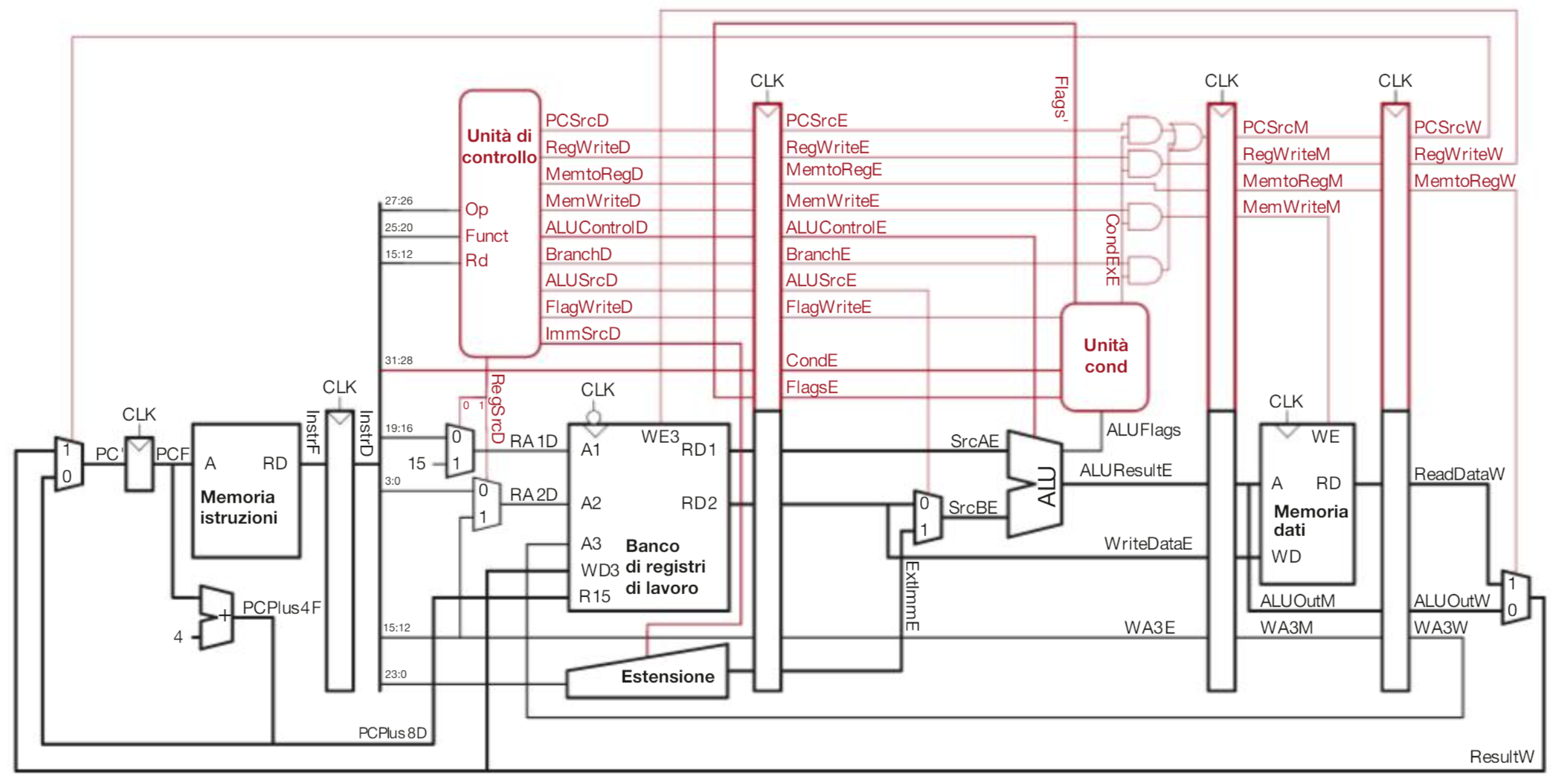
1. Il ciclo di **clock** è dettato dall’esecuzione del comando più **lungo.** (LDR)
2. Richiede **memorie separate** per la gestione dei dati e delle istruzioni.
3. Minimo **3 ALU.** 1 per la somma dei registri e 2 per il **PC**.

## Microarchitettura a ciclo Multiplo

Le debolezze del Processore a ciclo singolo vengono riviste e migliorate nel processore a ciclo Multiplo.

Il circuito è leggermente più complicato, e spuntano fuori alcuni registri intermedi.  
Come prima cosa si può vedere che c’è una sola ALU, che gestisce sia la logica del PC sia la logica delle elaborazioni Dati.  
Il processore esegue solo un’operazione alla volta, perciò il suo ciclo di clock è molto più piccolo rispetto a un processore a ciclo singolo, ma impiega dai 3 ai 5 cicli di clock per eseguire una sola operazione. I registri intermedi non amministrativi servono al processore a mantenere stabili le informazioni del processore durante i vari cicli di clock.

## microarchitettura pipeline



Il percorso del processore Pipeline è ottenuto dividendo il percorso del processore a ciclo singolo in 5 stadi (rappresentati dai 4 registri di stato):

* **Fetch (F):** Stadio di Fetch, recupera l’istruzione in memoria.
* **Decode (D):** Stadio di Decode, decodifica l’istruzione successiva da eseguire.
* **Execute (E):** Stadio di Execute, porta a termine l’istruzione.
* **Memory (M):** Accesso alla memoria, scrivo/leggo in memoria il dato.
* **WriteBack (W):** Ultimo stadio, scrivo nel banco dei registri il nuovo valore.

Questa impostazione qui a compartimenti stagni porta molti benefici ma anche qualche problematica:

##### Pro

A regime, il processore pipeline riesce ad eseguire molte più istruzioni del SingleCicle e del MultyCicle.

Grazie alla parte di controllo sezionata tramite i registri posso controllare le varie sezioni in modo indipendente.

##### Contro

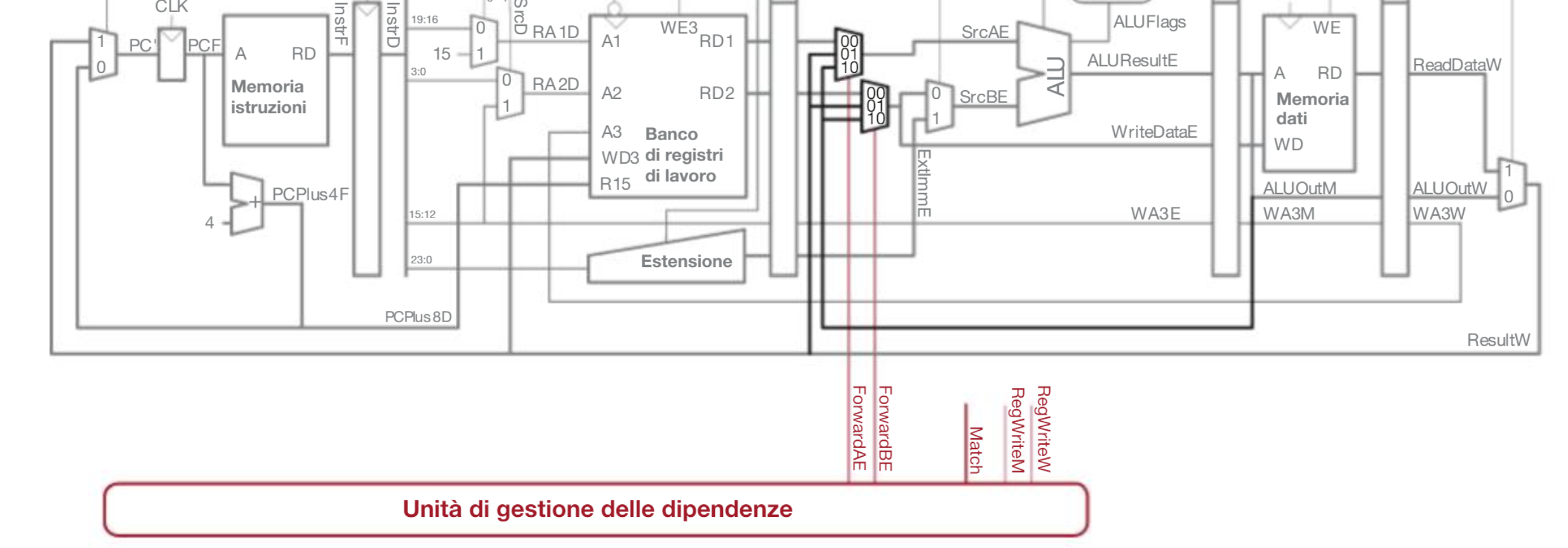
Il **WriteBack** può generare delle dipendenze a livello di codice. Se due istruzioni contigue accedono al solito registro, prima in scrittura e poi in lettura allora genero una dipendenza che può causare delle problematiche non indifferenti.

I salti (condizionati o meno) mi generano delle bolle, poiché devo andare a scartare minimo 2 fasi di fetch istruzioni per caricare il nuovo blocco di istruzioni.

Entrambe le problematiche vengono risolte con delle accortezze.

## forwarding

Per gestire le dipendenze della pipeline si può aggiungere una parte al processore chiamata **Unità di gestione delle dipendenze.**



Questa unità consiste nell’aggiungere un multiplexer davanti alla ALU che va a veder se devo recuperare il valore del registro dal **WriteBack** della memoria o dal **risultato della ALU**. In questo modo riesco a gestire quei casi in cui ho appena calcolato il nuovo valore del registro/appena inserito il nuovo valore in memoria ma non ho ancora aggiornato il suddetto valore nel banco dei registri.

Per fare ciò ho bisogno di un pezzo di logica aggiuntiva che mi va a controllare che l’intersezione cartesiana degli operandi (dell’operazione che ho appena decodificato) con l’uscita della memoria/ALU sia vuoto, se risulta qualcosa allora significa che ho una dipendenza, e devo usare il forwarding.

## Dipendenze

In poche parole, le dipendenze nella pipeline si generano nei seguenti casi:

1. **RAW:** Nella ReadAfterWrite creo una dipendenza funzionale perché sto cercando di leggere un dato che non è ancora stato calcolato e aggiornato nel blocco dei registri.
2. **Dipendenze di controllo:** Si verificano quando la decisione di quale sia la prossima istruzione da prelevare non sia ancora stata presa al momento della fetch

Per risolvere queste dipendenze ci sono alcuni accorgimenti che si possono sfruttare nella pipeline.

## Pipeline lunghe

Per ora abbiamo visto un processore pipeline a 5 stadi, ma nella realtà sono più comuni processori da 10/15 stadi.   
Riducendo la grandezza degli stadi è possibile velocizzare il tempo di clock (ogni stadio contiene meno logica).   
Pipeline più lunghe **aggiungono** ulteriori dipendenze, ma la maggior parte di esse viene **risolta tramite un inoltro (forwarding).**

Il limite delle pipeline lunghe (in termini di stadi) sono le dipendenze, il costo (tanti stadi => molti registri) e il sovraccarico di sequenziamento. Questo sovraccarico è dato dal ritardo del clock in uscita e dal ritardo di setup dello stadio.

## Previsione dei salti

Come visto, una delle grosse problematiche del Pipeline sta nella sua previsione dei salti. Se il processore “prevede” come andrà una istruzione e la sua previsione è sbagliata allora siamo in una situazione dove dobbiamo buttare tutti gli stadi precedenti e rincominciare da capo. In una pipeline lungo è ancora più nefasta come cosa e risulta in una crescita del nostro **CPI** (*Clock Per Instruction*).

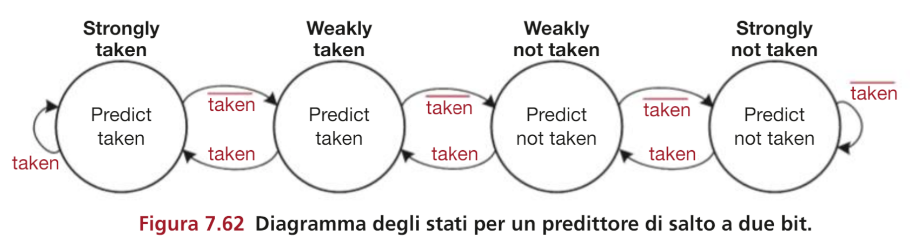
Per questo motivo vengono integrati dei previsori di salto, che possono essere di tipo **Statico** o **Dinamico.**

##### Previsore dei salti statico

La previsione dei salti statica si basa sulla **direzione del salto, principalmente se è all’indietro**.

Se la direzione del salto è all’indietro allora il processore suppone che il salto venga preso, dato che nella maggior parte dei casi si ha un salto all’indietro nei cicli **while/for.**   
Questo metodo è buono ma non infallibile, poiché non prende in considerazione i salti in avanti.

##### Previsore dei salti dinamico

Il previsore dei salti **Dinamico** si basa sulla storia del programma in memoria. Per fare ciò, tiene in memoria una **tabella** denominata ***buffer delle destinazioni di salto***, che include tutte le destinazioni relative a ciascun salto e tutti gli esiti, ovvero se è stato eseguito o meno in passato.  
Si usa un automa a stati finiti a 2 bit (a 1 bit risulta troppo semplice, causando il fallimento sistematico nella prima e ultima iterazione del salto), che dichiara 4 stati possibili:

**Strongly taken,*weakly taken, weakly NOT taken,* strongly NOT taken.**

Quando un ciclo viene eseguito per la prima volta l’automa entra subito nella sezione “**Weakly NOT taken**”, quindi ha ancora un po’di incertezza a riguardo. La seconda volta che si trova davanti al salto e non lo prende, si sposta nella sezione “**strongly NOT TAKEN**”, e rimarrà lì fino a che non finisce il ciclo, di fatto sbagliando solo l’ultima previsione.

# Processori superscalari

Un processore superscalare è un processore pipeline a n-stadi che contiene un numero x di copie dell’hardware (ALU) e memorie a *x* moduli interlacciate per poter eseguire in parallelo più operazioni. In un mondo ideale il processore superscalare a **2 vie** ha un **CPI**==0.5, perché in un ciclo di clock può eseguire due istruzioni in parallelo, ma purtroppo non è così.   
Si possono generare degli stalli dovuti a delle dipendenze di codice, dove un risultato di un registro è richiesto nell’esecuzione parallela dell’altra istruzione.

### Processori OUT-OF-ORDER

Spesso è un’operazione che fa il compilatore per rendere più fruibile il codice, ma è utile avere questo processo a run-time per migliorare ulteriormente le prestazioni del processore.

Pagando il prezzo di un blocco logico in più, il processore pipeline (normale, superscalare, lungo, ecc.) può esaminare un certo numero di istruzioni future e riordinarle in modo da eliminare le dipendenze, purché il risultato dell’esecuzione fuori posto non cambi il significato del programma.

## Sistemi multithread

### Cosa sono

I sistemi multithread sono processori moderni, evoluti da una necessita nata nel 2005 e dalle ovvie limitazioni del processore superscalare unico. Ogni CPU è composta da più **“Core”**, che altro non è che una copia di un processore ma di dimensione ridotta. Un sistema multiprocessore consiste di fondo in diversi processori interconnessi, che possono essere **omogenei** (tutti della stessa dimensione) ed eterogenei(di dimensione diversa tra loro, tipo Big.Little degli smartphone).

### multithreading interleaving (O Intel hyper-threading)

Per multithreading Interleaving (O hyper-threading) si intende un processore che esegue due o più processi contemporaneamente, schedulando prima un’istruzione di un processo, poi un’istruzione del secondo processo, poi la seconda istruzione del primo processo, e così via. Questo fa sì che, in processori pipeline, possa ridurre drasticamente le bolle di esecuzione e le dipendenze logiche date dalle **ReadAfterWrite**.

## Tipi di Località

* **Località spaziale:** Se il processore accede ad un dato in memoria, la probabilità che acceda ad un dato limitrofo è molto alta (istruzioni adiacenti stanno in blocchi di pagina adiacenti).
* **Località temporale:** Se il processore accede ad un dato in memoria, la probabilità che acceda allo stesso dato una seconda volta di lì a poco è molto alta (caso di un for)

## CAChe

Le cache sono memorie molto veloci di dimensione ridotta che stanno accanto al processore, e fanno da prima barriera di dati rispetto alla memoria principale. Non essendo di dimensioni tali da poter tenere in memoria l’intero programma si genera un problema: **La gestione delle pagine e il loro relativo fetch.**  
Se il processore chiede un’istruzione ed è **presente nella cache**, allora si parla di **cache HIT.**  
Se il processore chiede un’istruzione e questa **NON è presente nella cache,** allora si parla di **cache MISS** e di **politiche di rimpiazzo**.

Ogni **blocco di cache (B)** può contenere **b parole (istruzioni)** 🡸🡺 **b è la dimensione del blocco**.

### cache a mappatura diretta

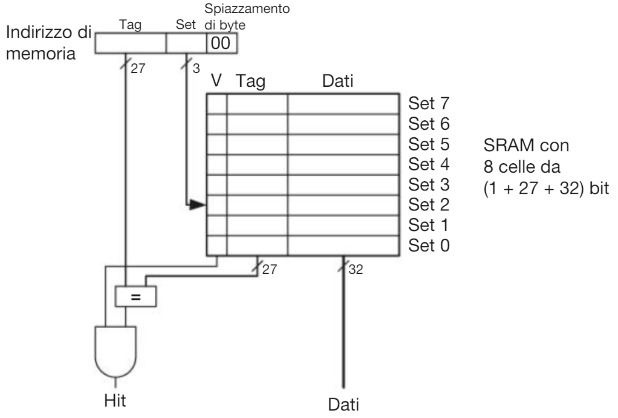
La cache a mappatura diretta è il tipo di cache più semplice e veloce. Se la cache ha dimensione 10 allora può contenere 10 blocchi di memoria (***S==B****)*.

Essendo a indirizzamento diretto, un indirizzo del **blocco** **0** in memoria principale viene mappato nel **set 0** della cache. Essendo la cache molto più piccola della memoria di molti ordini di grandezza, succede che ***n blocchi di memoria vengano mappati nel solito set***.

Per identificare e capire che blocco di memoria ho in cache, gestisco gli indirizzi a 32 bit nel seguente modo:

* **[31:5] Tag:** I 27 bit più significativi li uso come TAG e indicano l’indirizzo del dato contenuto nella cache.
* **[4:2] Set di byte:** Indicano in quale set viene mappato l’indirizzo.
* **[1:0] Spiazzamento:** indicano un byte all’interno della parola.

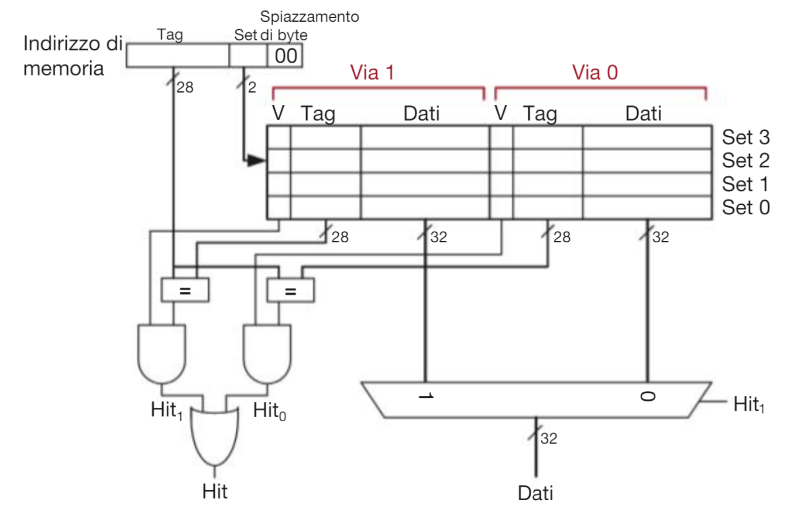
La cache a questo punto usa l’indirizzo di memoria dell’istruzione passata dal processore per determinare la presenza dell’informazione. Per fare ciò esegue i seguenti passi:

1. **Prende il set e recupera la riga nella cache.**
2. **Passa il Tag dell’indirizzo e il Tag del set di cache ad un confrontatore.**
3. **Passa il risultato del confrontatore ad un blocco AND insieme al bit di validità.**
4. **Se il risultato dell’AND == 1 allora abbiamo una cache HIT, altrimenti una cache MISS**

### CACHE a mappatura parzialmente associativa

Una cache a mappatura parzialmente associativa **riduce le collisioni** prevedendo **N blocchi per ogni set.** Di fondo, mette più SET di cache a mappamento diretto in parallelo, recupera da tutti le informazioni e poi con dei confrontatori determina le cache hit e cache miss.

Questo tipo di cache, a parità di numero di Set, ha generalmente un numero di **miss** inferiori rispetto a quella ad indirizzamento diretto, ma è leggermente più costosa e lenta in termini di hardware. Nonostante ciò la maggior parte dei sistemi usa memorie di questo tipo.

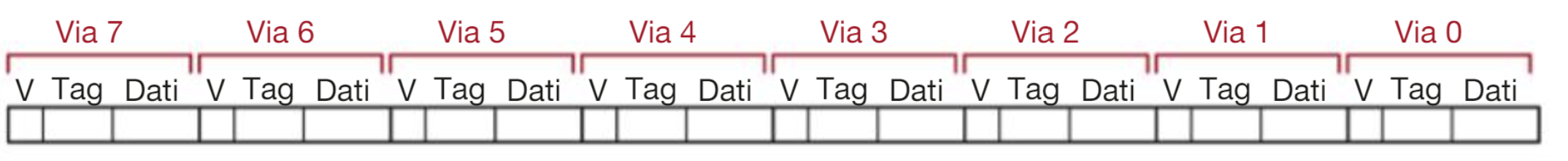
Più nel dettaglio, una **cache a due vie** funziona nel seguente modo:

1. Passo il Set dell’indirizzo alla cache, che in **parallelo recupera tutti i dati da tutti e due i Set**.
2. Entrambi i tag vengono passati ad un confrontatore.
3. Il valore di questi questi confrontatori viene passato ad un AND con il bit di validità.
4. Entrambi i risultati vengono mandati ad un OR, che decide se è stato fatto una Hit o una MISS.

### CACHE a mappatura completamente associativa

Questo tipo di cache è costituita da un unico set a B vie, perciò un indirizzamento di memoria può essere mappato ovunque.

Non è altro che una cache parziale con un solo set, multiplexer a B ingressi, B confrontatori, ecc…



### Politiche di aggiornamento della Cache

Ci sono due possibili politiche di aggiornamento delle informazioni presenti nella cache:

* **Politica Write-Through:** Aggiorna la cache e in maniera asincrona riporta l’aggiornamento ai livelli superiori.
* **Write Back:** Per ogni entry della cache tengo un bit in memoria che mi dice se il record è stato modificato o meno. Quando decido di dover eliminare quel particolare entry dalla cache controllo il bit, e se è settato a 1 vado a riportare il valore in memoria principale.

## Traduzione degli indirizzi

### Tabella delle pagine

Il processore usa la tabella delle pagine per tradurre in indirizzi fisici gli indirizzi virtuali. Ogni entry di questa tabella mappa un indirizzo di pagina fisica con quello virtuale, insieme ad un bit di validità. Siccome è una tabella molto grande viene memorizzata in memoria fisica.

### Translation lookaside buffer

Essendo la tabella delle pagine enorme, costerebbe tantissimo al processore interrogarla continuamente. Per ovviare a questo problema si usa una **Translation Lookaside Buffer,** o **TLB**.

È una tabella mappata esattamente come la tabella delle pagine, ma di dimensione nettamente più piccola, di solito dai 16 ai 512 record.

# Domande

### Logica Combinatoria: cosa si può fare con la rete sequenziale rispetto a quella combinatoria?

Le reti sequenziali estendono le reti combinatorie inglobandole e aggiungendoci dei registri di stato, che permettono di mantenere e salvare un informazioni al loro interno tra i vari cicli di clock.

### Memoria: come si traduce un ind virtuale in fisico?

Un indirizzo virtuale a 32 bit viene tradotto in indirizzo fisico utilizzando ***N*** bit più significativi per identificare la pagina di appartenenza, e i restanti pre recuperare l’entry giusta all’interno della pagina.

### Come avviene l'esecuzione di un istruzione condizionale in single cycle ?

Il Decoder e l’unità di controllo determinano se l’istruzione deve essere eseguita o meno dal processore, in base ai flag di stato, e mandano alle memorie un valore booleano di permesso alla scrittura in base alla condizione verificata o meno.

### Verilog: che differenza c'è tra simulazione e sintesi?

* **Simulazione:** Dato l’insieme di moduli verilog ed eventuali test posso simulare il comportamento del circuito in software come GTKWave, e veder che si comporti nel modo corretto
* **Sintesi:** Dato l’insieme di moduli verilog vado a sintetizzare il pattern di un circuito in silicio che eventualmente potrebbe essere stampato e si comporterebbe esattamente come descritto dalla simulazione dei componenti.

### Processore pipeline con controllo: Cosa serve la unità di controllo

L’unità di controllo della ALU serve per mandare indietro i flag ZNCV agli stadi precedenti, così da permettere una logica decisionale ed evitare delle dipendenze.

### Memoria: come funziona il meccanismo di paginazione dinamica?

Mantenere in cache tutta la tabella delle pagine sarebbe troppo costoso, perciò si mantiene in memoria. Per ovviare al ritardo di chiamata che sopraggiunge con questa posizione della tabella si usa una **TLB** di dimensioni piccole che sta in cache e tiene solo i riferimenti alle pagine utilizzate dal processore.

### Quanti livelli di porte AND e OR per un decoder sapendo il numero di ingressi?

Un decoder è un blocco di logica combinatoria da **N** ingressi, che presenta in uscita uscite, dove solo quella corrispondente al valore binario dell’ingresso è settata a 1. Ogni filo viene sdoppiato e uno dei due capi viene negato.

Gli AND sono tanti quante le uscite.

### Salto condizionale nel processore pipeline: conseguenze sul flusso di esecuzione

Il salto condizionale in un processore pipeline ha l’effetto di far buttare un numero x di cicli di clock all’esecuzione poiché deve caricare nuovamente un intero set di istruzioni.

### Memoria: come funziona la TLB?

La TLB è una tabella piccola contenente le ultime reference alla tabella di pagine in memoria.  
Ogni volta che la CPU richiede un riferimento di pagina controlla la TLB e, se presente e valida, gli restituisce tutte le informazioni, altrimenti esegue una chiamata alla memoria e, una volta recuperata la nuova pagina, sostituisce una pagina non più usata da tempo.

### Come si gestisce un fault in una memoria ad ind. diretto

Si prende il set e si recupera il tag della riga di riferimento, se i due tag non matchano o il bit di validità del record è uguale a 0 allora abbiamo un fault. La nuova entry della cache sostituirà il record nella stessa posizione, perché sono mappati in modo diretto.

### Processore single cycle (fig 7.14): perché logica e decoder sono divisi in due parti?

Il decoder principale può settare a 1 solo un wire tra tutti quello che escono dal blocco di logica, cosa che limiterebbe alcune operazioni all’interno del ciclo.

La logica condizionale deve essere in grado di gestire contemporaneamente il PC e la scrittura nei registri e nella memoria.

### convenzioni per passaggio registri e controllo quando si chiama una funzione.

I primi 4 Registri devono essere salvati dal chiamante della funzione, mentre per i registri R4:R12 deve essere premura del chiamato.

### Come si usano le mappe di karnaugh per minimizzare una rete?

Le mappe di Karnaugh sono un modo per ridurre al minimo una formula logica. Si dispongono gli operandi in una tabella righe/colonne, e per ogni riga/colonna si va ad associare un valore binario che deve variare di solo 1 carattere da quelli adiacenti.

Una volta disposta la tabella si raggruppano tutte le aree rettangolari lunghe , con n numero naturale.

### differenza tra interrupt e fast interrupt

Le interrupt sono dei meccanismi asincroni per comunicare con il processore. Servono ai dispositivi I/O e Processi per avvertire il processore che un determinato avvenimento è accaduto. Le **fast interrupt** sono così caratterizzate perché vengono eseguite prima rispetto alle interrupt normali, anche se arrivano dopo.

### Cosa sono le dipendenze logiche e che impatto hanno sul pipeline? Come si puo agire sul codice?

Le dipendenze logiche del codice hanno un effetto nefasto nelle prestazioni del pipeline, poiché creano delle bolle logiche dove il pipeline è costretto a mettere in stallo una serie di stadi attendendo che la dipendenza venga risolta.

Si può agire sia lato codice sia lato macchina:

* Forwarding
* Previsione dei Salti
* Out-Of-Order
* Hyperthreading

### Differenza tra rete sequenziale di mealy e di moore? Quale più veloce?

Le reti sequenziali si differenziano in due tipologie. Quelle di Moore determinano l’uscita della rete basandosi sullo stato interno della rete, quelle di Mealy invece determinano l’uscita della rete in base allo stato interno e al valore di ingresso della rete.

Sono più veloci quelle di Mealy di un ciclo di clock.

### Che cos'è un processore superscalare?

Un processore superscalare è un processore pipeline dove vengono duplicate le risorse hardware e le memorie vengono fatte interallacciate.

In questo modo è possibile eseguire un numero di operazioni in parallelo.

### Fare un esempio di come nella microarchitettura aggiungendo pezzi di hardware si aumentino le prestazioni

* Previsione dei salti
* Gestione delle dipendenze
* Forwarding