Microarchitettura

Rappresenta l'anello di congiunzione tra i circuiti logici e l'architettura, consiste infatti nella specifica combinazione tra registri, ALU, macchine a stati finiti, memorie e altri blocchi logici necessari per la realizzazione dell'architettura. L'architettura del calcolatore é quindi definita da un set di istruzioni e da uno stato architetturale.

Stato architetturale

Lo stato architetturale del processore ARM é definito dal contenuto di 16 registri a 32 bit e di un registro di stato. A partire dallo stato architetturale corrente, il processore esegue una particolare istruzione su un particolare insieme di dati per produrre un nuovo stato architetturale.
Alcune microarchitetture possono contenere anche uno stato non architetturale aggiuntivo, utile per semplificare le reti logiche o migliorare le prestazioni.

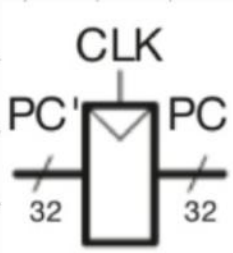
Tipologie di istruzioni

- Istruzioni di elaborazione dati: ADD, SUB, AND e ORR con indirizzamento a registro e immediato senza traslazioni.
- Istruzioni di accesso in memoria: LDR e STR con spiazzamento immediato positivo.
- Istruzioni di salto: B

Datapath e Control path

É opportuno dividere le microarchitetture in due parti: il percorso dati (datapath) e l'unità di controllo (control path). Il percorso dati opera su parole di dati: é costituito da strutture come le memorie, i registri, l'ALU e i multiplexer. L'unità di controllo riceve l'istruzione corrente dal percorso dati e comunica al percorso dati come eseguirla, attivando opportunamente gli ingressi di selezione del multiplexer, le abilitazioni dei registri e i segnali di lettura/scrittura. Dato che l'istruzione da eseguire viene letta da una zona di memoria, e LDR e STR leggono e scrivono dati in un'altra zona di memoria, é opportuno dividere tale memoria in due parti: una contenente istruzioni e l'altra contenente dati.

Program counter (PC)



Anche se parte del banco di registri, il PC viene letto e scritto in ogni ciclo di clock e conviene realizzarelo come un registro autonomo a 32 bit. La sua uscita, PC, punta all'istruzione corrente, mentre il suo ingresso, PC', é l'indirizzo della prossima istruzione da eseguire.

Memoria istruzioni

Ha una sola porta in lettura (simile ad una ROM), riceve in ingresso un indirizzo di istruzione a 32 bit (A) ed emette sull'uscita di lettura dato (RD) il dato a 32 bit contenuto nella parola di indirizzo A.

Register file

Il banco di registri di 15 elementi da 32 bit contiene contiene i registri R0 - R14 e ha un ingresso aggiuntivo per ricevere R15 dal PC. Ha due porte in lettura e una in scrittura; le porte in lettura ricevono in ingresso due indirizzi a 4 bit, A1 e A2, ciascuno dei quali specifica uno dei $2^4 = 16$ registri come operando sorgente ed emettono sulle uscite, RD1 e RD2, i valori a 32 bit dei registri indirizzati. La porta in scrittura riceve in ingresso un indirizzo a 4 bit, A3, un dato WD3, un segnale di abilitazione alla scrittura WE3 e il clock. Se il segnale è attivo il banco di registri memorizza il dato nel registro specificato in corrispondenza del fronte di salita del clock.

Memoria dati

La memoria dati ha una sola porta in lettura/scrittura. Se viene attivato il segnale di abilitazione, WE, il dato in ingresso WD viene scritto nella parola di indirizzo A in corrispondenza del fronte di salita del clock. Se WE = 0 il contenuto di A viene emesso sull'uscita RD.

Questi tre elementi di microarchitettura sono detti circuiti sequenziali sincroni in quanto gli elementi di stato modificano il loro stato solo in corrispondenza dei fronti di salita del clock.

Prestazioni

Il miglior modo per testare le prestazioni di un processore o microprocessore è attraverso i benchmark, ovvero una collezione di programmi simili a quelli che si pensa di dover eseguire. Il tempo totale per eseguirli rappresenta le prestazioni di quel processore (minore è, meglio è).

$$t_{exec} = (\#\ \text{istruzioni}) \left(\frac{\text{cicli}}{\text{istruzione}}\right)\left(\frac{\text{secondi}}{\text{ciclo}}\right)$$

$\frac{\text{cicli}}{\text{istruzione}} \rightarrow$ CPI

$\frac{\text{secondi}}{\text{ciclo}} \rightarrow T_c$

Il CPI (Cycles per Instruction) è il numero di cicli di clock richiesti in media per eseguire un'istruzione. È il reciproco della potenza di elaborazione (IPC, Instruction per cycle).
Il numero di secondi per ciclo è il periodo del clock, T_c. Tale periodo è determinato dal percorso critico attraverso i circuiti del processore.

Processore single cycle

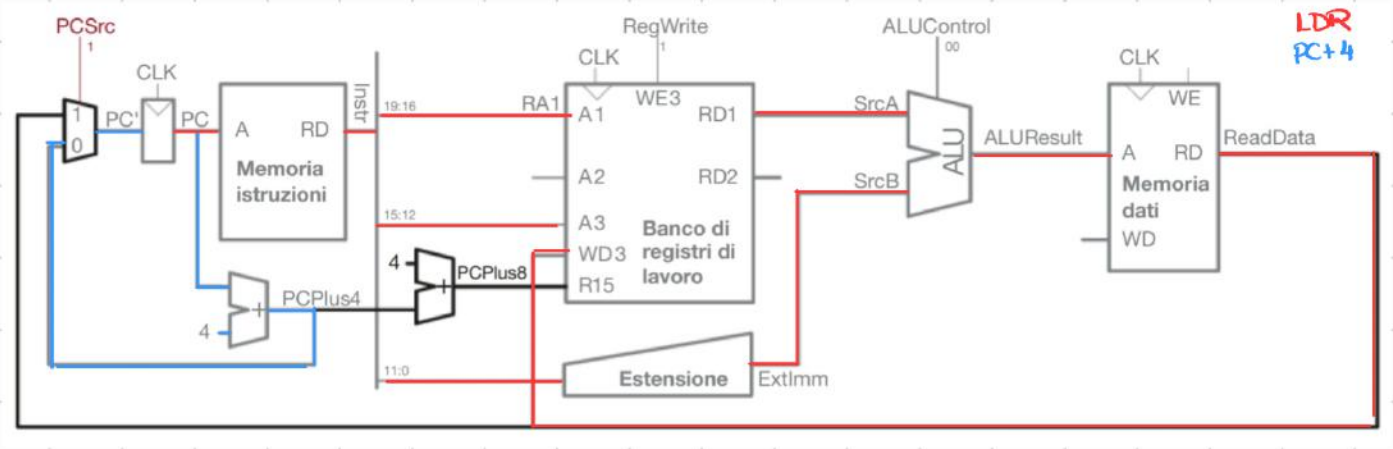
Esegue un'istruzione in un ciclo di clock, quindi non ha bisogno di alcuno stato non architetturale. Il tempo di ciclo è imposto dalla istruzione più lenta e il processore richiede memoria istruzione e memoria dati separate.

Percorso dati

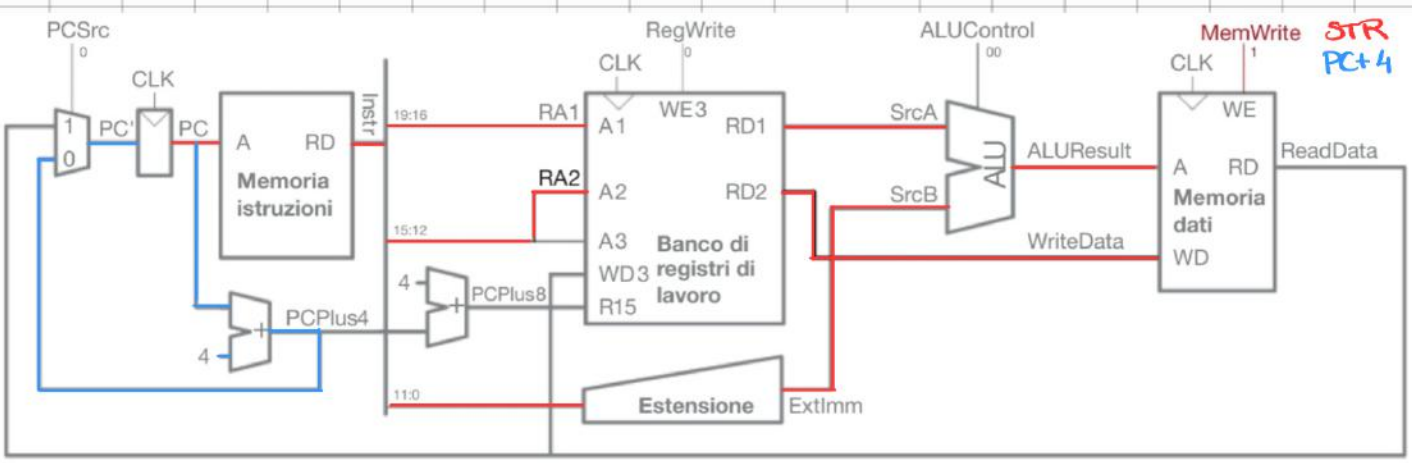
Il PC contiene l'indirizzo dell'istruzione da eseguire. Per prima cosa si deve dunque leggere l'istruzione dalla memoria istruzioni. Il PC è quindi collegato all'ingresso di indirizzo della memoria

istruzioni, che emette l'istruzione a 32 bit, denominata "Instr", in modo che possa essere prelevata dal processore. Questa fase prende il nome di fase di fetch e le successive attività del processore dipendono dall'istruzione prelevata:

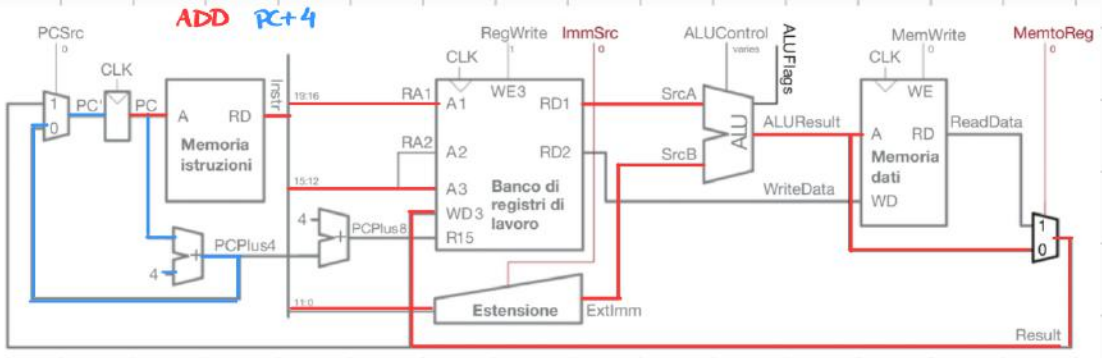
- LDR: per eseguire l'istruzione LDR, il prossimo passo è leggere il registro sorgente contenente l'indirizzo base (R_n) cioè $Instr_{19:16}$. Questi bit vengono collegati agli ingressi di indirizzo di una delle porte (A1) del register file.
L'istruzione LDR richiede anche uno spiazzamento, memorizzato nel campo immediato dell'istruzione $Instr_{11:0}$, valore senza segno esteso con zeri fino a 32 bit, denominato "ExtImm". Il processore a questo punto somma lo spiazzamento al registro base, grazie ad un'ALU, per ottenere l'indirizzo da cui leggere il dato.
Il dato viene emesso dalla memoria dati sul bus ReadData e scritto nel registro destinazione (R_d) cioè $Instr_{15:12}$, a fine ciclo.
Il bus ReadData è collegato agli ingressi di dato della porta di scrittura WD3 e i bit $Instr_{15:12}$ sono collegati agli ingressi di indirizzo A3.
Mentre l'istruzione viene eseguita, il processore deve calcolare l'indirizzo dell'istruzione successiva PC' = PC+4 che viene scritto nel PC in corrispondenza del prossimo fronte di salita del clock. Per fare ciò si aggiunge un altro sommatore per incrementare PC e passare il risultato alla porta R15 del register file.
Il valore PC può però provenire dal risultato dell'istruzione invece che da PC+4 quindi serve un multiplexer per scegliere tra le due possibilità. Il segnale di controllo PCSrc è messo a 0 per selezionare PCPlus4 e a 1 per selezionare ReadData.



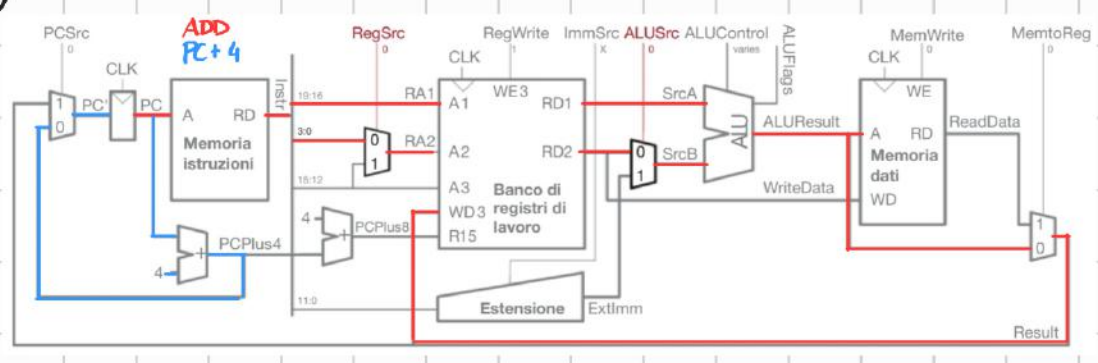
- STR : anche STR legge un indirizzo base dalla porta 1 del register file ed estende con zeri l'immediato. L'ALU somma l'immediato esteso all'indirizzo base per trovare l'indirizzo di memoria (come per la LDR). La STR legge anche un secondo registro dal register file e lo scrive nella memoria dati. Il registro è specificato nel campo Rd, cioè i bit $Instr_{15:12}$, collegati alla porta A2 del register file. Il valore del registro viene emesso sulla porta RD2 collegata alla porte di scrittura WD della memoria dati tramite il bus Write Data.
L'abilitazione alla scrittura della memoria dati, WE, è controllato dal segnal MemWrite: per STR, MemWrite = 1 per poter scrivere il dato, ALUContrd = 00 per fare la somma tra indirizzo base e spiazzamento e RegWrite = 0 perché non si deve scrivere nulla sul register file.



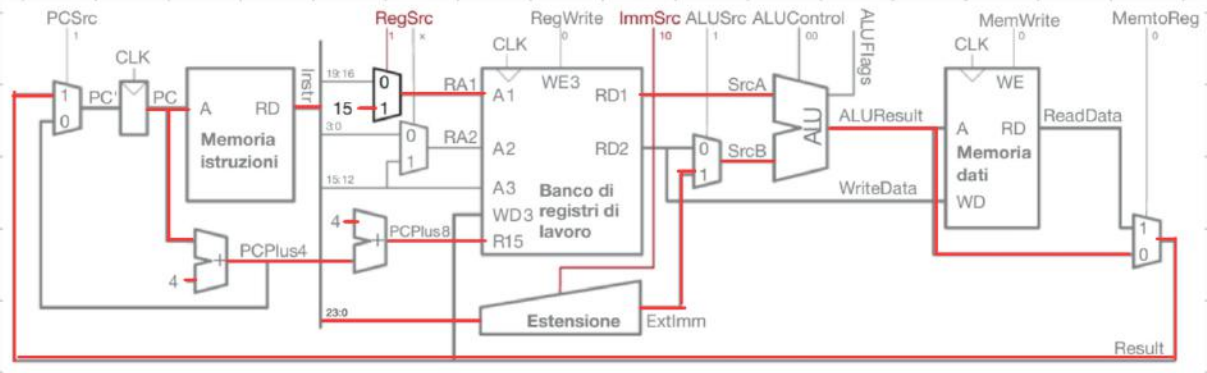
- **Istruzioni di elaborazione dati con indirizzamento immediato**: passiamo ora alla gestione delle istruzioni di elaborazione dati: ADD, SUB, AND e ORR con indirizzamento immediato.
Tutte le istruzioni leggono un registro sorgente dal register file e un immediato dai bit meno significativi dell'istruzione, eseguono un'operazione dell'ALU sui valori ricavati e scrivono il risultato in un registro.



- **Istruzioni di elaborazione dati con indirizzamento a registro**: tali istruzioni ricavano il secondo registro sorgente da Rm, indicato dai bit $Instr_{3:0}$, invece che dall'immediato. Servono quindi ulteriori multiplexer agli ingressi del register file e dell'ALU per selezionare questo secondo registro.
RA2 viene prelevato da Rd ($Instr_{15:12}$) per l'istruzione STR e dal campo Rm ($Instr_{3:0}$) per le istruzioni di elaborazione dati con indirizzamento a registro in base al valore del segnale di controllo RegSrc. Analogamente ALUSrc seleziona come secondo operando dell'ALU ExtImm per le istruzioni con indirizzamento immediato ed il register file per le istruzioni con indirizzamento a registro.



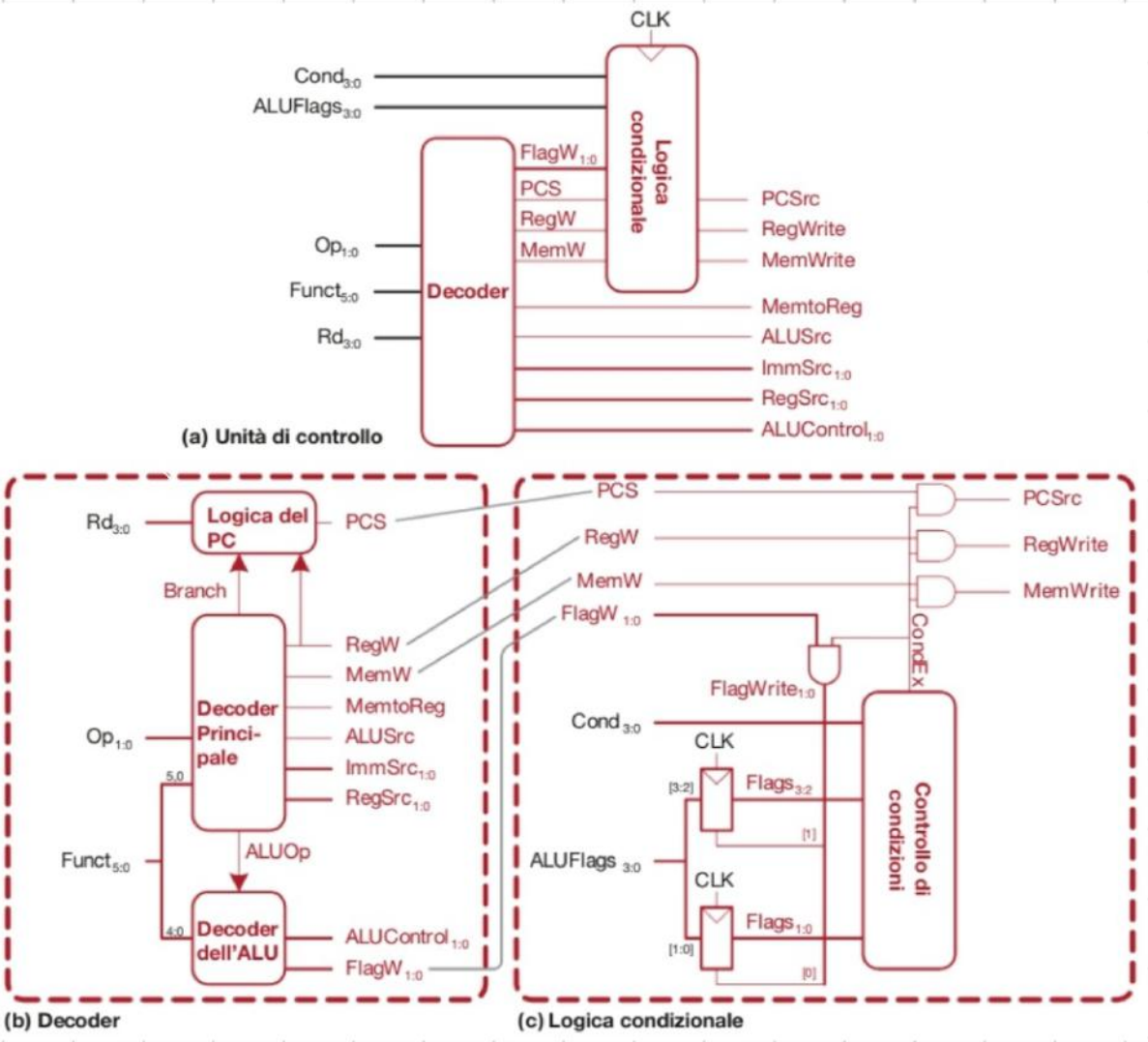
- Istruzione di salto (B): tale istruzione somma un immediato a 24 bit a PC+8 e scrive il risultato nel PC. L'immediato deve essere moltiplicato per 4 ed esteso con segno. PC+8 é letto dalla prima porta del register file quindi serve un multiplexer per selezionare R15 come ingresso RA1: questo multiplexer é pilotato da un altro bit di RegSrc che seleziona $Instr_{19:16}$ per le altre istruzioni e 15 per l'istruzione di salto (B).
MemtoReg = 0 e PCSrc = 1 per selezionare il nuovo valore di PC da ALUResult.



Unitá di controllo

L'unitá di controllo genera i segnali di controllo sulla base dei campi cond, op e funct dell'istruzione, come pure delle flag in uscita dall'ALU e del fatto che il registro destinazione sia il PC.
É divisa in due parti principali: il Decoder, che genera i segnali di controllo in base a Instr e la Logica condizionale che mantiene le flag di stato e abilitá gli aggiornamenti dello stato architetturale.
Il Decoder é costituito da un Decoder Principale che genera la maggior parte dei segnali di controllo, da un Decoder dell'ALU che usa il campo funct per determinare il tipo di operazione aritmetica e da una Logica del

PC per decidere se il PC deve essere modificato per una istruzione di salto o scrittura in R15.
Il Decoder principale determina il tipo di istruzione, genera opportunamente i segnali di controllo per il percorso dati e gli invia MemtoReg, ALUSrc, $ImmSrc_{1:0}$ e $RegSrc_{1:0}$. MemW e RegW devono invece passare dalla Logica prima di diventare i segnali di controllo MemWrite e RegWrite del percorso dati.
Il Decoder Principale genera anche i segnali Branch e ALUOp usati all'interno dell'unità di controllo per indicare se si tratta di un'istruzione di salto o di elaborazione dati.
La Logica del PC controlla se l'istruzione è una scrittura di R15 o di salto, che implicano aggiornamento del PC.
La logica condizionale determina se l'istruzione deve essere eseguita in base al campo cond dell'istruzione e ai valori delle flag.



Processore multi cycle

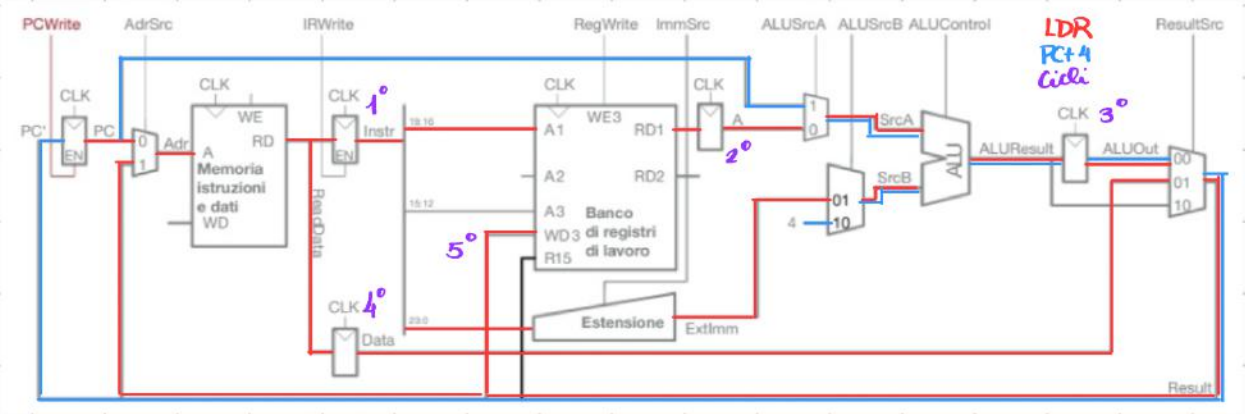
Un processore single cycle ha tre elementi di debolezza. In primo luogo richiede memorie separate per istruzioni e dati, richiede un ciclo di clock abbastanza lungo da consentire l'esecuzione dell'istruzione più lenta e infine, richiede tre circuiti sommatori.
Il processore multi cycle si propone di eliminare queste debolezze dividendo l'istruzione in una sequenza di passi più brevi.
L'istruzione viene letta da memoria in un passo e i dati possono essere letti o scritti in passi successivi, quindi è possibile usare una sola memoria per contenere entrambi.
Istruzioni diverse usano un numero diverso di passi e il processore richiede un solo circuito sommatore.

Percorso dati

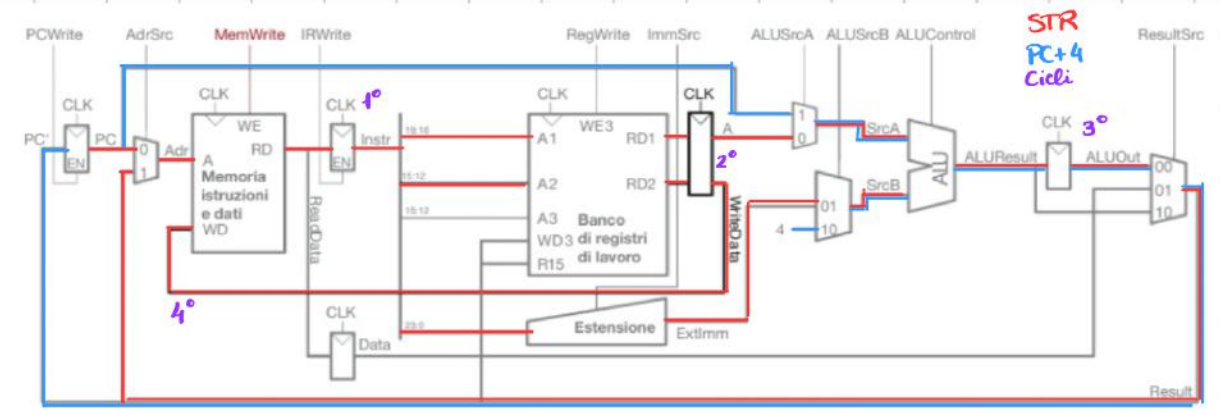
La soluzione, per un processore multi cycle, è molto più realistica ed è fattibile perché si può leggere l'istruzione in un ciclo, quindi leggere o scrivere un dato in un ciclo diverso. PC e register file sono gli stessi, infatti il PC contiene l'indirizzo della istruzione da eseguire ed il primo passo è proprio la lettura (fetch) di tale istruzione dalla memoria.
Il PC viene semplicemente connesso all'ingresso di indirizzo della memoria, l'istruzione viene letta e memorizzata in un registro non architetturale, chiamato "registro istruzioni" (IR), in modo da renderla disponibile nei passi successivi.
IR riceve un segnale di abilitazione IRWrite, che viene attivato quando IR deve essere caricato con la nuova istruzione.

- LDR: dopo la fase di fetch dell'istruzione, il passo successivo è la lettura del registro sorgente contenente l'indirizzo base. Tale registro è specificato nel campo R_n, cioè i bit $Instr_{19:16}$, collegati all'ingresso di indirizzo A1 del register file che emette il contenuto del registro indirizzato sull'uscita di dato RD1 e il valore viene memorizzato in un altro registro non architetturale A.
La LDR richiede anche uno spiazzamento a 12 bit, che si trova nel campo immediato $Instr_{11:0}$ e che deve essere esteso con zeri a 32 bit. L'immediato esteso è denominato ExtImm e dovrebbe essere memorizzato in un registro non architetturale, ma siccome non varia durante l'esecuzione non serve dedicargli un registro.
Si usa una ALU per fare la somma dell'indirizzo base e lo spiazzamento, trovando così l'indirizzo del dato da caricare. ALUControl = 00 per eseguire la somma e il risultato ALUResult viene memorizzato in un altro registro non architetturale, denominato ALUOut.
Il prossimo passo è il caricamento del dato, quindi si aggiunge un multiplexer prima della memoria per selezionare l'indirizzo Adr, dal PC, oppure da ALUOut in base al segnale AdrSrc. Il dato letto dalla memoria viene poi memorizzato in un altro registro non architetturale Data.
Infine il dato deve essere scritto nel register file. Il registro di destinazione è Rd, cioè i bit $Instr_{15:12}$. Il valore da scrivere proviene dal registro Data ma, invece di collegare direttamente Data a WD3, conviene aggiungere un multiplexer sul bus Result per poter scegliere ALUOut o Data prima di inoltrare Result a WD3 grazie al segnale ResultSrc.
Nel frattempo il processore deve aggiornare il PC. Nel

processore multi cycle un' ALU che non è utilizzata nella fase di fetch. Per fare questo bisogna inserire dei multiplexer per selezionare PC e la costante 4. Un multiplexer controllato da ALUSrcA seleziona PC o A come ingresso SrcA e un altro multiplexer seleziona 4 o ExtImm come ingresso SrcB. Per aggiornare il PC l'ALU somma SrcA e SrcB e il risultato deve essere scritto nel PC. Il multiplexer controllato da ResultSrc deve poter selezionare questa somma da ALUResult invece che da ALUOut, serve quindi un terzo ingresso al multiplexer. Il segnale di controllo PCWrite abilita la scrittura nel PC nei cicli opportuni.

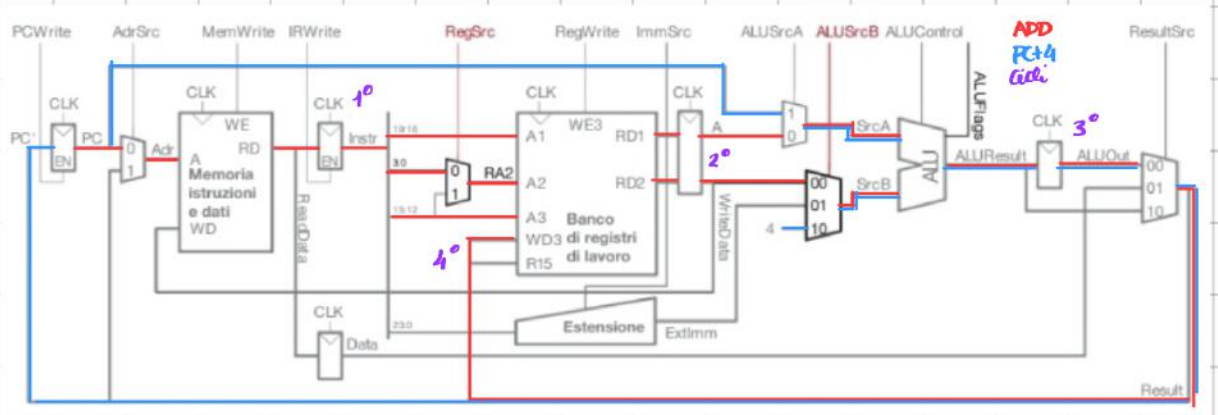


- STR: come LDR, legge un indirizzo base dalla porta 1 del register file ed estende un immediato. L'ALU somma l'indirizzo base all'immediato per trovare l'indirizzo di memoria.
L'unica nuova funzione di STR è la necessità di leggere un secondo registro e scrivere in memoria il valore letto. Tale registro è indicato nel campo Rd, cioè $Instr_{15:12}$ che viene collegato alle seconda porta del register file. Quando il registro viene letto, il valore viene memorizzato in un altro registro non architetturale WriteData. Al passo successivo, tale valore viene inviato alla porta di scrittura WD della memoria, la quale riceve il segnale di controllo MemWrite che attiva l'operazione di scrittura in memoria.

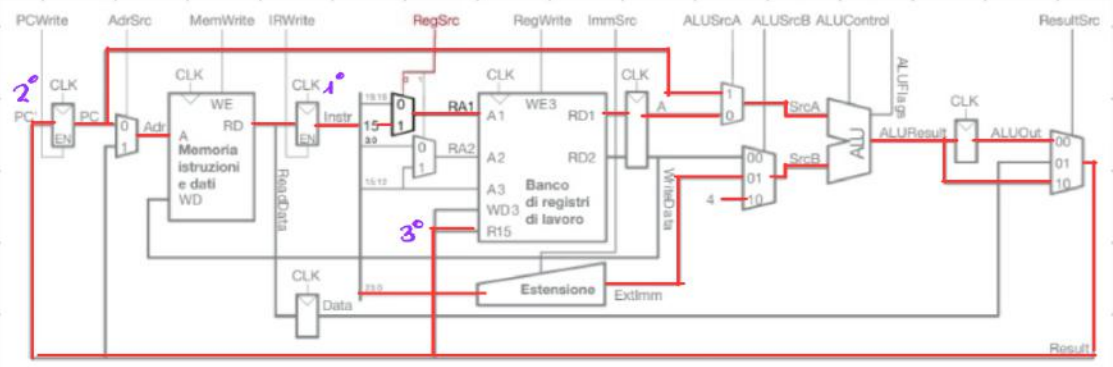


- Istruzioni di elaborazione dati con indirizzamento immediato: leggono il primo operando sorgente da R_n ed estendono il secondo operando sorgente da un immediato a 8 bit. Operano su tali operandi e scrivono il risultato nel register file. L'ALU usa usa il segnale di controllo ALUControl per determinare il tipo di operazione richiesta, le ALUFlags sono inviate alla unità di controllo per aggiornare il registro di stato.

- Istruzioni di elaborazione dati con indirizzamento a registro: queste istruzioni selezionano il secondo operando dal register file. Il registro è specificato nel campo R_m, ovvero $Instr_{3:0}$, quindi serve un multiplexer per selezionare questo campo come RA2 del register file. Serve inoltre estendere il multiplexer SrcB per ricevere il valore letto dal register file.



- Istruzione di salto (B): legge PC+8 e un immediato a 24 bit, li somma e somma il risultato al contenuto di PC.



Unitá di controllo

Come per il processore single cycle, l'unitá di controllo genera i segnali di controllo sulla base dei campi cond, op e funct, come pure delle flag in uscita dall'ALU e che il registro destinazione sia PC.
É divisa in due parti: Decoder e Logica Condizionale.
Il Decoder é a sua volta diviso in due parti, la FSM Principale (progettata come macchina di Moore) che produce la sequenza di segnali di controllo, anche se ImmSrc e RegSrc sono funzione di op invece che dello stato presente, quindi si useráun piccolo Decoder Istruzioni per generare tali segnali. Il Decoder dell'ALU e la Logica del PC sono identici a quelli del processore single cycle.
La Logica Condizionale e quasi identica a quella del processore single cycle: serve solo il segnale aggiuntivo NextPC per forzare una scrittura nel PC quando si calcola PC+4, serve anche ritardare di un ciclo CondEx prima di inviare tale segnale a PCWrite, RegWrite e MemWrite in modo che le flag aggiornate non siano visibili fino al termine dell'istruzione corrente.
Il primo passo di ogni istruzione é il fetch dalla memoria dell'indirizzo presente nel PC, e l'incremento del PC per puntare all'istruzione successiva.

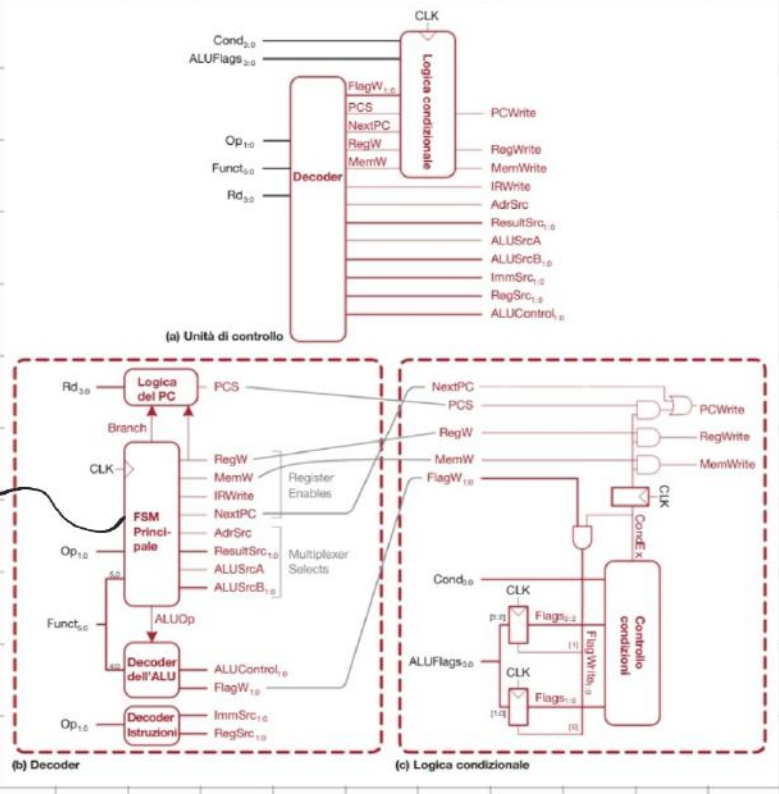
Per leggere dalla memoria AdrSrc = 0 in modo che l'indirizzo sia preso dal PC e IRWrite é attivato per salvare l'istruzione del registro istruzioni IR.
Viene usata l'ALU per calcolare PC+4 in parallelo alla fase di fetch: ALUSrcA = 1 e ALUSrcB = 10, $ALUOp = 0$ quindi l'unitá di controllo genera ALUControl = 00 per fare la somma.
Per aggiornare PC con PC+4, ResultSrc = 10 per selezionare ALUResult e NextPC = 1 per abilitare PCWrite.
Il secondo passo é la lettura del register file e/o dell'immediato e la decodifica delle istruzioni. I registri e l'immediato sono selezionati da RegSrc e ImmSrc, $RegSrc_0 = 1$ per i salti per leggere PC+8 come SrcA e $RegSrc_1 = 1$ per le istruzioni di scrittura in memoria, per leggere come SrcB il valore da memorizzare.
Per semplificare: $RegSrc_1 = (Op == 01)$ $RegSrc_0 = (Op == 10)$
$ImmSrc_{1:0} = Op$
Adesso la FSM procede a uno dei diversi possibili stati, in base op e Funct dell'istruzione esaminati durante la Decode:

- Accesso in memoria (LDR o STR, op = 01): si calcola l'indirizzo sommando all'indirizzo base lo spiazzamento esteso con zeri, quindi ALUSrcA = 0, ALUSrcB = 01 e ALUOp = 0. L'indirizzo ottenuto viene memorizzato nel registro ALUOut.
- LDR ($Funct_0 = 1$): il processore legge dalla memoria un dato e lo scrive nel register file. Per leggere dalla memoria ResultSrc = 00 e AdrSrc = 1, il contenuto della parola indirizzata viene letto e memorizzato nel registro Data durante il passo MemRead. Nel passo di scrittura finale MemWB, il contenuto di Data viene scritto nel register file. ResultSrc = 01 per selezionare Result da Data e viene attivato RegW per

scrivere nel register file.

- Istruzioni di elaborazione dati (op = 00): il processore calcola il risultato usando l'ALU e scrivendolo nel register file. Il primo operando sorgente è sempre un registro (ALUSrcA = 00) e ALUOp = 1 in modo che il Decoder dell'ALU selezioni il valore appropriato di ALUControl. Il secondo operando sorgente proviene dal register file per istruzioni con indirizzamento a registro e da ExtImm per istruzioni con indirizzamento a immediato. In entrambi i casi, l'istruzione avanza allo stato ALUWB di scrittura del risultato dell'ALU selezionato da ALUOut.
- Istruzione di salto: il processore calcola l'indirizzo di destinazione scrivendolo nel PC, durante il passo Decode PC + 8 era già stato calcolato e portato al register file come RD1. Nello stato Branch, l'unità di controllo use ALUSrcA = 0 per selezionare R15, ALUSrcB = 01 per selezionare ExtImm e ALUOp = 0 per fare la somma. Il multiplexer che produce Result seleziona ALUResult (ResultSrc = 10).

Moore



Processore pipeline

L'adozione di strutture pipeline, è una tecnica molto efficace per aumentare le prestazioni di un sistema digitale.
Si progetta un processore pipeline suddividendo il processore single cycle in cinque stadi di pipline. In questo modo, cinque istruzioni alla volta possono essere in esecuzione, una per ogni stadio.
Dal momento che ogni stadio ha circa un quinto della logica totale, la frequenza di clock dovrebbe risultare circa cinque volte superiore. Quindi la latenza rimane quasi inalterata ma la capacità di lavoro (throughput) svolto dal processore è idealmente cinque volte superiore.
I cinque stadi sono denominati Fetch, Decode, Execute, Memory e WriteBack. Nello stadio Fetch il processore legge l'istruzione dalla memoria, nello stadio Decode legge gli operandi dal register file e decodifica l'istruzione per generare i segnali di controllo, nello stadio Execute esegue i calcoli con l'ALU, nello stadio Memory legge o scrive i dati in memoria, nello stadio WriteBack scrive il risultato nel register file.
La durata di ogni stadio dipende dall'attività più lenta.
In ogni stadio viene indicato il componente principale relativo a tale stadio, ovvero memoria istruzioni (IM), lettura dal register file (RF), ALU, memoria dati (DM) e scrittura finale nel register file (RF).
Ogni riga fa vedere i cicli di clock nei quali l'istruzione si trova nei vari stadi.
Ogni colonna fa vedere cosa stanno facendo i vari stadi della pipeline in ogni ciclo di clock.
Nel processore pipeline si scrive nel register file nella prima parte del ciclo e si legge nella seconda parte del ciclo.

Un grosso problema presente nei sistemi pipeline è la gestione delle dipendenze, che si verificano se i risultati di un'istruzione servono ad un'istruzione successiva quando ancora l'istruzione che li deve produrre non è terminata.

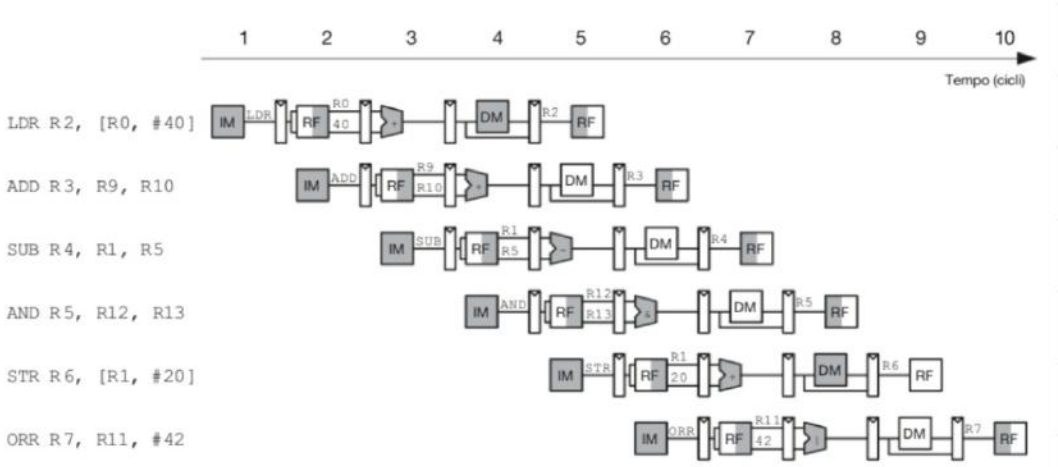
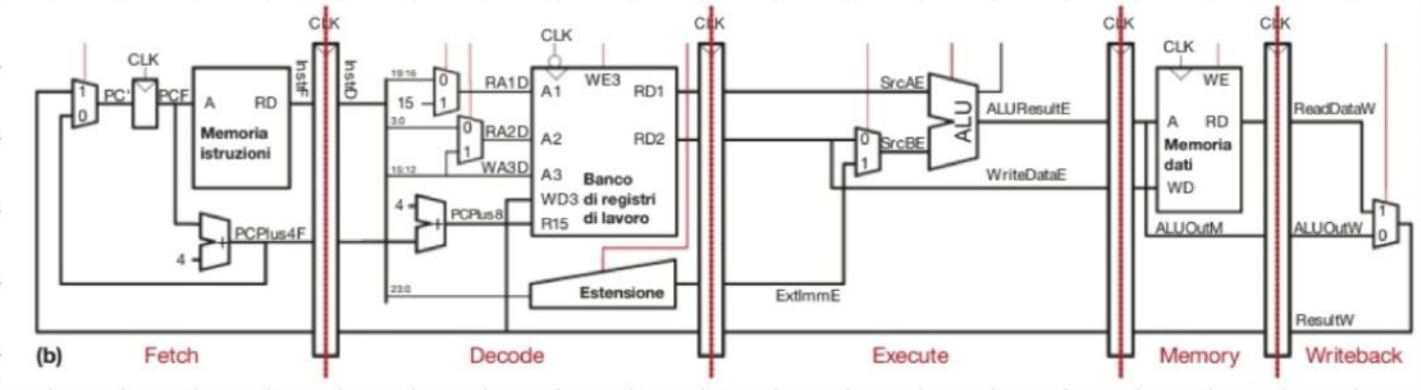


Figura 7.43 Rappresentazione schematica del funzionamento della pipeline.

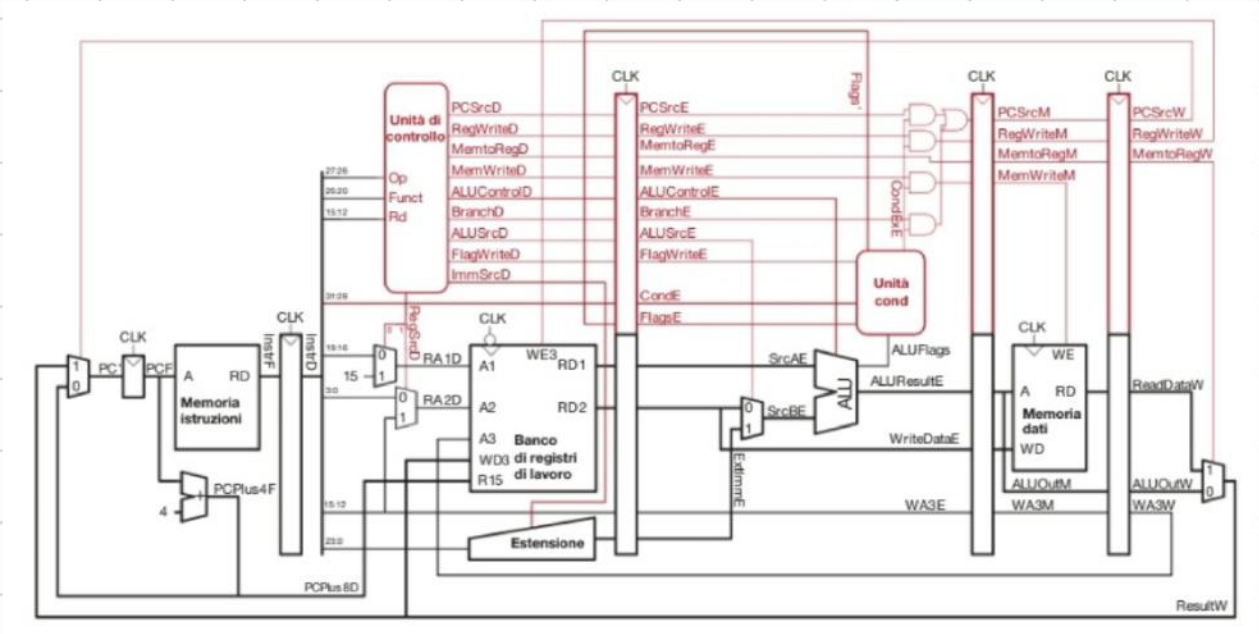
Percorso dati

Il percorso dati del processore pipeline è ottenuto dividendo il percorso dati del processore single cycle in cinque stadi, separati dai registri di pipeline.
Ai segnali è stato aggiunto un suffisso (F, D, E, M o W) per indicare in quale stadio tali segnali sono contenuti.
Uno degli aspetti critici nelle pipeline è il fatto che tutti i segnali, associati a una particolare istruzione, devono procedere all'unisono attraverso la pipeline



Unità di controllo

Il processore pipeline usa gli stessi segnali di controllo del processore single cycle, quindi ha la stessa unità di controllo, che prende in considerazione i campi op e funct dell'istruzione nello stadio Decode per generare i segnali di controllo. L'unità di controllo esamina anche il campo Rd per gestire le scritture nel registro R15(PC).



Dipendenze

Si verifica dipendenza quando un'istruzione dipende dai risultati di un'istruzione che la precede e che non è ancora conclusa. Quando un'istruzione scrive in un registro che deve essere letto da istruzioni successive si crea una dipendenza RAW (Read after Write). Una soluzione a livello software sarebbe quella di inserire un certo numero di istruzioni NOP (No OPeration) in modo che l'istruzione che ha dipendenza non legga il registro finché non sia disponibile nel register file.

Le dipendenze sono classificate in dipendenze di dato e dipendenze di controllo: una dipendenza di dato si verifica quando un'istruzione vuole leggere un registro che non è ancora stato aggiornato da una

istruzione precedente, una dipendenza di controllo si verifica quando la decisione di quale istruzione debba essere prelevata non è ancora stata presa al momento del fetch.
Esistono diversi modi per gestire le dipendenze:

- Gestione delle dipendenze di dato tramite inoltro: alcune dipendenze possono essere risolte mediante la tecnica dell'inoltro o "forwarding" di un risultato, disponibile negli stadi Memory o Writeback, all'istruzione che ha dipendenza e che si trova nello stadio Execute. Questo richiede l'aggiunta di un multiplexer davanti all'ALU per selezionare l'operando dal register file o dagli stadi Memory o Writeback.
  Il forwarding è necessario quando un'istruzione nello stadio Execute ha un registro sorgente coincidente con il registro destinazione delle istruzioni nello stadio Memory o Writeback.
  L'unità di gestione delle dipendenze riceve quattro segnali di uguaglianza dal percorso dati che indicano se un registro sorgente nello stadio Execute è uguale al registro destinazione negli stadi Memory o Writeback:

  - Match_1E_M = (RA1E == WA3M)
  - Match_1E_W = (RA1E == WA3W)
  - Match_2E_M = (RA2E == WA3M)
  - Match_2E_W = (RA2E == WA3W)

  L'unità di gestione delle dipendenze riceve anche i segnali RegWrite dagli stadi Memory o Writeback per sapere se il registro destinazione verrà davvero modificato. Genera inoltre i segnali per i multiplexer di inoltro per selezionare gli operandi dal register file o dai risultati negli stadi Memory o Writeback (ALUOutM e ResultW).

Se sia Memory che Writeback contengono registri destinazione uguali allo stesso registro sorgente di Execute, si da precedenza allo stadio Memory che contiene il valore più recente.

- Gestione delle dipendenze tramite stalli: il forwarding va bene solo per dipendenze di tipo RAW. Sfortunatamente la LDR finisce di leggere il dato alla fine dello stadio Memory, per cui il risultato non può essere inoltrato allo stadio Execute della istruzione successiva (LDR impiega 2 cicli).
La soluzione alternativa al forwarding è quindi quella di forzare in stallo la pipeline, sospendendo le operazioni fino all'arrivo del dato. Questo mancato utilizzo di uno stadio che si propaga lungo la pipeline è denominato bolla (bubble) e si comporta come l'istruzione NOP. La bolla viene generata azzerando i segnali di controllo dello stadio Execute durante lo stallo dello stadio Decode, in modo tale che non vengano eseguite operazioni, dalla bolla, che modifichino lo stato architetturale del processore.
Lo stallo di uno stadio viene effettuato disabilitando i registri pipeline in modo che lo stato corrente non sia modificato.
Ogni stadio che viene portato in stallo deve subire la stessa sorte tutti gli stadi precedenti, in modo che non si perde nessun'istruzione.
Il registro pipeline va svuotato subito dopo lo stallo in modo che informazioni fasulle non si propaghino nelle pipeline e bisogna tenere in considerazione che lo stallo degrada le prestazioni, quindi va usato solo quando strettamente necessario.
Gli stalli sono realizzati aggiungendo ingressi di abilitazione di pipeline degli stadi Fetch e Decode e un ingresso di reset sincrono (CLR) al registro pipeline dello stadio Execute. StallD e StallF mandano in stallo rispettivamente lo stadio Decode e Fetch e FlushE svuota il contenuto del registro pipeline dello stadio Execute.

- Gestione delle dipendenze di controllo: l'istruzione di salto B, come le scritture nel registro R15, presenta una dipendenza di controllo. Un modo per risolverela sarebbe usare lo stallo ma, dato che la decisione viene presa nello stadio Writeback, la pipeline dovrebbe essere messo in stallo per quattro cicli a ogni istruzione di salto e questo provocherebbe un notevole degrado delle prestazioni.
L'alternativa é prevedere se il salto dovrá essere fatto o meno e cominciare l'esecuzione delle istruzioni sulla base di tale previsione. Se il salto va fatto, le istruzioni dopo di esso devono essere scartate, cioé la pipeline va svuotata cancellando i registri pipeline per tali istruzioni. Queste istruzioni da eliminare sono la cosiddetta "penalizzazione da salto mal previsto".
Infine, serve generare i segnali di controllo per lo stallo e svuotamento per gestire salti e scritture nel PC. Quando si fa un salto si devono svuotare i registri pipeline delle due istruzioni successive negli stadi Fetch e Decode, quando nella pipeline é presente una scrittura nel PC bisogna mettere in stallo la pipeline fino al termine della scrittura (mettendo in stallo lo stadio di Fetch. PCWrPending é attivo quando é in corso una scrittura nel PC, in tal caso lo stadio Fetch va in stallo e Decode va svuotato. Quando la scrittura nel PC raggiunge lo stadio Writeback viene disattivato StallF ma FlushD rimane attivo per evitare che l'istruzione presente nello stadio Fetch possa avanzare.
Le istruzioni di salto non sono molto frequenti e anche una penalizzazione per salto mal previsto di due cicli ha un impatto non trascurabile sulle prestazioni.

Pipeline lunghe

Il modo più semplice per velocizzare il clock è quello di dividere la pipeline in più stadi: ogni stadio contiene meno logica quindi può essere più veloce. Il massimo numero di stadi di pipeline è limitato dalle dipendenze, dal sovraccarico di sequenziamento e dal costo. Pipeline lunghe introducono più dipendenze.
I registri pipeline tra uno stadio tra l'altro comportano un sovraccarico di sequenziamento per il ritardo del clock all'uscita e il ritardo di setup, tale sovraccarico fa sì che aggiungere stadi di pipeline porti vantaggi sempre minori.

Micro operazioni

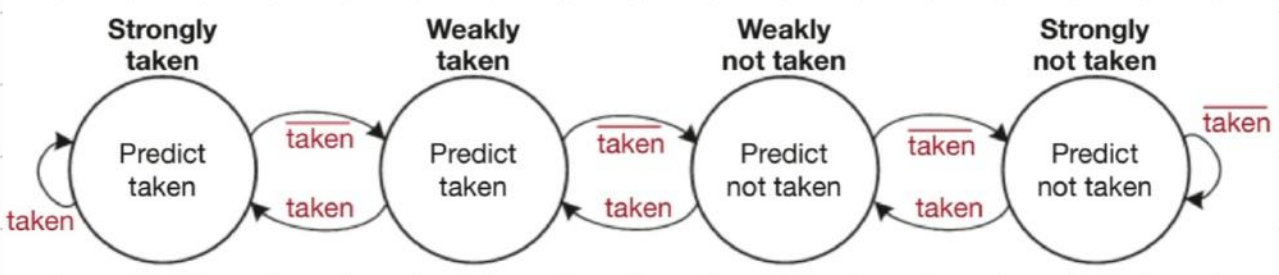
Le architetture a set di istruzioni ridotto (RISC) presentano solo istruzioni semplici eseguibili in un solo ciclo di clock su un percorso dati semplice e veloce, utilizzando un banco di registri a tre porte una sola ALU e un solo accesso in memoria dati.
Le architetture a set di istruzioni complesso (CISC) includono istruzioni che necessitano di più registri, più addizioni e più di un accesso in memoria per istruzione.
Gli architetti dei calcolatori rendono veloci i casi frequenti definendo un insieme di semplici micro operazioni eseguibili su percorsi dati semplici. Le istruzioni vere e proprie vengono scomposte in una o più micro operazioni.

Previsione dei salti

La causa principale dell'aumento del CPI è la penalizzazione per salti mal previsti e con l'allungarsi delle pipeline la decisione se

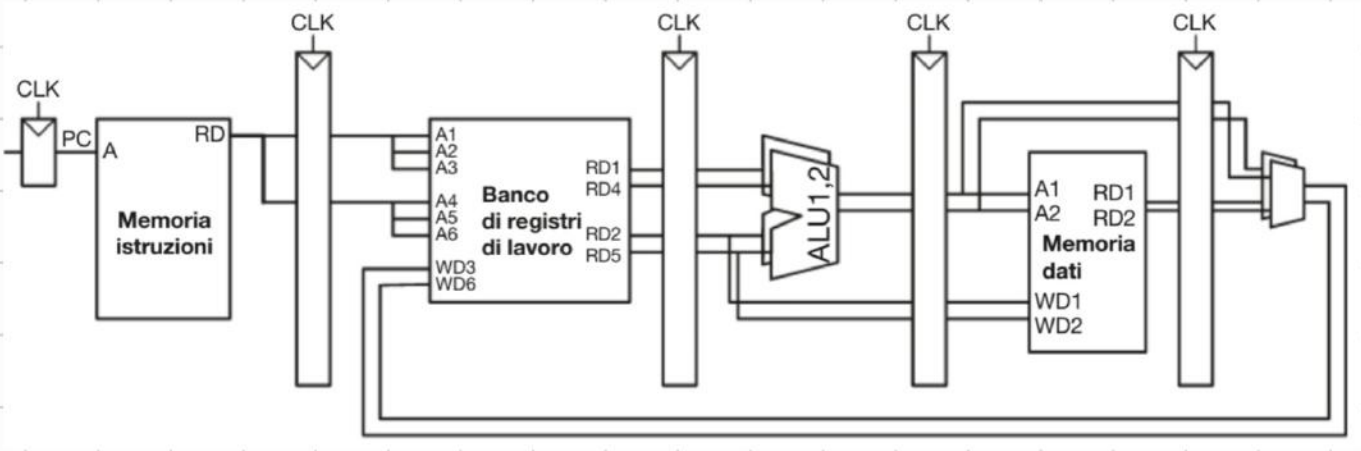
saltare o meno viene presa più avanti, con il risultato di aumentare tale penalizzazione.
Per affrontare questo problema, la maggior parte dei processori pipeline adotta un predittore di salto (branch predictor) per cercare di prevedere se il salto andrà fatto o meno.
Alcuni salti sono posizionati alla fine di un ciclo e saltano all'inizio del ciclo per ripeterlo. Il metodo più semplice per prevedere questo tipo di salti è la previsione statica dei salti, che verifica la direzione del salto e assume che i salti all'indietro debbano essere fatti (non dipende da ciò che è successo nell'esecuzione del programma).
I salti in avanti sono più difficili da prevedere senza conoscere la struttura del programma, quindi molti processori usano la previsione dinamica dei salti che si basa sulla storia del programma per cercare di indovinare se il salto vada fatto o meno.
La tabella, denominata buffer delle distinazioni di salto (o tabella di previsione dei salti) include la destinazione di ciascun salto e la storia del salto (se è stato eseguito o meno in passato).
Un predittore dinamico a un bit ricorda se il salto è stato eseguito o meno l'ultima volta che è stato incontrato, e quando lo incontra di nuovo ripete la scelta precedente. Tale previsione funziona fino all'ultima iterazione, quando il salto va fatto. Sfortunatamente, se si ritorna ad eseguire lo stesso ciclo, il predittore ricorda che l'ultima volta il salto era stato fatto e prevede di doverlo fare alla prima iterazione, quindi il predittore dinamico a un bit sbaglia la prima e l'ultima iterazione.
Un predittore dinamico a due bit risolve parzialmente il problema memorizzando quattro stati relativi ad ogni salto, strongly taken, weakly taken, strongly not taken e weakly not taken.
Quando il ciclo viene ripetuto il predittore entra nello stato strongly not taken, la previsione è corretta fino all'ultima

iterazione quando il salto va eseguito, cosa che sposta il predittore nello stato weakly not taken. Se si rientra una seconda volta nel ciclo il predittore prevede correttamente di non dovere saltare e si sposta nello stato strongly not taken, quindi sbaglia solo l'ultima iterazione.



Processori superscalari

Un processore superscalare contiene più copie dell'hardware del percorso dati, per poter eseguire più istruzioni contemporaneamente. Per esempio, in un processore superscalare a due vie il percorso dati esegue il fetch di due istruzioni alla volta dalla memoria istruzioni, ha un register file a sei porte per leggere quattro operandi sorgente e scrivere due risultati per ciclo e due ALU e una memoria dati a due porte per eseguire contemporaneamente due istruzioni. L'esecuzione simultanea di più istruzioni crea problemi per le dipendenze.



Processore out-of-order

Per far fronte al problema delle dipendenze, un processore out-of-order esamina un certo numero di prossime istruzioni per iniziare a eseguire istruzioni indipendenti il più rapidamente possibile: le istruzioni possono essere attivate in ordine diverso da quello scritto a patto che le dipendenze vengano rispettate e che vengano prodotti i risultati previsti.
Oltre alle dipendenze di tipo RAW (Read after write) esistono anche le dipendenze di tipo WAR (Write after read) anche note come antidipendenze e le dipendenze di tipo WAW (Write after write) note come dipendenze di uscita.
I processori out-of-order usano una tabella per tenere traccia delle istruzioni che attendono di essere attivate, la tabella contiene informazioni riguardanti le dipendenze e la sua dimensione determina il numero di istruzioni che possono essere considerate per decidere quali attivare. A ogni ciclo il processore esamina la tabella e attiva il maggior numero di istruzioni in base alle dipendenze e al numero di unità di esecuzione disponibili.
Il parallelismo a livello di istruzioni (ILP) è il numero di istruzioni che possono essere eseguite simultaneamente per un certo programma e una certa microarchitettura.

Ridenominazione dei registri

I processori out-of-order usano la ridenominazione dei registri (register renaming) per gestire le dipendenze di tipo WAR e WAW. Tale tecnica aggiunge alcuni registri non architetturali da usare per la ridenominazione.

Multithreading

Un programma in esecuzione su un calcolatore è chiamato processo, i calcolatori possono eseguire molti processi in parallelo.
Ogni processo a sua volta è composto da uno o più thread che possono essere anche loro eseguiti in parallelo.
Il numero di thread in cui un processo può essere suddiviso prende il nome di Thread level parallelism (TLP).
In un processore convenzionale, i thread danno solo l'illusione di essere eseguiti in parallelo: in realtà vengono eseguiti a turno dal processore, sotto il controllo del sistema operativo. Quando il turno di un thread finisce, il sistema operativo salva il suo stato architetturale, carica lo stato architetturale del prossimo thread ed inizia ad eseguirlo. Questa procedura è definita cambio di contesto (context switching).

Multiprocessori

Un multiprocessore consiste dunque di più processori e di una struttura di comunicazione tra essi. Le tre tipologie più comuni sono i multiprocessori simmetrici, i multiprocessori eterogenei e i cluster.

Multiprocessori simmetrici

Consistono di due o più processori identici che condividono la stessa memoria principale. Possono essere realizzati in chip diversi o diversi core nello stesso chip. Vengono usati sia per eseguire più thread in parallelo o per eseguire più rapidamente un singolo thread.
Quest'ultima cosa è più complicata in quanto bisognerebbe dividere il thread in molti thread e farli eseguire dai diversi processori,

ciò diventa problematico quando i processori devono comunicare tra loro.
I processori simmetrici sono relativamente semplici da progettare perché una volta disegnato un processore basta replicarlo un certo numero di volte.

Multiprocessori eterogenei

Dal momento che i processori per uso generale sono progettati per offrire buone prestazioni, non sono la soluzione più efficiente dal punto di vista energetico per svolgere un'operazione specifica.
I multiprocessori eterogenei affrontano questo problema incorporando tipi diversi di core e/o hardware specializzato in un singolo sistema. Ogni applicazione può quindi usare le risorse del sistema che forniscono le migliori prestazioni, o il miglior rapporto tra potenza e prestazioni.
Una strategia eterogenea resa popolare da ARM è big.LITTLE, che si basa su sistemi contenenti sia core a basso consumo sia core ad alte prestazioni. I core "LITTLE" sono in grado di di attivare una o due istruzioni alla volta, in ordine, con consumi limitati di potenza. I core "big" sono processori out-of-order superscalari più complessi in grado di offrire elevate prestazioni.
Un'altra strategia eterogenea sono gli acceleratori come ad esempio i processori di segnali digitali (DSP).
I processori eterogenei aggiungono però complessità sia nel progetto dei diversi elementi eterogenei sia nello sforzo di programmazione per decidere quando e come usare i diversi tipi di risorse.

Cluster

Nei multiprocessori a cluster ogni processore ha la sua struttura di memoria locale. Per esempio un gruppo di PC collegati in rete che eseguono insieme del software per risolvere un problema di grandi dimensioni.