Quarto capitolo

Il livello di microarchitettura

> L’obiettivo del livello di microarchitettura è di **implementare il livello ISA**

Esempio di microarchitettura

Introduciamo i principi generali su cui si basa la progettazione della microarchitettura: prenderemo come esempio la IJVM (Integer Java Virtual Machine - la JVM che opera solo sugli interi).

La IJVM contiene un **microprogramma:** si compone di una **sequenza di microistruzioni** che utilizzano variabili che costituiscono **lo stato del programma:** ogni “funzione” (istruzione) cambia il valore della variabile, quindi varia lo stato della macchina. Un esempio di variabile è il **Program Counter (PC)**, la quale contiene il riferimento della prossima microistruzione da eseguire.

Ogni microprogramma è composto da un **opcode (codice operativo),** il quale identifica la tipologia d’istruzione, e di un’ulteriore campo che specifica l’operando su cui si applicherà l’istruzione.

Il modello di esecuzione applicato è il cossidetto **fetch-decode-execute:**

- **fetch →** caricamento in memoria dell’istruzione;
- **decode →** riconoscimento del tipo di istruzione da eseguire;
- **execute →** esecuzione dell’istruzione.

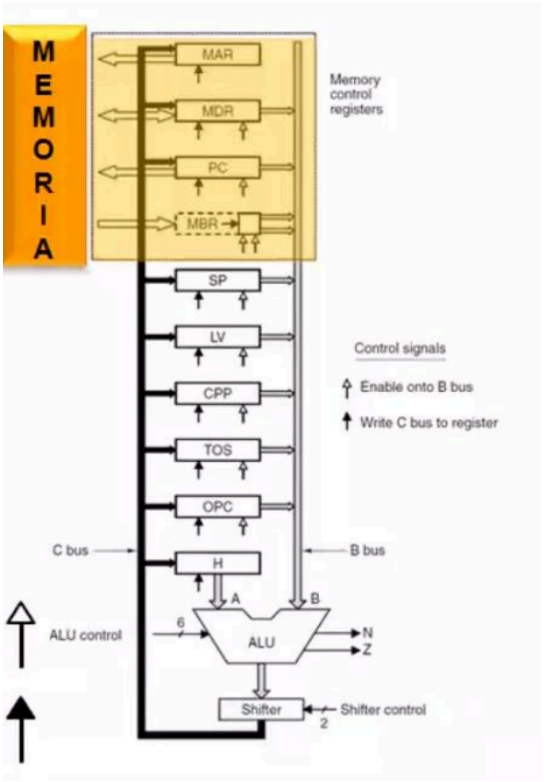
Il percorso dati

> Il **data path (o percorso dati)** è quella parte della CPU comporta da una serie di registri, la ALU, lo shifter, gli input e gli output. Quest’ultimi si trovano all’interno dei registri.

Principalmente si compone di due bus: il bus B, dal quale passa uno dei due input della ALU, e il bus C, dal quale passa l'output dello shifter e portato in uno dei registri rappresentati in figura. Il bus A preleva da un unico registro, chiamato registro di mantenimento (**holding**). Tale registro può contenere l'output finale della ALU (volendo anche uno dei dati presenti nei registri, quindi della ALU non verrà eseguita alcuna operazione).

Esistono due segnali di controllo:

- La freccia bianca indica che il segnale abilita la scrittura del registro sul bus B;
- La freccia nera indica che il segnale scrive il contenuto del bus C sul registro.



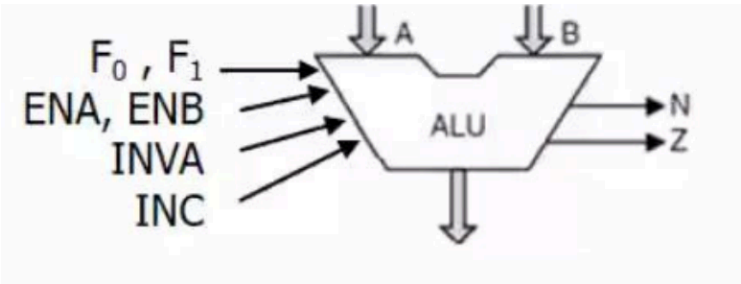
Possiamo notare che solo determinati registri memorizzano il contenuto presente nella memoria:

- **MAR (Memory Address Register);**
- **MDR (Memory Data Register);**
- **PC (Program Counter);**
- **MBR (Memory Byte Register).**

Dobbiamo specificare che i 4 registri fanno riferimento a parti differenti della memoria:

MAR e MDR contengono indirizzi espressi in parole e sono usati per leggere/scrivere parole di dati del livello ISA;

PC e MBR contengono indirizzi espressi in byte e sono usati per leggere il programma eseguibile del livello ISA. (Questi registri non possono scrivere!)



La funzione della ALU è determinata da 6 linee di controllo:

- F_0 e F_1 → Determinano l'operazione della ALU;
- ENA e ENB → Abilitano individualmente i due input;

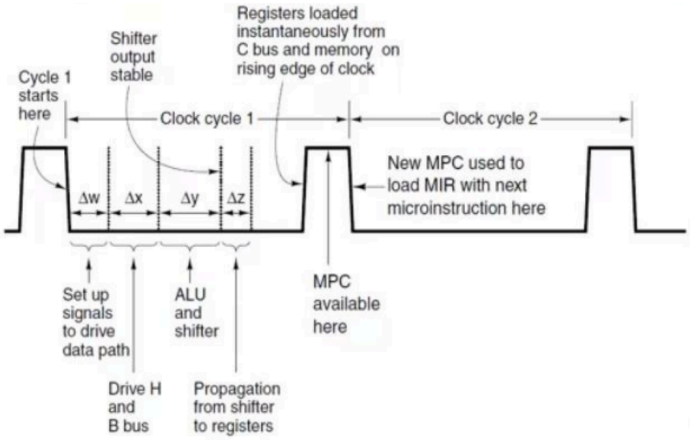
- INVA → Inverte l'input di sinistra;
- INC → Forza la presenza di un bit di riporto meno significativo.

F_0	F_1	ENA	ENB	INVA	INC	Function
0	1	1	0	0	0	A
0	1	0	1	0	0	B
0	1	1	0	1	0	$\bar{A}$
1	0	1	1	0	0	$\bar{B}$
1	1	1	1	0	0	A + B
1	1	1	1	0	1	A + B + 1
1	1	1	0	0	1	A + 1
1	1	0	1	0	1	B + 1
1	1	1	1	1	1	B – A
1	1	0	1	1	0	B – 1
1	1	1	0	1	1	–A
0	0	1	1	0	0	A AND B
0	1	1	1	0	0	A OR B
0	1	0	0	0	0	0
1	1	0	0	0	1	1
1	1	0	0	1	0	–1

L'output della ALU diventerà successivamente input dello shifter: il suo compito è di far scorrere i bit/byte del risultato verso destra o sinistra. In base alla tipologia di scorrimento, vengono abilitate uno dei due linee di controllo:

- SLL8 (Shift Left Logical - scorrimento logico a sinistra) → Trasla il valore a sx di un byte → 8 bit meno significativi a 0;
- SRA1 (Shift Right Arithmetic - scorrimento aritmetico a destra) → Trasla il valore a destra di un bit.

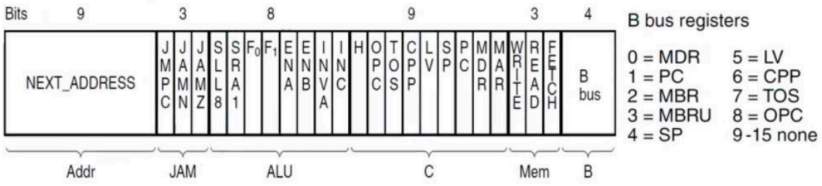
Temporizzazione del percorso dati



In tempo:

- **Δw →** vengono impostati i segnali che guideranno il percorso dati;
- **Δx →** viene selezionato il registro e il suo contenuto viene portato sul bus B;
- **Δy →** ALU e shifter operano e l'output finale diventa stabile;
- **Δz →** Il risultato si trova lungo il bus C;
- **Fronte di salita →** viene caricato il risultato nei registri e il registro che stava alimentando il bus B smette di farlo.

Il formato delle microistruzioni



Una microistruzione è composto da 36 segnali. Volendo, possiamo suddividerli in due gruppi di segnali, in base alla loro mansione principale:

- Il primo gruppo, composto dai **gruppi funzionali ALU, C, Mem e B,** specificano le operazioni da eseguire durante un ciclo di percorso dati;
- Il secondo gruppo, composto dai **gruppi funzionali Addr e JAM,** determinano cosa deve essere effettuato durante il ciclo successivo.

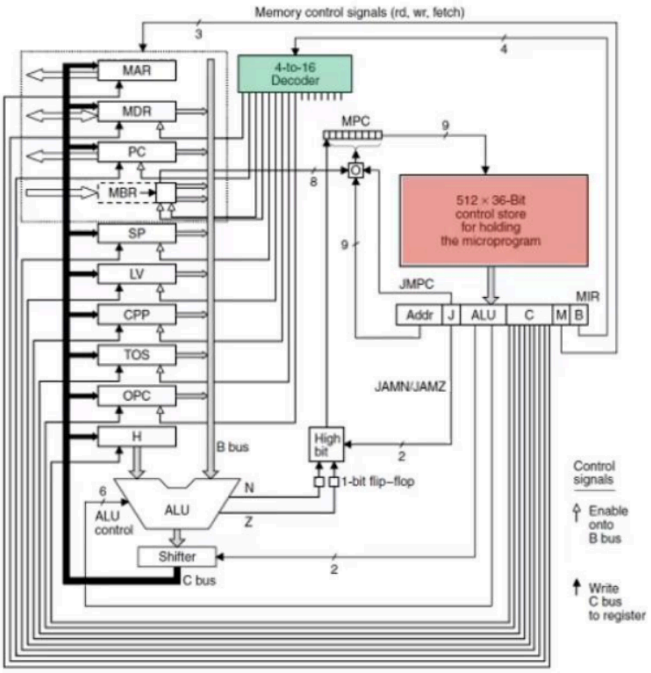
I **gruppi funzionali sono 6, ognuna con una propria funzione:**

- **Addr →**Contiene l'indirizzo di una potenziale successiva microistruzione;
- **JAM →** Determina come viene selezionata la successiva microistruzione;
- **ALU →** Seleziona le funzioni della ALU e dello shifter;
- **C →** Seleziona quali registri sono scritti dal bus C;
- **Mem →** Seleziona la funzione della memoria;
- **B →** Seleziona la sorgente del bus B.

Unità di controllo microprogrammata: Mic-I

La **microarchitettura Mic-I** è composto da due parti principali:

- Il **percorso dat**i, rappresentato a sinistra dell'immagine;
- Una **sezione di controllo.**

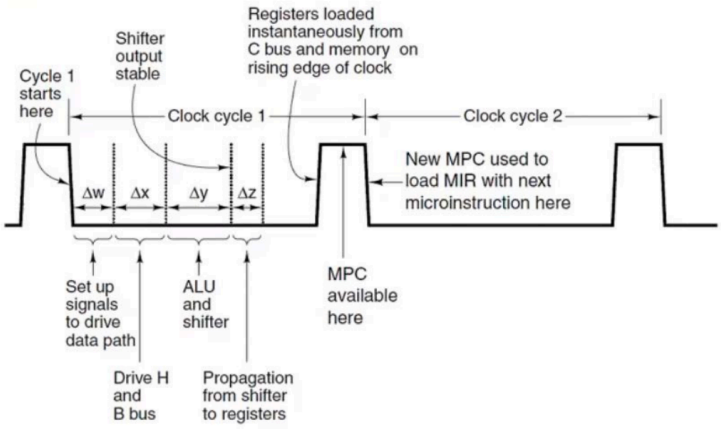


L'elemento principale della **sezione di controllo** è la **memoria di controllo**: possiamo considerarla come una memoria che contiene **l'intero microprogramma.** Quindi, dal punto di vista funzionale, la memoria di controllo è un circuito che memorizza le microistruzioni invece che le istruzioni ISA.

La memoria di controllo si differisce dalla **memoria centrale** per un aspetto cruciale: le istruzioni della memoria centrale sono sempre eseguite nell'ordine determinato dagli indirizzi, caso contrario invece nel caso delle microistruzioni, nel quale l'esecuzione della successiva istruzione corrisponde a quella che segue l'istruzione corrente all'interno della memoria.

Anche la memoria di controllo necessita di un **registro di indirizzo, che è chiamato MPC (MicroProgram Counter),** e di un **registro dei dati, chiamato MIR (MicroInstruction Register)**. L'obiettivo del MIR è di memorizzare la microistruzione in corso di esecuzione. Si può notare dall'immagine che il MIR è composto dai 6 gruppi funzionali descritti in precedenza.

Funzionamento del Mic-I



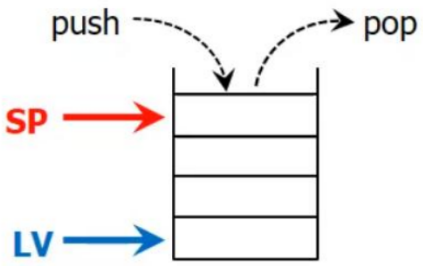
- **Δw** → la parola contenuta nella memoria di controllo e puntata dal MPC viene trasferita nel MIR;
- **Δw + Δx →** vari segnali si propagano nel percorso dati e inizia il processo;
- **Δy →** tutto il circuito si è stabilizzato (anche l'output della ALU, bit N,Z e lo shifter);
- **Δz →** l'output dello shifter raggiunge i registri attraverso il bus C;
- **Fine del ciclo →** i registri vengono caricati;
- Il sottociclo termina dopo il **fronte di salita del clock,** quando i risultati sono salvati, i risultati delle precedenti operazioni di memoria sono disponibili e MPC è stato caricato.

Un microprogramma, come detto in precedenza, oltre a guidare il percorso dati, deve anche determinare la prossima microistruzione: il calcolo dell'indirizzo della microistruzione avviene dopo che MIR è stato caricato ed è stabile. Inizialmente, i bit di NEXT_ADDRESS vengono copiati in MPC (ricordiamo che è il registro che mantiene l'indirizzo della successiva microistruzione da eseguire); mentre si svolge questa mansione, viene ispezionato il campo JAM: nel caso in cui il suo valore è 000, allora non viene eseguita alcuna mansione aggiuntiva; nel caso in cui uno o più dei 3 bit è uguale a 1, allora vengono eseguite alcune azioni.

Esempio di ISA: IJVM

Lo stack

> Lo **stack (o pila)** è una struttura dati utilizzata per memorizzare lo stato di una procedura. Segue la filosofia del LIFO (Last-In-First-Out).



Al suo interno si imposta un registro, chiamato **LV,** il quale punta alla base delle variabili locali per la procedura corrente. Un ulteriore registro, **SP,** punta alla parola che si trova nella locazione più alta. La struttura dati compresa tra LV e SP è chiamata **blocco delle variabili locali della procedura.**

Il modello di memoria IJVM

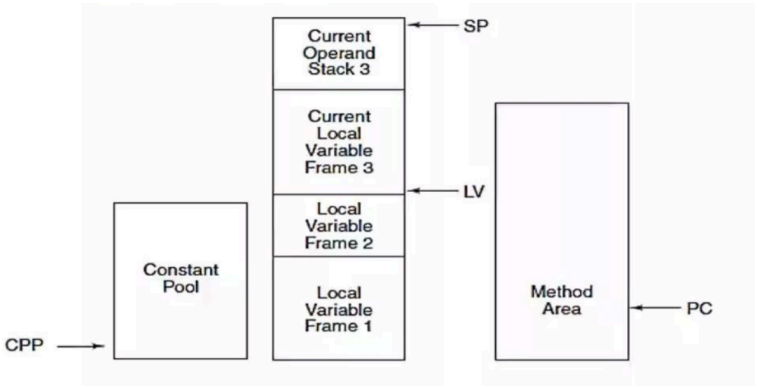
La memoria della IJVM può essere vista come 4 aree differenti:

- **Porzione costante di memoria →** contiene costanti, stringhe, puntatori ed altre aree di memoria cui è possibile far riferimento; I programmi IJVM non possono scriverci;
- **Blocco delle variabili locali →** Per ogni invocazione di un metodo, viene allocata un'area in cui memorizzare le variabili locali durante l'intero ciclo di vita dell'invocazione;
- **Stack degli operandi →** Allocato sopra il blocco delle variabili locali, memorizzano gli operandi durante il calcolo di un'espressione aritmetica. Brevemente:

  Consideriamo di voler calcolare

  $a_1 = a_2 + a_3$

  Poniamo prima a_2, successivamente a_3, in cima allo stack (quindi, eseguiamo un *push* prima con a_2, poi con a_3. Il calcolo effettivo può essere realizzato eseguendo un'istruzione che preleva le due parole dallo stack, le somma e inserisce il risultato nuovamente nello stack. Infine, il risultato che si trova in cima allo stack può essere rimossa e memorizzata nella variabile locale a_1.
- **Area dei metodi →** Regione di memoria in cui risiede il programma. È presente un registro implicito che contiene l'indirizzo della successiva istruzione da prelevare.



Insieme delle istruzioni della IJVM

Hex	Mnemonic	Meaning
0x10	BIPUSH *byte*	Push byte onto stack
0x59	DUP	Copy top word on stack and push onto stack
0xA7	GOTO *offset*	Unconditional branch
0x60	IADD	Pop two words from stack; push their sum
0x7E	IAND	Pop two words from stack; push Boolean AND
0x99	IFEQ *offset*	Pop word from stack and branch if it is zero
0x9B	IFLT *offset*	Pop word from stack and branch if it is less than zero
0x9F	IF_ICMPEQ *offset*	Pop two words from stack; branch if equal
0x84	IINC *varnum const*	Add a constant to a local variable
0x15	ILOAD *varnum*	Push local variable onto stack
0xB6	INVOKEVIRTUAL *disp*	Invoke a method
0x80	IOR	Pop two words from stack; push Boolean OR
0xAC	IRETURN	Return from method with integer value
0x36	ISTORE *varnum*	Pop word from stack and store in local variable
0x64	ISUB	Pop two words from stack; push their difference
0x13	LDC_W *index*	Push constant from constant pool onto stack
0x00	NOP	Do nothing
0x57	POP	Delete word on top of stack
0x5F	SWAP	Swap the two top words on the stack
0xC4	WIDE	Prefix instruction; next instruction has a 16-bit index