Architettura degli Elaboratori
10 - Il Livello di Microarchitettura

Ugo Dal Lago

Dipartimento di Scienze dell'Informazione
Università degli Studi di Bologna

Anno Accademico 2006/2007

Sommario

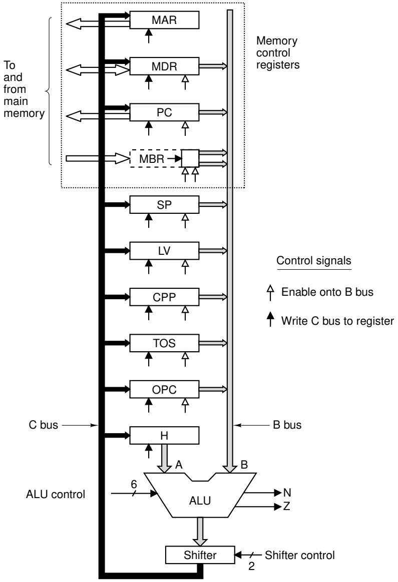
Introduzione al Livello di Microarchitettura

- Come sappiamo, il **livello di mircoarchitettura** si colloca tra il livello logico-digitale, che abbiamo appena finito di trattare e il livello dell'instruction set.
- Procederemo nello studio del livello di microarchitettura nel modo seguente:
  - Introdurremo prima di tutto una microarchitettura di esempio, detta **Mic-1**, che consiste in un percorso dati e in un'unità di controllo (quest'ultima contenente il microcodice vero e proprio).
  - Introdurremo poi un esempio di instruction set, detto **IJVM**.
  - Studieremo poi come scrivere microcodice per Mic-1 che interpreti le istruzioni IJVM.
  - Al passo successivo, cercheremo di migliorare Mic-1 tramite l'impiego di tecniche quali il buffer di prefetch e il pipelining.
  - Accenneremo poi ad alcune tecniche per il miglioramento delle prestazioni.
  - Daremo infine uno sguardo ad alcune microarchitetture concrete.

Il Percorso Dati di Mic-1

- Il percordo dati di Mic-1 comprende prima di tutto una **ALU**, del tutto simile a quella che abbiamo già visto parlando del livello logico-digitale.
  - L'ALU avrà due ulteriori output `N` e `Z` che indicano se il risultato è nullo oppure no.
- Vi sono poi un certo numero di **registri** a 32 bit, tra cui ricordiamo `PC`, `SP` e `MDR`.
- Le uscite dell'ALU contenenti il risultato vanno a finire in input ad un **registro a scorrimento**, comandato da due ingressi di controllo
- Vi saranno infine **tre bus**:
  - Il bus esterno A, che permette di comunicare con la memoria centrale.
  - Il bus interno B, che raccoglie i dati dai registri e li dà in pasto alla ALU.
  - Il bus interno C, che parte dalle uscite del registro a scorrimento e va a finire sulle entrate di (quasi) tutti i registri).

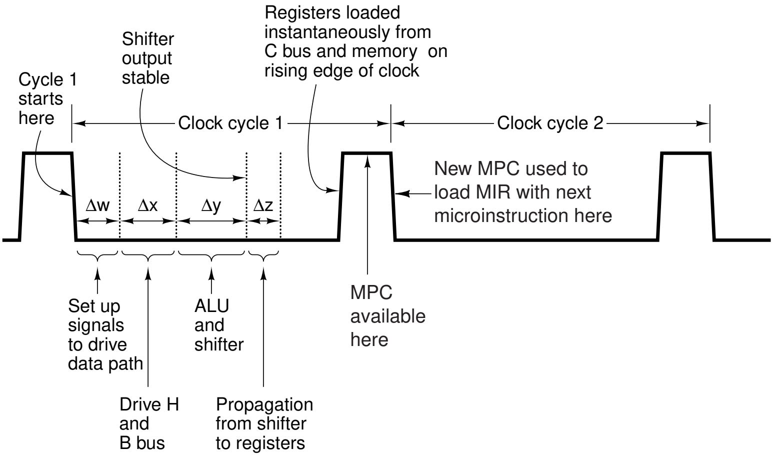
Il Percorso Dati di Mic-1



Temporizzazione del Percorso Dati

- La temporizzazione del percorso dati che abbiamo appena descritto è abbastanza semplice.
- Tutti i registri ricevono in ingresso un segnale proveniente da un singolo clock.
- Ogni ciclo di clock è suddiviso in un certo numero di fasi:
  - Nella **prima** fase, i segnali di controllo si propagano e giungono ai registri, alla ALU e al registro di scorrimento.
  - In una **seconda** fase, i 32 bit presenti nel registro `H` vanno a finire sul primo ingresso dell'ALU e i 32 bit presenti presenti in uno degli altri registri si propagano attraverso il bus B e vanno a finire sul secondo ingresso dell'ALU.
  - Nella **terza** fase, il segnala si propaga attraverso l'ALU e, successivamente, attraverslo il registro a scorrimento.
  - Nella **quarta** e ultima fase, il risultato prodotto dallo shifter si propaga attraverso il bus C e va a finire in uno (o più) registri.
- La frequenza di clock dovrà essere sufficientemente bassa da garantire che le quattro fasi possano essere eseguite in un unico ciclo di clock.

Temporizzazione del Percorso Dati



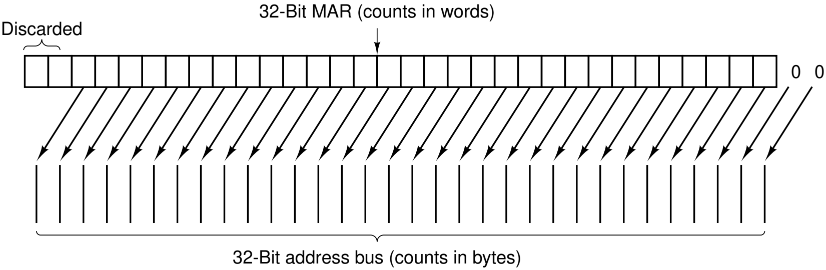
Interazione con la Memoria

- La memoria con cui Mic-1 interagisce è una memoria a 4 Gigabyte. Ogni byte corrisponderà ad un indirizzo a 32 bit.
- La microarchitettura Mic-1 può interagire con la memoria attraverso **due porte**.
- Abbiamo innanzitutto una porta a 32 bit.
  - È controllata dai registri `MAR` (Memory Address Register) e `MDR` (Memory Data Register).
  - Il registro `MAR` contiene indirizzi espressi in parole (a 32 bit).
  - Se si inserisce il valore n in `MAR`, sarà caricata in `MDR` la parola di memoria comprendente i quattro byte di indirizzo compreso tra $4n$ e $4n + 3$.
- Abbiamo poi una porta a 8 bit.
  - È controllata dai registri `PC` e `MBR`.
  - Il registro `PC` contiene indirizzi espressi in parole.
  - Il registro `MBR` è un registro a 8 bit e può essere copiato sul bus B in due modi diversi: con e senza segno. Nel secondo caso si utilizza la cosiddetta **estensione del segno**.

Interazione con la Memoria

- Un operazione di lettura o di scrittura in memoria inizia alla fine del ciclo di clock, dopo che `MAR` (o `PC`) sono stati caricati.
- L'operazione di lettura finisce al termine del ciclo di clock successivo a quello in cui è iniziata.
  - I dati saranno utilizzabili nel ciclo ancora successivo.
  - In altre parole, un'operazione di lettura iniziata alla fine del ciclo k trasmette dati che possono essere utilizzati a partire dal ciclo $k + 2$.
- I registri `MBR` e `MDR` possono essere **letti** nei cicli in cui si sta svolgendo una nuova lettura della memoria.

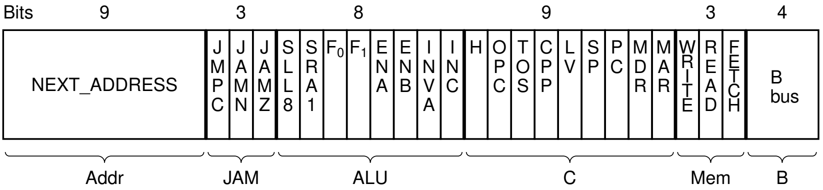
Il Registro MAR e gli Indirizzi



Microistruzioni

- Per controllare il percorso dati di Mic-1 abbiamo bisogno di 29 segnali suddivisibili in 5 gruppi:
  - 9 segnali per controllare la scrittura dei dati dal bus C sui registri.
  - 9 segnali per controllare l'abilitazione dei registri sul bus B.
  - 8 segnali per controllare ALU e registro a scorrimento.
  - 2 segnali per indicare alla memoria di leggere (o scrivere) attraverso la porta a 32 bit.
  - Un segnale per indicare il prelievo dalla memoria attraverso la porta a 8 bit.
- In realtà soltanto uno dei nove registri potrà essere abilitato sul bus B e quindi i relativi segnali di controllo potranno diventare 4.
- Le **microistruzioni** di Mic-1 consistono in una sequenza di 36 bit.
  - I 24 bit più significativi corrispondono ai segnali di controllo appena descritti.
  - I rimanenti 12 bit sono suddivisi in due campi chiamati `NEXT_ADDRESS` e `JAM`, che determinano come viene selezionata la successiva microistruzione.

Formato delle Microistruzioni di Mic-1



B bus registers

0 = MDR
1 = PC
2 = MBR
3 = MBRU
4 = SP
5 = LV
6 = CPP
7 = TOS
8 = OPC
9-15 none

La Memoria di Controllo

- La **memoria di controllo** è una componente essenziale di Mic-1.
- Consiste di 512 parole da 36 bit.
- Può essere implementata come:
  - Una memoria a sola lettura con indirizzi a 9 bit e dati a 36 bit.
  - Una rete combinatoria con 9 ingressi e 36 uscite.
- La memoria di controllo necessita di:
  - Un registro per gli indirizzi, che chiameremo `MPC`.
  - Un registro per i dati, che chiameremo `MIR`. In ogni momento, `MIR` contiene la microistruizone in esecuzione.

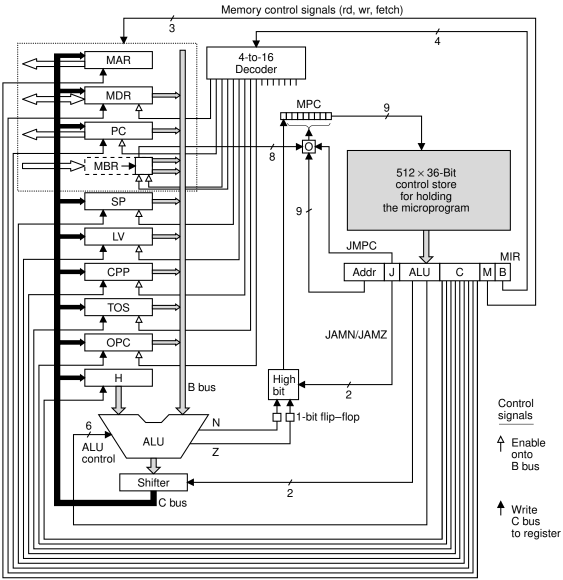
Determinazione della Microistruzione Successiva

- Oltre a guidare il percorso dati, il microprogramma deve parallelamente determinare quale sarà l'istruzione successiva.
- L'indirizzo della prossima microistruzione viene determinato considerando i valori di alcuni registri, ovvero `MIR`, `MBR`, `N`, e `Z` e viene inserito in `MPC`
- In particolare, si procede come segue:
  - Prima di tutto si copiano i 9 bit di `NEXT_ADDRESS` in `MPC`.
  - Nel frattempo, si ispezionano i 3 bit del campo `JAM` e si procede in modi diversi a seconda dei possibili valori di questi tre bit, chiamati `JMPC`, `JAMN` e `JAMZ`.
  - Il bit più significativo di `MPC` prende il valore dell'espressione

    $$(\mathtt{JAMZ} \cdot \mathtt{Z}) + (\mathtt{JAMN} \cdot \mathtt{N}) + \mathtt{NEXT\_ADDRESS}[8]$$

    dove `NEXT_ADDRESS`[8] è il bit più significativo tra quelli in `NEXT_ADDRESS`.
  - Gli 8 bit meno significativi di `MPC` prendono invece il valore degli 8 bit meno significativi di `NEXT_ADDRESS` (se $\mathtt{JMPC} = 0$) oppure la somma logica degli 8 bit meno significativi di `NEXT_ADDRESS` con il valore di `MBR` (se $\mathtt{JMPC} = 1$).

La Microarchitettura Mic-1



Temporizzazione di Mic-1

- Possiamo a questo punto ritornare a parlare della temporizzazione di Mic-1, tenendo conto del ruolo svolto dalla memoria di controlo e dai registri `MPC` e `MIR`.
- Ogni ciclo di clock è suddiviso nei seguenti **sottocicli**:
  1. In corrispondenza del fronte di discesa del clock, l'indirizzo della memoria di controllo contenuto in `MPC` viene letto e caricato in `MIR`.
  2. I segnali si propagano da `MIR` verso il percorso dati e il registro selezionato viene caricao sul bus B.
  3. La ALU e lo shifter svolgono le loro azioni e generano un risultato stabile.
  4. Il bus C e i bus di memoria diventano stabili.
  5. In corrispondenza del fronte di salita del clock, i registri su cui insiste il bus C, i flip-flop `N` e `Z` e i registri `MBR` e `MDR` vengono caricati.
  6. Non appena `MBR`, `N` e `Z` sono stabili, si carica `MPC` in previsione della sucessiva microistruzione.

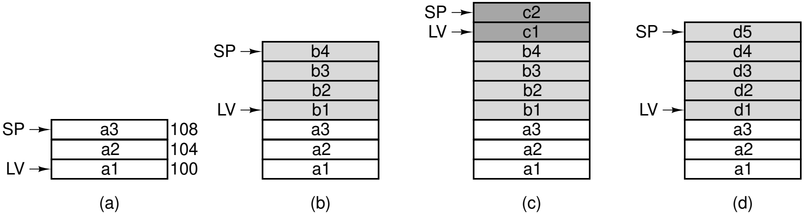
Introduzione a IJVM

- **IJVM** è una particolare macchina astratta, derivata dalla macchina virtuale JVM.
- In questa parte del corso costruiremo un interprete di IJVM nella microarchitettura Mic-1.
  - In altre parole, faremo vedere come sia possibile eseguire istruzioni IJVM presenti in memoria (micro)-programmando opportunamente Mic-1.
- Il linguaggio di IJVM comprende alcuni costrutti che sono tipici dei linguaggi del livello ISA e che analizzeremo qui per la prima volta.
  - Ci riferiamo, in particolare, alle **subroutine** (o **procedure** o **metodi**).
- La macchina virtuale IJVM, inoltre, utilizza un modello di memoria molto comune, basato sul concetto di **stack**.

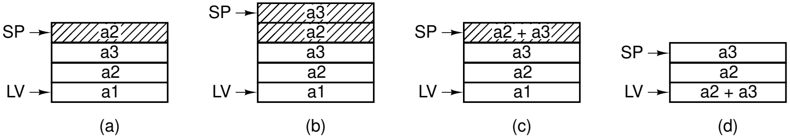
Stack

- Lo **stack** è un'area di memoria cui non si può accedere arbitrariamente (o in modo "casuale").
- Più nello specifico, lo stack si può leggere o scrivere solo ad una "estremità".
  - Proprio per questo si utilizza il termine **stack**, che significa pila.
- Lo stack viene utilizzato in IJVM per due scopi specifici:
  - Da una parte per tener traccia delle variabili locali di una subroutine.
  - Dall'altra per memorizzare gli operandi durante il calcolo di un'espressione aritmetica.
- I due registri `SP` e `LV` servono proprio ad accedere allo stack.
  - `LV` contiene l'indirizzo della prima parola di memoria tra quelle che contengono le variabili locali della subroutine attualmente in esecuzione.
  - `SP` contiene invece l'indirizzo dell'ultima parola di memoria tra quelle utilizzate per la subroutine attualmente in esecuzione.

Stack e Variabili Locali



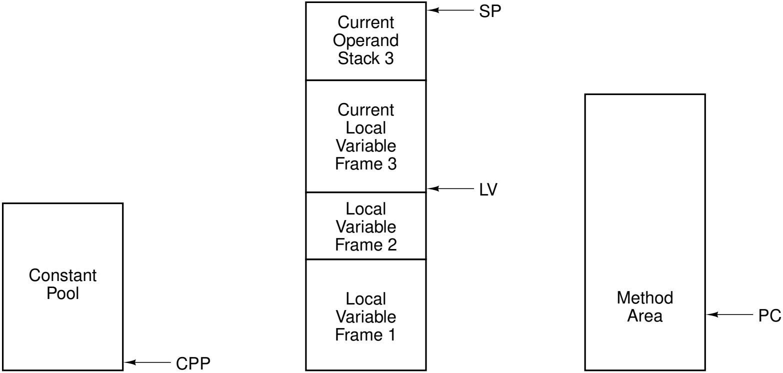
Stack e Operandi



Modello di Memoria di IJVM

- La memoria di IJVM può essere vista come un insieme di 2^{32} byte oppure come un insieme di 2^{30} parole da 4 byte ciascuna.
- Non si possono usare indirizzi espliciti per accedere alla memoria. Occorre fare sempre riferimento ad un indirizzo contenuto in un registro.
- distinguiamo le seguenti aree di memoria:
  - La **porzione costante di memoria**, che contiene dati che non verranno modificati durante l'esecuzione del programma. Vi sia accede tramite il registro `CPP`
  - Il **blocco delle variabili locali**, che contiene le variabili locali di tutte le invocazioni di metodi attive. Vi si accede tramite il registro `LV`.
  - Lo **stack degli operandi**, che noi supponiamo essere parte del blocco delle variabili locali. Vi si accede tramite il registro `SP`.
  - L'**area dei metodi**, che contiene il programma attualmente in esecuzione. Vi si acced tramite il registro `PC`.
- I registri `CPP`, `LV` e `SP` contengono riferimenti a parole (di 32 bit), mentre il registro `PC` è un puntatore a byte.

Modello di Memoria di IJVM



Istruzioni di IJVM

- Esistono prima di tutto un certo numero di istruzioni per **inserire nello stack** una parola proveniente da varie fonti:

`BIPUSH` *byte*	Scrive *byte* in cima allo stack.
`DUP`	Legge la parola in cima allo stack e la inserisce in cima allo stack.
`ILOAD` *varnum*	Scrive una variabile locale in cima allo stack.
`LDC_W` *index*	Scrive in cima allo stack una costante proveniente dalla porzione costante di memoria.

Istruzioni di IJVM

- Possiamo anche **togliere** una parola dalla cima dello stack:

`ISTORE` *varnum*	Preleva una parola dalla cima dello stack e la memorizza in una variabile locale.
`POP`	Rimuove la parola di memoria che sta in cima allo stack.

Istruzioni di IJVM

- Esistono poi un certo numero di istruzioni **logiche e aritmetiche**:

`IADD`	Sostituisce le due parole in cima allo stack con la loro somma.
`IAND`	Sostituisce le due parole in cima allo stack con la loro congiunzione logica.
`IINC` *varnum const*	Aggiunge *const* ad una variabile locale.
`IOR`	Sostituisce le due parole in cima allo stack con la loro disgiunzione logica.
`ISUB`	Sostituisce le due parole in cima allo stack con la loro differenza.

Istruzioni di IJVM

- Possiamo poi individuare un certo numero di istruzioni di **salto**:

`GOTO` *offset*	Diramazione incondizionata.
`IFEQ` *offset*	Estrae una parola in cima allo stack e effettua una diramazione se vale zero.
`IFLT` *offset*	Estrae una parola in cima allo stack e effettua una diramazione se ha valore negativo.
`IF_ICMPEQ` *offset*	Estrae due parole dalla cima dello stack e effettua una diramazione se sono uguali.

Istruzioni di IJVM

- In IJVM esiste la possibilità di **chiamare un metodo**:

`INVOKEVIRTUAL` *disp*	Invoca il metodo che si trova nell'area dei metodi con spiazzamento *disp*.
`IRETURN`	Termina un metodo restituendo un valore intero, che viene posto in cima allo stack.

Istruzioni di IJVM

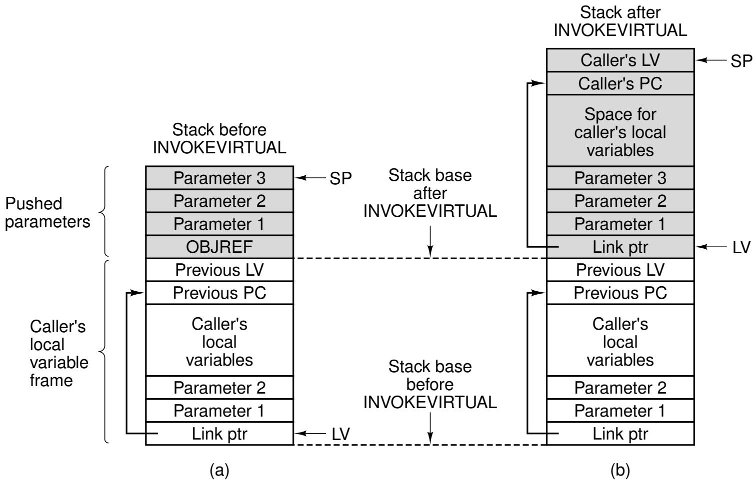
- Infine, esistono un certo numero di istruzioni **diverse**:

`NOP`	Non esegue nulla.
`SWAP`	Scambia le due parole in cima allo stack.
`WIDE`	Funge da prefisso. L'istruzione successiva ha un indice a 16 bit.

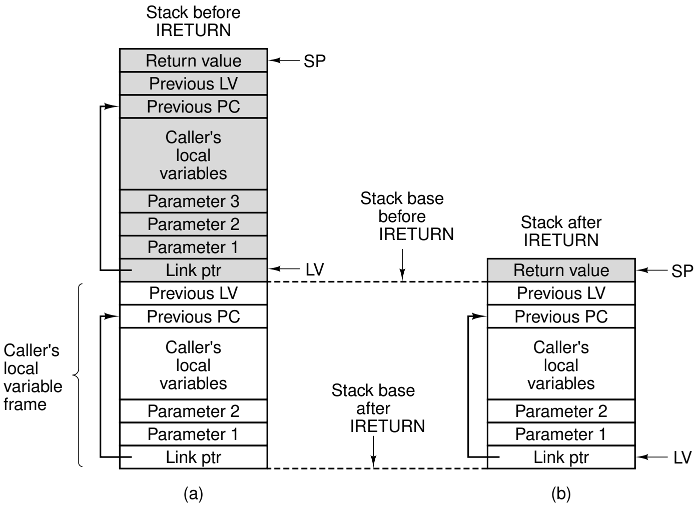
Invocazione di un Metodo e Ritorno da un Metodo

- Prima di invocare un metodo (tramite l'istruzione `INVOKEVIRTUAL`) occorre che sullo stack vi sia un **riferimento all'oggetto chiamante**, seguito dai **parametri attuali**.
  - In IJVM l'oggetto chiamante è sempre `OBJREF`.
- L'istruzione `INVOKEVIRTUAL` include uno spiazzamento all'interno della porzione costante di memoria, a sua volta contenente l'indirizzo del metodo vero e proprio.
- La porzione dell'area dei metodi relativa ad una particolare procedura inizia con 32 bit, dopo i quali troviamo il codice vero e proprio.
  - I primi quattro byte codificano il **numero di parametri** del metodo e la **dimensione del blocco delle variabili locali**.
- L'effetto di una istruzione `INVOKEVIRTUAL` è quello di riservare nuovo spazio per le variabili locali, salvando anche il contenuto di `LV` e `PC` al momento della chiamata. I registri `SP` e `LV` saranno modificati.
- Similarmente, l'istruzione `IRETURN` libera spazio sullo stack, copiando opportunamente il valore di ritorno. I registri `SP` e `LV` saranno riportati al valore che avevano prima della chiamata.

Invocazione di un Metodo



Ritorno da un Metodo



Una Notazione Compatta per le Microistruzioni

- A questo punto, possiamo descrivere le microistruzioni di Mic-1 che permettono di interpretare IJVM.
- In teoria, potremmo limitarci ad elencare le 512 parole a 36 bit che costituiscono il contenuto della memoria di controllo.
- È però sconveniente denotare le microistruizioni tramite mere sequenze di bit.
  - In particolare, si perde molto in termini di intuizione.
- Descriveremo le microistruzioni tramite un linguaggio chiamato **MAL** (Micro Assembly Language).
  - Ogni microistruzione corrisponde ad una riga di codice scritta in MAL.
  - L'istruzione fondamentale di MAL è l'assegnamento, la quale descrive in modo compatto un'operazione aritmetico-logica e i registri coinvolti.
  - Esistono poi altre istruzioni di MAL che permettono di indicare l'inizio di una lettura (o di una scrittura) dalla (o verso la) memoria.
  - Infine, esistono l'istruzione `goto`, che permette di catturare i salti incondizionati e condizionali.

Assegnamenti in MAL

- Tramite un assegnamento possiamo indicare in modo compatto un'operazione aritmetico-logica e i relativi registri.
  - Semplici esempi sono $\mathtt{MDR} = \mathtt{SP}$ oppure $\mathtt{MDR} = \mathtt{H} + \mathtt{SP}$.
- In ogni microistruzione, ovviamente, può essere presente al più un assegnamento.
- Solo alcuni assegnamenti sono leciti. In particolare, possono essere utilizzati solo gli assegnamenti che possano essere realizzati tramite il percorso dati.
  - Per esempio, l'assegnamento $\mathtt{MDR} = \mathtt{SP} + \mathtt{MDR}$ non è valido, perché uno dei due operandi di un'addizione deve essere sempre il registro $\mathtt{H}$.
- È possibile assegnare il risultato di un'operazione logica o aritmetica a più registri.
  - Per esempio, l'assegnamento $\mathtt{SP} = \mathtt{MDR} = \mathtt{SP} + 1$ è perfettamente lecito.

Interazioni con la Memoria e Salti in MAL

- Il fatto che in corrispondenza di una microistruzione inizi una lettura (o una scrittura) dalla (o verso la) memoria viene indicata in MAL con `rd` e `wr` (per la porta a 32 bit) oppure con `fetch` (per la porta a 8 bit).
- Ogni microistruzione deve specificare in modo esplicito l'indirizzo della successiva microistruzione. In MAL ciò può avvenire in vari modi:
  - Innanzitutto con un salto incodizionato `goto` *label* dove *label* è l'indirizzo (simbolico o numerico) della successiva istruzione.
  - Tramite un salto condizionato al valore di `N` oppure di `Z`, per esempio

    `if (N) goto` *label1* `else goto` *label2*.

    In tal caso occorre che *label1* e *label2* differiscano solo per il valore del bit più significativo. Stiamo implicitamente settando a 1 il bit `JAMZ` o il bit `JAMN`.
  - Tramite un salto incondizionato nella forma `goto (MBR OR` *valore*`)`. Stiamo implicitamente settando a 1 il bit `JMPC`.

Interpretazione di IJVM con Mic-1

- Il microprogramma Mic-1 che interpreta le istruzioni IJVM è composto da 112 microistruzioni.
  - Non analizzeremo l'interpretazione di tutte le istruzioni IJVM, concentrandoci su alcuni casi tipici.
  - Sul libro trovate il dettaglio relativo a tutte le altre istruzioni.
- I registri `CPP`, `LV` e `SP` contengono, rispettivamente, puntatori alla porzione costante di memoria, al blocco delle variabili locali e alla cima dello stack.
- All'inizio e alla fine dell'interpretazione di ogni istruzione IJVM, il registro `TOS` contiene il valore puntato da `SP`.
- Il registro `OPC` è un registro temporaneo.
- Il microprogramma Mic-1 che interpreta le istruzioni IJVM contiene una microistruzione che viene eseguita appena dopo che il codice operativo dell'istruzione IJVM successiva è stato caricato nel registro `MBR`:

$$\text{Main1} \qquad \mathtt{PC} = \mathtt{PC} + 1;\ \mathtt{fetch};\ \mathtt{goto}\ \ (\mathtt{MBR})$$

Interpretazione di IADD

iadd1	MAR = SP = SP − 1; rd
iadd2	H = TOS
iadd3	MDR = TOS = MDR + H; wr; goto Main1

Interpretazione di DUP

dup1	$\mathtt{SP} = \mathtt{SP} + 1$
dup2	$\mathtt{MDR} = \mathtt{TOS}$; wr; goto Main1

Interpretazione di BIPUSH

bipush1	$\mathtt{SP} = \mathtt{MAR} = \mathtt{SP} + 1$
bipush2	$\mathtt{PC} = \mathtt{PC} + 1$; `fetch`
bipush3	$\mathtt{MDR} = \mathtt{TOS} = \mathtt{MBR}$; `wr`; `goto` Main1

Interpretazione di ILOAD

iload1	$\texttt{H} = \texttt{LV}$
iload2	$\texttt{MAR} = \texttt{MBRU} + \texttt{H};\ \texttt{rd}$
iload3	$\texttt{MAR} = \texttt{SP} = \texttt{SP} + 1$
iload4	$\texttt{PC} = \texttt{PC} + 1;\ \texttt{fetch};\ \texttt{wr}$
iload5	$\texttt{TOS} = \texttt{MDR};\ \texttt{goto}\ \text{Main1}$

Interpretazione di WIDE

wide1	$\texttt{PC} = \texttt{PC} + 1$; `fetch`
wide2	`goto (MBR OR 0`x`100)`

wide_iload1	$\texttt{PC} = \texttt{PC} + 1$; `fetch`
wide_iload2	$\texttt{H} = \texttt{MBRU} << 8$
wide_iload3	$\texttt{H} = \texttt{MBRU OR H}$
wide_iload4	$\texttt{MAR} = \texttt{LV} + \texttt{H}$; `rd`; `goto` iload3

Codici di `ILOAD` e di `WIDE ILOAD`

ILOAD (0x15)	INDEX

(a)

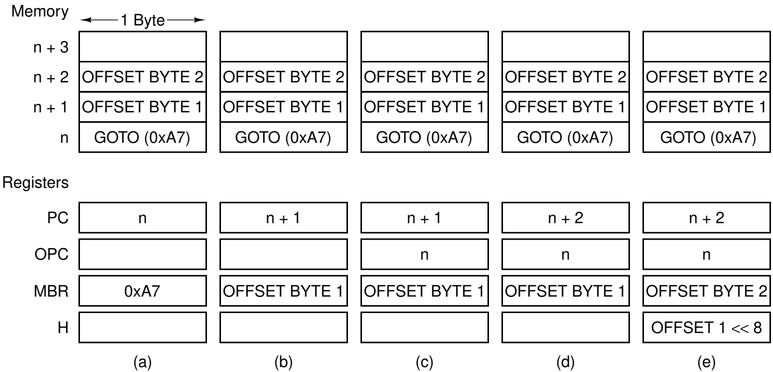
WIDE (0xC4)	ILOAD (0x15)	INDEX BYTE 1	INDEX BYTE 2

(b)

Interpretazione di GOTO

goto1	$\mathtt{OPC} = \mathtt{PC} - 1$
goto2	$\mathtt{PC} = \mathtt{PC} + 1$; `fetch`
goto3	$\mathtt{H} = \mathtt{MBR} << 8$
goto4	$\mathtt{H} = \mathtt{MBRU}\ \ \mathtt{OR}\ \ \mathtt{H}$
goto5	$\mathtt{PC} = \mathtt{OPC} + \mathtt{H}$; `fetch`
goto6	goto Main1

Interpretazione di GOTO



Interpretazione di IFLT

iflt1	$\texttt{MAR} = \texttt{SP} = \texttt{SP} - 1;\ \texttt{rd}$
iflt2	$\texttt{OPC} = \texttt{TOS}$
iflt3	$\texttt{TOS} = \texttt{MDR}$
iflt4	`if (N) goto` T `else goto` F.

T	$\texttt{OPC} = \texttt{PC} + 1;$ `goto` goto2

F	$\texttt{PC} = \texttt{PC} + 1$
F1	$\texttt{PC} = \texttt{PC} + 1;\ \texttt{fetch}$
F2	`goto` Main1

Interpretazione di IF_ICMPEQ

if_icmpeq1	$\texttt{MAR} = \texttt{SP} = \texttt{SP} - 1$; $\texttt{rd}$
if_icmpeq2	$\texttt{MAR} = \texttt{SP} = \texttt{SP} - 1$
if_icmpeq3	$\texttt{H} = \texttt{MDR}$; $\texttt{rd}$
if_icmpeq4	$\texttt{OPC} = \texttt{TOS}$
if_icmpeq5	$\texttt{TOS} = \texttt{MDR}$.
if_icmpeq6	$\texttt{Z} = \texttt{OPC} - \texttt{H}$; $\texttt{if (Z) goto}$ T $\texttt{else goto}$ F.

T	$\texttt{OPC} = \texttt{PC} + 1$; $\texttt{goto}$ goto2

F	$\texttt{PC} = \texttt{PC} + 1$
F1	$\texttt{PC} = \texttt{PC} + 1$; $\texttt{fetch}$
F2	$\texttt{goto}$ Main1

Interpretazione di `IRETURN`

ireturn1	`MAR = SP = LV; rd`
ireturn2	
ireturn3	`LV = MAR = MDR; rd`
ireturn4	`MAR = LV + 1`
ireturn5	`PC = MDR; rd; fetch.`
ireturn6	`MAR = SP.`
ireturn7	`LV = MDR.`
ireturn8	`MDR = TOS; wr; goto Main1.`

Velocità e Costi

- L'impiego di nuove tecnologie nel **livello dei dispositivi** ha prodotto un miglioramento notveole delle prestazioni.
- Noi siamo però interessati ai miglioramenti delle prestazioni indotti da cambiamenti nel livello della **architettura**.
- Una volta scelta una tecnologia per i circuiti e fissato un linguaggio al livello ISA, esistono principalmente tre approcci tramite i quali è possibile aumentare la velocità di esecuizione:
  - **Ridurre la lunghezza del percorso**, ossia il numero di cicli di clock necessari per eseguire un'istruzione.
  - **Semplificare l'organizzazione** in modo che il ciclo di clock possa essere più breve.
  - **Sovrapporre l'esecuzione** di più istruzioni, per esempio tramite meccanismi di pipelining.
- La riduzione della lunghezza del percorso può essere effettuata tramite l'impiego di unità hardware specializzate.
- La sovrapposizione dell'esecuzione di istruzione è di gran lunga la tecnica più efficace tra quelle elencate.
- Ad un aumento della velocità spesso corrisponde un aumento dei costi.

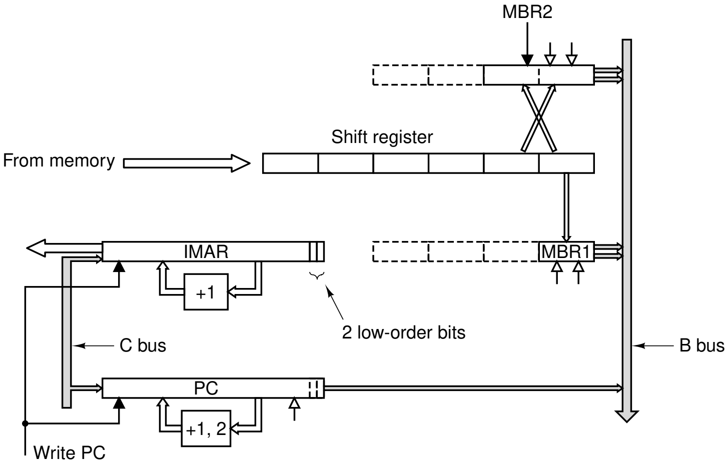
Riduzione della Lunghezza del Percorso

- Fondamentalmente, sono tre le tecniche più importanti tra quelle che permettono di ridurre la lunghezza del percorso di esecuzione.
- Prima di tutto, si può tentare di **sovrapporre l'esecuzione del ciclo principale** (ovvero della microistruzione Main1) all'esecuzione di una delle ultime microistruzioni relative ad ogni istruzione IJVM.
  - In taluni casi è possibile ridurre il numero di microistruzioni necessarie.
- Si può poi **passare ad un'architettura a tre bus**.
  - Anche in questo caso, è possibile che il numero di microistruzioni necessarie diminuisca.
  - In molte occasioni, infatti, occorre un'istruzione aggiuntiva per spostare il contenuto di un registro nel registro `H`.
- C'è infine la possibilità di utilizzare un'**unità di prelievo delle istruzioni** (o **IFU**) che memorizzi le successive istruzioni da eseguire in un buffer.
  - In questo modo possiamo ottenere miglioramenti netti nelle prestazioni.

Unità di Prelievo dell'Istruzione

- Nell'interpretazione di un'istruzione IJVM, la ALU viene utilizzata, oltre che per i calcoli veri e propri, anche per il **prelievo dell'istruzione**.
- Per poter sovrapporre il ciclo principale è necessario liberare la ALU da alcuni di questi compiti.
  - Si potrebbe introdurre un'altra ALU, anche se non sono in realtà necessarie tutte le funzionalità di una ALU.
- È possibile integrare facilmente in Mic-1 un'unità hardware che abbia il solo compito di incrementare il registro `PC` in modo indipendente, prelevando dalla memoria le istruzioni e i relativi operandi. Tale unità è chiamata **IFU** (o **Instruction Fetch Unit**).
- Il registro `MBR` è rimpiazzato da due registri `MBR1` (a 8 bit) e `MBR2` (a 16 bit), che vengono alimentati da un piccolo buffer a 6 byte.
  - Soltanto il buffer interagirà con la memoria.
- Oltre al registro `PC` esisterà anche un registro `IMAR`.
  - Soltanto il registro `IMAR` interagirà con la memoria e non sarà accessibile dal microcodice.

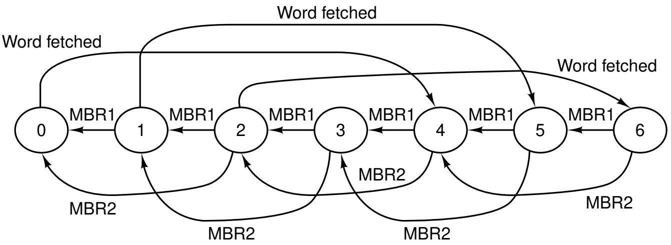
Unità di Prelievo per Mic-1



Architettura con Prefetching

- L'architettura a tre bus costruita a partire da Mic-1 e estendendola con una IFU è chiamata **Mic-2**.
- In Mic-2 non occorre preoccuparsi di modificare `PC` per farlo puntare all'istruzione successiva.
  - Ogniqualvolta uno dei registri `MBR1` oppure `MBR2` sono letti, la IFU aggiorna `PC` automaticamente e provvede a recuperare le parole di memoria necessarie.
- Mic-2 permette di risparmiare molti cicli di clock, anche se il miglioramento a livello prestazionale non è lo stesso per tutte le istruzioni.
  - `LDC_W` passa da nove a tre microistruzioni.
  - `SWAP` passa invece soltanto da otto a sei microistruzioni.
- Il prezzo da pagare consiste nei nuovi componenti hardware di cui abbiamo bisogno (in particolare, l'IFU).

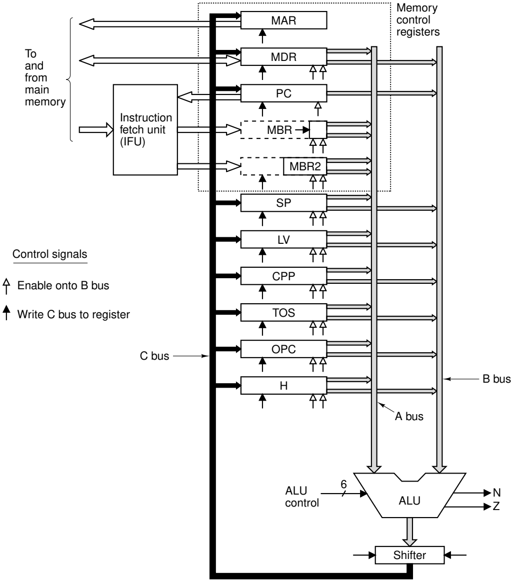
Automa a Stati Finiti che Modella il Funzionamento della IFU



Transitions

MBR1: Occurs when MBR1 is read
MBR2: Occurs when MBR2 is read
Word fetched: Occurs when a memory word is read and 4 bytes are put into the shift register

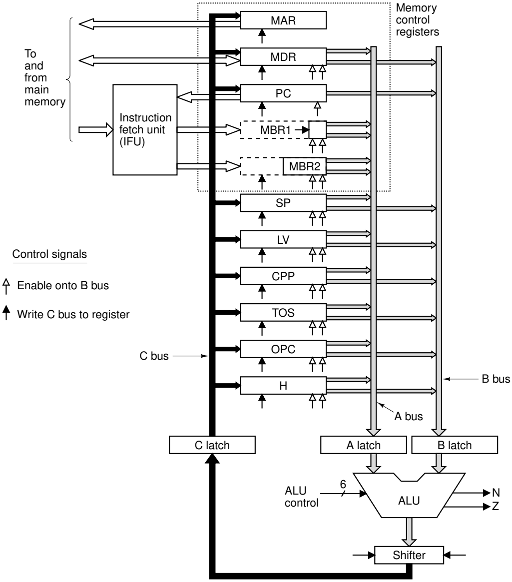
Percorso Dati di Mic-2



Architettura a Pipeline

- Un altra tecnica per migliorare le prestazioni della nostra microarchitettura di esempio, consiste nell'introdurre un certo livello di parallelismo.
- Si potrebbe aumentare la frequenza di clock.
  - Sappiamo, però, che il periodo di clock non può essere minore del tempo necessario ai segnali per propagarsi lungo il percorso dati.
- Il ciclo del percorso dati è composto da tre componenti principali:
  - Il tempo necessario a portare i registri selezionati sui bus A e B.
  - Il tempo impiegato dalla ALU e dallo shifter per compiere il loro lavoro.
  - Il tempo necessario per riportare i risultati nei registri.
- Introducendo tre latch "nel mezzo" dei bus A, B e C otteniamo i due vantaggi seguenti:
  - Prima di tutto, possiamo **accellerare** il clock.
  - Possiamo poi (potenzialmente) utilizzare **tutte le parti del percorso dati** durante ogni ciclo.

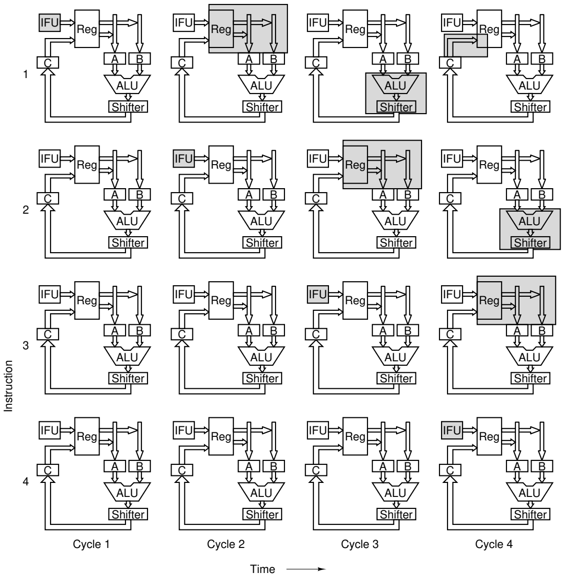
Percorso Dati di Mic-3



Funzionamento di Mic-3

- Ad ogni ciclo di clock, vi sono potenzialmente tre microistruzioni attive:
  - La **prima** comanda la parte di percorso dati compresa tra i registri e i latch A and B.
  - La **seconda** comanda la parte di percorso dati compresa tra i latch A e B e il latch C.
  - La **terza** comanda la parte di percorso dati compresa tra il latch C e i registri.
- In alcune situazioni, però, questo livello di parallelismo non è possibile, perché una delle microistuzioni potrebbe dipendere dal risultato di una delle microistruzioni precedenti.
  - Si parla in questo caso di **dipendenza RAW** o **dipendenza effettiva**.

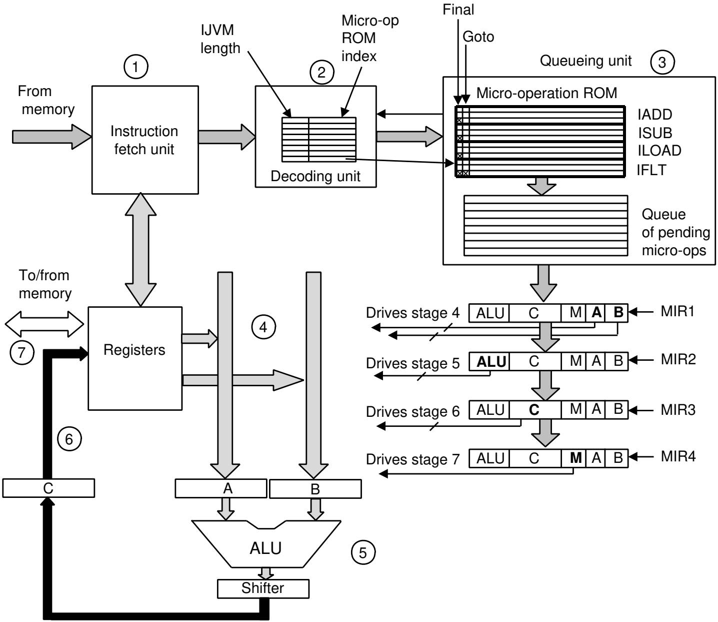
Rappresentazione Grafica del Funzionamento di un Pipeline



Pipeline a Sette Stadi

- Possiamo estendere la pipeline a quattro stadi di Mic-3 ad una pipeline a **sette** stadi, ottendendo quindi una microarchitettura chiamata Mic-4.
- In Mic-4, la IFU alimenta una nuova componente, chiamata **unità di decodifica**, dotata di una ROM indicizzata attraverso il codice operativo IJVM.
  - Per ogni codice operativo, la ROM tiene traccia del numero degli operandi e di un indice ad un'altra ROM, chiamata **ROM delle micro-operazioni**.
- L'unità di decodifica invia all'**unità di accodamento** l'indice relativo alla ROM delle micro-operazioni che ha trovato nella sua tabella.
  - L'unità di accodamento cerca nella ROM delle micro-operazioni la micro-operazione corrispondente e la copia in una coda interna, assieme a tutte le micro-operazioni successive
  - In questo modo, la sequenza di istruzioni IJVM in memoria viene convertita in una sequenza di micro-operazioni, che alimentano quattro registri `MIR1`, `MIR2`, `MIR3` e `MIR4`.

Componenti Principali di Mic-4



Pipeline di Mic-4