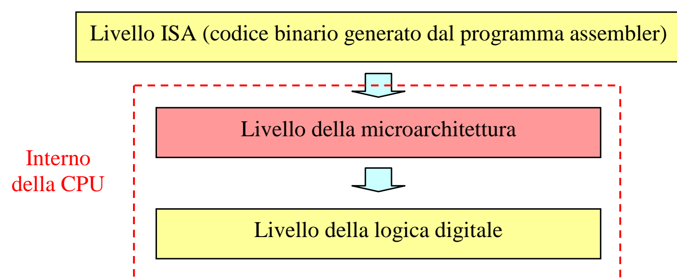
Il livello della microarchitettura

Il livello della **microarchitettura** descrive il funzionamento interno di una **CPU**, e in particolare come le istruzioni **ISA** (**I**nstruction **S**et **A**rchitecture) vengono interpretate ed eseguite dall'hardware (livello della logica digitale) che costituisce la CPU.



Diverse CPU moderne, in particolare quelle **RISC** (**R**educed **I**nstruction **S**et **C**omputer), hanno istruzioni semplici che possono essere eseguite in un singolo ciclo di clock. D'altro canto le CPU **CISC** (**C**omplex **I**nstruction **S**et **C**omputer) forniscono istruzioni anche molto complesse che sono interpretate da microprogrammi che richiedono diversi cicli di clock. In altre parole:

- le CPU **RISC** (es. SGI MIPS, Digital Alpha, Sun UltraSparc, IBM PowerPC) semplificano il disegno della CPU che per questo motivo è spesso molto "snella" e veloce, delegando ai compilatori il compito di tradurre con pochi mattoncini di base programmi anche molto complessi.
- le CPU **CISC** (Motorola 68xxx, Intel x86, Intel Pentium) forniscono ai compilatori molte più possibilità di traduzione e molte semplificazioni (ad. esempio gestione context-switch); questo però va spesso a discapito dell'efficienza e della semplicità di progetto.

Progetto di una microarchitettura

Risulta in generale estremamente difficile dare criteri generali per il progetto di microarchitetture. Per una comprensione dei problemi coinvolti è dunque preferibile analizzare un **esempio** semplice ma abbastanza generale da poter essere espanso e "complicato" a piacere...

Una CPU che implementa istruzioni IJVM

Java è un linguaggio di alto livello per calcolatori (simile al C) introdotto da **Sun** nei primi anni '90 e divenuto piuttosto popolare a partire dal '95 quando i browser Netscape e Internet Explorer decisero di adottarlo. La principale caratteristica del linguaggio Java è quella di essere indipendente dalla piattaforma hardware.

Come far eseguire a macchine con diversi ISA, programmi scritti nello stesso linguaggio ?

- Semplice, basta fornire un **interprete**, per ciascuna architettura, che traduca il codice Java in istruzioni ISA di quell'architettura. Questo approccio ha però il grosso svantaggio di non permettere la compilazione dei programmi e quindi di introdurre forte rallentamento a causa dell'interpretazione "run-time" del codice Java di alto livello.

- La soluzione scelta da Sun, più complessa ma molto più efficiente, consiste nel compilare il programma Java in un codice binario (di basso livello) detto **Bytecode** che viene eseguito da una **JVM** (**J**ava **V**irtual **M**achine). La compilazione risulta quindi indipendente dalla piattaforma hardware scelta per ognuna delle quali dovrà però essere fornita una JVM.

Come realizzare una JVM ?

Una JVM non è altro che un **interprete** (di basso livello) da istruzioni in formato Bytecode-Java a istruzioni ISA.

Esistono in realtà alcuni processori (es. PicoJava II) il cui ISA coincide con le istruzioni JVM; in questo caso non c'è necessità di interprete !

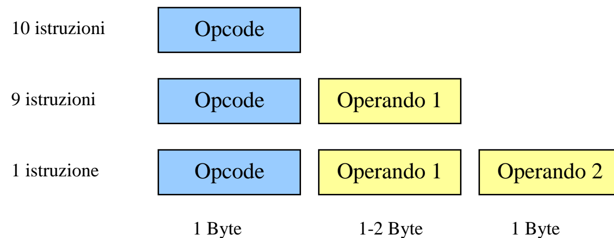
IJVM come ISA

Nel seguito ci concentreremo dunque sullo studio della microarchitettura di una CPU in grado di eseguire un sottoinsieme delle istruzioni Bytecode-Java: questo sottoinsieme contiene solo istruzioni su numeri interi (e non floating point) e per questo motivo è denominato **IJVM** (**I**nteger **JVM**).

Le istruzioni ISA della nostra CPU saranno dunque i Bytecode IJVM, chiamati nel seguito semplicemente **istruzioni IJVM**.

Le (**20**) istruzioni IJVM sono brevi; ogni istruzione è dotata di alcuni campi, solitamente 1 o 2, ognuno dei quali ha uno scopo preciso:

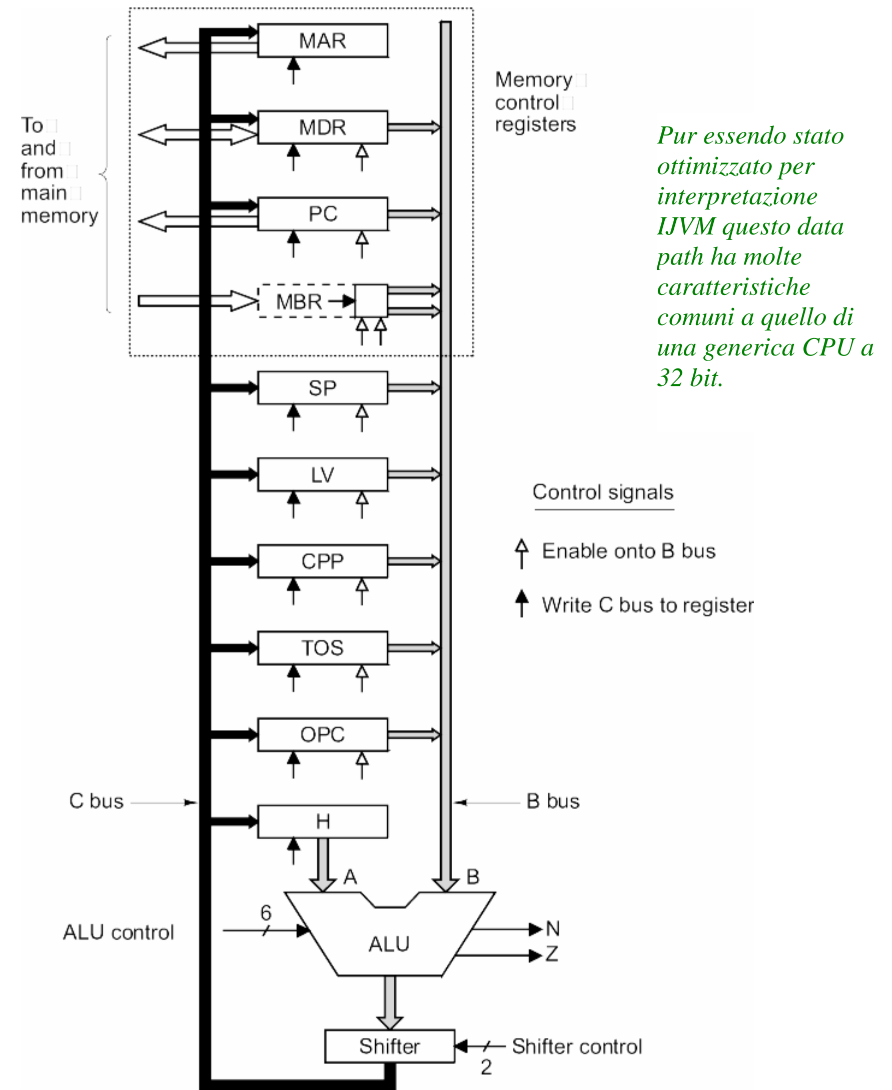
- Il primo campo dell'istruzione è l'**opcode** (abbreviazione di **op**eration **code**) che identifica l'istruzione specificando se si tratta di ADD, BRACH o altro.
- Molte istruzioni dispongono di un campo supplementare che specifica un **operando**. Ad esempio un'istruzione che deve accedere a una variabile locale deve specificare tramite un operando dove questa variabile si trova (indirizzo di memoria).



Come vedremo, le istruzioni IJVM fanno largo uso dello stack. Molte istruzioni vengono infatti eseguite prelevando operandi dalla cima dello stack e salvando sullo stack il risultato. Ciò consente di limitare (solitamente a max. 1) il numero di operandi delle istruzioni (*non è necessario specificare indirizzi di memoria*) e quindi di produrre **Bytecode** molto **compatti**.

Il Data Path

Il data path è il cuore della CPU, ovvero quella parte che contiene l'**ALU** con i suoi input e output, e i **registri** interni.



Pur essendo stato ottimizzato per interpretazione IJVM questo data path ha molte caratteristiche comuni a quello di una generica CPU a 32 bit.

Il Data Path (2)

Nel data path riportato in figura precedente:

- Sono presenti un certo numero di **registri a 32** bit i cui nomi simbolici (es. PC, SP, MDR, …), come sarà più chiaro nel seguito, ricordano la loro funzione.
- Il contenuto di gran parte dei registri può essere **scritto sul bus B**. La linea di controllo "enable to bus B" disponibile per ogni registro abilita il registro (*ovvero collega elettricamente il suo output*) sul bus B evitando conflitti: solo un registro per volta può ovviamente essere abilitato sul bus B !
- Viene utilizzata una **ALU a 32 bit** identica a quella introdotta in precedenza. La sua funzione viene determinata dalle 6 linee di controllo: F_0, F_1, ENA, ENB, INVA, INC. Non tutte le $2^6 = 64$ combinazioni sono utili; la tabella riporta quelle significative:

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	$\overline{A}$
1	0	1	1	0	0	$\overline{B}$
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
0	1	0	0	0	1	1
0	1	0	0	1	0	–1

F_0 ed F_1 definiscono la funzione, ENA ed ENB abilitano A e B rispettivamente, INVA nega l'input in A; infine INC forza un riporto nel bit meno significativo.

Il Data Path (3)

- L'output dell'**ALU**, viene inviato a uno **shift register** che a suo volta produce il suo output sul bus C. Tale output (attraverso il bus C) può essere scritto in input su più registri contemporaneamente.
- Il comportamento dello **shift register** è controllato da due linee:
  - ➢ **SLL8** (**S**hift **L**eft **L**ogical) che sposta il contenuto della parola a 32 bit di un byte a sinistra riempiendo con degli zeri gli 8 bit meno significativi entranti.
  - ➢ **SRA1** (**S**hift **R**ight **A**rithmetic) che sposta il contenuto di un bit a destra senza modificare però il bit più significativo (segno nel complemento a 2!)
- Mentre l'input B dell'**ALU** può in teoria provenire da qualunque dei registri interni, l'input A (in questa architettura) proviene obbligatoriamente dal registro H. Un'architettura alternativa con 2 bus completi in input all'ALU è decisamente più potente ma anche più complessa.

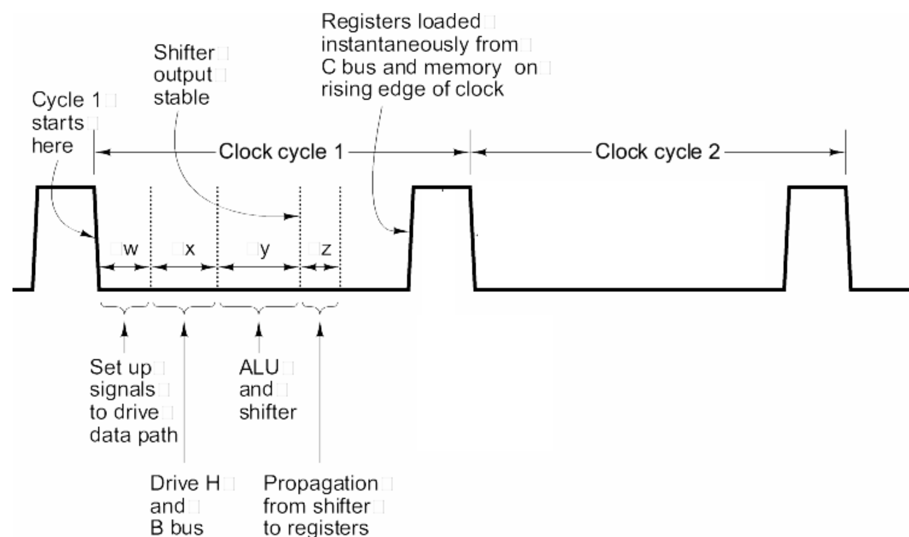
  Come caricare H con il valore di uno dei registri ?

  Semplice, è sufficiente abilitare su B il registro che ci interessa, selezionare come funzione della ALU la seconda configurazione della tabella precedente, passare l'output inalterato sullo shift register e abilitare H in input dal bus C.

- E' possibile leggere e poi scrivere lo stesso registro o anche registri diversi nello stesso ciclo di clock. Come questo possa fisicamente avvenire sarà più chiaro dall'analisi delle temporizzazioni…

  Ad esempio se vogliamo incrementare di 1 il contenuto di SP (Stack Pointer) è sufficiente abilitare su B il registro SP, selezionare come funzione della ALU l'ottava configurazione della tabella precedente (B+1), passare l'output inalterato sullo shift register e abilitare SP in input dal bus C.

Sincronizzazione del Data Path



- Ogni **nuovo ciclo** ha inizio sul fronte di discesa del clock (che come si può notare rimane basso per circa i ¾ del periodo).
- Durante il fronte di discesa vengono memorizzati i bit che controllano il funzionamento delle porte (segnali di controllo). E' necessario attendere un primo intervallo di tempo (**w**) affinché i segnali si propaghino e raggiungano uno stato di stabilità.
- Il registro selezionato per essere inviato sul bus B insieme ad H viene reso disponibile alla ALU. Dopo un intervallo di tempo (**x**) la ALU può iniziare ad operare sui dati.
- Un ulteriore intervallo di tempo (**y**) è necessario affinché le porte logiche di ALU e shift register producano un output stabile.
- Infine, dopo l'intervallo di tempo (**z**) l'output dello shift register è stato propagato lungo il bus C e in corrispondenza del fronte di salita del successivo impulso di clock i registri selezionati vengono caricati dal bus C.

Ovviamente w + x + y + z deve essere minore del tempo in cui il clock è 0.

Data Path e accesso alla memoria

La nostra CPU ha due modi per comunicare con la memoria:

- Una **porta da 32 bit** per la lettura/scrittura dei **dati** del livello ISA. La porta viene controllata da due registri:
  - **MAR** (**M**emory **A**ddress **R**egister) specifica l'indirizzo di memoria in cui si desidera leggere o scrivere una parola.
  - **MDR** (**M**emory **D**ata **R**egister) ospita la parola (32 bit) che sarà letta o scritta all'indirizzo di memoria specificato da MAR.
- Una **porta da 8 bit** per leggere (solo lettura!) il programma eseguibile (**fetch** delle istruzioni ISA). Anche questa porta è controllata da 2 registri:
  - **PC** (**P**rogram **C**ounter) è un registro a 32 bit che indica l'indirizzo di memoria della prossima istruzione ISA da caricare (fetch).
  - **MBR** (**M**emory **B**yte **R**egister) contiene il byte letto dalla memoria durante il fetch. MBR è in realtà un registro a 32 bit, pertanto il byte letto viene memorizzato negli 8 bit meno significativi.

In generale tutti i registri della nostra CPU vengono **controllati** da uno o due segnali di controllo. In figura la freccia vuota indica l'output del registro sul bus B mentre la freccia piena indica il caricamento (input) dal bus C.

MAR non è collegato con il bus B e quindi non necessita del corrispondente segnale di abilitazione. Lo stesso dicasi per H il cui output può essere diretto solo verso l'ALU. D'altro canto MBR non può essere caricato dal bus C è quindi è privo del corrispondente segnale di controllo (freccia piena).

MBR può essere scritto sul bus B in due modi diversi (da qui le due frecce vuote riportate in figura): **unsigned** e **signed**. Nel modo unsigned i 24 bit non utilizzati vengono impostati a 0 (e quindi il byte in MBR fornisce un valore compreso tra 0 e 255); nel modo signed il bit di segno (il settimo), viene copiato su tutti i 24 bit non utilizzati (il valore risultante è compreso tra –128 e + 127).

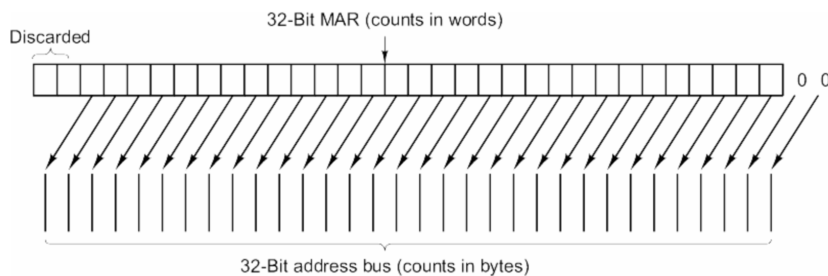
Data Path e accesso alla memoria (2)

Le due porte **MAR/MDR** e **PC/MBR** indirizzano la memoria a parole (32 bit) e a byte (8 bit) rispettivamente.

- Questo significa che gli indirizzi memorizzati in MAR sono espressi in termini di parole: *ad esempio se MAR contiene il valore 2, una lettura dalla memoria causa il caricamento in MDR dei byte di memoria 8, 9, 10 e 11.*
- Indirizzando invece PC la memoria in termini di byte, *la lettura dall'indirizzo 2, causa il trasferimento negli 8 bit meno significativi di MBR del byte di memoria 2.*

Questa apparente complicazione, semplifica in realtà il funzionamento interno in quanto il fetch delle istruzioni può avvenire un byte per volta, mentre risulta possibile leggere o scrivere in memoria una parola (32 bit) in un unico ciclo.

Fisicamente la memoria è realizzata come un unico spazio lineare organizzato in byte. Adottando il semplice accorgimento di figura è possibile indirizzare con MAR la memoria in termini di parole senza bisogno di introdurre nessun particolare circuito:



Il trucco consiste nel collegare MAR sul bus indirizzi sfalsato di due posizioni, ovvero di non utilizzare i due bit più significativi di MAR e di collegare il bit 0 di MAR con il bit 2 degli indirizzi, il bit 1 di MAR con il bit 3 degli indirizzi e così via. I due bit meno significativi degli indirizzi vengono semplicemente impostati a 0.

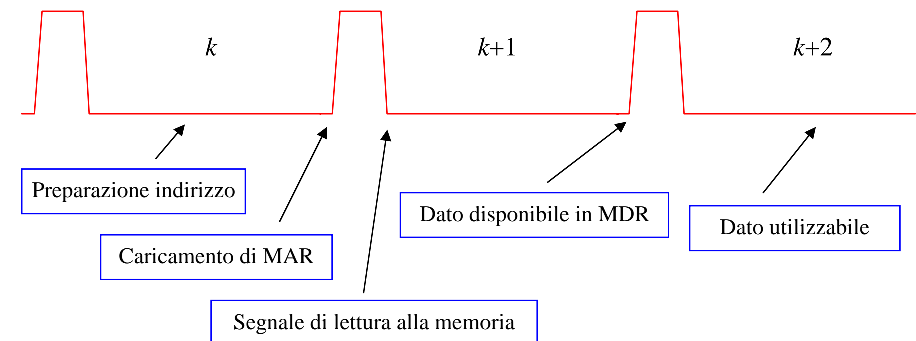
Data Path e accesso alla memoria (3)

Sebbene nello stesso ciclo di data path sia possibile eseguire più operazioni (lettura di registri, utilizzo dell'ALU e dello shift register e scrittura su registri), in un ciclo **NON è possibile completare** la lettura o scrittura di parole in memoria.

Infatti, come già detto a proposito di bus sincroni e asincroni, le memorie non sono in grado di far fronte istantaneamente a una richiesta di lettura o scrittura che non può quindi essere conclusa nello stesso ciclo di clock nel quale è stata inoltrata la richiesta.

Nella nostra microarchitettura se viene attivato un segnale di lettura dalla memoria dati (MAR/MDR), l'operazione di lettura ha inizio al termine del ciclo del data path, dopo aver caricato in MAR l'indirizzo. I dati sono disponibili al termine del ciclo seguente in MDR, e quindi possono essere utilizzati solo **due cicli dopo**.

In altre parole una lettura che ha inizio al ciclo k, fornisce dati alla fine del ciclo $k+1$ (quando oramai non possono più essere utilizzati in quel ciclo) e quindi potranno essere utilizzati solo al ciclo $k+2$.



Nel ciclo $k+1$, la CPU non deve necessariamente rimanere inattiva aspettando la memoria, ma può eseguire un ciclo di data path che non necessita del dato in corso di lettura.

Per il **fetch di istruzioni** (come vedremo) le cose sono diverse: il "ritardo" è di 1 solo ciclo.

Cicli di data path e microistruzioni

Abbiamo fino ad ora analizzato il comportamento del data path, delle sue sincronizzazioni e dei suoi segnali di controllo. Riepilogando, il nostro data path possiede **29 segnali di controllo**:

- **9** segnali (frecce piene) per controllare la scrittura dei dati dal bus C ai registri
- **9** segnali (frecce vuote) per abilitare i registri sul bus B (uno solo per volta può essere abilitato)
- **8** segnali per controllare le funzioni di ALU e shift register
- **2** segnali (che non appaiono in figura) per indicare lettura/scrittura della memoria per mezzo di MAR/MDR.
- **1** segnale (che non appare in figura) per il fetch (lettura) delle istruzioni dalla memoria tramite PC/MBR.

Come già detto in un singolo ciclo di data path è possibile leggere valori da registri, passarli alla ALU e memorizzare su registri il valore calcolato.

I **29 segnali** suddetti specificano il comportamento della CPU per un singolo ciclo di data path. Nella nostra CPU (come nella maggior parte delle CPU) il periodo di data path coincide con il periodo di clock.

Come vedremo, un ciclo di data path non corrisponde però a un'istruzione ISA, per l'implementazione della quale generalmente occorrono diversi cicli di data path (specie quando risulta necessario reperire valori dalla memoria).

La sequenza di cicli di data path necessari all'esecuzione di un'istruzione ISA prende il nome di **microprogramma** di quell'istruzione ISA. Un microprogramma è costituito da **microistruzioni**.

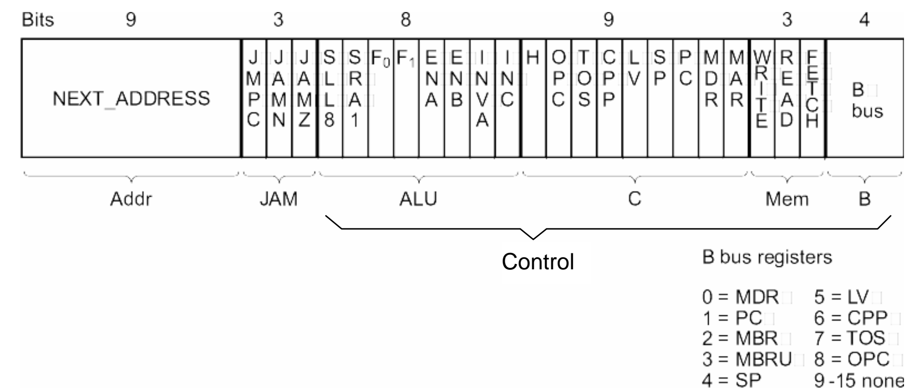
Ma che cos'è esattamente una microistruzione ?

Cicli di data path e microistruzioni (2)

I 29 segnali di controllo non sono sufficienti a specificare una microistruzione; infatti, è necessario specificare anche cosa fare nel ciclo seguente (flusso del microprogramma).

Una **microistruzione** è una sequenza di bit, composta da 3 parti:

- **Control**: stato dei segnali di controllo
- **Address**: indirizzo della prossima microistruzione da eseguire. Attenzione: come vedremo non si riferisce a un indirizzo di memoria esterna alla CPU, ma all'indirizzo di una ROM interna dove sono memorizzati i microprogrammi.
- **JAM**: bit per la gestione di salti condizionali a seconda dei bit di stato (N, Z) dell'ALU.



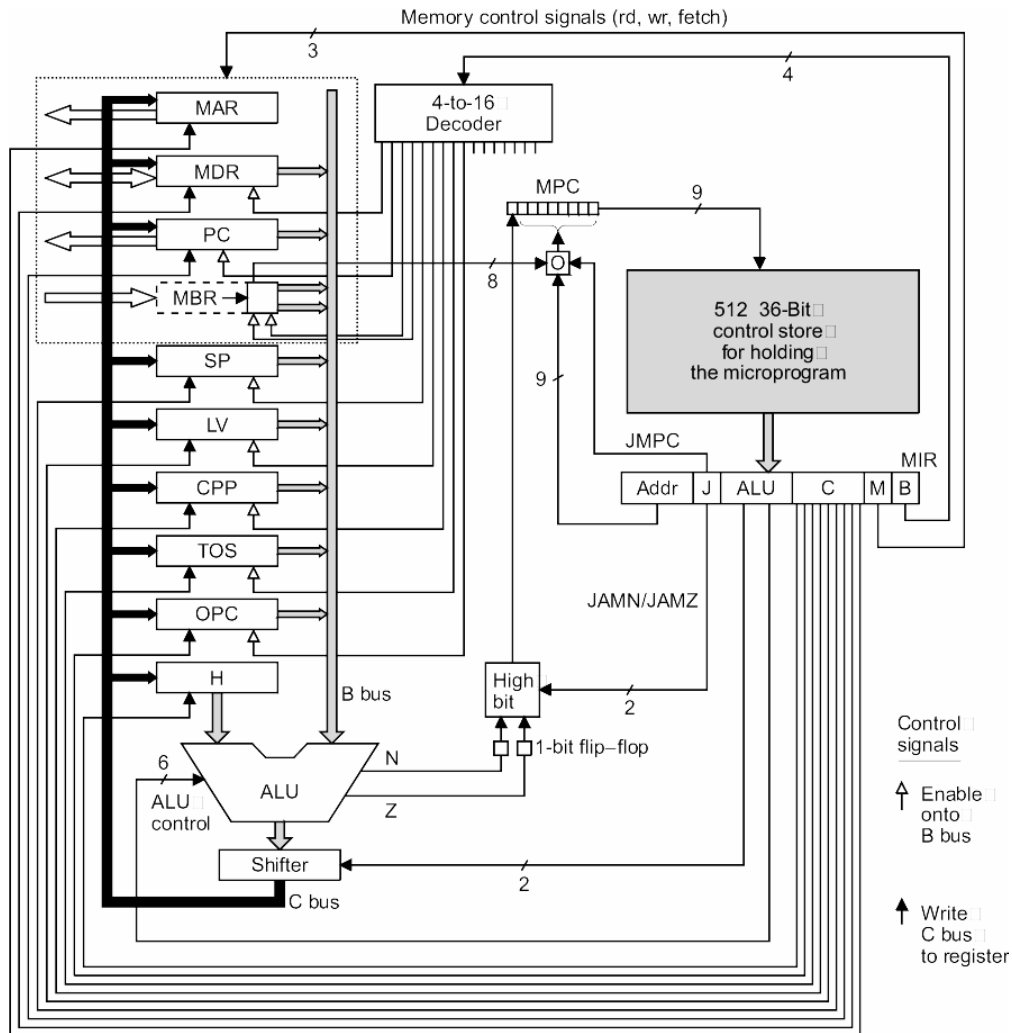
Per ridurre il numero di bit di controllo (da 29 a 24) la nostra microarchitettura utilizza un **decoder** che con soli 4 bit è in grado di specificare quale dei 9 registri abilitare sul bus B ($2^4 = 16 > 9$!).

Utilizzando 9 bit per l'indirizzo della prossima microistruzione, 3 bit per il JAM e 24 bit per il controllo, ciascuna delle nostre microistruzioni avrà una lunghezza pari a **36 bit**.

Ovviamente solo un piccolo sottoinsieme delle 2^{36} possibili microistruzioni saranno di una qualche utilità e verranno utilizzate dai microprogrammi.

Microarchitettura Mic-1

La figura mostra l'architettura completa della nostra CPU che chiameremo **Mic-1**. Sulla parte sinistra troviamo il data path già studiato, mentre quella destra è dedicata al controllo.



Microarchitettura Mic-1 (2)

Il cuore dell'architettura di controllo è una memoria chiamata **control store**. Si tratta di una memoria ROM interna alla CPU e non accessibile dall'esterno che contiene le microistruzioni che compongono i microprogrammi che codificano le istruzioni ISA (talvolta detto firmware della CPU).

- La memoria del nostro esempio contiene **512** parole di **36** bit ciascuna (lunghezza microistruzione). Possiamo quindi memorizzare un insieme di microprogrammi la cui lunghezza totale non supera le 512 microistruzioni.
- La memorizzazione dei microprogrammi nel control store è **piuttosto diversa** dalla memorizzazione dei programmi assembler (istruzioni ISA) nella memoria principale:
  - Infatti le istruzioni ISA vengono solitamente eseguite nello stesso ordine nel quale sono memorizzate (ad esclusione dei salti condizionali o incondizionati). Per questo motivo il registro Program Counter viene normalmente incrementato di una unità per puntare alla prossima istruzione da eseguire.
  - I microprogrammi richiedono invece più flessibilità in quanto le sequenze di microistruzioni devono essere più brevi possibile. Per questo motivo ogni **microistruzione specifica esplicitamente** il suo successore (che può essere ovunque nel control store).

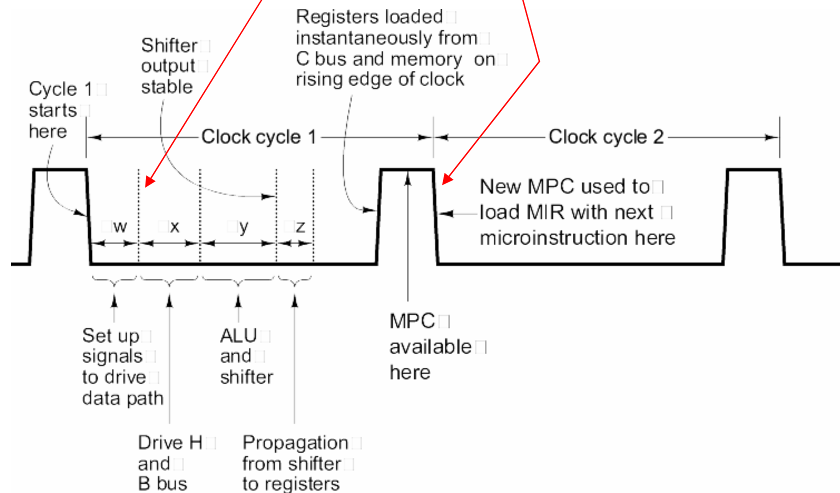
Il control store è interfacciato al resto della CPU tramite due registri:

- **MPC** (**M**icro**P**rogram **C**ounter) che specifica l'indirizzo della prossima microistruzione da eseguire. La sua lunghezza è di 9 bit (512 parole indirizzabili).
- **MIR** (**M**icro**I**nstruction **R**egister) che memorizza la microistruzione corrente i cui bit pilotano i segnali di controllo che attivano il data path. La sua lunghezza è di 36 bit. I gruppi di segnali ALU e C sono direttamente connessi al data path, B è connesso al data path tramite il decoder di cui abbiamo parlato, M sono i segnali di controllo della memoria esterna; le connessioni di ADDR e J sono spiegate nel seguito.

Microarchitettura Mic-1 (3)

Il **funzionamento** della microarchitettura Mic-1 è il seguente:

- All'inizio di ogni ciclo di clock (fronte di discesa) MIR viene caricato con il contenuto della parola indirizzata da MPC. Il caricamento deve terminare entro il tempo **w**.



Infatti dopo un tempo w il data path inizia il proprio **ciclo**: H e uno dei registri (attraverso il bus B) vengono inviati alla ALU che calcola la funzione richiesta e produce sul bus C il risultato; sul fronte di salita del clock i registri selezionati vengono caricati.

Sempre sul fronte di salita del clock i **bit di stato** N (Negative) e Z (Zero bit) vengono memorizzati temporaneamente su 2 Flip-flop per poter essere utilizzati in seguito (dopo il fronte di salita) quando i bus non sono pilotati e lo stato dell'ALU non è stabile.

Subito dopo il fronte di salita del clock il **ciclo è terminato**: tutti i risultati sono stati memorizzati, i risultati delle operazioni di memoria sono disponibili e MPC viene caricato con il nuovo valore (nel modo che vedremo).

Microarchitettura Mic-1 (4)

Il **calcolo dell'indirizzo della prossima microistruzione** da eseguire, ha inizio subito dopo che MIR è stato caricato ed è stabile, e avviene nel modo seguente:

- Il campo di 9 bit specificato da **Addr** (in MIR) viene copiato in **MPC**.
- Vengono a questo punto controllati i bit del campo **JAM**:
  - Se tutti e 3 i bit sono 0, non viene fatto altro e l'indirizzo della prossima microistruzione diviene semplicemente quello indicato da NEXT_ADDREASS.
  - Se JAMN è 1, il bit più significativo di MPC viene messo in OR con N (prelevato dal Flip-flop).
  - Se JAMZ è 1, il bit più significativo di MPC viene messo in OR con Z (prelevato dal Flip-flop).
  - Se entrambi JAMN e JAMZ sono a 1 si fa l'OR con entrambi:

  Detto con un'unica espressione booleana:

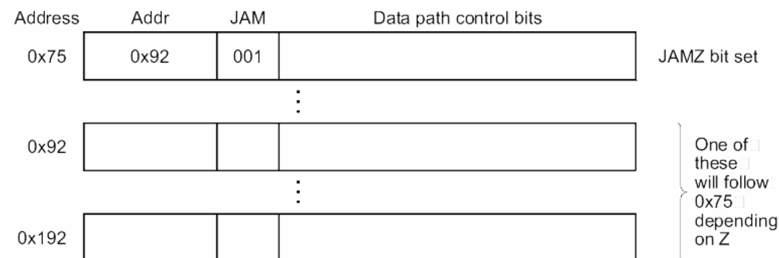
```
F = (JAMZ AND Z) OR (JAMN AND N) OR NEXT_ADDREASS[8]
```

- In ogni caso dunque MPC conterrà uno dei due valori:
  - Il valore di NEXT_ADDRESS
  - Il valore di NEXT_ADDRESS con il bit più significativo in OR con 1

L'**utilizzo** di **JAMZ** e **JAMN**, come sarà più chiaro nel seguito, permette di eseguire salti condizionali a seconda del valore dei bit di stato Z ed N.

Microarchitettura Mic-1 (5)

Nell'esempio seguente, la microistruzione corrente ha NEXT_ADDRESS uguale a 0x92 (92 esadecimale) e il bit JAMZ a 1. Ciò significa che la prossima istruzione sarà 0x92 se Z è zero oppure 0x192 se Z è 1.



Infine, il 3 bit del campo JAM denominato **JAMC** opera nel modo seguente: quando JAMC è a 1 viene eseguito l'OR bit-a-bit tra MBR e gli otto bit meno significativi di NEXT_ADDRESS.

- **Normalmente** quando JAMC è 1, i bit di NEXT_ADDRESS valgono 0 e l'opcode dell'istruzione ISA di cui è stato eseguito il fetch in MBR (avviato al ciclo precedente) determina la prossima microistruzione da eseguire (**inizio** di un nuovo microprogramma).
- La possibilità di eseguire l'OR con il valore di NEXT_ADDRESS consente in effetti di implementare un salto a più vie in modo efficiente.

Mic-1: Micro Assembly Language

Prima di studiare come scrivere microprogrammi per codificare le istruzioni ISA **IJVM** in termini di microistruzioni, introduciamo una notazione semplificativa denominata **MAL** (**M**icro **A**ssembly **L**anguage) per indicare cosa ogni microistruzione deve fare.

Scrivere microprogrammi usando un numero binario di 36 bit per ogni microistruzione risulterebbe infatti piuttosto difficile e soprattutto completamente incomprensibile!

La traduzione da MAL a codici binari 36 bit (**microassembling di MAL**) è un compito noioso ma molto semplice per un calcolatore.

Nella nostra notazione MAL tutto ciò che viene eseguito in un singolo ciclo deve essere scritto in un'unica riga.

Ad esempio se in un ciclo volessimo leggere SP sul bus B, eseguirne l'incremento di 1 tramite la ALU, memorizzare il risultato in SP, avviare una lettura da memoria e indicare 122 come prossimo indirizzo di microistruzione, potremmo scrivere:

```
ReadRegister = SP, ALU = INC, WSP, Read, NEXT_ADDRESS = 122
```

Questo rappresenta già un buona semplificazione ma meglio utilizzare semplicemente:

```
SP = SP + 1; rd; goto 122
```

Come abbiamo detto più volte in ogni ciclo si può scrivere su più registri, la nostra notazione si estende semplicemente; esempio:

```
SP = MDR = SP + 1
```

D'altro canto un solo registro può essere portato sull'ALU tramite il bus B; l'altro input dell'ALU (canale sinistro) può essere +1, 0, -1 o il registro H. Copiare un registro sull'altro (es: `SP = MDR`) equivale semplicemente a utilizzare 0 sul canale sinistro dell'ALU.

Mic-1: Micro Assembly Language (2)

Per sommare due registri (uno dei due deve per forza essere H):

```
SP = MDR = SP + 1
```

ATTENZIONE a utilizzare solo istruzioni legali:

```
MDR = SP + MDR
H = H – MDR
```

sono illegali; **perché ?**

La tabella riporta le **operazioni legali** dove:

- SOURCE può essere MDR, PC, MBR, MBRU (versione di MBR unsigned) SP, LV, CPP, TOS o OPC.
- DEST può essere MAR, MDR, PC, SP, LV, CPP, TOS, OPC o H. Sono concessi assegnamenti multipli.

DEST = H
DEST = SOURCE
DEST = $\overline{H}$
DEST = $\overline{SOURCE}$
DEST = H + SOURCE
DEST = H + SOURCE + 1
DEST = H + 1
DEST = SOURCE + 1
DEST = SOURCE – H
DEST = SOURCE -1
DEST = – H
DEST = H AND SOURCE
DEST = H OR SOURCE
DEST = 0
DEST = 1
DEST = -1

Le operazioni indicate dalla tabella possono essere estese con il contributo dello shift register; ad esempio:

`H = MBR << 8` (copia in H il contenuto di MBR spostato a sinistra di 8 bit)

Mic-1: Micro Assembly Language (3)

Un accesso alla memoria in lettura/scrittura tramite MAR/MDR viene semplicemente indicato rispettivamente con:

```
rd e wr
```

Ricordiamo che il dato letto non è utilizzabile nel ciclo successivo.

La lettura di opcode di 1 byte dal flusso di istruzioni ISA (IJVM) tramite la porta PC/MBR viene indicato da

```
fetch
```

Il byte letto (come vedremo) può essere utilizzato per il salto al termine del ciclo successivo.

Ambedue le letture dalla memoria (dati e fetch) possono procedere in **parallelo** (!?!).

ATTENZIONE alla sequenza illegale:

```
MAR = SP; rd
MDR = H
```

entrambe assegnano un valore a MDR al termine del secondo ciclo !!!

Direttive di salto

Per convenzione ogni riga di MAL che non contiene un salto esplicito, viene implicitamente tradotta impostando NEXT_ADDRESS all'indirizzo della microistruzione della riga seguente.

Per eseguire un salto incondizionato è sufficiente indicare al termine della riga:

```
goto label
```

forzando in questo modo il valore di NEXT_ADDRESS.

Mic-1: Micro Assembly Language (4)

Per i salti condizionali occorre una notazione diversa; ricordiamo che JAMN e JAMZ utilizzano i bit di stato N e Z. Ad esempio, se volessimo saltare nel caso in cui TOS vale 0, dovremmo:

- far transitare TOS attraverso la ALU (in modo tale che Z venga impostato)
- impostare JAMZ a 1 per prevedere due possibili prossime microistruzioni.

Queste due operazioni, che possono essere svolte nello stesso ciclo, in notazione MAL vengono indicate con:

```
TOS = TOS; if (Z) goto L1; else goto L2
```

Per maggiore chiarezza `TOS = TOS` può essere sostituito dal più intuitivo `Z = TOS` visto che l'unica ragione del transito di TOS attraverso la ALU è l'impostazione di Z.

Ricordiamo che per come è definita la microarchitettura del Mic-1, i due indirizzi L1 ed L2 devono avere gli stessi 8 bit meno significativi, ovvero L2 = L1 + 256 !

Infine la notazione per l'uso del bit JMPC è:

```
goto (MBR OR valore)
```

che indica al microassemblatore di utilizzare *valore* come NEXT_ADDRESS e di impostare a 1 il bit JMPC in modo che il nuovo valore di MPC sia calcolato come OR di MBR e *valore*. Quando valore è 0 (come nella maggior parte dei casi) possiamo scrivere semplicemente:

```
goto (MBR)
```

Come abbiamo già detto quest'ultimo tipo di salto viene utilizzato prevalentemente per saltare all'inizio del microprogramma della prossima istruzione ISA (IJVM). Per poter eseguire il salto al termine del ciclo corrente il fetch deve essere stato **avviato** almeno 1 ciclo prima !

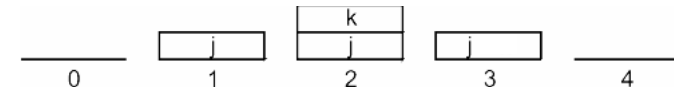
IJVM

Gestione della memoria e dello Stack

IJVM utilizza la memoria in un modo abbastanza particolare, e fa largo uso dello **stack**.

Lo **stack** è una parte della memoria dove i dati vengono impilati; i dati possono essere inseriti o prelevati solo dalla cima dello stack:

- Una scrittura (**Push**) sullo stack causa la crescita in altezza della pila di dati.
- Una lettura (**Pop**) dallo stack causa un accorciamento.

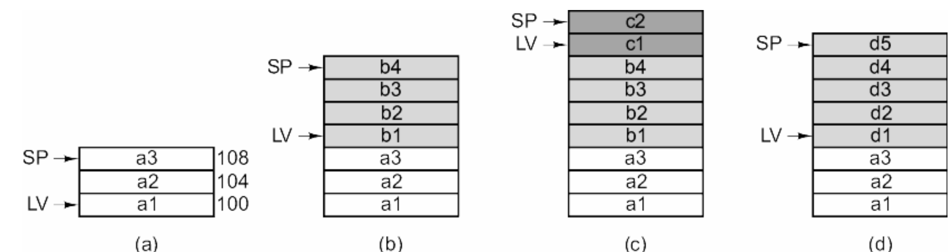


Nell'esempio di figura, vengono eseguite in sequenza le operazioni: Push j, Push k, Pop k, Pop j.

Lo stack è utilizzato dai linguaggi di programmazione per la gestione delle variabili locali delle procedure, per il passaggio di parametri durante la chiamata di procedure, per la valutazione di espressioni aritmetiche, ...

In IJVM lo stack è utilizzato per memorizzare variabili locali e per eseguire calcoli aritmetici. Vengono mantenuti due puntatori a indirizzi di memoria nello stack:

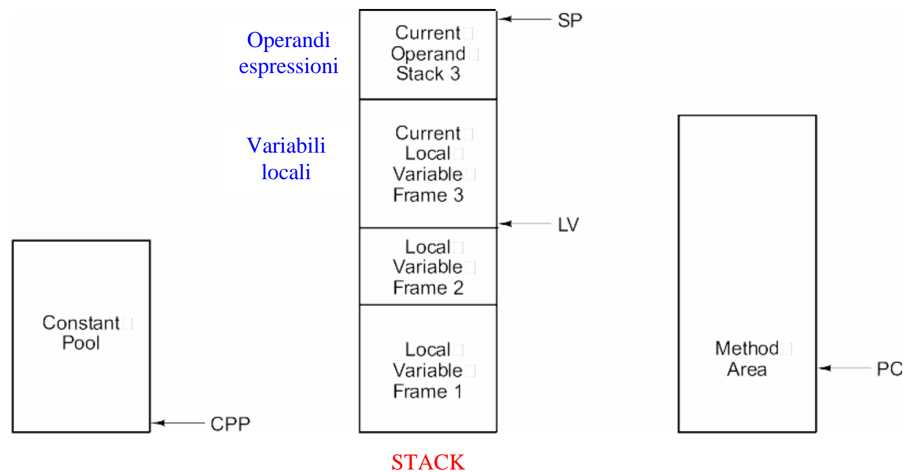
- un puntatore alla **base** attuale dello stack - **LV** (**L**ocal **V**ariable)
- un puntatore alla **cima** dello stack - **SP** (**S**tack **P**ointer).



IJVM (2)

Oltre allo stack, dove IJVM memorizza le variabili locali ed esegue operazioni aritmetiche, IJVM utilizza altre due regioni di memoria:

- **Constant Pool**: dove vengono memorizzate le costanti; la base di tale area è puntata dal registro **CPP**.
- **Method Area**: dove è memorizzato il codice (programmi) IJVM; i programmi Java prendono il nome di metodi (*come in generale in tutti i linguaggi di programmazione a oggetti*). Il puntatore all'istruzione corrente nella Method Area è **PC** (**P**rogram **C**ounter).



Fisicamente la memoria RAM è una sola; il fatto di considerarla composta da 3 regioni è una semplificazione.

IJVM (3)

Set di istruzioni

Il **set di istruzioni** IJVM è molto semplice e si compone di 20 istruzioni:

- 3 istruzioni per il la scrittura sulla cima allo stack di dati provenienti da sorgenti diverse (BIPUSH, ILOAD, LDCW)
- 1 istruzione per la lettura della parola sulla cima dello stack il salvataggio nell'area delle variabili locali (ISTORE)
- 3 istruzioni per la manipolazione dello stack: scambio parole, duplicazione parola, rimozione cima (SWAP, DUP, POP)
- 2 operazioni aritmetiche (IADD, ISUB) + 2 operazioni logiche (IAND, IOR)
- 1 istruzione di incremento del valore di una variabile locale (IINC)
- 4 istruzioni di salto: 1 incondizionata (GOTO), 3 condizionali (IFEQ, IFLT, IFICMPEQ)
- 1 istruzione per specificare ha un indice di 16 bit e non di 8 come di solito (WIDE)
- 2 istruzioni per la chiamata di metodi (INVOKEVIRTUAL, IRETURN)
- 1 istruzione nulla (NOP)

La **tabella seguente** elenca le istruzioni IJVM; la prima colonna contiene il codice (opcode) dell'istruzione, la seconda il codice mnemonico e gli eventuali operandi dell'istruzione; la terza colonna riporta una descrizione; le dimensioni degli operandi sono:

- 1 byte per `byte`, `const` e `varnum`
- 2 byte per `disp`, `index` e `offset`

IJVM (4)

Codice	Mnemonico	Significato
0x10	BIPUSH *byte*	Push del *byte* specificato sullo stack
0x15	ILOAD *varnum*	Push sullo stack della variabile il cui indirizzo è specificato dall'indice *varnum* (che è di 8 bit a meno che non sia stato utilizzato il prefisso WIDE)
0x13	LDCW *index*	Push della costante dalla constant area sullo stack. Index è l'indirizzo della costante nella costant area
0x36	ISTORE *varnum*	Pop di una parola dallo stack e memorizzazione come variabile locale, di cui *varnum* è l'indice.
0x5F	SWAP	Scambia le due parole in cima allo stack
0xA7	DUP	Duplica la prima parola sullo stack (push della copia)
0x57	POP	Cancella una parola dalla cima dello stack
0x60	IADD	Pop delle due parole dalla cima dello stack, somma e push del risultato
0x64	ISUB	Pop delle due parole dalla cima dello stack, sottrazione e push del risultato
0x7E	IAND	Pop delle due parole dalla cima dello stack, and logico e push del risultato
0x80	IOR	Pop delle due parole dalla cima dello stack, or logico e push del risultato
0x84	IINC *varnum const*	Somma la costante const alla variabile locale indirizzata da *varnum*
0xA7	GOTO *offset*	Salto non condizionato alla locazione corrente + offset
0x99	IFEQ *offset*	Pop di una parola e salto a locazione corrente + *offset* se la parola è 0
0x9B	IFTL *offset*	Pop di una parola e salto a locazione corrente + *offset* se la parola è negativa
0x9F	IFICMPEQ *offset*	Pop di due parole dallo stack e salto a locazione corrente + *offset* se sono uguali
0xC4	WIDE	Prefisso; l'istruzione ILOAD o ISTORE successiva ha un indice *varnum* a 16 bit che gli consente di indirizzare più di 256 variabili ...
0xB6	INVOKEVIRTUAL *disp*	Chiama un metodo; *disp* è l'offset della constant area dove reperire informazioni circa il metodo chiamato
0xAC	IRETURN	Ritorna da un metodo con un valore intero
0x00	NOP	Nessuna operazione

Compilazione Java → IJVM

Un esempio

Java	IJVM mnemonico	IJVM in versione esadecimale
i = j + k;	1 ILOAD j // i = j + k	0x15 0x02
if (i == 3)	2 ILOAD k	0x15 0x03
k = 0;	3 IADD	0x60
else	4 ISTORE i	0x36 0x01
j = j – 1;	5 ILOAD i // if (i < 3)	0x15 0x01
	6 BIPUSH 3	0x10 0x03
	7 IF_ICMPEQ L1	0x9F 0x00 0x0D
	8 ILOAD j // j = j – 1	0x15 0x02
	9 BIPUSH 1	0x10 0x01
	10 ISUB	0x64
	11 ISTORE j	0x36 0x02
	12 GOTO L2	0xA7 0x00 0x07
	13 L1: BIPUSH 0	// k = 0 0x10 0x00
	14 ISTORE k	0x36 0x03
	15 L2:	
(a)	(b)	(c)

IJVM su Mic-1

L'interprete IJVM per Mic-1 realizzato dal microprogramma MAL prevede come tutti gli interpreti un ciclo principale infinito che legge, decodifica ed esegue le istruzioni.

Il **ciclo principale** è costituito dalla sola riga **Main1** che:

- incrementa il Program Counter (`PC = PC +1`)
- inizia il fetch del prossimo byte (opcode successivo o operando istruzione corrente).
- salta all'indirizzo dell'istruzione presente in MBR. Si assume che quando ci si trova in Main1 l'opcode dell'istruzione sia già stato caricato in **MBR**; sarà dunque compito del microprogramma di ogni istruzione IJVC provvedere a ciò.

Etichetta	Microistruzione	Commenti
`Main1`	`PC = PC +1; fetch; goto (MBR)`	MBR contiene già l'opcode dell'istruzione corrente

Si assume inoltre che il registro **TOS** rimanga sempre aggiornato al contenuto della parola in cima allo stack. Ciò consentirà di risparmiare preziosi accessi alla memoria. Anche in questo caso ogni microprogramma di istruzione IJVC deve rispettare questa assunzione.

Gli indirizzi delle microistruzioni all'interno del control store non vengono qui riportati; al loro posto vengono utilizzate delle più pratiche etichette:

L'**etichetta** di ogni microistruzione è composta dal nome dell'istruzione IJVM cui il microprogramma si riferisce ed è affiancata da un intero crescente all'interno del microprogramma di quell'istruzione.

Le etichette vengono (automaticamente) tradotte in indirizzi dal microassemblatore ...

IJVM su Mic-1 (2)

Nel seguito sono riportati i frammenti di microprogramma relativi ad **alcune istruzioni IJVM**. Sul testo di riferimento (A.S. Tanenbaum, Architeture del Computer, 4ª edizione, UTET 2000) si può trovare l'intero microprogramma dell'interprete IJVM.

NOP

Etichetta	Microistruzione	Commenti
`Nop1`	`goto Main1`	Nop non esegue nessuna istruzione; è sufficiente saltare al ciclo principale

IADD

Etichetta	Microistruzione	Commenti
`iadd1`	`MAR = SP = SP -1; rd`	La prima delle 2 parole è già in TOS; avvia la lettura della seconda che si trova a SP -1
`iadd2`	`H = TOS`	Copia in H la prima parola, la seconda sarà in MDR al termine di questo ciclo
`iadd3`	`MDR = TOS = MDR + H; wr; goto Main1`	MDR viene aggiornato con la somma e si avvia la scrittura su SP -1; il multiassegnamento consente di mantenere aggiornato anche TOS

ISUB, IAND, IOR

I microprogrammi sono praticamente identici a IADD, eccezioni fatta per l'operazione eseguita alla terza microistruzione che invece di una somma è una sottrazione, AND o OR rispettivamente.

IJVM su Mic-1 (3)

DUP

Etichetta	Microistruzione	Commenti
dup1	`MAR = SP = SP + 1`	incrementa SP e prepara MAR per la scrittura
dup2	`MDR = TOS; wr; goto Main1`	Prepara MDR per la scrittura a SP +1; avvia la scrittura e salta a Main1

POP

Etichetta	Microistruzione	Commenti
pop1	`MAR = SP = SP - 1; rd`	legge la parola a SP-1 per il semplice motivo di dover tenere aggiornato TOS
pop2		deve attendere un ciclo per la lettura senza fare nulla !!!
pop3	`TOS = MDR; goto Main1`	può finalmente assegnare TOS e tornare al Main1

SWAP

Etichetta	Microistruzione	Commenti
swap1	`MAR = SP - 1; rd`	legge la parola a SP-1
swap2	`MAR = SP`	alla fine di questo ciclo ha [SP-1] in MDR
swap3	`H = MDR; wr`	salva in H temporaneamente il valore [SP-1] che al termine dell'operazione dovrà essere il nuovo valore di TOS; scrive [SP-1] a SP
swap4	`MDR = TOS`	Imposta MDR al valore TOS = [SP]
swap5	`MAR = SP -1; wr`	Scrive [SP] a SP-1
swap6	`TOS = H; goto Main1`	Aggiorna TOS col valore salvato temp. in H

IJVM su Mic-1 (4)

BIPUSH *byte*

Etichetta	Microistruzione	Commenti
bipush1	`SP = MAR = SP + 1`	lo stack deve crescere inserendo il nuovo byte
bipush2	`PC = PC +1; fetch`	il byte operando per questa istruzione è gia stato pre-caricato da Main1; devo comunque eseguire il fetch per caricare in MBR l'opcode successivo.
bipush3	`MDR = TOS = MBR; wr; goto Main1`	MBR viene esteso con segno a lunghezza parola che viene scritta a SP

ILOAD *varnum*

Etichetta	Microistruzione	Commenti
iload1	`H = LV`	LV è la base delle variabili locali
iload2	`MAR = MBRU + H; rd`	l'indirizzo a cui prelevare la variabile è LV + *varnum* (il cui valore è già in MBR). *varnum* deve essere considerato unsigned (pertanto utilizzo MBRU)
iload3	`MAR = SP = SP +1`	la variabile sarà salvata a SP +1
iload4	`PC = PC +1; fetch; wr`	esegue il fetch del prossimo opcode; avvia la scrittura nello stack in quanto MDR ora è disponibile con il valore della variabile
iload5	`TOS = MDR; goto Main1`	aggiorna TOS e torna al Main

ISTORE *varnum*

Operazione inversa di ILOAD.

IJVM su Mic-1 (5)

WIDE ILOAD *varnum*, WIDE ISTORE *varnum*

Nel caso in cui il prefisso WIDE (microistruzione) preceda ILOAD o ISTORE, l'operando *varnum* che segue l'opcode è di 2 byte (può indirizzare 65536 variabili locali).

LDCW *offset*

Etichetta	Microistruzione	Commenti
ldcw1	PC = PC +1; fetch	*offest* è di 2 byte; il primo è già contenuto in MBR; il secondo deve essere letto
ldcw2	H = MBRU << 8	l'indirizzo a cui prelevare la variabile è CPP + *offset* (che va inteso come valore a 16 bit, in formato BIG-ENDIAN)
ldcw3	H = MBRU OR H	ricompone *offset*
ldcw4	MAR = H + CCP; rd; goto iload3	prepara l'indirizzo per la lettura; prosegue da iload3; il quale ritornerà a Main1

GOTO *offset*

Etichetta	Microistruzione	Commenti
goto1	OPC = PC -1	L'indirizzo di salto è PC + *offest*; Il valore di PC è già stato incrementato da Main1 (quindi lo diminuisco di 1)
goto2	PC = PC +1; fetch	carico il secondo byte dell'*offest*; *offest* è di 2 byte e deve essere considerato signed (salti possibili da PC - 32768 a PC + 32767); il primo byte è già contenuto in MBR
goto3	H = MBR << 8	il primo byte (con segno) è scritto in H
goto4	H = MBRU OR H	il secondo byte (senza segno) è ora stato caricato e può essere messo negli 8 bit meno significativi di H
goto5	PC = OPC + H; fetch	Aggiorna PC ed esegue il fetch anticipato
goto6	goto Main1	Attende il fetch e salta a Main1

IJVM su Mic-1 (6)

IFLT *offset*

Etichetta	Microistruzione	Commenti
iflt1	MAR = SP = SP -1; rd	il salto avviene in base alla parola in cima allo stack, di cui devo in ogni caso fare il POP
iflt2	OPC = TOS	salva temporaneamente TOS in OPC
iflt3	TOS = MDR	Mette la nuova cima dello stack in TOS
iflt4	N = OPC; if (N) goto T; else goto F	OPC che contiene la testa dello stack setta il bit N (negative); se N è settato salta a T altrimenti a F

Etichetta	Microistruzione	Commenti
T	OPC = PC -1; fetch; goto goto2	uguale a goto1; deve fare il fetch del secondo byte di offset

Etichetta	Microistruzione	Commenti
F	PC = PC +1	salta il secondo byte di offset che non occorre
F2	PC = PC +1; fetch;	pre-carica il prossimo opcode
F3	goto Main1	torna a Main1

IFLQ *offset*, IFICMPEQ *offset*

Sono molto simili a IFLT.

Progetto di microarchitetture

Riduzione del path di esecuzione

In genere la riduzione del path di esecuzione di una o più istruzioni può essere ottenuta a discapito della semplicità della micro-architettura, aggiungendo:

- **registri** interni
- **bus** interni
- **unità funzionali** dedicate a compiti particolari (esempio fetch)

Talvolta (raramente) la semplice **ri-ottimizzazione del microcodice** consente alcuni miglioramenti. Consideriamo ad esempio l'istruzione POP nella microarchitettura Mic-1:

Etichetta	Microistruzione	Commenti
pop1	MAR = SP = SP - 1; rd	legge la parola a SP-1 per il semplice motivo di dover tenere aggiornato TOS
pop2		deve attendere un ciclo per la lettura senza fare nulla !!!
pop3	TOS = MDR; goto Main1	può finalmente assegnare TOS e tornare al Main1

durante il secondo ciclo il data path è inattivo; si potrebbe dunque anticipare ciò che verrà eseguito dopo l'istruzione POP, ovvero l'incremento del PC e il fetch eseguito da Main1 (con un aumento di prestazioni di 1 ciclo su 4, che considerando che POP è un'istruzione molto comune non è trascurabile):

Etichetta	Microistruzione	Commenti
pop1	MAR = SP = SP - 1; rd	legge la parola a SP-1 per il semplice motivo di dover tenere aggiornato TOS
pop2	PC = PC +1; fetch;	Anticipa incremento PC e fetch
pop3	TOS = MDR; goto (MBR)	Invece che a Main1 salta direttamente alla prossima istruzione

Progetto di microarchitetture

Riduzione del path di esecuzione (2)

Sempre analizzando i microprogrammi di Mic-1 ci rendiamo conto che molti cicli sono sprecati a causa dell'impossibilità di utilizzare l'input sinistro della ALU con un registro diverso da H.

Un'architettura a 3 bus (vedi lucido seguente) dove tutti i registri possono essere abilitati in scrittura anche sul bus A (input sinistro dell'ALU) è in grado di migliorare significativamente le prestazioni.

Consideriamo ad esempio il microprogramma dell'istruzione ILOAD:

Etichetta	Microistruzione	Commenti
iload1	H = LV	LV deve per forza transitare in H per poter essere sommato a MBRU e scritto in MAR
iload2	MAR = MBRU + H; rd	l'indirizzo a cui prelevare la variabile è LV + *varnum* (il cui valore è già in MBR). *varnum* deve essere considerato unsigned (pertanto utilizzo MBRU)
iload3	MAR = SP = SP +1	la variabile sarà salvata a SP +1
iload4	PC = PC +1; fetch; wr	esegue il fetch del prossimo opcode; avvia la scrittura nello stack in quanto MDR ora è disponibile con il valore della variabile
iload5	TOS = MDR; goto Main1	aggiorna TOS e torna al Main

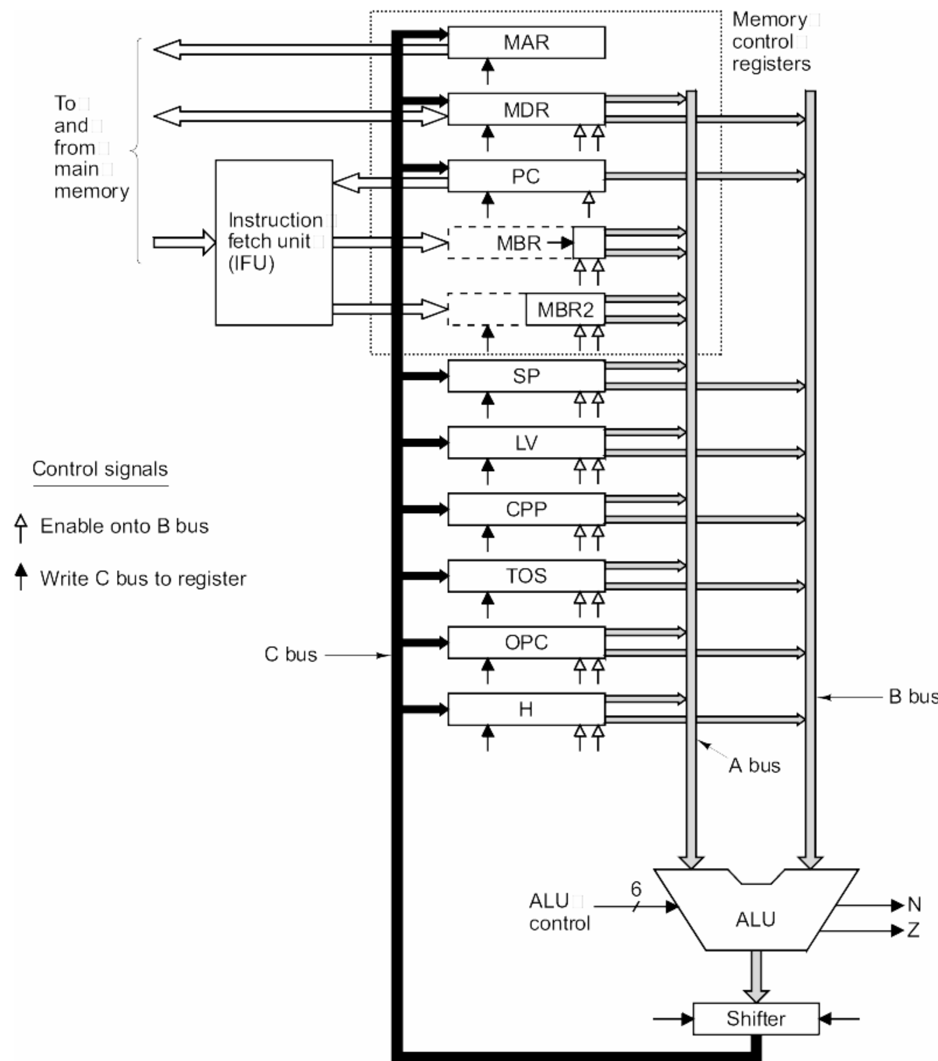
i cicli iload1 e iload2 possono essere sostituiti dall'unico ciclo:

MAR = MBRU + LV; rd

riducendo di 1 la lunghezza del path di esecuzione; lo stesso tipo di miglioramento può essere ottenuto per molte altre istruzioni che utilizzano H come registro "temporaneo"

Progetto di microarchitetture
Riduzione del path di esecuzione (3)

Un'architettura con **3 bus** e un'**unità** indipendente per il **fecth**:



Progetto di microarchitetture
Riduzione del path di esecuzione (4)

L'architettura a 3 bus del lucido precedente incorpora anche un'**unità indipendente per il fetch** delle istruzioni e degli operandi (**IFU**, **I**nstruction **F**etch **U**nit). Dall'analisi dei microprogrammi di Mic-1 appare infatti evidente che gran parte dei cicli del data path sono sprecati per:

- il fetch degli opcode e incremento del PC (Main1)
- fetch degli operandi e nel caso di operandi a 16 bit, nella ricostruzione dell'intero a 16 bit composto da 2 byte letti sequenzialmente.

L'unità **IFU**, parallelamente al normale funzionamento del data path, esegue:

- il fetch degli opcode e l'incremento di PC (per l'incremento non è necessario un sommatore completo ma un circuito "incrementatore" che è molto più semplice).
- il fetch degli operandi.

Come può l'IFU sapere se l'istruzione corrente ha operandi e in caso affermativo se sono di 1 o 2 byte ?

Vi sono almeno 2 modi:

1. l'IFU decodifica l'istruzione in modo da capire quanti extra byte leggere.
2. l'IFU legge sempre e comunque 2 extra byte (il massimo); tali byte verranno usati solo se necessario.

   Nello schema architetturale precedente è stata scelta questa seconda alternativa; MBR è stato affiancato da un nuovo registro MBR2 a 16 bit che viene utilizzato per il caricamento di operandi a 16 bit.

Nella nuova architettura molte delle istruzioni importanti possono essere eseguite con un notevole **riduzione di cicli**.