**Esercizio 8**

A partire dall’implementazione fornita di un processore operante secondo il modello IJVM,

1. si proceda all’analisi dell’architettura mediante simulazione e si approfondisca lo studio del suo funzionamento per due istruzioni a scelta,
2. si modifichi un codice operativo a scelta, documentando tutte le modifiche effettuate,
3. (opzionale) si descriva il funzionamento del processore in merito alle istruzioni di input/output,
4. (solo ove possibile) si sintetizzi il processore su FPGA.

L’IJVM:

E’ un linguaggio di livello macchina, semplificazione dell’Instruction Set Architecture JVM. La semplificazione consiste nell’utilizzo di sole operazioni su interi. L’IJVM organizza la memoria suddividendola in: (wiki)

-Constant Pool: area che non permette la scrittura da parte del programma IJVM. I dati qui caricati (costanti, puntatori e stringhe), possono essere scritti solamente quando il programma è portato in memoria ed è accessibile tramite il registro Constant Pool Pointer;

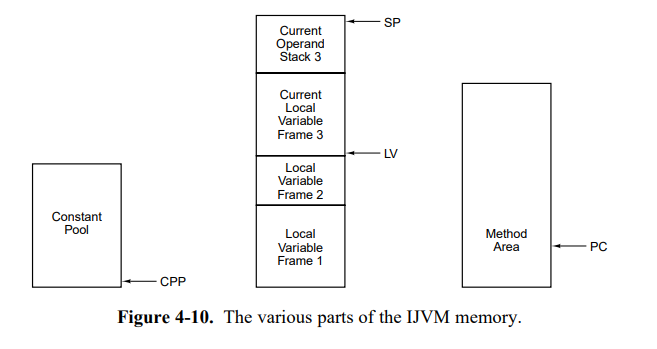
-Local Variable Frame: area dove vengono inserite le variabili locali al programma ed i parametri relativi ad esso. L’ indirizzo relativo all’inizio di quest’area è gestito tramite il registro LV;

-Stack degli operandi: è localizzato al di sopra del Local Variable Frame e permette di gestire gli operandi durante un’operazione. E’ accessibile tramite il registro SP, il quale punta sempre all’ultima locazione inserita nello stack;

-Method Area: area in cui risiedono i programmi IJVM da eseguire. Quest’area della memoria viene gestita come un array di byte, mentre tutte le precedenti sono gestite come array di parole da 4 byte.

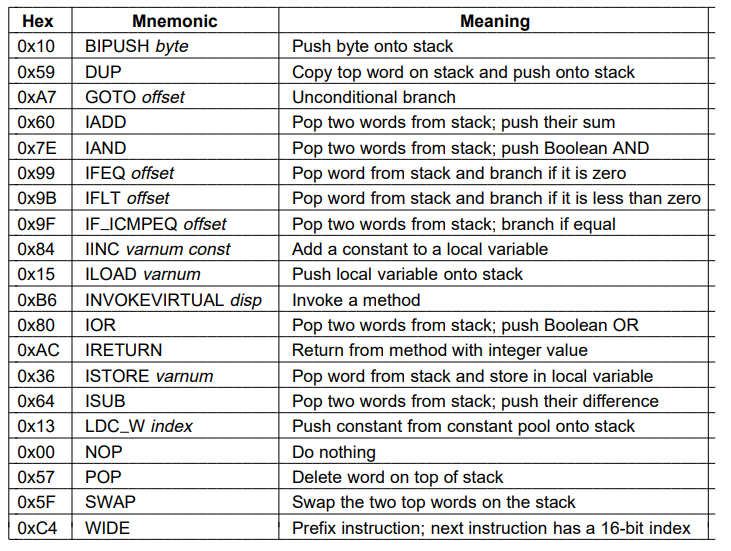
Per i registri LV,SP e CPP, indicare un offset equivale quindi ad indicare uno spiazzamento di 32 bit, mentre per il PC equivale ad indicare uno spiazzamento di 8 bit.

(Immagine memoria IJVM)



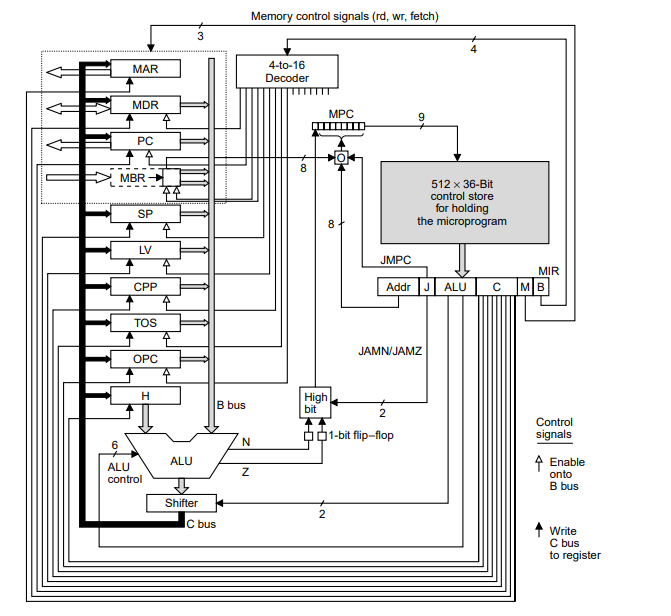
Le istruzioni IJVM sono a lunghezza variabile in base alla presenza di operandi o altri parametri; in generale possono occupare da 8 bit a 24 bit e vengono dunque memorizzate nella Method Area in locazioni contigue. Il primo byte di ogni istruzione è il Codice Operativo, ossia un indirizzo che verrà utilizzato per mappare l’istruzione IJVM con l’indirizzo start del microprogramma contenuto nella control\_store del processore. Esempi di istruzioni IJVM:

(elenco istruzioni IJVM)



Il processore MIC-1:

(Blocco completo img)

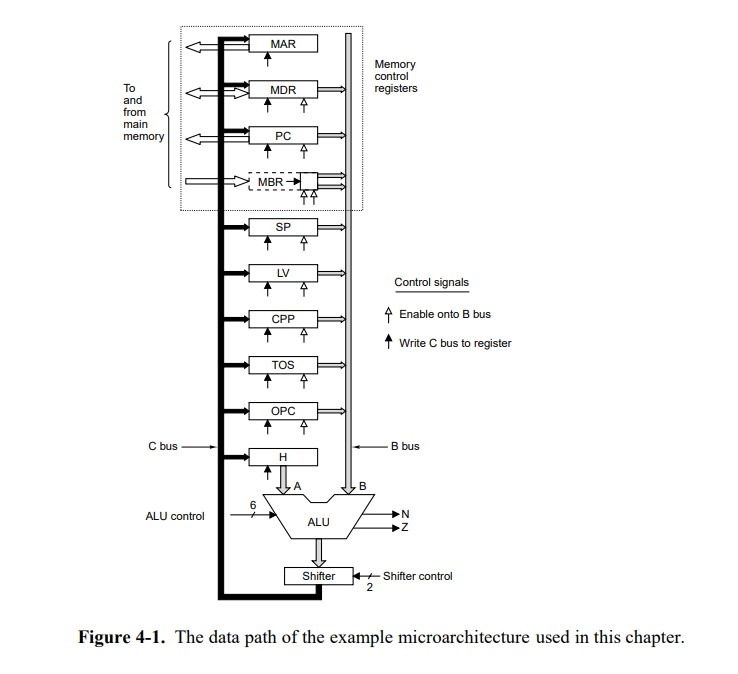


E’ una macchina a stack, ossia che non utilizza registri general purpose (ad esempio di tipo indirizzo o dato), ma le istruzioni presentano operandi contenuti in uno stack allocato nella memoria principale. Il processore MIC-1 è organizzato in parte operativa e parte di controllo.

Parte operativa:

La parte operativa realizza il datapath dell’architettura ed è composto da registri a 32 bit, 3 bus, ALU e shifter, come mostrato in figura:

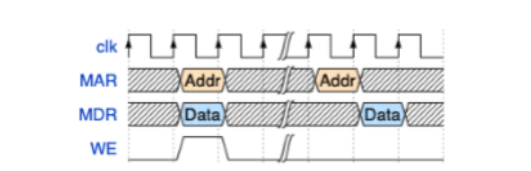
(immagine datapath).



Comunicazione in memoria:

La comunicazione in memoria differisce in tempificazione se si tratta di una read o di una write. In particolare nel caso della write, nello stesso periodo di clock avviene sia l’acquisizione sulle linee dell’indirizzo che del dato da scrivere in memoria, mentre nel caso della read è necessario un colpo di clk in più per la presentazione del dato in uscita alla memoria.

(img tempificazione mem)



I registri utilizzati per la comunicazione in memoria si suddividono in coppie:

-MAR-MDR: Quest’interfaccia è relativa alle aree di memoria accessibili in parole di 4 byte (Constant Pool,Local Variable, Stack). Questa consente di specificare l’indirizzo in memoria in MAR a partire dal quale leggere o scrivere 4 byte consecutivi nel MDR.

-PC-MBR Quest’interfaccia è relativa all’ aree di memoria in cui è contenuto il programma, la quale è accessibile in parole da 1 byte (Method Area). Questa consente di specificare l’indirizzo in memoria in PC a partire dal quale leggere 1 byte ponendolo nel MBR. Ciò significa che se un’istruzione prevede di specificare sia il codice operativo, che un operando, allora verrà prelevato prima il codice operativo e poi l’operando; le microprocedure sono realizzate tenendo conto proprio di questa modalità di comunicazione della memoria dell’interfaccia PC-MBR.

Bus:

L’unità operativa dispone di tre bus con parallelismo a 32 bit:

-A: per utilizzare un secondo operando ed effettuare le operazioni in ALU;

-B: è utilizzato per effettuare la lettura dai registri;

-C: utilizzato per la scrittura dei registri;

ALU:

L’ALU ha due ingressi: A,collegato direttamente al registro tampone H, e B collegato al bus B. Per poter definire l’operazione da effettuare a partire dagli operandi si utilizza una stringa di segnali di controllo di 6 bit. Inoltre, quando il risultato passa nello shifter, vi sono ulteriori 2 segnali che permettono di effettuare lo shift di quest’ultimo, prima che venga posto sul bus C.

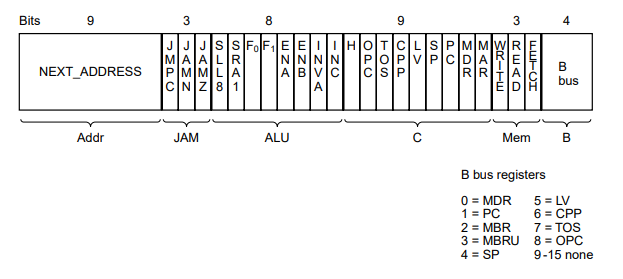
Parte di controllo:

E’ realizzata tramite una memoria di sola lettura, detta control store, un microPC ed un microIR. La control store memorizza tutti i microprogrammi relativi alle istruzioni del set IJVM, allocandoli in memoria all’indirizzo corrispondente al codice operativo dell’istruzione.

Ad ogni ciclo di clk, viene continuamente letta dalla control store la microistruzione all’indirizzo puntato dal mPC.

Il formato delle microistruzioni è dato dalla seguente figura:

(immagine microistruzioni)



Ogni microistruzione è legata alla successiva tramite i bit di NEXT\_ADDRESS e i bit di salto JAM. È necessario effettuare questo concatenamento delle microistruzioni poiché, quelle relative allo stesso microprogramma, potrebbero non essere salvate in locazioni contigue della control\_store (si pensi a POP e DUP). Oltre a questo concatenamento è possibile che ci sia la presenza dei salti in base al tipo di microistruzione (ad esempio quando vengono utilizzati i costrutti if, else e goto).

In particolare, se JAM = “000”, allora mPC = NEXT\_ADDRESS, altrimenti, se almeno uno dei flag di salto è alto, occorre calcolare il nuovo mPC:

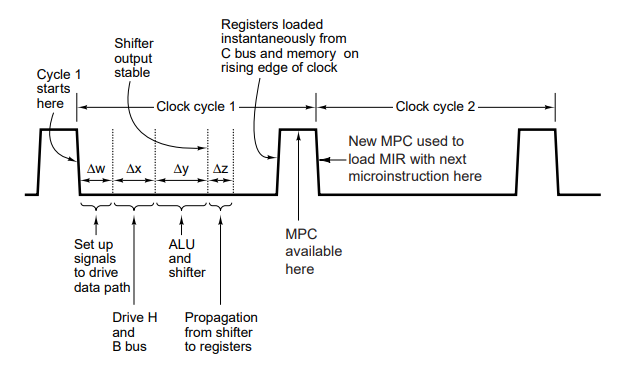
-- JAMN = 1: il bit più significativo di mPC viene messo in OR con il flag N dell’ALU (risultato negativo in uscita).

-- JAMZ = 1: il bit più significativo viene messo in OR con il flag Z dell’ALU (risultato nullo in uscita).

-- JAMC = 1: gli 8 bit meno significativi di NEXT\_ADDRESS sono messi in OR con MBR. In questo modo ottengo un salto di un valore pari a quello contenuto in MBR.

Per caricare la prossima microistruzione è necessario che trascorra un certo intervallo di tempo, necessario al caricamento del nuovo mPC. Il ciclo di clock inizia sul fronte di discesa in cui è disponibile il nuovo valore del mPC, il quale è stato caricato sul fronte di salita dello stesso impulso di clock. E’ necessario scegliere una frequenza adeguata del clock, in modo tale che tutti i ritardi dovuti ai vari componenti del datapath siano contenuti e quindi i segnali sui bus si stabiliscano.

(Immagine tempificazione clokc)



Flusso di esecuzione delle istruzioni:

Il processore MIC-1 dispone del set di istruzioni relative all’ Integer Java Virtual Machine e sfrutta la logica microprogrammata, ovvero realizza ogni istruzione tramite un microprogramma.

Per innescare l’esecuzione di un programma assembler è necessario effettuare prima la traduzione da assembler a microassembler e poi da microassembler a parole di controllo.

A partire dal set di istruzioni IJVM, l’assemblatore Java effettua la corrispondenza tra le istruzioni ed i microprogrammi, i quali verranno tradotti dal microassemblatore MAL in sequenze di segnali di controllo rispetto all’architettura MIC-1. Tutte le sequenze generate vengono inserite all’interno della control\_store del processore.

Una volta scritto il programma in AJVM, tutte le istruzioni IJVM sono tradotte in sequenze di bit che contengono il codice operativo dell'istruzione, ovvero l’indirizzo start del microprogramma contenuto nella control\_store del processore, ed eventuali operandi. Il programma tradotto viene poi inserito nella memoria del sistema.

Flusso di esecuzione di esempio:

.main

.var

a

.endvar

BIPUSH 0xA

BIPUSH 0xE

IADD

ISTORE a

HALT // HALT is a no operand instruction but translates as a GOTO (short value)

.endmethod

1 Per ogni programma vengono eseguite le microprocedure seguenti:

1.1 viene prima eseguita la routine mic1\_entry, la quale inizializza i registri del datapath ed inizializza la memoria locale del programma.

1.2 Viene inizializzato il valore a cui punta LV, il quale contiene l'indirizzo di memoria da cui deve partire lo stack pointer, ossia dopo l'allocazione di memoria per le variabili locali.

1.3 viene poi eseguita la procedura main, ed inizia l'effettivo programma.

2 Flusso di esecuzione BIPUSH

4 si effettua il push dell'operando 0xA sullo stack. L'op. bipush prevede:

1 Lo spostamento del puntatore SP una cella di memoria in avanti. Il nuovo valore di SP viene poi assegnato al registro MAR;

2 Fetch della 2a parte dell'istruzione (operando 0xA);

3 Pone in MDR il valore estratto 0xA ed effettua la scrittura in memoria all'indirizzo MAR, ovvero in cima allo stack;

4 go to main

5 si effettua il push dell'operando 0xE sullo stack. L'op. bipush è analoga.

3 Flusso di esecuzione IADD

6 si effettua l'operazione di IADD, la quale prende gli ultimi due valori nello stack (ossia quello in cima allo stack e quello immediatamente precedente) ed effettuare la somma tra i due, ponendola nello stack alla posizione del primo operando.

1 Decremento lo stack pointer per puntare al valore 0xA, lo assegno a MAR ed effettuo la lettura (in MDR avrò 0xA)

2 Il registro tampone viene assegnato con il valore in cima allo stack (H = TOS);

3 Effettuo la somma tra H e MDR e la pongo in MDR. Tale somma è assegnata al registro TOS dato che la cima dello stack è cambiata. Infine viene effettuata la write del risultato sullo stack alla posizione puntata dal registro MAR.

4 go to main

4 Flusso di esecuzione ISTORE

7 si effettua l'operazione di ISTORE, la quale memorizza nella memoria locale del programma il valore dell'operazione.

* 1 H = LV
* 2 In MAR viene inserito l’indirizzo della locazione di memoria nella quale sarà effettuata la store. Tale locazione è ricavata dalla somma tra MBRU (unsigned perchè MBR è un registro a 8 bit ed è necessario che abbia spiazzamento a 32 bit, in modo da essere un indirizzo di memoria) ed il registro H, che contiene il valore di LV.
* 3 Si inserisce in MDR il valore che si vuole scrivere in memoria locale, ossia quello della cima dello stack, contenente la somma, e si effettua la scrittura all'indirizzo puntato da MAR.
* 4 Si decrementa lo stack pointer effettuando una pop della somma, si legge il nuovo valore da inserire nel TOS e lo si pone in MDR.
* 5 Si effettua la fetch per spostarsi all'istruzione successiva, poichè l'istruzione ISTORE necessità di due fetch per poter essere eseguita completamente: una per saltare all'operando ed una per andare all'istruzione successiva (questa viene fatta nel main).
* 6 go to main

Analisi delle istruzioni IADD e ISTORE:

IADD

iadd = 0x65:

MAR = SP = SP - 1; rd

H = TOS

MDR = TOS = MDR + H; wr; goto main

La traduzione in segnali di controllo, rispetto al formato delle microistruzioni descritto precedentemente, è la seguente:

Microistruzione 1 MAR = SP = SP - 1; rd

next\_address: 102

jmp: 000

alu: 00110110

bus C: 000001001

mem: 010

bus B: 0100

Microistruzione 2 H = TOS

next\_address: 103

jmp: 000

alu: 00010100

bus C: 100000000

mem: 000

bus B: 0111

Microistruzione 3 MDR = TOS = MDR + H; wr; goto main

next\_address: 6

jmp: 000

alu: 00111100

bus C: 001000010

mem: 100

bus B: 0000

SIMULAZIONE IADD

Programma ajvm associato:

.main

.var

a

.endvar

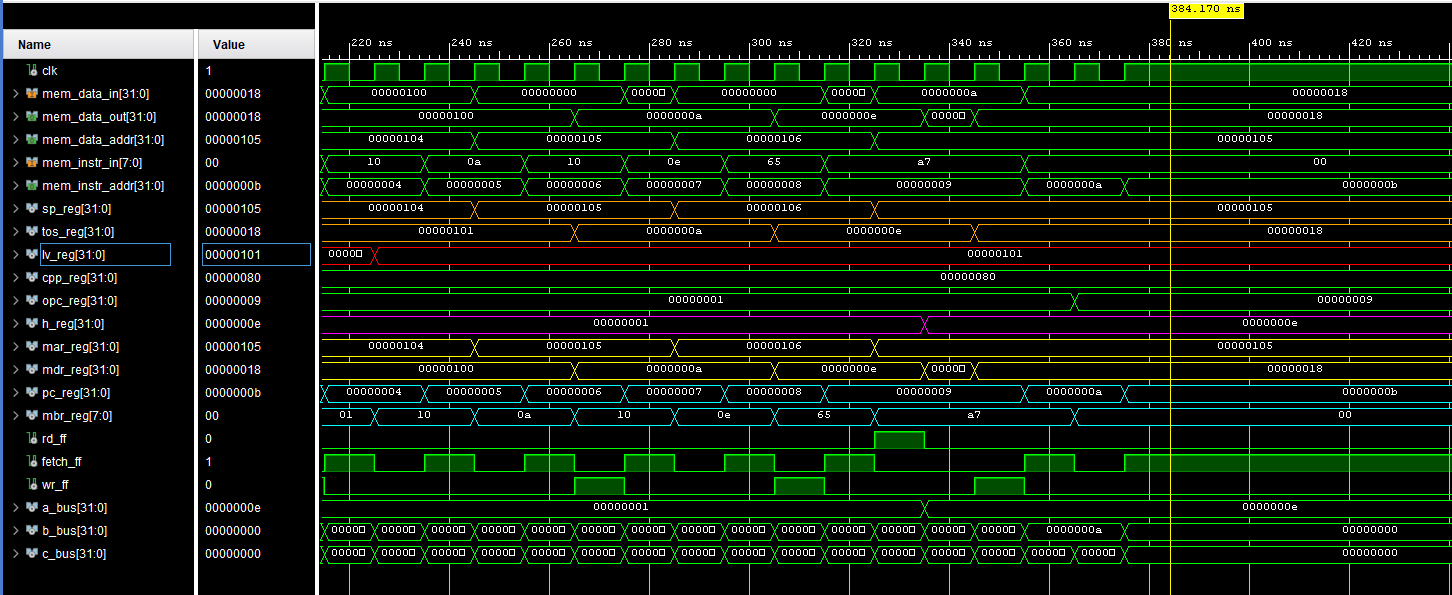
BIPUSH 0xA

BIPUSH 0xE

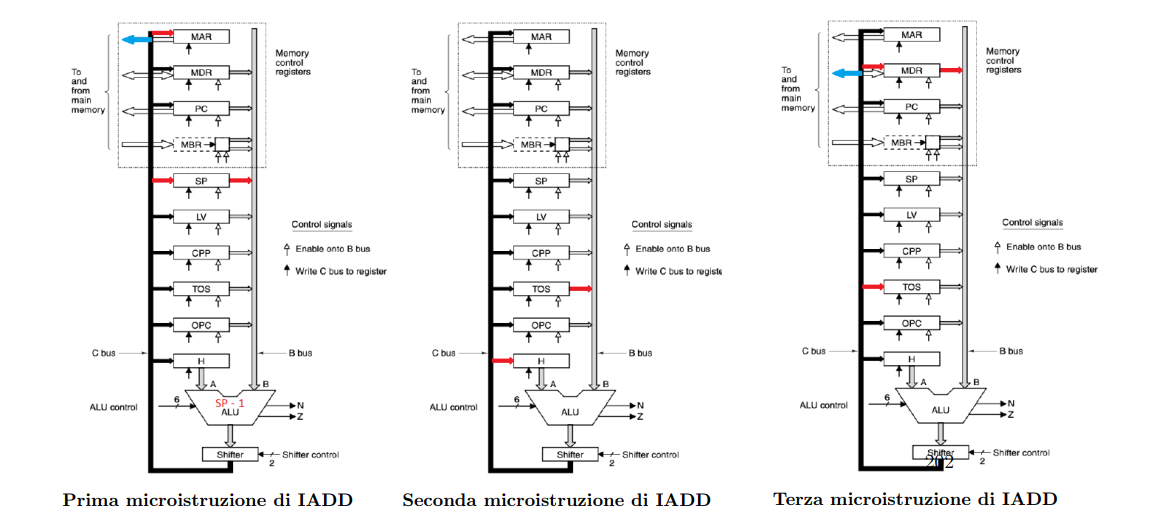
IADD

HALT // HALT is a no operand instruction but translates as a GOTO (short value)

.endmethod



Dalla simulazione si nota che all’esecuzione dell’istruzione IADD (0x36) vengono estratti: 0xA, decrementando lo stack pointer a 0x105, 0xE che viene mantenuto nel registro TOS e viene poi effettuata la somma tra 0xA e 0xE che produce in uscita 0x18, che verrà scritto sulla cima dello stack all’indirizzo 0x105, con un ritardo dovuto al posizionamento nei registri dei valori degli operandi e dalla scrittura in memoria. La simulazione termina appena PC raggiunge il valore 0xB.

(Iadd flusso di datapath)

ISTORE

istore = 0x36:

H = LV

MAR = MBRU + H

istore\_cont:

MDR = TOS; wr

SP = MAR = SP - 1; rd

PC = PC + 1; fetch

TOS = MDR; goto main

La traduzione in segnali di controllo, rispetto al formato delle microistruzioni descritto precedentemente, è la seguente:

ISTORE

Microistruzione 1 H = LV

next\_address: 55

jmp: 000

alu: 00010100

bus C: 100000000

mem: 000

bus B: 0101

Microistruzione 2 MAR = MBRU + H

next\_address: 56

jmp: 000

alu: 00111100

bus C: 000000001

mem: 000

bus B: 0011

Microistruzione 3 MDR = TOS; wr

next\_address: 57

jmp: 000

alu: 00010100

bus C: 000000010

mem: 100

bus B: 0111

Microistruzione 4 SP = MAR = SP - 1; rd

next\_address: 58

jmp: 000

alu: 00110110

bus C: 000001001

mem: 010

bus B: 0100

Microistruzione 5 PC = PC + 1; fetch

next\_address: 59

jmp: 000

alu: 00110101

bus C: 000000100

mem: 001

bus B: 0001

Microistruzione 6 TOS = MDR; goto main

next\_address: 6

jmp: 000

alu: 00010100

bus C: 001000000

mem: 000

bus B: 0000

SIMULAZIONE ISTORE

Programma ajvm associato

.main

.var

a

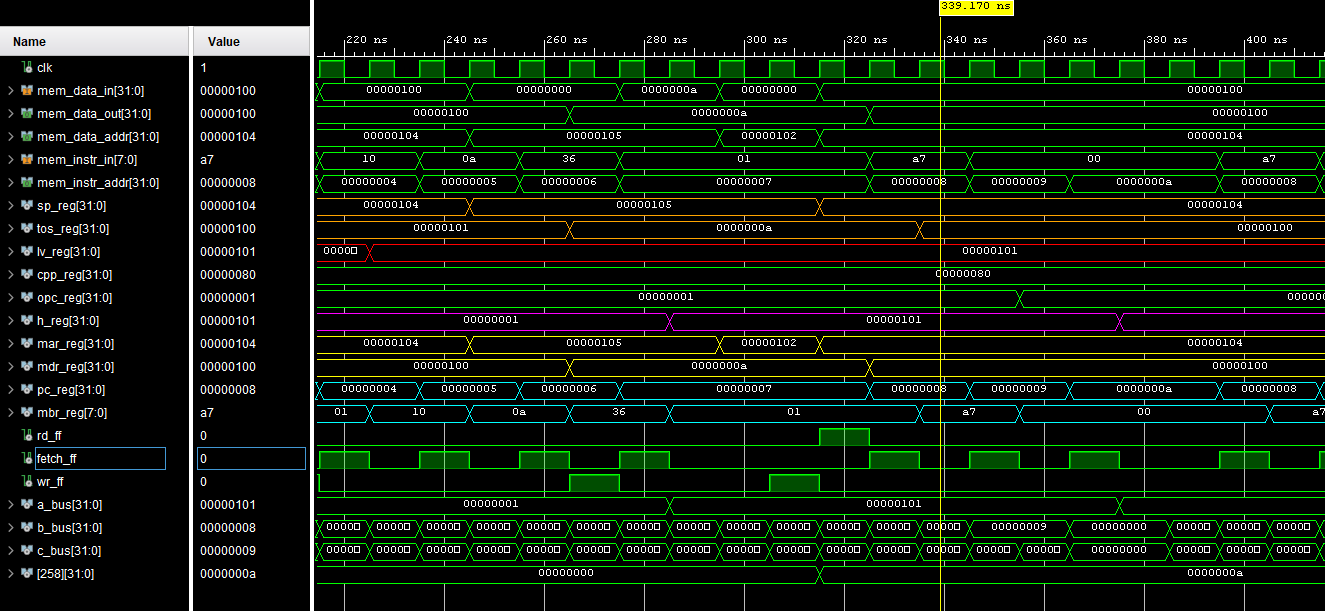
.endvar

BIPUSH 0xA

ISTORE a

HALT // HALT is a no operand instruction but translates as a GOTO (short value)

.endmethod



Dalla simulazione si nota che:

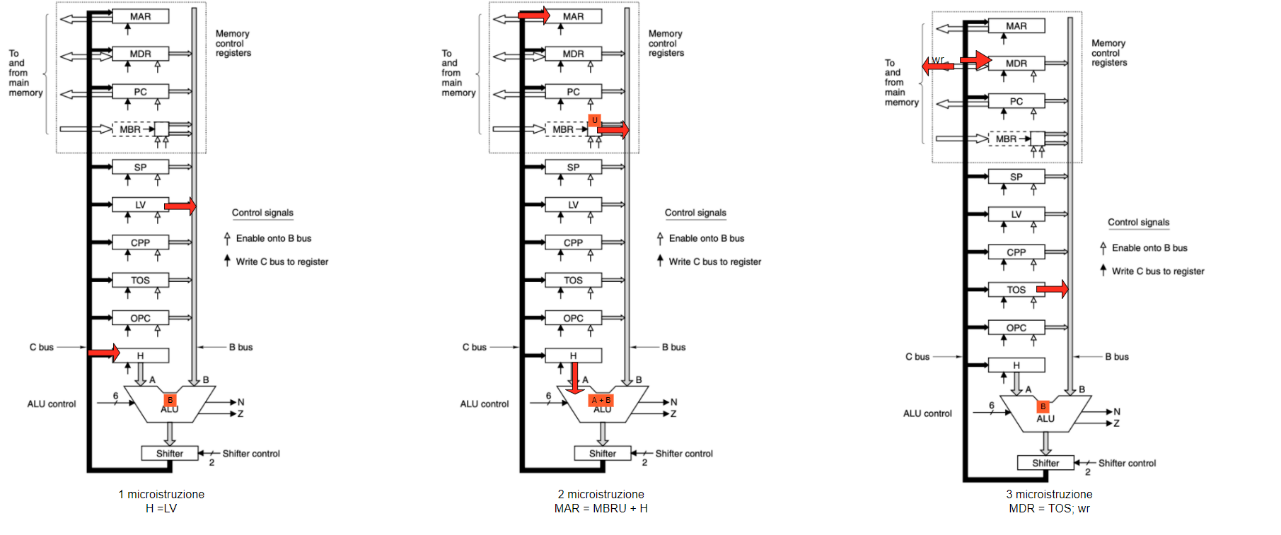
viene caricato il registro H con il valore di LV, ossia 0x101;

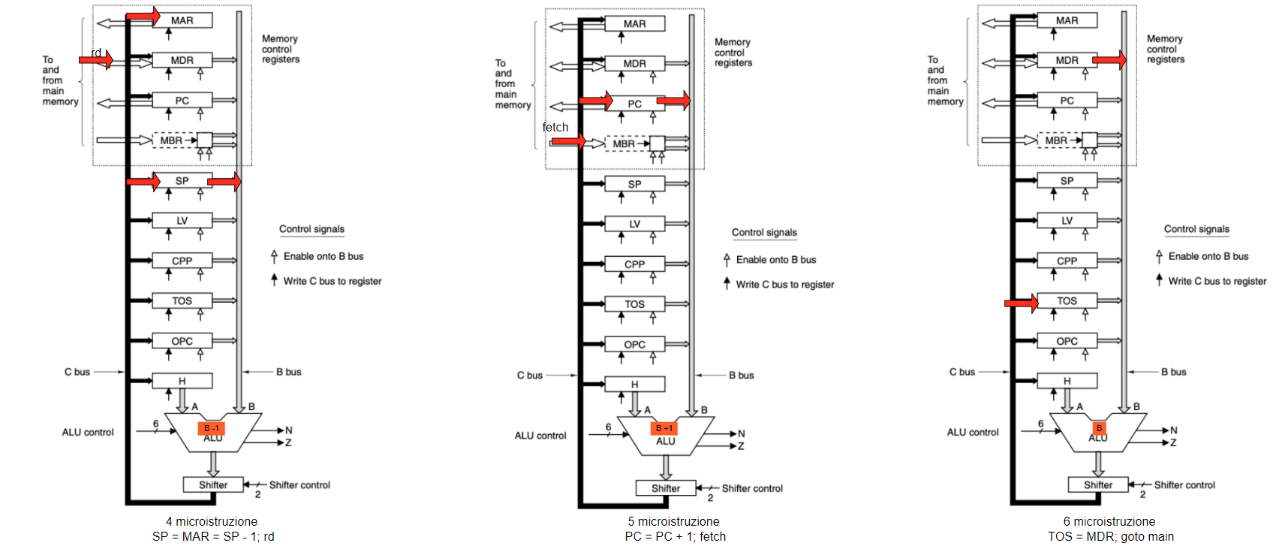
all’indirizzo base LV viene aggiunto l’offset di 0x01 per puntare alla variabile A nel Local Variable Frame e viene inserito nel MAR ottenendo 0x102;

si effettua poi la write in memoria all’indirizzo 0x102 del valore in TOS, ossia 0xA;

si decrementa poi lo stack pointer a 0x104 e si riaggiornano i registri.

(Istore flusso del datapath)





Modifica dell’istruzione IADD:

E’ stato modificato il microprogramma relativo all’istruzione IADD in questo modo:

iadd = 0x65:

MAR = SP = SP - 1; rd

H = TOS

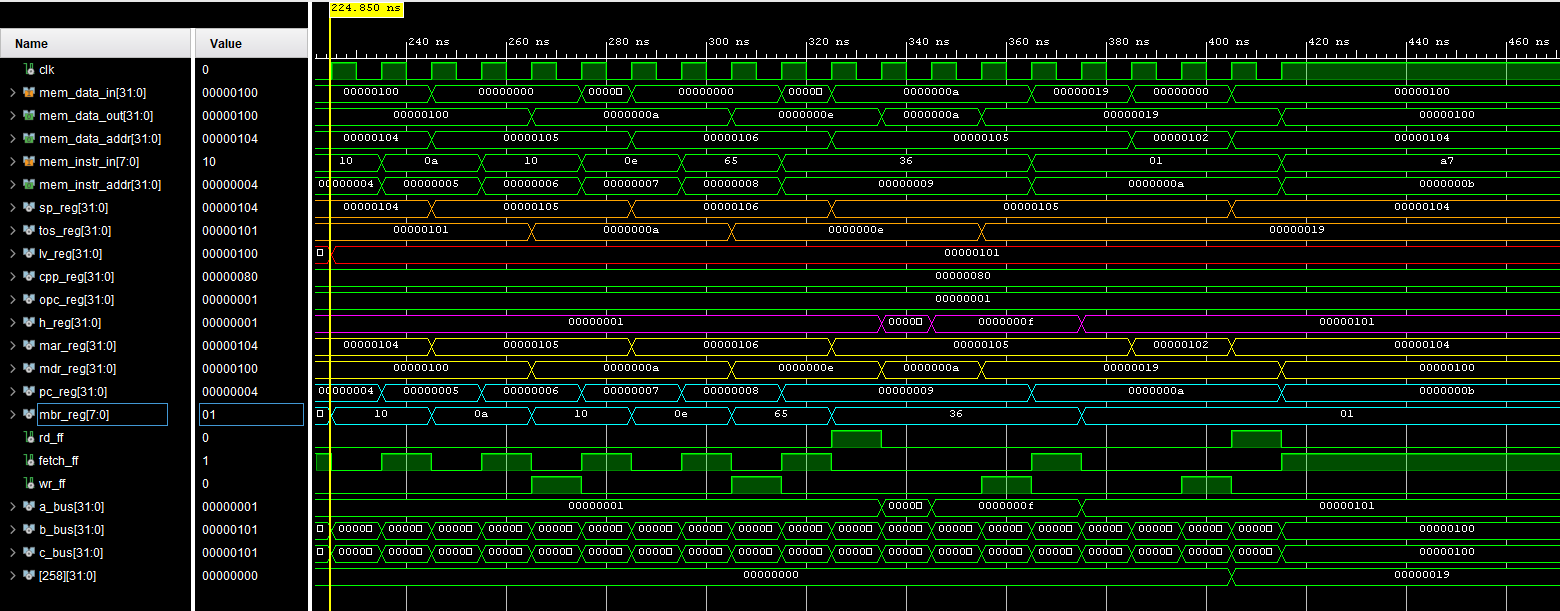
H = H+1

MDR = TOS = MDR + H; wr; goto main

L’istruzione di IADD, oltre ad effettuare la somma tra i due operandi, effettua anche l’incremento di 1 al valore ottenuto. Nel microprogramma viene incrementato di 1 il valore dell’operando in cima allo stack, contenuto nel registro H. L’istruzione IADD, in questo caso, impiegherà un ciclo di clk in più ad essere eseguita, per via della microistruzione di incremento.

Il programma ajvm associato alla istruzione IADD è lo stesso precedente.

SIMULAZIONE IADD MODIFICATA



Dalla simulazione si nota che l’esecuzione dell’istruzione IADD (0x36) è esattamente uguale a quella precedente, se non per l’incremento intermedio del valore in cima allo stack, che passa da 0xE a 0xF.