

MAL

MICRO ASSEMBLY LANGUAGE



A cura di:
Giuliana Franceschinis
giuliana@di.unito.it

Massimo Ubertini
massimo@fauser.edu

MICRO ARCHITETTURA

La micro architettura si appoggia sul livello logico ed interpreta le istruzioni definite all'interno dell'insieme delle istruzioni eseguibili da quella CPU: ISA-Level (Instruction Set Architecture).

Non esistono micro architetture "standard", ma famiglie di micro architetture, per esempio Motorola, Intel, AMD, cui corrispondono i relativi instruction set.

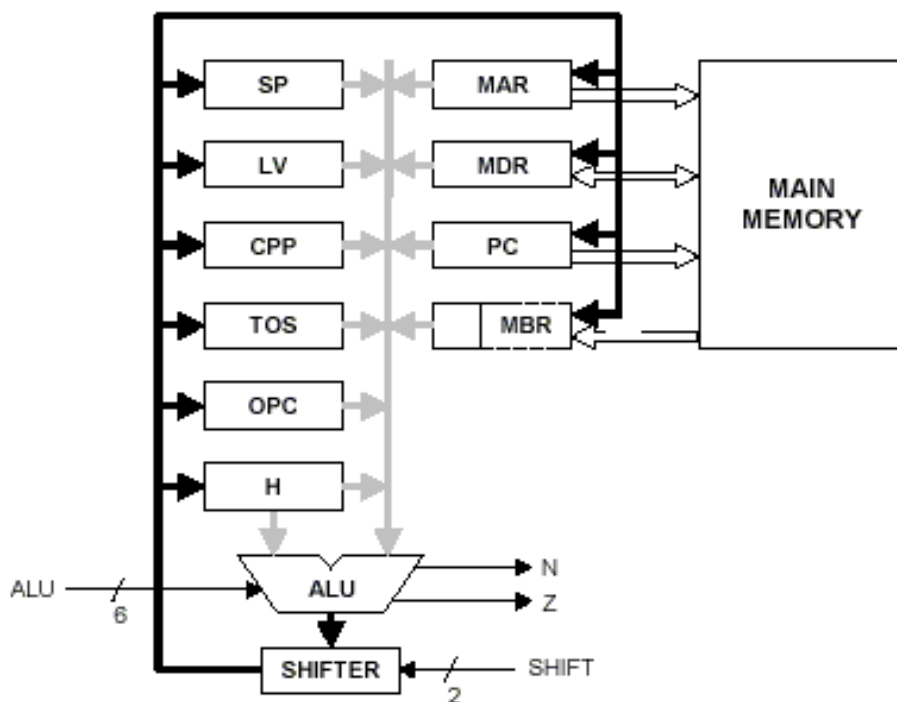
La micro architettura prevede:

1. micro programma presente nella memoria a sola lettura (ROM);
2. lettura delle singole istruzioni, che compongono il programma da eseguire, dalla memoria centrale (RAM), è la fase di fetch (prelievo);
3. decodifica (riconoscimento) della singola istruzione, specificata tramite il suo codice operativo (opcode);
4. esecuzione della singola istruzione.

Il micro programma è composto da micro istruzioni che controllano il data path (è quella parte di CPU che contiene l'ALU, con i suoi ingressi e le sue uscite) o unità di calcolo, i segnali di controllo del data path sono ventinove:

- 9 per controllare la memorizzazione (write) dei valori dal bus C nei registri;
- 9 per controllare la propagazione (read) sul bus B dei valori dei registri;
- 8 per controllare le funzioni dell'ALU e dello shifter;
- 2 per indicare read/write attraverso MAR/MDR;
- 1 per indicare il fetch attraverso PC/MBR.

Durante ogni ciclo sono letti i ventinove segnali di controllo.



MICRO ISTRUZIONI

Le micro istruzioni realizzano:

1. il fetch delle istruzioni assembly, in altre parole le istruzioni fornite a livello ISA;
2. il decode delle istruzioni del livello ISA;
3. l'execute delle istruzioni del livello ISA.

Una microistruzione è definita da (e specifica) l'insieme dei segnali di controllo nel data path

per un certo ciclo clock.

Una micro istruzione è eseguita in un ciclo clock.

Si può scrivere il contenuto del bus C in più di un registro.

Non si deve mai abilitare più di un registro sul bus B nello stesso momento.

Ci sono nove registri che possono essere abilitati sul bus B, i nove segnali di controllo possono essere codificati su quattro bit (16 segnali, 7 dei quali non servono).

Poiché un solo registro può scrivere sul bus B, i segnali di controllo del data path sono:

- 9 lettura dal bus C;
- 4 codifica assegnamento al bus B;
- 8 controllo ALU e shifter;
- 2 read e write;
- 1 fetch.

Questi 24 (9+4+8+2+1) segnali (bit) controllano il data path per un solo ciclo.

Oltre ai 24 bit dei segnali di controllo, ogni micro istruzione deve indicare: **l'operazione da compiere nel ciclo successivo.**

Tale informazione è specificata attraverso due campi aggiuntivi.

1. Addr: indirizzo della micro istruzione seguente (9 bit), a meno di salti.
2. JAM: codici di condizione per i salti (3 bit), JMPC, JAMN e JAMZ indicano i criteri per selezionare la prossima micro istruzione, ad esempio in base al risultato dell'ultima operazione effettuata dall'ALU che ha definito i valori delle uscite N e Z.

I bit di JAM sono tutti zero

Addr indica la prossima micro istruzione, MPC (*Micro Program Counter*) copia il valore di Addr.

JAMN o JAMZ sono diversi da zero

La micro istruzione successiva dipende dai valori memorizzati nei bistabili N e Z.

1. N (Negative) = 1 indica che l'ultima operazione ha restituito un numero negativo, altrimenti N = 0 numero positivo.
2. Z (Zero) = 1 indica che l'ultima operazione ha restituito un numero nullo, altrimenti Z = 0 numero diverso da zero.

JAMN o JAMZ non sono tutti a zero

La micro istruzione successiva, dipende dai valori memorizzati nei bistabili N e Z, può essere esclusivamente:

1. il valore indicato da Addr;
2. il valore indicato da Addr + 256, in pratica messo in OR con il numero esadecimale 0X100.

JAMZ = 0

Z = 0 la micro istruzione successiva è Addr.

Per esempio, stiamo eseguendo la micro istruzione 0x75, Z = 0

Indirizzo	Addr	JAM	Bit di controllo del data path
0X75	0X92	000

La micro istruzione successiva è Addr, in pratica 0X92.

JAMZ = 1

Z = 1 la micro istruzione successiva è Addr + 0X100

Per esempio, stiamo eseguendo la micro istruzione 0x75, Z = 1

Indirizzo	Addr	JAM	Bit di controllo del data path
0X75	0X92	001

La micro istruzione successiva è Addr + 0X100, in pratica 0X192.

JAMN = 0

N = 0 la micro istruzione successiva è Addr.

Per esempio, stiamo eseguendo la micro istruzione 0x75, N = 0

Indirizzo	Addr	JAM	Bit di controllo del data path
0X75	0X92	000

La micro istruzione successiva è Addr, in pratica 0X92.

JAMN = 1

N = 1 la micro istruzione successiva è Addr + 0X100.

Per esempio, stiamo eseguendo la micro istruzione 0x75, N = 1

Indirizzo	Addr	JAM	Bit di controllo del data path
0X75	0X92	010

La micro istruzione successiva è Addr + 0X100, in pratica 0X192.

JMPC = 1

Si esegue l'or bit a bit tra gli otto bit meno significativi di Addr e MBR.

Tale tecnica è molto utile per effettuare la selezione multipla, Select Case (BASIC), case (Pascal), switch (C, C++, Java).

Posso saltare alla micro istruzione relativa all'opcode appena letto in MBR.

Per esempio, istruzione a livello ISA: *istruzione1*.

Dopo il fetch MBR = 0000 0001 e Addr = 0 1010 1100 eseguendo l'or tra gli otto bit meno significativi di Addr e MBR si ottiene Addr = 0 1010 1101.

Si ottiene così un'iniziale decodifica dell'istruzione, eseguendo: la micro istruzione d'indirizzo 0 1010 1101 per l'istruzione *istruzione1* a livello ISA.

Per esempio, istruzione a livello ISA: *istruzione2*.

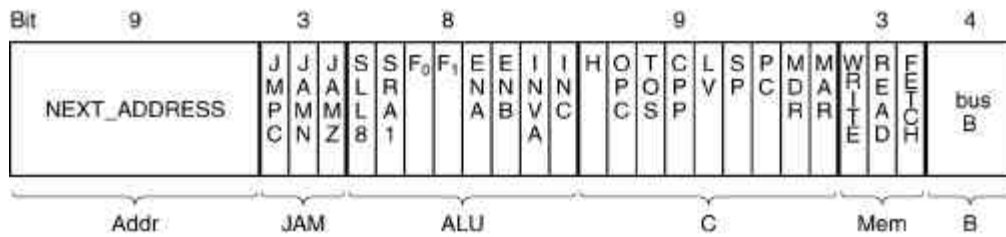
Dopo il fetch MBR = 0000 0010 e Addr = 0 1010 1100 eseguendo l'OR tra gli otto bit meno significativi di Addr e MBR si ottiene Addr = 0 1010 1110.

Si ottiene così un'iniziale decodifica dell'istruzione, eseguendo: la micro istruzione d'indirizzo 0 1010 1110 per l'istruzione *istruzione2* a livello ISA.

FORMATO MICRO ISTRUZIONI

I trentasei bit sono raggruppati in:

- Addr = 9;
- JAM = 3;
- ALU = 8;
- C = 9;
- Mem = 3;
- B = 4.



Registri del bus B

0 = MDR	5 = LV
1 = PC	6 = CPP
2 = MBR	7 = TOS
3 = MBRU	8 = OPC
4 = SP	9-15 nessuno

CONTROLLO MICRO ISTRUZIONI

Ogni istruzione ISA-Level è realizzata mediante l'esecuzione di una sequenza di micro istruzioni.

L'emissione dei segnali di controllo durante ogni ciclo è effettuata da un sequenziatore, che in ogni ciclo, basandosi sul contenuto di una memoria (*control store*), provvede a definire due informazioni.

1. Lo stato di ogni segnale di controllo del sistema.
2. L'indirizzo della micro istruzione seguente.

Le micro istruzioni sono memorizzate in una ROM, chiamata *control store*, formata da 512 (2^9) parole di 36 bit. Tutte queste parole, indicano i passi della sequenza di controllo corrispondenti all'esecuzione di ogni istruzione ISA-Level.

Ogni micro istruzione specifica esplicitamente la micro istruzione successiva: non sono eseguite nell'ordine in cui sono memorizzate.

Dispongono di un apposito campo che indica l'indirizzo della micro istruzione successiva.

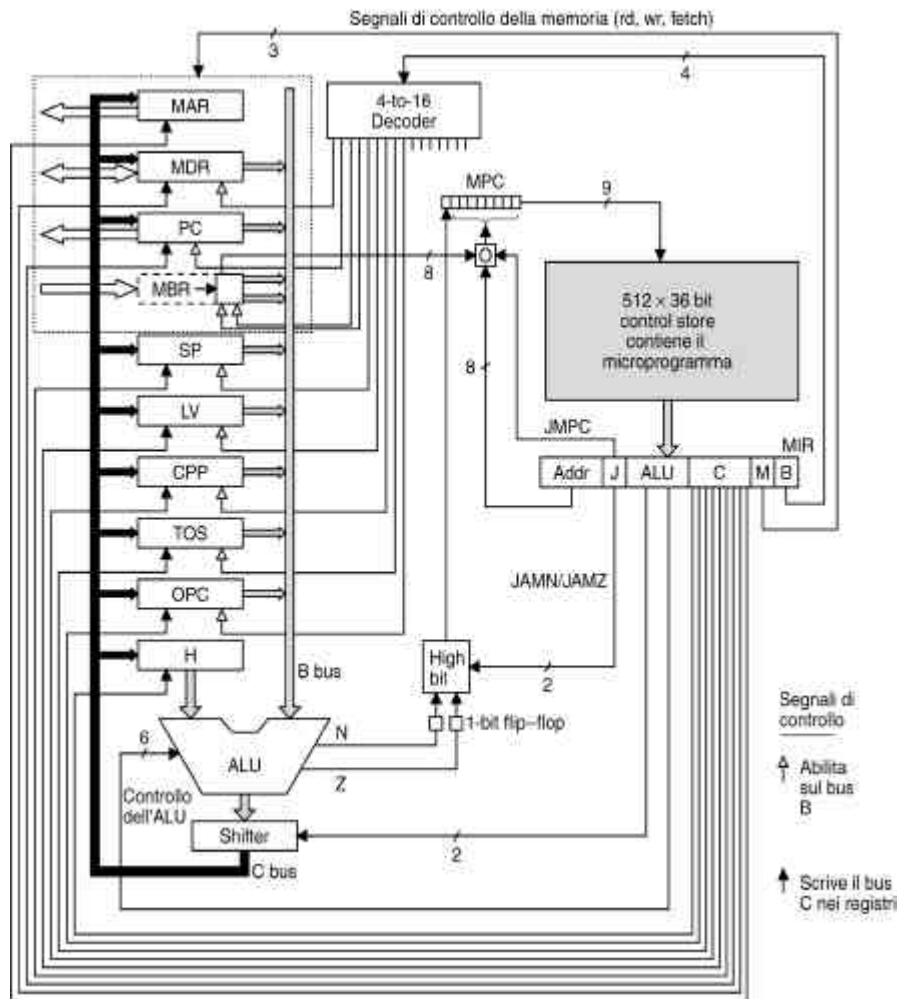
Il control store è una memoria a sola lettura e come tutte le memorie ha:

1. un registro d'indirizzo, MPC (*Micro Program Counter*) e indica, ad ogni passo di controllo, quale micro istruzione leggere, indirizza 512 parole, MPC ha dimensione nove bit.
2. un registro dati, MIR (*Micro Instruction Register*) per contenere la micro istruzione corrente i cui bit controllano i segnali del data path, MIR ha dimensione trentasei bit.

Nella figura seguente si vede:

- il control store ed i registri MPC e MIR;
- il decoder per i segnali di controllo, bus B;
- la logic per il calcolo degli indirizzi.

MIC1 si compone di due parti: a sinistra, il data path e a destra, la sezione controllo.



Il blocco di logica denominato *High Bit* determina il bit MPC[0] di MPC.

Svolge la seguente funzione.

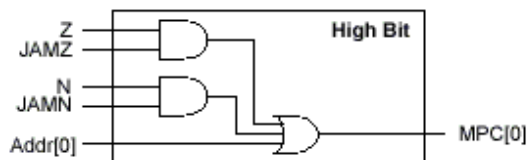
```

if (JAMN=0 and JAMZ=0) then MBR[0]=Addr[0];
if (JAMN=0 and JAMZ=1) then MBR[0]=Addr[0]+Z;
if (JAMN=1 and JAMZ=0) then MBR[0]=Addr[0]+N;
if (JAMN=1 and JAMZ=1) then MBR[0]=Addr[0]+N+Z

```

La funzione che ne risulta è:

$$MPC[0] = (JAMZ \text{ and } Z) \text{ or } (JAMN \text{ and } N) \text{ or } Addr[0]$$



Il blocco di logica denominato O determina il valore dei bit MPC[1..8] di MPC e svolge la seguente funzione.

```

if (JMPC=0) then MPC[1..8]=Addr[1..8];
if (JMPC=1) then MPC[1..8]=Addr[1..8] or MBR[0..7]

```

La funzione per la linea i-esima di MPC è la seguente.

$$MPC[i] = Addr[i] + JAMZ \text{ and } MBR[i-1]$$

MBR contiene un opcode, quindi porre (JMPC=1), significa caricare il valore corrispondente

all'opcode in MPC. Questo meccanismo consente un efficiente salto all'inizio del micro codice per una specifica istruzione.

Nella figura non è indicato che va eseguito l'or fra il nono bit di NEXT_ADDRESS e ((JAMZ and Z) or (JAMN and N)).

Il nono bit di NEXT_ADDRESS dovrebbe entrare nel blocco High Bit, in altre parole quello che esegue il calcolo del bit più significativo di MPC.

Il campo Addr, contenente NEXT_ADDRESS, è di nove bit. Di fatto, solo gli otto bit meno significativi sono utilizzati (ed eventualmente messi in or con gli otto bit provenienti da MBR se JMPC=1), mentre il computo del bit più significativo è eseguito nella seguente maniera: (JAMN and N) or (JAMZ and Z).

Quindi il bit più significativo sarà 0 o 1 solo in funzione di JAMN e JAMZ, e del contenuto dei registri N e Z, riducendo il campo Addr da nove a otto bit e, conseguentemente, il registro MIR da 36 a 35 bit.

La figura va bene per capire il funzionamento dell'hardware di MIC1, ma non è il massimo della chiarezza per capire in dettaglio la parte hardware che riguarda il calcolo della prossima micro istruzione.

Per quanto riguarda il calcolo degli otto bit meno significativi di MPC bisogna utilizzare un selettore pilotato da JMPC e avente in ingresso: gli otto bit meno significativi di NEXT_ADDRESS e gli otto bit formati dall'or logico tra il registro MBR e gli otto bit meno significativi di NEXT_ADDRESS. Così facendo, se JMPC è zero, l'uscita sarà formata dagli otto bit NEXT_ADDRESS, se JMPC è uno l'uscita sarà formata dagli otto bit dell'altro ingresso (MBR or NEXT_ADDRESS).

Per quanto riguarda il calcolo del bit più significativo di MPC, questo va calcolato tramite l'espressione booleana

$$F = (JAMZ \text{ and } Z) \text{ or } (JAMN \text{ and } N) \text{ or } NEXT_ADDRESS[8]$$

dove NEXT_ADDRESS[8] indica il nono (il più significativo) bit di NEXT_ADDRESS.

NOTAZIONE MICRO ISTRUZIONI

Una micro istruzione è:

- una parola di 36 bit (programmazione software);
- un insieme di segnali di controllo.

Abilità del progettista: selezionare gli indirizzi in modo efficiente, la memoria non è ordinata!

La rappresentazione di una micro istruzione può essere:

- esplicita, una sequenza di bit o un elenco dei segnali con valore 0 o 1;
- simbolica, una notazione più leggibile, semplice ed immediata.

Esempio: incrementare il valore di SP in un ciclo, iniziare un'operazione di lettura e salto alla micro istruzione seguente all'indirizzo 122.

Esplicita

ReadRegister=SP, ALU=INC, Write SP, Read, NEXTADDRESS=122

Simbolica

SP=SP+1; rd; goto 122

Questo è il MAL (Micro Assembly Language) che riflette le caratteristiche del MIC1.

L'architettura del MIC1 consente due tipi di salti.

1. Salti incondizionati: *goto A*, forza il micro assemblatore ad assegnare un valore specifico a NEXT_ADDRESS.
2. Salti condizionati: *if (cond) goto A1; else goto A2*

Se il bit cond è a 1 carica l'indirizzo A1 in MPC.

Se il bit cond è a 0 carica l'indirizzo A2 in MPC.

Gli indirizzi A1 e A2 devono differire solo per MSB.

Il bit di condizione *cond* può essere N o Z.

Per fissare il valore di N e Z si usano le micro istruzioni:

$Z=TOS$ $N=TOS$

Per esempio, $Z=TOS; \text{if } (Z) \text{ goto } L1; \text{else goto } L2$

Per saltare ad un indirizzo letto dalla memoria, si usa: $\text{goto } (MBR)$.

L'hardware richiede che questi due indirizzi siano uguali nei loro otto bit meno significativi.

$\text{goto } (MBR \text{ or valore})$

Forza il micro assemblatore ad usare valore come NEXT_ADDRESS, mette $JMPC = 1$

JAMN e JAMZ usano i bit N e Z

$Z=TOS$ $N=TOS$

MAL genera una micro istruzione in cui il valore di TOS passa attraverso l'ALU (non è memorizzato) in modo da calcolare il valore del bit Z (N) che è messo in or con il bit più significativo di MPC, forzando così la lettura all'indirizzo della micro istruzione L2 o L1 (che deve essere maggiore di L2 esattamente di 256).

IMPLEMENTAZIONE

L'esecuzione di ogni istruzione ISA-Level consiste:

- lettura dell'istruzione dalla memoria;
- salto alle micro istruzioni dell'istruzione ISA-Level;
- esecuzione della micro istruzione;
- ritorno alle micro istruzione per il caricamento della prossima istruzione.

I segnali rd , wr e $fetch$ hanno significati differenti:

rd (wr) sono usati per leggere (scrivere) una word dalla memoria in MAR/MDR.

$fetch$ è usato per leggere un opcode di un byte dal flusso d'istruzioni in PC/MBR.

ESECUZIONE SU MIC1

Micro Assembly Language (MAL): riflette le caratteristiche dell'architettura.

Una micro istruzione per ogni ciclo di clock.

Tipicamente un unico registro è scritto ad ogni ciclo.

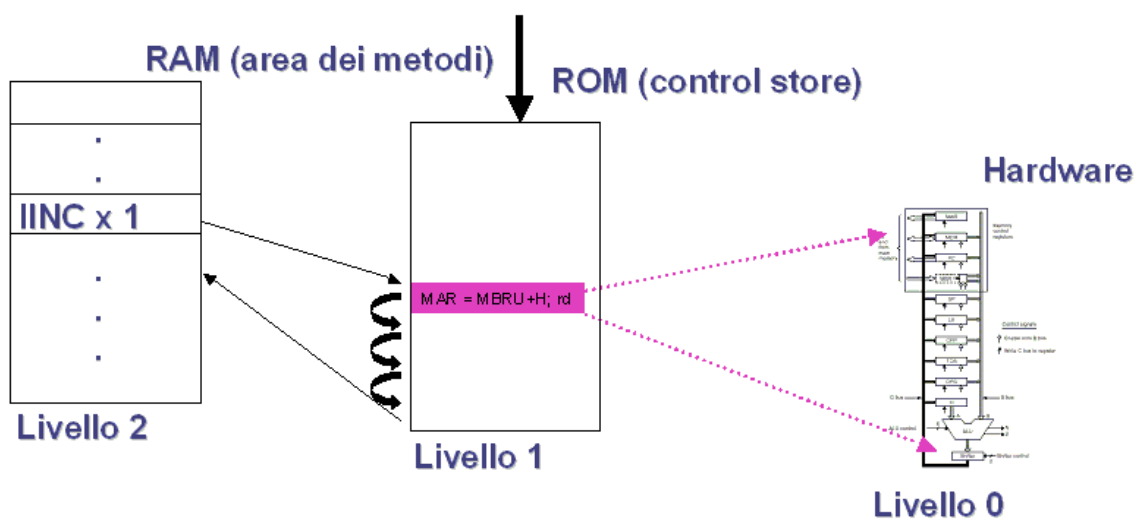
Ingresso alla ALU:

-lato A: registro H, oppure +1, -1, 0

-lato B: un registro (MDR, PC, MBR, MBRU, SP, LV, CPP, TOS, OPC).

Destinazione del calcolo della ALU:

MAR, MDR, PC, SP, LV, CPP, TOS, OPC, H.



ASSEGNAMENTI ILLEGALI

$MDR = SP + MDR$

è illegale, perché uno degli operandi deve essere H.

$H = H - MDR$

è illegale perché il sottraendo deve essere nel registro H.

$MAR = SP; rd$

$MDR = H$

il risultato è indefinito: entrambe le operazioni scrivono su MDR contemporaneamente (l'operazione di lettura da memoria infatti termina alla fine della prima microistruzione, e può essere usata al clock successivo).

OPERAZIONI DI LETTURA E SCRITTURA IN MEMORIA

Method area: solo lettura (fetch) di un byte

-indirizzo: nel PC;

-dato: nel registro MBR.

Constant pool: solo lettura di una parola.

Operazioni di lettura/scrittura di una parola: rd/wr

-indirizzo: nel registro MAR;

-dato: nel registro MDR.

OPERAZIONI DI CONTROLLO DI FLUSSO

goto label

goto (MBR)

goto (MBR OR value)

if (Z) goto label1; else goto label2

if (N) goto label1; else goto label2

Il valore ai bit N e Z è assegnato non abilitando alcun registro al caricamento in uscita dalla ALU: esempio $Z = TOS$

MICRO ISTRUZIONI

La micro istruzione successiva non è necessariamente alla riga consecutiva: in tal caso è presente un esplicito goto

Main1 PC = PC + 1; fetch; goto(MBR)

bipush1 SP = MAR = SP + 1

bipush2 PC = PC + 1; fetch

bipush3 MDR = TOS = MBR; wr; goto Main1

Micro programma che funziona sul MIC1 e interpreta IJVM (112 microistruzioni).

L'esecuzione di un'istruzione in termini di cicli clock.

Si consideri, ad esempio, l'istruzione:

$MDR = TOS + SP$

$T = 1 \quad rd \ SP, \ wr \ H$

$T = 2 \quad rd \ TOS, \ ALU = ADD, \ wr \ MDR$

A livello implementativo, le micro istruzioni non sono necessariamente consecutive nella control store, ad esempio:

.label pop1 0x57

.label dup1 0x59

Le locazioni 57 e 59 sono riservate; la traduzione della POP richiede tuttavia tre micro istruzioni: se fossero memorizzate in locazioni consecutive, la terza interferirebbe con la prima micro istruzione d'interpretazione della DUP.

È il micro assembler che riloca le micro istruzioni, e le linka fra loro; solo la prima micro istruzione di ogni insieme (dichiarata nello statement .label) è in una posizione predefinita.

Main1 (1 ciclo clock)

Ogni set di micro istruzioni che interpreta un'istruzione IJVM termina con un goto a Main1. L'opcode della nuova istruzione è già in MBR (risultato di un'operazione di fetch precedente): Main1 esegue un salto a tale indirizzo nel control store. Questo implica che, prima di tornare a Main1, PC deve puntare al nuovo opcode e l'operazione di fetch deve essere stata già lanciata.

```
Main1      PC = PC + 1; fetch; goto(MBR)
```

Incrementa il PC, che punterà al primo byte dopo l'opcode. Inizia un'operazione di fetch di tale byte in MBR. Salta al valore che MBR conteneva all'inizio del ciclo di clock (collocato da un fetch precedente): opcode di una micro istruzione uguale indirizzo del control store. All'inizio della terza micro istruzione, il byte di cui Main1 ha lanciato il fetch sarà disponibile. Main1 non è all'indirizzo zero del control store. Se il byte in MBR è zero, Main1 salta alla *NOP*. Il ciclo si ripete finché non è disponibile l'opcode di un'effettiva micro istruzione in MBR.

NOP (2 cicli clock)

Non esegue nessun'istruzione.

```
nop1 goto Main1
```

IADD (4 cicli clock)

Devo leggere l'elemento sotto il top stack; in tale locazione scriverò anche la somma.

```
iadd1     MAR = SP = SP - 1; rd
```

Copio top stack (memorizzato in TOS) in H per eseguire la somma.

```
iadd2     H = TOS
```

Ora l'operazione di lettura è terminata: posso sommare H ed MDR, che contiene il secondo operando, porre il risultato in MDR (e in TOS che è sempre la copia del top stack), e iniziare la scrittura. Intanto salto a Main1.

```
iadd3     MDR = TOS = MDR + H; wr; goto Main1
```

ISUB (4 cicli clock)

Devo leggere l'elemento sotto il top stack; in tale locazione scriverò anche la differenza.

```
isub1     MAR = SP = SP - 1; rd
```

Copio top stack (memorizzato in TOS) in H per eseguire la differenza.

```
isub2     H = TOS
```

Ora l'operazione di lettura è terminata: posso sottrarre H ed MDR, che contiene il secondo operando, porre il risultato in MDR (e in TOS che è sempre la copia del top stack), e iniziare la scrittura. Intanto salto a Main1.

```
isub3     MDR = TOS = MDR - H; wr; goto Main1
```

IAND (4 cicli clock)

Devo leggere l'elemento sotto il top stack; in tale locazione scriverò anche il risultato.

```
iand1     MAR = SP = SP - 1; rd
```

Copio top stack (memorizzato in TOS) in H per eseguire l'and.

```
iand2     H = TOS
```

Ora l'operazione di lettura è terminata: posso fare l'and tra H ed MDR, che contiene il secondo operando, porre il risultato in MDR (e in TOS che è sempre la copia del top stack), e iniziare la scrittura. Intanto salto a Main1.

iband3 *MDR = TOS = MDR AND H; wr; goto Main1*

IOR (4 cicli clock)

Devo leggere l'elemento sotto il top stack; in tale locazione scriverò anche il risultato.

ior1 *MAR = SP = SP - 1; rd*

Copio top stack (memorizzato in TOS) in H per eseguire l'or.

ior2 *H = TOS*

Ora l'operazione di lettura è terminata: posso fare l'or tra H ed MDR, che contiene il secondo operando, porre il risultato in MDR (e in TOS che è sempre la copia del top stack), e iniziare la scrittura. Intanto salto a Main1.

ior3 *MDR = TOS = MDR OR H; wr; goto Main1*

DUP (3 cicli clock)

Incremento SP, e lo copio in MAR.

dup1 *MAR = SP = SP + 1*

Scrivo la nuova parola in top stack.

dup2 *MDR = TOS; wr; goto Main1*

POP (4 cicli clock)

Leggo la parola vicina alla cima dello stack.

pop1 *MAR = SP = SP - 1; rd*

Aspetto che TOS sia letto dalla memoria.

pop2

Copio una nuova parola in TOS.

pop3 *TOS = MDR; goto Main1*

SWAP (7 cicli clock)

Metto MAR a SP - 1; leggo la seconda parola dallo stack.

swap1 *MAR = SP - 1; rd*

Metto MAR in top stack.

swap2 *MAR = SP*

Salvo TOS in H; scrivo la seconda parola in top stack.

swap3 *H = MDR; wr*

Copio il vecchio TOS in MDR.

swap4 *MDR = TOS*

Metto MAR a SP - 1; scrivo la seconda parola dello stack.

swap5 *MAR = SP - 1; wr*

Aggiorno il valore TOS.

swap6 *TOS = H; goto Main1*

BIPUSH (4 cicli clock)

Incremento SP, poiché devo scrivere una nuova parola.

bipush1 *SP = MAR = SP + 1*

PC deve essere incrementato e deve essere iniziato il fetch del nuovo opcode.

bipush2 *PC = PC + 1; fetch*

Il byte da collocare sullo stack è stato letto con un'operazione di fetch eseguita da Main1: ora il risultato è disponibile; tale byte deve essere esteso a 32 bit (estensione con segno: MBR) per essere collocato sullo stack; TOS deve essere aggiornato.

bipush3 *MDR = TOS = MBR; wr; goto Main1*

ILOAD (6 cicli clock)

Può identificare solo le prime 256 parole nel local variable frame. Per leggere la variabile, deve sommare a LV il valore dell'offset contenuto in MBR: operazione di fetch lanciata da Main1: LV è prima copiato in H.

iload1 $H = LV$

Nel frattempo l'operazione di fetch è terminata; si esegue la somma con MBRU (MBR esteso senza segno: l'offset non è mai negativo).

iload2 $MAR = MBRU + H; rd$

Mentre la variabile è letta, aggiorno MAR e SP per scrittura in cima allo stack.

iload3 $MAR = SP = SP + 1$

Incremento PC e inizio l'operazione di fetch del prossimo opcode; scrivo la variabile (la lettura intanto è terminata) in cima allo stack.

iload4 $PC = PC + 1; fetch; wr$

TOS è aggiornato.

iload5 $TOS = MDR; goto Main1$

ISTORE (7 cicli clock)

MBR contiene l'indice; copia LV in H.

istore1 $H = LV$

MAR contiene l'indirizzo della variabile locale nella quale scrivere.

istore2 $MAR = MBRU + H$

Copio TOS in MDR; scrivo la parola.

istore3 $MDR = TOS; wr$

Leggo la parola in top stack.

istore4 $SP = MAR = SP - 1; rd$

Incremento PC; leggo il prossimo opcode.

istore5 $PC = PC + 1; fetch$

Aggiorno TOS.

istore6 $TOS = MDR; goto Main1$

WIDE

L'offset che segue l'istruzione retta dalla WIDE diventa di 16 bit.

wide1 $PC = PC + 1; fetch; goto (MBR OR 0x100)$

Il micro codice d'interpretazione dell'istruzione estesa dalla WIDE inizia 256 indirizzi (0x100) dopo il micro codice dell'istruzione non estesa.

Esempio

ILOAD: 0x15; WIDE ILOAD: 0x115

WIDE ILOAD (9 cicli clock)

WIDE ILOAD index1 index2

Posso indirizzare più variabili nel local variable frame.

MBR contiene il primo byte d'indice, ora non devo estendere l'index usando MBRU, ma devo concatenare due byte: leggo anche il secondo byte.

wide_ildoad1 $PC = PC + 1; fetch$

Shifto di otto posizioni il primo byte (più significativo), e lo sommo al secondo, la cui lettura ora è terminata, esteso senza segno (or con zeri equivale a sommare).

wide_ildoad2 $H = MBRU \ll 8$

H contiene l'indice a sedici bit della variabile locale.

wide_ildoad3 $H = MBRU OR H$

MAR contiene l'indirizzo della variabile locale da mettere in top stack, accedo in lettura e poi procedo come con ILOAD.

```
wide_iloa4    MAR = LV + H; rd; goto iload3
```

WIDE ISTORE (10 cicli clock)

MBR contiene il primo byte d'indice; leggo il secondo.

```
wide_istore1  PC = PC + 1; fetch
```

Shifto di otto posizioni il primo byte (più significativo), e lo sommo al secondo, la cui lettura ora è terminata, esteso senza segno (or con zeri equivale a sommare).

```
wide_istore2  H = MBRU << 8
```

H contiene l'indice a sedici bit della variabile locale.

```
wide_istore3  H = MBRU OR H
```

MAR contiene l'indirizzo della variabile nella quale scrivere, poi procedo come con ISTORE.

```
wide_istore4  MAR = LV + H; goto istore3
```

LDC_W (8 cicli clock)

MBR contiene il primo byte d'indice; leggo il secondo.

```
ldc_w1    PC = PC + 1; fetch
```

Shifto di otto posizioni il primo byte (più significativo), e lo sommo al secondo, la cui lettura ora è terminata, esteso senza segno (or con zeri equivale a sommare).

```
ldc_w2    H = MBRU << 8
```

H contiene l'indice a sedici bit nella constant pool.

```
ldc_w3    H = MBRU OR H
```

MAR contiene l'indirizzo della costante nella constant pool, accedo in lettura e poi procedo come con ILOAD.

```
ldc_w4    MAR = H + CPP; rd; goto iload3
```

IINC (7 cicli clock)

MBR contiene l'indice; copio LV in H.

```
iinc1     H = LV
```

Copio LV più indice in MAR; leggo la variabile.

```
iinc2     MAR = MBRU + H; rd
```

Leggo la costante.

```
iinc3     PC = PC + 1; fetch
```

Copio la variabile in H.

```
iinc4     H = MDR
```

Leggo il successivo opcode.

```
iinc5     PC = PC + 1; fetch
```

Metto la somma in MDR; aggiorno la variabile.

```
iinc6     MDR = MBR + H; wr; goto Main1
```

GOTO (7 cicli clock)

Devo memorizzare il valore di PC quando punta all'opcode GOTO, perché è ad esso che devo sommare l'offset: lo metto in OPC.

```
goto1     OPC = PC - 1
```

Leggo i due byte di offset (il primo era già stato messo in MBR da Main1) e li sommo come nella WIDE ILOAD.

```
goto2     PC = PC + 1; fetch
```

```
goto3     H = MBRU << 8
```

```
goto4     H = MBRU OR H
```

Sommo OPC all'offset, ed ho la locazione dove saltare; inizio operazione di fetch di tale opcode.

```
goto5    PC = OPC + H; fetch
```

Aspetto la lettura del successivo opcode.

```
goto6    goto Main1
```

IFLT (8 cicli clock per false, 11 per true)

La parola da testare è già in TOS, ma TOS va riscritto perché si esegue una POP.

```
iflt1    MAR = SP = SP - 1; rd
```

La parola da testare è salvata temporaneamente in OPC.

```
iflt2    OPC = TOS
```

Metto il nuovo top stack in TOS.

```
iflt3    TOS = MDR
```

OPC passa senza modifiche nella ALU e setta il bit N, senza essere salvata in alcun registro.

```
iflt4    N = OPC; if (N) goto T; else goto F
```

IFEQ (8 cicli clock per false, 11 per true)

Leggo la parola in top stack.

```
ifeq1    MAR = SP = SP - 1; rd
```

Salvo temporaneamente TOS in OPC.

```
ifeq2    OPC = TOS
```

Metto il nuovo top stack in TOS.

```
ifeq3    TOS = MDR
```

Salto a molte direzioni in funzione del bit Z.

```
ifeq4    Z = OPC; if (Z) goto T; else goto F
```

IF_ICMPEQ (10 cicli clock per false, 13 per true)

Leggo la parola in top stack.

```
if_icmpeq1    MAR = SP = SP - 1; rd
```

Preparo MAR per leggere il nuovo top stack.

```
if_icmpeq2    MAR = SP = SP - 1
```

Copio la seconda parola dello stack in H.

```
if_icmpeq3    H = MDR; rd
```

Salva temporaneamente TOS in OPC.

```
if_icmpeq4    OPC = TOS
```

Mette il nuovo top stack in TOS.

```
if_icmpeq5    TOS = MDR
```

Se le due parole in top stack sono uguali, goto T, else goto F.

```
if_icmpeq6    Z = OPC - H; if (Z) goto T; else goto F
```

Se è negativo, prosegue come per GOTO.

```
T    OPC = PC - 1; fetch; goto goto2
```

Se è positivo:

salto il primo byte di offset.

```
F1    PC = PC + 1
```

PC ora punta al successivo opcode.

```
F2    PC = PC + 1; fetch
```

Aspetto la lettura dell'opcode.

```
F3    goto Main1
```

T e F sono posizionati dal micro assembler in indirizzi che differiscono solo per il bit più

significativo.

INVOKEVIRTUAL (23 cicli clock)

MBR contiene il primo byte d'indice; incremento PC; leggo il secondo byte.

invokevirtual1 $PC = PC + 1; \text{fetch}$

Sposto e salvo il primo byte in H.

invokevirtual2 $H = MBRU \ll 8$

H contiene l'offset da CPP del puntatore al metodo.

invokevirtual3 $H = MBRU \text{ OR } H$

Leggo il puntatore al metodo nella constant pool.

invokevirtual4 $MAR = CPP + H; rd$

Salvo temporaneamente il PC di ritorno in OPC.

invokevirtual5 $OPC = PC + 1$

PC punta al nuovo metodo; leggo il numero di parametri.

invokevirtual6 $PC = MDR; \text{fetch}$

Leggo il secondo byte del numero di parametri.

invokevirtual7 $PC = PC + 1; \text{fetch}$

Sposto e salvo il primo byte in H.

invokevirtual8 $H = MBRU \ll 8$

H contiene il numero di parametri.

invokevirtual9 $H = MBRU \text{ OR } H$

Leggo il primo byte del numero di variabili locali.

invokevirtual10 $PC = PC + 1; \text{fetch}$

TOS contiene l'indirizzo di OBJREF - 1.

invokevirtual11 $TOS = SP - H$

TOS contiene l'indirizzo di OBJREF, il nuovo LV.

invokevirtual12 $TOS = MAR = TOS + 1$

Leggo il secondo byte del numero di variabili locali.

invokevirtual13 $PC = PC + 1; \text{fetch}$

Sposto e salvo il primo byte in H.

invokevirtual14 $H = MBRU \ll 8$

H contiene il numero di variabili locali.

invokevirtual15 $H = MBRU \text{ OR } H$

Sovrascrivo OBJREF con il "link pointer".

invokevirtual16 $MDR = SP + H + 1; wr$

Scrivo SP, MAR indirizza la cella dove mettere il vecchio PC.

invokevirtual17 $MAR = SP = MDR;$

Salvo il vecchio PC "sopra" le variabili locali.

invokevirtual18 $MDR = OPC; wr$

SP punta alla cella che contiene il vecchio LV.

invokevirtual19 $MAR = SP = SP + 1$

Salvo il vecchio LV "sopra" il vecchio PC.

invokevirtual20 $MDR = LV; wr$

Leggo il primo opcode del nuovo metodo.

invokevirtual21 $PC = PC + 1; \text{fetch}$

Faccio puntare LV al frame LV.

invokevirtual22 $LV = TOS; \text{goto Main1}$

IRETURN (9 cicli clock)

Resetto SP, MAR riceve il link pointer.

ireturn1 $MAR = SP = LV; rd$
 Aspetto che finisca la lettura.
ireturn2
 Faccio puntare LV al link pointer; leggo il vecchio PC.
ireturn3 $LV = MAR = MDR; rd$
 Scrivo MAR per leggere il vecchio LV.
ireturn4 $MAR = LV + 1$
 Ripristino il PC; leggo il successivo opcode.
ireturn5 $PC = MDR; rd; fetch$
 Scrivo in MAR l'indirizzo di TOS.
ireturn6 $MAR = SP$
 Ripristino LV.
ireturn7 $LV = MDR$
 Salvo il valore di ritorno sul top stack originale.
ireturn8 $MDR = TOS; wr; goto Main1$

HALT

halt1 $goto halt1$

OUT

out1 $OPC = H = - 1$
 $OPC = H + OPC$
 Calcolo OUT address.
 $MAR = H + OPC$
 Scrivo nello standard output.
 $MDR = TOS; wr$
nop
 Decremento SP.
 $MAR = SP = SP - 1; rd$
nop
 $TOS = MDR; goto Main1$

IN

in1 $OPC = H = - 1$
 $OPC = H + OPC$
 Calcolo IN address; leggo dallo standard input.
 $MAR = H + OPC; rd$
 Incremento SP; attendo la lettura.
 $MAR = SP = SP + 1$
 Scrivo.
 $TOS = MDR; wr; goto Main1$

Esercizio N° 1

Quanto tempo ci vuole perché un sistema MIC1 con clock a 200 MHz esegua la seguente istruzione C:

$i = j + k;$

dare la risposta in nano secondi.

Si traduce l'istruzione in linguaggio IJVM e si conta quante micro istruzioni occorrono per la sua esecuzione.

```
ILOAD j    // 6 micro istruzioni
ILOAD k    // 6 micro istruzioni
IADD       // 4 micro istruzioni
ISTORE i   // 7 micro istruzioni
           // Totale = 23 micro istruzioni
```

Con un clock a 200 MHz, ogni micro istruzione impiega un tempo pari a un periodo di clock:

$T = 1 / F = 1 / (200 * 10^6) = 5 \text{ ns}$ durata di una micro istruzione.

Per cui il tempo totale per l'esecuzione dell'istruzione C è:

$5 * 23 = 115 \text{ nano secondi}$

Esercizio N° 2

Nel MIC1 ci vuole 1 ns per preparare MIR, 1 ns per mettere il contenuto di un registro sul bus B, 3 ns per far funzionare l'ALU e lo shifter e 1 ns perché i risultati ritornino ai registri. La larghezza dell'impulso di clock è di 2 ns. Questo sistema può funzionare a 100 MHz? E a 150 MHz?

Il data path occupa 6 ns così ripartiti:

- 1 ns per preparare MIR;
- 1 ns per mettere il contenuto di un registro sul bus B;
- 3 ns per far funzionare l'ALU e lo shifter;
- 1 ns perché i risultati ritornino ai registri.

Ai 6 ns del data path bisogna aggiungere i 2 ns della larghezza dell'impulso di clock. Quindi, per far funzionare correttamente il sistema, il periodo di clock deve essere superiore a 8 ns.

A 100 MHz, il periodo di clock è: $T = 1 / F = 1 / (100 * 10^6) = 10 \text{ ns}$ ed è quindi sufficiente a far funzionare correttamente il sistema.

A 150 MHz, il periodo di clock è: $T = 1 / F = 1 / (150 * 10^6) = 6,667 \text{ ns}$ ed non è quindi sufficiente a far funzionare correttamente il sistema.

Esercizio N° 3

Sintetizzare un circuito combinatorio che realizzi l'interfaccia tra il registro MBR e il bus B dell'architettura HW del MIC1, che permetta cioè di caricare sul bus B il valore di MBR o quello di MBRU.

Gli ingressi del circuito sono:

- i segnali di controllo SMBR e SMBRU;
- gli otto bit del registro MBR (A1, A2, A3, A4, A5, A6, A7, A8).

Le uscite sono:

- le 32 linee che vanno sul BUS B (U1, U2, ... U32).

Bisogna innanzitutto fare in modo che le uscite siano disconnesse dal Bus B se nessun segnale di controllo è abilitato (cioè quando $SMBR=SMBRU=0$). Questo è realizzato da tanti buffer tri-state quante sono le uscite e da una porta OR applicata ai segnali SMBR e SMBRU. Quindi, il segnale di controllo del buffer tri-state deve essere zero (in modo da disconnettere le uscite) quando $SMBR=SMBRU=0$.

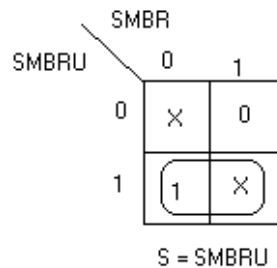
Una volta risolto questo problema bisogna fare in modo che:

- se SMBR è attivo, i bit da U9 a U32 siano uguali al bit A8;
- se SMBRU è attivo, i bit da U9 a U32 siano uguali a 0.

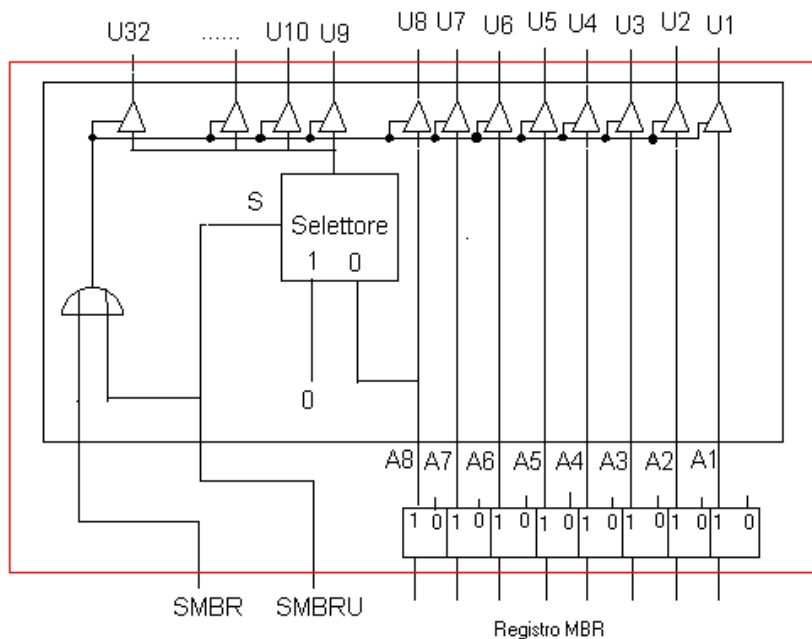
Questo si realizza tramite un selettore che, in ingresso ha il bit A8 e 0 e, in uscita ha il bit U9 che è ricopiato in tutti i bit successivi fino a U32. Naturalmente questo selettore va pilotato da una linea che abbia la seguente tabella di verità:

SMBR	SMBRU	S
0	0	X
0	1	1
1	0	0
1	1	X

L'uscita corrispondente a SMBR=SMBRU=0 è non specificata perché in questo caso le uscite sono disconnesse dal Bus B tramite il circuito descritto precedentemente. Anche l'uscita corrispondente a SMBR=SMBRU=0 è non specificata perché questo è un caso che non si dovrebbe mai verificare. Semplificando tramite mappa di Karnaugh si ottiene:



Sarà quindi SMBRU a pilotare il selettore. Il circuito finale è il seguente:



Esercizio N° 4

Quando è attivato il campo JMPC in una micro istruzione, MBR è messo in OR con NEXT-ADDRESS per formare l'indirizzo della micro istruzione seguente. Vi sono circostanze in cui ha senso che NEXT-ADDRESS valga 0X1FF e usare JMPC?

Considerando il campo Addr di nove bit, NEXT-ADDRESS potrebbe essere 0x1FF, tuttavia il bit più significativo del campo Addr non è utilizzato in nessun modo. Infatti non avrebbe senso se realmente il calcolo del bit[8] di MPC avvenisse secondo lo schema:

$$(\text{JAMN AND N}) \text{ OR } (\text{JAMZ AND Z}) \text{ OR NEXT-ADDRESS}[8]$$

poiché se NEXT-ADDRESS fosse 0x1FF, per esempio, non sarebbe possibile specificare in alcuna maniera un secondo indirizzo cui il micro programma dovrebbe saltare nel caso in cui il test su N o Z sia positivo. Infatti (JAMN AND N) OR (JAMZ AND Z) può solo modificare il bit[8] di MPC da zero a uno, ma se questo venisse preso già come uno da Addr non sarebbe possibile specificare nessun altro indirizzo cui effettuare il salto e così sia che il test sia positivo, sia che si verifichi negativo, la prossima micro istruzione eseguita sarà comunque 0x1FF. Possiamo quindi più semplicemente considerare il caso in cui NEXT-ADDRESS sia 0xFF e JMPC = 1. Concludiamo che non ha senso poiché MPC conterrà sempre e comunque 0xFF nei bit da 0 a 7, sia che JMPC = 1, e quindi 0xFF messo in OR con MBR restituendo comunque 0xFF, sia che JMPC = 0.

Esercizio N° 5

Quando si traduce la micro istruzione:

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

in codice binario, L2 deve trovarsi fra le ultime 256 parole del control store. Non sarebbe anche possibile avere L1 a, diciamo, 0X40 e L2 a 0X140?

Sappiamo, invece, che è L1 che deve essere più in basso di 256 word, proprio perchè se il test su Z è positivo il bit più significativo di MPC sarà uno, altrimenti zero. Quindi le alternative sono:

- L2, se il test è negativo.

- L1, se il test è positivo, dove $L1 = L2 \text{ OR } 0x100$

perciò, rispondendo alla domanda, no, non è ugualmente possibile avere L1 a 0x40 e L2 a 0x140, proprio per i motivi sopra spiegati.

Esercizio N° 6

Si potrebbe pensare di realizzare la SWAP nel modo seguente:

```
swap1    MAR=SP-1; rd  
swap2    MDR=TOS; wr  
swap3    MAR=SP; wr  
swap4    TOS=MDR; goto main1
```

Può funzionare?

Non può funzionare perché per eseguire l'istruzione $\text{MDR} = \text{TOS}; \text{wr}$ s'impiegano due cicli di clock e quindi dopo il primo ciclo (in cui si è eseguito $\text{MDR} = \text{TOS}$) il valore di MDR non è TOS, ma il valore della lettura nell'indirizzo $\text{SP} - 1$ (poiché l'istruzione precedente era $\text{MAR} = \text{SP} - 1; \text{rd}$). Perciò la *wr* scrive il valore letto dalla *rd*.

Il micro codice di "swap" enunciato nel testo dell'esercizio è errato perchè non aggiorna il TOS con il corretto valore, in pratica con il valore che sta in cima allo stack. questo micro codice potrebbe funzionare se tra *swap3* e l'assegnamento $\text{TOS} = \text{MDR}$; intercorresse

un'altra micro istruzione, giusto il tempo di aggiornare MDR. Per capire meglio analizziamo il seguente esempio: abbiamo uno stack i cui primi due elementi sono rispettivamente 4 e 3

swap1 indica -> MAR = (indirizzo di 3); rd (leggi 3)

swap2 indica -> MDR = 4; wr (memorizza 4 all'indirizzo di 3)

swap3 indica -> MAR = (indirizzo di 4); wr (memorizza all'indirizzo di 4 il valore 3)

swap4 indica -> TOS = 4 (errore !!!!!!!);

MDR cui si riferisce l'ultimo assegnamento è quello di swap2, infatti, quello di swap3 sarà disponibile nella micro istruzione successiva a swap4.

Esercizio N° 7

È possibile che in un micro programma MIC1 MAL la micro istruzione seguente:

label1 TOS = H + SP; if (N) goto labelA; else goto labelB

corrisponda in binario alla sequenza di 36 bit, di cui i primi nove siano i seguenti? 100110101

I primi nove bit di una micro istruzione MIC1 binaria corrispondono all'indirizzo della prossima istruzione da eseguire, nel caso di assenza di salti o in caso di salto incondizionato. Nel caso di salto condizionato, corrisponde all'indirizzo dell'istruzione da eseguire in caso la condizione non sia verificata (nel nostro caso 100110101). L'indirizzo cui andare nel caso la condizione è verificata sarà dato dai primi nove bit dell'istruzione con il bit più significativo messo in or con 1. Questo significa che nel nostro esempio, sia in caso di condizione verificata che non verificata, l'indirizzo sarebbe lo stesso. Il che non ha senso, a meno che labelA e labelB non denotino la stessa istruzione o siano la stessa etichetta.

MICRO ASSEMBLY LANGUAGE (MAL)

ELEMENTI LESSICALI

Come la maggior parte dei linguaggi assembly, il linguaggio MAL è un linguaggio orientato alla linea. Ogni micro istruzione è definita su una singola linea del programma. Il terminatore di linea è generalmente significativo: terminare il micro programma con una linea vuota.

MAL è un linguaggio case-sensitive. Il micro programma è in loop infinito, non esiste un'istruzione di HALT (tasto Stop del simulatore).

Commenti

Tutti i commenti cominciano con due caratteri slash ("/") e continuano fino alla fine della riga. Le linee vuote e linee consistenti solo di spazi bianchi e commenti sono ignorate dal micro assembler.

Direttive

Non sono parole nel Control store, sono informazioni al micro assembler, cominciano con un carattere punto (".") e possono contenere caratteri alfabetici. Ci sono due principali direttive per il micro assembler: ".default" e ".label". Queste sono usate per guidare il comportamento del micro assembler, e non corrispondono a parole nel control store.

```
.default goto stampa
```

Permette di specificare un'istanza di default per passare in qualche indirizzo non usato del control store. È messa in tutte le locazioni del control store non utilizzate.

```
.label nop1 0x00
```

Sono "ancorate" ad uno specifico indirizzo del control store.

Parole riservate

I nomi dei registri sono riservati. Per l'architettura MIC1, sono riservate le seguenti parole.

MAR

MDR

PC

MBR

MBRU

SP

LV

CPP

TOS

OPC

H

Z

N

rd

wr

fetch

if

else

goto

nop

AND
OR
NOT

Costanti letterali decimali

Le costanti letterali decimali sono stringhe numeriche: "0", "1", "8", sono usate come costanti numeriche nel MAL.

Costanti letterali esadecimali

Le costanti letterali esadecimali sono stringhe che cominciano col carattere "0x" e seguite da uno o più caratteri esadecimali ("0"- "9") oppure lettere ("a"- "f" o "A"- "F"), sono usate come indirizzi nel MAL.

Caratteri speciali

I caratteri seguenti hanno uno speciale significato nel linguaggio MAL.

(
)
+
-
=
;
<
>

ELEMENTI SINTATTICI

La grammatica seguente descrive il linguaggio accettato dal MIC1, *eol*, *label*, e *address* (costanti letterali esadecimali), sono simboli terminali, così come tutte le stringhe sono comprese tra apici ("). Tutti gli altri simboli non sono terminali. "*program*" è il simbolo d'inizio.

```
program ::= line_sequence  
          ;
```

```
line_sequence ::= line line_sequence  
                 |  
                 ;
```

```
line ::= instruction eol  
         | directive eol  
         | eol  
         ;
```

```
instruction ::= label statement_sequence  
               | statement_sequence  
               | label  
               ;
```

```
directive ::= ".label" label address  
              | ".default" statement_sequence  
              ;
```

```

statement_sequence ::= statement ";" statement_sequence
                    | statement ";"
                    | statement
                    ;

```

```

statement ::= io_statement
           | control_statement
           | assignment_statement
           | nop_statement
           ;

```

```

io_statement ::= "rd"
              | "wr"
              | "fetch"
              ;

```

```

control_statement ::= if_statement
                   | multiway_branch_statement
                   | goto_statement
                   ;

```

```

if_statement ::= "if" "(" condition ")" "goto" label ";" "else" "goto"
label
              ;

```

```

condition ::= "N"
            | "Z"
            ;

```

```

multiway_branch_statement ::= "goto" "(" mb_expr ")"
                           ;

```

```

mb_expr ::= "MBR" "OR" address
          | "MBR"
          ;

```

```

goto_statement ::= "goto" label
                 ;

```

```

assignment_statement ::= target "=" assignment_statement
                     | expr
                     ;

```

```

target ::= c_register
         | "N"
         | "Z"
         ;

```

```

c_register ::= "MAR"
             | "MDR"
             | "PC"

```

```

        | "SP"
        | "LV"
        | "CPP"
        | "TOS"
        | "OPC"
        | "H"
;

expr ::= operation
      | operation "<" "<" "8"
      | operation ">" ">" "1"
;

operation ::= a_term "AND" b_term
           | b_term "AND" a_term
           | a_term "OR" b_term
           | b_term "OR" a_term
           | "NOT" b_term
           | "NOT" a_term
           | b_term "+" a_term
           | a_term "+" b_term
           | b_term "+" "1"
           | a_term "+" "1"
           | b_term "-" a_term
           | "-" a_term
           | b_term "-" "1"
           | b_term "+" a_term "+" "1"
           | a_term "+" b_term "+" "1"
           | b_term
           | a_term
           | "-" "1"
           | "0"
           | "1"
;

b_term ::= "MDR"
        | "PC"
        | "MBR"
        | "MBRU"
        | "SP"
        | "LV"
        | "CPP"
        | "TOS"
        | "OPC"
;

a_term ::= "H"
;

nop_statement ::= "nop"
;

```

SEMANTICHE

Direttive Semantiche

.default

Questa direttiva permette di specificare un'istanza di default per passare in qualche indirizzo inutilizzato della memoria di controllo (control store).

.default goto err1

.label statement

Ancora gli statement (la prima micro istruzione corrispondente alla traduzione di ogni istruzione IJVM) a precise locazioni nel control store.

Ogni istruzione avrà un'etichetta collocata nella specifica locazione del control store (sono "ancorate" ad uno specifico indirizzo). Questa direttiva permette l'istruzione di salto come "*goto (MBR)*" oppure "*goto (MBR OR 0x100)*" per saltare ad una locazione conosciuta.

.label nop1 0x00

.label bipush1 0x10

La traduzione di NOP inizia all'indirizzo 0x00 (in esadecimale) del control store.

La traduzione di BIPUSH inizia all'indirizzo 0x10.

.default statement

Istruzione di default che può essere messa in ogni indirizzo non utilizzato del control store.

.default goto err1

Istruzioni Semantiche

Le linee che contengono un'etichetta, una sequenza d'istruzioni, o entrambe le cose sono da considerarsi una micro istruzione, ossia, una parola nel control store.

Se una linea inizia con una *label*, ci deve essere un'istruzione *goto*.

Se un'istruzione contiene un'esplicita etichetta *goto*, ci deve essere un'istruzione avente quell'etichetta.

I registri che si trovano alla sinistra del segno di uguale ("="), corrispondono a segnali di controllo abilitati a caricare i registri dal bus C.

I registri che si trovano senza il segno di uguale ("="), oppure alla destra del segno di uguale corrispondono a segnali di controllo abilitati a mettere i valori sul bus B come input per l'ALU.

I segni "+", "-", "<", ">", "AND", "OR", "NOT", "0", "1", e "8" che si trovano senza un segno di uguale ("=") oppure alla destra del segno di uguale, corrispondono a segnali di controllo d'input all'ALU e allo SHIFT register.

"*rd*", "*wr*" e "*fetch*", generano i bit da settare nel control store per abilitare i corrispondenti segnali di controllo.

"*if*", "*else*" e "*goto*" sono usati per settare *JAMN*, *JAMZ* e *JMPC*, bit delle micro istruzioni, con il campo NEXT_ADDRESS.

"*nop*" è un'istruzione "do-nothing" senza una label.

USO DEL SIMULATORE

Consente di visualizzare come un'istruzione JVM è implementata attraverso una serie di micro istruzioni.

È possibile visualizzare il contenuto dei registri durante l'esecuzione delle micro istruzioni.

È possibile visualizzare le operazioni eseguite dalla ALU.

Finestra che visualizza la method area.

Finestra che visualizza la memoria:

constant pool (da indirizzo 0x4000);

stack (da indirizzo 0x8000);

local variable frame (da indirizzo 0x8000).

Finestra che visualizza il control store: contiene il micro programma.

Scrivo il testo di un programma JVM.

Per esempio, il file lezione.jas.

È composto da:

-main (con una variabile locale x);

-il metodo *minimo* che riceve due parametri locx e locy e non ha variabili locali;

-il metodo confronta locx e locy e restituisce il minimo dei due.

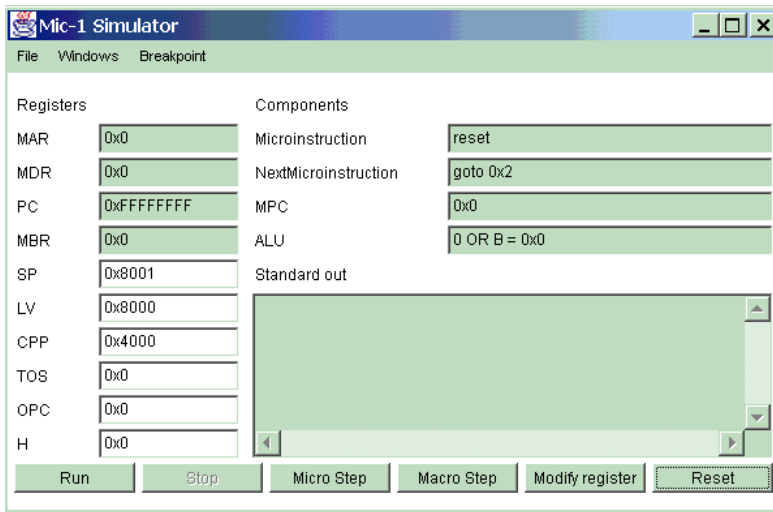
```
constant
OBJREF 0x40          // costante per chiamata metodo
.end-constant

.main              // inizio programma
.var              // variabili locali main
x
.end-var
  BIPUSH 0x41      // inizializzo x al valore ASCII A
  ISTORE x
  LDC_W OBJREF    // preparo per chiamata metodo
  ILOAD x         //parametri metodo: x e il valore B
  BIPUSH 0x42
  INVOKEVIRTUAL minimo
  OUT             //verifico che stampi A
  HALT
.end-main

.method minimo(locx,locy)
//due parametri, nessuna variabile locale
  ILOAD locx
  ILOAD locy
  ISUB
  IFLT L1
  ILOAD locx
  ILOAD locy
  ISTORE locx
  ISTORE locy
L1: ILOAD locx
  IRETURN          //ritorna locx
.end-method
```

Inizio esecuzione

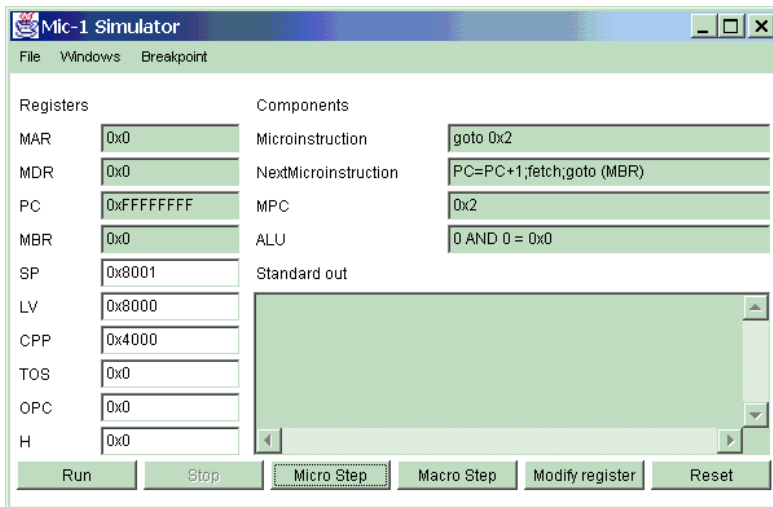
Alla partenza, la micro istruzione corrente è reset, che si limita a puntare all'indirizzo 0x00 del control store.



Address	Instruction	Hex instruction
0	goto 0x2	10000000
1	PC=PC+1;goto 0x40	200350201
2	PC=PC+1,fetch,goto (MBR)	4350211
3	H=TOS;goto 0x4	20148007
4	TOS=MDR=H+MDR;wr;goto 0x2	103C2140
5	H=TOS;goto 0x6	30148007
6	TOS=MDR=MDR-H;wr;goto 0x2	103F2140
7	H=TOS;goto 0x8	40148007
8	TOS=MDR=H AND MDR;wr;goto 0x2	100C2140
9	H=TOS;goto 0xA	50148007
a	TOS=MDR=H OR MDR;wr;goto 0x2	101C2140
b	MDR=TOS;wr;goto 0x2	10140147
c	goto 0xD	68000000
d	TOS=MDR;goto 0x2	10142000
e	MAR=SP;goto 0xF	78140084
f	H=MDR;wr;goto 0x11	88148040
10	SP=MAR=SP+1;goto 0x16	B0350484
11	MDR=TOS;goto 0x12	90140107

La micro istruzione all'indirizzo 0x00 del control store è nop1, che non fa niente; la successiva micro istruzione è all'indirizzo 0x2, ed è:

Main1: PC = PC + 1; fetch; goto (MBR)



Address	Instruction	Hex instruction
0	goto 0x2	10000000
1	PC=PC+1;goto 0x40	200350201
2	PC=PC+1,fetch,goto (MBR)	4350211
3	H=TOS;goto 0x4	20148007
4	TOS=MDR=H+MDR;wr;goto 0x2	103C2140
5	H=TOS;goto 0x6	30148007
6	TOS=MDR=MDR-H;wr;goto 0x2	103F2140

MICROSTEP

Esegue

nop1

Address	Instruction	Hex instruction
0	goto 0x2	10000000
1	PC=PC+1;goto 0x40	200350201
2	PC=PC+1;fetch;goto (MBR)	4350211
3	H=TOS;goto 0x4	20148007
4	TOS=MDR=H+MDR;wr;goto 0x2	103C2140

MICROSTEP

Esegue

Main1: PC = PC + 1; fetch; goto (MBR)

PC era 0xFFFFFFFF ora vale zero: il primo opcode della method area. Poiché l'operazione di fetch non è ancora terminata (il risultato sarà disponibile solo al prossimo ciclo di clock), MBR vale ancora zero; la prossima istruzione è di nuovo la nop1.

Esegue la nop1; nel frattempo l'operazione di fetch è terminata (MBR contiene l'opcode della prima istruzione del mio programma, che è BIPUSH = 0x10); la prossima istruzione è Main1.

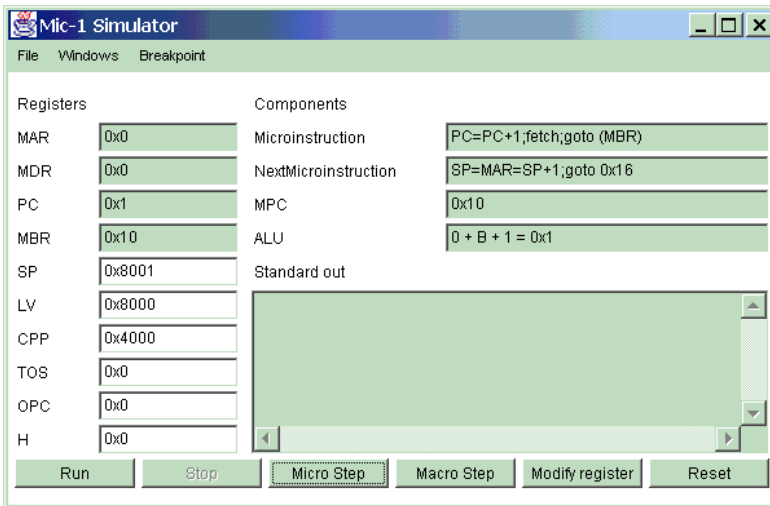
Address	Instruction	Hex instruction
0	goto 0x2	10000000
1	PC=PC+1;goto 0x40	200350201
2	PC=PC+1;fetch;goto (MBR)	4350211
3	H=TOS;goto 0x4	20148007
4	TOS=MDR=H+MDR;wr;goto 0x2	103C2140
5	H=TOS;goto 0x6	30148007

MICROSTEP

Esegue

Main1: PC = PC + 1; fetch; goto (MBR)

Ora goto (MBR) significa mettere in MPC 0x10, ed eseguire la BIPUSH. PC intanto è incrementato ad uno: l'operazione di fetch recupererà il byte da porre sullo stack.



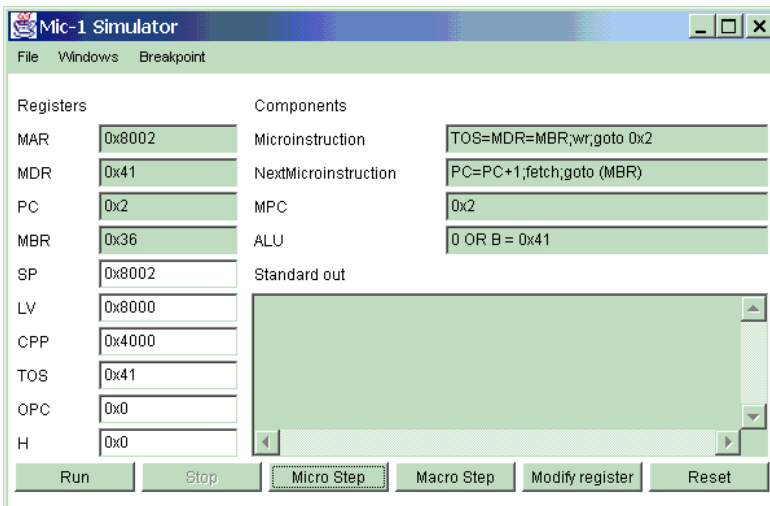
Address	Instruction	Hex instruction
0	goto 0x2	10000000
1	PC=PC+1;goto 0x40	200350201
2	PC=PC+1;fetch;goto (MBR)	4350211
3	H=TOS;goto 0x4	20148007
4	TOS=MDR=H+MDR;wr;goto 0x2	103C2140
5	H=TOS;goto 0x6	30148007
6	TOS=MDR=MDR-H;wr;goto 0x2	103F2140
7	H=TOS;goto 0x8	40148007
8	TOS=MDR=H AND MDR;wr;goto ...	100C2140
9	H=TOS;goto 0xA	50148007
a	TOS=MDR=H OR MDR;wr;goto 0x2	101C2140
b	MDR=TOS;wr;goto 0x2	10140147
c	goto 0xD	68000000
d	TOS=MDR;goto 0x2	10142000
e	MAR=SP;goto 0xF	78140084
f	H=MDR;wr;goto 0x11	88148040
10	SP=MAR=SP+1;goto 0x16	B0350484
11	MDR=TOS;goto 0x12	90140107
12	MAR=SP-1;wr;goto 0x14	A03600C4

MICROSTEP

Inizia ad eseguire la BIPUSH

ALU: $0 \text{ OR } B =$ passo il contenuto del registro (B) invariato; serve per gli assegnamenti (esempio $\text{MDR} = \text{TOS}$).

NOT A + B +1: differenza tra due registri (esempio $Z = \text{OPC} - \text{H}$): sottrarre un numero binario significa sommare la sua negazione bit a bit e sommare uno.



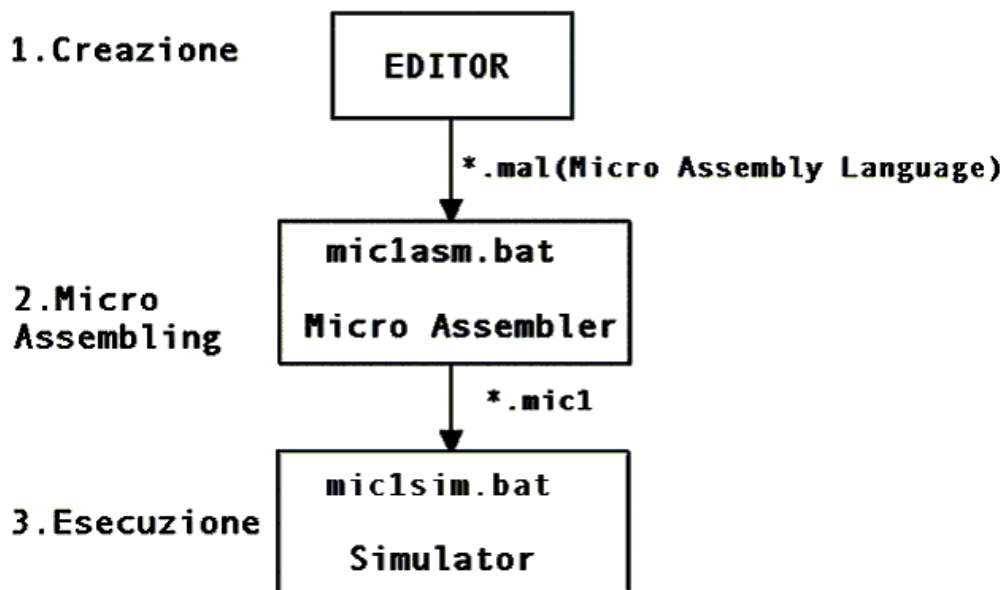
Address	Instruction	Hex instruction
0	goto 0x2	10000000
1	PC=PC+1;goto 0x40	200350201
2	PC=PC+1;fetch;goto (MBR)	4350211
3	H=TOS;goto 0x4	20148007
4	TOS=MDR=H+MDR;wr;goto 0x2	103C2140

DIFFERENZE TRA MIC1 E IJVM

Il linguaggio MAL è un linguaggio come qualsiasi altro e quindi, nel control store, è possibile mettere un qualsiasi programma. In pratica, però, si mette un micro interprete per eseguire i programmi IJVM. La differenza sostanziale è nel tempo di esecuzione:

- *.ijvm, numero cicli clock maggiore;
- *.mic1, non ho accesso alla RAM, quindi non ho ciclo di fetch.

SVILUPPO DEL PROGRAMMA



```
// Nome del programma: sequenza1.mal
// Programmatore:
// Descrizione
// Programma che visualizza la stringa "Ciao, Mondo".
```

```
.label stampa 0x00
.default goto halt1
```

```
stampa OPC=H=-1
  OPC=H+OPC
  MAR=H+OPC // calcolo IO address
  OPC=H=1 // 1
  OPC=H=H+OPC // 10
  OPC=H=H+OPC // 100
  OPC=H=H+OPC // 1000
  OPC=H=H+OPC // 10000
  OPC=H=H+OPC+1 // 100001
  MDR=H+OPC+1;wr // 1000011 'C'
//
  OPC=H=1 // 1
  OPC=H=H+OPC+1 // 11
  OPC=H=H+OPC // 110
  OPC=H=H+OPC+1 // 1101
  OPC=H=H+OPC // 11010
  OPC=H=H+OPC // 110100
  MDR=H+OPC+1;wr // 1101001 'i'
//
  OPC=H=1 // 1
```

```

OPC=H=H+OPC+1 // 11
OPC=H=H+OPC // 110
OPC=H=H+OPC // 1100
OPC=H=H+OPC // 11000
OPC=H=H+OPC // 110000
MDR=H+OPC+1;wr // 1100001 'a'
//
OPC=H=1 // 1
OPC=H=H+OPC+1 // 11
OPC=H=H+OPC // 110
OPC=H=H+OPC+1 // 1101
OPC=H=H+OPC+1 // 11011
OPC=H=H+OPC+1 // 110111
MDR=H+OPC+1;wr // 1101111 'o'
//
OPC=H=1 // 1
OPC=H=H+OPC // 10
OPC=H=H+OPC+1 // 101
OPC=H=H+OPC+1 // 1011
OPC=H=H+OPC // 10110
MDR=H+OPC;wr // 101100 ', '
//
OPC=H=1 // 1
OPC=H=H+OPC // 10
OPC=H=H+OPC // 100
OPC=H=H+OPC // 1000
OPC=H=H+OPC // 10000
MDR=H+OPC;wr // 100000 'spazio'
//
OPC=H=1 // 1
OPC=H=H+OPC // 10
OPC=H=H+OPC // 100
OPC=H=H+OPC+1 // 1001
OPC=H=H+OPC+1 // 10011
OPC=H=H+OPC // 100110
MDR=H+OPC+1;wr // 1001101 'M'
//
OPC=H=1 // 1
OPC=H=H+OPC+1 // 11
OPC=H=H+OPC // 110
OPC=H=H+OPC+1 // 1101
OPC=H=H+OPC+1 // 11011
OPC=H=H+OPC+1 // 110111
MDR=H+OPC+1;wr // 1101111 'o'
//
OPC=H=1 // 1
OPC=H=H+OPC+1 // 11
OPC=H=H+OPC // 110
OPC=H=H+OPC+1 // 1101
OPC=H=H+OPC+1 // 11011
OPC=H=H+OPC+1 // 110111

```

```

MDR=H+OPC;wr          // 1101110 'n'
//
OPC=H=1               // 1
OPC=H=H+OPC+1       // 11
OPC=H=H+OPC         // 110
OPC=H=H+OPC         // 1100
OPC=H=H+OPC+1       // 11001
OPC=H=H+OPC         // 110010
MDR=H+OPC;wr        // 1100100 'd'
//
OPC=H=1               // 1
OPC=H=H+OPC+1       // 11
OPC=H=H+OPC         // 110
OPC=H=H+OPC+1       // 1101
OPC=H=H+OPC+1       // 11011
OPC=H=H+OPC+1       // 110111
MDR=H+OPC+1;wr      // 1101111 'o'
halt1      goto halt1

```

TEST DELL'INSTALLAZIONE

Per verificare che le varie parti dell'installazione stanno lavorando correttamente, assemblare e lanciare il micro programma *sequenza1.mal*.

Test dell'installazione da Unix

A) Assemblare il micro programma *sequenza1.mal*

Dal prompt dei comandi:

```
$ java mic1asm mic1ijvm.mal mic1ijvm.mic1
```

Si può usare la versione GUI dell'assemblatore IJVM dal prompt dei comandi sotto Unix

```
$ java gmic1asm
```

Test dell'installazione da Windows XP (HE, P)

A) Assemblare il micro programma *sequenza1.mal*

Lanciare semplicemente *mic1asm.bat* con un doppio clic.

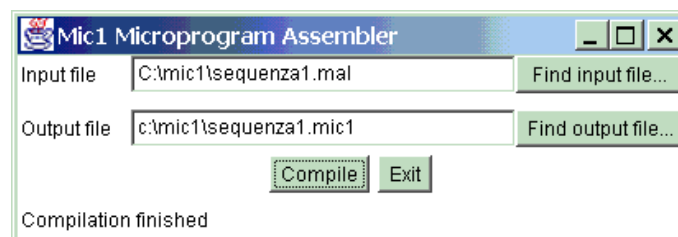
Questo comando lancia il micro assemblatore MIC1 (*mic1asm.bat*) per leggere il file di testo *sequenza1.mal*, che contiene il codice sorgente e produce un file binario *sequenza1.mic1*, che contiene il micro codice che deve essere caricato nel control store di una macchina che implementa l'architettura MIC1, questo è un file binario che contiene il codice oggetto per un micro programma MIC1.

Digitare *sequenza1.mal* come input file.

Digitare *sequenza1.mic1* come output file.

Clic sul bottone *Compile*.

Si selezionano i file d'input e di output da una dialog box.



B) Lanciare il simulatore MIC1, mic1sim

Eseguire *mic1sim.bat* con un doppio clic.

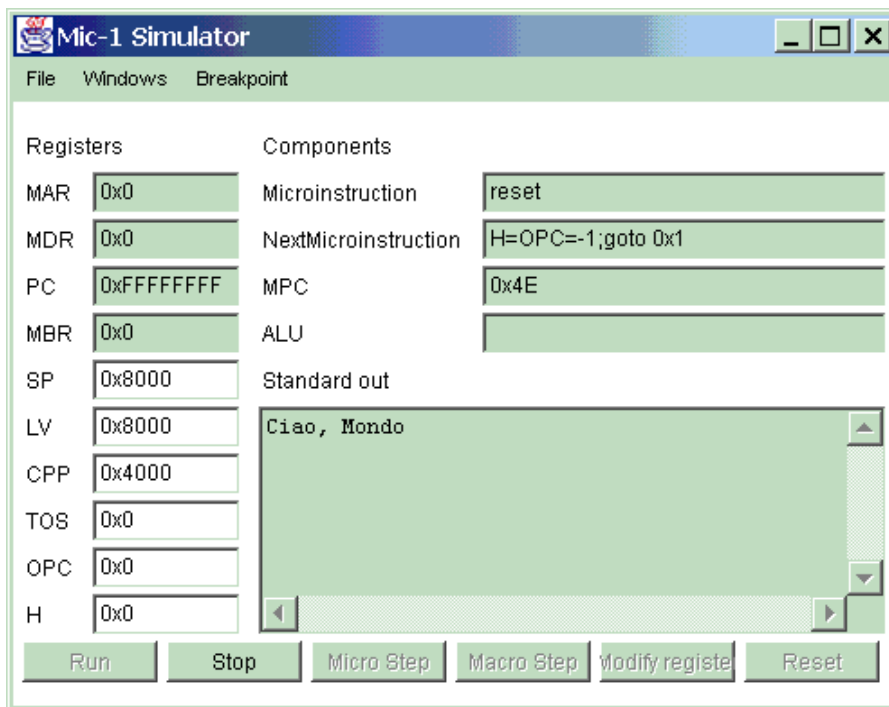
File/Load Microprogram/sequenza1.mic1 come micro programma da caricare.

C) Clic sul bottone Run

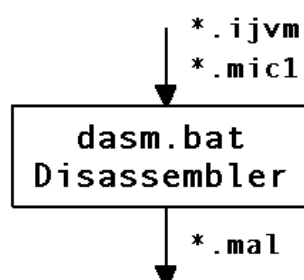
Inizia l'interpretazione del micro programma. Dopo un breve periodo, mentre il simulatore è in esecuzione, si dovrebbero vedere le seguenti parole sull'area "Standard out" di testo:

Ciao, mondo

Il simulatore è un programma Java con un'interfaccia grafica (GUI)



D) Disassemblare un file *.ijvm o un *.mic1



```
C:\mic1\Esercizi>dasm sequenza1.mic1 sequenza1.txt
micldasm V0.0
0x0: N=OPC=TOS=SP=PC=MDR=MAR=1>>1;if (N) goto 0x124; else goto 0x24
0x1: N=0>>1<<8;if (N) goto 0x102; else goto 0x102
0x2: LV=0>>1;goto (MBR OR 0x7)
0x3: H=LV=0;goto (MBR OR 0x107)
0x4: Z=0>>1<<8;if (Z) goto 0x102; else goto 0x2
0x5: LV=0>>1<<8;goto (MBR OR 0x107)
0x6: LV=0>>1<<8;goto (MBR OR 0x7)
0x7: LV=0>>1<<8;goto (MBR OR 0x107)
```

```

0x8: LV=0>>1<<8;goto (MBR OR 0x7)
0x9: Z=LV=0>>1<<8;if (Z) goto 0x107; else goto 0x107
0xa: Z=OPC=LV=1;if (Z) goto 0x107; else goto 0x7
0xb: Z=0>>1<<8;if (Z) goto 0x102; else goto 0x102
0xc: Z=LV=0>>1<<8;if (Z) goto 0x107; else goto 0x7
0xd: LV=0>>1<<8;goto (MBR OR 0x107)
0xe: Z=LV=0>>1<<8;if (Z) goto 0x107; else goto 0x7
0xf: LV=0>>1<<8;goto (MBR OR 0x107)
0x10: LV=0>>1<<8;goto (MBR OR 0x7)
0x11: Z=OPC=LV=1;if (Z) goto 0x107; else goto 0x107
0x12: Z=0>>1<<8;if (Z) goto 0x102; else goto 0x2
0x13: Z=LV=0>>1<<8;if (Z) goto 0x107; else goto 0x107
0x14: LV=0>>1<<8;goto (MBR OR 0x7)
0x15: LV=0>>1<<8;goto (MBR OR 0x107)
0x16: LV=0>>1<<8;goto (MBR OR 0x7)
0x17: LV=0>>1<<8;goto (MBR OR 0x107)
0x18: Z=OPC=LV=1;if (Z) goto 0x107; else goto 0x7
0x19: Z=0>>1<<8;if (Z) goto 0x102; else goto 0x102
0x1a: Z=LV=0>>1<<8;if (Z) goto 0x107; else goto 0x7
0x1b: LV=0>>1<<8;goto (MBR OR 0x107)
0x1c: Z=LV=0>>1<<8;if (Z) goto 0x107; else goto 0x7
0x1d: Z=LV=0>>1<<8;if (Z) goto 0x107; else goto 0x107
0x1e: Z=LV=0>>1<<8;if (Z) goto 0x107; else goto 0x7
0x1f: Z=OPC=LV=1;if (Z) goto 0x107; else goto 0x107
0x20: Z=0>>1<<8;if (Z) goto 0x102; else goto 0x2
0x21: LV=0>>1<<8;goto (MBR OR 0x107)
0x22: Z=LV=0>>1<<8;if (Z) goto 0x107; else goto 0x7
0x23: Z=LV=0>>1<<8;if (Z) goto 0x107; else goto 0x107
0x24: LV=0>>1<<8;goto (MBR OR 0x7)
0x25: OPC=LV=1;goto (MBR OR 0x107)
0x26: Z=0>>1<<8;if (Z) goto 0x102; else goto 0x2
0x27: LV=0>>1<<8;goto (MBR OR 0x107)
0x28: LV=0>>1<<8;goto (MBR OR 0x7)
0x29: LV=0>>1<<8;goto (MBR OR 0x107)
0x2a: LV=0>>1<<8;goto (MBR OR 0x7)
0x2b: OPC=LV=1;goto (MBR OR 0x107)
0x2c: Z=0>>1<<8;if (Z) goto 0x102; else goto 0x2
0x2d: LV=0>>1<<8;goto (MBR OR 0x107)
0x2e: LV=0>>1<<8;goto (MBR OR 0x7)
0x2f: Z=LV=0>>1<<8;if (Z) goto 0x107; else goto 0x107
0x30: Z=LV=0>>1<<8;if (Z) goto 0x107; else goto 0x7
0x31: LV=0>>1<<8;goto (MBR OR 0x107)
0x32: Z=OPC=LV=1;if (Z) goto 0x107; else goto 0x7
0x33: Z=0>>1<<8;if (Z) goto 0x102; else goto 0x102
0x34: Z=LV=0>>1<<8;if (Z) goto 0x107; else goto 0x7
0x35: LV=0>>1<<8;goto (MBR OR 0x107)
0x36: Z=LV=0>>1<<8;if (Z) goto 0x107; else goto 0x7
0x37: Z=LV=0>>1<<8;if (Z) goto 0x107; else goto 0x107
0x38: Z=LV=0>>1<<8;if (Z) goto 0x107; else goto 0x7
0x39: Z=OPC=LV=1;if (Z) goto 0x107; else goto 0x107
0x3a: Z=0>>1<<8;if (Z) goto 0x102; else goto 0x2
0x3b: Z=LV=0>>1<<8;if (Z) goto 0x107; else goto 0x107
0x3c: LV=0>>1<<8;goto (MBR OR 0x7)
0x3d: Z=LV=0>>1<<8;if (Z) goto 0x107; else goto 0x107
0x3e: Z=LV=0>>1<<8;if (Z) goto 0x107; else goto 0x7
0x3f: Z=LV=0>>1<<8;if (Z) goto 0x107; else goto 0x107
0x40: OPC=LV=1;goto (MBR OR 0x7)
0x41: Z=0>>1<<8;if (Z) goto 0x102; else goto 0x102
0x42: Z=LV=0>>1<<8;if (Z) goto 0x107; else goto 0x7
0x43: LV=0>>1<<8;goto (MBR OR 0x107)
0x44: LV=0>>1<<8;goto (MBR OR 0x7)

```

```
0x45: Z=LV=0>>1<<8;if (Z) goto 0x107; else goto 0x107
0x46: LV=0>>1<<8;goto (MBR OR 0x7)
0x47: OPC=LV=1;goto (MBR OR 0x107)
0x48: Z=0>>1<<8;if (Z) goto 0x102; else goto 0x2
0x49: Z=LV=0>>1<<8;if (Z) goto 0x107; else goto 0x107
0x4a: LV=0>>1<<8;goto (MBR OR 0x7)
0x4b: Z=LV=0>>1<<8;if (Z) goto 0x107; else goto 0x107
0x4c: Z=LV=0>>1<<8;if (Z) goto 0x107; else goto 0x7
0x4d: Z=LV=0>>1<<8;if (Z) goto 0x107; else goto 0x107
0x4e: Z=OPC=LV=1;if (Z) goto 0x107; else goto 0x7
0x4f: goto 0x0
0x50: goto 0x0
...
0x1fe: goto 0x0
0x1ff: goto 0x0
```

```

// Nome del programma: bi2push.mal
// Programmatore:
// Descrizione: push sullo stack di una costante con segno di 16 bit (2 byte).
// MDR = costante

```

```

.label    bi2push1    0x00
.default  goto        fine

```

```

// legge il secondo byte il primo è già in MDR
bi2push1  PC = PC + 1; fetch; goto bi2push2
// carica il primo byte in H e prepara lo spazio per il secondo
bi2push2  H = MDR<<8; goto bi2push3
// completa la costante a 16 bit, scrive in TOS
bi2push3  TOS = MDR = H = MBRU OR H; goto bi2push4
// aggiorna SP e scrive il risultato in top stack
bi2push4  MAR = SP = SP + 1; wr; goto bi2push5
// fetch della prossima micro istruzione
bi2push5  PC = PC + 1; fetch; goto fine
fine goto fine

```

```

// Nome del programma: contabit.mal
// Programmatore:
// Descrizione programma che dato un numero binario di 32 bit, conta il numero di bit a uno.
// OPC = numero binario di 32 bit, in TOS = risultato

```

```

.label    inizio     0x00
.default  goto        fine

```

```

inizio    TOS = 0; goto conta0
conta0    Z = OPC; if (Z) goto fine; else goto conta1
conta1    N = OPC; if (N) goto inc; else goto conta2
conta2    H = OPC; goto conta3
conta3    OPC = OPC + H; goto conta0
inc       TOS = TOS + 1; goto conta2
fine goto fine

```

```

// Nome del programma: c.mal
// Programmatore:
// Descrizione TOS = A e SP = B, il risultato è in OPC = 2 * (A - B)

```

```

.label    c1         0x00
.default  goto        fine

```

```

c1       H = TOS
c2       H = SP = SP - H
c3       OPC = SP + H
fine goto fine

```

```

// Nome del programma: per1.mal
// Programmatore:
// Descrizione
// Programma che dati due numeri li moltiplica.
// TOS = primo fattore
// OPC = secondo fattore
// SP = prodotto

.label    inizio    0x00
.default  goto     fine

inizio    SP = 0; goto POS
POS       Z = TOS; if (Z) goto fine; else goto per1
per1      TOS = TOS -1; goto per2
per2      H = OPC; goto per3
per3      SP = SP + H; goto POS
fine      goto fine

```

```

// Nome del programma: per2.mal
// Programmatore:
// Descrizione
// Programma che dati due numeri li moltiplica.
// TOS = primo fattore
// OPC = secondo fattore
// SP = prodotto

.label    inizio    0x00
.default  goto     fine

inizio    SP = 0; goto per2
per2      LV = TOS; goto per3
per3      N = TOS; if (N) goto NEG; else goto POS
NEG       H = TOS; goto per4
per4      TOS = -H
POS       Z = TOS; if (Z) goto FINE; else goto CONT
CONT      TOS = TOS -1; goto per5
per5      H = OPC; goto per6
per6      SP = SP + H; goto POS
FINE      N = LV; if (N) goto NEG1; else goto fine
NEG1     H = SP;
          SP = -H;goto POS1
fine      goto fine

```

```

// Nome del programma: sommavet2.mal
// Programmatore:
// Descrizione
// Programma che somma due vettori.
// Dati due vettori v1, v2 della stessa lunghezza n, calcolare v3 = v1 + v2.
// Supponiamo che:
// SP = indirizzo di partenza di v1 8000-8002 [SP, SP+TOS-1]
// LV = indirizzo di partenza di v2 8005-8007 [LV, LV+TOS-1]
// CPP = indirizzo di partenza di v3 800A-800C [CPP, CPP+TOS-1]
// TOS = n, in questo caso TOS = 3

```

```

.label    inizio    0x00
.default  goto     fine

```

```

inizio Z = TOS; if (Z) goto fine; else goto somma1
// v1
somma1 MAR = SP; rd; goto somma2
somma2 SP = SP + 1; goto somma3
somma3 H = MDR; goto somma4
// v2
somma4 MAR = LV; rd; goto somma5
somma5 LV = LV + 1; goto somma6
somma6 MDR = H + MDR; goto somma7
// v3
somma7 MAR = CPP; wr; goto somma8
somma8 CPP = CPP + 1; goto somma9
//
somma9 TOS = TOS - 1; goto inizio
fine goto fine

```

```

C:\mic1\Esercizi>dasm sommavet2.mic1 sommavet2.txt

```

```

micldasm V0.0
0x0: N=OPC=TOS=SP=PC=MDR=MAR=1>>1;if (N) goto 0x124; else goto 0x24
0x1: SP=PC=MDR=0;goto (MBR OR 0x122)
0x2: H=TOS=SP=0;goto (MBR OR 0x2)
0x3: N=SP=PC=MDR=0;if (N) goto 0x106; else goto 0x106
0x4: 0>>1;goto (MBR OR 0x2)
0x5: H=TOS=SP=MDR=0;goto (MBR OR 0x102)
0x6: Z=SP=0;if (Z) goto 0x106; else goto 0x6
0x7: 0<<8;goto (MBR OR 0x102)
0x8: H=SP=PC=0;goto (MBR OR 0x2)
0x9: OPC=LV=1;goto (MBR OR 0x107)
0xa: Z=SP=MDR=0;if (Z) goto 0x106; else goto 0x6
0xb: Z=SP=PC=0;if (Z) goto 0x106; else goto 0x6
0xc: goto 0x100
0xd: goto 0x100
...
0x1fd: goto 0x100
0x1fe: goto 0x100
0x1ff: goto 0x100

```

Mic-1 Simulator - Registers

Register	Value	Component	Value
MAR	0x0	Microinstruction	reset
MDR	0x0	NextMicroinstruction	Z=TOS;if (Z) goto 0x101; else goto 0x1
PC	0xFFFFFFFF	MPC	0x0
MBR	0x0	ALU	
SP	0x8000	Standard out	
LV	0x8005		
CPP	0x800A		
TOS	0x4		
OPC	0x0		
H	0x0		

Control Store

Address	Instruction	Hex instruction
0	Z=TOS;if (Z) goto 0x101; else got..	9140007
1	MAR=SP;rd;goto 0x2	101400A4
2	SP=SP+1;goto 0x3	18350404
3	H=MDR;goto 0x4	20148000
4	MAR=LV;rd;goto 0x5	281400A5
5	LV=LV+1;goto 0x6	30350805
6	MDR=H+MDR;goto 0x7	383C0100
7	MAR=CPP;wr;goto 0x8	401400C6
8	CPP=CPP+1;goto 0x9	48351006
9	TOS=TOS-1;goto 0x0	362007
a	goto 0x101	80800000
b	goto 0x101	80800000
c	goto 0x101	80800000
d	goto 0x101	80800000
e	goto 0x101	80800000
f	goto 0x101	80800000
10	goto 0x101	80800000
11	goto 0x101	80800000
12	goto 0x101	80800000
13	goto 0x101	80800000

Main Memory (4 byte words)

Word Address (hex)	Content (hex)	Pointers
8000	00000001	
8001	00000002	← SP
8002	00000003	
8003	00000004	
8004	00000000	
8005	00000010	← LV
8006	00000020	
8007	00000030	
8008	00000040	
8009	00000000	
800a	00000000	← CPP
800b	00000000	
800c	00000000	
800d	00000000	
800e	00000000	
800f	00000000	
8010	00000000	
8011	00000000	
8012	00000000	
8013	00000000	
8014	00000000	
8015	00000000	
8016	00000000	
8017	00000000	
8018	00000000	
8019	00000000	
801a	00000000	
801b	00000000	
801c	00000000	
801d	00000000	
801e	00000000	
801f	00000000	
8020	00000000	
8021	00000000	
8022	00000000	
8023	00000000	

Displayed memory address range: [initial address , initial address+1] modify the initial address insert a new hexadecimal value (smaller than 0xff7f)

First address OK

Mic-1 Simulator - Registers

Register	Value	Component	Value
MAR	0x800D	Microinstruction	goto 0x101
MDR	0x44	NextMicroinstruction	goto 0x101
PC	0xFFFFFFFF	MPC	0x101
MBR	0x0	ALU	0 AND 0 = 0x0
SP	0x8004	Standard out	
LV	0x8009		
CPP	0x800E		
TOS	0x0		
OPC	0x0		
H	0x4		

Control Store

Address	Instruction	Hex instruction
0	Z=TOS;if (Z) goto 0x101; else got..	9140007
1	MAR=SP;rd;goto 0x2	101400A4
2	SP=SP+1;goto 0x3	18350404
3	H=MDR;goto 0x4	20148000
4	MAR=LV;rd;goto 0x5	281400A5
5	LV=LV+1;goto 0x6	30350805
6	MDR=H+MDR;goto 0x7	383C0100
7	MAR=CPP;wr;goto 0x8	401400C6
8	CPP=CPP+1;goto 0x9	48351006
9	TOS=TOS-1;goto 0x0	362007
a	goto 0x101	80800000
b	goto 0x101	80800000
c	goto 0x101	80800000
d	goto 0x101	80800000
e	goto 0x101	80800000
f	goto 0x101	80800000
10	goto 0x101	80800000
11	goto 0x101	80800000
12	goto 0x101	80800000
13	goto 0x101	80800000

Main Memory (4 byte words)

Word Address (hex)	Content (hex)	Pointers
8000	00000001	
8001	00000002	← SP
8002	00000003	
8003	00000004	
8004	00000000	← SP #1
8005	00000010	
8006	00000020	
8007	00000030	
8008	00000040	
8009	00000000	← LV #1
800a	00000011	
800b	00000022	
800c	00000033	
800d	00000044	
800e	00000000	← CPP
800f	00000000	
8010	00000000	
8011	00000000	
8012	00000000	
8013	00000000	
8014	00000000	
8015	00000000	
8016	00000000	
8017	00000000	
8018	00000000	
8019	00000000	
801a	00000000	
801b	00000000	
801c	00000000	
801d	00000000	
801e	00000000	
801f	00000000	
8020	00000000	
8021	00000000	
8022	00000000	
8023	00000000	

Displayed memory address range: [initial address , initial address+1] modify the initial address insert a new hexadecimal value (smaller than 0xff7f)

First address OK

Le tre micro istruzioni sovrastanti sono codificate con i seguenti bit:

```
000000001 000 00010100 001000000 000 0000
000000010 000 00110101 000000100 001 0001
000000100 001 00011000 000000000 000 1001
```

Consideriamo il segmento di codice MIC1 dell'interprete che si occupa dell'implementazione dell'istruzione IJVM ISUB:

```
iSub1 MAR=SP=SP-1; rd
iSub2 H=TOS
iSub3 MDR=TOS=MDR-H; wr; goto Main1
```

L'opcode di ISUB è 0x64 quindi si ottiene:

```
0x64 001100101 000 00110111 000001001 010 0100
0x65 001100110 000 00010100 100000000 000 0111
0x66 000000001 000 00111111 001000010 100 0000
```

Avendo supposto che main1 si trovi all'indirizzo 0x01.

Consideriamo il codice MIC1 che si occupa dell'implementazione dell'istruzione IJVM SWAP:

```
swap1 MAR=SP-1;wr
swap2 MAR=SP
swap3 H=MDR;wr
swap4 MDR=TOS
swap5 MAR=SP-1;wr
swap6 TOS=H;goto Main1
```

Poiché l'opcode di SWAP è 0x5F, si ottiene la seguente traduzione in binario:

```
0x5F 001100000 000 00110111 000000001 010 0100
0x60 001100001 000 00010100 000000001 000 0100
0x61 001100010 000 00010100 100000000 100 0000
0x62 001100011 000 00010100 000000010 000 0111
0x63 001100100 000 00110111 000000001 100 0100
0x64 000000001 000 00011000 001000000 000 0000
```

Esercizio N° 3

Poiché un'istruzione MIC1 è composta da tutti i segnali di controllo per le varie componenti dell'hardware, questo significa che qualsiasi sequenza di 36 bit corrisponde ad un'istruzione MIC1 in formato MAL. È vero? Perché?

Non è vero che ad una qualsiasi sequenza di 36 bit corrisponde un'istruzione MIC1 in formato MAL perché esiste almeno un caso in cui non è vero. Infatti, se si verifica che i bit JAMN, JAMZ e JAMC sono tutti a 1 non ci può essere un'istruzione in formato MAL che la decodifica.

Dare due sequenze di 36 bit che codificano la stessa istruzione, spiegandone il perché.

Due istruzioni che producono TOS = 3D 0 sono:

```
011100101 000 00110000 001000000 000 0010
011100101 000 01000000 001000000 000 0011
```

Nella prima l'ALU esegue $0 + 0 = 3D 0$ e il risultato è memorizzato in TOS.

Nella seconda l'ALU esegue $(0 \text{ AND } 0) \gg 1 = 3D 0$ e il risultato è memorizzato in TOS. Abbiamo anche modificato i bit del BUS B perché (considerando che EnB è zero) è indifferente modificarne il valore.

Dire quali sono le micro istruzioni in formato MAL che corrispondono alle seguenti stringhe di 36 bit:

```

011100101000110101110110001010100010
11111111111111111111110000000000000000
00000000000000000000001111111111111111
101100101001010110101001111011010010

```

```
011100101 000 11010111 011000101 010 0010
```

L'istruzione MAL è:

$OPC = 3D$ TOS = $3D$ MAR = $3D$ PC = D 0; rd; goto L1
dove L1 ha indirizzo 0x0E5.

```
111111111 111 11111111 100000000 000 0000
```

A quest'istruzione non può corrispondere nessun'istruzione MAL perché sono abilitati contemporaneamente JAMC, JAMN e JAMZ.

```
000000000 000 00000000 011111111 111 1111
```

L'istruzione MAL è:

$OPC = 3D$ TOS = $3D$ CPP = $3D$ LV = $3D$ SP = $3D$ PC = $3D$ MDR = $3D$
MAR = $3D$ 0; rd; wr; fetch; goto L2
dove L2 ha indirizzo 0x000.

```
101100101 001 01011010 100111101 101 0010
```

L'istruzione mal è:

$H = 3D$ CPP = $3D$ LV = $3D$ SP = $3D$ PC = $3D$ MAR = $3D$
 $H' \gg 1$; = if(Z) goto L3; else goto L4
dove L3 ha indirizzo 0x165 e L4 ha indirizzo 0x165 e H indica la negazione di H.

Esercizio N° 4

L'opcode di NOP è 0x00. Si potrebbe allora pensare di mettere l'istruzione con etichetta Main1 del micro interprete all'indirizzo zero del control store. In questo modo se l'istruzione NOP è quella da eseguire, non dovendo fare nulla si può saltare direttamente di nuovo a Main1. L'idea pare buona, ma non funziona. Perché?

Quello che succede normalmente quando si deve eseguire una NOP è: supponiamo che inizialmente PC = 0 e che le prossime istruzioni IJVM siano senza argomenti.

```

Main1 PC = PC + 1; fetch;goto MBR // MBR = nop PC=1
Nop goto Main1 // MBR = nop PC=1
Main1 PC = PC + 1; fetch;goto MBR // MBR = next istruzione IJVM PC=2
prossima istruzione1
prossima istruzione2 // MBR = due istruzioni IJVM dopo
prossima istruzione3
prossima istruzione4

```

Mentre quello che succede nel caso proposto è: supponiamo che inizialmente PC = 0 e che le prossime istruzioni IJVM siano senza argomenti.

```

Main1 PC = PC + 1; fetch;goto MBR // MBR = Main1 PC=1
Main1 PC = PC + 1; fetch;goto MBR // MBR = Main1 PC=2
Main1 PC = PC + 1; fetch;goto MBR // MBR = next istruzione IJVM PC=3
prossima istruzione1 // MBR = due istruzioni IJVM dopo
prossima istruzione2 // MBR = tre istruzioni IJVM dopo
prossima istruzione3
prossima istruzione4

```

In definitiva, quello che succede è che sono prelevate troppe istruzioni successive. Ciò causa un malfunzionamento nell'eseguire le istruzioni IJVM successive.

ALTRA SOLUZIONE PROPOSTA

Analizziamo la modifica nel dettaglio: poniamo all'indirizzo 0x00 nel control store (opcode di NOP) la riga etichettata Main1 e supponiamo che l'attuale opcode da eseguire sia proprio quello di NOP:

```
1 0x00 Main1 PC = PC + 1; fetch; goto(0x00)
```

incrementa PC e salta nuovamente a 0x00 attivando un ciclo di fetch.

```
2 0x00 Main1 PC = PC + 1; fetch; goto(0x00)
```

Il nuovo OP-CODE in MBR non è ancora disponibile e così salta nuovamente a 0x00 incrementando però nuovamente PC.

```
3 0x00 Main1 PC = PC + 1; fetch; goto(nuovo OP-CODE)
```

Salta al nuovo opcode che intanto si è reso disponibile, ma vediamo che PC è stato incrementato una volta di troppo, così alla prossima esecuzione di Main1 sarà stato saltato un opcode. Ciò chiaramente pregiudica il corretto funzionamento dell'interprete.

Esercizio N° 5

Si consideri il seguente codice per un'ipotetica nuova micro istruzione:

```
i1 MDR = TOS
```

```
i2 MAR = SP = SP + 1; wr
```

```
i3 MAR = SP = SP + 1 wr; goto Main1
```

Quale operazione svolge?

La micro istruzione crea due copie aggiuntive dell'elemento in top stack, aggiornando SP di conseguenza: DUP DUP.

Si consideri il seguente codice per un'ipotetica nuova micro istruzione:

```
i1 MAR = SP; rd
```

```
// scrivo il nuovo valore di MDR (quello prodotto da i1) dato che rd  
// termina durante il secondo ciclo e comunque prima di wr.
```

```
// Non sto usando MDR per il salvataggio nei registri, quindi non
```

```
// devo attendere un ciclo vuoto per utilizzare il valore letto in i1
```

```
i2 MAR = SP = SP + 1; wr
```

```
i3 MAR = SP = SP + 1 wr; goto Main1
```

Quale operazione svolge?

Il risultato ottenuto è lo stesso dell'esercizio precedente, al costo di una lettura in più dalla memoria, che è di fatto inutile dato che l'elemento da replicare è già contenuto in TOS.

Esercizio N° 6

Supponete che per strani motivi, a noi sconosciuti, non sia possibile inserire la prima micro istruzione del codice che interpreta un'istruzione IJVM all'indirizzo del control store corrispondente all'opcode dell'istruzione IJVM. Questo significa che la fase di decodifica dell'istruzione IJVM non è immediata, ma il micro interprete avrà una serie d'istruzioni che effettueranno tale fase di decodifica dell'istruzione. Scrivere il segmento di codice MIC1 che effettua tale decodifica. Per semplicità supponiamo che:

- l'indirizzo di iadd1 sia 0x03

- l'indirizzo di isub1 sia 0x06

- l'indirizzo di iand1 sia 0x09

- l'indirizzo di `ior1` sia `0x0C`

e che `MBR` contenga l'opcode dell'istruzione `IJVM`. La parte di micro interprete che realizza la fase di decodifica dell'istruzione `IJVM` è costituita dal seguente codice `MIC1`:

```
OPC = 1
OPC = OPC + 1
H= OPC = OPC + 1           // OPC = H = 3
Z = MBR - H; if (Z) goto iadd1 // if (MBR = 0x03) goto iadd1
H = H + OPC
Z = MBR - H; if (Z) goto isub1 // if (MBR = 0x06) goto iadd1
H = H + OPC
Z = MBR - H; if (Z) goto iand1 // if (MBR = 0x09) goto iadd1
H = H + OPC
Z = MBR - H; if (Z) goto ior1  // if (MBR = 0x0C) goto iadd1
e così via.
```

Esercizio N° 7

Sappiamo che `TOS` contiene sempre il valore in cima allo stack. Ogni volta che lo stack è modificato, per esempio dall'istruzione `POP`, si deve ovviamente aggiornare `TOS`. Se però osserviamo il segmento di micro codice che implementa l'istruzione `INVOKEVIRTUAL`, tale aggiornamento non è fatto, è un errore del micro programmatore?

L'istruzione `INVOKEVIRTUAL` è corretta. Sappiamo infatti che essa serve per invocare un nuovo metodo, e quindi lo stack, dei dati, deve contenere le variabili relative a quest'ultimo. All'invocazione del metodo lo stack è considerato logicamente vuoto, quindi non avrebbe senso richiedere informazioni su `TOS`.

`IRETURN` non fa nessun assegnamento a `TOS` poiché lo stack è considerato inizialmente vuoto. Il comando `IRETURN` presuppone che alla fine del metodo in `TOS` sia contenuto il top stack (relativo al local variable frame utilizzato dal metodo), quindi non è `IRETURN` ad occuparsi di mantenere l'invariante su `TOS` (cioè che `TOS` contenga la testa dello stack), ma è il metodo stesso invocato che alla fine della sua esecuzione inserirà in `TOS` la testa dello stack relativo al local variable frame su cui ha operato. Infine c'è da dire che il metodo invocato lavora supponendo lo stack vuoto, quindi come prima istruzione non potrà essere utilizzato un comando che prevede l'utilizzo della variabile contenuta in `TOS`.

```

// Nome del programma: pervett.mal
// Programmatore:
// Descrizione
// Programma che moltiplica due vettori.
// Dati due vettori v1, v2 della stessa lunghezza n, calcolare v3 = v1 * v2.
// Supponiamo che:
// LV = indirizzo di partenza di v1 8000-8002 [SP, SP+TOS-1]
// SP = indirizzo di partenza di v2 8005-8007 [LV, LV+TOS-1]
// CPP = indirizzo di partenza di v3 800A-800C [CPP, CPP+TOS-1]
// TOS = n, in questo caso n=3

.label      inizio      0x00
.default   goto        fine

inizio Z = TOS; if (Z) goto fine; else goto pvett1
// v1
pvett1 MAR = LV; rd; goto pvett2
pvett2 LV = LV + 1; goto pvett3
pvett3 H = MDR; goto pvett4
// v2
pvett4 MAR = SP; rd; goto pvett5
pvett5 SP = SP + 1; goto pvett6
pvett6 TOS = MDR; goto passo1

//prodotto
passo1 OPC = TOS; goto passo2
passo2 TOS = 0; goto passo3
passo3 Z = OPC; if (Z) goto passo6; else goto passo4
passo4 TOS = H + TOS; goto passo5
passo5 OPC = OPC-1; goto passo3
passo6 MDR = TOS; wr; goto pvett7

// v3
pvett7 MAR = CPP; wr; goto pvett8
pvett8 CPP = CPP + 1; goto pvett9
//
pvett9 PC = PC - 1; goto inizio
fine goto fine

```

```

// Nome del programma: addlcl.mal
// Programmatore:
// Descrizione
// Aggiungere la seguente istruzione all'IJVM: ADDLCL varnum1 varnum2 varnum3
// la cui semantica è la seguente:  $M[LV + \text{varnum1}] = M[LV + \text{varnum2}] + M[LV + \text{varnum3}]$ 
// si pone nella variabile locale indicata da varnum1 la somma delle variabili locali indicate da
// varnum2 e varnum3. I campi varnum solo lunghi un byte.
// SP = 8000 varnum2 e in 8001 varnum3

```

```

.label    a1    0x00
.default  goto fine

a1 PC = PC+1; fetch          // fetch varnum2
a2 OPC = MBRU                // OPC = varnum1
a3 H = LV
a4 MAR = MBRU+H; rd          // read M[LV + varnum2]
a5 PC = PC+1; fetch          // fetch varnum3
a6 TOS = MDR                 // TOS = M[LV + varnum2]
a7 MAR = MBRU+H; rd          // read M[LV + varnum3]
a8 H = TOS                   // H = M[LV + varnum2]
a9 MDR = MDR+H               // MDR = M[LV + varnum2] + M[LV + varnum3]
a10 H =LV
// write M[LV + varnum2] + M[LV + varnum3] in M[LV + varnum1]
a11 MAR = OPC+H; wr a12 PC = PC+1; fetch;
fine goto fine

```

POPTWO

INTRODUZIONE

Volendo si può estendere la funzionalità dell'assemblatore IJVM, in pratica aggiungere nuovi comandi simili nel formato alle esistenti istruzioni. Bisogna semplicemente aggiungere le nuove istruzioni al file *ijvm.conf*. Naturalmente, occorre implementare le nuove micro istruzioni nel micro programma, il file *mic1ijvm.mal* e assemblarlo nuovamente.

Per esempio, in IJVM non esiste l'istruzione POPTWO, che cancella due parole dal top stack.

CODICE IJVM

```
// Nome del programma: poptwo.jas
```

```
// Programmatore:
```

```
// Descrizione
```

```
// Programma che verifica la nuova istruzione IJVM assembly.
```

```
.main
    BIPUSH 0X43          // codice ASCII "C"
    BIPUSH 0X69          // codice ASCII "i"
    POPTWO
    BIPUSH 0X20          // codice ASCII " "
    BIPUSH 0X6D          // codice ASCII "m"
    BIPUSH 0X6F          // codice ASCII "o"
    BIPUSH 0X6E          // codice ASCII "n"
    POPTWO
    HALT
.end-main
```

CODICE ISA -LEVEL

```
0 1d 29
1 ea 234
2 df 223
3 ad 173
4 0 0
5 1 1
6 0 0
7 0 0
8 0 0
9 0 0
10 0 0
11 0 0
12 0 0
13 0 0
14 0 0
15 0 0
16 0 0
17 0 0
18 0 0
```

```

19 f 15   contatore (32 bit, 4 byte), indica il numero di byte del
        codice da caricare nella method area, (F)H = (15)10 34 - 19 = 15
20 10 16  main
21 43 67
22 10 16
23 69 105
24 40 64  POPTWO
25 10 16
26 20 32
27 10 16
28 6d 109
29 10 16
30 6f 111
31 10 16
32 6e 110
33 40 64  POPTWO
34 ff 255

```

CODICE MAL

```

// Nome del programma: poptwo.mal
// Programmatore:
// Descrizione
// Si vuole aggiungere all'insieme d'istruzioni JVM Assembly la nuova istruzione POPTWO,
// che cancella due parole dal top stack.
// SP = indirizzo del top stack.

```

```

.label    poptwo1    0x00
.default  goto      fine

```

```

poptwo1   SP=SP-1           // sposta SP una prima volta
poptwo2   MAR=SP=SP-1;rd   // sposta SP una seconda volta e legge
poptwo3
poptwo4   TOS=MDR          // aggiorna il contenuto di TOS
fine goto fine

```

Soluzione alternativa

```

poptwo1   SP=SP-1; goto pop1 // riusa il codice esistente della POP

```

Compilo il MAL per ottenere il MIC1.

Word Address (hex)	Content (hex)	Pointers
8000	00000000	<-- SP, LV
8001	00000001	
8002	00000002	<-- SP
8003	00000000	

CODICE IJVM.CONF

```
// configuration file for IJVM Assembler
0x10 BIPUSH byte      // Push byte onto stack
0x59 DUP              // Copy top word on stack; push onto stack
0xA7 GOTO label      // Unconditional jump
0x60 IADD             // Pop two words from stack; push their sum
0x7E IAND             // Pop two words from stack; push Boolean AND
0x99 IFEQ label      // Pop word from stack; branch if it is zero
0x9B IFLT label      // Pop word from stack; branch if it is less than zero
0x9F IF_ICMPEQ label // Pop two words from stack; branch if equal
0x84 IINC varnum const // Add a constant to a local variable
0x15 ILOAD varnum    // Push local variable onto stack
0xB6 INVOKEVIRTUAL offset // Invoke a method
0xB0 IOR              // Pop two words from stack; push Boolean OR
0xAC IRETURN         // Return from method with integer value
0x36 ISTORE varnum   // Pop word from stack; store in local variable
0x64 ISUB            // Pop two words from stack; push their difference
0x13 LDC_W index     // Push constant from constant pool onto stack
0x00 NOP              // Do nothing
0x57 POP             // Delete word on top of stack
0x5F SWAP            // Swap the two top words on the stack
0xC4 WIDE            // Prefix instruction; next instruction has 16-bit index
0xFF HALT           // halt the simulator
0xFE ERR            // print ERROR and halt
0xFD OUT            // Pop a word from the stack and use the low order 8-
// bits as an ASCII character to display on screen
0xFC IN             // Read a character from standard input and put it in
// the low order 8-bits of a word pushed onto the stack
0x40 POPTWO         // Toglie due parole dal top dello stack.
```

CODICE MIC1IJVM.MAL

```
// note that this is nearly identical to the example
// given in Tanenbaum. Note:
// 1) SlashSlash-style ("/") comment characters have been added.
// 2) "nop" has been added as a pseudo-instruction to indicate that
// nothing should be done except goto the next instruction. It
// is a do-nothing sub-instruction that allows us to have MAL
// statements without a label.
// 3) instructions are "anchored" to locations in the control
// store as defined below with the ".label" pseudo-instruction
// 4) a default instruction may be specified using the ".default"
// pseudo-instruction. This instruction is placed in all
// unused locations of the control store by the mic1 MAL assembler.

// labeled statements are "anchored" at the specified control store address
.label nop1          0x00
.label bipush1       0x10
.label ldc_w1        0x13
.label iload1        0x15
.label wide_ild1     0x115
.label istore1       0x36
.label wide_istore1  0x136
.label pop1          0x57
.label dup1          0x59
.label swap1         0x5F
.label iadd1         0x60
.label isub1         0x64
.label iand1         0x7E
.label iinc1         0x84
.label ifeq1         0x99
```

```

.label      iflt1          0x9B
.label      if_icmpeq1     0x9F
.label      goto1         0xA7
.label      ireturn1       0xAC
.label      ior1          0xB0
.label      invokevirtual1 0xB6
.label      wide1         0xC4
.label      halt1         0xFF
.label      err1          0xFE
.label      out1          0xFD
.label      in1           0xFC
.label     poptwo1       0x40

```

```

// default instruction to place in any unused addresses of the control store
.default    goto err1

```

```

Main1 PC = PC + 1; fetch; goto (MBR) //MBR holds opcode; get next byte; dispatch

```

```

nop1 goto Main1 // Do nothing

```

```

iadd1 MAR = SP = SP - 1; rd // Read in next-to-top word on stack
iadd2 H = TOS // H = top of stack
iadd3 MDR = TOS = MDR + H; wr;goto Main1//Add top 2 words;write to top of stack

```

```

isub1 MAR = SP = SP - 1; rd // Read in next-to-top word on stack
isub2 H = TOS // H = top of stack
isub3 MDR = TOS = MDR - H; wr;goto Main1 //Do subtraction; write to top of stack

```

```

iand1 MAR = SP = SP - 1; rd // Read in next-to-top word on stack
iand2 H = TOS // H = top of stack
iand3 MDR = TOS = MDR AND H; wr; goto Main1 // Do AND; write to new top of stack

```

```

ior1 MAR = SP = SP - 1; rd // Read in next-to-top word on stack
ior2 H = TOS // H = top of stack
ior3 MDR = TOS = MDR OR H; wr; goto Main1 // Do OR; write to new top of stack

```

```

dup1 MAR = SP = SP + 1 // Increment SP and copy to MAR
dup2 MDR = TOS; wr; goto Main1 // Write new stack word

```

```

pop1 MAR = SP = SP - 1; rd // Read in next-to-top word on stack
pop2 // Wait for new TOS to be read from memory
pop3 TOS = MDR; goto Main1 // Copy new word to TOS

```

```

swap1 MAR = SP - 1; rd // Set MAR to SP - 1; read 2nd word from stack
swap2 MAR = SP // Set MAR to top word
swap3 H = MDR; wr // Save TOS in H; write 2nd word to top of stack
swap4 MDR = TOS // Copy old TOS to MDR
swap5 MAR = SP - 1; wr // Set MAR to SP - 1; write as 2nd word on stack
swap6 TOS = H; goto Main1 // Update TOS

```

```

bipush1 SP = MAR = SP + 1 // MBR = the byte to push onto stack
bipush2 PC = PC + 1; fetch // Increment PC, fetch next opcode
bipush3 MDR = TOS = MBR; wr;goto Main1 //Signextend constant and push on stack

```

```

iload1 H = LV // MBR contains index; copy LV to H
iload2 MAR = MBRU + H; rd // MAR = address of local variable to push
iload3 MAR = SP = SP + 1 // SP points to new top of stack; prepare write
iload4 PC = PC + 1; fetch; wr // Inc PC; get next opcode; write top of stack
iload5 TOS = MDR; goto Main1 // Update TOS

```

```

istore1 H = LV // MBR contains index; Copy LV to H

```

```

istore2    MAR = MBRU + H    // MAR = address of local variable to store into
istore3    MDR = TOS; wr     // Copy TOS to MDR; write word
istore4    SP = MAR = SP - 1; rd // Read in next-to-top word on stack
istore5    PC = PC + 1; fetch // Increment PC; fetch next opcode
istore6    TOS = MDR; goto Main1 // Update TOS
wide1     PC = PC + 1; fetch;goto (MBR OR 0x100)// Multiway branch with high bit set

wide_iloa1 PC = PC + 1; fetch // MBR contains 1st index byte; fetch 2nd
wide_iloa2 H = MBRU << 8    // H = 1st index byte shifted left 8 bits
wide_iloa3 H = MBRU OR H    // H = 16-bit index of local variable
wide_iloa4 MAR = LV + H; rd;goto iload3// MAR=address of local variable to push

wide_istore1 PC = PC + 1; fetch // MBR contains 1st index byte; fetch 2nd
wide_istore2 H = MBRU << 8    // H = 1st index byte shifted left 8 bits
wide_istore3 H = MBRU OR H    // H = 16-bit index of local variable
wide_istore4 MAR = LV + H; goto istore3// MAR=address of lv to store into

ldc_w1     PC = PC + 1; fetch // MBR contains 1st index byte; fetch 2nd
ldc_w2     H = MBRU << 8    // H = 1st index byte << 8
ldc_w3     H = MBRU OR H    // H = 16-bit index into constant pool
ldc_w4     MAR = H + CPP; rd;goto iload3// MAR = address of constant in pool

iinc1     H = LV            // MBR contains index; Copy LV to H
iinc2     MAR = MBRU + H; rd // Copy LV + index to MAR; Read variable
iinc3     PC = PC + 1; fetch // Fetch constant
iinc4     H = MDR          // Copy variable to H
iinc5     PC = PC + 1; fetch // Fetch next opcode
iinc6     MDR = MBR + H; wr; goto Main1 // Put sum in MDR; update variable

goto1     OPC = PC - 1     // Save address of opcode.
goto2     PC = PC + 1; fetch // MBR = 1st byte of offset; fetch 2nd byte
goto3     H = MBR << 8    // Shift and save signed first byte in H
goto4     H = MBRU OR H    // H = 16-bit branch offset
goto5     PC = OPC + H; fetch // Add offset to OPC
goto6     goto Main1      // Wait for fetch of next opcode

iflt1     MAR = SP = SP - 1; rd // Read in next-to-top word on stack
iflt2     OPC = TOS        // Save TOS in OPC temporarily
iflt3     TOS = MDR        // Put new top of stack in TOS
iflt4     N = OPC; if (N) goto T; else goto F // Branch on N bit

ifeq1     MAR = SP = SP - 1; rd // Read in next-to-top word of stack
ifeq2     OPC = TOS        // Save TOS in OPC temporarily
ifeq3     TOS = MDR        // Put new top of stack in TOS
ifeq4     Z = OPC; if (Z) goto T; else goto F // Branch on Z bit

if_icmpeq1 MAR = SP = SP - 1; rd // Read in next-to-top word of stack
if_icmpeq2 MAR = SP = SP - 1    // Set MAR to read in new top-of-stack
if_icmpeq3 H = MDR; rd         // Copy second stack word to H
if_icmpeq4 OPC = TOS          // Save TOS in OPC temporarily
if_icmpeq5 TOS = MDR          // Put new top of stack in TOS
if_icmpeq6 Z = OPC - H; if (Z) goto T; else goto F
// If top 2 words are equal, goto T, else goto F

T         OPC = PC - 1; fetch; goto goto2// Same as goto1; needed for target address

F         PC = PC + 1        // Skip first offset byte
F2        PC = PC + 1; fetch // PC now points to next opcode
F3        goto Main1        // Wait for fetch of opcode

invokevirtual1 PC = PC + 1; fetch// MBR = index byte 1; inc. PC, get 2nd byte

```

```

invokevirtual2  H = MBRU << 8      // Shift and save first byte in H
invokevirtual3  H = MBRU OR H        // H = offset of method pointer from CPP
invokevirtual4  MAR = CPP + H; rd // Get pointer to method from CPP area
invokevirtual5  OPC = PC + 1    // Save Return PC in OPC temporarily
invokevirtual6  PC = MDR; fetch // PC points to new method; get param count
invokevirtual7  PC = PC + 1; fetch // Fetch 2nd byte of parameter count
invokevirtual8  H = MBRU << 8      // Shift and save first byte in H
invokevirtual9  H = MBRU OR H        // H = number of parameters
invokevirtual10 PC = PC + 1; fetch // Fetch first byte of # locals
invokevirtual11 TOS = SP - H        // TOS = address of OBJREF - 1
invokevirtual12 TOS = MAR = TOS + 1 // TOS = address of OBJREF (new LV)
invokevirtual13 PC = PC + 1; fetch // Fetch second byte of # locals
invokevirtual14 H = MBRU << 8      // Shift and save first byte in H
invokevirtual15 H = MBRU OR H        // H = # locals
invokevirtual16 MDR = SP + H + 1; wr // Overwrite OBJREF with link pointer
invokevirtual17 MAR = SP = MDR;    // Set SP, MAR to location to hold old PC
invokevirtual18 MDR = OPC; wr      // Save old PC above the local variables
invokevirtual19 MAR = SP = SP + 1 // SP points to location to hold old LV
invokevirtual20 MDR = LV; wr       // Save old LV above saved PC
invokevirtual21 PC = PC + 1; fetch // Fetch first opcode of new method.
invokevirtual22 LV = TOS; goto Main1 // Set LV to point to LV Frame

ireturn1  MAR = SP = LV; rd // Reset SP, MAR to get link pointer
ireturn2  // Wait for read
ireturn3  LV = MAR = MDR; rd // Set LV to link ptr; get old PC
ireturn4  MAR = LV + 1      // Set MAR to read old LV
ireturn5  PC = MDR; rd; fetch // Restore PC; fetch next opcode
ireturn6  MAR = SP          // Set MAR to write TOS
ireturn7  LV = MDR          // Restore LV
ireturn8  MDR = TOS; wr;goto Main1//Save return value on original top of stack

halt1 goto halt1

err1  OPC=H--1
      OPC=H+OPC
      MAR=H+OPC // compute IO address
      OPC=H=1 // 1
      OPC=H=H+OPC // 10
      OPC=H=H+OPC // 100
      OPC=H=H+OPC // 1000
      OPC=H=H+OPC+1 // 10001
      OPC=H=H+OPC // 100010
      MDR=H+OPC+1;wr // 1000101 'E'
      OPC=H=1 // 1
      OPC=H=H+OPC // 10
      OPC=H=H+OPC+1 // 101
      OPC=H=H+OPC // 1010
      OPC=H=H+OPC // 10100
      OPC=H=H+OPC+1 // 101001
      MDR=H+OPC;wr // 1010010 'R'
      nop
      MDR=H+OPC;wr // 1010010 'R'
      OPC=H=1 // 1
      OPC=H=H+OPC // 10
      OPC=H=H+OPC // 100
      OPC=H=H+OPC+1 // 1001
      OPC=H=H+OPC+1 // 10011
      OPC=H=H+OPC+1 // 100111
      MDR=H+OPC+1;wr // 1001111 'O'
      OPC=H=1 // 1
      OPC=H=H+OPC // 10

```

```

        OPC=H=H+OPC+1           // 101
        OPC=H=H+OPC             // 1010
        OPC=H=H+OPC             // 10100
        OPC=H=H+OPC+1           // 101001
        MDR=H+OPC;wr             // 1010010 'R'
        goto halt1

out1   OPC=H=-1
        OPC=H+OPC
        MAR=H+OPC                // compute OUT address
        MDR=TOS; wr              // write to output
        nop
        MAR=SP=SP-1; rd          // decrement stack pointer
        nop
        TOS=MDR; goto Main1

in1    OPC=H=-1
        OPC=H+OPC
        MAR=H+OPC;rd            // compute IN address ; read from input
        MAR=SP=SP+1             // increment SP; wait for read
        TOS=MDR;wr ; goto Main1 // Write

poptwo1    SP=SP-1
poptwo2    MAR=SP=SP-1;rd
poptwo3
poptwo4    TOS=MDR; goto Main1

```

Compilo il MAL per ottenere il MIC1
 Eseguo il programma al simulatore.

IMAX

CODICE IJVM

```
// Nome del programma: imax.jas
// Programmatore:
// Descrizione
// Programma che verifica la nuova istruzione IJVM Assembly - IMAX -

.constant
OBJREF 0x20
.end-constant

.main
.var
DEPO
DEPO2
.end-var
    LDC_W OBJREF
    LDC_W OBJREF // Acquisizione valore hex in input
    INVOKEVIRTUAL getnum
    ISTORE DEPO // Salvataggio in DEPO
    INVOKEVIRTUAL getnum
    ISTORE DEPO2
    LDC_W OBJREF
    ILOAD DEPO // Richiamo valore immesso in input
    ILOAD DEPO2
    IMAX
    INVOKEVIRTUAL print // Stampa di IMAX
    HALT
.end-main
```

CODICE MAL

```
// Nome del programma: imax.mal
// Programmatore:
// Descrizione
// Programma che ricerca il massimo tra due numeri affioranti nello stack.
// TOS = primo numero, SP = 7FFF, il secondo numero è sullo stack.
```

```
.label    max1 0x00
.default  goto fine

imax1    MAR=SP=SP-1; rd // Mi posiziono sotto il top stack
imax2    H=TOS // Memorizzo il primo valore
imax3    OPC=MDR // Memorizzo il secondo valore
imax4    TOS=OPC-H // Memorizzo in TOS la differenza
imax5    N=TOS; if (N) goto imax7; else goto imax6
imax6    MDR=TOS=OPC; wr;goto fine// Secondo valore
imax7    MDR=TOS=H;wr // Primo valore
fine goto fine
```



```

LDC_W OBJREF
ILOAD total          // push total, parametro per il metodo print
INVOKEVIRTUAL print
GOTO start
.end-main

```

CODICE MAL

// Nome del programma: fat.mal

// Programmatore:

// Descrizione

// Programma che calcola il fattoriale di un numero intero.

// TOS = fattoriale

```

.label fatt1      0x00
.default goto     fine

fatt1 Z=TOS;if(Z)goto fatt15 ;else goto fatt2      // 0! = 1
fatt2 OPC=TOS
fatt3 H=OPC-1
fatt4 Z=H;if(Z)goto fatt14;else goto fatt5      // while (H !=0) OPC*H
fatt5 TOS=0
fatt6 Z=OPC;if(Z)goto fatt9;else goto fatt7      // while (OPC !=0) OPC*H
fatt7 TOS=TOS+H
fatt8 OPC=OPC-1;goto fatt6
fatt9 OPC=TOS
fatt10 MDR=H          // H = H - 1, mi appoggio a MDR
fatt11 MDR=MDR-1
fatt12 H=MDR
fatt13 TOS=0;goto fatt4
fatt14 TOS=OPC;goto fatt16          // TOS = fattoriale
fatt15 TOS=OPC=1;goto fatt16;
fatt16 MDR=TOS
fatt17 MAR=SP;wr;
fine goto fine

```

OPERAZIONI LOGICHE

INTRODUZIONE

Per quanto riguarda la INOT si è utilizzata una delle istruzioni presenti e già implementate nell'ALU che consiste nell'ottenere $\neg A$ nel risultato. Ciò significa che si prende in ingresso sul Bus A (soltanto il registro H) il dato, lo si complementa bit a bit e si somma 1. Siccome per quanto riguarda la INOT interessava soltanto la parte del complemento, si è ricorsi ad un "trucchetto" che consiste nel sommare 1 al valore di H originale, in questo modo quando si effettua il complemento bit a bit l'uno aggiunto precedentemente diventa forzatamente 0 e quindi la sommatoria automatica presente nell'istruzione ALU utilizzata non fa altro che riportare il vero complemento bit a bit del valore originale di H. In particolare dato A presente in H si effettua $H = -H$ dove $-H \equiv H + 1$.

Incrementando prima il valore di H si ha $A + 1 = -A$ ovvero $A = A + 1 + 1$ ovvero $A = A + 1 - 1$ ottenendo il risultato finale voluto di $A = A$.

In particolare per quanto riguarda l'istruzione addizionale INOT è necessaria la presenza di una parola in cima allo stack e successivamente al richiamo di tale istruzione, sarà presente in cima allo stack la NOT bit a bit della parola.

Per quanto riguarda la realizzazione della IEXOR si è utilizzato il principio di De Morgan, che consiste nel vedere tramite la tabella di verità quali sono i mintermini utilizzati per avere come risultato 1. In pratica si prende il prodotto logico (AND) di ogni mintermine (negato se preso come 0) e alla fine si realizza una somma logica (OR) di tutti i mintermini e si ottiene il risultato finale secondo la tabella di verità dell'EXOR.

La tabella di verità della EXOR è questa:

X1	X2	X1 EXOR X2
0	0	0
0	1	1
1	0	1
1	1	0

Osservando la tabella di verità della EXOR si può notare che soltanto in presenza degli ingressi X1 AND X2 OR X1 AND X2 si ottiene un 1 come risultato.

Sfruttando appunto le istruzioni possibili già implementate nell'ALU di negazione, di AND e di OR si è costruita l'EXOR.

Per quanto riguarda la IEXOR sono necessarie due parole in cima allo stack (una puntata da SP ed una da SP-1) e il richiamo di questa istruzione produrrà come risultato il POP delle due parole ed una push del valore della loro EXOR bit a bit.

In IJVM non sono presenti le istruzioni di SHIFT, che spostano di un bit verso destra o verso sinistra un numero. Lo spostamento di un bit corrisponde ad una divisione o moltiplicazione per due. Questo si ricava ragionando su sequenze di bit. Se sul numero binario 101101 eseguiamo uno shift verso sinistra otteniamo il numero 101101 0 abbiamo in pratica inserito uno zero nella posizione meno significativa. Ma ciò corrisponde a moltiplicare per due il numero, infatti, il suo valore decimale prima dello shift era 45 che diventa 90 dopo lo spostamento. L'operazione contraria avviene con lo shift verso destra. Invece di una moltiplicazione per due otteniamo una divisione, sempre per due.

CODICE C

```
#include <stdio.h>
#include <conio.h>
int shl (int a);
int shr (int a);
int main (void)
{ int tos;
  clrscr();
  printf ("Inserisci il valore da shiftare: "); scanf ("%d", &tos);
  printf ("\nIl valore shiftato a sinistra: %d %d ",tos<<1,shl(tos));
  printf ("\nIl valore shiftato a destra: %d %d ",tos>>1,shr(tos));
  printf ("\nIl valore in complemento a 1: %d ",~tos);
  printf ("\nOR esclusivo (%d EXOR 10): %d",tos,tos^10);
  getch();
  return (0);
}
int shl (int a)
{ return ( a + a);}
int shr (int a)
{ int h = 0;
  a -= 2;
  while (a >= 0) { h++;a -= 2;}
  return (h);
}
```

CODICE IJVM

```
// Nome del programma: inot.jas
// Programmatore:
// Descrizione: programma che verifica la nuova istruzione IJVM Assembly INOT.
```

```
.constant
OBJREF 0x20
.end-constant

.main
.var
DEPO
.end-var
  LDC_W OBJREF
  LDC_W OBJREF //Acquisizione valore hex in input
  INVOKEVIRTUAL getnum
  DUP //Salvataggio in DEPO e sua stampa
  ISTORE DEPO
  INVOKEVIRTUAL print
  LDC_W OBJREF
  ILOAD DEPO //Richiamo valore immesso in input
  INOT //NOT sul valore dato in input
  INVOKEVIRTUAL print //Stampa del risultato della NOT
  HALT
```

```

.end-main

// Nome del programma: iexor.jas
// Programmatore:
// Descrizione: programma che verifica la nuova istruzione IJVM Assembly IEXOR.

.constant
OBJREF 0x20
.end-constant

.main
.var
DEPO
DEPO2
.end-var
    LDC_W OBJREF
    LDC_W OBJREF //Acquisizione valori hex in input
    INVOKEVIRTUAL getnum
    ISTORE DEPO //Salvataggio del primo in DEPO
    INVOKEVIRTUAL getnum
    ISTORE DEPO2 //Salvataggio del secondo in DEPO2
    LDC_W OBJREF
    ILOAD DEPO //Richiamo valori immessi in input
    ILOAD DEPO2
    IEXOR //EXOR sui valori dati in input
    INVOKEVIRTUAL print //Stampa del risultato della EXOR
    HALT
.end-main

```

```

// Nome del programma: shift.jas
// Programmatore:
// Descrizione
// Programma che verifica le nuove istruzioni IJVM Assembly - ISHL e ISHR -

```

```

.constant
OBJREF 0x20
.end-constant

.main
.var
sx
dx
.end-var
    LDC_W OBJREF
    INVOKEVIRTUAL getnum
    ISHL
    ISTORE sx
    LDC_W OBJREF
    ILOAD sx
    INVOKEVIRTUAL print
    POP

```

```

LDC_W OBJREF
INVOKEVIRTUAL getnum
ISHR
ISTORE dx
LDC_W OBJREF
ILOAD dx
INVOKEVIRTUAL print
HALT
.end-main

```

CODICE MAL

```

// Nome del programma: inot.mal
// Programmatore:
// Descrizione
// INOT esegue il NOT bit a bit della parola in top stack.
// TOS = operando da complementare.

```

```

.label    inot1      0x00
.default  goto      fine

```

```

inot1     H=TOS+1    // Salvo in H l'operando presente sullo stack + 1
inot2     MDR=TOS=-H // Complemento bit a bit del primo elemento
           // dello stack
inot3     MAR=SP;wr; // Scrivo il complemento a 1 in top stack
fine goto fine

```

```

// Nome del programma: iexor.mal
// Programmatore:
// Descrizione
// Eseguo l'EXOR bit a bit delle prime due parole sullo stack.
// SP = 7FFF indirizzo del secondo operando.
// TOS = primo operando.

```

```

.label    iexor1     0x00
.default  goto      fine

```

```

// A è il primo operando in cima allo stack e B il secondo
iexor1    MAR=SP=SP-1;rd // in TOS A, MAR punta a SP-1
iexor2    H=OPC=TOS+1    // in OPC e H=A+1
iexor3    H=-H           // in MDR B, in H c'è A(Negato)
iexor4    OPC=OPC-1     // in OPC c'è A
iexor5    TOS=H AND MDR // TOS = A(NEG) AND B Primo mintermine
iexor6    H=MDR=MDR+1   // in H e MDR c'è B+1
iexor7    H=-H           // H = B(negato)
iexor8    H=H AND OPC   // H = B(NEG) AND A Secondo mintermine
// MDR=TOS=H=A EXOR B, aggiornamento della memoria
iexor9    MDR=TOS=H=H OR TOS;wr;
fine goto fine

```

```
// Nome del programma: ishl.mal
// Programmatore:
// Descrizione
// ISHL esegue lo shift left di un bit del top stack.
// TOS = operando.
```

```
.label    ishl1    0x00
.default  goto    fine
```

```
ishl1    H = TOS
ishl2    H = H + TOS
ishl3    MDR=TOS = H
ishl4    MAR=SP;wr;
fine goto fine
```

```
// Nome del programma: ishr.mal
// Programmatore:
// Descrizione
// ISHR esegue lo shift right di un bit del top stack.
// TOS = operando.
```

```
.label    shr1 0x00
.default  goto fine
```

```
ishr1    H=0
ishr2    TOS=TOS-1
ishr3    TOS=TOS-1; if(N) goto ishr5; else goto ishr4
ishr4    H=H+1; goto ishr2
ishr5    MDR=TOS=H
ishr6    MAR=SP;wr;
fine goto fine
```