

Corso di *Architettura degli Elaboratori* a.a. 2017/2018

Il livello della microarchitettura:
il microprogramma che implementa lJVM

Un interprete per l'ISA (again)

- **Interprete**: un programma per eseguire le istruzioni di un altro programma
 - **Ciclo fetch-execute**
 - L'interprete per l'ISA IJVM è **scritto nel control store**, con microistruzioni da 36 bit
 - **MAL**: linguaggio mnemonico per le microistruzioni
- | | |
|-----------------------------|--|
| Ciclo fetch-execute: | Fetch istruzione e incremento PC |
| | Decodifica del tipo di istruzione |
| | Eventuale caricamento di dati aggiuntivi |
| | Esecuzione dell'istruzione |

Un interprete per l'ISA in MIC

Ciclo fetch-execute – realizzazione in MIC:

- **Fetch istruzione e incremento PC**
 - Comando di fetch (che porta l'istruzione disponibile in MBR *due cicli* dopo) e incremento del registro PC
- **Decodifica del tipo di istruzione**
 - Realizzato usando il codice operativo dell'istruzione (*goto(MBR)*)
 - Eventuale caricamento di dati aggiuntivi
 - Dipende dall'istruzione che, in IJVM, ha o non ha dati aggiuntivi, quindi per certe istruzioni saranno necessarie uno o piu' ulteriori istruzioni di fetch e incremento PC
- **L'interprete esegue l'istruzione**
 - Sequenza di microistruzioni (non necessariamente contigue, ma eseguite in modo sequenziale), due istruzioni possono anche condividere porzioni di microcodice

Un esempio

- Vediamo l'esecuzione del calcolo dell'espressione a lato, si supponga che i e j abbiano offset 1 e 2 rispettivamente all'interno della procedura
- Il risultato dell'espressione si troverà sul top dello operand stack, da qui prelevato e memorizzato in i

...
 $i = 3 + j$
...

0x0	BIPUSH	0x10
0x1	3	0x03

Un esempio

- Vediamo l'esecuzione del calcolo dell'espressione a lato, si supponga che i e j abbiano offset 1 e 2 rispettivamente all'interno della procedura
- Il risultato dell'espressione si troverà sul top dello operand stack, da qui prelevato e memorizzato in *i*

...
 $i = 3 + j$
...

0x0	BIPUSH	0x10
0x1	3	0x03
0x2	ILOAD	0x15
0x3	j	0x02

Un esempio

- Vediamo l'esecuzione del calcolo dell'espressione a lato, si supponga che i e j abbiano offset 0 e 1 rispettivamente all'interno della procedura
- Il risultato dell'espressione si troverà sul top dello operand stack, da qui prelevato e memorizzato in i

...
 $i = 3 + j$
...

0x0	BIPUSH	0x10
0x1	3	0x03
0x2	ILOAD	0x15
0x3	j	0x02
0x4	IADD	0x60

Un esempio

- Vediamo l'esecuzione del calcolo dell'espressione a lato, si supponga che i e j abbiano offset 1 e 2 rispettivamente all'interno della procedura
- Il risultato dell'espressione si troverà sul top dello operand stack, da qui prelevato e memorizzato in *i*

...
 $i = 3 + j$
...

0x0	BIPUSH	0x10
0x1	3	0x03
0x2	ILOAD	0x15
0x3	j	0x02
0x4	IADD	0x60
0x5	ISTORE	0x36
0x6	i	0x01

All'avvio della macchina

Label	Operations	Comments
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 two 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

MPC = 0 (indirizzo della nop1), MBR == 0; PC == -1

Al primo ciclo non si fa nulla, alla fine il nuovo MPC = Main1

Al secondo ciclo si esegue Main1, la fetch viene eseguita sul PC incrementato di 1 e il salto viene eseguito sul valore di MBR di inizio, cioe' a 0

Al terzo ciclo si esegue di nuovo nop1, e si salta a Main1

Al quarto ciclo si esegue la Main1, la fetch viene eseguita sul PC incrementato di 1 e si salta al microcodice corrispondente alla prima istruzione IJVM

Cycle 1: Via!

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

nop1 goto Main1 // Do nothing

*In realtà questa istruzione
si trova all'indirizzo 0x0!*

All'inizio MBR = 0; PC = -1

BIPUSH

3

ILOAD

j

IADD

ISTORE

i

-----Start cycle 1-----

goto 0x2

MDR: 0x0

H: 0x0

ALU: 0 AND 0 = 0x0

Goto ADDR: 0x2

Cycle 2: esegue Main1

```
Main1  PC = PC + 1; fetch; goto (MBR) // MBR holds opcode; get next byte; dispatch  
nop1  goto Main1                  // Do nothing
```

BIPUSH

3

richiesta la lettura di

ILOAD

j

IADD

ISTORE

i

-----Start cycle 2-----

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

H: 0x0

old PC: -1

ALU: $0 + PC + 1 = 0x0$

new PC: 0x0

MEM: Fetch from byte# 0x0

requested. Processing...

Goto MBR: 0x0

Cycle 3

```
Main1  PC = PC + 1; fetch; goto (MBR) // MBR holds opcode; get next byte; dispatch  
nop1  goto Main1                  // Do nothing
```

BIPUSH

3

ILOAD

-----Start cycle 3-----

goto 0x2

j

MDR: 0x0

IADD

H: 0x0

ALU: 0 AND 0 = 0

ISTORE

MEM: Fetch value 0x10 from address 0x0

i

MBR: 0x10

Goto ADDR: 0x2

Cycle 4: torna a Main1

```
Main1  PC = PC + 1; fetch; goto (MBR) // MBR holds opcode; get next byte; dispatch  
nop1  goto Main1                  // Do nothing
```

BIPUSH

3

ILOAD

j

IADD

ISTORE

i

-----Start cycle 4-----

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

old PC: 0

H: 0

ALU: $0 + PC + 1 = 0x1$

new PC: 0x1

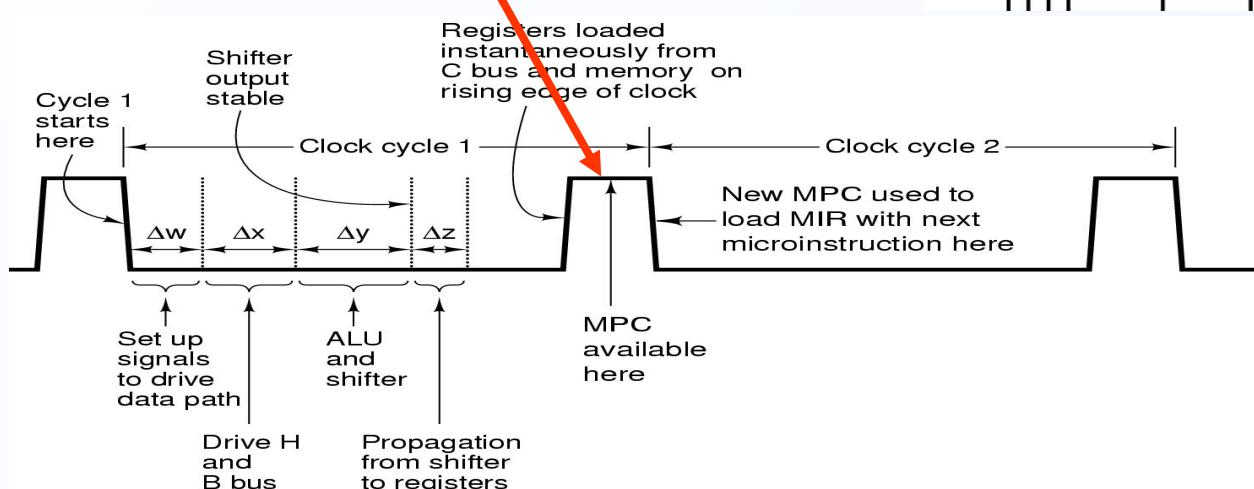
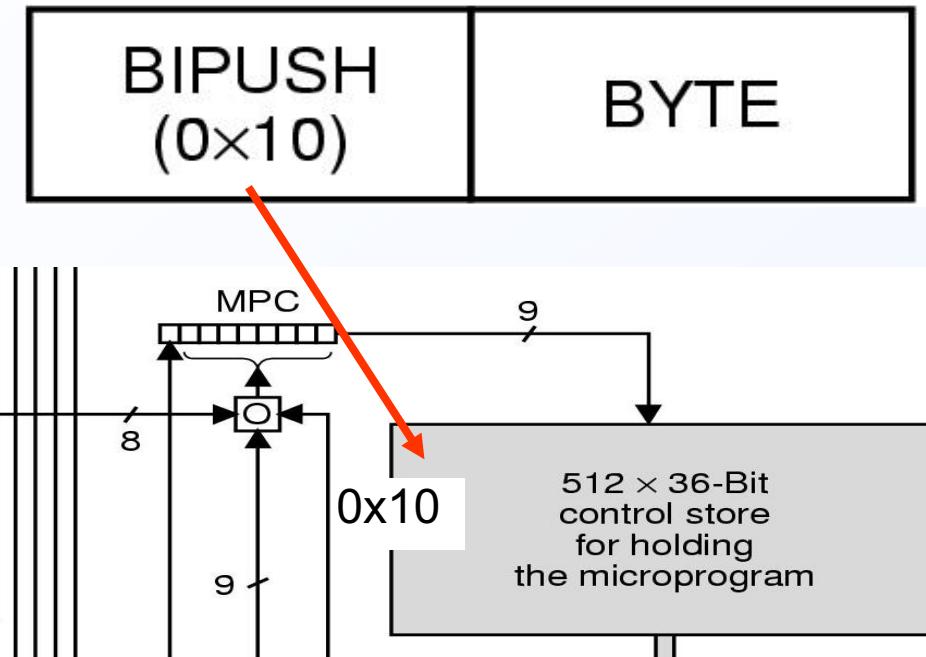
MEM: Fetch from byte# 0x1

requested. Processing...

Goto MBR: 0x10

Determinare la microistruzione successiva

- Esempio l'istruzione ISA **BIPUSH** è codificata dal byte 0x10
- Nel *Control Store* la sequenza delle microistruzioni che interpretano **BIPUSH** inizia all'indirizzo 0x10
- E' quindi importante che MPC venga caricato solo dopo che MBR, N e Z siano pronti (cioè dopo il fronte di salita del ciclo successivo)



Cycle 5: inizia la BI~~PUSH~~

```
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 // Sign-extend constant and push on stack
```

BIPUSH

3

ILOAD

j

-----Start cycle 5-----

SP=MAR=SP+1;goto 0x16

IADD

Old SP: 0x8000

ISTORE

H: 0x0

i

ALU: 0 + SP + 1 = 0x8001

new MAR: 0x8001

new SP: 0x8001

MEM: Fetch value 0x3 from
address 0x1

MBR: 0x3

Goto ADDR: 0x16

Cycle 6

```
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 // Sign-extend constant and push on stack
```

BIPUSH

3

ILOAD

j

richiesta la lettura di

IADD

ISTORE

i

-----Start cycle 6-----

PC=PC+1;fetch;goto 0x17

old PC: 1

H: 0x0

ALU: 0 + PC + 1 = 0x2

new PC: 0x2

PC: Fetch byte 2

MEM: Fetch from byte# 0x2

requested. Processing...

Goto ADDR: 0x17

Cycle 7

```
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 // Sign-extend constant and push on stack
```

BIPUSH

3

-----Start cycle 7-----

TOS=MDR=MBR;wr;goto 0x2

old MBR: 0x3

durante il ciclo

H: 0x0

MAR: 0x8001

ALU: 0 OR MBR = 0x3

MDR: 0x3

TOS: 0x3

MEM: Write value 0x3 to word#

0x20004 requested. Processing...

MEM: Fetch value 0x15 from

address 0x2

new MBR: 0x15

Goto ADDR: 0x2

*nota: MBR conterrà l'opcode
di ILOAD solo alla fine di questo
ciclo*

ILOAD

j

IADD

ISTORE

i

Cycle 8: torna a Main1

```
Main1  PC = PC + 1; fetch; goto (MBR) // MBR holds opcode; get next byte; dispatch  
nop1  goto Main1                  // Do nothing
```

BIPUSH

3

ILOAD

j

IADD

ISTORE

i

-----Start cycle 8-----

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

old PC: 0x2

H: Put 0x0

ALU: $0 + PC + 1 = 0x3$

new PC: 0x3

PC: Fetch byte 3

MEM: Write value 0x3 to
address 0x20004

MEM: Fetch from byte# 0x3
requested. Processing...

Goto MBR: 0x15

*Nota: qui si salta a 0x15
(attuale valore di MBR)
e si richiede di caricare
MBR (attendo il valore j)*

Invarianti di ciclo per Main1

Prima dell'esecuzione di Main1:

- MBR contiene il codice della prossima istruzione IJVM da eseguire
- PC è l'indirizzo dell'istruzione IJVM presente in MBR
- SP punta all'elemento in cima allo stack (coerentemente con le istruzioni IJVM)
- TOS = mem[SP]

Nota: nel caso di si arrivi dall'esecuzione di una BIPOPUSH, IADD, ISUB, IAND, IOR, DUP, IINC questa uguaglianza vale solo dal fronte di salita del clock al termine dell'istruzione Main1

Cycle 9: inizia la ILOAD

```
i load1 H = LV          // MBR contains index; copy LV to H  
i load2 MAR = MBRU + H; rd // MAR - address of local variable to push  
i load3 MAR = SP = SP + 1 // SP points to new top of stack; prepare write  
i load4 PC = PC + 1; fetch; wr // Inc PC; get next opcode; write top of stack  
i load5 TOS = MDR; goto Main1 // Update TOS
```

BIPUSH

3

ILOAD

j

IADD

ISTORE

i

-----Start cycle 9-----
H=LV;goto 0x18

LV: 0xC000

old H: 0x0

ALU: 0 OR LV = 0xC000

new H: Store 0xC000

MEM: Fetch value 0x2

from address 0x3

MBR: 0x2

Goto ADDR: 0x18

*e` un offset! va sommato
a LV per recuperare il
valore di j*

*a fine ciclo si ha
il valore di MBR dalla
memoria*

Cycle 10

```
i load1 H = LV // MBR contains index; copy LV to H  
i load2 MAR = MBRU + H; rd // MAR = address of local variable to push  
i load3 MAR = SP = SP + 1 // SP points to new top of stack; prepare write  
i load4 PC = PC + 1; fetch; wr // Inc PC; get next opcode; write top of stack  
i load5 TOS = MDR; goto Main1 // Update TOS
```

BIPUSH

3

ILOAD

j

IADD

ISTORE

i

-----Start cycle 10-----

MAR=H+MBRU;rd;goto 0x19

MBR: 0x2

H: 0xC000

ALU: H + MBRU = 0xC002

MAR: 0xC002

MEM: Read from word# 0x30008

requested. Processing...

Goto ADDR: 0x19

MAR ora contiene l'indirizzo di j e puo' quindi recuperarne il valore

MAR * 4!

e` un offset! va sommato
a LV per recuperare il
valore di j



Cycle 11

```

i load1 H = LV           // MBR contains index; copy LV to H
i load2 MAR = MBRU + H; rd // MAR = address of local variable to push
i load3 MAR = SP = SP + 1 // SP points to new top of stack; prepare write
i load4 PC = PC + 1; fetch; wr // Inc PC; get next opcode, write top of stack
i load5 TOS = MDR; goto Main1 // Update TOS

```

-----Start cycle 11-----

SP=MAR=SP+1;goto 0x1A

old SP: 0x8001

H: 0xC000

ALU: $0 + SP + 1 = 0x8002$

MAR: 0x8002

new SP: 0x8002

MEM: Read value 0x0 from
address 0x30008

MDR: 0x0

Goto ADDR: 0x1A

*In MDR, a fine ciclo,
e` disponibile in
valore di j*

*prepara lo spazio sul top
dello stack per mettere il
valore di MDR*

BIPUSH

3

ILOAD

j

IADD

ISTORE

i

Cycle 12

i load1 H = LV	// MBR contains index; copy LV to H
i load2 MAR = MBRU + H; rd	// MAR = address of local variable to push
i load3 MAR = SP = SP + 1	// SP points to new top of stack; prepare write
i load4 PC = PC + 1; fetch; wr	// Inc PC; get next opcode; write top of stack
i load5 TOS = MDR; goto Main1	// Update TOS

BIPUSH

3

ILOAD

j

IADD

ISTORE

i

Ordina il caricamento della prossima istruzione e la scrittura del valore in MDR sul top dello stack

-----Start cycle 12-----

PC=PC+1;wr;fetch;goto 0x1B

Old PC: 3

H: Put 0xC000

ALU: $0 + PC + 1 = 0x4$

New PC: 0x4

PC: Fetch byte 4

MAR: 0x8002

MDR: 0x0

MEM: Write value 0x0 to word#

0x20008 requested. Processing...

MEM: Fetch from byte# 0x4

requested. Processing...

Goto ADDR: 0x1B

Cycle 13

i load1 H = LV	// MBR contains index; copy LV to H
i load2 MAR = MBRU + H; rd	// MAR = address of local variable to push
i load3 MAR = SP = SP + 1	// SP points to new top of stack; prepare write
i load4 PC = PC + 1; fetch; wr	// Inc PC; get next opcode; write top of stack
i load5 TOS = MDR; goto Main1	// Update TOS

BIPUSH

3

ILOAD

j

IADD

ISTORE

i

-----Start cycle 13-----

TOS=MDR;goto 0x2

MDR: 0x0

H: 0xC000

ALU: 0 OR MDR = 0x0

TOS: 0x0

MEM: Write value 0x0 to address

0x20008

MEM: Fetch value 0x60 from
address 0x4

MBR: 0x60

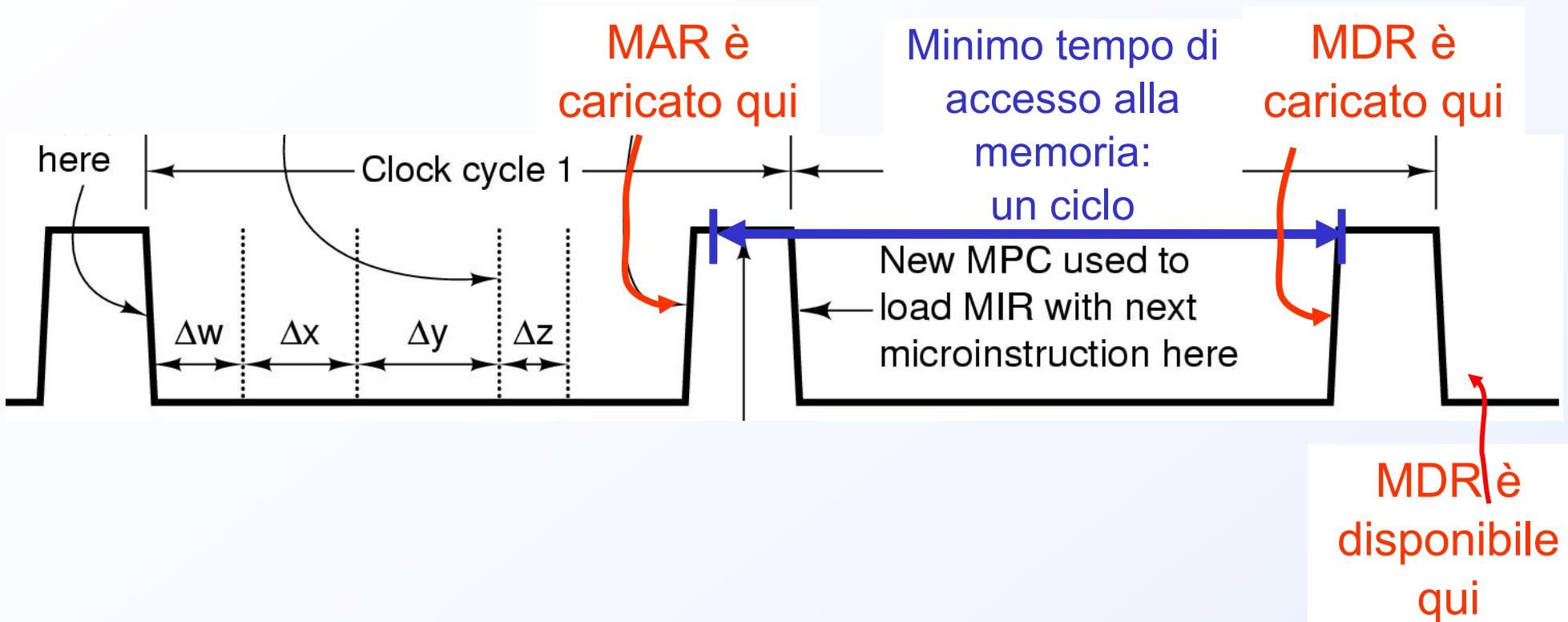
Goto ADDR: 0x2

*Aggiorna il valore del registro TOS
(Top Of the Stack) e torna al ciclo
principale*

opcode di IADD

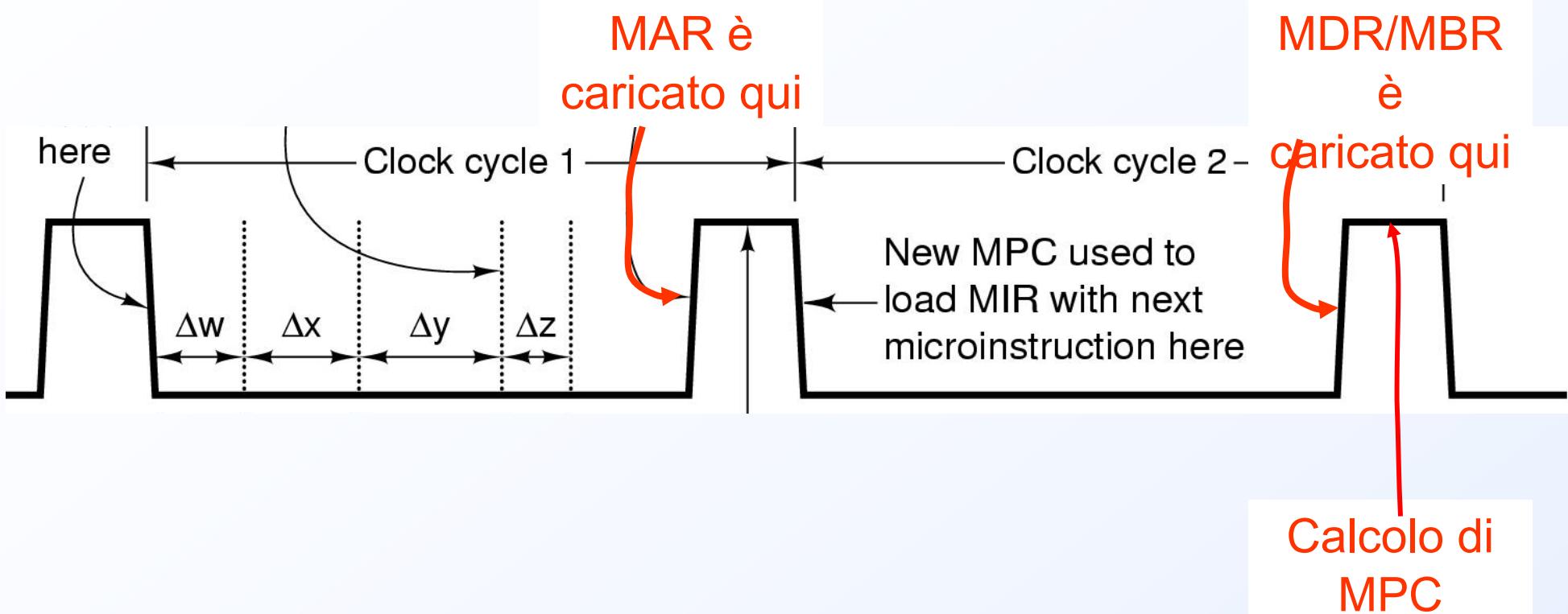
Accesso alla memoria

- Se viene inviata una richiesta di lettura di memoria nel ciclo k i dati saranno disponibili in MDR solo nel ciclo $k + 2$
- Dati sono disponibili in MDR (un ciclo di data path tra l'avvio della lettura e l'utilizzo dei dati)



Accesso alla memoria

- I dati sono disponibili in MDR (MBR) sul fronte di salita del clock nel ciclo successivo a quello in cui il comando di read (fetch) e' stato dato



Accesso alla memoria

Durante il tempo di accesso alla memoria:

- non è necessario mantenere asserito il segnale di controllo (rd, wr, fetch)
- non è necessario mantenere i valori di MAR e/o MDR, ne' di PC durante il ciclo di attesa
- i dati letti da MDR/MBR sono quelli vecchi (ciò non è un male se si desidera proprio questo, vedi esempio della microistruzione *bipush3* precedente)

Si possono anche effettuare due richieste di lettura/scrittura consecutiva

Cycle 14: torna al Main1

```
Main1  PC = PC + 1; fetch; goto (MBR) // MBR holds opcode; get next byte; dispatch  
nop1  goto Main1                  // Do nothing
```

BIPUSH

3

ILOAD

j

IADD

ISTORE

i

-----Start cycle 14-----

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

old PC: 0x4

H: 0xC000

ALU: $0 + PC + 1 = 0x5$

new PC: 0x5

MEM: Fetch from byte# 0x5

requested. Processing...

Goto MBR: 0x60

*richiede la prossima
istruzione*

Cycle 15: inizia IADD

```
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 two words; write to top of stack
```

BIPUSH

3

ILOAD

j

IADD

ISTORE

i

-----Start cycle 15-----

SP=MAR=SP-1;rd;goto 0x3

old SP: 0x8002

H: 0xC000

ALU: NOT 0 + SP = 0x8001

new SP: 0x8001

MAR: 0x8001

MEM: Read from word# 0x20004

requested. Processing...

MEM: Fetch value 0x36

from address 0x5

MBR: 0x36

Goto ADDR: 0x3

*in MBR e` arrivato
l'opcode di ISTORE*

*Richiede in memoria il valore
del top dello stack - 1 (il secondo
addendo, il primo e` gia` in TOS)*

Cycle 16

```
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 two words; write to top of stack
```

BIPUSH

3

Sposta TOS in H mentre attende
il secondo addendo che sara` in
MDR a fine di questo ciclo

ILOAD

j

IADD

ISTORE

i

-----Start cycle 16-----

H=TOS;goto 0x4

TOS: 0x0

H: 0xC000

ALU: 0 OR TOS = 0x0

H: 0x0

MEM: Read value 0x3 from

address 0x20004

Goto ADDR: 0x4

Il valore in top dello
stack - 1

Cycle 17

```
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 two words; write to top of stack
```

BIPUSH

3

*Somma il valore in MDR a H, lo
rimette in MDR e ordina la scrittura
e quindi torna a Main1*

ILOAD

j

IADD

ISTORE

i

-----Start cycle 17-----

TOS=MDR=H+MDR;wr;goto 0x2

old MDR: 0x3

H: 0x0

ALU: H + MDR = 0x3

new MDR: 0x3

TOS: 0x3

MAR: 0x8001

MEM: Write value 0x3 to word#
0x20004 requested. Processing...

Goto ADDR: 0x2

*Nota: MAR conteneva
già l'indice corretto
per la memorizzazione
del risultato*

Cycle 18: torna a Main1

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

BIPUSH

nop1 goto Main1 // Do nothing

3

*Richiede l'operando di ISTORE e
salta al suo codice*

ILOAD

-----Start cycle 18-----

j

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

IADD

old PC: 0x5

ISTORE

H: 0x0

i

ALU: $0 + PC + 1 = 0x6$

new PC: 0x6

MEM: Write value 0x3 to address

0x20004

MEM: Fetch from byte# 0x6

requested. Processing...

Goto MBR: 0x36

Cycle 19: inizia ISTORE

```
istore1 H = LV  
istore2 MAR = MBRU + H  
istore3 MDR = TOS; wr  
istore4 SP = MAR = SP - 1; rd  
istore5 PC = PC + 1; fetch  
istore6 TOS = MDR; goto Main1
```

```
// MBR contains index; Copy LV to H  
// MAR = address of local variable to store into  
// Copy TOS to MDR; write word  
// Read in next-to-top word on stack  
// Increment PC; fetch next opcode  
// Update TOS
```

BIPUSH

3

ILOAD

j

IADD

ISTORE

i

-----Start cycle 19-----

H=LV;goto 0x1C

LV: 0xC000

old H: 0x0

ALU: 0 OR LV = 0xC000

new H: 0xC000

MEM: Fetch value 0x1 from

address 0x6

MBR: 0x1

Goto ADDR: 0x1C

*A fine ciclo e` arrivato
l'offset della variabile i*

Cycle 20

```
istore1 H = LV          // MBR contains index; Copy LW 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
```

BIPUSH

3

ILOAD

*Sommo a H (base del record di
attivazione) il valore dell'offset
della variabile i*

j

IADD

ISTORE

i

-----Start cycle 20-----

MAR=H+MBRU;goto 0x1D

MBR: 0x1

H: 0xC000

ALU: H + MBRU = 0xC001

MAR: 0xC001

Goto ADDR: 0x1D

Cycle 21

```
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
```

BIPUSH

3

ILOAD

j

IADD

ISTORE

i

-----Start cycle 21-----

MDR=TOS;wr;goto 0x1E

TOS: 0x3

H: 0xC000

ALU: 0 OR TOS = 0x3

MDR: 0x3

MAR: 0xC001

MEM: Write value 0x3 to word#

0x30004 requested. Processing...

Goto ADDR: 0x1E

Preparo in MDR il valore da scrivere nella variabile i, quindi richiedo la scrittura

Nota: TOS mi fa risparmiare qui una lettura in memoria!



Cycle 22

```
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
```

BIPUSH

3

ILOAD

j

IADD

ISTORE

i

-----Start cycle 22-----

SP=MAR=SP-1;rd;goto 0x1F

old SP: 0x8001

H: 0xC001

ALU: NOT H + SP = 0x8000

MAR: 0x8000

new SP: 0x8000

MEM: Write value 0x3 to address
0x30004

MEM: Read from word# 0x20000
requested. Processing...

Goto ADDR: 0x1F

*Richiedo in memoria il nuovo valore
del top dello stack*

*L'ALU fa semplicemente
passare non modificato
il valore*

Cycle 23

```
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
```

BIPUSH

3

ILOAD

j

IADD

ISTORE

i

...

*Carico il prossimo opcode mentre
attendo che il valore richiesto in
memoria arrivi alla fine di questo ciclo*

-----Start cycle 23-----

PC=PC+1;fetch;goto 0x20

Old PC: 0x 6

H: 0xC000

ALU: $0 + B + 1 = 0x7$

New PC: 0x 7

MEM: Read value 0x0 from
address 0x20000

MEM: Fetch from byte# 0x7
requested. Processing...

Goto ADDR: 0x20

Cycle 24

```
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
```

BIPUSH

3

ILOAD

j

IADD

ISTORE

i

-----Start cycle 24-----

TOS=MDR;goto 0x2

MDR:0x0

H: 0xC000

ALU: 0 OR MDR = 0x0

TOS: 0x0

MBR: 0x0

Goto ADDR: 0x2

*Salvo il valore del top dello stack
attuale in TOS e torno a Main1*

...

Cycle 25

```
Main1 PC = PC + 1; fetch; goto (MBR) // MBR holds opcode; get next byte; dispatch
nop1 goto Main1 // Do nothing
```

BIPUSH

3

ILOAD

j

IADD

ISTORE

i

...

-----Start cycle 25-----

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

old PC: 0x7

H: 0xC000

ALU: $0 + PC + 1 = 0x8$

new PC: 0x8

MEM: Fetch from byte# 0x8

requested. Processing...

Goto MBR: 0x0

*Parte per nuove frontiere...
di microinterpretazione!!*

Il microprogramma Mic-1

```
Main1    PC = PC + 1; fetch; goto (MBR) // MBR holds opcode; get next byte; dispatch  
nop1    goto Main1                  // Do nothing
```

- ✗ **Main1:** *microistruzione che si occupa di effettuare un salto multiplo alla parte di microprogramma che microinterpreta le varie istruzioni del livello ISA*
- ✗ **NOP:** nessuna operazione

Il microprogramma Mic-1

```
|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 two 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
```

- **IADD:** pop di due parole dallo stack; push della loro somma
- **ISUB:** pop due parole dallo stack; push la loro differenza
- **IAND:** pop di due parole dallo stack; push dell'AND logico
- **IOR:** pop di due parole dallo stack; push dell'OR logico

Il microprogramma Mic-1

```
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 // Sign-extend constant and push on stack
```

- **DUP**: copia della prima parola sullo stack e push
- **POP**: cancella una parola
- **SWAP**: scambia le due parole in cima allo stack
- **BIPUSH byte**: push di un byte sullo stack

BIPUSH (0x10)	BYTE
------------------	------

Il microprogramma Mic-1

i load1	H = LV	// MBR contains index; copy LV to H	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>ILOAD (0x15)</td><td>INDEX</td></tr> </table>	ILOAD (0x15)	INDEX	
ILOAD (0x15)	INDEX					
i load2	MAR = MBRU + H; rd	// MAR = address of local variable to push				
i load3	MAR = SP = SP + 1	// SP points to new top of stack; prepare write				
i load4	PC = PC + 1; fetch; wr	// Inc PC; get next opcode; write top of stack				
i load5	TOS = MDR; goto Main1	// Update TOS				
	istore1 H = LV	// MBR contains index; Copy LV to H	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>ISTORE (0x1A)</td><td>INDEX</td><td>WORD</td></tr> </table>	ISTORE (0x1A)	INDEX	WORD
ISTORE (0x1A)	INDEX	WORD				
	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				
i inc1	H = LV	// MBR contains index; Copy LV to H	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>IINC (0x84)</td><td>INDEX</td><td>CONST</td></tr> </table>	IINC (0x84)	INDEX	CONST
IINC (0x84)	INDEX	CONST				
i inc2	MAR = MBRU + H; rd	// Copy LV + index to MAR; Read variable				
i inc3	PC = PC + 1; fetch	// Fetch constant				
i inc4	H = MDR	// Copy variable to H				
i inc5	PC = PC + 1; fetch	// Fetch next opcode				
i inc6	MDR = MBR + H; wr; goto Main1	// Put sum in MDR; update variable				

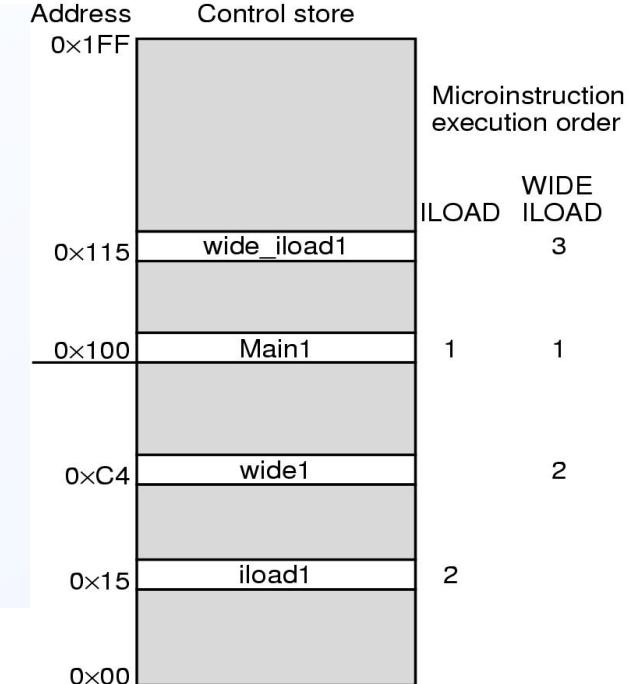
- **ILOAD varnum**: push di una variabile locale sullo stack
- **ISTORE varnum**: pop una parola dallo stack e scrittura in variabile locale
- **IINC varnum const**: somma una costante a una variabile locale

Il microprogramma Mic-1

wide1	PC = PC + 1; fetch;	Fetch operand byte or next opcode
wide2	goto (MBR OR 0x100)	Multiway branch with high bit set

wide_iload1	PC = PC + 1; fetch	// MBR contains 1st index byte; fetch 2nd
wide_iload2	H = MBRU << 8	// H = 1st index byte shifted left 8 bits
wide_iload3	H = MBRU OR H	// H = 16-bit index of local variable
wide_iload4	MAR = LV + H; rd; goto iload3	// MAR = address of local variable to push

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



WIDE: prefisso; l'istruzione seguente ha un indirizzo di 16 bit

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 local variable to store into

Il microprogramma Mic-1

```
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
```

- **LDCW index:** push di una costante dalla constant pool sullo stack
- Analoga alla ILOAD ma:
 - carica dal constant pool
 - usa un offset di 16 bit

Il microprogramma Mic-1

```

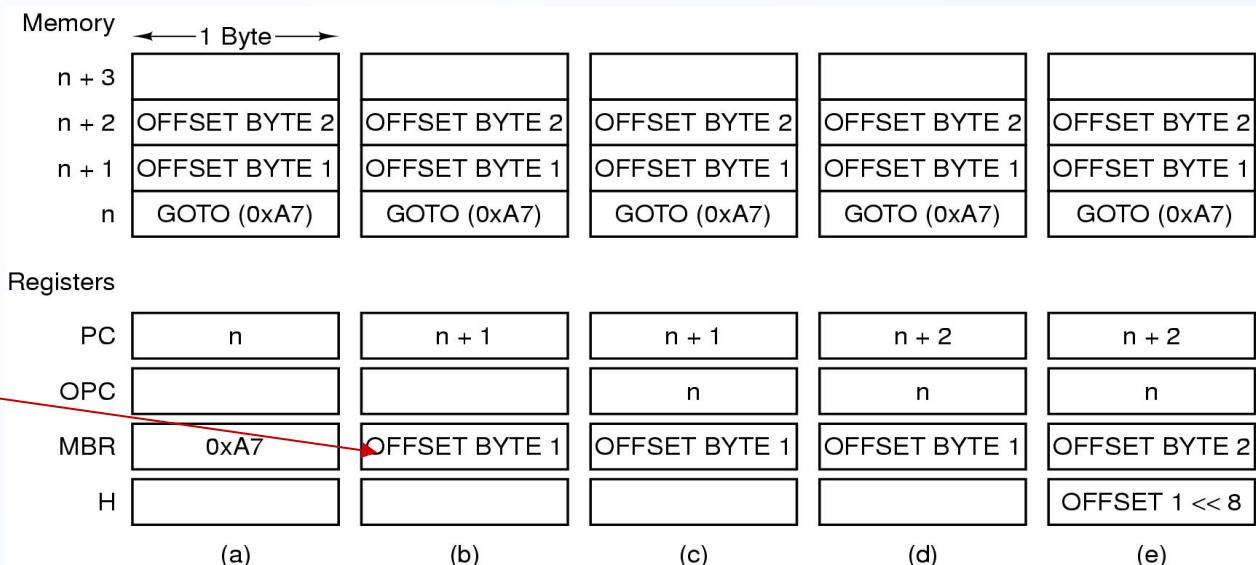
| goto1  OPC = PC - 1
| goto2  PC = PC + 1; fetch
| goto3  H = MBR << 8
| goto4  H = MBRU OR H
| goto5  PC = OPC + H; fetch
| goto6  goto Main1
  
```

```

// Save address of opcode.
// MBR = 1st byte of offset; fetch 2nd byte
// Shift and save signed first byte in H
// H = 16-bit branch offset
// Add offset to OPC
// Wait for fetch of next opcode
  
```

L'offset è relativo all'istruzione di GOTO, quindi devo prelevare tale indirizzo, il microprogramma salta a questa microprocedura abbiamo già ordinato la lettura del prossimo byte (che sarà disponibile a fine di questa istruzione)

offset con segno!



GOTO offset: branch non condizionato

Il microprogramma Mic-1

```
|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
```

passa nell'ALU senza
essere memorizzata
(attiva solo l'opportuno
bit di controllo)

stesso di goto1

```
|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
```

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

- **IFEQ offset: pop di una parola e branch se è zero**
- **IFLT offset: pop di una parola e branch se è negativa**

Il microprogramma Mic-1

```
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 gotol; 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

- **IFICMPEQ offset:** pop di due parole dallo stack; branch se sono uguali

Il microprogramma Mic-1

```
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; 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

- IFICMPEQ offset:** pop di due parole dallo stack; branch se sono uguali

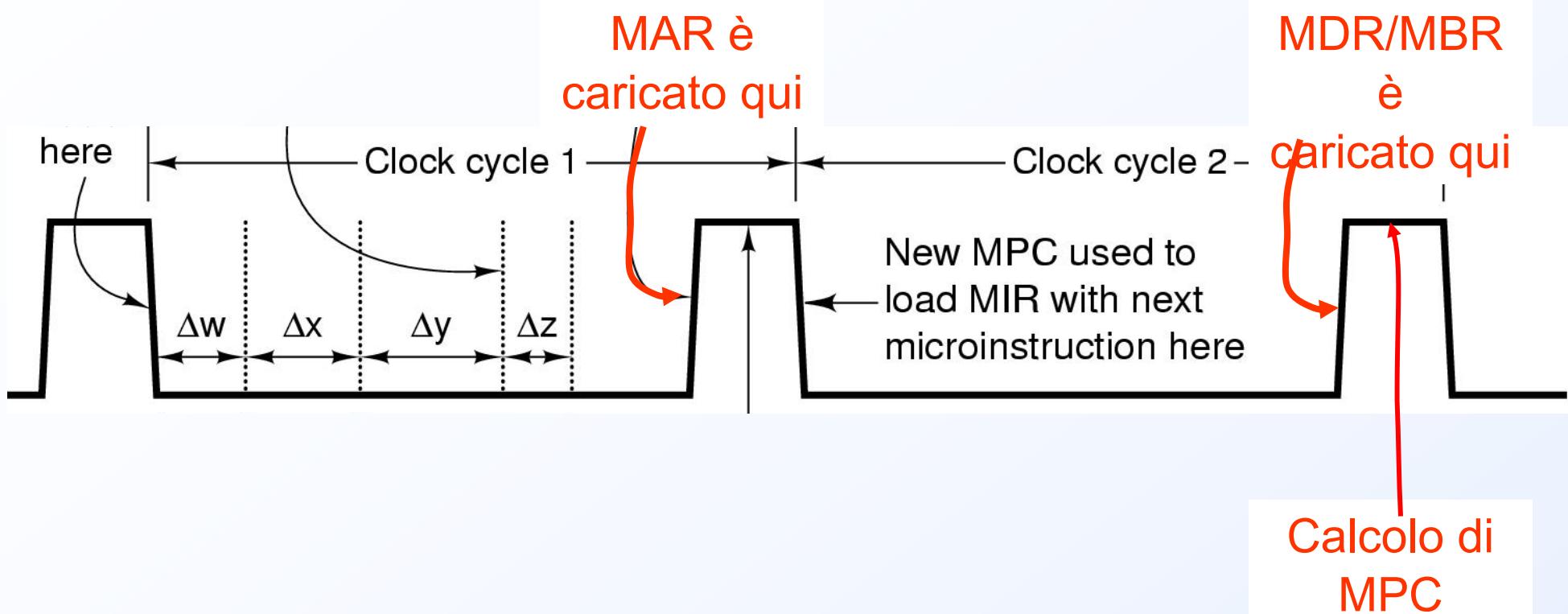
Considerazioni a posteriori

```
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 // Sign-extend constant and push on stack
```

- È possibile scambiare le istruzioni bipush1 e bipush2 preservando la funzionalità di questa parte di codice (interpretare la BIPUSH)?
- È possibile scambiare l'istruzione bipush2 con bipush3 (tranne la “*goto Main1*”)?

Accesso alla memoria (richiamo)

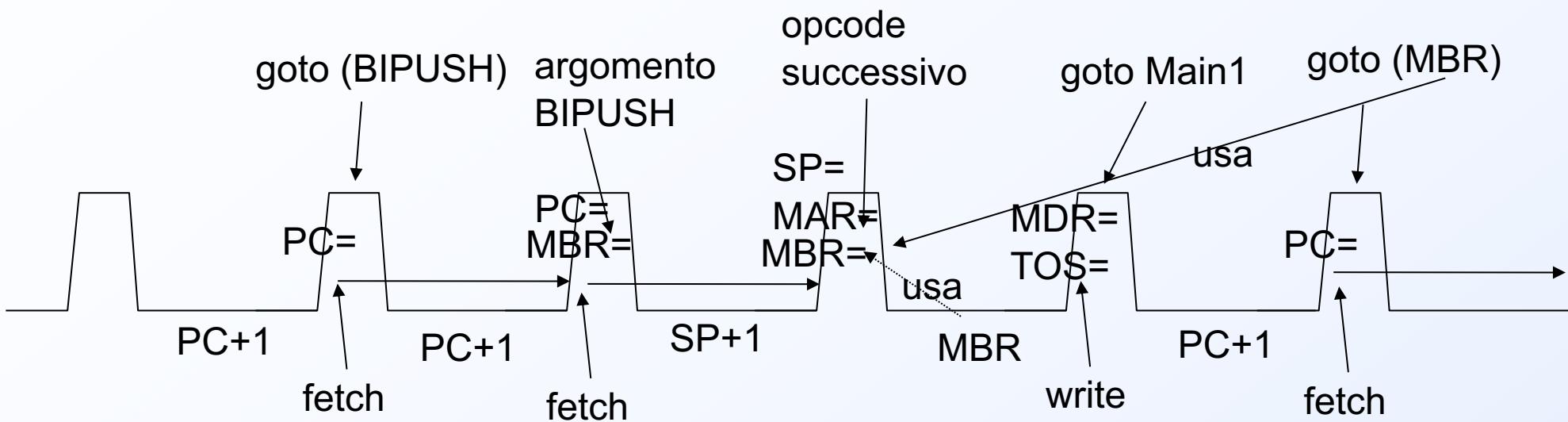
I dati sono disponibili in MDR (MBR) sul fronte di salita del clock nel ciclo successivo a quello in cui il comando di read (fetch) è stato dato



Considerazioni a posteriori

```
Main1  PC = PC + 1; fetch; goto (MBR) // MBR holds opcode; get next byte; dispatch  
bipush2 PC = PC + 1; fetch           // Increment PC, fetch next opcode  
bipush1 SP = MAR = SP + 1           // MBR = the byte to push onto stack  
bipush3 MDR = TOS = MBR; wr; goto Main1 // Sign-extend constant and push on stack  
Main1  PC = PC + 1; fetch; goto (MBR) // MBR holds opcode; get next byte; dispatch
```

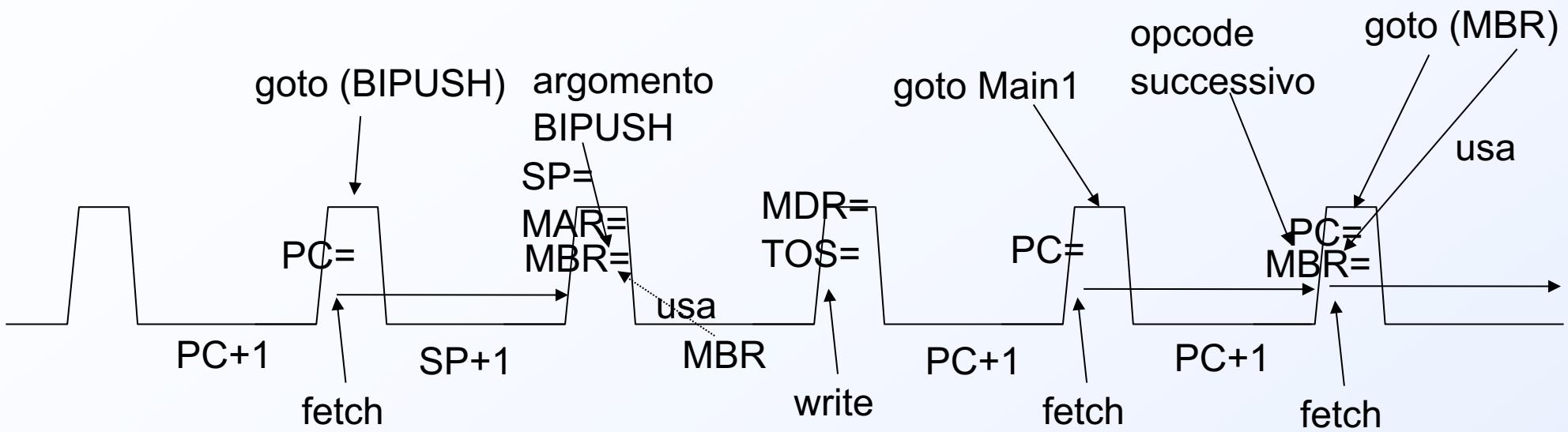
- E` possibile scambiare le istruzioni bipush1 e bipush2? **NO, perché MBR in bipush3 conterrebbe opcode successivo anzichè l'operando immediato della BIPUSH**



Considerazioni a posteriori

```
Main1  PC = PC + 1; fetch; goto (MBR) // MBR holds opcode; get next byte; dispatch  
bipush1 SP = MAR = SP + 1           // MBR = the byte to push onto stack  
bipush3 MDR = TOS = MBR; wr        // Sign-extend constant and push on stack  
bipush2 PC = PC + 1; fetch goto Main1 // Increment PC, fetch next opcode  
Main1  PC = PC + 1; fetch; goto (MBR) // MBR holds opcode; get next byte; dispatch
```

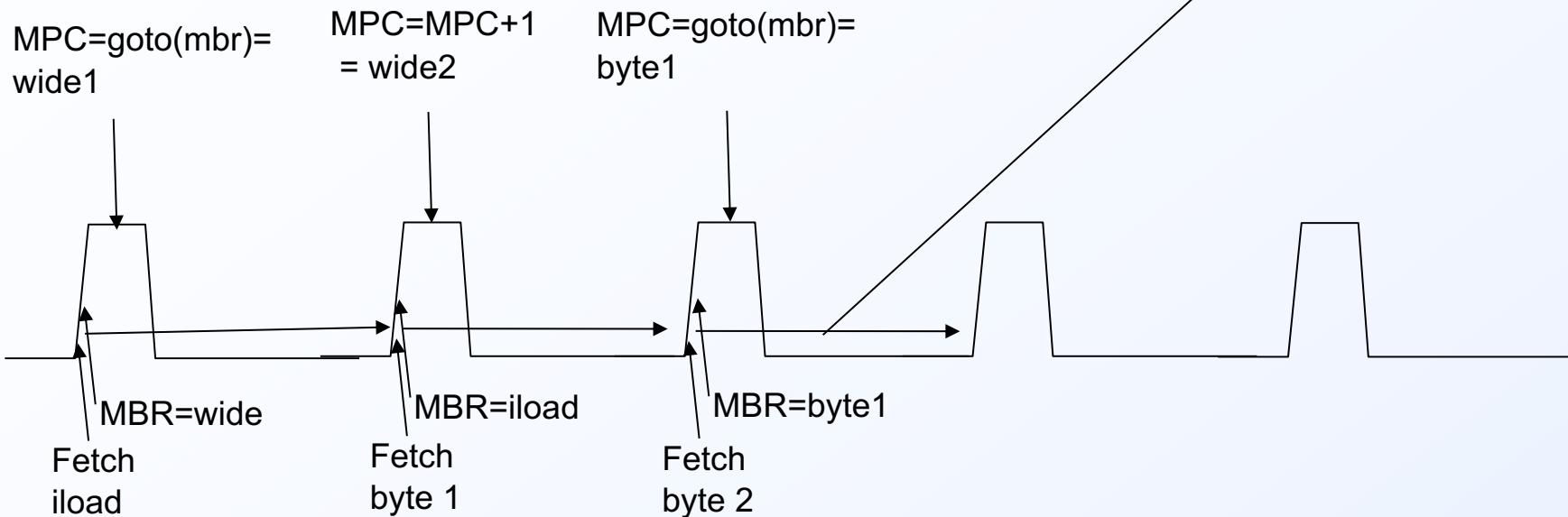
- E' possibile scambiare l'istruzione bipush2 con bipush3 (tranne la "goto Main1")? **Sì, MBR nella seconda Main1 contiene già l'opcode della istruzione successiva che ha appena caricato sul fronte di salita del clock!**



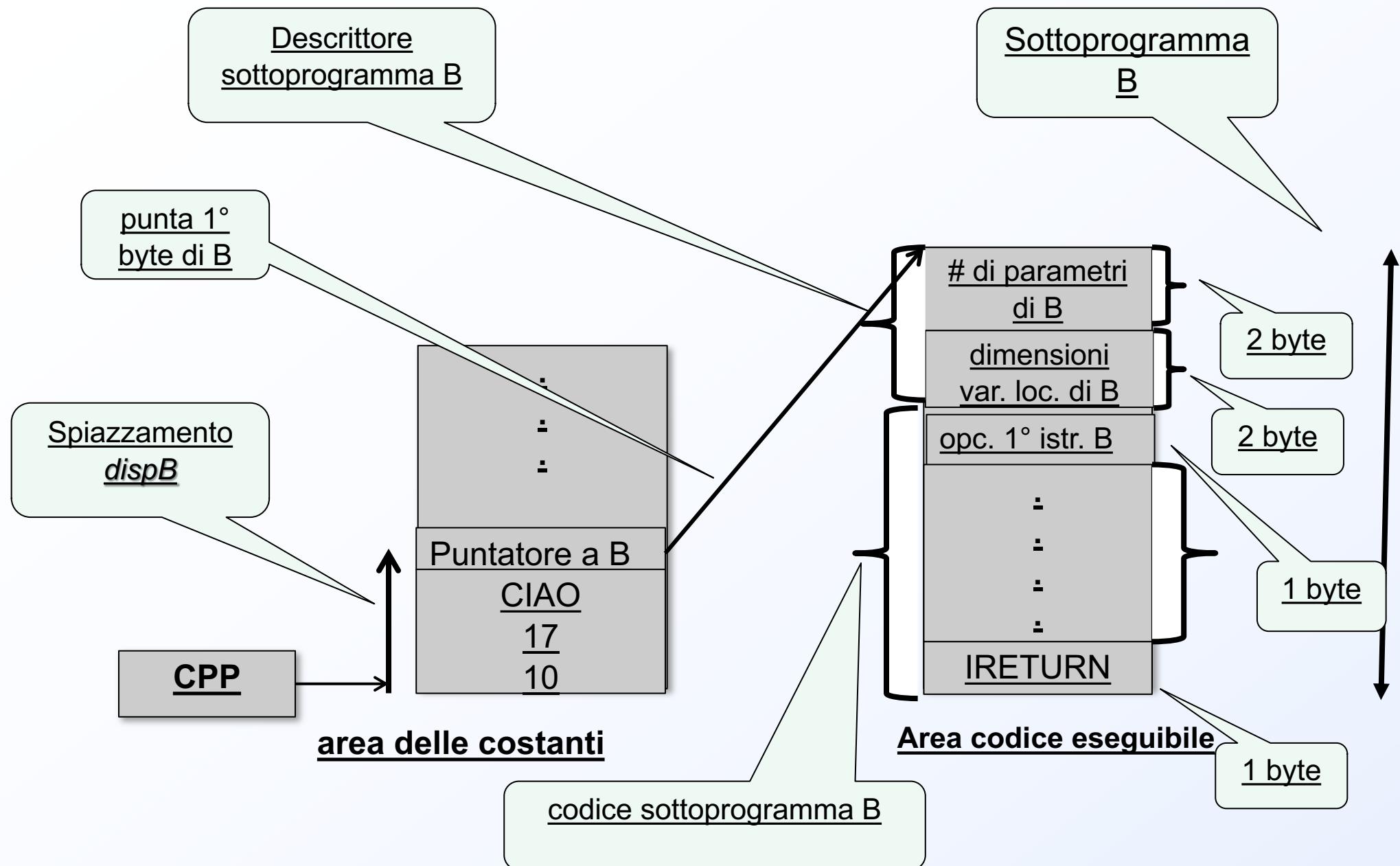
Considerazioni a posteriori

- E' corretta/necessaria l'istruzione wide2?
- Non è nè corretta nè, ovviamente, necessaria
 - wide1: $PC = PC + 1$; fetch;
 - wide2: goto (MBR OR 0x100)
- deve diventare:
 - wide: $PC = PC + 1$; fetch; goto (MBR OR 0x1000)

Si esegue la microistruzione all'indirizzo byte1
ERRATO!!



INVOKEVIRTUAL *dispB*

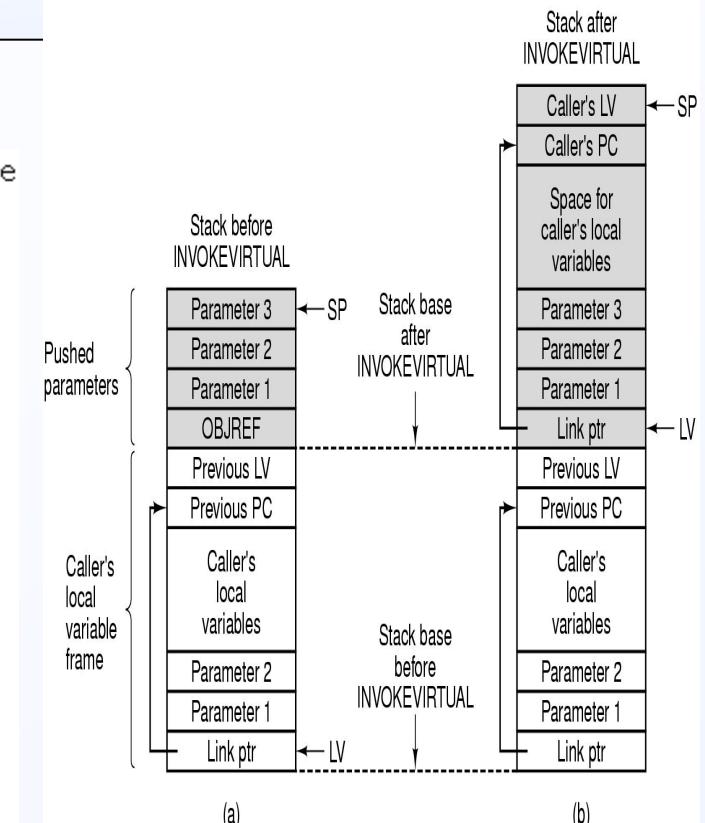


Il microprogramma Mic-1

```

invokevirtual11 PC = PC + 1; fetch      // MBR = index byte 1; inc. PC, get 2nd byte
invokevirtual12 H = MBRU << 8          // Shift and save first byte in H
invokevirtual13 H = MBRU OR H           // H = offset of method pointer from CPP
invokevirtual14 MAR = CPP + H; rd       // Get pointer to method from CPP area
invokevirtual15 OPC = PC + 1             // Save Return PC in OPC temporarily
invokevirtual16 PC = MDR; fetch         // PC points to new method; get param count
invokevirtual17 PC = PC + 1; fetch       // Fetch 2nd byte of parameter count
invokevirtual18 H = MBRU << 8          // Shift and save first byte in H
invokevirtual19 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

```



2 byte!

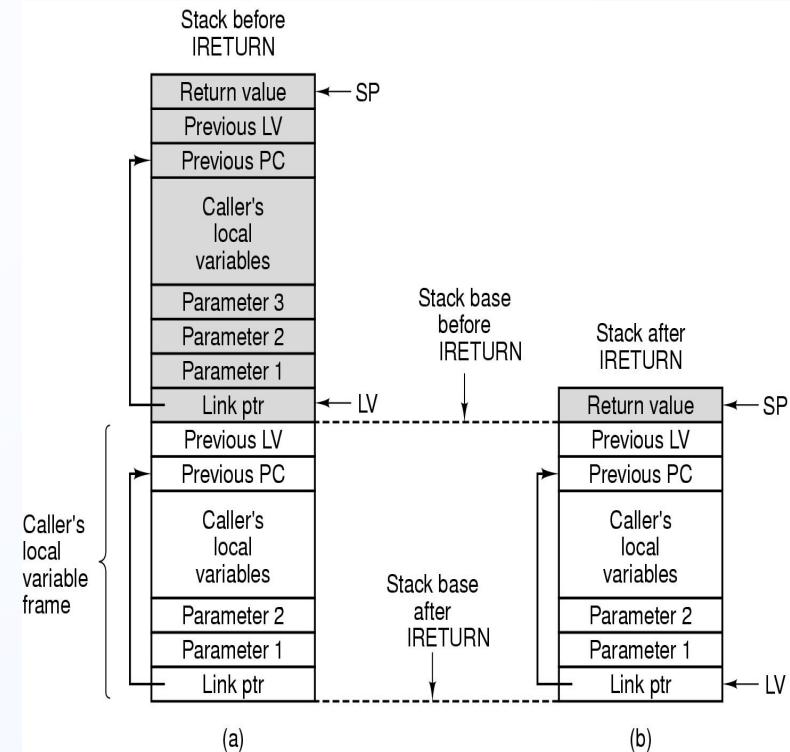
✗ **INVOKEVIRTUAL disp**

Il microprogramma Mic-1

```

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

```



- * **IRETURN:** ritorno da un metodo con un valore intero