Re7MAR

Architettura degli elaboratori 2

Relazione Lavoro di Gruppo

Simone Caggese*, Andrea Ierardi, Edoardo Favorido, Alessio Famiani
* Coordinatore

Sommario

Parte 1: Estensione del microinterprete

IREM (0x70)

Descrizione

L'istruzione IREM *(Codice operativo 0x70)* effettua la divisione euclidea tra i valori delle due parole poste in cima allo stack, di cui effettua successivamente una POP, e restituisce come risultato il resto tra i due operandi in cima allo stack. Il valore in cima allo stack corrisponde al divisore, mentre il valore sottostante in posizione SP-1 rappresenta il dividendo. Inoltre, si ipotizza che i due valori siano di segno **positivo**.

Casi significativi

Per la realizzazione della nuova operazione sono state considerate le seguenti casistiche:

- **Caso significativo 1, Dividendo maggiore del divisore**: In caso il dividendo risulti essere maggiore del divisore, si dovrà sottrarre il divisore al dividendo finché la condizione per cui il resto debba essere compreso tra zero e il divisore non verrà soddisfatta;
- **Caso significativo 2, Dividendo minore del divisore**: In caso il dividendo risulti essere minore rispetto al divisore, il resto sarà il dividendo stesso.

Il caso in cui il dividendo sia uguale al divisore non è stato considerato come caso significativo poiché non costituisce un caso particolare nell'esecuzione del microcodice.

Inoltre, dalla definizione di divisione euclidea, non si potrà mai verificare una condizione di overflow perché il resto risulta essere sempre compreso tra zero e il divisore.

Estensione del microinterprete

Modifiche apportate ai file ijvm.conf e mic1.mal

Per poter implementare IREM è stato necessario modificare i file *ijvm.conf* e *mic1.mal*.

Il file *ijvm.conf* è stato modificato aggiungendo la seguente label:

```
0x70  IREM       // Pop two words from stack; push their rest
```

Il file *mic1.mal* è stato invece modificato aggiungendo la label `.label irem1 0x70` in cima al file e il seguente microcodice:

```
irem1 MAR = SP = SP - 1 ;rd
irem2 H = TOS
irem3 OPC = MDR
irem4 MDR = MDR - H;
irem5 N = MDR ; if (N) goto resto; else goto irem3
```

```
resto MDR = TOS = OPC ; wr ; goto Main1
```

Descrizione del microcodice IREM

La parte iniziale del codice preleva principalmente il dividendo e il divisore dalla cima dello stack ed effettua la sottrazione. Arrivati verso la fine, viene effettuato un confronto, che stabilirà se dovrà essere eseguita un'ulteriore sottrazione oppure restituire direttamente il resto, scrivendolo in memoria e salvandolo sulla cima dello stack.

Descrizione di ogni istruzione

Premessa: Main1 ha già incrementato PC e ha iniziato il fetch dell'istruzione successiva alla IREM.

1. `irem1 MAR = SP = SP - 1 ;rd`: *prepara MAR alla lettura della penultima parola in cima allo stack, che corrisponde al dividendo (decrementando il puntatore alla cima dello stack, si otterrà anche l'indirizzo di dove avverrà la scrittura del risultato nella posizione SP-1);*
2. `irem2 H = TOS`: *copia in H il valore della parola situata in cima allo stack, ovvero il divisore, ottenuto prelevando il contenuto del registro ausiliario TOS. Mentre viene eseguita la lettura del dato precedente;*
3. `irem3 OPC = MDR`: *in questo ciclo, la lettura viene terminata, quindi in MDR si troverà il dividendo, successivamente verrà salvato temporaneamente il contenuto di MDR in OPC, così da avere una copia del dividendo, che verrà decrementato nella prossima istruzione;*
4. `irem4 MDR = MDR - H`: *l'ALU esegue una sottrazione tra H, il divisore, e il registro MDR, il dividendo, e salva il tutto in MDR;*
5. `irem5 N = MDR ; if (N) goto resto; else goto irem3`: *dopo aver effettuato la sottrazione, questa istruzione effettua un confronto: se il contenuto del registro MDR risulta negativo, si dovrà saltare all'etichetta resto , altrimenti si effettuerà un salto a `irem3` per sottrarre nuovamente il divisore al dividendo;*
6. `resto MDR = TOS = OPC ; wr ; goto Main1`: *scrive il risultato (resto) che è contenuto in OPC nel registro MDR per avviare successivamente una scrittura e aggiornare il registro TOS, che ha il compito di contenere una copia del valore in cima allo stack. Infine, con l'ultima istruzione, torna all'inizio del ciclo del microinterprete (MBR contiene il codice operativo della successiva istruzione IJVM).*

Debug IREM

Traccia di esecuzione A>B (utilizzo IREM_AmagB.jas)

CASO A>B A=10 B=3 → MCD =1

(utilizzo AmagB.jas)

----------Start cycle 16---------- Main1 (in control store all'indirizzo 0x2)

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

PC: Put 5

H: Put 0x0

ALU: 0 + B + 1 = 0x6

PC: Store 6

PC: Fetch byte 6

MEM: Write value 0x3 to address 0x2000C

MEM: Fetch from byte# 0x6 requested. Processing...

JMPC is set: true MBR: 0x70 ADDR: 0x0

NEXT MPC: 0x70

----------Start cycle 17---------- IREM1 (in control store all'indirizzo 0x70)

SP=MAR=SP-1;rd;goto 0xA0

SP: Put 0x8003

H: Put 0x0

ALU: NOT 0 + B = 0x8002

MAR: Store 0x8002

SP: Store 0x8002

MAR: Read from word 0x8002

MEM: Read from word# 0x20008 requested. Processing...

MEM: Fetch value 0x10 from address 0x6

MBR: Store 0x10

Goto ADDR: 0xA0

----------Start cycle 18---------- IREM2

H=TOS;goto 0x6

TOS: Put 0x3

H: Put 0x0

ALU: 0 OR B = 0x3

H: Store 0x3

MDR: Read 0xA

MEM: Read value 0xA from address 0x20008

Goto ADDR: 0x6

----------Start cycle 19---------- IREM3

OPC=MDR;goto 0xA1

MDR: Put 0xA

H: Put 0x3

ALU: 0 OR B = 0xA

OPC: Store 0xA

Goto ADDR: 0xA1

----------Start cycle 20---------- IREM4

MDR=MDR-H;goto 0xA2

MDR: Put 0xA

H: Put 0x3

ALU: NOT A + B + 1 = 0x7

MDR: Store 0x7

Goto ADDR: 0xA2

----------Start cycle 21---------- IREM5

N=MDR;if (N) goto 0x106; else goto 0x6

MDR: Put 0x7

H: Put 0x3

ALU: 0 OR B = 0x7

Goto ADDR: 0x6

----------Start cycle 22---------- IREM3

OPC=MDR;goto 0xA1

MDR: Put 0x7

H: Put 0x3

ALU: 0 OR B = 0x7

OPC: Store 0x7

Goto ADDR: 0xA1

----------Start cycle 23---------- IREM4

MDR=MDR-H;goto 0xA2

MDR: Put 0x7

H: Put 0x3

ALU: NOT A + B + 1 = 0x4

MDR: Store 0x4

Goto ADDR: 0xA2

----------Start cycle 24---------- IREM5

N=MDR;if (N) goto 0x106; else goto 0x6

MDR: Put 0x4

H: Put 0x3

ALU: 0 OR B = 0x4

Goto ADDR: 0x6

----------Start cycle 25---------- IREM3

OPC=MDR;goto 0xA1

MDR: Put 0x4

H: Put 0x3

ALU: 0 OR B = 0x4

OPC: Store 0x4

Goto ADDR: 0xA1

----------Start cycle 26---------- IREM4

MDR=MDR-H;goto 0xA2

MDR: Put 0x4

H: Put 0x3

ALU: NOT A + B + 1 = 0x1

MDR: Store 0x1

Goto ADDR: 0xA2

----------Start cycle 27---------- IREM5

N=MDR;if (N) goto 0x106; else goto 0x6

MDR: Put 0x1

H: Put 0x3

ALU: 0 OR B = 0x1

Goto ADDR: 0x6

----------Start cycle 28---------- IREM3

OPC=MDR;goto 0xA1

MDR: Put 0x1

H: Put 0x3

ALU: 0 OR B = 0x1

OPC: Store 0x1

Goto ADDR: 0xA1

----------Start cycle 29---------- IREM4

MDR=MDR-H;goto 0xA2

MDR: Put 0x1

H: Put 0x3

ALU: NOT A + B + 1 = 0xFFFFFFFE

MDR: Store 0xFFFFFFFE

Goto ADDR: 0xA2

----------Start cycle 30---------- IREM5

N=MDR;if (N) goto 0x106; else goto 0x6

MDR: Put 0xFFFFFFFE

H: Put 0x3

ALU: 0 OR B = 0xFFFFFFFE

Goto ADDR: 0x6

----------Start cycle 31---------- RESTO

TOS=MDR=OPC;wr;goto 0x2

OPC: Put 0x1

H: Put 0x3

ALU: 0 OR B = 0x1

MDR: Store 0x1

TOS: Store 0x1

MAR: Write to word 0x8002

MDR: Write 0x1

MEM: Write value 0x1 to word# 0x20008 requested. Processing...

Goto ADDR: 0x2

Traccia di esecuzione B>A (utilizzo IREM_BmagA.jas)

CASO B>A A=4 B=10 MCD→ 4

----------Start cycle 16---------- MAIN1

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

PC: Put 5

H: Put 0x0

ALU: 0 + B + 1 = 0x6

PC: Store 6

PC: Fetch byte 6

MEM: Write value 0x3 to address 0x2000C

MEM: Fetch from byte# 0x6 requested. Processing...

JMPC is set: true MBR: 0x70 ADDR: 0x0

NEXT MPC: 0x70

----------Start cycle 17---------- IREM1

SP=MAR=SP-1;rd;goto 0xA0

SP: Put 0x8003

H: Put 0x0

ALU: NOT 0 + B = 0x8002

MAR: Store 0x8002

SP: Store 0x8002

MAR: Read from word 0x8002

MEM: Read from word# 0x20008 requested. Processing...

MEM: Fetch value 0x10 from address 0x6

MBR: Store 0x10

Goto ADDR: 0xA0

----------Start cycle 18---------- IREM2

H=TOS;goto 0x6

TOS: Put 0xA

H: Put 0x0

ALU: 0 OR B = 0xA

H: Store 0xA

MDR: Read 0x4

MEM: Read value 0x4 from address 0x20008

Goto ADDR: 0x6

----------Start cycle 19---------- IREM3

OPC=MDR;goto 0xA1

MDR: Put 0x4

H: Put 0xA

ALU: 0 OR B = 0x4

OPC: Store 0x4

Goto ADDR: 0xA1

----------Start cycle 20---------- IREM4

MDR=MDR-H;goto 0xA2

MDR: Put 0x4

H: Put 0xA

ALU: NOT A + B + 1 = 0xFFFFFFFA

MDR: Store 0xFFFFFFFA

Goto ADDR: 0xA2

----------Start cycle 21---------- IREM5

N=MDR;if (N) goto 0x106; else goto 0x6

MDR: Put 0xFFFFFFFA

H: Put 0xA

ALU: 0 OR B = 0xFFFFFFFA

Goto ADDR: 0x6

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

TOS=MDR=OPC;wr;goto 0x2

OPC: Put 0x4

H: Put 0xA

ALU: 0 OR B = 0x4

MDR: Store 0x4

TOS: Store 0x4

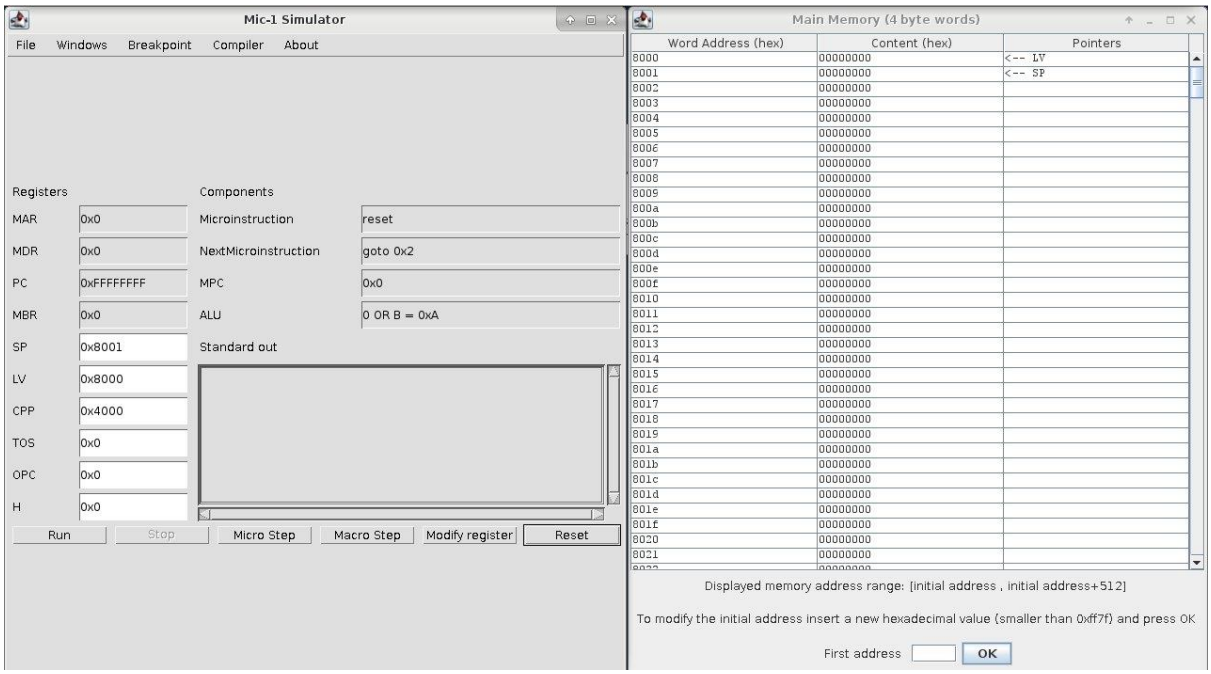
MAR: Write to word 0x8002

MDR: Write 0x4

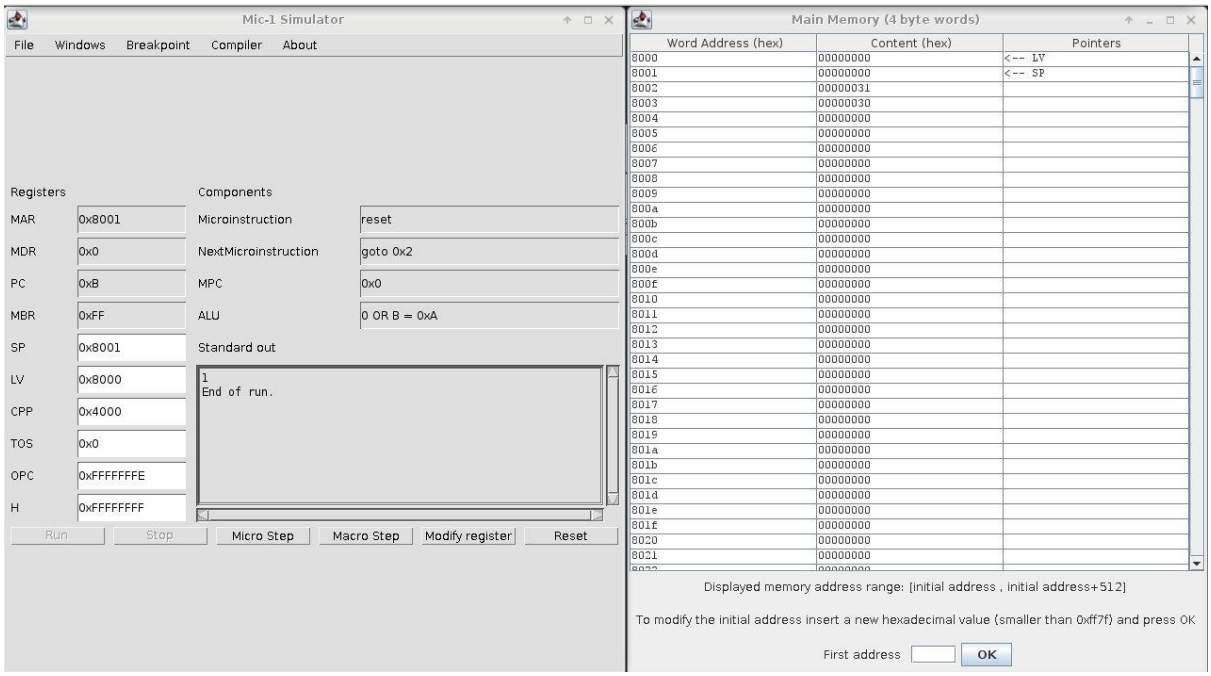
MEM: Write value 0x4 to word# 0x20008 requested. Processing...

Goto ADDR: 0x2

Registri e valori in memoria

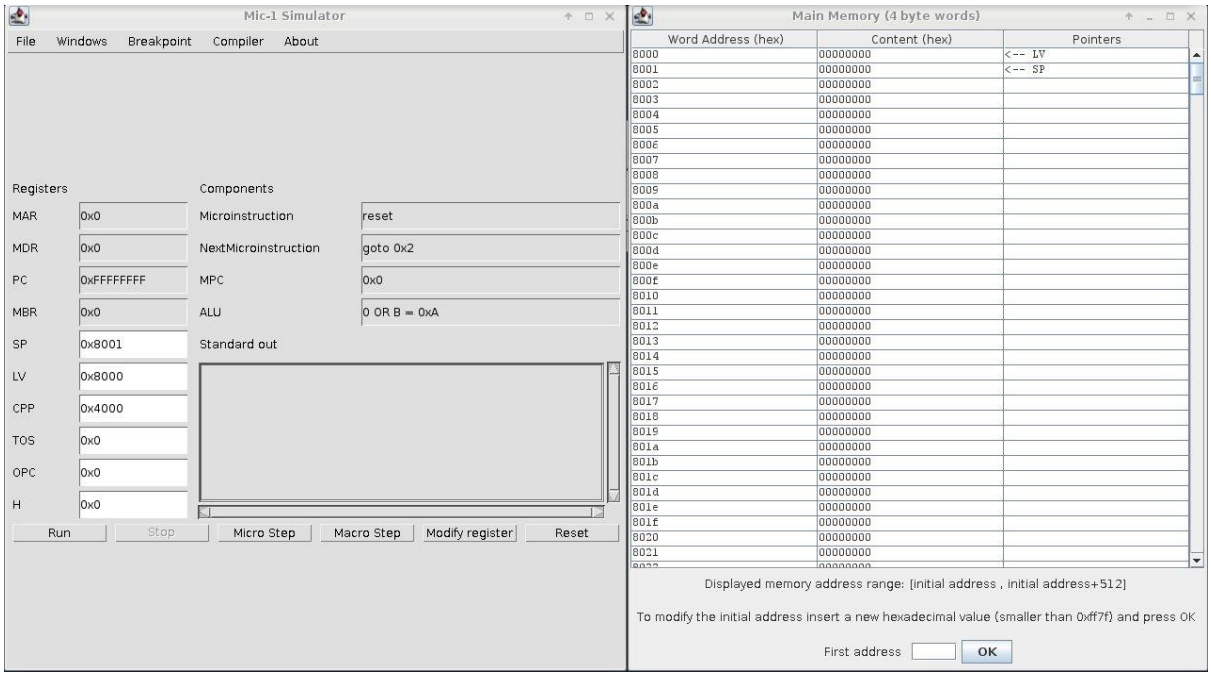


Registri e valori in memoria prima dell'esecuzione di IREM - caso A>B

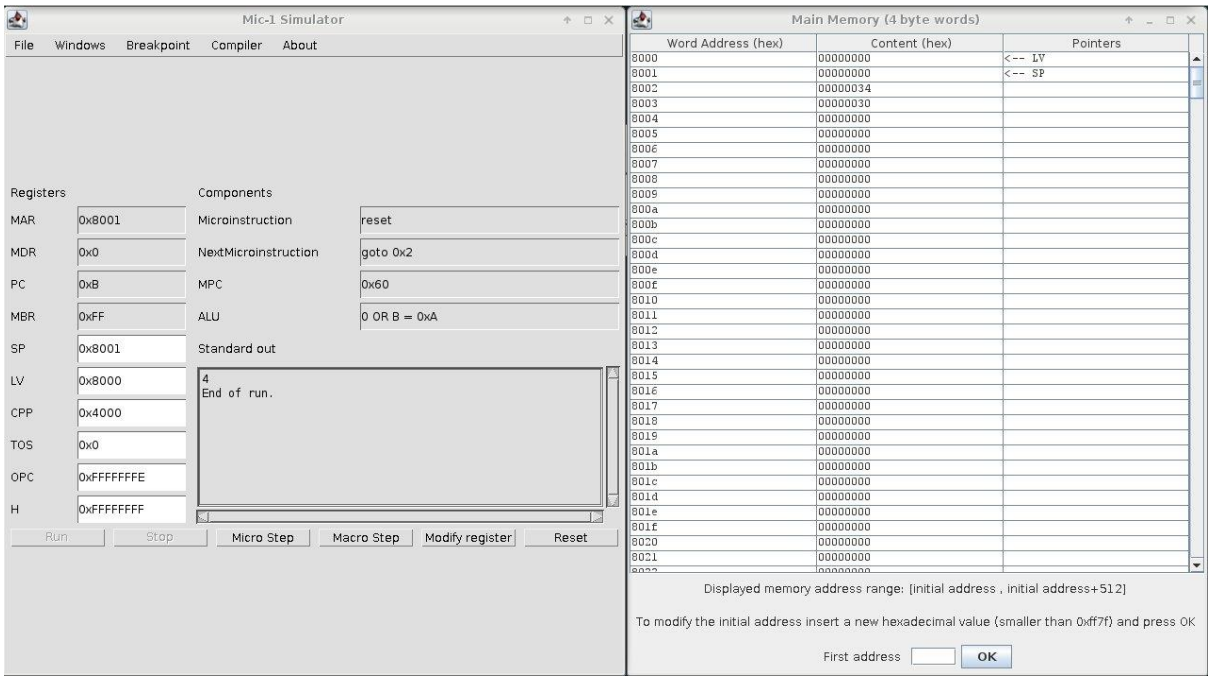


Registri e valori in memoria dopo l'esecuzione di IREM - caso A>B

Come si può vedere dagli screenshot, nell'indirizzo `8002` è contenuto il resto tra i due valori (`0x30` è una costante necessaria per la stampa a video di caratteri compresi tra 0 e 9), in questo caso tra 10 e 3 il resto è 1.



Registri e valori in memoria prima dell'esecuzione di IREM - caso B>A



Registri e valori in memoria dopo l'esecuzione di IREM - caso B>A

In quest'altro caso, il resto tra a=4 b=10 è 4, sempre all'indirizzo `8002` troviamo il resto sommato alla costante `0x30` necessaria per la stampa a video di caratteri compresi da 0 a 9.

IFGE (0x9C)

Descrizione

L'istruzione IFGE (*Codice operativo 0x9C*) rappresenta un'istruzione di salto condizionato che ha il compito di controllare se il valore posto in cima allo stack sia maggiore o uguale a zero.

Casi significativi

I casi presi in considerazioni per la realizzazione della IFGE sono:

- **Caso significativo 1, Valore maggiore o uguale a 0**
- **Caso significativo 2, Valore non maggiore**

Estensione del microinterprete

Per poter implementare IFGE si son dovuti modificare i file *ijvm.conf* e *mic1.mal*.

Modifiche apportate ai file ijvm.conf e mic1.mal

Per poter implementare IFGE si son dovuti modificare i file *ijvm.conf* e *mic1.mal*.

Il file *ijvm.conf* è stato modificato aggiungendo la seguente label:

```
0x9C  IFGE label //Pop word from stack; branch if it is >= than zero
```

Il file *mic1.mal* è stato invece modificato aggiungendo la label `.label IFGE 0x9C` in cima al file e il seguente microcodice:

```
ifge1    MAR = SP = SP - 1; rd
ifge2    OPC = TOS
ifge3    TOS = MDR
ifge4    N = OPC; if (N) goto FBIS;
FBIS     PC = PC + 1; goto F2
F2   PC = PC + 1; fetch
F3   goto Main1
T2       OPC = PC - 1; goto goto2
goto2    PC = PC + 1; fetch
goto3    H = MBR << 8
```

```
goto4   H = MBRU OR H

goto5   PC = OPC + H; fetch

goto6   goto Main1
```

Descrizione del microcodice IFGE

L'istruzione preleva semplicemente il valore della parola in cima allo stack ed effettua un confronto: se la condizione della `ifge4` (ovvero si sta verificando che il valore in input sia negativo) risulta soddisfatta, prosegue all'istruzione successiva, altrimenti salta alla label indicata.

Descrizione di ogni istruzione

PREMESSA: Main1 ha già incrementato PC e ha iniziato il fetch dell'indirizzo di memoria specificato dalla label.

1. `ifge1  MAR = SP = SP - 1; rd`: *prepara MAR alla lettura della penultima parola in cima allo stack, che corrisponde al valore in posizione SP-1 , ovvero il secondo operando sulla cima dello stack;*
2. `ifge2  OPC = TOS`: *salva temporaneamente il contenuto della cima dello stack nel registro OPC, più in particolare in OPC verrà salvata una copia del valore che si trova in cima allo stack per poterlo poi utilizzare nella condizione all'istruzione* `ifge4`*;*
3. `ifge3  TOS = MDR`: *aggiorna la cima dello stack con il valore contenuto in MDR; ovvero effettua una copia del dato richiesto in lettura dalla microistruzione precedente, in modo da avere il top dello stack aggiornato alla fine dell'esecuzione dell' IFGE;*
4. `ifge4  N = OPC; if (N) goto FBIS; else goto T2`: *effettua un confronto: se il valore prelevato in precedenza dalla cima dello stack (salvato in opc) risulta essere negativo, si salterà all'etichetta* `FBIS` *(*non deve eseguire il salto*), diversamente, a* `T2` *(deve eseguire il salto);*
5. `FBIS  PC = PC + 1; goto F2`: *fa saltare semplicemente all'istruzione successiva, saltando il primo byte dell' offset successivo, puntando perciò al codice operativo dell'operazione seguente;*
6. `F2  PC = PC + 1; fetch`: *fa puntare il programma all'OPCODE successivo;*
7. `F3  goto Main1`: *ritorna a* `Main1` *e aspetta di finire il ciclo di fetch una volta tornato in Main1;*
8. `T2  OPC = PC - 1; goto goto2`: *salva il codice operativo dell'istruzione di salto;*
9. `goto2  PC = PC + 1; fetch`*; esegue il fetch del secondo byte (in MBR si ha già il primo byte dell'offset);*
10. `goto3  H = MBR << 8`: *Shifta MBR di 8 bit a sinistra e salva il valore nel registro H;*
11. `goto4  H = MBRU OR H`: *effettua la somma tra MBRU e H e salva il nuovo valore in H;*
12. `goto5  PC = OPC + H; fetch`: *aggiorna il registro PC ed esegue il fetch in anticipo*;
13. `goto6  goto Main1`: *attende il fetch e ritorna al* `Main1`;

Debug IFGE

Traccia di esecuzione A>0 (utilizzo IFGEAmag0.jas)

CASO A>=0 A=10

(utilizzo IFGE_magg0.jas)

----------Start cycle 8----------main1 (in control store all'indirizzo 0x2)

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

PC: Put 2

H: Put 0x0

ALU: 0 + B + 1 = 0x3

PC: Store 3

PC: Fetch byte 3

MEM: Write value 0xA to address 0x20004

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

JMPC is set: true MBR: 0x9C ADDR: 0x0

NEXT MPC: 0x9C

----------Start cycle 9----------IFGE1 (in control store all'indirizzo 0x9C)

SP=MAR=SP-1;rd;goto 0xA4

SP: Put 0x8001

H: Put 0x0

ALU: NOT 0 + B = 0x8000

MAR: Store 0x8000

SP: Store 0x8000

MAR: Read from word 0x8000

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

MEM: Fetch value 0x0 from address 0x3

MBR: Store 0x0

Goto ADDR: 0xA4

----------Start cycle 10----------IFGE2

OPC=TOS;goto 0xA5

TOS: Put 0xA

H: Put 0x0

ALU: 0 OR B = 0xA

OPC: Store 0xA

MDR: Read 0x0

MEM: Read value 0x0 from address 0x20000

Goto ADDR: 0xA5

----------Start cycle 11----------IFGE3

TOS=MDR;goto 0xA6

MDR: Put 0x0

H: Put 0x0

ALU: 0 OR B = 0x0

TOS: Store 0x0

Goto ADDR: 0xA6

----------Start cycle 12----------IFGE4

N=OPC;if (N) goto 0x107; else goto 0x7

OPC: Put 0xA

H: Put 0x0

ALU: 0 OR B = 0xA

Goto ADDR: 0x7

----------Start cycle 13----------t2

OPC=PC-1;fetch;goto 0x34

PC: Put 3

H: Put 0x0

ALU: NOT 0 + B = 0x2

OPC: Store 0x2

PC: Fetch byte 3

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

Goto ADDR: 0x3

----------Start cycle 14----------goto2

PC=PC+1;fetch;goto 0x35

PC: Put 3

H: Put 0x0

ALU: 0 + B + 1 = 0x4

PC: Store 4

PC: Fetch byte 4

MEM: Fetch value 0x0 from address 0x3

MEM: Fetch from byte# 0x4 requested. Processing...

MBR: Store 0x0

Goto ADDR: 0x35

----------Start cycle 15----------goto3

H=MBR<<8;goto 0x37

MBR: Put 0x0

H: Put 0x0

ALU: 0 OR B = 0x0

H: Store 0x0

MEM: Fetch value 0x7 from address 0x4

MBR: Store 0x7

Goto ADDR: 0x37

----------Start cycle 16----------goto4

H=H OR MBRU;goto 0x38

MBR: Put 0x7

H: Put 0x0

ALU: A OR B = 0x7

H: Store 0x7

Goto ADDR: 0x38

----------Start cycle 17----------goto5

PC=H+OPC;fetch;goto 0x39

OPC: Put 0x2

H: Put 0x7

ALU: A + B = 0x9

PC: Store 9

PC: Fetch byte 9

MEM: Fetch from byte# 0x9 requested. Processing...

Goto ADDR: 0x39

----------Start cycle 18----------goto6

goto 0x2

MDR: Put 0x0

H: Put 0x7

ALU: 0 AND 0 = 0x0

MEM: Fetch value 0x10 from address 0x9

MBR: Store 0x10

Goto ADDR: 0x2

Traccia di esecuzione A<0 (utilizzo IFGEAmin0.jas)

CASO A<0 A=-120

(utilizzo IFGE_min0.jas)

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

PC: Put 2

H: Put 0x0

ALU: 0 + B + 1 = 0x3

PC: Store 3

PC: Fetch byte 3

MEM: Write value 0xA to address 0x20004

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

JMPC is set: true MBR: 0x9C ADDR: 0x0

NEXT MPC: 0x9C

----------Start cycle 9----------

SP=MAR=SP-1;rd;goto 0xA4

SP: Put 0x8001

H: Put 0x0

ALU: NOT 0 + B = 0x8000

MAR: Store 0x8000

SP: Store 0x8000

MAR: Read from word 0x8000

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

MEM: Fetch value 0x0 from address 0x3

MBR: Store 0x0

Goto ADDR: 0xA4

Legge il penultimo elemento in cima allo stack, che corrisponde al valore da verificare (10 in questo esempio). In questo modo , MDR dopo la lettura conterrà il byte da caricare sullo stack.

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

OPC=TOS;goto 0xA5

TOS: Put 0xFFFFFF88

H: Put 0x0

ALU: 0 OR B = 0xFFFFFF88

OPC: Store 0xFFFFFF88

MDR: Read 0x0

MEM: Read value 0x0 from address 0x20000

Goto ADDR: 0xA5

Copia il valore di TOS in OPC temporaneamente.

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

TOS=MDR;goto 0xA6

MDR: Put 0x0

H: Put 0x0

ALU: 0 OR B = 0x0

TOS: Store 0x0

Goto ADDR: 0xA6

Copia il nuovo valore in cima allo stack in TOS.

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

N=OPC;if (N) goto 0x107; else goto 0x7

OPC: Put 0xFFFFFF88

H: Put 0x0

ALU: 0 OR B = 0xFFFFFF88

Goto ADDR: 0x7

Viene controllato se il valore risulta negativo, in questo ciclo in OPC vi è -120, un valore negativo, quindi procede ad un'operazione di salto descritto nel ramo di vero.

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

PC=PC+1;goto 0x4A

PC: Put 3

H: Put 0x0

ALU: 0 + B + 1 = 0x4

PC: Store 4

Goto ADDR: 0x4A

Viene incrementato il valore nel registro PC in modo tale che salti il primo byte di offset.

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

PC=PC+1;fetch;goto 0x4B

PC: Put 4

H: Put 0x0

ALU: 0 + B + 1 = 0x5

PC: Store 5

PC: Fetch byte 5

MEM: Fetch from byte# 0x5 requested. Processing...

Goto ADDR: 0x4B

Viene incrementato PC in modo tale che punti all'indirizzo della prossima istruzione.

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

goto 0x2

MDR: Put 0x0

H: Put 0x0

ALU: 0 AND 0 = 0x0

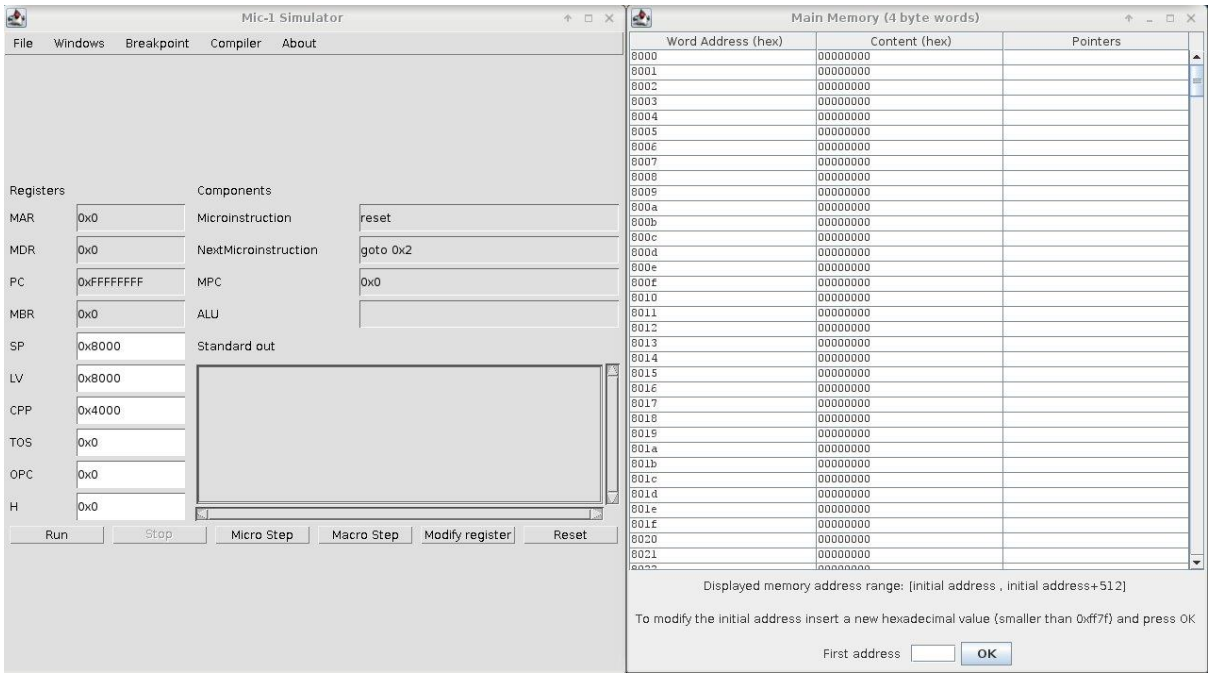
MEM: Fetch value 0x10 from address 0x5

MBR: Store 0x10

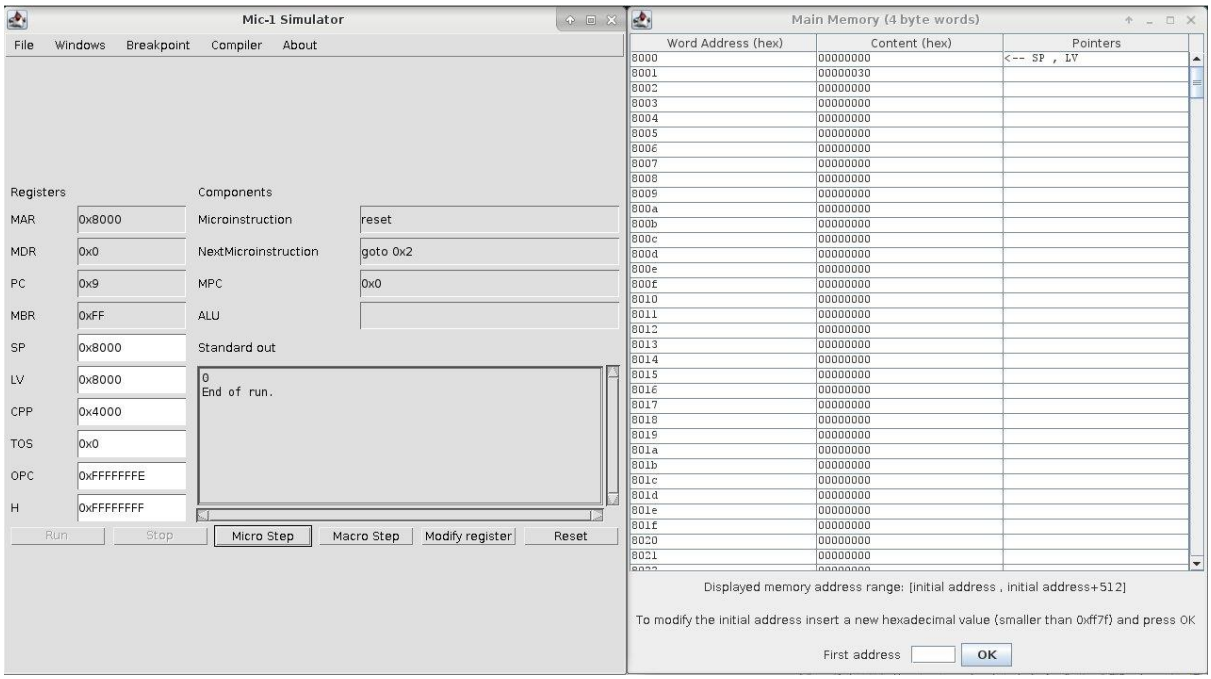
Goto ADDR: 0x2

Avviene il salto a Main1 e procede all'esecuzione della prossima istruzione.

Registri e valori in memoria

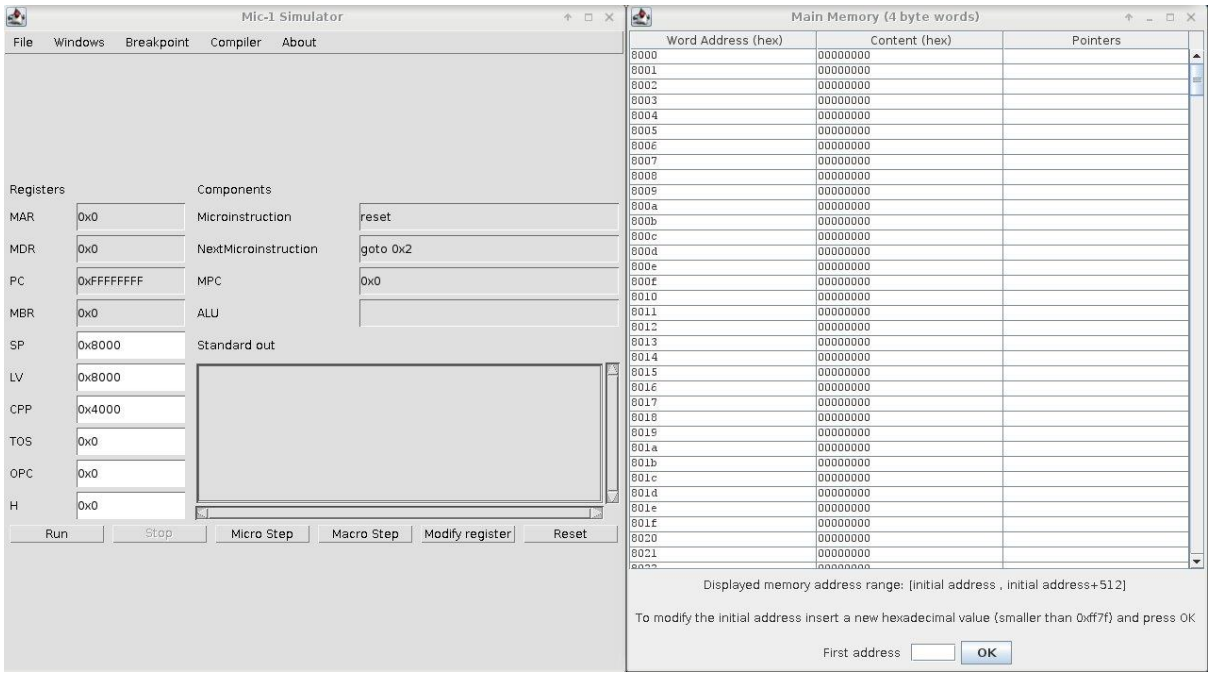


Registri e valori in memoria prima dell'esecuzione di IFGE - caso valore < 0

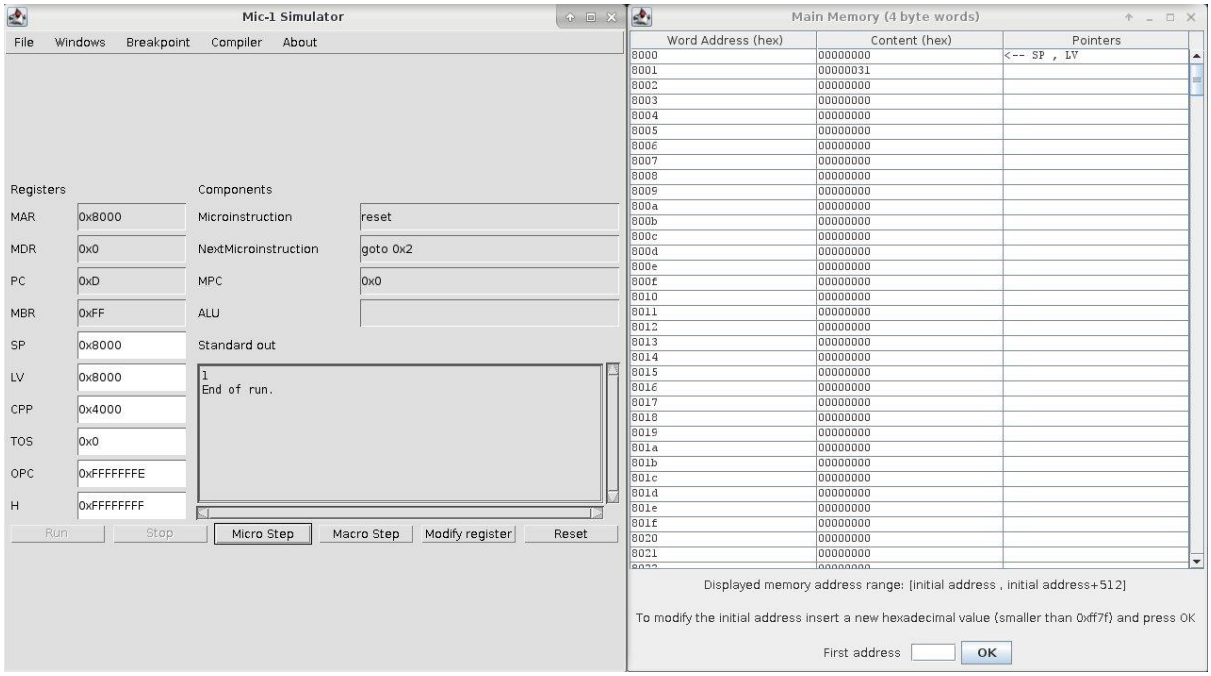


Registri e valori in memoria dopo l'esecuzione di IFGE - caso valore < 0

In questo caso è stato creato un semplice main che ritornasse il valore 0 o 1 in base all'esito del confronto del valore posto in cima allo stack. In questo caso il risultato è un numero negativo e quindi viene stampato "0" a video.



Registri e valori in memoria prima dell'esecuzione di IFGE - caso valore > 0



Registri e valori in memoria dopo l'esecuzione di IFGE - caso valore > 0

In questo caso è stato creato un semplice main che ritornasse il valore 0 o 1 in base all'esito del confronto del valore posto in cima allo stack. In questo caso il risultato è un numero negativo e quindi viene stampato "1" a video.

Parte 2: Realizzazione metodi in linguaggio IJVM

Sia nella versione ricorsiva, sia in quella iterativa, ci si è basati sul metodo di Euclide, costruito sulle seguenti proprietà:

- **MCD(a,b)= MCD(b, a%b)**, *con % viene considerato il resto della divisione euclidea)*;
- **MCD(a,b)= a**, *se b risulta uguale a zero;*

Inoltre si suppone che i valori contenuti nelle due variabili siano positivi e che NON esista il caso a==b==0, come da consegna. Tuttavia, la versione del programma proposta funziona anche in questi due casi particolari.

Massimo Comune Divisore - Versione ricorsiva

La funzione ricorsiva MCD, contenuta nel file MCD.jas, presenta le seguenti etichette:

- **`start`**: rappresenta la parte iniziale del programma e contiene la parte di codice che ha il compito di caricare il valore di 'a' e controllare se quest'ultimo sia maggiore o uguale a zero attraverso all'operazione `IFGE`, realizzata in precedenza. In caso 'a' sia effettivamente maggiore o uguale a zero, si effettuerà un salto all'etichetta `controllo_b`, altrimenti si proseguirà all'istruzione successiva, che manderà all'etichetta `neg-a`;
- **`controllo_b`**: ha il compito di caricare il valore di 'b' e, su quest'ultimo, effettua tre controlli:
  - Controlla se 'b' sia uguale a zero, in caso affermativo si verrà indirizzati alla label `equaltozero`, in caso contrario si passerà al secondo controllo;
  - Controlla se 'b' sia negativo, se lo è si salterà all'etichetta `neg-b`, altrimenti si proseguirà al terzo, e ultimo, controllo;
  - Controlla se 'b' sia maggiore o uguale a zero. In caso affermativo, si proseguirà l'esecuzione saltando all'etichetta `resto`;
- **`equaltozero`**: il segmento di codice contenuto in questa etichetta ha il compito di effettuare un pop di 'b' e restituire 'a' come risultato del resto;
- **`resto`**: semplicemente, calcola il resto utilizzando l'istruzione `IREM`, realizzata in precedenza, e richiamando il metodo stesso (MCD), e ritorna il risultato saltando all'etichetta `return`;
- **`neg-a`**: calcola e restituisce il valore assoluto di 'a';
- **`neg-b`**: calcola e restituisce il valore assoluto di 'b';
- **`return`**: ritorna il valore della variabile `risultato` lasciandola in cima allo stack.

Le variabili presenti in questo metodo sono solo due: `resto` e `risultato`.

Tabella sulla profondità di ricorsione del MCD ricorsivo

Caso	Record di attivazione	Profondità di ricorsione	Valore parametro a	Valore parametro b	Indirizzo parametro a	Indirizzo parametro b	Resto
	1	0	3 (0x03)	20 (0x14)	8007	8008	3
a > 0 b > 0 con b>a	2	1	20 (0x14)	3 (0x03)	800e	800f	2
	3	2	3 (0x03)	2 (0x02)	8015	8016	1
	4	3	2 (0x02)	1 (0x01)	801c	801d	0
	5	4	1 (0x01)	0 (0x00)	8023	8024	RETURN 1
Questo caso trattato, con b>a è il caso in cui si ha la maggior profondità di ricorsione perchè viene effettuato uno swap iniziale tra 'a' e 'b' per ritrovarsi nel caso (a>b) e poi viene richiamata ricorsivamente la funzione al fine di calcolarne l'MCD.							
a = 0 b > 0	1	0	0 (0x00)	5 (0x05)	8007	8008	0
	2	1	5 (0x05)	0 (0x00)	800e	800f	RETURN 5

In questo caso abbiamo a=0, b>0 , avverrà soltanto un richiamo ricorsivo, che calcolerà il resto tra 0 e 5 e farà un richiamo ricorsivo. Successivamente, quando entrerà nell'etichetta controlla-b, controllerà che il valore al suo interno sia diverso da 0, ma poiché è stato fatto uno swap, 'a' varrà 5 e 'b' varrà 0. Quindi entrerà nell'etichetta 'equaltozero', con una return del valore di 'a' che verrà propagato per tutti i record di attivazione che son stati creati.

a = 0 b = 0	1	0	0 (0x00)	0 (0x00)	8007	8008	RETURN 0

In questo caso, che è quello non richiesto dalla consegna, al momento dell'esecuzione dell'etichetta controlla-b, farà un controllo sul valore di 'b' che è uguale a 0, quindi entrerà nell'etichetta 'equaltozero' che tornerà il valore di 'a' che è anch'esso 0.

Nota* : Nel codice sono state aggiunte due etichette facoltative che si occupano di controllare che i valori inseriti all'interno del programma non siano negativi, in tal caso, verrebbe calcolato il valore assoluto (ricordiamo che l'MCD tra due numeri negativi è uguale all'MCD tra numeri positivi). Il motivo per cui non sono stati inseriti come esempi all'interno della profondità di ricorsione è che dovendo utilizzare la funzione leggi_interi, non è possibile inserire numeri negativi durante l'esecuzione di tale funzione.

-Due positivi:

Con 20 e 3 restituisce 1

Massima profondità di ricorsione: 4

1° record: resto tra 3 e 20 → 3

Indirizzo parametro 'a' con valore uguale a 3 (0x03) → 8007

Indirizzo parametro 'b' con valore uguale a 20 (0x14) → 8008

2° record : resto tra 20 e 3 → 2

Indirizzo parametro 'a' con valore uguale a 20 (0x14) → 800e

Indirizzo parametro 'b' con valore uguale a 3 (0x03) → 800f

3°record : resto tra 3 e 2 → 1

Indirizzo parametro 'a' con valore uguale a 3 (0x03) → 8015

Indirizzo parametro 'b' con valore uguale a 2 (0x02) → 8016

4°record : resto tra 2 e 1 → 0

Indirizzo parametro ‘a’ con valore uguale a 2 (0x02) → 801c

Indirizzo parametro ‘b’ con valore uguale a 1 (0x01) → 801d

5° record : resto tra 1 e 0→ ‘b’ è uguale a 0 quindi restituisce ‘a’

Indirizzo parametro ‘a’ con valore uguale a 1 (0x01) → 8023

Indirizzo parametro ‘b’ con valore uguale a 0 (0x00) → 8024

-Il valore di b è uguale a 0

Con 0 e 5 restituisce 5

Massima profondità di ricorsione : 1

1° record : resto tra 0 e 5 → 0

Indirizzo parametro ‘a’ con valore uguale a 0 (0x00) → 8007

Indirizzo parametro ‘b’ con valore uguale a 5 (0x05) → 8008

2° record : resto tra 5 e 0 → 0 // Quindi avviene la Swap tra i due parametri

Indirizzo parametro ‘a’ con valore uguale a 5 (0x05) → 800e

Indirizzo parametro ‘b’ con valore uguale a 0 (0x00) → 800f

- Entrambi i parametri sono = 0

Con 0 e 0 restituisce 0

Massima profondità di ricorsione: 0 poichè non viene eseguita la ricorsione

1° record : resto tra 0 e 0 → 0

Indirizzo parametro ‘a’ con valore uguale a 0 (0x00) → 8007

Indirizzo parametro ‘b’ con valore uguale a 0 (0x00) → 8008

Parte 3 (Facoltativa): MCD iterativo

Massimo Comune Divisore - Versione iterativa

Il metodo iterativo MCD, contenuto nel file MCDiterativo.jas, contiene le seguenti etichette:

- **`input`**: ha il compito di caricare sullo stack il valore di 'a' e controllare se quest'ultimo sia negativo o meno. In caso risultasse negativo, verrà effettuato un salto all'etichetta `neg_a`;
- **`input2`**: ha il compito di eseguire le stesse operazioni di `input` ma con il valore di 'b';
- **`input3`**: carica i nuovi valori di 'a' e 'b', qualora fossero stati modificati, per poterli passare all'istruzione `IF_ICMPGT`;
- **`neg_a`**: calcola e restituisce il valore assoluto di 'a';
- **`neg_b`**: calcola e restituisce il valore assoluto di 'b';
- **`a_mag_b`**: controlla che 'a' sia maggiore di 'b', in caso lo fosse, verrà effettuato un salto all'etichetta `b_control`, che controllerà se b sia uguale a zero. Se così non fosse, quindi in caso 'b' sia maggiore di 'a', si dovranno scambiare i due operandi, saltando così all'etichetta `swap`;
- **`swap`**: scambia i valori di 'a' e 'b', ricaricandoli sullo stack;
- **`b_control`**: effettua due confronti, se b risulta essere uguale a zero, allora si dovrà restituire come valore di ritorno il valore di 'a', saltando all'etichetta `output_a`. Diversamente, l'esecuzione salterà all'etichetta `resto`;
- **`output_a`**: predispone il valore di 'a' sullo stack per poi saltare all'etichetta `return`, che ritornerà il valore al metodo chiamante;
- **`resto`**: semplicemente, calcola il resto utilizzando l'istruzione `IREM`, realizzata in precedenza, e richiamando il metodo stesso (MCD), e ritorna il risultato saltando all'etichetta `return`;
- **`euclide`**: assegna il valore di 'b' ad 'a', salva il valore del resto, ottenuto dall'istruzione `IREM`, nella variabile b, e carica sullo stack i valori di 'a' e 'b' per poter saltare all'etichetta `resto`;
- **`output_b`**: predispone il valore di 'b' sullo stack per poi saltare all'etichetta `return`, che ritornerà il valore al metodo chiamante;
- **`return`**: ritorna il valore della variabile `risultato` lasciandola in cima allo stack.

Le variabili presenti in questo metodo sono solo due: `resto` e `risultato`.