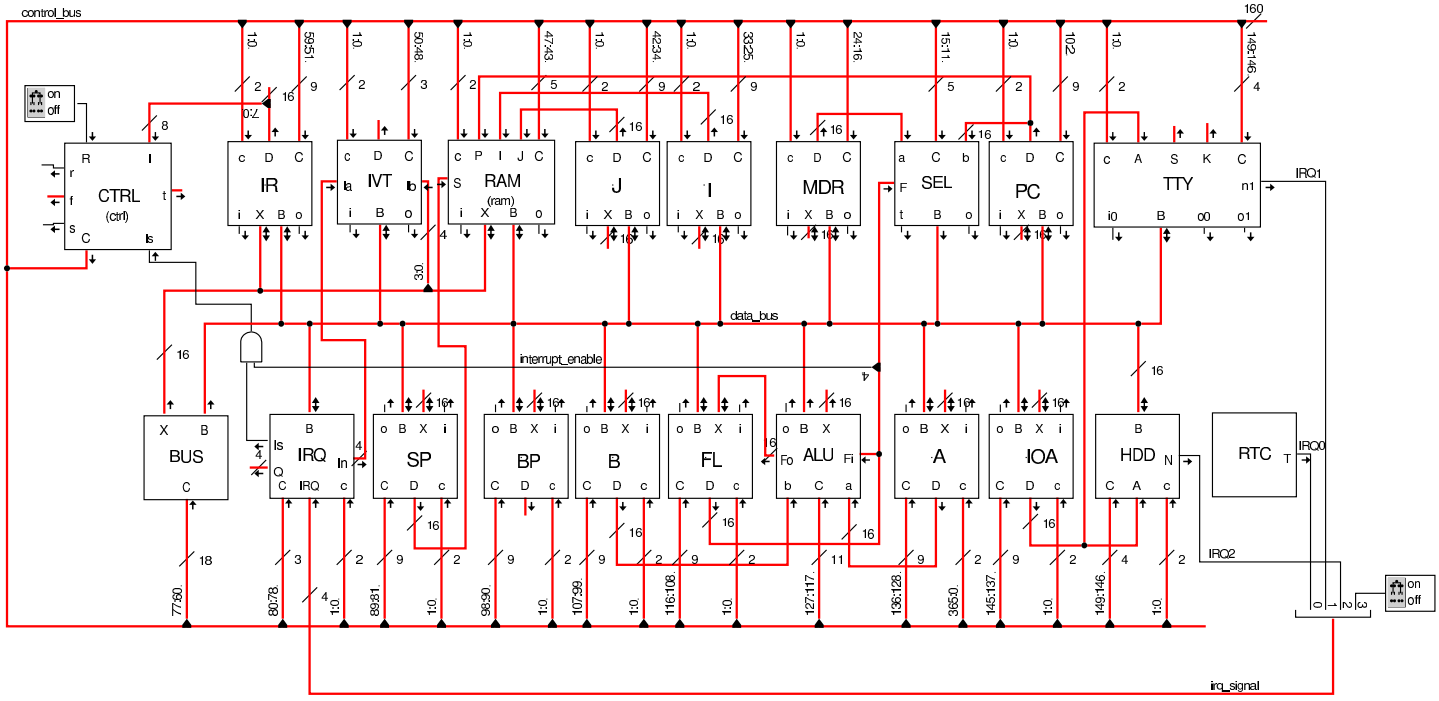
Versione K: 16 bit «little-endian»

Viene proposta un'estensione ulteriore del progetto con registri a 16 bit, pur continuando a gestire una memoria organizzata a blocchi da 8 bit. Dal momento che il compilatore di microcodice e macrocodice di Tkgate memorizza i valori a 16 bit invertendo l'ordine dei

byte (o almeno lo fa nella versione compilata per architettura x86), questa versione della CPU (che ormai è un elaboratore completo di dispositivi) è organizzata in modalità *little-endian*.

Figura u116.1. CPU dimostrativa, versione «K».



Tra le varie novità, nella figura si può osservare la presenza del registro ***BP*** il cui scopo è quello di agevolare l'uso della pila dei dati quando si eseguono chiamate di funzione. Tale registro andrebbe usato in modo simile a quello con lo stesso nome e si trova nelle CPU 8086-8088.

Registri a 16 bit

«

I registri di questa versione della CPU dimostrativa sono da 16 bit, ma sono divisi in due byte, i cui contenuti sono accessibili separatamente. Inoltre, è possibile incrementare e ridurre il valore di tali registri, di una o di due unità.

Figura u116.2. Aspetto esterno dei registri a 16 bit.

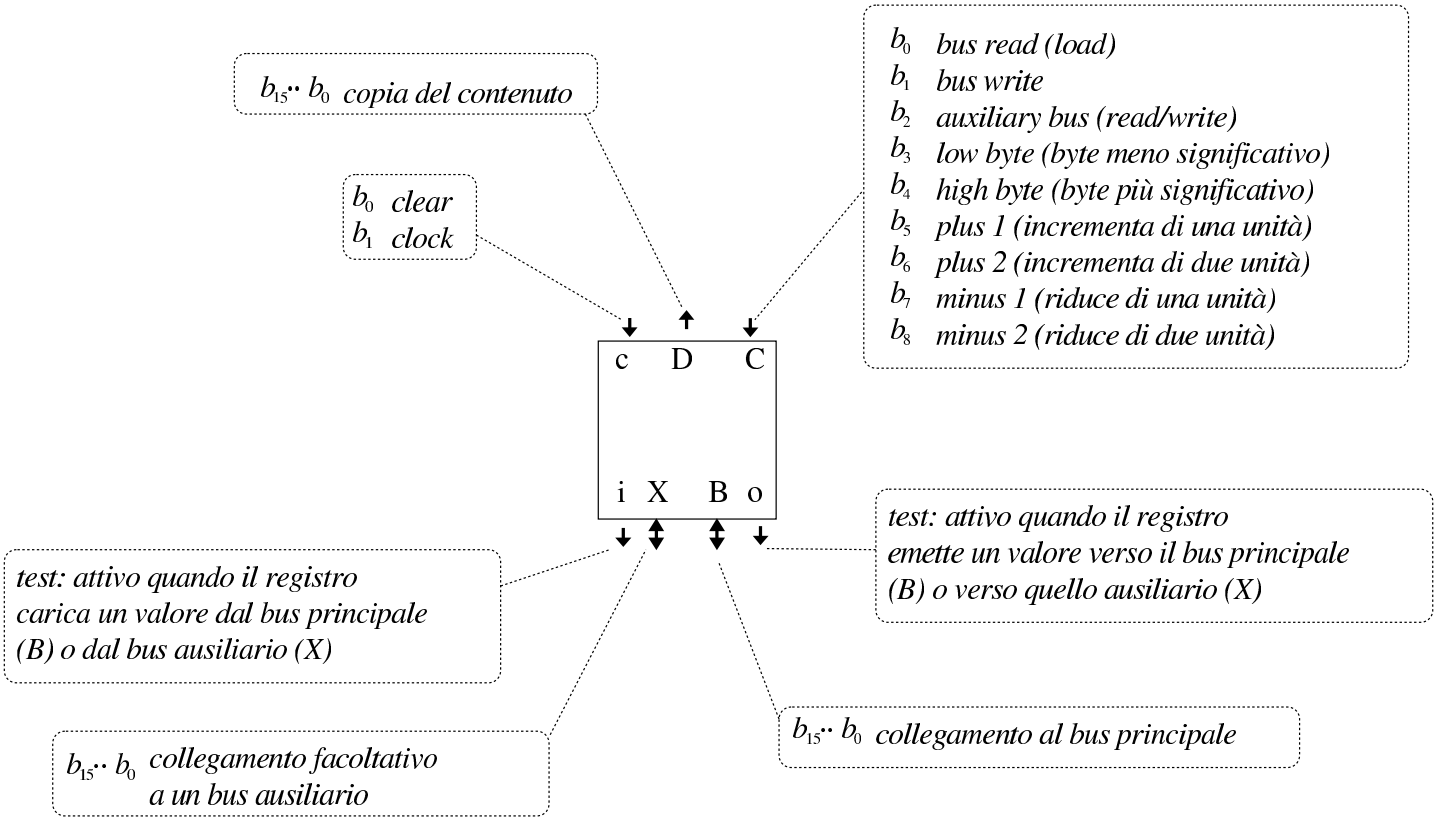


Figura u116.3. Struttura dei registri a 16 bit.

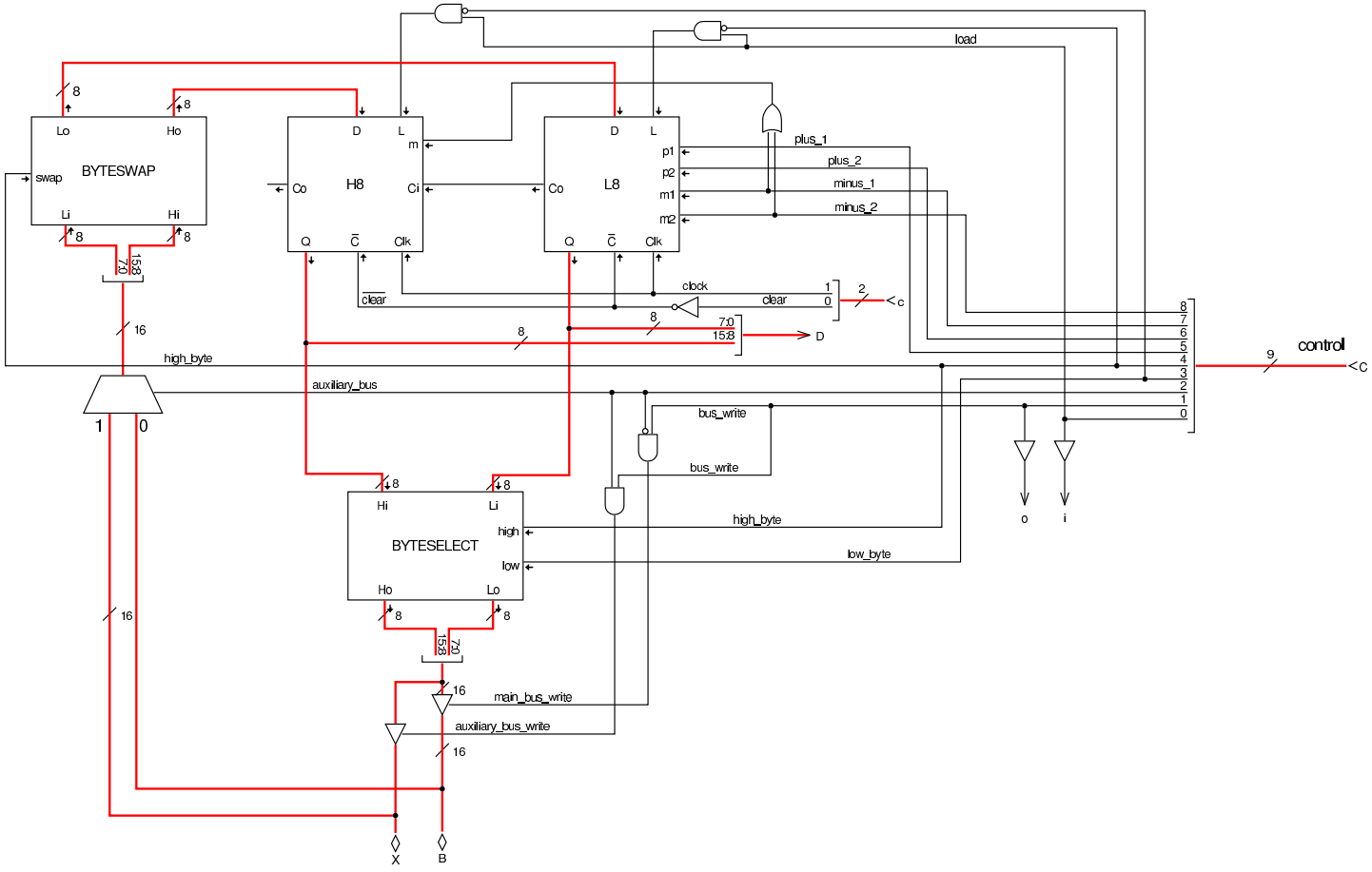
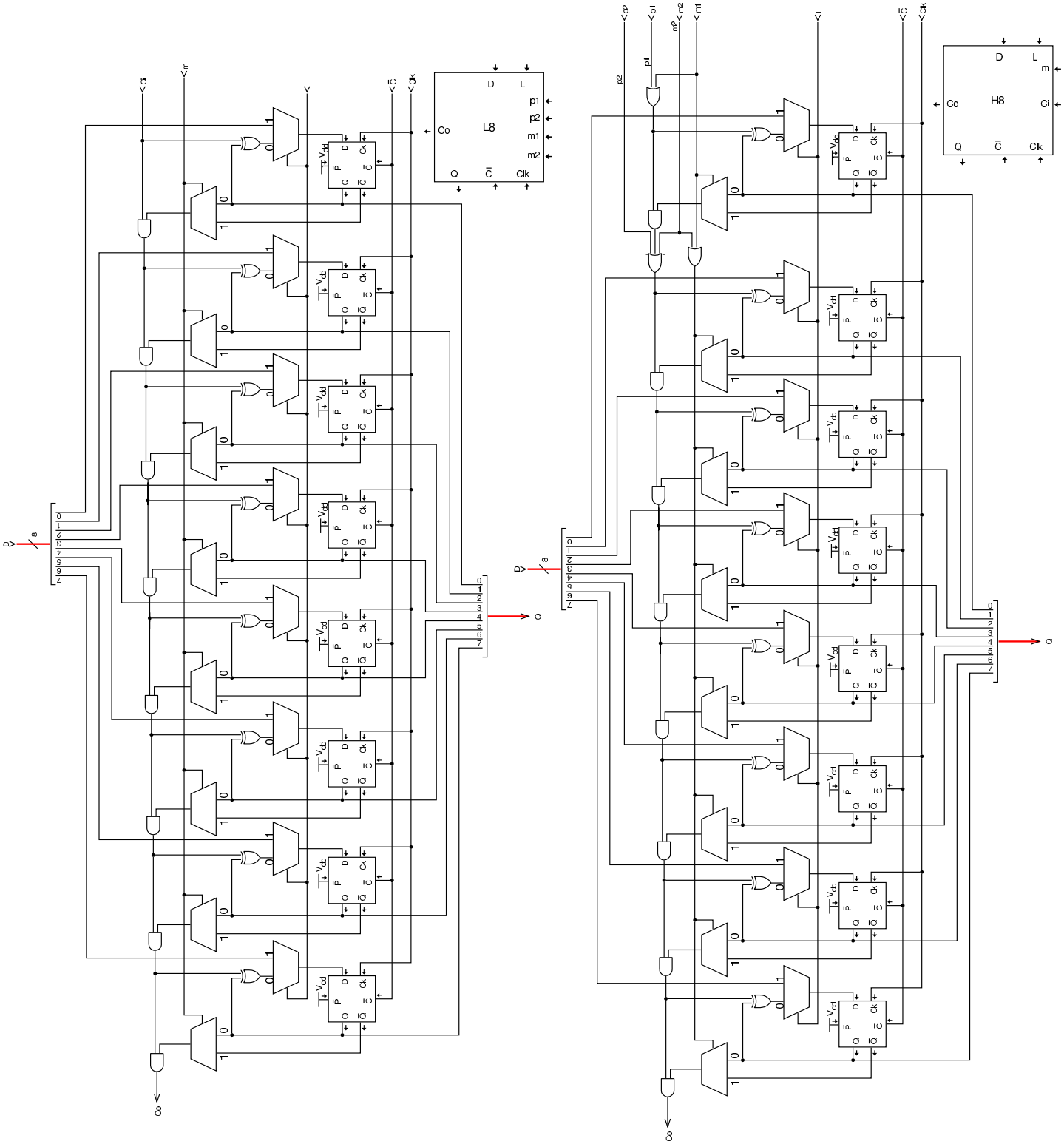


Figura u116.4. Struttura dei moduli **H8** e **L8**.

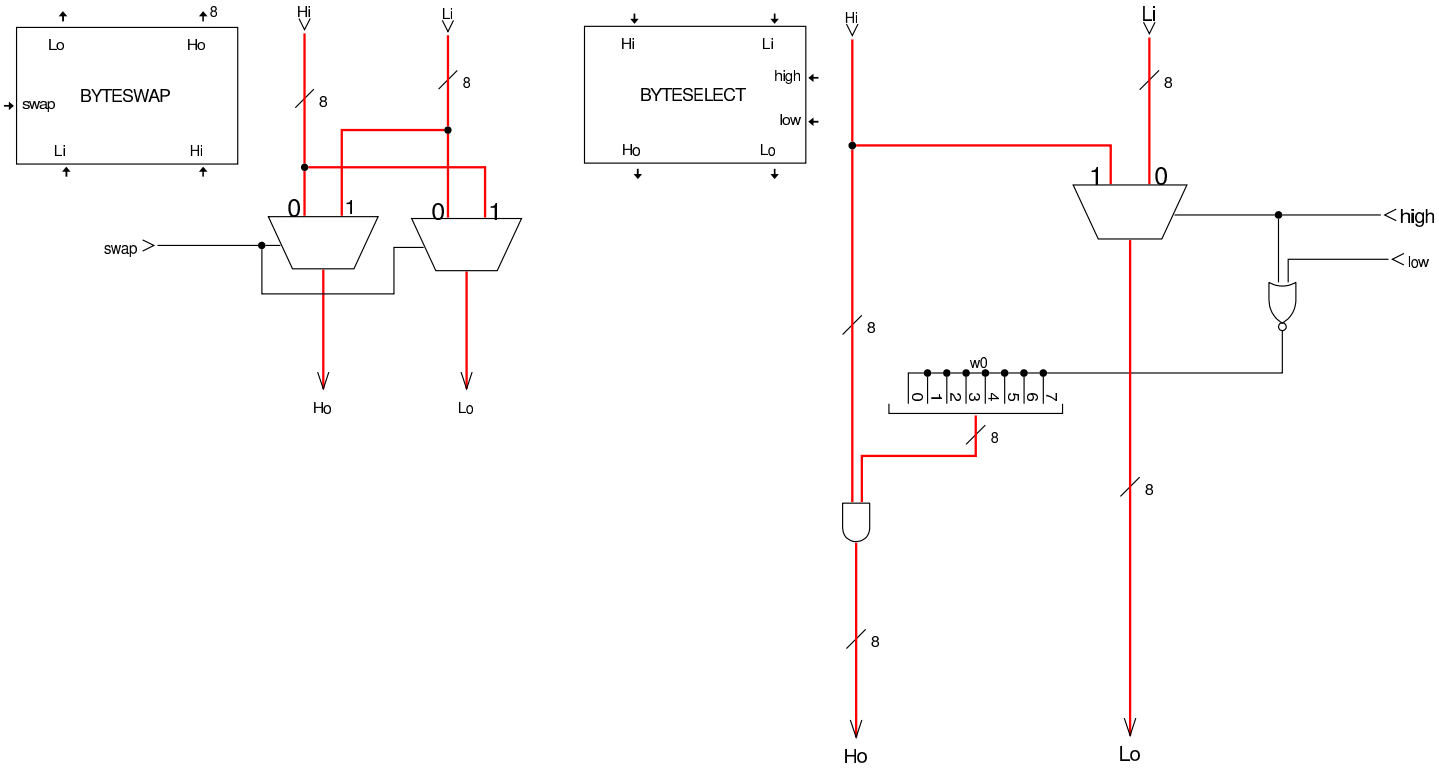


Il modulo **BYTESELECT** contenuto nei registri a 16 bit, serve a limitare la lettura del contenuto del registro a soli 8 bit, scegliendo tra

la parte meno significativa o quella più significativa. Per esempio, se il registro contiene il valore $ABCD_{16}$ e si seleziona il byte meno significativo, si ottiene $00CD_{16}$, al contrario, se si seleziona il byte più significativo, si ottiene $00AB_{16}$.

Quando si inserisce un valore nel registro, è possibile scrivere solo nella la porzione inferiore o solo in quella superiore. Per questo si utilizza il modulo **`BYTESWAP`** che permette di scambiare i byte del valore recepito dal bus; poi sta ai moduli **`L8`** o **`H8`** attivarsi per caricare la porzione rispettiva se ciò è richiesto dai segnali del bus di controllo. In pratica, quando si sta ricevendo dal bus dati un valore a 8 bit che deve essere collocato nella porzione superiore del registro, i segnali del bus di controllo attivano lo scambio dei byte con il modulo **`BYTESWAP`** e attivano il caricamento dell'informazione solo nel registro ***H8***.

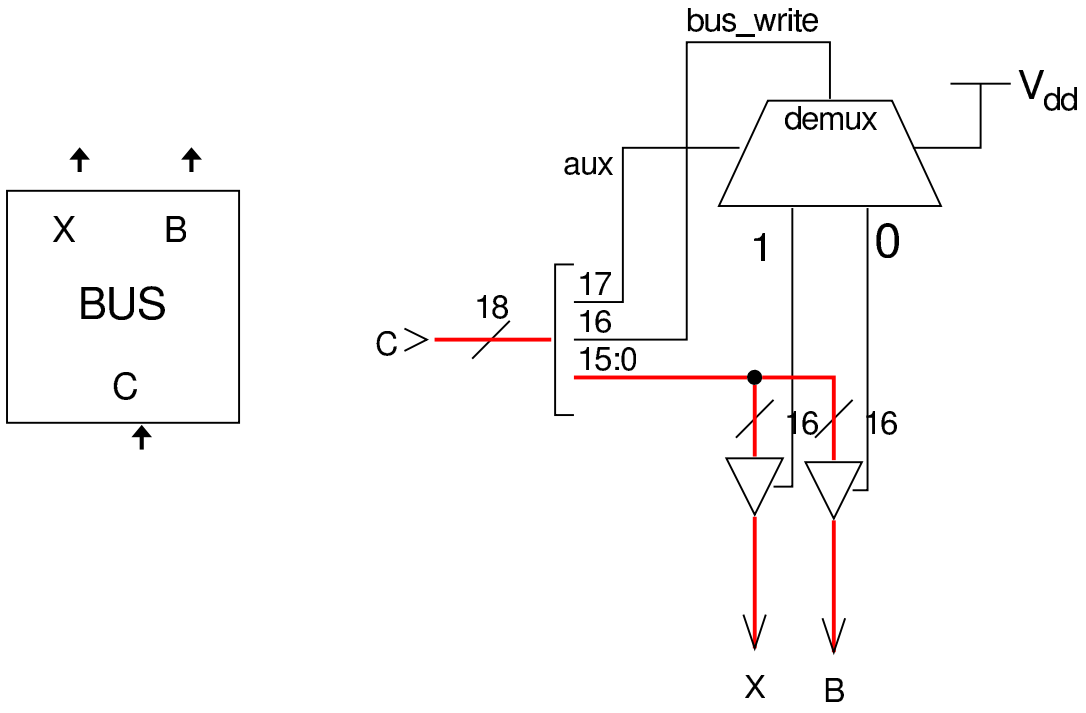
Figura u116.5. Struttura interna dei moduli **`BYTESWAP`** e **`BYTESELECT`** per lo scambio o la selezione dei byte.



Modulo «BUS»

Rispetto alla versione precedente della CPU dimostrativa, si aggiunge un modulo molto semplice che consente al sistema di controllo di inserire un valore nel bus, scegliendo tra quello principale (***B***) o quello ausiliario (***X***).

Figura u116.6. Modulo **BUS**.



Modulo «ALU»

«

L'unità ALU è stata ridisegnata, allo scopo di gestire valori a 16 bit e per poter disporre di qualche funzionalità in più. In particolare, si distinguono indicatori diversi per le operazioni che riguardano 8 bit rispetto a quelle che vanno intese a 16.

Figura u116.7. Struttura complessiva della ALU.

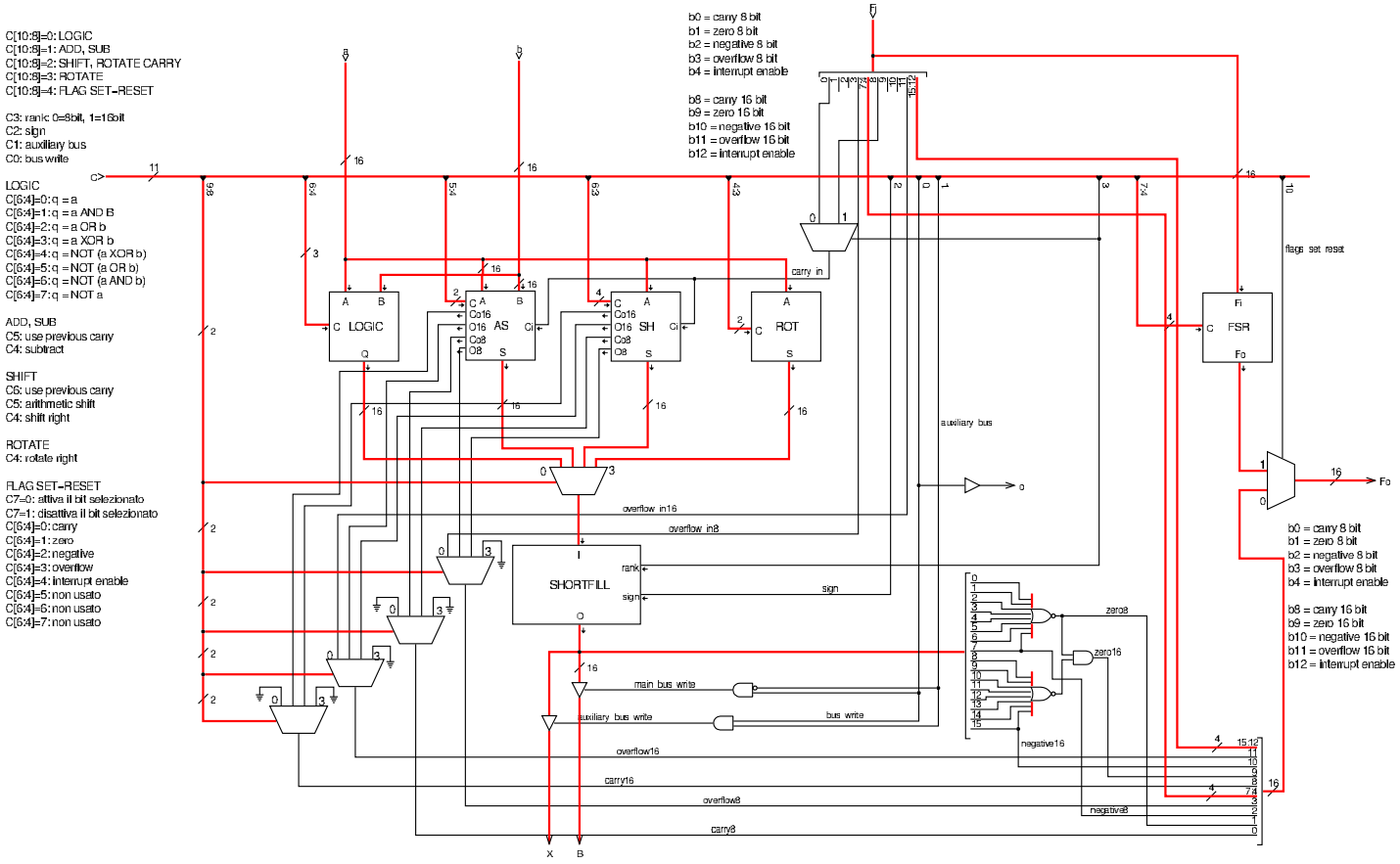
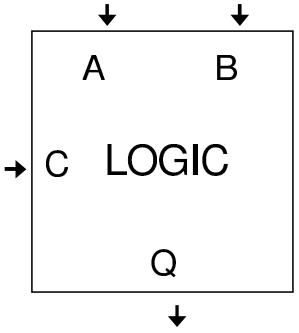


Figura u116.8. Struttura dell'unità logica interna alla ALU.



C=0: q = a
C=1: q = a AND B
C=2: q = a OR b
C=3: q = a XOR b
C=4: q = NOT (a XOR b)
C=5: q = NOT (a OR b)
C=6: q = NOT (a AND b)
C=7: q = NOT a

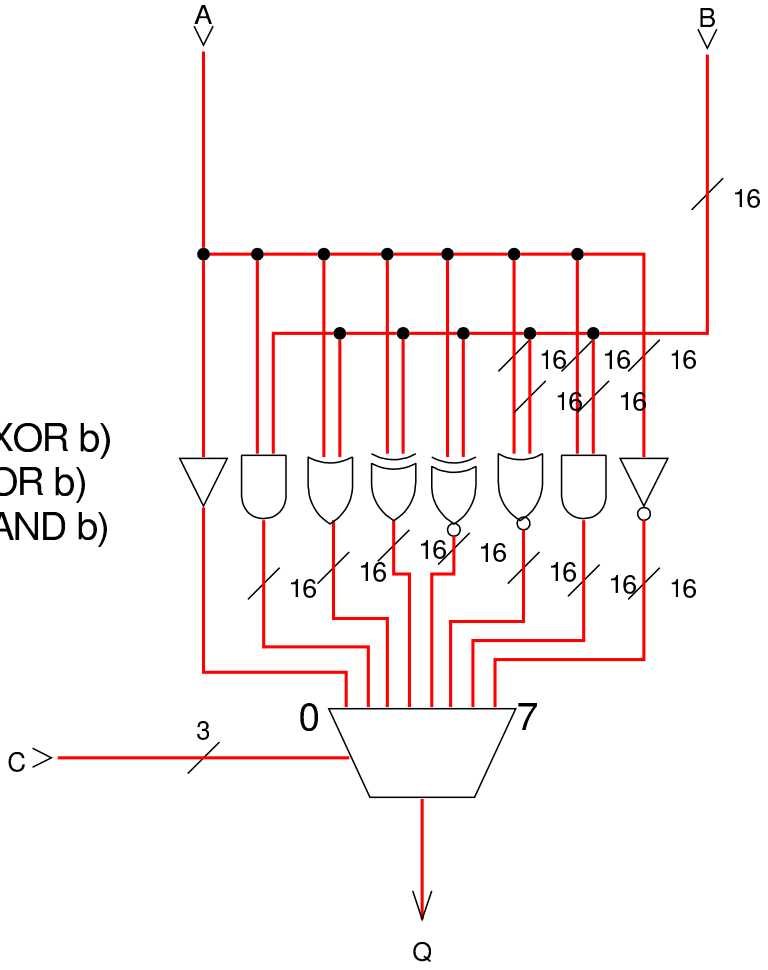


Figura u116.9. Struttura del modulo di addizione e sottrazione **AS**.

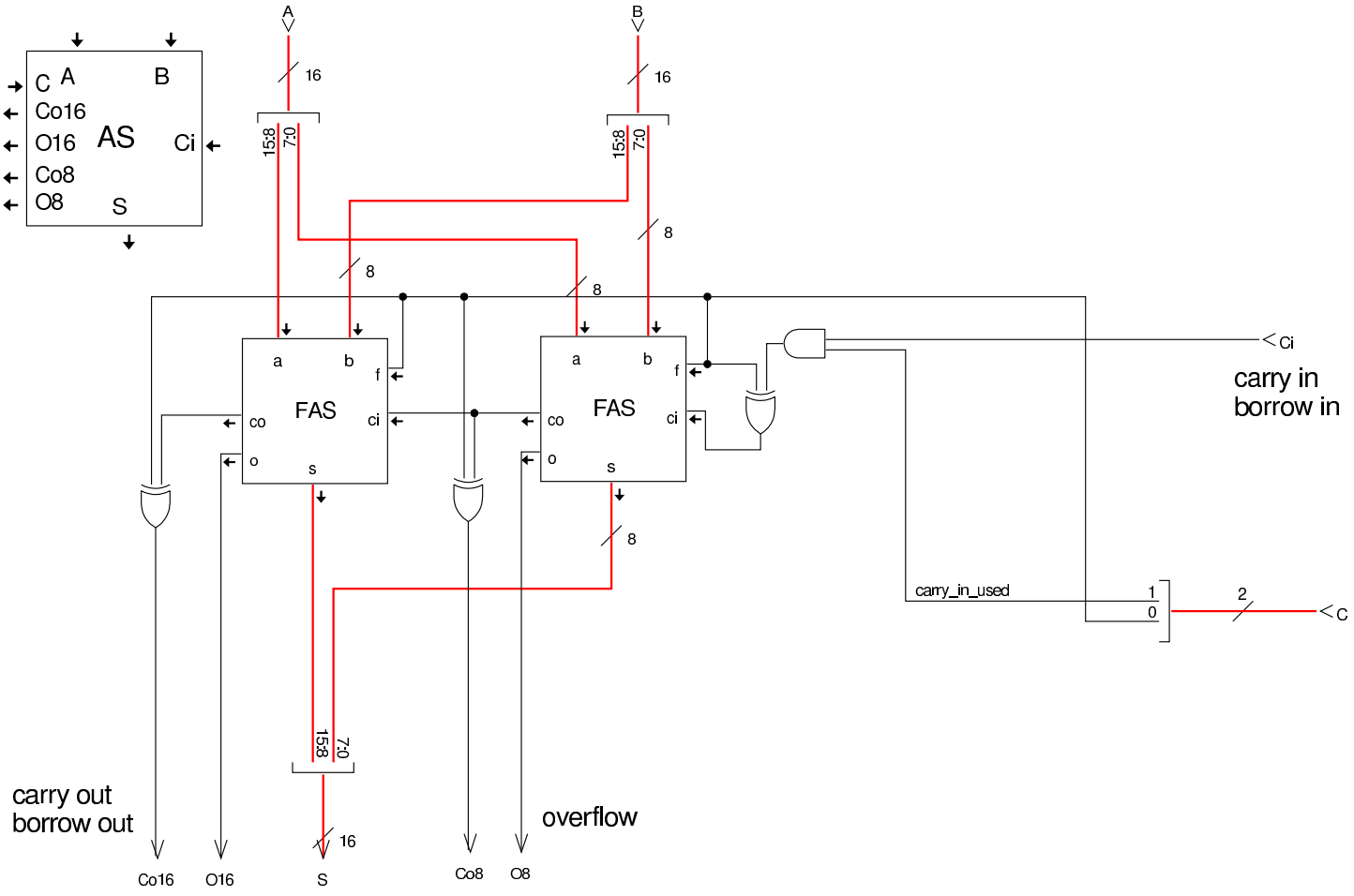


Figura u116.10. Modulo **`FAS`** (*full adder-subtractor*) contenuto nell'unità aritmetica. I moduli **`fa`** sono degli addizionatori completi (*full adder*).

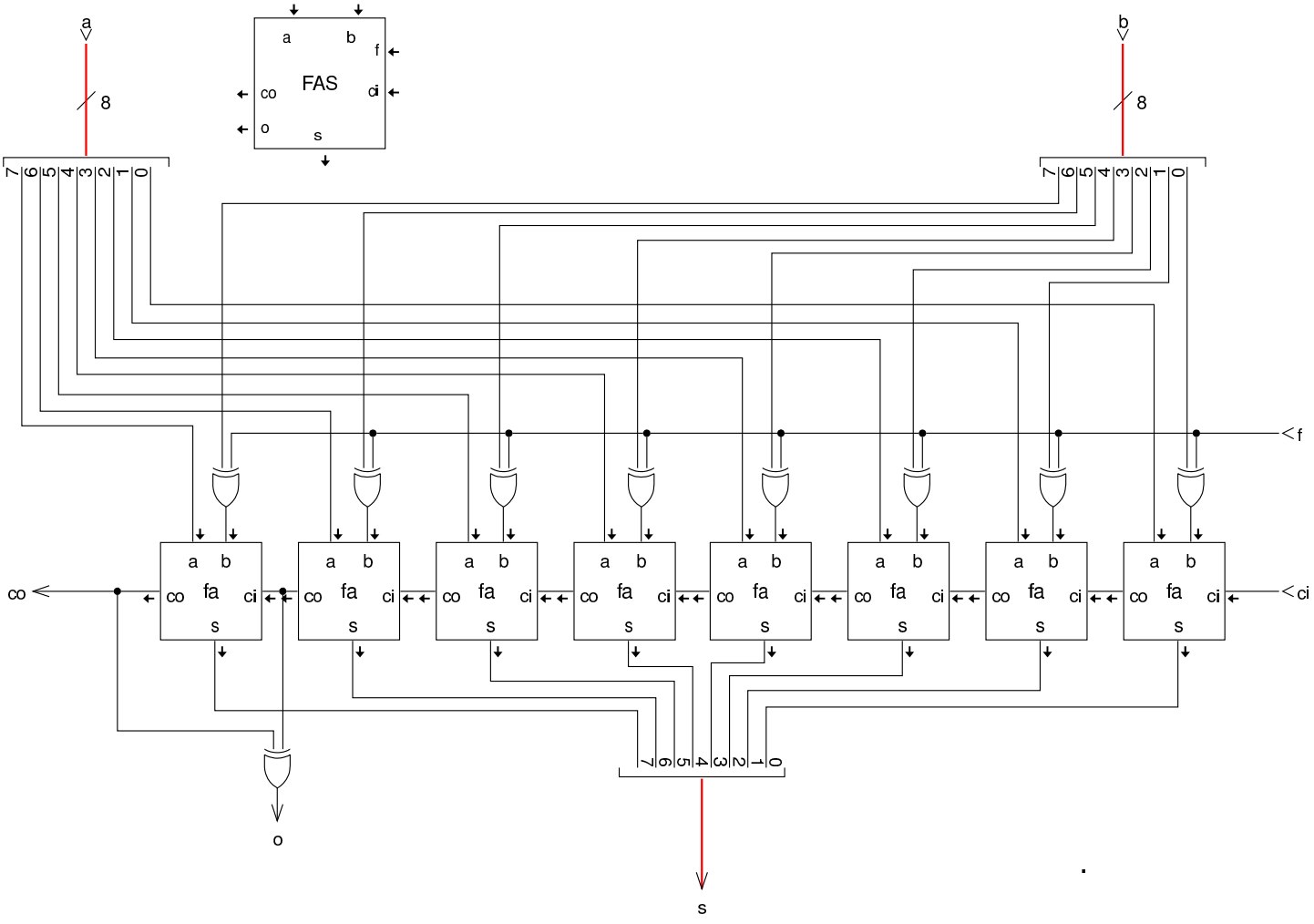


Figura u116.11. Struttura del modulo di scorrimento (**SH**), il quale si occupa anche della rotazione con riporto.

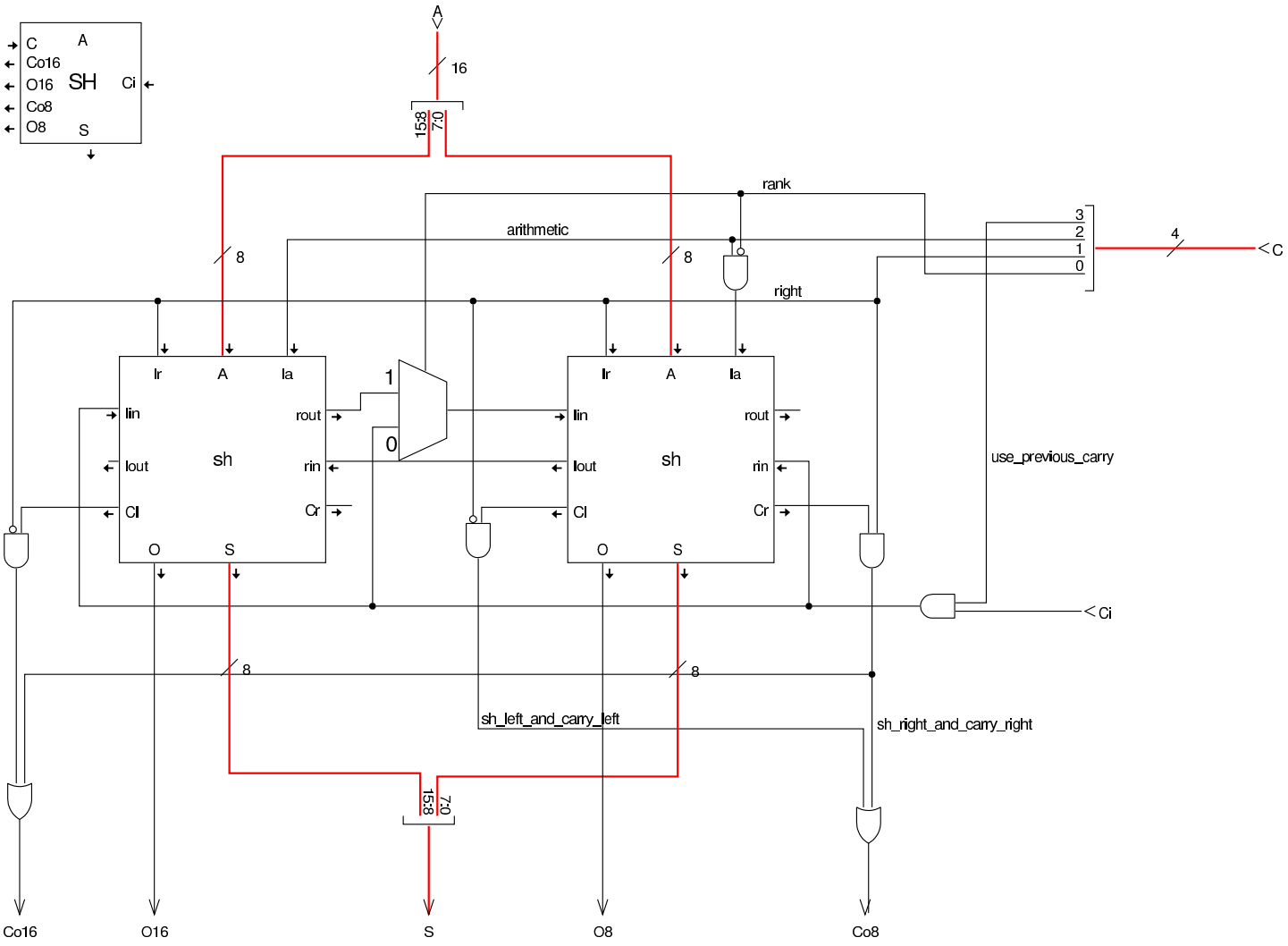


Figura u116.12. Modulo di rotazione (**ROT**): questo modulo esegue esclusivamente la rotazione del contenuto, senza utilizzare il riporto.

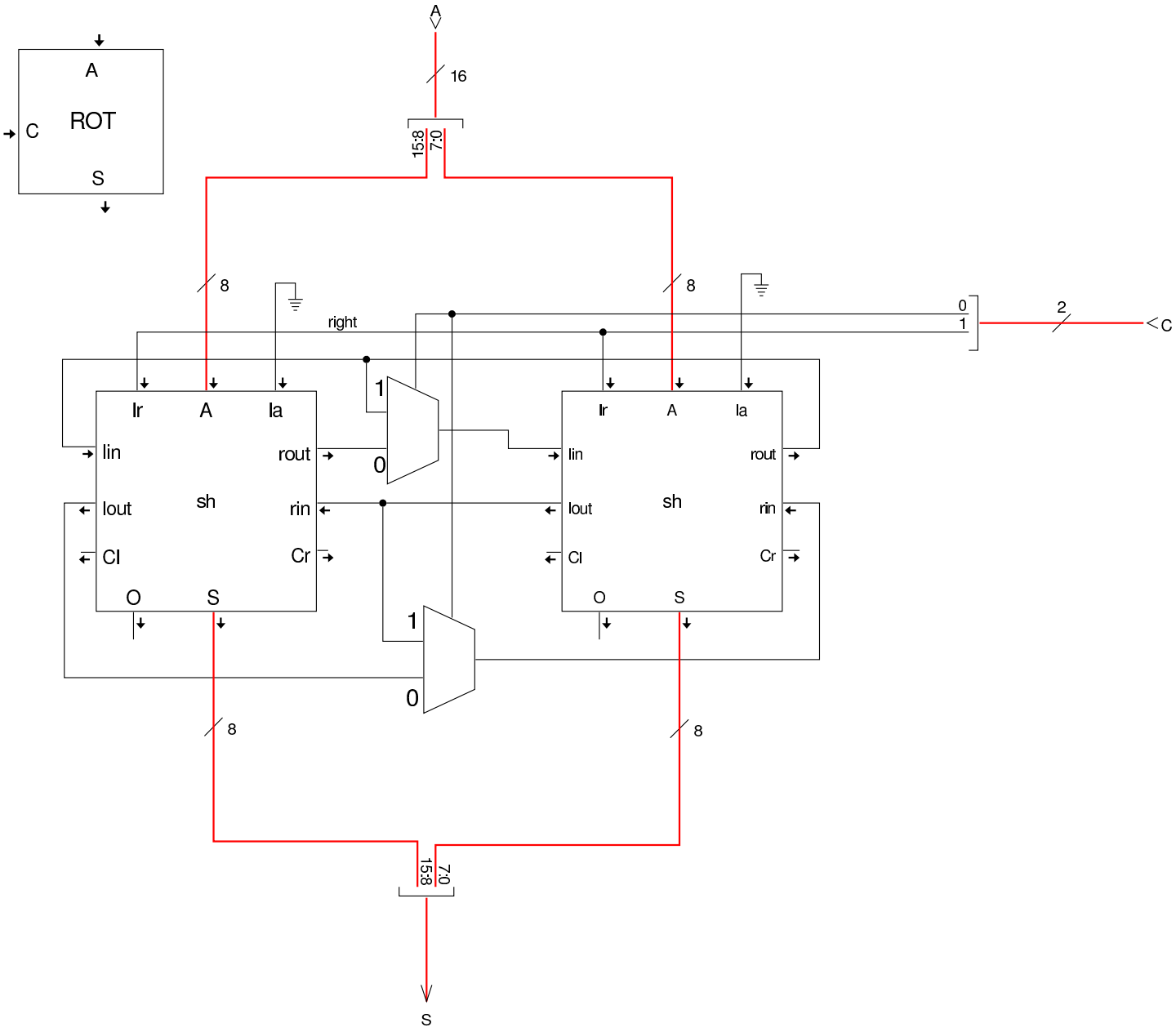
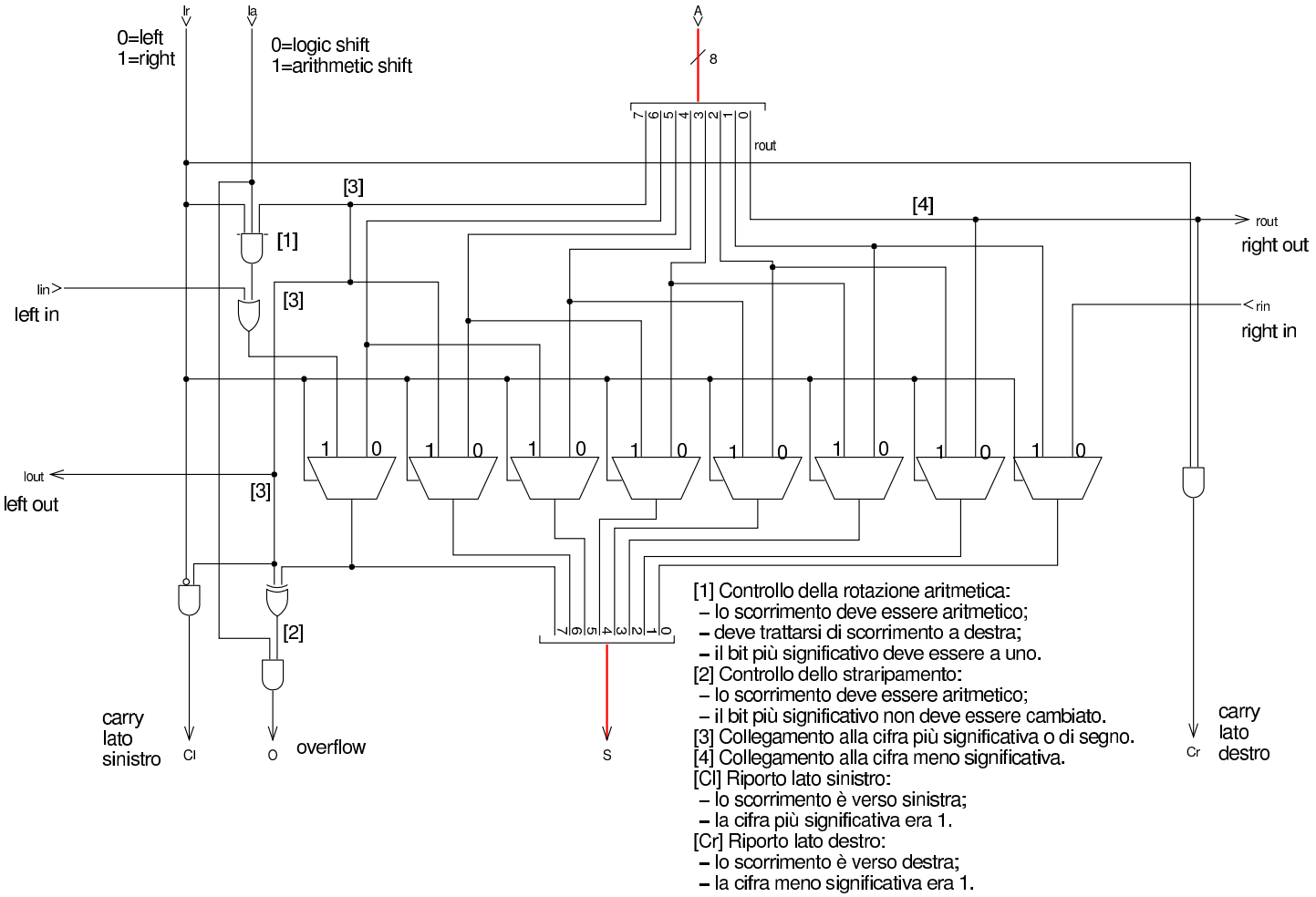


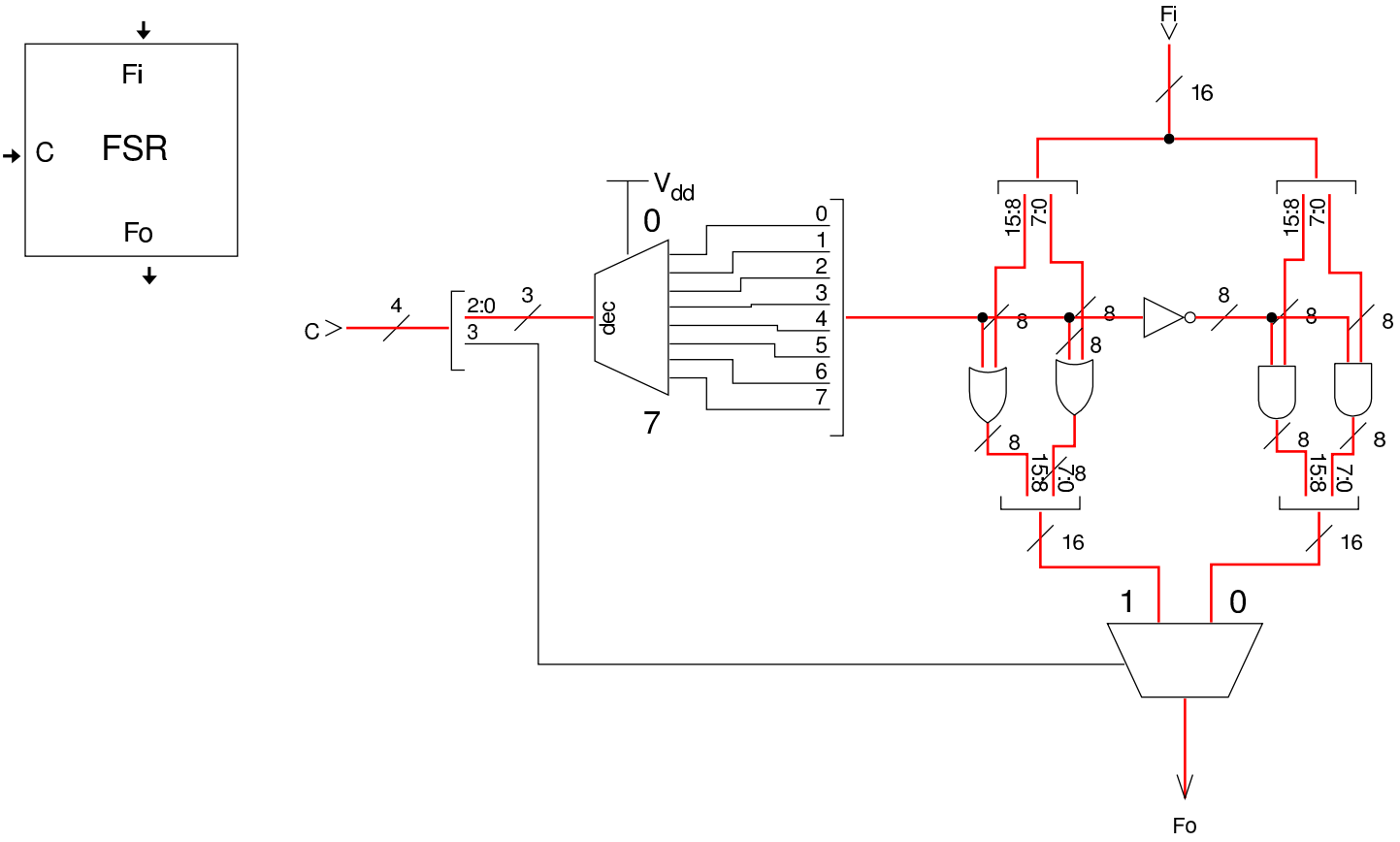
Figura u116.13. Modulo **sh** contenuto nei moduli di scorrimento e di rotazione.



La ALU di questa versione della CPU dimostrativa consente di modificare lo stato degli indicatori, attraverso il modulo **FAS** (*flags add-subtract*). Prima di tutto va osservato che gli indicatori sono doppi, su due gruppi da 8 bit, per consentire di distinguere quando alcune operazioni producono l'alterazione degli indicatori in modo diverso se si considerano valori a 8 bit o valori a 16 bit; per esempio esiste un indicatore di riporto a 8 bit e un altro a 16 bit. Quando si interviene per modificare lo stato degli indicatori, si agisce simultaneamente in entrambi i gruppi, attivandoli o disattivandoli assieme. Il modulo **FAS** riceve quindi una maschera da 8 bit e la funzione da applica-

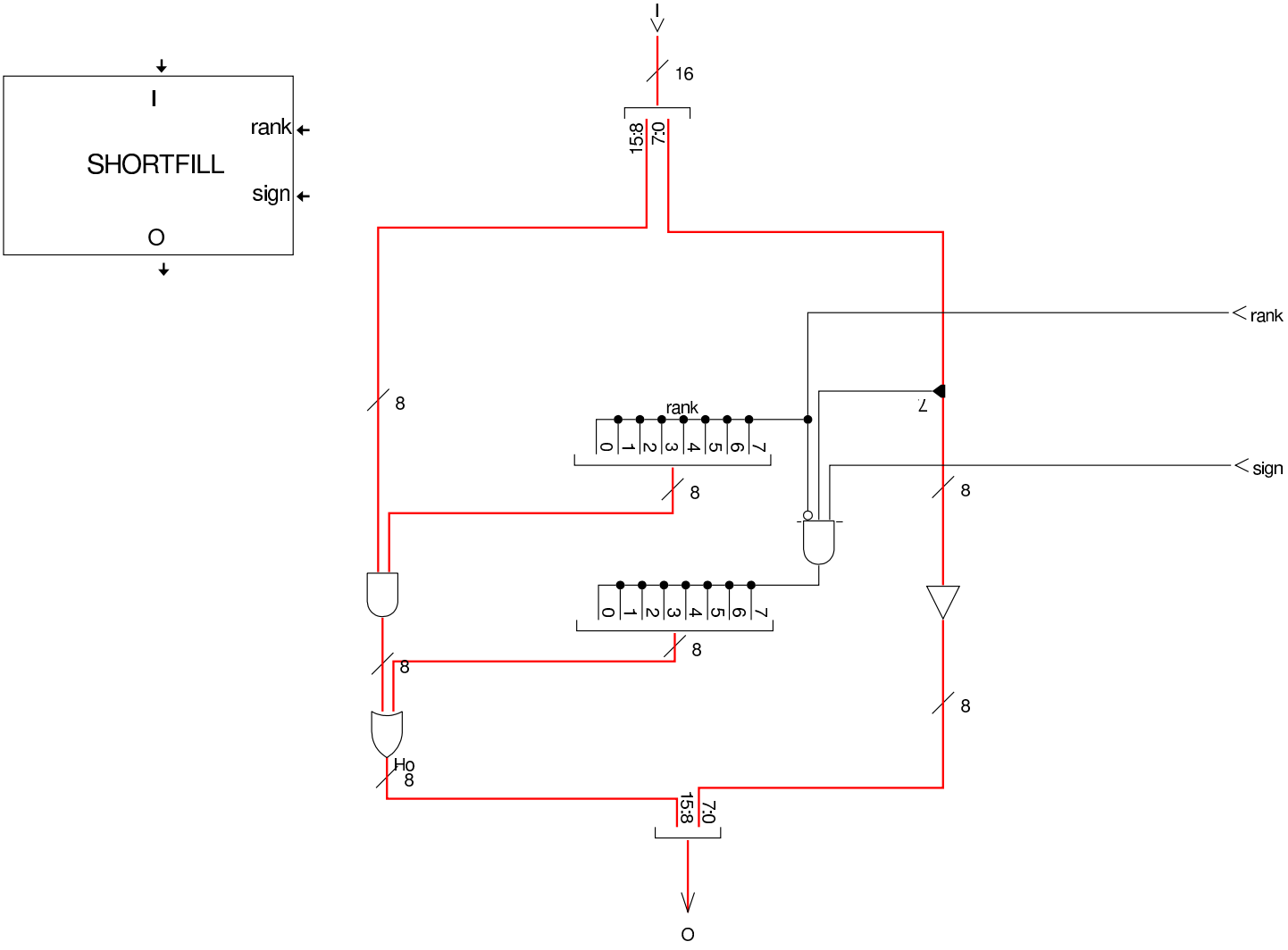
re a questa maschera: si può applicare l'operatore OR o l'operatore AND e si aggiorna di conseguenza lo stato dei registri.

Figura u116.14. Modulo **`FSR`** per l'alterazione diretta degli indicatori.



In modo simile a quello che succede nei registri, la ALU dispone del modulo **`SHORTFILL`** che può essere usato per adattare un valore, quando si sa che questo va considerato a 8 bit, per estendere il segno correttamente.

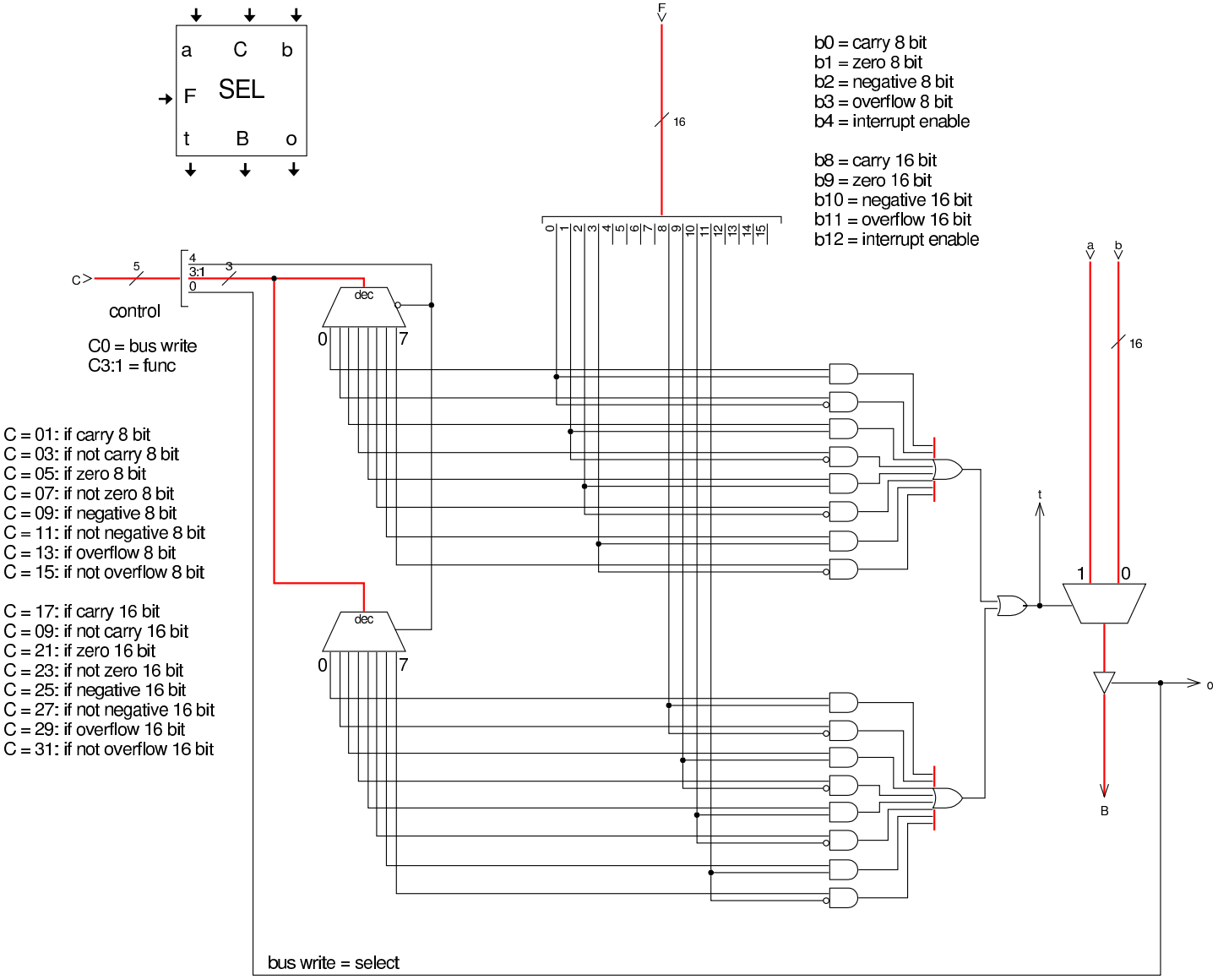
Figura u116.15. Modulo **SHORTFILL**, usato per sistemare il contenuto degli otto byte più significativi, quando si richiede di gestore solo operazioni a otto bit.



Modulo «SEL»

Il modulo **SEL** si estende per gestire gli indicatori distinti, a otto o sedici bit. Tra gli indicatori ne appare uno nuovo, relativo all'attivazione o meno delle interruzioni hardware (IRQ), ma su questo valore non si prevedono valutazioni, quindi il modulo **SEL** lo ignora.

Figura u116.16. Modulo **SEL**.

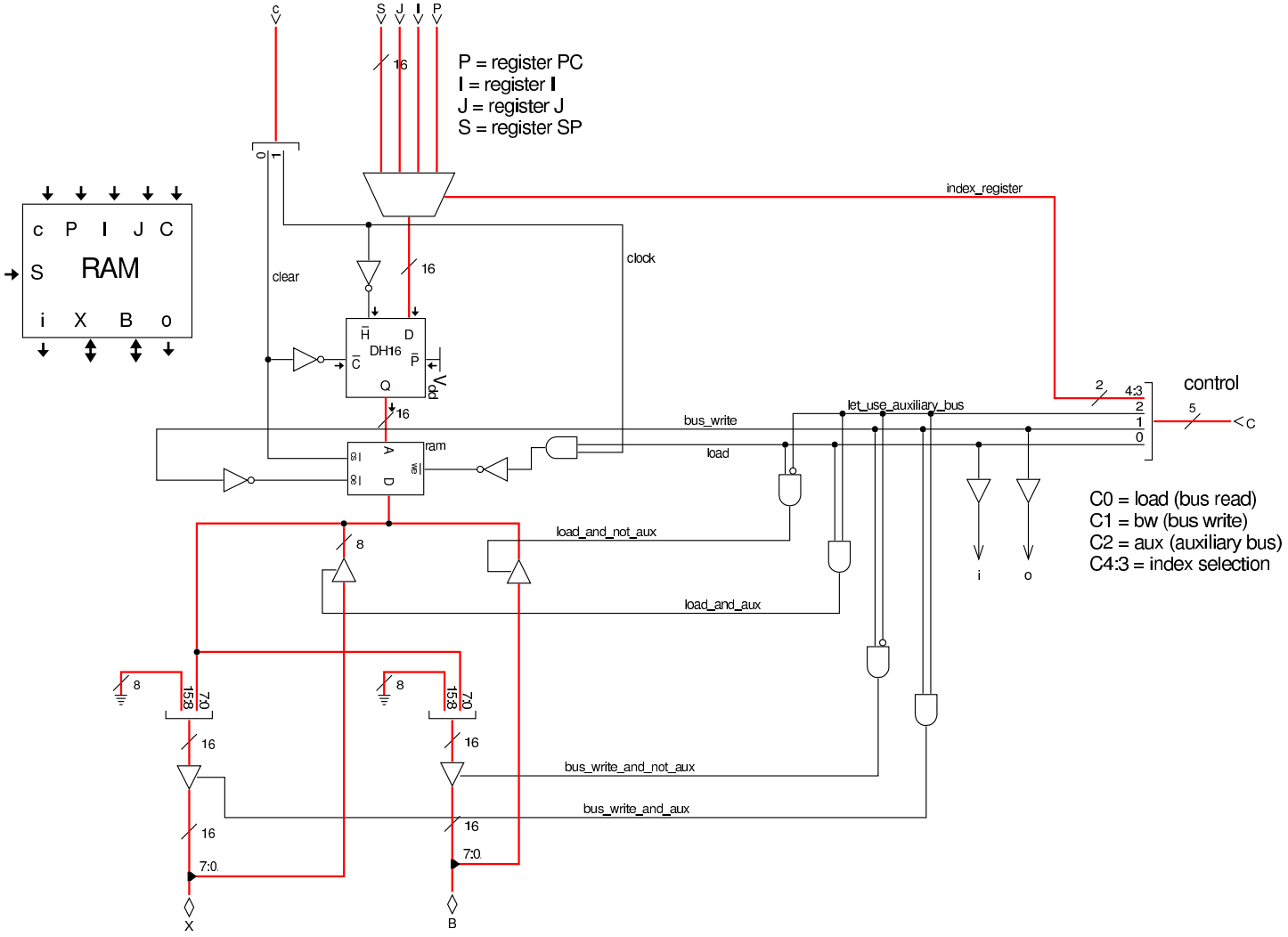


Modulo «RAM»

«

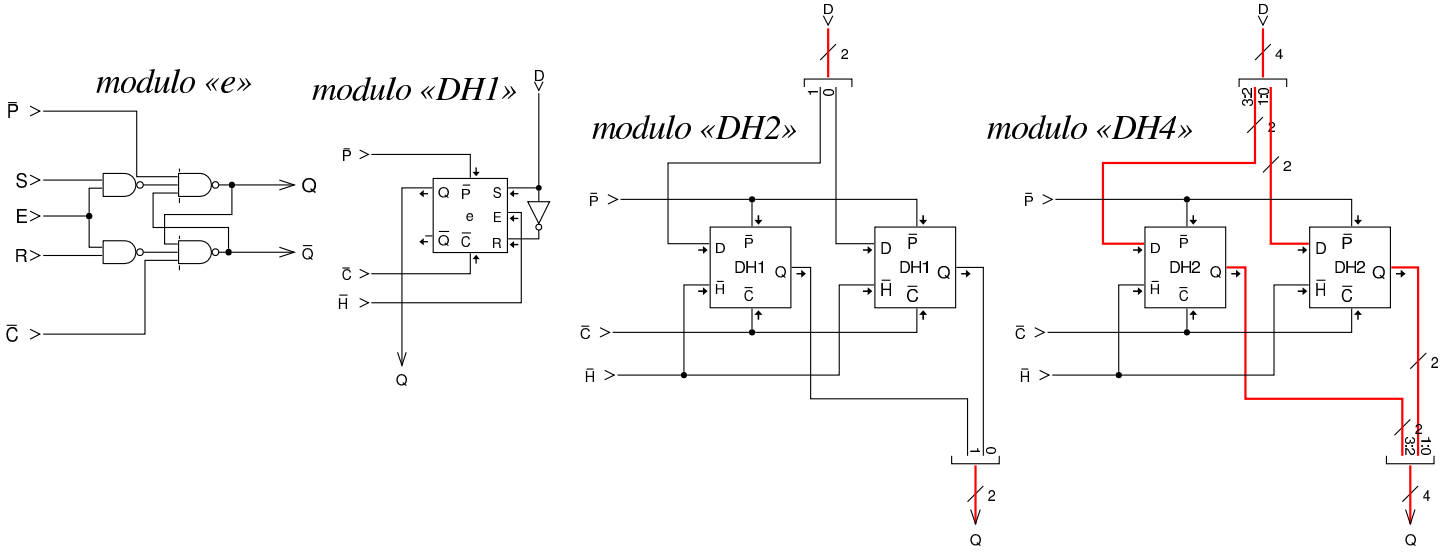
La memoria RAM continua a funzionare a blocchi di otto bit, come avviene nelle architetture comuni. Per leggere o scrivere valori a 16 bit occorre eseguire due operazioni successive; inoltre, tenendo conto che si lavora secondo l'ordine *little endian*, lettura e scrittura partono sempre dal byte meno significativo.

Figura u116.17. Modulo **RAM**.



Il modulo **RAM** contiene il registro ***DH16*** che si lascia attraversare dal valore che riceve dall'ingresso ***D*** quando l'ingresso ***H'*** è attivo, altrimenti, se ***H'*** non è attivo, mantiene in uscita il valore recepito precedentemente.

Figura u116.18. Da sinistra a destra, si vedono le fasi realizzative dei moduli **`DH`**...: si parte da un flip-flop SR con ingresso di abilitazione, quindi si realizza un flip-flop D con ingresso di abilitazione, poi si mettono in parallelo i flip-flop D. Si intende che il registro ***DH16*** è composto con due registri ***DH8***, il quale, a sua volta, è composto da due registri ***DH4***.

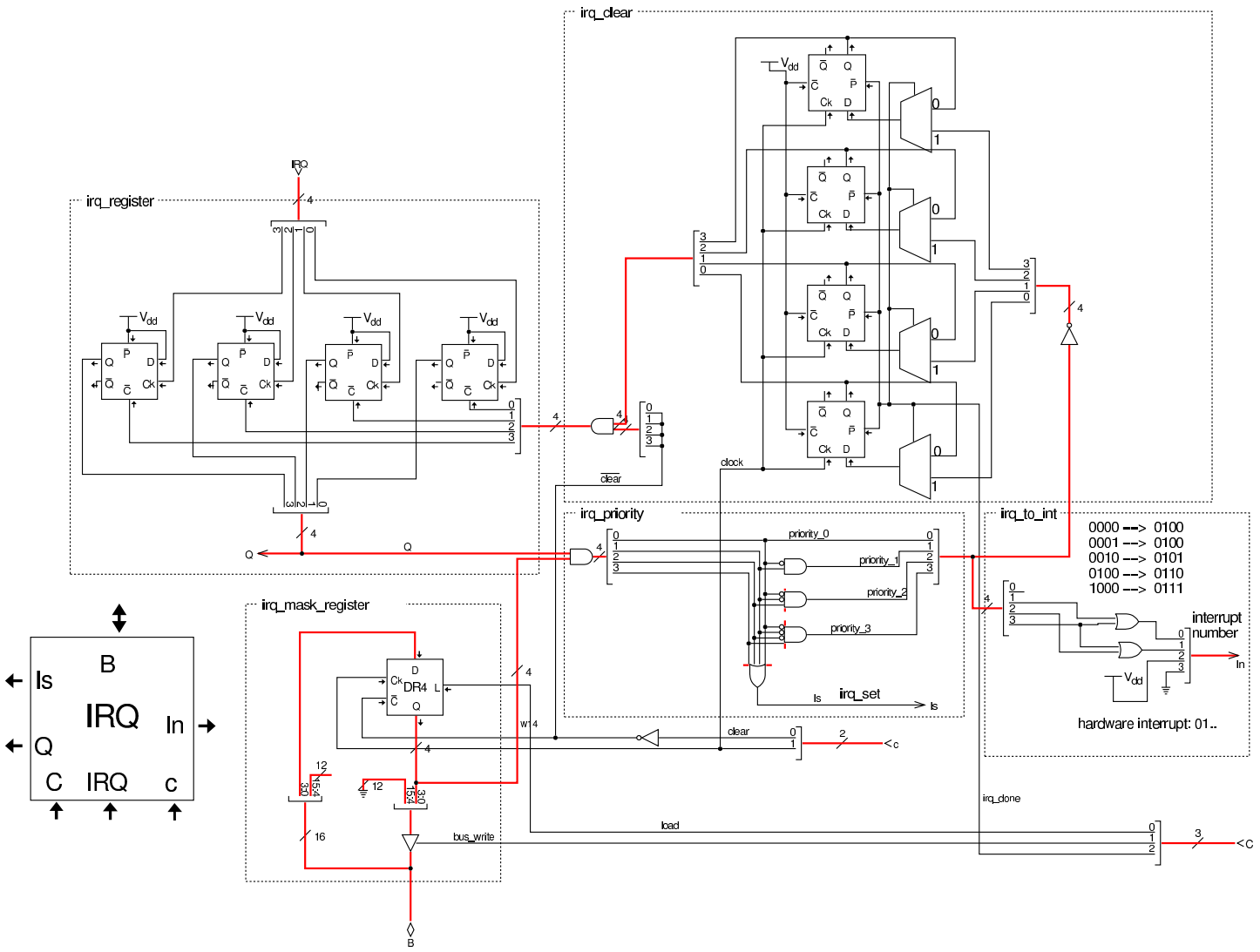


Modulo «IRQ»

«

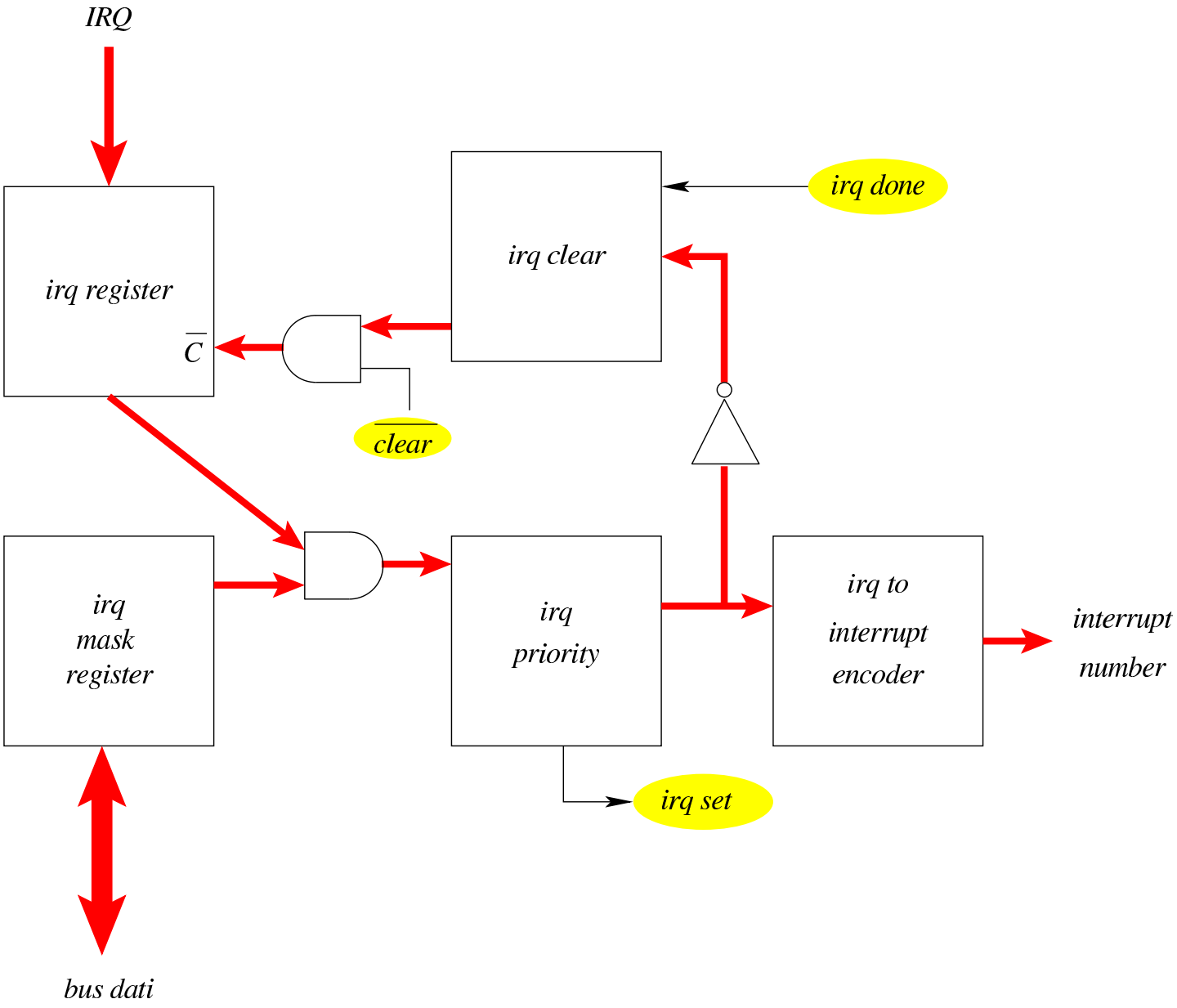
Questa versione della CPU dimostrativa gestisce le interruzioni, distinguendo tra quelle prodotte internamente dalla CPU stessa, quelle provenienti da dispositivi esterni e quelle gestite via software. Il modulo **`IRQ`** si occupa di ricevere le interruzioni hardware dai dispositivi per fornirle al circuito di controllo che deve poi attuare l'interruzione. In breve, il modulo **`IRQ`** riceve le interruzioni in modo asincrono, le memorizza e determina quale sia l'interruzione da servire per prima. Il modulo appare esternamente come se fosse un registro, in quanto deve poter ricevere una maschera delle interruzioni ammissibili; la stessa maschera può essere letta dal modulo.

Figura u116.19. Schema complessivo del modulo **IRQ**.



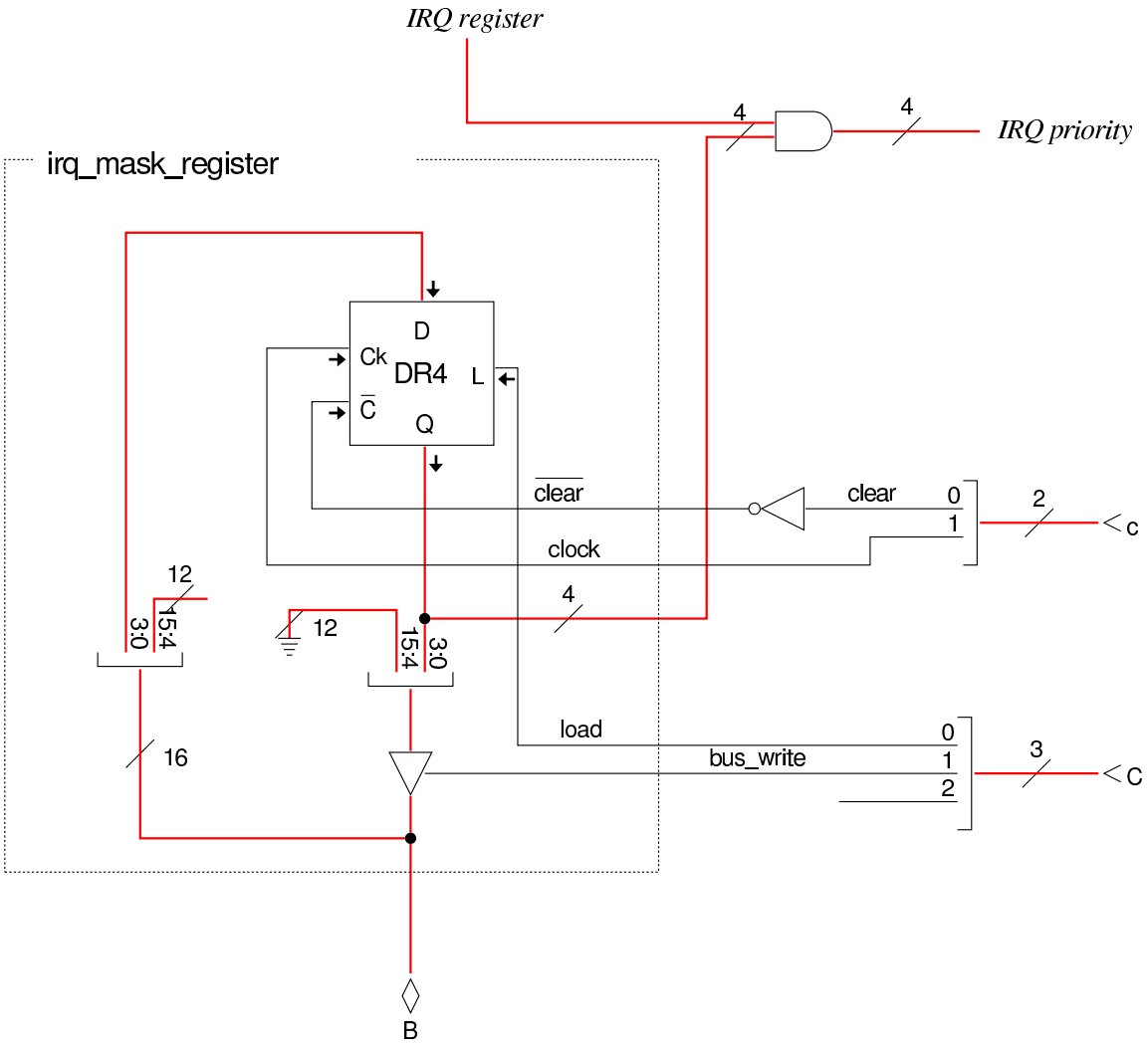
Per poter comprendere cosa fa il modulo **IRQ** è necessario analizzare i suoi vari componenti, con l'aiuto di uno schema a blocchi che riproduce in modo più semplice il suo disegno effettivo. Questo schema a blocchi è visibile nella figura successiva.

Figura u116.20. Schema a blocchi del modulo **IRQ**.



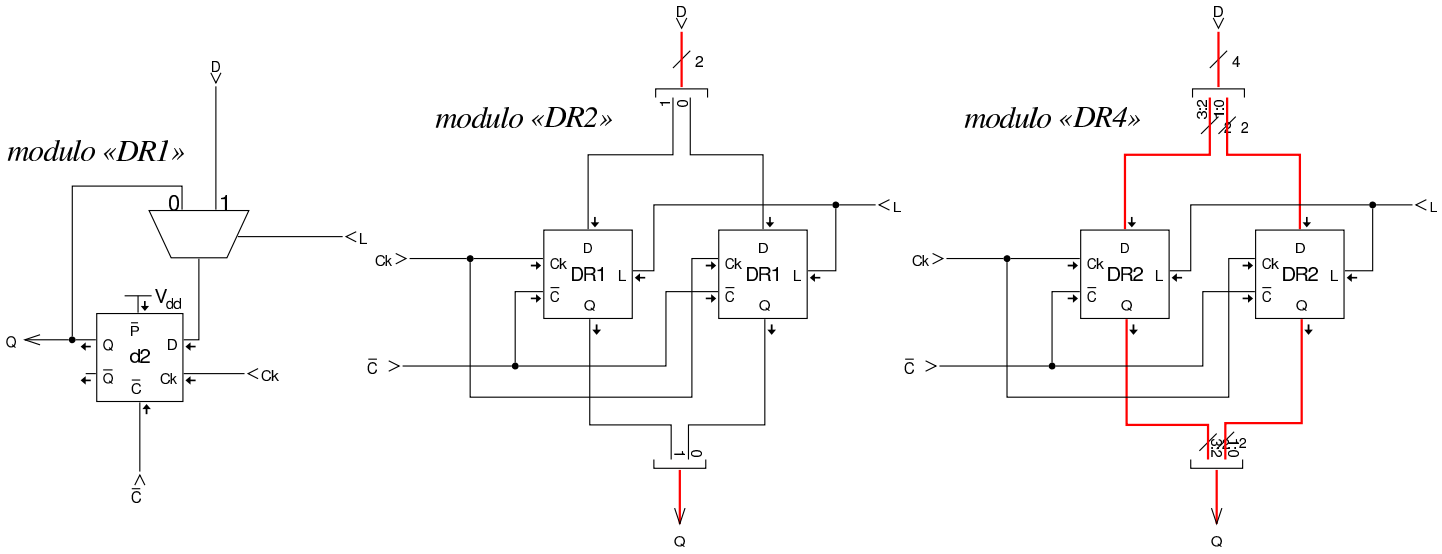
Conviene partire dall'analisi del registro che contiene la maschera degli IRQ ammissibili che appare in basso a sinistra nello schema complessivo: si tratta di un registro a 4 bit (uno per ogni IRQ gestito) che legge dal bus dati per aggiornare il proprio valore e scrive sul bus dati, per consentire di conoscere il valore che contiene (ammesso che ciò possa servire).

Figura u116.21. Dettaglio del registro della maschera degli IRQ.



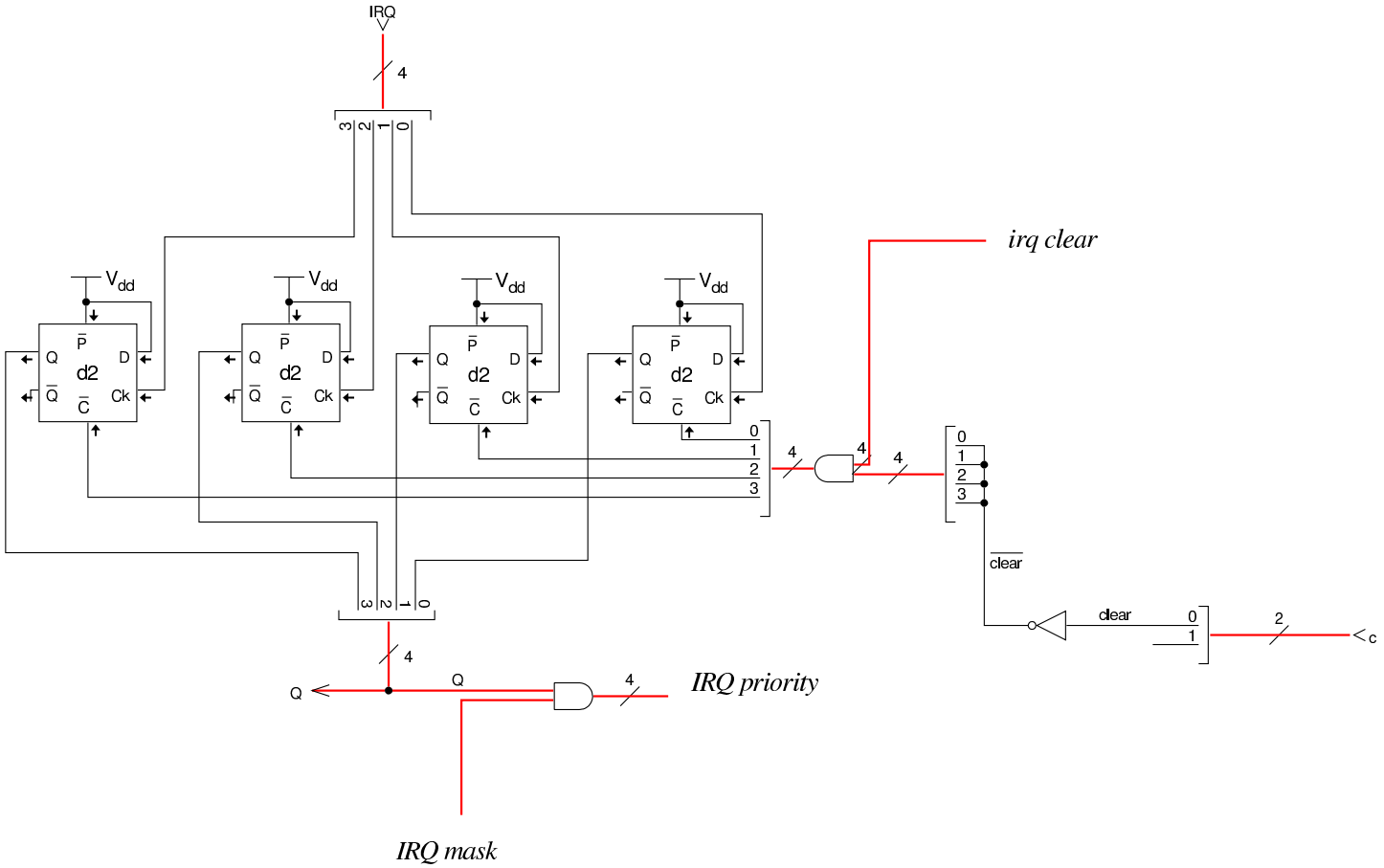
Il modulo **DR4** è un registro a quattro bit, realizzato con flip-flop D, come si vede nella figura successiva, attraverso passaggi successivi.

Figura u116.22. Costruzione dei moduli **DR**….



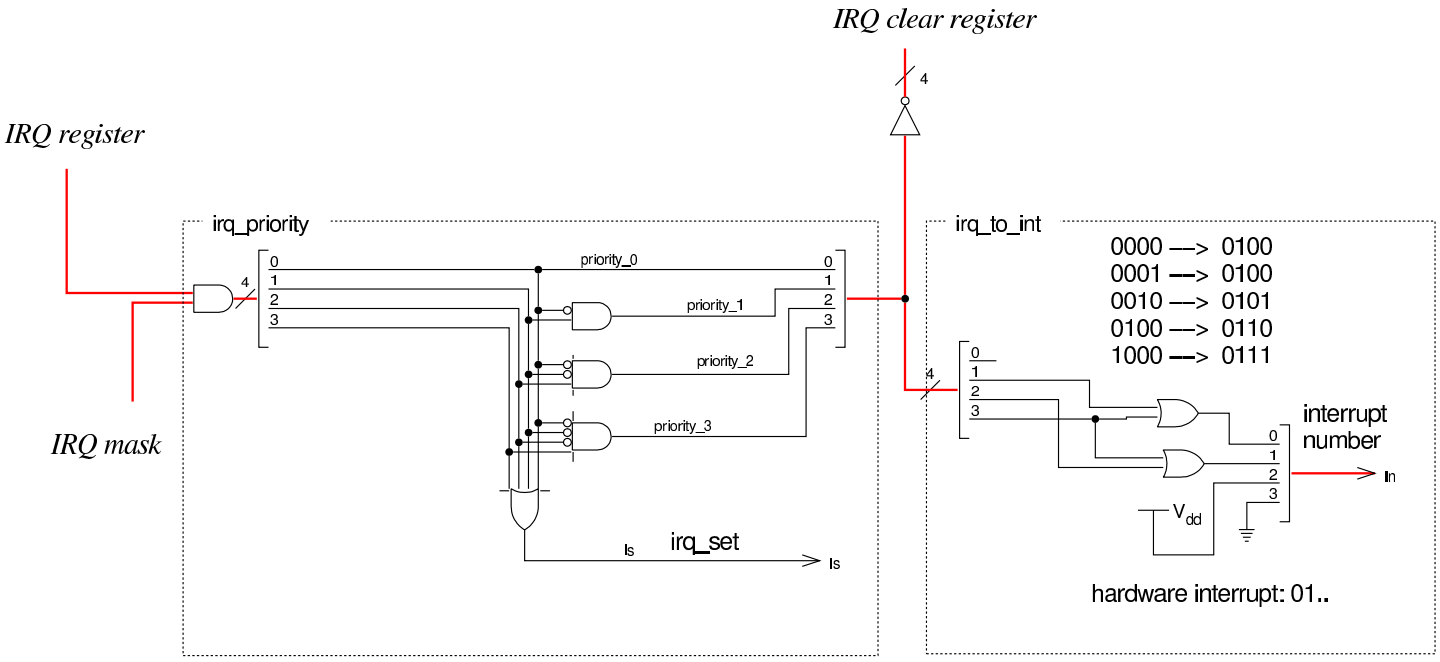
In alto a sinistra, nello schema generale, appare il registro degli IRQ, il cui scopo è quello di memorizzare le interruzioni hardware ricevute dall'ingresso IRQ. Questo registro è costruito in modo insolito, perché è costituito da flip-flop D a margine positivo, ma l'ingresso ***D*** di tali flip-flop è collegato in modo da essere sempre attivo, mentre l'ingresso di clock viene usato per ricevere il segnale di IRQ. In pratica, un segnale di IRQ che giunge all'ingresso clock del flip-flop, lo attiva stabilmente. I flip-flop del registro IRQ possono essere azzerati solo attraverso l'ingresso ***C'***.

Figura u116.23. Dettaglio del registro degli IRQ ricevuti.



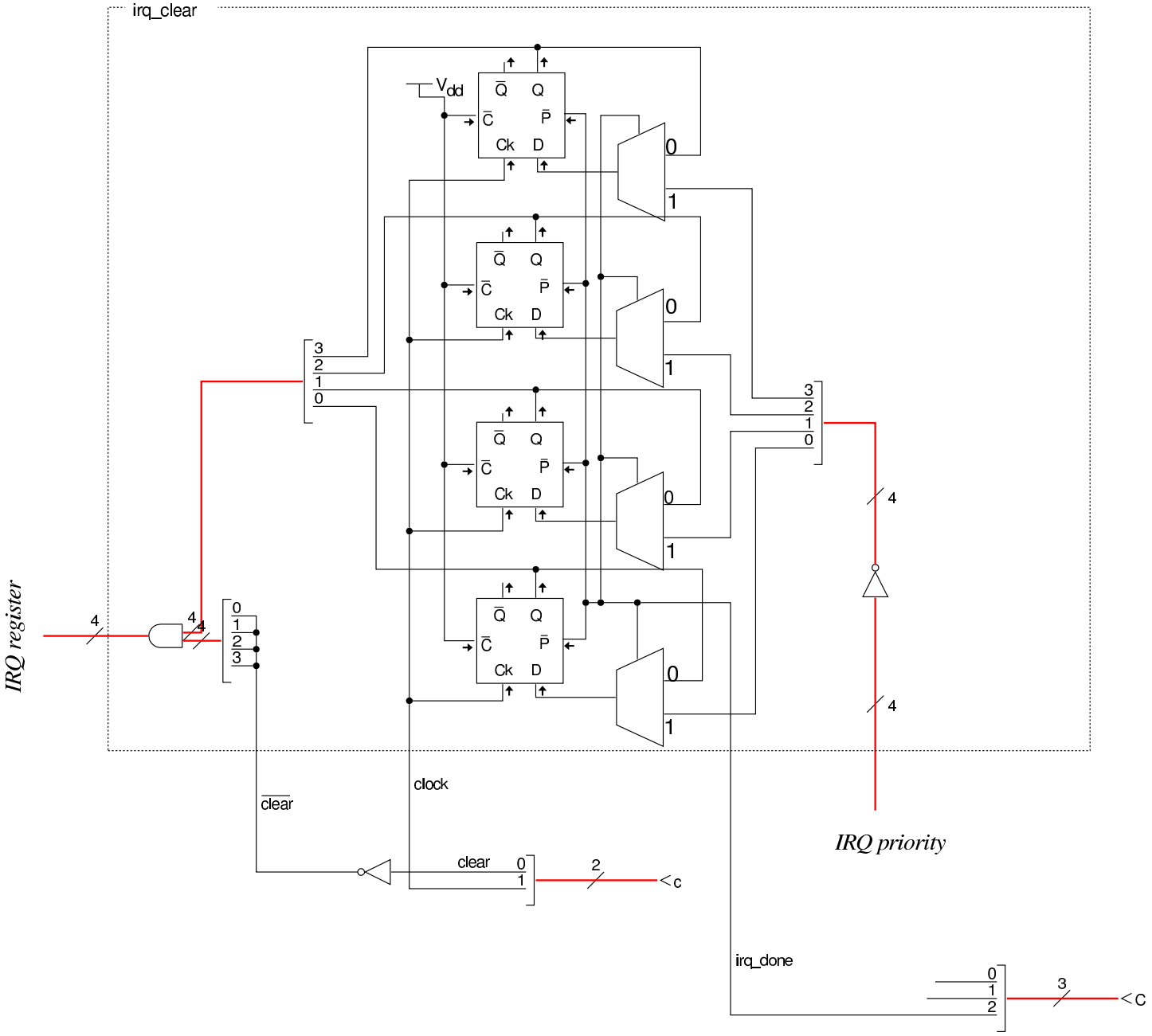
Il valore memorizzato nel registro IRQ e quello della maschera sottostante, vengono confrontati con una porta AND multipla (una porta distinta per ogni linea di IRQ) e quindi passati a un modulo che ne seleziona uno solo in base alla priorità: si sceglie il numero di IRQ più basso disponibile. Il modulo che ha selezionato la priorità comunica con un codificatore che si occupa di trasformare l'IRQ scelto in un numero di interruzione, per cui, IRQ0 diventa INT4, IRQ2 diventa INT5, fino a IRQ3 che diventa INT7. Si può osservare che il modulo di selezione della priorità emette un segnale (*irq set*) per informare della presenza effettiva di un IRQ che necessita di essere servito, dato che l'assenza di un IRQ produce comunque nel codificatore il valore INT4.

Figura u116.24. Dettaglio del selettore di priorità e del codificatore.



Quando un IRQ è stato servito, c'è la necessità di azzerare il flip-flop corrispondente nel registro degli IRQ (in alto a sinistra). Per ottenere questo risultato si utilizza il registro di azzeramento che si vede in alto a destra. Questo è composto da flip-flop D (a margine positivo) che in condizioni normali (quando l'ingresso *irq done* è pari a zero) producono in uscita un valore pari a uno, in quanto risultano inizializzati a uno (ingresso ***P'*** a zero). L'uscita di questo registro di azzeramento è collegato all'ingresso ***C'*** del registro degli IRQ, per cui, finché offre valori a uno, il registro degli IRQ mantiene il proprio valore memorizzato. Quando invece il registro di azzeramento riceve il segnale *irq done*, allora recepisce il complemento a uno del valore selezionato in base alla priorità di IRQ; in tal modo, si azzera al suo interno il bit corrispondente, azzerando di conseguenza il flip-flop del registro degli IRQ. Di conseguenza, il modulo che valuta la priorità può mettere in evidenza un altro IRQ, se disponibile.

Figura u116.25. Dettaglio del registro di cancellazione degli IRQ serviti.



L'azzeramento del registro degli IRQ deve poter avvenire anche simultaneamente per tutti i flip-flop che contiene, pertanto il suo ingresso ***C'*** è collegato con una porta AND che consente di agire in tal modo. Il segnale ***clear'*** risulta come complemento del segnale ***clear*** proveniente dal bus di controllo.

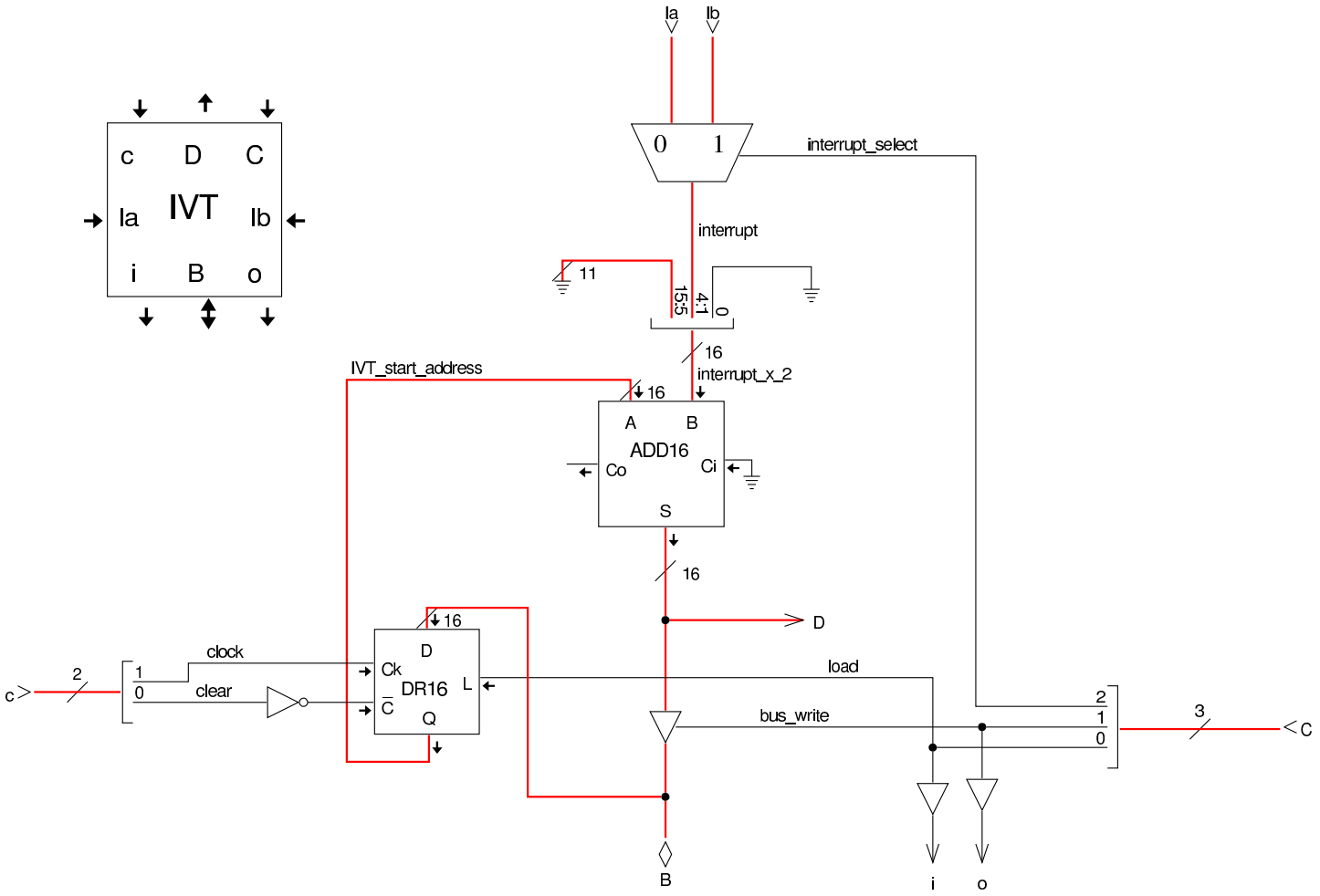
Modulo «IVT»

«

Per poter gestire le interruzioni (di CPU, hardware e software), questa versione della CPU dimostrativa ha la necessità di disporre di una tabella «IVT» (*interrupt vector table*), da quale parte nella memoria RAM. La tabella IVT deve essere realizzata come un array di interi a 16 bit (*little-endian*), ognuno dei quali rappresenta l'indirizzo di una routine da eseguire quando viene attivata l'interruzione corrispondente. Pertanto, ***IVT[n]*** deve corrispondere all'indirizzo che si deve occupare di svolgere l'attività richiesta dall'interruzione ***n***.

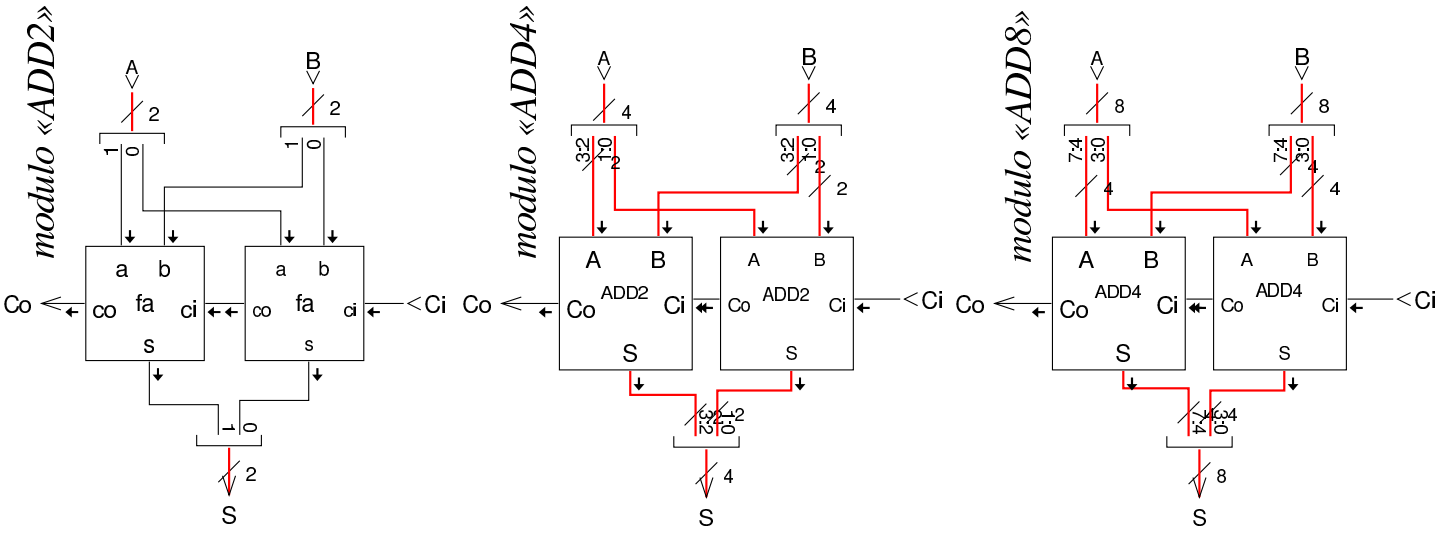
Il registro ***IVT*** serve a memorizzare la collocazione della tabella IVT, corrispondente precisamente a ***IVT[0]***. Da due ingressi indipendenti, il modulo riceve il numero di una certa interruzione, la quale viene trasformata nell'indirizzo corrispondente in memoria che contiene il riferimento alla routine da avviare.

Figura u116.26. Modulo **IVT**.



Nel modulo **IVT** si utilizza un registro **DR16** per memorizzare l'indirizzo di partenza della tabella IVT. Questo registro è realizzato nella stessa modalità già descritta in relazione al registro di tipo **DR4**, nella sezione precedente. Il modulo **ADD16** è composto da sedici addizionatori completi, messi in parallelo, con il riporto in cascata. Anche questo modulo viene realizzato per fasi successive, come già fatto per **DR16**.

Figura u116.27. Costruzione dei moduli **ADD**….



Modulo «CTRL»

«

Il modulo **CTRL** ha solo piccole modifiche rispetto alla versione precedente: il codice operativo rimane a otto bit (ingresso ***I***); il registro contatore (***CNT9***) è ridotto a soli nove bit, perché nel microcodice non si superano le 512 righe; le righe del microcodice richiedono molti più bit, quindi si utilizzano cinque moduli di memoria che assieme permettono di pilotare un bus di controllo da 160 bit. Nell'ingresso al contatore ***CNT9*** c'è la mediazione di un multiplatore che consente di immettere un indirizzo quando il flip-flop D che appare sulla destra è attivo. Questo indirizzo deve corrispondere al punto in cui nel microcodice si descrive la procedura necessaria a iniziare un'interruzione hardware (IRQ); in pratica deve corrispondere alla collocazione dell'etichetta '**irq:**', come si può determinare dai file prodotti dalla compilazione con Tkgate.

Figura u116.28. Modulo **CTRL**.

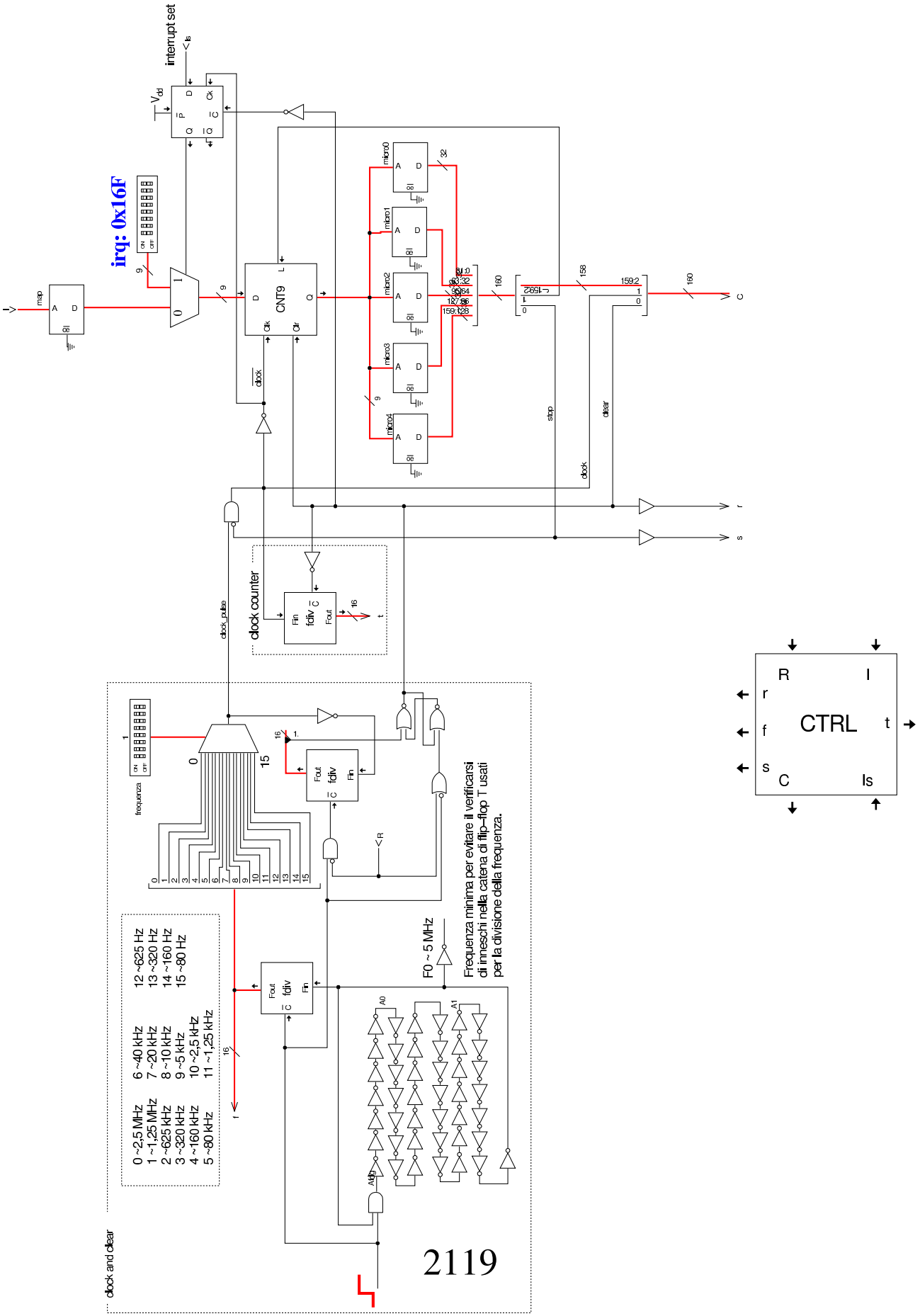
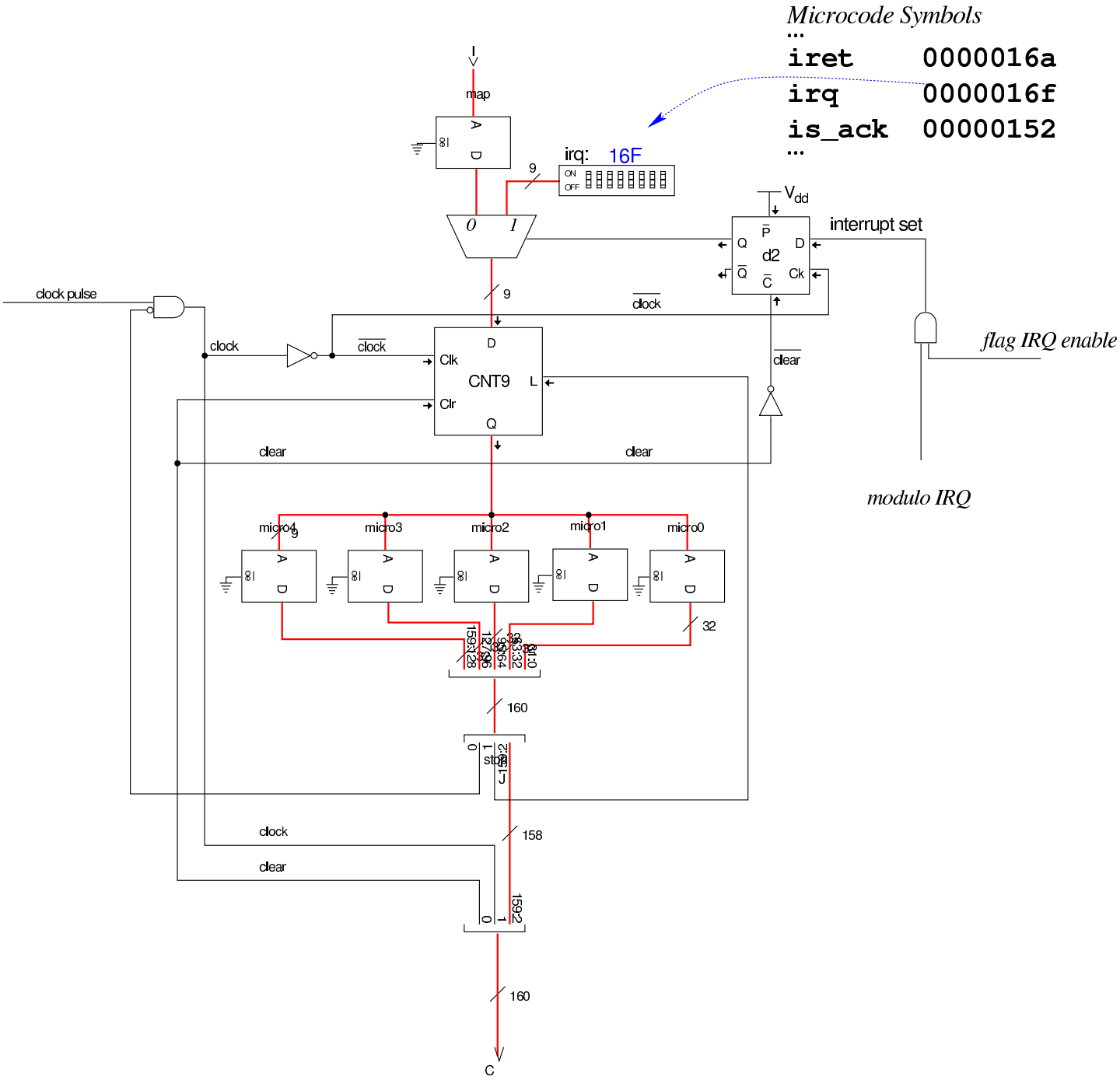


Figura u116.29. Dettaglio dell'unità di controllo che si occupa di gestire il codice.



Osservando la figura precedente, è importante chiarire cosa accade quando viene recepita un'interruzione hardware: il modulo **`IRQ`**, già descritto nella sezione u0.6, trasmette al modulo **`IVT`** il numero dell'interruzione corrispondente al numero di IRQ selezionato e attiva

la propria uscita ***Is*** (*IRQ set*) che arriva all'unità di controllo solo se è attivo anche l'indicatore ***I*** (*interrupt enable*). Se le cose stanno proprio così, questa richiesta viene memorizzata nel flip-flop D (a margine positivo) che appare in alto a destra nello schema; quindi, alla prima occasione in cui l'unità di controllo deve eseguire un nuovo codice operativo, si trova invece diretta a eseguire le istruzioni corrispondenti all'etichetta '**`irq:`**'.

Listato u116.30. Dichiarazione delle memorie utilizzate.

```
map       bank[7:0]     ctrl.map;
microcode bank[31:0]    ctrl.micro0;
microcode bank[63:32]   ctrl.micro1;
microcode bank[95:64]   ctrl.micro2;
microcode bank[127:96]  ctrl.micro3;
microcode bank[159:128] ctrl.micro4;
macrocode bank[15:0]    ram.ram;
```

Codici operativi

I codici operativi usati in questa versione della CPU dimostrativa, utilizzano sempre solo otto bit, ma invece di usare semplicemente un numero sequenziale per distinguerli, si va a strutturare con un certo criterio lo spazio binario disponibile. Per prima cosa si definisce una conversione tra i registri utilizzabili nella programmazione e un numero intero, in modo tale da usare tre bit per la loro distinzione:

```
registers I=0, J=1, A=2, B=3, BP=4, SP=5, MDR=6, FL=7;
```

Gli operandi associati ai codici operativi sono di vario tipo; i casi più semplici sono dichiarati all'inizio:

```
operands op_0 {
    -  = { };
```

```
};
operands op_8 {
    #1 = { +1=#1[7:0]; };
};
operands op_16 {
    #1 = { +1=#1[7:0]; +2=#1[15:8]; };
};
```

Si comprende, intuitivamente, che '**op_0**' rappresenti la mancanza di operandi, che '**op_8**' rappresenti un operando di soli 8 bit, e che '**op_16**' rappresenti un operando da 16 bit, tenendo conto che la memoria RAM è però organizzata in blocchi da otto bit e l'accesso alla stessa avviene in modalità *little endian* (quindi il byte meno significativo si trova prima di quello più significativo).

Listato u116.33. Dichiarazione dei codici operativi.

```
op nop {
  map nop:        0x00;  // not operate
  +0[7:0]=0x00;
  operands op_0;
};
op mv {                  // 00...... = mv
  map nop:        0x00;  // 00000000 = mv %I %I non valido => nop
  map mv_i_j:     0x01;  // 00000001 = mv %I %J
  map mv_i_a:     0x02;  // 00000010 = mv %I %A
  map mv_i_b:     0x03;  // 00000011 = mv %I %B
  map mv_i_bp:    0x04;  // 00000100 = mv %I %BP
  map mv_i_sp:    0x05;  // 00000101 = mv %I %SP
  map mv_i_mdr:   0x06;  // 00000110 = mv %I %MDR
  map mv_i_fl:    0x07;  // 00000111 = mv %I %FL
  map mv_j_i:     0x08;  // 00001000 = mv %J %I
  map op_error:   0x09;  // 00001001 = mv %J %J non valido
  map mv_j_a:     0x0A;  // 00001010 = mv %J %A
  map mv_j_b:     0x0B;  // 00001011 = mv %J %B
```

```
map mv_j_bp:    0x0C;  // 00001100 = mv %J %BP
map mv_j_sp:    0x0D;  // 00001101 = mv %J %SP
map mv_j_mdr:   0x0E;  // 00001110 = mv %J %MDR
map mv_j_fl:    0x0F;  // 00001111 = mv %J %FL
map mv_a_i:     0x10;  // 00010000 = mv %A %I
map mv_a_j:     0x11;  // 00010001 = mv %A %J
map op_error:   0x12;  // 00010010 = mv %A %A non valido
map mv_a_b:     0x13;  // 00010011 = mv %A %B
map mv_a_bp:    0x14;  // 00010100 = mv %A %BP
map mv_a_sp:    0x15;  // 00010101 = mv %A %SP
map mv_a_mdr:   0x16;  // 00010110 = mv %A %MDR
map mv_a_fl:    0x17;  // 00010111 = mv %A %FL
map mv_b_i:     0x18;  // 00011000 = mv %B %I
map mv_b_j:     0x19;  // 00011001 = mv %B %J
map mv_b_a:     0x1A;  // 00011010 = mv %B %A
map op_error:   0x1B;  // 00011011 = mv %B %B non valido
map mv_b_bp:    0x1C;  // 00011100 = mv %B %BP
map mv_b_sp:    0x1D;  // 00011101 = mv %B %SP
map mv_b_mdr:   0x1E;  // 00011110 = mv %B %MDR
map mv_b_fl:    0x1F;  // 00011111 = mv %B %FL
map mv_bp_i:    0x20;  // 00100000 = mv %BP %I
map mv_bp_j:    0x21;  // 00100001 = mv %BP %J
map mv_bp_a:    0x22;  // 00100010 = mv %BP %A
map mv_bp_b:    0x23;  // 00100011 = mv %BP %B
map op_error:   0x24;  // 00100100 = mv %BP %BP non valido
map mv_bp_sp:   0x25;  // 00100101 = mv %BP %SP
map mv_bp_mdr:  0x26;  // 00100110 = mv %BP %MDR
map mv_bp_fl:   0x27;  // 00100111 = mv %BP %FL
map mv_sp_i:    0x28;  // 00101000 = mv %SP %I
map mv_sp_j:    0x29;  // 00101001 = mv %SP %J
map mv_sp_a:    0x2A;  // 00101010 = mv %SP %A
map mv_sp_b:    0x2B;  // 00101011 = mv %SP %B
map mv_sp_bp:   0x2C;  // 00101100 = mv %SP %BP
map op_error:   0x2D;  // 00101101 = mv %SP %SP non valido
map mv_sp_mdr:  0x2E;  // 00101110 = mv %SP %MDR
map mv_sp_fl:   0x2F;  // 00101111 = mv %SP %FL
```

```
  map mv_mdr_i:  0x30;  // 00110000 = mv %MDR %I
  map mv_mdr_j:  0x31;  // 00110001 = mv %MDR %J
  map mv_mdr_a:  0x32;  // 00110010 = mv %MDR %A
  map mv_mdr_b:  0x33;  // 00110011 = mv %MDR %B
  map mv_mdr_bp: 0x34;  // 00110100 = mv %MDR %BP
  map mv_mdr_sp: 0x35;  // 00110101 = mv %MDR %SP
  map op_error:  0x36;  // 00110110 = mv %MDR %MDR non valido
  map mv_mdr_fl: 0x37;  // 00110111 = mv %MDR %FL
  map mv_fl_i:   0x38;  // 00111000 = mv %FL %I
  map mv_fl_j:   0x39;  // 00111001 = mv %FL %J
  map mv_fl_a:   0x3A;  // 00111010 = mv %FL %A
  map mv_fl_b:   0x3B;  // 00111011 = mv %FL %B
  map mv_fl_bp:  0x3C;  // 00111100 = mv %FL %BP
  map mv_fl_sp:  0x3D;  // 00111101 = mv %FL %SP
  map mv_fl_mdr: 0x3E;  // 00111110 = mv %FL %MDR
  map op_error:  0x3F;  // 00111111 = mv %FL %FL non valido
  +0[7:0]=0x00;
  operands {
    %1,%2 = { +0[5:3]=%1; +0[2:0]=%2; };
  };
};
op load8 {              // 010001.. = load8
  map load8_i:   0x44;  // 01000100 = load8 %I
  map load8_j:   0x45;  // 01000101 = load8 %J
  map load8:     0x46;  // 01000110 = load8 #...
  +0[7:0]=0x44;
  operands {
    %1 = { +0[1]=0; +0[0]=%1; };
    #1 = { +0[1]=1; +0[0]=0; +1=#1[7:0]; +2=#1[15:8]; };
  };
};
op load16 {             // 010010.. = load16
  map load16_i:  0x48;  // 01001000 = load16 %I
  map load16_j:  0x49;  // 01001001 = load16 %J
  map load16:    0x4A;  // 01001010 = load16 #...
  +0[7:0]=0x48;
```

```
  operands {
    %1 = { +0[1]=0; +0[0]=%1; };
    #1 = { +0[1]=1; +0[0]=0; +1=#1[7:0]; +2=#1[15:8]; };
  };
};
op store8 {                 // 010011.. = store8
  map store8_i:  0x4C;      // 01001100 = store8 %I
  map store8_j:  0x4D;      // 01001101 = store8 %J
  map store8:    0x4E;      // 01001110 = store8 #...
  +0[7:0]=0x4C;
  operands {
    %1 = { +0[1]=0; +0[0]=%1; };
    #1 = { +0[1]=1; +0[0]=0; +1=#1[7:0]; +2=#1[15:8]; };
  };
};
op store16 {                // 010100.. = store16
  map store16_i: 0x50;      // 01010000 = store16 %I
  map store16_j: 0x51;      // 01010001 = store16 %J
  map store16:   0x52;      // 01010010 = store16 #...
  +0[7:0]=0x50;
  operands {
    %1 = { +0[1]=0; +0[0]=%1; };
    #1 = { +0[1]=1; +0[0]=0; +1=#1[7:0]; +2=#1[15:8]; };
  };
};
op cp8 {                    // 0101010. = cp8
  map cp8_ij:    0x54;      // 01010100 = cp8 %I
  map cp8_ji:    0x55;      // 01010101 = cp8 %J
  +0[7:0]=0x54;
  operands {
    %1 = { +0[0]=%1; };
  };
};
op cp16 {                   // 0101011. = cp16
  map cp16_ij:   0x56;      // 01010110 = cp16 %I
  map cp16_ji:   0x57;      // 01010111 = cp16 %J
```

```
  +0[7:0]=0x56;
  operands {
    %1 = { +0[0]=%1; };
  };
};
op return {                 // 010110..
  map return:     0x58;    // 01011000 = return
  +0[7:0]=0x58;
  operands op_0;
};
op call {                   // 010110..
  map call:       0x59;    // 01011001 = call #...
  map call_i:     0x5A;    // 01011010 = call %I
  map call_j:     0x5B;    // 01011011 = call %J
  +0[7:0]=0x58;
  operands {
    #1 = { +0[1]=0; +0[0]=1; +1=#1[7:0]; +2=#1[15:8]; };
    %1 = { +0[1]=1; +0[0]=%1; };
  };
};
op int {
  map int:        0x5C;    // 01011100 = int #...
  +0[7:0]=0x5C;
  operands op_8;
};
op iret {
  map iret:       0x5D;    // 01011101 = iret
  +0[7:0]=0x5D;
  operands op_0;
};
op cleari {
  map cleari:     0x5E;    // 01011110 = clear interrupt flag
  +0[7:0]=0x5E;
  operands op_0;
};
op seti {
```

```
  map seti:       0x5F;  // 01011111 = set interrupt flag
  +0[7:0]=0x5F;
  operands op_0;
};
op ivtl {
  map ivtl:       0x60;  // 01100000 = load IVT location
  +0[7:0]=0x60;
  operands op_16;
};
op jump {
  map jump:       0x61;  // 01100001 = jump #...
  +0[7:0]=0x61;
  operands op_16;
};
op jump8c {
  map jump8c:     0x62;  // 01100010 = jump8c #...
  +0[7:0]=0x62;
  operands op_16;
};
op jump8nc {
  map jump8nc:    0x63;  // 01100011 = jump8nc #...
  +0[7:0]=0x63;
  operands op_16;
};
op jump8z {
  map jump8z:     0x64;  // 01100100 = jump8z #...
  +0[7:0]=0x64;
  operands op_16;
};
op jump8nz {
  map jump8nz:    0x65;  // 01100101 = jump8nz #...
  +0[7:0]=0x65;
  operands op_16;
};
op jump8o {
  map jump8o:     0x66;  // 01100110 = jump8o #...
```

```
  +0[7:0]=0x66;
  operands op_16;
};
op jump8no {
  map jump8no:    0x67;  // 01100111 = jump8no #...
  +0[7:0]=0x67;
  operands op_16;
};
op jump8n {
  map jump8n:     0x68;  // 01101000 = jump8n #...
  +0[7:0]=0x66;
  operands op_16;
};
op jump8nn {
  map jump8nn:    0x69;  // 01101001 = jump8nn #...
  map op_error:   0x6A;  // 01101010 = non valido
  map op_error:   0x6B;  // 01101011 = non valido
  map op_error:   0x6C;  // 01101100 = non valido
  map op_error:   0x6D;  // 01101101 = non valido
  map op_error:   0x6E;  // 01101110 = non valido
  map op_error:   0x6F;  // 01101111 = non valido
  map op_error:   0x70;  // 01110000 = non valido
  map op_error:   0x71;  // 01110001 = non valido
  +0[7:0]=0x67;
  operands op_16;
};
op jump16c {
  map jump16c:    0x72;  // 01110010 = jump16c #...
  +0[7:0]=0x72;
  operands op_16;
};
op jump16nc {
  map jump16nc:   0x73;  // 01110011 = jump16nc #...
  +0[7:0]=0x73;
  operands op_16;
};
```

```
op jump16z {
  map jump16z:   0x74;  // 01110100 = jump16z #...
  +0[7:0]=0x74;
  operands op_16;
};
op jump16nz {
  map jump16nz:  0x75;  // 01110101 = jump16nz #...
  +0[7:0]=0x75;
  operands op_16;
};
op jump16o {
  map jump16o:   0x76;  // 01110110 = jump16o #...
  +0[7:0]=0x76;
  operands op_16;
};
op jump16no {
  map jump16no:  0x77;  // 01110111 = jump16no #...
  +0[7:0]=0x77;
  operands op_16;
};
op jump16n {
  map jump16n:   0x78;  // 01111000 = jump16n #...
  +0[7:0]=0x76;
  operands op_16;
};
op jump16nn {
  map jump16nn:  0x79;  // 01111001 = jump16 #...
  map op_error:  0x7A;  // 01111010 = non valido
  map op_error:  0x7B;  // 01111011 = non valido
  map op_error:  0x7C;  // 01111100 = non valido
  map op_error:  0x7D;  // 01111101 = non valido
  map op_error:  0x7E;  // 01111110 = non valido
  map op_error:  0x7F;  // 01111111 = non valido
  +0[7:0]=0x77;
  operands op_16;
};
```

```
op push8 {                  // 10000... = push8
  map push8_i:   0x80;      // 10000000 = push8 %I
  map push8_j:   0x81;      // 10000001 = push8 %J
  map push8_a:   0x82;      // 10000010 = push8 %A
  map push8_b:   0x83;      // 10000011 = push8 %B
  map push8_bp:  0x84;      // 10000100 = push8 %BP
  map op_error:  0x85;      // 10000101 = push8 %SP non valido
  map push8_mdr: 0x86;      // 10000110 = push8 %MDR
  map push8_fl:  0x87;      // 10000111 = push8 %FL
  +0[7:0]=0x80;
  operands {
    %1 = { +0[2:0]=%1; };
  };
};
op pop8 {                   // 10001... = pop8
  map pop8_i:    0x88;      // 10001000 = pop8 %I
  map pop8_j:    0x89;      // 10001001 = pop8 %J
  map pop8_a:    0x8A;      // 10001010 = pop8 %A
  map pop8_b:    0x8B;      // 10001011 = pop8 %B
  map pop8_bp:   0x8C;      // 10001100 = pop8 %BP
  map op_error:  0x8D;      // 10001101 = pop8 %SP non valido
  map pop8_mdr:  0x8E;      // 10001110 = pop8 %MDR
  map pop8_fl:   0x8F;      // 10001111 = pop8 %FL
  +0[7:0]=0x88;
  operands {
    %1 = { +0[2:0]=%1; };
  };
};
op push16 {                 // 10010... = push16
  map push16_i:   0x90;     // 10010000 = push16 %I
  map push16_j:   0x91;     // 10010001 = push16 %J
  map push16_a:   0x92;     // 10010010 = push16 %A
  map push16_b:   0x93;     // 10010011 = push16 %B
  map push16_bp:  0x94;     // 10010100 = push16 %BP
  map op_error:   0x95;     // 10010101 = push16 %SP non valido
  map push16_mdr: 0x96;     // 10010110 = push16 %MDR
```

```
  map push16_fl:  0x97; // 10010111 = push16 %FL
  +0[7:0]=0x90;
  operands {
    %1 = { +0[2:0]=%1; };
  };
};
op pop16 {                // 10011... = pop16
  map pop16_i:   0x98;  // 10011000 = pop16 %I
  map pop16_j:   0x99;  // 10011001 = pop16 %J
  map pop16_a:   0x9A;  // 10011010 = pop16 %A
  map pop16_b:   0x9B;  // 10011011 = pop16 %B
  map pop16_bp:  0x9C;  // 10011100 = pop16 %BP
  map op_error:  0x9D;  // 10011101 = pop16 %SP non valido
  map pop16_mdr: 0x9E;  // 10011110 = pop16 %MDR
  map pop16_fl:  0x9F;  // 10011111 = pop16 %FL
  +0[7:0]=0x98;
  operands {
    %1 = { +0[2:0]=%1; };
  };
};
op c8to16u {
  map c8to16u:   0xA0;  // 10100000
  +0[7:0]=0xA0;
  operands op_0;
};
op c8to16s {
  map c8to16s:   0xA1;  // 10100001
  +0[7:0]=0xA1;
  operands op_0;
};
op equal {
  map equal:     0xA2;  // 10100010
  +0[7:0]=0xA2;
  operands op_0;
};
op not {
```

```
  map not:        0xA3;  // 10100011
  +0[7:0]=0xA3;
  operands op_0;
};
op and {
  map and:        0xA4;  // 10100100
  +0[7:0]=0xA4;
  operands op_0;
};
op nand {
  map nand:       0xA5;  // 10100101
  +0[7:0]=0xA5;
  operands op_0;
};
op or {
  map or:         0xA6;  // 10100110
  +0[7:0]=0xA6;
  operands op_0;
};
op nor {
  map nor:        0xA7;  // 10100111
  +0[7:0]=0xA7;
  operands op_0;
};
op xor {
  map xor:        0xA8;  // 10101000
  +0[7:0]=0xA8;
  operands op_0;
};
op nxor {
  map nxor:       0xA9;  // 10101001
  +0[7:0]=0xA9;
  operands op_0;
};
op add {
  map add:        0xAA;  // 10101010
```

```
  +0[7:0]=0xAA;
  operands op_0;
};
op sub {
  map sub:        0xAB;  // 10101011
  +0[7:0]=0xAB;
  operands op_0;
};
op addc8 {
  map addc8:      0xAC;  // 10101100
  +0[7:0]=0xAC;
  operands op_0;
};
op subb8 {
  map subb8:      0xAD;  // 10101101
  +0[7:0]=0xAD;
  operands op_0;
};
op addc16 {
  map addc16:     0xAE;  // 10101110
  +0[7:0]=0xAE;
  operands op_0;
};
op subb16 {
  map subb16:     0xAF;  // 10101111
  +0[7:0]=0xAF;
  operands op_0;
};
op lshl8 {
  map lshl8:      0xB0;  // 10110000
  +0[7:0]=0xB0;
  operands op_0;
};
op lshr8 {
  map lshr8:      0xB1;  // 10110001
  +0[7:0]=0xB1;
```

```
  operands op_0;
};
op ashl8 {
  map ashl8:      0xB2;  // 10110010
  +0[7:0]=0xB2;
  operands op_0;
};
op ashr8 {
  map ashr8:      0xB3;  // 10110011
  +0[7:0]=0xB3;
  operands op_0;
};
op rotcl8 {
  map rotcl8:     0xB4;  // 10110100
  +0[7:0]=0xB4;
  operands op_0;
};
op rotcr8 {
  map rotcr8:     0xB5;  // 10110101
  +0[7:0]=0xB5;
  operands op_0;
};
op rotl8 {
  map rotl8:      0xB6;  // 10110110
  +0[7:0]=0xB6;
  operands op_0;
};
op rotr8 {
  map rotr8:      0xB7;  // 10110111
  +0[7:0]=0xB7;
  operands op_0;
};
op lshl16 {
  map lshl16:     0xB8;  // 10111000
  +0[7:0]=0xB8;
  operands op_0;
```

```
};
op lshr16 {
  map lshr16:    0xB9;  // 10111001
  +0[7:0]=0xB9;
  operands op_0;
};
op ashl16 {
  map ashl16:    0xBA;  // 10111010
  +0[7:0]=0xBA;
  operands op_0;
};
op ashr16 {
  map ashr16:    0xBB;  // 10111011
  +0[7:0]=0xBB;
  operands op_0;
};
op rotcl16 {
  map rotcl16:   0xBC;  // 10111100
  +0[7:0]=0xBC;
  operands op_0;
};
op rotcr16 {
  map rotcr16:   0xBD;  // 10111101
  +0[7:0]=0xBD;
  operands op_0;
};
op rotl16 {
  map rotl16:    0xBE;  // 10111110
  +0[7:0]=0xBE;
  operands op_0;
};
op rotr16 {
  map rotr16:    0xBF;  // 10111111
  +0[7:0]=0xBF;
  operands op_0;
};
```

```
op in {
  map in:         0xC0;  // 11000000
  +0[7:0]=0xC0;
  operands op_8;
};
op out {
  map out:        0xC1;  // 11000001
  +0[7:0]=0xC1;
  operands op_8;
};
op is_ack {
  map is_ack:     0xC2;  // 11000010
  map op_error:   0xC3;  // 11000011
  +0[7:0]=0xC2;
  operands {
    #1,#2 = { +1=#1[7:0]; +2=#1[7:0]; +3=#2[17:8]; };
  };
};
op clearc {
  map clearc:     0xC4;  // 11000100
  +0[7:0]=0xC4;
  operands op_0;
};
op setc {
  map setc:       0xC5;  // 11000101
  +0[7:0]=0xC5;
  operands op_0;
};
op cmp {
  map cmp:        0xC6;  // 11000110
  +0[7:0]=0xC6;
  operands op_0;
};
op test {
  map test:       0xC7;  // 11000111
```

```
  +0[7:0]=0xC7;
  operands op_0;
};
op imrl {
  map imrl:       0xC8;  // 11001000     // IMR load
  map op_error:   0xC9;  // 11001001
  map op_error:   0xCA;  // 11001010
  map op_error:   0xCB;  // 11001011
  map op_error:   0xCC;  // 11001100
  map op_error:   0xCD;  // 11001101
  map op_error:   0xCE;  // 11001110
  map op_error:   0xCF;  // 11001111
  +0[7:0]=0xC8;
  operands op_8;
};
op inc {
  map inc_i:      0xD0;  // 11010000     // inc %I
  map inc_j:      0xD1;  // 11010001     // inc %J
  map inc_a:      0xD2;  // 11010010     // inc %A
  map inc_b:      0xD3;  // 11010011     // inc %B
  map inc_bp:     0xD4;  // 11010100     // inc %BP
  map inc_sp:     0xD5;  // 11010101     // inc %SP
  map inc_mdr:    0xD6;  // 11010110     // inc %MDR
  map inc_fl:     0xD7;  // 11010111     // inc %FL
  +0[7:0]=0xD0;
  operands {
    %1 = { +0[2:0]=%1; };
  };
};
op dec {
  map dec_i:      0xD8;  // 11011000     // dec %I
  map dec_j:      0xD9;  // 11011001     // dec %J
  map dec_a:      0xDA;  // 11011010     // dec %A
  map dec_b:      0xDB;  // 11011011     // dec %B
  map dec_bp:     0xDC;  // 11011100     // dec %BP
  map dec_sp:     0xDD;  // 11011101     // dec %SP
```

```
  map dec_mdr:    0xDE;  // 11011110      // dec %MDR
  map dec_fl:     0xDF;  // 11011111      // dec %FL
  map op_error:   0xE0;  // 11100001
  map op_error:   0xE1;  // 11100010
  map op_error:   0xE2;  // 11100011
  map op_error:   0xE4;  // 11100100
  map op_error:   0xE5;  // 11100101
  map op_error:   0xE6;  // 11100110
  map op_error:   0xE7;  // 11100111
  map op_error:   0xE8;  // 11101000
  map op_error:   0xE9;  // 11101001
  map op_error:   0xEA;  // 11101010
  map op_error:   0xEB;  // 11101011
  map op_error:   0xEC;  // 11101100
  map op_error:   0xED;  // 11101101
  map op_error:   0xEE;  // 11101110
  map op_error:   0xEF;  // 11101111
  map op_error:   0xF0;  // 11110001
  map op_error:   0xF1;  // 11110010
  map op_error:   0xF2;  // 11110011
  map op_error:   0xF4;  // 11110100
  map op_error:   0xF5;  // 11110101
  map op_error:   0xF6;  // 11110110
  map op_error:   0xF7;  // 11110111
  map op_error:   0xF8;  // 11111000
  map op_error:   0xF9;  // 11111001
  map op_error:   0xFA;  // 11111010
  map op_error:   0xFB;  // 11111011
  map op_error:   0xFC;  // 11111100
  map op_error:   0xFD;  // 11111101
  map op_error:   0xFE;  // 11111110
  +0[7:0]=0xD8;
  operands {
    %1 = { +0[2:0]=%1; };
  };
};
```

Tabella u116.34. Elenco completo dei codici operativi con le macroistruzioni corrispondenti.

Sintassi del macrocodice	Codice operativo binario	Descrizione
`nop`	00000000	Non fa alcunché.
`mv %`***src***`, %`***dst***	`00`***sss ddd***	Copia il contenuto del registro ***src*** nel registro ***dst***; nel codice operativo i bit ***sss*** rappresentano il primo registro, mentre i bit ***ddd*** rappresentano il secondo. Tra i codici operativi sono esclusi quelli che copierebbero lo stesso registro su se stesso; pertanto, il codice 00000000_2 rimane destinato all'istruzione '**`nop`**', in quanto rappresenterebbe la copia del registro ***A*** su se stesso.

Sintassi del macrocodice	Codice operativo binario	Descrizione

`load8 %`***indice***
`load8 #`***indice***
`load16 %`***indice***
`load16 #`***indice***
	`0100010`***i***
	`01000110`
	`0100100`***i***
	`01001010`
		Carica nel registro ***MDR*** un valore a 8 o 16 bit dalla memoria. L'argomento può essere un registro indice (***I*** o ***J***) e in tal caso si utilizza il bit ***i*** del codice operativo per distinguerlo; altrimenti può essere direttamente l'indirizzo della memoria a cui ci si riferisce e i due bit meno significativi del codice operativo sono pari a 10_2.

`store8 %`***indice***
`store8 #`***indice***
`store16 %`***indice***
`store16 #`***indice***
	`0100110`***i***
	`01001110`
	`0101000`***i***
	`01010010`
		Registra in memoria (8 o 16 bit), all'indirizzo corrispondente all'argomento, il valore contenuto nel registro ***MDR***. Quando l'argomento è un registro, può trattarsi solo di ***I*** o ***J*** e tale informazione si colloca nel bit ***i*** del codice operativo. Se l'argomento è rappresentato invece un numero, la parte finale del codice operativo diventa 10_2.

Sintassi del macrocodice	Codice operativo binario	Descrizione
`cp8 %`***src*** `cp16 %`***src***	`0101010`***s*** `0101011`***s***	Copia in memoria, dalla posizione indicata nel registro indice ***src*** (che può essere ***I*** o ***J***), alla posizione indicata dall'altro registro indice, 8 o 16 bit, incrementando successivamente i due registri indice.

`return`
 	
`call #`***indirizzo***
 	
`call #`***indice***
	`01011000`
	
	`01011001`
	
	`0101101`***i***
		Uscita e chiamata di una routine. Il bit ***i*** rappresenta un registro indice (***I*** o ***J***).

Sintassi del macrocodice	Codice operativo binario	Descrizione

`int #`***n_interrupt***
 	
`iret`
	`01011100`
	
	`01011101`
		Esegue una chiamata di interruzione e il ritorno dalla stessa; il numero ***n_interrupt*** è a 8 bit, ma può andare solo da 0 a 16. Nella chiamata vengono salvati nella pila dei dati il registro ***FL*** e il registro ***PC***, quindi nel registro ***FL*** vengono disattivati i bit che consentono la generazione di interruzioni hardware (IRQ); al ritorno dalla chiamata, vengono ripristinati il registro ***PC*** e il registro ***FL***.

`cleari`
 	
`seti`
	`01011110`
	
	`01011111`
		Azzera o attiva i bit di abilitazione delle interruzioni hardware (IRQ) contenuti nel registro ***FL***.

`ivtl #`***indirizzo***	`01100000`	Carica nel registro ***IVT*** l'indirizzo della tabella IVT (*interrupt vector table*) in memoria.
`jump #`***indirizzo***	`01100001`	Salta all'esecuzione dell'istruzione contenuta nell'indirizzo indicato. L'argomento è a 16 bit.

Sintassi del macrocodice	Codice operativo binario	Descrizione
`jump8c #`***indirizzo***	`01100010`	Salta all'esecuzione dell'istruzione contenuta nell'indirizzo indicato se: l'indicatore di riporto a 8 bit è attivo; l'indicatore di riporto a 8 bit non è attivo; l'indicatore di zero a 8 bit è attivo; l'indicatore di zero a 8 bit non è attivo; l'indicatore di straripamento a 8 bit è attivo; l'indicatore di straripamento a 8 bit non è attivo; l'indicatore di segno a 8 bit è attivo; l'indicatore di segno a 8 bit non è attivo.
`jump8nc #`***indirizzo***	`01100011`	
`jump8z #`***indirizzo***	`01100100`	
`jump8nz #`***indirizzo***	`01100101`	
`jump8o #`***indirizzo***	`01100110`	
`jump8no #`***indirizzo***	`01100111`	
`jump8n #`***indirizzo***	`01101000`	
`jump8nn #`***indirizzo***	`01101001`	

Sintassi del macrocodice	Codice operativo binario	Descrizione

`jump16c #`***indirizzo***
`jump16nc #`***indirizzo***
`jump16z #`***indirizzo***
`jump16nz #`***indirizzo***
`jump16o #`***indirizzo***
`jump16no #`***indirizzo***
`jump16n #`***indirizzo***
`jump16nn #`***indirizzo***
	`01110010`
	`01110011`
	`01110100`
	`01110101`
	`01110110`
	`01110111`
	`01111000`
	`01111001`
		Salta all'esecuzione dell'istruzione contenuta nell'indirizzo indicato se: l'indicatore di riporto a 16 bit è attivo; l'indicatore di riporto a 16 bit non è attivo; l'indicatore di zero a 16 bit è attivo; l'indicatore di zero a 16 bit non è attivo; l'indicatore di straripamento a 16 bit è attivo; l'indicatore di straripamento a 16 bit non è attivo; l'indicatore di segno a 16 bit è attivo; l'indicatore di segno a 16 bit non è attivo.

`push8 #`***registro*** `pop8 #`***registro***	`10000`***rrr*** `10001`***rrr***	Inserisce nella pila dei dati, oppure recupera dalla pila dei dati, gli otto bit meno significativi del registro indicato, il quale è annotato negli ultimi tre bit del codice operativo. Il caso del registro ***SP*** non è valido, in quanto si tratta dell'indice della pila che non ha senso accantonare.

Sintassi del macrocodice	Codice operativo binario	Descrizione
`push16 #`***registro*** `pop16 #`***registro***	`10010`***rrr*** `10011`***rrr***	Inserisce nella pila dei dati, oppure recupera dalla pila dei dati, gli otto bit meno significativi del registro indicato, il quale è annotato negli ultimi tre bit del codice operativo. Il caso del registro ***SP*** non è valido, in quanto si tratta dell'indice della pila che non ha senso accantonare.

`c81016u`
 	
`c81016s`
	`10100000`
	
	`10100001`
		Estende il contenuto a 8 bit del registro ***A*** in un valore a 16 bit: nel primo caso trattando il valore senza segno, nel secondo trattandolo con segno.

Sintassi del macrocodice	Codice operativo binario	Descrizione
`equal`	`10100010`	Operazione logica a partire dal valore dei registri ***A*** e ***B***, mettendo il risultato nel registro ***A***.
`not`	`10100011`	
`and`	`10100100`	
`nand`	`10100101`	
`or`	`10100110`	
`nor`	`10100111`	
`xor`	`101001000`	
`nxor`	`10101001`	
`add`	`10101010`	Esegue l'operazione ***A*+*B***, oppure ***A*−*B***, senza tenere conto del riporto preesistente. Il risultato aggiorna il registro ***A***.
`sub`	`10101011`	
`addc8`	`10101100`	Esegue l'operazione ***A*+*B***, oppure ***A*−*B***, tenendo conto del riporto preesistente e dell'estensione dei valori da sommare o sottrarre. Il risultato aggiorna il registro ***A***.
`subb8`	`10101101`	
`addc16`	`10101110`	
`subb16`	`10101111`	

Sintassi del macrocodice	Codice operativo binario	Descrizione
`lshl8`	`10110000`	Scorrimenti e rotazioni a 8 bit, sul valore del registro ***A***, mettendo il risultato nello stesso registro ***A***.
`lshr8`	`10110001`	
`ashl8`	`10110010`	
`ashr8`	`10110011`	
`rotcl8`	`10110100`	
`rotcr8`	`10110101`	
`rotl8`	`10110110`	
`rotr8`	`10110111`	
`lshl16`	`10111000`	Scorrimenti e rotazioni a 16 bit, sul valore del registro ***A***, mettendo il risultato nello stesso registro ***A***.
`lshr16`	`10111001`	
`ashl16`	`10111010`	
`ashr16`	`10111011`	
`rotcl16`	`10111100`	
`rotcr16`	`10111101`	
`rotl16`	`10111110`	
`rotr16`	`10111111`	

Sintassi del macrocodice	Codice operativo binario	Descrizione

`in #`***io***
`out` ***io***
`is_ack #`***io***`, #`***indirizzo***
	`11000000`
	`11000001`
	`11000010`
		Legge o scrive un valore nell'indirizzo di I/O indicato. L'indirizzo di I/O è un numero a 8 bit. Nel caso di '**`is_ack`**', si valuta la presenza di una conferma da parte del dispositivo e, se presente, si salta all'istruzione corrispondente all'indirizzo che appare come ultimo argomento. Attualmente solo il dispositivo dello schermo prevede questo tipo di interrogazione.

`clearc` `setc`	`11000100` `11000101`	Azzera o attiva il bit di riporto nel registro ***FL***.
`cmp` `test`	`11000110` `11000111`	Confronta il contenuto di ***A*** e ***B*** simulando una sottrazione o l'operatore logico AND, al solo scopo di aggiornare il registro ***FL***.

Sintassi del macrocodice	Codice operativo binario	Descrizione
`imrl #`***maschera***	`11001000`	Carica la maschera delle interruzioni hardware accettate. L'argomento è da 8 bit, ma valgono solo i 4 bit meno significativi, in quanto esistono solo quattro interruzioni hardware.
`inc %`***registro*** `dec %`***registro***	`11010`***rrr*** `11011`***rrr***	Incrementa o decrementa, di una unità, il registro indicato, il quale si annota negli ultimi tre bit del codice operativo.
`stop`	`11111111`	Ferma la CPU, bloccando il flusso del segnale di clock.

Microcodice

Listato u116.35. Campi in cui si suddividono i bit che rappresentano il microcodice, nelle memorie da ***micro0*** a ***micro4***.

```
field ctrl[1:0]     = {nop=0, stop=1, load=2};
field pc[10:2]      = {br=1, bw=2, aux=4, low=8, high=16,
                       p1=32, p2=64, m1=128, m2=256};
field sel[15:11]    = {if_carry_8=1, if_not_carry_8=3,
                       if_zero_8=5, if_not_zero_8=7,
                       if_negative_8=9, if_not_negative_8=11,
                       if_overflow_8=13, if_not_overflow_8=15,
                       if_carry_16=1, if_not_carry_16=3,
                       if_zero_16=5, if_not_zero_16=7,
```

```
                        if_negative_16=9, if_not_negative_16=11,
                        if_overflow_16=13, if_not_overflow_16=15};
field mdr[24:16]   = {br=1, bw=2, aux=4, low=8, high=16,
                        p1=32, p2=64, m1=128, m2=256};
field i[33:25]     = {br=1, bw=2, aux=4, low=8, high=16,
                        p1=32, p2=64, m1=128, m2=256};
field j[42:34]     = {br=1, bw=2, aux=4, low=8, high=16,
                        p1=32, p2=64, m1=128, m2=256};
field ram[47:43]   = {br=1, bw=2, aux=4, p=0, i=8, j=16, s=24};
field ivt[50:48]   = {br=1, bw=2, inta=0, intb=4};
field ir[59:51]    = {br=1, bw=2, aux=4, low=8, high=16,
                        p1=32, p2=64, m1=128, m2=256};
field bus[77:60]   = {bw=0x10000, aux=0x20000};
field irq[80:78]   = {br=1, bw=2, done=4};
field sp[89:81]    = {br=1, bw=2, aux=4, low=8, high=16,
                        p1=32, p2=64, m1=128, m2=256};
field bp[98:90]    = {br=1, bw=2, aux=4, low=8, high=16,
                        p1=32, p2=64, m1=128, m2=256};
field b[107:99]    = {br=1, bw=2, aux=4, low=8, high=16,
                        p1=32, p2=64, m1=128, m2=256};
field fl[116:108]  = {br=1, bw=2, aux=4, low=8, high=16,
                        p1=32, p2=64, m1=128, m2=256};
field alu[127:117] = {bw=1, aux=2, sign=4, rank8=0, rank16=8,
                        a=0, and=16, or=32, xor=48,
                        nxor=64, nor=80, nand=96, not=112,
                        add=256, sub=228, addc=320, subb=352,
                        lshl=512, lshr=528, ashl=484, ashr=560,
                        rotcl=576, rotcr=592,
                        rotl=768, rotr=784,
                        clearc=1024, clearz=1040, clearn=1056,
                        clearo=1072, cleari=1088,
                        setc=1152, setz=1168, setn=1184,
                        seto=1200, seti=1216};
field a[136:128]   = {br=1, bw=2, aux=4, low=8, high=16,
                        p1=32, p2=64, m1=128, m2=256};
field ioa[145:137] = {br=1, bw=2, aux=4, low=8, high=16,
```

```
                          p1=32, p2=64, m1=128, m2=256};
field ioc[149:146] = {br=1, bw=2, req=4, isack=8};
```

Listato u116.36. Dichiarazione del microcodice.

```
begin microcode @ 0
//
// fetch:
//      IR  <-- RAM[pc++]; load;
//
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
//
nop:
    // fetch:
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
op_error:
    // INT 0
    // push FL
    sp=m2;                                      // SP <-- (SP - 2)
    ram=br ram=s fl=bw fl=low  sp=p1;           // RAM[sp++] <- FL[7:0];
    ram=br ram=s fl=bw fl=high sp=m1;           // RAM[sp--] <- FL[15:8];
    // reset interrupt enable flag
    fl=br fl=aux alu=cleari;
    // push PC
    sp=m2;                                      // SP <-- (SP - 2)
    ram=br ram=s pc=bw pc=low  sp=p1;           // RAM[sp++] <- PC[7:0];
    ram=br ram=s pc=bw pc=high sp=m1;           // RAM[sp--] <- PC[15:8];
    // push I
    sp=m2;                                      // SP <-- (SP - 2)
    ram=br ram=s i=bw i=low  sp=p1;             // RAM[sp++] <- I[7:0];
    ram=br ram=s i=bw i=high sp=m1;             // RAM[sp--] <- I[15:8];
    //
    i=br ivt=bw ivt=intb  bus=bw bus=aux bus=0;     // I <- IVT <- 0;
    pc=br pc=low  ram=bw ram=i i=p1;            // PC[7:0]  <-- RAM[i++]
    pc=br pc=high ram=bw ram=i i=m1;            // PC[15:7] <-- RAM[i--]
    // pop I
    i=br i=low  ram=bw ram=s sp=p1;             // I[7:0] <-- RAM[sp++];
    i=br i=high ram=bw ram=s sp=p1;             // I[15:0] <-- RAM[sp++];
    // fetch
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_i_j:
    j=br i=bw                                   // J <- I, fetch;
```

```
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_i_a:
    a=br i=bw                                   // A <- I, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_i_b:
    b=br i=bw                                   // B <- I, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_i_bp:
    bp=br i=bw                                  // BP <- I, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_i_sp:
    sp=br i=bw                                  // SP <- I, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_i_mdr:
    mdr=br i=bw                                 // MDR <- I, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_i_fl:
    fl=br i=bw                                  // FL <- I, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_j_i:
    i=br j=bw                                   // I <- J, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_j_a:
    a=br j=bw                                   // A <- J, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_j_b:
    b=br j=bw                                   // B <- J, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_j_bp:
    bp=br j=bw                                  // BP <- J, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_j_sp:
    sp=br j=bw                                  // SP <- J, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_j_mdr:
    mdr=br j=bw                                 // MDR <- J, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_j_fl:
    fl=br j=bw                                  // FL <- J, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_a_i:
    i=br a=bw                                   // I <- A, fetch;
```

```
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_a_j:
    j=br a=bw                                   // J <- A, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_a_b:
    b=br a=bw                                   // B <- A, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_a_bp:
    bp=br a=bw                                  // BP <- A, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_a_sp:
    sp=br a=bw                                  // SP <- A, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_a_mdr:
    mdr=br a=bw                                 // MDR <- A, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_a_fl:
    fl=br a=bw                                  // FL <- A, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_b_i:
    i=br b=bw                                   // I <- B, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_b_j:
    j=br b=bw                                   // J <- B, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_b_a:
    a=br b=bw                                   // A <- B, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_b_bp:
    bp=br b=bw                                  // BP <- B, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_b_sp:
    sp=br b=bw                                  // SP <- B, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_b_mdr:
    mdr=br b=bw                                 // MDR <- B, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_b_fl:
    fl=br b=bw                                  // FL <- B, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_bp_i:
    i=br bp=bw                                  // I <- BP, fetch;
```

```
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_bp_j:
    j=br bp=bw                                  // J <- BP, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_bp_a:
    a=br bp=bw                                  // A <- BP, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_bp_b:
    b=br bp=bw                                  // B <- BP, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_bp_sp:
    sp=br bp=bw                                 // SP <- BP, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_bp_mdr:
    mdr=br bp=bw                                // MDR <- BP, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_bp_fl:
    fl=br bp=bw                                 // FL <- BP, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_sp_i:
    i=br sp=bw                                  // I <- SP, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_sp_j:
    j=br sp=bw                                  // J <- SP, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_sp_a:
    a=br sp=bw                                  // A <- SP, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_sp_bp:
    bp=br sp=bw                                 // BP <- SP, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_sp_b:
    b=br sp=bw                                  // B <- SP, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_sp_mdr:
    mdr=br sp=bw                                // MDR <- SP, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_sp_fl:
    fl=br sp=bw                                 // FL <- SP, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_mdr_i:
    i=br mdr=bw                                 // I <- MDR, fetch;
```

```
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_mdr_j:
    j=br mdr=bw                                  // J <- MDR, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_mdr_bp:
    bp=br mdr=bw                                 // BP <- MDR, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_mdr_sp:
    sp=br mdr=bw                                 // SP <- MDR, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_mdr_b:
    b=br mdr=bw                                  // B <- MDR, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_mdr_a:
    a=br mdr=bw                                  // A <- MDR, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_mdr_fl:
    fl=br mdr=bw                                 // FL <- MDR, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_fl_i:
    i=br fl=bw                                   // I <- FL, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_fl_j:
    j=br fl=bw                                   // J <- FL, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_fl_a:
    a=br fl=bw                                   // A <- FL, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_fl_bp:
    bp=br fl=bw                                  // BP <- FL, fetch
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_fl_sp:
    sp=br fl=bw                                  // SP <- FL, fetch
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_fl_mdr:
    mdr=br fl=bw                                 // MDR <- FL, fetch
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
mv_fl_b:
    fl=br b=bw                                   // B <- FL, fetch
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
load8_i:
    mdr=br ram=bw ram=i;                         // MDR <- RAM[i];
```

```
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
load8_j:
    mdr=br ram=bw ram=j;                    // MDR <- RAM[j];
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
load8:
    i=br i=low  ram=bw ram=p pc=p1;         // I[7:0] <- RAM[pc++];
    i=br i=high ram=bw ram=p pc=p1;         // I[15:8] <- RAM[pc++];
    mdr=br ram=bw ram=i;                    // MDR <- RAM[i];
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
load16_i:
    mdr=br mdr=low  ram=bw ram=i i=p1;      // MDR[7:0] <- RAM[i++];
    mdr=br mdr=high ram=bw ram=i i=m1;      // MDR[15:8] <- RAM[i--];
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
load16_j:
    mdr=br mdr=low  ram=bw ram=j j=p1;      // MDR[7:0] <- RAM[j++];
    mdr=br mdr=high ram=bw ram=j j=m1;      // MDR[15:8] <- RAM[j--];
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
load16:
    i=br i=low  ram=bw ram=p pc=p1;         // I[7:0] <- RAM[pc++];
    i=br i=high ram=bw ram=p pc=p1;         // I[15:8] <- RAM[pc++];
    mdr=br mdr=low  ram=bw ram=i i=p1;      // MDR[7:0] <- RAM[i++];
    mdr=br mdr=high ram=bw ram=i i=m1;      // MDR[15:8] <- RAM[i--];
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
store8_i:
    ram=br ram=i mdr=bw;                    // RAM[i] <- MDR[7:0];
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
store8_j:
    ram=br ram=j mdr=bw;                    // RAM[j] <- MDR[7:0];
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
store8:
    i=br i=low  ram=bw ram=p pc=p1;         // I[7:0] <- RAM[pc++];
    i=br i=high ram=bw ram=p pc=p1;         // I[15:8] <- RAM[pc++];
    ram=br ram=i mdr=bw;                    // RAM[i] <- MDR[7:0];
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
store16_i:
    ram=br ram=i mdr=bw mdr=low  i=p1;      // RAM[i++] <- MDR[7:0];
    ram=br ram=i mdr=bw mdr=high i=m1;      // RAM[i--] <- MDR[15:8];
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
store16_j:
    ram=br ram=j mdr=bw mdr=low  j=p1;      // RAM[j++] <- MDR[7:0];
    ram=br ram=j mdr=bw mdr=high j=m1;      // RAM[j--] <- MDR[15:8];
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
```

```
store16:
    i=br i=low  ram=bw ram=p pc=p1;          // I[7:0] <- RAM[pc++];
    i=br i=high ram=bw ram=p pc=p1;          // I[15:8] <- RAM[pc++];
    ram=br ram=i mdr=bw mdr=low  i=p1;       // RAM[i++] <- MDR[7:0];
    ram=br ram=i mdr=bw mdr=high i=m1;       // RAM[i--] <- MDR[15:8];
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
cp8_ij:
    mdr=br ram=bw ram=i i=p1;                // MDR[7:0] <- RAM[i++];
    ram=br ram=j mdr=bw j=p1;                // RAM[j++] <- MDR[7:0];
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
cp8_ji:
    mdr=br ram=bw ram=j j=p1;                // MDR[7:0] <- RAM[j++];
    ram=br ram=i mdr=bw i=p1;                // RAM[i++] <- MDR[7:0];
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
cp16_ij:
    mdr=br mdr=low  ram=bw ram=i i=p1;       // MDR[7:0] <- RAM[i++];
    mdr=br mdr=high ram=bw ram=i i=p1;       // MDR[15:8] <- RAM[i++];
    ram=br ram=j mdr=bw mdr=low  j=p1;       // RAM[j++] <- MDR[7:0];
    ram=br ram=j mdr=bw mdr=high j=p1;       // RAM[j++] <- MDR[15:0];
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
cp16_ji:
    mdr=br mdr=low  ram=bw ram=j j=p1;       // MDR[7:0] <- RAM[j++];
    mdr=br mdr=high ram=bw ram=j j=p1;       // MDR[15:8] <- RAM[j++];
    ram=br ram=i mdr=bw mdr=low  i=p1;       // RAM[i++] <- MDR[7:0];
    ram=br ram=i mdr=bw mdr=high i=p1;       // RAM[i++] <- MDR[15:0];
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
jump:
    i=br pc=bw;                              // I <- PC
    pc=br pc=low  ram=bw ram=i i=p1;         // PC[7:0]  <-- RAM[i++]
    pc=br pc=high ram=bw ram=i i=m1;         // PC[15:7] <-- RAM[i--]
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
jump8c:
    mdr=br mdr=low  ram=bw ram=p pc=p1;      // MDR[7:0] <-- RAM[pc++]
    mdr=br mdr=high ram=bw ram=p pc=p1;      // MDR[15:7] <-- RAM[pc++]
    pc=br sel=if_carry_8;                    // PC = (carry8?MDR:PC)
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
jump8nc:
    mdr=br mdr=low  ram=bw ram=p pc=p1;      // MDR[7:0] <-- RAM[pc++]
    mdr=br mdr=high ram=bw ram=p pc=p1;      // MDR[15:7] <-- RAM[pc++]
    pc=br sel=if_not_carry_8;                // PC = (not_carry8?MDR:PC)
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
jump8z:
```

```
    mdr=br mdr=low  ram=bw ram=p pc=p1;      // MDR[7:0] <-- RAM[pc++]
    mdr=br mdr=high ram=bw ram=p pc=p1;      // MDR[15:7] <-- RAM[pc++]
    pc=br sel=if_zero_8;                     // PC = (zero8?MDR:PC)
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
jump8nz:
    mdr=br mdr=low  ram=bw ram=p pc=p1;      // MDR[7:0] <-- RAM[pc++]
    mdr=br mdr=high ram=bw ram=p pc=p1;      // MDR[15:7] <-- RAM[pc++]
    pc=br sel=if_not_zero_8;                 // PC = (not_zero8?MDR:PC)
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
jump8o:
    mdr=br mdr=low  ram=bw ram=p pc=p1;      // MDR[7:0] <-- RAM[pc++]
    mdr=br mdr=high ram=bw ram=p pc=p1;      // MDR[15:7] <-- RAM[pc++]
    pc=br sel=if_overflow_8;                 // PC = (overflow8?MDR:PC)
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
jump8no:
    mdr=br mdr=low  ram=bw ram=p pc=p1;      // MDR[7:0] <-- RAM[pc++]
    mdr=br mdr=high ram=bw ram=p pc=p1;      // MDR[15:7] <-- RAM[pc++]
    pc=br sel=if_not_overflow_8;             // PC = (not_overflow8?MDR:PC)
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
jump8n:
    mdr=br mdr=low  ram=bw ram=p pc=p1;      // MDR[7:0] <-- RAM[pc++]
    mdr=br mdr=high ram=bw ram=p pc=p1;      // MDR[15:7] <-- RAM[pc++]
    pc=br sel=if_negative_8;                 // PC = (negative8?MDR:PC)
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
jump8nn:
    mdr=br mdr=low  ram=bw ram=p pc=p1;      // MDR[7:0] <-- RAM[pc++]
    mdr=br mdr=high ram=bw ram=p pc=p1;      // MDR[15:7] <-- RAM[pc++]
    pc=br sel=if_not_negative_8;             // PC = (not_negative8?MDR:PC)
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
jump16c:
    mdr=br mdr=low  ram=bw ram=p pc=p1;      // MDR[7:0] <-- RAM[pc++]
    mdr=br mdr=high ram=bw ram=p pc=p1;      // MDR[15:7] <-- RAM[pc++]
    pc=br sel=if_carry_16;                   // PC = (carry16?MDR:PC)
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
jump16nc:
    mdr=br mdr=low  ram=bw ram=p pc=p1;      // MDR[7:0] <-- RAM[pc++]
    mdr=br mdr=high ram=bw ram=p pc=p1;      // MDR[15:7] <-- RAM[pc++]
    pc=br sel=if_not_carry_16;               // PC = (not_carry16?MDR:PC)
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
jump16z:
    mdr=br mdr=low  ram=bw ram=p pc=p1;      // MDR[7:0] <-- RAM[pc++]
    mdr=br mdr=high ram=bw ram=p pc=p1;      // MDR[15:7] <-- RAM[pc++]
```

```
    pc=br sel=if_zero_16;                    // PC = (zero16?MDR:PC)
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
jump16nz:
    mdr=br mdr=low  ram=bw ram=p pc=p1;      // MDR[7:0] <-- RAM[pc++]
    mdr=br mdr=high ram=bw ram=p pc=p1;      // MDR[15:7] <-- RAM[pc++]
    pc=br sel=if_not_zero_16;                // PC = (not_zero16?MDR:PC)
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
jump16o:
    mdr=br mdr=low  ram=bw ram=p pc=p1;      // MDR[7:0] <-- RAM[pc++]
    mdr=br mdr=high ram=bw ram=p pc=p1;      // MDR[15:7] <-- RAM[pc++]
    pc=br sel=if_overflow_16;                // PC = (overflow16?MDR:PC)
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
jump16no:
    mdr=br mdr=low  ram=bw ram=p pc=p1;      // MDR[7:0] <-- RAM[pc++]
    mdr=br mdr=high ram=bw ram=p pc=p1;      // MDR[15:7] <-- RAM[pc++]
    pc=br sel=if_not_overflow_16;            // PC = (not_overflow16?MDR:PC)
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
jump16n:
    mdr=br mdr=low  ram=bw ram=p pc=p1;      // MDR[7:0] <-- RAM[pc++]
    mdr=br mdr=high ram=bw ram=p pc=p1;      // MDR[15:7] <-- RAM[pc++]
    pc=br sel=if_negative_16;                // PC = (negative16?MDR:PC)
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
jump16nn:
    mdr=br mdr=low  ram=bw ram=p pc=p1;      // MDR[7:0] <-- RAM[pc++]
    mdr=br mdr=high ram=bw ram=p pc=p1;      // MDR[15:7] <-- RAM[pc++]
    pc=br sel=if_not_negative_16;            // PC = (not_negative16?MDR:PC)
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
call:
    i=br i=low  ram=bw ram=p pc=p1 sp=m1;    // I[7:0] <-- RAM[pc++], SP--
    i=br i=high ram=bw ram=p pc=p1 sp=m1;    // I[15:7] <-- RAM[pc++], SP--
    ram=br ram=s pc=bw pc=low  sp=p1;        // RAM[sp++] <- PC[7:0], SP++
    ram=br ram=s pc=bw pc=high sp=m1;        // RAM[sp--] <- PC[15:8], SP--
    pc=br i=bw;                              // PC <- I;
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
call_i:
    sp=m2;                                   // SP <-- (SP - 2)
    ram=br ram=s pc=bw pc=low  sp=p1;        // RAM[sp++] <- PC[7:0], SP++
    ram=br ram=s pc=bw pc=high sp=m1;        // RAM[sp--] <- PC[15:8], SP--
    pc=br i=bw;                              // PC <- I;
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
call_j:
    sp=m2;                                   // SP <-- (SP - 2)
```

```
    ram=br ram=s pc=bw pc=low  sp=p1;        // RAM[sp++] <- PC[7:0], SP++
    ram=br ram=s pc=bw pc=high sp=m1;        // RAM[sp--] <- PC[15:8], SP--
    pc=br j=bw;                              // PC <- J;
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
return:
    pc=br pc=low  ram=bw ram=s sp=p1;        // PC[7:0] <- RAM[sp++];
    pc=br pc=high ram=bw ram=s sp=p1;        // PC[15:8] <- RAM[sp++];
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
push8_i:
    sp=m1;                                   // SP--;
    ram=br ram=s i=bw i=low;                 // RAM[sp] <- I[7:0];
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
push8_j:
    sp=m1;                                   // SP--;
    ram=br ram=s j=bw j=low;                 // RAM[sp] <- J[7:0];
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
push8_a:
    sp=m1;                                   // SP--;
    ram=br ram=s a=bw a=low;                 // RAM[sp] <- A[7:0];
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
push8_b:
    sp=m1;                                   // SP--;
    ram=br ram=s b=bw b=low;                 // RAM[sp] <- B[7:0];
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
push8_bp:
    sp=m1;                                   // SP--;
    ram=br ram=s bp=bw bp=low;               // RAM[sp] <- BP[7:0];
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
push8_mdr:
    sp=m1;                                   // SP--;
    ram=br ram=s mdr=bw mdr=low;             // RAM[sp] <- MDR[7:0];
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
push8_fl:
    sp=m1;                                   // SP--;
    ram=br ram=s fl=bw fl=low;               // RAM[sp] <- FL[7:0];
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
pop8_i:
    i=br i=low  ram=bw ram=s sp=p1           // I[7:0] <-- RAM[sp++];
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
pop8_j:
    j=br j=low  ram=bw ram=s sp=p1;          // J[7:0] <-- RAM[sp++];
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
```

```
pop8_a:
    a=br a=low  ram=bw ram=s sp=p1;          // A[7:0] <-- RAM[sp++];
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
pop8_b:
    b=br b=low  ram=bw ram=s sp=p1;          // B[7:0] <-- RAM[sp++];
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
pop8_bp:
    bp=br bp=low  ram=bw ram=s sp=p1;        // BP[7:0] <-- RAM[sp++];
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
pop8_mdr:
    mdr=br mdr=low  ram=bw ram=s sp=p1;      // MDR[7:0] <-- RAM[sp++];
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
pop8_fl:
    fl=br fl=low  ram=bw ram=s sp=p1;        // FL[7:0] <-- RAM[sp++];
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
push16_i:
    sp=m2;                                   // SP <-- (SP - 2)
    ram=br ram=s i=bw i=low  sp=p1;          // RAM[sp++] <- I[7:0];
    ram=br ram=s i=bw i=high sp=m1;          // RAM[sp--] <- I[15:8];
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
push16_j:
    sp=m2;                                   // SP <-- (SP - 2)
    ram=br ram=s j=bw j=low  sp=p1;          // RAM[sp++] <- J[7:0];
    ram=br ram=s j=bw j=high sp=m1;          // RAM[sp--] <- J[15:8];
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
push16_a:
    sp=m2;                                   // SP <-- (SP - 2)
    ram=br ram=s a=bw a=low  sp=p1;          // RAM[sp++] <- A[7:0];
    ram=br ram=s a=bw a=high sp=m1;          // RAM[sp--] <- A[15:8];
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
push16_b:
    sp=m2;                                   // SP <-- (SP - 2)
    ram=br ram=s b=bw b=low  sp=p1;          // RAM[sp++] <- B[7:0];
    ram=br ram=s b=bw b=high sp=m1;          // RAM[sp--] <- B[15:8];
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
push16_bp:
    sp=m2;                                   // SP <-- (SP - 2)
    ram=br ram=s bp=bw bp=low  sp=p1;        // RAM[sp++] <- BP[7:0];
    ram=br ram=s bp=bw bp=high sp=m1;        // RAM[sp--] <- BP[15:8];
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
push16_mdr:
    sp=m2;                                   // SP <-- (SP - 2)
```

```
    ram=br ram=s mdr=bw mdr=low  sp=p1;      // RAM[sp++] <- MDR[7:0];
    ram=br ram=s mdr=bw mdr=high sp=m1;      // RAM[sp--] <- MDR[15:8];
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
push16_fl:
    sp=m2;                                   // SP <-- (SP - 2)
    ram=br ram=s fl=bw fl=low  sp=p1;        // RAM[sp++] <- FL[7:0];
    ram=br ram=s fl=bw fl=high sp=m1;        // RAM[sp--] <- FL[15:8];
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
pop16_i:
    i=br i=low  ram=bw ram=s sp=p1;          // I[7:0] <-- RAM[sp++];
    i=br i=high ram=bw ram=s sp=p1;          // I[15:0] <-- RAM[sp++];
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
pop16_j:
    j=br j=low  ram=bw ram=s sp=p1;          // J[7:0] <-- RAM[sp++];
    j=br j=high ram=bw ram=s sp=p1;          // J[15:0] <-- RAM[sp++];
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
pop16_a:
    a=br a=low  ram=bw ram=s sp=p1;          // A[7:0] <-- RAM[sp++];
    a=br a=high ram=bw ram=s sp=p1;          // A[15:0] <-- RAM[sp++];
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
pop16_b:
    b=br b=low  ram=bw ram=s sp=p1;          // B[7:0] <-- RAM[sp++];
    b=br b=high ram=bw ram=s sp=p1;          // B[15:0] <-- RAM[sp++];
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
pop16_bp:
    bp=br bp=low  ram=bw ram=s sp=p1;        // BP[7:0] <-- RAM[sp++];
    bp=br bp=high ram=bw ram=s sp=p1;        // BP[15:0] <-- RAM[sp++];
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
pop16_mdr:
    mdr=br mdr=low  ram=bw ram=s sp=p1;      // MDR[7:0] <-- RAM[sp++];
    mdr=br mdr=high ram=bw ram=s sp=p1;      // MDR[15:0] <-- RAM[sp++];
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
pop16_fl:
    fl=br fl=low  ram=bw ram=s sp=p1;        // FL[7:0] <-- RAM[sp++];
    fl=br fl=high ram=bw ram=s sp=p1;        // FL[15:0] <-- RAM[sp++];
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
c8to16u:
    a=br alu=bw alu=a alu=rank8 fl=br fl=aux        // A[15:0] <- A[7:0],
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;    // fetch;
c8to16s:
    a=br alu=bw alu=a alu=rank8 alu=sign fl=br fl=aux   // A[15:0] <- A[7:0],
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;    // fetch;
```

```
equal:
    a=br alu=bw alu=a alu=rank16 fl=br fl=aux       // A <- A, fetch
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
not:
    a=br alu=bw alu=not alu=rank16 fl=br fl=aux     // A <- NOT A, fetch
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
and:
    a=br alu=bw alu=and alu=rank16 fl=br fl=aux     // A <- A AND B, fetch
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
nand:
    a=br alu=bw alu=nand alu=rank16 fl=br fl=aux    // A <- A NAND B, fetch
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
or:
    a=br alu=bw alu=or alu=rank16 fl=br fl=aux      // A <- A OR B, fetch
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
nor:
    a=br alu=bw alu=nor alu=rank16 fl=br fl=aux     // A <- A NOR B, fetch
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
xor:
    a=br alu=bw alu=xor alu=rank16 fl=br fl=aux     // A <- A XOR B, fetch
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
nxor:
    a=br alu=bw alu=nxor alu=rank16 fl=br fl=aux    // A <- A NXOR B, fetch
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
add:
    a=br alu=bw alu=add alu=rank16 fl=br fl=aux     // A <- A+B, fetch
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
sub:
    a=br alu=bw alu=sub alu=rank16 fl=br fl=aux     // A <- A-B, fetch
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
addc8:
    a=br alu=bw alu=addc alu=rank8 fl=br fl=aux     // A <- A+B+carry, fetch
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
subb8:
    a=br alu=bw alu=subb alu=rank8 fl=br fl=aux     // A <- A-B-borrow, fetch
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
addc16:
    a=br alu=bw alu=addc alu=rank16 fl=br fl=aux    // A <- A+B+carry, fetch
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
subb16:
    a=br alu=bw alu=subb alu=rank16 fl=br fl=aux    // A <- A-B-borrow, fetch
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
```

```
lshl8:
    a=br alu=bw alu=lshl alu=rank8 fl=br fl=aux    // A <- A <<, fetch
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
lshr8:
    a=br alu=bw alu=lshr alu=rank8 fl=br fl=aux    // A <- A >>, fetch
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
ashl8:
    a=br alu=bw alu=ashl alu=rank8 alu=sign fl=br fl=aux    // A <- A*2,
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;    // fetch:
ashr8:
    a=br alu=bw alu=ashr alu=rank8 alu=sign fl=br fl=aux   // A <- A/2, fetch
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
rotcl8:
    a=br alu=bw alu=rotcl alu=rank8 fl=br fl=aux   // A <- rotcl(A), fetch
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
rotcr8:
    a=br alu=bw alu=rotcr alu=rank8 fl=br fl=aux   // A <- rotcr(A), fetch
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
rotl8:
    a=br alu=bw alu=rotl alu=rank8 fl=br fl=aux    // A <- rotl(A), fetch
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
rotr8:
    a=br alu=bw alu=rotr alu=rank8 fl=br fl=aux    // A <- rotr(A), fetch
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
lshl16:
    a=br alu=bw alu=lshl alu=rank16 fl=br fl=aux   // A <- A <<, fetch
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
lshr16:
    a=br alu=bw alu=lshr alu=rank16 fl=br fl=aux   // A <- A >>, fetch
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
ashl16:
    a=br alu=bw alu=ashl alu=rank16 alu=sign fl=br fl=aux   // A <- A*2,
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;    // fetch;
ashr16:
    a=br alu=bw alu=ashr alu=rank16 alu=sign fl=br fl=aux   // A <- A/2,
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;    // fetch;
rotcl16:
    a=br alu=bw alu=rotcl alu=rank16 fl=br fl=aux  // A <- rotcl(A), fetch
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
rotcr16:
    a=br alu=bw alu=rotcr alu=rank16 fl=br fl=aux  // A <- rotcr(A), fetch
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
```

```
rotl16:
    a=br alu=bw alu=rotl alu=rank16 fl=br fl=aux   // A <- rotl(A), fetch
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
rotr16:
    a=br alu=bw alu=rotr alu=rank16 fl=br fl=aux   // A <- rotr(A), fetch
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
in:
    ioa=br ram=bw ram=p pc=p1;                    // IOA <- RAM[pc++];
    ioc=req;                                      // I/O request;
    ctrl=nop;                                     // non fa alcunché
    a=br ioc=bw                                   // A <- I/O, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
out:
    ioa=br ram=bw ram=p pc=p1;                    // IOA <- RAM[pc++];
    ioc=br a=bw;                                  // I/O <- A
    ioc=req                                       // I/O request, fetch;
      ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
is_ack:
    ioa=br ram=bw ram=p pc=p1;                    // IOA <- RAM[pc++];
    mdr=br mdr=low  ram=bw ram=p pc=p1;           // MDR[7:0] <- RAM[pc++];
    mdr=br mdr=high ram=bw ram=p pc=p1;           // MDR[15:8] <- RAM[pc++];
    a=br ioc=bw ioc=isack;                        // A <- I/O is ack;
    a=br alu=a alu=rank8 alu=sign fl=br fl=aux;   // A[15:0] <- A[7:0];
    pc=br sel=if_not_zero_8;                      // PC = (not_zero8?MDR:PC);
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
int:
    // push FL
    sp=m2;                                        // SP <-- (SP - 2)
    ram=br ram=s fl=bw fl=low  sp=p1;             // RAM[sp++] <- FL[7:0];
    ram=br ram=s fl=bw fl=high sp=m1;             // RAM[sp--] <- FL[15:8];
    // reset interrupt enable flag, pc++
    // (PC viene incrementato per saltare l'argomento, prima di
    // salvare il suo valore nella pila).
    fl=br fl=aux alu=cleari pc=p1;
    // push PC
    sp=m2;                                        // SP <-- (SP - 2)
    ram=br ram=s pc=bw pc=low  sp=p1;             // RAM[sp++] <- PC[7:0];
    ram=br ram=s pc=bw pc=high sp=m1;             // RAM[sp--] <- PC[15:8];
    // push I
    sp=m2;                                        // SP <-- (SP - 2)
    ram=br ram=s i=bw i=low  sp=p1;               // RAM[sp++] <- I[7:0];
    ram=br ram=s i=bw i=high sp=m1;               // RAM[sp--] <- I[15:8];
```

```
    // riporta PC al valore corretto per individuare
    // l'argomento che contiene il numero di interruzione.
    pc=m1;
    //
    i=br ivt=bw ivt=intb ram=bw ram=aux ram=p pc=p1; // I <- IVT <- RAM[pc++];
    pc=br pc=low  ram=bw ram=i i=p1;         // PC[7:0]  <-- RAM[i++]
    pc=br pc=high ram=bw ram=i i=m1;         // PC[15:7] <-- RAM[i--]
    // pop I
    i=br i=low  ram=bw ram=s sp=p1;          // I[7:0] <-- RAM[sp++];
    i=br i=high ram=bw ram=s sp=p1;          // I[15:0] <-- RAM[sp++];
    // fetch
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
iret:
    // pop PC
    pc=br pc=low  ram=bw ram=s sp=p1;        // PC[7:0] <- RAM[sp++];
    pc=br pc=high ram=bw ram=s sp=p1;        // PC[15:8] <- RAM[sp++];
    // pop FL
    fl=br fl=low  ram=bw ram=s sp=p1;        // FL[7:0] <-- RAM[sp++];
    fl=br fl=high ram=bw ram=s sp=p1;        // FL[15:0] <-- RAM[sp++];
    // fetch
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
irq:
    // push FL
    sp=m2;                                   // SP <-- (SP - 2)
    ram=br ram=s fl=bw fl=low  sp=p1;        // RAM[sp++] <- FL[7:0];
    ram=br ram=s fl=bw fl=high sp=m1;        // RAM[sp--] <- FL[15:8];
    // reset interrupt enable flag
    fl=br fl=aux alu=cleari;
    // ripristina il valore corretto di PC:
    // PC è collocato dopo il codice operativo
    // di un'istruzione al posto della quale si sta
    // eseguendo il codice dell'interruzione; pertanto,
    // il valore corretto di PC da salvare è PC-1.
    pc=m1;                                   // PC--;
    // push PC
    sp=m2;                                   // SP <-- (SP - 2)
    ram=br ram=s pc=bw pc=low  sp=p1;        // RAM[sp++] <- PC[7:0];
    ram=br ram=s pc=bw pc=high sp=m1;        // RAM[sp--] <- PC[15:8];
    // push I
    sp=m2;                                   // SP <-- (SP - 2)
    ram=br ram=s i=bw i=low  sp=p1;          // RAM[sp++] <- I[7:0];
    ram=br ram=s i=bw i=high sp=m1;          // RAM[sp--] <- I[15:8];
```

```
    //.
    i=br ivt=bw ivt=inta;                    // I <- IVT <- IRQ;
    pc=br pc=low  ram=bw ram=i i=p1;         // PC[7:0]  <-- RAM[i++]
    pc=br pc=high ram=bw ram=i i=m1;         // PC[15:7] <-- RAM[i--]
    // pop I
    i=br i=low  ram=bw ram=s sp=p1;          // I[7:0] <-- RAM[sp++];
    i=br i=high ram=bw ram=s sp=p1;          // I[15:0] <-- RAM[sp++];
    //
    irq=done;
    // fetch
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
ivtl:
    i=br i=low  ram=bw ram=p pc=p1;          // I[7:0] <- RAM[pc++];
    i=br i=high ram=bw ram=p pc=p1;          // I[15:8] <- RAM[pc++];
    ivt=br i=bw;                             // IVT <- MDR;
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
imrl:
    irq=br ram=bw ram=p pc=p1;               // IRQ <- RAM[pc++];
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;  // fetch
cleari:
    fl=br fl=aux alu=cleari irq=done
        ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
seti:
    fl=br fl=aux alu=seti irq=done
        ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
clearc:
    fl=br fl=aux alu=clearc
        ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
setc:
    fl=br fl=aux alu=setc
        ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
cmp:
    fl=br fl=aux alu=sub                     // FL(A - B);
        ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
test:
    fl=br fl=aux alu=and                     // FL(A AND B);
        ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
inc_i:
    i=p1                                     // I++, fetch;
        ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
inc_j:
    j=p1                                     // J++, fetch;
```

```
        ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
inc_a:
    a=p1                                        // A++, fetch;
        ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
inc_b:
    b=p1                                        // B++, fetch;
        ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
inc_bp:
    bp=p1                                       // BP++, fetch;
        ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
inc_sp:
    sp=p1                                       // SP++, fetch;
        ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
inc_mdr:
    mdr=p1                                      // MDR++, fetch;
        ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
inc_fl:
    fl=p1                                       // FL++, fetch;
        ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
dec_i:
    i=m1                                        // I--, fetch;
        ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
dec_j:
    j=m1                                        // J--, fetch;
        ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
dec_a:
    a=m1                                        // A--, fetch;
        ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
dec_b:
    b=m1                                        // B--, fetch;
        ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
dec_bp:
    bp=m1                                       // BP--, fetch;
        ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
dec_sp:
    sp=m1                                       // SP--, fetch;
        ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
dec_mdr:
    mdr=m1                                      // MDR--, fetch;
        ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
dec_fl:
    fl=m1                                       // FL--, fetch;
```

```
        ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
//
stop:
    ctrl=stop;                                      // stop clock
    // if resumed, fetch:
    ir=aux ir=br ram=aux ram=bw ram=p pc=p1 ctrl=load;
end
```

Gestione delle interruzioni

Si distinguono tre tipi di interruzioni: quelle generate internamente dalla CPU, quelle hardware (IRQ) e quelle software. Le interruzioni interne di CPU vanno da INT0 a INT3, ma attualmente è previsto solo INT0 che corrisponde all'individuazione di un codice operativo non valido. Le interruzioni hardware vanno da INT4 a INT7 e corrispondono rispettivamente all'intervallo da IRQ0 a IRQ3. Le interruzioni software vanno da INT8 a INT15. La tabella IVT (*interrupt vector table*) va predisposta nel macrocodice e va inizializzato il registro ***IVT*** con l'indirizzo della sua collocazione, come nell'esempio seguente:

```
begin macrocode @ 0
        jump #start
        nop
interrupt_vector_table:
        .short  0x0025  // CPU # op_code_error
        .short  0x0024  // CPU # default_interrupt_routine
        .short  0x0024  // CPU # default_interrupt_routine
        .short  0x0024  // CPU # default_interrupt_routine
        .short  0x0024  // IRQ # default_interrupt_routine
        .short  0x0024  // IRQ # default_interrupt_routine
        .short  0x0024  // IRQ # default_interrupt_routine
        .short  0x0024  // IRQ # default_interrupt_routine
        .short  0x0024  // software # default_interrupt_routine
```

```
        .short  0x0024  // software # default_interrupt_routine
        .short  0x0024  // software # default_interrupt_routine
        .short  0x0024  // software # default_interrupt_routine
        .short  0x0024  // software # default_interrupt_routine
        .short  0x0024  // software # default_interrupt_routine
        .short  0x0024  // software # default_interrupt_routine
        .short  0x0024  // software # default_interrupt_routine
default_interrupt_routine:
        iret
op_code_error:
        stop
start:
        load16 #sp_base
        mv %MDR, %SP
        ivtl #interrupt_vector_table
        imrl 0x0F // tutti
        seti
        …
        …
sp_base:
        .short 0x0080
end
```

Eventualmente, la tabella può essere più breve, se non si vogliono utilizzare le interruzioni software.

Il manifestarsi di un'interruzione (di CPU, hardware o software) comporta il salvataggio nella pila dei dati del registro ***FL*** (azzerando subito dopo l'indicatore di abilitazione delle interruzioni hardware) e del registro ***PC***, che l'istruzione **`iret`** va poi a recuperare. Ma mentre la conclusione di un'interruzione avviene sempre nello stesso modo, attraverso la descrizione del codice operativo **`iret`**, l'inizio è diverso nei tre casi. Se si tratta di un'interruzione dovuta a un codice operativo errato, viene eseguito il microcodice a partire

dall'etichetta '**`op_error:`**'; se si tratta di un'interruzione hardware, viene eseguito il microcodice a partire dall'etichetta '**`irq:`**', quando l'unità di controllo starebbe per passare all'esecuzione di un nuovo codice operativo, ma viene invece dirottata a causa dell'interruzione; se si tratta di un'interruzione software, viene eseguito il microcodice a partire dall'etichetta '**`int:`**', corrispondente al codice operativo **`int`**. Le tre situazioni sono diverse:

- L'interruzione dovuta a un codice operativo errato, comporta un errore nel codice contenuto nella memoria RAM e non si può conoscere l'entità di questo danno. Non potendo fare ipotesi, la scelta migliore per la routine associata all'interruzione dovrebbe coincidere con l'arresto della CPU; diversamente, se si accetta di ritornare all'esecuzione del codice si passa a quanto contenuto nella cella di memoria successiva, senza poter sapere se lì si trova eventualmente un argomento (errato) per il codice operativo errato precedente.
- L'interruzione dovuta a un IRQ avviene in modo asincrono rispetto all'attività della CPU e viene servita quando la CPU stessa starebbe invece per acquisire un nuovo codice operativo. In questa condizione, il registro ***PC*** punta già alla posizione di memoria successiva al codice operativo che avrebbe dovuto essere eseguito; pertanto, prima di salvare il registro ***PC*** nella pila dei dati, occorre farlo arretrare di una posizione, in modo che corrisponda alla posizione del codice operativo che deve essere eseguito al termine dell'interruzione.

  Il microcodice che serve un'interruzione hardware ha anche il compito, una volta letto l'indirizzo corrispondente alla cella della tabella IVT corrispondente, di cancellare la richiesta nel modulo

IRQ. Ciò avviene inviando un segnale tramite il bus di controllo, che nel modulo **IRQ** viene recepito come *irq done*. È poi compito della logica del modulo **IRQ** sapere qual è effettivamente il segnale di IRQ da azzerare. Contestualmente, il modulo **IRQ** potrebbe richiedere la gestione di un'altra interruzione, ma temporaneamente tale gestione risulterebbe sospesa, perché l'indicatore di abilitazione delle interruzioni hardware si trova sicuramente a essere disabilitato (ciò avviene subito dopo il salvataggio del registro ***FL*** nella pila dei dati).

- L'interruzione software è più semplice da governare, perché avviene in modo prevedibile, senza interrompere veramente l'attività dell'unità di controllo.

Quando si verifica un'interruzione esiste anche la necessità di saltare correttamente alla routine prevista nella tabella IVT. Il registro ***IVT*** ha due ingressi distinti per ricevere il numero di interruzione da convertire in indirizzo di memoria: uno è collegato al bus ausiliario, a cui è collegato anche il modulo **bus** e il modulo della memoria RAM; l'altro è collegato a modulo **IRQ**. Quando si tratta di un'interruzione interna di CPU, il modulo **IVT** viene pilotato direttamente dall'unità di controllo, attraverso il modulo **bus**; quando si tratta di un'interruzione hardware, il modulo **IVT** viene pilotato dal modulo **IRQ**; quando invece si tratta di un'interruzione software, il modulo **IVT** viene pilotato dalla RAM, dalla quale si preleva il numero dell'interruzione, fornito in qualità di argomento del codice operativo **int**. A questo proposito va anche osservato che con il codice operativo **int** è possibile attivare qualunque tipo di interruzione, anche se non sarebbe di competenza del software.

Orologio: modulo «RTC»

Il modulo **RTC** (*real time clock*) produce un impulso al secondo e si limita a fornirlo attraverso l'interruzione hardware IRQ0. Se nel modulo **IRQ** risulta abilitata questa linea di interruzione, a ogni secondo viene richiesta l'interruzione saltando all'indirizzo contenuto nella quinta posizione della tabella IVT; in pratica, IRQ0 corrisponde a INT4 nella tabella IVT.

Il modulo **RTC** è costruito semplicemente attraverso codice Verilog:

Listato u116.38. Dichiarazione del modulo **RTC**.

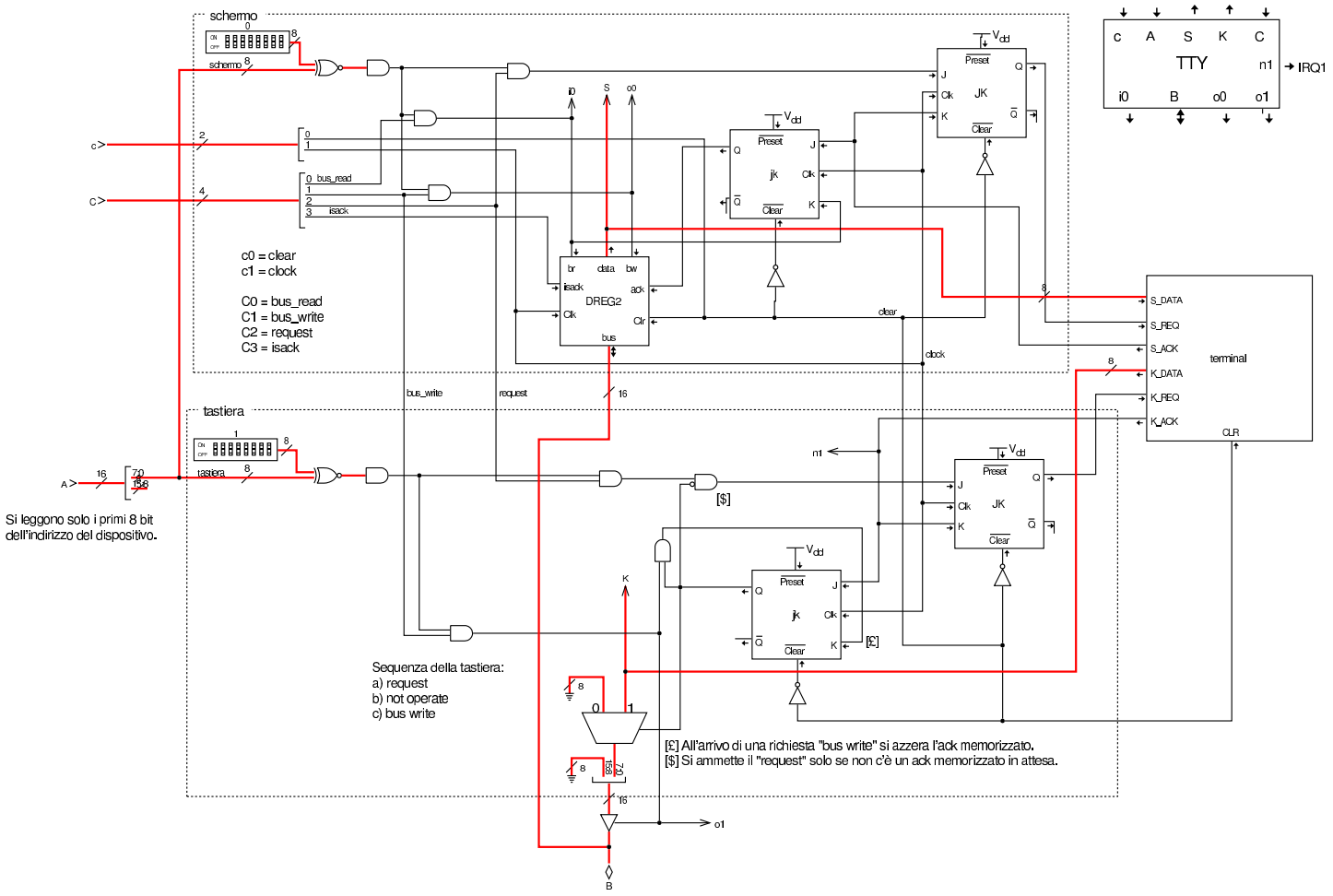
```
module RTC (T);
  output T;
  reg p;
  always
    begin
      p = 0;
      $tkg$wait (500);
      p = 1;
      $tkg$wait (500);
    end
  assign T = p;
endmodule
```

Modulo «TTY»

Il modulo **TTY**, per la gestione del terminale video-tastiera, è quasi identico alla versione precedente della CPU dimostrativa: si aggiunge un'uscita collegata al segnale di conferma (*acknowledge*) della tastiera, per pilotare il segnale IRQ1. In tal modo, quando si preme un tasto sulla tastiera si produce anche un'interruzione IRQ1, la

quale può servire per eseguire il codice necessario a prelevare quanto digitato.

Figura u116.39. Modulo **`TTY`**.

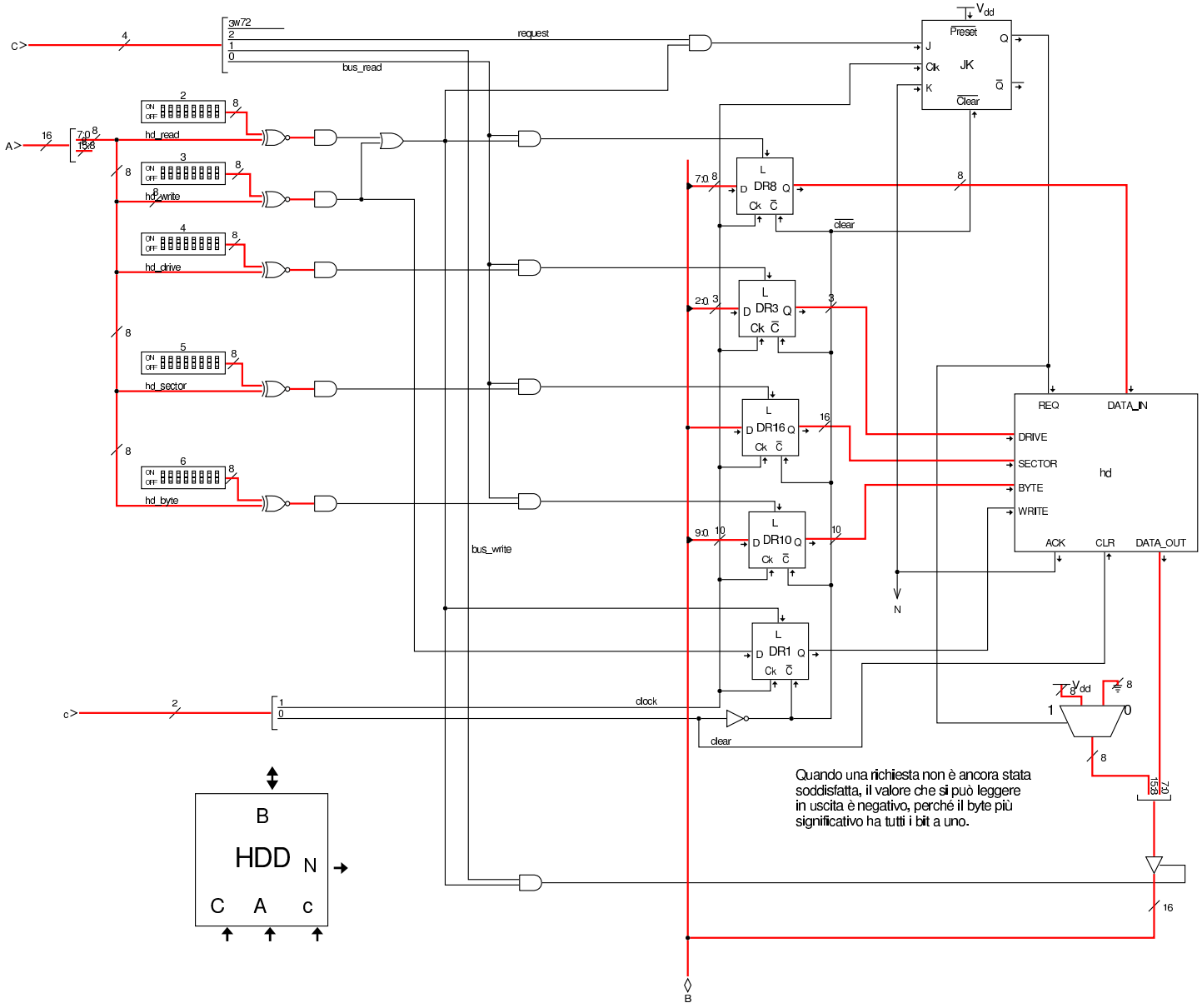


Modulo «HDD»

«

Il modulo **`HDD`** è nuovo rispetto alla versione precedente: si tratta di un'interfaccia che simula un insieme di otto unità di memorizzazione di massa, suddivise a loro volta in settori da 512 byte ognuno. Al dispositivo si accede con indirizzi di I/O differenti, a seconda del tipo di operazione che si deve svolgere.

Figura u116.40. Schema del modulo **HDD**.



Listato u116.41. Codice Verilog che descrive il modulo **hd**.

```
module hd(DRIVE, SECTOR, BYTE, WRITE, DATA_IN, DATA_OUT, REQ, ACK, CLR);
input [2:0] DRIVE;
input WRITE, REQ, CLR;
input [15:0] SECTOR;
input [9:0] BYTE;
input [7:0] DATA_IN;
output [7:0] DATA_OUT;
output ACK;
//
```

```
integer _data_out;
integer _ack;
//
reg [7:0] buffer[0:1023];
reg [8*24-1:0] filename = "hd0_sector_000000000.mem";
//
integer i;
integer sector_8;
integer sector_7;
integer sector_6;
integer sector_5;
integer sector_4;
integer sector_3;
integer sector_2;
integer sector_1;
integer sector_0;
integer x;
//
  initial
    begin
      for (i=0; i<1024; i=i+1)
        begin
          //
          // Initial buffer reset with 00.
          //
          buffer[i] = 8'h00;
        end
      _ack = 0;
      _data_out = 0;
      x = 0;
    end
  //
  always
    begin
      @(posedge CLR)
      _ack = 0;
      _data_out = 0;
      x = 0;
    end
  //
```

```
  //
  //
  always
    begin
      //
      // Start after a positive edge from REQ!.
      //
      @(posedge REQ);
      # 10;
      //
      // Define the sector file name.
      //
      x = SECTOR;
      sector_0 = x%10;
      x = x/10;
      sector_1 = x%10;
      x = x/10;
      sector_2 = x%10;
      x = x/10;
      sector_3 = x%10;
      x = x/10;
      sector_4 = x%10;
      x = x/10;
      sector_5 = x%10;
      x = x/10;
      sector_6 = x%10;
      x = x/10;
      sector_7 = x%10;
      x = x/10;
      sector_8 = x%10;
      //
      // La stringa parte da destra verso sinistra!
      //
      filename[12*8+7:12*8] = sector_8 + 8'd48;
      filename[11*8+7:11*8] = sector_7 + 8'd48;
      filename[10*8+7:10*8] = sector_6 + 8'd48;
      filename[9*8+7:9*8] = sector_5 + 8'd48;
      filename[8*8+7:8*8] = sector_4 + 8'd48;
      filename[7*8+7:7*8] = sector_3 + 8'd48;
      filename[6*8+7:6*8] = sector_2 + 8'd48;
```

```
      filename[5*8+7:5*8] = sector_1 + 8'd48;
      filename[4*8+7:4*8] = sector_0 + 8'd48;
      //
      filename[21*8+7:21*8] = DRIVE + 8'd48;
      //
      if (WRITE)
        begin
          //
          // Put data inside the buffer.
          //
          buffer[BYTE] = DATA_IN;
          //
          // Save the buffer to disk.
          // Please remember that $writememh() must be enabled inside
          // Tkgate configuration!
          //
          $writememh(filename, buffer);
          //
          // Return the same data read.
          //
          _data_out = buffer[BYTE];
        end
      else
        begin
          //
          // Get data from disk to the buffer.
          //
          $readmemh(filename, buffer);
          //
          // Return the data required.
          //
          _data_out = buffer[BYTE];
        end
      //
      // Acknowledge.
      //
      _ack = 1;
      //
      // Wait the end of request (the negative edge)
      // before restarting the loop.
```

```
        //
        @(negedge REQ);
        # 10;
        //
        // Now become ready again.
        //
        _ack = 0;
      end
      //
    assign DATA_OUT = _data_out;
    assign ACK = _ack;
    //
endmodule
```

Trattandosi di un modulo nuovo, è necessario descrivere prima il comportamento di **hd**, di cui è appena stato mostrato il sorgente Verilog: gli ingressi ***DRIVE***, ***SECTOR*** e ***BYTE*** servono a individuare in modo univoco un certo byte, appartenente a un certo settore di una certa unità di memorizzazione. In pratica, ogni unità di memorizzazione virtuale è divisa in settori, dal primo, corrispondente a zero, all'ultimo, corrispondente a 65536. Dal momento che ogni settore è da 512 byte, queste unità di memorizzazione virtuali hanno una capacità massima di 32 Mibyte.

L'ingresso ***WRITE*** consente di selezionare un accesso in scrittura al dispositivo di memorizzazione, altrimenti si intende un accesso in lettura. L'accesso all'unità avviene un byte alla volta e si deve utilizzare l'uscita ***DATA_OUT*** per la lettura, oppure l'ingresso ***DATA_IN*** per la scrittura. Gli ingressi e le uscite ***REQ***, ***ACK*** e ***CLR*** funzionano in modo prevedibile, conformemente a quanto già visto a proposito del dispositivo del terminale (tastiera e schermo).

Per poter usare il dispositivo **HDD**, è necessario fornire inizialmente le coordinate del byte a cui si è interessati, scrivendo nelle porte di

I/O 4, 5 e 6, rispettivamente per l'unità di memorizzazione, il settore e il byte. Quindi si può chiedere un'operazione di lettura (indirizzo di I/O 2) o di scrittura (indirizzo di I/O 3). Quando un'operazione di lettura o scrittura è stata completata, il segnale di conferma (*acknowledge*) viene emesso dal modulo **HDD** e diretto al modulo **IRQ**, diventando un segnale IRQ2. Tuttavia si può fare a meno di usare le interruzioni con il modulo **HDD**, perché la lettura è sempre possibile, con la differenza che se il dato ottenuto non è ancora valido, il valore letto è negativo. Allo stesso modo, dopo la scrittura si può verificare che l'operazione sia stata completata attraverso una lettura: se il valore che si ottiene fosse negativo, significherebbe che occorre attendere ancora un po'.

Il modulo **hd** permette di usare i dispositivi di memorizzazione virtuali in modo libero, senza bisogno di creare prima dei file: quando si accede per la prima volta, in scrittura, a un settore che non era mai stato usato prima, viene creato al volo il file che lo rappresenta, nella directory in cui sta lavorando Tkgate. Se invece si legge un settore che non esiste, il dispositivo si limita a produrre il valore nullo. I file che vengono creati corrispondono al modello 'hd***n***_sector_***ssssssss*****.**mem'.

Macrocodice: esempio di uso del terminale con le interruzioni

«

Il codice seguente esegue la lettura della tastiera, attraverso l'interruzione generata dalla stessa, e la rappresentazione del testo digitato attraverso lo schermo. La parte iniziale del codice definisce la collocazione della tabella IVT e del codice associato ai suoi vari elementi.

Listato u116.42. Macrocodice per la gestione del terminale attraverso le interruzioni della tastiera.

```
begin macrocode @ 0
        jump #start
        nop
interrupt_vector_table:
        .short  0x001D  // CPU
        .short  0x001C  // CPU
        .short  0x001C  // CPU
        .short  0x001C  // CPU

        .short  0x001C  // IRQ
        .short  0x001E  // IRQ keyboard
        .short  0x001C  // IRQ
        .short  0x001C  // IRQ

        .short  0x001C  // software
        .short  0x001C  // software
        .short  0x001C  // software
        .short  0x001C  // software
default_interrupt_routine:
        iret
op_code_error:
        stop
keyboard:
        in 1            // Legge dalla tastiera.
        equal
        jump8z #keyboard_end
        out 0           // Altrimenti emette lo stesso
                        //   valore sullo schermo.
        jump #keyboard
keyboard_end:
        iret
start:
```

```
        load16 #data_1
        mv %MDR, %SP
        //
        ivtl #interrupt_vector_table
        imrl 0x0F // tutti
        seti
keyboard_reset:
        in 1            // Legge dalla tastiera.
        equal
        jump8nz #start // Ripete fino a svuotare il buffer
ciclo:
        jump #ciclo
stop:                   // Non raggiunge mai questo punto.
        stop
data_0:
        .short 0
data_1:
        .short 0x0080
end
```

Figura u116.43. Inserimento da tastiera e visualizzazione sullo schermo. Video: *http://www.youtube.com/watch?v=dgIfZHNTedM*