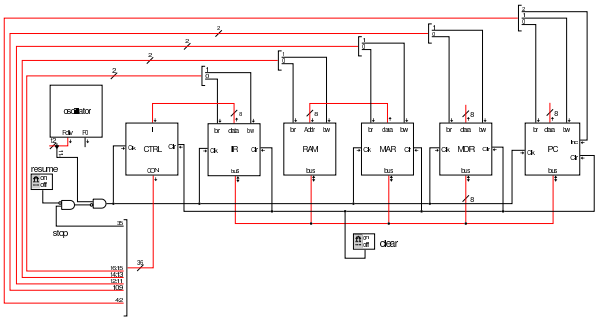
Viene qui introdotto lo sviluppo di una CPU dimostrativa, aggiungendo progressivamente componenti e funzioni, fino ad arrivare a un elaboratore molto semplice. Inizialmente si tratta solo di una CPU con registri a 8 bit, inclusi quelli relativi all'indirizzamento della memoria RAM, la quale è limitata così a un massimo di 256 byte.

Versione A: caricamento ed esecuzione del codice

Nella sua prima versione, la CPU si compone soltanto di registri utili ad accedere alla memoria per leggere il codice operativo da eseguire, come di vede nella figura successiva.

Figura u106.1. Il bus della CPU nella sua prima fase realizzativa.



Il modulo più semplice che si può analizzare è l'oscillatore che serve a produrre il segnale di clock. Si tratta di un oscillatore costruito con una serie di porte logiche invertenti, per creare un ritardo di propagazione sufficiente a produrre un'oscillazione a una frequenza gestibile. Per attivare l'oscillazione si richiede un impulso iniziale che, dopo una breve pausa a zero, si attiva stabilmente. La figura successiva mostra l'oscillatore e l'impulso di avvio necessario per l'attivazione. È importante osservare che la serie di porte invertenti deve essere in numero dispari, come se si trattasse di una sola porta invertente, ma con un lungo ritardo di propagazione. Il risultato viene poi passato a un divisore di frequenza, composto in questo caso da una catena di flip-flop T, sincroni, in modo da non sfasare l'oscillazione a ogni divisione; in uscita si hanno tante linee raggruppate assieme, ognuna delle quali permette di prelevare un'oscillazione a una frequenza differente. Il divisore di frequenza è inizializzato dallo stesso impulso iniziale, il quale parte da uno stato a zero. Nel caso degli esempi viene usata una frequenza molto bassa, corrispondente all'ultimo stadio di divisione.

Figura u106.2. Oscillatore utilizzato per il segnale di clock.

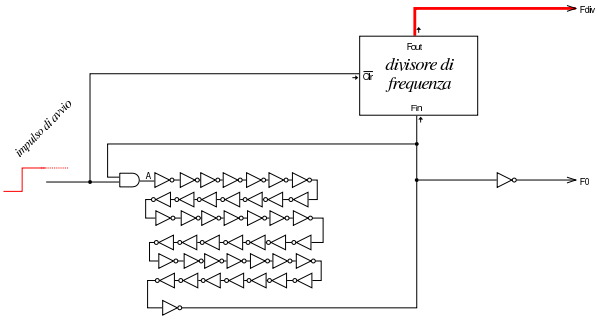
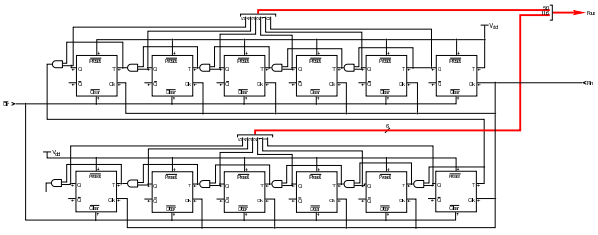


Figura u106.3. Divisore utilizzato nel modulo dell'oscillatore.



L'impulso iniziale viene prodotto da un componente sintetizzato attraverso del codice Verilog, in quanto diversamente servirebbero

componenti elettronici non logici e la loro trattazione esula dallo scopo di questo studio.

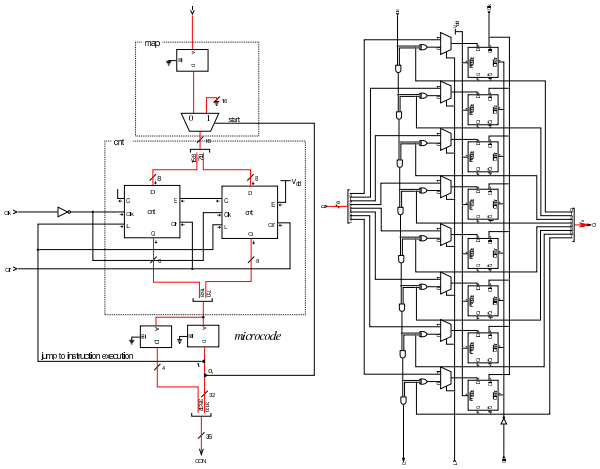
Figura u106.4. Codice Verilog per Tkgate, relativo al modulo di innesco dell'oscillazione: l'uscita è inizialmente a zero e dopo un breve istante passa a uno, rimanendo così stabilmente. Il tempo di attesa iniziale è configurabile attraverso il parametro ***W***.

```
module one_up #(.W(1000)) (Z);
  output Z;
  reg Z;
  initial
    begin
      Z = 1'b0;
      $tkg$wait(W);
      Z = 1'b1;
    end
endmodule
```

L'unità di controllo, contenuta nel modulo **`CTRL`**, è molto simile a quella descritta nella sezione u0.3, con la differenza che l'ingresso è individuato dalla variabile ***I*** a 8 bit (la lettera «I» sta per «istruzione») e che l'uscita ha un rango molto maggiore, costringendo a utilizzare due unità di memoria in parallelo. Il contatore che serve a scandire le istruzioni nel blocco finale di memoria è complessivamente a 16 bit, ma per convenienza, ne sono stati usati due da 8 in cascata.

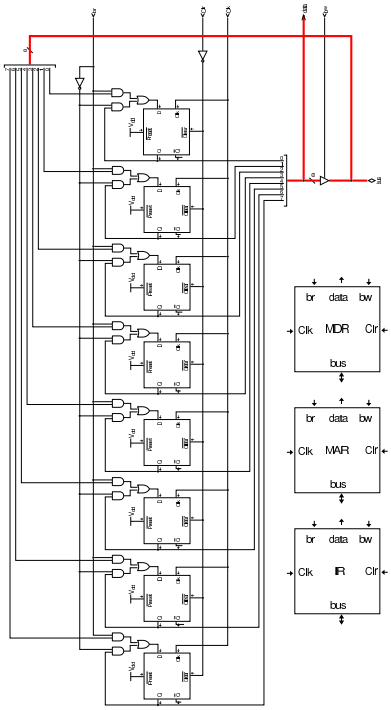
L'ingresso ***I*** dell'unità di controllo è alimentato dal contenuto del registro ***IR*** (*instruction register*).

Figura u106.5. Unità di controllo, evidenziando a destra la struttura del modulo **`cnt`** che rappresenta un contatore, basato su flip-flop D, estensibile per ottenere ranghi maggiori.



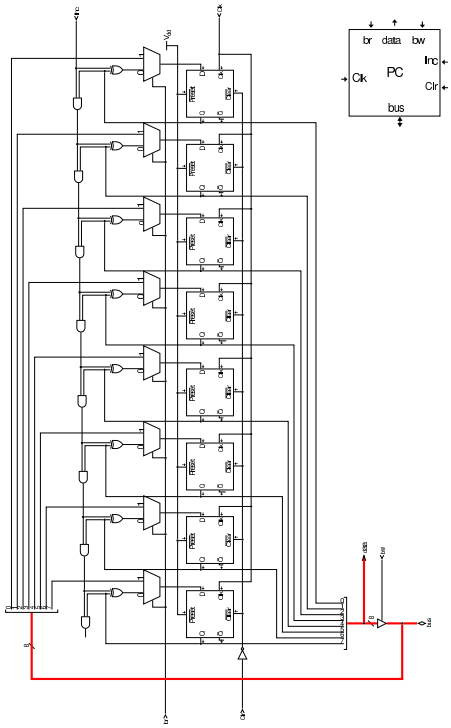
I moduli **`IR`**, **`MAR`** e **`MDR`**, sono registri semplici, costruiti con flip-flop D, connessi al bus attraverso dei buffer a tre stati, dai quali è possibile prelevare copia del valore memorizzato da un'uscita supplementare, denominata ***data***. Il registro ***IR*** (*instruction register*), a cui si è già accennato, ha lo scopo di conservare il codice operativo che l'unità di controllo deve eseguire; il registro ***MAR*** (*memory address register*) ha lo scopo di conservare l'indirizzo di memoria a cui si vuole accedere; il registro ***MDR*** (*memory data register*) serve ad accumulare quanto viene letto dalla memoria per qualche motivo o ciò che vi deve essere scritto.

Figura u106.6. Registri ***IR***, ***MAR*** e ***MDR***.



Il modulo **`PC`** è un registro simile agli altri, con la differenza che può incrementare il valore che contiene quando è attivo l'ingresso ***Inc***. Il registro ***PC*** (*program counter*) ha lo scopo di contenere l'indirizzo di memoria del codice successivo da eseguire.

Figura u106.7. Registro contatore ***PC***.

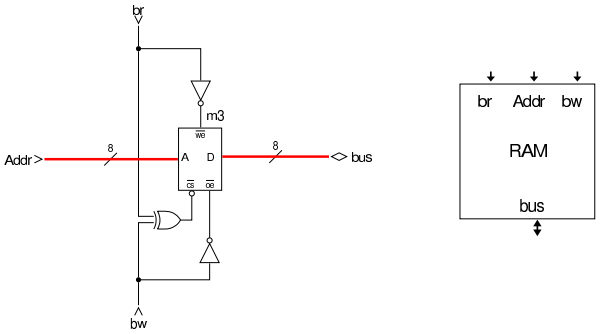


Il modulo **`RAM`** è sostanzialmente differente dagli altri, in quanto racchiude la memoria RAM usata dalla CPU. A tale memoria si accede attraverso l'indirizzo fornito tramite l'ingresso ***Addr***, a 8 bit, e anche il contenuto della memoria è organizzato a celle da 8 bit. Il modulo condivide con gli altri gli ingressi di controllo dell'accesso

al bus; tuttavia, quando il modulo riceve l'indirizzo ed è abilitata la lettura dal bus, il valore contenuto in memoria viene aggiornato subito (salvo il ritardo di propagazione), senza attendere l'impulso di clock.

Il modulo **RAM** riceve l'indirizzo dal registro ***MAR*** (*memory address register*), il quale è così dedicato a contenere e conservare l'indirizzo di memoria a cui si vuole accedere.

Figura u106.8. Modulo **RAM**. La rete logica che controlla gli ingressi ***br*** e ***bw***, serve a impedire che si possa mettere in pratica la lettura e scrittura simultanea del bus.



La prima cosa di cui si deve occupare la struttura appena descritta, consiste nel caricamento di un'istruzione, seguito poi dall'esecuzione della stessa: ciò è noto come ***ciclo di caricamento*** (*fetch*). Nella struttura in questione, il registro ***PC*** contiene l'indirizzo dell'istruzione da eseguire: questo valore deve essere trasferito nel registro ***MAR*** e il registro ***PC*** viene incrementato; dalla memoria RAM si ottiene l'istruzione contenuta nell'indirizzo ***MAR*** che viene copiata nel registro ***IR***. Ciò si può rappresentare sinteticamente come segue:

1. ***MAR*** = ***PC***
2. ***PC***++
3. ***IR*** = ***RAM***[***MAR***]

Le figure successive mostrano proprio questi tre passaggi, evidenziando i valori degli ingressi ***br***, ***bw*** e ***Inc***, attraverso dei LED che diventano rossi nel momento dell'attivazione della linea a cui sono connessi. Le figure mostrano sempre solo il momento in cui il segnale di clock diventa attivo.

Figura u106.9. Prima fase: si richiede al registro ***PC*** di inviare il suo valore al bus e al registro ***MAR*** di leggerlo. Si attua in pratica l'operazione ***MAR***=***PC***.

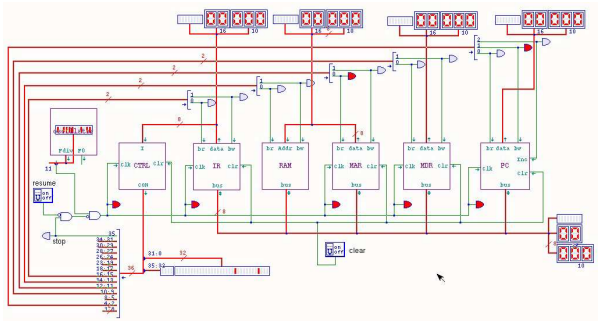


Figura u106.10. Seconda fase: si richiede al registro ***PC*** di incrementarsi di una unità. Si attua in pratica l'operazione ***PC***++.

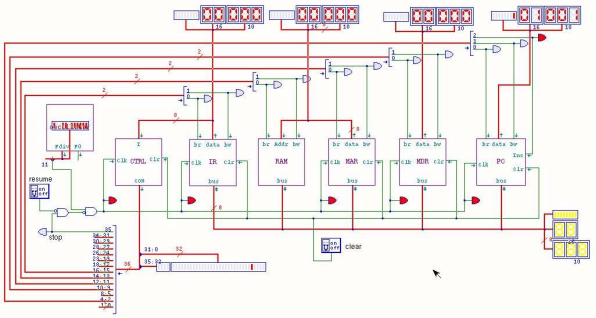
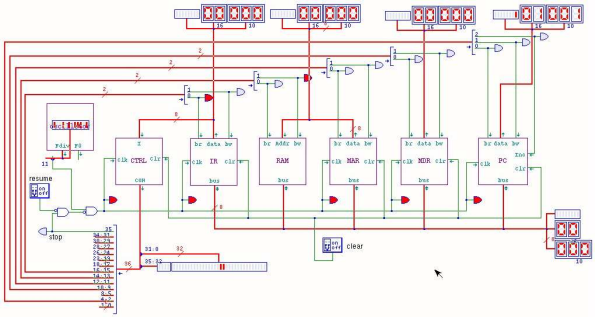


Figura u106.11. Terza fase: si richiede alla RAM di inviare il valore corrispondente all'indirizzo recepito dal registro ***MDR*** al bus e al registro ***IR*** di accumulare questo valore. Si attua in pratica l'operazione ***IR***=***RAM***[***MAR***].



All'interno dell'unità di controllo (il modulo **CTRL**) il tempo è scandito allo stesso modo, a parte il fatto che i contatori ***cnt*** sono pilotati da un segnale di clock invertito, per anticipare l'attivazione delle linee di controllo rispetto all'impulso relativo alla gestione del bus dati. Inizialmente i contatori dell'unità di controllo si trovano a essere azzerati e per questo vanno a ricercare nella memoria sottostante la prima microistruzione, corrispondente alla richiesta di eseguire l'operazione ***MAR***=***PC***. Successivamente il complesso dei due contatori ***cnt*** viene incrementato e ciò fa passare alla seconda microistruzione, corrispondente alla richiesta di incremento del registro ***PC***. Nel terzo istante si ha un incremento ulteriore, facendo emergere la microistruzione ***IR***=***RAM***[***MAR***].

Figura u106.12. Prima fase: i contatori dell'unità di controllo sono azzerati e la microistruzione iniziale corrisponde a ***MAR***=***PC***.

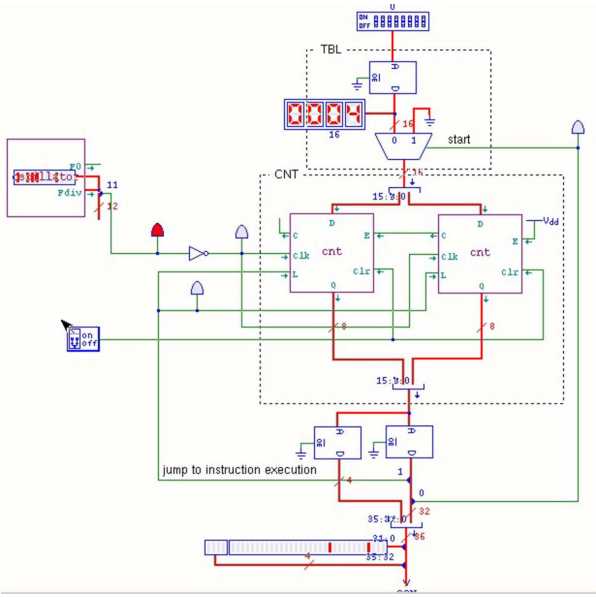


Figura u106.13. Seconda fase: il complesso dei contatori è stato incrementato e la microistruzione prodotta corrisponde a ***PC***++.

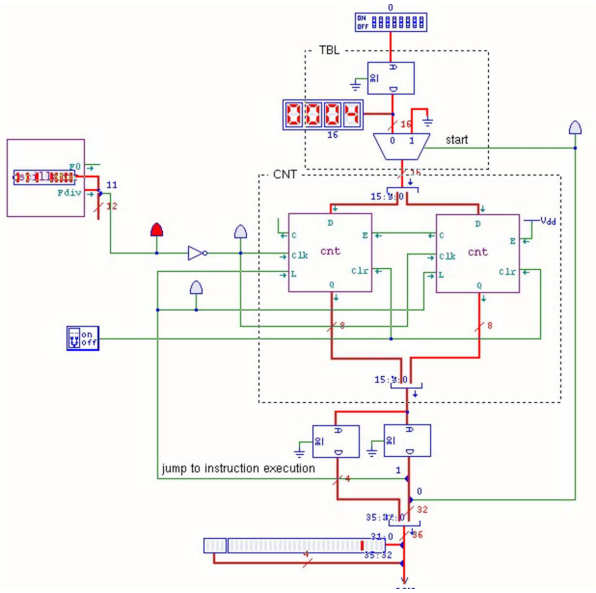
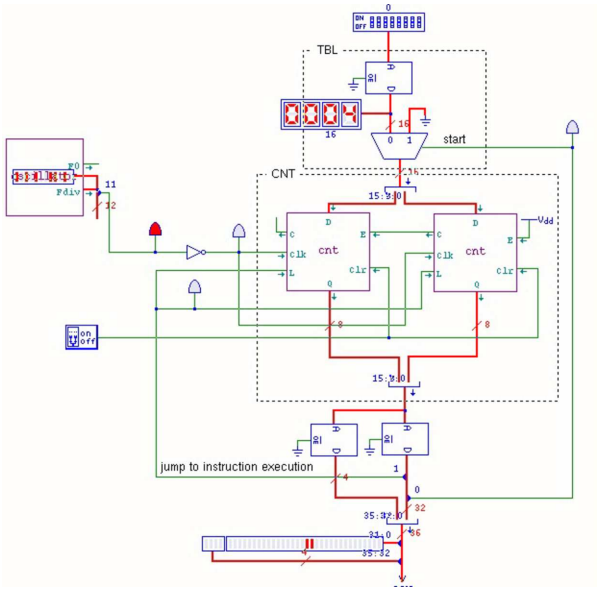


Figura u106.14. Terza fase: il complesso dei contatori è stato incrementato e la microistruzione prodotta corrisponde a ***IR***=***RAM***[***MAR***].



A questo punto, l'unità di controllo dispone dell'istruzione da eseguire nell'ingresso ***I*** ed è pronta per recepirla. Per farlo, la microistruzione successiva richiede al contatore interno di accettare il valore in ingresso. Questo valore corrisponde al contenuto della memoria ***m0***, la quale tratta l'istruzione in ingresso come indirizzo, dal quale produce a sua volta l'indirizzo del microcodice successivo a cui saltare. Negli esempi delle figure, l'istruzione in questione corrisponde al codice operativo 00000000_2, ovvero all'istruzione nulla (`not_operate`).

Figura u106.15. Quarta fase: i contatori dell'unità di controllo sono caricati con il valore proveniente dalla memoria che traduce l'istruzione in indirizzo del microcodice.

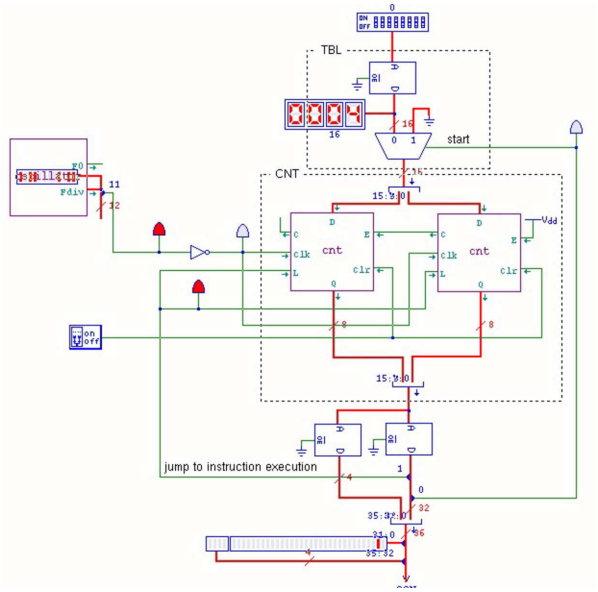
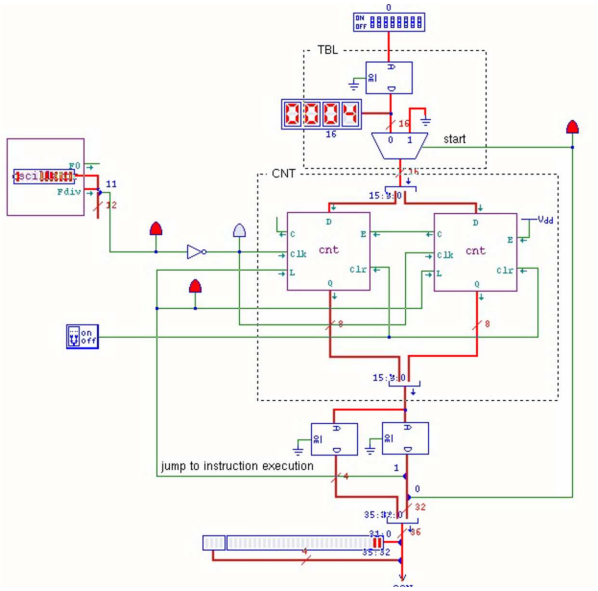


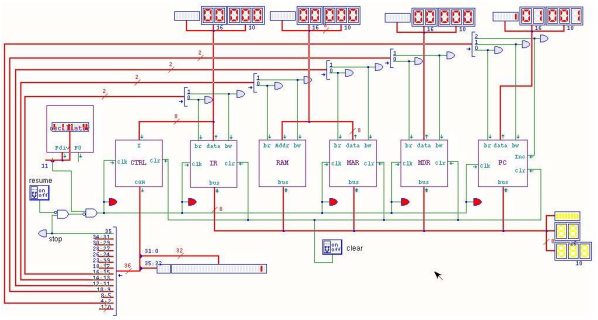
Figura u106.16. Fase conclusiva: i contatori dell'unità di controllo sono stati incrementati e puntano alla microistruzione successiva. Dal momento che l'istruzione originale (**not_operate**) non richiedeva lo svolgimento di alcuna operazione nel bus dati, ci si trova al termine della procedura per tale istruzione, incontrando la microistruzione che richiede al complesso di contatori dell'unità di controllo di azzerarsi. L'azzeramento avviene facendo caricare ai contatori il valore zero, tramite il multiplatore che controlla l'ingresso di tali contatori.



Dopo l'azzeramento dei contatori dell'unità di controllo, si ricomincia dal microcodice iniziale (le prime tre fasi) con il quale si richiede il caricamento di una nuova istruzione.

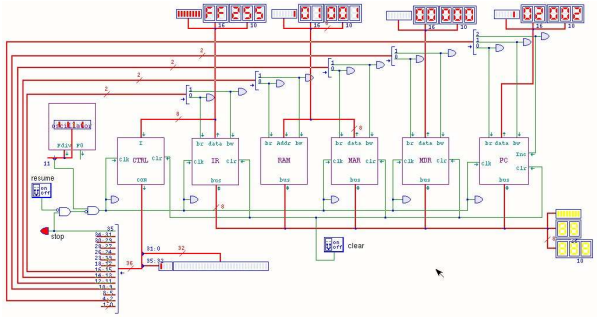
Va osservato che durante la quarta fase (salto al microcodice di esecuzione dell'istruzione richiesta) e durante la fase conclusiva (salto al microcodice iniziale che attua il ciclo di caricamento), nel bus dati non succede nulla.

Figura u106.17. Durante la fase di salto al microcodice di esecuzione dell'istruzione richiesta e durante il salto al microcodice del ciclo di caricamento, nel bus dati non succede nulla.



Per fermare il funzionamento del circuito descritto, esiste l'istruzione **stop** (11111111_2), con la quale viene fermato il segnale di clock. La figura successiva mostra questa situazione.

Figura u106.18. La situazione in cui si trova il bus dati quando viene eseguita l'istruzione **stop**: la linea di controllo CON_{35} si attiva e va a bloccare il segnale di clock. Per far riprendere l'esecuzione da quel punto, superando lo stop, occorrerebbe intervenire nell'interruttore situato vicino al LED che risulta attivo.



Dovrebbero essere disponibili due video, nei quali si dimostra l'esecuzione di due sole istruzioni (macroistruzioni):

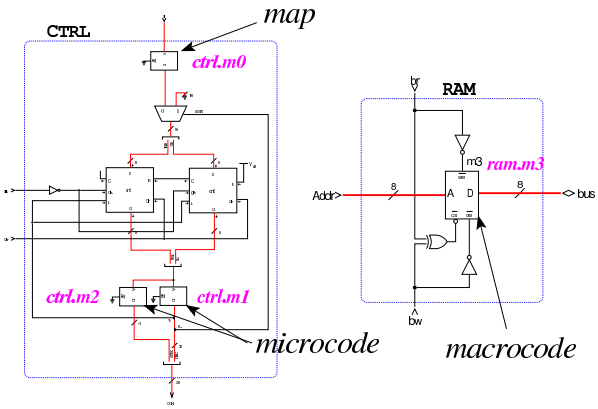
1. **not_operate**
2. **stop**

Il primo video *http://www.youtube.com/watch?v=Z8bTO8WjYYc* mostra ciò che accade nel bus dati; il secondo, invece, mostra l'interno dell'unità di controllo *http://www.youtube.com/watch?v=pPxCQz7IFbM* .

Per descrivere il contenuto delle memorie, incluso quello della memoria RAM, viene usato un file sorgente scritto secondo la sintassi adatta a '**gmac**' di Tkgate 2. Le prime direttive descrivono i banchi di memoria, i quali sono organizzati così: ***ctrl.m0*** corrisponde alla prima memoria in alto dell'unità di controllo; ***ctrl.m1*** e ***ctrl.m2*** sono le due memorie che contengono il microcodice e che si trovano in basso nello schema dell'unità di controllo; ***ram.m3*** è invece la memoria contenuta nel modulo **RAM** del bus dati e ospita il macrocodice che inizialmente si limita solo a **not_operate** e **stop**.

```
map       bank[7:0]   ctrl.m0;
microcode bank[31:0]  ctrl.m1;
microcode bank[35:32] ctrl.m2;
macrocode bank[7:0]   ram.m3;
```

Figura u106.20. Dove si trovano concretamente i banchi di memoria.



Si passa quindi alla descrizione dei campi in cui è suddivisa ogni cella di memoria che rappresenta il microcodice (***ctrl.m1*** e ***ctrl.m2***). Per esempio, il bit meno significativo si chiama ***ctrl_start***, mentre il più significativo si chiama ***stop***. Va osservato che non sono descritti tutti i 36 bit della cella che rappresenta una microistruzione, perché al momento il codice si limita a rappresentare la riduzione della CPU nella sua prima versione.

```
field ctrl_start[0];    // parte dall'indirizzo 0
field ctrl_load[1];     // carica l'indirizzo nel contatore.
field pc_br[2];         // PC <-- bus
field pc_bw[3];         // PC --> bus
field pc_Inc[4];        // PC++
field mdr_br[9];        // MDR <-- bus
field mdr_bw[10];       // MDR --> bus
field mar_br[11];       // MAR <-- bus
field mar_bw[12];       // MAR --> bus
field ram_br[13];       // RAM[mar] <-- bus
field ram_bw[14];       // RAM[mar] --> bus
field ir_br[15];        // IR <-- bus
field ir_bw[16];        // IR --> bus
field stop[35];         // stop clock
```

Vengono poi descritti i tipi di operandi che possono avere le istruzioni (le macroistruzioni). Si prevede di gestire istruzioni senza operandi, oppure con un solo operando di 8 bit. Il significato della sintassi utilizzata per descrivere il tipo ***op_0*** e il tipo ***op_1***, va approfondito, eventualmente, nella documentazione di Tkgate.

```
operands op_0 {
    //
    // [........]
    //
    -  = { };
};
operands op_1 {
    //
    // [........][nnnnnnnn]
    //
    #1 = { +1=#1[7:0]; };
};
```

Si passa poi alla descrizione dei codici operativi; per esempio, si vede che l'istruzione **`not_operate`** corrisponde al codice zero (00000000_2), mentre l'istruzione **`jump`** ha il codice 15 (00001111_2). Va osservato che nel primo caso (**`not_operate`**) non ci sono argomenti, mentre nel secondo si richiede un argomento.

```
op not_operate {
  map not_operate : 0;
  +0[7:0]=0;
  operands op_0;
};
op jump {
  map jump : 15;                    // jump to #nn
  +0[7:0]=15;
  operands op_1;
};
op stop {
  map stop : 255;                   // stop
  +0[7:0]=255;
  operands op_0;
};
```

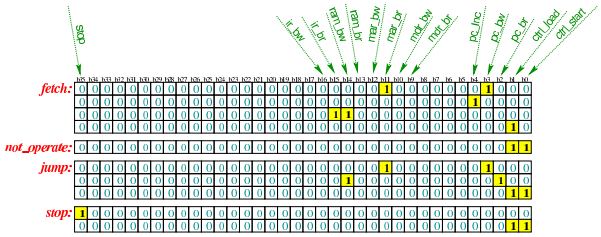
Inizia quindi la definizione del microcodice, il quale viene collocato a partire dall'indirizzo zero della coppia di memorie ***ctrl.m1*** e ***ctrl.m2***. Si può osservare che si inizia proprio dalla descrizione del ciclo di caricamento (*fetch*) che si conclude con il salto alla microistruzione che inizia la procedura che mette in pratica la macroistruzione recepita; inoltre, alla fine della descrizione di ogni macroistruzione (in forma di microcodice), viene richiesto di saltare nuovamente alla prima microistruzione, con la quale si ripete il ciclo di caricamento.

```
begin microcode @ 0
//
fetch:
    mar_br pc_bw;               // MAR = PC
    pc_Inc;                     // PC++
    ir_br ram_bw;               // IR = RAM[MAR]
    ctrl_load;                  // salta alla
                                // microistruzione
                                // corrispondente
//
not_operate:
    ctrl_start ctrl_load;       // salta a «fetch»
//
```



```
jump:
    mar_br pc_bw;               // MAR = PC
    pc_br ram_bw;               // PC <-- RAM[MAR]
    ctrl_start ctrl_load;       // salta a «fetch»
//
stop:
    stop;                       // stop clock.
    // Se il clock fosse
    // riabilitato manualmente:
    ctrl_start ctrl_load;       // salta a «fetch»
//
end
```

Figura u106.25. Corrispondenza con il contenuto della memoria che rappresenta il microcodice (la coppia ***m1*** e ***m2*** dell'unità di controllo).



Infine, inizia il macrocodice, ovvero il codice assemblatore da immettere nella memoria RAM:

```
begin macrocode @ 0
start:
        not_operate
        stop

end
```

Figura u106.27. Macrocodice contenuto nella memoria RAM. Le celle indicate con «xx» hanno un valore indifferente.

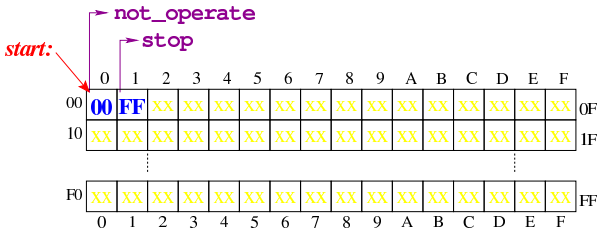


Tabella u106.28. Elenco delle macroistruzioni di questa prima versione della CPU dimostrativa.

Sintassi	Descrizione
`not_operate`	Non fa alcunché, limitandosi a passare all'istruzione successiva.
`jump` ***indirizzo***	Salta all'istruzione che si trova in memoria all'indirizzo specificato.
`stop`	Ferma l'afflusso degli impulsi di clock.

Il file descritto dovrebbe essere disponibile all'indirizzo *allegati/circuiti-logici/scpu-sub-a.gm* . Per compilarlo con '**`gmac`**' di Tkgate 2, si dovrebbe procedere con il comando successivo:

```
$ gmac20 -o scpu-sub-a.mem -m scpu-sub-a.map ↩
↪       scpu-sub-a.gm [Invio]
```

Il file '`scpu-sub-a.mem`' che si ottiene è quello che serve a Tkgate 2 per caricare i contenuti delle memorie previste. Eventualmente, dovrebbe essere disponibile anche il file *allegati/circuiti-logici/scpu-sub-a.v* che contiene la rappresentazione completa di questa prima versione della CPU dimostrativa nel formato di Tkgate 2.

Prima di concludere la descrizione della versione iniziale della CPU dimostrativa, va osservato che esiste una terza istruzione che non è ancora stata usata in un esempio: **`jump`**. Questa si realizza semplicemente con i passaggi seguenti:

1. ***MAR*** = ***PC***

2. ***PC*** = ***RAM***[***MAR***]

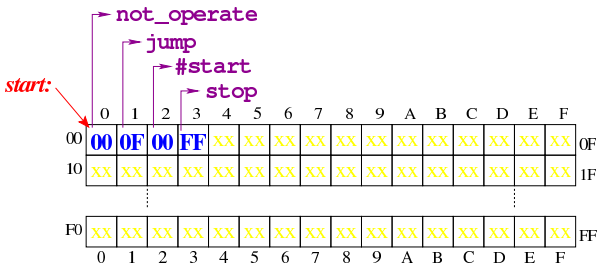
In pratica: nel registro ***MAR*** viene copiato l'indirizzo contenuto nel registro ***PC***, il quale corrisponde all'indirizzo successivo all'istruzione appena letta e in corso di esecuzione (**`jump`**), ma il contenuto della memoria corrispondente a tale indirizzo, viene copiato di nuovo nel registro ***PC*** (senza incrementarlo).

L'istruzione **`jump`** precede un argomento, costituito dall'indirizzo a cui si vuole saltare incondizionatamente; pertanto, tale indirizzo si colloca subito dopo il codice dell'istruzione e viene letto attraverso l'indice del registro ***PC***, come se si trattasse di un'istruzione; poi, però, il contenuto della memoria in corrispondenza di quell'indirizzo, non viene inviato al registro ***IR***, ma viene immesso nuovamente nel registro ***PC***, in maniera tale che la prossima istruzione a essere caricata sia quella a cui si vuole saltare.

A titolo di esempio, il macrocodice (ovvero il codice assemblatore) potrebbe essere modificato come segue:

```
begin macrocode @ 0
start:
        not_operate
        jump #start
        stop
end
```

Figura u106.30. Contenuto della memoria RAM. Le celle indicate con «xx» hanno un valore indifferente.



Durante la compilazione, '**`#start`**' viene rimpiazzato dall'indirizzo corrispondente all'etichetta '**`start:`**' che in pratica è semplicemente zero. Questo piccolo programma si limita a non fare nulla (**`not_operate`**) e a ripeterlo indefinitivamente, tanto che l'istruzione **`stop`** non può mai essere eseguita. Le figure successive mostrano ciò che accade dopo l'esecuzione dell'istruzione **`not_operate`** nel bus dati.

Figura u106.31. La situazione in cui si trova il bus dati quando è stata caricata l'istruzione **`jump`** e il registro ***PC***, puntando all'indirizzo che segue l'istruzione **`jump`**, immette il suo valore nel registro ***MAR***.

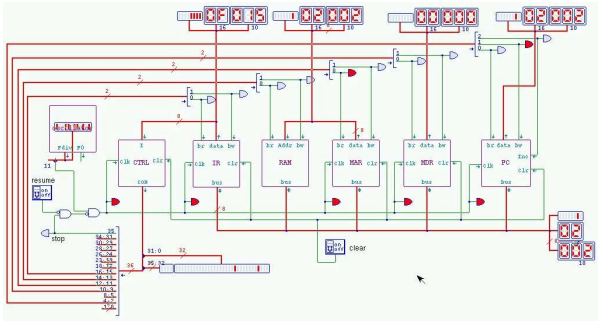
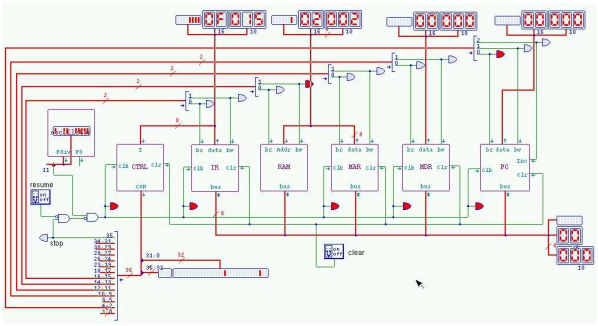


Figura u106.32. Il valore contenuto nella memoria, in corrispondenza dell'indirizzo di salto, viene immesso nel registro ***PC***, facendo in modo che si riparta poi da quella posizione.

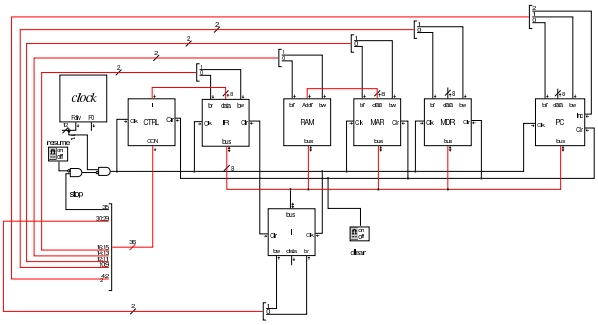


Dovrebbe essere disponibile un video che mostra l'esecuzione del macrocodice descritto: *http://www.youtube.com/watch?v=Z8bTO8WjYYc* .

Versione B: indice della memoria

Nella seconda versione della CPU dimostrativa, viene aggiunto soltanto un registro speciale, denominato ***I***, il cui scopo è quello di contenere un indice della memoria. Nello specifico, serve a poter leggere o scrivere nella memoria RAM, attraverso un indice che possa essere gestito. Il registro ***I*** è realizzato nello stesso modo di ***MDR***, ***MAR*** e ***IR***.

Figura u107.1. Il bus della CPU nella sua seconda fase realizzativa.



Nel codice che descrive i campi del bus di controllo, si aggiungono quelli seguenti, i quali servono specificatamente a gestire il registro ***I***:

```
field i_br[29];          // I <-- bus
field i_bw[30];          // I --> bus
```

Nell'elenco dei codici operativi si aggiungono istruzioni nuove e lo stesso poi nella descrizione del microcodice:

```
op load_imm
{
  map load_imm : 1;                 // load from address #nn
  +0[7:0]=1;
  operands op_1;
};
op load_reg
{
  map load_reg : 2;                 // load from address %I
  +0[7:0]=2;
  operands op_0;
};
op store_imm {
  map store_imm : 3;                // store to address #nn
  +0[7:0]=3;
  operands op_1;
};
op store_reg {
  map store_reg : 4;                // store to address I
  +0[7:0]=4;
  operands op_0;
};
op move_mdr_i {
  map move_mdr_i : 11;              // move MDR to I
  +0[7:0]=11;
  operands op_0;
};
op move_i_mdr {
  map move_i_mdr : 12;              // move I to MDR
  +0[7:0]=12;
  operands op_0;
};
```

```
begin microcode @ 0
…
load_imm:
    mar_br pc_bw;               // MAR <-- PC
    pc_Inc;                     // PC++
    // La memoria non ha un clock,
```

```
    // quindi, non si può passare
    // direttamente a MAR.
    i_br ram_bw;                // I <-- RAM[MAR]
    mar_br i_bw;                // MAR <-- I
    mdr_br ram_bw;              // MDR <-- RAM[MAR]
    ctrl_start ctrl_load;       // CNT <-- 0
//
load_reg:
    mar_br i_bw;                // MAR <-- I
    mdr_br ram_bw;              // MDR <-- RAM[MAR]
    ctrl_start ctrl_load;       // CNT <-- 0
//
store_imm:
    mar_br pc_bw;               // MAR <-- PC
    pc_Inc;                     // PC++
    i_br ram_bw;                // I <-- RAM[MAR]
    mar_br i_bw;                // MAR <-- I
    ram_br mdr_bw;              // RAM[MAR] <-- MDR
    ctrl_start ctrl_load;       // CNT <-- 0
//
store_reg:
    mar_br i_bw;                // MAR <-- I
    ram_br mdr_bw;              // RAM[MAR] <-- MDR
    ctrl_start ctrl_load;       // CNT <-- 0
//
move_mdr_i:
    i_br mdr_bw;                // I <-- MDR
    ctrl_start ctrl_load;       // CNT <-- 0
//
move_i_mdr:
    mdr_br i_bw;                // MDR <-- I
    ctrl_start ctrl_load;       // CNT <-- 0
…
end
```

Figura u107.5. Corrispondenza con il contenuto della memoria che rappresenta il microcodice (la coppia ***m1*** e ***m2*** dell'unità di controllo).

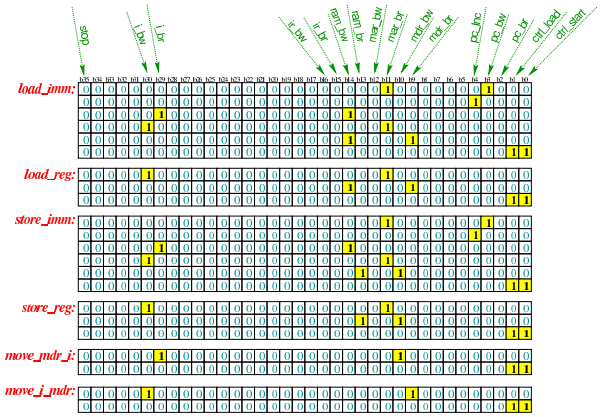


Tabella u107.6. Elenco delle macroistruzioni aggiunte in questa versione della CPU dimostrativa.

Sintassi	Descrizione
`load_imm` ***indirizzo***	*load immediate*: carica nel registro ***MDR*** il contenuto della cella di memoria che corrisponde all'indirizzo indicato dall'argomento. Contestualmente, il registro ***I*** viene modificato e alla fine contiene l'indirizzo di memoria in questione.
`load_reg`	*load register*: carica nel registro ***MDR*** il contenuto della cella di memoria che corrisponde all'indirizzo indicato dal valore contenuto nel registro ***I***.
`store_imm` ***indirizzo***	*store immediate*: salva in memoria, all'indirizzo specificato come argomento, il valore contenuto nel registro ***MDR***. Contestualmente, il registro ***I*** viene modificato e alla fine contiene l'indirizzo di memoria in questione.

Sintassi	Descrizione
`store_reg`	*store register*: salva in memoria, all'indirizzo specificato dal registro ***I***, il valore contenuto nel registro ***MDR***.
`move_mdr_i`	Copia il contenuto del registro ***MDR*** nel registro ***I***.
`move_i_mdr`	Copia il contenuto del registro ***I*** nel registro ***MDR***.

Istruzioni «load»

Come primo esempio viene proposto il macrocodice seguente:

```
begin macrocode @ 0
start:
        load_imm #data_1
        move_mdr_i
        load_reg
stop:
        stop
data_1:
       .byte 3
end
```

In pratica, viene caricato nel registro ***MDR*** il valore corrispondente all'indirizzo in cui si trova l'etichetta '`data_1:`' (facendo i conti si tratta dell'indirizzo 5); successivamente, il valore di ***MDR*** viene copiato nel registro ***I*** e quindi viene caricato nel registro ***MDR*** quanto contenuto nell'indirizzo di memoria corrispondente al valore di ***I***: dal momento che a quel indirizzo si trova il valore 2, corrispondente al codice operativo dell'istruzione **`load_reg`**, al termine, il registro ***MDR*** contiene tale valore. Il file completo che descrive le memorie per Tkgate dovrebbe essere disponibile all'indirizzo *allegati/circuiti-logici/scpu-sub-b.gm*

Figura u107.8. Contenuto della memoria RAM. Le celle indicate con «xx» hanno un valore indifferente.

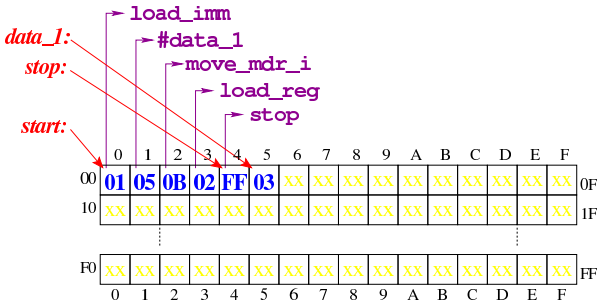
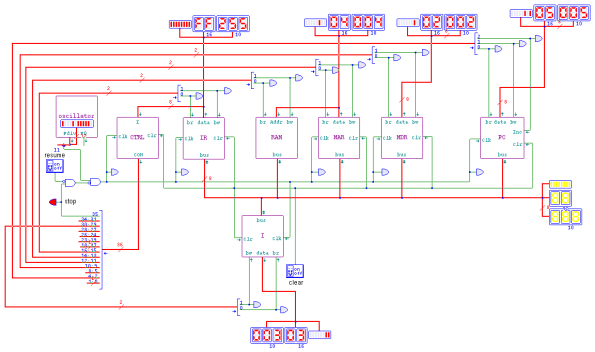


Figura u107.9. Situazione conclusiva del bus dati, dopo l'esecuzione delle istruzioni di caricamento. Video: *http://www.youtube.com/watch?v=AXUSrH49cF49w*



Istruzioni «store»

Viene proposto un altro esempio di macrocodice, nel quale si sperimentano le istruzioni **`store_imm`** e **`store_reg`**:

```
begin macrocode @ 0
start:
        load_imm #data_1
        store_imm #data_2
        move_mdr_i
        store_reg
stop:
        stop
data_1:
       .byte 15
data_2:
       .byte 0
end
```

In questo caso, si carica nel registro ***MDR*** il valore contenuto in memoria in corrispondenza dell'etichetta '**`data_1:`**'; quindi si memorizza, in corrispondenza della posizione di memoria corrispondente all'etichetta '**`data_2:`**', il valore contenuto in ***MDR*** (in pratica, in quella destinazione che prima conteneva il valore zero, viene copiato il valore 15, ovvero $0F_{16}$); quindi il contenuto del registro ***MDR*** viene copiato nel registro ***I*** e poi viene memorizzato il contenuto di ***MDR*** (che è rimasto sempre 15) nella posizione di memoria corrispondente al valore del registro ***I***. In pratica, alla fine si va a scrivere anche nella posizione 15 ($0F_{16}$) della memoria, e ci si mette il valore 15.

Figura u107.11. Contenuto della memoria RAM all'inizio dell'esecuzione.

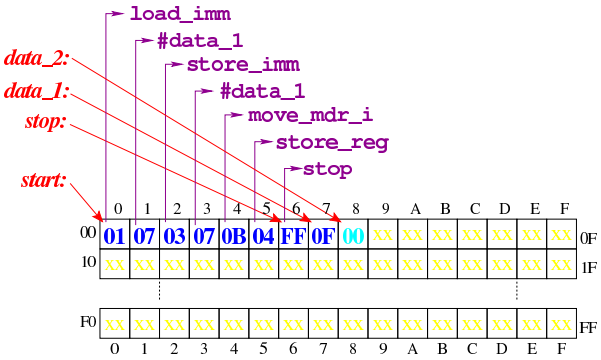


Figura u107.12. Contenuto della memoria RAM dopo l'esecuzione dell'istruzione **`store_imm`**.

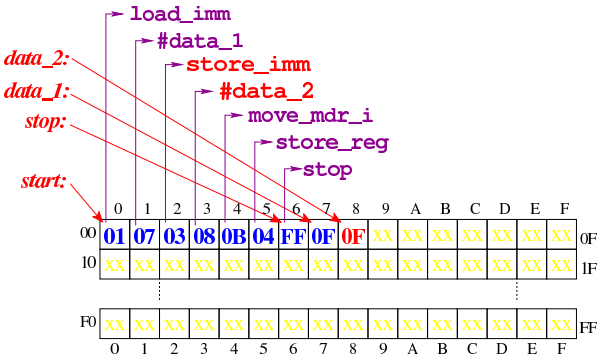


Figura u107.13. Contenuto della memoria RAM al termine dell'esecuzione.

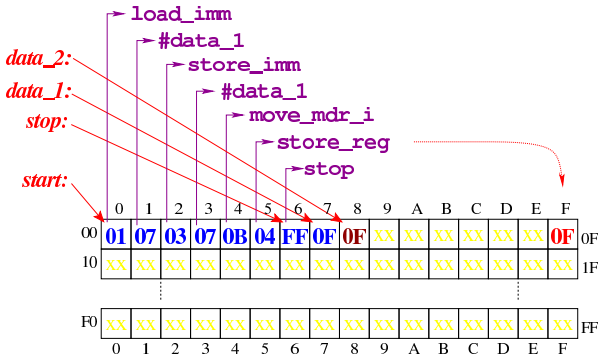
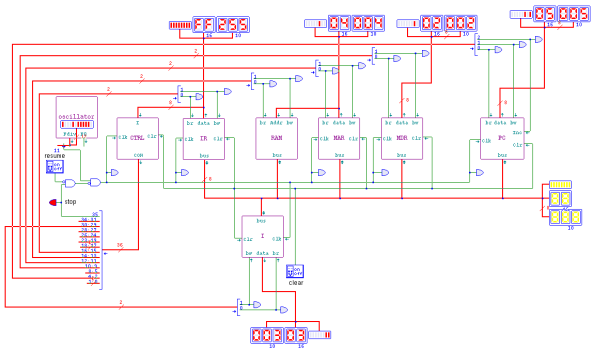
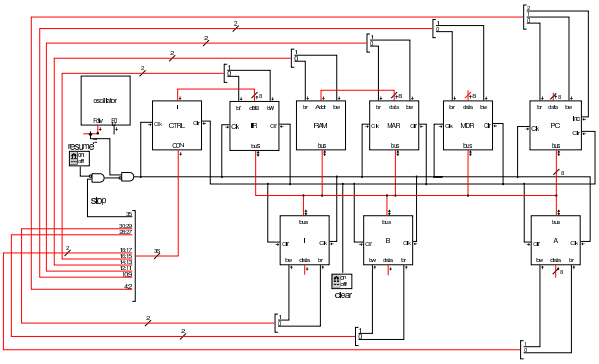


Figura u107.14. Situazione conclusiva del bus dati, dopo l'esecuzione delle istruzioni di memorizzazione. Video: *http://www.youtube.com/watch?v=lHxx3SR56hE56*



Nella terza versione della CPU dimostrativa, vengono aggiunti due registri che per il momento non hanno alcuno scopo particolare: ***A*** e ***B***. Tali registri sono realizzati nello stesso modo di ***I***, ***MDR***, ***MAR*** e ***IR***.

Figura u108.1. Il bus della CPU nella sua terza fase realizzativa.



Nel codice che descrive i campi del bus di controllo, si aggiungono quelli seguenti, i quali servono specificatamente a gestire i registri ***A*** e ***B***:

```
field a_br[17];          // A <-- bus
field a_bw[18];          // A --> bus
field b_br[27];          // B <-- bus
field b_bw[28];          // B --> bus
```

Nell'elenco dei codici operativi si aggiungono istruzioni nuove e lo stesso poi nella descrizione del microcodice:

```
op move_mdr_a {
  map move_mdr_a : 5;                  // move MDR to A
  +0[7:0]=5;
  operands op_0;
};
op move_a_mdr {
  map move_a_mdr : 6;                  // move A to MDR
  +0[7:0]=6;
  operands op_0;
};
op move_mdr_b {
  map move_mdr_b : 7;                  // move MDR to B
  +0[7:0]=7;
  operands op_0;
};
op move_b_mdr {
  map move_b_mdr : 8;                  // move B to MDR
  +0[7:0]=8;
  operands op_0;
};
```

```
begin microcode @ 0
…
move_mdr_a:
    a_br mdr_bw;                 // A <-- MDR
    ctrl_start ctrl_load;        // CNT <-- 0
//
move_a_mdr:
    mdr_br a_bw;                 // MDR <-- A
    ctrl_start ctrl_load;        // CNT <-- 0
//
move_mdr_b:
    b_br mdr_bw;                 // B <-- MDR
    ctrl_start ctrl_load;        // CNT <-- 0
//
move_b_mdr:
    mdr_br b_bw;                 // MDR <-- B
    ctrl_start ctrl_load;        // CNT <-- 0
…
end
```

Figura u108.5. Corrispondenza con il contenuto della memoria che rappresenta il microcodice (la coppia ***m1*** e ***m2*** dell'unità di controllo).

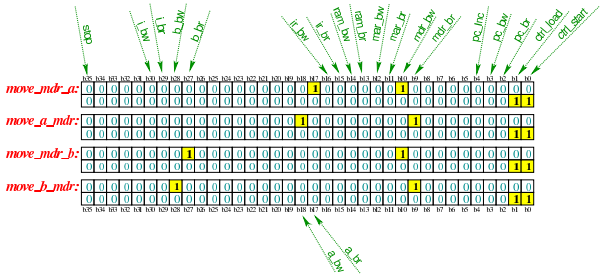


Tabella u108.6. Elenco delle macroistruzioni aggiunte in questa versione della CPU dimostrativa.

Sintassi	Descrizione
`move_mdr_a`	Copia il contenuto del registro ***MDR*** nel registro ***A***.
`move_a_mdr`	Copia il contenuto del registro ***A*** nel registro ***MDR***.
`move_mdr_b`	Copia il contenuto del registro ***MDR*** nel registro ***B***.
`move_b_mdr`	Copia il contenuto del registro ***B*** nel registro ***MDR***.

Come esempio viene proposto il macrocodice seguente:

```
begin macrocode @ 0
start:
        load_imm #data_1
        move_mdr_a
        load_imm #data_2
        move_mdr_b
stop:
        stop
data_1:
       .byte 17
data_2:
       .byte 11
end
```

In pratica, viene caricato nel registro ***MDR*** il valore corrispondente all'indirizzo in cui si trova l'etichetta '**`data_1:`**' (facendo i conti si tratta dell'indirizzo 7); successivamente, il valore di ***MDR*** viene copiato nel registro ***A***; quindi viene caricato nel registro ***MDR*** quanto contenuto nell'indirizzo di memoria corrispondente all'etichetta '**`data_2:`**' (indirizzo 8) e poi copiato nel registro ***B***. Il file completo che descrive le memorie per Tkgate dovrebbe essere disponibile all'indirizzo *allegati/circuiti-logici/scpu-sub-c.gm*

Figura u108.8. Contenuto della memoria RAM. Le celle indicate con «xx» hanno un valore indifferente.

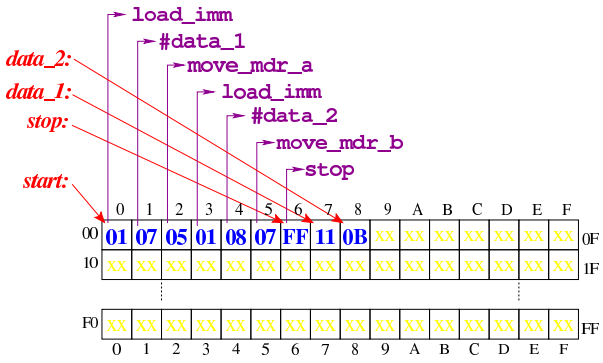
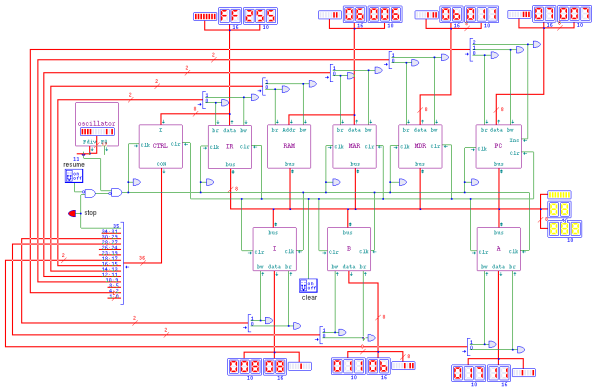


Figura u108.9. Situazione conclusiva del bus dati, dopo l'esecuzione delle istruzioni copia nei registri ***A*** e ***B***. Video: *http://www.youtube.com/watch?v=9qVsCKmxcdk*



Dalle istruzioni introdotte in questa versione della CPU dimostrativa, si può intendere che i dati contenuti nei registri possano essere copiati soltanto con la mediazione del registro ***MDR***; pertanto non esiste un'istruzione **`move_a_b`**. Questa è una semplificazione per evitare di dover dichiarare tante istruzioni nel macrocodice, ma in condizioni normali, tale scelta non sarebbe utile.

Nella quarta versione della CPU dimostrativa, viene aggiunta un'unità aritmetica, logica e di scorrimento (ALU), ma per il momento senza gestire gli indicatori (riporto, segno, zero e straripamento).

Figura u109.1. Il bus della CPU con l'aggiunta dell'unità ALU.

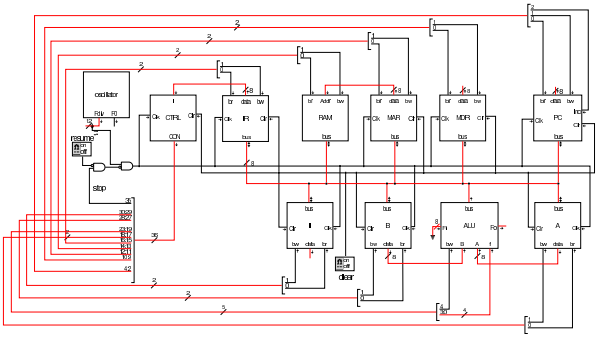


Figura u109.2. La struttura della ALU: si deve fare attenzione a non confondere le linee da un solo bit (di colore nero), rispetto a quelle che ne raccolgono in ranghi maggiori (di colore rosso).

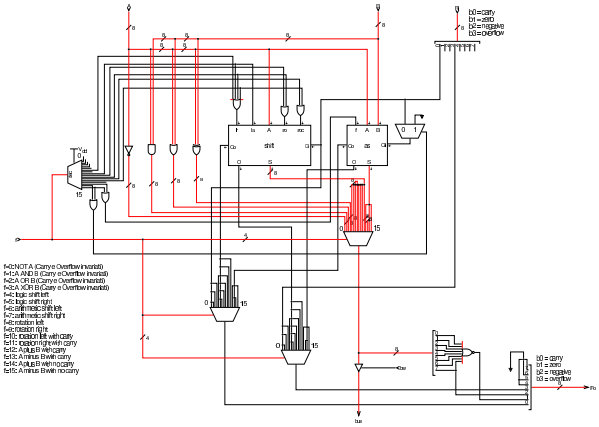


Figura u109.3. Modulo **shift** che si occupa di gestire gli scorrimenti e le rotazioni dei bit.

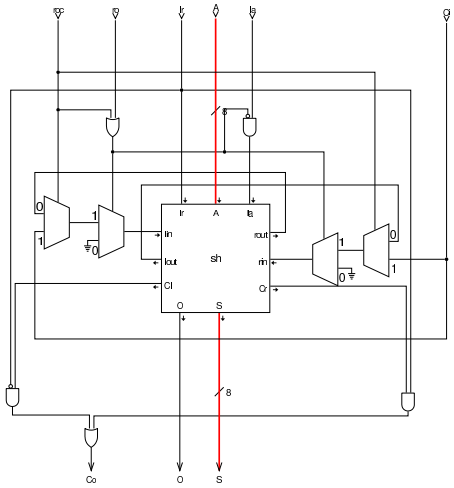


Figura u109.4. Modulo **sh**, contenuto nel modulo **shift**, per lo scorrimento dei bit.

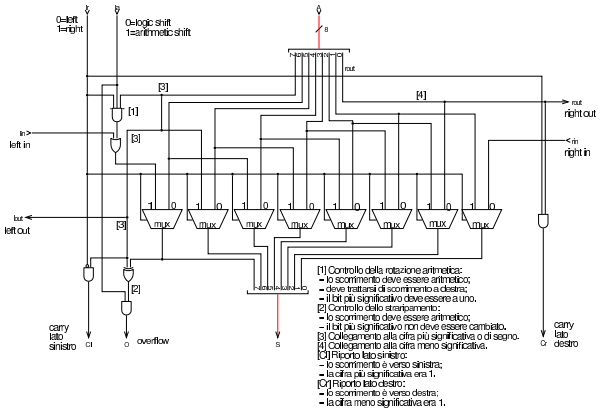
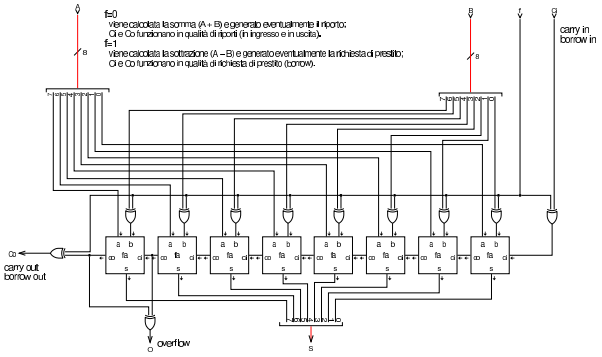


Figura u109.5. Modulo **as** della ALU che ha il compito di sommare o sottrarre gli ingressi.



Nel codice che descrive i campi del bus di controllo, si aggiungono quelli seguenti, i quali servono specificatamente a gestire la ALU. Si può osservare che la ALU ha il controllo di scrittura nel bus, ma non quello di lettura, dato che dal bus non riceve dati, e richiede il controllo della funzione che vi si vuole svolgere :

```
field alu_f[22:19]={
        not_a=0,
        a_and_b=1,
        a_or_b=2,
        a_xor_b=3,
        logic_shift_left=4,
        logic_shift_right=5,
        arith_shift_left=6,
        arith_shift_right=7,
        rotate_left=8,
        rotate_right=9,
        rotate_carry_left=10,
        rotate_carry_right=11,
        a_plus_b_carry=12,
        a_minus_b_borrow=13,
        a_plus_b=14,
        a_minus_b=15
};
field alu_bw[23];       // ALU --> bus
field fl_ar[24];        // FL <-- ALU
```


Tra i campi del bus di controllo si vede anche ***fl_ar*** che per ora può essere ignorato: viene chiarito il suo utilizzo quando nella prossima versione della CPU dimostrativa si aggiunge il registro ***FL***. Attualmente, nel microcodice vi si fa già riferimento, perché le microistruzioni prese ora in considerazione, in un secondo momento devono avere a che fare con tale registro.

Nell'elenco dei codici operativi si aggiungono istruzioni nuove e lo stesso poi nella descrizione del microcodice:

```
op not {
  map not : 32;                         // A = NOT A
  +0[7:0]=32;
  operands op_0;
};
op and {
  map and : 33;                         // A = A AND B
  +0[7:0]=33;
  operands op_0;
};
op or {
  map or : 34;                          // A = A OR B
  +0[7:0]=34;
  operands op_0;
};
op xor {
  map xor : 35;                         // A = A OR B
  +0[7:0]=35;
  operands op_0;
};
op lshl {
  map lshl : 36;                        // A = A << 1
  +0[7:0]=36;
  operands op_0;
};
op lshr {
  map lshr : 37;                        // A = A >> 1
  +0[7:0]=37;
  operands op_0;
};
op ashl {
  map ashl : 38;                        // A = A << 1
  +0[7:0]=38;
  operands op_0;
};
op ashr {
  map ashr : 39;                        // A = +/-A >> 1
  +0[7:0]=39;
  operands op_0;
};
op rotl {
  map rotl : 40;                        // A = A rotate left
  +0[7:0]=40;
  operands op_0;
};
op rotr {
  map rotr : 41;                        // A = A rotate right
  +0[7:0]=41;
  operands op_0;
};
op add {
  map add : 46;                         // A = A + B
  +0[7:0]=46;
  operands op_0;
};
op sub {
  map sub : 47;                         // A = A - B
  +0[7:0]=47;
```

```
  operands op_0;
};
```

```
begin microcode @ 0
…
not:
    a_br alu_f=not_a alu_bw fl_ar;        // A <-- NOT A
    ctrl_start ctrl_load;                 // CNT <-- 0
//
and:
    a_br alu_f=a_and_b alu_bw fl_ar;      // A <-- A AND B
    ctrl_start ctrl_load;                 // CNT <-- 0
//
or:
    a_br alu_f=a_or_b alu_bw fl_ar;       // A <-- A OR B
    ctrl_start ctrl_load;                 // CNT <-- 0
//
xor:
    a_br alu_f=a_xor_b alu_bw fl_ar;      // A <-- A XOR B
    ctrl_start ctrl_load;                 // CNT <-- 0
//
lshl:
    a_br alu_f=logic_shift_left alu_bw fl_ar; // A <-- A << 1
    ctrl_start ctrl_load;                    // CNT <-- 0
//
lshr:
    a_br alu_f=logic_shift_right alu_bw fl_ar; // A <-- A >> 1
    ctrl_start ctrl_load;                     // CNT <-- 0
//
ashl:
    a_br alu_f=arith_shift_left alu_bw fl_ar; // A <-- A*2
    ctrl_start ctrl_load;                    // CNT <-- 0
//
ashr:
    a_br alu_f=arith_shift_right alu_bw fl_ar; // A <-- A/2
    ctrl_start ctrl_load;                     // CNT <-- 0
//
rotl:
    a_br alu_f=rotate_left alu_bw fl_ar; // A <-- A rot. left
    ctrl_start ctrl_load;                // CNT <-- 0
//
rotr:
    a_br alu_f=rotate_right alu_bw fl_ar; // A <-- A rot. right
    ctrl_start ctrl_load;                 // CNT <-- 0
//
add:
    a_br alu_f=a_plus_b alu_bw fl_ar;     // A <-- A + B
    ctrl_start ctrl_load;                 // CNT <-- 0
//
sub:
    a_br alu_f=a_minus_b alu_bw fl_ar;    // A <-- A - B
    ctrl_start ctrl_load;                 // CNT <-- 0
…
end
```

Figura u109.9. Corrispondenza con il contenuto della memoria che rappresenta il microcodice (la coppia ***m1*** e ***m2*** dell'unità di controllo).

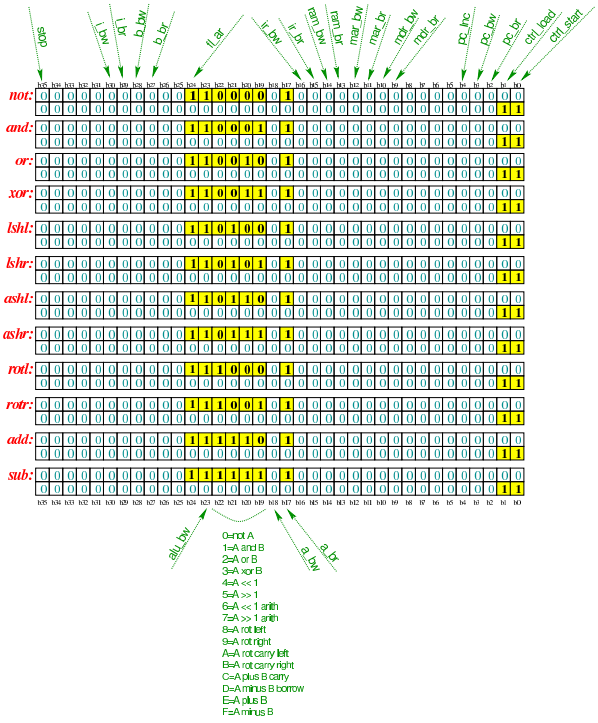


Tabella u109.10. Elenco delle macroistruzioni aggiunte in questa versione della CPU dimostrativa. Nella descrizione sintetica delle operazioni si usa la notazione del linguaggio C.

Sintassi	Descrizione
`not`	***A*** = ~***A*** Complemento a uno del contenuto di ***A***.
`and`	***A*** = ***A*** & ***B*** Si assegna ad ***A*** il risultato di ***A*** AND ***B***, bit per bit.
`or`	***A*** = ***A*** | ***B*** Si assegna ad ***A*** il risultato di ***A*** OR ***B***, bit per bit.
`xor`	***A*** = ***A*** ^ ***B*** Si assegna ad ***A*** il risultato di ***A*** XOR ***B***, bit per bit.
`lshl`	***A*** = ***A*** << 1 Si assegna ad ***A*** il risultato dello scorrimento logico a sinistra dei bit di ***A***.
`lshr`	***A*** = ***A*** >> 1 Si assegna ad ***A*** il risultato dello scorrimento logico a destra dei bit di ***A***.
`ashl`	***A*** = ***A*** << 1 Si assegna ad ***A*** il risultato dello scorrimento aritmetico a sinistra dei bit di ***A*** (in pratica è identico a **`lshl`**).
`ashr`	***A*** = ***A*** >> 1 Si assegna ad ***A*** il risultato dello scorrimento aritmetico a destra dei bit di ***A***.
`rotl`	Si assegna ad ***A*** il risultato della rotazione a sinistra dei bit di ***A***.
`rotr`	Si assegna ad ***A*** il risultato della rotazione a destra dei bit di ***A***.
`add`	***A*** = ***A*** + ***B*** Si assegna ad ***A*** il risultato della somma di ***A*** e ***B***, senza tenere conto del riporto precedente.

Sintassi	Descrizione
`sub`	***A*** = ***A*** − ***B*** Si assegna ad ***A*** il risultato della sottrazione di ***A*** e ***B***, senza tenere conto della richiesta di prestito precedente.

Nelle sezioni successive, vengono proposti diversi esempi, nei quali si sperimentano tutte le istruzioni nuove introdotte.

Istruzione «not»

Listato u109.11. Macrocodice per sperimentare l'istruzione **`not`**: si carica un valore dalla memoria, lo si copia nel registro ***A***, si calcola il complemento a uno e il risultato va ad aggiornare il registro ***A***. Il file completo che descrive le memorie per Tkgate dovrebbe essere disponibile presso *allegati/circuiti-logici/scpu-sub-d-not.gm* .

```
begin macrocode @ 0
start:
        load_imm #data_1
        move_mdr_a
        not
stop:
        stop
data_1:
        .byte 17
end
```

Figura u109.12. Contenuto della memoria RAM. Le celle indicate con «xx» hanno un valore indifferente.

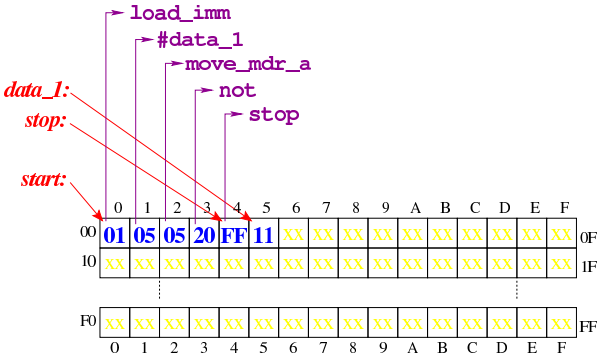
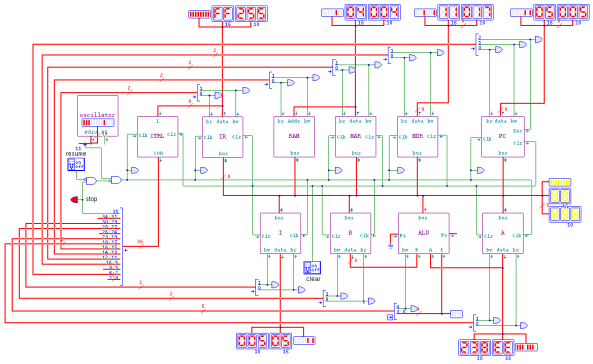


Figura u109.13. Situazione conclusiva del bus dati, dopo l'esecuzione dell'istruzione **`not`**. Video: *http://www.youtube.com/watch?v=x5Vnhd72vh728*



Istruzione «and»

Listato u109.14. Macrocodice per sperimentare l'istruzione **and**: si caricano dalla memoria i valori da assegnare ai registri ***A*** e ***B***, quindi si esegue un AND binario che va ad aggiornare il registro ***A***. Il file completo che descrive le memorie per Tkgate dovrebbe essere disponibile presso *allegati/circuiti-logici/scpu-sub-d-and.gm* .

```
begin macrocode @ 0
start:
        load_imm #data_1
        move_mdr_a
        load_imm #data_2
        move_mdr_b
        and
stop:
        stop
data_1:
        .byte 17
data_2:
        .byte 11
end
```

Figura u109.15. Contenuto della memoria RAM. Le celle indicate con «xx» hanno un valore indifferente.

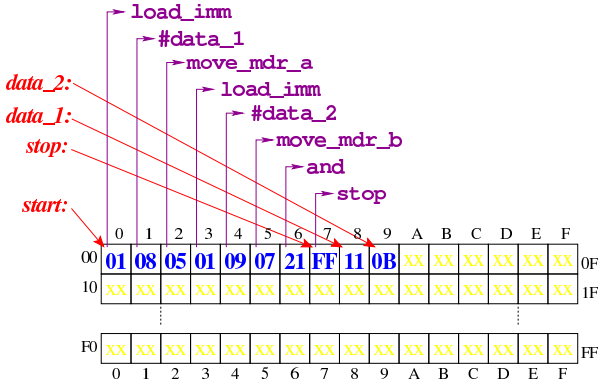
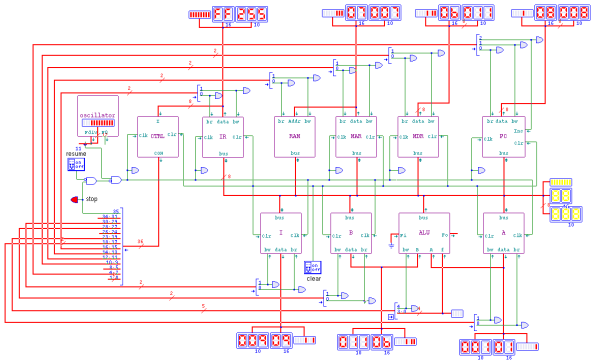


Figura u109.16. Situazione conclusiva del bus dati, dopo l'esecuzione dell'istruzione **and**. Video: *http://www.youtube.com/watch?v=2ra7SHxBvYY*



Istruzione «or»

Listato u109.17. Macrocodice per sperimentare l'istruzione **`or`**: si caricano dalla memoria i valori da assegnare ai registri ***A*** e ***B***, quindi si esegue un OR binario che va ad aggiornare il registro ***A***. Il file completo che descrive le memorie per Tkgate dovrebbe essere disponibile presso *allegati/circuiti-logici/scpu-sub-d-or.gm* .

```
begin macrocode @ 0
start:
        load_imm #data_1
        move_mdr_a
        load_imm #data_2
        move_mdr_b
        or
stop:
        stop
data_1:
        .byte 17
data_2:
        .byte 11
end
```

Figura u109.18. Contenuto della memoria RAM. Le celle indicate con «xx» hanno un valore indifferente.

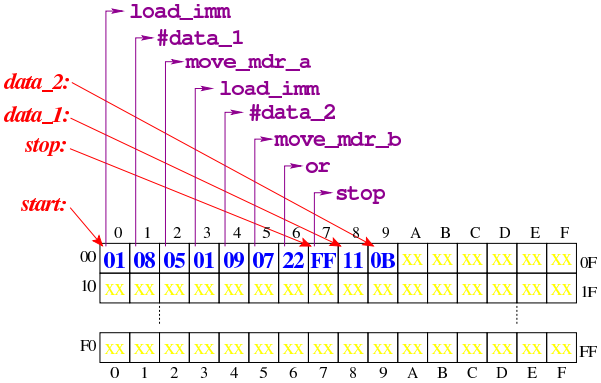
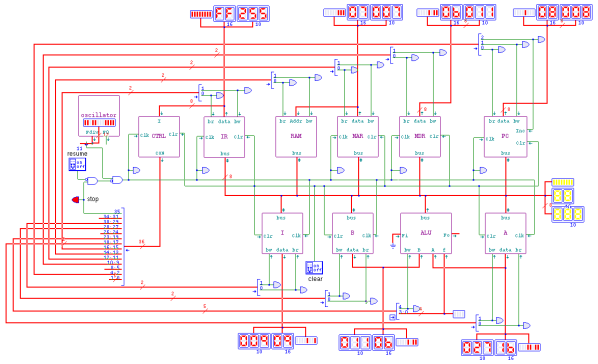


Figura u109.19. Situazione conclusiva del bus dati, dopo l'esecuzione dell'istruzione **or**. Video: *http://www.youtube.com/watch?v=7E-2uA6fVoY*



Istruzione «xor»

Listato u109.20. Macrocodice per sperimentare l'istruzione **xor**: si caricano dalla memoria i valori da assegnare ai registri ***A*** e ***B***, quindi si esegue un XOR binario che va ad aggiornare il registro ***A***. Il file completo che descrive le memorie per Tkgate dovrebbe essere disponibile presso *allegati/circuiti-logici/scpu-sub-d-xor.gm* .

```
begin macrocode @ 0
start:
        load_imm #data_1
        move_mdr_a
        load_imm #data_2
        move_mdr_b
        xor
stop:
        stop
data_1:
        .byte 17
data_2:
        .byte 11
end
```


Figura u109.21. Contenuto della memoria RAM. Le celle indicate con «xx» hanno un valore indifferente.

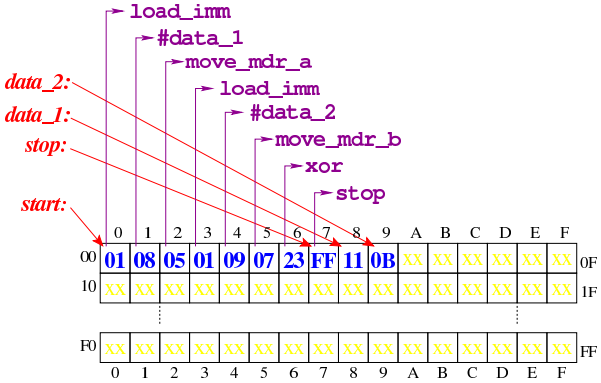
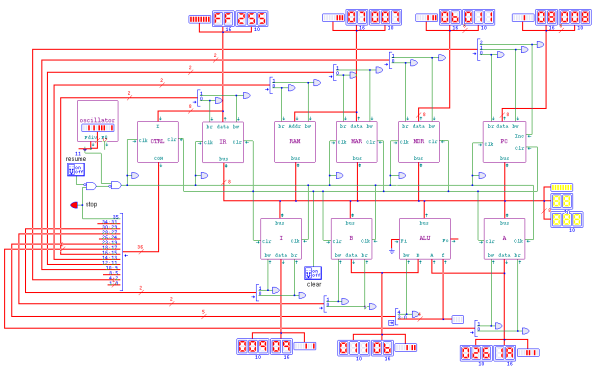


Figura u109.22. Situazione conclusiva del bus dati, dopo l'esecuzione dell'istruzione **xor**. Video: *http://www.youtube.com/watch?v=TuzknbyeabQ*



Istruzioni «lshl» e «lshr»

Listato u109.23. Macrocodice per sperimentare le istruzioni di scorrimento logico: si carica un valore dalla memoria, lo si copia nel registro ***A***, si esegue lo scorrimento a sinistra e il risultato va ad aggiornare il registro ***A***; si copia il risultato nel registro ***B*** e si carica nuovamente il valore originale per eseguire lo scorrimento a destra (che va ad aggiornare sempre il registro ***A***). Il file completo che descrive le memorie per Tkgate dovrebbe essere disponibile presso *allegati/circuiti-logici/scpu-sub-d-lsh.gm* .

```
begin macrocode @ 0
start:
        load_imm #data_1
        move_mdr_a
        lshl
        move_a_mdr
        move_mdr_b
        load_imm #data_1
        move_mdr_a
        lshr
stop:
        stop
data_1:
        .byte 17
end
```

Figura u109.24. Contenuto della memoria RAM. Le celle indicate con «xx» hanno un valore indifferente.

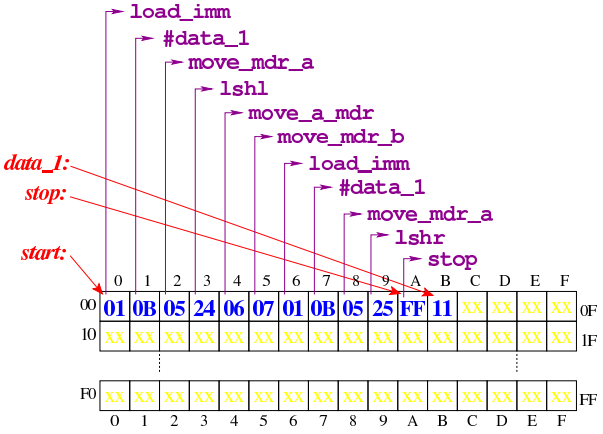
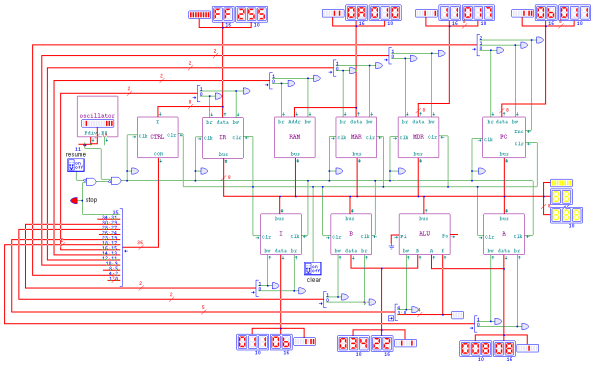


Figura u109.25. Situazione conclusiva del bus dati. Video: *http://www.youtube.com/watch?v=pkRfWYqGeB4*



Istruzioni «ashl» e «ashr»

Listato u109.26. Macrocodice per sperimentare le istruzioni di scorrimento aritmetico: si carica un valore dalla memoria, lo si copia nel registro ***A***, si esegue lo scorrimento a sinistra e il risultato va ad aggiornare il registro ***A***; si copia il risultato nel registro ***B*** e si carica nuovamente il valore originale per eseguire lo scorrimento a destra (che va ad aggiornare sempre il registro ***A***). Il file completo che descrive le memorie per Tkgate dovrebbe essere disponibile presso *allegati/circuiti-logici/scpu-sub-d-ash.gm* .

```
begin macrocode @ 0
start:
        load_imm #data_1
        move_mdr_a
        ashl
        move_a_mdr
        move_mdr_b
        load_imm #data_1
        move_mdr_a
        ashr
stop:
        stop
data_1:
        .byte 143
end
```


Figura u109.27. Contenuto della memoria RAM. Le celle indicate con «xx» hanno un valore indifferente.

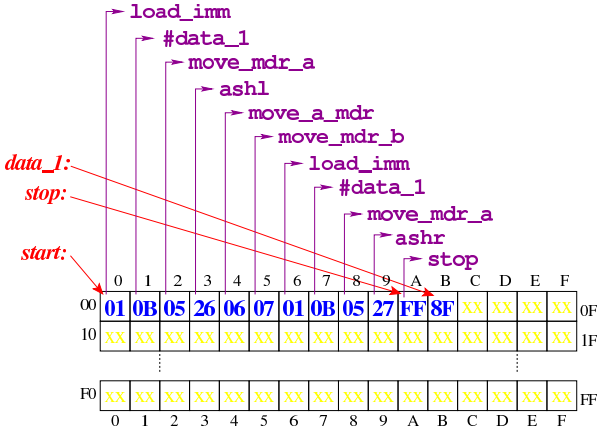
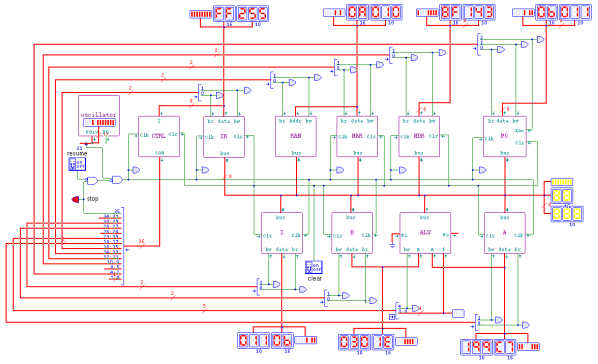


Figura u109.28. Situazione conclusiva del bus dati. Video: *http://www.youtube.com/watch?v=3rvR1WvWd1k*



Istruzioni «rotl» e «rotr»

Listato u109.29. Macrocodice per sperimentare le istruzioni di scorrimento logico: si carica un valore dalla memoria, lo si copia nel registro ***A***, si esegue la rotazione a sinistra e il risultato va ad aggiornare il registro ***A***; si copia il risultato nel registro ***B*** e si carica nuovamente il valore originale per eseguire la rotazione a destra (che va ad aggiornare sempre il registro ***A***). Il file completo che descrive le memorie per Tkgate dovrebbe essere disponibile presso *allegati/circuiti-logici/scpu-sub-d-rot.gm* .

```
begin macrocode @ 0
start:
        load_imm #data_1
        move_mdr_a
        rotl
        move_a_mdr
        move_mdr_b
        load_imm #data_1
        move_mdr_a
        rotr
stop:
        stop
data_1:
        .byte 17
end
```

Figura u109.30. Contenuto della memoria RAM. Le celle indicate con «xx» hanno un valore indifferente.

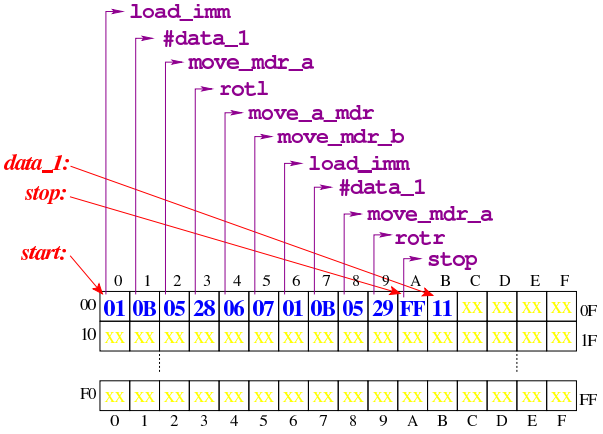
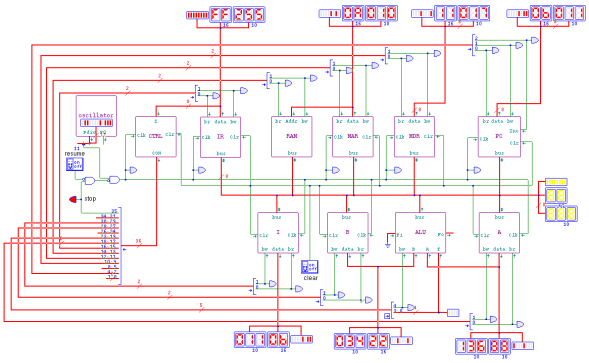


Figura u109.31. Situazione conclusiva del bus dati. Video: *http://www.youtube.com/watch?v=KCi8n6bnLQo*



Istruzione «add»

Listato u109.32. Macrocodice per sperimentare l'istruzione **add**: si caricano dalla memoria i valori da assegnare ai registri ***A*** e ***B***, quindi si esegue la somma che va ad aggiornare il registro ***A***. Il file completo che descrive le memorie per Tkgate dovrebbe essere disponibile presso *allegati/circuiti-logici/scpu-sub-d-add.gm* .

```
begin macrocode @ 0
start:
        load_imm #data_1
        move_mdr_a
        load_imm #data_2
        move_mdr_b
        add
stop:
        stop
data_1:
        .byte 17
data_2:
        .byte 11
end
```


Figura u109.33. Contenuto della memoria RAM. Le celle indicate con «xx» hanno un valore indifferente.

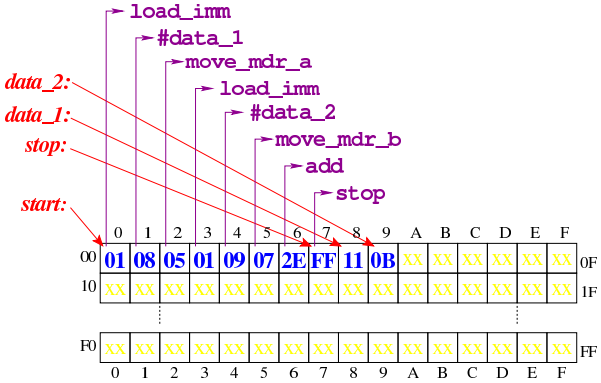
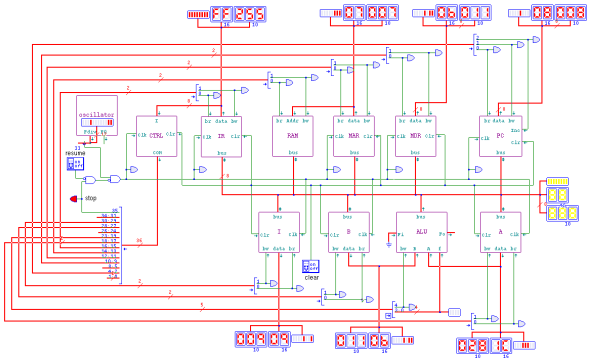


Figura u109.34. Situazione conclusiva del bus dati, dopo l'esecuzione dell'istruzione **add**. Video: *http://www.youtube.com/watch?v=QQJwz2yVwA8*



Istruzione «sub»

Listato u109.35. Macrocodice per sperimentare l'istruzione **sub**: si caricano dalla memoria i valori da assegnare ai registri ***A*** e ***B***, quindi si esegue la sottrazione (***A*−*B***) che va ad aggiornare il registro ***A***. Il file completo che descrive le memorie per Tkgate dovrebbe essere disponibile presso *allegati/circuiti-logici/scpu-sub-d-sub.gm* .

```
begin macrocode @ 0
start:
        load_imm #data_1
        move_mdr_a
        load_imm #data_2
        move_mdr_b
        sub
stop:
        stop
data_1:
        .byte 17
data_2:
        .byte 11
end
```

Figura u109.36. Contenuto della memoria RAM. Le celle indicate con «xx» hanno un valore indifferente.

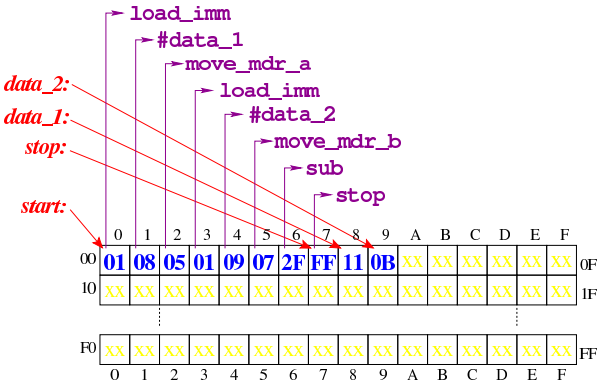
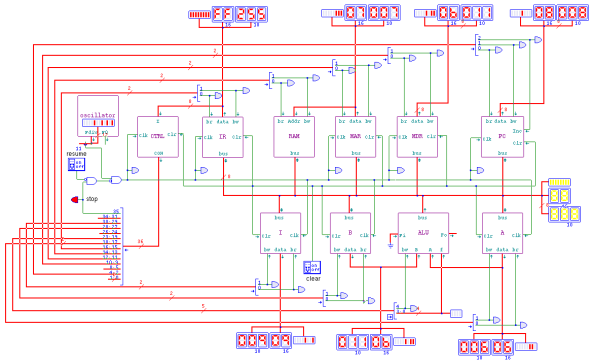


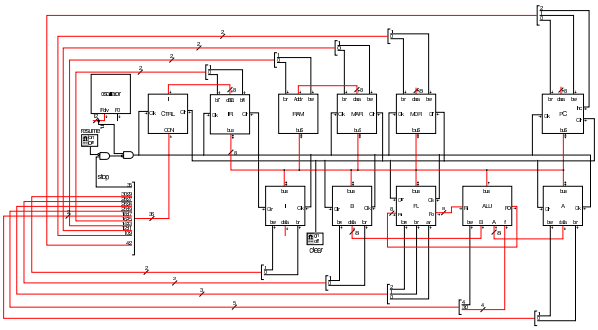
Figura u109.37. Situazione conclusiva del bus dati, dopo l'esecuzione dell'istruzione **sub**. Video: *http://www.youtube.com/watch?v=VRd8ilJbK_Y*



Versione E: indicatori

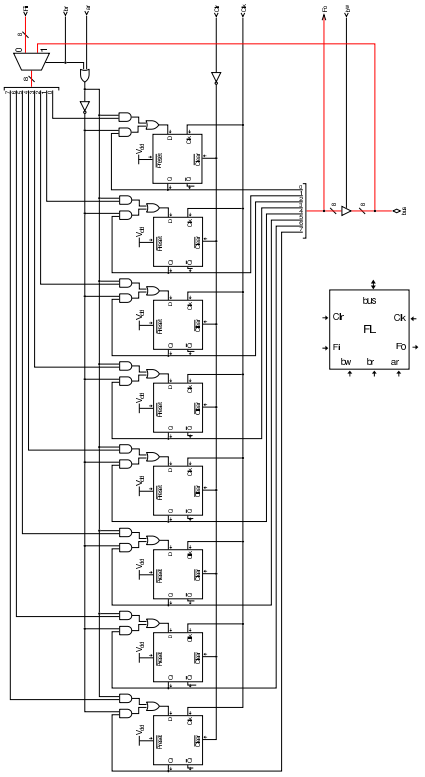
Nella quinta versione della CPU dimostrativa, viene aggiunto un registro per annotare lo stato degli indicatori, relativi all'esito di alcune operazioni svolte dalla ALU: riporto, segno, zero e straripamento.

Figura u110.1. Il bus della CPU con l'aggiunta del registro ***FL*** per la gestione degli indicatori.



Come si può comprendere dagli ingressi e dalle uscite che possiede, il registro ***FL*** può immettere dati nel bus e può essere modificato leggendo dati dal bus; inoltre, può leggere direttamente dalla ALU (ingresso ***Fi***), e per questo esiste un ingresso di abilitazione ulteriore, denominato ***ar*** (*ALU read*), mentre fornisce in ogni istante il proprio valore memorizzato alla ALU stessa (uscita ***Fo***).

Figura u110.2. La struttura interna del registro ***FL***: gli otto moduli che si vedono sono flip-flop D.



Nel codice che descrive i campi del bus di controllo, si aggiungono quelli seguenti (a parte ***fl_ar*** già apparso nella sezione precedente), i quali servono specificatamente a gestire il registro ***FL***:

```
field fl_ar[24];        // FL <-- ALU
field fl_br[25];        // FL <-- bus
field fl_bw[26];        // FL --> bus
```

Nell'elenco dei codici operativi si aggiungono istruzioni nuove e lo stesso poi nella descrizione del microcodice:

```
op move_mdr_fl {
  map move_mdr_fl : 9;          // move MDR to FL
  +0[7:0]=9;
  operands op_0;
};
op move_fl_mdr {
  map move_fl_mdr : 10;         // move FL to MDR
  +0[7:0]=10;
  operands op_0;
};
op rotcl {
  map rotcl : 42;               // A = A rotate carry left
  +0[7:0]=42;
  operands op_0;
};
op rotcr {
  map rotcr : 43;               // A = A rotate carry right
  +0[7:0]=43;
  operands op_0;
};
op add_carry {
  map add_carry : 44;           // A = A + B + carry
  +0[7:0]=44;
  operands op_0;
};
op sub_borrow {
  map sub_borrow : 45;          // A = A - B - borrow
  +0[7:0]=45;
  operands op_0;
};
```

```
begin microcode @ 0
…
//
move_mdr_fl:
    fl_br mdr_bw;               // FL <-- MDR
    ctrl_start ctrl_load;       // CNT <-- 0
//
move_fl_mdr:
    mdr_br fl_bw;               // MDR <-- FL
    ctrl_start ctrl_load;       // CNT <-- 0
//
rotcl:
    a_br alu_f=rotate_carry_left alu_bw fl_ar; // A <-- A rot. carry l
    ctrl_start ctrl_load;                     // CNT <-- 0
//
rotcr:
    a_br alu_f=rotate_carry_right alu_bw fl_ar; // A <-- A rot. carry r
    ctrl_start ctrl_load;                      // CNT <-- 0
//
add_carry:
    a_br alu_f=a_plus_b_carry alu_bw fl_ar; // A <-- A + B + carry
    ctrl_start ctrl_load;                  // CNT <-- 0
//
sub_borrow:
    a_br alu_f=a_minus_b_borrow alu_bw fl_ar; // A <-- A - B - borrow
    ctrl_start ctrl_load;                    // CNT <-- 0
…
end
```


Figura u110.6. Corrispondenza con il contenuto della memoria che rappresenta il microcodice (la coppia ***m1*** e ***m2*** dell'unità di controllo).

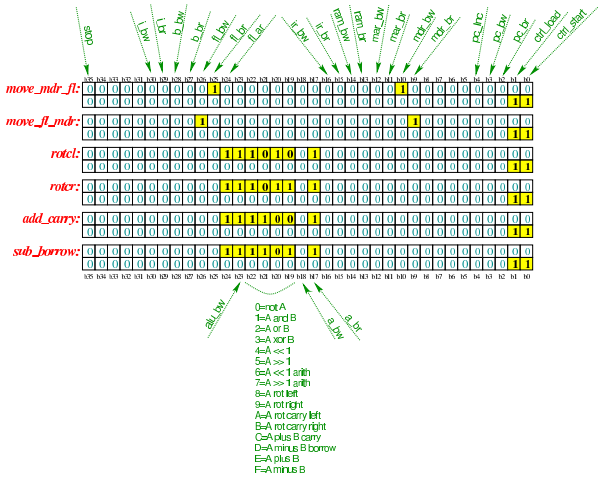


Tabella u110.7. Elenco delle macroistruzioni aggiunte in questa versione della CPU dimostrativa.

Sintassi	Descrizione
`move_mdr_fl`	Copia il contenuto del registro ***MDR*** nel registro ***FL***.
`move_fl_mdr`	Copia il contenuto del registro ***FL*** nel registro ***MDR***.
`rotcl`	Esegue la rotazione a sinistra del contenuto del registro ***A***, utilizzando anche l'indicatore di riporto.
`rotcr`	Esegue la rotazione a destra del contenuto del registro ***A***, utilizzando anche l'indicatore di riporto.
`add_carry`	Esegue la somma dei registri ***A*** e ***B***, tenendo conto del riporto precedente, aggiornando di conseguenza lo stesso registro ***A***.
`sub_carry`	Esegue la sottrazione ***A*−*B***, tenendo conto di un'eventuale richiesta di prestito precedente, aggiornando di conseguenza lo stesso registro ***A***.

Nelle sezioni successive, vengono proposti alcuni esempi, nei quali si sperimentano tutte le istruzioni nuove introdotte.

Istruzione «rotcl» e «rotcr»

Listato u110.8. Macrocodice per sperimentare le istruzioni **`rotcl`** e **`rotcr`**: si carica in memoria il valore da assegnare al registro ***A***, si eseguono cinque scorrimenti a sinistra, con l'uso del riporto e il risultato viene copiato nel registro ***B***; poi, con il valore presente in quel momento nel registro ***A***, si eseguono altri cinque rotazioni a destra, sempre con l'uso del riporto. Il file completo che descrive le memorie per Tkgate dovrebbe essere disponibile presso *allegati/circuiti-logici/scpu-sub-d-rotc.gm* .

```
begin macrocode @ 0
start:
        load_imm #data_1
        move_mdr_a
        rotcl
        rotcl
        rotcl
        rotcl
        rotcl
        move_a_mdr
        move_mdr_b
        rotcr
        rotcr
        rotcr
        rotcr
        rotcr
```

```
stop:
        stop
data_1:
        .byte 160
end
```

Figura u110.9. Contenuto della memoria RAM. Le celle indicate con «xx» hanno un valore indifferente.

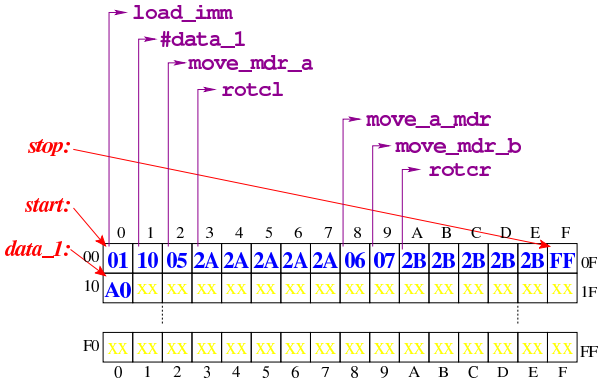
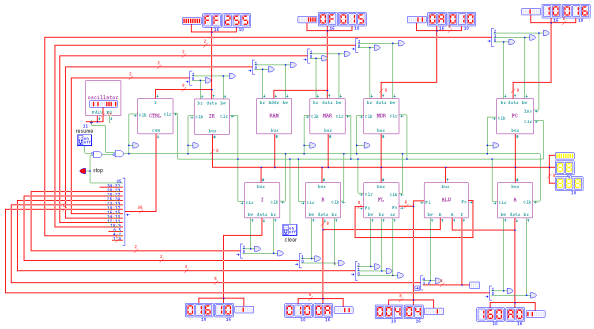


Figura u110.10. Situazione conclusiva del bus dati, dopo l'esecuzione delle istruzioni di rotazione con riporto. Video: *http://www.youtube.com/watch?v=Zl3d-Tg5C1Q*



Istruzione «add_carry»

«

Listato u110.11. Macrocodice per sperimentare l'istruzione **add_carry**: si vogliono sommare due numeri $12FF_{16}$ e $11EE_{16}$, necessariamente in due passaggi. Prima viene sommata la coppia FF_{16} e EE_{16}, con l'istruzione **add**, la quale produce il risultato ED_{16} con riporto, quindi viene sommata la coppia 12_{16} e 11_{16}, assieme al riporto precedente, ottenendo 24_{16}. In pratica, il risultato completo sarebbe $24ED_{16}$ che viene collocato in memoria dividendolo in due byte distinti. Il file completo che descrive le memorie per Tkgate dovrebbe essere disponibile presso *allegati/circuiti-logici/scpu-sub-d-add_carry.gm* .

```
begin macrocode @ 0
start:
        load_imm #data_0
        move_mdr_a
        load_imm #data_2
        move_mdr_b
        add
        move_a_mdr
        store_imm #data_4
        load_imm #data_1
        move_mdr_a
        load_imm #data_3
        move_mdr_b
        add_carry
        move_a_mdr
        store_imm #data_5
stop:
        stop
// 0x12FF = 4863
data_0:
        .byte 0xFF
```



```
data_1:
        .byte 0x12
// 0x11EE = 4590
data_2:
        .byte 0xEE
data_3:
        .byte 0x11
data_4:
        .byte 0
data_5:
        .byte 0
end
```

Figura u110.12. Contenuto della memoria RAM prima dell'esecuzione. Le celle indicate con «xx» hanno un valore indifferente.

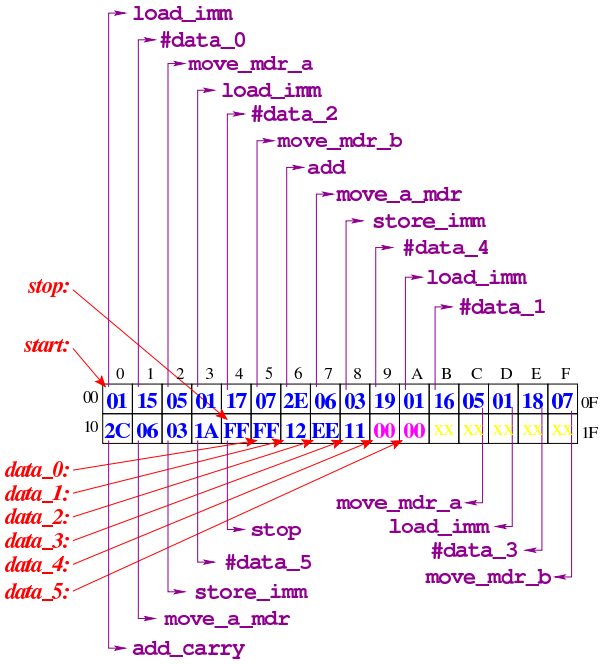


Figura u110.13. Al termine dell'esecuzione, le celle di memoria che devono contenere il risultato riportano il contenuto che si può vedere evidenziato qui.

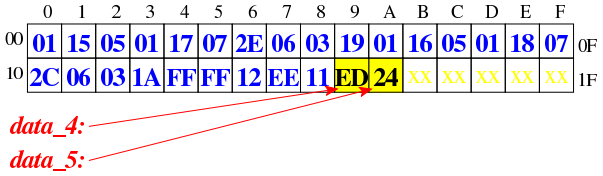
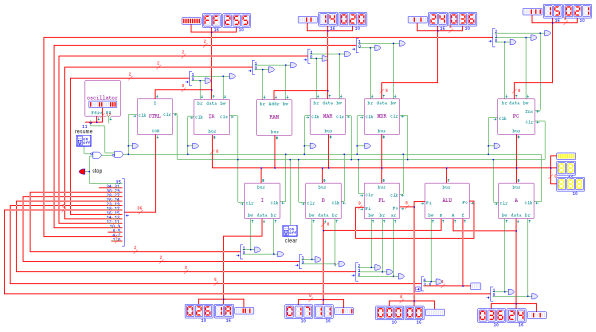


Figura u110.14. Situazione conclusiva del bus dati. Video: *http://www.youtube.com/watch?v=1Xu4MxWBwW4*



«

Listato u110.15. Macrocodice per sperimentare l'istruzione **sub_borrow**: si vuole eseguire la sottrazione $12EE_{16}-11FF_{16}$ e la si deve svolgere necessariamente in due passaggi. Prima viene sottratta la coppia EE_{16} e FF_{16}, con l'istruzione **sub**, la quale produce il risultato EF_{16} con richiesta di un prestito, quindi viene sottratta la coppia 12_{16} e 11_{16}, tenendo conto della richiesta del prestito dalle cifre precedenti, ottenendo 00_{16}. In pratica, il risultato completo sarebbe $00EF_{16}$ che viene collocato in memoria dividendolo in due byte distinti. Il file completo che descrive le memorie per Tkgate dovrebbe essere disponibile presso *allegati/circuiti-logici/scpu-sub-d-sub_borrow.gm* .

```
begin macrocode @ 0
start:
        load_imm #data_0
        move_mdr_a
        load_imm #data_2
        move_mdr_b
        sub
        move_a_mdr
        store_imm #data_4
        load_imm #data_1
        move_mdr_a
        load_imm #data_3
        move_mdr_b
        sub_borrow
        move_a_mdr
        store_imm #data_5
stop:
        stop
// 0x12EE = 4846
data_0:
        .byte 0xEE
data_1:
        .byte 0x12
// 0x11FF = 4607
data_2:
        .byte 0xFF
data_3:
        .byte 0x11
data_4:
        .byte 0
data_5:
        .byte 0
end
```

Figura u110.16. Contenuto della memoria RAM prima dell'esecuzione. Le celle indicate con «xx» hanno un valore indifferente.

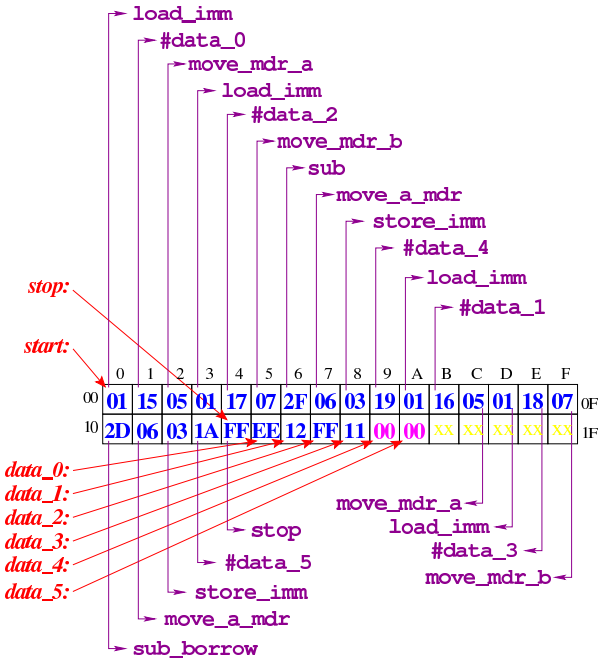


Figura u110.17. Al termine dell'esecuzione, le celle di memoria che devono contenere il risultato riportano il contenuto che si può vedere evidenziato qui.

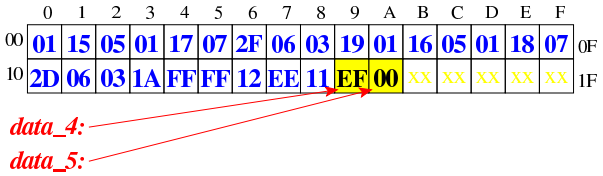
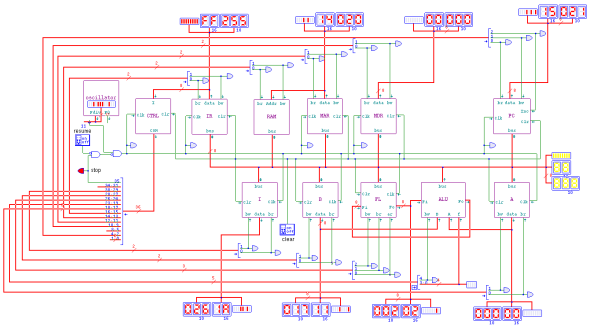
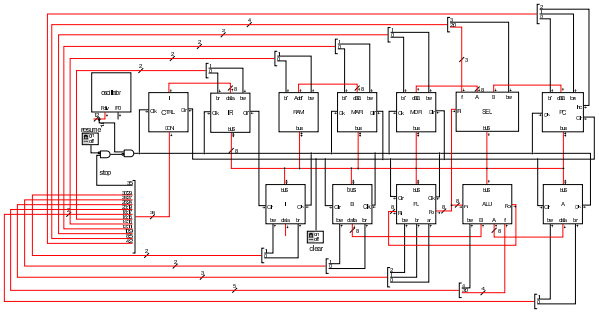


Figura u110.18. Situazione conclusiva del bus dati. Video: *http://www.youtube.com/watch?v=ofPUzdIids8*



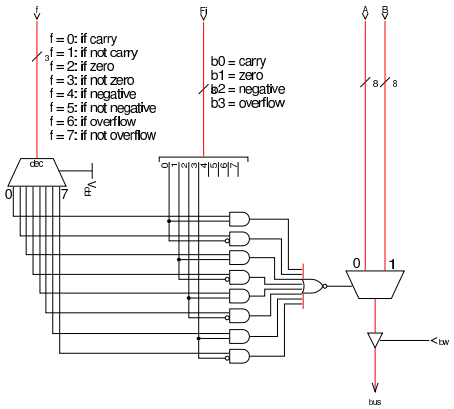
Nella sesta versione della CPU dimostrativa, viene aggiunto un modulo che consente di eseguire delle comparazioni, sulla base dello stato degli indicatori annotati nel registro ***FL***, scegliendo tra due valori che in questo progetto sono costituiti dal contenuto del registro ***PC*** o dal contenuto del registro ***MDR***.

Figura u111.1. Il bus della CPU con l'aggiunta del modulo ***SEL*** per la gestione condizioni.



Il modulo **SEL** riceve due valori dagli ingressi ***A*** e ***B***; dall'ingresso ***Fi*** riceve lo stato degli indicatori, così come emesso dal registro ***FL***. Sulla base della funzione che si seleziona attraverso l'ingresso ***f***, quando è attivo l'ingresso ***bw***, il modulo immette nel bus uno dei due valori disponibili negli ingressi ***A*** e ***B***. Quando la condizione rappresentata dalla funzione si avvera, viene scelto il valore dell'ingresso ***A***, altrimenti si prende ***B***.

Figura u111.2. La struttura interna del modulo **SEL**.



Nel codice che descrive i campi del bus di controllo, si aggiungono quelli seguenti, i quali servono specificatamente a gestire il modulo **SEL**:

```
field sel_f[7:5]={
        if_carry=0,
        if_not_carry=1,
        if_zero=2,
        if_not_zero=3,
        if_negative=4,
        if_not_negative=5,
        if_overflow=6,
        if_not_overflow=7
};
field sel_bw[8];        // SEL --> bus
```

Nell'elenco dei codici operativi si aggiungono istruzioni nuove e lo stesso poi nella descrizione del microcodice:

```
op jump_if_carry {
  map jump_if_carry : 16;        // jump to #nn if carry==1
  +0[7:0]=16;
  operands op_1;
```

```
};
op jump_if_not_carry {
  map jump_if_not_carry : 17;    // jump to #nn if carry==0
  +0[7:0]=17;
  operands op_1;
};
op jump_if_zero {
  map jump_if_zero : 18;         // jump to #nn if zero==1
  +0[7:0]=18;
  operands op_1;
};
op jump_if_not_zero {
  map jump_if_not_zero : 19;     // jump to #nn if zero==0
  +0[7:0]=19;
  operands op_1;
};
op jump_if_negative {
  map jump_if_negative : 20;     // jump to #nn if negative==1
  +0[7:0]=20;
  operands op_1;
};
op jump_if_not_negative {
  map jump_if_not_negative : 21; // jump to #nn if negative==0
  +0[7:0]=21;
  operands op_1;
};
op jump_if_overflow {
  map jump_if_overflow : 22;     // jump to #nn if overflow==1
  +0[7:0]=22;
  operands op_1;
};
op jump_if_not_overflow {
  map jump_if_not_overflow : 23; // jump to #nn if overflow==0
  +0[7:0]=23;
  operands op_1;
};
```

```
begin microcode @ 0
…
jump_if_carry:
    mar_br pc_bw;                       // MAR <-- PC
    pc_Inc;                             // PC++
    mdr_br ram_bw;                      // MDR <-- RAM[mar]
    pc_br sel_f=if_carry sel_bw         // PC = (carry ? MAR : PC)
    ctrl_start ctrl_load;               // CNT <-- 0
//
jump_if_not_carry:
    mar_br pc_bw;                       // MAR <-- PC
    pc_Inc;                             // PC++
    mdr_br ram_bw;                      // MDR <-- RAM[mar]
    pc_br sel_f=if_not_carry sel_bw     // PC = (not_carry ? MAR : PC)
    ctrl_start ctrl_load;               // CNT <-- 0
//
jump_if_zero:
    mar_br pc_bw;                       // MAR <-- PC
    pc_Inc;                             // PC++
    mdr_br ram_bw;                      // MDR <-- RAM[mar]
    pc_br sel_f=if_zero sel_bw          // PC = (zero ? MAR : PC)
    ctrl_start ctrl_load;               // CNT <-- 0
//
jump_if_not_zero:
    mar_br pc_bw;                       // MAR <-- PC
    pc_Inc;                             // PC++
    mdr_br ram_bw;                      // MDR <-- RAM[mar]
    pc_br sel_f=if_not_carry sel_bw     // PC = (not_zero ? MAR : PC)
    ctrl_start ctrl_load;               // CNT <-- 0
//
jump_if_negative:
    mar_br pc_bw;                       // MAR <-- PC
    pc_Inc;                             // PC++
    mdr_br ram_bw;                      // MDR <-- RAM[mar]
    pc_br sel_f=if_negative sel_bw      // PC = (negative ? MAR : PC)
    ctrl_start ctrl_load;               // CNT <-- 0
//
jump_if_not_negative:
    mar_br pc_bw;                       // MAR <-- PC
    pc_Inc;                             // PC++
    mdr_br ram_bw;                      // MDR <-- RAM[mar]
    pc_br sel_f=if_not_negative sel_bw  // PC = (not_negative ? MAR : PC)
    ctrl_start ctrl_load;               // CNT <-- 0
//
jump_if_overflow:
    mar_br pc_bw;                       // MAR <-- PC
    pc_Inc;                             // PC++
    mdr_br ram_bw;                      // MDR <-- RAM[mar]
    pc_br sel_f=if_overflow sel_bw      // PC = (overflow ? MAR : PC)
    ctrl_start ctrl_load;               // CNT <-- 0
//
jump_if_not_overflow:
    mar_br pc_bw;                       // MAR <-- PC
    pc_Inc;                             // PC++
```



```
    mdr_br ram_bw;                      // MDR <-- RAM[mar]
    pc_br sel_f=if_not_overflow sel_bw  // PC = (not_overflow ? MAR : PC)
    ctrl_start ctrl_load;               // CNT <-- 0
…
end
```

Figura u111.6. Corrispondenza con il contenuto della memoria che rappresenta il microcodice (la coppia ***m1*** e ***m2*** dell'unità di controllo).

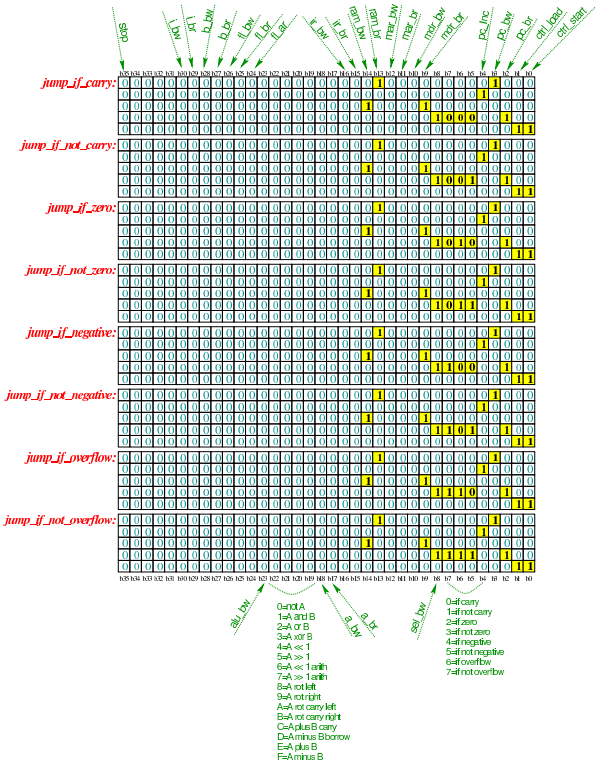


Tabella u111.7. Elenco delle macroistruzioni aggiunte in questa versione della CPU dimostrativa.

Sintassi	Descrizione
`jump_if_carry` ***indirizzo***	Salta all'indirizzo specificato se l'indicatore di riporto è attivo.
`jump_if_not_carry` ***indirizzo***	Salta all'indirizzo specificato se l'indicatore di riporto non è attivo.
`jump_if_zero` ***indirizzo***	Salta all'indirizzo specificato se l'indicatore zero è attivo.
`jump_if_not_zero` ***indirizzo***	Salta all'indirizzo specificato se l'indicatore zero non è attivo.
`jump_if_negative` ***indirizzo***	Salta all'indirizzo specificato se l'indicatore di valore negativo è attivo.
`jump_if_not_negative` ***indirizzo***	Salta all'indirizzo specificato se l'indicatore di valore negativo non è attivo.
`jump_if_overflow` ***indirizzo***	Salta all'indirizzo specificato se l'indicatore di straripamento è attivo.
`jump_if_not_overflow` ***indirizzo***	Salta all'indirizzo specificato se l'indicatore di straripamento non è attivo.

Listato u111.8. Macrocodice per sperimentare l'uso del modulo di selezione, nel quale si crea un ciclo che incrementa una variabile, di una unità alla volta, fino a quando questa variabile contiene il risultato della somma con un'altra. Il file completo che descrive le memorie per Tkgate dovrebbe essere disponibile presso *allegati/circuiti-logici/scpu-sub-f-jump-if.gm* .

```
begin macrocode @ 0
start:
        load_imm #costante_zero
        move_mdr_b
```

```
        load_imm #variabile_x
        move_mdr_a
        add
ciclo:
        jump_if_zero #stop
        load_imm #variabile_y
        move_mdr_a
        load_imm #costante_uno
        move_mdr_b
        add
        move_a_mdr
        store_imm #variabile_y
        load_imm #variabile_x
        move_mdr_a
        load_imm #costante_uno
        move_mdr_b
        sub
        move_a_mdr
        store_imm #variabile_x
        jump #ciclo
stop:
        stop
costante_zero:
        .byte 0
costante_uno:
        .byte 1
variabile_x:
        .byte 3
variabile_y:
        .byte 7
end
```

Figura u111.9. Contenuto della memoria RAM prima dell'esecuzione. Le celle indicate con «xx» hanno un valore indifferente. Al termine dell'esecuzione, la cella di memoria all'indirizzo 23_{16}, corrispondente a ***variabile_y***, contiene il valore 10 ($0A_{16}$).

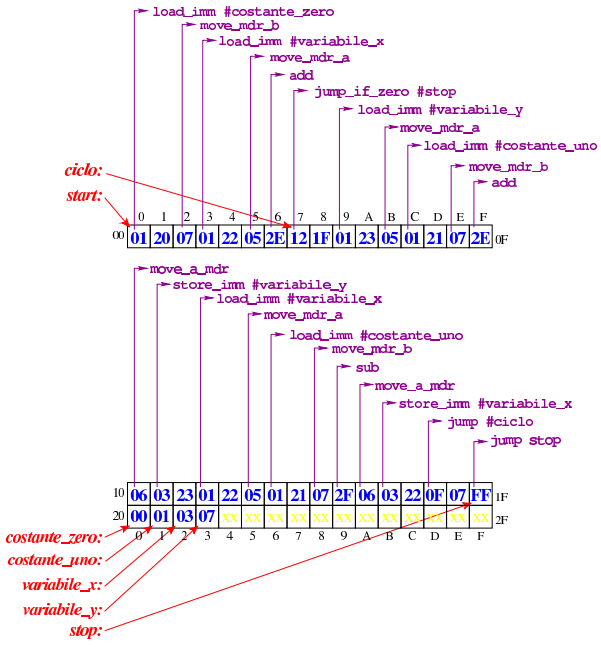
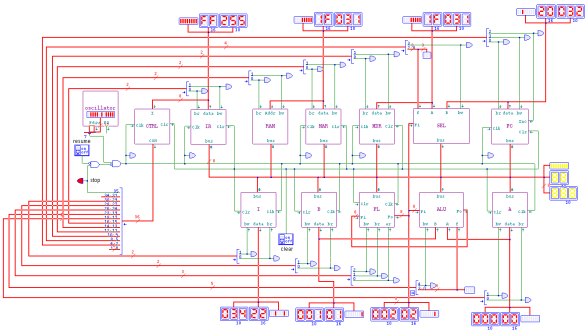
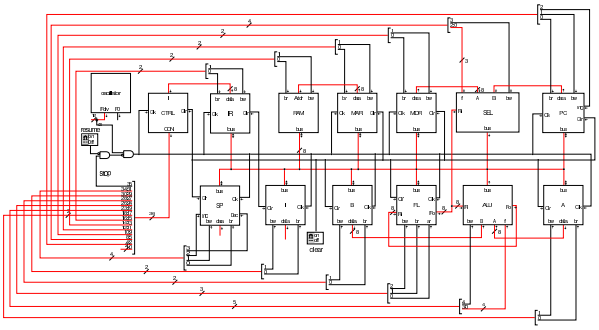


Figura u111.10. Situazione conclusiva del bus dati. Video: *http://www.youtube.com/watch?v=hFoOoGf86t0*



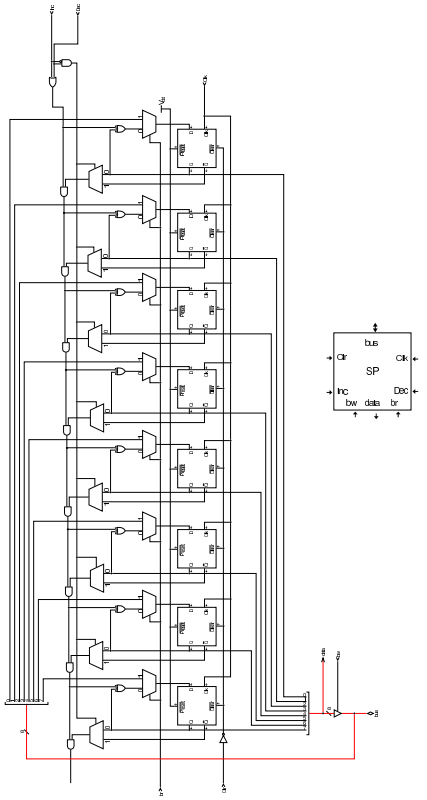
Nella settima versione della CPU dimostrativa, viene aggiunto il registro ***SP*** (*stack pointer*), utilizzato come indice per la pila dei dati. La pila serve principalmente a consentire le chiamate di procedure, tramite istruzioni **`call`** e **`return`**, oltre che a poter salvare e recuperare lo stato dei altri registri.

Figura u112.1. Il bus della CPU con l'aggiunta del registro ***SP*** per la gestione della pila.



Il registro ***SP*** ha due ingressi supplementari, ***Inc*** e ***Dec***, con lo scopo, rispettivamente, di incrementare o diminuire il valore memorizzato nel registro stesso, di una unità.

Figura u112.2. La struttura interna del registro **SP**.



Nel codice che descrive i campi del bus di controllo, si aggiungono quelli seguenti, i quali servono specificatamente a gestire il registro ***SP***:

```
field sp_br[31];        // SP <-- bus
field sp_bw[32];        // SP --> bus
field sp_Inc[33];       // SP++
field sp_Dec[34];       // SP--
```

Nell'elenco dei codici operativi si aggiungono istruzioni nuove e lo stesso poi nella descrizione del microcodice:

```
op call_imm {
  map call_imm : 24;                    // call #nn
  +0[7:0]=24;
  operands op_1;
};
op call_reg {
  map call_reg : 25;                    // call I
  +0[7:0]=25;
  operands op_0;
};
op return {
  map return : 26;                      // return
  +0[7:0]=26;
  operands op_0;
};
op push_mdr {
  map push_mdr : 27;                    // push MDR
  +0[7:0]=27;
  operands op_0;
};
op push_a {
  map push_a : 28;                      // push A
  +0[7:0]=28;
  operands op_0;
};
op push_b {
  map push_b : 29;                      // push B
  +0[7:0]=29;
  operands op_0;
};
op push_fl {
  map push_fl : 30;                     // push FL
  +0[7:0]=30;
  operands op_0;
};
op push_i {
  map push_i : 31;                      // push I
  +0[7:0]=31;
  operands op_0;
};
op pop_mdr {
  map pop_mdr : 48;                    // pop MDR
  +0[7:0]=48;
  operands op_0;
};
op pop_a {
  map pop_a : 49;                      // pop A
  +0[7:0]=49;
  operands op_0;
};
op pop_b {
  map pop_b : 50;                      // pop B
  +0[7:0]=50;
  operands op_0;
};
op pop_fl {
  map pop_fl : 51;                     // pop FL
  +0[7:0]=51;
  operands op_0;
};
op pop_i {
  map pop_i : 52;                      // pop I
  +0[7:0]=52;
  operands op_0;
};
```

```
begin microcode @ 0
…
call_imm:
    mar_br pc_bw;                   // MAR <-- PC
    pc_Inc;                         // PC++
    mdr_br ram_bw;                  // MDR <-- RAM[mar]
    sp_Dec;                         // SC--
    mar_br sp_bw;                   // MAR <-- SP
    ram_br pc_bw;                   // RAM[mar] <-- PC
    pc_br mdr_bw;                   // PC <-- MDR
    ctrl_start ctrl_load;           // CNT <-- 0
//
call_reg:
    sp_Dec;                         // SP--
    mar_br sp_bw;                   // MAR <-- SP
    ram_br pc_bw;                   // RAM[mar] <-- PC
    pc_br i_bw;                     // PC <-- I
    ctrl_start ctrl_load;           // CNT <-- 0
```



```
//
return:
    mar_br sp_bw;                   // MAR <-- SP
    sp_Inc;                         // SP++;
    pc_br ram_bw;                   // PC <-- RAM[mar]
    ctrl_start ctrl_load;           // CNT <-- 0
//
push_mdr:
    sp_Dec;                         // SP--
    mar_br sp_bw;                   // MAR <-- SP
    ram_br mdr_bw;                  // RAM[mar] <-- MDR
    ctrl_start ctrl_load;           // CNT <-- 0
//
push_a:
    sp_Dec;                         // SP--
    mar_br sp_bw;                   // MAR <-- SP
    ram_br a_bw;                    // RAM[mar] <-- A
    ctrl_start ctrl_load;           // CNT <-- 0
//
push_b:
    sp_Dec;                         // SP--
    mar_br sp_bw;                   // MAR <-- SP
    ram_br b_bw;                    // RAM[mar] <-- B
    ctrl_start ctrl_load;           // CNT <-- 0
//
push_fl:
    sp_Dec;                         // SP--
    mar_br sp_bw;                   // MAR <-- SP
    ram_br fl_bw;                   // RAM[mar] <-- FL
    ctrl_start ctrl_load;           // CNT <-- 0
//
push_i:
    sp_Dec;                         // SP--
    mar_br sp_bw;                   // MAR <-- SP
    ram_br i_bw;                    // RAM[mar] <-- I
    ctrl_start ctrl_load;           // CNT <-- 0
//
pop_mdr:
    mar_br sp_bw;                   // MAR <-- SP
    sp_Inc;                         // SP++
    mdr_br ram_bw;                  // MDR <-- RAM[mar]
    ctrl_start ctrl_load;           // CNT <-- 0
//
pop_a:
    mar_br sp_bw;                   // MAR <-- SP
    sp_Inc;                         // SP++
    a_br ram_bw;                    // A <-- RAM[mar]
    ctrl_start ctrl_load;           // CNT <-- 0
//
pop_b:
    mar_br sp_bw;                   // MAR <-- SP
    sp_Inc;                         // SP++
    b_br ram_bw;                    // B <-- RAM[mar]
    ctrl_start ctrl_load;           // CNT <-- 0
//
pop_fl:
    mar_br sp_bw;                   // MAR <-- SP
    sp_Inc;                         // SP++
    fl_br ram_bw;                   // FL <-- RAM[mar]
    ctrl_start ctrl_load;           // CNT <-- 0
//
pop_i:
    mar_br sp_bw;                   // MAR <-- SP
    sp_Inc;                         // SP++
    i_br ram_bw;                    // I <-- RAM[mar]
    ctrl_start ctrl_load;           // CNT <-- 0
…
end
```

Figura u112.6. Corrispondenza con il contenuto della memoria che rappresenta il microcodice (la coppia ***m1*** e ***m2*** dell'unità di controllo).

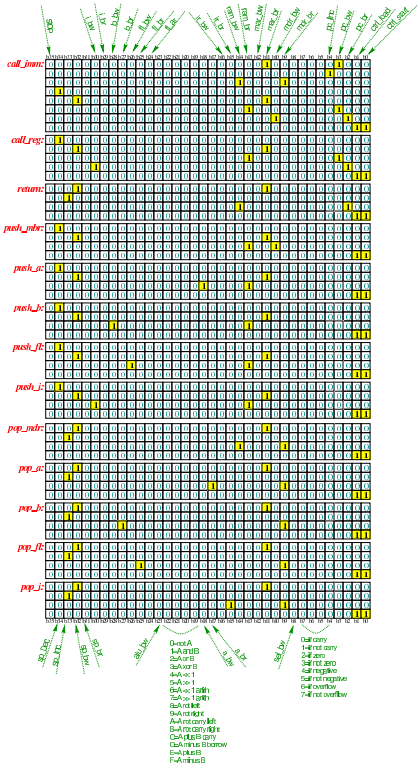


Tabella u112.7. Elenco delle macroistruzioni aggiunte in questa versione della CPU dimostrativa.

Sintassi	Descrizione
`call_imm` ***indirizzo***	Decrementa il registro ***SP*** e annota nella posizione di memoria a cui questo punta l'indirizzo dell'istruzione successiva, quindi aggiorna il registro ***PC*** con l'indirizzo indicato come argomento. In pratica esegue la chiamata di una procedura, annotando nella pila dei dati l'indirizzo di ritorno.
`call_reg`	Decrementa il registro ***SP*** e annota nella posizione di memoria a cui questo punta l'indirizzo dell'istruzione successiva, quindi aggiorna il registro ***PC*** con l'indirizzo contenuto nel registro ***I***. In pratica esegue la chiamata di una procedura, annotando nella pila dei dati l'indirizzo di ritorno.
`return`	Incrementa il registro ***SP*** e preleva l'indirizzo annotato in memoria, corrispondente all'indice della pila, quindi aggiorna il registro ***PC*** con tale valore. In pratica, consente di concludere una chiamata precedente.
`push_mdr`	Inserisce nella pila il valore del registro ***MDR***.
`push_a`	Inserisce nella pila il valore del registro ***A***.
`push_b`	Inserisce nella pila il valore del registro ***B***.
`push_fl`	Inserisce nella pila il valore del registro ***FL***.
`push_i`	Inserisce nella pila il valore del registro ***I***.
`pop_mdr`	Recupera dalla pila il valore del registro ***MDR***.
`pop_a`	Recupera dalla pila il valore del registro ***A***.

Sintassi	Descrizione
`pop_b`	Recupera dalla pila il valore del registro ***B***.
`pop_fl`	Recupera dalla pila il valore del registro ***FL***.
`pop_i`	Recupera dalla pila il valore del registro ***I***.

Istruzioni «push» e «pop»

Listato u112.8. Macrocodice per sperimentare l'uso delle istruzioni di inserimento ed estrazione di valori dalla pila dei dati: in questo caso, viene inserito nella pila il valore contenuto nel registro ***A*** e poi ripescato, ma nel registro ***B***. Il file completo che descrive le memorie per Tkgate dovrebbe essere disponibile presso *allegati/circuiti-logici/scpu-sub-g-push-pop.gm* .

```
begin macrocode @ 0
start:
        load_imm #sp_bottom
        move_mdr_sp
        load_imm #data_0
        move_mdr_a
        push_a
        pop_b

stop:
        stop
sp_bottom:
        .byte 0x10
data_0:
        .byte 0xCC
end
```

Figura u112.9. Contenuto della memoria RAM: la cella nella posizione $0F_{16}$ viene scritta dall'istruzione **push_a**. Le celle indicate con «xx» hanno un valore indifferente.

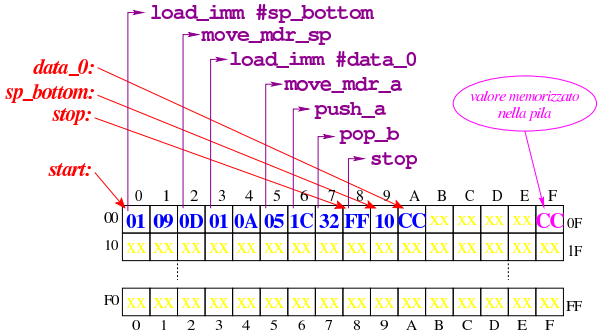
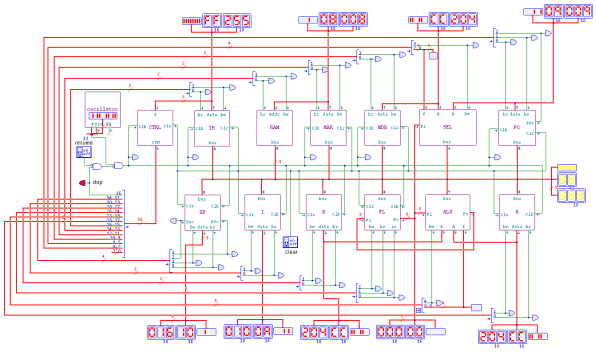


Figura u112.10. Situazione conclusiva del bus dati. Video: *http://www.youtube.com/watch?v=vAdVww3lD7I*



Istruzioni «call» e «return»

Listato u112.11. Macrocodice per sperimentare l'uso delle istruzioni di chiamata e ritorno dalle procedure. Il file completo che descrive le memorie per Tkgate dovrebbe essere disponibile presso *allegati/circuiti-logici/scpu-sub-g-call-return.gm* .

```
begin macrocode @ 0
start:
        load_imm #sp_bottom
        move_mdr_sp
        call_imm #elabora
        move_a_mdr
        move_mdr_b
        jump #stop
elabora:
        load_imm #data_0
        move_mdr_a
        load_imm #data_1
        move_mdr_b
        add
        return
stop:
        stop
sp_bottom:
        .byte 0x20
data_0:
        .byte 0x0A
data_1:
        .byte 0x0B
end
```

Figura u112.12. Contenuto della memoria RAM: la cella nella posizione $1F_{16}$ viene scritta dall'istruzione **`call_imm`**. Le celle indicate con «xx» hanno un valore indifferente.

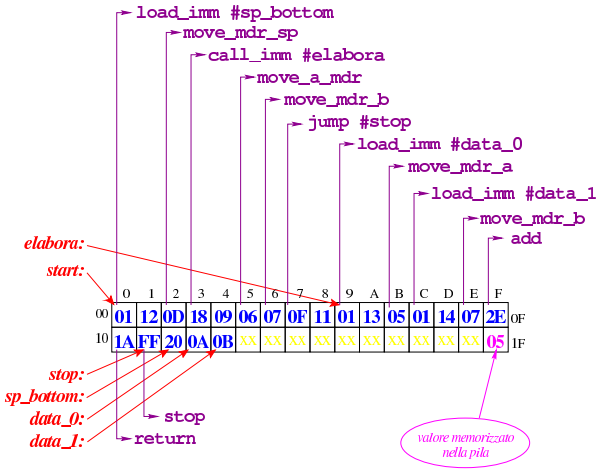
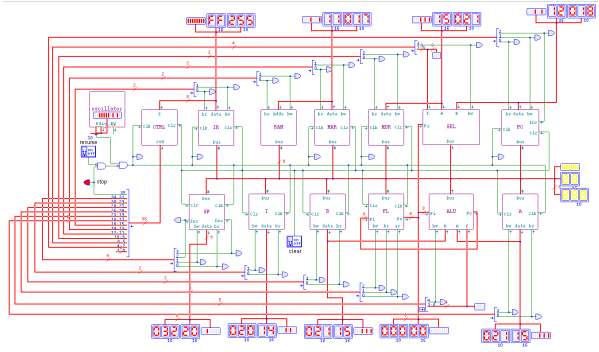


Figura u112.13. Situazione conclusiva del bus dati. Video: *http://www.youtube.com/watch?v=nWdXMvegkjc*



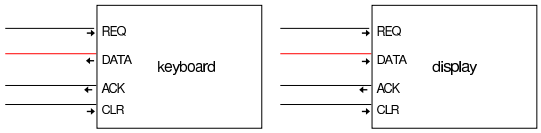
Versione H: I/O

La versione precedente è abbastanza completa per dimostrare le funzionalità principali di una CPU; in questa versione si passa a estendere il progetto iniziale, per consentire il collegamento con dispositivi di input-output. Per fare questo viene aggiunto un bus dati e un bus indirizzi per l'I/O (input-output), inoltre, si aggiungono alcune linee nel bus di controllo (***CON***), da utilizzare per i componenti che costituiscono l'interfaccia con i dispositivi di I/O. Il progetto della CPU dimostrativa si basa su Tkgate e nelle sezioni successive si mostra anche il codice utilizzato per realizzare i dispositivi in questione con questo simulatore.

Generalizzazione della comunicazione con i dispositivi

I dispositivi di I/O sono normalmente asincroni rispetto alla CPU (nel senso che non sono regolati dal clock che amministra la CPU), pertanto si rende necessario un protocollo per richiedere un'azione al dispositivo e per recepirne il risultato, tipico dei componenti asincroni. I dispositivi di I/O utilizzati in questo progetto dimostrativo hanno esteriormente le connessioni che si possono vedere nella figura successiva.

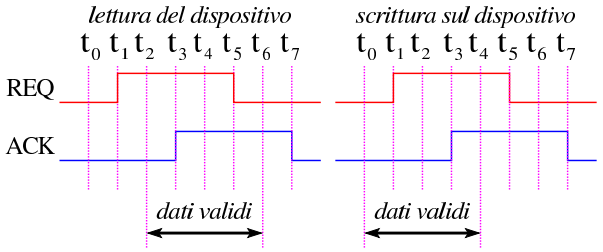
Figura u113.1. Un dispositivo di input e un dispositivo di output.



L'ingresso o l'uscita ***DATA*** consente di movimentare l'informazione che serve al dispositivo o che viene fornita dallo stesso. Gli ingressi ***REQ*** (*request*) e ***ACK*** (*acknowledge*) servono a negoziare il movimento dell'informazione (l'ingresso ***CLR*** serve ad azzerare il dispositivo). Vanno fatti due esempi, a seconda che si debba leggere un'informazione dal dispositivo, oppure che vi si voglia scrivere.

Per comunicare con un **dispositivo di input**, dal quale si deve leggere un'informazione, si comincia attivando l'ingresso ***REQ*** (t_1), con il quale si intende richiedere il dato. Il dispositivo riceve la richiesta e predispone l'uscita ***DATA*** con l'informazione (t_2); subito dopo, attiva l'uscita ***ACK*** (t_3). A questo punto, trovando attiva l'uscita ***ACK*** si può leggere il valore dall'uscita ***DATA*** (t_4) e al termine si può disattivare l'ingresso ***REQ*** (t_5). Il dispositivo, osservando la disattivazione dell'ingresso ***REQ*** sa che l'informazione è stata recepita, quindi smettere di fornire l'informazione richiesta (t_6) e disabilita a sua volta l'uscita ***ACK*** (t_7).

Figura u113.2. Fasi della lettura da un dispositivo di input e della scrittura su un dispositivo di output.



Per comunicare con un **dispositivo di output**, nel quale si deve scrivere un'informazione, si comincia fornendo l'informazione sull'ingresso ***DATA*** (t_0); subito dopo, si attiva l'ingresso ***REQ*** (t_1) per informare il dispositivo della disponibilità dell'informazione. Quindi il dispositivo recepisce l'informazione (t_2) e poi dà conferma attivando l'uscita ***ACK*** (t_3). Ricevendo la conferma, non è più necessario trattenere l'informazione disponibile nell'ingresso ***DATA*** (t_4), quindi viene disattivato l'ingresso ***REQ*** (t_5) e dopo di questo il dispositivo disattiva l'uscita ***ACK*** (t_7) concludendo l'operazione.

Realizzazione dei dispositivi di I/O

In questo progetto, per il momento, si realizzano soltanto due dispositivi di I/O: una tastiera e uno schermo. Dato che il progetto è sviluppato con Tkgate, i dispositivi vanno dichiarati attraverso codice Tcl/Tk. Il codice che viene mostrato qui è stato ottenuto modificando un dispositivo di esempio che fa parte della distribuzione di Tkgate.

Listato u113.3. File '`share/tkgate/vpd/kbd.tcl`' per simulare l'input di una tastiera. Il programma si limita a mostrare una piccola finestra vuota, selezionando la quale è possibile digitare da tastiera qualcosa che deve essere recepito dal dispositivo virtuale corrispondente.

```
VPD::register KBD
VPD::allow KBD::post

namespace eval KBD {
    # Dichiarazione delle variabili pubbliche. $kbd_w è un array
    # di cui si utilizza solo l'elemento $n, il quale identifica
    # univocamente l'istanza dell'interfaccia in funzione.
    variable kbd_w
    variable KD
    # Funzione richiesta da Tkgate per creare l'interfaccia.
    proc post {n} {
        variable kbd_w
        # Crea una finestra e salva l'oggetto in un elemento dell'array
        # $kbd_w.
        set kbd_w($n) [VPD::createWindow "KBD $n" -shutdowncommand "KBD::unpost $n"]
        # Collega la digitazione della tastiera, relativa all'oggetto
        # rappresentato da $kbd_w($n), alla funzione sendChar.
        bind $kbd_w($n) <KeyPress> "KBD::sendChar $n \"%A\""
        # Apre un canale di scrittura, denominato «KD».
        if {[info exists ::tkgate_isInitialized]} {
            VPD::outsignal $n.KD KBD::KD($n)
        }
    }
    # Funzione che recepisce la digitazione e la immette nel canale
    # denominato «KD», relativo all'istanza attuale dell'interfaccia.
    proc sendChar {n key} {
        variable KD
        if { [string length $key ] == 1 } {
            binary scan $key c c
            set KBD::KD($n) $c
        }
    }
    # Funzione richiesta da Tkgate per distruggere l'interfaccia.
    proc unpost {n} {
        variable kbd_w
        destroy $kbd_w($n)
        unset kbd_w($n)
    }
}
```

Listato u113.4. File '`share/tkgate/vpd/scr.tcl`' per simulare l'output su schermo a caratteri. Il programma mostra una finestra sulla quale possono poi apparire i caratteri trasmessi. Nel codice si fa riferimento al file '`textcurs.b`' che è già disponibile nella distribuzione di Tkgate.

```
image create bitmap txtcurs -file "$bd/txtcurs.b"

VPD::register SCR
VPD::allow SCR::post
VPD::allow SCR::data

namespace eval SCR {
    # Dichiarazione delle variabili pubbliche: si tratta di array
    # dei quali si utilizza solo l'elemento $n, il quale identifica
    # univocamente l'istanza dell'interfaccia in funzione.
    variable scr_w
    variable scr_pos
    # Funzione richiesta da Tkgate per creare l'interfaccia.
    proc post {n} {
        variable scr_w
        variable scr_pos
        # Crea una finestra e salva l'oggetto in un elemento dell'array
        # $scr_w.
        set scr_w($n) [VPD::createWindow "SCR $n" -shutdowncommand "SCR::unpost $n"]
        # Per maggiore comodità, copia il riferimento all'oggetto nella
        # variabile locale $w e in seguito fa riferimento all'oggetto
        # attraverso questa seconda variabile.
        set w $scr_w($n)
        text $w.txt -state disabled
        pack $w.txt
        # Mette il cursore alla fine del testo visualizzato.
        $w.txt image create end -image txtcurs
        # Apre un canale di lettura, denominato «RD», associandolo
        # alla funzione «data».
        if {[info exists ::tkgate_isInitialized]} {
            VPD::insignal $n.RD -command "SCR::data $n" -format %d
        }
        # Azzera il contatore che tiene conto dei caratteri visualizzati
        # sullo schermo.
        set scr_pos($n) 0
    }
    # Funzione richiesta da Tkgate per distruggere l'interfaccia.
    proc unpost {n} {
        variable scr_w
        destroy $scr_w($n)
        unset scr_w($n)
    }
    # Funzione usata per recepire i dati da visualizzare.
    proc data {n c} {
        variable scr_w
        variable scr_pos
        # Per maggiore comodità, copia il riferimento all'oggetto che
        # rappresenta l'interfaccia nella variabile $w.
        set w $scr_w($n)
        catch {
            # La variabile $c contiene il carattere da visualizzare.
            if { $c == 7 } {
                # BEL
                bell
                return
            } elseif { $c == 127 || $c == 8 } {
                # DEL | BS
                if { $scr_pos($n) > 0 } {
                    # Cancella l'ultimo carattere visualizzato, ma solo
                    # se il contatore dei caratteri è maggiore di zero,
                    # altrimenti sparirebbe il cursore e la
                    # visualizzazione verrebbe collocata in un'area
                    # non visibile dello schermo.
                    $w.txt configure -state normal
                    $w.txt delete "end - 3 chars"
                    $w.txt see end
                    $w.txt configure -state disabled
                    set scr_pos($n) [expr {$scr_pos($n) - 1}]
                }
                return
            } elseif { $c == 13 } {
                # CR viene trasformato in LF.
                set c 10
            }
            # Converte il numero del carattere in un simbolo
            # visualizzabile.
            set x [format %c $c]
            # Visualizza il simbolo.
            $w.txt configure -state normal
            $w.txt insert "end - 2 chars" $x
            $w.txt see end
            $w.txt configure -state disabled
            # Aggiorna il contatore dei caratteri visualizzati.
            set scr_pos($n) [expr {$scr_pos($n) + 1}]
        }
    }
}
```

I due programmi Tcl/Tk servono a fornire due moduli software, a cui poi si fa riferimento attraverso del codice Verilog. Nei listati successivi si vedono i moduli **`keyboard`** e **`display`** che graficamente si mostrano esattamente come nella figura u113.1.

Listato u113.5. Codice Verilog per il modulo **`keyboard`**.

```
module keyboard(DATA, REQ, ACK, CLR);
output ACK;
output [7:0] DATA;
input REQ;
input CLR;
reg ready;
reg [7:0] key;

  initial
    begin
```

```
      ready = 0;
      key = 0;
    end

  always
    begin
      @(posedge CLR)
      ready = 0;
      key = 0;
    end

  initial $tkg$post("KBD","%m");

  always
    begin
      @ (posedge REQ);
      # 5;
      key = $tkg$recv("%m.KD");
      # 5;
      ready = 1'b1;
      # 5;
      @ (negedge REQ);
      # 5;
      ready = 1'b0;
    end

  assign DATA = key;
  assign ACK = ready;

endmodule
```

Listato u113.6. Codice Verilog per il modulo **`display`**.

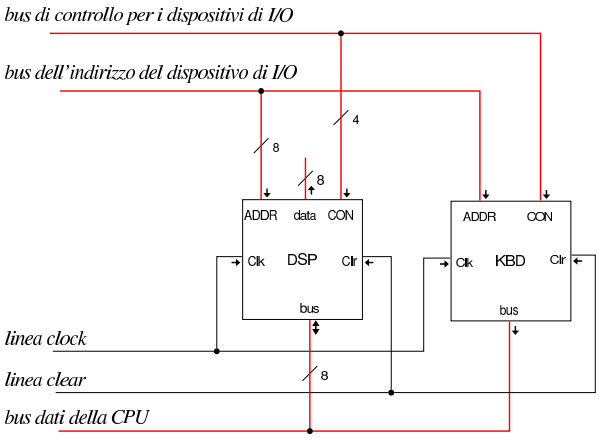
```
module display(DATA, REQ, ACK, CLR);
output ACK;
input [7:0] DATA;
input REQ;
input CLR;
reg ready;

  initial
    begin
      ready = 0;
    end

  initial $tkg$post("SCR","%m");

  always
    begin
      @(posedge CLR)
      ready = 0;
    end

  always
    begin
      @(posedge REQ);
      # 5;
      $tkg$send("%m.RD",DATA);
      # 5;
      ready = 1'b1;
      # 5;
      @(negedge REQ);
      # 5;
      ready = 1'b0;
    end

  assign ACK = ready;

endmodule
```

Fino a qui, i dispositivi descritti funzionano in modo asincrono, ma per essere utilizzati devono essere adattati per poter funzionare in modo sincrono.

Aspetto e funzionamento esteriore delle interfacce sincrone

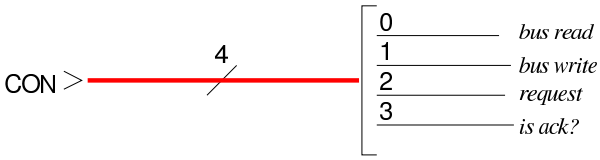
Le interfacce sincrone dei dispositivi di I/O devono potersi collegare al bus della CPU come gli altri moduli già presenti; tuttavia, per semplificare il cablaggio, invece di disporre di linee di controllo personali, viene creato un bus aggiuntivo, in cui indicare l'indirizzo del modulo a cui si vuole fare riferimento.

Figura u113.7. Connessione dei moduli di I/O.



Come si vede dal disegno, il bus connesso agli ingressi ***ADDR*** serve a selezionare il dispositivo, mentre il bus connesso agli ingressi ***CON*** serve a uniformare le linee di controllo per tutti i moduli di I/O. In pratica, viene mostrato in seguito che questi due bus aggiuntivi provengono dallo stesso bus di controllo complessivo.

Gli ingressi ***CON*** sono uniformi, ma ogni modulo utilizza solo ciò che gli serve, ignorando il resto:

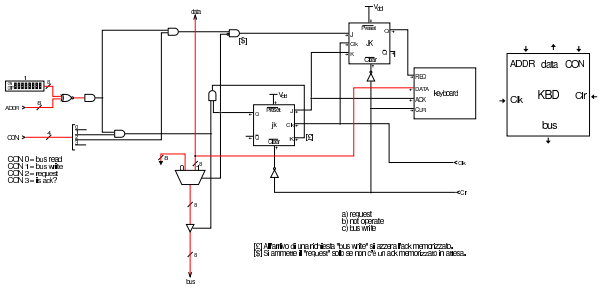


Le linee ***bus read*** e ***bus write*** sono le stesse degli altri moduli già descritti, riferendosi all'ordine di leggere o di scrivere sul bus della CPU. La linea ***request*** serve a richiedere l'operazione di lettura o scrittura del dispositivo, ma senza la necessità di mantenerla attiva come nel protocollo di comunicazione asincrona. La linea ***is ack?*** serve a ottenere, in qualche modo, l'informazione sul fatto che sia stata ricevuta la conferma da parte del dispositivo.

Interfaccia sincrona della tastiera

Il modulo di interfaccia della tastiera utilizza solo due delle quattro linee di controllo: ***bus write*** e ***request***.

Figura u113.9. Modulo **`KBD`** che collega la tastiera al bus della CPU.



Per comunicare con il modulo della tastiera, è necessario inizialmente un segnale ***request*** che all'arrivo dell'impulso di clock viene memorizzato nel flip-flop JK superiore, attivandolo; tale flip-flop ha il compito di trasferire e fissare tale valore sull'ingresso ***REQ*** del dispositivo. Il secondo flip-flop JK, in posizione centrale, ha invece il compito di memorizzare l'esito emesso dall'uscita ***ACK*** e, se tale valore risulta memorizzato, non permette la ricezione di una nuova richiesta. Dopo la ricezione di un segnale ***request*** acquisito correttamente, nella migliore delle ipotesi, il dispositivo ha già accumulato

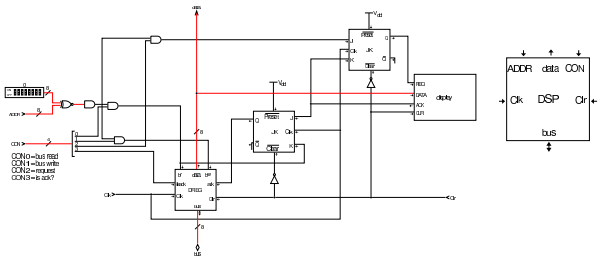
un carattere digitato da tastiera e risponde quasi subito mettendo tale valore nella sua uscita ***DATA*** e poi attivando la sua uscita ***ACK***. Al secondo impulso di clock, il flip-flop che manteneva attivo l'ingresso ***REQ*** si azzera e invece si attiva il secondo flip-flop al centro del disegno; tuttavia, il dispositivo mantiene il valore dell'uscita ***DATA***. A questo punto si riceve il segnale ***bus write*** che consente di immettere il valore ricevuto dalla tastiera nel bus della CPU, azzerando contestualmente il flip-flop centrale. Se invece non si riesce a ottenere un carattere dalla tastiera, nel tempo stabilito, si ottiene il valore nullo (00_{16}), dato che manca l'attivazione del flip-flop che conserva lo stato di ***ACK***.

A livello di macrocodice, se la lettura della tastiera produce un carattere nullo, significa che non c'è alcun carattere pronto e occorre ripetere la lettura.

Interfaccia sincrona dello schermo

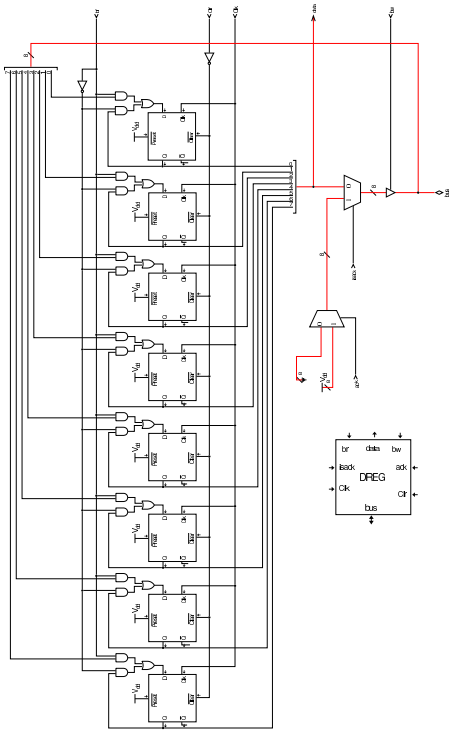
Il modulo **DSP** utilizza tutte le linee dell'ingresso di controllo, in quanto deve poter leggere dal bus della CPU, quando si vuole visualizzare un carattere sullo schermo, ma deve anche poter scrivere sul bus, per fornire un codice di conferma della riuscita della visualizzazione.

Figura u113.10. Modulo **DSP** che collega lo schermo al bus della CPU.



Il modulo **DSP** utilizza un registro modificato che serve principalmente per memorizzare il carattere da rappresentare sullo schermo; tuttavia, quando si attiva il suo ingresso ***isack***, può immettere nel bus della CPU l'esito della visualizzazione: 00_{16} vuol dire che questa non è ancora stata confermata, mentre FF_{16} indica una rappresentazione avvenuta correttamente.

Figura u113.11. Registro **DREG** che accumula il carattere da visualizzare.



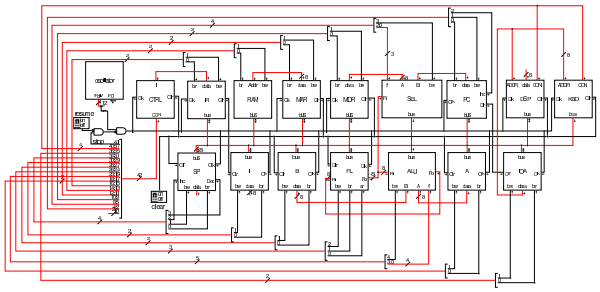
Per visualizzare un carattere sullo schermo, si comincia attivando la linea ***bus read***: all'arrivo dell'impulso di clock il registro accumula il carattere leggendolo dal bus della CPU, mentre il flip-flop JK centrale si azzera, azzerando così l'ingresso ***ack*** del registro ***DREG***. Quindi deve essere attivata la linea ***request*** e all'arrivo dell'impulso di clock questo valore viene memorizzato nel flip-flop JK superiore, il quale attiva così l'ingresso ***REQ*** del dispositivo. A quel punto, ritardo di propagazione permettendo, il dispositivo mostra il carattere già presente nel suo ingresso ***DATA*** (proveniente dal registro ***DREG*** e a un certo punto risponde attivando la sua uscita ***ACK***. Quando l'uscita ***ACK*** del dispositivo si attiva e sopraggiunge un impulso di clock, il registro che manteneva il segnale ***REQ*** si azzera, mentre si attiva il flip-flop JK centrale, attivando di conseguenza l'ingresso ***ack*** del registro ***DREG***.

Per la visualizzazione di un carattere, sono sufficienti i due cicli di clock iniziali, ma per verificare che la visualizzazione sia avvenuta effettivamente, occorre intervenire nuovamente con un'interrogazione. In tal caso si attiva la linea ***is ack?*** e ***bus write***, in modo da immettere nel bus della CPU il valore che può essere 00_{16} o FF_{16}, a seconda del fatto che non sia ancora stata ottenuta la conferma oppure che invece questa ci sia stata.

Il bus della CPU con i dispositivi di I/O

Nel bus della CPU, oltre ai moduli dei dispositivi di I/O, si aggiunge un registro, denominato ***IOA***, con lo scopo di conservare l'indirizzo del dispositivo di I/O con il quale si vuole comunicare.

Figura u113.12. Il bus della CPU con l'aggiunta del registro ***IOA*** e dei moduli di I/O.



Dal disegno si può vedere che il bus di controllo complessivo è stato modificato, inserendo delle linee per il controllo del registro ***IOA*** e le quattro linee necessarie a controllare i dispositivi di I/O, spostando di conseguenza la linea usata per fermare il segnale di clock. Pertanto, nel codice della dichiarazione delle memorie e in quello che descrive i campi del bus di controllo, si apportano le modifiche seguenti:

```
map       bank[7:0]   ctrl.m0;
microcode bank[31:0]  ctrl.m1;
microcode bank[41:32] ctrl.m2;
macrocode bank[7:0]   ram.m3;
…
field ioa_br[35];        // IOA <-- bus
field ioa_bw[36];        // IOA --> bus
field io_br[37];         // I/O <-- bus
field io_bw[38];         // I/O --> bus
field io_req[39];        // I/O request
field io_isack[40];      // I/O is ack?
field stop[41];          // stop clock
```

Nell'elenco dei codici operativi si aggiungono istruzioni nuove e lo stesso poi nella descrizione del microcodice; in particolare, si ammettono macroistruzioni che richiedono due argomenti:

```
operands op_2 {
    //
    // [........][mmmmmmmm][nnnnnnnn]
    //
    #1,#2 = { +1=#1[7:0]; +2=#2[15:8]; };
};
…
op in {
  map in : 48;                    // read input from I/O bus
  +0[7:0]=48;
  operands op_1;
};
op out {
  map out : 49;                   // write output to I/O bus
  +0[7:0]=49;
  operands op_1;
};
op io_is_ack {
  map io_is_ack : 50;
  +0[7:0]=50;
  operands op_2;
};
```

```
begin microcode @ 0
…
in:
    mar_br pc_bw;               // MAR <-- PC
    pc_Inc;                     // PC++
    mdr_br ram_bw;              // MDR <-- RAM[mar]
    ioa_br mdr_bw;              // IOA <-- MDR
    io_req;                     //
    io_br;                      // non fa alcunché
    a_br io_bw;                 // A <-- I/O
    ctrl_start ctrl_load;       // CNT <-- 0

out:
    mar_br pc_bw;               // MAR <-- PC
    pc_Inc;                     // PC++
    mdr_br ram_bw;              // MDR <-- RAM[mar]
    ioa_br mdr_bw;              // IOA <-- MDR
    io_br a_bw;                 // IO <-- A
    io_req;                     //
    ctrl_start ctrl_load;       // CNT <-- 0

io_is_ack:
    mar_br pc_bw;               // MAR <-- PC
```



```
    pc_Inc;                     // PC++
    mdr_br ram_bw;              // MDR <-- RAM[mar]
    ioa_br mdr_bw;              // IOA <-- MDR
    //
    mar_br pc_bw;               // MAR <-- PC
    pc_Inc;                     // PC++
    mdr_br ram_bw;              // MDR <-- RAM[mar]
    a_br io_bw io_isack;        // A <-- I/O is ack
    a_br alu_f=not_a alu_bw fl_ar;  // A <-- NOT A
    a_br alu_f=not_a alu_bw fl_ar;  // A <-- NOT A
    pc_br sel_f=if_not_zero sel_bw  // PC = (not_zero ? MAR : PC)
    ctrl_start ctrl_load;       // CNT <-- 0
…
end
```

Figura u113.16. Corrispondenza con il contenuto della memoria che rappresenta il microcodice (la coppia ***m1*** e ***m2*** dell'unità di controllo).

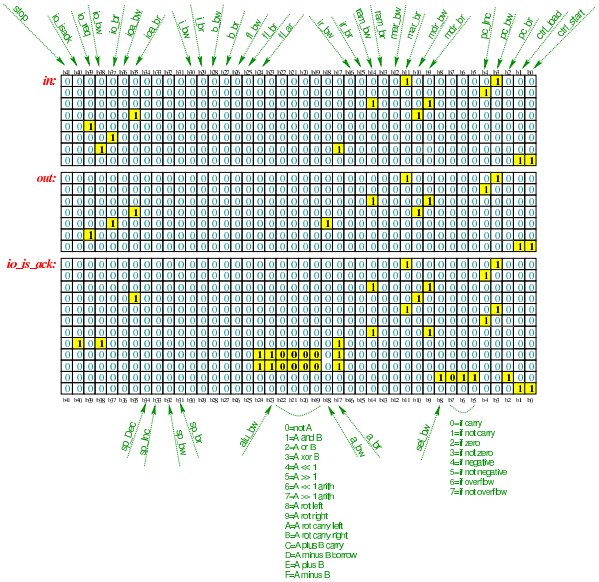


Tabella u113.17. Elenco delle macroistruzioni aggiunte in questa versione della CPU dimostrativa.

Sintassi	Descrizione
`in` ***indirizzo_io***	Legge un byte dal dispositivo a cui corrisponde l'indirizzo.
`out` ***indirizzo_io***	Scrive un byte sul dispositivo a cui corrisponde l'indirizzo.
`io_is_ack` ***indirizzo_io indirizzo***	Legge dal dispositivo indicato dal primo argomento, un codice che consente di verificare il ricevimento della conferma (*acknowledge*); se l'esito è valido, salta all'indirizzo indicato come secondo argomento.

Istruzione «out»

In questa sezione viene mostrato l'uso della macroistruzione '**`out`**', per visualizzare un carattere attraverso il dispositivo **`DSP`**. Nel listato successivo si mostra l'uso di '**`out`**' e poi anche '**`io_is_ack`**' per verificare che il carattere da visualizzare sia stato effettivamente mostrato. Il programma si limita a mostrare la lettera «H», ripetutamente, senza fermarsi.

Listato u113.18. Macrocodice per sperimentare l'istruzione **`out`** e **`io_is_ack`**: si vuole visualizzare la lettera «H», ripetutamente, controllando ogni volta il completamento dell'operazione. Il file completo che descrive le memorie per Tkgate dovrebbe essere disponibile presso *allegati/circuiti-logici/scpu-sub-h-out.gm* .

```
begin macrocode @ 0
start:
        load_imm #carattere
```

```
        move_mdr_a
        out 0                   // display
check_ack:
        io_is_ack 0, #start
        jump #check_ack
stop:
        stop
carattere:
        .byte 0x48              // 'H'
end
```

Anche per questo esempio è disponibile un video: *http://www.youtube.com/watch?v=S9XqmTMYAj4* .

Istruzione «in»

In questa sezione viene mostrato l'uso della macroistruzione '**in**', per recepire la digitazione da tastiera, attraverso il modulo **KBD**. Nel listato successivo si usa anche l'istruzione '**out**', usata per riemettere il carattere ricevuto.

Listato u113.19. Macrocodice per sperimentare l'istruzione **in**: si vuole recepire la digitazione da tastiera, la quale viene riemessa attraverso l'istruzione **out** sul dispositivo **DSP**. Il file completo che descrive le memorie per Tkgate dovrebbe essere disponibile presso *allegati/circuiti-logici/scpu-sub-h-in.gm* .

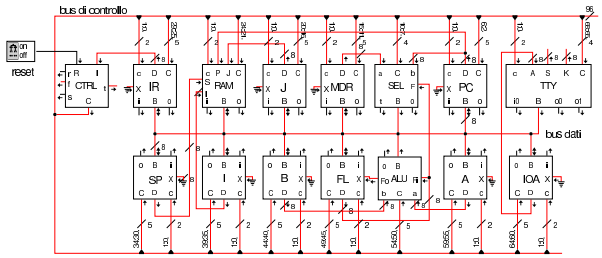
```
begin macrocode @ 0
start:
        in 1
        not
        not
        jump_if_zero #start
        out 0
        jump #start
stop:
        stop
end
```

Video: *http://www.youtube.com/watch?v=JhGoQhssWQM* .

Versione I: ottimizzazione

Viene proposta una ristrutturazione della CPU dimostrativa sviluppata fino a questo punto, per riordinarne e semplificarne il funzionamento. Si parte dalla realizzazione uniforme dei registri, raccogliendo dove possibile le linee di controllo, per arrivare a un'ottimizzazione del funzionamento, evitando cicli di clock inutili.

Figura u114.1. CPU dimostrativa, versione «I».



Registri uniformi

I registri della nuova versione della CPU dimostrativa, hanno tutti la possibilità di incrementare o ridurre il valore che contengono, di una unità; inoltre, hanno la possibilità di leggere un dato dal bus (***B***) oppure da un ingresso ausiliario (***X***). Per poter monitorare la loro attività, dispongono di due uscite a cui si potrebbero collegare dei led, i quali si attivano in corrispondenza di una fase di lettura o di scrittura.

Figura u114.2. Aspetto esterno del registro generalizzato.

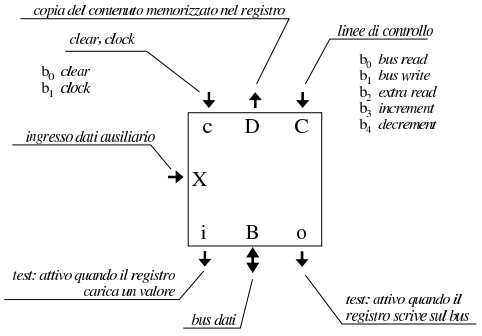
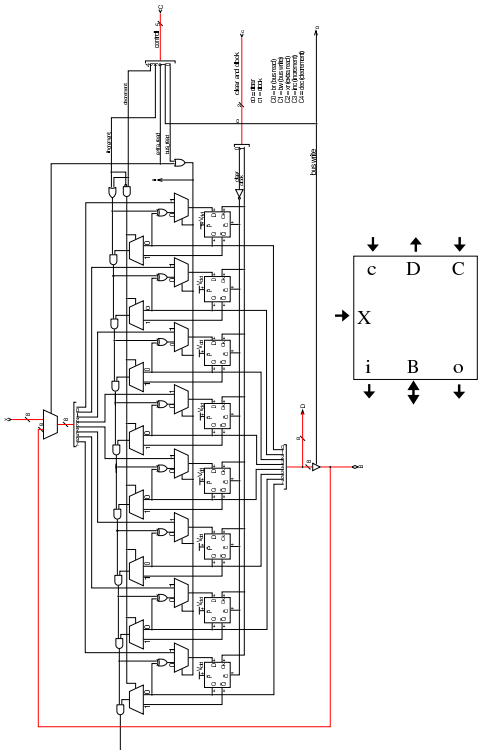


Figura u114.3. Schema interno del registro generalizzato.

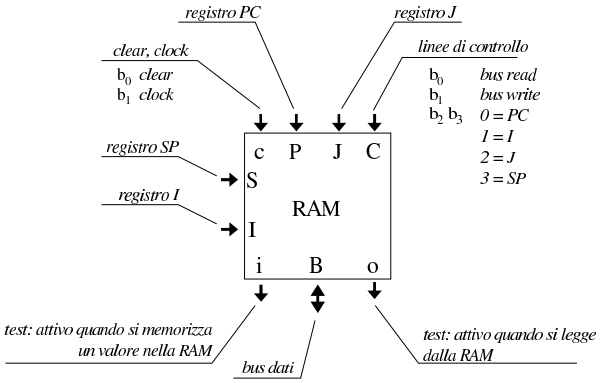


RAM

«

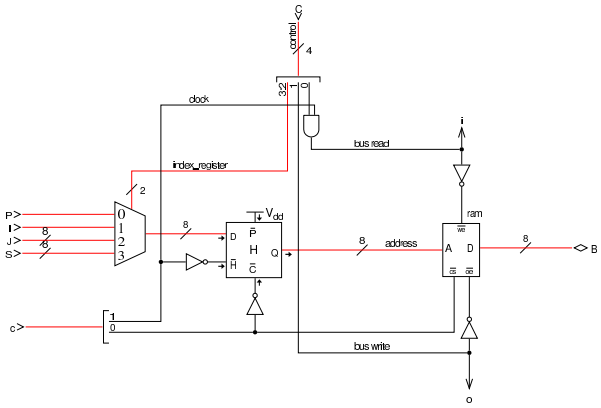
Il modulo **RAM** può ricevere l'indirizzo, direttamente dai registri ***PC***, ***SP***, ***I*** e ***J***, senza mediazioni; pertanto, il registro ***MAR*** utilizzato fino alla versione precedente è stato rimosso. La scelta del registro da cui leggere l'indirizzo dipende dal codice contenuto nel gruppo di linee dell'ingresso ***C***.

Figura u114.4. Aspetto esterno del modulo **RAM**.



Lo schema interno del modulo **RAM** cambia sostanzialmente, per consentire di utilizzare l'indirizzo proveniente dal registro selezionato, ma solo allo stato in cui questo dato risulta valido. Nello schema si vede l'aggiunta di un modulo, denominato ***H***, corrispondente a un registro controllato da un ingresso di abilitazione. Pertanto, tale registro non reagisce alla variazione dell'impulso di clock, ma si limita a mantenere memorizzato un valore per tutto il tempo in cui l'ingresso ***H'*** risulta azzerato.

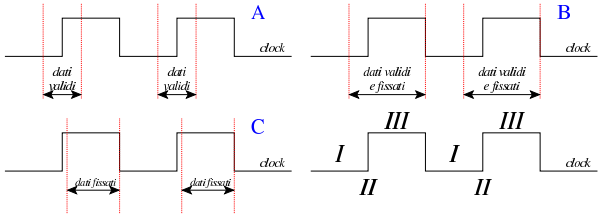
Figura u114.5. Schema interno del modulo **RAM**.



In questa versione della CPU, durante un ciclo di clock, l'indirizzo che serve a individuare la cella di memoria a cui si è interessati, può non essere stabile, a causa di vari fattori. Prima di tutto, l'indirizzo viene scelto attraverso un multiplatore, pescandolo da quattro registri diversi, ma questa selezione avviene all'inizio della fase «I» della figura successiva; pertanto, in questa prima fase l'informazione subisce un cambiamento nella maggior parte dei casi. Inoltre, quando scatta il segnale di clock, passando da zero a uno, il registro da cui si attinge l'informazione dell'indirizzo potrebbe essere indotto ad aggiornarsi, in preparazione della fase successiva. Quindi, l'informazione valida sull'indirizzo da utilizzare per la memoria RAM appare a cavallo della variazione positiva del segnale di clock (fase «II»). Tuttavia, quando si richiede di scrivere nella RAM un valore, la RAM stessa ha bisogno di disporre dell'indirizzo per un certo tempo, durante il quale questo indirizzo non deve cambiare; pertanto, si utilizza il registro ***H*** che è trasparente quando il segnale di clock è a zero, mentre blocca il proprio valore quando il segnale di clock è attivo. Per questo, la RAM viene abilitata a ricevere la richiesta di lettura o di scrittura soltanto durante il periodo attivo del segnale di clock (fase «III»). Quando si tratta invece di leggere dalla RAM, è sufficiente che la RAM abbia avuto il tempo di fornire il dato corrispondente all'indirizzo selezionato, nel momento in cui l'informazione viene poi attinta dal bus da un altro registro.

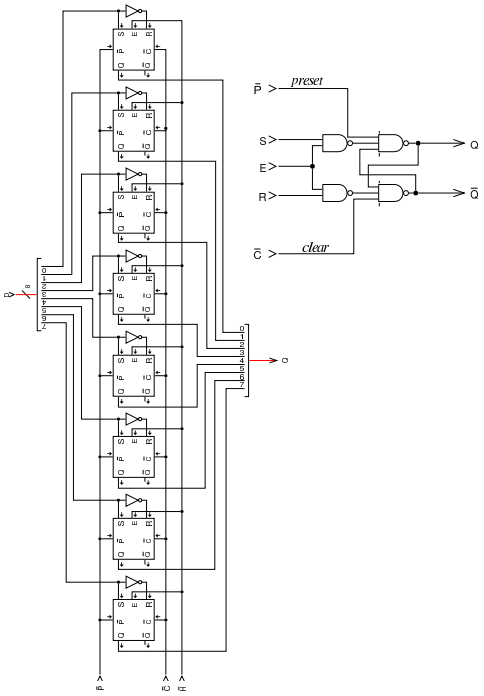
Nella figura, il grafico «A» si riferisce agli intervalli di validità dell'informazione degli indirizzi, a cavallo della variazione positiva del segnale di clock. Il grafico «B» mostra la situazione all'uscita del registro ***H***, che estende la validità dell'indirizzo ricevuto, in ingresso, perché quando il segnale di clock diventa positivo, blocca il valore alla sua uscita. Il grafico «C» mostra il periodo in cui è concesso alla memoria RAM di operare per modificare il proprio contenuto.

Figura u114.6. Fasi nel funzionamento del modulo **RAM**.



Il registro ***H*** è fatto di flip-flop SR semplici, collegati in modo da operare in qualità di flip-flop D, con ingresso di abilitazione. L'uso di flip-flop semplici, in questo caso, serve a evitare di introdurre latenze eccessive.

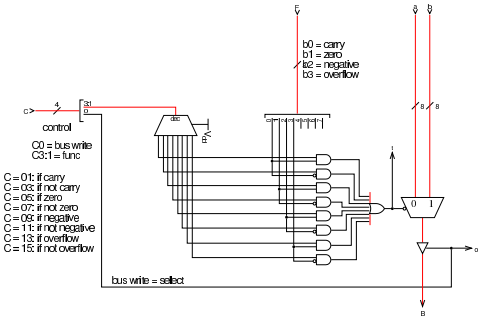
Figura u114.7. Schema interno del registro ***H*** (*hold*), contenuto del modulo **RAM**.



Modulo «SEL»

Il modulo di selezione non è cambiato, a parte la riorganizzazione del cablaggio e l'aggiunta di un'uscita diagnostica per sapere quando la condizione sottoposta a valutazione risulta avverarsi (uscita ***t***).

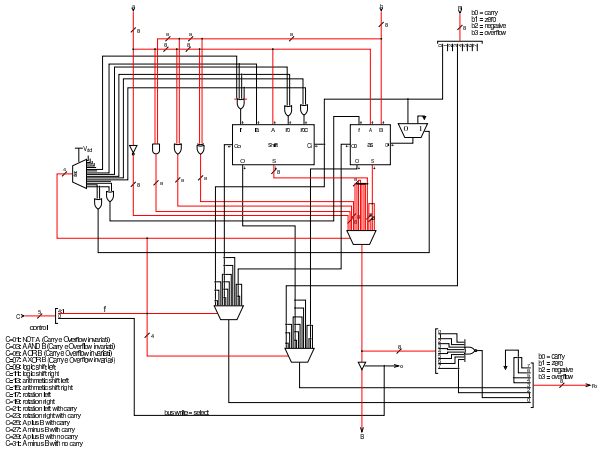
Figura u114.8. Schema interno del modulo **SEL**.



ALU

Anche la ALU non ha subito cambiamenti, a parte il fatto di avere riunito le linee di controllo e di disporre di un indicatore (uscita ***o***) che si attiva quando la ALU scrive sul bus dati un valore.

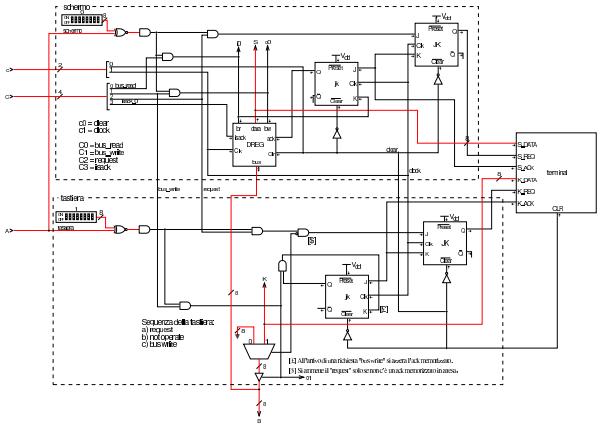
Figura u114.9. Schema interno del modulo **ALU**.



Terminale

Il terminale, costituito dal complesso tastiera-schermo, cambia rispetto alla versione precedente della CPU dimostrativa, in quanto torna a unificarsi, così come è realizzato nella versione già disponibile per Tkgate. Tuttavia, l'unificazione mantiene internamente la distinzione circuitale della versione precedente e anche la stessa logica di funzionamento; in pratica, si gestiscono sempre tastiera e schermo separatamente, ma nella realizzazione del codice TCL/Tk, si ha un modulo unico, che si manifesta così in una sola finestra durante la simulazione di Tkgate.

Figura u114.10. Circuito interno del modulo **TTY**: il registro ***DREG*** è identico a quello usato nella versione precedente.



Nella figura che mostra il circuito del modulo **TTY**, si può osservare la delimitazione tra le due porzioni, relative a tastiera e schermo: va notato che i due blocchi sono attivati attraverso indirizzi diversi (ingresso ***A***), esattamente come nella versione precedente. Il modulo **terminal** è scritto in Verilog, come già fatto nella versione precedente, solo che in questo caso si tratta di un modulo unico, per tastiera e schermo. A sua volta, il modulo **terminal** si avvale di codice TCL/Tk, costituito dal file '`terminal.tcl`' che viene mostrato subito dopo.

Listato u114.11. Modulo **terminal**, scritto in Verilog.

```
module terminal(K_DATA, K_REQ, K_ACK, S_DATA, S_REQ, S_ACK, CLR);

output K_ACK;
output S_ACK;
output [7:0] K_DATA;
input [7:0] S_DATA;
input K_REQ;
input S_REQ;
input CLR;
reg k_ready;
reg [7:0] key;
```

```
reg s_ready;

  initial
    begin
      k_ready = 0;
      s_ready = 0;
      key = 0;
    end

  always
    begin
      @(posedge CLR)
      k_ready = 0;
      s_ready = 0;
      key = 0;
    end

  initial $tkg$post("TERMINAL","%m");

  always
    begin
      @ (posedge K_REQ);
      # 5;
      key = $tkg$recv("%m.KD");
      # 5;
      k_ready = 1'b1;
      # 5;
      @ (negedge K_REQ);
      # 5;
      k_ready = 1'b0;
    end

  always
    begin
      @(posedge S_REQ);
      # 5;
      $tkg$send("%m.SD",S_DATA);
      # 5;
      s_ready = 1'b1;
      # 5;
      @(negedge S_REQ);
      # 5;
      s_ready = 1'b0;
    end

  assign S_ACK = s_ready;
  assign K_DATA = key;
  assign K_ACK = k_ready;

endmodule
```

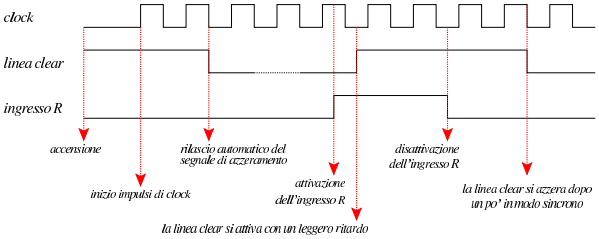
Listato u114.12. File '`share/tkgate/vpd/terminal.tcl`'. Il file è molto simile a quello fornito assieme a Tkgate, per la gestione di un terminale.

```
image create bitmap txtcurs -file "$bd/txtcurs.b"

VPD::register TERMINAL
VPD::allow TERMINAL::post
VPD::allow TERMINAL::data

namespace eval TERMINAL {
    # Dichiarazione delle variabili pubbliche: le variabili
    # $terminal_... sono array dei quali si utilizza solo
    # l'elemento $n, il quale identifica univocamente l'istanza
    # dell'interfaccia in funzione.
    variable terminal_w
    variable terminal_pos
    #
    variable KD
    # Funzione richiesta da Tkgate per creare l'interfaccia.
    proc post {n} {
        variable terminal_w
        variable terminal_pos
        # Crea una finestra e salva l'oggetto in un elemento dell'array
        # $terminal_w.
        set terminal_w($n) [VPD::createWindow "TERMINAL $n" -shutdowncommand "TERMINAL::unpost $n"]
        # Per maggiore comodità, copia il riferimento all'oggetto nella
        # variabile locale $w e in seguito fa riferimento all'oggetto
        # attraverso questa seconda variabile.
        set w $terminal_w($n)
        text $w.txt -state disabled
        pack $w.txt
        # Mette il cursore alla fine del testo visualizzato.
        $w.txt image create end -image txtcurs
        # Collega la digitazione della tastiera, relativa all'oggetto
        # rappresentato da $terminal_w($n), alla funzione sendChar.
        bind $w <KeyPress> "TERMINAL::sendChar $n \"%A\""
        # Apre un canale di lettura, denominato «SD» (screen data),
        # associandolo alla funzione «data»; inoltre, apre un canale
        # di scrittura, denominato «KD» (keyboard data).
        if {[info exists ::tkgate_isInitialized]} {
            VPD::outsignal $n.KD TERMINAL::KD($n)
            VPD::insignal $n.SD -command "TERMINAL::data $n" -format %d
        }
        # Azzera il contatore che tiene conto dei caratteri visualizzati
        # sullo schermo.
        set terminal_pos($n) 0
    }
    # Funzione che recepisce la digitazione e la immette nel canale
    # denominato «KD», relativo all'istanza attuale dell'interfaccia.
    proc sendChar {n key} {
        variable KD
        if { [string length $key ] == 1 } {
            binary scan $key c c
            set TERMINAL::KD($n) $c
        }
    }
    # Funzione richiesta da Tkgate per distruggere l'interfaccia.
    proc unpost {n} {
        variable terminal_w
        variable terminal_pos
        destroy $terminal_w($n)
        destroy $terminal_pos($n)
        unset terminal_w($n)
        unset terminal_pos($n)
    }
    # Funzione usata per recepire i dati da visualizzare.
    proc data {n c} {
        variable terminal_w
        variable terminal_pos
        # Per maggiore comodità, copia il riferimento all'oggetto che
        # rappresenta l'interfaccia nella variabile $w.
        set w $terminal_w($n)
        catch {
            # La variabile $c contiene il carattere da visualizzare.
            if { $c == 7 } {
                # BEL
                bell
                return
            } elseif { $c == 127 || $c == 8 } {
                # DEL | BS
                if { $terminal_pos($n) > 0 } {
                    # Cancella l'ultimo carattere visualizzato, ma solo
                    # se il contatore dei caratteri è maggiore di zero,
                    # altrimenti sparirebbe il cursore e la
                    # visualizzazione verrebbe collocata in un'area
                    # non visibile dello schermo.
                    $w.txt configure -state normal
                    $w.txt delete "end - 3 chars"
                    $w.txt see end
                    $w.txt configure -state disabled
                    set terminal_pos($n) [expr {$terminal_pos($n) - 1}]
                }
                return
            } elseif { $c == 13 } {
                # CR viene trasformato in LF.
                set c 10
            }
            # Converte il numero del carattere in un simbolo
            # visualizzabile.
            set x [format %c $c]
            # Visualizza il simbolo.
            $w.txt configure -state normal
            $w.txt insert "end - 2 chars" $x
            $w.txt see end
            $w.txt configure -state disabled
            # Aggiorna il contatore dei caratteri visualizzati.
            set terminal_pos($n) [expr {$terminal_pos($n) + 1}]
        }
    }
}
```

Unità di controllo

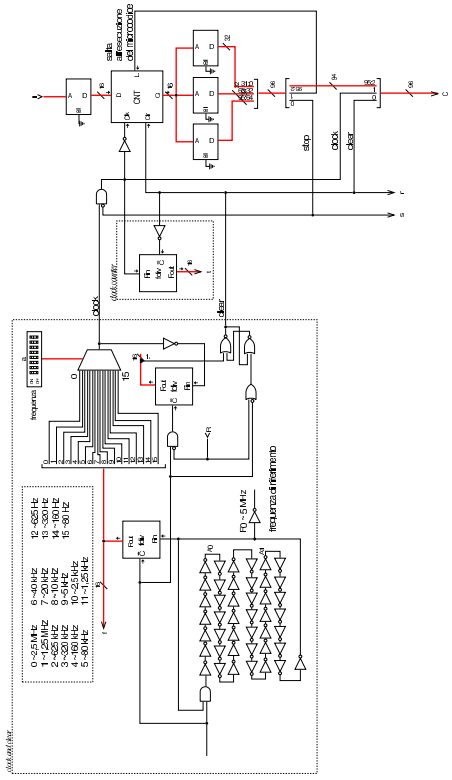
Per semplificare l'organizzazione del cablaggio, l'unità di controllo incorpora anche il generatore degli impulsi di clock; inoltre, il generatore di impulsi di clock incorpora la gestione del segnale di azzeramento, in modo che venga tolto solo nel momento più adatto rispetto all'impulso di clock: fino alla versione precedente della CPU dimostrativa, il circuito richiedeva un azzeramento manuale prima di poter iniziare a lavorare correttamente, inoltre il rilascio del segnale di azzeramento poteva avvenire in un momento inadatto che rendeva instabile il funzionamento.

Figura u114.13. Tempistica del funzionamento della linea ***clear***.



La figura successiva mostra lo schema dell'unità di controllo che integra le funzionalità di clock. Nella parte sinistra si trova il circuito che serve a generare gli impulsi di clock e a controllare la linea di azzeramento (*clear*). Va osservato che il modulo **`fdiv`** è esteso rispetto alla versione precedente, in modo da poter dividere la frequenza maggiormente; inoltre, la selezione della frequenza avviene attraverso un interruttore multiplo collegato a un multiplatore che si vede in alto. Tuttavia, dagli esperimenti fatti con Tkgate, la CPU funziona con una frequenza di clock non superiore a 1,25 MHz, pari al valore 1 per questo interruttore multiplo.

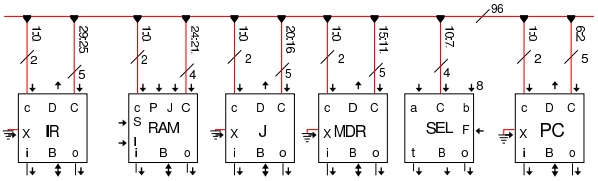
Figura u114.14. Schema completo dell'unità di controllo.



Nel circuito si usano diversi moduli **`fdiv`**: quello centrale serve a contare gli impulsi per sincronizzare la linea di azzeramento; quello più a destra serve a contare gli impulsi di clock a partire dall'avvio della CPU e consentirne il monitoraggio attraverso l'uscita ***t***: si tratta quindi soltanto di un ausilio diagnostico.

Nella parte destra che rappresenta l'unità di controllo originale, si vede un modulo contatore unico, a 16 bit (**`CNT`**), ma senza altre modifiche; inoltre, si vede che manca la possibilità di riportare l'esecuzione del microcodice all'indirizzo zero. Nelle linee che costituiscono assieme il bus di controllo, le prime due sono utilizzate per portare l'impulso di clock e il segnale di azzeramento (*clear*); le linee corrispondenti che escono dalla memoria che contiene il microcodice, servono per controllare l'unità stessa e non riguardano il resto della CPU. Allo stato attuale, questa versione dell'unità di controllo non permette di far riprendere il segnale di clock quando si attiva la linea interna di stop.

Figura u114.15. Connessione al bus di controllo.



Nella figura precedente si vedono alcuni componenti della CPU dimostrativa connessi al bus di controllo. Tutti questi componenti hanno in comune gli ingressi ***c*** (minuscola) e ***C*** (maiuscola). L'ingresso ***c*** è collegato sempre alle prime due linee del bus di controllo, dalle quali si ottiene, rispettivamente, il segnale di azzeramento e il segnale di clock. L'ingresso ***C***, invece, va connesso alle linee del bus di controllo che riguardano specificatamente il modulo. Nel caso dei registri uniformati, queste linee sono sempre cinque: *bus read*, *bus write*, *extra read*, *increment*, *decrement*. Il registro ***PC*** collega il proprio ingresso ***C*** alle linee da 2 a 6, del bus di controllo; il modulo **`SEL`** (che usa solo quattro linee di controllo) si collega alle linee da

7 a 10, e così si prosegue con gli altri componenti.

Memorie, campi, argomenti e codici operativi

Il sorgente Tkgate che serve a descrivere il contenuto delle memorie utilizzate con la CPU dimostrativa, inizia sempre con la definizione delle dimensioni di queste, assieme al loro nome:

```
map       bank[7:0]   ctrl.map;
microcode bank[31:0]  ctrl.micro0;
microcode bank[63:32] ctrl.micro1;
microcode bank[91:64] ctrl.micro2;
macrocode bank[7:0]   ram.ram;
```

In questa versione della CPU dimostrativa vengono cambiati leggermente i nomi delle memorie, in modo da rendere più chiaro il compito rispettivo. Va osservato che si utilizzano tre moduli di memoria, ognuno da 32 bit, per il microcodice, perché il bus di controllo prevede l'uso di molte linee.

```
field ctrl[1:0]  = {nop=0, stop=1, load=2};
field pc[6:2]    = {br=1, bw=2, xr=4, inc=8, dec=16};
field sel[10:7]  = {if_carry=1, if_not_carry=3,
                    if_zero=5, if_not_zero=7,
                    if_negative=9, if_not_negative=11,
                    if_overflow=13, if_not_overflow=15};
field mdr[15:11] = {br=1, bw=2, xr=4, inc=8, dec=16};
field j[20:16]   = {br=1, bw=2, xr=4, inc=8, dec=16};
field ram[24:21] = {br=1, bw=2, p=0, i=4, j=8, s=12};
field ir[29:25]  = {br=1, bw=2, xr=4, inc=8, dec=16};
field sp[34:30]  = {br=1, bw=2, xr=4, inc=8, dec=16};
field i[39:35]   = {br=1, bw=2, xr=4, inc=8, dec=16};
field b[44:40]   = {br=1, bw=2, xr=4, inc=8, dec=16};
field fl[49:45]  = {br=1, bw=2, xr=4, inc=8, dec=16};
field alu[54:50] = {not=1, and=3, or=5, xor=7, lshl=9, lshr=11,
                    ashl=13, ashr=15, rotl=17, rotr=19, rotcl=21,
                    rotcr=23, add_c=25, sub_b=27, add=29, sub=31};
field a[59:55]   = {br=1, bw=2, xr=4, inc=8, dec=16};
field ioa[64:60] = {br=1, bw=2, xr=4, inc=8, dec=16};
field ioc[68:65] = {br=1, bw=2, req=4, isack=8};
```

I campi delle linee di controllo sono scritti in modo più compatto. Va osservato che i valori rappresentabili in ogni campo possono sommarsi con l'operatore OR binario. In pratica, in relazione al campo ***pc***, il quale si riferisce alle linee di controllo specifiche del registro ***PC***, è possibile attivare sia la scrittura sul bus dati (**`pc=bw`**), sia incrementare il valore del registro (**`pc=inc`**), nello stesso ciclo di clock.

Nella dichiarazione della memoria si vede che le prime due linee sono relative all'unità di controllo, ma va ricordato che poi quelle due linee non vengono convogliate al bus di controllo esterno, perché al loro posto si fa transitare la linea di azzeramento e quella di clock.

In questa versione esiste la possibilità di dichiarare una microistruzione nulla, al solo scopo di far passare un ciclo di clock, indicando **`ctrl=nop`**.

Il codice operativo delle istruzioni rimane a 8 bit, poi ci possono essere un massimo di due argomenti (da 8 bit ognuno):

```
operands op_0 {
    -  = { };
};
operands op_1 {
    #1 = { +1=#1[7:0]; };
};
operands op_2 {
    #1,#2 = { +1=#1[7:0]; +2=#2[15:8]; };
};
```

Il codice operativo delle istruzioni disponibili è semplicemente un numero intero che parte da zero con l'istruzione **`nop`** e arriva a 255 con l'istruzione **`stop`**, senza altri accorgimenti:

```
op nop {                  // not operate
  map nop: 0;
  +0[7:0]=0;
  operands op_0;
};
op load                   // MDR <-- RAM[arg]
{
  map load: 1;
```

```
  +0[7:0]=1;
  operands op_1;
};
op load_i               // MDR <-- RAM[i]
{
  map load_i: 2;
  +0[7:0]=2;
  operands op_0;
};
op load_j               // MDR <-- RAM[j]
{
  map load_j: 3;
  +0[7:0]=3;
  operands op_0;
};
op store {              // RAM[arg] <-- MDR
  map store: 4;
  +0[7:0]=4;
  operands op_1;
};
op store_i {            // RAM[i] <-- MDR
  map store_i: 5;
  +0[7:0]=5;
  operands op_0;
};
op store_j {            // RAM[j] <-- MDR
  map store_j: 6;
  +0[7:0]=6;
  operands op_0;
};
op cp_ij {              // RAM[j++] <-- MDR <-- RAM[i++]
  map cp_ij: 7;
  +0[7:0]=7;
  operands op_0;
};
op cp_ji {              // RAM[i++] <-- MDR <-- RAM[j++]
  map cp_ji: 8;
  +0[7:0]=8;
  operands op_0;
};
op mv_mdr_a {           // A <-- MDR
  map mv_mdr_a: 9;
  +0[7:0]=9;
  operands op_0;
};
op mv_mdr_b {           // B <-- MDR
  map mv_mdr_b: 10;
  +0[7:0]=10;
  operands op_0;
};
op mv_mdr_fl {          // FL <-- MDR
  map mv_mdr_fl: 11;
  +0[7:0]=11;
  operands op_0;
};
op mv_mdr_sp {          // SP <-- MDR
  map mv_mdr_sp: 12;
  +0[7:0]=12;
  operands op_0;
};
op mv_mdr_i {           // I <-- MDR
  map mv_mdr_i: 13;
  +0[7:0]=13;
  operands op_0;
};
op mv_mdr_j {           // J <-- MDR
  map mv_mdr_j: 14;
  +0[7:0]=14;
  operands op_0;
};
op mv_a_mdr {           // A <-- MDR
  map mv_a_mdr: 15;
  +0[7:0]=15;
  operands op_0;
};
op mv_a_b {             // B <-- A
  map mv_a_b: 16;
  +0[7:0]=16;
  operands op_0;
};
op mv_a_fl {            // FL <-- A
```



```
  map mv_a_fl: 17;
  +0[7:0]=17;
  operands op_0;
};
op mv_a_sp {            // SP <-- A
  map mv_a_sp: 18;
  +0[7:0]=18;
  operands op_0;
};
op mv_a_i {             // I <-- A
  map mv_a_i: 19;
  +0[7:0]=19;
  operands op_0;
};
op mv_a_j {             // J <-- A
  map mv_a_j: 20;
  +0[7:0]=20;
  operands op_0;
};
op mv_b_a {              // A <-- B
  map mv_b_a: 21;
  +0[7:0]=21;
  operands op_0;
};
op mv_b_mdr {           // MDR <-- B
  map mv_b_mdr: 22;
  +0[7:0]=22;
  operands op_0;
};
op mv_b_fl {            // FL <-- B
  map mv_b_fl: 23;
  +0[7:0]=23;
  operands op_0;
};
op mv_b_sp {            // SP <-- B
  map mv_b_sp: 24;
  +0[7:0]=24;
  operands op_0;
};
op mv_b_i {             // I <-- B
  map mv_b_i: 25;
  +0[7:0]=25;
  operands op_0;
};
op mv_b_j {             // J <-- B
  map mv_b_j: 26;
  +0[7:0]=26;
  operands op_0;
};
op mv_fl_a {            // A <-- FL
  map mv_fl_a: 27;
  +0[7:0]=27;
  operands op_0;
};
op mv_fl_b {            // B <-- FL
  map mv_fl_b: 28;
  +0[7:0]=28;
  operands op_0;
};
op mv_fl_mdr {          // MDR <-- FL
  map mv_fl_mdr: 29;
  +0[7:0]=29;
  operands op_0;
};
op mv_fl_sp {           // SP <-- FL
  map mv_fl_sp: 30;
  +0[7:0]=30;
  operands op_0;
};
op mv_fl_i {            // I <-- FL
  map mv_fl_i: 31;
  +0[7:0]=31;
  operands op_0;
};
op mv_fl_j {            // J <-- FL
  map mv_fl_j: 32;
  +0[7:0]=32;
  operands op_0;
};
op mv_sp_a {            // A <-- SP
  map mv_sp_a: 33;
```

```
  +0[7:0]=33;
  operands op_0;
};
op mv_sp_b {            // B <-- SP
  map mv_sp_b: 34;
  +0[7:0]=34;
  operands op_0;
};
op mv_sp_fl {           // FL <-- SP
  map mv_sp_fl: 35;
  +0[7:0]=35;
  operands op_0;
};
op mv_sp_mdr {          // MDR <-- SP
  map mv_sp_mdr: 36;
  +0[7:0]=36;
  operands op_0;
};
op mv_sp_i {            // I <-- SP
  map mv_sp_i: 37;
  +0[7:0]=37;
  operands op_0;
};
op mv_sp_j {            // J <-- SP
  map mv_sp_j: 38;
  +0[7:0]=38;
  operands op_0;
};
op mv_i_a {             // A <-- I
  map mv_i_a: 39;
  +0[7:0]=39;
  operands op_0;
};
op mv_i_b {             // B <-- I
  map mv_i_b: 40;
  +0[7:0]=40;
  operands op_0;
};
op mv_i_fl {            // FL <-- I
  map mv_i_fl: 41;
  +0[7:0]=41;
  operands op_0;
};
op mv_i_sp {            // SP <-- I
  map mv_i_sp: 42;
  +0[7:0]=42;
  operands op_0;
};
op mv_i_mdr {           // MDR <-- I
  map mv_i_mdr: 43;
  +0[7:0]=43;
  operands op_0;
};
op mv_i_j {             // J <-- I
  map mv_i_j: 44;
  +0[7:0]=44;
  operands op_0;
};
op mv_j_a {             // A <-- J
  map mv_j_a: 45;
  +0[7:0]=45;
  operands op_0;
};
op mv_j_b {             // B <-- J
  map mv_j_b: 46;
  +0[7:0]=46;
  operands op_0;
};
op mv_j_fl {            // FL <-- J
  map mv_j_fl: 47;
  +0[7:0]=47;
  operands op_0;
};
op mv_j_sp {            // SP <-- J
  map mv_j_sp: 48;
  +0[7:0]=48;
  operands op_0;
};
op mv_j_i {             // I <-- J
  map mv_j_i: 49;
  +0[7:0]=49;
```



```
  operands op_0;
};
op mv_j_mdr {           // MDR <-- J
  map mv_j_mdr: 50;
  +0[7:0]=50;
  operands op_0;
};
op jump {               // PC <-- arg
  map jump: 51;
  +0[7:0]=51;
  operands op_1;
};
op jump_c {             // if carry, PC <-- arg
  map jump_c: 52;
  +0[7:0]=52;
  operands op_1;
};
op jump_nc {            // if not carry, PC <-- arg
  map jump_nc: 53;
  +0[7:0]=53;
  operands op_1;
};
op jump_z {             // if zero, PC <-- arg
  map jump_z: 54;
  +0[7:0]=54;
  operands op_1;
};
op jump_nz {            // if not zero, PC <-- arg
  map jump_nz: 55;
  +0[7:0]=55;
  operands op_1;
};
op jump_n {             // if negative, PC <-- arg
  map jump_n: 56;
  +0[7:0]=56;
  operands op_1;
};
op jump_nn {            // if not negative, PC <-- arg
  map jump_nn: 57;
  +0[7:0]=57;
  operands op_1;
};
op jump_o {             // if overflow, PC <-- arg
  map jump_o: 58;
  +0[7:0]=58;
  operands op_1;
};
op jump_no {            // if not overflow, PC <-- arg
  map jump_no: 59;
  +0[7:0]=59;
  operands op_1;
};
op call {
  map call : 60;
  +0[7:0]=60;
  operands op_1;
};
op call_i {             // call I
  map call_i: 61;
  +0[7:0]=61;
  operands op_0;
};
op call_j {             // call J
  map call_j: 62;
  +0[7:0]=62;
  operands op_0;
};
op return {
  map return : 63;
  +0[7:0]=63;
  operands op_0;
};
op push_mdr {
  map push_mdr: 64;
  +0[7:0]=64;
  operands op_0;
};
op push_a {
  map push_a: 65;
  +0[7:0]=65;
  operands op_0;
```

```
};
op push_b {
  map push_b: 66;
  +0[7:0]=66;
  operands op_0;
};
op push_fl {
  map push_fl: 67;
  +0[7:0]=67;
  operands op_0;
};
op push_i {
  map push_i: 68;
  +0[7:0]=68;
  operands op_0;
};
op push_j {
  map push_j: 69;
  +0[7:0]=69;
  operands op_0;
};
op pop_mdr {
  map pop_mdr: 70;
  +0[7:0]=70;
  operands op_0;
};
op pop_a {
  map pop_a: 71;
  +0[7:0]=71;
  operands op_0;
};
op pop_b {
  map pop_b: 72;
  +0[7:0]=72;
  operands op_0;
};
op pop_fl {
  map pop_fl: 73;
  +0[7:0]=73;
  operands op_0;
};
op pop_i {
  map pop_i: 74;
  +0[7:0]=74;
  operands op_0;
};
op pop_j {
  map pop_j: 75;
  +0[7:0]=75;
  operands op_0;
};
op not {
  map not: 76;
  +0[7:0]=76;
  operands op_0;
};
op and {
  map and: 77;
  +0[7:0]=77;
  operands op_0;
};
op or {
  map or: 78;
  +0[7:0]=78;
  operands op_0;
};
op xor {
  map xor: 79;
  +0[7:0]=79;
  operands op_0;
};
op lshl {
  map lshl: 80;
  +0[7:0]=80;
  operands op_0;
};
op lshr {
  map lshr: 81;
  +0[7:0]=81;
  operands op_0;
};
```



```
op ashl {
  map ashl: 82;
  +0[7:0]=82;
  operands op_0;
};
op ashr {
  map ashr: 83;
  +0[7:0]=83;
  operands op_0;
};
op rotl {
  map rotl: 84;
  +0[7:0]=84;
  operands op_0;
};
op rotr {
  map rotr: 85;
  +0[7:0]=85;
  operands op_0;
};
op rotcl {
  map rotcl: 86;
  +0[7:0]=86;
  operands op_0;
};
op rotcr {
  map rotcr: 87;
  +0[7:0]=87;
  operands op_0;
};
op add_c {
  map add_c: 88;
  +0[7:0]=88;
  operands op_0;
};
op sub_b {
  map sub_b: 89;
  +0[7:0]=89;
  operands op_0;
};
op add {
  map add: 90;
  +0[7:0]=90;
  operands op_0;
};
op sub {
  map sub: 91;
  +0[7:0]=91;
  operands op_0;
};
op in {
  map in : 92;
  +0[7:0]=92;
  operands op_1;
};
op out {
  map out: 93;
  +0[7:0]=93;
  operands op_1;
};
op is_ack {
  map is_ack: 94;
  +0[7:0]=94;
  operands op_2;
};
op stop {
  map stop : 255;
  +0[7:0]=255;
  operands op_0;
};
```

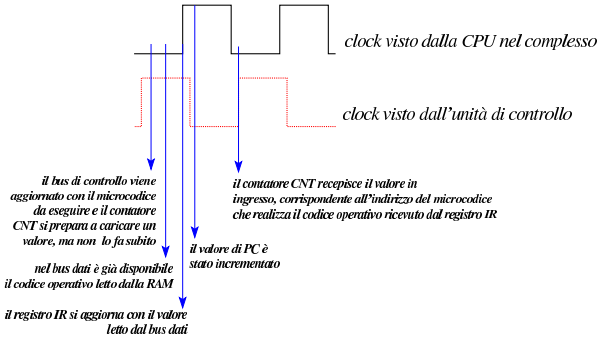
Microcodice

Nella descrizione del microcodice vero e proprio, si inizia con ciò che serve al caricamento del primo codice operativo dalla memoria, ma in questa realizzazione può avvenire tutto in un solo ciclo di clock:

```
begin microcode @ 0
//
    ir=br ram=bw ram=p pc=inc ctrl=load;   // IR <- RAM[pc++],
                                           // jump MAP[ir];
```

In pratica, il registro ***IR*** carica dal bus dati quanto emesso dal modulo **RAM**, il quale a sua volta riceve l'indirizzo dal registro ***PC***, il quale viene incrementato contestualmente. Oltre a questo, si richiede all'unità di controllo di aggiornare il proprio contatore con il valore proveniente dalla memoria ***ctrl.map*** in corrispondenza dell'indirizzo che rappresenta il codice operativo. Si può fare tutto questo in un solo ciclo di clock perché la struttura della CPU è cambiata rispetto alla versione precedente. Vanno considerate le diverse fasi del ciclo di clock, che intervengono in modo differente nell'unità di controllo rispetto ai componenti che poi sono connessi al bus di controllo, come si vede nella figura successiva.

Figura u114.21. Il ciclo di clock dell'operazione di caricamento e messa in esecuzione di un'istruzione contenuta nella memoria RAM.



Pertanto, con un solo ciclo di clock si realizza quello che è noto come *fetch*. Nella descrizione successiva delle istruzioni, alla fine di ogni procedimento, si ripete la microistruzione di *fetch*, senza bisogno di far ripartire il contatore ***CNT*** dalla prima microistruzione, come necessario, invece, nelle versioni precedenti della CPU dimostrativa.

Per la macroistruzione **nop**, in pratica, c'è solo la microistruzione di *fetch*:

```
nop:
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
```

Segue la descrizione delle altre macroistruzioni:

```
load:
    i=br ram=bw ram=p pc=inc;                   // I <- RAM[pc++];
    mdr=br ram=bw ram=i;                        // MDR <- RAM[i];
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
load_i:
    mdr=br ram=bw ram=i;                        // MDR <- RAM[i];
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
load_j:
    mdr=br ram=bw ram=j;                        // MDR <- RAM[j];
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
store:
    i=br ram=bw ram=p pc=inc;                   // I <- RAM[pc++];
    ram=br ram=i mdr=bw;                        // RAM[i] <- MDR;
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
store_i:
    ram=br ram=i mdr=bw;                        // RAM[i] <- MDR;
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
store_j:
    ram=br ram=j mdr=bw;                        // RAM[j] <- MDR;
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
cp_ij:
    mdr=br ram=bw ram=i i=inc;                  // MDR <- RAM[i++];
    ram=br ram=j mdr=bw j=inc;                  // RAM[j++] <- MDR;
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
cp_ji:
    mdr=br ram=bw ram=j j=inc;                  // MDR <- RAM[j++];
    ram=br ram=i mdr=bw i=inc;                  // RAM[i++] <- MDR;
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
mv_mdr_a:
    a=br mdr=bw;                                // A <- MDR;
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
mv_mdr_b:
    b=br mdr=bw;                                // B <- MDR;
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
mv_mdr_fl:
    fl=br mdr=bw;                               // FL <- MDR;
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
mv_mdr_sp:
    sp=br mdr=bw;                               // SP <- MDR;
```



```
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
mv_mdr_i:
    i=br mdr=bw;                                // I <- MDR;
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
mv_mdr_j:
    j=br mdr=bw;                                // J <- MDR;
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
mv_a_mdr:
    mdr=br a=bw;                                // MDR <- A;
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
mv_a_b:
    b=br a=bw;                                  // B <- A;
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
mv_a_fl:
    fl=br a=bw;                                 // FL <- A;
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
mv_a_sp:
    sp=br a=bw;                                 // SP <- A;
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
mv_a_i:
    i=br  a=bw;                                 // I <- A;
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
mv_a_j:
    j=br  a=bw;                                 // J <- A;
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
mv_b_a:
    a=br b=bw;                                  // A <- B;
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
mv_b_mdr:
    mdr=br b=bw;                                // MDR <- B;
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
mv_b_fl:
    fl=br b=bw;                                 // FL <- B;
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
mv_b_sp:
    sp=br b=bw;                                 // SP <- B;
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
mv_b_i:
    i=br b=bw;                                  // I <- B;
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
mv_b_j:
    j=br b=bw;                                  // J <- B;
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
mv_fl_a:
    a=br fl=bw;                                 // A <- FL;
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
mv_fl_b:
    b=br fl=bw;                                 // B <- FL;
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
mv_fl_mdr:
    mdr=br fl=bw;                               // MDR <- FL;
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
mv_fl_sp:
    sp=br fl=bw;                                // SP <- FL;
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
mv_fl_i:
    i=br fl=bw;                                 // I <- FL;
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
mv_fl_j:
    j=br fl=bw;                                 // J <- FL;
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
mv_sp_a:
    a=br sp=bw;                                 // A <- SP;
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
mv_sp_b:
    b=br sp=bw;                                 // B <- SP;
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
mv_sp_fl:
    fl=br sp=bw;                                // FL <- SP;
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
mv_sp_mdr:
    mdr=br sp=bw;                               // MDR <- SP;
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
mv_sp_i:
    i=br sp=bw;                                 // I <- SP;
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
mv_sp_j:
    j=br sp=bw;                                 // J <- SP;
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
mv_i_a:
    a=br i=bw;                                  // A <- I;
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
mv_i_b:
    b=br i=bw;                                  // B <- I;
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
mv_i_fl:
    fl=br i=bw;                                 // FL <- I;
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
mv_i_sp:
    sp=br i=bw;                                 // SP <- I;
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
mv_i_mdr:
    mdr=br i=bw;                                // MDR <- I;
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
mv_i_j:
    j=br i=bw;                                  // J <- I;
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
mv_j_a:
    a=br j=bw;                                  // A <- J;
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
```

```
mv_j_b:
    b=br j=bw;                                  // B <- J;
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
mv_j_fl:
    fl=br j=bw;                                 // FL <- J;
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
mv_j_sp:
    sp=br j=bw;                                 // SP <- J;
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
mv_j_i:
    i=br j=bw;                                  // I <- J;
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
mv_j_mdr:
    mdr=br j=bw;                                // MDR <- J;
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
jump:
    i=br pc=bw;                                 // I <- PC
    pc=br ram=bw ram=i;                         // PC <-- RAM[i]
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
jump_c:
    mdr=br ram=bw ram=p pc=inc;                 // MDR <-- RAM[pc++]
    pc=br sel=if_carry;                         // PC = (carry?MDR:PC)
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
jump_nc:
    mdr=br ram=bw ram=p pc=inc;                 // MDR <-- RAM[pc++]
    pc=br sel=if_not_carry;                     // PC = (not_carry?MDR:PC)
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
jump_z:
    mdr=br ram=bw ram=p pc=inc;                 // MDR <-- RAM[pc++]
    pc=br sel=if_zero;                          // PC = (zero?MDR:PC)
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
jump_nz:
    mdr=br ram=bw ram=p pc=inc;                 // MDR <-- RAM[pc++]
    pc=br sel=if_not_zero;                      // PC = (not_zero?MDR:PC)
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
jump_n:
    mdr=br ram=bw ram=p pc=inc;                 // MDR <-- RAM[pc++]
    pc=br sel=if_negative;                      // PC = (negative?MDR:PC)
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
jump_nn:
    mdr=br ram=bw ram=p pc=inc;                 // MDR <-- RAM[pc++]
    pc=br sel=if_not_negative;                  // PC = (not_negative?MDR:PC)
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
jump_o:
    mdr=br ram=bw ram=p pc=inc;                 // MDR <-- RAM[pc++]
    pc=br sel=if_overflow;                      // PC = (overflow?MDR:PC)
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
jump_no:
    mdr=br ram=bw ram=p pc=inc;                 // MDR <-- RAM[pc++]
    pc=br sel=if_not_overflow;                  // PC = (not_overflow?MDR:PC)
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
call:
    i=br ram=bw ram=p pc=inc sp=dec;            // I <- RAM[pc++], SP--;
    ram=br ram=s pc=bw;                         // RAM[sp] <- PC;
    pc=br i=bw;                                 // PC <- I;
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
call_i:
    sp=dec;                                     // SP--;
    ram=br ram=s pc=bw;                         // RAM[sp] <- PC;
    pc=br i=bw;                                 // PC <- I;
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
call_j:
    sp=dec;                                     // SP--;
    ram=br ram=s pc=bw;                         // RAM[sp] <- PC;
    pc=br j=bw;                                 // PC <- J;
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
return:
    pc=br ram=bw ram=s sp=inc;                  // PC <- RAM[sp++];
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
push_mdr:
    sp=dec;                                     // SP--;
    ram=br ram=s mdr=bw;                        // RAM[sp] <- MDR;
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
push_a:
    sp=dec;                                     // SP--;
    ram=br ram=s a=bw;                          // RAM[sp] <- A;
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
push_b:
    sp=dec;                                     // SP--;
    ram=br ram=s b=bw;                          // RAM[sp] <- B;
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
push_fl:
    sp=dec;                                     // SP--;
    ram=br ram=s fl=bw;                         // RAM[sp] <- FL;
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
push_i:
    sp=dec;                                     // SP--;
    ram=br ram=s i=bw;                          // RAM[sp] <- I;
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
push_j:
    sp=dec;                                     // SP--;
    ram=br ram=s j=bw;                          // RAM[sp] <- J;
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
pop_mdr:
    mdr=br ram=bw ram=s sp=inc;                 // MDR <- RAM[sp++];
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
pop_a:
    a=br ram=bw ram=s sp=inc;                   // A <- RAM[sp++];
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
pop_b:
```



```
    b=br ram=bw ram=s sp=inc;                   // B <- RAM[sp++];
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
pop_fl:
    fl=br ram=bw ram=s sp=inc;                  // FL <- RAM[sp++];
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
pop_i:
    i=br ram=bw ram=s sp=inc;                   // I <- RAM[sp++];
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
pop_j:
    j=br ram=bw ram=s sp=inc;                   // J <- RAM[sp++];
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
not:
    a=br alu=not fl=xr;                         // A <- NOT A;
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
and:
    a=br alu=and fl=xr;                         // A <- A AND B
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
or:
    a=br alu=or fl=xr;                          // A <- A OR B
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
xor:
    a=br alu=xor fl=xr;                         // A <- A XOR B
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
lshl:
    a=br alu=lshl fl=xr;                        // A <- A << 1
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
lshr:
    a=br alu=lshr fl=xr;                        // A <- A >> 1
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
ashl:
    a=br alu=ashl fl=xr;                        // A <- A*2
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
ashr:
    a=br alu=ashr fl=xr;                        // A <- A/2
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
rotl:
    a=br alu=rotl fl=xr;                        // A <- rotl(A)
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
rotr:
    a=br alu=rotr fl=xr;                        // A <- rotr(A)
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
rotcl:
    a=br alu=rotcl fl=xr;                       // A <- rotcl(A)
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
rotcr:
    a=br alu=rotcr fl=xr;                       // A <- rotcr(A)
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
add_c:
    a=br alu=add_c fl=xr;                       // A <- A+B+carry
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
sub_b:
    a=br alu=sub_b fl=xr;                       // A <- A-B-borrow
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
add:
    a=br alu=add fl=xr;                         // A <- A+B
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
sub:
    a=br alu=sub fl=xr;                         // A <- A-B
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
in:
    ioa=br ram=bw ram=p pc=inc;                 // IOA <- RAM[pc++];
    ioc=req;                                    // I/O request;
    ctrl=nop;                                   // non fa alcunché
    a=br ioc=bw;                                // A <- I/O
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
out:
    ioa=br ram=bw ram=p pc=inc;                 // IOA <- RAM[pc++];
    ioc=br a=bw;                                // I/O <- A
    ioc=req;                                    // I/O request;
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
is_ack:
    ioa=br ram=bw ram=p pc=inc;                 // IOA <- RAM[pc++];
    mdr=br ram=bw ram=p pc=inc;                 // MDR <- RAM[pc++];
    a=br ioc=bw ioc=isack;                      // A <- I/O is ack;
    a=br alu=not fl=xr;                         // A <- NOT A;
    a=br alu=not fl=xr;                         // A <- NOT A;
    pc=br sel=if_not_zero;                      // PC = (not_zero?MDR:PC);
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
stop:
    ctrl=stop;                                  // stop clock
    // if resumed:
    ir=br ram=bw ram=p pc=inc ctrl=load;        // fetch
end
```

L'insieme delle macroistruzioni è cambiato leggermente ed è anche esteso, in considerazione delle modifiche apportate alla CPU, come descritto nella tabella successiva.

Tabella u114.24. Elenco completo delle macroistruzioni disponibili nella versione attuale della CPU dimostrativa.

Sintassi	Cicli di clock	Descrizione
`nop`	1	Non fa alcunché.

Sintassi	Cicli di clock	Descrizione
`load` ***indirizzo***	3	Carica nel registro ***I*** l'argomento e nel registro ***MDR*** il contenuto della memoria all'indirizzo specificato.
`load_i` `load_j`	2	Carica nel registro ***MDR*** il contenuto della memoria all'indirizzo specificato dal registro ***I*** o ***J*** rispettivamente.
`store` ***indirizzo***	3	Carica nel registro ***I*** l'argomento e scrive in memoria, in corrispondenza dell'indirizzo indicato, quanto contenuto nel registro ***MDR***.
`store_i` `store_j`	2	Scrive in memoria, in corrispondenza dell'indirizzo contenuto nel registro ***I*** o ***J*** rispettivamente, quanto contenuto nel registro ***MDR***.
`cp_ij` `cp_ji`	3	Nel primo caso, copia il contenuto della memoria RAM in corrispondenza dell'indirizzo contenuto nel registro ***I***, all'indirizzo rappresentato dal registro ***J***, incrementando successivamente i due registri; nel secondo caso, la copia avviene in senso inverso, ma sempre con incremento successivo degli indici.

`mv_mdr_a`
`mv_mdr_b`
`mv_mdr_fl`
`mv_mdr_sp`
`mv_mdr_i`
`mv_mdr_j`
	2	Copia il contenuto del registro ***MDR*** nel registro ***A***, ***B***, ***FL***, ***SP***, ***I*** o ***J***, rispettivamente.

`mv_a_mdr`
`mv_a_b`
`mv_a_fl`
`mv_a_sp`
`mv_a_i`
`mv_a_j`
	2	Copia il contenuto del registro ***A*** nel registro ***MDR***, ***B***, ***FL***, ***SP***, ***I*** o ***J***, rispettivamente.

`mv_b_a`
`mv_b_mdrb`
`mv_b_fl`
`mv_b_sp`
`mv_b_i`
`mv_b_j`
	2	Copia il contenuto del registro ***B*** nel registro ***A***, ***MDR***, ***FL***, ***SP***, ***I*** o ***J***, rispettivamente.

`mv_fl_a`
`mv_fl_b`
`mv_fl_mdr`
`mv_fl_sp`
`mv_fl_i`
`mv_fl_j`
	2	Copia il contenuto del registro ***FL*** nel registro ***A***, ***B***, ***MDR***, ***SP***, ***I*** o ***J***, rispettivamente.

Sintassi	Cicli di clock	Descrizione

`mv_sp_a`
`mv_sp_b`
`mv_sp_fl`
`mv_sp_mdr`
`mv_sp_i`
`mv_sp_j`
	2	Copia il contenuto del registro ***SP*** nel registro ***A***, ***B***, ***FL***, ***MDR***, ***I*** o ***J***, rispettivamente.

`mv_i_a`
`mv_i_b`
`mv_i_fl`
`mv_i_sp`
`mv_i_mdr`
`mv_i_j`
	2	Copia il contenuto del registro ***I*** nel registro ***A***, ***B***, ***FL***, ***SP***, ***MDR*** o ***J***, rispettivamente.

`mv_j_a`
`mv_j_b`
`mv_j_fl`
`mv_j_sp`
`mv_j_i`
`mv_j_mdr`
	2	Copia il contenuto del registro ***J*** nel registro ***A***, ***B***, ***FL***, ***SP***, ***I*** o ***MDR***, rispettivamente.

`jump` ***indirizzo***	3	Mette nel registro ***I*** l'argomento e poi salta all'esecuzione dell'istruzione che si trova all'indirizzo indicato.
`jump_c` ***indirizzo*** `jump_nc` ***indirizzo*** `jump_z` ***indirizzo*** `jump_nz` ***indirizzo*** `jump_n` ***indirizzo*** `jump_nn` ***indirizzo*** `jump_o` ***indirizzo*** `jump_no` ***indirizzo***	3	Mette nel registro ***MDR*** l'argomento e poi, se la condizione si avvera, salta all'esecuzione dell'istruzione che si trova all'indirizzo indicato. Le condizioni sono, nell'ordine: esistenza di un riporto, assenza di un riporto, valore a zero, valore diverso da zero, valore negativo, valore non negativo, straripamento, non straripamento.
`call` ***indirizzo***	4	Mette nel registro ***I*** l'argomento e riduce di una unità il valore del registro ***SP***; poi, in corrispondenza dell'indirizzo di memoria contenuto nel registro ***SP***, salva il valore contenuto nel registro ***PC***: in pratica salva nella pila il valore di ***PC***. Subito dopo, salta all'indirizzo memorizzato nel registro ***I***.
`call_i` `call_j`	4	Riduce di una unità il valore del registro ***SP***; poi, in corrispondenza dell'indirizzo di memoria contenuto nel registro ***SP***, salva il valore contenuto nel registro ***PC***: in pratica salva nella pila il valore di ***PC***. Subito dopo, salta all'indirizzo memorizzato nel registro ***I*** o ***j***, rispettivamente.

Sintassi	Cicli di clock	Descrizione
`return`	2	Legge il valore contenuto nella posizione di memoria indicato dal registro ***SP*** e lo mette nel registro ***PC***, quindi incrementa di una unità il registro ***SP***. In pratica, estrae dalla pila l'indirizzo di ritorno di una chiamata precedente.

`push_mdr`
`push_a`
`push_b`
`push_fl`
`push_sp`
`push_i`
`push_j`
	3	Riduce di una unità il valore del registro ***SP***; poi, in corrispondenza dell'indirizzo di memoria contenuto nel registro ***SP***, salva il valore contenuto nel registro ***MDR***, ***A***, ***B***, ***FL***, ***SP***, ***I*** o ***J***, rispettivamente.

`pop_mdr`
`pop_a`
`pop_b`
`pop_fl`
`pop_sp`
`pop_i`
`pop_j`
	3	Copia, rispettivamente nel registro ***MDR***, ***A***, ***B***, ***FL***, ***SP***, ***I*** o ***J***, il contenuto dell'area di memoria corrispondente all'indirizzo indicato dal registro ***SP***, incrementando poi ***SP*** di una unità.

`not`	2	Esegue l'operazione logica NOT ***A***, bit per bit, mettendo il risultato nello stesso registro ***A***.

`and`
`or`
`xor`
	2	Esegue, rispettivamente, l'operazione logica ***A*** AND ***B***, ***A*** OR ***B***, ***A*** XOR ***B***, mettendo il risultato nel registro ***A***.

`lshl` `lshr`	2	Esegue, rispettivamente, lo scorrimento logico a sinistra o a destra, del contenuto del registro ***A***.
`ashl` `ashr`	2	Esegue, rispettivamente, lo scorrimento aritmetico a sinistra o a destra, del contenuto del registro ***A***.
`rotl` `rotl`	2	Esegue, rispettivamente, la rotazione a sinistra o a destra, del contenuto del registro ***A***.
`rotcl` `rotcl`	2	Esegue, rispettivamente, la rotazione con riporto a sinistra o a destra, del contenuto del registro ***A*** e dell'indicatore di riporto.
`add_c` `sub_b`	2	Esegue, rispettivamente, la somma (***A***+***B***) o la sottrazione (***A***−***B***), utilizzando il riporto o la richiesta di prestito precedente, mettendo il risultato nel registro ***A***.
`add` `sub`	2	Esegue, rispettivamente, la somma (***A***+***B***) o la sottrazione (***A***−***B***), ignorando il riporto o la richiesta di prestito precedente, mettendo il risultato nel registro ***A***.

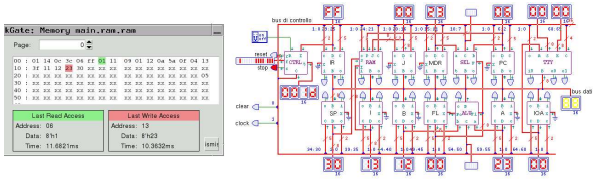
Sintassi	Cicli di clock	Descrizione
`in` ***indirizzo_io***	5	Legge un valore dal dispositivo di I/O individuato dall'indirizzo fornito, mettendo questo valore nel registro ***A***.
`out` ***indirizzo_io***	4	Scrive un valore sul dispositivo di I/O individuato dall'indirizzo fornito, utilizzando il valore contenuto nel registro ***A***.
`is_ack` ***indirizzo_io indirizzo***	7	Carica l'indirizzo rappresentato dal secondo argomento nel registro ***MDR*** e poi interroga un dispositivo di I/O per ottenere un codice di conferma da mettere nel registro ***A***, aggiornando gli indicatori: se si ottiene un valore diverso da zero, corrispondente al codice di conferma, si salta all'indirizzo di memoria indicato come secondo argomento.
`stop`	1	Ferma il segnale di clock per tutta la CPU.

Nelle sezioni successive vengono proposti alcuni esempi per verificare il funzionamento delle macroistruzioni della versione attuale della CPU dimostrativa. Gli esempi sono dimostrati attraverso dei video che mettono in evidenza anche l'accesso alla memoria RAM.

Macrocodice: chiamata di una routine

```
begin macrocode @ 0
start:
        load #data_3          // Colloca la pila dei dati.
        mv_mdr_sp
        call #somma           // Chiama la funzione somma().
        stop
somma:
        load #data_0
        mv_mdr_a
        load #data_1
        mv_mdr_b
        add
        mv_a_mdr
        store #data_2
        return
data_0:
        .byte 0x11
data_1:
        .byte 0x12
data_2:
        .byte 0x00
data_3:
        .byte 0x30
end
```

Figura u114.26. Situazione conclusiva del bus dati dopo l'esecuzione del codice contenuto nel listato precedente. Video: *http://www.youtube.com/watch?v=eATz3XLYWbc*

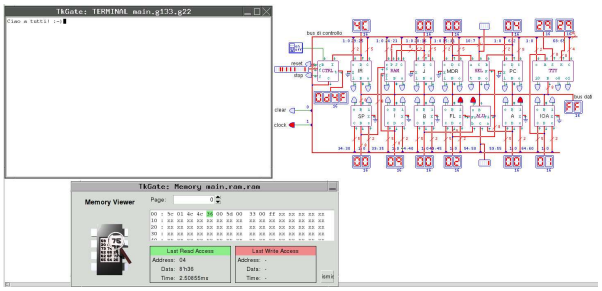


Macrocodice: inserimento da tastiera e visualizzazione sullo schermo

```
begin macrocode @ 0
start:
        in 1          // Legge dalla tastiera.
        not           // Inverte per aggiornare
        not           //   gli indicatori.
```

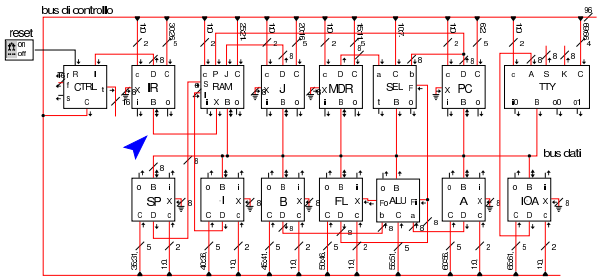
```
        jump_z #start // Se il valore è zero,
                      //   ripete la lettura.
        out 0         // Altrimenti emette lo stesso
                      //   valore sullo schermo.
        jump #start   // Ricomincia.
stop:                 // Non raggiunge mai questo punto.
        stop
end
```

Figura u114.28. Inserimento da tastiera e visualizzazione sullo schermo. Video: *http://www.youtube.com/watch?v=m22oK22ULTwWo*



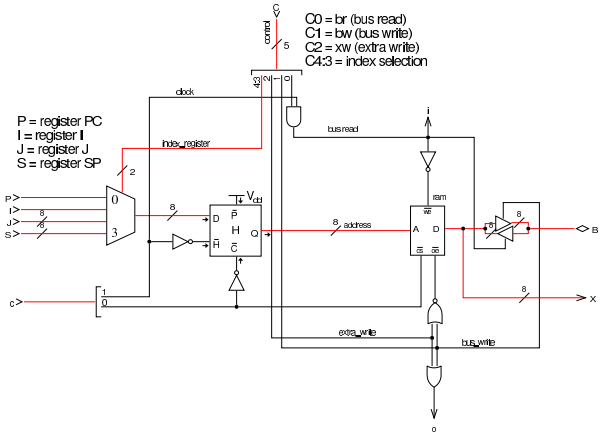
Viene proposta una modifica ulteriore della CPU dimostrativa, con lo scopo di migliorare leggermente la sua efficienza, attraverso il collegamento diretto tra il modulo **RAM** e il registro ***IR***, per non interferire con il bus dati quando si può trasferire un codice operativo dalla memoria al registro relativo.

Figura u115.1. CPU dimostrativa, versione «J».



Per realizzare il collegamento evidenziato nella figura, il modulo **RAM** viene modificato, per poter pilotare un'uscita ausiliaria (***X***); di conseguenza di ingrandisce il suo collegamento al bus di controllo, costringendo a riadattare i collegamenti degli altri moduli.

Figura u115.2. Modulo **RAM** modificato con l'aggiunta dell'uscita ausiliaria.



La dichiarazione dei campi del bus di controllo richiede quindi il cambiamento del significato delle linee di controllo relative al modulo **RAM**, assieme allo slittamento in avanti del collegamento degli altri moduli che seguono:

```
field ctrl[1:0]  = {nop=0, stop=1, load=2};
field pc[6:2]    = {br=1, bw=2, xr=4, inc=8, dec=16};
field sel[10:7]  = {if_carry=1, if_not_carry=3,
                    if_zero=5, if_not_zero=7,
                    if_negative=9, if_not_negative=11,
                    if_overflow=13, if_not_overflow=15};
field mdr[15:11] = {br=1, bw=2, xr=4, inc=8, dec=16};
field j[20:16]   = {br=1, bw=2, xr=4, inc=8, dec=16};
field ram[25:21] = {br=1, bw=2, xw=4, p=0, i=8, j=16, s=24};
field ir[30:26]  = {br=1, bw=2, xr=4, inc=8, dec=16};
field sp[35:31]  = {br=1, bw=2, xr=4, inc=8, dec=16};
field i[40:36]   = {br=1, bw=2, xr=4, inc=8, dec=16};
field b[45:41]   = {br=1, bw=2, xr=4, inc=8, dec=16};
field fl[50:46]  = {br=1, bw=2, xr=4, inc=8, dec=16};
field alu[55:51] = {not=1, and=3, or=5, xor=7, lshl=9, lshr=11,
                    ashl=13, ashr=15, rotl=17, rotr=19, rotcl=21,
                    rotcr=23, add_c=25, sub_b=27, add=29, sub=31};
field a[60:56]   = {br=1, bw=2, xr=4, inc=8, dec=16};
field ioa[65:61] = {br=1, bw=2, xr=4, inc=8, dec=16};
field ioc[69:66] = {br=1, bw=2, req=4, isack=8};
```

Nella dichiarazione del microcodice, cambia il fatto che la RAM,

per poter comunicare con il registro ***IR***, deve avere abilitata la linea ***xr*** (*extra write*), pertanto, tutte le microistruzioni di caricamento del codice operativo successivo (*fetch*), vanno cambiate nel modo successivo:

```
    ir=br ram=xw ram=p pc=inc ctrl=load;
```

Infine, dove possibile, la microistruzione di caricamento (*fetch*) che appare alla fine di ogni macroistruzione, si fonde con la penultima macroistruzione. Si tratta precisamente delle istruzioni relative alla copia di un valore da un registro a un altro e quelle che utilizzano la ALU: si riducono tutte a un solo ciclo di clock:

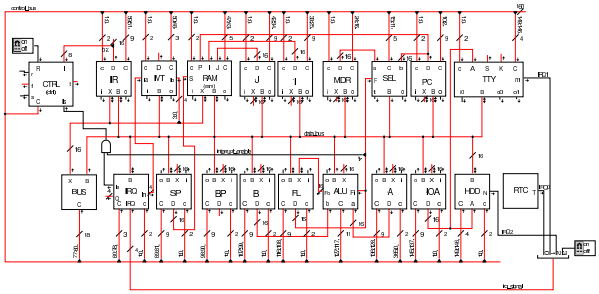
```
mv_mdr_a:
    a=br mdr=bw ir=br ram=xw ram=p pc=inc ctrl=load;
mv_mdr_b:
    b=br mdr=bw ir=br ram=xw ram=p pc=inc ctrl=load;
…
mv_j_i:
    i=br j=bw ir=br ram=xw ram=p pc=inc ctrl=load;
mv_j_mdr:
    mdr=br j=bw ir=br ram=xw ram=p pc=inc ctrl=load;
```

```
not:
    a=br alu=not fl=xr ir=br ram=xw ram=p pc=inc ctrl=load;
and:
    a=br alu=and fl=xr ir=br ram=xw ram=p pc=inc ctrl=load;
…
add:
    a=br alu=add fl=xr ir=br ram=xw ram=p pc=inc ctrl=load;
sub:
    a=br alu=sub fl=xr ir=br ram=xw ram=p pc=inc ctrl=load;
```


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.

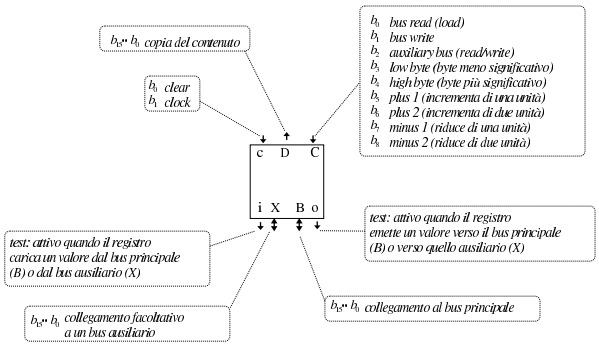


Figura u116.3. Struttura dei registri a 16 bit.

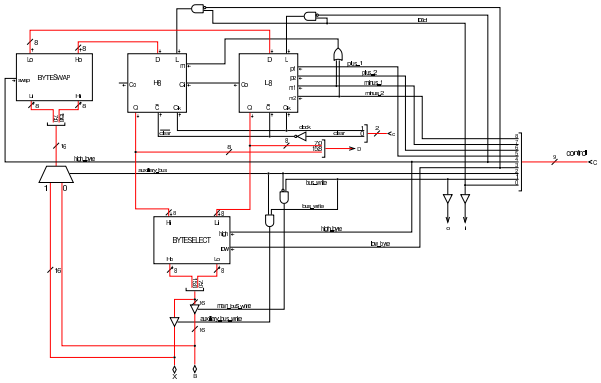
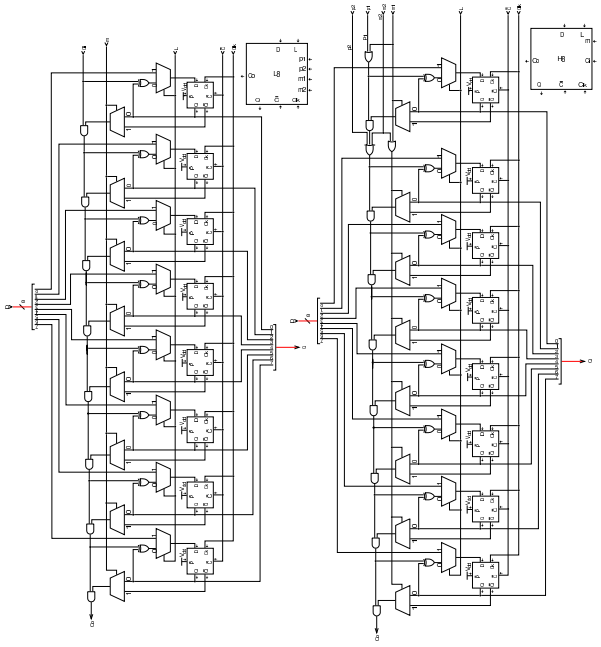


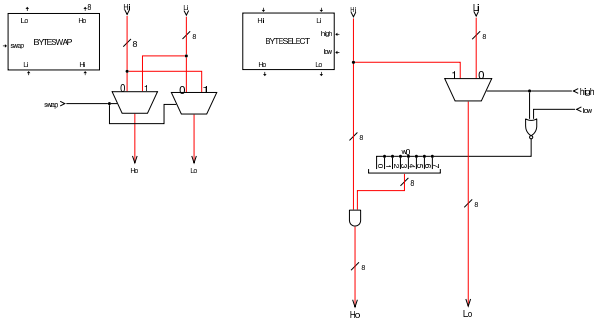
Figura u116.4. Struttura dei moduli **H8** e **L8**.



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***.

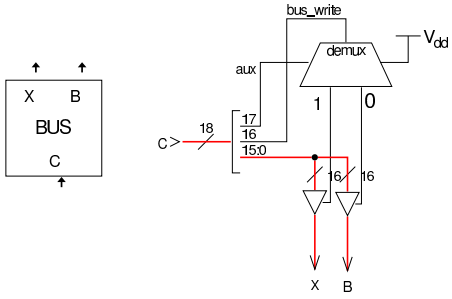
Figura u116.5. Struttura interna dei moduli **BYTESWAP** e **BYTESELECT** per lo scambio o la selezione dei byte.



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.

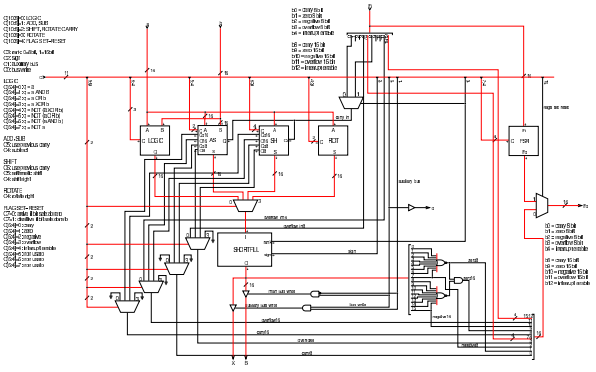


Figura u116.8. Struttura dell'unità logica interna alla ALU.

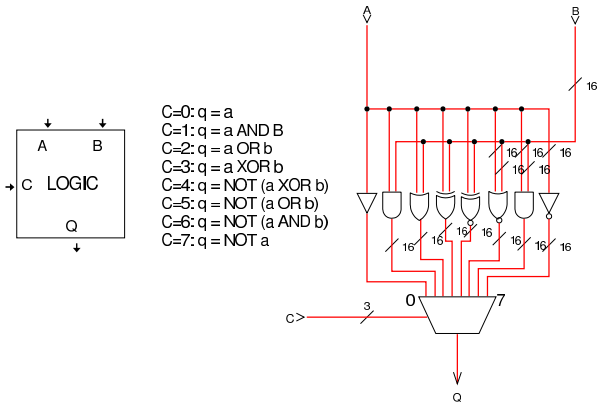


Figura u116.9. Struttura del modulo di addizione e sottrazione **AS**.

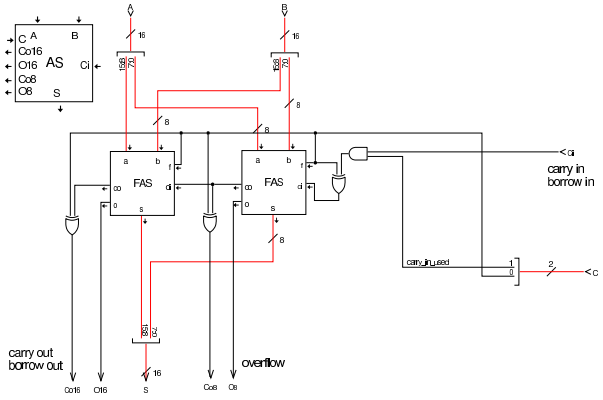


Figura u116.10. Modulo **FAS** (*full adder-subtractor*) contenuto nell'unità aritmetica. I moduli **fa** sono degli addizionatori completi (*full adder*).

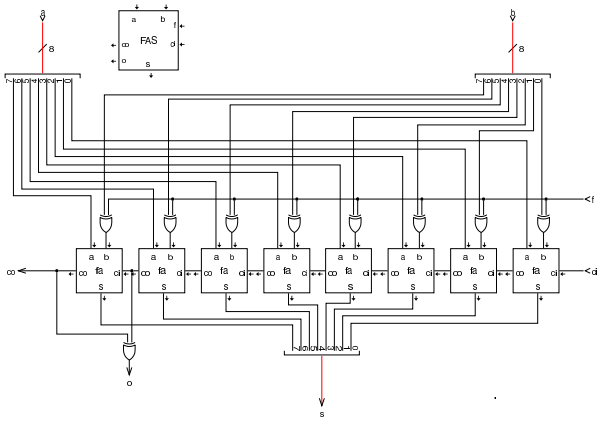


Figura u116.11. Struttura del modulo di scorrimento (**SH**), il quale si occupa anche della rotazione con riporto.

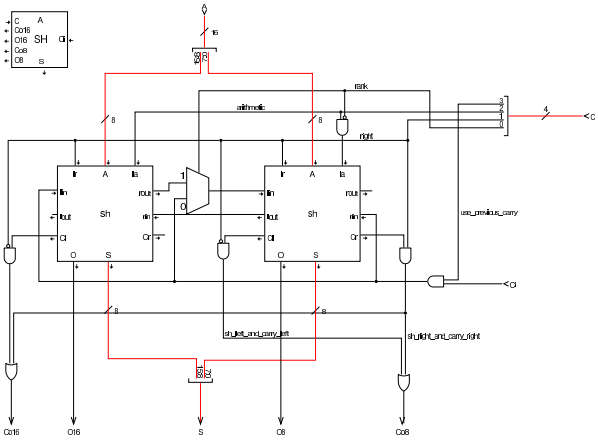


Figura u116.12. Modulo di rotazione (**ROT**): questo modulo esegue esclusivamente la rotazione del contenuto, senza utilizzare il riporto.

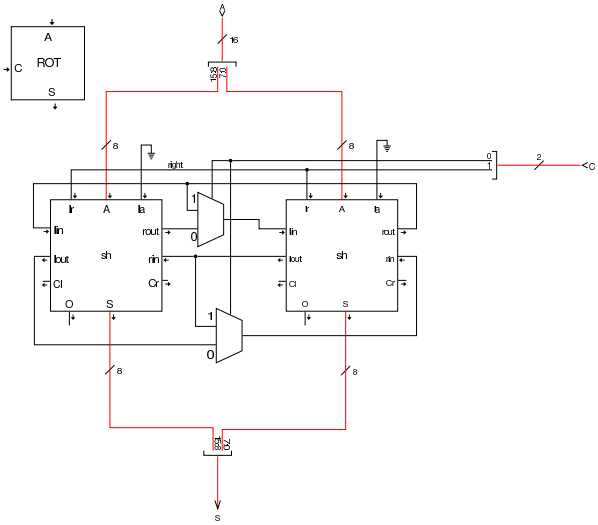
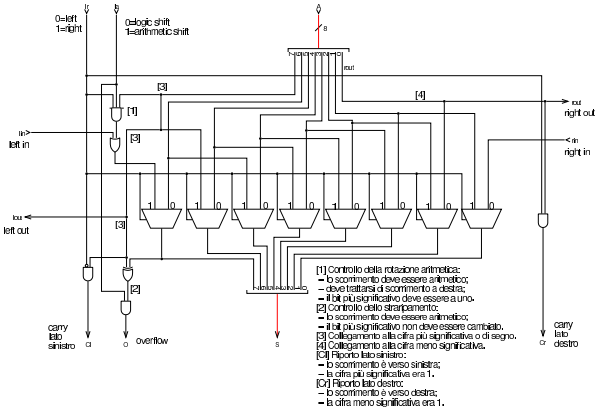


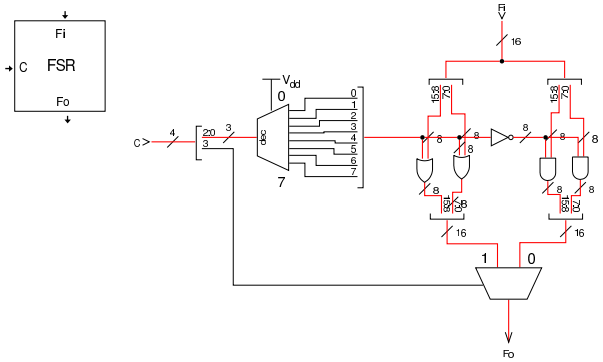
Figura u116.13. Modulo **sh** contenuto nei moduli di scorrimento e di rotazione.



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

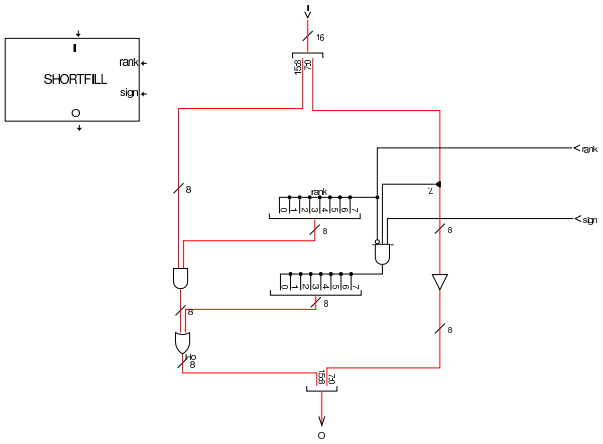
un indicatore di riporto a 8 bit e un altro a 16 bit. Quando si interviene per modificare lo stato degli indicatori, si agisce simultaneamente in entrambi i gruppi, attivandoli o disattivandoli assieme. Il modulo **FAS** riceve quindi una maschera da 8 bit e la funzione da applicare a questa maschera: 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.

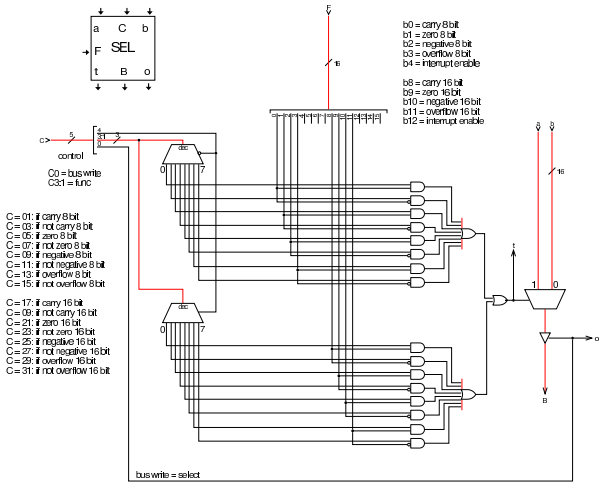
Figura u116.15. Modulo **SHORTFILL**, usato per sistemare il contenuto degli otto byte più significativi, quando si richiede di gestore solo operazioni a otto bit.



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.

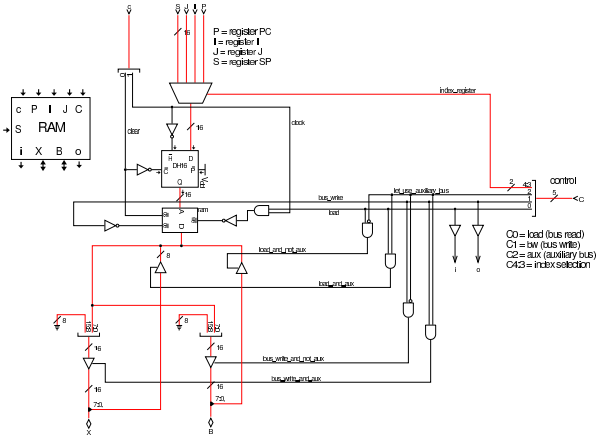
Figura u116.16. Modulo **SEL**.



Modulo «RAM»

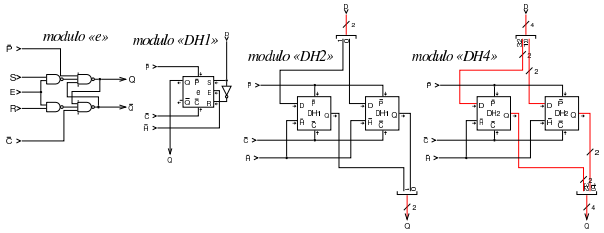
La memoria RAM continua a funzionare a blocchi di otto bit, come avviene nelle architetture comuni. Per leggere o scrivere valori a 16 bit occorre eseguire due operazioni successive; inoltre, tenendo conto che si lavora secondo l'ordine *little endian*, lettura e scrittura partono sempre dal byte meno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

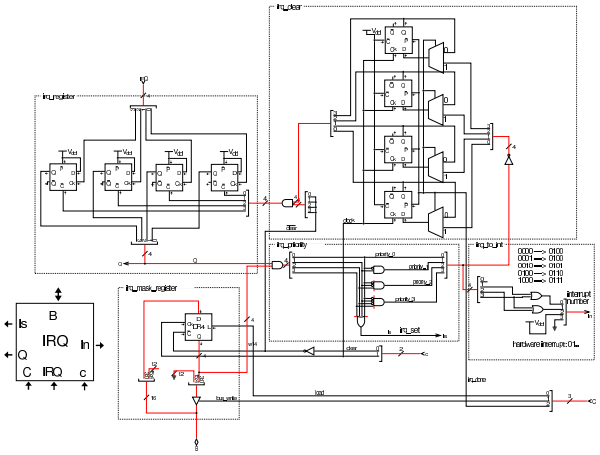
Figura u116.18. Da sinistra a destra, si vedono le fasi realizzative dei moduli **DH**...: si parte da un flip-flop SR con ingresso di abilitazione, quindi si realizza un flip-flop D con ingresso di abilitazione, poi si mettono in parallelo i flip-flop D. Si intende che il registro ***DH16*** è composto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



Modulo «IRQ»

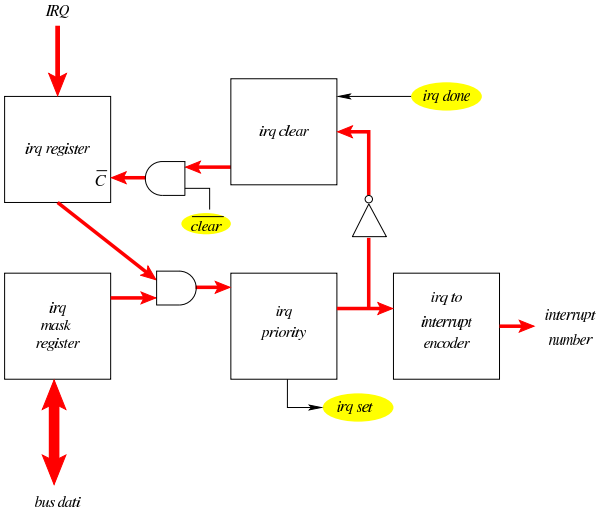
Questa versione della CPU dimostrativa gestisce le interruzioni, distinguendo tra quelle prodotte internamente dalla CPU stessa, quelle provenienti da dispositivi esterni e quelle gestite via software. Il modulo **IRQ** si occupa di ricevere le interruzioni hardware dai dispositivi per fornirle al circuito 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**.



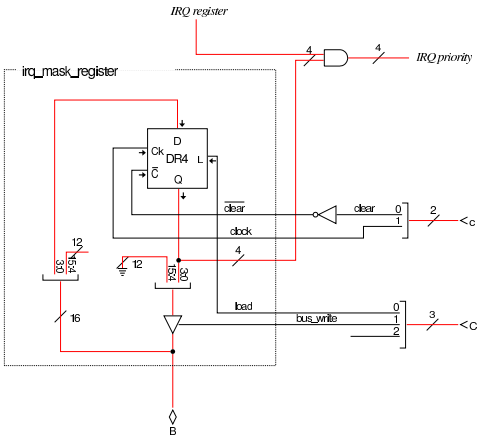
Per poter comprendere cosa fa il modulo **IRQ** è necessario analizzare i suoi vari componenti, con l'aiuto di uno schema a blocchi che riproduce in modo più semplice il suo disegno effettivo. Questo schema a blocchi è visibile nella figura successiva.

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



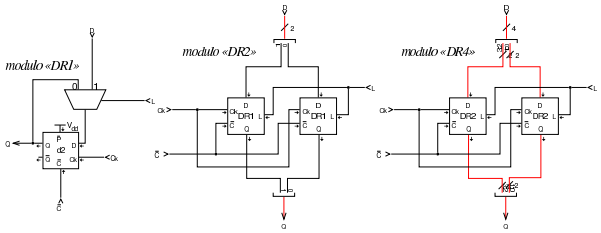
Conviene partire dall'analisi del registro che contiene la maschera degli IRQ ammissibili che appare in basso a sinistra nello schema complessivo: si tratta di un registro a 4 bit (uno per ogni IRQ gestito) che legge dal bus dati per aggiornare il proprio valore e scrive sul bus dati, per consentire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

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



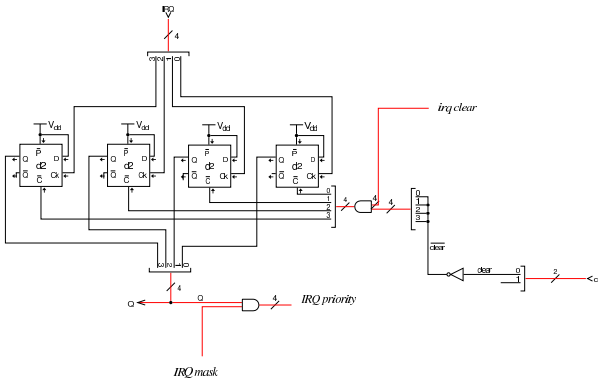
Il modulo **DR4** è un registro a quattro bit, realizzato con flip-flop D, come si vede nella figura successiva, attraverso passaggi successivi.

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'***.

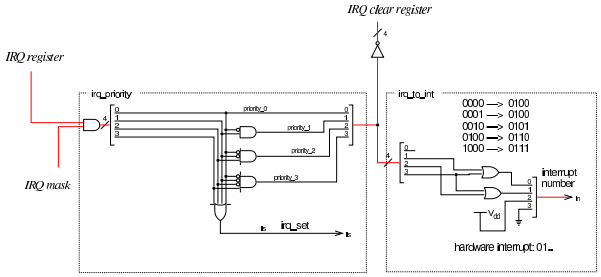
Figura u116.23. Dettaglio del registro degli IRQ ricevuti.



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

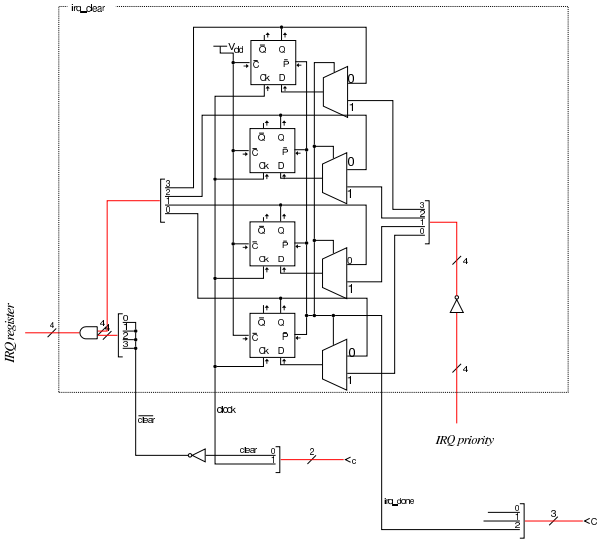
codificatore il valore INT4.

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.



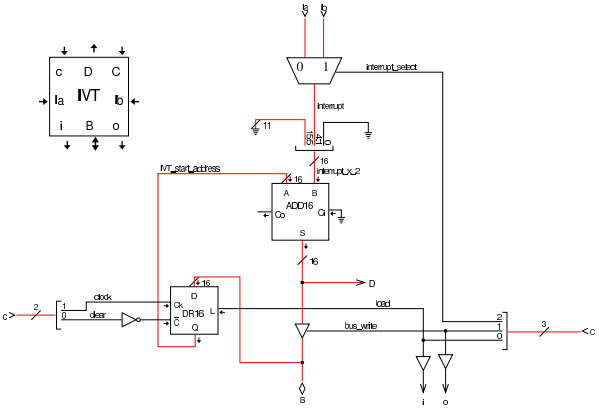
L'azzeramento del registro degli IRQ deve poter avvenire anche simultaneamente per tutti i flip-flop che contiene, pertanto il suo ingresso ***C'*** è collegato con una porta AND che consente di agire in tal modo. Il segnale ***clear'*** risulta come complemento del segnale ***clear*** proveniente dal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

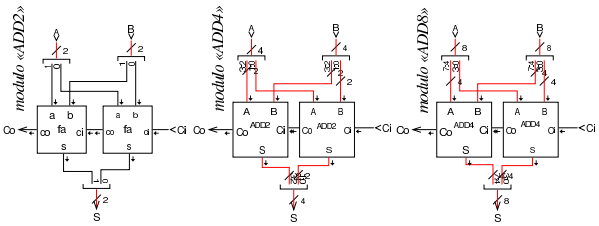
Il registro ***IVT*** serve a memorizzare la collocazione della tabella IVT, corrispondente precisamente a ***IVT[0]***. Da due ingressi indipendenti, il modulo riceve il numero di una certa interruzione, la quale viene trasformata nell'indirizzo corrispondente in memoria che contiene il riferimento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

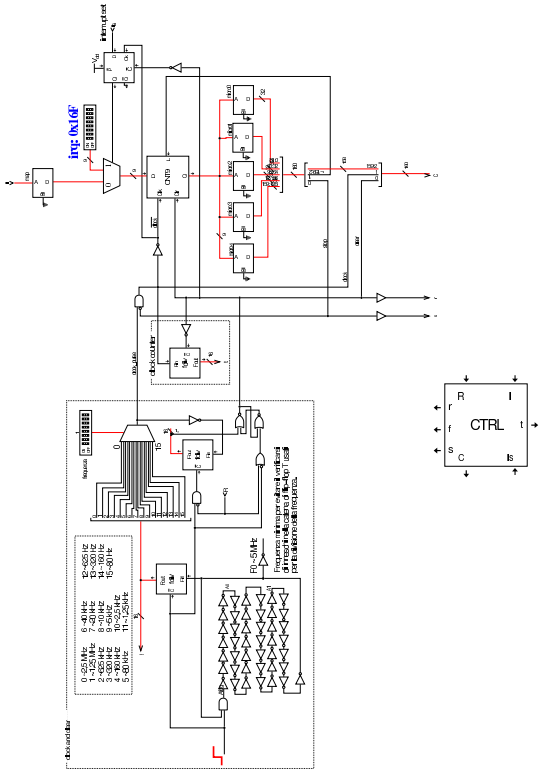
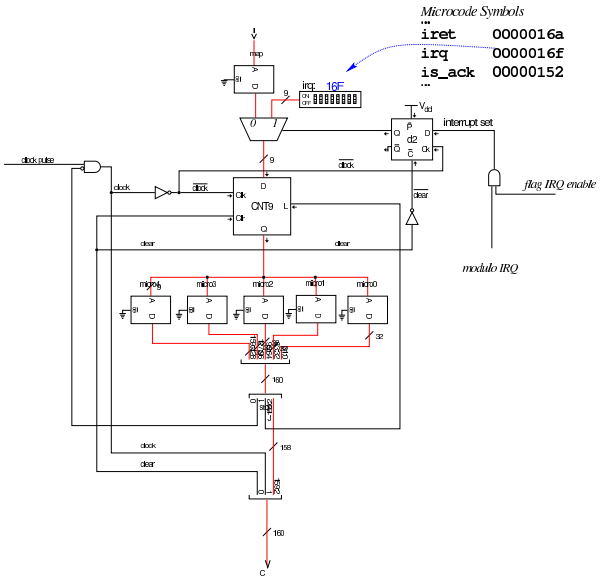


Figura u116.29. Dettaglio dell'unità di controllo che si occupa di gestire il codice.



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.

`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.

Sintassi del macrocodice	Codice operativo binario	Descrizione

`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.

`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***).

`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`	
`jump16c #`***indirizzo***	`01110010`	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.
`jump16nc #`***indirizzo***	`01110011`	
`jump16z #`***indirizzo***	`01110100`	
`jump16nz #`***indirizzo***	`01110101`	
`jump16o #`***indirizzo***	`01110110`	
`jump16no #`***indirizzo***	`01110111`	
`jump16n #`***indirizzo***	`01111000`	
`jump16nn #`***indirizzo***	`01111001`	
`push8 #`***registro***	`10000`***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.
`pop8 #`***registro***	`10001`***rrr***	
`push16 #`***registro***	`10010`***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.
`pop16 #`***registro***	`10011`***rrr***	
`c81016u`	`10100000`	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.
`c81016s`	`10100001`	

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`	
`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`	
`in #`***io***	`11000000`	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.
`out` ***io***	`11000001`	
`is_ack #`***io***`, #`***indirizzo***	`11000010`	

Sintassi del macrocodice	Codice operativo binario	Descrizione
`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***.
`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:

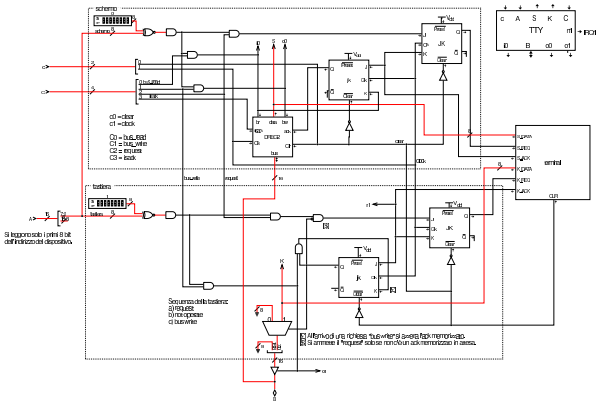
Listato u116.38. Dichiarazione del modulo **`RTC`**.

```
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.

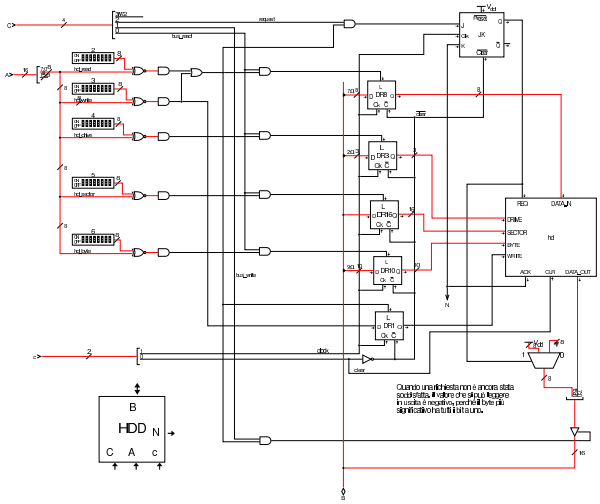
Figura u116.39. Modulo **`TTY`**.



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

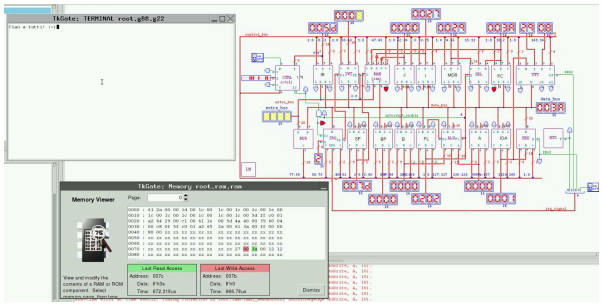
Il codice seguente esegue la lettura della tastiera, attraverso l'interruzione generata dalla stessa, e la rappresentazione del testo digitato attraverso lo schermo. La parte iniziale del codice definisce la collocazione della tabella IVT e del codice associato ai suoi vari elementi.

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*



Riferimenti

- *Tkgate*, *http://www.tkgate.org*
- Albert Paul Malvino, Jerard A. Brown, *Digital Computer Electronics*, Glencoe/Mcgraw-Hill *http://www.amazon.it/Digital-Computer-Electronics-Albert-Malvino/dp/0028005945*
- Olivier Carton, *Circuits et architecture des ordinateurs*, *http://www.liafa.jussieu.fr/~carton/Enseignement/Architecture/archi.pdf*