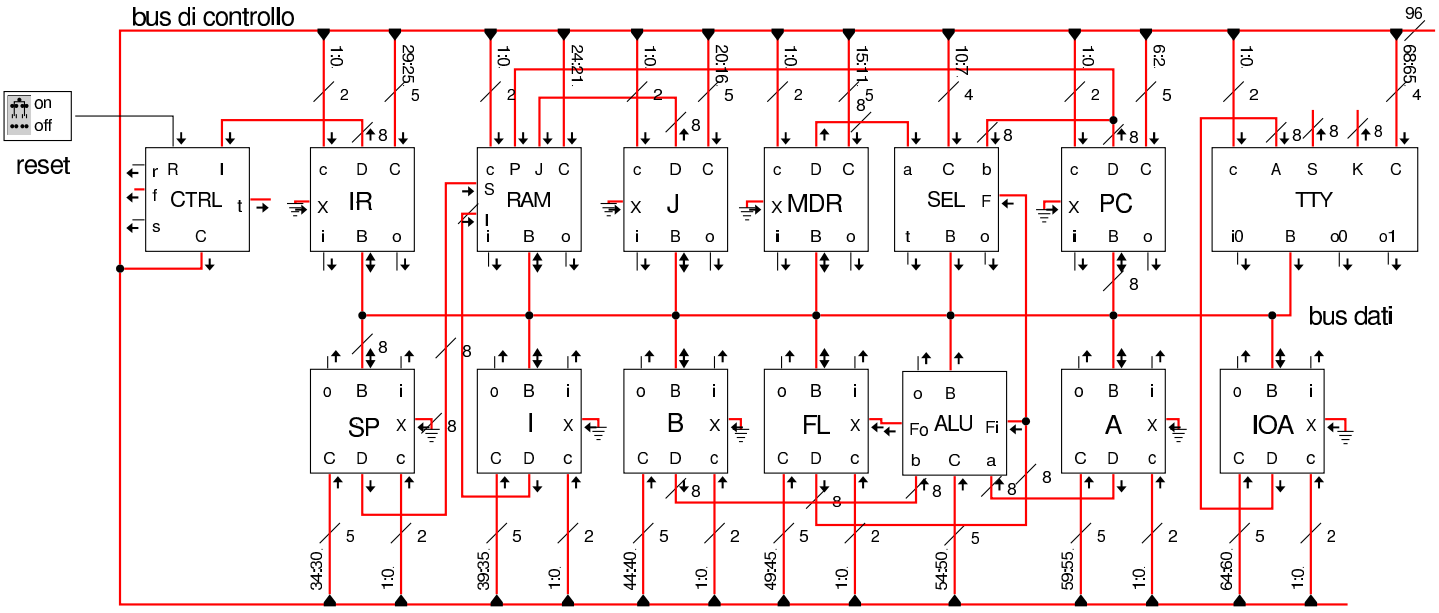
Versione I: ottimizzazione

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.

«a2» 2013.11.11 --- Copyright © Daniele Giacomini -- appunti2@gmail.com http://informaticalibera.net

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.

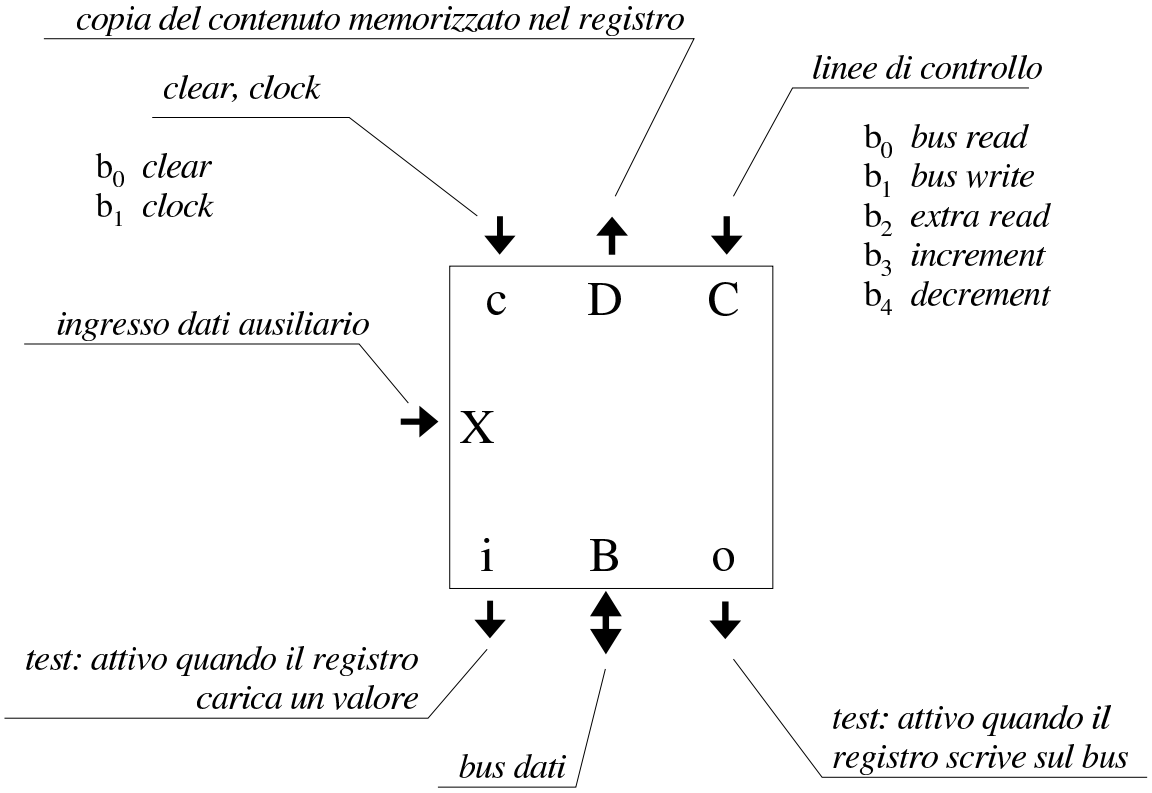
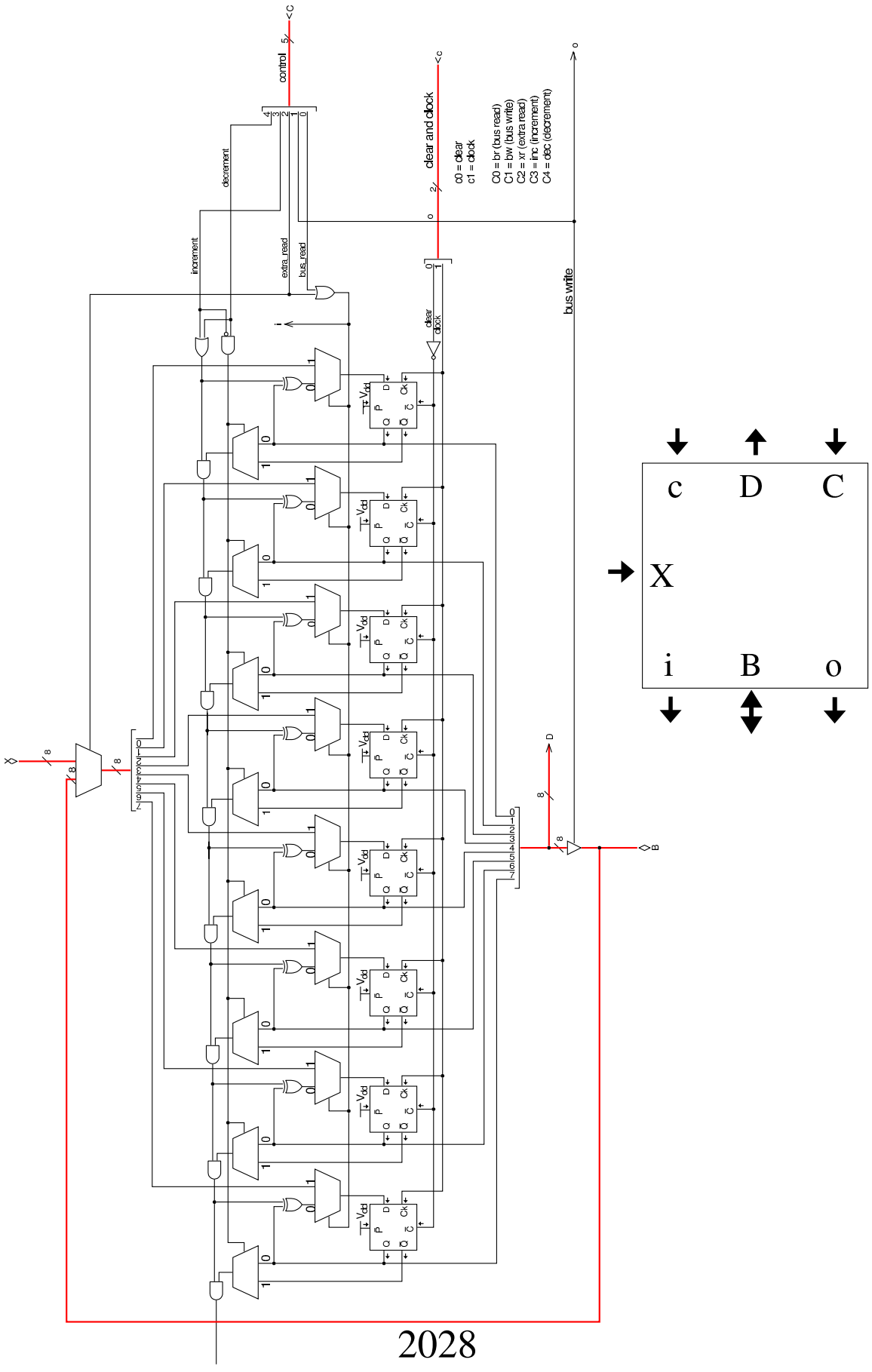
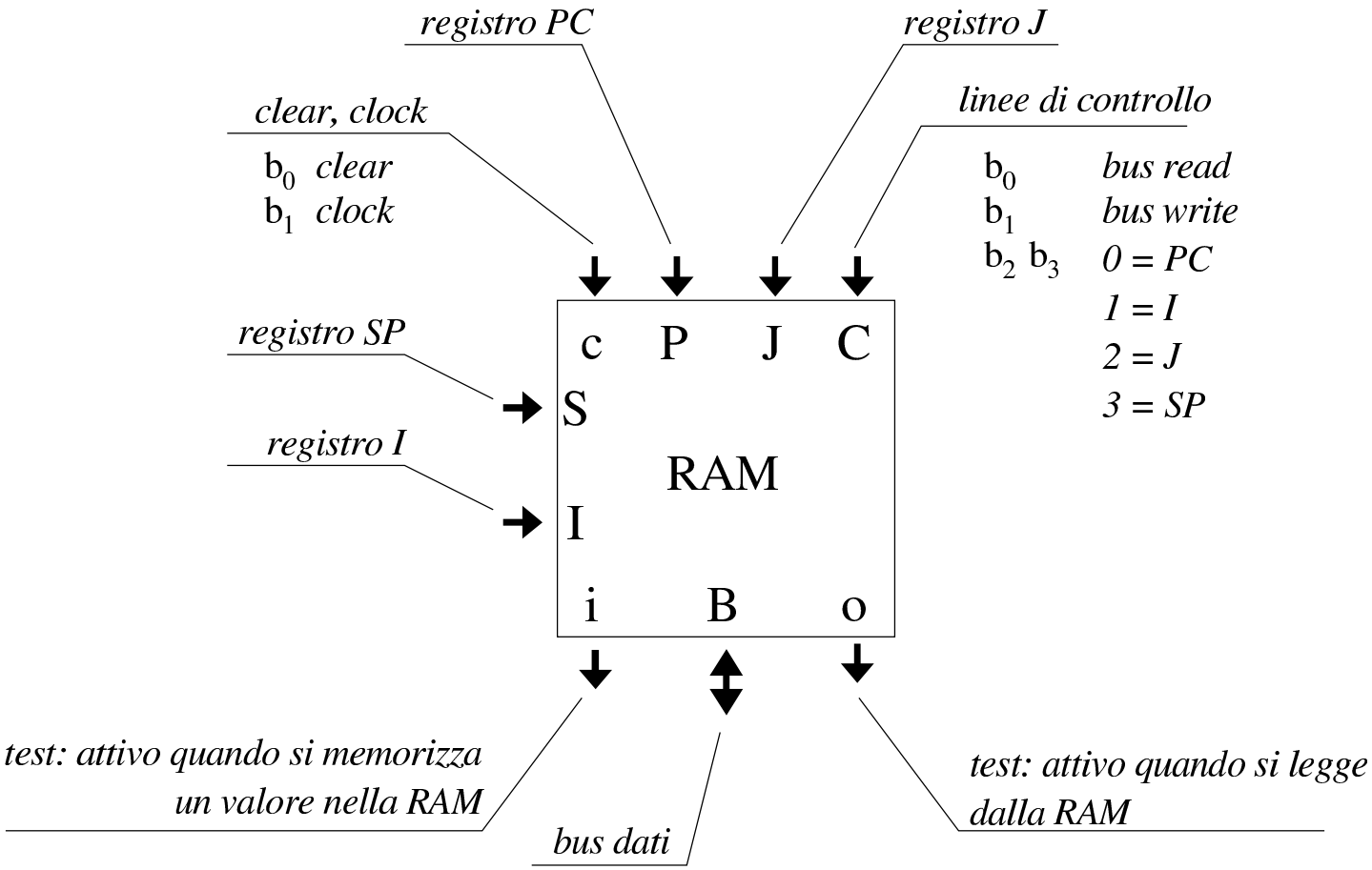


Figura u114.3. Schema interno del registro generalizzato.



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***.

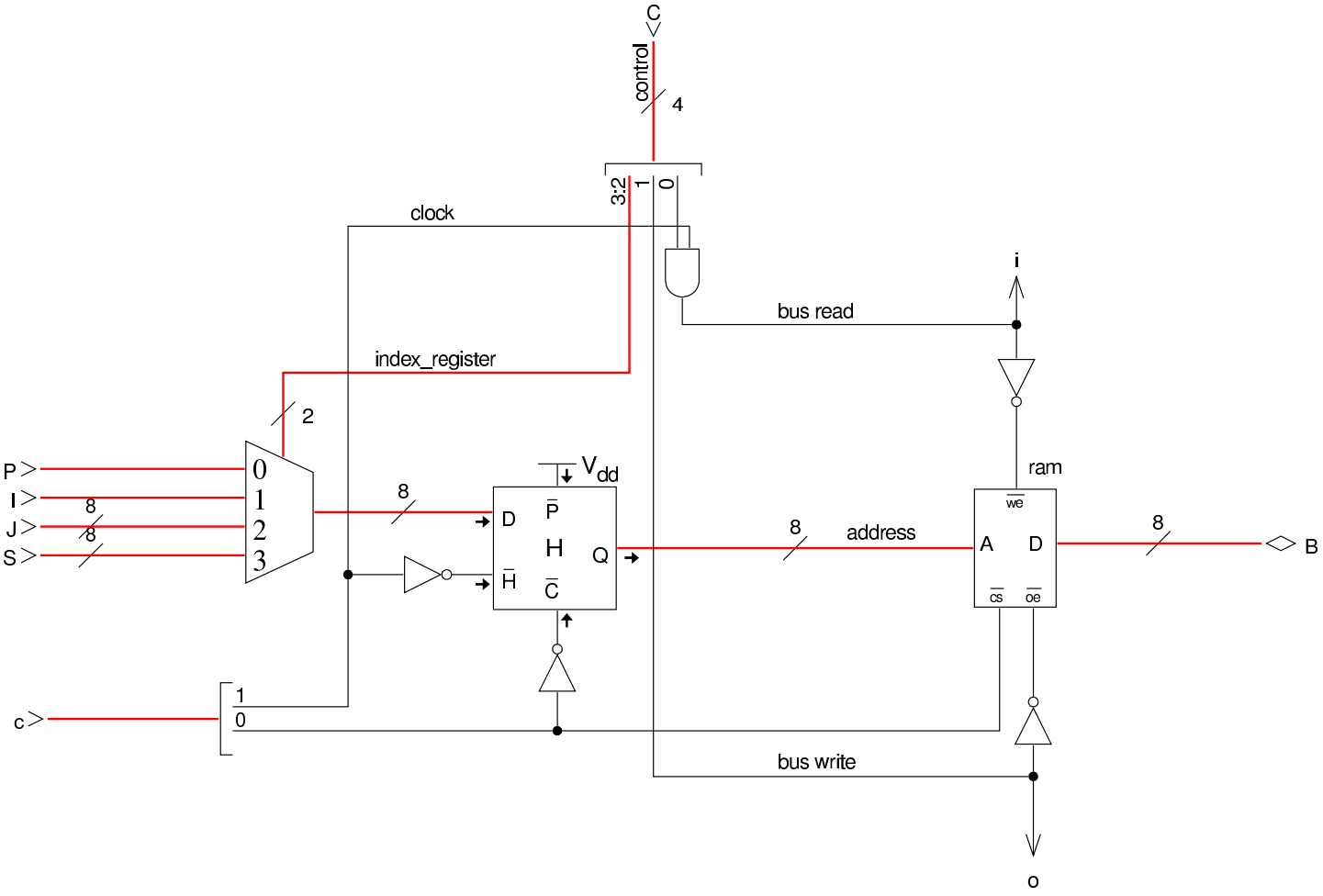
Figura u114.4. Aspetto esterno del modulo **RAM**.



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**.

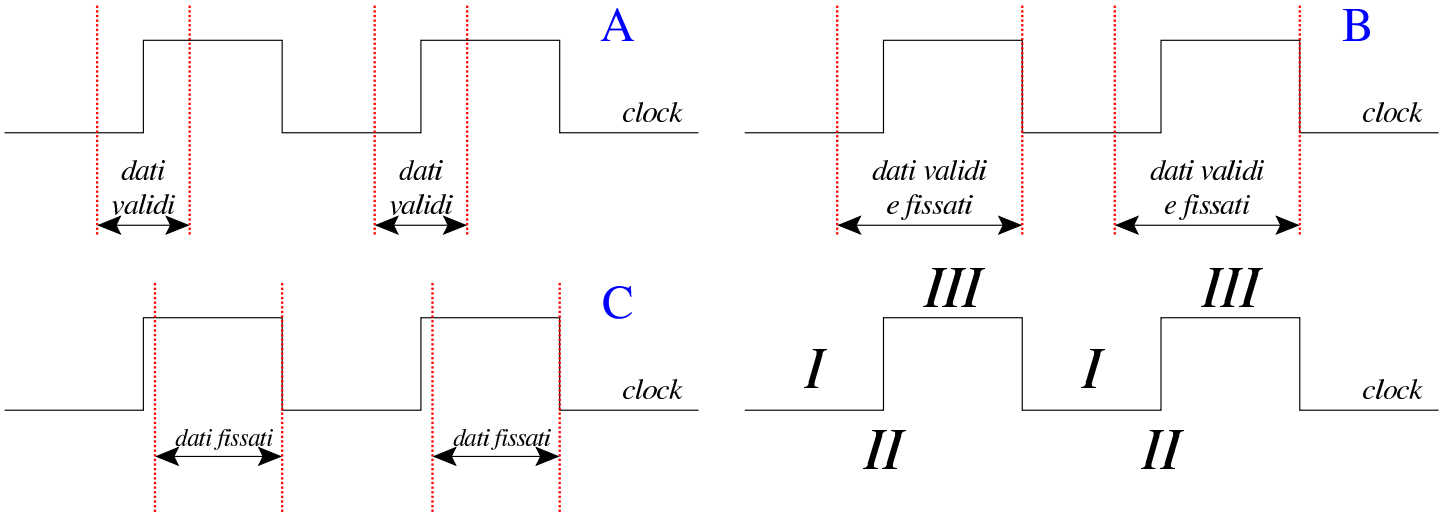


In questa versione della CPU, durante un ciclo di clock, l'indirizzo che serve a individuare la cella di memoria a cui si è interessati, può non essere stabile, a causa di vari fattori. Prima di tutto, l'indirizzo viene scelto attraverso un multiplatore, pescandolo da quattro registri diversi, ma questa 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.

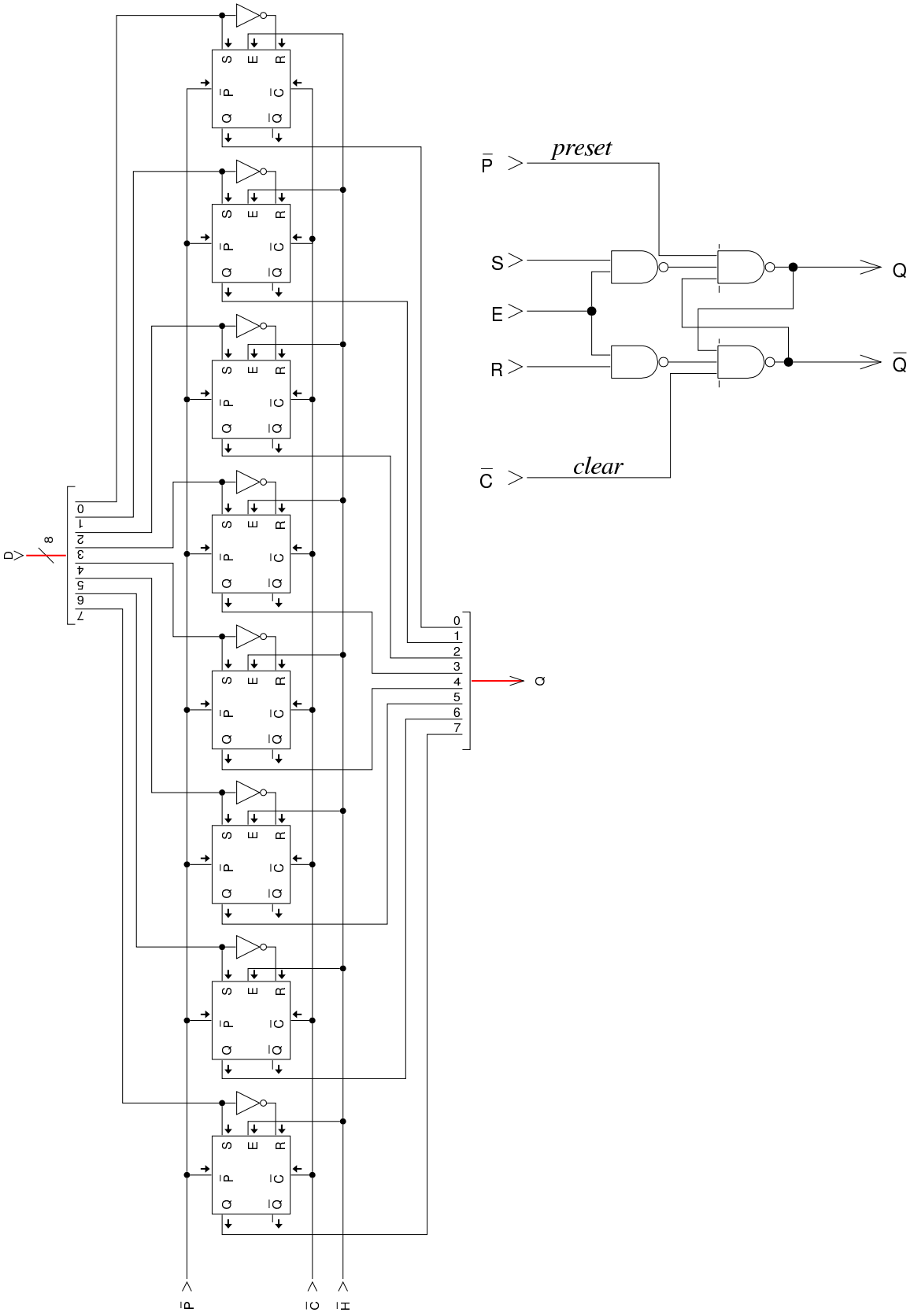
Nella figura, il grafico «A» si riferisce agli intervalli di validità dell'informazione degli indirizzi, a cavallo della variazione positiva del segnale di clock. Il grafico «B» mostra la situazione all'uscita del registro ***H***, che estende la validità dell'indirizzo ricevuto, in ingresso, perché 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.

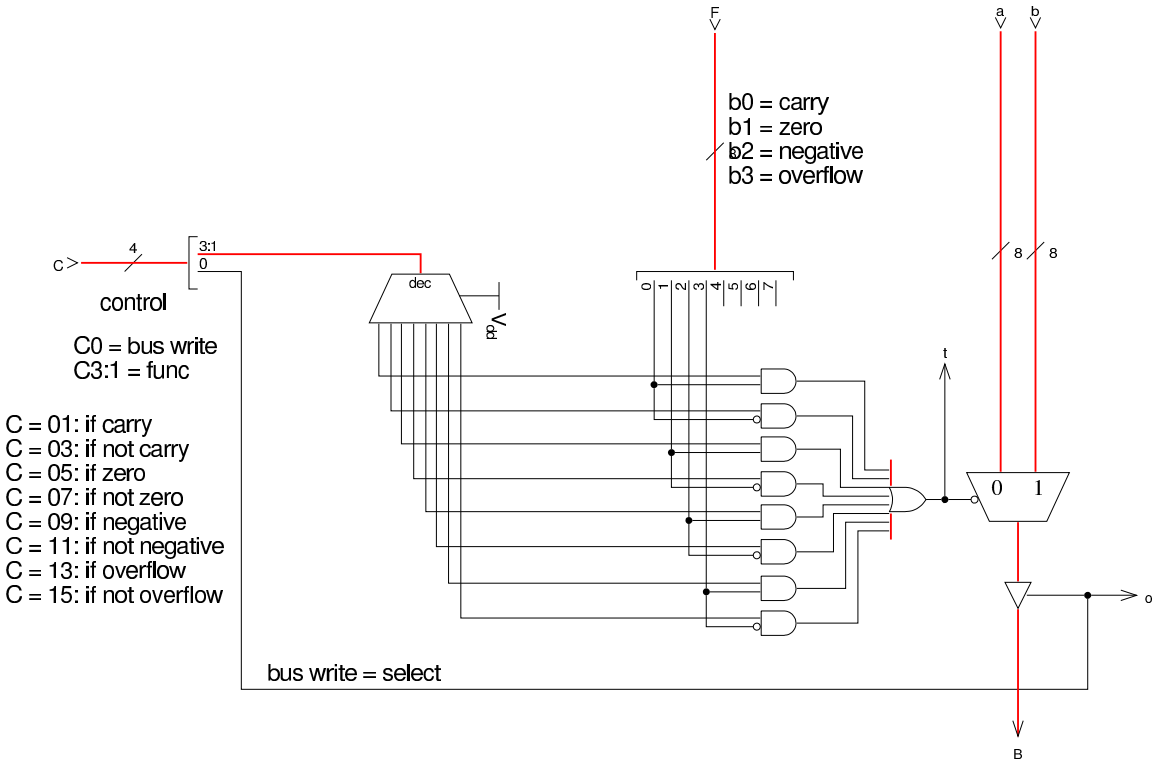
Figura u114.7. Schema interno del registro ***H*** (*hold*), contenuto del modulo **`RAM`**.



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***).

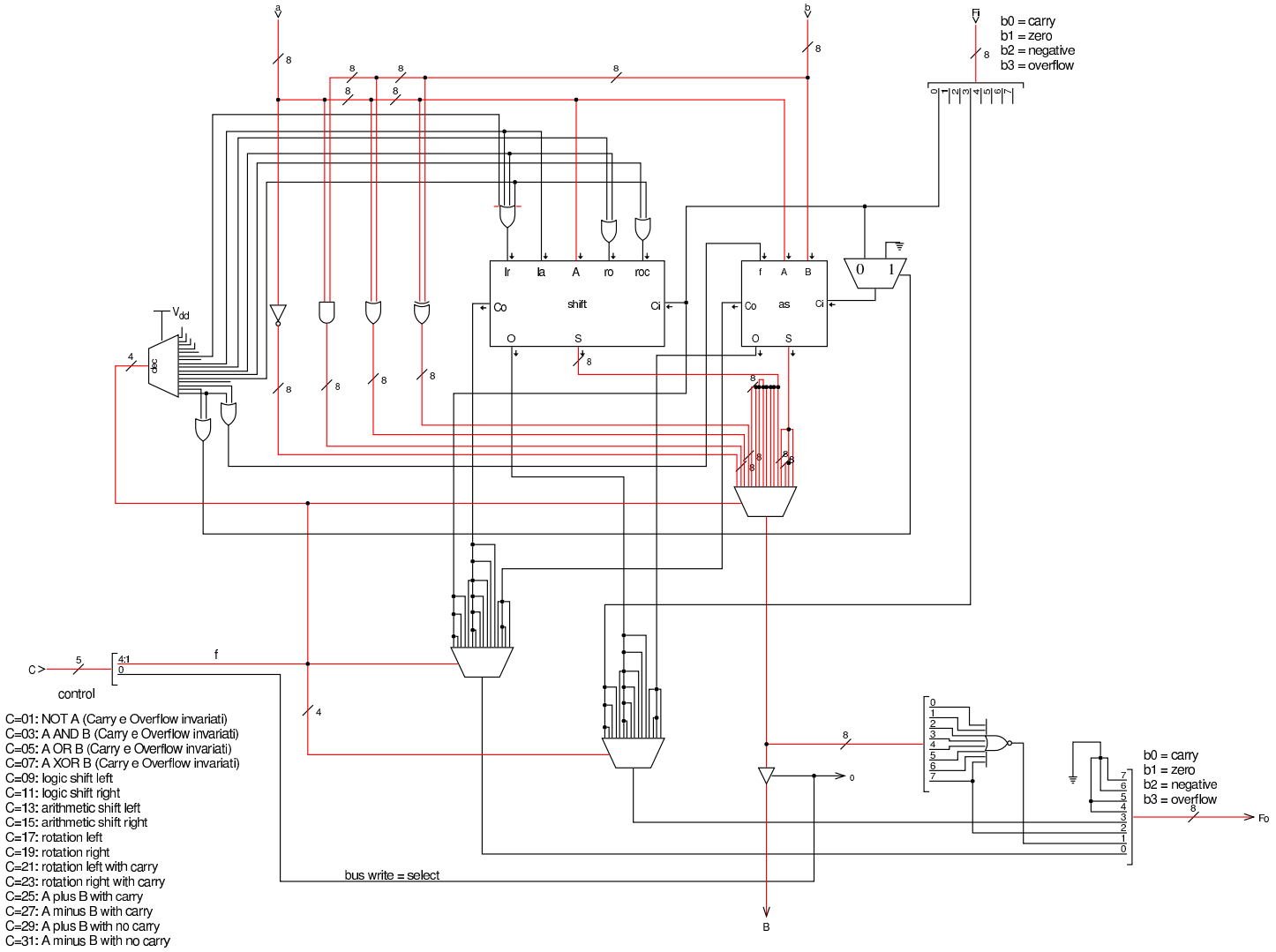
Figura u114.8. Schema interno del modulo **`SEL`**.



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**.

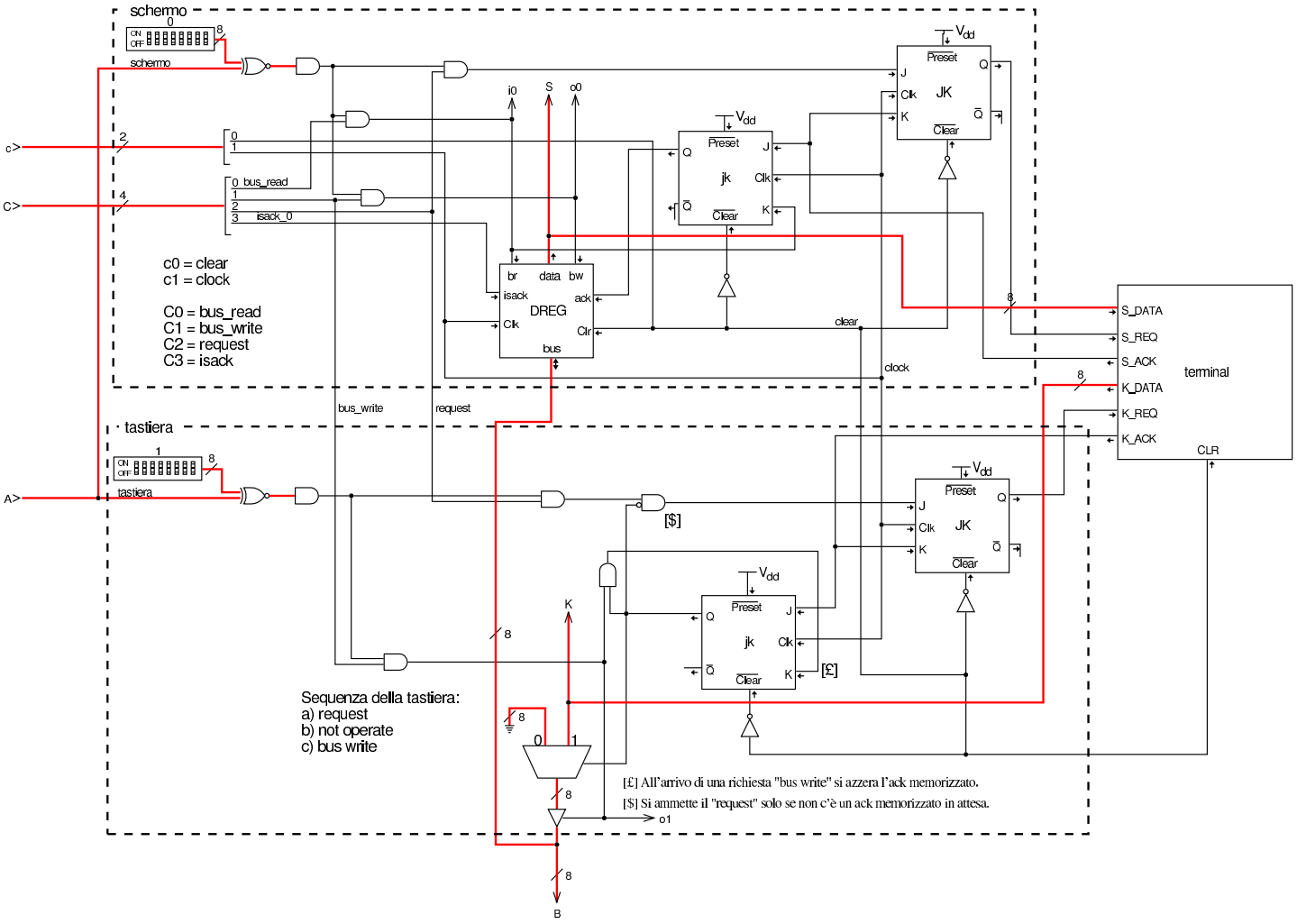


Terminale

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.

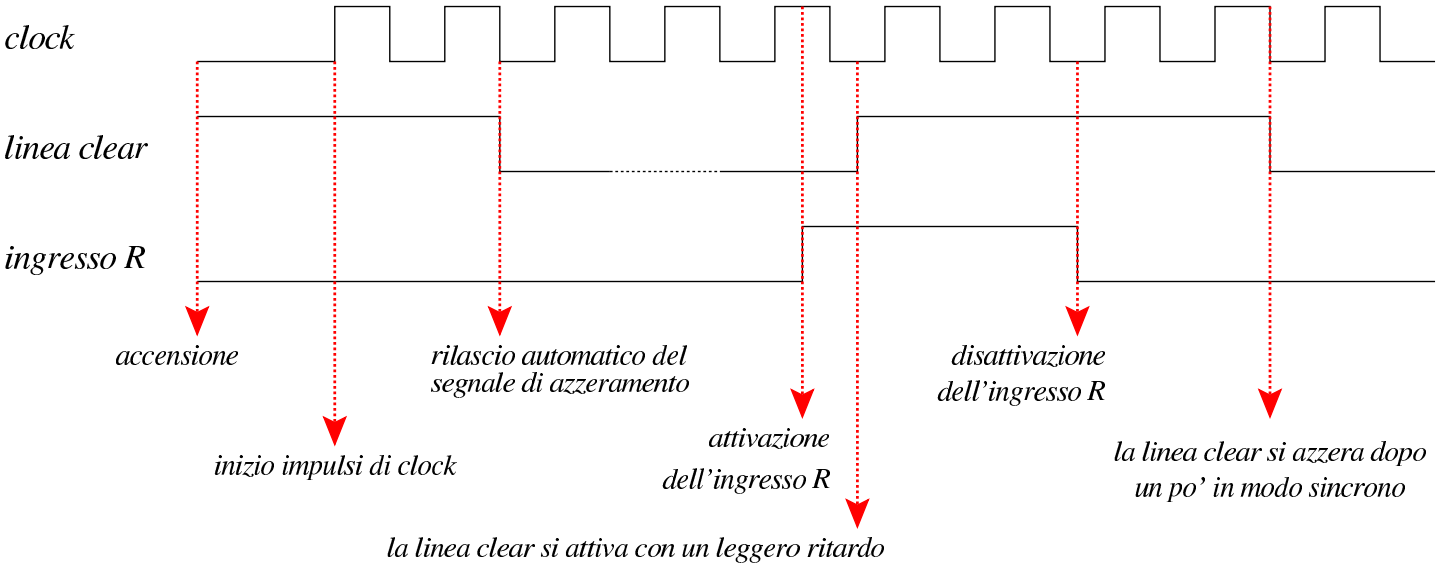
```
image create bitmap txtcurs -file "$bd/txtcurs.b"

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}]
        }
    }
}
```

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.

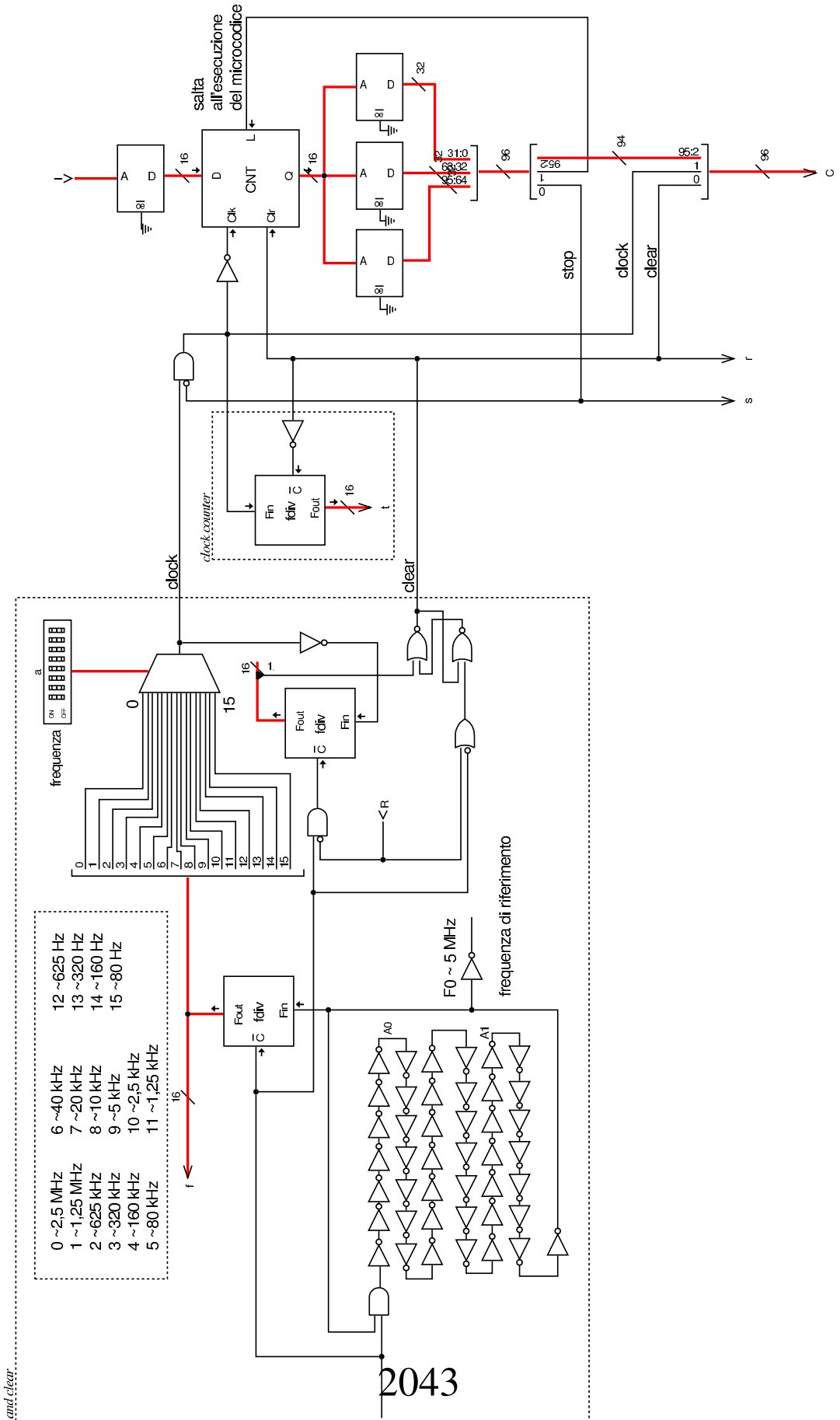
Figura u114.13. Tempistica del funzionamento della linea ***clear***.



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.

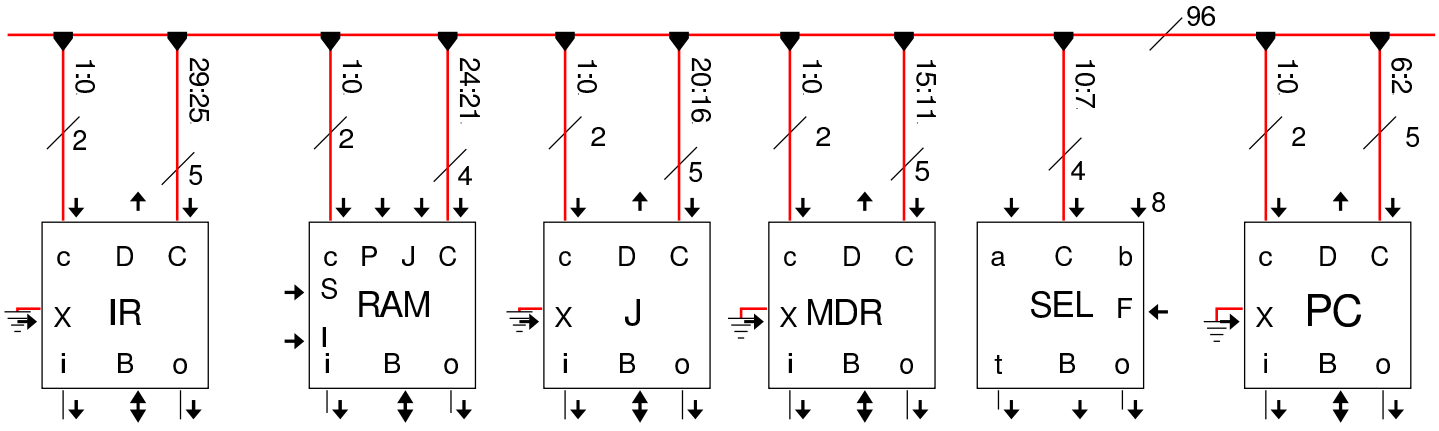
Figura u114.14. Schema completo dell'unità di controllo.



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

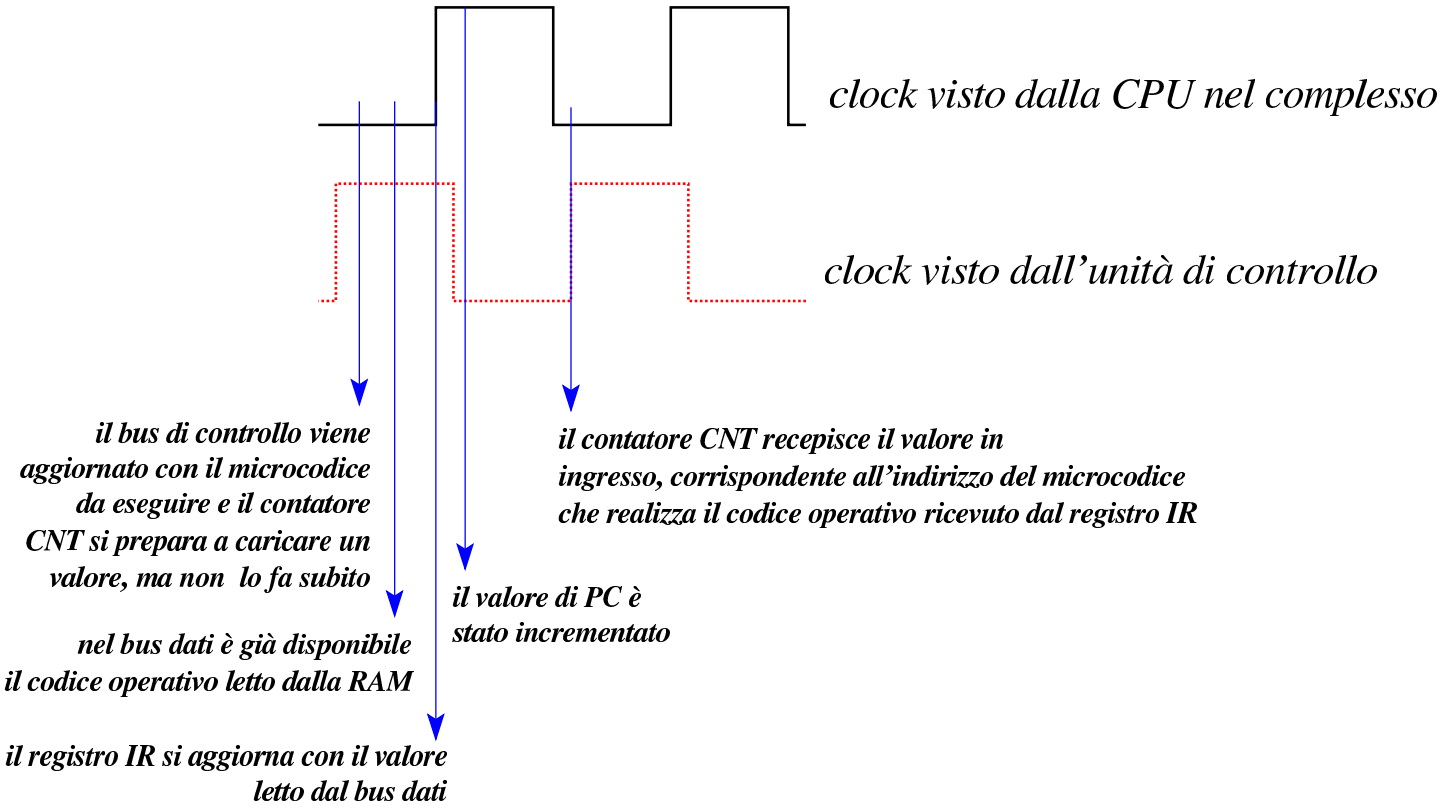
Nella descrizione del microcodice vero e proprio, si inizia con ciò che serve al caricamento del primo codice operativo dalla memoria, ma in questa realizzazione può avvenire tutto in un solo ciclo di clock:

```
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 al-

l'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.
`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***.

Sintassi	Cicli di clock	Descrizione

`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.

Sintassi	Cicli di clock	Descrizione

`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.

Sintassi	Cicli di clock	Descrizione

`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.

`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.

Sintassi	Cicli di clock	Descrizione

`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.

Sintassi	Cicli di clock	Descrizione
`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***.

Sintassi	Cicli di clock	Descrizione

`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.

`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.

Sintassi	Cicli di clock	Descrizione

`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***.

Sintassi	Cicli di clock	Descrizione

`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***.

Sintassi	Cicli di clock	Descrizione

`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***.

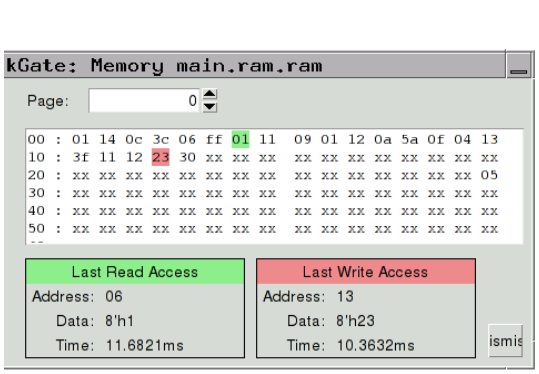
`in` ***indirizzo_io***	5	Legge un valore dal dispositivo di I/O individuato dall'indirizzo fornito, mettendo questo valore nel registro ***A***.
`out` ***indirizzo_io***	4	Scrive un valore sul dispositivo di I/O individuato dall'indirizzo fornito, utilizzando il valore contenuto nel registro ***A***.
`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.

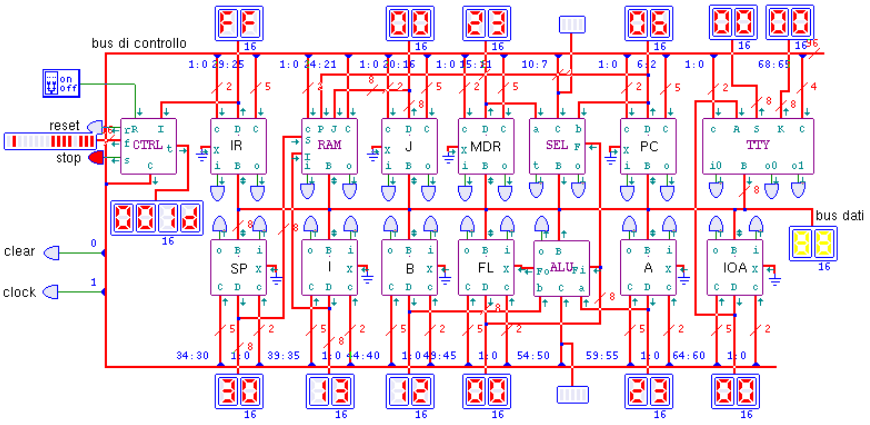
Nelle sezioni successive vengono proposti alcuni esempi per verificare il funzionamento delle macroistruzioni della versione attuale della CPU dimostrativa. Gli esempi sono dimostrati attraverso dei video che mettono in evidenza anche l'accesso alla memoria RAM.

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*