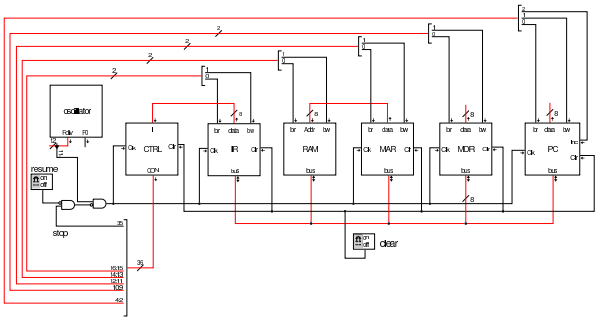
Nella sua prima versione, la CPU si compone soltanto di registri utili ad accedere alla memoria per leggere il codice operativo da eseguire, come di vede nella figura successiva.

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.

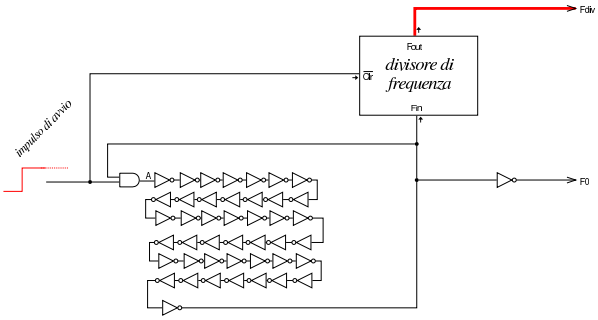
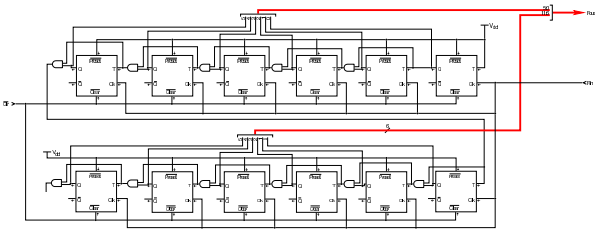


Figura u106.3. Divisore utilizzato nel modulo dell'oscillatore.



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

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

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

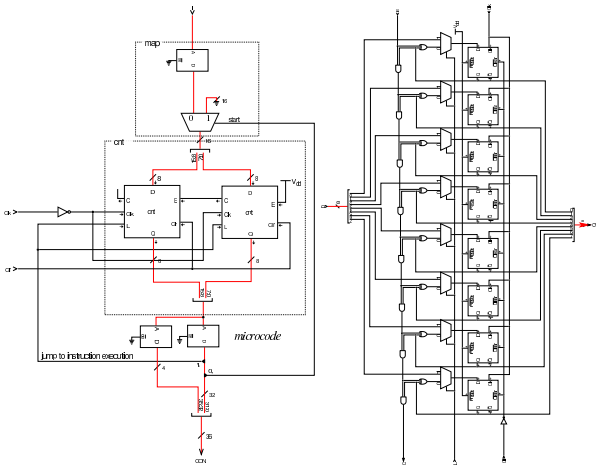
Figura u106.4. Codice Verilog per Tkgate, relativo al modulo di innesco dell'oscillazione: l'uscita è inizialmente a zero e dopo un breve istante passa a uno, rimanendo così stabilmente. Il tempo di attesa iniziale è configurabile attraverso il parametro ***W***.

```
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.

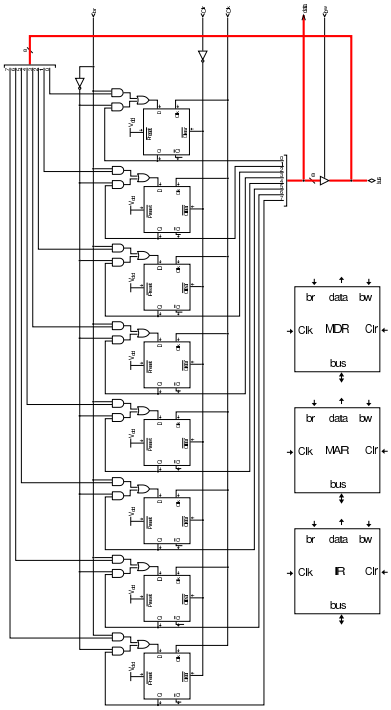
L'ingresso ***I*** dell'unità di controllo è alimentato dal contenuto del registro ***IR*** (*instruction register*).

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.



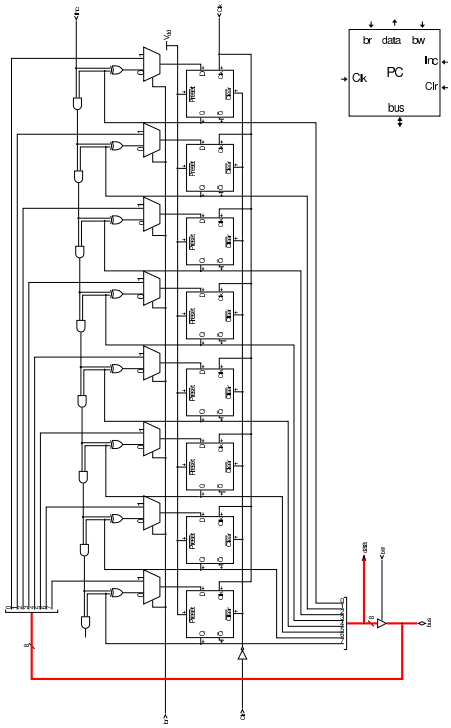
I moduli **`IR`**, **`MAR`** e **`MDR`**, sono registri semplici, costruiti con flip-flop D, connessi al bus attraverso dei buffer a tre stati, dai quali è possibile prelevare copia del valore memorizzato da un'uscita supplementare, denominata ***data***. Il registro ***IR*** (*instruction register*), 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***.

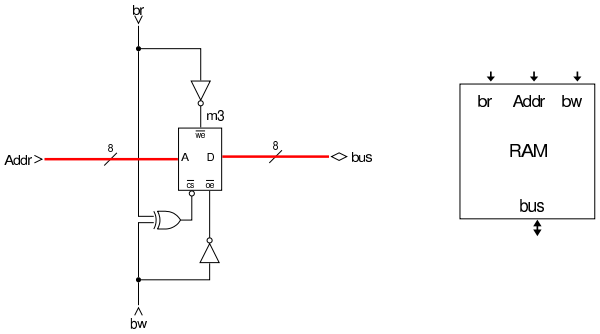


Il modulo **`RAM`** è sostanzialmente differente dagli altri, in quanto racchiude la memoria RAM usata dalla CPU. A tale memoria si accede attraverso l'indirizzo fornito tramite l'ingresso ***Addr***, a 8 bit, e anche il contenuto della memoria è organizzato a celle da 8 bit. Il modulo condivide con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

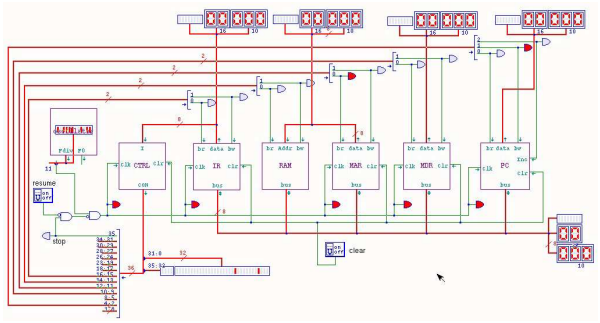


Figura u106.10. Seconda fase: si richiede al registro ***PC*** di incrementarsi di una unità. Si attua in pratica l'operazione ***PC***++.

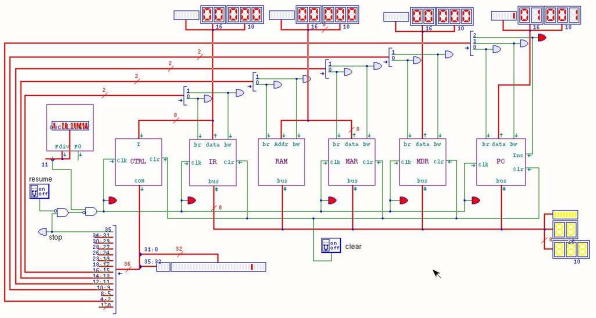
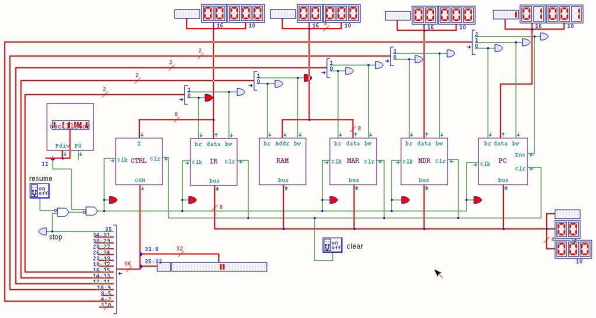


Figura u106.11. Terza fase: si richiede alla RAM di inviare il valore corrispondente all'indirizzo recepito dal registro ***MDR*** al bus e al registro ***IR*** di accumulare questo valore. Si attua in pratica l'operazione ***IR***=***RAM***[***MAR***].



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***.

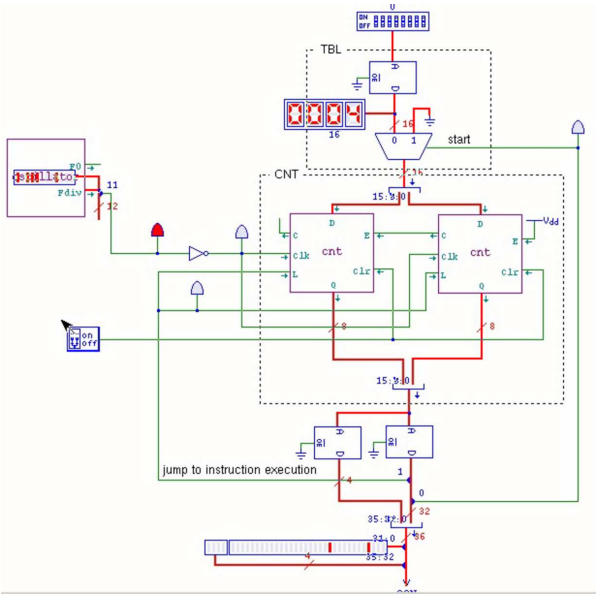


Figura u106.13. Seconda fase: il complesso dei contatori è stato incrementato e la microistruzione prodotta corrisponde a ***PC***++.

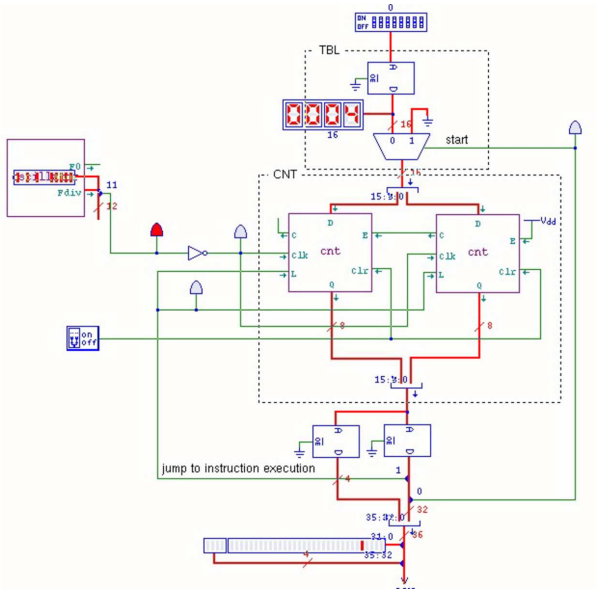
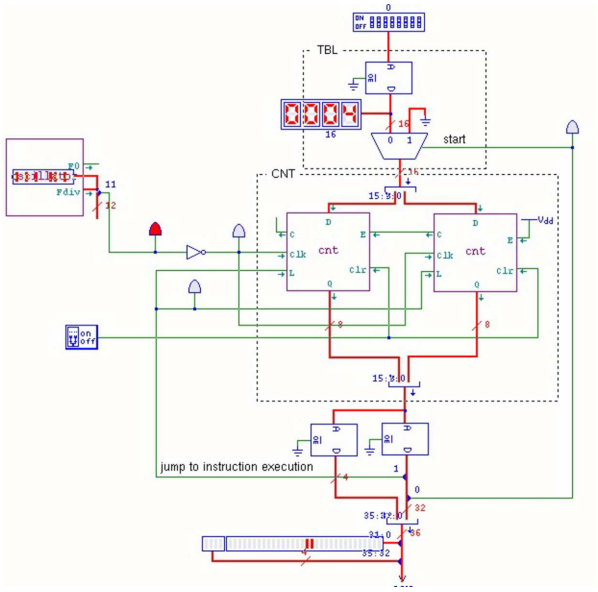


Figura u106.14. Terza fase: il complesso dei contatori è stato incrementato e la microistruzione prodotta corrisponde a ***IR***=***RAM***[***MAR***].



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.

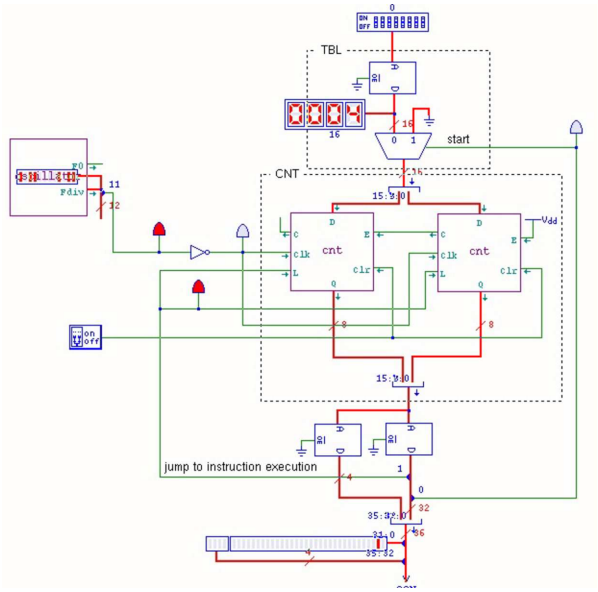
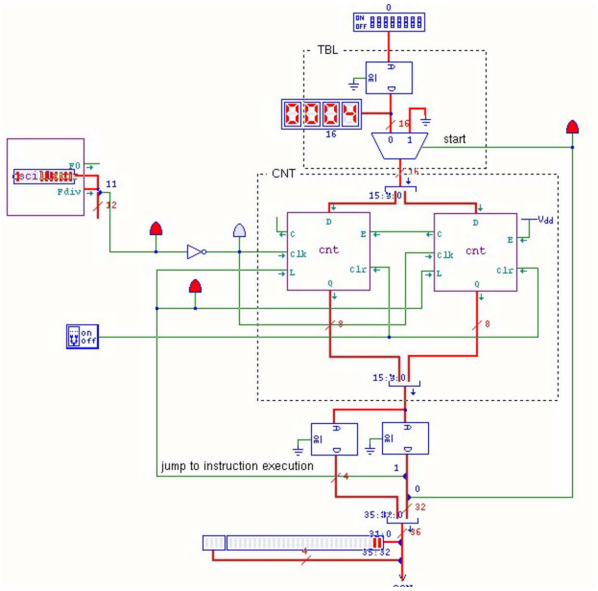


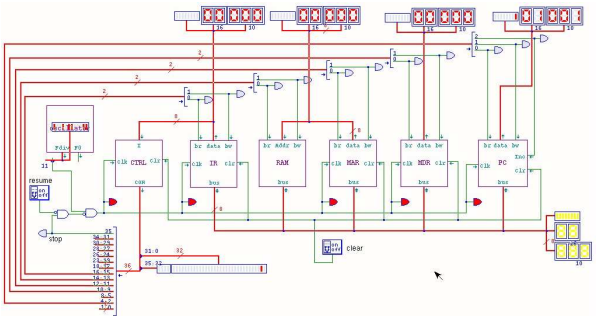
Figura u106.16. Fase conclusiva: i contatori dell'unità di controllo sono stati incrementati e puntano alla microistruzione successiva. Dal momento che l'istruzione originale (**not_operate**) non richiedeva lo svolgimento di alcuna operazione nel bus dati, ci si trova al termine della procedura per 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.

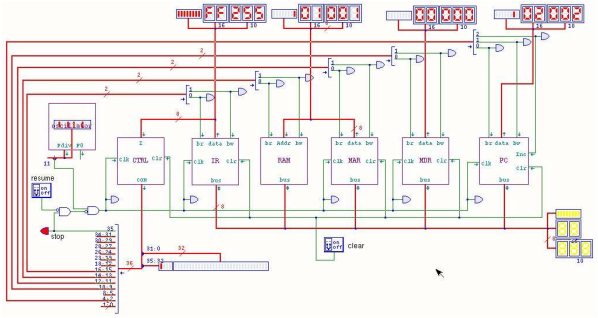
Va osservato che durante la quarta fase (salto al microcodice di esecuzione dell'istruzione richiesta) e durante la fase conclusiva (salto al microcodice iniziale che attua il ciclo di caricamento), nel bus dati non succede nulla.

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):

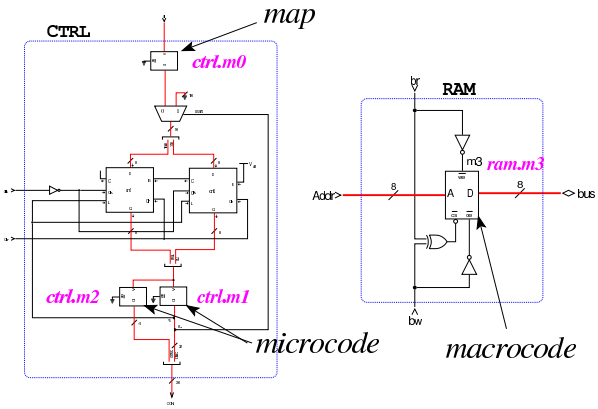
1. **not_operate**
2. **stop**

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;
};
```

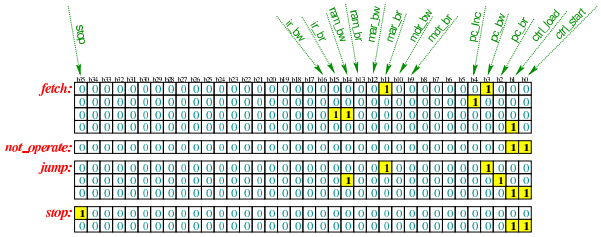
Inizia quindi la definizione del microcodice, il quale viene collocato a partire dall'indirizzo zero della coppia di memorie ***ctrl.m1*** e ***ctrl.m2***. Si può osservare che si inizia proprio dalla descrizione del ciclo di caricamento (*fetch*) che si conclude con il salto alla microistruzione che 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.

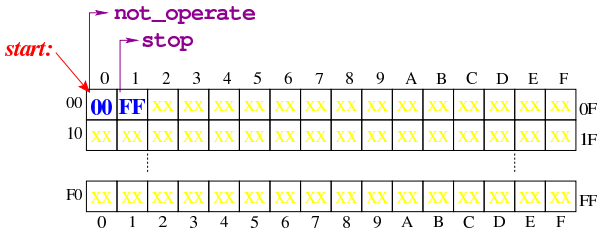


Tabella u106.28. Elenco delle macroistruzioni di questa prima versione della CPU dimostrativa.

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***]

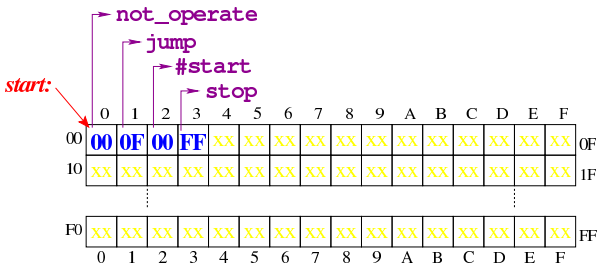
In pratica: nel registro ***MAR*** viene copiato l'indirizzo contenuto nel registro ***PC***, il quale corrisponde all'indirizzo successivo all'istruzione appena letta e in corso di esecuzione (**`jump`**), ma il contenuto della memoria corrispondente a tale indirizzo, viene copiato di nuovo nel registro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

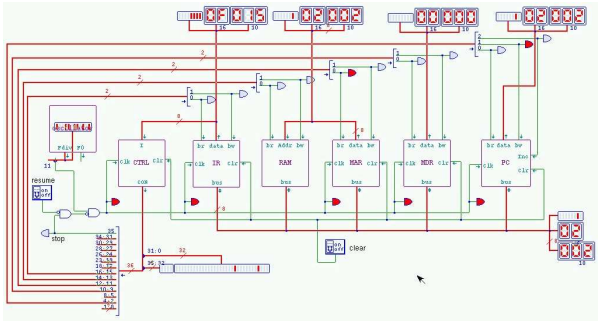
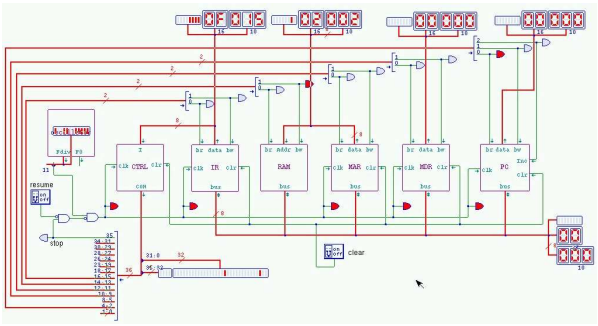


Figura u106.32. Il valore contenuto nella memoria, in corrispondenza dell'indirizzo di salto, viene immesso nel registro ***PC***, facendo in modo che si riparta poi da quella posizione.



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