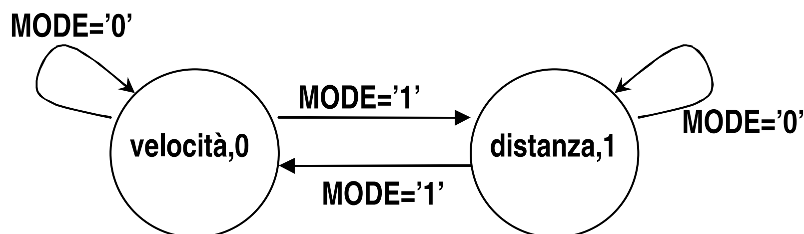
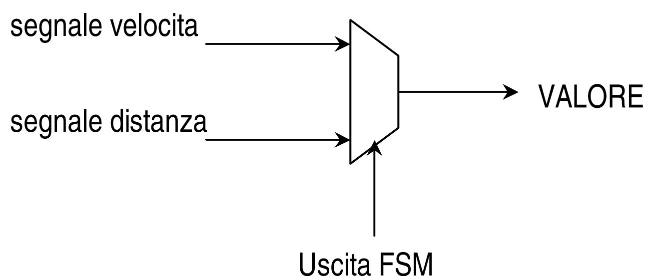
Ciclocomputer. Soluzione Dettagliata.

1. Progetto della rete.
La rete da progettare deve essere in grado di calcolare la distanza percorsa e la velocità del velocipede. Siamo grandemente facilitati dal fatto che gli tutti gli ingressi vengono attivati ogni volta per un solo ciclo di clock. Quindi nessuna oscillazione e nessun impulso di durata sconosciuta sugli ingressi e alla pressione dei pulsanti. Si noti che è una situazione alquanto differente dal mondo reale, in cui queste problematiche vanno gestite opportunamente! (si pensi come, come ulteriore esercizio….)

Il pulsante di RESET deve azzerare il conteggio attuale della distanza. Non è strettamente necessario azzerare la velocità poiché viene aggiornata ogni secondo, purchè si possa tollerare un valore casuale nel primo secondo.

Una pressione del pulsante MODE fa commutare il segnale di uscita VALORE tra la distanza percorsa e la velocità "istantanea". La rete ha dunque "memoria".
Per implementare questo blocco, è lecito immaginare un multiplexer controllato da una piccola macchina a stati. Lo schema di Moore risulta adeguato a questa implementazione.





Senza preoccuparci per ora di come calcolare velocita e distanza potremmo scrivere il seguente codice VHDL.

```
[…]
type modo_funzionamento is (velocita, distanza);
signal current_state, next_state : modo_funzionamento;

signal select_output : std_logic;
[…]

-- macchina a stati per commutare tra distanza e velocita
process(clock, reset)
begin
   if clock'event and clock = '1' then
      if RESET = '1' then
         current_state <= distanza;
      else
         current_state <= next_state;
     end if;
   end if;
end process;

process(current_state, mode)
begin
   case current_state is
      when velocita =>
          if mode = '1' then
             next_state <= distanza;
          else
             next_state <= velocita;
          end if;

          select_output <= '0';


      when distanza =>
          if mode = '1' then
             next_state <= velocita;
          else
             next_state <= distanza;
          end if;

          select_output <= '1';


      when others =>
          next_state <= distanza;
          select_output <= '0';
   end case;
end process;



-- realizzo il multiplexer controllato dalla FSM
VALORE <= segnale_distanza when select_output='1' else segnale_velocita;
```

Nel codice precedente dovremo poi ovviamente sostituire `segnale_distanza` e `segnale_velocita` con dei segnali opportunamente calcolati.

Una prima parte di esercizio è stata svolta. Passiamo ora a calcolare la distanza percorsa, che sembra più facile. Ogni impulso del segnale in ingresso corrisponde ad un giro di ruota. La pressione del tasto RESET azzera il conteggio. L'idea che allora può venire in mente è quella di utilizzare un contatore con enable, dove l'abilitazione viene proprio data dall'ingresso GIROCOMPIUTO, ed il reset viene collegato all'ingresso RESET. Si noti però che cosi si contano i giri, mentre l'uscita deve essere calcolata in metri. E' un dato del problema che 1 giro = 2 m. Dovremo quindi moltiplicare per due il numero di giri. Siamo fortunati, perché per moltiplicare per due basta aggiungere uno '0' a destra del numero binario. E' immediato capire allora che VALORE, quando mostrerà la distanza avrà sempre il bit meno significativo a '0', e che per contare i bit servirà un segnale con ampiezza pari a quella di VALORE meno un bit.

```
[…]
-- uso degli unsigned, poiché devo effettuare delle somme aritmetiche
signal cont_distanza, next_distanza : unsigned(14 downto 0);
[…]

-- contatore della distanza
process(clock, reset)
begin
  if clock'event and clock = '1' then
       if RESET = '1' then
          cont_distanza <= conv_unsigned(0,15);
       elsif GIROCOMPIUTO = '1' then
          cont_distanza <= next_distanza;
       end if;
    end if;
end process;

next_distanza <= cont_distanza + conv_unsigned(1,15);
```

A questo punto il segnale cont_distanza rappresenta il numero di giri di ruota effettuati dall'ultima pressione del tasto RESET, e andrà moltiplicato per due.

Passiamo ora a calcolare la velocità. E' un po' più difficile. La velocità da calcolare non è proprio una velocità istantanea, poiché viene aggiornata ogni secondo. Allora basta contare quanti giri vengono fatti in un secondo, e poi ricominciare.

Potremmo usare un contatore con enable, abilitato da GIROCOMPIUTO, dove il reset viene controllato da un segnale periodico con frequenza pari a 1 secondo, così da azzerarlo. Ipotizziamo di avere questo segnale, chiamiamolo `wave1Hz`. Esso dovrà restare attivo un ciclo ogni secondo.

In questo modo alla fine del secondo il contatore memorizza la velocità espressa in giri/sec.

```
[…]
signal giri_sec, next_giri_sec: unsigned(14 downto 0);
[…]
-- contatore della velocita, ogni secondo azzero il conteggio
process(clock, reset)
begin
  if clock'event and clock = '1' then
       if reset = '1' or wave1Hz = '1' then
          giri_sec <= conv_unsigned(0,15);
       elsif girocompiuto = '1' then
          giri_sec <= next_giri_sec;
       end if;
    end if;
end process;
```

```
next_giri_sec <= giri_sec + conv_unsigned(1,15);
```

Tuttavia, ad ogni giro di ruota il valore dell'uscita cambia, ed il valore corretto si avrà solo allo scadere di ogni secondo. Le specifiche dicono che la velocità calcolata deve restare stabile fino al secondo successivo. Questo è facile, basta campionare il segnale `giri_sec` con un registro con enable, dove l'abilitazione è data dal segnale `wave1Hz`.

```
[…]
signal giri_sec_frozen: unsigned(14 downto 0);
[…]
-- ogni secondo vado a campionare i giri calcolati
process(clock, reset)
begin
  if clock'event and clock = '1' then
        if reset = '1' then
           giri_sec_frozen <= conv_unsigned(0,15);
        elsif wave1Hz = '1' then
           giri_sec_frozen <= giri_sec;
        end if;
     end if;
end process;
```

Noto inoltre che la velocità in m/sec andrà ottenuta moltiplicando per due. Quindi anche in questo caso VALORE dovrà avere il bit meno significativo a '0'.

Possiamo completare il codice del multiplexer nel seguente modo:

```
-- seleziono il valore da portare in uscita.
-- Devo convertire da giri a metri,
-- ma essendo 1 giro = 2 metri, basta shiftare
-- a sinistra di 1;
valore(0) <= '0';
valore(15 downto 1) <= cont_distanza when select_output = '0' else
                       giri_sec_frozen;
```

Ci manca ancora da realizzare il segnale `wave1Hz`, che ha frequenza 1 secondo ed è attivo per un ciclo. La frequenza di clock di sistema è data, ed è pari a:

$$f_{CK} = 10kHZ$$

Risulta quindi:

$$T_{CK} = 100\mu s$$

Quanti cicli dovremo contare per arrivare ad 1 sec?

$$Ncicli = \frac{1s}{100\mu s} = 10000$$

Possiamo utilizzare un contatore, che conta ininterrottamente da 0 a 9999 (10000 cicli), dopodiché ricomincia. Come farlo ricominciare una volta arrivato a 9999? Beh… per esempio agendo sul calcolo combinatorio del valore di conteggio futuro….

[…]
```
	-- uso 14 bit perchè maxint = 2^14-1 = 16383
	signal cicli, next_cicli : unsigned(13 downto 0);
	[…]
	process(clock, reset)
	begin
	  if clock'event and clock = '1' then
	        if RESET = '1' then
	           cicli <= conv_unsigned(0,14);
	        else
	           cicli <= next_cicli;
	        end if;
	     end if;
	end process;

	next_cicli <= conv_unsigned(0,14) when cicli = conv_unsigned(9999,14) else
	              cicli + conv_unsigned(1,14);
```

Ora è sufficiente selezionare un ciclo fra i 10000 del contatore ed attivare l'uscita in questo modo:

```
	wave1Hz <= '1' when cicli = conv_unsigned(0,14) else '0';
```

Il segnale wave1Hz sarà attivo ogni volta che il contatore ha la sua uscita a 0.

Tuttavia, poiché dovrò simulare il circuito per verificarne il circuito e simulare 10000 cicli sarebbe inutile e impegnerebbe risorse in maniera ingiustificata, è lecito effettuare questa piccola modifica, motivandola opportunamente nel codice:

```
	next_cicli <= conv_unsigned(0,14) when cicli = conv_unsigned(9,14) else
	              cicli + conv_unsigned(1,14);
	-- scrivo 9 invece di 9999 per agevolare la fase
	-- di simulazione e verifica!
```

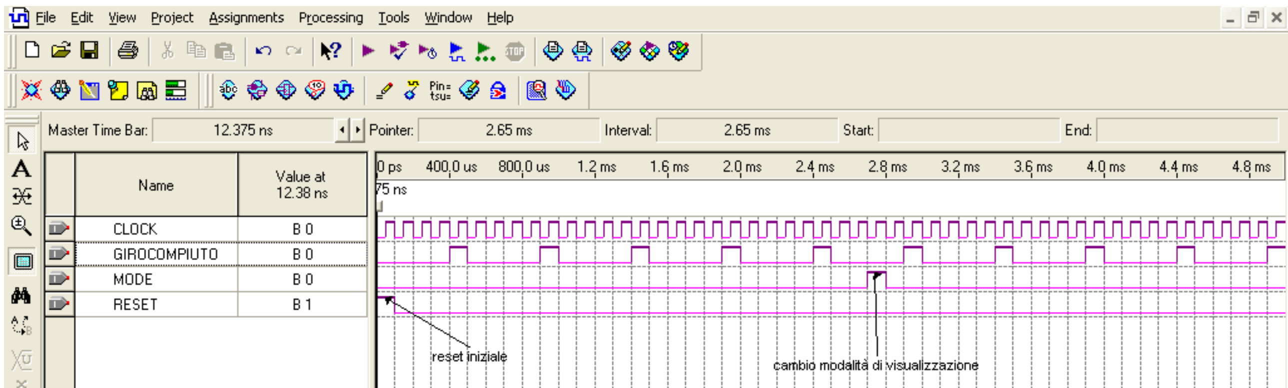
2. Simulazione funzionale.

Impostato un nuovo progetto in Quartus, e scritto il codice VHDL, andiamo a vedere se il nostro circuito funziona.
Per prima cosa compiliamo il circuito. Se tutto va bene procediamo.
Impostiamo la modalità di simulazione su *Functional,* e aggiungiamo per cominciare i pin di ingresso e uscita del dispositivo.

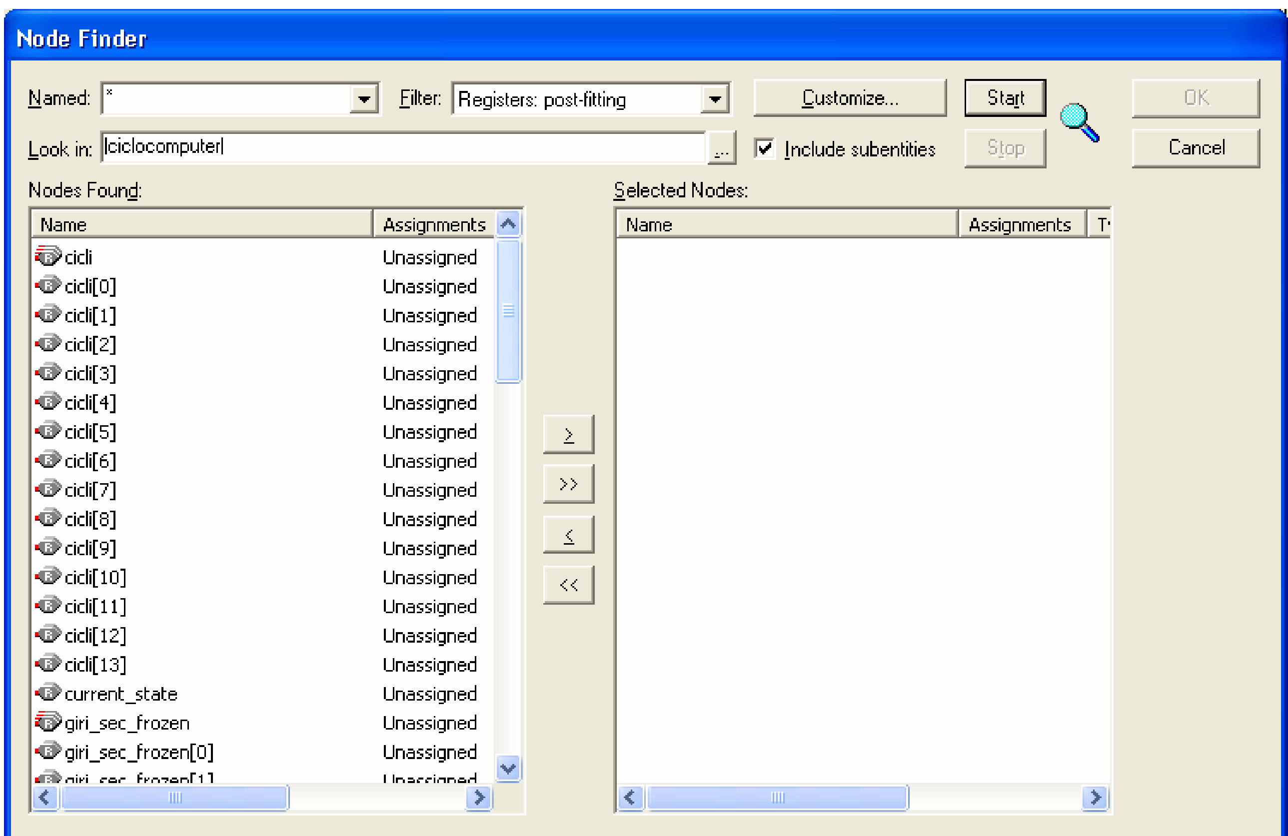
Ora impostiamo la risoluzione del simulatore, e l'istante finale di simulazione. E' inutile discretizzare al ns se il clock è dell'ordine del kHz! La risoluzione si imposta in termini di massima frequenza (o minimo periodo) di clock con essa rappresentabile.
Scegliamo l'istante finale. Ogni 10 cicli calcolo la velocità, supponiamo di voler visualizzare diversi valori di velocità, oltre a qualche ciclo per il reset iniziale, e per un reset intermedio per verificare se la distanza si azzera. Quindi consideriamo circa 50 cicli, che corrispondono a circa 10 calcoli di velocità.
Scegliamo allora una risoluzione compatibile con i 10kHz del clock:
Edit, End Time.
e impostiamo 50*Tck = 5000us
Ora selezioniamo una risoluzione adeguata al nostro clock di 100us:
Edit, Grid Size, e specifichiamo 100us

Ora aggiungiamo il clock sul segnale CLOCK, e pilotiamo il segnale RESET.

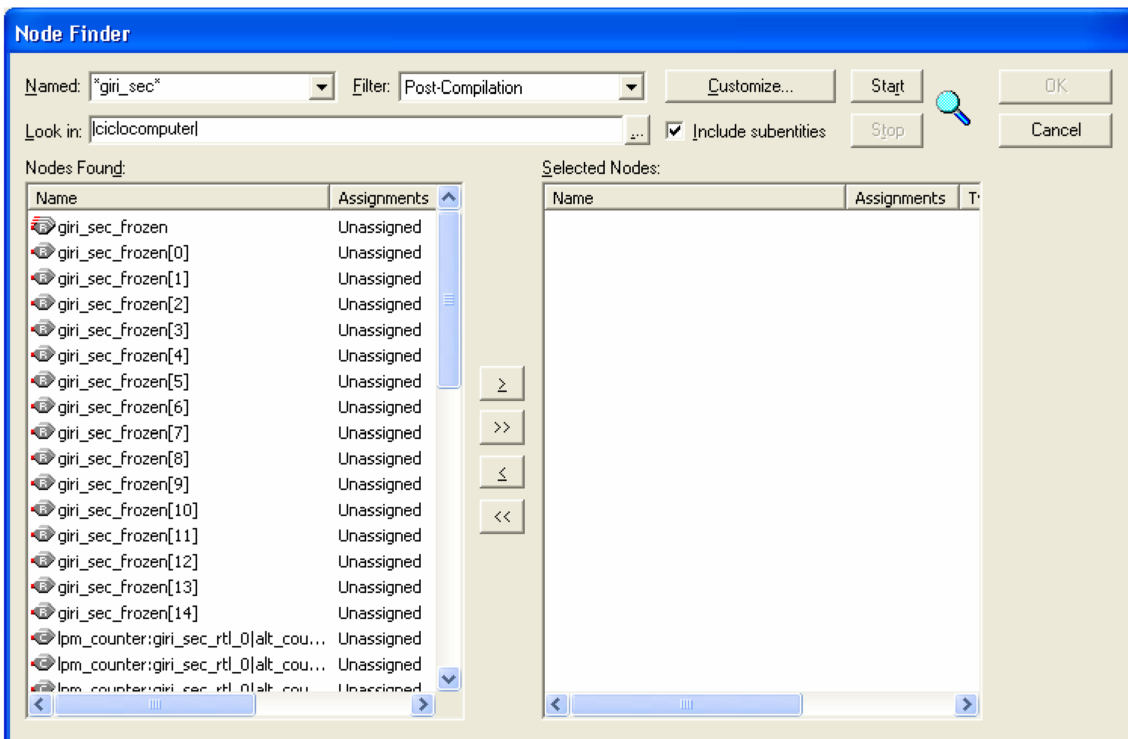
Andiamo a definire per GIROCOMPIUTO una forma d'onda adeguata: se lo definiamo come un clock con periodo 500us, e duty cycle 20% otteniamo una forma d'onda attiva per un ciclo di clock (100us) ogni 5.
Utilizzare il pulsante "A" a sinistra per inserire commenti e rendere piu leggibili le forme d'onda, come mostrato in figura.



Tuttavia al fine di effettuare un debug più accurato in caso di malfunzionamento, è utile e spesso necessario visualizzare anche i segnali interni all'*architecture*. I segnali che sono uscite di registri vengono di solito mantenuti da Quartus, con nomi uguali o simili. Utilizzeremo per individuarli il tool *Node Finder*, selezionando *Registers:PostFitting* nella casella *Filter*.



Inoltre è possibile effettuare ricerche su nomi specifici di segnali utilizzando il carattere jolly *, come mostrato in figura. Selezioniamo *Post-Compilation* e cerchiamo tutti i segnali che contengono i caratteri "giri_sec", come indicato nella seguente figura:



Si può osservare che Quartus non ha preservato i nomi di tutti i segnali in uscita dai registri poichè alcuni sono stati inclusi in contatori ottimizzati. E' il caso di `cont_distanza` e `giri_sec`. Li si individui cercando "`*cont_distanza*`", oppure "`*giri_sec*`" oppure "`*q`".
L'ultima stringa individua i registri ottimizzati da Quartus, poiché hanno l'uscita denominata "`q`" a default.
Individuatili li si aggiunga alla finestra di simulazione

Tuttavia, pur mantenendo i segnali sequenziali, Quartus per via del mappaggio tecnologico su LUT e FF perde il dettaglio sui segnali combinatori, di cui viene effettuata una sintesi ottimizzata. Questi segnali potrebbero non essere presenti nel *Node Finder*, (chiaramente alla voce Post-Compilation!e non *Registers*!)
Per esempio, vorremmo visualizzare il segnale `wave1Hz`, per visualizzare la forma d'onda periodica di 10 cicli, ma questo segnale non è presente nella lista.

E' possibile allora ricorrere ad un piccolo artificio, per la fase di debug, in cui i segnali combinatori che desideriamo visualizzare e non sono presenti vengono mappati su pin di output aggiuntivi, con qualche modifica al codice VHDL. Per esempio:

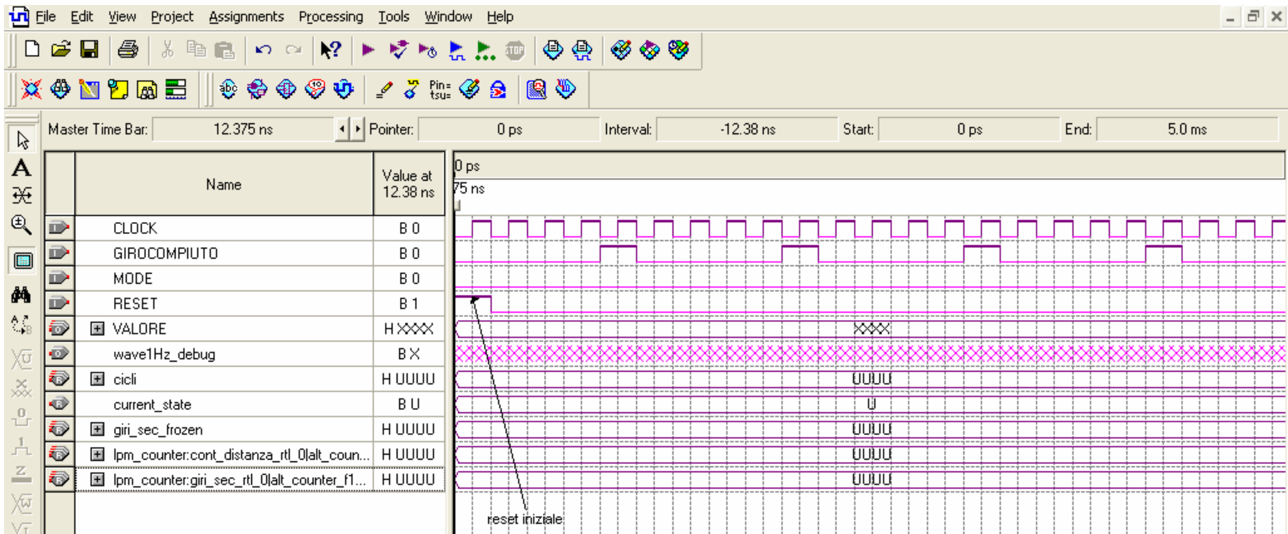
```
entity CICLOCOMPUTER is
   port (
      GIROCOMPIUTO  : in  std_logic;
      CLOCK         : in  std_logic;
      RESET         : in  std_logic;
      MODE          : in  std_logic;
      VALORE        : out unsigned(15 downto 0)

      wave1Hz_debug : out std_logic;
   );
end CICLOCOMPUTER;

architecture A of CICLOCOMPUTER is
[…]
    wave1Hz_debug <= wave1Hz;
[…]
```

Deve essere chiaro che queste modifiche servono per ovviare ad una limitazione del tool, nella sua versione 2.2. Queste modifiche impediscono al tool di effettuare ottimizzazioni spinte, e quindi non verranno ottenute prestazioni ottime. Infatti i segnali devono essere preservati (e quindi questo significa sintesi combinatoria non ottima) e portati su dei pin (e questo implica un maggiore sforzo di routing, e maggiore capacità di carico).

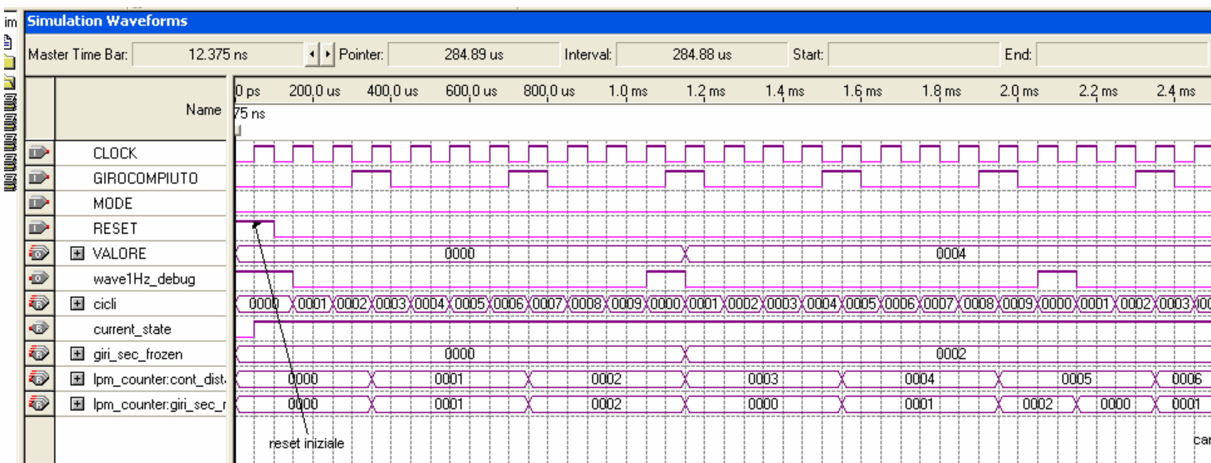
Ora è possibile visualizzare il segnale di debug, insieme a tutti i registri interni appena individuati.

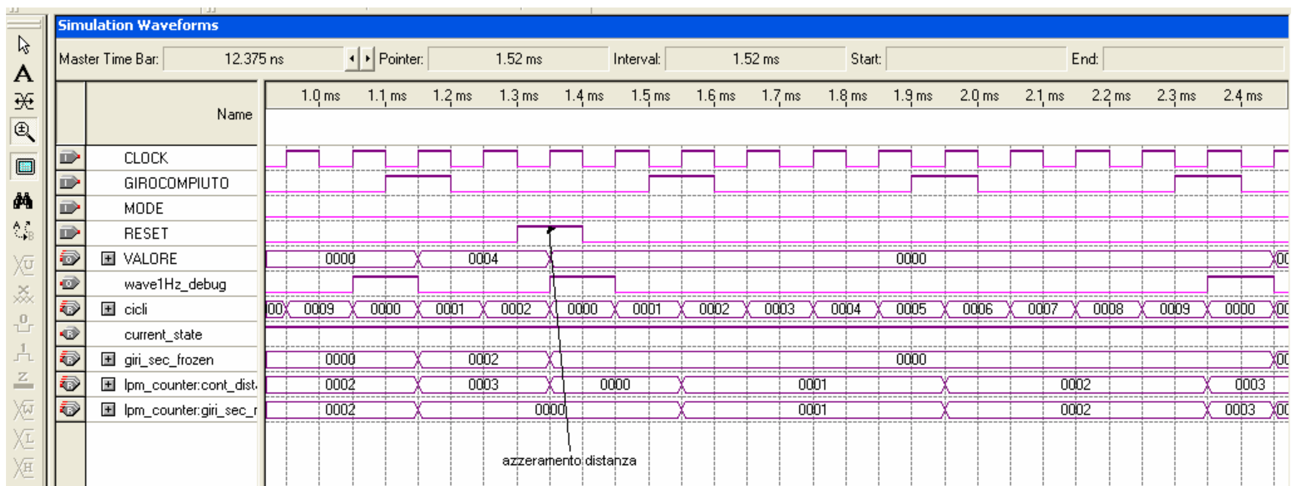


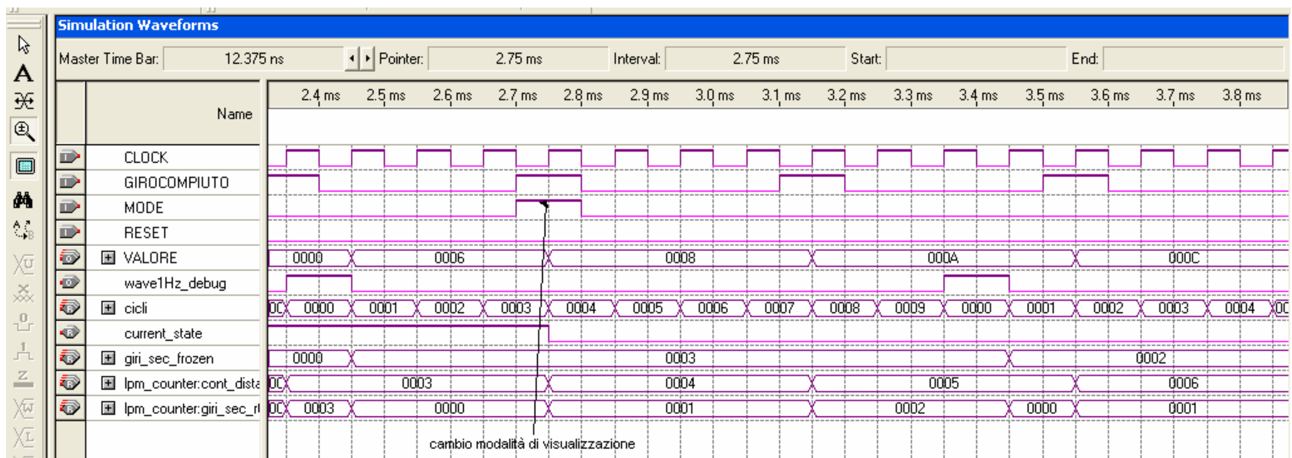
Salvare le forme d'onda, verrà proposto un nome di file identico a quello della top-level entità, con estensione VWF (Vector Waveform File).

Secondo la documentazione Altera, per ottenere che venga mantenuta corrispondenza con i nomi di un maggior numero di segnali:
Assignments, Settings, Compiler Settings, Mode
e deselezionare la casella "*Preserve fewer node names*".

Ora simuliamo.
Processing, Run Simulation. Il risultato è il seguente.







Osservando le forme d'onda le specifiche sembrano rispettate. La pressione del tasto MODE causa la commutazione tra velocità e distanza dell'uscita, mentre il tasto RESET azzera la distanza. La forma d'onda periodica azzera periodicamente il registro `giri_sec` che incrementa ad ogni giro compiuto dalla ruota, mentre `giri_sec_frozen` mantiene l'ultima velocità calcolata.

Le forme d'onda in uscita si attivano visualizzando la finestra *Window*, *Simulation Report*.

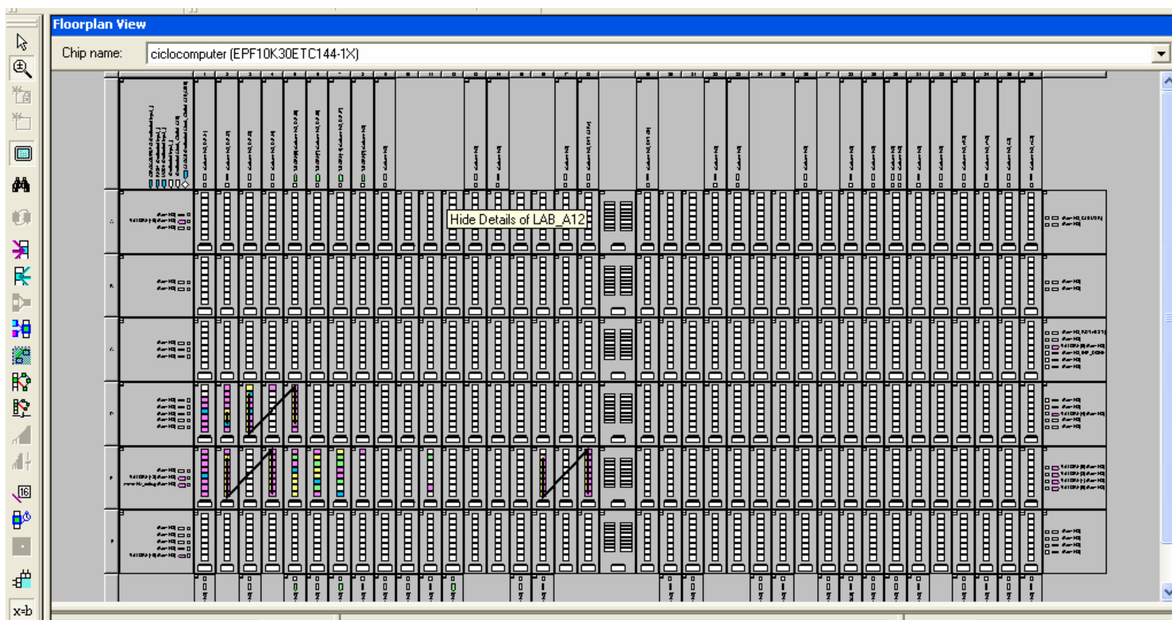
3.Sintesi Logica

Abbiamo il vincolo (non stringente) che la frequenza di lavoro sia compatibile con un oscillatore a 10kHz. Questo vincolo può essere indicato al tool di sintesi nel menu:
Assignments, Timing Settings, Clocks
Impostiamo, garantendoci un certo margine, un periodo di clock di 50us, corrispondenti al doppio della frequenza nominale.
Compiliamo il design, e andiamo ad analizzare i risultati di sintesi.

Apriamo il *Compilation Report*, dal menu *Window*.
Andiamo in *Synthesis Section, Resource Utilization by Entity*.
Qui troviamo un riassunto dell'utilizzo di LE nel dispositivo, in particolare quante sono puramente combinatorie (LUT), quante sequenziali (Registers) e quante miste.

<table>
<tr><th colspan="10">Resource Utilization by Entity</th></tr>
<tr><th></th><th>Compilation Hierarchy Node</th><th>Logic Cells</th><th>Registers</th><th>Memory Bits</th><th>Pins</th><th>LUT-Only LCs</th><th>Register-Only LCs</th><th>LUT/Register LCs</th><th>Full Hierarchy Name</th></tr>
<tr><td>1</td><td>|ciclocomputer</td><td>96 (53)</td><td>60</td><td>0</td><td>21</td><td>36 (23)</td><td>0 (0)</td><td>60 (30)</td><td>|ciclocomputer</td></tr>
<tr><td>2</td><td>|lpm_add_sub:add_234|</td><td>13 (0)</td><td>0</td><td>0</td><td>0</td><td>13 (0)</td><td>0 (0)</td><td>0 (0)</td><td>|ciclocomputer|lpm_add_sub:add_234</td></tr>
<tr><td>3</td><td>|addcore:adder|</td><td>13 (1)</td><td>0</td><td>0</td><td>0</td><td>13 (1)</td><td>0 (0)</td><td>0 (0)</td><td>|ciclocomputer|lpm_add_sub:add_234|addcore:adder</td></tr>
<tr><td>4</td><td>|a_csnbuffer:result_node|</td><td>12 (12)</td><td>0</td><td>0</td><td>0</td><td>12 (12)</td><td>0 (0)</td><td>0 (0)</td><td>|ciclocomputer|lpm_add_sub:add_234|addcore:adder</td></tr>
<tr><td>5</td><td>|lpm_counter:cont_distanza_rtl_0|</td><td>15 (0)</td><td>15</td><td>0</td><td>0</td><td>0 (0)</td><td>0 (0)</td><td>15 (0)</td><td>|ciclocomputer|lpm_counter:cont_distanza_rtl_0</td></tr>
<tr><td>6</td><td>|alt_counter_f10ke:wysi_counter|</td><td>15 (15)</td><td>15</td><td>0</td><td>0</td><td>0 (0)</td><td>0 (0)</td><td>15 (15)</td><td>|ciclocomputer|lpm_counter:cont_distanza_rtl_0|alt_c</td></tr>
<tr><td>7</td><td>|lpm_counter:giri_sec_rtl_0|</td><td>15 (0)</td><td>15</td><td>0</td><td>0</td><td>0 (0)</td><td>0 (0)</td><td>15 (0)</td><td>|ciclocomputer|lpm_counter:giri_sec_rtl_0</td></tr>
<tr><td>8</td><td>|alt_counter_f10ke:wysi_counter|</td><td>15 (15)</td><td>15</td><td>0</td><td>0</td><td>0 (0)</td><td>0 (0)</td><td>15 (15)</td><td>|ciclocomputer|lpm_counter:giri_sec_rtl_0|alt_counter</td></tr>
</table>

Alla voce Floorplan View è possibile vedere una rappresentazione schematica del dispositivo e del suo utilizzo:



Andando invece su *Timing Analyses, Fmax* è possibile verificare la frequenza massima raggiunta dal dispositivo dopo la sintesi logica. Vengono indicati tutti i "Critical Path" cioè i percorsi combinatori più lunghi. Il più lungo tra questi limita la frequenza massima dell'intero sistema.
Premendo sui tasti "+", è possibile individuarli con la seguente notazione:

NOME DEL SEGNALE DI CLOCK
        -NOME REGISTRO DESTINAZIONE
                -NOME REGISTRO SORGENTE.

(Essendo i critical paths combinatori, sono necessariamente compresi tra 2 registri!)

	fmax (not incl. delays to/from pins)			
	Clock Name -- Destination Register Name -- Source Register Name	Required fmax	Actual fmax (period)	
1	⊟ CLOCK	0.02 MHz	153.85 MHz (period = 6.500 ns)	
2	⊟ giri_sec_frozen[12]	0.02 MHz	153.85 MHz (period = 6.500 ns)	
3	cicli[4]	0.02 MHz	153.85 MHz (period = 6.500 ns)	
4	cicli[1]	0.02 MHz	161.29 MHz (period = 6.200 ns)	
5	cicli[2]	0.02 MHz	163.93 MHz (period = 6.100 ns)	
6	cicli[6]	0.02 MHz	166.67 MHz (period = 6.000 ns)	
7	cicli[7]	0.02 MHz	169.49 MHz (period = 5.900 ns)	
8	cicli[8]	0.02 MHz	169.49 MHz (period = 5.900 ns)	
9	cicli[5]	0.02 MHz	169.49 MHz (period = 5.900 ns)	
10	cicli[10]	0.02 MHz	185.19 MHz (period = 5.400 ns)	
11	cicli[12]	0.02 MHz	188.68 MHz (period = 5.300 ns)	
12	cicli[9]	0.02 MHz	188.68 MHz (period = 5.300 ns)	
13	Timing analysis results restricted.		To change the limit use Timing Settings (Assignments menu)	
14	⊞ giri_sec_frozen[11]	0.02 MHz	153.85 MHz (period = 6.500 ns)	
26	⊞ giri_sec_frozen[10]	0.02 MHz	153.85 MHz (period = 6.500 ns)	
38	⊞ giri_sec_frozen[9]	0.02 MHz	153.85 MHz (period = 6.500 ns)	
50	⊞ giri_sec_frozen[6]	0.02 MHz	153.85 MHz (period = 6.500 ns)	
62	⊞ giri_sec_frozen[7]	0.02 MHz	156.25 MHz (period = 6.400 ns)	
74	⊞ giri_sec_frozen[5]	0.02 MHz	156.25 MHz (period = 6.400 ns)	

In figura il percorso critico va dall’uscita 4 del registro cicli, ed arriva fino all’uscita 12 del registro giri_sec_frozen. Questo calcolo tiene conto sia del ritardo attraverso le porte logiche, sia del ritardo introdotto dagli switch e dalle linee di interconnessioni. Dipende quindi sia dalla sintesi logica che dal Place&Route.

Oltre alla frequenza massima raggiungibile internamente, un altro dato importante è dato dal ritardo di commutazione dei pin di uscita rispetto al clock. Viene assunta una situazione di carico standard di riferimento ed elencati, alla voce”tco” (Clock to Output Delay) i ritardi dal più grande in ordine decrescente.

	tco (Clock to Output Delays)		
	Output Name -- Register Name -- Clock Name	Actual tco	
1	⊞ wave1Hz_debug	10.700 ns	
23	⊞ VALORE[4]	9.400 ns	
30	⊞ VALORE[6]	9.200 ns	
37	⊞ VALORE[8]	9.100 ns	
44	⊞ VALORE[1]	9.100 ns	
51	⊞ VALORE[9]	9.000 ns	
58	⊞ VALORE[10]	8.800 ns	
65	⊞ VALORE[15]	8.500 ns	
72	⊞ VALORE[13]	8.200 ns	
79	⊞ VALORE[14]	8.000 ns	
86	⊞ VALORE[7]	8.000 ns	
93	⊞ VALORE[2]	8.000 ns	
100	⊞ VALORE[5]	7.900 ns	
107	⊞ VALORE[12]	7.800 ns	
114	⊞ VALORE[11]	7.800 ns	
121	⊞ VALORE[3]	7.800 ns	

Da questo valore si evince la frequenza di commutazione dei pin di uscita, si noti come in questo caso sia minore di quella di clock. Secondo voi come mai?

4. Simulazione Post-Sintesi

Consente di verificare se il VHDL è stato sintetizzato correttamente e dà una visione più realistica del funzionamento del circuito. *Assignments, Settings, Simulator, Mode, Timing* e ripetere la simulazione con *Processing, Run Simulation*. Compariranno glitch, e ritardi rappresentativi delle commutazioni nei vari nodi del dispositivo.

5. Generazione del bitstream

E’ l’ultimo passo, in cui viene generato un file POF (Programmer Object File) che contiene la configurazione del dispositivo FPGA utilizzato per implementare il codice sviluppato con i vincoli specificati.