PROGETTAZIONE DI SISTEMI EMBEDDED

UNIVERSITA' DEGLI STUDI DI VERONA
DIPARTTIENTO DI INFORMATICA

A.A. 2014/2015

Nicolò Danzi
VR387906

Indice

Modelling al livello RTL

Lezione 2

Il progetto consiste nella realizzazione a livello RTL un sistema per la moltiplicazione floating point. Il modulo verrà testato attraverso il testbench che simulerà l'esecuzione del modulo forzando i segnali con i valori richiesti.

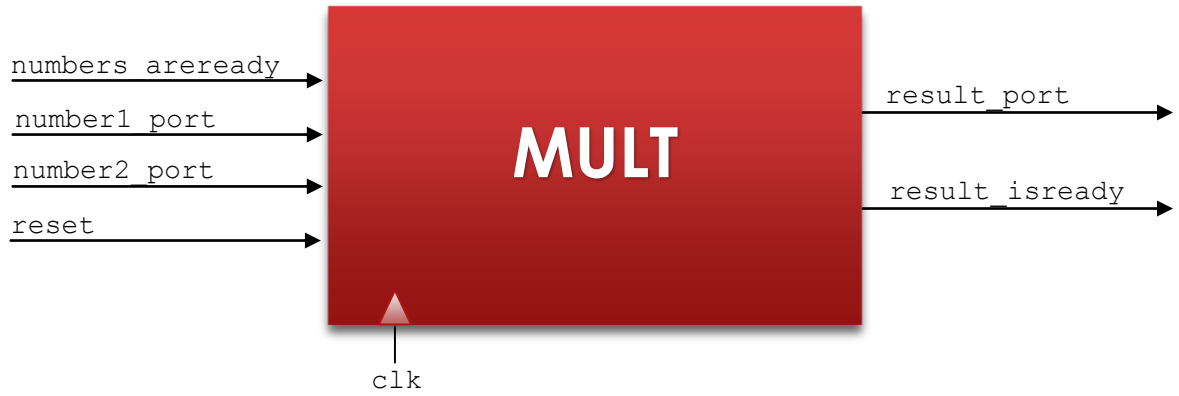
Il sistema è suddiviso in cinque file:

- **main_mult_RTL.cc:** rappresenta l'interfaccia di collegamento tra il mult_RTL e il testbench;
- **mult_RTL.hh:** rappresenta il file di dichiarazione dell'EFSM che descrive come è costituito il modulo RTL;
- **mult_RTL.cc**: rappresenta il file elaborativo dell'EFSM, ovvero il vero e proprio modulo mult_RTL;
- **mult_RTL_testbench.hh:** rappresenta la libreria del testbench, descrive come è costituito il modulo di simulazione.
- **mult_RTL_testbench.cc:** rappresenta il modulo di simulazione, contiene al suo interno l'implementazione del testing di sistema con le modalità richieste.

Interfaccia del modulo

L'interfaccia del modulo mult è costituita da 7 porte:

- **`numbers_areready`:** è un segnale di 1 bit messo a 1 dal testbench quando due nuovi numeri sono disponibili nelle `number1_port` e `number2_port`. Il modulo mult deve attendere finché `numbers_areready` non è messo a 1 per iniziare il calcolo della moltiplicazione.
- **`number1_port`:** è un segnale a 64 bit che è settato dal tetstbench. Rappresenta il primo numero dato in input su cui il modulo esegue la moltiplicazione.
- **`number2_port`:** è un segnale a 64 bit che è settato dal tetstbench. Rappresenta il secondo numero dato in input su cui il modulo esegue la moltiplicazione.
- **`result_port`:** è un segnale a 64 bit che è settato dal modulo mult. Rappresenta il risultato dato in output dopo che il modulo ha eseguito la moltiplicazione.
- **`result_isready`:** è un segnale di 1 bit messo a 1 dal modulo mult quando ha calcolato la moltiplicazione e il risultato è disponibile nelle `result_port`.
- **`reset`:** è un segnale di tipo bool usato per resettare il modulo mult.
- **`Clk`:** è il clock.



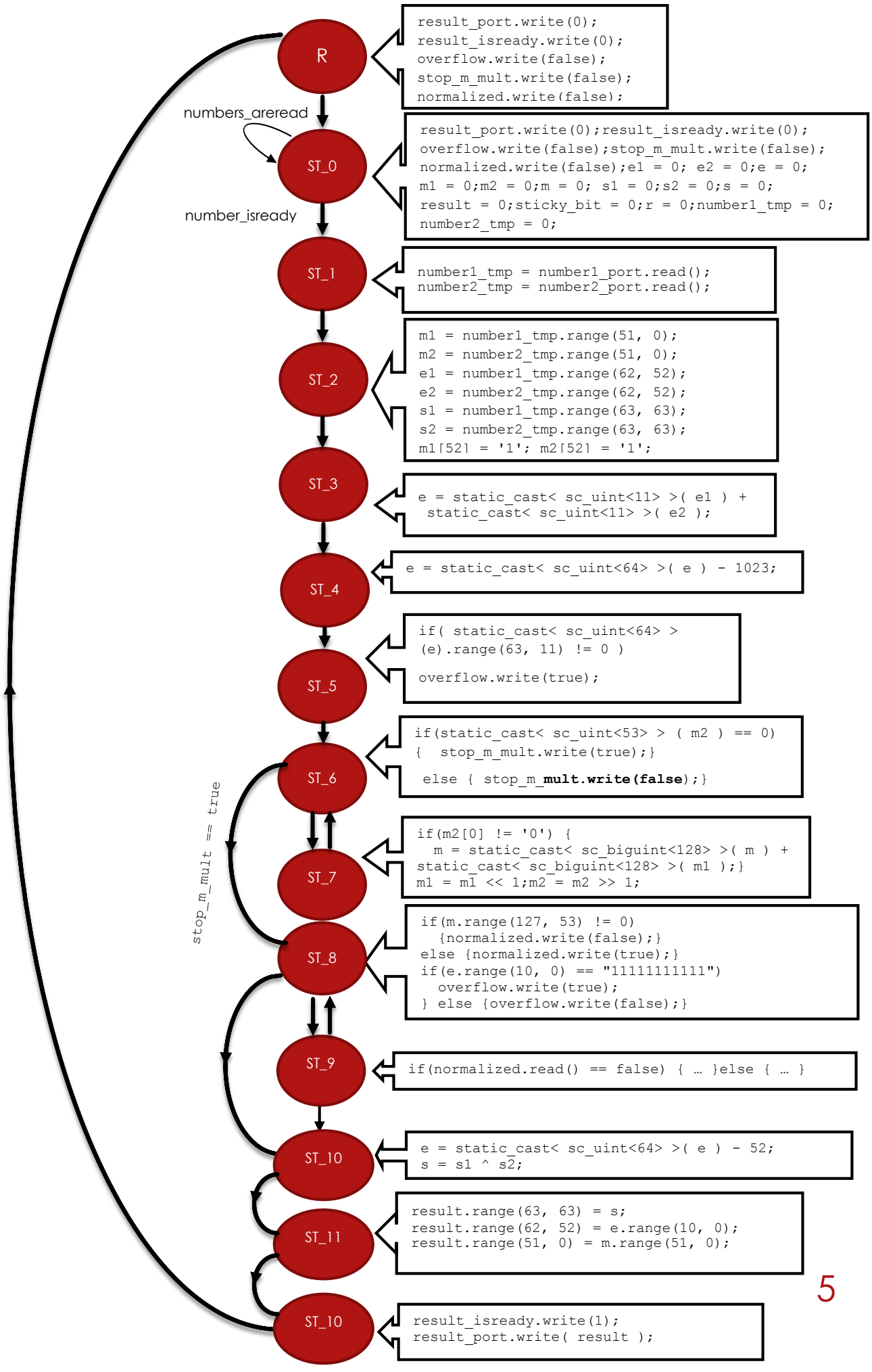
Funzionalità

Le funzionalità che devo implementare sono quelle del moltiplicatore seguendo l'algoritmo visto a lezione. Segue lo pseudocidice dell'algoritmo che ho utilizzato:

```
BEGIN
#inizializzazione:
...
-----------
#recupero di mantissa esponente e segno:
e = e_1 + e_2 - 1023
m_1 = 1 + m_1
m_2 = 1 + m_2
m = 0
-----------
#controllo overflow di e:
...
-----------
#moltiplicazione delle mantisse:
while(m_2 != 0)
----------
        if(m_2[0] != 0)
                m = m + m_1
        m_1 = m_1 << 1;
        m_2 = m_2 >> 1;
----------
#normalizzazione:
while(m.range(63, 53) != 0 && m[52] != 1)
        r = m[0];
        sticky_bit = sticky_bit OR r;
----------
        m = m >> 1;
        if(e != max_e)
                e = e + 1;
        else
            overflow;
---------
#fine normalizzazione:
      if(r == "1" && sticky_bit == "1")
            m = m + 1;
      else if(sticky_bit == "0" && r == "1" && m[0] == 1)
            m = m + 1;
---------
#calcolo del segno e esponente:
      e = e - 52;
      s = s1 XOR s2;
---------
#assegnamento del risultato:
...
--------
END
```

EFSM

Un modulo può essere rappresentato da un *clocked EFSM*. Io ho pensato di strutturarlo nel seguente modo:


R
result_port.write(0);
result_isready.write(0);
overflow.write(false);
stop_m_mult.write(false);
normalized.write(false);
numbers_areread
ST_0
result_port.write(0);result_isready.write(0);
overflow.write(false);stop_m_mult.write(false);
normalized.write(false);e1 = 0; e2 = 0;e = 0;
m1 = 0;m2 = 0;m = 0; s1 = 0;s2 = 0;s = 0;
result = 0;sticky_bit = 0;r = 0;number1_tmp = 0;
number2_tmp = 0;
number_isready
ST_1
number1_tmp = number1_port.read();
number2_tmp = number2_port.read();
ST_2
m1 = number1_tmp.range(51, 0);
m2 = number2_tmp.range(51, 0);
e1 = number1_tmp.range(62, 52);
e2 = number2_tmp.range(62, 52);
s1 = number1_tmp.range(63, 63);
s2 = number2_tmp.range(63, 63);
m1[52] = '1'; m2[52] = '1';
ST_3
e = static_cast< sc_uint<11> >(e1) +
static_cast< sc_uint<11> >(e2);
ST_4
e = static_cast< sc_uint<64> >(e) - 1023;
ST_5
if(static_cast< sc_uint<64> >
(e).range(63, 11) != 0)
overflow.write(true);
ST_6
if(static_cast< sc_uint<53> > (m2) == 0)
{ stop_m_mult.write(true);}
else { stop_m_mult.write(false);}
stop_m_mult == true
ST_7
if(m2[0] != '0') {
m = static_cast< sc_biguint<128> >(m) +
static_cast< sc_biguint<128> >(m1);}
m1 = m1 << 1;m2 = m2 >> 1;
ST_8
if(m.range(127, 53) != 0)
{normalized.write(false);}
else {normalized.write(true);}
if(e.range(10, 0) == "11111111111")
overflow.write(true);
} else {overflow.write(false);}
ST_9
if(normalized.read() == false) { … }else { … }
ST_10
e = static_cast< sc_uint<64> >(e) - 52;
s = s1 ^ s2;
ST_11
result.range(63, 63) = s;
result.range(62, 52) = e.range(10, 0);
result.range(51, 0) = m.range(51, 0);
ST_10
result_isready.write(1);
result_port.write(result);

Implementazione

Il mio modulo è codificato in due SC_METHODS, uno per calcolare i valori interni delle variabili e uno per calcolare lo stato prossimo.

```
SC_METHOD(elaborate_MULT_FSM);
    sensitive << reset.neg();
    sensitive << clk.pos();

SC_METHOD(elaborate_MULT);
    sensitive << STATUS << numbers_areready << number1_port << number2_port;
```

Il primo SC_METHOD è sensibile alle variazioni di reset e clock. Il secondo invece è sensibile allo stato, i segnali number1_port, number2_port, e number_areready.

Elaborate_MULT_FSM

void mult_RTL :: elaborate_MULT_FSM(void){

```
  cout<<"\t"<<sc_time_stamp()<<" - root: MULT_FSM"<<endl;
  static sc_lv<11>   e1;
  static sc_lv<11>   e2;
  static sc_lv<64>   e;
  static sc_lv<128>   m1;
  static sc_lv<128>   m2;
  static sc_lv<128>  m;
  static sc_lv<1>    s1;
  static sc_lv<1>    s2;
  static sc_lv<1>    s;
  static sc_lv<1>   sticky_bit;
  static sc_lv<1>   r;
  static sc_lv<64>   result;
  static sc_lv<64>   number1_tmp;
  static sc_lv<64>   number2_tmp;

  if (reset.read() == 0){
    cout<< "\t" <<sc_time_stamp()<< " - mult: reset" << endl;
    STATUS = Reset_ST;
  }
  else if (clk.read() == 1) {


    STATUS = NEXT_STATUS;

    switch(STATUS){
      //reset state
      case Reset_ST:
        cout<< "\t" <<sc_time_stamp()<< "mult: ST_reset" << endl;

        result_port.write(0);
        result_isready.write(0);
        overflow.write(false);
        stop_m_mult.write(false);
        normalized.write(false);

        break;
```

```
//initialization state
case ST_0:
  cout<< "\t" <<sc_time_stamp()<< "mult: ST_0" << endl;

  result_port.write(0);
  result_isready.write(0);
  overflow.write(false);
  stop_m_mult.write(false);
  normalized.write(false);


  e1 = 0; //11 bit
  e2 = 0; //11 bit
  e = 0; //64 bit

  m1 = 0; //53 bit
  m2 = 0; //53 bit
  m = 0; //128 bit

  s1 = 0; //1 bit
  s2 = 0; //1 bit
  s = 0; // 1 bit

  result = 0; // 64 bit
  sticky_bit = 0;
  r = 0;
  number1_tmp = 0;
  number2_tmp = 0;
  break;

//take the numbers
case ST_1:
  cout<< "\t" <<sc_time_stamp()<< "mult: ST_1" << endl;

  number1_tmp = number1_port.read();
  number2_tmp = number2_port.read();

  break;

//divide numbers' parts into s e m
case ST_2:
  cout<< "\t" <<sc_time_stamp()<< "mult: ST_2" << endl;

  m1 = number1_tmp.range(51, 0);
  m2 = number2_tmp.range(51, 0);

  e1 = number1_tmp.range(62, 52);
  e2 = number2_tmp.range(62, 52);

  s1 = number1_tmp.range(63, 63);
  s2 = number2_tmp.range(63, 63);

  m1[52] = '1';
  m2[52] = '1';
  break;

//sum the exponents
case ST_3:
```

```
        cout<< "\t" <<sc_time_stamp()<< "mult: ST_3" << endl;


        e = static_cast< sc_uint<11> >( e1 ) + static_cast< sc_uint<11> >(
e2 );
        break;

      //subtract the bias
      case ST_4:
        cout<< "\t" <<sc_time_stamp()<< "mult: ST_4" << endl;

        e = static_cast< sc_uint<64> >( e ) - 1023;
        break;

      //check exp overflow
      case ST_5:
        cout<< "\t" <<sc_time_stamp()<< "mult: ST_5" << endl;

        if( static_cast< sc_uint<64> > (e).range(63, 11) != 0 )
          overflow.write(true);
        break;

      //multiplication of significand .1: check if finished
      case ST_6:
        cout<< "\t" <<sc_time_stamp()<< "mult: ST_6" << endl;

        if(static_cast< sc_uint<53> > ( m2 ) == 0) {
          stop_m_mult.write(true);
        } else {
          stop_m_mult.write(false);
        }
        break;


      //multiplication of significand .2: multiplication
      case ST_7:
        cout<< "\t" <<sc_time_stamp()<< "mult: ST_7" << endl;

        if(m2[0] != '0') {
          //cout << "sum m1 to m" << endl;
          m = static_cast< sc_biguint<128> >( m ) + static_cast<
sc_biguint<128> >( m1 );
        }

        m1 = m1 << 1;
        m2 = m2 >> 1;

        break;

      //normalization .1
      case ST_8:
        cout<< "\t" <<sc_time_stamp()<< "mult: ST_8" << endl;

        if(m.range(127, 53) != 0) {
          normalized.write(false);
        } else {
          normalized.write(true);
        }
```

```
    if(e.range(10, 0) == "11111111111") {
      overflow.write(true);
    } else {
      overflow.write(false);
    }

    break;

  //normalization .2
  case ST_9:
    cout<< "\t" <<sc_time_stamp()<< "mult: ST_9" << endl;
    if(normalized.read() == false) {

      r[0] = m[0];
      sticky_bit = sticky_bit | r;

      m = m >> 1;

      e = static_cast< sc_uint<64> >( e ) + 1;


    } else {

      if(r == "1" && sticky_bit == "1")
        m = static_cast< sc_biguint<128> >( m ) + 1;
      else if(sticky_bit == "0" && r == "1" && m[0] == 1)
        m = static_cast< sc_biguint<128> >( m ) + 1;


    }
    break;


  //calculate sign
  case ST_10:
    cout<< "\t" <<sc_time_stamp()<< "mult: ST_10" << endl;
    e = static_cast< sc_uint<64> >( e ) - 52;
    s = s1 ^ s2;
    break;

  //compose result
  case ST_11:
    cout<< "\t" <<sc_time_stamp()<< "mult: ST_11" << endl;

    result.range(63, 63) = s;
    result.range(62, 52) = e.range(10, 0);
    result.range(51, 0) = m.range(51, 0);
    break;

  case ST_12:
    cout<< "\t" <<sc_time_stamp()<< "mult: ST_12" << endl;

    result_isready.write(1);
    result_port.write( result );
    break;
```

```
    }
  }
}
```

Elaborate_MULT

```
void mult_RTL :: elaborate_MULT(void){
cout<<"\t"<<sc_time_stamp()<<" - mult: MULT"<<endl;


NEXT_STATUS = STATUS;
switch(STATUS){
   case Reset_ST:
     NEXT_STATUS = ST_0;
     break;
   case ST_0:
     if (numbers_areready.read() == 1){
       NEXT_STATUS = ST_1;
     } else {
       NEXT_STATUS = ST_0;
     }
     break;

    case ST_1:
     NEXT_STATUS = ST_2;
      break;
    case ST_2:
     NEXT_STATUS = ST_3;
      break;
    case ST_3:
      NEXT_STATUS = ST_4;
      break;
    case ST_4:
      NEXT_STATUS = ST_5;
     break;
   case ST_5:
      NEXT_STATUS = ST_6;
      break;
   case ST_6:
     if(stop_m_mult.read() == true) {
       NEXT_STATUS = ST_8;
     } else {
        NEXT_STATUS = ST_7;
     }
      break;
   case ST_7:
      NEXT_STATUS = ST_6;
      break;
    case ST_8:
      if(normalized.read() == true) {
       NEXT_STATUS = ST_10;
```

```
    } else {
      NEXT_STATUS = ST_9;
    }
    break;
  case ST_9:
    if(normalized.read() == true) {
      NEXT_STATUS = ST_10;
    } else {
      NEXT_STATUS = ST_8;
    }
    break;
  case ST_10:
     NEXT_STATUS = ST_11;
     break;
   case ST_11:
    NEXT_STATUS = ST_12;
     break;
   case ST_12:
    NEXT_STATUS = ST_0;
    break;

}
```

SystemC timing evolution

Lezione 3

L'obiettivo di questa lezione è studiare la simulazione del kernel di SystemC e l'algoritmo di scheduling che regola la simulazione.

RTL

La simulazione di SystemC

Durante l'esecuzione del codice opportunamente commentato noi possiamo distinguere alcune fasi:

- Inizializzazione dello scheduling: tutti SC_METHODS eseguiti una volta.
- il costrutto wait: sospende il testbench da 0s a 40 ns.
- la sequenza di esecuzione del MULT:
  - Il processo MULT è svegliato dal processo MULT_FSM, che scrive nei segnali interni del modulo moltiplicatore.
  - Il processo MULT_FSM è svegliato dal clock ad ogni ciclo di clock

Qui si può vedere l'output dell'esecuzione con le varie fasi evidenziate:

```
0 s - mult: MULT_FSM

0 s - mult: reset
```

Inizializzazion

0 s - mult: MULT

0 s - tb: begin run()

Reset the design!

0 s - tb: reset

0 s - mult: MULT

0 s - mult: MULT_FSM

0 s - mult: reset

10 ns - mult: MULT_FSM

10 ns - mult: reset

20 ns - mult: MULT_FSM

20 ns - mult: reset

30 ns - mult: MULT_FSM

30 ns - mult: reset

40 ns - mult: MULT_FSM

40 ns - mult: reset

40 ns - tb: after wait(5)

Wait(5)

The multiplication between -47.178 and 106.04

40 ns - tb: write 1100000001000111100101101100100010110100001110010101100000010000

40 ns - tb: write 0100000001011010100000101000111101011100001010001111010111000011

40 ns - mult: MULT

50 ns - mult: MULT_FSM

50 ns - mult: ST_Reset

50 ns - tb: wait result

50 ns - mult: MULT

60 ns - mult: MULT_FSM

MULT_FSM → MULT

Clock → MULT_FSM

60 ns - mult: ST_0

60 ns - mult: MULT

70 ns - mult: MULT_FSM

70 ns - mult: ST_1

…

2290 ns - mult: ST_10

2290 ns - mult: MULT

2300 ns - mult: MULT_FSM

2300 ns - mult: ST_11

2300 ns - mult: MULT

2310 ns - mult: MULT_FSM

2310 ns - mult: ST_12

2310 ns - mult: MULT

2320 ns - mult: MULT_FSM

2320 ns - mult: ST_0

is: -5002.7551199999998061684891581535339355468750000000000

2320 ns - tb: result available

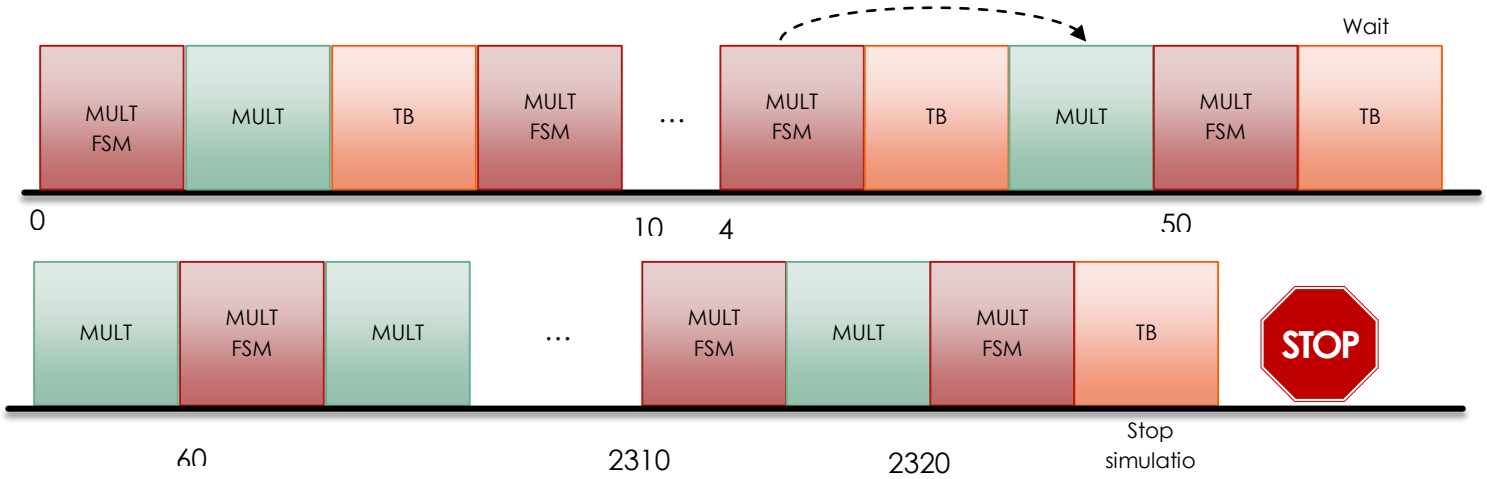
multiplication was right

****

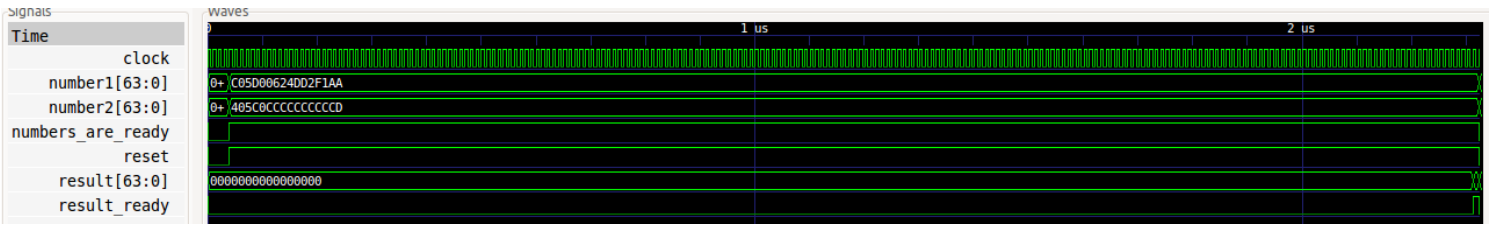
All multiplications were right!!

2320 ns - tb: reset

L'esecuzione può essere schematizzata nel modo seguente:



File VDC



Come si può vedere da questa immagine tutti i segnali sono bassi fino al tempo 40ns. Al tempo 40ns, il testbench mette `reset` e `number_areready` a 1 e setta `number1` e `number2` ai valori di input per il design. Al tempo 2320 ns, il moltiplicatore mette `result_areready` a 1 e setta il valore di output `result`.

TLM UT

Con il codice TLM UT opportunamente commentato con le `sc_time_stamp` ottengo il seguente risultato:

```
0 s - [TB:] Calculating the product between -81.937 and -23.927
        0 s - [TB:] Invoking the b_transport primitive - write
                0 s - [MULT:] Received invocation of the b_transport primitive - write
                0 s - [MULT:] Invoking the mult_function to calculate the product
                0 s - [MULT:] Calculating mult_function ...
                0 s - [MULT:] Returning result:
01000000100111101010001000000110110000011110001101100100101111110
        0 s - [TB:] TLM protocol correctly implemented
        0 s - [TB:] Result is: 1960.51
Info: /OSCI/SystemC: Simulation stopped by user.
```

Come si può notare la notazione di tempo non è necessaria, i tempi del simulatore sono infatti tutti a zero.

TLM LT

Con il codice TLM LT opportunamente commentato con le `sc_time_stamp` ottengo il seguente risultato:

```
0 - top.initiator - run
        number1:        115.126
        number2:        78.722
        0 s - [TB:] Calculating the product between 115.126 and 78.722
        0 s - [TB:] Invoking the b_transport primitive - write
                0 s - [MULT:] Received invocation of the b_transport primitive - write
                0 s - [MULT:] Invoking the mult_function to calculate the product
                0 s - [MULT:] Calculating mult_function ...
                0 s - [MULT:] Returning result:
01000000110000011011001101111001011101111111010100001110001101001
        0 s - [TB:] TLM protocol correctly implemented
        0 s - [TB:] Result is: 9062.95
Time: 0 s + 100 ns

Info: /OSCI/SystemC: Simulation stopped by user.
```

Come può notare anche qui il tempo del Simulatore è pari a zero. La notazione di tempo è data dal programmatore attraverso Timing annotation. Il simulatore continua a dare tempo zero per un quanto di tempo cioè fino a che non si sincronizza e prende il tempo del timimg annotion, ciò avviene per il temporal decoupling.

TLM AT4

Con il codice TLM AT4 opportunamente commentato con le `sc_time_stamp` ottengo il seguente risultato:

```
                number1:    125.523
                number2:    -126.874

        0 s - [TB:] Calculating the product between 125.523 and -126.874
        0 s - [TB:] Invoking the nb_transport_fw primitive of mult - write
                0 s - [MULT:] Received invocation of the nb_transport_fw primitive
                0 s - [MULT:] Activating the IOPROCESS
                0 s - [MULT:] End of the nb_transport_fw primitive
        0 s - [TB:] Waiting for nb_transport_bw to be invoked
                0 s - [MULT:] IOPROCESS has been activated
                100 ns - [MULT:] Invoking the mult_function to calculate the product
                100 ns - [MULT:] Calculating mult_function ...
                100 ns - [TB:] Invoking the nb_transport_bw primitive - write
        100 ns - [TB:] Invoking the nb_transport_fw primitive of mult - read
                100 ns - [MULT:] Received invocation of the nb_transport_fw primitive
                100 ns - [MULT:] Activating the IOPROCESS
                100 ns - [MULT:] End of the nb_transport_fw primitive
        100 ns - [TB:] Waiting for nb_transport_bw to be invoked
                100 ns - [MULT:] IOPROCESS has been activated
                200 ns - [MULT:] Returning result:
11000000110011110001101011001101011100111111101101111010010111111
                200 ns - [TB:] Invoking the nb_transport_bw primitive - write
        200 ns - [TB:] TLM protocol correctly implemented
        200 ns - [TB:] Result is: -15925.6
Info: /OSCI/SystemC: Simulation stopped by user.
```

Come si può notare qui il tempo del simulatore avanza grazie a wait() messe appositamente dal programmatore.

Comparazione dei tempi

Come abbiamo notato abbiamo un avanzamento del tempo dato dal simulatore solo su RTL. Per i TLM le nozioni di tempo sono più astratte e date da annotazioni del programmatore.

SystemC Modelling al livello TLM

Lezione 4

Il SystemC TLM standard definisce i protocolli di comunicazione per sviluppo di design al livello più astratto di RTL. Il livello TLM prevede l'utilizzo di due tipologie di modulo per il funzionamento di un sistema, queste due tipologie sono rispettivamente: Initiator e Target. Initiator e Target comunicano utilizzando specifiche primitive e sincronizzandosi tra loro attraverso socket di comunicazione.

Untimed (UT)

Nella tipologia di TLM level UT il modulo Initiator è implementato nel file "mult_UT_testbench.cc", mentre il modulo Target nel file "mult_UT.cc". Il modulo Target rappresenta nel nostro sistema, l'elaborazione vera e propria dell'algoritmo, il modulo Initiator richiama l'elaborazione per eseguire determinate simulazioni. L'Untimed utilizza la primitiva "b_transport()", differenziata in modalità Read e Write, per la comunicazione tra i due moduli. La b_transport è implementata nel modulo Target e viene richiamata dal modulo Initiator per effettuare una simulazione (dopo ovviamente aver preparato adeguatamente i segnali di ingresso al modulo Target) in modalità Write, successivamente il Target comincerà ad elaborare i segnali in ingresso per calcolarne il risultato dell'algoritmo di moltiplicazione. Siccome Target e Initiator sono concorrenti, l'Initiator richiama una seconda volta la primitiva b_transport in modalità Read per poter leggere il risultato del modulo Target. Questo darà la disponibilità alla lettura del payload, (pacchetto di dati che trasferisce la b_transport), solo successivamente al "TLM_OK_RESPONSE" che rappresenta la fine dell'elaborazione del Target.

Loosley Timed (LT)

Anche in questo caso l'elaborazione si divide in modulo Target e Initiator che comunicano tra loro attraverso i rispettivi socket. Come per l'UT l'Initiator richiama la primitiva b_transport, implementata nel modulo Target, per richiedere al Target l'esecuzione dell'elaborazione. Il nome stesso del coding style identifica come il tempo giochi un ruolo chiave nell'esecuzione del sistema, infatti, si differenzia dall'Untimed in quanto è necessario considerare una sincronizzazione a livello temporale tra i due moduli Target e Initiator. Anche qui l'Initiator richiama la b_transport in modalità Write per inviare il payload contenente gli input per l'elaborazione al Target. Il Target ricevuto il payload comincia la sua elaborazione e una volta terminata risponde all'Initiator con il messaggio "TLM_OK_RESPONSE". Siccome i moduli sono concorrenti l'Initiator richiama una seconda volta la b_transport in modalità Read per leggere i risultati dell'esecuzione che saranno disponibili solo dopo il "TLM_OK_RESPONSE". Il tempo viene gestito in modalità di sincronizzazione dei due moduli e visualizzato in fase di esecuzione, in modo da determinare chi, e in quale istante di tempo, sta facendo cosa.

APPROXIMATELY TIMED (AT4)

L'Approximately Timed si distingue dagli stili precedenti per l'utilizzo di due primitive:

- nb_transport_fw: implementata nel Target viene invocata dall'Initiator per richiedere l'esecuzione dell'algoritmo di moltiplicazione al Target o richiederne i risultati (sostituisce approssimativamente la b_transport nei coding styles precedenti);
- nb_transport_bw: implementata nell'Initiator, viene invocata dal Target quando il Target ha terminato il calcolo dell'algoritmo e per segnalare all'Initiator la possibilità di ricevere i risultati dell'algoritmo.

L'Approximately Timed distingue quattro fasi operative che identificano richiesta e risposta della transport forward e richiesta e risposta della transport backward. Il tempo anche qui gioca un ruolo fondamentale.

Comparazione dei tempi

	TLM UT	TLM LT	TLM AT4	RTL
Real Time	0.044	0.067	0.090	0.537
User Time	0.003	0.003	0.005	0.321
System Time	0.023	0.035	0.009	0.164

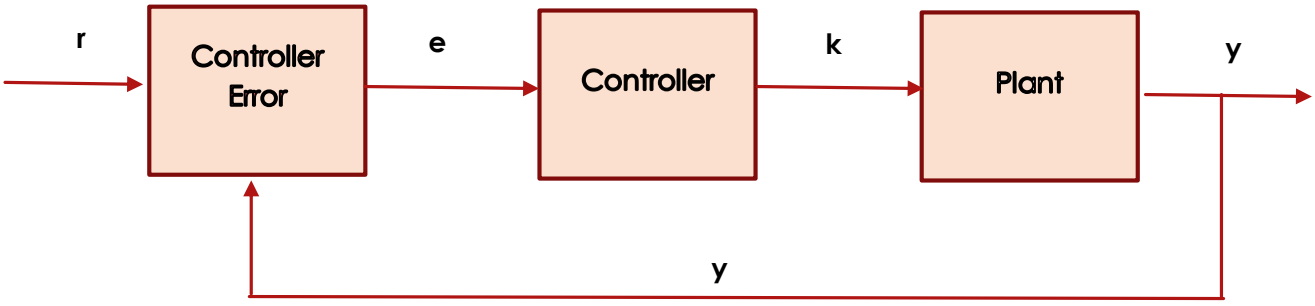
Nel TLM la simulazione è molto veloce e legata solo alla complessità della funzionalità del modulo, infatti abbiamo tempi molto bassi. Il TLM – UT è quello che ci impiega il tempo minore. Invece in RTL la simulazione è più lenta infatti con la stessa quantità di moltiplicazioni impiega 0.321s.

SystemC-AMS

Lezione 6 – SystemC-AMS

L'assegnamento prevede la realizzazione di un impianto/controllore in SystemC-AMS. Dati una serie di input il Sistema deve prevedere un controllore che calcoli l'errore e la funzione k, che viene passata all'impianto il quale calcola la funzione di trasferimento e crea quindi l'output.

Implementazione



Ho implementato il Sistema come un cluster TDF poichè con questo MoC riuscivo a implementare più facilmente i collegamenti tra I vari component del Sistema. Avrei potuto anche utilizzare il MoC LSF per rappresentare l'impianto. Inoltre in questo modo è più facile l'interazione con SystemC tramite porte specifiche (questo mi sarà utile nella creazione del platform). Il Sistema quindi è descritto da una serie di blocchi con modelli a eventi discreti che comunicano tra di loro. Lo scheduler utilizzato è quello a eventi del SystemC, il quale è più efficiente.

È presente un modulo di testbench che legge i dati di input da un file e li passa all'impianto/controllore.

Come da specifiche il controllore dell'errore è separato dal controllore della funzione k.

Poichè nel Sistema è presente un ciclo, è stato necessario creare un delay sulla porta y_out dell'impianto, in modo tale da permettere uno scheduling accettabile del moduli.

Ciascun modulo ha un time_step di 20 ms.

Il Sistema è implementato dai seguenti moduli:

- **controller error:** possiede le porte:
  - **r_input** che riceve la referenza di input dal testbench
  - **y_input** che riceve la y calcolata dall'impianto
  - **err_out** che fornisce l'errore calcolato al controller

  Questo modulo calcola l'errore tramite la formula

$$e(t) = r(t) - y(t)$$

- **controller:** possiede le porte:
  - **err_input** che riceve l'errore calcolato
  - **k_out** che fornisce il valore della funzione k

  Questo modulo calcola la funzione k dato l'errore, salvandosi in una variabile il valore di k(t-1), inizializzato a 0. La funzione k è la seguente

  $$k(t) = k(t-1) + 100 * [e(t) - e(t-1)] + Ts * e(t)$$

  Dove Ts è pari al time_step in secondi

- **plant:** possiede le porte:
  - **k_input** che riceve il valore della funzione k
  - **y_out** che fornisce il risultato della funzione di trasferimento

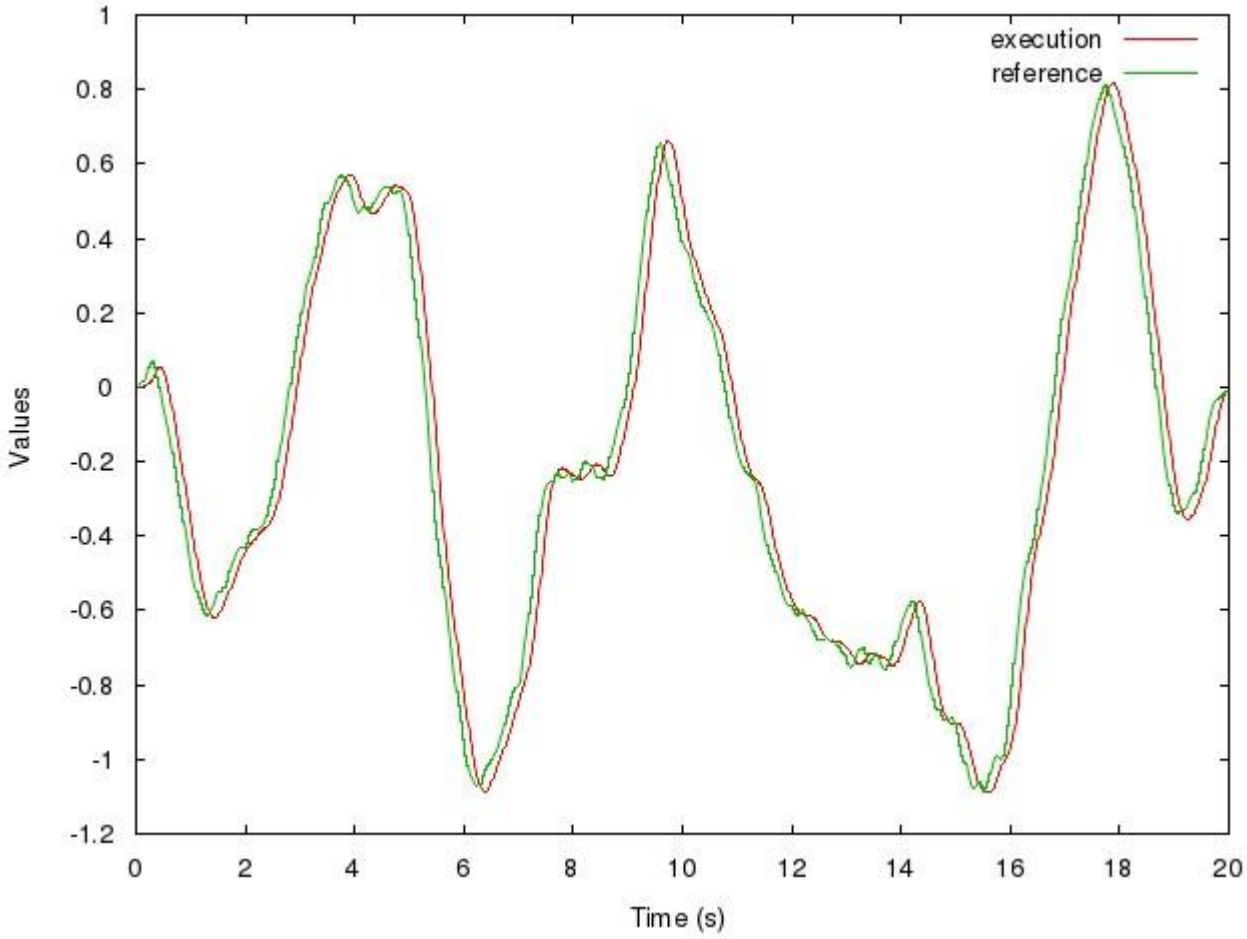
  Questo modulo calcola la funzione di trasferimento:

  $$P(s) = \frac{1}{13s + s^2}$$

  La funzione è calcolata tramite la funzione `sca_tdf::sca_ltf_nd ltf_nd`

Grafico

Il Sistema quindi calcola una y che segue la referenza, come si può vedere nel seguente grafico:



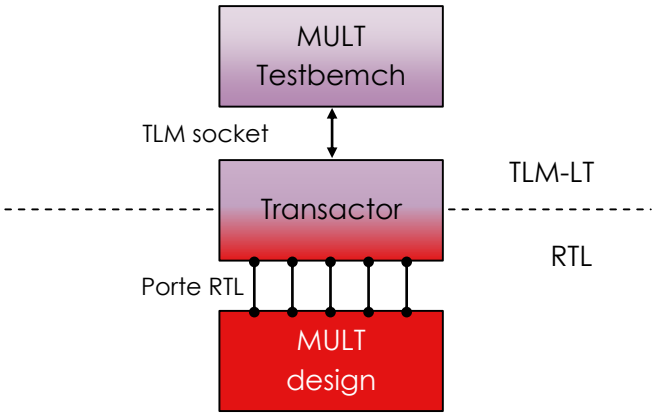
Mixed RTL/TLM modelling : Transactor

Lezione 7a – Transactor TLM/RTL

L'obbiettivo del progetto è realizzare un'interfaccia di collegamento tra il modulo RTL e TLM-LT. Tale interfaccia prende il nome di *transattore*.

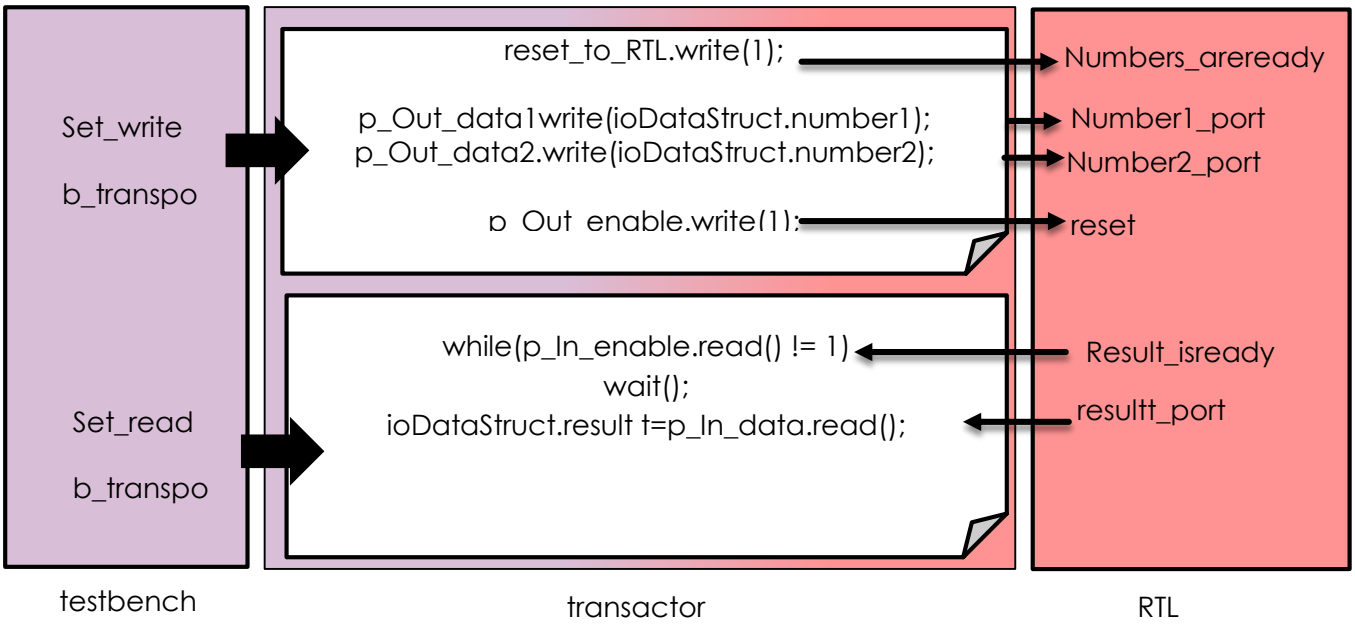
Mapping di porte e segnali tra TLM e RTL

Il transattore, in quanto modulo di collegamento tra i moduli testbench, a livello TLM-LT e moltiplicatore RTL deve mappare correttamente tra loro ingressi e uscite del TLM e dell'RTL. Il transattore riceve i segnali di input dal TLM quando questo invoca la b_transport implementata all'interno del transattore stesso. L'implementazione dei segnali che il TLM passa al transattore, è contenuta nel payload che la b_transport richiede come parametro. Il transattore a sua volta collega il valore dei segnali passati dal TLM con i valori di input dei segnali del modulo RTL, in più pone il segnale numbers_areready uguale a 1 così da permettere l'esecuzione del modulo RTL. A esecuzione terminata il modulo RTL risponde con il segnale di result di result_isready inizializzato a 1, (indicante la fine dell'esecuzione RTL), al transattore, il quale notifica al modulo TLM attraverso lo status "TLM_OK_RESPONSE". Come si può vedere le flags di collegamento tra testbencher e RTL non sono più necessarie in quanto il transattore (modulo intermedio) è in grado di stabilire, e notificare ai rispettivi moduli, partenza e termine dell'esecuzione.



Traduzione da funzioni a operazioni

Il transattore, oltre a mappare i segnali, deve tradurre funzioni a livello TLM in operazioni a livello RTL. Il transattore implementa due thread che identificano rispettivamente la Write e la Read. Tali thread vengono attivate quando il modulo TLM invoca la b_transport settando il parametro trans_command rispettivamente a Write o a Read. Sostanzialmente, quando il TLM richiama la b_transport implementata nel transattore, all'interno della b_transport vi sarà una funzione notify che, a seconda che il trans_command sia Read o Write, "sveglia" la thread Read o Write. Così facendo il transattore fa da perfetto bridge tra primitive ad alto livello TLM e operazioni a basso livello RTL.



Comportamento del transattore

Come già detto in precedenza il transattore è un modulo che si interpone tra testbench TLM e moltiplicatore_RTL. Il suo compito è quello di fare il mapping tra modulo TLM e i segnali del modulo RTL, inoltre deve tradurre le funzioni TLM in operazioni RTL. Il TLM, che è il testbench, fornisce al transattore i valori dei segnali di ingresso per il calcolo dell'algoritmo. L'implementazione di questi segnali è passata al transattore attraverso il payload della b_transport. Il payload, che conterrà quindi i segnali sopra elencati, sarà una struttura i cui campi sono i valori dei segnali da inviare, tale struttura è definita nel file define.h. Successivamente, la b_transport, richiamata dal TLM in modalità di scrittura, esegue un notify a WRITEPROCESS che "sveglia" la thread e esegue l'RTL. Tale thread, che è associata alla funzione di scrittura della b_transport, ha il compito di implementare le porte in ingresso al modulo RTL con i valori del payload e di "lanciare" l'esecuzione del modulo RTL ponendo il segnale number_isready uguale a 1. A questo punto il modulo RTL può procedere con il calcolo dell'algoritmo. In contemporanea il TLM richiama nel transattore la b_transport in modalità Read.
Analogamente per il caso Write, la b_transport conterrà una notify che "sveglia" un READPROCESS che ha il compito di prelevare i risultati dell'algoritmo calcolato dal modulo RTL e inerirli nella structure predefinita. Fatto ciò il READPROCESS notifica alla b_transport che i risultati sono pronti ed essa a sua volta invia al TLM il comando "TLM_OK_RESPONSE", il quale consente al TLM di leggere i risultati contenuti nel payload. Tutto ciò si può vedere nell'output seguente:

o - topl.i_mult_RTL_transactor - reset
o - topl.i_mult_RTL - MULT_FSM
o - topl.i_mult_RTL - MULT

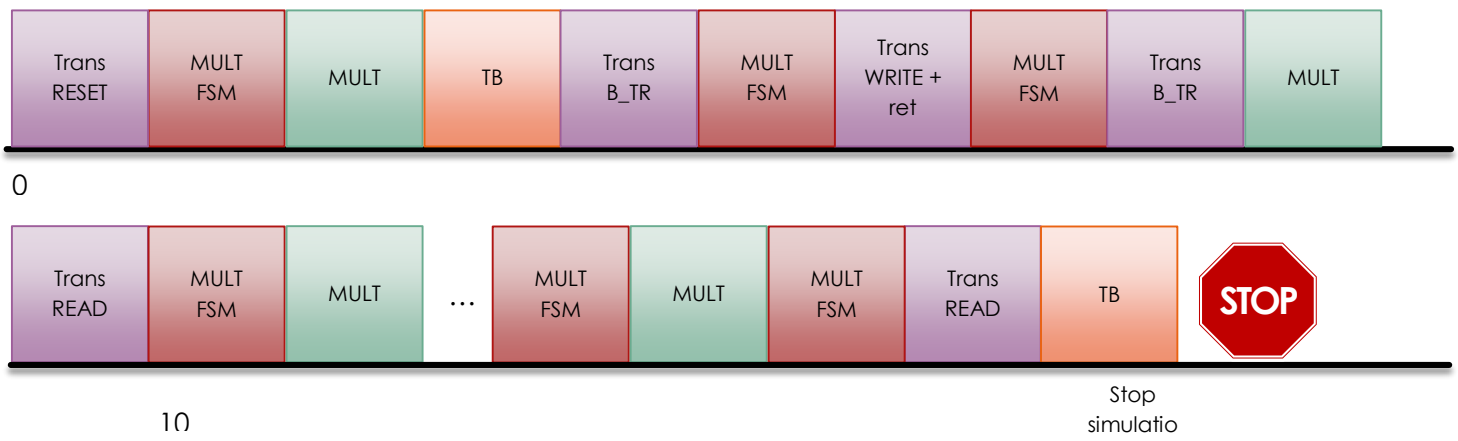
Inizializzazione

o - topl.i_src_LT - run
The multiplication between 2.99 and -71.598
o - topl.i_mult_RTL_transactor - b_transport
o - topl.i_mult_RTL - MULT_FSM

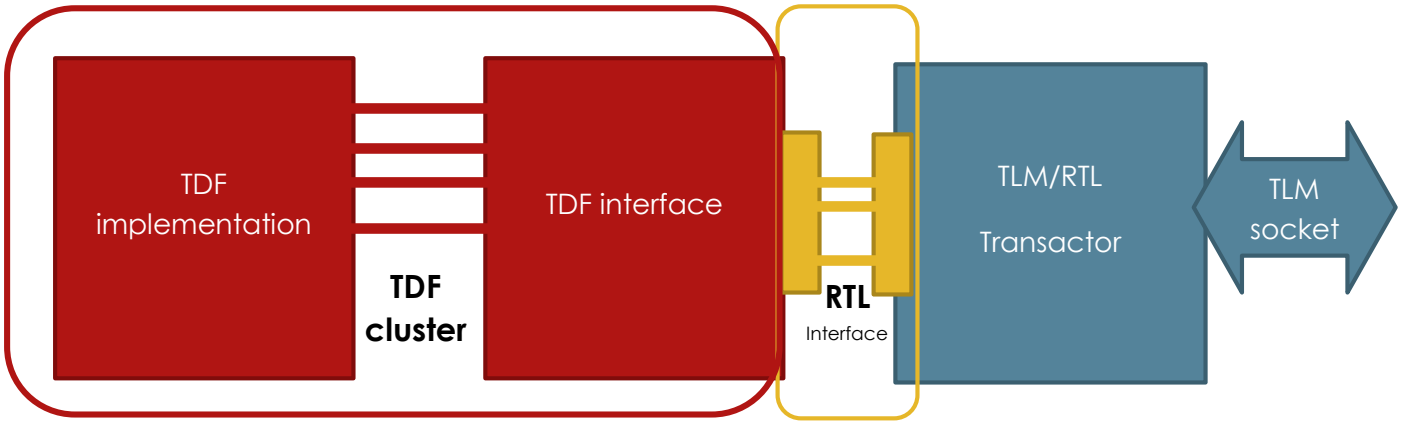
```
        write notified
0 - topl.i_mult_RTL_transactor - notify received
0 - topl.i_mult_RTL_transactor - b_transport ended
0 - topl.i_mult_RTL_transactor - b_transport
0 - topl.i_mult_RTL - MULT
        read notified
0 - topl.i_mult_RTL_transactor - notify received
10 - topl.i_mult_RTL - MULT_FSM
10 - topl.i_mult_RTL - MULT
20 - topl.i_mult_RTL - MULT_FSM
20 - topl.i_mult_RTL - MULT
30 - topl.i_mult_RTL - MULT_FSM
30 - topl.i_mult_RTL - MULT
40 - topl.i_mult_RTL - MULT_FSM
40 - topl.i_mult_RTL - MULT
50 - topl.i_mult_RTL - MULT_FSM
50 - topl.i_mult_RTL - MULT
60 - topl.i_mult_RTL - MULT_FSM
...
2220 - topl.i_mult_RTL - MULT_FSM
2220 - topl.i_mult_RTL - MULT
2230 - topl.i_mult_RTL - MULT_FSM
2230 - topl.i_mult_RTL - MULT
2240 - topl.i_mult_RTL - MULT_FSM
2240 - topl.i_mult_RTL - MULT
2250 - topl.i_mult_RTL - MULT_FSM
2250 - topl.i_mult_RTL - MULT
2260 - topl.i_mult_RTL - MULT_FSM
2260 - topl.i_mult_RTL_transactor - b_transport ended
is: -214.078
Time: 2260 ns + 0 s
SYNCHRONIZING
2260 - topl.i_mult_RTL - MULT
#####################

Info: /OSCI/SystemC: Simulation stopped by user.
```

Elaborazione RTL



Lezione 7a – Transactor TLM/AMS



Per quanto riguarda il transattore TLM/AMS il funzionamento è simile a quanto descritto per il transattore TLM/RTL, con la differenza che il transattore non può comunicare direttamente con il cluster TDF ma deve passare I segnali ad un ulteriore adattatore RTL il quale "traduce" il segnale dalle porte RTL a quelle sca_de, le quali permettono di comunicare direttamente con AMS. Un'altra differenza rispetto al transattore TLM/RTL è che I processi READPROCESS e WRITEPROCESS non sono soggetti al clock, in quanto non è presente il clock in AMS ma si vuole simulare un Sistema continuo di elaborazione dati. Il Sistema AMS è lo stesso della Lezione 6, mentre il testbench TLM è molto simile. Quest'ultimo crea il payload con la referenza letta da file, e chiama la b_trasport sul transattore TLM, il quale scrive la referenza sull'interfaccia RTL che la passa ad AMS. Il testbench poi richiama la b_transport per la lettura del risultato il quale passa sempre dall'interfaccia RTL.

Assertion

Lezione 7b - Assertion

L'obbiettivo di questa consegna è creare un simulatore che verifichi la corretta, o meno, esecuzione del sistema, valutando se esso soddisfa o meno tre proprietà. Le proprietà, come da richiesta, devono rispettivamente: le prime due definire una caratteristica sempre vera del sistema e l'ultima una caratteristica sempre falsa.

Ho implementato tre proprietà:

Proprietà 1

La proprietà 1 è la seguente:

```
void mult_RTL :: property1(void){
  // if numberss_areready == 1 then result_isready == 1
```

```
  // in less than 100 clock cycles

  int count = 0;
  bool true_property = false;

  while(true){
    if (numbers_areready.read() == 1){
      while((count < 100)&(!true_property)){
      wait();
      if (result_isready.read() == 1)
            true_property = true;
      }

      if (true_property)
        cout<<"\033[34m"<<"[CHECK: ] property 1 is true"<<"\033[0m"<<endl;
      else {
           cout<<"\033[31m"<<"[CHECK: ] property 1 is false"<<"\033[0m"<<endl;
              sc_stop();
      }
      count = 0;
      true_property = false;
    }
    wait();
  }
}
```

Tale proprietà è sempre vera nel mio codice. Come si vede nella simulazione sottostante.

Proprietà 2

La proprietà 2 è la seguente:

```
void mult_RTL :: property2(void){

// if STATUS = ST_8 and the number is still not normalized

  bool true_property = false;
  while(true){
   if(STATUS.read() == ST_8) {
     if(normalized.read() == false)
       cout<<"\033[34m"<<"[CHECK: ] property 2 is true"<<"\033[0m"<<endl;
      else{
     cout<<"\033[31m"<<"[CHECK: ] property 2 is false"<<"\033[0m"<<endl;
           //sc_stop();
     }
    }
      wait();
  }
```

Tale proprietà è sempre vera nel mio codice. Come si vede nella simulazione sottostante.

Proprietà 3

La proprietà 3 è la seguente:

```
void mult_RTL :: property3(void){
```

```
// il STATUS = ST_8 and the multiplication of the significands hasn't
finished yet
bool true_property = false;
 while(true){
   if(STATUS.read() == ST_8) {
     if(stop_m_mult.read() == false)
       cout<<"\033[34m"<<"[CHECK: ] property 3 is true"<<"\033[0m"<<endl;
     else{
      cout<<"\033[31m"<<"[CHECK: ] property 3 is false"<<"\033[0m"<<endl;
            //sc_stop();
     }
   }
    wait();
}
```

Tale proprietà è sempre falsa nel mio codice. Come si vede nella parte di simulazione sotto.

Simulazione

```
[CHECK: ] property 3 is false
[CHECK: ] property 2 is true
       2230 ns - mult: ST_8
       2230 ns - mult: MULT
       2240 ns - mult: ST_9
       2240 ns - mult: MULT
[CHECK: ] property 3 is false
[CHECK: ] property 2 is true
       2250 ns - mult: ST_8
       2250 ns - mult: MULT
       2260 ns - mult: ST_9
       2260 ns - mult: MULT
[CHECK: ] property 3 is false
[CHECK: ] property 2 is true
       2270 ns - mult: ST_8
       2270 ns - mult: MULT
       2280 ns - mult: ST_9
       2280 ns - mult: MULT
       2290 ns - mult: ST_10
       2290 ns - mult: MULT
       2300 ns - mult: ST_11
       2300 ns - mult: MULT
       2310 ns - mult: ST_12
       2310 ns - mult: MULT
       2320 ns - mult: ST_0
       is: 8688.291385999998965417034924030303955078125000000000000
       2320 ns - tb: result available
       multiplication was right


********************************************************************************


       2320 ns - tb: reset
[CHECK: ] property 1 is true
```

Platform

Lezione 8 – Platform

Creare una Piattaforma che unisca tutti i componenti creati precedentemente in modo da formare un sistema unico. La piattaforma deve avere le seguenti caratteristiche:

- Il risultato del moltiplicatore è la referenza per l'impianto.
- Il testbench del moltiplicatore è un component "SW" implementato in TLM
  - Fornisce input al moltiplicatore
  - Legge gli output del moltiplicatore
  - In accordo con il ritardo di input/output, controlla se il risultato dell'impianto è all'interno di una certa threshold (e.g., 10%).
  - Stampa un errore nell'stdout o stderr in caso di threshold mancata
- L'impianto implementa i processi fisici descritti nelle lezioni precedente.

Implementazione

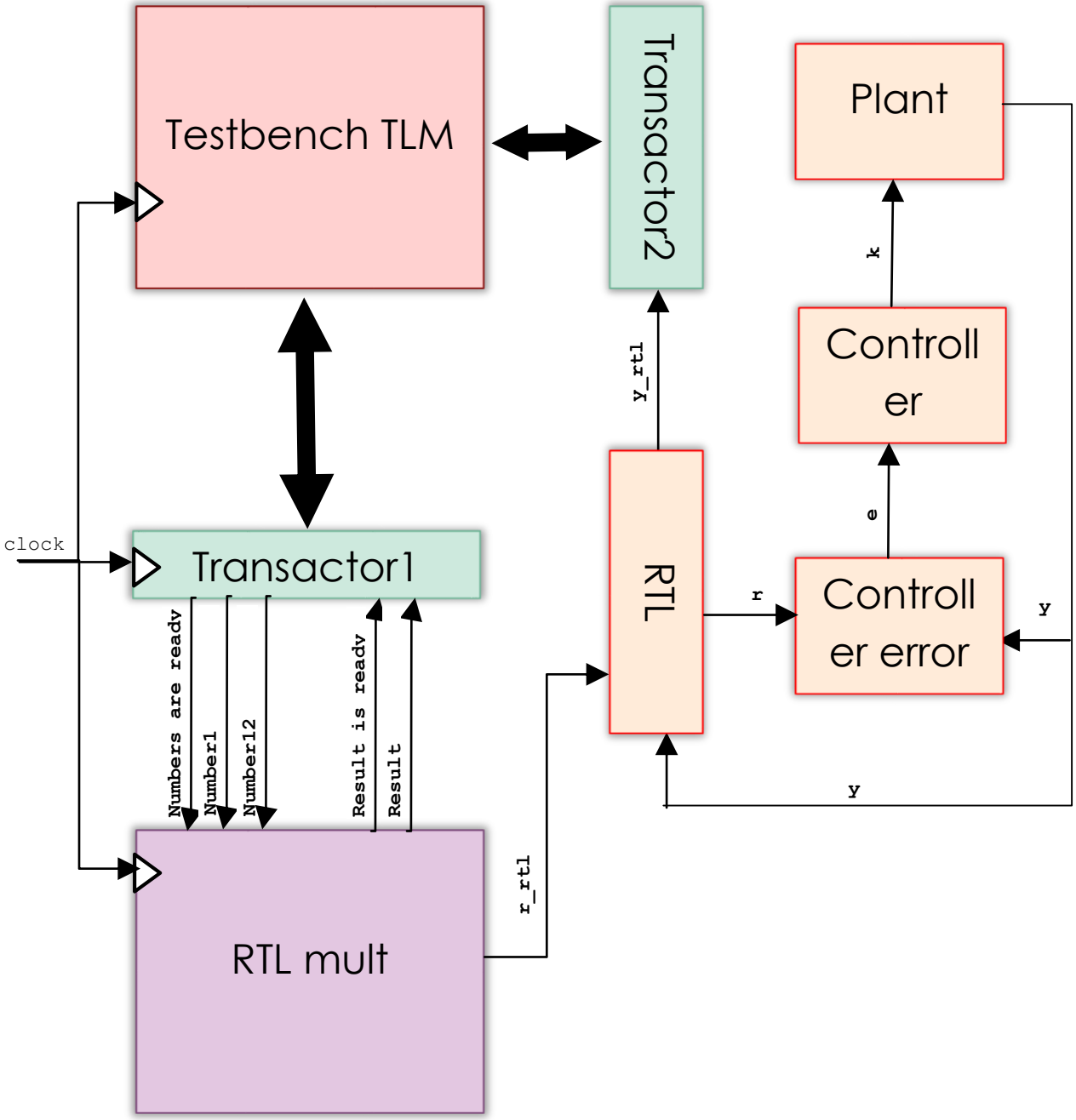
Prendendo i vari componenti dagli assegnamenti precedenti, ho collegato il testbench TLM a due transattore: uno per l'impianto/controllore AMS e uno per il moltiplicatore RTL.
L'interfaccia RTL di comunicazione con AMS è collegata al moltiplicatore, il quale fornisce l'input per l'impianto e con il transattore2 il quale comunica con il socket TLM. Il testbench fornisce 360 coppie di numeri al moltiplicatore RTL. Questi numeri sono tali che se moltiplicati formano la funzione seno di un angolo progressivo:

$$n1 = \frac{1}{i} * \sin(i)$$

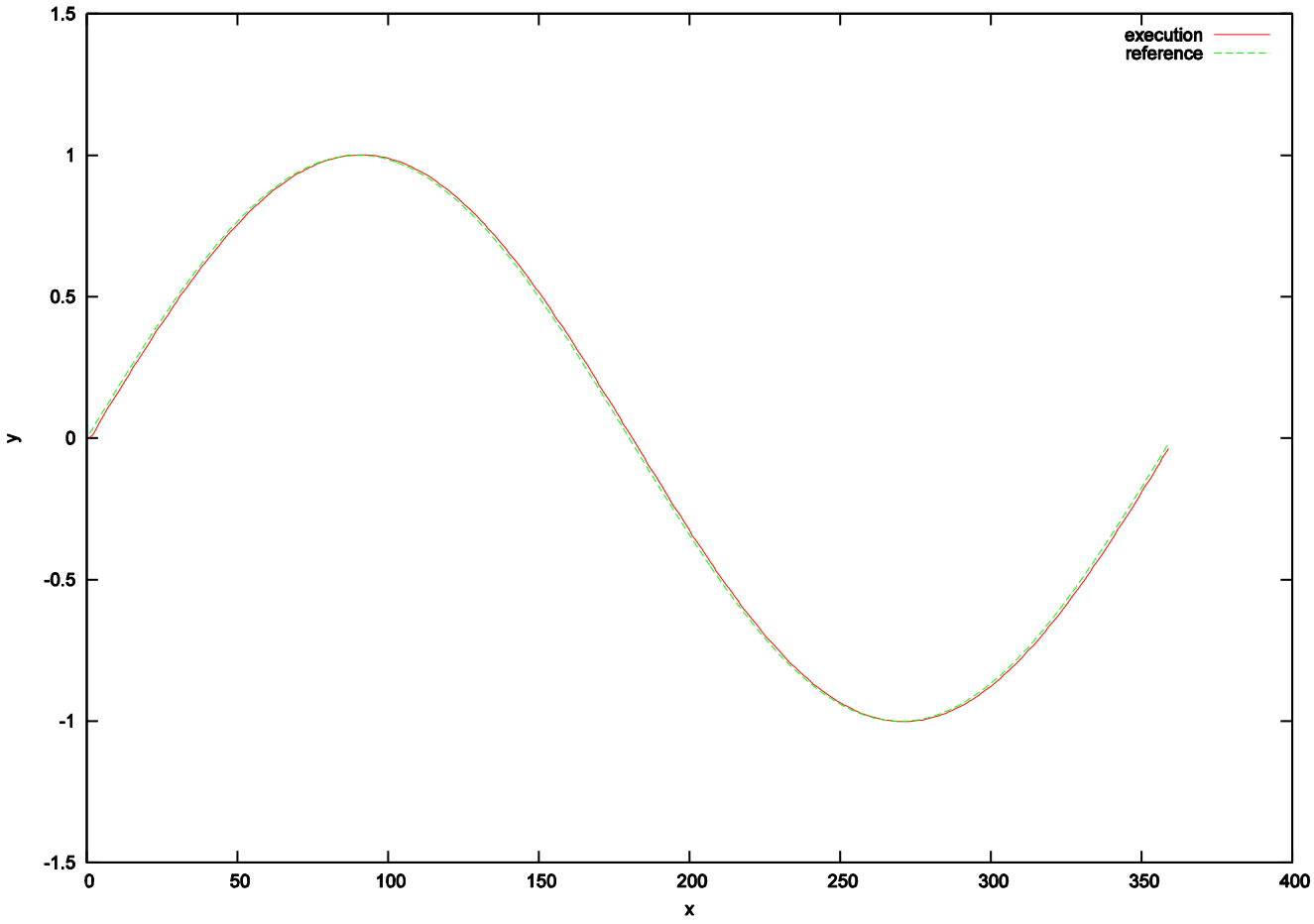
$$n2 = i$$

In questo modo forniscono una referenza di una funzione abbastanza regolare per l'impianto controllore, il quale non ha difficoltà a seguirla.
Il testbench richiama la b_transport sul socket del moltiplicatore per passare i due numeri da moltiplicare. Successivamente richiama due b_transport in lettura sui due socket, separate da una wait di 40 ms in modo da permettere all'impianto (che ha un timestep di 20 ms) di riuscire a calcolare i dati prima che il moltiplicatore finisca di moltiplicare tutti i 360 numeri. Infatti quest'ultimo ha una frequenza molto più alta dell'impianto (il periodo del clock è stato modificato a 0.1 ms, in modo da rincronizzare meglio la piattaforma). Il moltiplicatore fornisce i risultati che vengono passati sul socket TLM e salvati in un buffer nel testbench. L'impianto non inizia subito a produrre i risultati, ma dopo alcune moltiplicazioni. Segnando quindi con un contatore il ritardo dell'impianto rispetto al moltiplicatore, si riesce a determinare la referenza corrispondente all'output dell'impianto. Questi due vengono quindi confrontati per vedere se l'errore rispetto alla referenza è superiore al 10%, in questo caso viene stampato un messaggio di errore. Dall'output si può notare che all'inizio l'impianto ha un errore superiore al 10%, che però viene corretto poco dopo.

Schema dei componenti



Segue il grafico referenza/output dell'impianto AMS:



VHDL modelling al livello RTL

Lezione 10 – VHDL modeling

Scrivere il modulo Mult in VHDL.

VHDL

La descrizione del device è divisa in tre parti. La prima è la *package declaration*. Nel mio caso, il packages è usato per definire alcune costanti:

```
PACKAGE mult_pack IS
  CONSTANT SIZE : INTEGER := 64;
  CONSTANT Reset_ST : INTEGER := 0;
  CONSTANT ST_0 : INTEGER := 1;
  CONSTANT ST_1 : INTEGER := 2;
  CONSTANT ST_2 : INTEGER := 3;
  CONSTANT ST_3 : INTEGER := 4;
  CONSTANT ST_4 : INTEGER := 5;
  CONSTANT ST_5 : INTEGER := 6;
  CONSTANT ST_6 : INTEGER := 7;
  CONSTANT ST_7 : INTEGER := 8;
  CONSTANT ST_8 : INTEGER := 9;
  CONSTANT ST_9 : INTEGER := 10;
  CONSTANT ST_10 : INTEGER := 11;
  CONSTANT ST_11 : INTEGER := 12;
  CONSTANT ST_12 : INTEGER := 13;
END mult_pack;
```

La seconda parte della descrizione del mio modulo mult è la *entity definition*:

```
entity mult is
  port (
    clock, reset   : in  bit;
    numbers_areready: in std_logic;
    number1_port: in std_logic_vector (SIZE-1 DOWNTO 0);
    number2_port: in std_logic_vector (SIZE-1 DOWNTO 0);

    result_isready: out bit;
    result_port: out std_logic_vector (SIZE-1 DOWNTO 0)
  );

end mult;
```

Una volta che la sua interfaccia è stata specificata in una *entity declaration*, una o più implementazioni di una entity possono essere descritte in più *architecture bodies*. La dichiarazione in un architecture body definisce gli oggetti che saranno usati per costruire una descrizione strutturale.

```
architecture mult of mult is

       subtype status_t is integer range 0 to 13;

       --definisco i segnali
       signal STATUS: status_t;
       signal NEXT_STATUS: status_t;
       signal overflow: std_logic;
       signal normalized: std_logic;
       signal stop_m_mult: std_logic;

       --definisco le costanti
       CONSTANT ZERO11 : std_logic_vector (10 DOWNTO 0) := "00000000000";
  CONSTANT ONE11 : std_logic_vector (10 DOWNTO 0) :=  "11111111111";
       CONSTANT ZERO53 : std_logic_vector (52 DOWNTO 0) :=
"0000000000000000000000000000000000000000000000000000";
       CONSTANT ZERO64 : std_logic_vector (63 DOWNTO 0) :=
"0000000000000000000000000000000000000000000000000000000000000000";
       CONSTANT ZERO75 : std_logic_vector (74 DOWNTO 0) :=
"000000000000000000000000000000000000000000000000000000000000000000000000000";
       CONSTANT ZERO128 : std_logic_vector (127 DOWNTO 0) :=
"00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000";



  --inizio a descrivere il comportamento della mia architettura
 begin …
```

Il modulo mult è implementato attraverso due processi:

- Il primo è del quarto tipo: esso è sensibile al clock e ha un segnale di reset asincrono. Il processo aggiorna il valore dello stato corrente, esso legge dalle porte di input e scrive nelle porte di output.
- Il secondo è di tipo uno: esso è sensibile a tutti i senali letti e descrive un circuito combinatorio. Questo processo calcola lo stato prossimo e l'evoluzione del sistema.

L'algoritmo utilizzato è lo stesso dell'implementazione in SystemC RTL.

Simulazione

Per la simulazione ho utilizzato il file qui sotto riscritto:

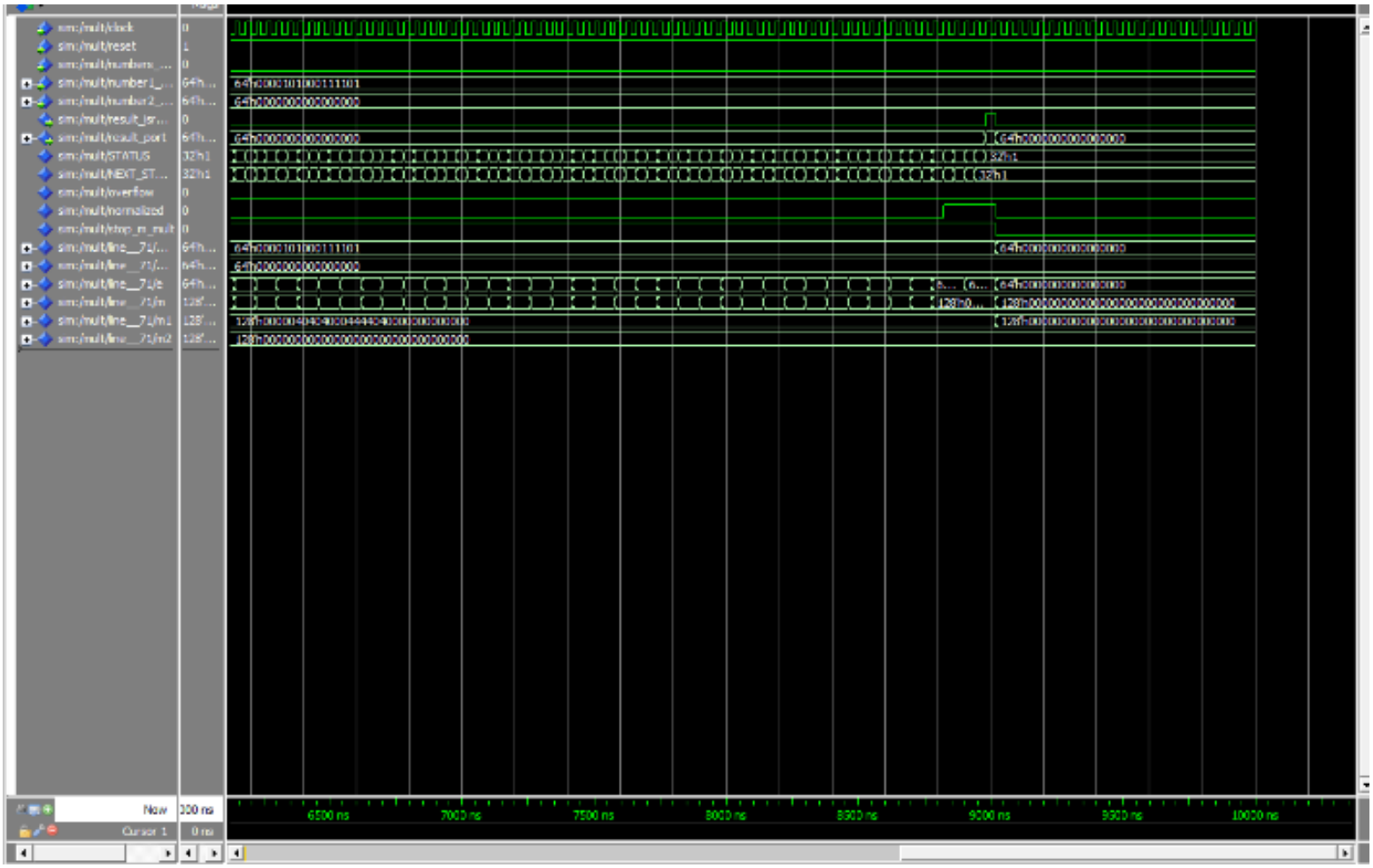
```
add wave *
add wave -position insertpoint /mult/line__71/number1_tmp
add wave -position insertpoint /mult/line__71/number2_tmp
add wave -position insertpoint /mult/line__71/e
add wave -position insertpoint /mult/line__71/m
add wave -position insertpoint /mult/line__71/m1
add wave -position insertpoint /mult/line__71/m2

force clock 1 20 ns, 0 40 ns -repeat 40
force reset 1 0

force number1_port
0100000001011111110111010111000010100011110101110000101000111101 0
force number2_port
0100000001000101000000000000000000000000000000000000000000000000 0
force numbers_areready 1 0 ns, 0 120 ns
run 10000
```

Ottengo quindi il seguente waveform:

Come si può notare la waveform i segnali di STATUS e NEXT_STATUS cambiano seconda dei due processi scritti.

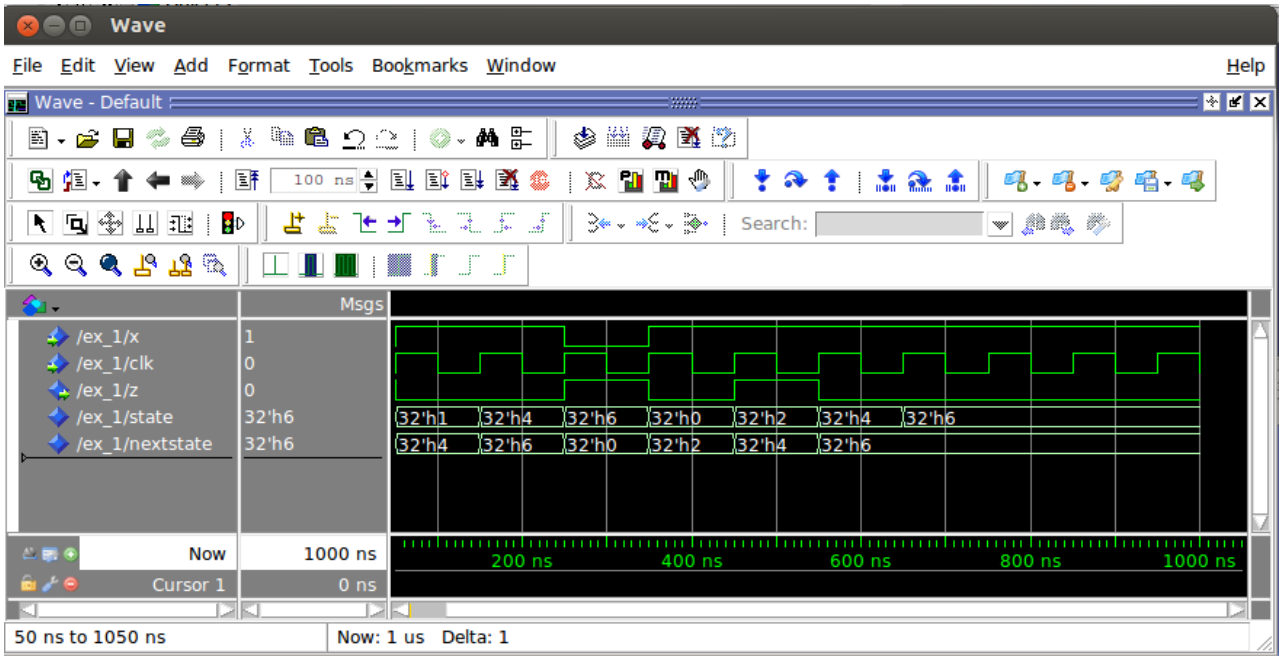


Timing simulation in VHDL

Lezione 11- VHDL timing simulation

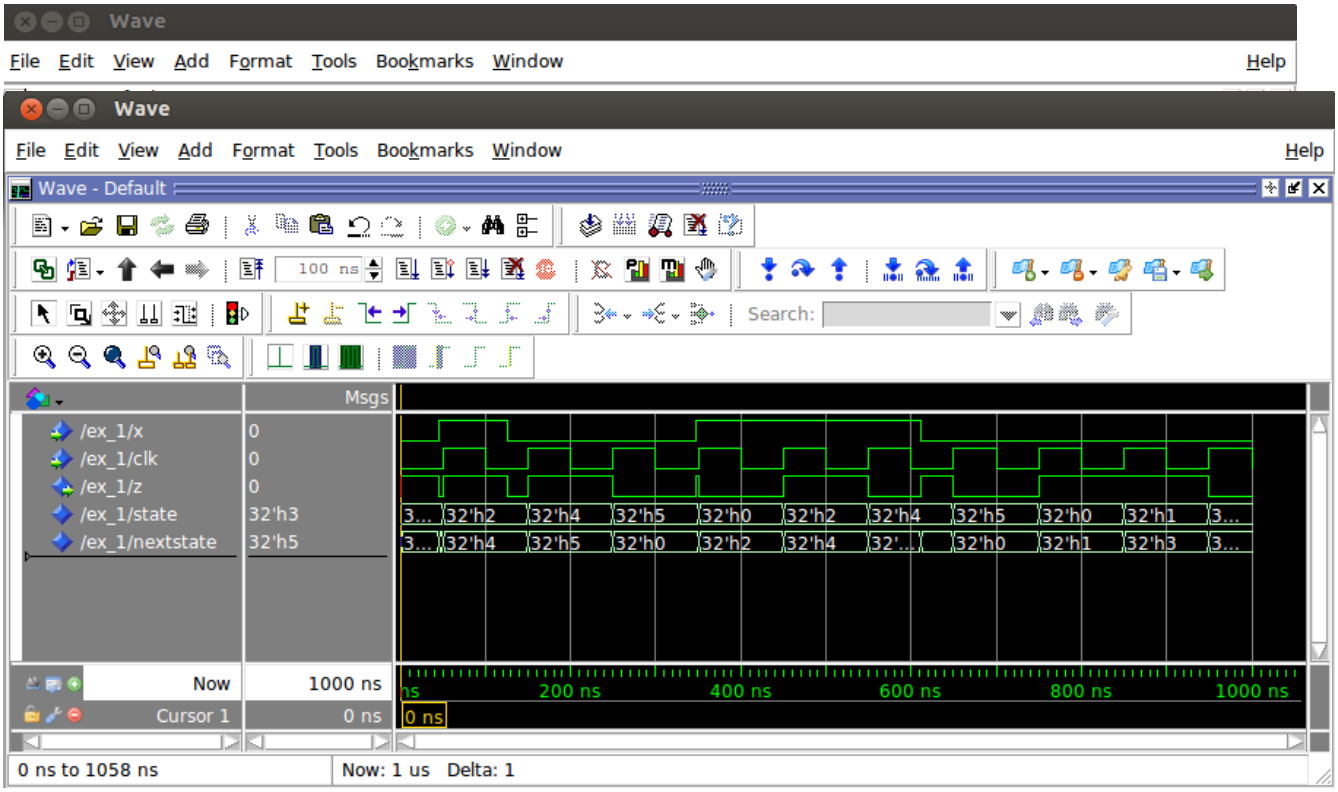
Questa lezione ha come obbiettivo l'analisi dei comportamento dei segnali in VHDL. Attraverso i seguenti esercizi:

Ex_1



Ex_1 stimuli_1

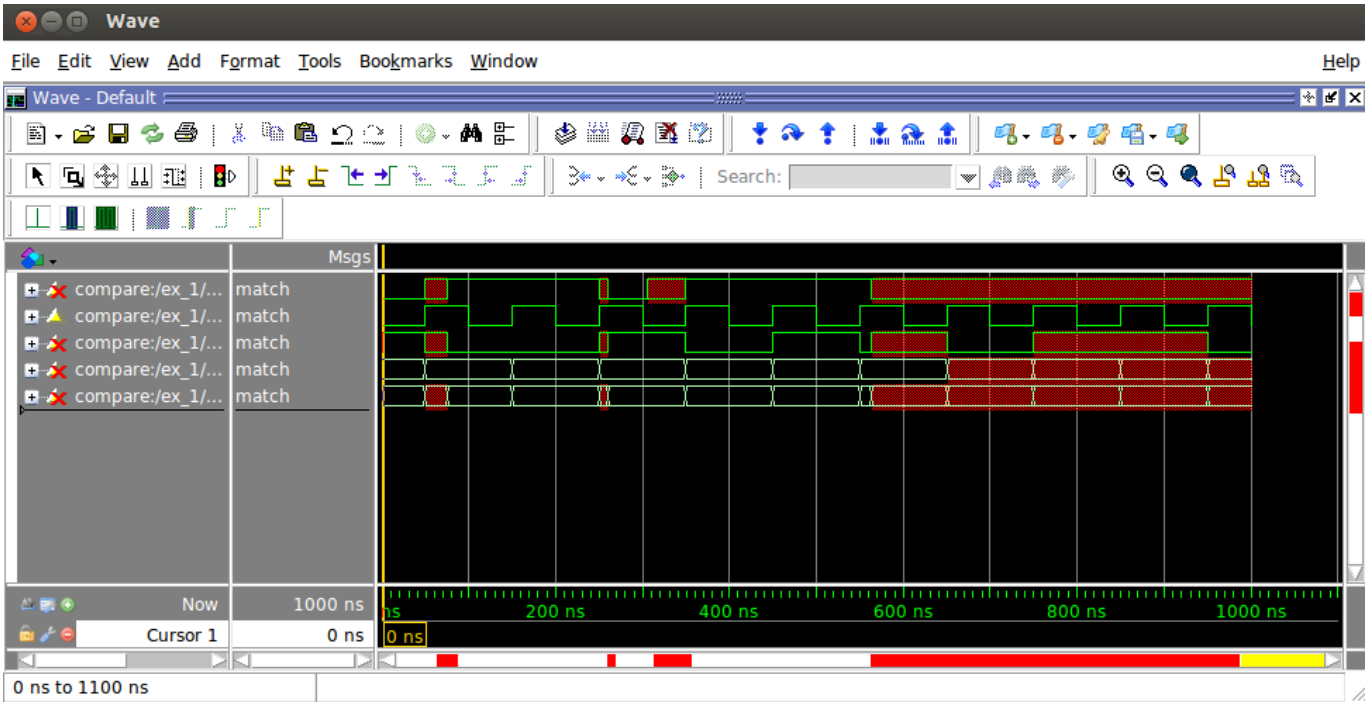
Ex_1 stimuli_2



Ex_1 stimuli_3

Confronti

Ex_1 stimuli_1 e Ex_1 stimuli_2

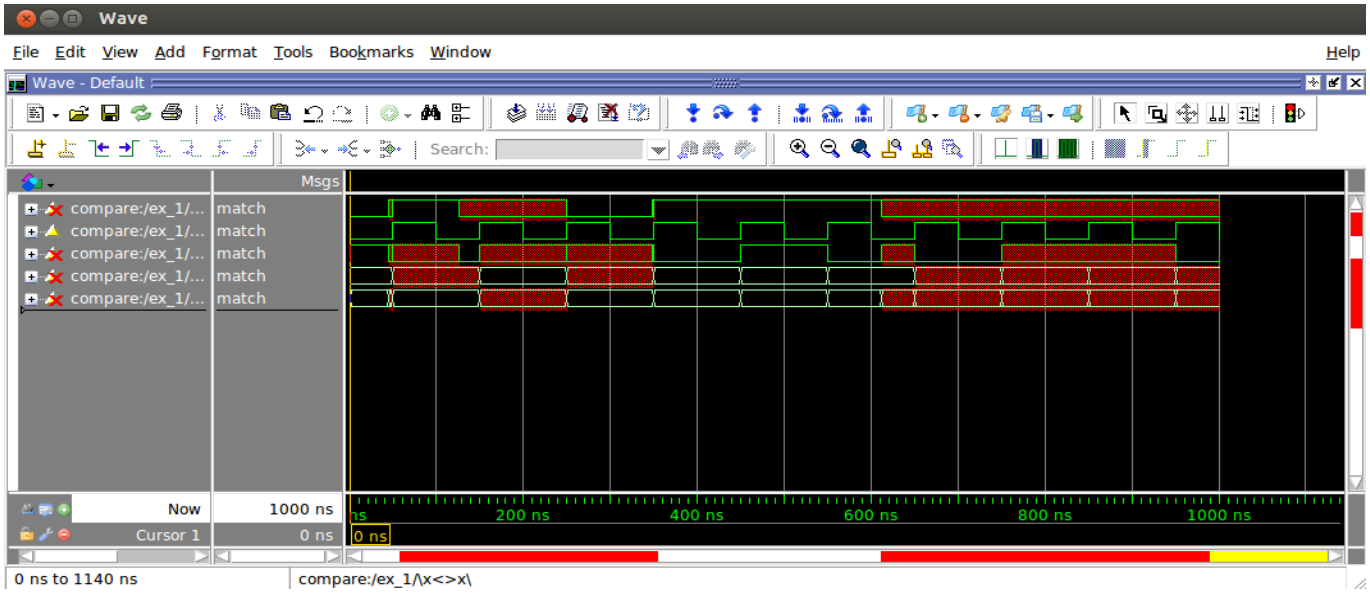


Come si può vedere ci sono molteplici differenze. In particolare si può notare come un ritardo nell'assegnamento del segnale provochi il medesimo ritardo nel cambiamento del valore z e di nextstate ma non influisce sullo stato in quanto il cambiamento è inferiore ad un ciclo di clock. Negli istanti tra 305 e 350 invece il cambiamento non si propaga su z e nextstate perchè essendo nello stato 6 non o una transazione per x='1' ma ho solo quella per x='0'come si vede dal codice:

```
when 6 => if x='0' then z<='1'; nextstate<=0; end if;
```

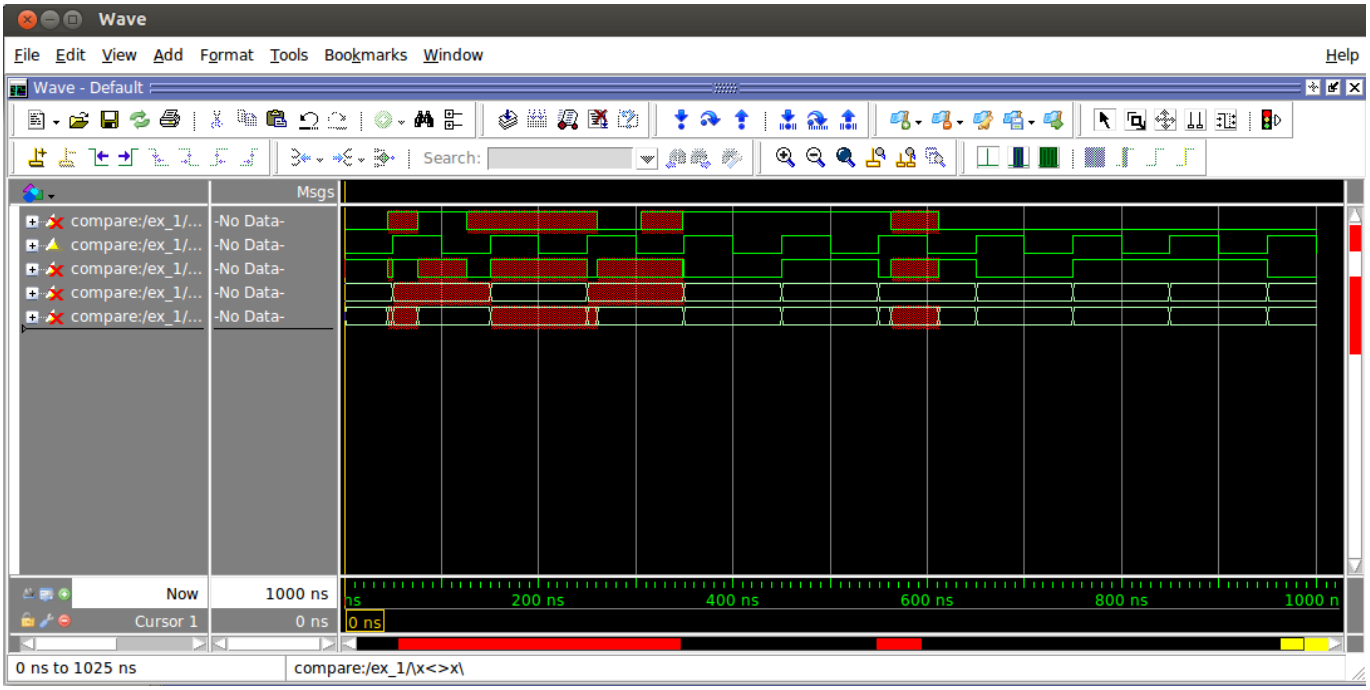
Alla fine invece un cambiamento permanente del segnale influisce immediatamente su nextstate e dal ciclo successivo anche su state.

Ex_1 stimuli_1 e Ex_1 stimuli_3



Come si può vedere ci sono molteplici differenze. In particolare si può notare come un anticipo nell'assegnamento del segnale x provochi cambiamenti del valore z, di nextstate e influisce anche sullo stato. Alla fine invece un cambiamento permanente del segnale influisce immediatamente su nextstate e dal ciclo successivo anche su state.

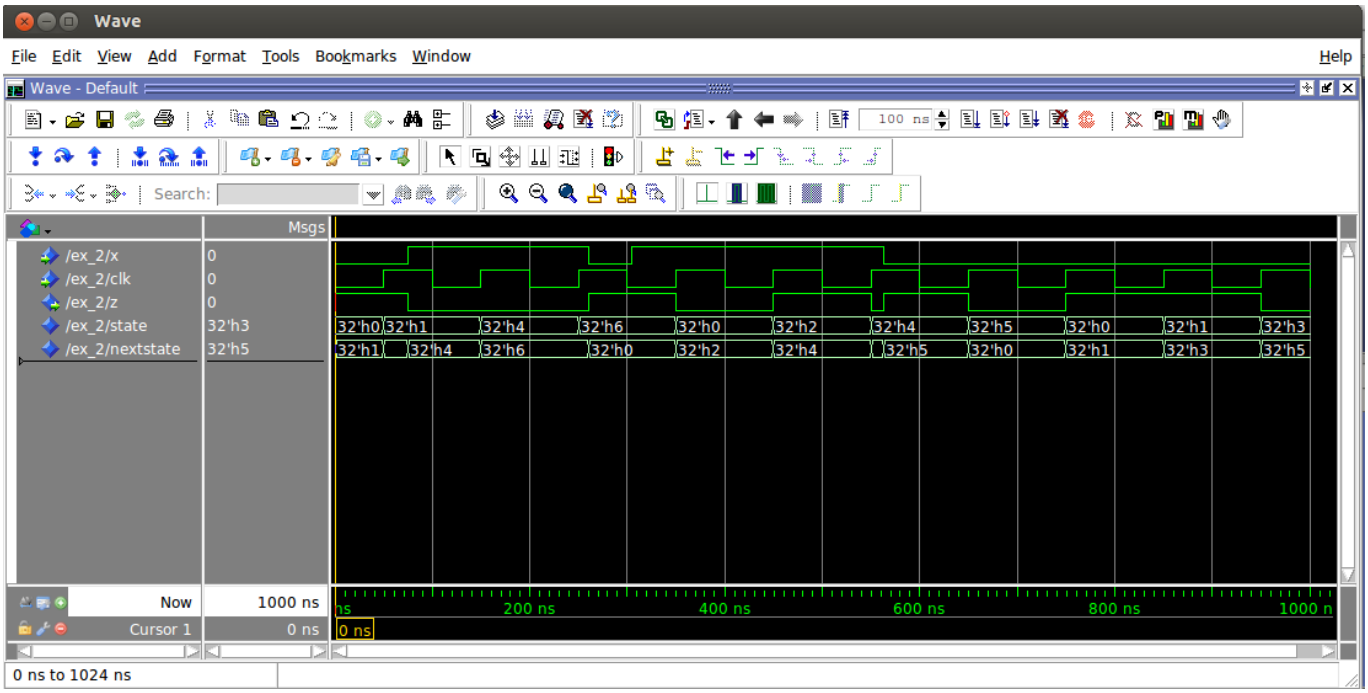
Ex_1 stimuli_2 e Ex_1 stimuli_3



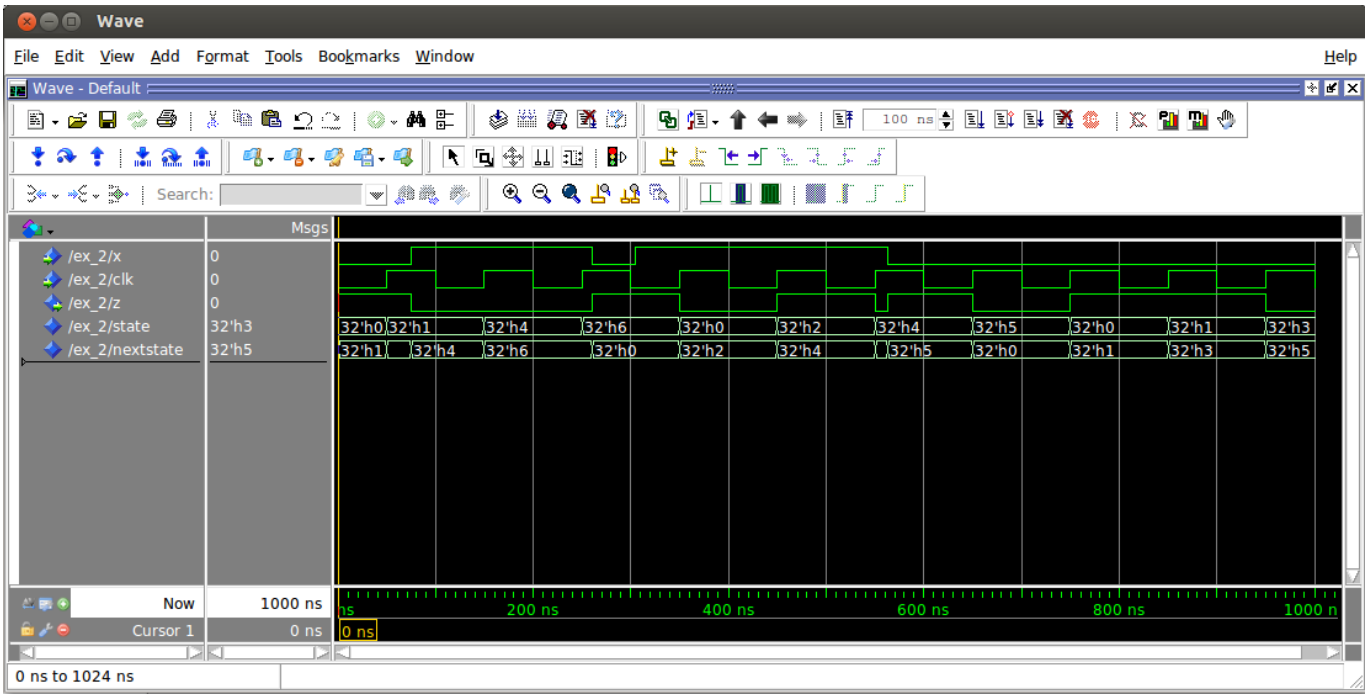
Come si può vedere ci sono molteplici differenze. In particolare si può notare come un anticipo nell'assegnamento del segnale x provochi cambiamenti del valore z, di nextstate e influisce anche sullo stato. Alla fine invece in entrambi gli stimoli a x si assegna 0 a istanti diversi (563ns e 612ns) questo provoca cambiamenti sul tempo di assegnamento di z e nextstate ma non varia per lo state in quanto il cambiamento è inferiore ad un ciclo di clock e il cambiamento non si propaga.

Ex_2

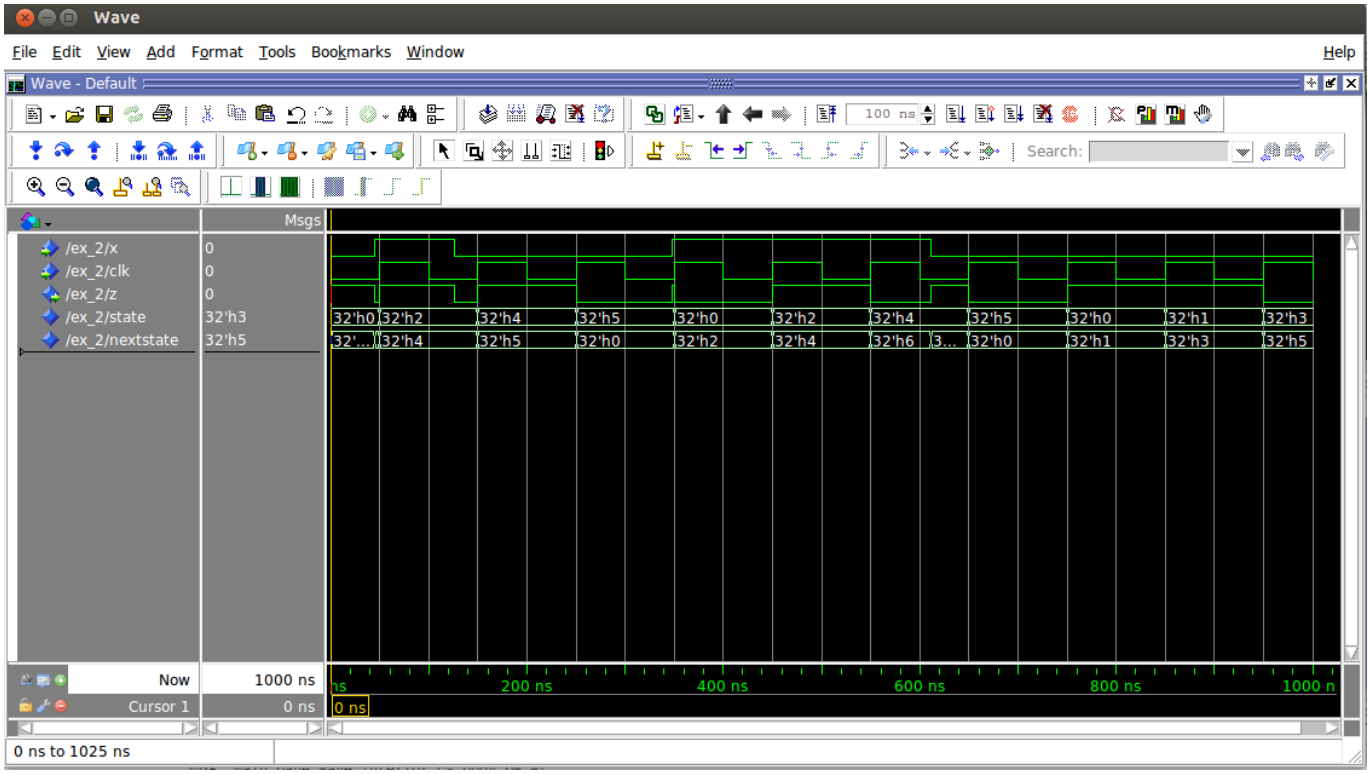
Ex_2 stimuli_1



Ex_2 stimuli_2



Ex_2 stimuli_3



Confronti

Le differenze tra ex_1 e ex_2 consiste nel fatto che nell'ex_2 al posto di usare il process sensibile al clock utilizzo una wait on clock:

```
1. process(clk) -- state Register
     begin
         if clk='1' then -- rising edge of clock
           state <= nextstate;
         end if;
     end process;

2. wait on clk, x;
   if (clk'event and clk = '1') then
          state <= nextstate;
          wait for 0 ns;        -- wait for state to be updated
```

Ex_1 stimuli_1 e Ex_2 stimuli_1

Come si può vedere dalle simulazioni il comportamento per ex_1 e ex_2 è uguale.

Ex_1 stimuli_2 e Ex_2 stimuli_2

Come si può vedere dalle simulazioni il comportamento per ex_1 e ex_2 è uguale.

Ex_1 stimuli_3 e Ex_2 stimuli_3

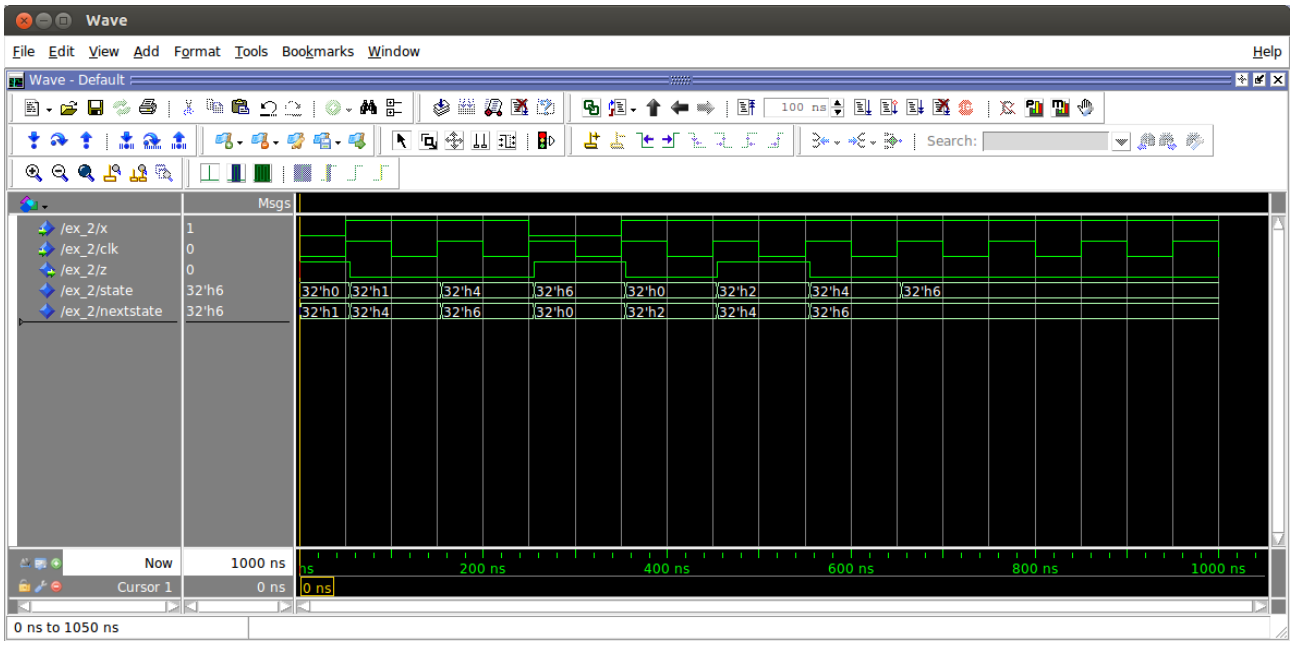
Come si può vedere dalle simulazioni il comportamento per ex_1 e ex_2 è uguale.

Ex_3

Nell'esercizio 3 devo considerare le varie versioni dell'esercizio 2:

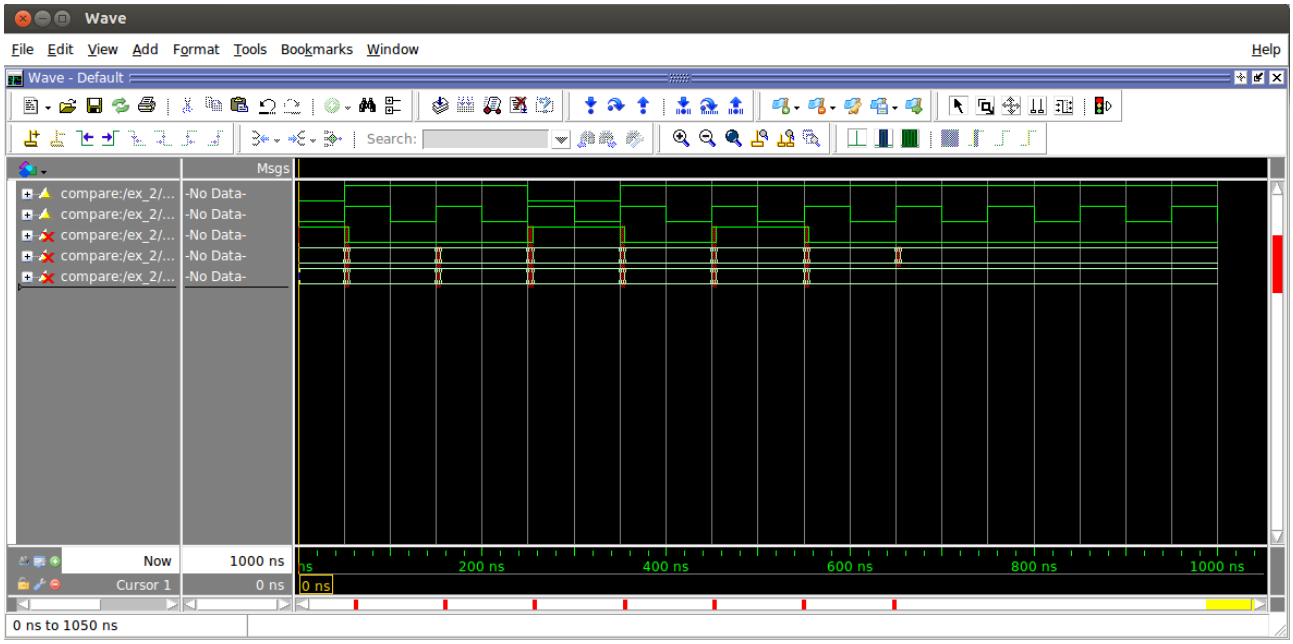
Versione a)

Questo è il risultato della versione a) simulato con il stimuli_1.



Differenza con ex_2

Questa è la differenza tra l'esercizio_2 simulato con lo stimuli_1 e la versione a) dell'esercizio_2 con lo stesso simuli:

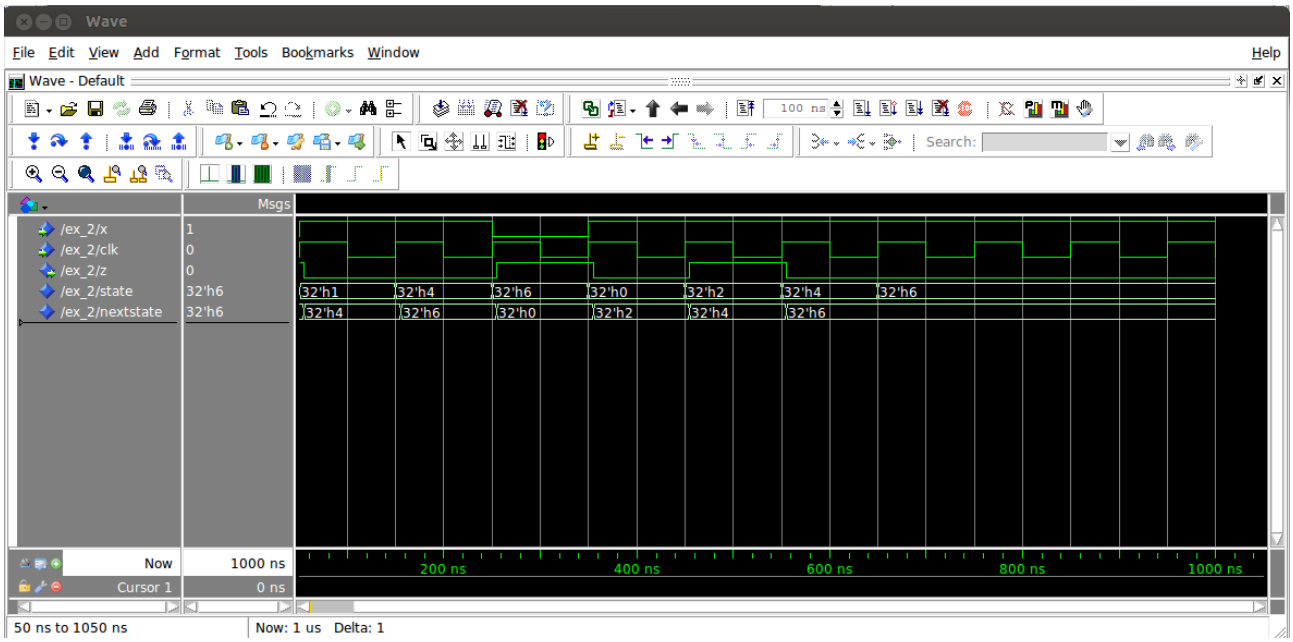


Come si può notare la variazione è solo nei segnali nextstate, z che vengono ritardati 5ns. Questo ritardo è dovuto al wait per aggiornare lo stato che da 0 ns in questa versione viene assegnato a 5ns. L'altra modifica fatta in questa versione non ha influenza sui tempi di x e

neststate come si può vedere nella versione b) ma ha influenza sui nanosecondi in cui viene aggiornato state che subiscono 5 ns di ritardo.

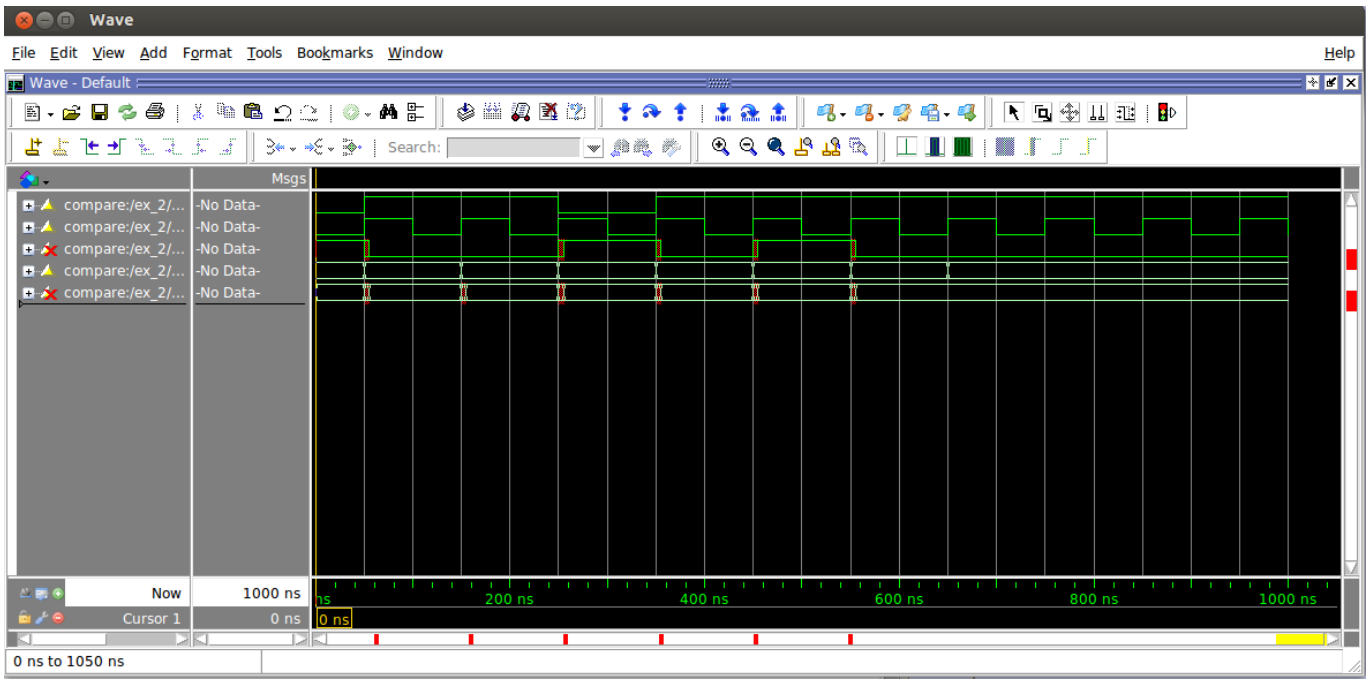
Versione b)

Questo è il risultato della versione b) simulato con il stimuli_1, come si può notare è identica quella della versione a) tranne che per il segnale state.



Differenza con ex_2

Questa è la differenza tra l'esercizio_2 simulato con lo stimuli_1 e la versione b) dell'esercizio_2 con lo stesso simuli

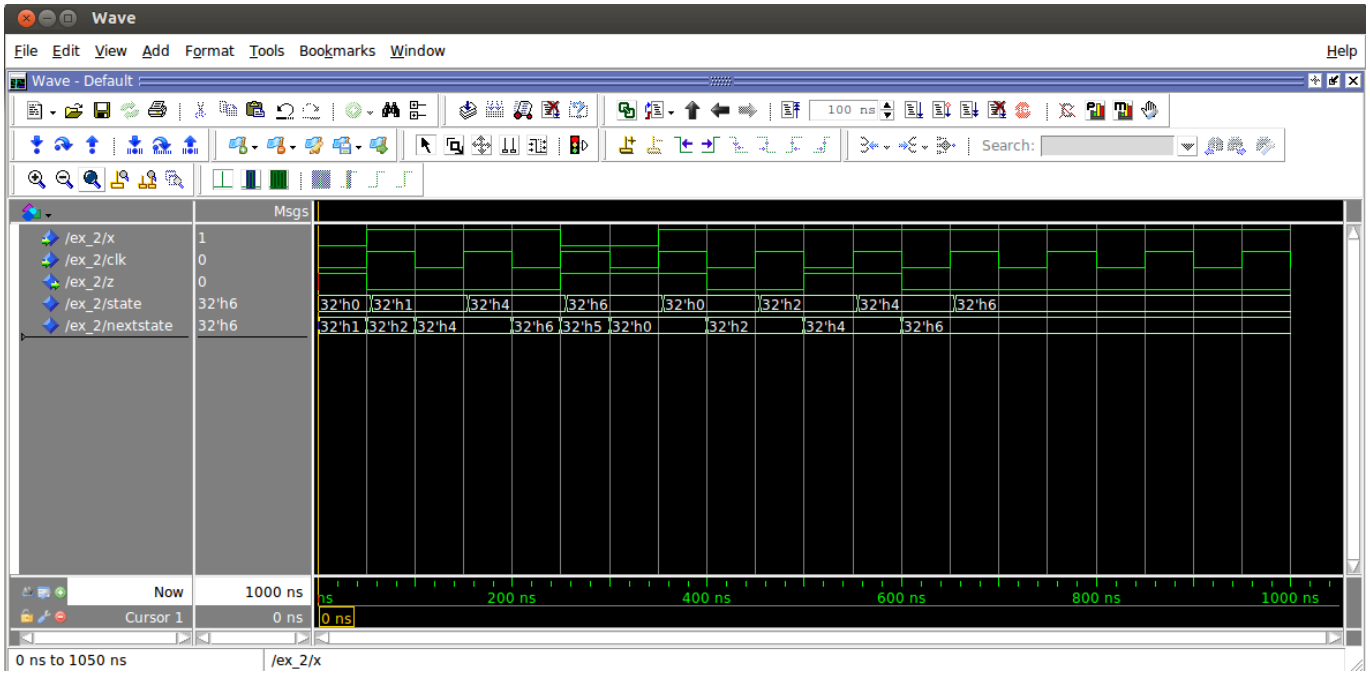


Come si può notare la variazione è solo nei segnali z e nextstate che vengono ritardati 5ns. Questo ritardo è dovuto al wait per aggiornare lo stato che da 0 ns in questa versione viene

assegnato a 5ns. Il segnale state non viene ritardato in quanto l'assegnamento a state è pressoché immediato.

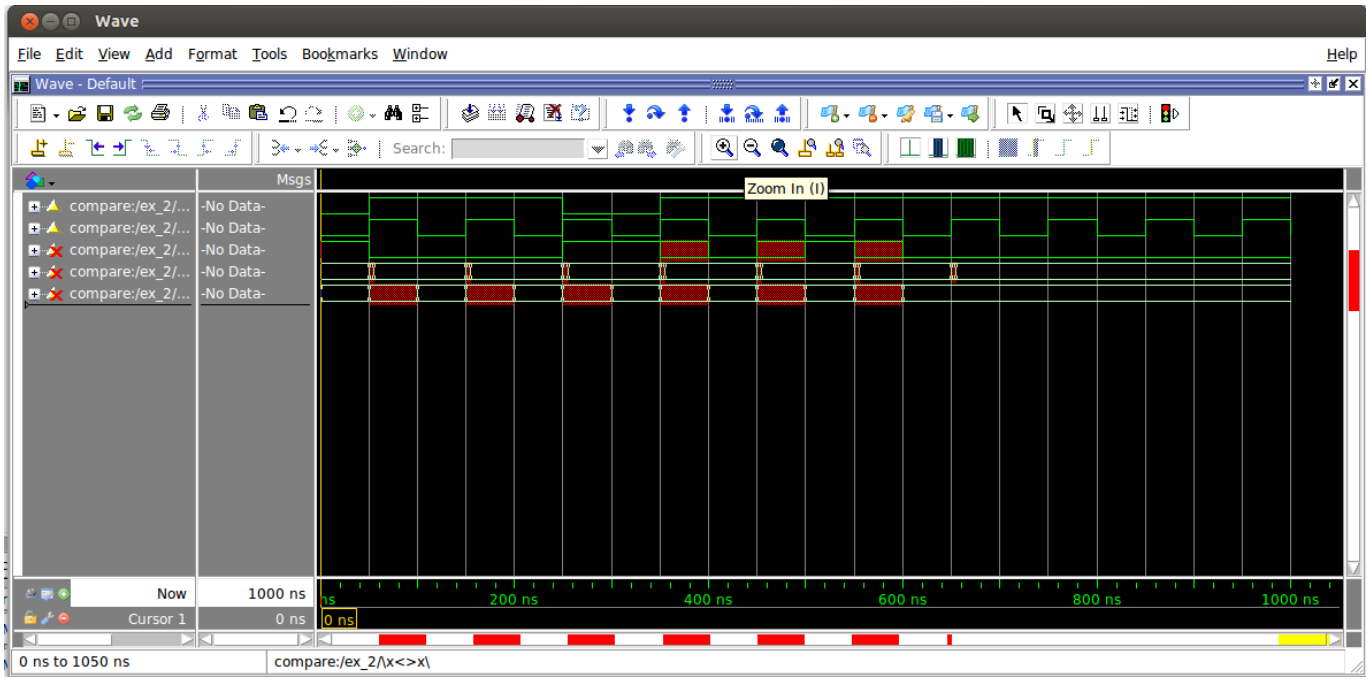
Versione c)

Questo è il risultato della versione c) simulato con il stimuli_1:



Differenza con ex_2

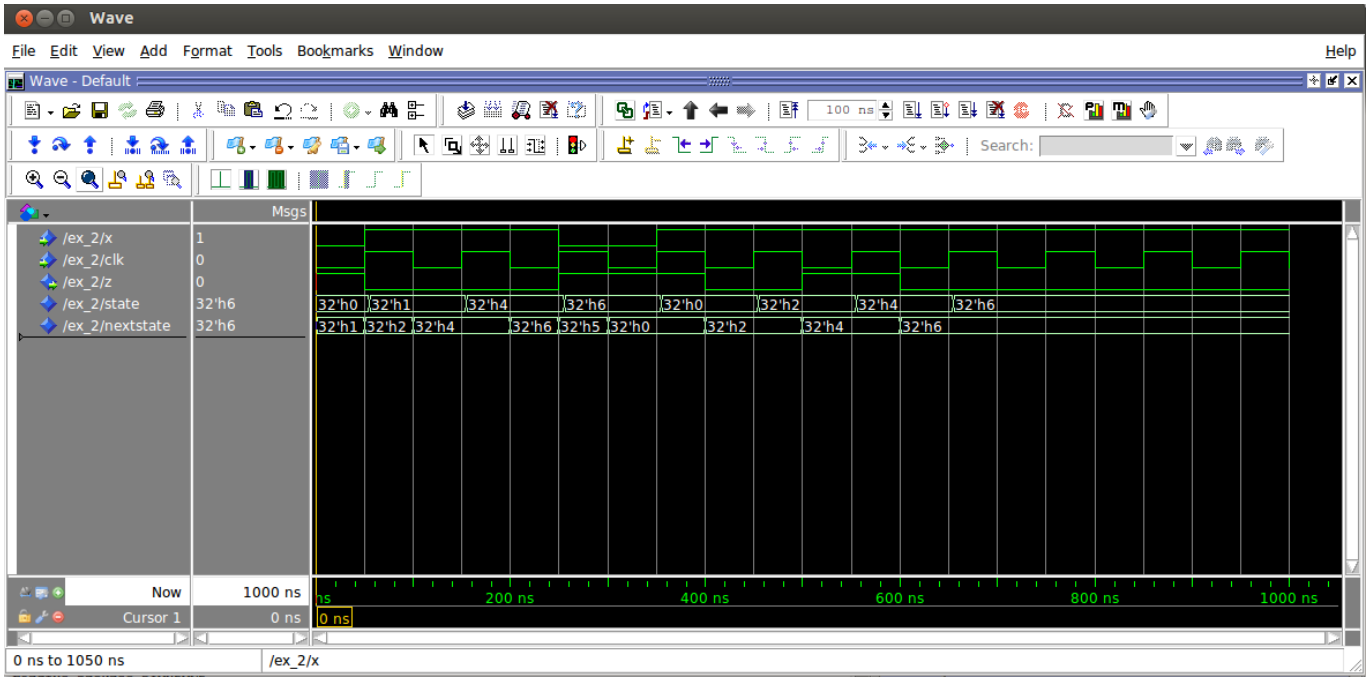
Questa è la differenza tra l'esercizio_2 simulato con lo stimuli_1 e la versione c) dell'esercizio_2 con lo stesso simuli



Come si può notare la variazione è sui segnali z, state e nextstate. A causa di un ritardo di aggiornamento del segnale state di 5 ns il segnale nextstate ha una variazione di 50 ns lo stesso vale per z.

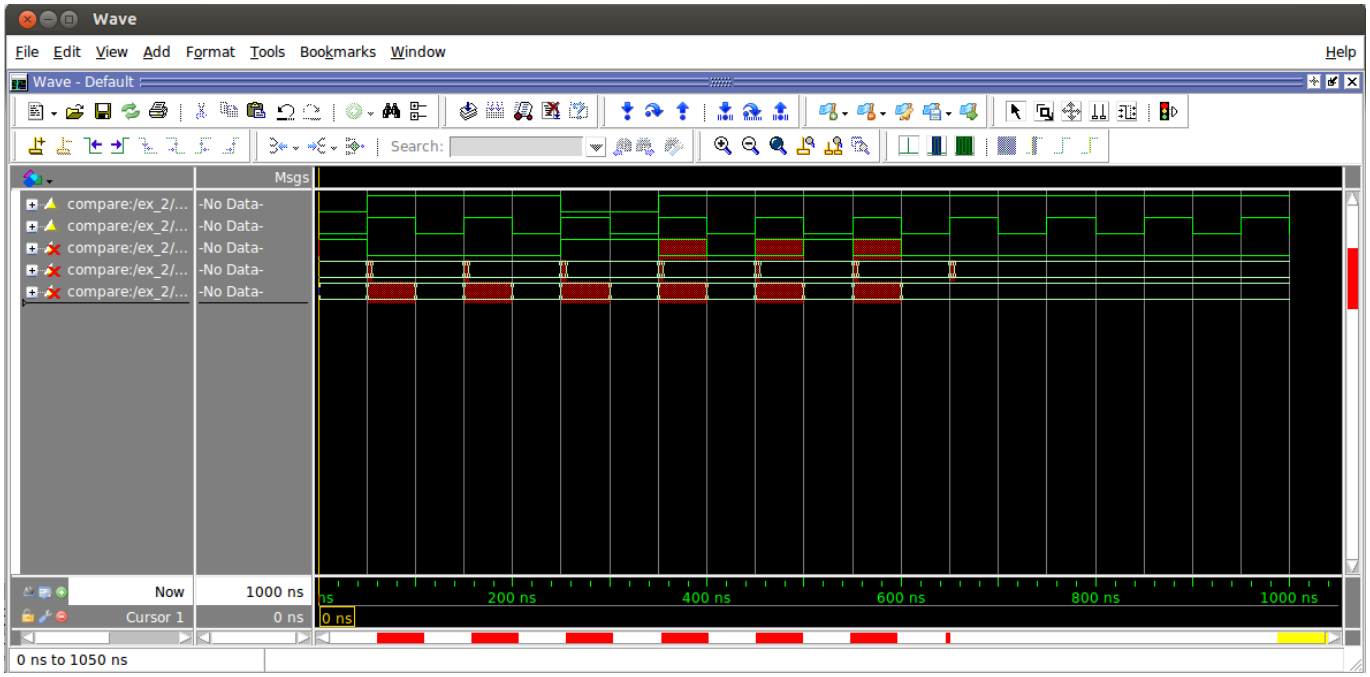
Versione d)

Questo è il risultato della versione d) simulato con il stimuli_1:



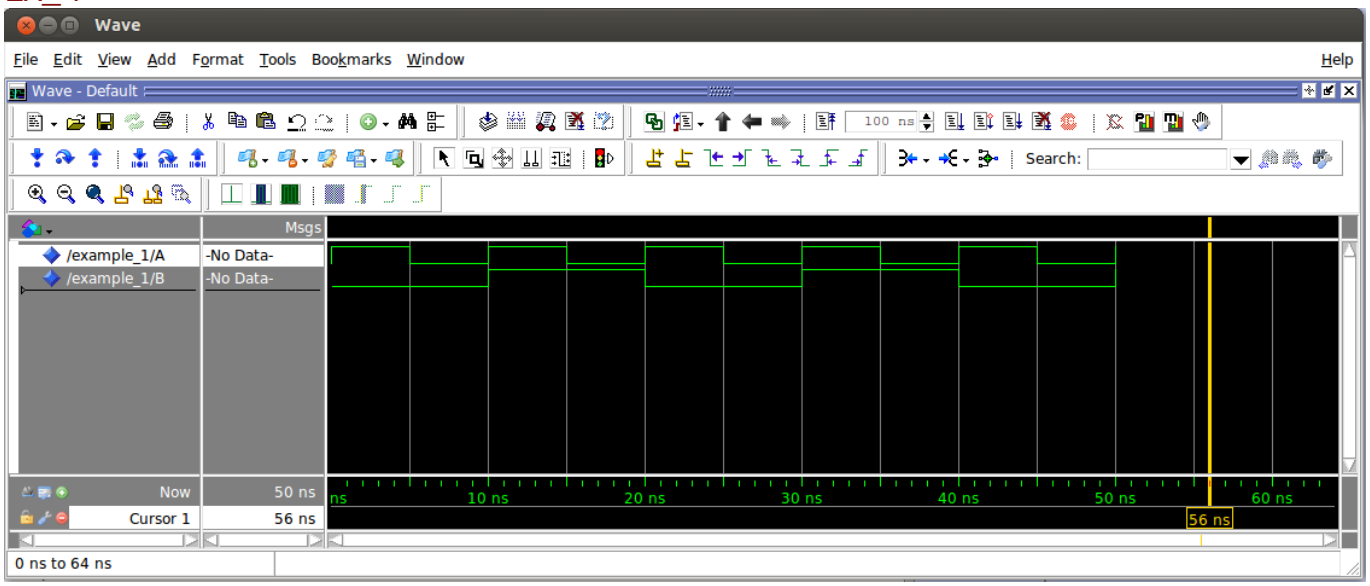
Differenza con ex_2

Questa è la differenza tra l'esercizio_2 simulato con lo stimuli_1 e la versione c) dell'esercizio_2 con lo stesso simuli.

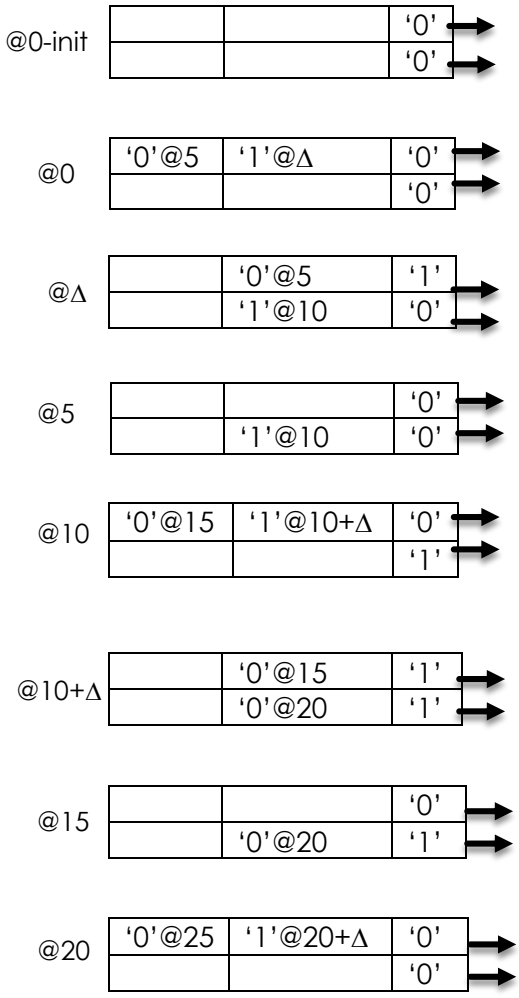


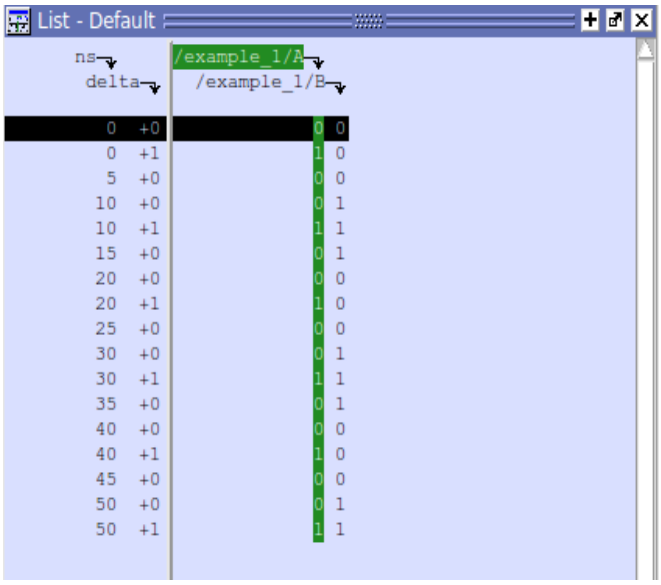
Come si può notare la variazione è sui segnali z, state e nextstate. A causa di un ritardo di aggiornamento del segnale state di 5 ns il segnale nextstate ha una variazione di 50 ns lo stesso vale per z. Ed è identica alla simulazione della versione d).

Ex_4



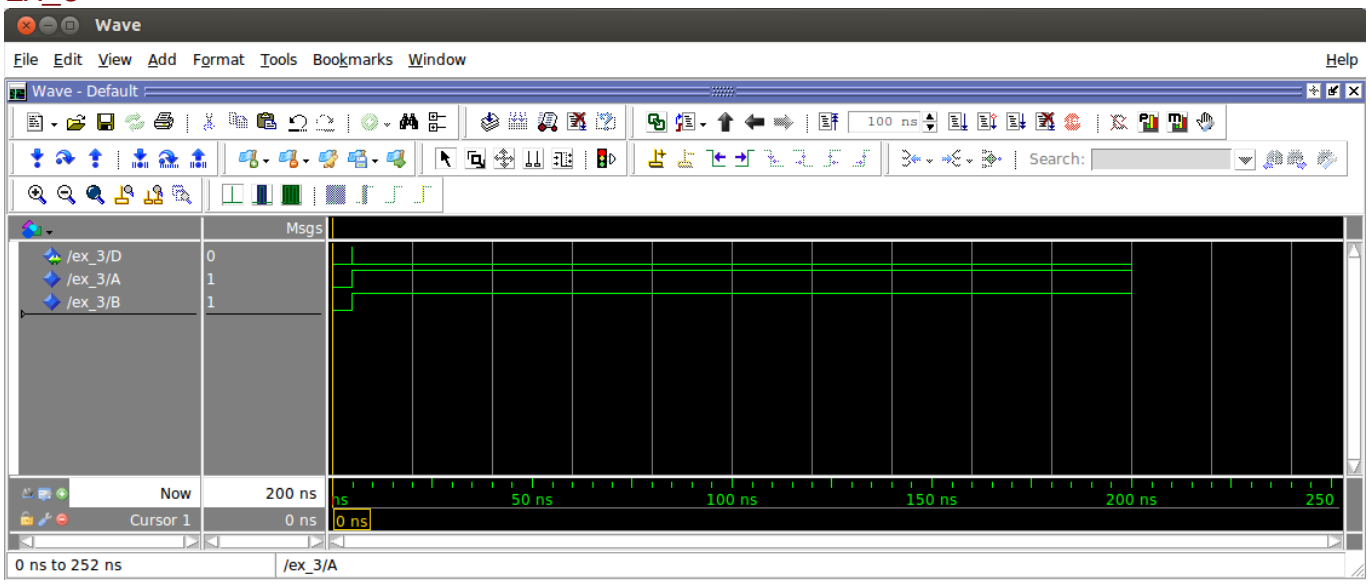
Signal drivers





Come si può notare il comportamento è ciclico.

Ex_5

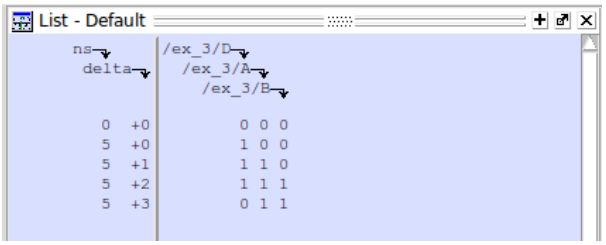


Signal drivers

Considero i segnali A, B,C,D,E,F in questo ordine.

@0-init

		‘0’
		‘0’
		‘0’
		‘0’
		‘0’
		‘0’



@0

	‘0’@0+Δ	‘0’
	‘0’@0+Δ	‘0’
	‘0’@0+Δ	‘0’
	‘1’@5	‘0’
	‘0’@0+Δ	‘0’
	‘0’@0+Δ	‘0’

@0+Δ

		‘0’
		‘0’
		‘0’
	‘1’@5	‘0’
		‘0’
		‘0’

@5

	‘1’@5+Δ	‘0’
		‘0’
		‘0’
		‘1’
		‘0’
		‘0’

@5+Δ

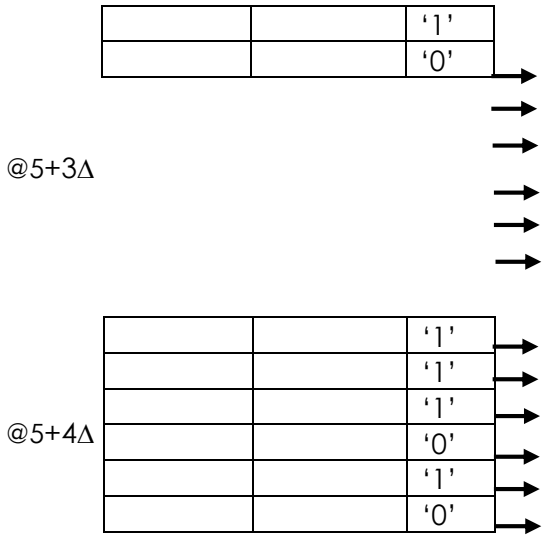
@5+Δ

		‘1’
	‘1’@5+2Δ	‘0’
	‘1’@0+2Δ	‘0’
		‘1’
		‘0’
		‘0’

@5+2Δ

	‘1’@5+3Δ	‘1’
		‘1’
		‘1’
	‘0’@5+3Δ	‘1’
	‘1’@5+3Δ	‘0’
	‘0’@5+3Δ	‘0’

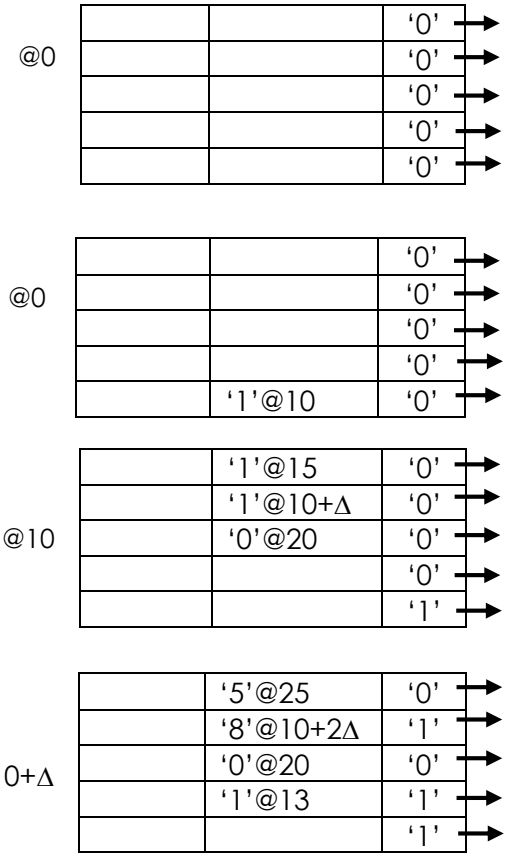
	‘1’@5+4Δ	‘1’
		‘1’
		‘1’
		‘0’



Il segnale da qui non cambia più valore.

Ex_6

Considero i segnali A,B,C,D,E in questo ordine.



Essendo un assegnamento inerziale cancella assegnamenti precedenti. Tolgo quindi '1'@15.

@10+2

	‘5’@25	‘0’
		‘8’
	‘0’@20	‘0’
	‘1’@13	‘1’
		‘1’

@13

	‘5’@25	‘0’
		‘8’
	‘0’@20	‘0’
		‘1’
		‘1’

@20

	‘5’@25	‘1’
		‘8’
		‘1’
		‘1’
		‘1’

@25

		‘5’
		‘8’
		‘1’
		‘1’
		‘1’