ELABORATI DI PROGETTAZIONE SISTEMI EMBEDDED

Enrico Giordano VR386687

Indice

Progetto 1: implementazione moltiplicatore in SystemC RTL

Descrizione del progetto

Il progetto consisteva nella realizzazione di un moltiplicatore in virgola mobile, doppia precisione, tra due numeri con virgola.

Il risultato finale è formato da due moduli SystemC: un moltiplicatore implementato come FSMD (controllore + datapath) e un testbench per stimolare e verificare il progetto; tutto questo è stato diviso in più file:

- main.cc : descrive la modalità di collegamento tra il modulo mul_RTL e il testbench;
- mul_RTL.cc: descrive il modulo che esegue la moltiplicazione tra due numeri e genera in uscita il risultato;
- mul_RTL_testbench.cc : descrive il modulo testbench.

Interfaccia

L'interfaccia è descritta nel file main.cc, che si occupa di collegare il modulo che esegue la moltiplicazione e il testbench per testarlo.

L'interfaccia si occupa di :

- creare le istanze del modulo del moltiplicatore e del modulo di testing;
- dichiarare le porte di ingresso e uscita del sistema e i segnali che permettono la comunicazione dei moduli precedentemente istanziati;
- eseguire il binding delle porte, ovvero di associare le porte ai rispettivi moduli garantendone la comunicazione.

I segnali istanziati nell'interfaccia sono i seguenti:

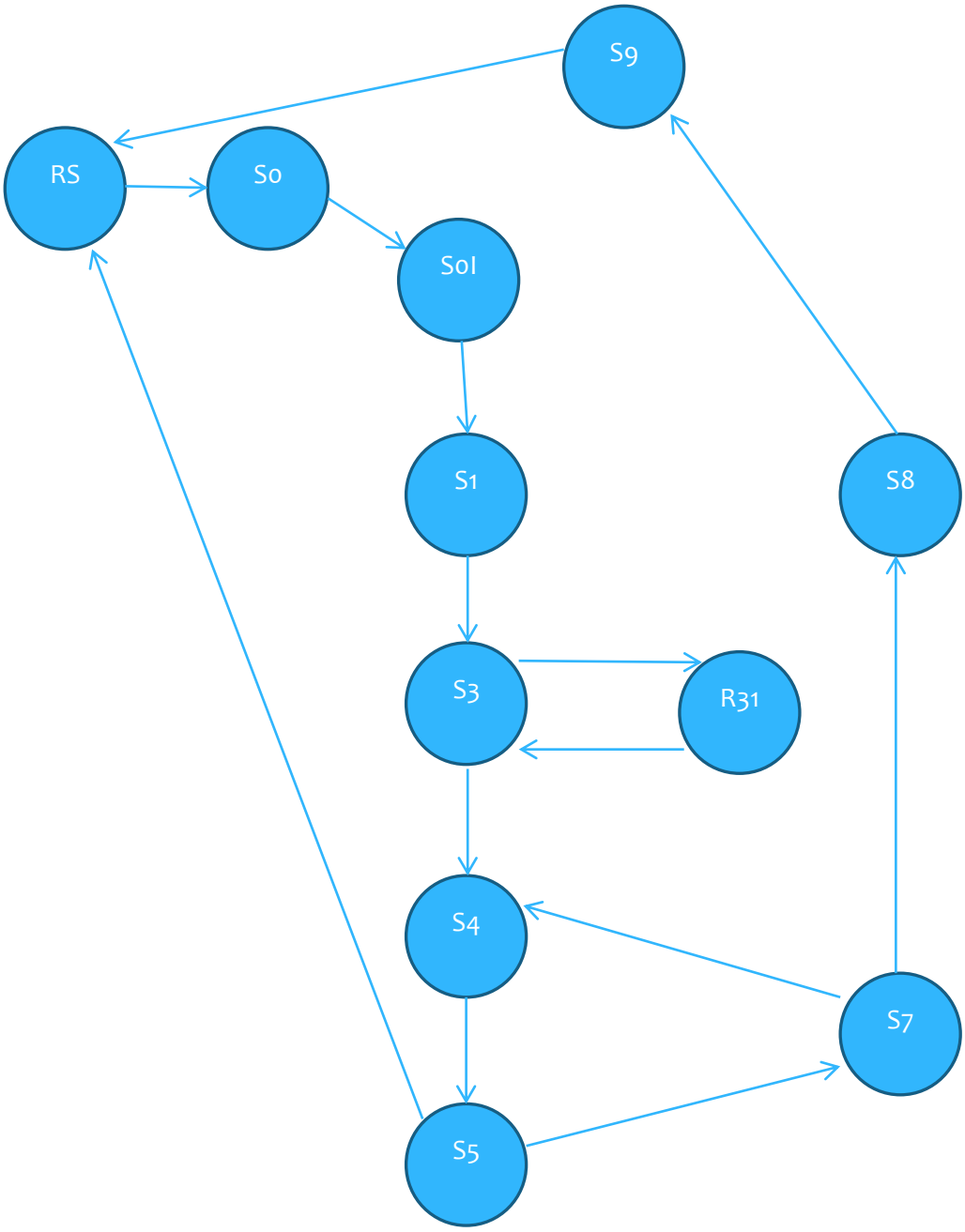
- clock: <sc_logic> rappresenta il segnale sincrono di clock che gestisce lo scorrere degli istanti di tempo del sistema;
- reset_signal: <bool> viene utilizzato per resettare il sistema.
- p_In_data1: <sc_lv<64>> rappresenta un numero da moltiplicare;
- p_In_data2: <sc_lv<64>> rappresenta un numero da moltiplicare;
- p_In_enable: <sc_uint<1>> indica se può iniziare la computazione;
- p_Out_enable: <sc_uint<1>> indica se è pronto un dato in uscita;
- p_result: <sc_lv<64>> rappresenta il risultato della moltiplicazione;

p_In_enable e p_Out_enable sono segnali che determinano rispettivamente l'inizio e la fine dell'esecuzione dell'algoritmo; di conseguenza vengono utilizzati come flag di collegamento e sincronizzazione tra il modulo di esecuzione e quello di test.

Per effettuare il collegamento, è stato necessario istanziare nel main i due moduli (mul_RTL_testbench e mul_RTL) e collegare ogni loro segnale, input e output con i segnali sopra descritti. Dopo averli collegati, per far partire la simulazione si esegue la funzione sc_start().

Descrizione EFSM

L'EFSM, descritta nel file mul_RTL.cc, è il modulo di elaborazione del sistema e rappresenta l'unione di datapath e macchina a stati finiti dell'algoritmo della moltiplicazione.



L'algoritmo che rappresenta questa macchina a stati è il seguente:

1. SR: reset di tutti i segnali e uscite;
2. S0: preparazione del sistema e inizializzazione delle porte di uscita e di ingresso (si scrive 0 in queste porte in modo da eliminare ogni segnale "sporco");
3. S0I: lettura dei numeri ed estrapolazione delle singole parti di questi (segno, esponente e mantissa in variabili interne per facilitare i calcoli);
4. S1: somma degli esponenti per ottenere l'esponente del risultato;
5. S2: polarizzazione dell'esponente risultante (togliere 1023 da esso);
6. S3: prodotto tra mantisse in interazione (parte di controllo), ovvero controllo se si devono fare altri passi per la moltiplicazione (in tutto si fanno 52 controlli perché si deve controllare ogni bit della mantissa del secondo numero);
7. S31: prodotto tra mantisse in interazione (parte di esecuzione), ovvero se la mantissa del secondo numero all'indice i-esimo vale '1', si somma alla mantissa risultante la mantissa del primo numero shiftandola di i volte;
8. S4: normalizzazione del risultato in interazione (parte di esecuzione), ovvero si controlla che non ci siano numeri dopo il 104-esimo bit, se ce ne sono si shifta la mantissa risultante a destra di 1 e si incrementa di 1 l'esponente risultante, altrimenti si shifta di 1 a sinistra la mantissa e si decrementa l'esponente;
9. S5: controllo di overflow, ovvero se l'esponente vale 0 o se la mantissa vale 0;
10. S7: normalizzazione del risultato in interazione (parte di controllo), ovvero si controlla se bisogna normalizzare ancora;
11. S8: calcolo del segno;
12. S9: preparazione del risultato da mandare in output.

Il file che contiene l'ESFSM contiene 2 metodi:

- void mul_RTL :: elaborate_mul_FSM(void), che contiene la parte datapath (di calcolo);
- void mul_RTL :: elaborate_mul(void), che contiene la parte di fsm (cambio di stato).

Descrizione dei processi

La procedura di esecuzione del sistema prevede l'istanziazione di due metodi descritti attraverso il comando "SC_METHOD", che sono rispettivamente i due processi sopra descritti. Questi processi sono dichiarati di tipo method poiché la loro esecuzione viene eseguita totalmente e può essere richiamata dal sistema più volte, senza mai essere interrotta.

I metodi sono sensibili ai cambiamenti di valore di una serie di segnali che compongono, per ogni metodo, una sensitivity list; al variare di valore di ogni segnale appartenente alla sensitivity list di un metodo, il metodo viene eseguito nuovamente. Un ulteriore procedura di istanziazione dei processi è quella che fa uso delle thread attraverso il comando "SC_THREAD". Il processo di testing è istanziato come thread, questo vuol dire che può essere lanciato una volta sola e interrotto attraverso il comando wait(), questo perché il testbench deve chiamare ed attendere il termine dell'esecuzione del modulo mul_RTL.

Tutti i processi sono sensibili al variare del clock.

mul_RTL ha la sensitivity list descritta in questo modo:

```
sensitive << STATUS << isready << number_a << number_b;
```

mentre elaborate_mul_FSM ha questa sensitivity list:

```
sensitive << reset.neg();

sensitive << clk.pos();
```

Esecuzione

Nel testbench è cablato l'input da eseguire, quindi basta eseguire il programma dopo averlo compilato per vedere il corretto funzionamento.

Progetto 3: implementazione moltiplicatore in SystemC TLM

Descrizione del progetto

Si vuole implementare l'algoritmo della moltiplicazione tra due numeri in virgola mobile doppia precisione a livello TLM, utilizzando le tre tipologie di implementazione del livello TLM: UT, LT e AT4.

In realtà, poichè a livello TLM è più importante osservare come comunicano i differenti moduli tra di loro, senza dare enfasi all'effettivo funzionamento algoritmico già descritto a livello RTL, è stata implementata la moltiplicazione come una normale moltiplicazione tra due numeri in C++, mentre si è implementato il protocollo di comunicazione tra moduli secondo lo standard TLM.

TLM

Il livello TLM prevede l'utilizzo di due tipologie di modulo per il funzionamento di un sistema.

Queste due tipologie sono **Initiator** e **Target**. Questi comunicano utilizzando specifiche primitive e sincronizzandosi tra di loro attraverso socket di comunicazione. Di seguito si vedranno le differenti modalità di comunicazione, ovvero Untimed (senza la nozione di tempo), Loosely Time e Aproximately Time.

Interfacce

In questo modello di progettazione, le interfacce sono praticamente uguali tra le differenti modalità di comunicazione, cambia invece la descrizione di Initiator e Target. Vengono creati due moduli, rispettivamente per il moltiplicatore e il suo testbench, e istanziate nel main chiamandole eloquentemente m_target e m_initiator. L'initiator inizializza il socket di comunicazione collegandosi con il socket del target; in questo modo si riprende in parte il design pattern dell'Observer, che permette la comunicazione tra classi chiamando, con il metodo di invio, il metodo di ricezione della classe che deve ricevere.

Modello di comunicazione

La modalità di comunicazione è molto simile tra moduli di diversi modelli descrittivi, infatti per il TLM in generale è possibile utilizzare dei modelli per progettare sia l'initiator sia il target. Ciò che cambia poi è la parte di esecuzione dell'algoritmo e il payload, ovvero la parte del messaggio che verrà inviato che contiene i dati calcolati dalla parte di esecuzione. Nella progettazione di questo elaborato quindi sono stati presi i template dei tre diversi tipi di comunicazione, cambiando quindi solo la parte di esecuzione (esecuzione della moltiplicazione dei dati ricevuti) e la parte di composizione del payload (invio del dato della moltiplicazione). Questo è uno dei punti importanti dell'utilizzo del TLM, in quanto permette di progettare velocemente il sistema, senza implementare molto codice in quanto si utilizza una parte templatica.

Untimed (UT)

Nella tipologia di TLM il modulo Initiator è implementato nel file "mul_UT_testbench.cc", mentre il modulo Target nel file "mul_UT.cc". Il modulo Target rappresenta nel nostro sistema, l'elaborazione vera e propria della moltiplicazione. Il modulo Initiator richiama l'elaborazione per eseguire l'algoritmo. L'Untimed utilizza la primitiva "b_transport()", differenziata in modalità Read e Write per la comunicazione tra i due moduli.

La b_transport è stata implementata nel modulo Target e viene richiamata dal modulo Initiator per effettuare una simulazione in modalità Write, successivamente il Target comincerà ad elaborare i segnali in ingresso per calcolarne il risultato della moltiplicazione. Poichè sono concorrenti, l'Initiator richiama ulteriormente la primitiva b_transport in modalità Read per poter leggere il risultato del modulo Target. Questo è visibile nel payload, (pacchetto di dati che viene inviato dalla b_transport) solo successivamente al "TLM_OK_RESPONSE" che rappresenta la fine dell'elaborazione del Target.

Loosely Time (LT)

Anche in questo caso l'elaborazione si divide in modulo Target e Initiator, i quali 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. Questo modello 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 del MUL 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.

Approximately Timed (AT4)

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

- nb_transport_fw: viene implementata nel Target e viene invocata dall'Initiator per richiedere l'esecuzione della moltiplicazione al Target o richiederne i risultati; sostituisce la b_transport dei modelli precedenti;
- nb_transport_bw: viene implementata nell'Initiator e viene invocata dal Target quando il Target ha terminato il calcolo della moltiplicazione e per notificare 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.

Comparazione di tempi tra TLM e RTL

Le tempistiche di esecuzione del sistema a livello RTL e TLM consentono di vedere come il sistema sia ottimizzato in termini di ritardo maggiormente a livello RTL rispetto TLM-UT. Si nota infatti, che l'esecuzione del sistema a livello RTL impiega circa `1240 ns`, mentre a livello TLM-UT è immediato e non ha il concetto di tempo, AT 1130 ns e LT 1130. Questo perché, il livello UT non è ottimizzato in termini di ritardo contenendo una wait che blocca l'Initiator durante tutta l'esecuzione del Target. Analogamente, lo stesso tempo lo riporta il TLM-LT. Diversamente il più vicino alle prestazioni di esecuzione dell' RTL è il TLM-AT4 che, sfruttando le sue quattro fasi, permette un'esecuzione concorrenziale "non completamente bloccante" dei due moduli Target e Initiator.

La scelta di utilizzare un modello TLM rispetto a RTL è utile per analizzare come i diversi moduli possono comunicare tra di loro, senza avere un sistema che rappresenta effettivamente quello che succede all'interno dei moduli. Infatti la moltiplicazione è stata rappresentata come una semplice moltiplicazione in C++, mentre si è data molta più enfasi alla comunicazione tra moduli.

Progetto2: tracciare l'evoluzione dei modelli RTL e TLM

Scheduler

Lo scheduler in SystemC è una parte del Kernel che ha il compito di:

- eseguire la simulazione;
- far avanzare il tempo;
- gestire i processi (thread);
- aggiornare i valori su porte e segnali e quindi i processi secondo la gerarchia dei segnali.

Lo Scheduling è diviso in fasi, in ognuna di esse si simula ciò che dovrebbe accadere durante l'evoluzione di una vera esecuzione su silicio. In particolare, la fase di Elaborazione ha queste fasi:

- INIT: vengono eseguiti tutti i processi fino alla loro terminazione o fino alla prima wait(), creando una coda di processi eseguibili, facendo il binding delle porte;
- EVALUATION: vengono eseguiti tutti i processi in coda, fino allo svuotamento di essa; tutti vengono eseguiti senza interruzioni e, in caso di notifiche immediate, i processi sensibili a questi eventi vengono aggiunti alla coda di esecuzione;
- UPDATE: vengono aggiornati i valori di porte e segnali, attivando (mettendo nella coda di esecuzione) i processi sensibili a questi;
- DELTA NOTIFICATION: vengono aggiunti alla coda di esecuzione tutti i processi sensibili alle delta notification;
- TIMED NOTIFICATION: se ci sono timed notification, il tempo di esecuzione avanza fino alla prima di queste e i processi sensibili a tali notifiche vengono aggiunti alla coda di esecuzione.

Finchè la coda non è vuota, si ritorna sempre alla fase di Evaluation.

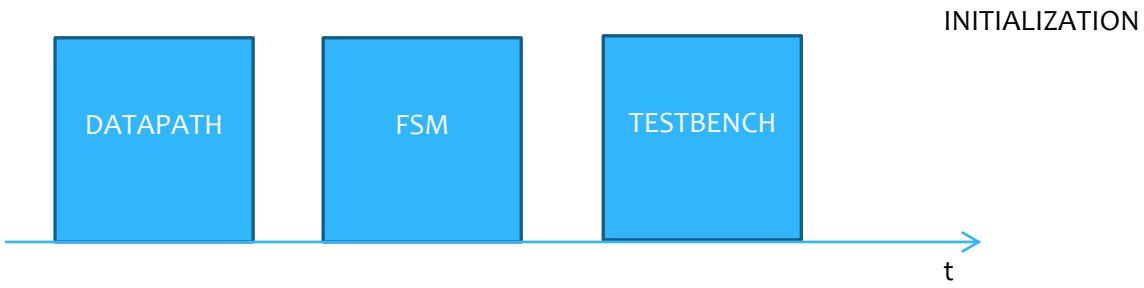
Flusso di esecuzione RTL

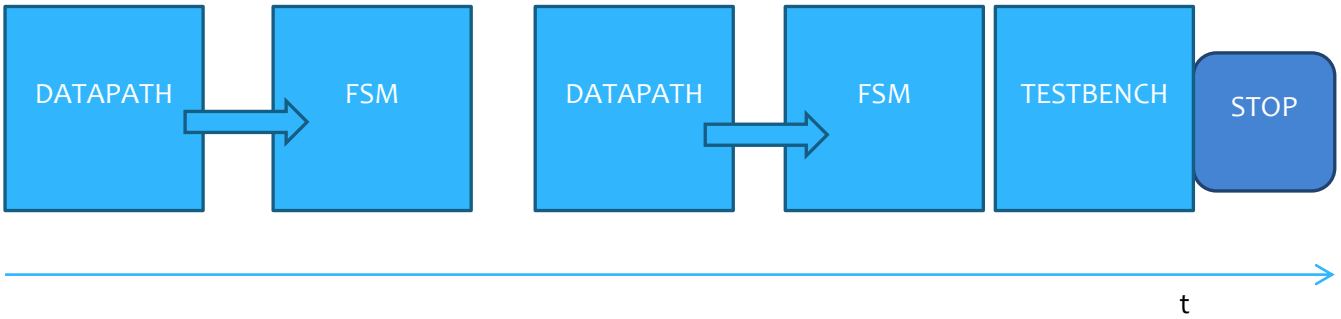
```
0 s - mul: elaborate_MUL

0 s - mul: elaborate_MUL_FSM

0 s - mul: SR

0 s - mul: elaborate_MUL

10 ns - mul: elaborate_MUL_FSM

10 ns - mul: S0

10 ns - mul: elaborate_MUL

20 ns - mul: elaborate_MUL_FSM

20 ns - mul: SOI

20 ns - mul: elaborate_MUL

30 ns - mul: elaborate_MUL_FSM

30 ns - mul: S1

30 ns - mul: elaborate_MUL

40 ns - mul: elaborate_MUL_FSM

40 ns - mul: S2

40 ns - mul: elaborate_MUL

…

1220 ns - mul: elaborate_MUL

1230 ns - mul: elaborate_MUL_FSM

1230 ns - mul: S9

1230 ns - mul: elaborate_MUL

1240 ns - mul: elaborate_MUL_FSM

1240 ns - mul: SR
```

Per poter tracciare l'evoluzione dei processi nel tempo, sono state inserite invocazioni alla funzione sc_time_stamp(), che stampa il corrente tempo di simulazione del sistema.

Il flusso d'esecuzione del sistema si divide in:

- INITIALIZATION: Inizializzazione dello Scheduler, in cui tutti i processi (SC METHOD e SC THREAD) vengo eseguiti una volta;
- Sequenza di esecuzione dei processi, dove:
    - il processo `elaborate_MUL_FSM` viene risvegliato dal processo `elaborate_MUL`, il quale aggiorna i segnali interni al modulo presenti nella sensitivity list del primo;
    - il processo `elaborate_MUL` viene risvegliato dal segnale di clk ad ogni ciclo di clock.





Al tempo 0 viene effettuata l'inizializzazione di tutti i processi.

Dal tempo 0 al tempo `1240` viene eseguito il design: vengono eseguiti nell'ordine il Datapath e la FSM.

Al tempo `1240` la FSM raggiunge lo stato finale S9, il DataPath scrive sulla porta result

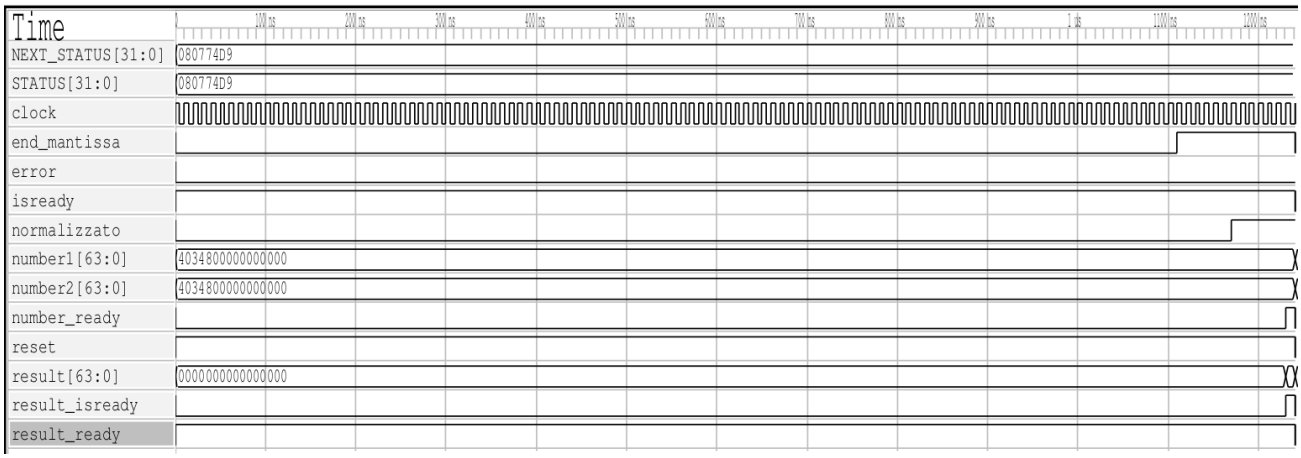
il risultato e abilita la porta di uscita: a questo punto il controllo ritorna al TestBench.

Evoluzione dei segnali

Creando una waveform in formato VCD è possibile tracciare i segnali. Per fare ciò, basta invocare il metodo sc_create_vcd_trace_file("wave") che crea un file con tali caratteristiche.

Inoltre, è possibile identificare i segnali con variabili assegnandogli i nomi corrispondenti:

```
sc_trace(fp, clock, "clock");

sc_trace(fp, reset_signal, "reset");

sc_trace(fp, p_In_enable, "number_ready");

sc_trace(fp, p_Out_enable, "result_ready");

sc_trace(fp, p_In_data1, "number1");

sc_trace(fp, p_In_data2, "number2");

sc_trace(fp, p_result, "result");
```



Flusso di esecuzione UT

Come si può notare, qui non esiste la nozione di tempo, avviene semplicemente l'invocazione della b_transport da parte del testbench e viene calcolata la moltiplicazione

```
[TB:] Calculating the multiplication of:  2.5 * 1.3

 [TB:] Invoking the b_transport primitive - write

              [MULTIPLIER:] Received invocation of the b_transport primitive - write

              [MULTIPLIER:] Invoking the multiplier_function to calculate the
moltiplication

              [MULTIPLIER:] Calculating multiplier_function ...

              [MULTIPLIER:] Returning result: 3.25
```

```
[TB:] TLM protocol correctly implemented

[TB:] Result is:  3.25
```

Flusso di esecuzione LT

Qui esiste la nozione di tempo, lo scambio di messaggi avviene allo stesso modo, ovvero avviene semplicemente l'invocazione della b_transport da parte del testbench e viene calcolata la moltiplicazione

```
0 s - top.initiator - run

0 s-[TB:] Calculating the multiplication of:

        2.4 * 1.3 = 0 s-[TB:] Invoking the b_transport primitive - write

              [MULTIPLIER:] Received invocation of the b_transport primitive - write

              [MULTIPLIER:] Invoking the multiplier_function to calculate the
moltiplication

              [MULTIPLIER:] Calculating multiplier_function ...

              [MULTIPLIER:] Returning result: 3.12

0 s-[TB:] TLM protocol correctly implemented

0 s-[TB:] Result is:

        = 3.12

Time: 0 s + 100 ns
```

Flusso di esecuzione AT

In questo caso, il processo è diviso in 4 fasi, cioè invio-ricezione nb_transport_fw e invio-ricezione nb_transport_bw. Questo modello è più realistico e quindi impiega più tempo ad essere eseguito.

```
0 s-[TB:] Calculating the  multiplication of

        3.7 * 6.6 = 0 s-[TB:] Invoking the nb_transport_fw primitive of multiplier -
write

              [MUL:] Received invocation of the nb_transport_fw primitive

              [MUL:] Activating the IOPROCESS

              [MUL:] End of the nb_transport_fw primitive

0 s-[TB:] Waiting for nb_transport_bw to be invoked

              [MUL:] IOPROCESS has been activated

              [MUL:] Invoking the multiplier_function to calculate the multiplication

              [MUL:] Calculating multiplication_function ...

[TB:] Invoking the nb_transport_bw primitive - write

100 ns-[TB:] Invoking the nb_transport_fw primitive of multiplier - read
```

```
                [MUL:] Received invocation of the nb_transport_fw primitive

                [MUL:] Activating the IOPROCESS

                [MUL:] End of the nb_transport_fw primitive

                [MUL:] IOPROCESS has been activated

                [MUL:] Returning result
[TB:] Invoking the nb_transport_bw primitive - write
200 ns-[TB:] TLM protocol correctly implemented
200 ns-[TB:] Result is: 24.42
```

Progetto4: SystemC AMS

Descrizione del progetto

Si vuole implementare un modello di sistema analogico composto da un controllore, che ha la funzionalità di pilotare e controllare in base a stimoli derivati dall'impianto in autoanello, e dal sistema fisico che tratta segnali analogici.

L'impianto fisico evolve in base agli stimoli secondo questa funzione di trasferimento:

$P(s) = 1 / (13s + s2)$

mentre il controllore segue questa equazione alle differenze:

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

Il Controller riceve due segnali: un segnale di riferimento $r(t)$, ottenuto in questo caso da un file in cui per ogni riga c'è un valore, e un segnale di "errore" $e(t)$ che deriva dal calcolo dell'impianto. Il controller, in base al calcolo dell'errore e del segnale, esegue l'equazione alle differenze. Il calcolo dell'errore deve derivare da questa formula:

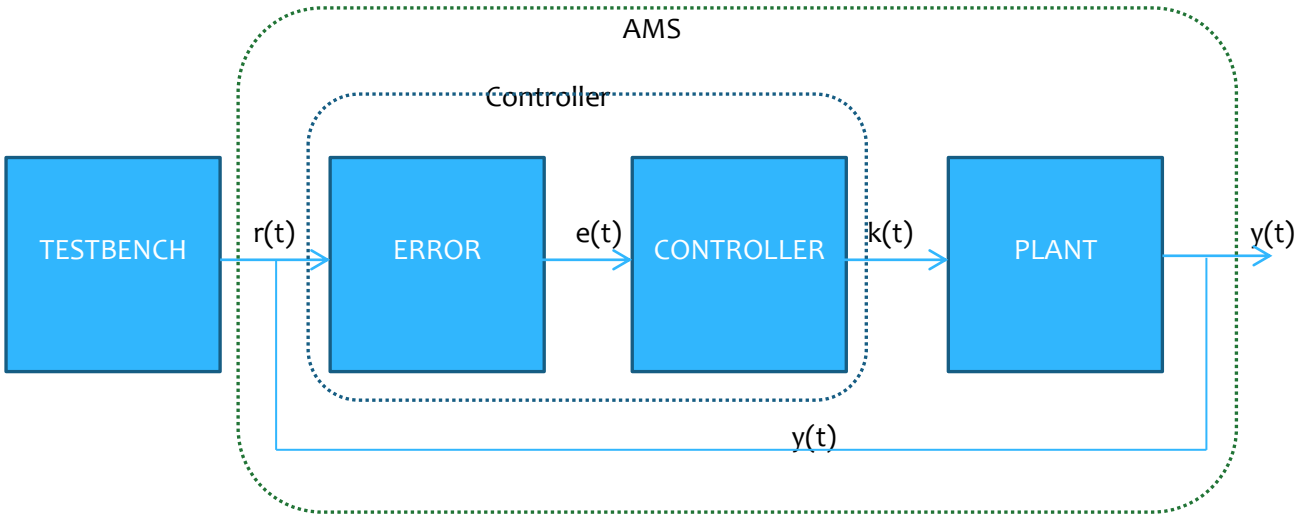
$e(t) = r(t) - y(t)$

Gli step di tempo devono essere di 20 ms, quindi ogni nuovo riferimento $r(t)$ viene preso dal file ogni 20 ms, quindi il controllore deve essere abbastanza veloce per permettere all'impianto di campionare regolarmente ogni valore.

Per implementare il tutto, è stato deciso di utilizzare come MoC il TDF, in quanto l'impianto è a tempo discreto e il controllore esegue un'equazione alle differenze. Il controllore è stato diviso in due moduli: il primo esegue la computazione dell'errore e il secondo computa $k(t)$.

Sarebbe stato possibile usare anche come MoC il TDF per il Controllore e il LSF per l'Impianto, un quanto l'impianto poteva essere interpetato come sistema completamente analogico; ci sarebbe stata facilità di collegamento tra i due moduli in quanto nel LSF non è necessario descrivere la velocità di campionamento, ma per semplicità (e calandosi in un contesto in cui il Time To Market è importante) è stato deciso di implementare entrambi in TDF, in modo da avere un template comune semplice da implementare.

Implementazione



Controllore

Il Controller è diviso in due moduli:

- un modulo **controller_err**, che calcola l'errore derivato dalla differenza di segnale di riferimento e del segnale di calcolo dell'impianto;
- un modulo **controller**, che calcola l'equazione alle differenze precedentemente descritta.

Visto che è stato deciso di utilizzare il MoC TDF, è stato necessario implementare le 3 funzioni:

- *set_attributes()*, in cui si imposta il timestep (20 ms) e il delay della porta di uscita;
- *initialize()*, in cui si inizializza il sistema (quindi le variabili interne);
- *processing()*, in cui si esegue l'algoritmo (calcolo dell'equazione alle differenze per controller e)

La dichiarazione e il costruttore dei moduli del controllore avvengono nei file controller.h e controller_err.h e differiscono da quelli di SystemC non AMS, in quanto bisogna lavorare con oggetti e variabili di SystemC AMS (contraddistinti dalla 'a' di "analog" in parte a "sc"), opportunamente scelti in base al MoC di riferimento (in questo caso sono "sca_tdf"). Il costruttore del modulo è chiamato SCA_CTOR.

Impianto

Questo viene descritto dal modulo **p_plant**, nei file p_plant.h e p_plant.cc. Si è deciso di utilizzare un MoC di tipo TDF per i motivi descritti sopra.

La differenza sostanziale tra questo modulo e quello del controllore è l'utilizzo della funzione di trasferimento per calcolare l'output: prima di calcolarla è necessario impostare i valori del numeratore e del denominatore, in particolare vanno impostati i coefficienti delle variabili della formula fino al grado massimo descritto (partendo da 0 per il coefficiente senza variabile) sia del numeratore che del denominatore; successivamente nel metodo di calcolo effettivo si esegue la funzione di trasferimento

tramite il metodo *ltf_nd*, che riceve in input il numeratore precedentemente impostato, il denominatore, ciò che è stato letto in quell'istante dall'input e il DC gain.

Testbench

Questo è l'unico modulo del sistema che non è stato descritto in SystemC AMS, in quanto deve solamente leggere i valori da file per generare il valore di riferimento del controllore. La modalità di collegamento tra SystemC e SystemC AMS verrà discussa nella sezione "Interfaccia". Come per gli altri moduli in SystemC, è stato implementato un modulo operativo che si occupa della lettura del file e l'invio dei dati e un generatore di clock che permetta di dare all'intero sistema un clock a cui essere sensibili.

Interfaccia

Nel file main.cc viene descritta la modalità di collegamento tra tutti i moduli. I moduli TDF sono stati collegati semplicemente usando segnali del SystemC AMS, mentre il segnale *r_in*, proveniente da un modulo SystemC, è stato dichiarato in ambiente SystemC.

Per permettere la comunicazione tra un modulo SystemC e SystemC AMS, è stato deciso di integrare la parte "digitale" in ambiente "analogico" e non viceversa, in quanto si tratta di una semplice lettura di un modulo analogico di un valore digitale. Quindi il testbench invia il valore utilizzando un normale output, mentre il controller_err riceve il valore tramite l'input dichiarato in questo modo:

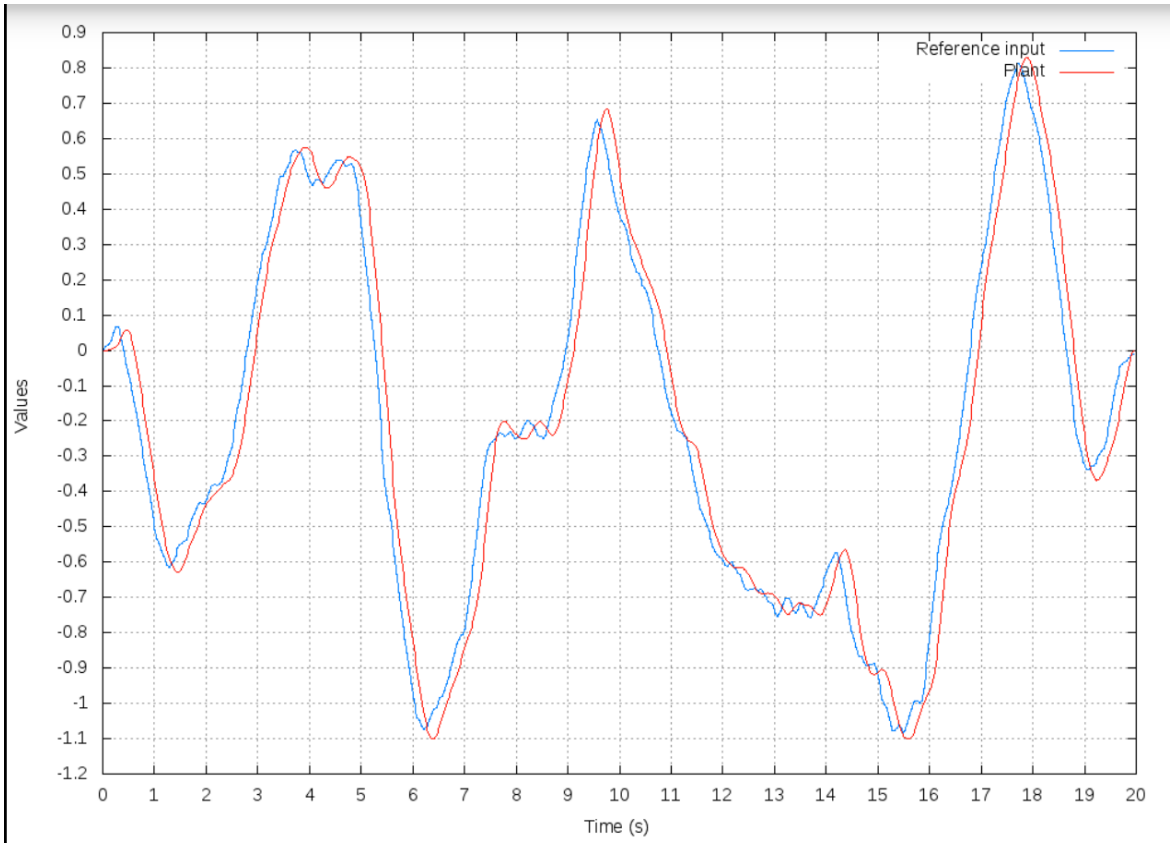
```
sca_tdf::sca_de::sca_in<double> r_input;
```

In questo modo, il file di interfaccia non deve fare altro che collegare in maniera semplice i due tipi di moduli, non corrompendo la normale esecuzione dello scheduler di SystemC. Facendo il contrario invece (cioè facendo un "cast" all'interno del controller_err in modo da convertire un segnale analogico in ambiente digitale) non avrebbe dato errore di compilazione, ma lo scheduler non sarebbe stato in grado di eseguire tutti i binding di segnali e quindi avrebbe dato errori in esecuzione.

Come da prassi, in questo file è stato istanziato ogni modulo e creato un segnale per ogni coppia di input/output per collegare i vari moduli.

Tracing

Per tracciare il corretto funzionamento dell'impianto risultante, è stato posto dentro il main un tracing dei regnali r_in e y_out, in modo da tracciare sia il segnale ricevuto dal testbench sia il segnale prodotto dall'impianto.



Il grafico è stato generato tramite GnuPlot a partire dal file generato dal tracing (usando uno script per velocizzare il disegno). Come si può notare, il segnale "Reference input" è più grezzo del segnale "Plant", che risulta più smussato perché è intervenuto l'impianto.

Progetto 5: Transattori

I Transattori sono dei modelli in SystemC che permettono di far comunicare e collegare moduli di diversa natura e descrizione. In questo progetto sono stati implementati due tipi di transattori: uno tra SystemC RTL e SystemC TLM e uno tra SystemC TLM e SystemC AMS. Da notare che non è stato implementato un transattore tra SystemC AMS e SystemC RTL in quanto non serve, esistono già delle primitive in SystemC AMS per comunicare con un modulo RTL, discusse nel precedente progetto di SystemC AMS.

Un Transattore è una sorta di "traduttore" di segnale: si deve mettere in comunicazione con i due moduli di diversa natura, riceve un segnale e lo modifica in modo da essere comprensibile al modulo che deve riceverlo. Come si vedrà di seguito, per la parte TLM/RTL comunicherà secondo lo standard TLM con il modulo TLM, mentre comunicherà con RTL secondo lo scambio di segnali tipico di RTL; invece per TLM/AMS, mentre per la parte TLM sarà uguale alla descrizione precedente, con il modulo AMS comunicherà con lo standard AMS.

Da notare che, una volta creato il transattore, sarà "portabile" tra diversi moduli con la stessa descrizione di porte: se si dovessero modificare i moduli che deve far comunicare, a meno di cambiamenti sostanziali di configurazione di input/output, sarà ancora valido per farli comunicare. Questo permette una buona scalabilità dell'architettura, quindi ha un ottimo impatto esplorativo del sistema come deve essere per SystemC.

TLM – RTL

I moduli TLM e RTL da cui partire sono gli stessi dei precedenti progetti; ciò che cambia è il file di interfaccia tra i moduli, che dovrà connettere TLM, Transattore e RTL, e il nuovo modulo del Transattore.

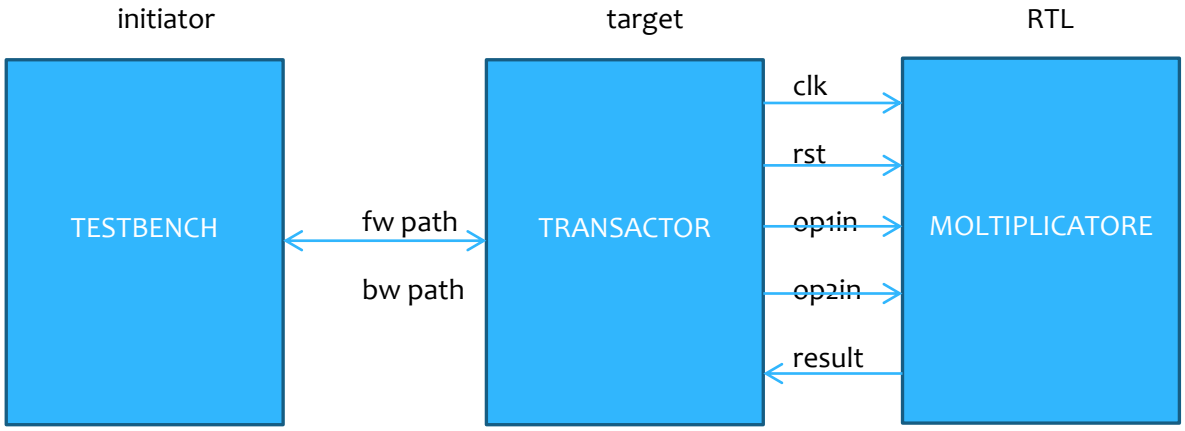
Il modulo Transattore ha una parte tipica della descrizione TLM, ovvero tutta la struttura del messaggio TLM (socket e messaggio con payload), un'interfaccia TLM per usare i metodi tipici del TLM, come la b_transport, dichiarata come *virtual*, e una parte descritta in RTL, quindi con le porte di ingresso, di uscita e i segnali. I processi utilizzati sono:

- *WRITEPROCESS()*, che si occupa di scrivere in ambiente RTL le informazioni passate dal modulo TLM verso il modulo RTL;
- *READPROCESS()*, che si occupa di leggere il risultato calcolato dal modulo RTL e inoltrarlo al modulo TLM tramite ioDataStruct, notificando la fine dell'eborazione tramite il metodo *notify()*.

Per la comunicazione effettiva con il modulo TLM, è stato utilizzata la b_transport, in modo da rimanere attinenti allo standard TLM. Entrambi i processi sono sensibili al clock.

Il file di interfaccia, il main_root_RTL.cc, è stato modificato in modo da collegare tutte le parti del sistema come è da prassi, ricordandosi che il collegamento deve essere fatto tra TLM - Transattore e Transattore – RTL.

Graficamente può essere rappresentato in questo modo:



TLM - AMS

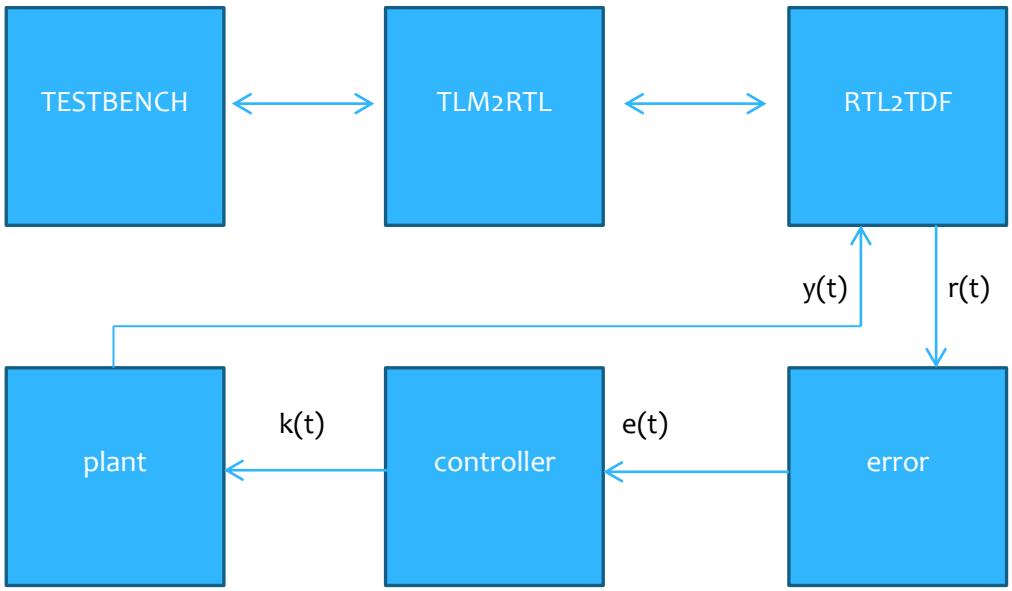
In questo caso, si doveva far comunicare un modulo TLM con un modulo AMS, quindi il Transattore doveva utilizzare la parte tipica del TLM (messaggi e socket) e la parte AMS, più vicina a RTL. I processi READPROCESS() e WRITEPROCESS() quindi sono molto simili a quelli della descrizione precedente, considerando AMS come oggetto simile ad una descrizione RTL; come è stato detto nel progetto di SystemC AMS, è necessario convertire un segnale digitale in ambito analogico e non viceversa. Questo passaggio è stato fatto nel file *rtl_2_tdf.hh*, nella variabile r_rtl:

```
sca_tdf::sca_de::sca_in<double>                r_rtl;
```

Si è deciso di dividere in più file questa descrizione rispetto a quella precedente in quanto è molto più chiara la divisione in parti e il procedimento di "conversione" dei valori passati, in modo che sia tutto più scalabile; infatti la parte TLM è nei file *testbench_TLM*, la parte di conversione di segnali è nei file *rtl_2_tdf* e *tlm_2_rtl*, mentre la parte AMS è nei file *error*, *controller* e *plant*.

Anche in questo caso, il file di configurazione permette di interfacciare tramite segnali tutti i moduli, collegando TLM – Transattore – AMS.

Graficamente può essere rappresentato in questo modo:



Progetto 6: Asserzioni

Le asserzioni hanno lo scopo di testare un'architettura descritta in SystemC in modo da evidenziarne le qualità e i difetti. In particolare, un'asserzione, come in un normale programma in SystemC, possono bloccare l'esecuzione qualora ci fosse qualche condizione importante violata (controllata appunto dalle asserzioni).

Si è deciso di definire tre proprietà, come da specifiche, per il moltiplicatore e non per AMS, poiché non è testabile con le asserzioni.

Le proprietà da verificare sono state dichiarate come funzioni che poi verranno eseguite sotto forma di thread per non intaccare il normale funzionamento del sistema. Poichè però vengono schedulate insieme al resto della simulazione, si è deciso di non interrompere la simulazione per la proprietà che risulta sempre falsa. Questo è stato deciso soprattutto perché altrimenti non si sarebbero viste le altre asserzioni (lo scheduler può mandare in esecuzione prima l'asserzione sempre falsa e, interrompendo la simulazione, non avrebbe reso visibili le altre sempre vere).

Le asserzioni hanno una loro sensitivity list, in modo da essere risvegliate al momento opportuno e per fare quindi i relativi test.

property1

Questa controlla che non passano più di 500 cicli di clock dalla ricezione dei valori in input e il calcolo dei valori. Questo è sempre vero perché la fsm e datapath vengono attivati ad ogni ciclo di clock, considerando che ci sono 11 stati, il sistema impiegherà almeno 22 cicli di clock se dovesse percorrerli tutti almeno una volta e al massimo circa 100 cicli se si dovesse normalizzare il massimo numero normalizzabile, ma mai 500 cicli.

property2

Questa controlla che nella situazione iniziale, quando i numeri non sono pronti per essere calcolati, quindi con il bit di conferma inizio calcolo posto a 0, il risultato e il bit di conferma di risultato calcolato non valgono 1 (valgono X...X e 0). Questo è evidentemente sempre vero perché quando non è pronto l'input da calcolare, di sicuro non è pronto l'output di risultato.

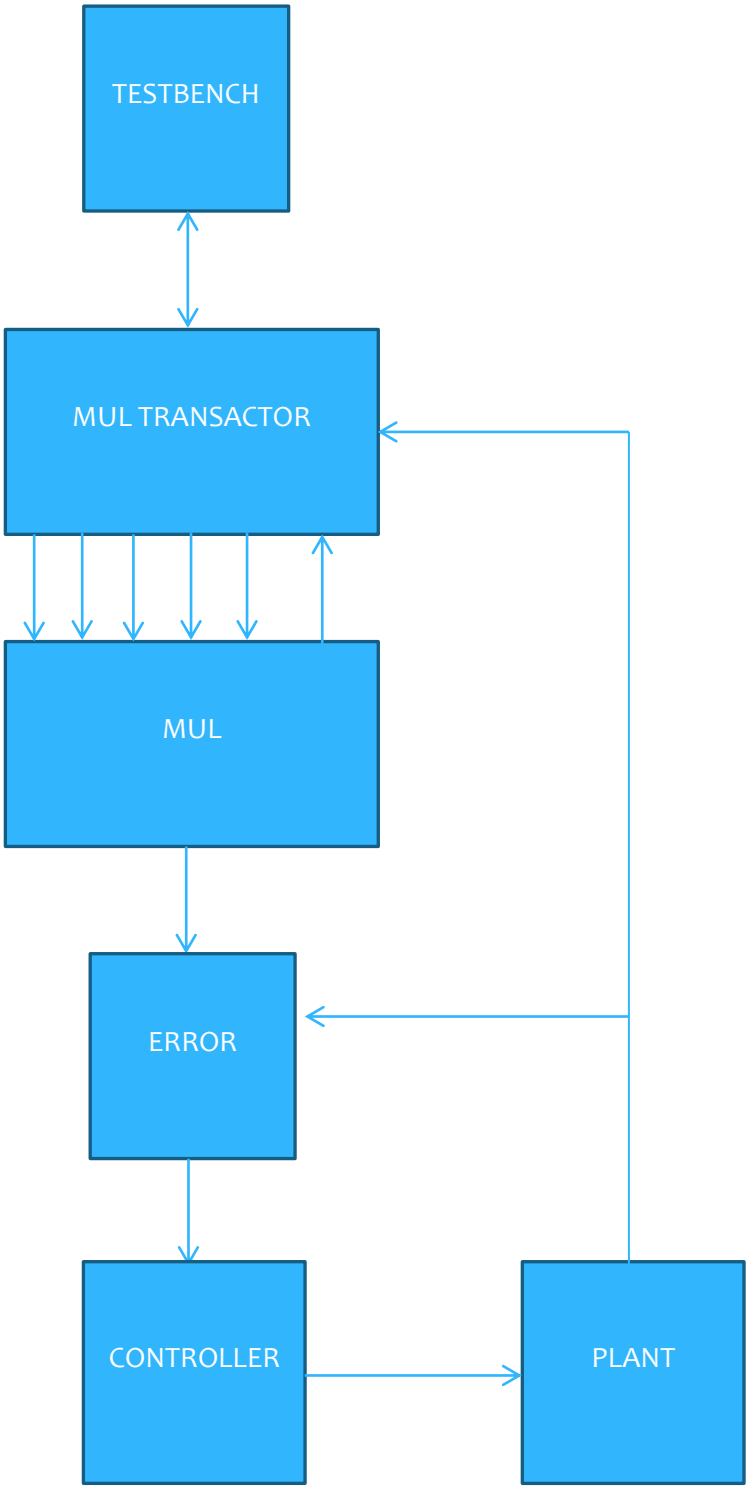
property3

Questa che, se la mantissa è stata normalizzata, il bit di verifica che notifica alla FSM se la mantissa è stata vale 0. Questo è un paradosso perché se la mantissa è stata normalizzata, il sistema imposta quel bit a 1, altrimenti la FSM non saprebbe quando cambiare stato. La condizione pertanto risulta sempre falsa. Da questa è stata tolta però la funzione sc_stop() altrimenti la property1 non darebbe mai il risultato: questo perché la sensitivity list di property1 contiene il clock e deve effettivamente contare i cicli di clock che passano; quindi viene schedulata per più tempo di questa e, quando questa sarebbe fallita, non si sarebbe potuto vedere il risultato di property1. Un altro motivo per cui non si sarebbe visto property1 è il fatto che deve aspettare che tutta la computazione sia completata, mentre property3 testa una condizione a metà computazione.

Progetto 7: Platform

Questo progetto consisteva nell'unire tutti i "pezzi" hardware generati con i progetti precedenti, quindi riutilizzandoli per ottenere il prodotto finito.

L'unione di tutti i pezzi si può riassumere con il seguente disegno:



Flusso di esecuzione

Il Testbench, scritto in TLM, è l'Initiator del sistema.

Questo, invocando la primitiva b transport in modalita write, invia i dati tramite il payload che li invierà al multiplier RTL transactor. Al momento dell'invocazione della b transport in modalita read, il Testbench riceve i dati relativi all'output del moltiplicatore e l'output dell'impianto.

TLM/RTL

Questo componente permette la comunicazione tra il Testbench TLM e il moltiplicatore RTL. Il transattore mantiene la sincronizzazione tra il livello RTL e il livello AMS e genera il segnale di clock per il Multiplier.

L'impianto genera valori ogni 20ms e il tempo di campionamento deve essere doppio rispetto quello di esecuzione. Per questo motivo il Multiplier deve generare i references per l'impianto ogni 40ms.

I dati del transattore vengono passati a questo modulo che si occupa della moltiplicazione dei due valori passati. Dopo questa operazione, il risultato verra scritto sulla porta result alla quale sono connessi il modulo sottostante

Il doppio collegamento Plant -> Error e Plant -> Mul è stato possibile grazie al fatto che nel mondo analogico si riesce a generare un segnale digitale (quindi da SystemCAMS a SystemC) ma non il viceversa; infatti viene generato il calcolo dell'impianto e inviato direttamente al sistema analogico per elaborare la stima dell'errore e al transattore del moltiplicatore, proprio per il fatto che non si necessita di ulteriore conversione di tipi di segnali.

Progetto 8: VHDL

Nella progettazione finale di un sistema embedded, utilizzando lo standard di progettazione visto nel corso di Progettazione Sistemi Embedded, il passo finale è la "traduzione" del modello di simulazione SystemC RTL in un effettivo sistema descritto in VHDL. Storicamente il VHDL è nato per la simulazione più che per la sintesi, ma propone un ottimo modello per tradurre il sistema simulato su PC in un effettivo circuito. In questo progetto si vedrà come si è sviluppato questo processo.

Traduzione

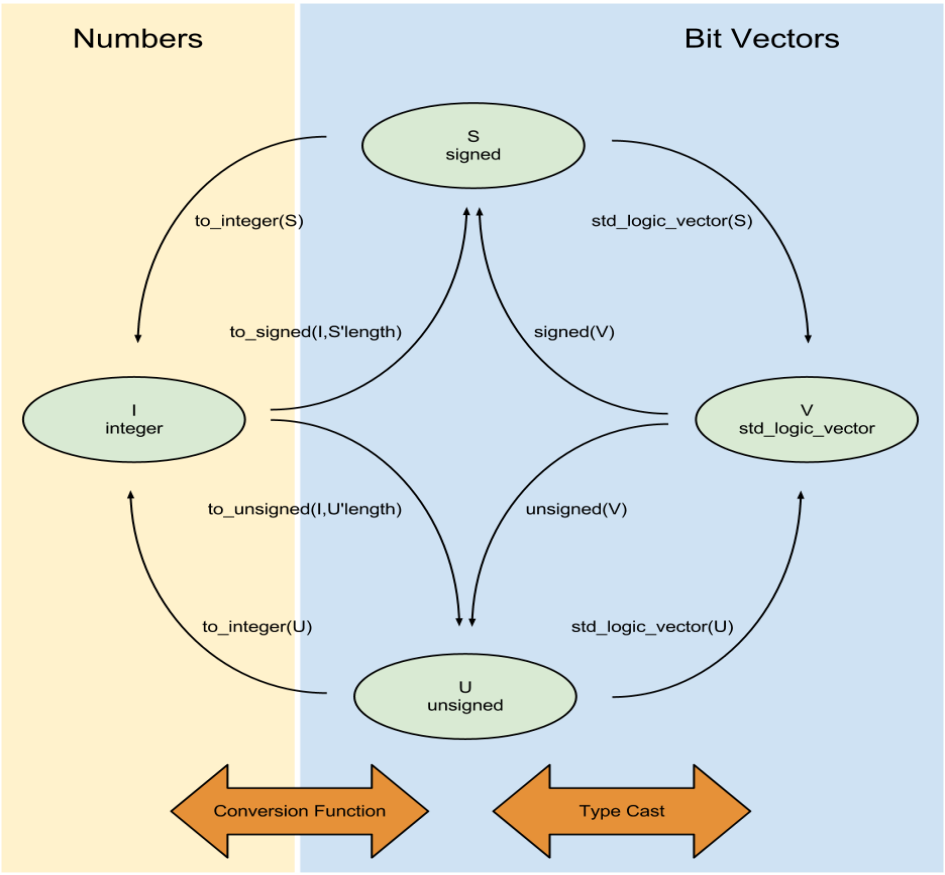
Si dice che, qualora il modello descritto in SystemC sia stato progettato bene, la descrizione del medesimo modello in VHDL dovrebbe essere una "mera" traduzione di linguaggio. In realtà, poichè la simulazione in VHDL rappresenta effettivamente cosa succede in un circuito, con i segnali elettrici che variano, oltre ad una semplice traduzione si esegue anche una procedura di riadattamento del proprio modello.

Particolare attenzione bisogna prestare al linguaggio VHDL, che si presenta fortemente tipato e quindi può prestare problemi rispetto al C++ che esegue spesso in automatico la conversione di tipi tramite polimorfismo delle funzioni stesse; il codice quindi risulta modificato da continui "cast" tra tipi, qualora fossero stati implementati.

Infine, problema principale di tutta la progettazione, bisogna prestare attenzione al fine ultimo di questa progettazione: visto che bisogna creare un effettivo sistema, quindi un circuito, tutta la descrizione deve essere sintetizzabile: non deve quindi contenere variabili di tipo non sintetizzabile (tipo integer), non deve contenere cicli while e deve rispettare i quattro modelli di sintesi. Sapendo queste cose a priori, basta stare attenti durante la descrizione in SystemC a non riprodurre queste situazioni non sintetizzabili, altrimenti bisogna effettivamente modificare il codice e non si ha più una traduzione ma una riprogettazione di tutto il sistema.

Le variabili utilizzate quindi sono state prettamente std_logic_vector, bit e unsigned, poichè nella tabella di cast descritta nella documentazione del VHDL sono i tipi sintetizzabili e utilizzabili per qualunque cast (seguendo un ordine di cast specifico).

L'unico tipo diverso che è stato utilizzato è uno definito dall'utente, quindi enumerativo, FSM_ST, per rappresentare gli stati della FSM.



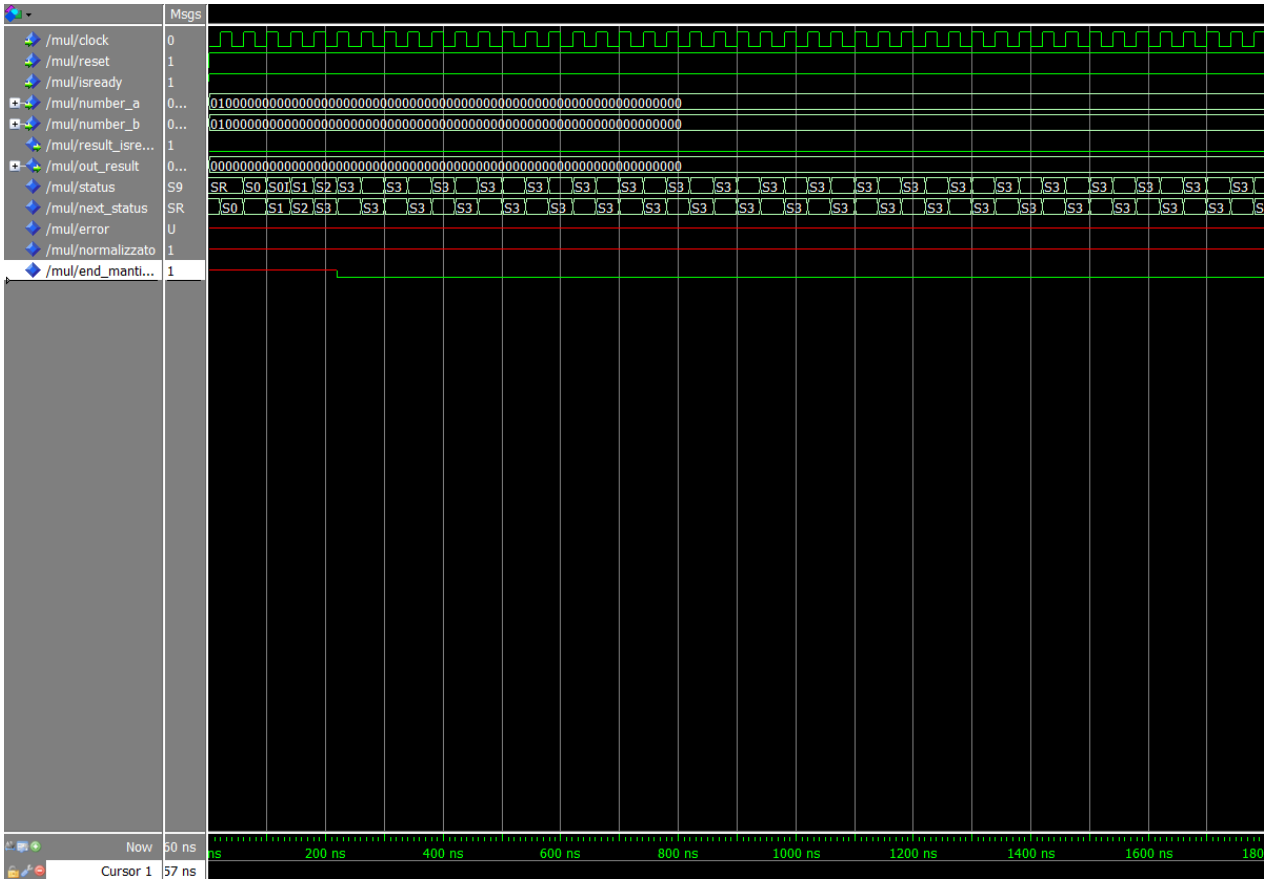
Struttura

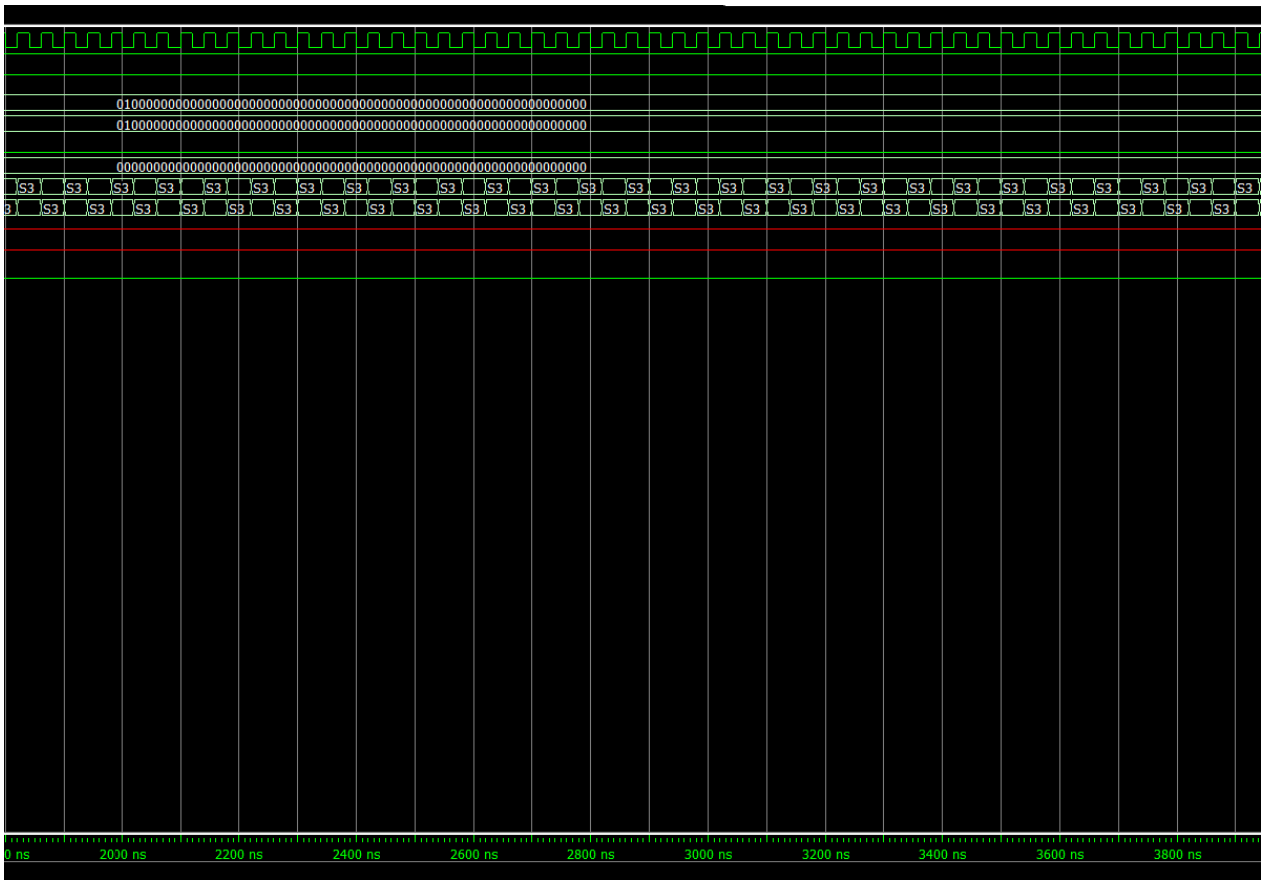
Poichè si doveva costruire un modello di FSMD, è stata descritta una entity che ha come porte clock, reset, isready (pronto per calcolare), number_a e number_b (numeri presi in input da cui calcolare il risultato), out_result (output del risultato). L'architecture contiene i segnali che fanno comunicare la FSM descritta e il Datapath della stessa, che sono gli stessi segnali descritti nel modello SystemC.

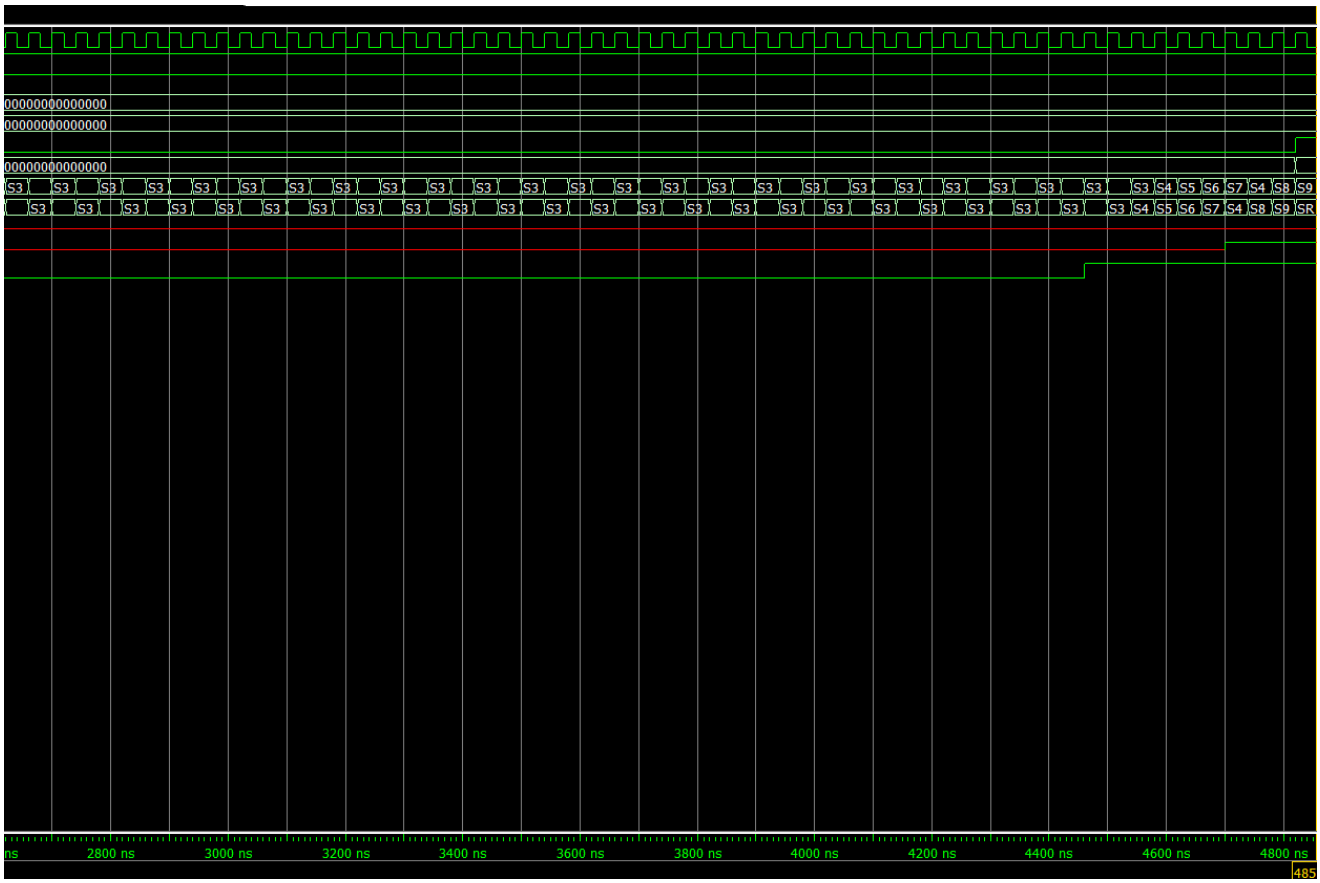
La parte sequenziale algoritmica, descritta con i relativi process, descrive ciò che effettivamente accade dentro alla FSM, che cambierà stato rispetto ai segnali corrispondenti, e al Datapath.

La struttura è quindi praticamente identica a quella descritta nel modello in SystemC.

Tracing


Msgs
/mul/clock
/mul/reset
/mul/isready
/mul/number_a
/mul/number_b
/mul/result_isre...
/mul/out_result
/mul/status
/mul/next_status
/mul/error
/mul/normalizzato
/mul/end_manti...
Now
Cursor 1
200 ns
400 ns
600 ns
800 ns
1000 ns
1200 ns
1400 ns
1600 ns


2000 ns
2200 ns
2400 ns
2600 ns
2800 ns
3000 ns
3200 ns
3400 ns
3600 ns
3800 ns



Stimuli.do

```
add wave *

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

force reset 1 0

force number_a 01000000000000000000000000000000000000000000000000000000000000000

force number_b 01000000000000000000000000000000000000000000000000000000000000000

force isready 1 1

run 4860

quit
```

Spiegazione

Per stimolare il design, sono stati creati 3 file stimuli.do, tutti con lo stesso comportamento ma che rappresentano numeri di input diversi. Il file Stimuli, oltre a “stimolare” il sistema, ha anche il compito di eseguire il trace dei segnali, rendendo disponibile al simulatore di Modelsim tutti i segnali da considerare.

Gli screenshot precedenti mostrano il funzionamento del sistema eseguendo stimuli.do.

In particolare, i comandi indicano questo:

- add wave *: Aggiunge tutte le porte e tutti i segnali definiti all'interno dell'architettura nella Waveform;
- force clock 1 20 ns, 0 40 ns -repeat 40: Forza il valore del clock ad 1 e lo ripete ogni 20 ns;
- force reset 1 0: Forza il valore della porta di reste ad 1 al tempo 0;
- force number_a 0100000000000000000000000000000000000000000000000000000000000000: Assegna il valore 2 in double;
- force number_b 0100000000000000000000000000000000000000000000000000000000000000: Assegna il valore 2 in double.
- force isready 1 1: Assegna il valore 1 al tempo 1 alla porta isready;
- run 4860: L'esecuzione verrà eseguita per un t=4860ns.

Significato degli stati

Anche se è stata una traduzione sintattica, è bene spiegare il significato degli stati dei due sottocomponenti, ovvero del datapath e della fsm.

```
when SR =>
          for i in 0 to 63 loop
             out_result(i) <= '0';
      end loop;
          for i in 0 to 127 loop
             mantissa_tot(i) := '0';
      end loop;
          for i in 0 to 127 loop
             mantissa1(i) := '0';
      end loop;
          for i in 0 to 127 loop
             mantissa2(i) := '0';
      end loop;
          for i in 0 to 63 loop
             buff(i) := '0';
      end loop;
          for i in 0 to SIZE-1 loop
             exp_tot(i) := '0';
```

Nello stato SR, si inizializzano tutte le variabili e i segnali. Da ricordare che, visto che si parla di hardware, bisogna inizializzare i singoli bit; i cicli for, al momento della sintesi, vengono "srotolati".

```
when S0 =>

          for i in 0 to 63 loop

             out_result(i) <= '0';

       end loop;
```

Nello stato So, si inizializza, per sicurezza, l'output risultante.

```
            when S0I =>

              vcl_number_a := number_a;

              vcl_number_b := number_b;


              exp1(10 downto 0) := number_a(62 downto 52);

              exp2(10 downto 0) := number_b(62 downto 52);


              mantissa1(51 downto 0) := number_a(51 downto 0);

              mantissa2(51 downto 0) := number_b(51 downto 0);


              mantissa1(52) := '1';

              mantissa2(52) := '1';


              sign1 := number_a(63);

              sign2 := number_b(63);


              counter := to_unsigned(0, 33);
```

Nello stato SoI, si preparano le variabili e i segnali per il calcolo, estrapolando dagli input ciò che serve per il calcolo.

```
            WHEN S1 =>

              --magic cast

              exp_tot := std_logic_vector((unsigned(exp1)) +
(unsigned(exp2)));




            WHEN S2 =>

              exp_tot := std_logic_vector((unsigned(exp_tot) -
1023));




            WHEN S3 =>

              if (counter <= 52) then

                end_mantissa <= '0';


              else

                end_mantissa <= '1';

              end if;
```

Nello stato S1, si calcola l'esponente.

```
            WHEN S3 =>

              if (counter <= 52) then

                end_mantissa <= '0';


              else

                end_mantissa <= '1';

              end if;
```

Nello stato S3, si è nel controllo del ciclo in cui si calcola la mantissa.

```
            WHEN S31 =>

              if(mantissa2(to_integer(counter)) = '1') then

                buff := std_logic_vector((unsigned(mantissa_tot) +
(unsigned(mantissa1) sll to_integer(unsigned(counter)))));


                mantissa_tot := buff;


              end if;


              counter := counter + 1;


            WHEN S4 =>

              if(unsigned(mantissa_tot(127 downto 105)) /= 0) then

                mantissa_tot := std_logic_vector(unsigned(mantissa_tot) srl 1);

                buff :=
std_logic_vector(to_unsigned(to_integer(unsigned(exp_tot(63 downto 0))) + 1,
128));

                exp_tot := buff(63 downto 0);


              elsif(mantissa_tot(104) /= '1') then

                mantissa_tot := std_logic_vector(unsigned(mantissa_tot) sll 1);

                buff(63 downto 0) := std_logic_vector(unsigned(exp_tot(63 downto
0)) - 1);

                exp_tot := buff(63 downto 0);


              end if;
```

Nello stato S31 si esegue il calcolo della mantissa e dell'esponente in base alla normalizzazione.

```
                WHEN S5 =>

                  if(unsigned(exp_tot) = 0 or unsigned(mantissa_tot)
= 0) then

                    error <= '1';

                     end if;
```

Nello stato S5 si controlla che non ci siano stati errori.

```
                WHEN S7 =>

                  if(unsigned(mantissa_tot(127 downto 105)) = 0 and
mantissa_tot(104) = '1') then

                    normalizzato <= '1';



                  else

                      if(unsigned(mantissa_tot(127 downto 105)) = 0
and unsigned(mantissa_tot(104 downto 0)) = 0) then

                        normalizzato <= '1';

                      else

                        normalizzato <= '0';

                      end if;

                  end if;
```

Nello stato S7 si controlla che la mantissa sia stata correttamente normalizzata.

```
                WHEN S8 =>

                  sign_tot := sign1 xor sign2;
```

Nello stato S8 si calcola il segno.

```
                WHEN S9 =>

                  result_tot(63) := sign_tot;

                  result_tot(62 downto 52) := exp_tot(10 downto 0);

                  result_tot(51 downto 0) := mantissa_tot(103 downto
52);  --prima era downto 52

                  out_result <= result_tot;

                  result_isready <= '1';
```

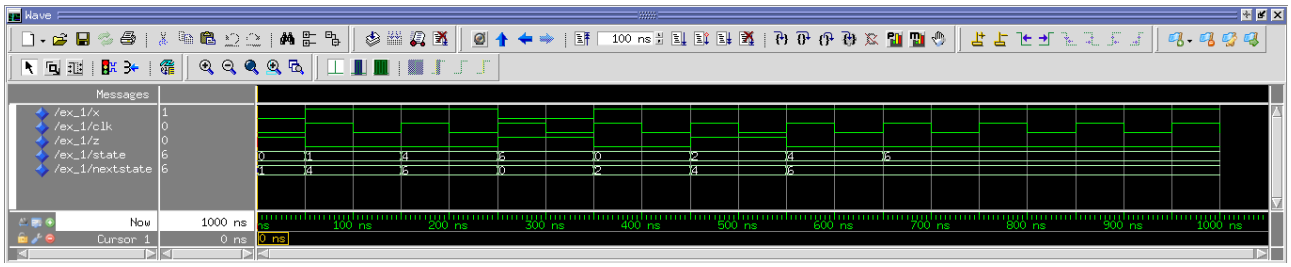
Nello stato S9 invio l'output calcolato.

Progetto 8: VHDL Timing Simulation

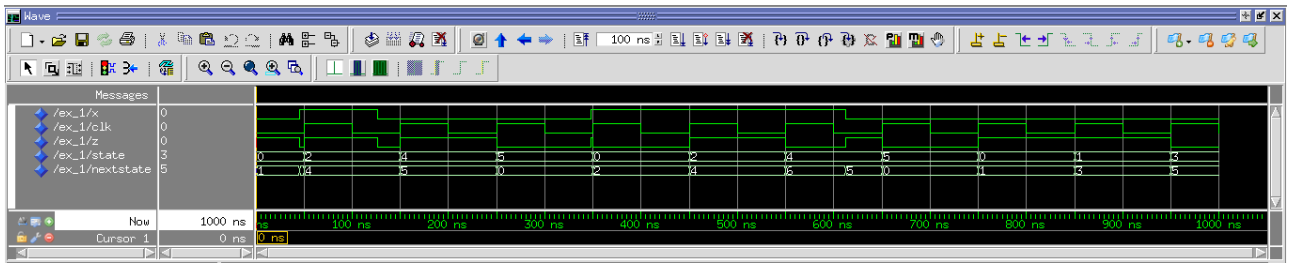
ES1:

stimuli1 VS stimuli3

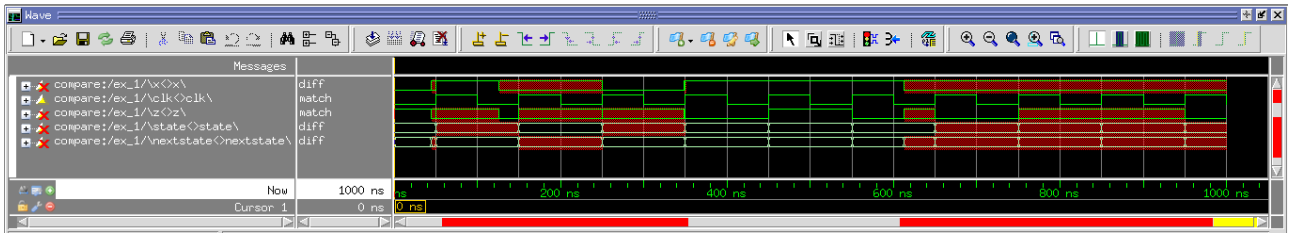
stimuli_1



stimuli_3



confronto:



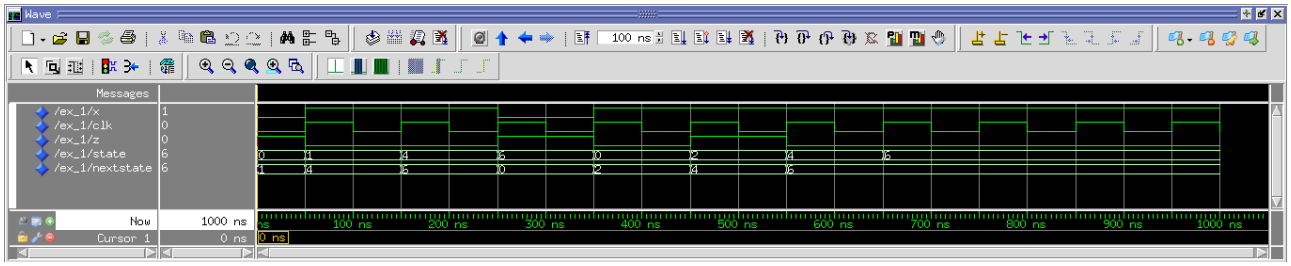
Nella prima simulazione, vengono eseguiti i processi sensibili a “x” e al “clock”, quindi al fronte di salita di quest’ultimo vengono assegnati in questo modo: x=1, state <= nextstate, nextstate = 4, z <= 0.

Nella seconda simulazione, viene attivato il processo sensibile al fronte di salita del clock, viene aggiornato lo stato (state <= nextstate) che fa attivare il processo sensibile a state; dal momento che x vale 0, viene assegnato a nextstate 3 e z a 1; infine x cambia valore e viene eseguito il processo sensibile a x, assegnando 4 a nextstate e 0 a z.

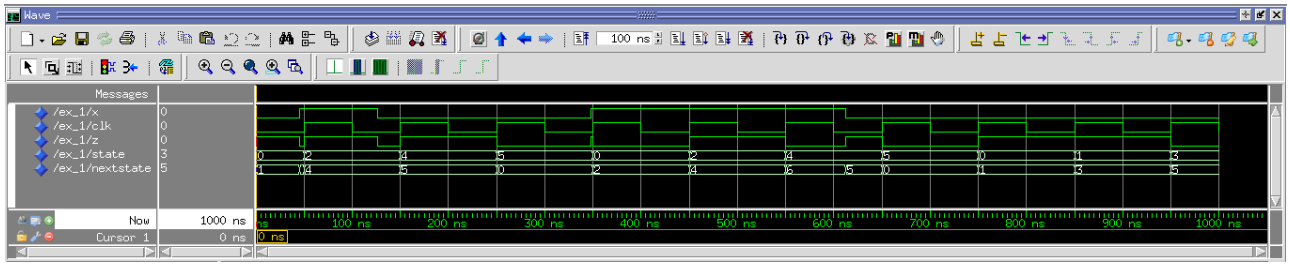
Gli spazi rossi della figura che rappresenta il confronto tra le simulazioni mostra questo, anche per i successivi rising edge del clock.

Stimuli2 VS Stimuli3

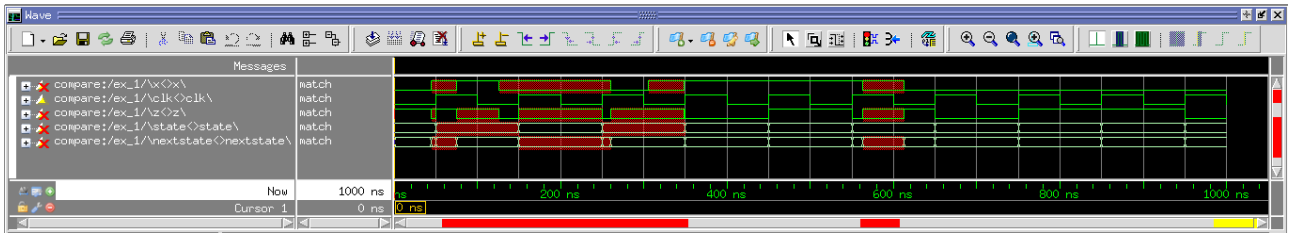
stimuli_2



stimuli_3



Confronto:



Nella prima simulazione vengono eseguiti i processi sensibili al segnale “x” e al “clock”, quindi sensibili al fronte di salita. Dopo il primo fronte di salita del clock si ha x a 1, state a nextstate, nextstate a 4 e z a 0.

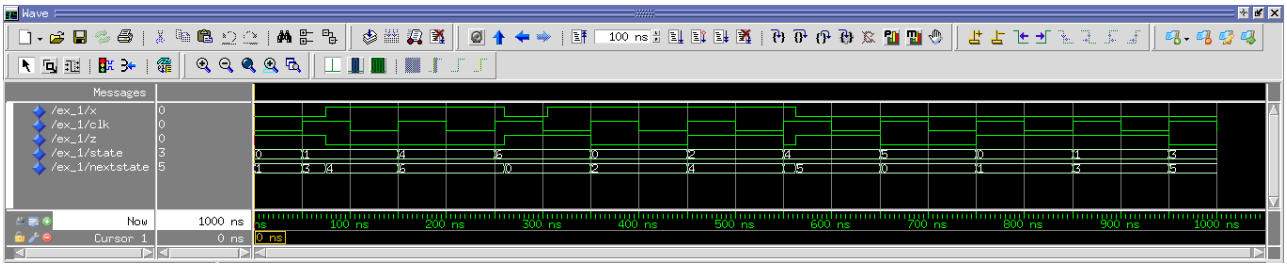
Nella seconda simulazione, visto che x cambia prima del fronte di salita del clock, si attiva il processo sensibile ad esso poichè cambia lo stato (state vale 0) e x (vale 1); quindi assegna 2 a nextstate e 0 a z.

Infine al primo fronte di salita del clock si attiva il processo sensibile ad esso, assegnando nextstate a state e svegliando il processo sensibile a state, assegnando 4 a nextstate e 1 a z.

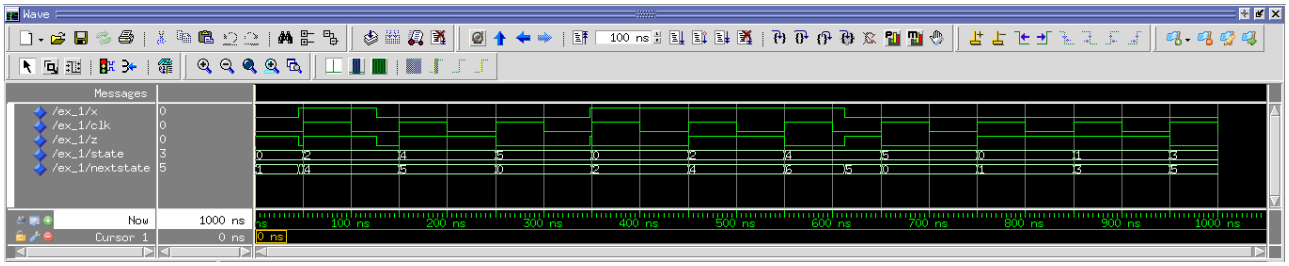
Gli spazi rossi della figura che rappresenta il confronto tra le simulazioni mostra questo, anche per i successivi rising edge del clock.

Stimuli1 VS Stimuli3

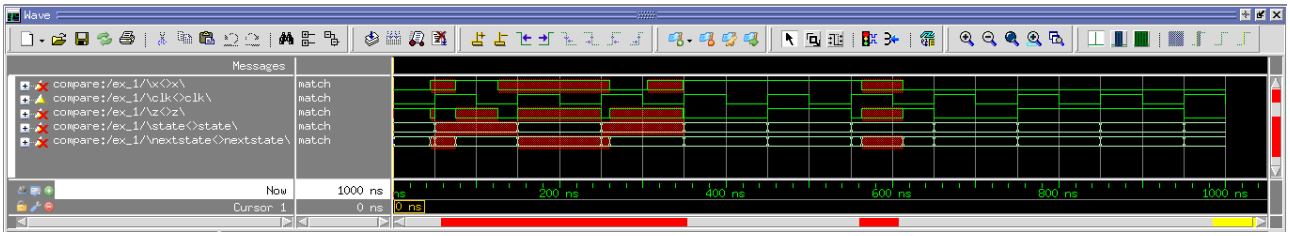
stimuli1



stimuli3



Confronto:



Nella prima simulazione viene eseguito il processo sensibile al "clock", aggiornando state con nextstate e attivando quindi il processo sensibile a quest'ultimo dopo l'assegnamento di 0 a x. Infine assegna 3 a nextstate e 1 a z; x cambia valore e viene eseguito il processo sensibile a questo, impostando a 4 il segnale nextstate e z a 0.

Nella seconda simulazione x cambia prima del fronte di salita del clock, quindi “poco prima” che ci sia il rising edge; quindi si attiva il processo sensibile con state a 0 e x a 1, assegnando 2 a nextstate e 0 a z.

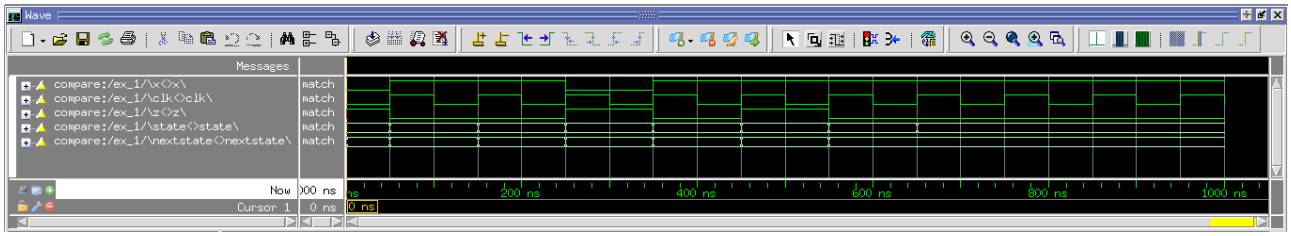
Infine, al primo fronte di salita del “clock” si attiva il processo sensibile ad esso, assegnando state a nextstate e svegliando il processo sensibile a quest’ultimo, che assegna 4 a nextstate e 1 a z.

Gli spazi rossi della figura che rappresenta il confronto tra le simulazioni mostra questo, anche per i successivi rising edge del clock.

ES2:

La parte in comune delle due simulazioni (es1 e es2) è la fase di inizializzazione, con la differenza che nella versione con 1 processo senza sensitivity list (es2) l’esecuzione viene interrotta nella “wait on clk, x”. In questa fase si assegnano i valori z <= 1 e nextstate <=1.

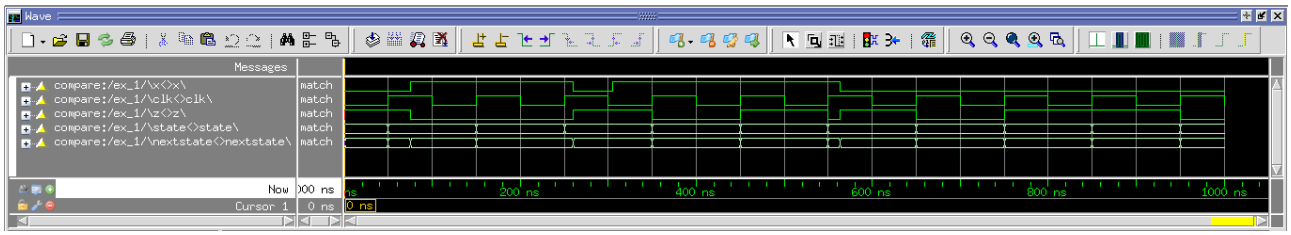
stimuli1



Processo ('ex2'): Al primo fronte di salita del “clock” avviene l’assegnamento state <= nextstate e con 'WAIT FOR 0 ns' viene eseguito il CASE aggiornando “state”, che assegna z <= 0 e nextstate <= 4.

Processo ('ex1’): Al primo fronte di salita del “clock”, sia nel caso in cui si esegue prima il processo sensibile ad “x” sia nel caso in cui venga eseguito quello sensibile al clock prima degli altri, al termine dell'istante di simulazione corrente si ha z <= 0 e nextstate <= 4 , con la differenza che nel primo caso avviene in un delta cycle in più rispetto al secondo.

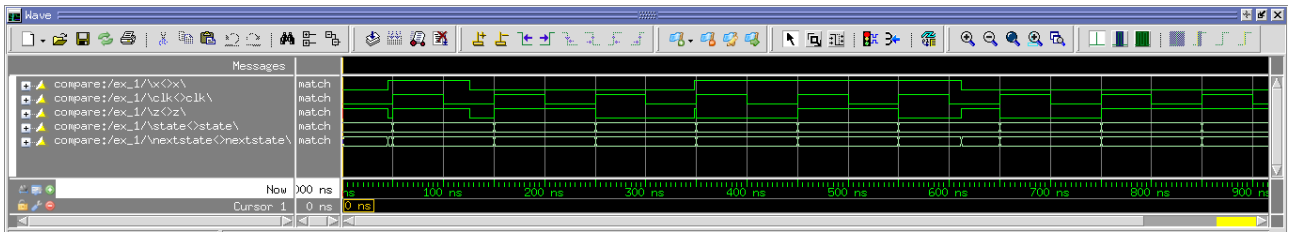
stimuli2



Processo ('ex2'): Al primo fronte di salita del "clock" viene assegnato state <= nextstate e con 'WAIT FOR 0 ns' viene eseguito il CASE con il valore aggiornato per state, che assegna z <= 1 e nextstate <= 3. Dopo ciò, il valore di x diventa 1 sbloccando la WAIT su di esso e ritornando sul CASE (in quanto non ci sono ancora eventi sul clock), il quale, poichè state = 1 e x = 1, assegna z <= 0 e nextstate <= 4.

Processo ('ex1'): Al primo fronte di salita del "clock" viene assegnato state <= nextstate; questo attiva il processo sensibile a state che, vedendo state = 1, assegna z <= 1 e nextstate <= 3. A questo punto x diventa 1 sbloccando il processo sensibile ad esso, che vedendo ancora state = 1 e x = 1, assegna z <= 0 e nextstate <= 4.

stimuli3



Processo ('ex2'): x si alza prima del fronte di salita del clock, quindi viene sbloccata la WAIT su di esso ed il CASE, vedendo state = 0 e x = 1, assegna z <= 0 e nextstate <= 2. Al primo fronte di salita del clock viene assegnato state <= nextstate e grazie al 'WAIT FOR 0 ns' viene eseguito il CASE con il valore aggiornato per state (2), il quale assegna z <= 1 e nextstate <= 4.

Processo ('ex1'): x si alza prima del fronte di salita del clock, quindi viene eseguito il processo sensibile ad esso con state = 0 e x = 1, assegnando z <= 0 e nextstate <= 2. Al primo fronte di salita del clock viene assegnato state <= nextstate sbloccando il processo sensibile ad esso, il quale vedendo state = 2 assegna z<=1 e nextstate <= 4 .

1 Processo vs. 2 Processi

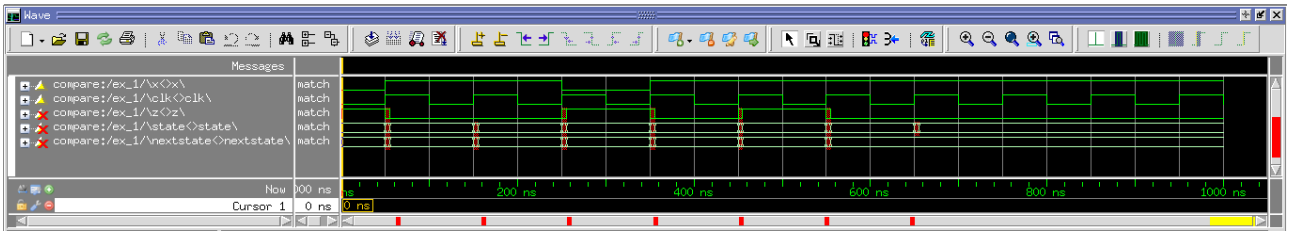
Come si può notare nei confronti tra le due versioni del modulo, nelle waveform dei 3 casi precedentemente considerati il design dell'Esercizio 1 implementato con 2 processi con Sensitive

List, produce gli stessi risultati del design con 1 singolo processo senza Sensitive List ma con il costrutto WAIT. Nel caso in cui nella versione con 1 processo (Esercizio 2) fosse mancato il costrutto 'WAIT

FOR 0 ns', il segnale state non sarebbe stato visto aggiornato dal CASE nello stesso tempo di simulazione (anche se con un Delta di ritardo), potendo provocare comportamenti differenti nell'esecuzione.

ES3:

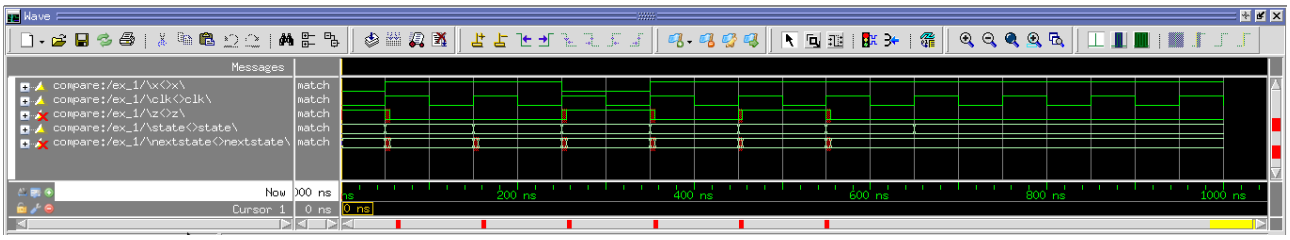
Version A



Dal confronto delle waveform si può vedere che nella Versione A del modulo, gli assegnamenti dei valori aggiornati ai segnali z, state e nextstate avvengono con 5ns di ritardo rispetto alla versione originale, mantenendo comunque gli stessi valori.

Il ritardo 'AFTER 5 ns' sull'assegnamento 'state <= nextstate' fa in modo che state e nextstate si aggiornino insieme dopo i 5ns, istante in cui viene sbloccata la WAIT e si aggiorna nextstate.

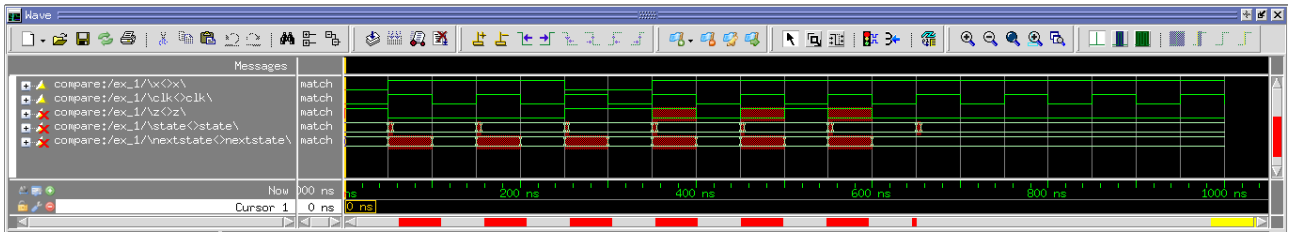
Version B



Dal confronto delle waveform si puo vedere che nella Versione B del modulo, gli assegnamenti dei valori aggiornati ai segnali z e nextstate avvengono con 5ns di ritardo rispetto alla versione originale, mantenendo comunque gli stessi valori e non ritardando l'aggiornamento del segnale state.

L'assegnamento state <= nextstate senza ritardi specificati fa in modo che state venga aggiornato al successivo ciclo delta, mentre nextstate deve attendere 5ns prima che venga sbloccato il processo dalla WAIT per aggiornarlo.

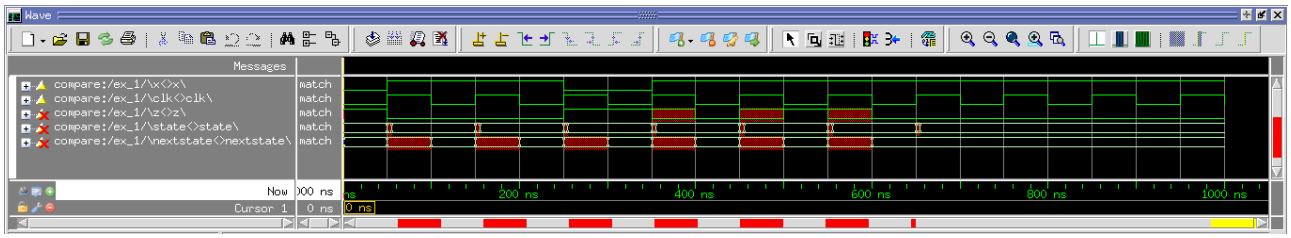
Version C



Dal confronto delle waveform si puo vedere che nella Versione C del modulo, l'assegnamento del valore aggiornato al segnale state avviene con 5ns di ritardo rispetto alla versione originale; nel frattempo il processo non attende l'aggiornamento dello state (attende solo un ciclo delta con una 'WAIT FOR 0 ns) ed entra nel CASE aggiornando i segnali nextstate e z in base al vecchio valore di state e di x.

Come si può notare dalle zone in rosso sulla waveform di confronto, si hanno valori differenti assegnati ai segnali z e nextstate con ritardi di anche un ciclo di clock, dovuti al fatto che mentre si attende l'aggiornamento di state il processo prosegue la sua esecuzione.

Version D



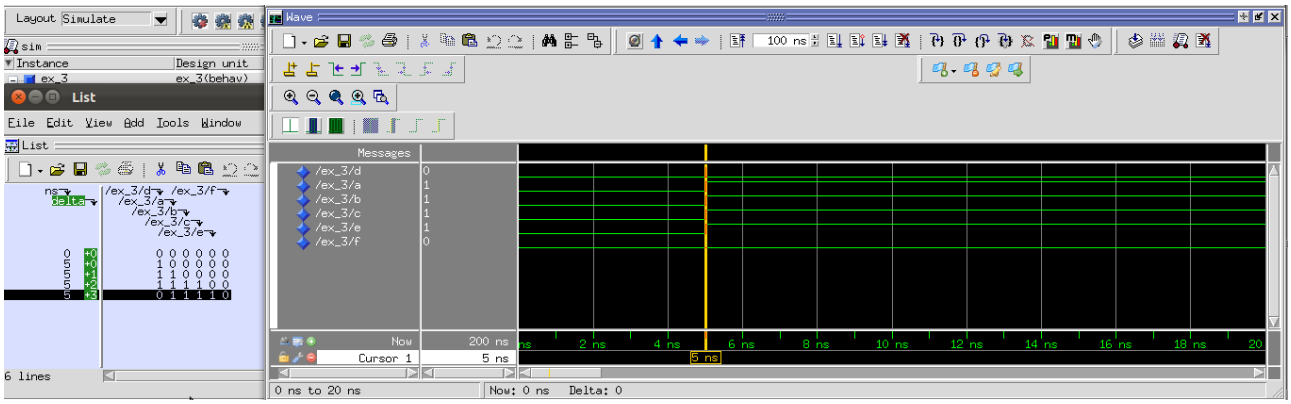
Dal confronto delle waveform si può vedere che nella Versione D del modulo si ha un comportamento uguale al caso della Versione C, in quanto nonostante non sia presente una 'WAIT FOR 0 ns' dopo l'assegnamento 'state <= nextstate, l'aggiornamento di state avviene comunque dopo 5ns indipendentemente da qualsiasi attesa (se questa è minore di 5ns).

ES4:

In questo esercizio bisogna creare la tabella di simulazione mostrando, per come cambia il tempo di simulazione, il valore corrente del singolo segnale e la coda dei valori

TIME	QUEUED VALUES	QUEUED VALUES	CURRENT VALUE	SIGNAL
@0			0	A
@0			0	B
@0	0 @5	1 @delta	0	A
@0			0	B
@delta		0 @5	1	A
@delta		1 @10	0	B
@5			0	A
@5		1 @10	0	B
@10	0 @15	1 @10+delta	0	A
@10			1	B
@10+delta		0 @15	1	A
@10+delta		0 @20	1	B
@15			0	A
@15		0 @20	1	B

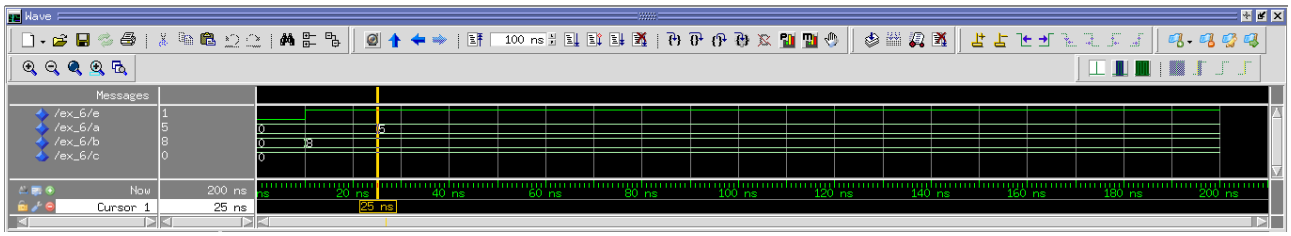
ES5:



TIME	QUEUED VALUES	QUEUED VALUES	CURRENT VALUE	SIGNAL
@0			0	A
@0			0	B
@0			0	C
@0		1 @5	0	D
@0			0	E
@0			0	F
@5		1 @5+delta	0	A
@5			0	B
@5		0@5+delta	0	C
@5			1	D
@5			0	E
@5			0	F
@5+delta		1 @5+2delta	1	A
@5+delta		1 @5+2delta	0	B
@5+delta		1 @5+2delta	0	C
@5+delta			1	D
@5+delta			0	E
@5+delta			0	F
@5+2delta		1 @5+3delta	1	A
@5+2delta			1	B
@5+2delta		1 @5+3delta	1	C
@5+2delta		1 @5+3delta	1	D
@5+2delta		1 @5+3delta	0	E
@5+2delta		1 @5+3delta	0	F
@5+3delta		1 @5+4delta	1	A
@5+3delta			1	B
@5+3delta		1 @5+4delta	1	C
@5+3delta			0	D
@5+3delta			1	E
@5+3delta			0	F
@5+4delta		1 @5+5delta	1	A
@5+4delta			1	B
@5+4delta		1 @5+5delta	1	C
@5+4delta		1 @5+5delta	0	D
@5+4delta		1 @5+5delta	1	E
@5+4delta		1 @5+5delta	0	F

Negli istanti di tempo successivi i segnali smettono di variare i loro valori. Si può notare come i Driver dei Segnali rispecchino esattamente il loro comportamento nel tempo indicato nella List View e nella Wave View di Modelsim.

ES6:



TIME	QUEUED VALUES	QUEUED VALUES	CURRENT VALUE	SIGNAL
@0			0	A
@0			0	B
@0			0	C
@0			0	D
@0		1 @10	0	E
@10		1 @15	0	A
@10		1@10+delta	0	B
@10		0@20	0	C
@10			0	D
@10			1	E
@10+delta		1 @25	0	A
@10+delta		8 @10+2delta	1	B
@10+delta		0 @20	0	C
@10+delta		1@13	0	D
@10+delta			1	E
@10+2delta		5 @25	0	A
@10+2delta			8	B
@10+2delta		0@20	0	C
@10+2delta		1 @13	0	D
@10+2delta			1	E
@13		5@25	0	A
@13			8	B
@13		0@20	0	C
@13			1	D
@13			1	E
@20		5@25	0	A
@20			8	B
@20			0	C
@20			1	D
@20			1	E
@25			5	A
@25			8	B
@25			0	C
@25			1	D
@25			1	E