Modellazione e Sintesi di un Moltiplicatore Floating-point Single Precision

Enrico Sgarbanti - VR446095

***Sommario*—Questo documento mostra la realizzazione di un moltiplicatore in virgola mobile a precisione singola realizzato in VHDL, Verilog e SystemC ed un componente che permetta di eseguire due moltiplicazioni in parallelo. Il tutto è accompagnato da testbench, sintesi dei componenti VHDL e verilog ed un confronto con l'High-level-Synthesis di un moltiplicatore scritto in c++.**

I. Introduzione

Il progetto consiste nella realizzazione in hardware di un sistema, che attraverso il protocollo di handshake, utilizza due moltiplicatori in virgola mobile a precisione singola, secondo lo standard IEEE754, per eseguire due moltiplicazioni in parello. Esso deve essere sintetizzabile sulla scheda FPGA "xc7z020clg400-1" che possedendo solo 125 porte I/O, obbliga a serializzare input e output.
Il sistema è stato realizzato in diversi linguaggi al fine di vedere le differenze tra i vari linguaggi utilizzabili per descrivere hardware e capirne i pro e contro. I risultati sono stati poi analizzati e confrontati con quelli ottenuti dall'high level syntesis del codice in c++.
L'approccio utilizzato è bottom-up, cioè si è partiti dal moltiplicatore per poi arrivare al top level. L'implementazione è preceduta dall'analisi dei requisiti e dalla stesura della EFSM, cioè la parte più importante in quanto è dove viene tradotto l'algoritmo, descritto il flusso e scelti i vari segnali e registri necessari. Una buona EFSM permette di evitare di scrivere varie righe di codice per poi accorgersi in simulazione che qualcosa non funziona.
Ci si aspetta che la versione RTL sia significativamente più performante di quella con l'high level syntesis e che il sistema occupi una minima parte della FPGA in quanto molto piccolo.

II. Background

A. Progettazione hardware

Per la realizzazione di componenti hardware si possono utilizzare diverse tecniche e linguaggi.
Un primo approccio è descrivere i componenti a livello RT utilizzando linguaggi di descrizione hardware (**HDL**) come VHDL e Verilog. Un HDL è un linguaggio specializzato per la descrizione della struttura e del comportamento di circuiti elettronici, in particolare circuiti logici digitali, e la loro analisi e simulazione. Permette inoltre la sintesi di una descrizione HDL in una netlist (una specifica di componenti elettronici fisici e il modo in cui sono collegati insieme), che può quindi essere posizionata e instradata per produrre l'insieme di maschere utilizzate per creare un circuito integrato[1].
Un secondo approccio è descrivere le funzionalità del componente con linguaggi più ad alto livello come C, C++ o SystemC[2] e fare High Level Syntesis (**HLS**) per ottenere una descrizione dell'hardware a livello RT[3].
Entrambi gli approcci hanno vantaggi e svantaggi. In particolare HLS riduce i tempi, ma la descrizione hardware generata sarà meno ottimizzata rispetto a quella che si potrebbe ottenere usando un HDL.

B. IEEE 754 single-precision binary floating-point format

Questo standard definisce il **formato** per la rappresentazione dei numeri in virgola mobile (compreso ± 0 e i numeri denormalizzati; gli infiniti e i NaN, "not a number"), ed un set di operazioni effettuabili su questi.
In particolare la versione a precisione singola descrive il numero con 32 bit: 1 bit per segno (sign), 8 bit per l'esponente (esp) e 23 bit per la mantissa (mant)[4].

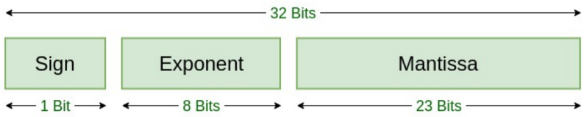

Figura 1. IEEE 754 single precision

Per la **codifica** in numero binario:

- Dal segno si ricava il bit più significativo (1 se negativato 0 altrimenti).
- Si converte il numero in binario.
- Si sposta la virgola a sinistra fino ad avere un numero nella forma $1, \ldots \cdot 2^E$ dove E è il numero di spostamenti.
- La mantissa è la parte a destra della virgola, con zeri a destra fino a riempire i 23 bit.
- L'esponente esp è uguale a $127 + E$ dove 127 è il *bias* di questo standard.

Per la **decodifica** del numero binario:

$$(-1)^{sign} \cdot 2^{(esp-127)} \cdot (1 + \sum_{i=1}^{23} b_{23-i} \cdot 2^{-i})$$

C. Moltiplicazione di due numeri floating-point

Qui è riportato un algoritmo utilizzabile per la moltiplicazione fra floating point. Guardare qui per ulteriori dettagli[5].

Categoria	Esp.	Mantissa
Zeri	0	0
Numeri denormalizzati	0	non zero
Numeri normalizzati	1-254	qualunque
Infiniti	255	0
Nan (not a number)	255	non zero

Figura 2. IEEE 754 special case

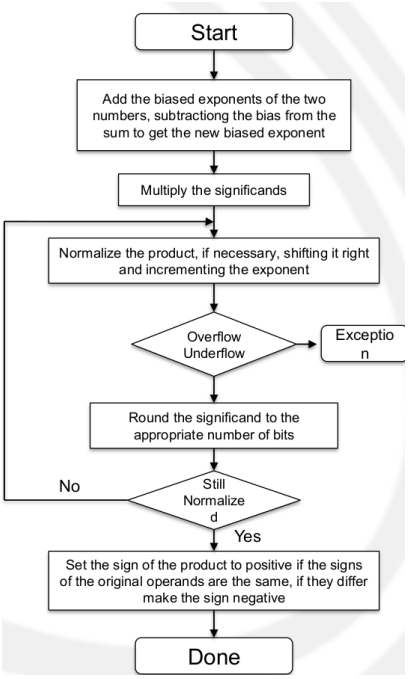

Figura 3. IEEE 754 multiplication

III. Metodologia applicata

A. Struttura progetto

- **cpp/**
  - **multiplier.cpp** file dove è presente una funzione che esegue la moltiplicazione in c++.
- **SystemC/** contiene il progetto del sistema scritto interamente in SystemC. La struttura è approfondita qui III-G.
- **VHDL_verilog/**
  - **stimuli/** contiene gli script TCL usati per dei test.
  - **waves/** contiene le configurazioni grafiche per visualizzare le simulazioni come negli screenshot.
  - **constrains/** contiene i vincoli per la sintesi.
  - **double_multiplier** file Verilog del top level del sistema
  - **verilog_multiplier** file Verilog del moltiplicatore IEEE754
  - **vhdl_multiplier** file VHDL del moltiplicatore VHDL
  - **testbench** file Verilog contenente il testbench

B. Procedimento

Il primo passo è stato la realizzazione della EFSM di *multiplier* e *double_multiplier* che ha portato ad aquisire una visione generale del sistema.
Poi si è passati all'implementazione a livello RT con Vivado[6] del *multiplier* in Verilog e VHDL. La loro correttezza è stata testata subito grazie ad uno script TCL osservando gli output a determinati input.
Consolidati questi moduli è stato poi possibile realizzare in Verilog il *double_multiplier* che prende i due componenti e li usa per calcolare due moltiplicazioni. Anch'esso è stato testato con uno script TCL.
Completato il sistema è stato fatto un testbench in Verilog per verificare che i moltiplicatori si comportassero allo stesso modo e che reagissero correttamente a casi particolari e numeri arbitrari.
In seguito è stato riscritto tutto in SystemC dove si è potuto fare un testbench più fine grazie alla potenza del c++. In particolare sono stati confrontati i valori ottenuti dal sistema con quelli derivati dal risultato convertito in binario della moltiplicazione degli stessi numeri convertiti in float.
Infine è stata fatta l'high level syntesis da un semplice codice c++ per confrontare i risultati ottenuti.

C. Vincoli ed Architettura

Il progetto presenta diversi vincoli:

- Il multiplier deve essere scritto in VHDL, verilog e systemC.
- Il double_multiplier deve essere scritto in systemC e un linguaggio a scelta tra VHDL e verilog.
- Gli operandi e il risultato devono essere a 32 bit.
- I due componenti devono essere sintetizzabili sulla FPGA "xc7z020clg400-1" la quale ha a disposizione solo 125 porte.

Per far fronte al limite delle porte logiche è stato necessario serializzare input e output. Vengono quindi utilizzati gli stessi 32 bit per il risultato e altri 64 bit per le due coppie di operandi. Grazie al protocollo di *handshake* si sono sincronizzati i vari componenti. Quando il flag *ready* diventa uguale a 1, vengono usati i valori di *op1* e *op2* a quel ciclo di clock per il primo moltiplicatore e quelli del ciclo di clock successivo per il secondo moltiplicatore. Non appena entrambi i moltiplicatori finiranno, il flag *done* del sistema diventerà uguale a 1 e allo stesso ciclo di clock *res* conterrà il risultato della prima moltiplicazione e al ciclo di clock successivo quello della seconda.
L'architettura con VHDL e Verilog è mostrata in figura 4. Quella per SystemC è analoga.
I segnali intermedi sono stati omessi da questa figura, ma vengono descritti nelle sezioni successive. La FSMD è realizzata con due processi:

- **fsm:** processo asincrono col compito di calcolare e aggiornare lo stato prossimo.
- **datapath:** processo sincrono che ha il compito di aggiornare lo stato attuale ed elaborare gli output. Esso viene però attivato asincronamente dal fronte di salita del reset al fine di riportare lo stato a quello iniziale.

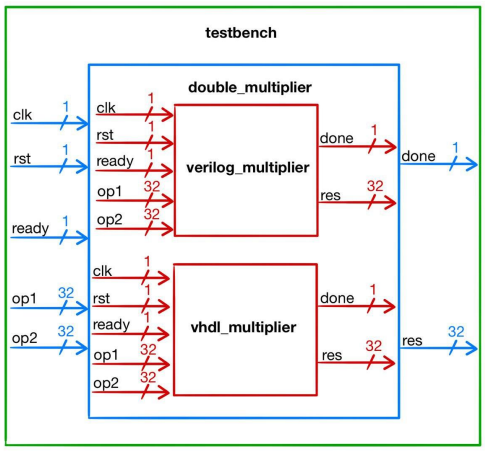

Figura 4. Architettura RTL

D. multiplier

Questo componente esegue la moltiplicazione tra numeri floating point a precisione singola.
È stato supposto che gli operandi passati non siano denormalizzati, ma in caso sarebbe facilmente estendibile aggiungendo qualche stato e un segnale di errore in uscita.
L'interfaccia è mostrata in figura 4 ed è la stessa per tutte le implementazioni VHDL, Verilog e SystemC:

- **op1** (32 bit input): primo operando.
- **op2** (32 bit input): secondo operando.
- **clk** (1 bit input): segnale di clock.
- **rst** (1 bit input): segnale di reset. Riporta il sistema allo stato iniziale.
- **ready** (1 bit input): segnale che permette al sistema di uscire dallo stato iniziale. Nello specifico indica che *op1* e *op2* contengono i valori dei due operatori.
- **done** (1 bit output): segnale che indica che il valore su *res* è il risultato.
- **res** (32 bit output): risultato.

Gli altri segnali/registri intermedi utilizzati sono:

- **norm_again** (1 bit): indica che la mantissa ha bisogno di essere ulteriormente normalizzata.
- **res_type** (2 bit): indica il tipo del risultato. Solo nel caso in cui sia un numero si procede all'elaborazione, mentre negli altri casi si passa direttamente allo stato finale.
- **STATE e NEXT_STATE** (4 bit): rappresentano lo stato attuale e lo stato prossimo.
- **op1_type e op2_type** (2 bit): indicano il tipo degli operandi ovvero 0, NaN, ∞ oppure un numero.
- **esp_tmp** (10 bit): permette di eseguire le operazioni per ricavare l'esponente finale senza perdere informazioni.
- **mant_tmp** (48 bit): permette di eseguire le operazioni per ricavare la mantissa finale senza perdere informazioni.
- **sign1, sign2, esp1, esp2, mant1, mant2**: rappresentano le componenti dei due operandi. *mant1* e *mant2* sono a 24 bit perchè aggiunguno 1. alla mantissa che nella rappresentazione IEE754 è omesso.

L'algoritmo della moltiplicazione è descritto grazie alla EFSM [Figura 5] la quale è formata 14 stati:

- **ST_START:** stato di partenza. Qui vengono resettati i segnali a zero e inizializzati i registri/variabili contenenti le informazioni di segno, esponente e mantissa degli operanti attuali. In caso di segnale di reset si torna in questo stato. In caso il segnale *ready* vada a 1 si passa a *ST_EVAL1*.
- **ST_EVAL1:** qui viene ricavato *op1_type* cioè se *op1* è 0, ∞, NaN o un numero. Poi si passa a *ST_EVAL2*.
- **ST_EVAL2:** qui viene ricavato *op2_type* cioè se *op2* è 0, ∞, NaN o un numero. Poi si passa a *ST_EVAL3*.
- **ST_EVAL3:** qui viene ricavato *res_type* cioè se *res* è 0, ∞, NaN o un numero. Poi si passa a *ST_CHECK1*.
- **ST_CHECK1:** se *res* è un numero si continua con l'elaborazione andando in *ST_ELAB* altrimenti il risultato è noto e si passa a *ST_FINISH*.
- **ST_ELAB:** qui si ricava l'esponente temporaneo di *res* sommando gli esponenti degli operandi e sottraendo il bias. Esso è a 10bit anzichè 8bit per permettere di compiere l'operazione senza perdere informazioni. Viene ricavato anche la mantissa temporanea che sarà di 48bit per contenere la moltiplicazione delle mantesse dei due operandi. Poi si passa a *ST_UNDERF*.
- **ST_UNDERF:** qui si controlla se il bit più significativo di *esp_tmp* è 1. Essendo codificato in complemento a 2, il fatto che sia 1 implica che è un numero negativo e quindi si è verificato un underflow e *res_type* sarà 0.
- **ST_CHECK2:** se *res* non è zero si continua con l'elaborazione andando in *ST_NORM1* altrimenti il risultato è noto e si passa a *ST_FINISH*.
- **ST_NORM1:** qui si effettua la normalizzazione della mantissa. Se il bit più significativo è 1 allora vuol dire la mantissa ha due cifre a sinistra della virgola e quindi deve essere normalizzata cioè riporta alla forma in cui c'è solo un 1 a sinistra. Per normalizzare basta incrementare l'esponente e considera la virgola spostata a sinistra. Nel caso in cui il bit più significativo sia 0 si fa uno shift a sinistra al fine di avere la virgola a destra del bit più significativo. Poi si passa a *ST_ROUND*.
- **ST_ROUND:** qui si controlla se c'è bisogno di arrotondare la mantissa, che da 48bit dovrà passare a 23bit, e in caso affermativo si pone a 1 *norm_again*. Si è deciso di arrotondare solo se il 23esimo bit, cioè quello a destra del punto di taglio, vale 1. Poi si passa a *ST_CHECK3*.
- **ST_CHECK3:** se *norm_again* vale 1 si passa a *ST_NORM2* altrimenti a *ST_OVERF*.
- **ST_NORM2:** qui avviene l'incremento della mantissa a causa dell'arrotondamento. Nel caso in cui la mantissa fosse formata da tutti 1 l'incremento la riporterebbe a tutti 0 e l'esponente andrebbe incrementato. Poi si passa a *ST_OVERF*.
- **ST_OVERF:** qui avviene il controllo dell'overflow guardando se il secondo bit più significativo di *esp_tmp* vale 1 e in caso affermativo *res_type* sarà ∞. Infine si passa a *ST_FINISH*.
- **ST_FINISH:** qui si ricava il segno del risultato facendo

lo XOR fra quello degli operandi, mentre si ricava il resto in base al valore di *res_type*. Infine si imposta *done* a 1 e si torna in *ST_START*.

E. double_multiplier

Questo componente esegue due moltiplicazioni tra numeri floating point a precisione singola.
La scelta di realizzarlo in Verilog è stata del tutto arbitraria.
L'interfaccia è mostrata in figura 4 ed è la stessa sia per l'implementazione Verilog che quella SystemC:

- **op1** (32 bit input): primo operando.
- **op2** (32 bit input): secondo operando.
- **clk** (1 bit input): segnale di clock.
- **rst** (1 bit input): segnale di reset. Riporta il sistema allo stato iniziale.
- **ready** (1 bit input): segnale che permette al sistema di uscire dallo stato iniziale. Nello specifico indica che in questo ciclo di clok *op1* e *op2* contengono gli operandi della prima moltiplicazione e nel ciclo di clock successivo ci saranno quelli per la seconda moltiplicazione.
- **done** (1 bit output): segnale che indica che il valore su *res* è il risultato.
- **res** (32 bit output): risultato.

Gli altri segnali/registri intermedi utilizzati sono:

- **ready1** (1 bit) segnale che pone il *ready* del primo multiplier (quello in Verilog) a 1.
- **ready2** (1 bit) segnale che pone il *ready* del secondo multiplier (quello in VHDL) a 1.
- **done1** (1 bit) segnale che indica che il valore su *res1* è il risultato della prima moltiplicazione.
- **done2** (1 bit) segnale che indica che il valore su *res2* è il risultato della seconda moltiplicazione.
- **op1_tmp, op2_tmp** (32 bit) che servono a memorizzare temporaneamente gli operandi per la prima moltiplicazione.

L'algoritmo è descritto grazie alla EFSM [Figura 6] la quale è formata 8 stati:

- **ST_START:** pone *done*, *ready1* e *ready2* uguali a 0 e inizializza *op1_tmp1* e *op2_tmp1* rispettivamenete con i valori di *op1* e *op2* i quali serviranno per il primo moltiplicatore. Si rimane qui finchè *ready* vale 0 altrimensi si passa a *ST_RUN1*. Se in qualsiasi stato si riceve *reset* uguale a 1 allora si passa a questo stato e si pone *rst* uguale a 1 entrambi i *rst* dei multiplier.
- **ST_RUN1:** pone *ready1* uguale a 1, attivando quindi il primo moltiplicatore, e inizializza *op1_tmp2* e *op2_tmp2* rispettivamenete con i valori di *op1* e *op2* i quali serviranno per il secondo moltiplicatore.
- **ST_RUN2:** pone *ready1* uguale a 0 e *ready2* uguale a 1, attivando quindi il secondo moltiplicatore.
- **ST_WAIT:** pone *ready2* uguale a 0. Si rimane in questo stato finchè $done1 = 1$ e in quel caso si passa a *ST_WAIT2* oppure che $done2 = 1$ passando a ST_WAIT1. Nel caso in cui sia $done1 = 1$ che $done2 = 1$ allora si passa direttamente a *ST_RET1*.
- **ST_WAIT1:** si resta qui finchè non finisce anche il primo moltiplicatore, cioè finchè $ready1 = 0$.
- **ST_WAIT2:** si resta qui finchè non finisce anche il secondo moltiplicatore, cioè finchè $ready2 = 0$.
- **ST_RET1:** pone *done* uguale a 1 e *res* uguale al risultato del primo moltiplicatore cioè *res1*.
- **ST_RET2:** pone *res* uguale al risultato del secondo moltiplicatore cioè *res2* e ritorna allo stato iniziale.

F. Implementazione RTL con Verilog e VHDL

In **verilog_multiplier** e *double_multiplier*

- Sono definiti come "wire" tutti i segnali collegati alle porte di input mentre come "reg" tutti i registri collegati alle porte di output e quelli intermedi.
- Gli stati e *op1_type, op2_type, res_type* sono stati definiti come "parameter".

In **vhdl_multiplier:**

- Sono usate le librerie "IEEE.STD_LOGIC_1164.ALL" per abilitare i tipi std_logic e "use IEEE.NUMERIC_STD.ALL" per usare funzioni aritmetiche con valori signed e unsigned
- Sono definiti come "signal" tutti i segnali collegati alle porte di input e output.
- Sono definiti come "signal" tutti i segnali interni di communicazione per la FSM.
- Sono definite come "variable" *sign1, sign2, esp1, esp2, esp_tmp, mant1, mant2, mant_tmp, op1_type, op2_type* perchè utilizzati solo all'interno del processo "datapath".
- Gli stati e *op1_type, op2_type, res_type* sono stati definiti all'interno del *package* rispettivamente come "MULT_STATE" e "MULT_TYPE".
- L'architettura utilizzata segue lo stile "behavioral", cioè quello più "program-like" in quanto più semplice e chiaro per descrivere una FSMD con due processi.

G. Implementazione RTL con SystemC

Si creano i seguenti files e directory:

- **Makefile:** tool per la compilazione automatica del progetto. Richiede che la variabile d'ambiente SYSTEMC_HOME contenga il path alla libreria di SystemC.
- **bin:** directory che contiene l'eseguibile *double_multiplier_RLT.x* (generato dopo la compilazione) e *wave.vcd* (generato dopo l'esecuzione dell'eseguibile).
- **obj:** directory che contiene i files oggetto (generati dopo la compilazione)
- **include:** directory che contiene gli headers *double_multiplier_RTL.hh*, *multiplier_RTL.hh*, *testbench_RTL.hh*. Qui sono definite tutte le porte, segnali, variabili ed enumerazioni dei vari componenti
- **src:** directory che contiene i files sorgenti *double_multiplier_RTL.cc*, *multiplier_RTL.cc*, *testbench_RTL.cc* e *main_RTL.cc*.

In **double_multiplier_RTL.hh**

- Sono definiti come "sc_signal" tutti i segnali collegati alle porte di input e output.
- Sono definiti come "sc_signal" tutti i segnali interni di communicazione per la FSM.

- Gli stati sono stati definiti come “enumerazioni”.

In **multiplier_RTL.hh**

- Sono definiti come “sc_signal” tutti i segnali collegati alle porte di input e output.
- Sono definiti come “sc_signal” tutti i segnali interni di communicazione per la FSM.
- Sono definite come variabili di SystemC *sign1, sign2, esp1, esp2, esp_tmp, mant1, mant2, mant_tmp, op1_type, op2_type* perchè utilizzati solo all’interno del processo “datapath”.
- Gli stati e *op1_type, op2_type, res_type* sono stati definiti come “enumerazioni”.

A differenza di verilog e VHDL, in SystemC è necessario un file “main” che contenga il metodo *sc_main* e che permetta di collegare il componente da testare con il testbench. In esso si utilizza *sc_create_vcd_trace_file* per salvare le tracce necessarie a lanciare una simulazione con tools come gtkwave. Per eseguire il componente bisogna usare i comandi:

- **make:** per compilare.
- **./bin/double_multiplier_RTL.x:** per eseguire il programma.
- **gtkwave/wave.vcd:** per lanciare la simulazione.

IV. Testbench

Il **testbench in Verilog**, [Figura reffig:SIM1], aspetta un po di tempo, perchè altrimenti si verificherebbero problemi dovuti allo startup della FPGA nella simulazione post-sintesi, e poi esegue due volte il *double_multiplier*, prima con due coppie di operandi che danno come risultato dei numeri normali, e poi con due coppie di operandi che danno come risultati dei casi speciali.

Il **testbench in SystemC** mette a disposizione tre thread da attivare togliendo i commenti nel costruttore del “TestbenchModule”:

- **targeted_test:** test analogo a quello in Verilog. [Figura reffig:SIM2].
- **rnd_test:** test che prova *TESTS_NUM* moltiplicazioni generate casualmente tra un intervallo modificabile. [Figura reffig:SIM3].
- **run_all:** che prova tutte le possibili combinazioni cioè $2^{32} * 2^{32}$. Si può limitare il numero di combinazioni evitando di contare il bit del segno, in quanto il calcolo è un semplice xor. Poi si possono escludere tutti i numeri denormalizzati. In ogni caso risulta troppo pesante per essere eseguito.

V. risultati

A. Simulazioni con script TCL

B. Simulazione con testbench in Verilog

I test precedenti servivano ad assistere la progettazione, questo invece a verificare la correttezza del sistema. Sono infatti testati tutti i casi particolari e valori scelti arbitrariamente.

VI. Conclusioni

Riferimenti bibliografici

[1] “Hdl,” https://en.wikipedia.org/wiki/Hardware_description_language.
[2] Accellera Systems Initiative *et al.*, “Systemc,” *Online, December*, 2013.
[3] “Hls,” https://en.wikipedia.org/wiki/High-level_synthesis.
[4] “Ieee 754,” https://en.wikipedia.org/wiki/IEEE_754.
[5] “Ieee 754 multiplication,” https://en.wikipedia.org/wiki/Single-precision_floating-point_format.
[6] “Vivado,” https://www.xilinx.com/products/design-tools/vivado.html.

Appendice

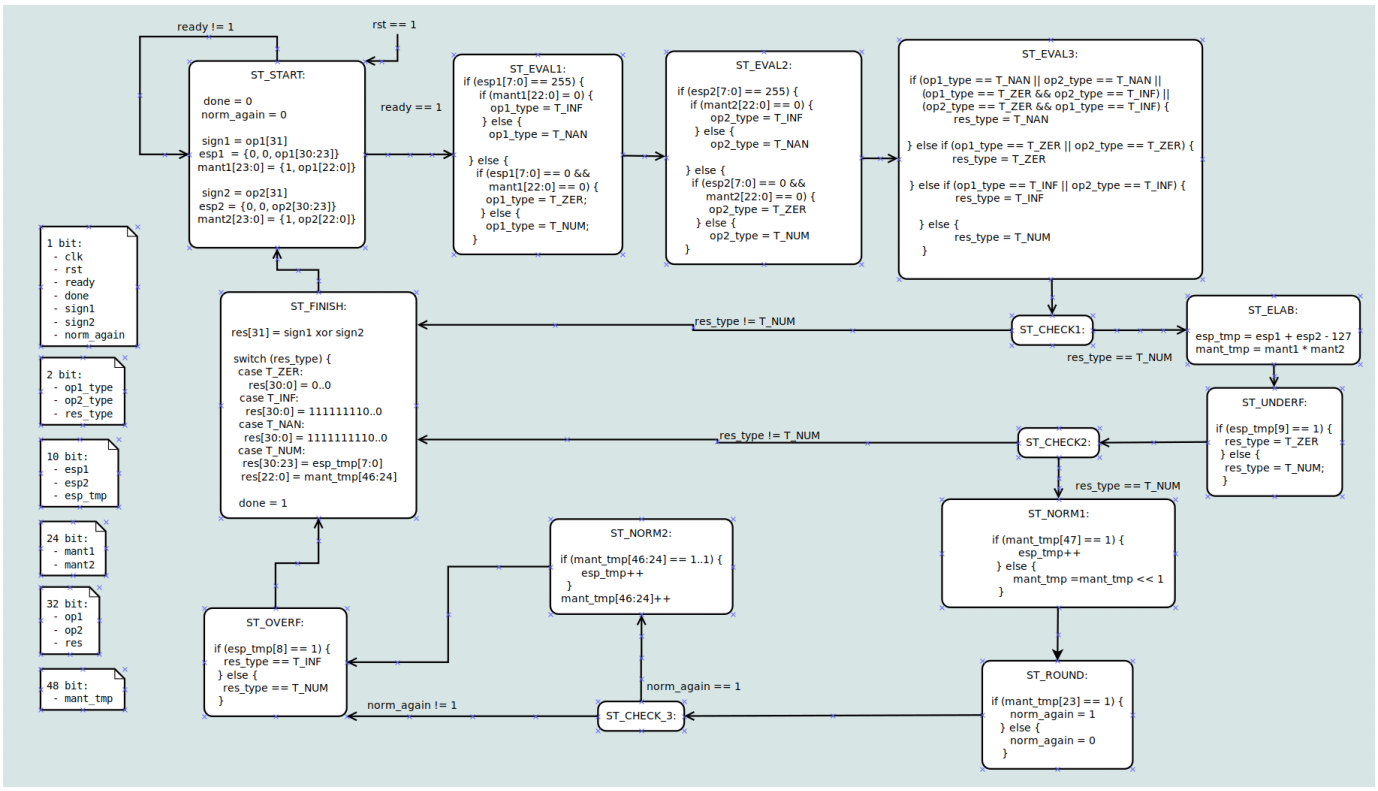

Figura 5. EFSM del multiplier

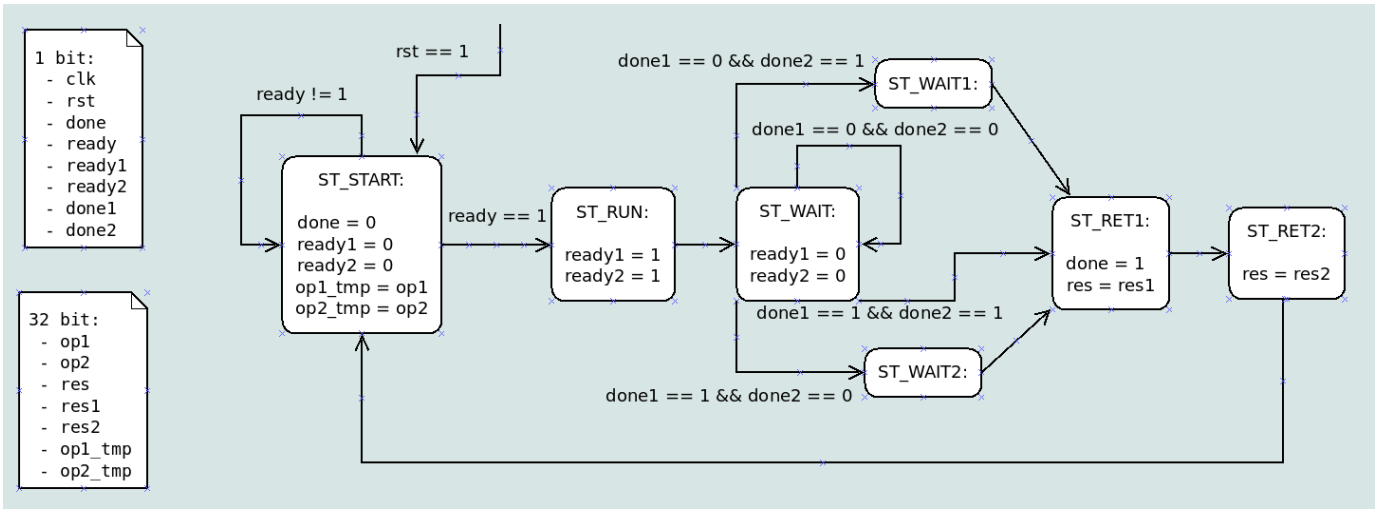

Figura 6. EFSM del double_multiplier

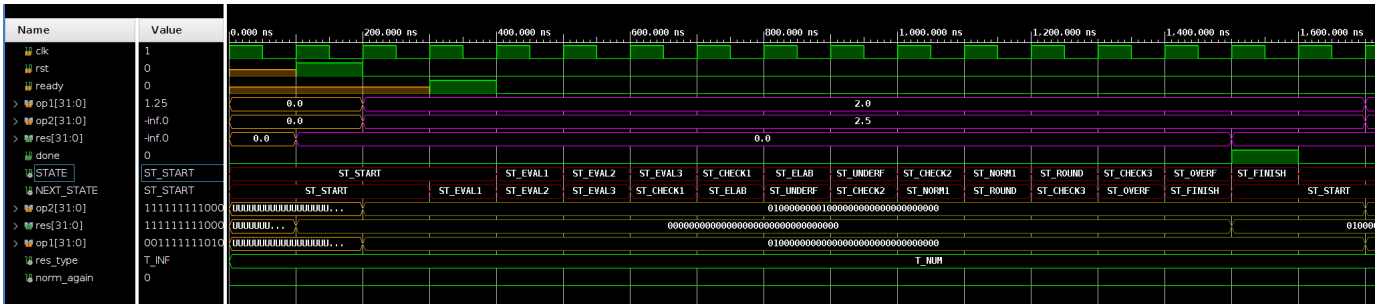

Figura 7. Simulazione multiplier in VHDL con script TCL con più zoom

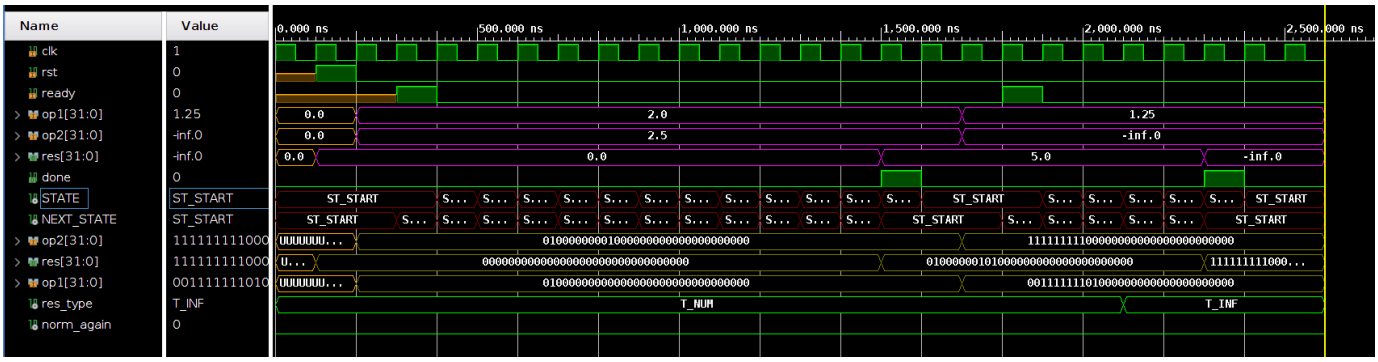

Figura 8. Simulazione multiplier in VHDL con script TCL

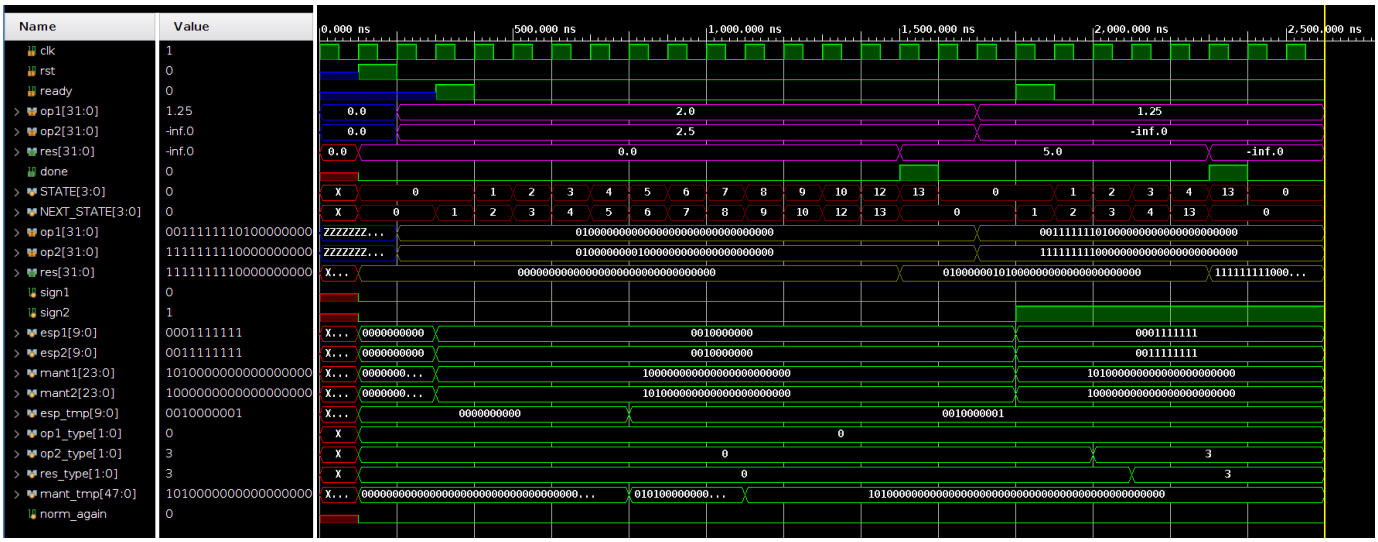

Figura 9. Simulazione multiplier in Verilog con script TCL

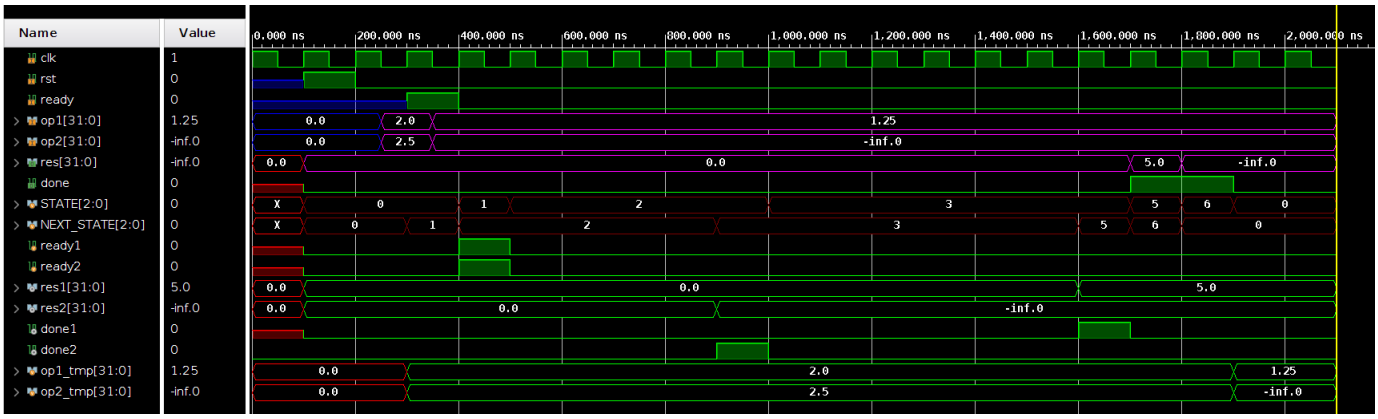

Figura 10. Simulazione double_multiplier in Verilog con script TCL

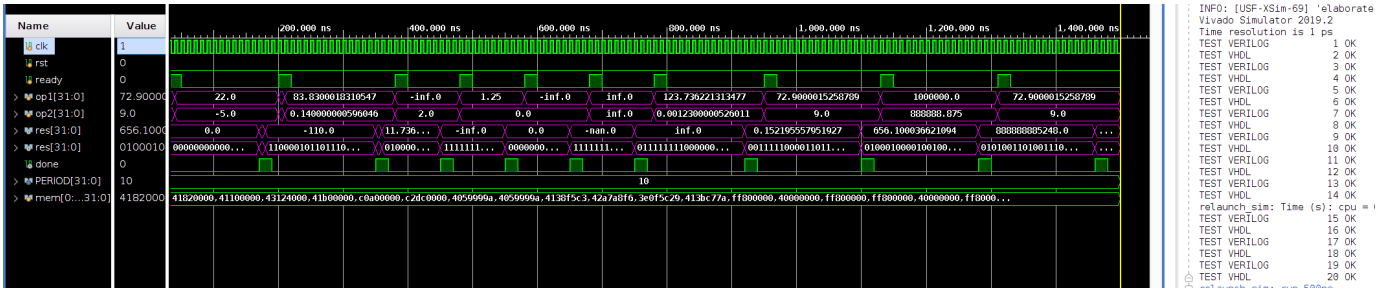

Figura 11. Simulazione double_multiplier in Verilog con testbench

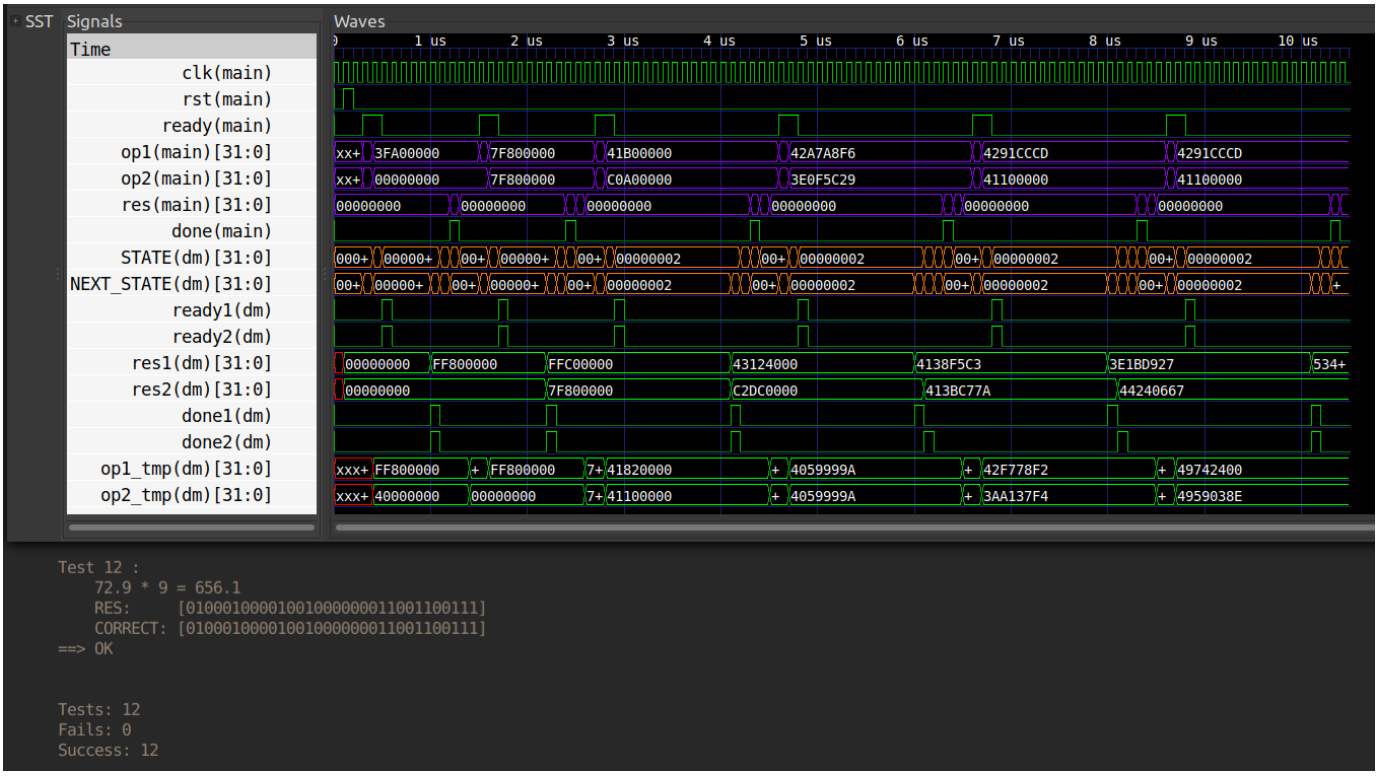

Figura 12. Simulazione double_multiplier in SystemC con “targeted test”

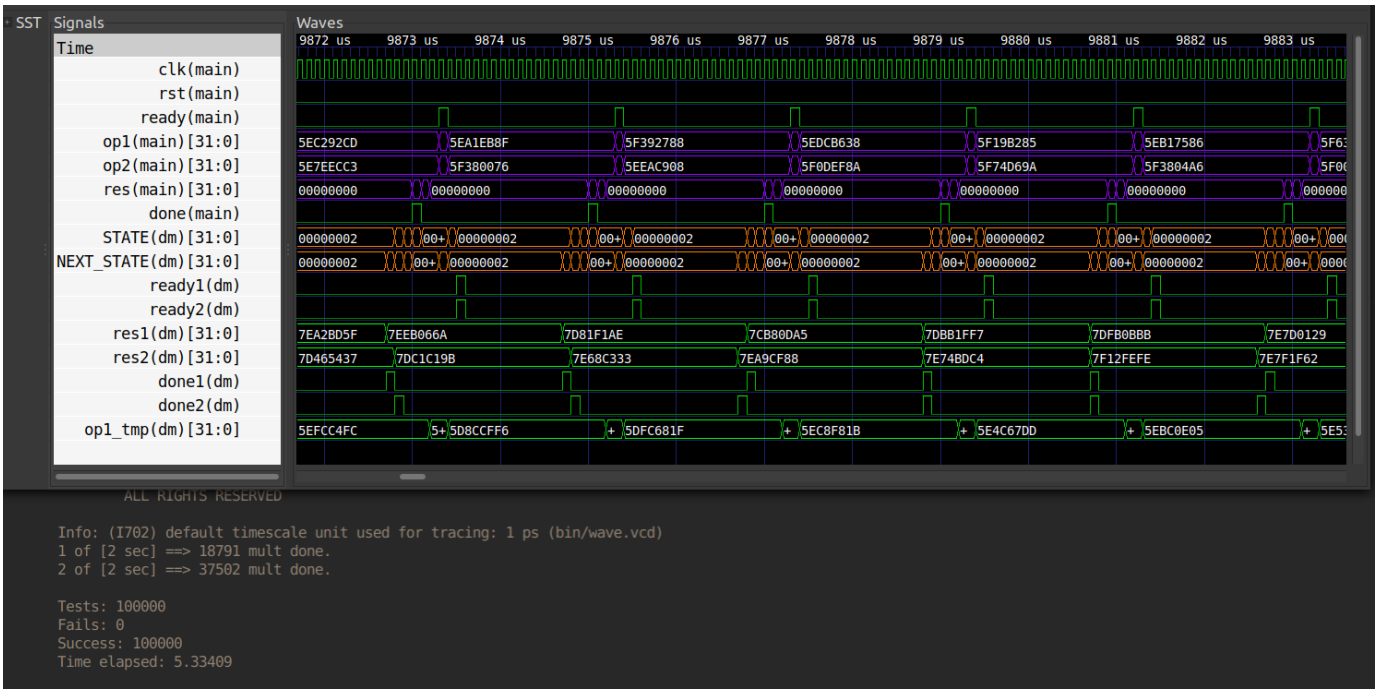

Figura 13. Simulazione double_multiplier in SystemC con “random test”