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 utilizzi due moltiplicatori in virgola mobile a precisione singola, secondo lo standard IEEE754, per eseguire due moltiplicazioni in parello. Esso deve essere sintetizzabile sulla 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 scoprirne le differenze e capirne i pro e contro.
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 intermedi 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 del 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[3] (**HLS**) per ottenere una descrizione dell'hardware a livello RT.
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

Lo standard 754[4] è la rappresentazione più comune per i numeri reali. Esso definisce il **formato** per la rappresentazione dei numeri in virgola mobile (compreso ± 0 e i numeri denormalizzati (o subnormali), gli infiniti e i NaN, "not a number"), ed un set di operazioni effettuabili su questi [5].
La rappresentazione in virgola fissa ha una "finestra" di rappresentazione che gli impedisce di rappresentare sia numeri molto grandi che molto piccoli. Invece la rappresentazione in virgola mobile utilizza una sorta di "finestra scorrevole" di precisione adeguata alla scala del numero permettendogli di massimizzare la precisione su entrambe le estremità della scala [6].
La versione a precisione singola dello standard IEEE754 descrive il numero con 32 bit: 1 bit per segno (sign), 8 bit per l'esponente (exp) e 23 bit per la mantissa (mant).
Per la **codifica** in numero binario normalizzato:

- Il bit del segno *sign* è 1 se il numero è negativo 0 altrimenti.
- Si converte il numero in binario in virgola fissa.
- Si sposta la virgola a sinistra o destra fino ad avere un numero nella forma $1.x \cdot 2^E$.
- I bit della mantissa *mant* sono la parte a destra della virgola, con zeri a destra fino a riempire i 23 bit. Il bit a 1 a sinistra della virgola è omesso.
- I bit dell'esponente *exp* sono uguali a $127 + E$ dove 127 è il *bias* di questo standard per la versione a precisione singola.

Per la **decodifica** del numero binario normalizzato:

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

Ci sono però dei casi particolari rappresentati in modo diverso:

- **zero:** è rappresentato mettendo tutti i bit di esponente e mantissa a 0.
- **infinito:** è rappresentato mettendo tutti i bit dell'esponente a 1 e quelli della mantissa a 0.
- **NaN:** sono rappresentati mettendo tutti i bit dell'esponente a 1, ma non hanno tutti i bit della mantissa a 0. Essi vengono utilizzati per rappresentare un valore che non rappresenta un numero reale. Esistono due categorie di NaN:

  - **Quiet NaN:** esso ha il bit più significativo della mantissa a 1. Indica un valore indeterminato generato da operazioni aritmetiche il cui risultato non è matematicamente definito.
  - **Signal NaN:** esso ha il bit più significativo della mantissa a 0. Indica un valore non valido. Può essere utilizzato per segnalare eccezioni causate da operazioni o per indicare variabili non inizializzate.
- **Numeri denormalizzati:** sono rappresentati mettendo tutti i bit dell'esponente a 0, ma non hanno tutti i bit della mantissa a 0. Essi sono decodificati in modo differente dai numeri normalizzati:

$$(-1)^{sign} \cdot 2^{(-126)} \cdot (0 + \sum_{i=1}^{23} b_{23-i} \cdot 2^{-i})$$

In particolare il bit omesso vale 0 anzichè 1 e l'esponente effettivo del numero vale sempre -126 [6].

Float Values (b = bias)

Sign	Exponent (e)	Fraction (f)	Value
0	00⋯00	00⋯00	+0
0	00⋯00	00⋯01 ⋮ 11⋯11	Positive Denormalized Real $0.f \times 2^{(-b+1)}$
0	00⋯01 ⋮ 11⋯10	XX⋯XX	Positive Normalized Real $1.f \times 2^{(e-b)}$
0	11⋯11	00⋯00	+∞
0	11⋯11	00⋯01 ⋮ 01⋯11	SNaN
0	11⋯11	1X⋯XX	QNaN
1	00⋯00	00⋯00	−0
1	00⋯00	00⋯01 ⋮ 11⋯11	Negative Denormalized Real $-0.f \times 2^{(-b+1)}$
1	00⋯01 ⋮ 11⋯10	XX⋯XX	Negative Normalized Real $-1.f \times 2^{(e-b)}$
1	11⋯11	00⋯00	−∞
1	11⋯11	00⋯01 ⋮ 01⋯11	SNaN
1	11⋯11	1X⋯XX	QNaN

Figura 1: IEEE 754 casi possibili. (Steve Hollasch, Online [6])

III. Metodologia applicata

A. Struttura progetto

- **cpp/** contiene il file **multiplier.cpp** dove è presente una funzione che esegue la moltiplicazione in c++.
- **SystemC/**
  - **Makefile:** tool per la compilazione automatica del progetto. Richiede che la variabile d'ambiente SYSTEMC_HOME contenga il path alla libreria di SystemC.
  - **include:** contiene gli headers del progetto. Qui sono definite tutte le porte, segnali ed enumerazioni dei vari componenti.
  - **src:** contiene i file sorgenti del progetto.
  - **bin:** contiene l'eseguibile **double_multiplier_RLT.x** (generato dopo la compilazione) e la simulazione **wave.vcd** (generata dopo l'esecuzione dell'eseguibile).
  - **obj:** contiene i file oggetto (generati dopo la compilazione).
- **VHDL_verilog/**
  - **stimuli/** contiene gli script TCL usati per dei test.
  - **waves/** contiene i file per visualizzare le simulazioni come negli screenshot.
  - **constrains/** contiene i vincoli per l'implementazione su FPGA.
  - **double_multiplier** file Verilog del top level del sistema.
  - **verilog_multiplier** file Verilog del moltiplicatore IEEE754.
  - **vhdl_multiplier** file VHDL del moltiplicatore IEEE754.
  - **testbench** file Verilog del testbench.

B. Procedimento

Per prima cosa è stato studiato lo standard IEE754 ed è stato definito l'algoritmo ad alto livello per la moltiplicazione. Dopodichè è stata realizzata la EFSM di *double_multiplier* e *multiplier*.
A questo punto si è passati alla realizzazione su Vivado[7] dei componenti hardware, partendo con un approccio bottom-up dal *multiplier* per poi passare al *double_multiplier* e infine al *testbench* tutti scritti in Verilog. È stato scelto questo linguaggio per realizzare le varie componenti in quanto ritenuto più semplice da utilizzare e con una sintassi molto più chiara del VHDL. Ogni componente è stato testato con uno script TCL e in seguito con il testbench. Verificata la correttezza del sistema con due moltiplicatori Verilog si è passati a scrivere *multiplier* anche in VHDL ed è stato aggiunto al testbench il confronto fra i valori ottenuti dai due componenti.
In seguito è stato riscritto tutto in SystemC dove è stato fatto anche un testbench con numeri random.
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.
- 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.
I vari componenti comunicano fra loro grazie al protocollo di *handshake*: con *ready* uguale a 1 si attiva il componente, il quale quando avrà finito metterà *done* uguale a 1.

L'architettura con VHDL e Verilog è mostrata in figura 2. Quella per SystemC è analoga.
I segnali intermedi sono stati omessi da questa figura, ma vengono descritti nelle sezioni successive.

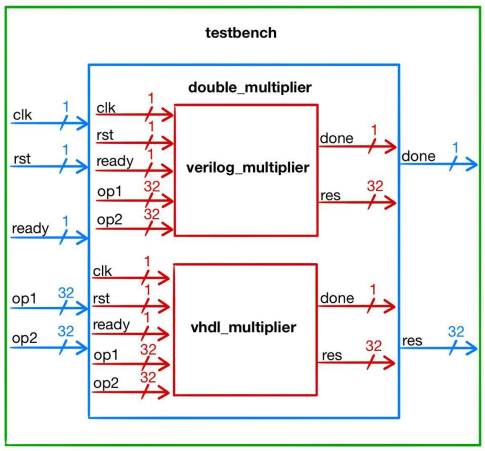


Figura 2: Architettura RTL

Le FSMD realizzate usano tutte 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.

D. multiplier

Questo componente esegue la moltiplicazione tra numeri floating point a precisione singola.
L'interfaccia è mostrata in figura 2:

- **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 al prossimo fronte di salita del clock *op1* e *op2* conterranno i valori degli operandi.
- **done** (1 bit output): segnale che indica che il valore su *res* è il risultato.
- **res** (32 bit output): risultato.

Gli altri segnali intermedi utilizzati sono:

- **STATE e NEXT_STATE** (5 bit): rappresentano lo stato attuale e lo stato prossimo.
- **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** (1 bit): rappresentano il segno dei due operandi.
- **esp1, esp2** (8 bit): rappresentano gli esponenti dei due operandi
- **mant1, mant2** (24 bit): rappresentano le mantisse dei due operandi. Esse presentano anche il bit omesso che sarà 1 nel caso di operandi normalizzati, 0 altrimenti.

L'algoritmo della moltiplicazione è descritto grazie alla EFSM [Figura 20] la quale è formata da 19 stati:

- **ST_START:** stato di partenza. Qui vengono resettati i segnali interni e gli output a zero. In caso di segnale *reset* a 1 si torna in questo stato. In caso di segnale *ready* a 1 si passa a *ST_INIT*.
- **ST_INIT:** qui vengono estrapolate le informazioni di segno, esponente e mantissa, la quale presenta anche il bit 24 a 1, degli operandi. Avviene anche la valutazione dei casi speciali, ovvero la presenza di NaN, ∞, 0 o numeri subnormali che porteranno nei rispetti stati. In caso il numero sia normalizzato si va allo stato *ST_ELAB*.
- **ST_SNAN1:** si arriva a questo stato se il primo operando è un Signal NaN. In concordanza col metodo di testing e con l'articolo [8]: "Un signal NaN può essere copiato senza problemi, ma qualsiasi altra operazione è invalida e deve essere intercettata o prodotto un Nan non signal", *res* avrà il valore di *op1* con il bit più significativo della mantissa a 1. Poi si passa a *ST_FINISH*.
- **ST_SNAN2:** si arriva a questo stato se il secondo operando è un Signal NaN. Analogamente allo stato *ST_SNAN1*, viene riscritto su *res* il valore di *op2* con il bit più significativo della mantissa a 1. Poi si passa a *ST_FINISH*. In caso ci siano due SNAN come operandi è stato deciso di dare priorità a questo stato.
- **ST_QNAN:** si arriva a questo stato se gli operandi sono 0 e ∞. In questo caso i primi 10 bit di *res* sono posti a 1 e gli altri a 0. Poi si passa a *ST_FINISH*.
- **ST_ZERO:** si arriva a questo stato se un operando è 0 e l'altro non è ∞. In questo caso il bit più significativo di *res* è dato dallo xor del segno dei due operandi e gli altri bit sono posti a 0. Poi si passa a *ST_FINISH*.
- **ST_INF:** si arriva a questo stato se un operando è ∞ e l'altro non è 0. In questo caso il bit più significativo di *res* è dato dallo xor del segno, quelli dell'esponente sono posti tutti a 1 e quelli della mantissa a 0. Poi si passa a *ST_FINISH*.
- **ST_ADJ1:** si arriva a questo stato se il primo operando è un numero subnormale e l'altro è normalizzato. In questo caso si pone il bit 24 di *mant1* a 0 ed *exp1* uguale a 1 perchè $1 - 127 = -126$ cioè il valore corretto in caso di numero subnormale. Questa scelta è stata fatta per utilizzare lo stato *ST_ELAB* indipendentemente che il numero fosse normalizzato o subnormalizzato. Poi si passa a *ST_ELAB*.
- **ST_ADJ2:** analogo a *ST_ADJ2* ma qui è *op2* subnormale.
- **ST_ADJ3:** analogo a *ST_ADJ1* e *ST_ADJ2* ma qui entrambi gli operandi sono subnormali.
- **ST_ELAB:** qui viene ricavato l'esponente temporaneo come $exp\_tmp = exp1 + exp2 - 127$. Si sottrae 127 perchè altrimenti *exp_tmp* sarebbe composto dall'esponente reale e la somma di due bias cioè $127 +$

127. Viene anche ricavata la mantissa temporanea come $mant\_tmp = mant1 * mant2$. Poi si passa a *ST_SHIFTR* se il bit più significativo di *mant_tmp* vale 1, a *ST_SHIFTL* se i due bit più significativi sono "00" oppure *ST_CHECK* se sono "01".

- **ST_SHIFTR:** qui si esegue lo shift a destra di *mant_tmp* per portarla nella forma 1.x, con conseguente incremento di *exp_tmp*. Poi si passa a *ST_CHECK*.
- **ST_SHIFTL:** qui si esegue lo shift a sinistra di *mant_tmp* nella speranza di portarla nella forma 1.x, con conseguente decremento di *exp_tmp*. Poi si passa a *ST_NORM*.
- **ST_NORM:** se *mant_tmp* è diventato nella forma 1.x allora si passa a *ST_CHECK*. Se *exp_tmp* è maggiore di 0 allora c'è ancora speranza di riuscire a ottenere un numero nella forma 1.x e quindi si ritorna a *ST_SHIFTL*, altrimenti si passa a *ST_CHECK* per valutare se è possibile rappresentare il risultato con un numero subnormale.
- **ST_CHECK:** qui si controlla ciò che si è ottenuto dalle elaborazioni precedenti:
  Se *exp_tmp* ha tutti i bit a zero allora il numero è subnormale e si passa a *ST_SUBNORM*. Se *exp_tmp* ha i due bit più significativi a "01" allora c'è stato overflow e si passa a *ST_INF*. Se *exp_tmp* ha i due bit più significativi a "00" e il bit 22 di *mant_tmp* a 0 allora il risultato è un numero normalizzato e si passa a *ST_WRITE*. Se *exp_tmp* ha i due bit più significativi a "00" e il bit 22 di *mant_tmp* a 1 allora c'è bisogno di arrontare la mantissa e si passa a *ST_ROUND*. Se il bit più significativo di *exp_tmp* è 1 e la somma dell'esponente temporaneo con 21 è maggiore o uguale a 0 allora il risultato è rappresentabile con un numero subnormale, ma non è ancora pronto quindi si passa a *ST_SHIFTR* per portarlo alla forma giusta. Nel caso pessimo infatti si avrebbe *mant_tmp* 00.10...0 (i primi due bit sono zero perchè altrimenti il numero sarebbe stato normalizzato e si sarebbe andati in un altro stato) e quindi il numero massimo di shift a destra necessari per riportare *exp_tmp* a 0 senza che i 23 bit a destra della virgola siano tutti 0 (caso di underflow) è 22, al quale va tolto 1 in quanto c'è un altro shift a destra inevitabile nello stato successivo. Se invece il bit più significativo di *exp_tmp* è 1 e la somma dell'esponente temporaneo con 21 è minore di 0 allora si è verificato un underflow e si passa a *ST_ZERO*.
- **ST_SUBNORM:** si arriva a questo stato quando il risultato è un numero subnormale. Analogamente a quanto detto per *ST_ADJ1*, l'obiettivo è arrivare ad avere l'esponente reale -126 che è ottenuto da $1 - 127$. Per questo motivo bisogna compiere un ulteriore shift a destra della *mant_tmp*, mentre l'esponente va tenuto a 0 in quanto è già nella forma corretta dell'esponente per i numeri subnormali. Poi si passa a *ST_WRITE*.
- **ST_ROUND:** qui avviene l'arrotondamento della mantissa, che avviene solo se il bit a destra del punto di taglio di *mant_tmp* è 1. In questo caso si incrementa di 1 la parte a sinistra del punto di taglio della mantissa temporanea. nel caso in cui questo incremento faccia diventare il bit più significivo di *mant_tmp* a 1 si passa a *ST_SHIFTR* altrimenti si prosegue in *ST_WRITE*. Ci sono vari metodi di arrotondamento possibili, questo è stato scelto perchè molto semplice.
- **ST_WRITE:** qui si assembla il risultato finale *res*. In particolare il segno è dato dallo xor fra il segno degli operandi, l'esponente è dato da *exp_tmp* senza i due bit più significativi e la mantissa da *mant_tmp* senza i due bit più significativi e con solo i 23 successivi.
- **ST_FINISH:** qui si pone *done* a 1.

E. double_multiplier

Questo componente esegue due moltiplicazioni tra numeri floating point a precisione singola.
L'interfaccia è mostrata in figura 2:

- **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 al prossimo fronte di salita del clock *op1* e *op2* conterranno i valori degli operandi per la prima moltiplicazione e in quello 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 intermedi utilizzati sono:

- **STATE e NEXT_STATE** (3 bit): rappresentano lo stato attuale e lo stato prossimo.
- **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.

L'algoritmo è descritto grazie alla EFSM [Figura 21] la quale è formata 8 stati:

- **ST_START:** stato di partenza. Qui vengono resettati i segnali interni e gli output a zero. In caso di segnale *reset* a 1 si torna in questo stato. In caso di segnale *ready* a 1 si passa a *ST_RUN1*.
- **ST_RUN1:** qui viene posto *ready1* a 1, attivando quindi il primo moltiplicatore. Poi si passa a *ST_RUN2*.
- **ST_RUN2:** qui viene posto *ready2* a 1, attivando quindi il secondo moltiplicatore. Poi si passa a *ST_WAIT*.
- **ST_WAIT:** qui vengono posti *ready1* e *ready2* uguali a 0. Si rimane in questo stato finchè *done1* e *done2* rimangono a 0. Se *done1* diventa 1 allora si passa a *ST_WAIT*2, se *done2* diventa 1 allora si passa a *ST_WAIT1* e nel caso si attivino contemporaneamente allora si passa direttamente a *ST_RET1*.

- **ST_WAIT1:** si resta qui finchè non finisce anche il primo moltiplicatore, cioè finchè *ready1* è uguale a 0. Poi si passa a *ST_RET1*.
- **ST_WAIT2:** si resta qui finchè non finisce anche il secondo moltiplicatore, cioè finchè *ready2* è uguale a 0. Poi si passa a *ST_RET1*.
- **ST_RET1:** pone *done* uguale a 1 e *res* uguale al risultato del primo moltiplicatore cioè *res1*. Poi si passa a *ST_RET2*.
- **ST_RET2:** pone *res* uguale al risultato del secondo moltiplicatore cioè *res2* e ritorna allo stato iniziale.

F. testbench

Questo componente è un test automatizzato che incorpora il *double_multiplier* come mostrato in figura 2.
La versione in Verilog ha il compiti di verificare:

- che i sottocomponenti in VHDL e Verilog diano gli stessi risultati.
- che il *double_multiplier* esegua correttamente due moltiplicazioni con operandi diversi.
- che vengano provati per ogni nodo della EFSM tutti gli archi.

Nella versione in SystemC sono state messe a disposizione tre thread (da attivare togliendo i commenti del costruttore del *TestbenchModule* nel file testbench_RTL.cc):

- **target_test:** testa l'esecuzione di due specifici operandi scelti da un array.
- **full_target_test:** testa tutti gli elementi dell'array, i quali sono scelti appositamente per provare tutti gli archi di ogni nodo della EFSM.
- **rnd_test:** testa l'esecuzione del *double_multiplier* con valori generati randomicamente.

Per i testbench in entrambi i linguaggi è stato usato un array di bits contenenti gli inputs da testare. In SystemC sono state create due funzioni ausiliarie "binary_to_float()" e "float_to_binary()" per confrontare il risultato ottenuto dal *double_multiplier* con quelli ottenuti da una semplice moltiplicazione in c++. I risultati ottenuti sono stati inseriti nell'array del testbench in Verilog per realizzare il test automatico.

IV. RISULTATI

A. Simulazioni con script TCL

Dopo aver realizzato il *multiplier* in Verilog, questo è stato subito testato con un semplice script TCL. In questo modo si è potuto controllare facilmente che il componente esegua i passaggi aspettati. In particolare sono stati ricercati e provati gli input necessari a testare ogni arco della EFSM. Nel file "/stimuli/multiplier.tcl" si possono vedere i vari input testati, con gli stati percorsi e il risultato obiettivo ricavato grazie alle funzioni ausiliarie del testbench SystemC. Nelle figure 22 e 23 si possono guardare le simulazioni dei moltiplicatori con lo script TCL.
Lo script TCL per il *double_multiplier* analogamente è stato usato per controllare il corretto funzionamento del componente e in particolare per verificare che i moltiplicatori ricevano i giusti input. In figura 24 si può guardare la simulazione con lo script TCL.

B. Simulazione con testbench in Verilog

I test con gli script TCL sono serviti principalmente a controllare nel dettaglio il comportamento dei componenti e a ricavare gli input per questo testbench.
Questo test ha due compiti:

- Verificare che il *multiplier* VHDL dia gli stessi valori di quello Verilog.
- Verificare che il *double_multiplier* esegua correttamente due moltiplicazioni diverse.

In particolare è utilizzato per capire in modo automatico se i componenti fanno ciò che ci si aspetta, restituendo "OK" in caso il test sia andato a buon fine e FAIL, con il risultato ottenuto e quello corretto, in caso di errore. In figura 25 si può vedere un pezzo della simulazione con questo testbench a livello Behavioral. In figura 26 si può vedere un pezzo della "post-implementation timing simulation" cioè la simulazione più simile al comportamento effettivo sulla FPGA e in figura 27 i risultati ottenuti che sono gli stessi in entrambe le simulazioni.

C. Simulazione con testbench in SystemC

In questo testbench sono stati messi a disposizione tre tipi di test differenti.
Il primo, mostrato in figura 28 ha lo scopo di testare semplicemente due input. Questo è stato utilizzato per ricavare il risultato in bit da utilizzare per il testbench in Verilog e analogamente allo script TCL, per permettere di controllare che il sistema percorra gli stati corretti.
Il secondo, mostrato in figura 29 è analogo al testbench in Verilog e testa tutti gli input selezionati appositamente e disposti in un array. Ha il compito di testare la correttezza del sistema con gli input ritenuti più significativi.
Il terzo, mostrato in figura 30 ha lo scopo di dare una maggiore sicurezza sulla correttezza del sistema e in particolare trovare operandi che causano moltiplicazioni errate, dovute a casi non tenuti in considerazione durante la progettazione della EFSM. Come mostrato in figura 31 il test fallisce in alcuni casi a causa del metodo di arrotondamento utilizzato. Essendoci però più metodi di arrotondamento usabili, è stato deciso di tenere questo in quanto molto semplice.

D. Sintesi double_multiplier

La sintesi è stata fatta per la scheda Pynq xc7z020clg400-1. In figura 3 sono mostrati i valori riguardanti le risorse di area utilizzate nella post-syntesis del *double_multiplier*.

Utilization — Post-Synthesis | Post-Implementation

Graph | Table

Resource	Estimation	Available	Utilization %
LUT	500	53200	0.94
FF	362	106400	0.34
DSP	4	220	1.82
IO	100	125	80.00
BUFG	1	32	3.13

Figura 3: Valori di utilizzo risorse post-syntesis del *double_multiplier*

Il periodo di clock minimo raggiungibile prima che WNS diventi negativo (e che quindi avvengano violazioni) è 8ns. Con questo valore la simulazione post-implementation funzionale funziona, ma quella post-implementation del tempo da valori sbagliati. Per questo motivo è stato usato come periodo di clock minimo 10ns.
Nelle figure 4, 5 e 6 sono mostrati i valori riguardanti le risorse di tempo utilizzate post-implementation che risultato essere peggiori con l'aumentare del periodo di clock.

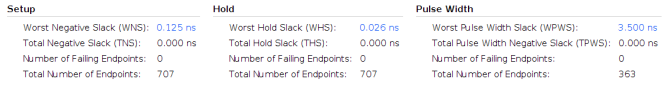


Figura 4: Valori di timing del *double_multiplier* con periodo minimo di 8ns

Figura 5: Valori di timing del *double_multiplier* con periodo minimo di 9ns

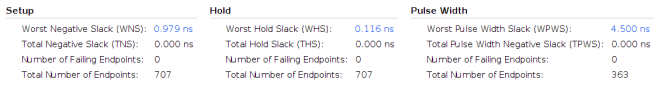


Figura 6: Valori di timing del *double_multiplier* con periodo minimo di 10ns

Nelle figure 7, 8 e 9 sono mostrati i valori riguardanti le risorse di area utilizzate post-implementation. Si nota subito che il numero di LUT utilizzate aumenta con il diminuire del periodo di clock scelto. Ciò sottolinea il fatto che per guadagnare in tempo si perde in area

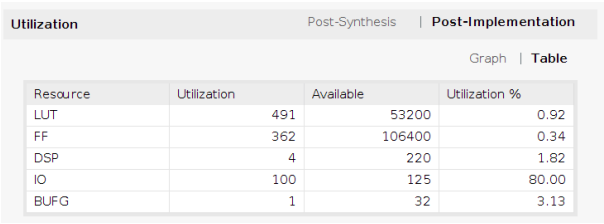


Figura 7: valori di utilizzo risorse post-implementation del *double_multiplier* con periodo minimo di 8ns

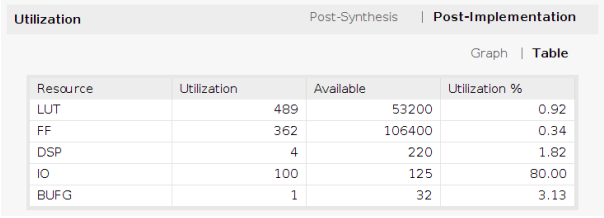


Figura 8: valori di utilizzo risorse post-implementation del *double_multiplier* con periodo minimo di 9ns

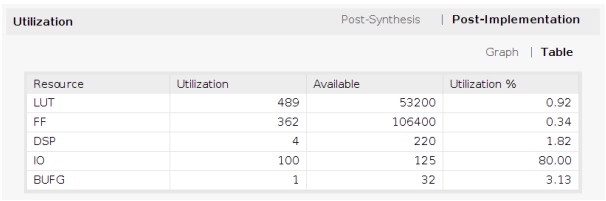


Figura 9: valori di utilizzo risorse post-implementation del *double_multiplier* con periodo minimo di 10ns

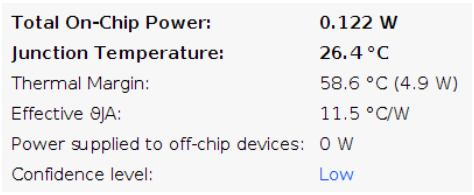


Figura 10: Power summary

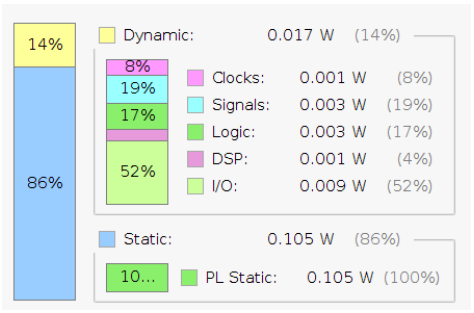


Figura 11: Power on-Chip

Nelle figure 10 e 11 si possono vedere i valori di potenza. Nella figura 12 si vede chiaramente che l'utilizzao della FPGA è molto ridotto, proprio come previsto.

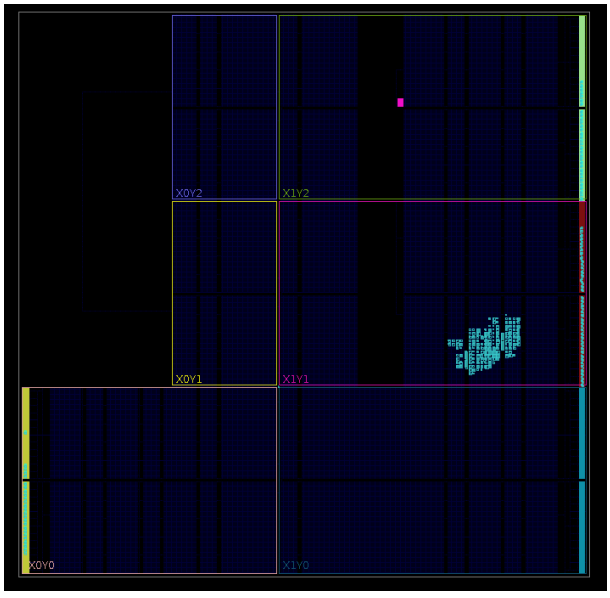
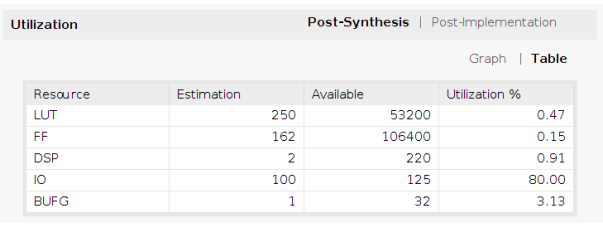


Figura 12: FPGA con *double_multiplier*

E. Sintesi multiplier e HLS

Nelle figure 13 e 14 sono mostrati i valori riguardanti le risorse di area utilizzate nella post-syntesis del *verilog_multiplier*. L'unica differenza fra le due sono le 4 LTU in meno dopo l'implementazione. In figura 15 sono mostrate le risorse di tempo utilizzate.

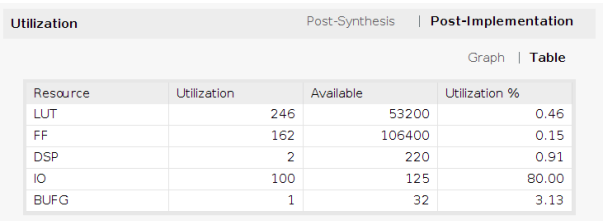


Utilization — Post-Synthesis | Post-Implementation

Graph | Table

Resource	Estimation	Available	Utilization %
LUT	250	53200	0.47
FF	162	106400	0.15
DSP	2	220	0.91
IO	100	125	80.00
BUFG	1	32	3.13

Figura 13: Valori di utilizzo risorse post-syntesis del *verilog_multiplier*

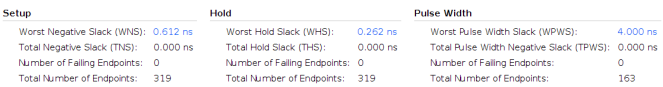


Utilization — Post-Synthesis | Post-Implementation

Graph | Table

Resource	Utilization	Available	Utilization %
LUT	246	53200	0.46
FF	162	106400	0.15
DSP	2	220	0.91
IO	100	125	80.00
BUFG	1	32	3.13

Figura 14: valori di utilizzo risorse post-implementation del *verilog_multiplier*



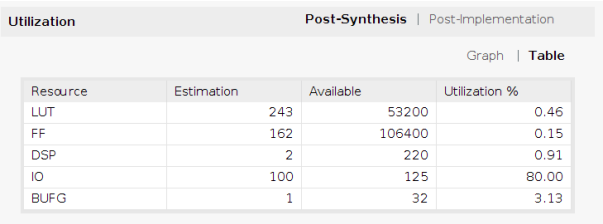
Setup
Worst Negative Slack (WNS): 0.612 ns
Total Negative Slack (TNS): 0.000 ns
Number of Failing Endpoints: 0
Total Number of Endpoints: 319

Hold
Worst Hold Slack (WHS): 0.262 ns
Total Hold Slack (THS): 0.000 ns
Number of Failing Endpoints: 0
Total Number of Endpoints: 319

Pulse Width
Worst Pulse Width Slack (WPWS): 4.000 ns
Total Pulse Width Negative Slack (TPWS): 0.000 ns
Number of Failing Endpoints: 0
Total Number of Endpoints: 163

Figura 15: Valori di timing del *verilog_multiplier*

Nelle figure 16 e 17 sono mostrati i valori riguardanti le risorse di area utilizzate nella post-syntesis del *vhdl_multiplier*. L'unica differenza fra le due sono le 6 LTU in meno dopo l'implementazione. In figura 18 sono mostrate le risorse di tempo utilizzate.



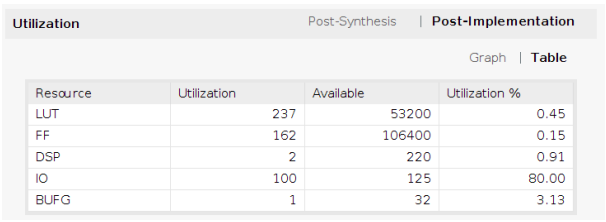
Utilization — Post-Synthesis | Post-Implementation

Graph | Table

Resource	Estimation	Available	Utilization %
LUT	243	53200	0.46
FF	162	106400	0.15
DSP	2	220	0.91
IO	100	125	80.00
BUFG	1	32	3.13

Figura 16: Valori di utilizzo risorse post-syntesis del *vhdl_multiplier*

Si nota subito come la versione VDHL dia risultati migliori sia a livello di area che di tempo rispetto a quella in Verilog, ciò fa capire che la scelta del linguaggio non è indifferente dal punto di vista delle performante.
In figura 19 sono riportati i risultati dell' High Level Syntesis. Sono state usate circa 100 LUT in più rispetto ai moltiplicatori in Verilog e VHDL, ma circa 10 FF in meno.



Utilization — Post-Synthesis | Post-Implementation

Graph | Table

Resource	Utilization	Available	Utilization %
LUT	237	53200	0.45
FF	162	106400	0.15
DSP	2	220	0.91
IO	100	125	80.00
BUFG	1	32	3.13

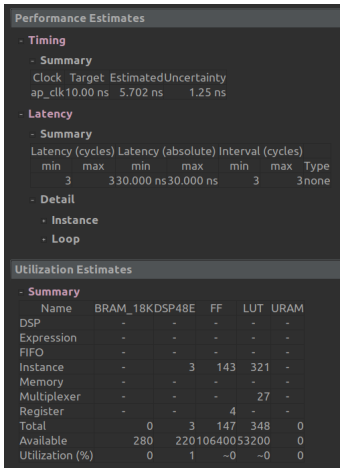
Figura 17: valori di utilizzo risorse post-implementation del *vhdl_multiplier*

Setup
Worst Negative Slack (WNS): 0.594 ns
Total Negative Slack (TNS): 0.000 ns
Number of Failing Endpoints: 0
Total Number of Endpoints: 319

Hold
Worst Hold Slack (WHS): 0.223 ns
Total Hold Slack (THS): 0.000 ns
Number of Failing Endpoints: 0
Total Number of Endpoints: 319

Pulse Width
Worst Pulse Width Slack (WPWS): 4.000 ns
Total Pulse Width Negative Slack (TPWS): 0.000 ns
Number of Failing Endpoints: 0
Total Number of Endpoints: 163

Figura 18: Valori di timing del *vhdl_multiplier*



Performance Estimates
- Timing
  - Summary

Clock	Target	EstimatedUncertainty
ap_clk	10.00 ns	5.702 ns	1.25 ns

- Latency
  - Summary

Latency (cycles)		Latency (absolute)		Interval (cycles)		
min	max	min	max	min	max	Type
3	3	30.000 ns	30.000 ns	3	3	none

  - Detail
    - Instance
    - Loop

Utilization Estimates
- Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	-	-	-
FIFO	-	-	-	-	-
Instance	-	3	143	321	-
Memory	-	-	-	-	-
Multiplexer	-	-	-	27	-
Register	-	-	4	-	-
Total	0	3	147	348	0
Available	280	220	106400	53200	0
Utilization (%)	0	1	~0	~0	0

Figura 19: High level syntesis con Vivado

V. Conclusioni

Il progetto è stato abbastanza impegnativo, ma stimolante. Ho imparato molte cose, tra cui la difficoltà di dimostrare la correttezza del sistema, più volte infatti sono dovuto ritornare sui miei passi, perchè ho inserito dei bug durante l'implementazione della EFSM o perchè non avevo pensato a determinati casi speciali. Sono soddisfatto dei risultati ottenuti dalla sintesi, anche se penso che si possa fare ancora meglio dal punto di vista del tempo. Sono poi rimasto particolarmente sorpreso dei risultati ottenuti con la HLS che pensavo sarebbero stati peggiori.

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] I. C. Society, "Ieee standard 754 for binary floating-point arithmetic," *Online*, 1985.
[5] "Ieee 754," https://en.wikipedia.org/wiki/IEEE_754.
[6] "Ieee 754 lectures notes," "http://steve.hollasch.net/cgindex/coding/ieeefloat.html".
[7] "Vivado," https://www.xilinx.com/products/design-tools/vivado.html.
[8] "Ieee 754 lectures notes," "http://li.mit.edu/Archive/Activities/Archive/CourseWork/Ju_Li/MITCourses/18.335/Doc/IEEE754/ieee754.pdf".

Appendice

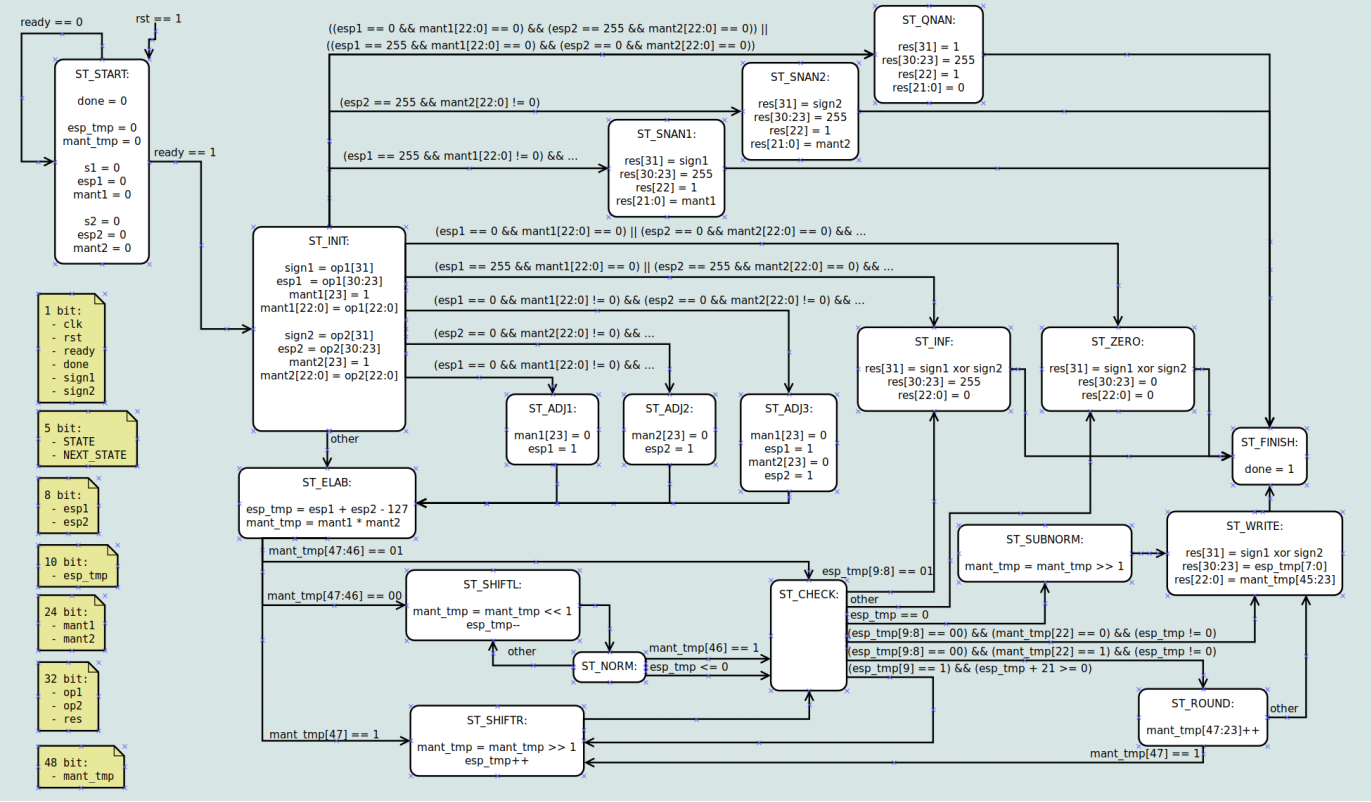


Figura 20: EFSM del multiplier

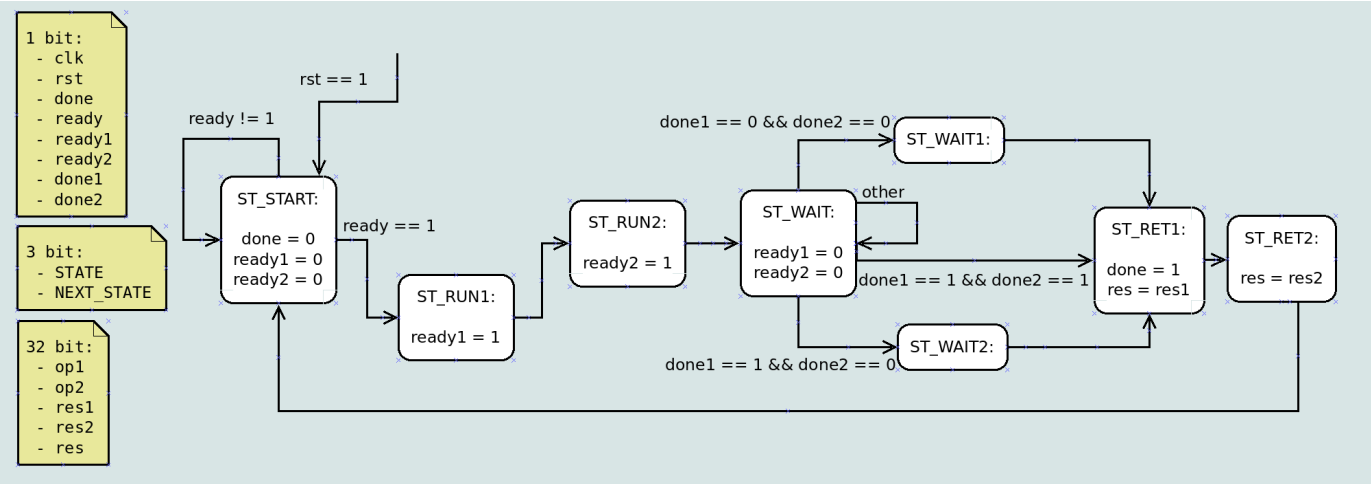


Figura 21: EFSM del double_multiplier (... indica che ci sono alcune condizioni implicite omesse per non appesantire troppo la rappresentazione)

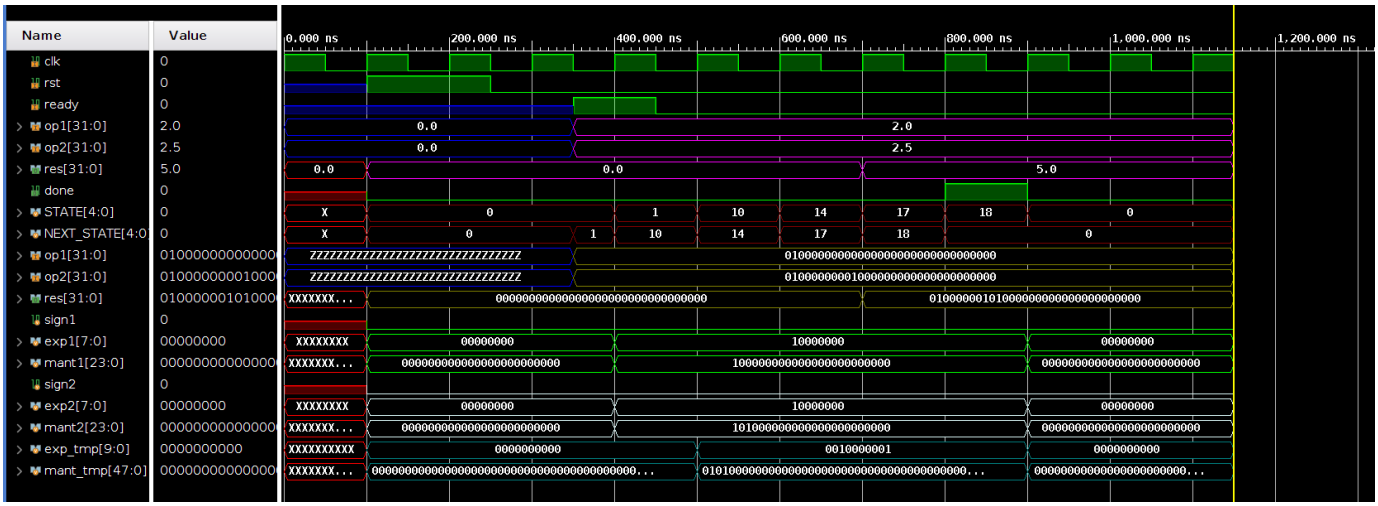


Figura 22: Simulazione multiplier in Verilog con script TCL

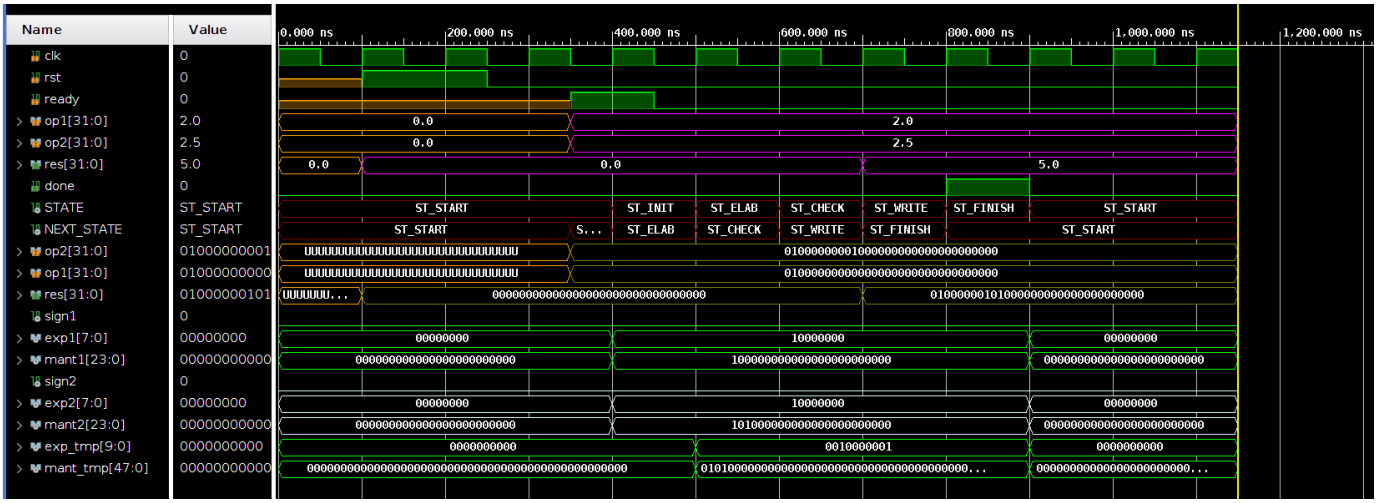


Figura 23: Simulazione multiplier in VHDL con script TCL

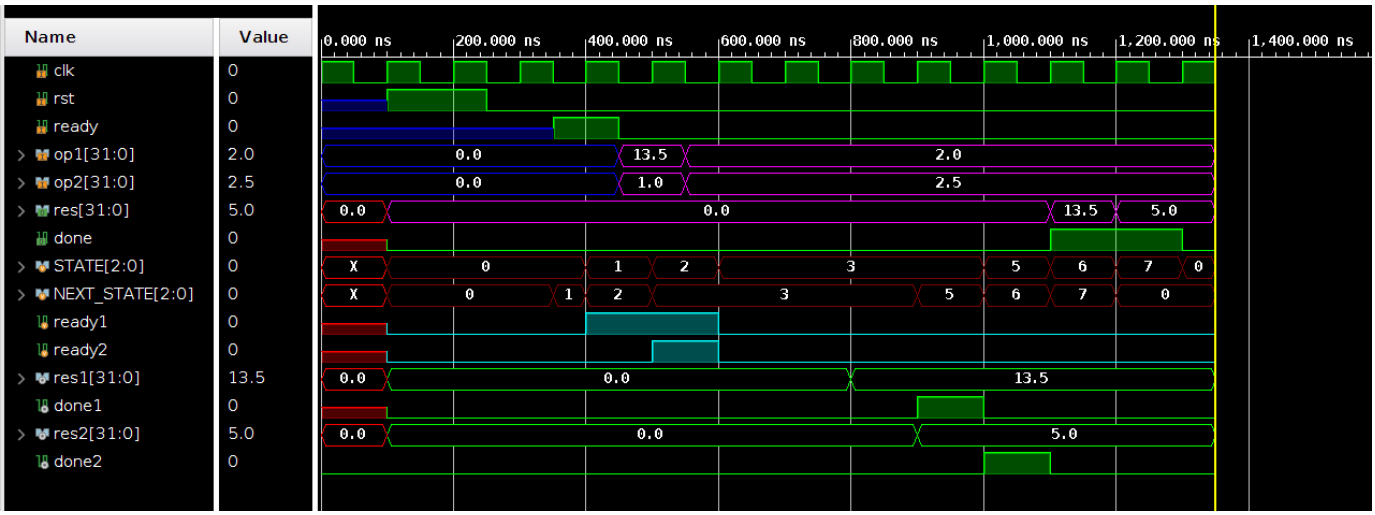


Figura 24: Simulazione double_multiplier in Verilog con script TCL

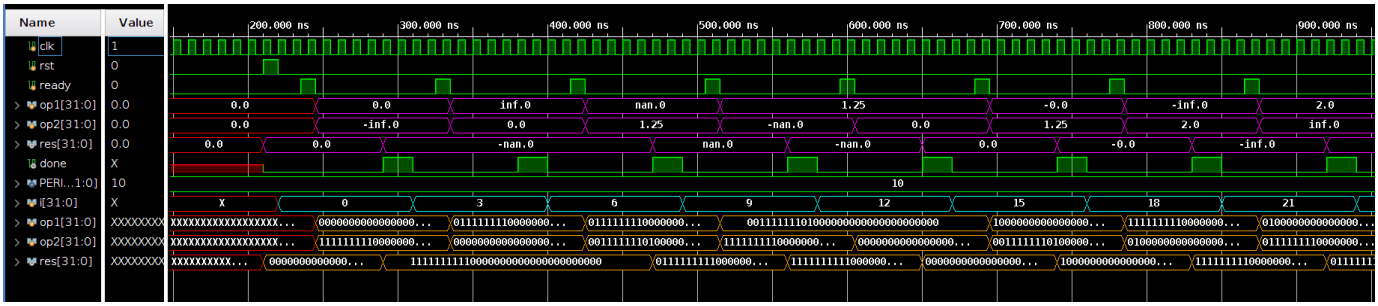


Figura 25: Behavioral simulation double_multiplier in Verilog con testbench

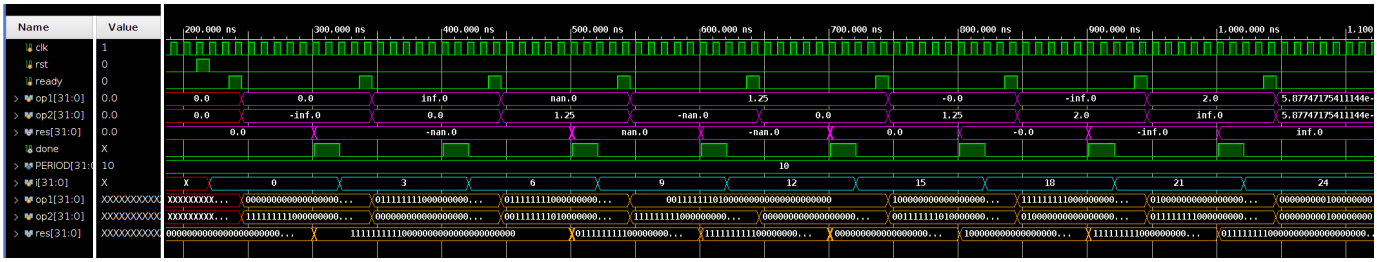


Figura 26: Post-Implementation Timing simulation double_multiplier in Verilog con testbench

```
Time resolution is 1 ps
################
VERILOG vs VHDL
TEST VERILOG           1 OK
TEST VHDL              1 OK
TEST VERILOG           2 OK
TEST VHDL              2 OK
TEST VERILOG           3 OK
TEST VHDL              3 OK
TEST VERILOG           4 OK
TEST VHDL              4 OK
TEST VERILOG           5 OK
TEST VHDL              5 OK
TEST VERILOG           6 OK
TEST VHDL              6 OK
TEST VERILOG           7 OK
TEST VHDL              7 OK
TEST VERILOG           8 OK
TEST VHDL              8 OK
relaunch_sim: Time (s): cpu =
TEST VERILOG           9 OK
TEST VHDL              9 OK
TEST VERILOG          10 OK
TEST VHDL             10 OK
TEST VERILOG          11 OK
TEST VHDL             11 OK
TEST VERILOG          12 OK
TEST VHDL             12 OK
TEST VERILOG          13 OK
TEST VHDL             13 OK
TEST VERILOG          14 OK
TEST VHDL             14 OK
TEST VERILOG          15 OK
TEST VHDL             15 OK
TEST VERILOG          16 OK
TEST VHDL             16 OK
TEST VERILOG          17 OK
TEST VHDL             17 OK
TEST VERILOG          18 OK
TEST VHDL             18 OK
TEST VERILOG          19 OK
TEST VHDL             19 OK
TEST VERILOG          20 OK
TEST VHDL             20 OK
```

(a) VHDL vs Verilog

```
################
VERILOG and VHDL
TEST VERILOG           1 OK
TEST VHDL              2 OK
TEST VERILOG           3 OK
TEST VHDL              4 OK
TEST VERILOG           5 OK
TEST VHDL              6 OK
TEST VERILOG           7 OK
TEST VHDL              8 OK
TEST VERILOG           9 OK
TEST VHDL             10 OK
TEST VERILOG          11 OK
TEST VHDL             12 OK
TEST VERILOG          13 OK
TEST VHDL             14 OK
TEST VERILOG          15 OK
TEST VHDL             16 OK
TEST VERILOG          17 OK
TEST VHDL             18 OK
TEST VERILOG          19 OK
TEST VHDL             20 OK
```

(b) VHDL and Verilog

Figura 27: Risultati testbench

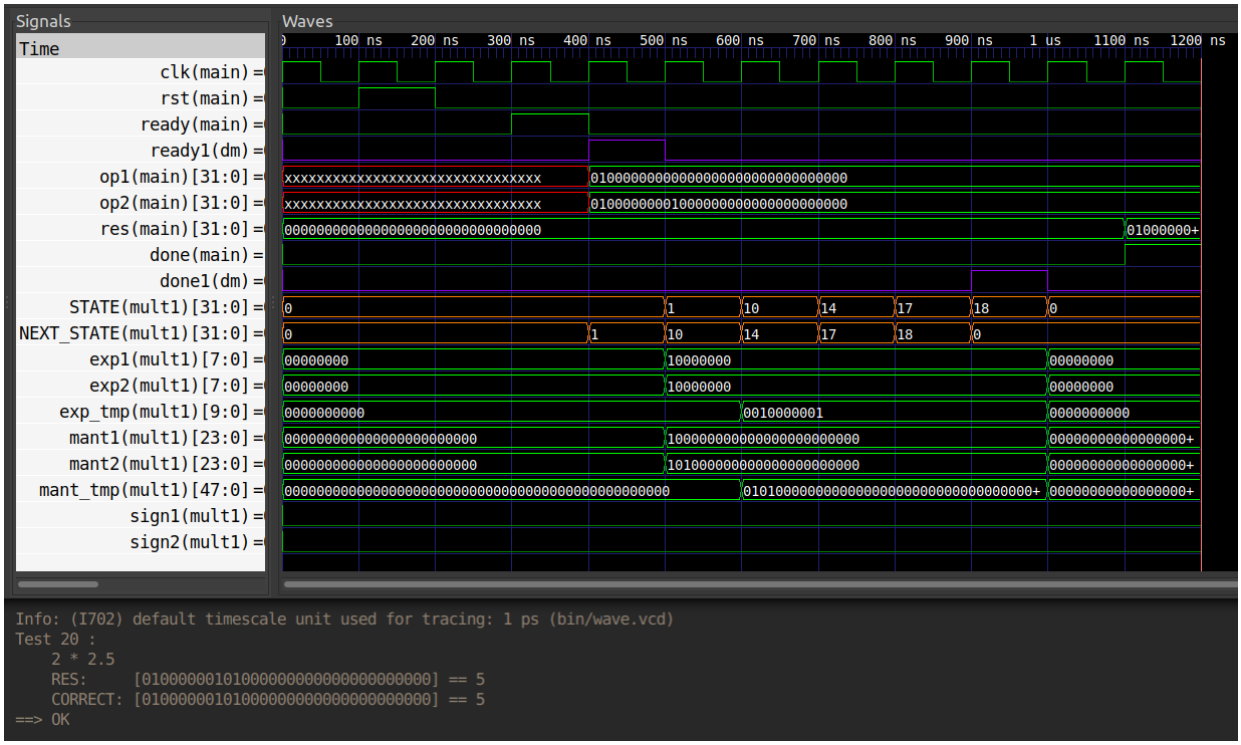


Figura 28: Simulazione double_multiplier in SystemC con “target test”

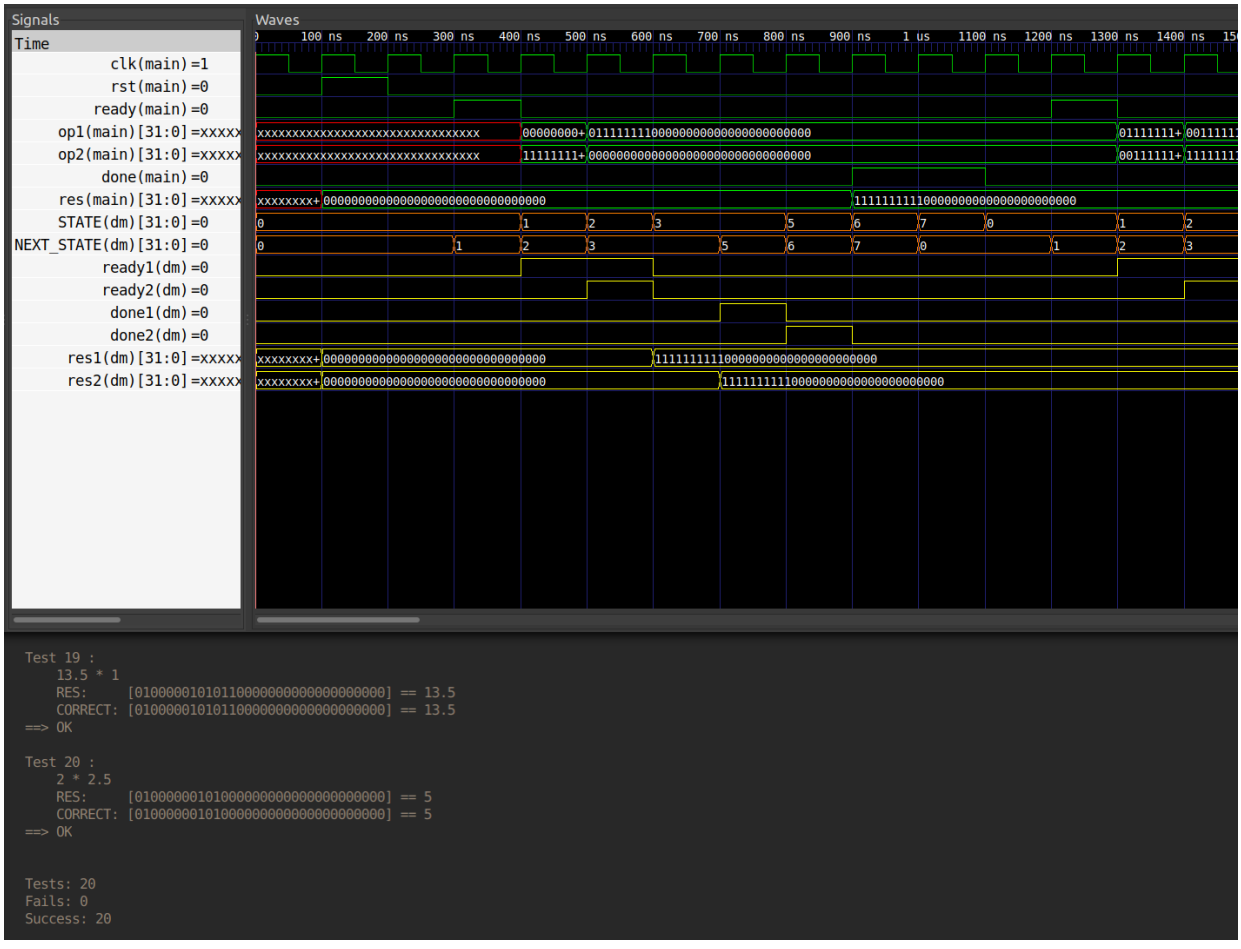


Figura 29: Simulazione double_multiplier in SystemC con “full target test”

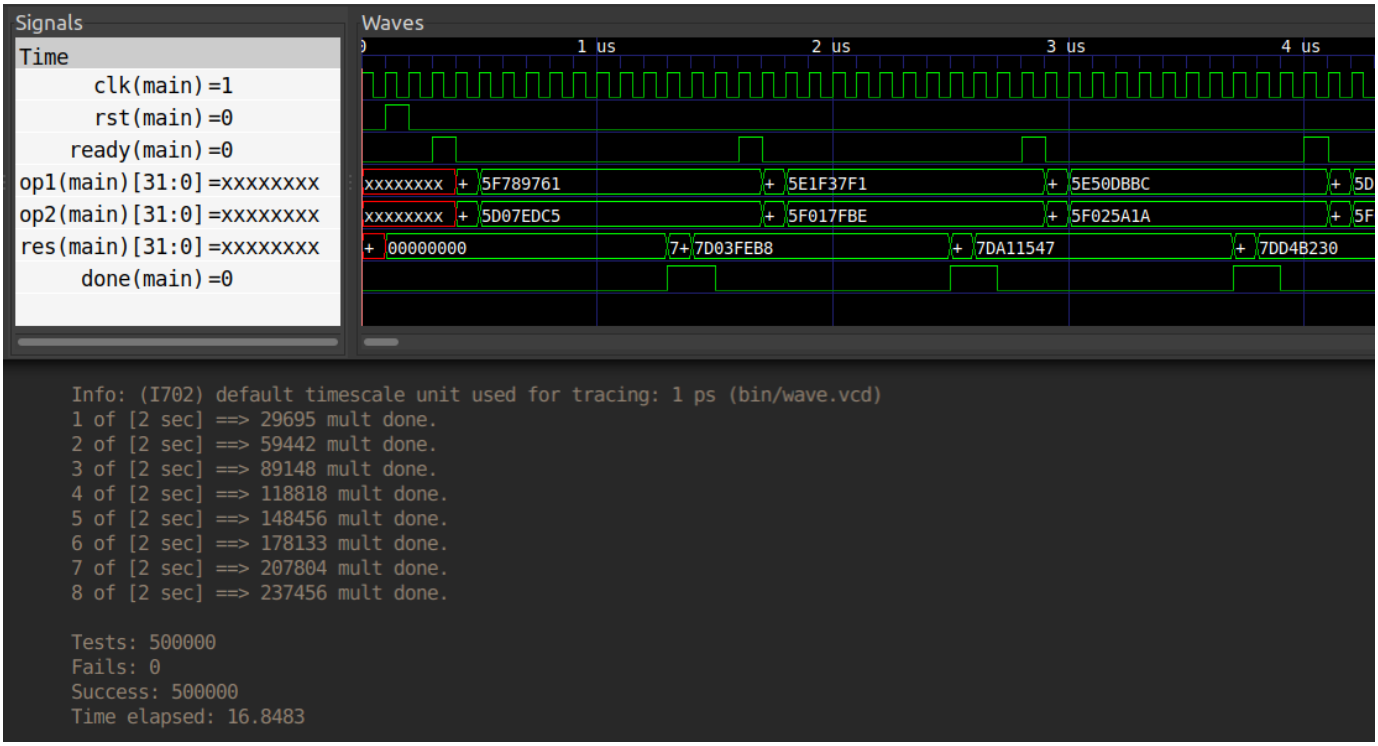


Figura 30: Simulazione double_multiplier in SystemC con “random test”

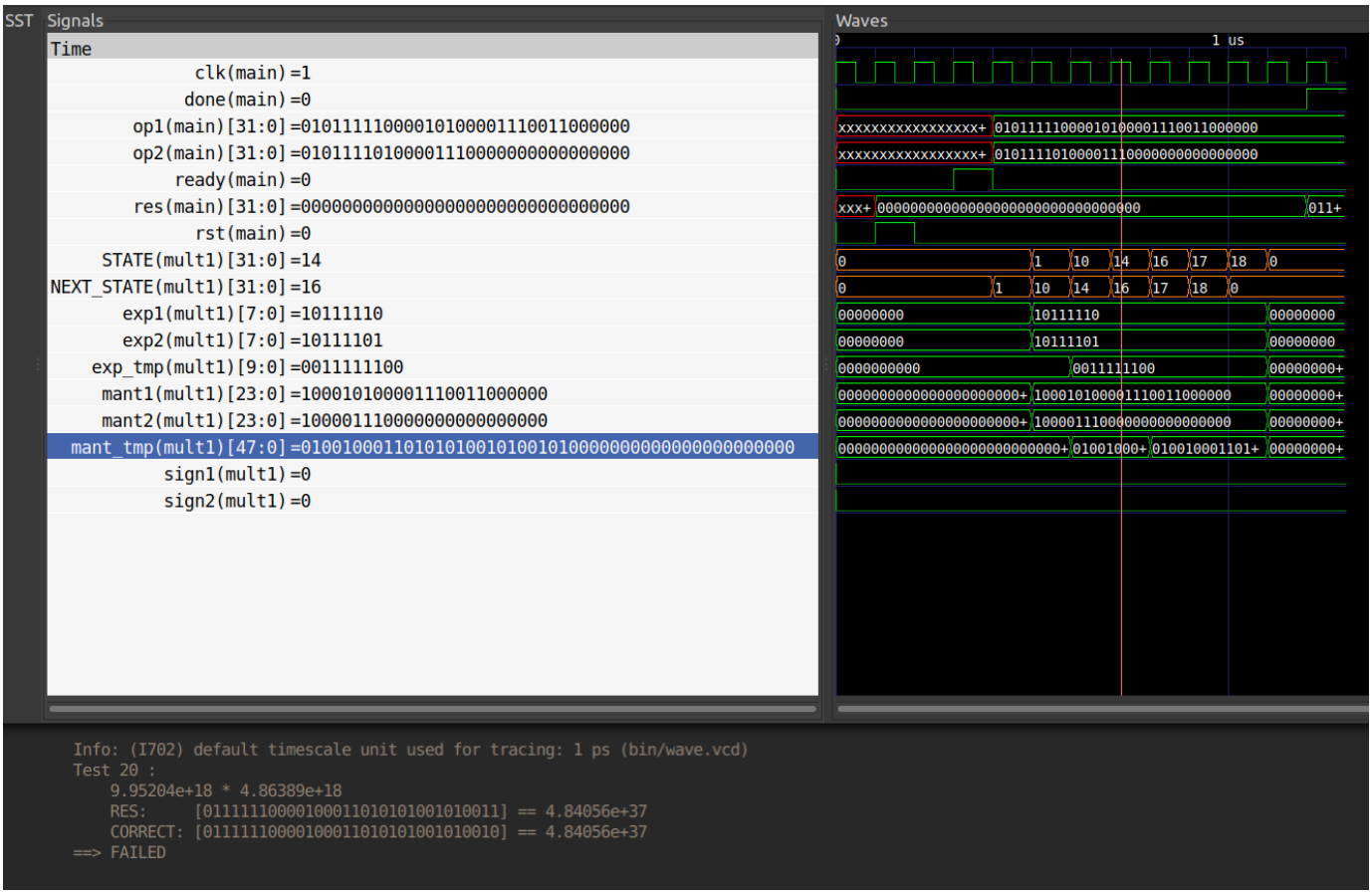


Figura 31: Simulazione double_multiplier in SystemC con errore dovuto all’arrotondamento