

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 parallelo. 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 synthesis 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 synthesis 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 Synthesis (**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

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 i” 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].

In particolare 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 omissso.
- 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:** presentano 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:** presentano 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:

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

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

Float Values ($b = \text{bias}$)

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	$+\infty$
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	$-\infty$
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** (generato 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 la sintesi.
 - **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 contenente il testbench.

B. Procedimento

Per prima cosa è stato studiato lo standard IEEE754 ed è stato definito l'algoritmo ad alto livello per la moltiplicazione. Dopodichè è stata realizzata la EFSM di *double_multiplier* e *double_multiplier*.

A questo punto si è passati all'implementazione 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 synthesis 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. Le FSMD realizzate

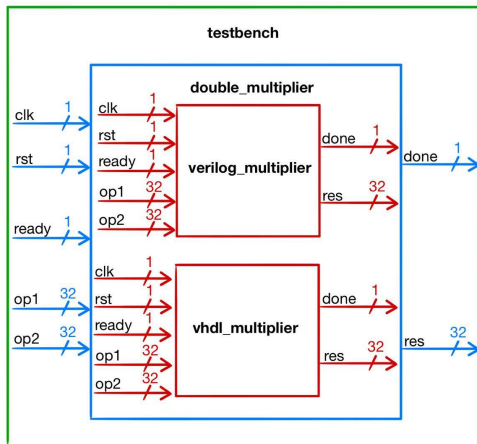


Figura 2. Architettura RTL

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** (4 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 omissso che sarà 1 nel caso di operandi normalizzati, 0 altrimenti.

L'algoritmo della moltiplicazione è descritto grazie alla EFMS [Figura 3] la quale è formata 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 degli operandi di segno, esponente e mantissa la quale presenta anche il bit 24 a 1 (quello omissso nei 32 bit). Qui avviene anche la valutazione dei casi speciali, ovvero la generazione di NaN, ∞ , 0, numeri subnormali che porterà 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 [7]: "Un signal NaN può essere copiato senza problemi, ma qualsiasi altra operazione è invalida e deve essere intercettata o prodotto un Nan non signal". Quindi viene riscritto su *res* 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 si entra lo stesso in 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_ADJ1* 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 avrebbe sommato un bias di $127 + 127$. 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 valido e quindi si passa 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 arrotondare 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 48 (cioè in) 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. Se il bit più significativo di exp_tmp è 1 e la somma dell'esponente temporaneo con 48 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. Il risultato però per essere pronto deve eseguire un ulteriore shift a destra della $mant_tmp$. Di conseguenza anche exp_tmp dovrebbe essere incrementato diventando 1 però complementariamente a quanto accade negli stati *ST_ADJ1*, *ST_ADJ2*, *ST_ADJ3* si lascia l'esponente a zero. Poi si passa a *ST_WRITE*.
- **ST_ROUND:** qui avviene l'arrotondamento della mantissa, il quale 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 significativo di $mant_tmp$ a 1 si passa a *ST_SHIFTR* altrimenti si prosegue in *ST_WRITE*.
- **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 il *done* a 1.

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 2 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 clock *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 4] la quale è formata 8 stati:

- **ST_START:** stato di partenza. Qui vengono resettati i segnali a zero e inizializzati *op1_tmp*, *op2_tmp* rispettivamente con i valori di *op1* e *op2* i quali serviranno per il primo moltiplicatore. In caso di segnale di reset si torna in questo stato e si pongono a 1 i segnali di reset dei due moltiplicatori. In caso il segnale *ready* vada a 1 si passa a *ST_RUN*.
- **ST_RUN:** qui vengono posti *ready1* e *ready2* uguali a 1, attivando quindi i due moltiplicatori. Poi si passa a *ST_WAIT*.
- **ST_WAIT:** qui vengono posti *ready1* e *ready2* uguali 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* 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.

Nella variante Verilog prova casi speciali, valori scelti arbitrariamente e in particolare controlla che i sottocomponenti Verilog e VHDL si comportino allo stesso modo.

Nella variante SystemC sono stati messi a disposizione tre thread da attivare togliendo i commenti nel costruttore del “TestbenchModule”:

- **targeted_test:** test analogo a quello in Verilog.
- **rnd_test:** test che prova *TESTS_NUM* moltiplicazioni generate casualmente tra un intervallo modificabile.
- **run_all:** test 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.

G. 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 utilizzate le librerie “IEEE.STD_LOGIC_1164.ALL” per abilitare i tipi *std_logic* e “use IEEE.NUMERIC_STD.ALL” al fine di poter 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 comunicazione 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 FSM con due processi.

H. Implementazione RTL con SystemC

Sono stati creati 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’eseguitibile *double_multiplier_RTL.x* (generato dopo la compilazione) e *wave.vcd* (generato dopo l’esecuzione dell’eseguitibile).
- **obj:** directory che contiene i file 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 file 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 comunicazione 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 comunicazione 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 dispositivo bisogna usare i comandi:

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

IV. RISULTATI

A. Simulazioni con script TCL

Dopo aver realizzato un componente questo è stato subito testato con un semplice script TCL. In questo modo si possono provare facilmente alcuni input e controllare che all’interno funzioni tutto come ci si aspetta. In particolare modificando lo script sono stati controllati casi particolari come la reazione del sistema al variare inaspettato degli operandi o di *ready*, operazioni con operatori speciali e il comportamento con la richiesta di una nuova elaborazione.

Un altro controllo fatto è il verificare che l’esecuzione del multiplier Verilog sia analoga a quella del multiplier VHDL. Le varie simulazioni sono riportate nelle figure 6 7 8.

B. Simulazione con testbench in Verilog

I test con gli script TCL sono serviti principalmente ad assistere l'implementazione, questo invece a verificare la correttezza del sistema. Sono stati testati i casi particolari e un po' di valori scelti arbitrariamente e confrontati col risultato che ci si aspettava. Questo risultato è stato ottenuto grazie al testbench in SystemC.

In figura 9 si può guardare la simulazione.

C. Simulazione con testbench in SystemC

Grazie al c++ è stato possibile realizzare dei testbench più accurati.

Sono stati provati gli stessi valori del testbench in Verilog i quali però sono stati confrontati con il risultato convertito in binario della moltiplicazione degli operandi convertiti in float [Figura 10].

Poi è stato fatto un testbench più sofisticato che genera un numero di valori arbitrario e confronta i risultati ottenuti dal dispositivo e dalla moltiplicazione in c++ [Figura 11].

SystemC è stato particolarmente comodo per testare il sistema grazie alla possibilità di guardare i valori di tutti i segnali evitando quindi di modificare le interfacce o controllare i componenti separatamente come con gli script TCL.

D. Sintesi

V. 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] 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] "Ieee 754 lectures notes," "http://li.mit.edu/Archive/Activities/Archive/CourseWork/Ju_Li/MITCourses/18.335/Doc/IEEE754/ieee754.pdf".
- [8] "Ieee 754 multiplication," https://en.wikipedia.org/wiki/Single-precision_floating-point_format.
- [9] "Vivado," <https://www.xilinx.com/products/design-tools/vivado.html>.

APPENDICE

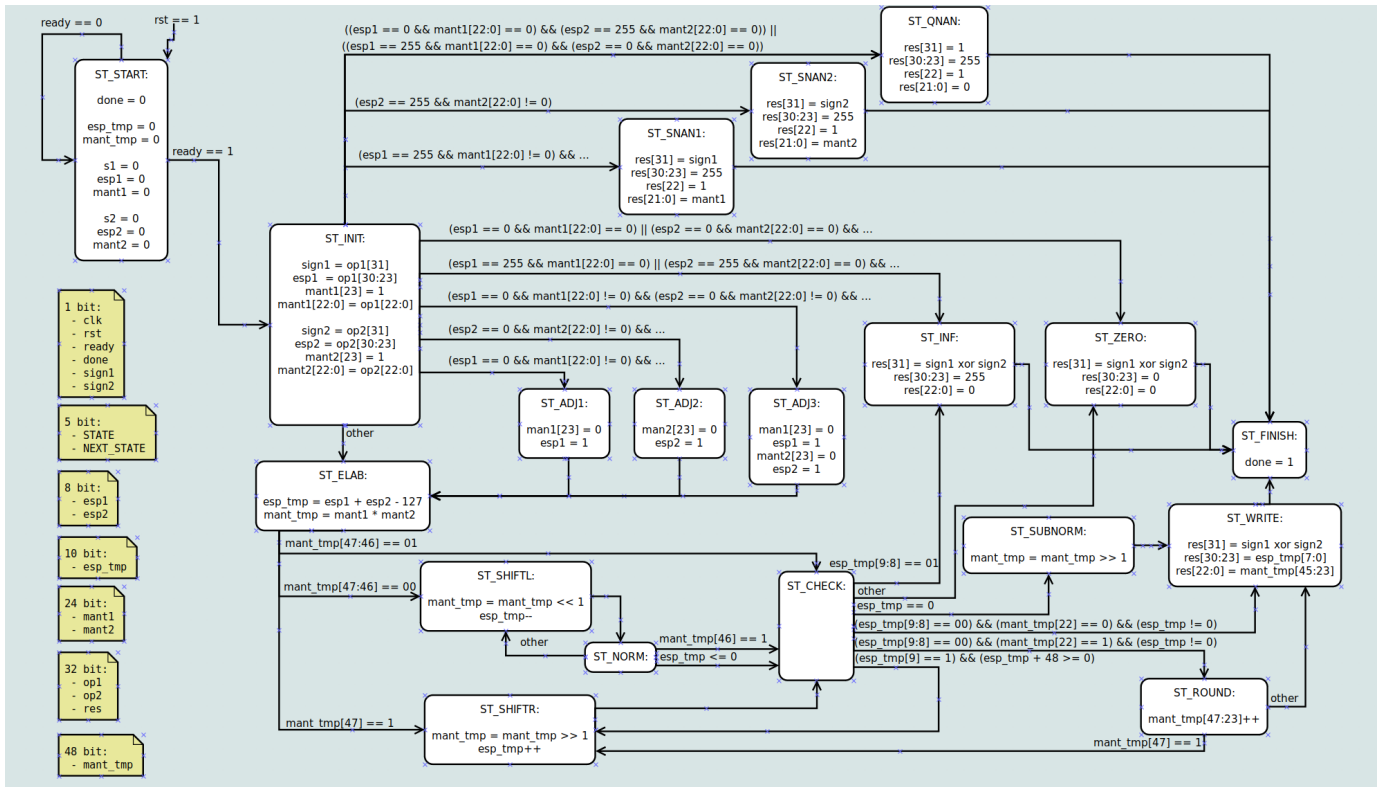


Figura 3. EFSM del multiplier

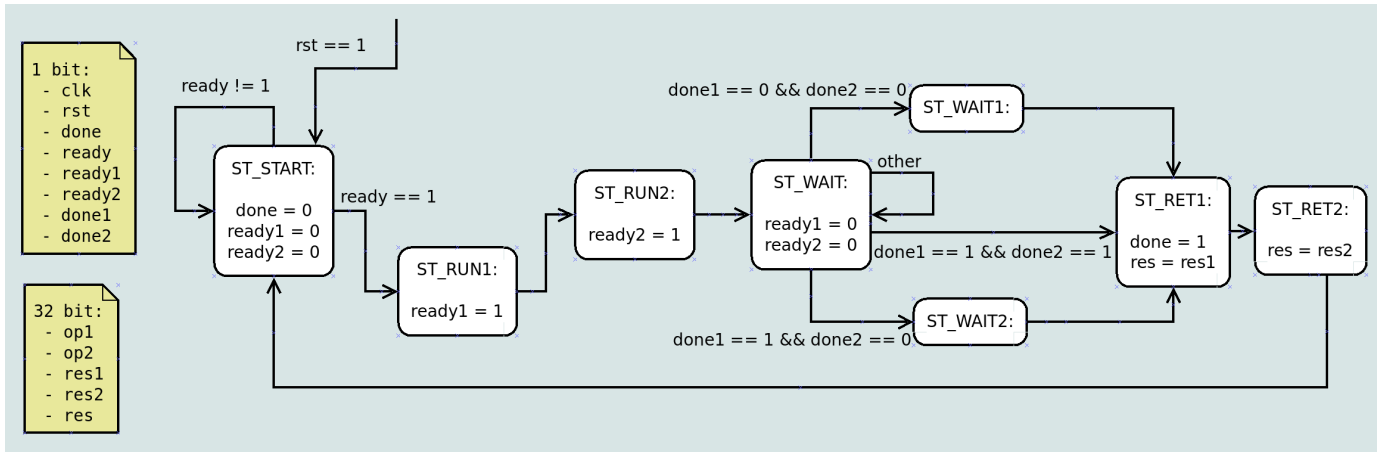


Figura 4. EFSM del double_multiplier

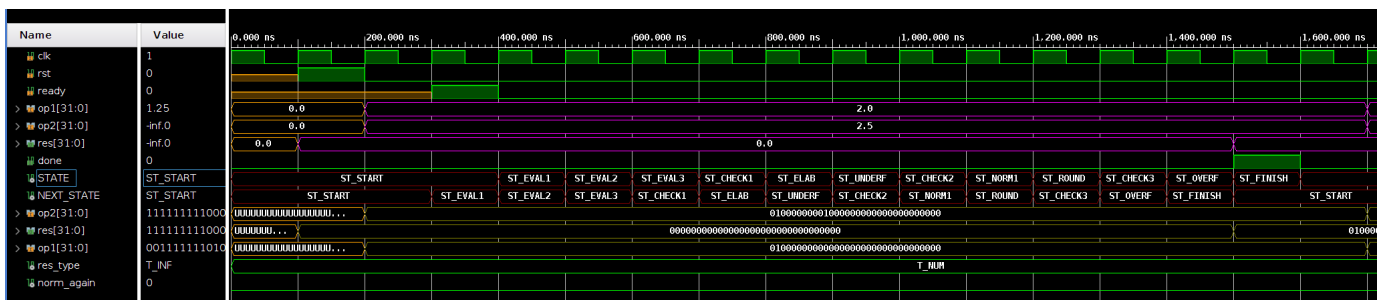


Figura 5. Simulazione multiplier in VHDL con script TCL con più zoom

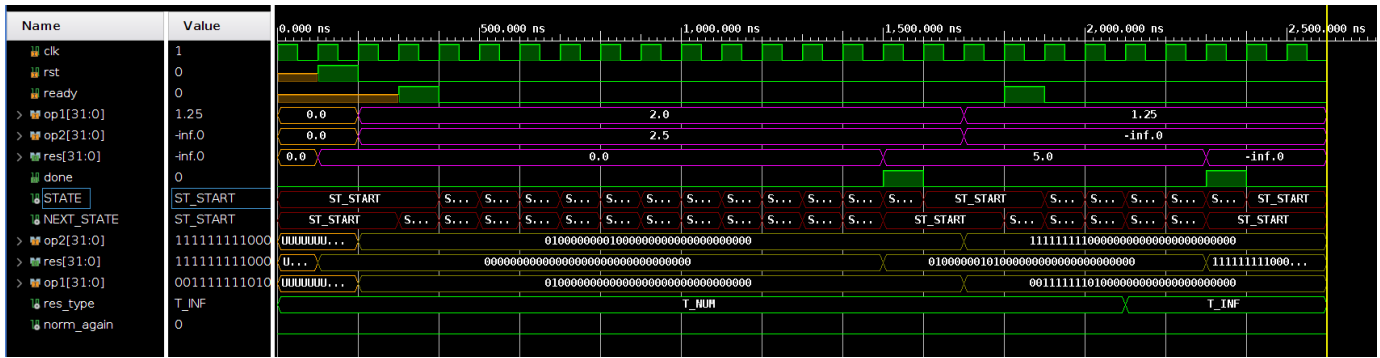


Figura 6. Simulazione multiplier in VHDL con script TCL

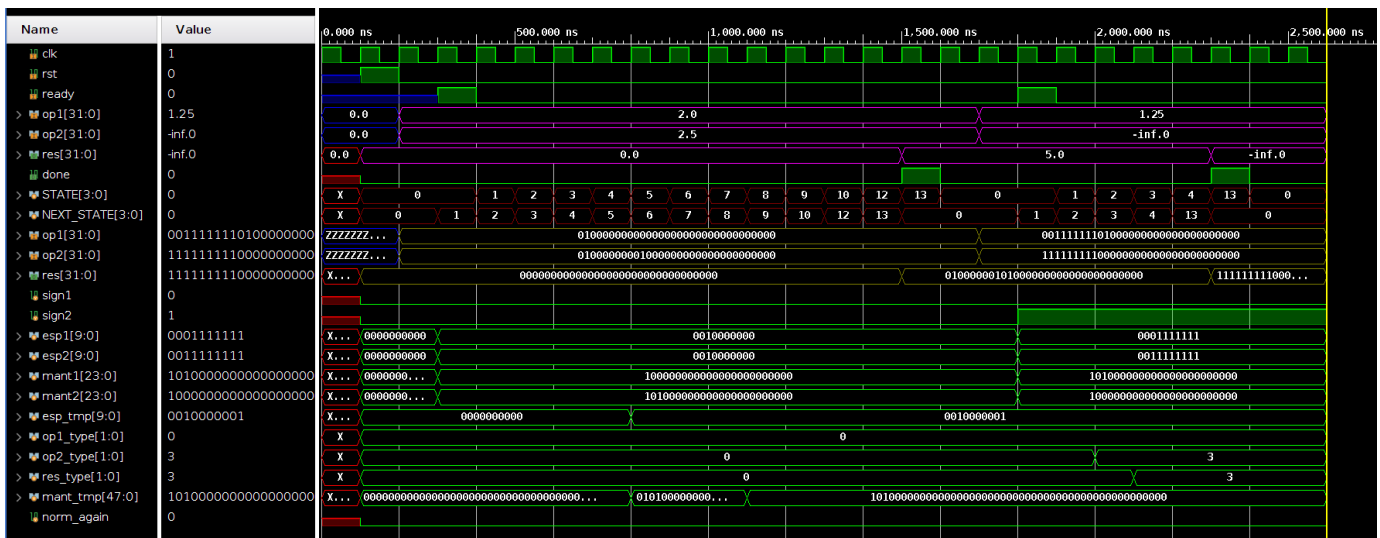


Figura 7. Simulazione multiplier in Verilog con script TCL

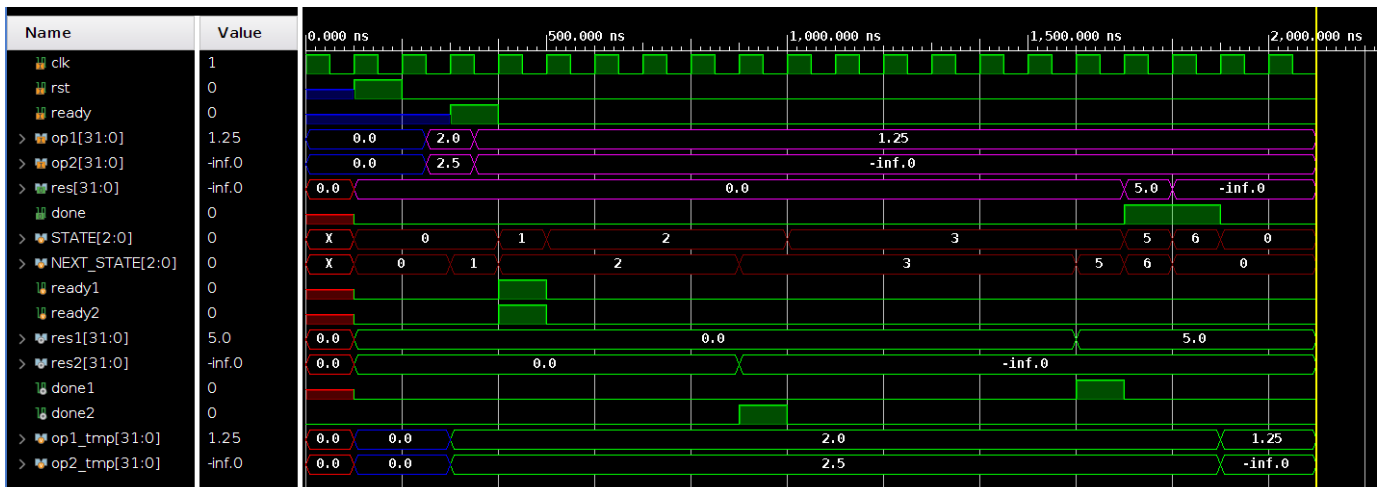


Figura 8. Simulazione double_multiplier in Verilog con script TCL

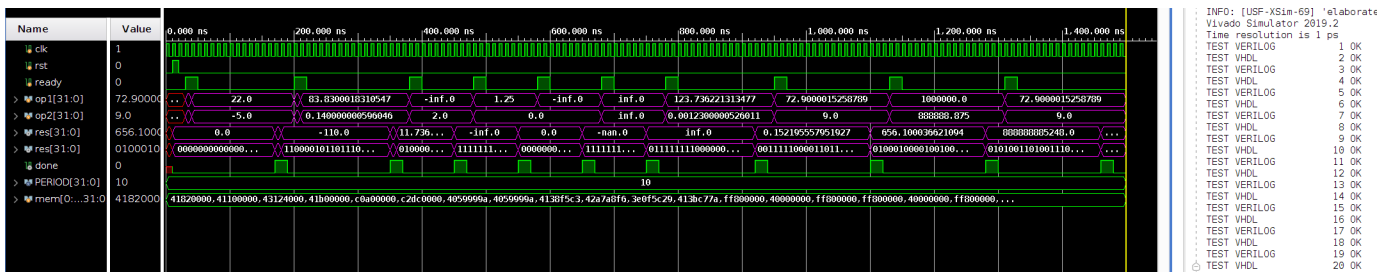


Figura 9. Simulazione double_multiplier in Verilog con testbench

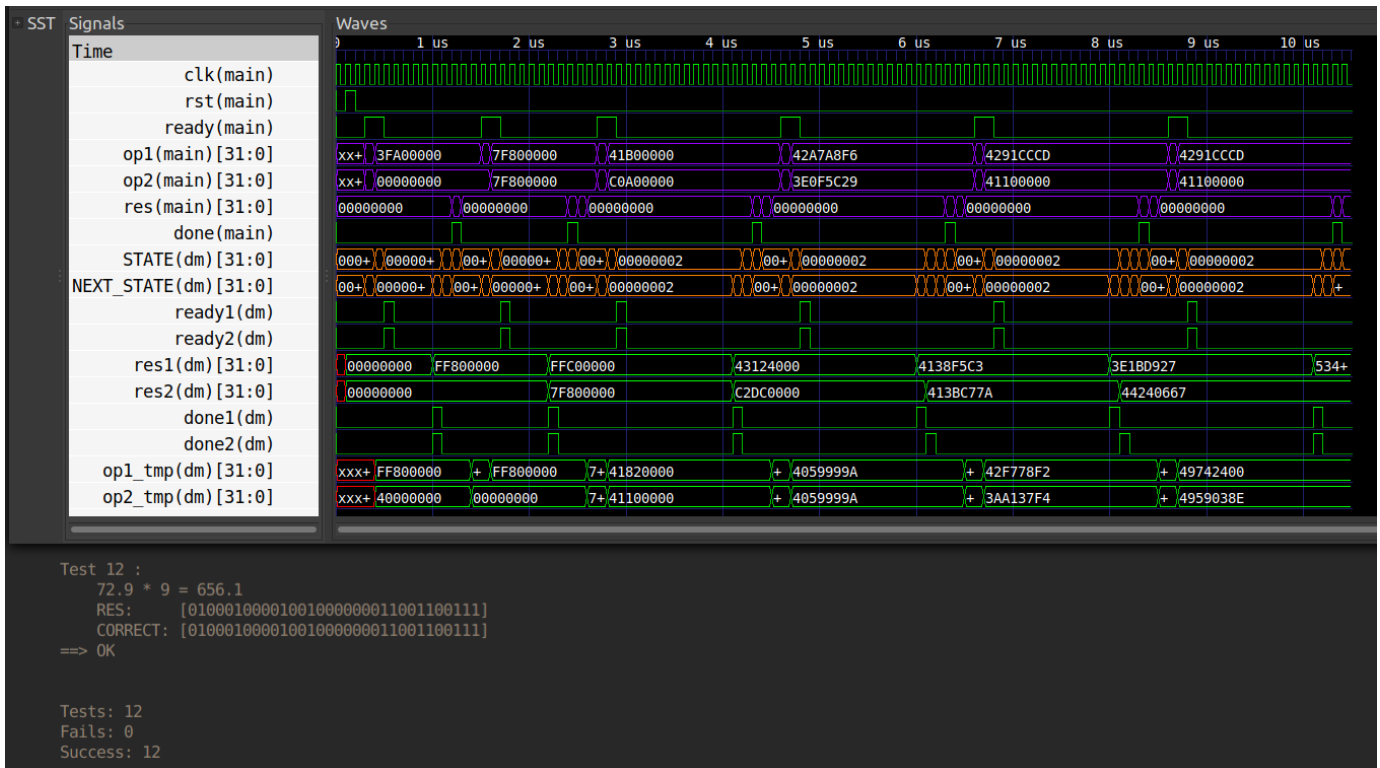


Figura 10. Simulazione double_multiplier in SystemC con “targeted test”

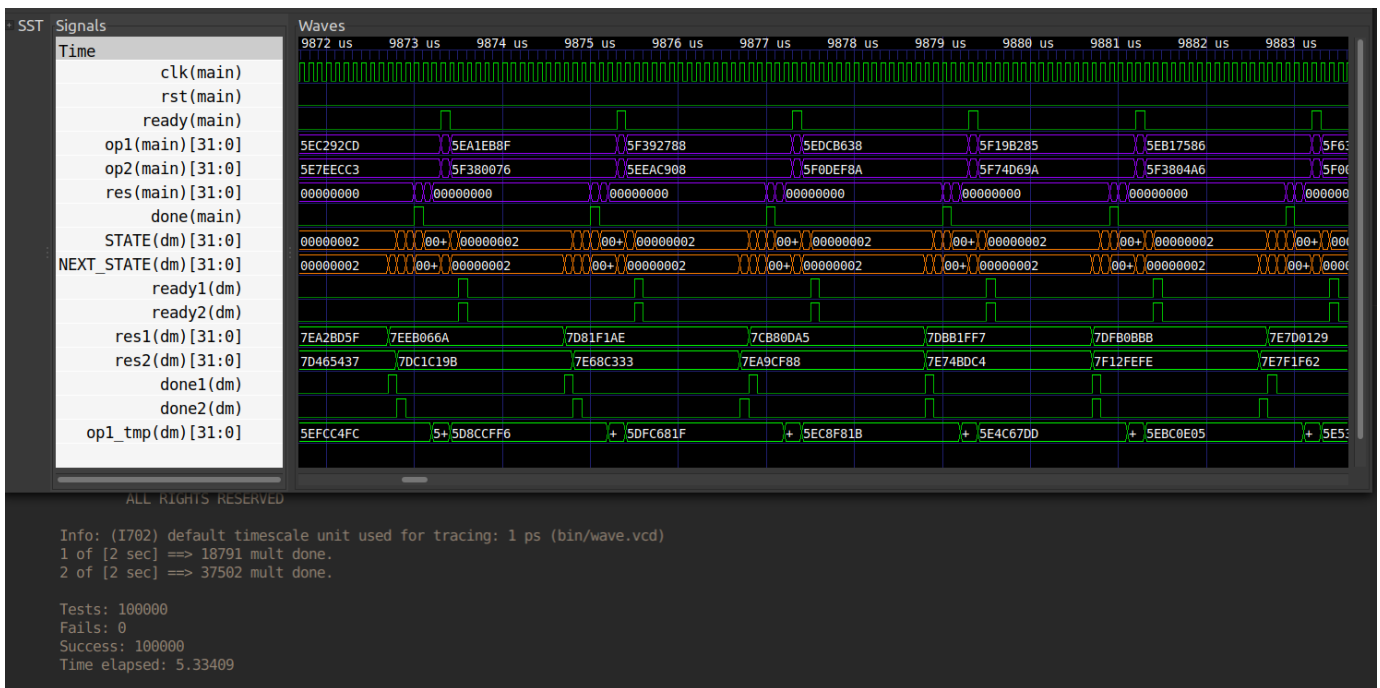


Figura 11. Simulazione double_multiplier in SystemC con “random test”