

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 sistema è composto da un modulo top-level chiamato “double_multiplier” il quale esegue la moltiplicazione di due operandi dati in input nello stesso ciclo di clock in cui ready viene posto a 1, e i due operandi passati al ciclo di clock successivo.

Nell’introduzione viene descritto in maniera astratta quello che poi viene dettagliato nel seguito del report. Una buona scaletta per l’introduzione può essere la seguente:

- Descrizione ad alto livello delle principali caratteristiche del sistema che si vuole modellare.
- Descrizione delle motivazioni principali per l’utilizzo delle tecnologie descritte nel corso. Qual è il problema che si vuole risolvere?
- Descrizione dei passi utilizzati per arrivare all’implementazione finale. Descrivere la motivazione di ciascun passo. La descrizione dei passi dovrebbe formare la descrizione del flusso di lavoro svolto per completare l’assignment.
- Rapidissima descrizione dei risultati principali.

L’introduzione non dovrebbe andare oltre la metà della seconda colonna (nel caso a due colonne), o la prima pagina (nel caso a colonna singola): bisogna cercare di essere concisi (e chiari). Alla fine, l’introduzione è solo “chiacchiere”: deve semplicemente rendere chiari quali sono gli obiettivi del lavoro (e nel caso del corso, deve far capire a me che avete gli obiettivi chiari in testa). Consiglio: l’introduzione (e spesso l’abstract) è l’ultima parte che viene completata.

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 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 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].

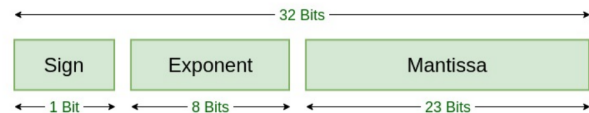


Figura 1. IEEE 754 single precision

Per la codifica in numero binario:

- Dal segno si ricava il bit più significativo (1 se negativo 0 altrimenti).
- Si converte il numero in binario.
- Si sposta la virgola a sinistra fino ad avere un numero nella forma $1, \dots \cdot 2^E$.
- La mantissa è la parte a destra della virgola, riempita con zeri a destra fino a riempire i 23 bit.
- L’esponente è $127 + E$ dove E è l’esponente ricavato dallo shift.

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 l’algoritmo usato 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

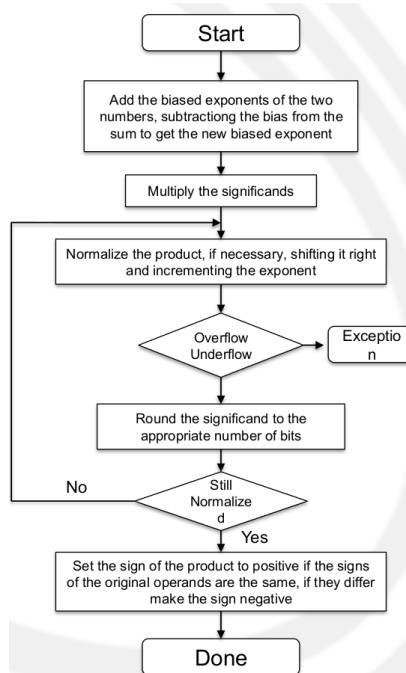


Figura 3. IEEE 754 multiplication

III. METODOLOGIA APPLICATA

Il primo passo è stato la realizzazione della EFSM di *multiplier* e *double_multiplier* che ha portato ad acquisire una visione generale.

Dopo si è passati all'implementazione a livello RT con Vivado[6] del *multiplier* sia in Verilog che in VHDL. Un primo test è stato fatto con TLC-script 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. A questo punto è stato realizzato un test più accurato in verilog controllando tutti i casi particolari per poi passare alla sintesi.

È anche stato rifatto tutto in SystemC dove si è potuto fare un testbench più fine grazie alla potenza del c++.

Infine si è provato a fare l'high level synthesis da un semplice codice c++ per confrontare i risultati ottenuti.

A. Vincoli ed Architettura

Il progetto a 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 utilizzato il protocollo di handshake. Vengono quindi utilizzati gli stessi 32 bit per il risultato e altri 64 bit per le due coppie di operandi. Al primo ciclo di clock, con il flag "ready" uguale a 1, verranno trasmessi i primi due operandi e al ciclo successivo gli altri due. Dopodiché si aspetterà il complementamento delle moltiplicazioni, segnalato col flag "done" uguale a 1, per poi trasmettere il primo risultato, e il secondo al ciclo di clock successivo.

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 FSM è realizzata

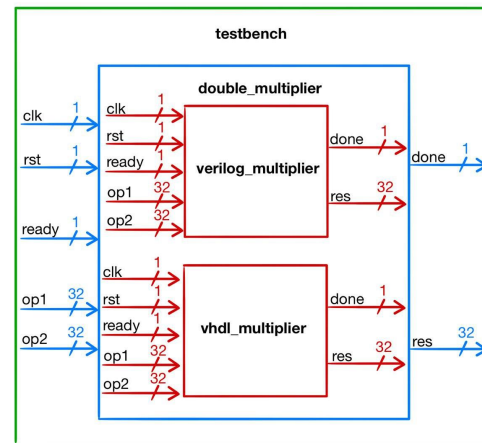


Figura 4. Architettura RTL

con due processi:

- **fsm:** processo asincrono attivato con la variazione di qualche segnale interno. Esso ha il compito di calcolare e aggiornare lo stato prossimo.
- **datapath:** processo sincrono che ha il compito di aggiornare lo stato attuale, elaborare gli output. Esso viene anche attivato dal fronte di salita del reset al fine di riportare lo stato a quello iniziale.

B. multiplier

Questo componente esegue la moltiplicazione tra numeri floating point a precisione singola.

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 il prossimo ciclo di clock ci sarà il primo operando in *op1* e il ciclo di clock successivo in *op2*. (?)
- **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 il numero ha bisogno di essere ulteriormente normalizzato.
- **res_type** (2 bit): indicare 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. Il caso del numero denormalizzato è gestito come se fosse normalizzato.
- **STATE e NEXT_STATE** (4 bit): rappresentano lo stato attuale e lo stato prossimo.
- **op1_type e op2_type** (2 bit): per indicare il tipo degli operandi ovvero 0, NaN, ∞ oppure un numero. Solo se sono entrambi sono dei numeri *res_type* sarà 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.

Nello specifico in Verilog sono rappresentati tutti come reg, mentre in VHDL *res_type*, *norm_again*, *STATE*, *NEXT_STATE* come signal e gli altri come variable.

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

- **ST_START**: pone *done* e *norm_again* a 0 ed estrae le informazioni di segno, esponente e mantissa dei due operandi. Si rimane qui finché *ready* vale 0 altrimenti si passa a *ST_EVAL1*. Se in qualsiasi stato si riceve *reset* uguale a 1 allora si passa a questo stato.
- **ST_EVAL1 e ST_EVAL2**: ricava rispettivamente il tipo di *op1* e *op2* fra *T_ZER*, *T_INF*, *T_NAN* e *T_NUM*
- **ST_EVAL3**: ricava il tipo del risultato in base al tipo di *op1* e *op2*.
- **ST_CHECK1**: controlla se *res_type* è uguale a *T_NUM* e in tal caso passa a *ST_ELAB* altrimenti a *ST_FINISH*.
- **ST_ELAB**: esegue la somma tra i due esponenti e la sottrazione di 127 in quanto entrambi, per lo standard, sono incrementati di 127: $(esp + 127) = (esp1 + 127) + (esp2 + 127) - 127$. Per compiere la somma è necessario usare una variabile *esp_tmp* di 10 bit perché altrimenti con 8 bit si andrebbe in overflow con valori che dopo la sottrazione sarebbero rappresentabili con 8 bit e quindi validi. Viene poi eseguita anche la moltiplicazione delle due mantisse, che essendo a 23 bit più un bit che vale sempre 1, necessita di una variabile *mant_tmp* di almeno 48 bit
- **ST_UNDERF**: controlla se *esp_tmp* è in uno stato di underflow, ovvero guardando se il bit *esp_tmp*[9], che nel complemento a 2 indica il segno, è 1. Infatti i valori disponibili per l'esponente vanno da 0 a 255. Il controllo

dell'overflow viene fatto in seguito all'arrotondamento per evitare di farlo due volte.

- **ST_CHECK2**: passa allo stato *ST_FINISH* se *res_type* è diverso da *T_NUM*, perché diventato *T_ZER* per l'underflow.
- **ST_NORM1**: compie la normalizzazione della mantissa che deve sempre essere della forma *1.bits*. Essendo la virgola posta tra il 46esimo bit e il 45esimo, si verificano due casi: Se il 47esimo bit vale 1 bisogna incrementare l'esponente, altrimenti il valore è già corretto, ma viene effettuato uno shift a sinistra per trattare allo stesso modo i due casi durante l'arrotondamento.
- **ST_ROUND**: effettua l'eventuale arrotondamento dovuto al fatto che il valore della mantissa è attualmente a 48 bit, ma bisogna portarlo a 24 bit. L'arrotondamento è fatto per eccesso, quindi si incrementa *mant_tmp*[47:24] solo se *mant_tmp*[23:00] è \geq a "01..1". L'arrotondamento effettivo verrà fatto nello stato *ST_NORM2*, qui ci si limita a porre *norm_again* uguale a 1 per poterci andare.
- **ST_CHECK3**: passa allo stato *ST_NORM2* se *norm* è uguale a 1 oppure allo stato *ST_OVERF* se *norm* è uguale a 0.
- **ST_NORM2**: effettua il vero arrotondamento della mantissa. Bisogna tenere conto del caso in cui sia della forma "1..1" e che quindi con l'incremento vada a "0..0" e venga incrementato l'esponente.
- **ST_OVERF**: verifica se l'esponente del risultato è in uno stato di overflow, ovvero guardando se il bit *esp_tmp*[8] vale 1 ovvero se corrisponde ad un valore maggiore di 255.
- **ST_FINISH**: pone *done* uguale 1 e ricava *res*[31], ovvero il segno del risultato facendo lo XOR fra i segni degli operandi. In base al valore di *res_type* si ricavano gli altri bits e si torna allo stato iniziale.

C. 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 il prossimo ciclo di clock ci sarà il primo operando in *op1* e il ciclo di clock successivo in *op2*. (?)
- **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_tmp1, op2_tmp1, op1_tmp2, op2_tmp2:** (32 bit) che servono a memorizzare temporaneamente gli operandi che arrivano nei vari cicli di clock. (?)

Nello specifico in Verilog *STATE*, *NEXT_STATE*, *ready1* e *ready2* sono rappresentati come reg mentre *res1*, *res2*, *done1* e *done2* come wire.

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* rispettivamente con i valori di *op1* e *op2* i quali serviranno per il primo moltiplicatore. Si rimane qui finchè *ready* vale 0 altrimenti 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* rispettivamente 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.

D. Implementazione RTL

L'interfaccia del *double_multiplier* è formata dai segnali di input: *clk*, *rst*, *ready*, *op1*, *op2*; E i segnali di output *res*, *done*. L'interfaccia del *multiplier* è formata anch'essa dai segnali di input: *clk*, *rst*, *ready*, *op1*, *op2*; E i segnali di output *res*, *done*, dove:

- *clk*, *rst* sono collegati ai segnali analoghi del *double_multiplier*.
- *ready* è collegato al segnale interno del *double_multiplier* *ready1*, per multiplier1 e *ready2* per multiplier2.
- *op1* e *op2* sono collegati ai segnali interni del *double_multiplier* *op1_tmp1*, *op2_tmp1*, per multiplier1 e *op1_tmp2*, *op2_tmp2* per multiplier2.
- *res* è collegato al segnale interno del *double_multiplier* *res1*, per multiplier1 e *res2* per multiplier2.

- *done* è collegato al segnale interno del *double_multiplier* *done1*, per multiplier1 e *done2* per multiplier2.

op1_tmp1, *op2_tmp1*, *op1_tmp2*, *op2_tmp2* sono necessari al fine di conservare i valori della prima coppia di operandi e della seconda, che verranno passati ai moltiplicatori dei cicli di clock dopo.

E. Implementazione RTL con Verilog e VHDL

Si creano i seguenti files:

- **verilog_multiplier:** implementazione verilog del moltiplicatore (sintetizzabile)
- **vhdl_multiplier:** implementazione vhdl del moltiplicatore (sintetizzabile)
- **double_multiplier:** implementazione verilog del doppio moltiplicatore (sintetizzabile)
- **testbench:** implementazione verilog di un testbench da usare solo in simulazione

In *verilog_multiplier* e *double_multiplier*

- Sono definiti come "wire" tutti i segnali collegati alle porte di input mentre come "reg" tutti quelli collegati alle porte di output.
- Sono definiti come "reg" tutti i segnali interni di comunicazione.
- 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 comunicazione.
- 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.

F. 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
- **testbench:** 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 comunicazione.
- 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.
- 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.

IV. RISULTATI

Il testbench verilog, *Figure 7*, 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:** che analogamente a quello in Verilog, testa due volte il *double_multiplier*. *Figure 8*.
- **rnd_test:** che prova *TESTS_NUM* moltiplicazioni generate casualmente tra un intervallo modificabile. *Figure 9*.
- **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. Ma anche così il tempo necessario a completarlo è troppo elevato per la mia macchina.

Con la sintesi si sono ottenuti i seguenti risultati:

Questa sezione può contenere anche riflessioni personali sui risultati ottenuti. Importante: tutte le affermazioni devono essere supportate da numeri¹.

V. CONCLUSIONI

Le conclusioni dovrebbero riassumere in poche righe tutto ciò che è stato fatto. Un paio di righe descrivono i risultati osservati, in modo da introdurre poi la conclusione “vera e propria”. Nel caso del corso, la “lezione da portare a casa” sarà quello che si è imparato svolgendo l’elaborato.

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] “Iee 754,” https://en.wikipedia.org/wiki/IEEE_754.
- [5] “Iee 754 multiplication,” https://en.wikipedia.org/wiki/Single-precision_floating-point_format.
- [6] “Vivado,” <https://www.xilinx.com/products/design-tools/vivado.html>.

APPENDICE

¹Richard Feynman on Scientific Method (1964) - <https://www.youtube.com/watch?v=OL6-x0modwY>

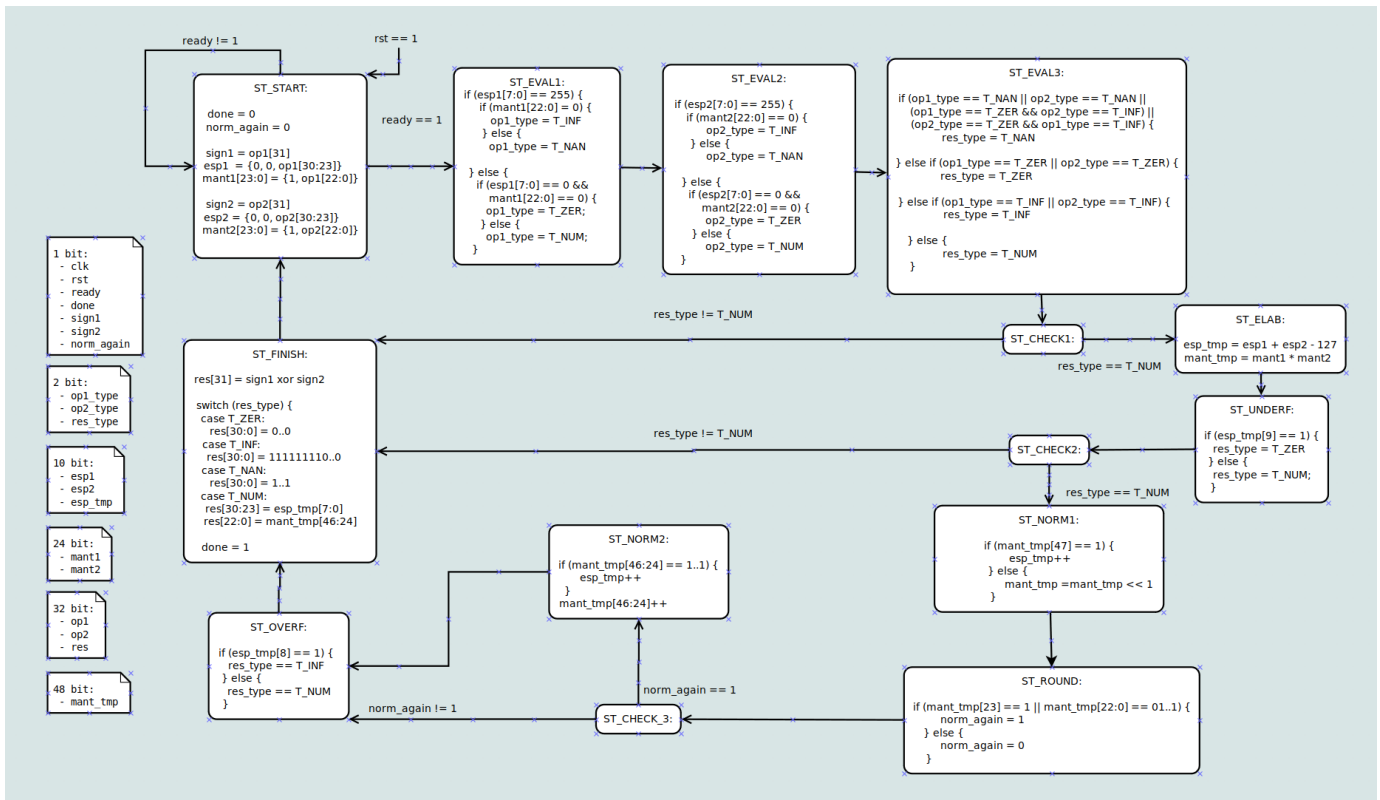


Figura 5. EFSM del multiplier

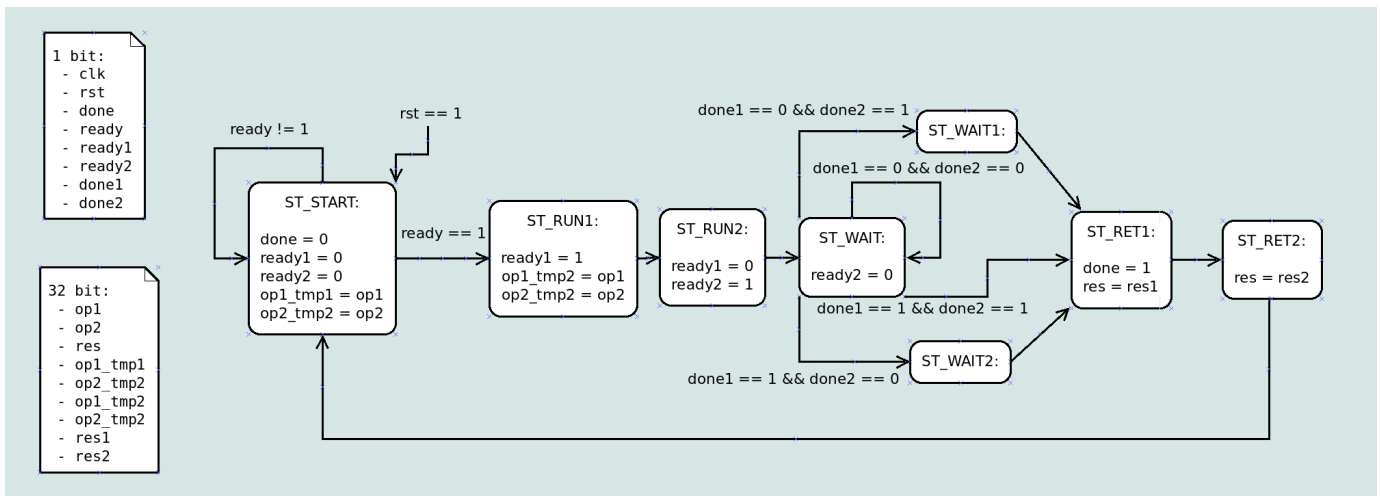


Figura 6. EFSM del double_multiplier

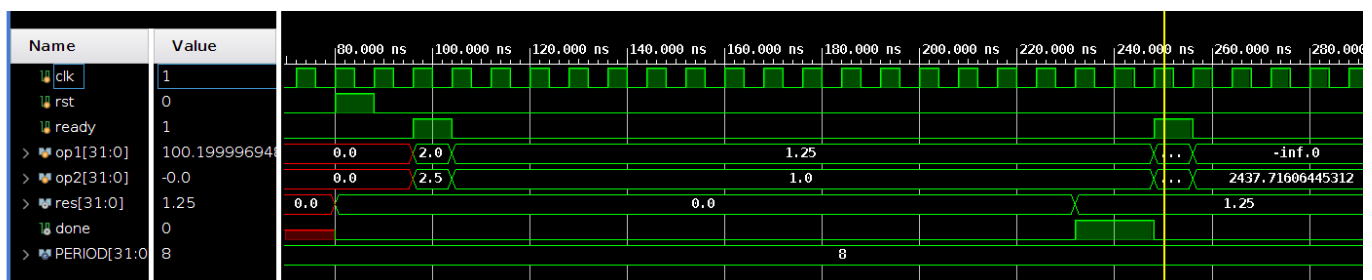


Figura 7. Simulazione in Verilog

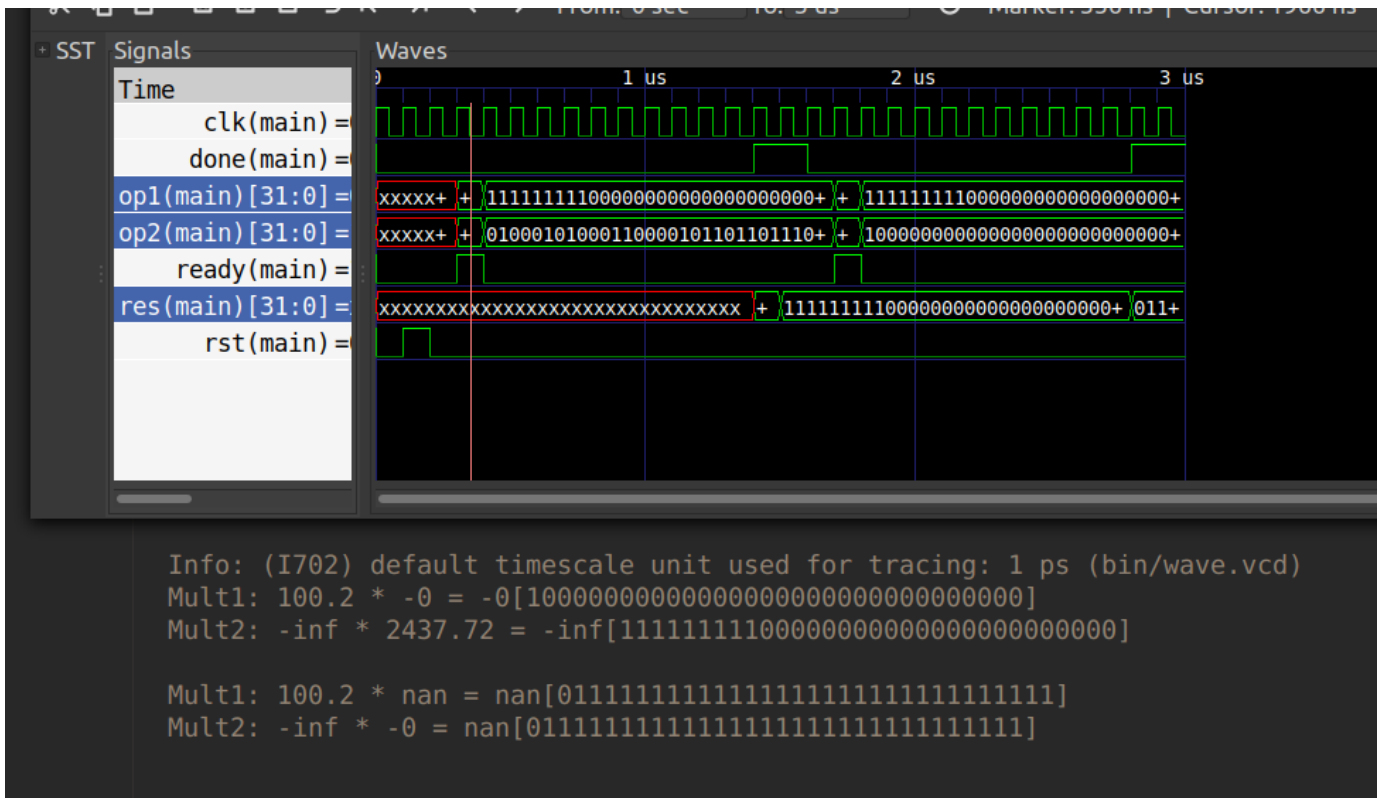


Figura 8. Simulazione in SystemC con targeted_test

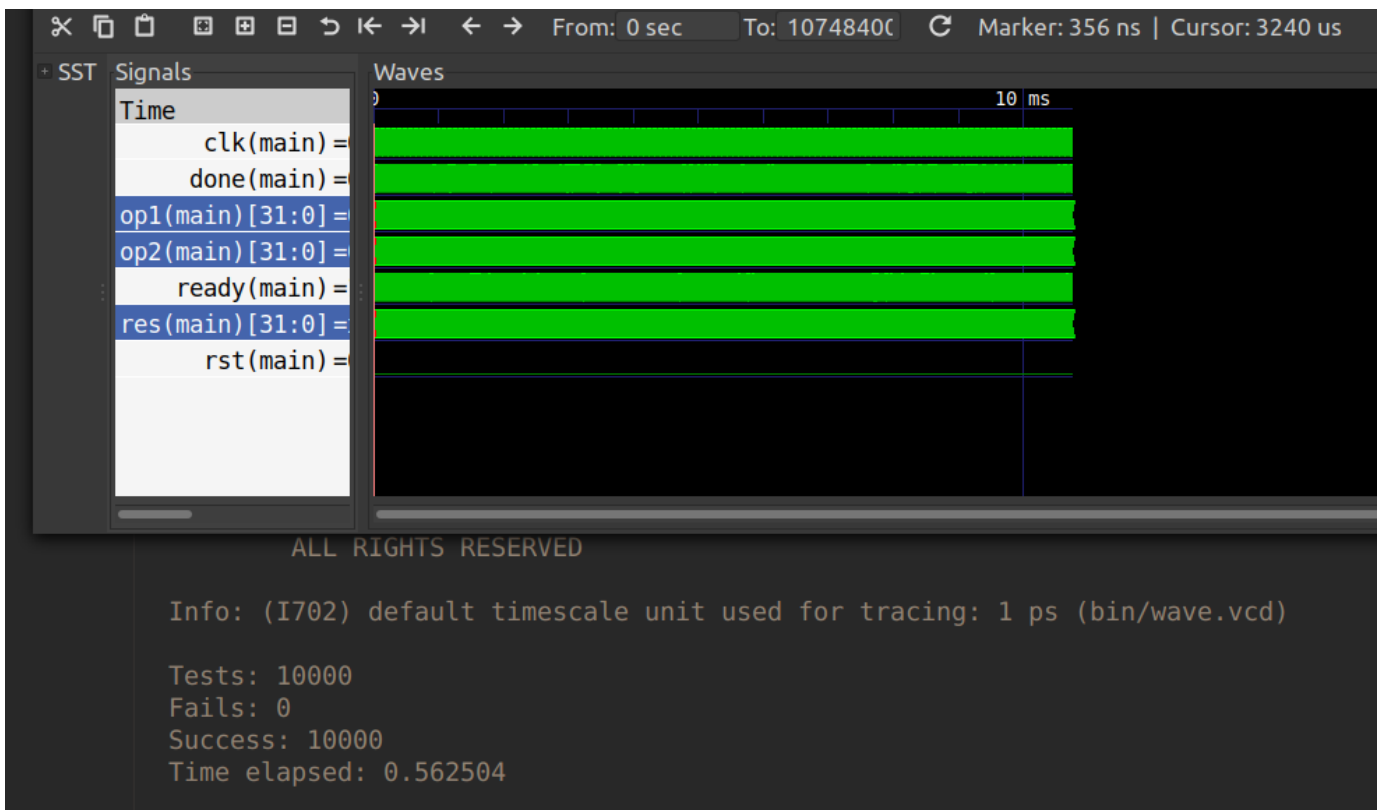


Figura 9. Simulazione in SystemC con rnd_test