Modellazione e sintesi di un accelleratore hardware per la moltiplicazione di numeri in virgola mobile a singola precisione

Luigi Capogrosso - VR445456

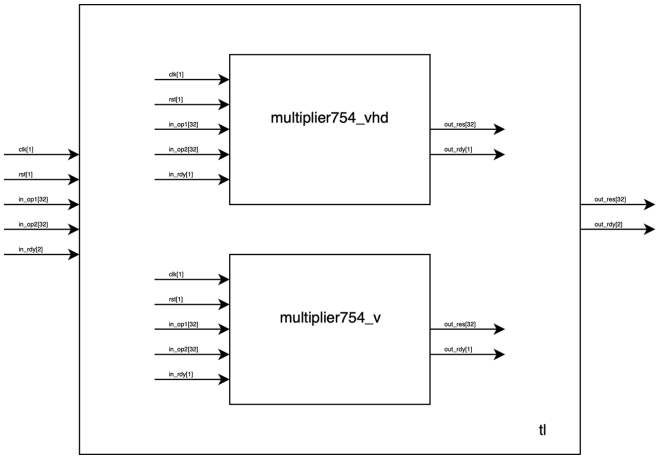

Figura 1. Organizzazione del sistema.

***Sommario*—Questo documento presenta l'implementazione, ai fini didattici, di un accelleratore hardware per la moltiplicazione di numeri espressi secondo lo standard IEEE for Binary Floating-Point Arithmetic (ANSI/IEEE Std 754-1985) a 32 bit.**

Il progetto è sviluppato facendo uso dei linguaggi VHDL, Verilog, SystemC e C++.

I. Introduzione

Il sistema descritto esegue due moltiplicazioni (fra i medesimi fattori) in virgola mobile a singola precisione e, restituisce i prodotti serializzati. La Figura 1 ne mostra la struttura, organizzata nei seguenti moduli:

- `multiplier754_vhd` in Figura 2, si occupa di eseguire la moltiplicazione di due numeri secondo lo standard IEEE for Binary Floating-Point Arithmetic (ANSI/IEEE Std 754-1985) a 32 bit, implementato facendo uso del linguaggio VHDL.
- `multiplier754_v` in Figura Figura 2, si occupa di eseguire la moltiplicazione di due numeri secondo lo standard IEEE for Binary Floating-Point Arithmetic (ANSI/IEEE Std 754-1985) a 32 bit, implementato facendo uso del linguaggio Verilog.
- `top_level`, si occupa della sincronizzazione dei due componenti rispetto alle porte di ingresso e uscita.

L'elaborato ha come obiettivo la creazione di un modulo hardware, sintetizzabile, progettato a diversi livelli di astrazione. Nello specifico, per quanto riguarda lo sviluppo per mezzo dei linguaggi VHDL e Verilog è richiesta una implementazione

Figura 2. Modulo VHDL e Verilog del moltiplicatore floatig-point a singola precisione.

a **livello RT**, mentre, invece, per quanto riguarda lo sviluppo per mezzo dei linguaggi SystemC e C++ è richiesta una implementazione a **livello algoritmico**.

Il secondo scopo del progetto è quello di, attraverso il tool per lo sviluppo hardware di Xilinx, eseguire la sintesi su una FPGA (`PYNQ xc7z020clg400-1`) dei modelli realizzati così da verificare anzitutto la corretta funzionalità dei moduli ed, inoltre, comprendere il perché l'iniziale spinta innovativa dell'HLS (acronimo di *High Level Syntesis Language*) non ha poi trovato riscontro nei risultati prodotti.

II. Background

Il **VHDL** (acronimo di *VHSIC Hardware Description Language*, dove "VHSIC" è la sigla di *Very High Speed Integrated Circuits*) e **Verilog** sono linguaggi di descrizione hardware (in lingua inglese *Hardware Description Language*, in acronimo "HDL"), ovvero, dei linguaggi di programmazione specializzati per *descrivere* la stuttura ed il comportamento dei circuiti digitali. Gli HDL sono, inoltre, anche utilizzati per *simulare* il sistema e controllarne la risposta.

Il **SystemC**, invece, è un insieme di librerie e macro che introduce in ambiente C++ i concetti tipici della descrizione hardware quali segnali, moduli, porte, concorrenza e molti altri ancora, nato ed utilizzato per la progettazione di sistemi embedded di alto livello, ovvero, a livelli di astrazione superiore.

In particolare, i software utilizzati per per la fase di modellazione hardware ed i concetti di HLS sono:

- **Xilinx Vivado**, per la modellazione, verifica e sintesi dei moduli progettati a livello RT;
- **Xilinx Vivado HL**, per la modellazione, verifica e sintesi, attraverso il processo di HLS, partendo da una descrizione ad alto livello sviluppata nei linguaggi SystemC e C++.

III. Struttura di un numero in virgola mobile

Lo standard ANSI/IEEE Std 754-1985 specifica il formato e i metodi per l'aritmetica in virgola mobile nei sistemi informatici. Questo, inoltre, specifica le eventuali condizioni di errore, eccezione, e, la loro gestione predefinita.

Considerando il livello di dettaglio dello standard, e quindi l'impossibilità di una implementazione fedele, questa sezione risulta essere necessaria per la spiegazione delle scelte relativamente al formato e le condizioni di eccezione utilizzate in questo progetto. Si è considerato, quindi:

1) La rappresentazione di **numeri finiti** descritti da:
   - Un bit di **segno** S
   - Un campo per l'**esponente** E
   - Un campo per la **mantissa** M.
2) **Due valori di zero**, chiamati zeri firmati: il bit di segno specifica se uno zero è $+0$ (zero positivo) o -0 (zero negativo).
3) **Due valori di infinito**: $+\infty$, $-\infty$.
4) **Un valore di NaN**: acronimo di *Not a Number*, è utilizzato per segnalare che non si può rappresentare in forma numerica ciò che si vorrebbe. Ad esempio, in questo caso, lo standard ANSI/IEEE Std 754-1985 prevede la presenza di *due* (e non uno) diversi valori di NaN: *qNaN* e *sNaN*.

Il valore del numero rappresentato è calcolabile come:

$$(-1)^S \times 2^E \times M$$

Il campo S specifica il segno del numero: 0 per i numeri positivi, 1 per i numeri negativi.

Il campo E contiene l'esponente del numero in forma intera rappresentato in eccesso 127 (*polarizzazione* o *bias*). Essendo costituito da 8 bit e, considerando che i valori 0 e 255 sono utilizzati per funzioni speciali, l'intervallo di rappresentazione è $[-127, 128]$.

I restanti 23 bit sono utilizzati per la definizione della mantissa M, ovvero, la sequenza utilizzata per rappresentare le cifre dopo la virgola. Convenzione vuole che la prima cifra significativa si trovi *sempre* immediatamente a sinistra del punto decimale. Questo, si ottiene, aumentando o diminuendo il valore dell'esponente di tante unità quante sono le posizioni di cui si è spostato il punto. A seconda del valore dell'esponente abbiamo due diverse forme di rappresentazione di un numero: *normalizzati* e *denormalizzati*.

Un numerale si intende in **rappresentazione normalizzata** quando $E \neq 00000000$. In questa rappresentazione, la mantissa è normalizzata tra: $1 \leq M < 2$, quindi, sempre nella forma $1.XXXXXX...X$. L'1 prima della virgola è implicito e si utilizzano tutti i 23 bit per rappresentare la sola parte frazonaria. Gli intervalli rappresentabili sono pertanto: $(-2^{128}, -2^{-126}][2^{-126}, 2^{128})$.

Un numerale si intende in **rappresentazione denormalizzata**, invece, quando $E = 00000000$. In questa rappresentazione, la mantissa è normalizzata tra: $0 \leq M < 1$, quindi, sempre nella forma $0.XXXXXX...X$. Si utilizzano tutti i 23 bit per rappresentare la sola parte frazonaria e la più piccola mantissa vale 2^{-23}. Gli intervalli rappresentabili sono pertanto: $(-2^{-126}, -2^{-149}][2^{-149}, 2^{-126})$.

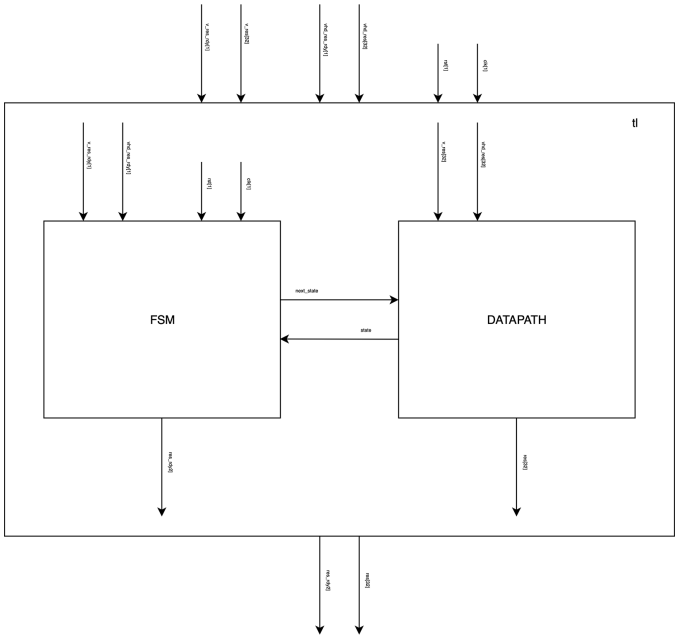

Figura 3. Scomposizione del modulo `top_level` in FSM e datapath.

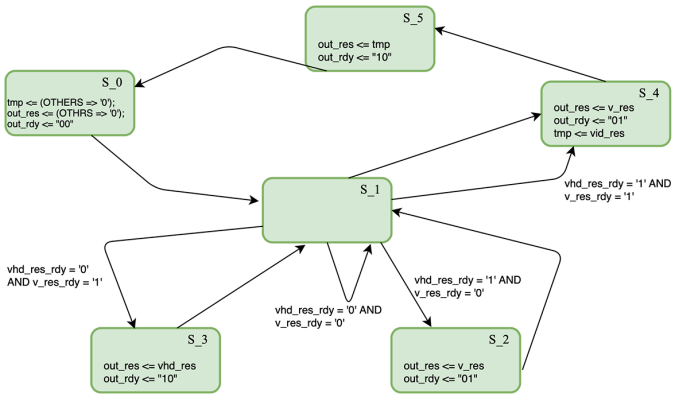

Figura 4. EFSM del modulo `top_level`.

Lo standard definisce, inoltre, cinque differenti regole di arrotondamento. Le prime due arrotondano ad un valore "più vicino", mentre, le restanti tre, invece, sono chiamate arrotondamenti diretti. La metodologia che si è deciso di implementare all'interno di questo progetto, fra le cinque a disposizione, è nello specifico la **round to nearest, ties away from zero**, che, consiste nell'arrotondare il numero al valore più vicino "sopra".

IV. Metodologia applicata

Il sistema da progettare segue diverse fasi prima del suo completamento nella versione finale.

A. Implementazione

L'implementazione a livello RT (VHDL e Verilog) dei moduli `multiplier754_vhd`, `multiplier754_v` e `top_level` segue la scomposizione in **FSM** (acronimo di

Finite State Machine), per il controllo del flusso del programma e, **datapath** per l'elaborazione dati. In Figura 3 la scomposizione del modulo `top_level` in FSM e datapath.

Di seguito, l'interfaccia del modulo `multiplier754_v` (l'implementazione VHDL presenta i medesimi ingressi ed uscite) e del `top_level`:

```
module multiplier754_v(
    input clk,
    input rst,
    input in_op1,
    input in_op2,
    input in_rdy,

    output out_res,
    output out_rdy
    );
    // Inputs.
    wire clk;
    wire rst;
    wire [SIZE - 1:0] in_op1;
    wire [SIZE - 1:0] in_op2;
    wire in_rdy;
    // Outputs.
    reg [SIZE - 1:0] out_res;
    reg out_rdy;
endmodule
```

In particolare, tutte le interfacce presentano due ingressi a 32 bit ciascuno per i due fattori e, un'uscita, sempre a 32 bit, per il prodotto. Sono presenti, inoltre, i segnali `in_rdy` ed `out_rdy` per l'implementazione dell'*handshake protocol* atto a gestire la comunicazione fra ingressi ed uscite, `rst` per il reset e `clk` per il clock. Per quanto riguarda il modulo `top_level`, nello specifico, è necessario per il corretto funzionamento serializzare l'output in differenti cicli di clock e, poter direzionare l'input tra i due moltiplicatori.

```
ENTITY tl IS
    PORT(
        -- Inputs.
        clk     :   IN std_logic;
        rst     :   IN std_logic;
        in_op1  :   IN
        std_logic_vector(SIZE-1 DOWNTO 0);
        in_op2  :   IN
        std_logic_vector(SIZE-1 DOWNTO 0);
        in_rdy  :   IN
        std_logic_vector(1 DOWNTO 0);
        -- Outputs.
        out_res :   OUT
        std_logic_vector(SIZE-1 DOWNTO 0);
        out_rdy :   OUT
        std_logic_vector(1 DOWNTO 0)
    );
end tl;
```

Per quanto concerne, invece, la fase di progettazione, i due processi `FSM` e `datapath` dei due moltiplicatori sono implementati secondo lo schema rappresentato nella EFSM in Figura 6, mentre, quelli relativamente al modulo del `top_level` sono sono sviluppati secondo lo schema rappresentato nella EFSM in Figura 4. In particolare, il processo che implementa la FMS è sensibile ai segnali `clk` e `rst` e si occupa di aggiornare lo stato prossimo del sistema, mentre, invece, il processo che implementa il datapath, è sensibile al segnale `next_state` e si occupa di eseguire le operazioni logico-aritmetiche definite dall'algoritmo, infine, di scrivere il risultato sulle porte di uscita.

L'implementazione per l'HLS (SystemC e C++), invece, avviene in modo **algoritmico**, quindi, senza alcuna netta suddivisione tra controllo e calcolo, con il principale vantaggio di avere quindi una strutturazione di progetto più "semplice" partendo da file sorgenti C++.

L'implementazione in SystemC segue la gerarchia che vuole il file composto da una classe definita usando la macro `SC_MODULE` all'interno della quale sono dichiarate le porte di ingresso e di uscita. Successivamente, viene diciarata la funzione o, le funzioni che "svlogono il lavoro". Le porte di input e output includono i metodi `read()` and `write()` per consentire le operazioni di lettura e scrittura. Infine, c'è un costruttore per l'istanza di `SC_MODULE`. SystemC fornisce un modo abbreviato per farlo, usando la macro `SC_CTOR`. Si utilizza, poi, `SC_METHOD` per far si che le semplici funzioni C++ adibite all'elaborazione si comportino come processi e, inoltre, si rendono queste sensibili ai loro input. Di seguito, l'interfaccia del file `multiplier754_sc.cpp`:

```
SC_MODULE(multiplierIEEE754_sc)
{
    // Inputs.
    sc_in_clk               clk;
    sc_in< sc_logic >       rst;
    sc_in< sc_lv<SIZE> >    in_op1;
    sc_in< sc_lv<SIZE> >    in_op2;
    sc_in< sc_logic >       in_rdy;
    // Outputs.
    sc_out< sc_lv<SIZE> >   out_res;
    sc_out< sc_logic >      res_rdy;
    // Do the multiplication.
    void makeMul()
    {
        ...
    }
    SC_CTOR(multiplierIEEE754_sc)
        : in_op1("in_op1_sc")
        , in_op2("in_op2_sc")
        , in_rdy("in_rdy_sc")
        , clk("clk_sc")
        , rst("rst_sc")
        , out_res("res_sc")
        , res_rdy("res_sc")
    {
        SC_METHOD(makeMul);
        sensitive_pos << clk;
        sensitive << rst;
    }
};
```

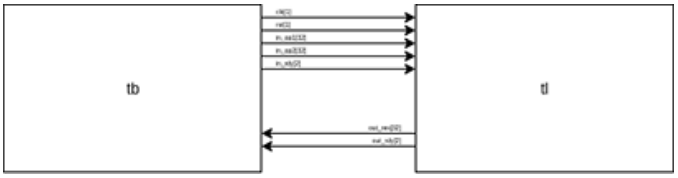

Figura 5. Interazione tra il TB ed il DUT.

L'implementazione in C++, invece, risulta essere quella più beve ed immediata. Essa consiste in una semplice funzione che, presi in input due numeri, ne calcola il prodotto e restituisce il risultato. Ovvero, un semplice programma, o, porzione di tale, che rispetti l'insieme delle regole definite dalla sintassi C++. Di seguito, ne viene riportata una possibile implementazione:

```
void multiplier_754(float op1,
                    float op2,
                    float *res)
{
    *res = op1 * op2;
}
```

B. Simulazione

Per verificare che un componente si comporti secondo le specifiche del progetto lo si sottopone ad una serie di prove, quindi:

- Si forzano in ingresso stimoli controllati;
- Si verifica che le uscite corrispondano.

Un **testbench** è un modulo HDL utilizzato per testare un altro modulo, chiamato DUT (acronimo di *Device Under Test*). L'interazione fra i due moduli è mostrata in Figura 5. In primo luogo, quindi, si predispone un opportuno modulo che descriva il test da eseguire. Le caratteristiche di questo sono la l'assenza di porte di ingresso o di uscita e, il fatto che, se le configurazoni dei valori in ingresso sono troppo numerose il test viene limitato alle configurazioni più significative.Il testbench ingloba al proprio interno l'entità sottoposta a verifica (DUT) e contiene la descrizione degli stimoli da applicare.

Per riuscire a simulare il modello sviluppato a livello RT è poi necessario utilizzare il software Xilinx Vivado citato precedentemente. Nel momento in cui si esegue il comando per la simulazione, Vivado, inanzitutto, analizza e compila il progetto in un'istantanea di simulazione. Poi, se tale processo è andato a buon fine, viene effettivamente eseguita la simulazione. Nell'IDE di Vivado si apre quindi, infine, l'interfaccia grafica del simulatore come mostrato in Figura 7 che riporta i risultati della simulazione effettuata.

Per simulare le implementazioni a livello algoritmico, invece, basta compilare il codice `.cpp` facendo uso di un compilatore (es. `g++`). Si traducono così le istruzioni scritte in un linguaggio di programmazione ad alto livello in linguaggio macchina, producendo codice oggetto, questo, poi, viene passato al linker che genera un file eseguibile.

C. Sintesi

La sintesi è il processo di trasformazione di un progetto specificato a livello RT in una rappresentazione a livello di gate. L'IDE di Vivado include un ambiente di sintesi e implementazione che facilita il flusso del processo per mezzo di un'interfaccia grafica consentendo così, ad esempio, ripetuti tentativi di esecuzione con diverse versioni di sorgenti di Register Transfer Level (RTL), dispositivi di destinazione, opzioni di sintesi o di implementazione e vincoli fisici o temporali. Inoltre, è possibile monitorare l'avanzamento della sintesi o dell'implementazione, visualizzare i rapporti di log e annullare le esecuzioni.

Quindi, una volta caricato il file e impostata la piattaforma di destinazione `PYNQ xc7z020clg400-1` è possibile lanciare il processo sintesi.

D. Sintesi ad alto livello

La sintesi ad alto livello (HLS), è un processo automatico di progettazione che, interpreta una descrizione algoritmica, e, crea un hardware che implementa tale descrizione.

Per generare il risultato finale è stato necessario utilizzare il tool Xilinx Vivado HLS. Il processo di HLS inizia con una prima analisi del codice, questo, poi, "programmato" per essere trasformato in un design a livello RT in linguaggio VHDL e Verilog, che, a sua volta, è a sintetizzato a livello di gate.

Quindi, anche in questo caso, una volta caricato il file e impostata la piattaforma di destinazione `PYNQ xc7z020clg400-1` è possibile lanciare il processo di HLS.

V. Risultati

Confrontando risultati ottenuti dal processo di sintesi e di HLS, relativamente alla latenza e all'area occupata dal design, possiamo dire che le versioni dei moduli sviluppati a livello RT risultano essere migliori delle versioni dei moduli sviluppati a livello algoritmico sia in termini di latenza che di area.

Di seguito, si riportano i risultati ottenuti:

- In Figura 9 il risultato del processo di sintesi relativamente al modulo `multiplier754_vhd`, impostando 4ns come minimo *Clock Period* facendo uso dei *Timing Constraints*.
- In Figura 10 il risultato del processo di sintesi relativamente al modulo `multiplier754_v`, impostando 3ns come minimo *Clock Period*.
- In Figura 11 il risultato del processo di sintesi relativamente al modulo `top_level` contenente i due moltiplicatori, impostando 4ns come minimo *Clock Period*.
- In Figura 12 il risultato del processo di HLS relativamente al file `multiplier754_algorithmic.cpp`, ottenendo come risultato 5.702ns come tempo stimato *Clock Period* con una incertezza di 1.25ns.
- In Figura 8 il risultato del processo di HLS relativamente al file `multiplier754_systemc.cpp`, ottenendo come risultato 5.702ns come tempo stimato *Clock Period* con una incertezza di 1.25ns.

VI. Conclusioni

In conclusione, si può desumere che, l'obiettivo di HLS è quello di permettere ai progettisti di hardware di costruire e verificare l'hardware in modo efficiente, dando loro un migliore controllo sull'ottimizzazione della loro architettura di design, e attraverso la natura di permettere al progettista di descrivere il design a un livello di astrazione più alto, mentre lo "strumento" fa l'implementazione RTL. Progettare hardware a livello RT, invece, risulta essere decisamente più "difficile" poiché, appunto, non si dispone di tutta la versalità, e gli strumenti messi a disposizione dai linguaggi ad alto livello, questo però, con vantaggi in termini ottimizzazione e affidabilità del progetto hardware.

Questo progetto mi ha permesso di capire come un codice automatico generalmente si presenta essere molto più complesso, impegnativo da gestire e mantenere, e soprattutto molto più oneroso dal punto di vista dell'utilizzo delle risorse hardware, dato che partendo da un codice ad alto livello, si hanno informazioni più generiche sul modello da costruire. Questo è difatti uno dei motivi che fanno si che ancora oggi, nonostante i diversi tool in commercio, l'HLS risulti essere poco utilizzata.

Il modo migliore di procedere, dunque, per costruire un circuito digitale risulta essere quello percorso in questo documento, ovvero passare da una versione VHDL/Verilog più dettagliata, verificando una iniziale correttezza del componente prodotto con primi test diretti su hardware programmabile, poi, prima di un eventuale impiego su larga scala, sviluppando un progetto ad alto livello in SystemC/C++ verificare l'hardware in modo efficiente.

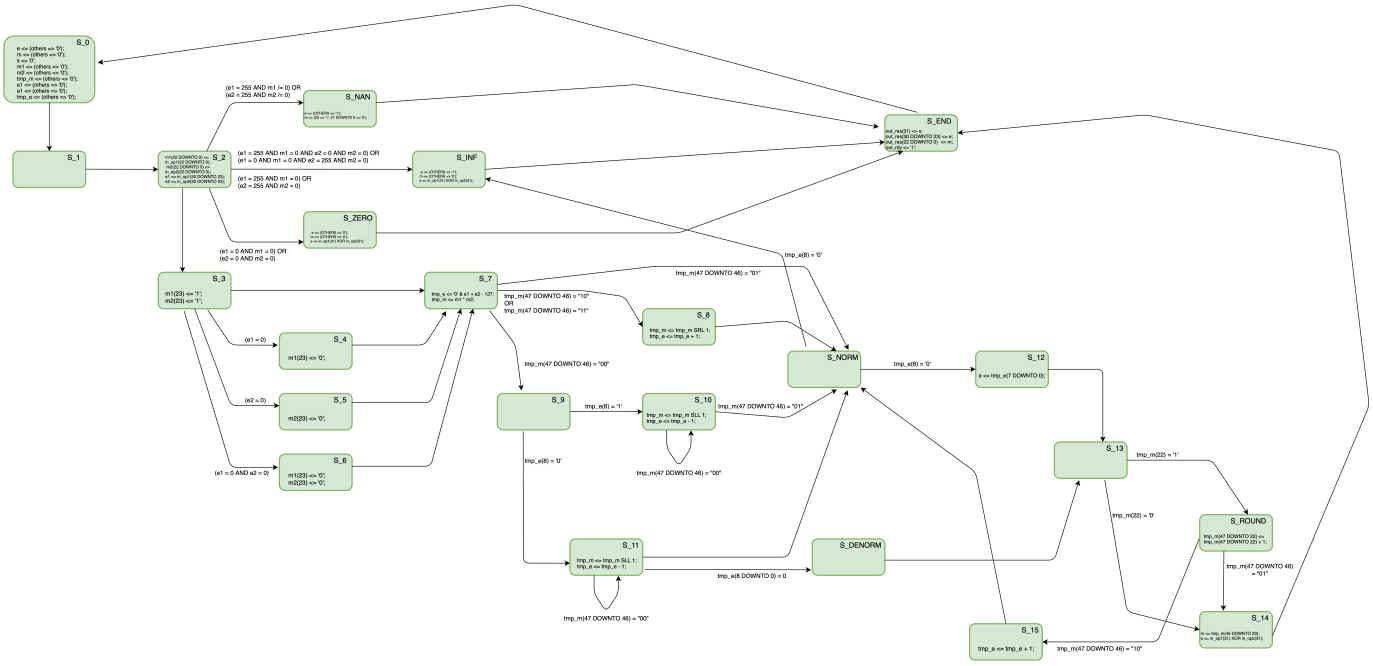

Figura 6. EFSM del modulo `multiplier754_v` e `multiplier754_v`.

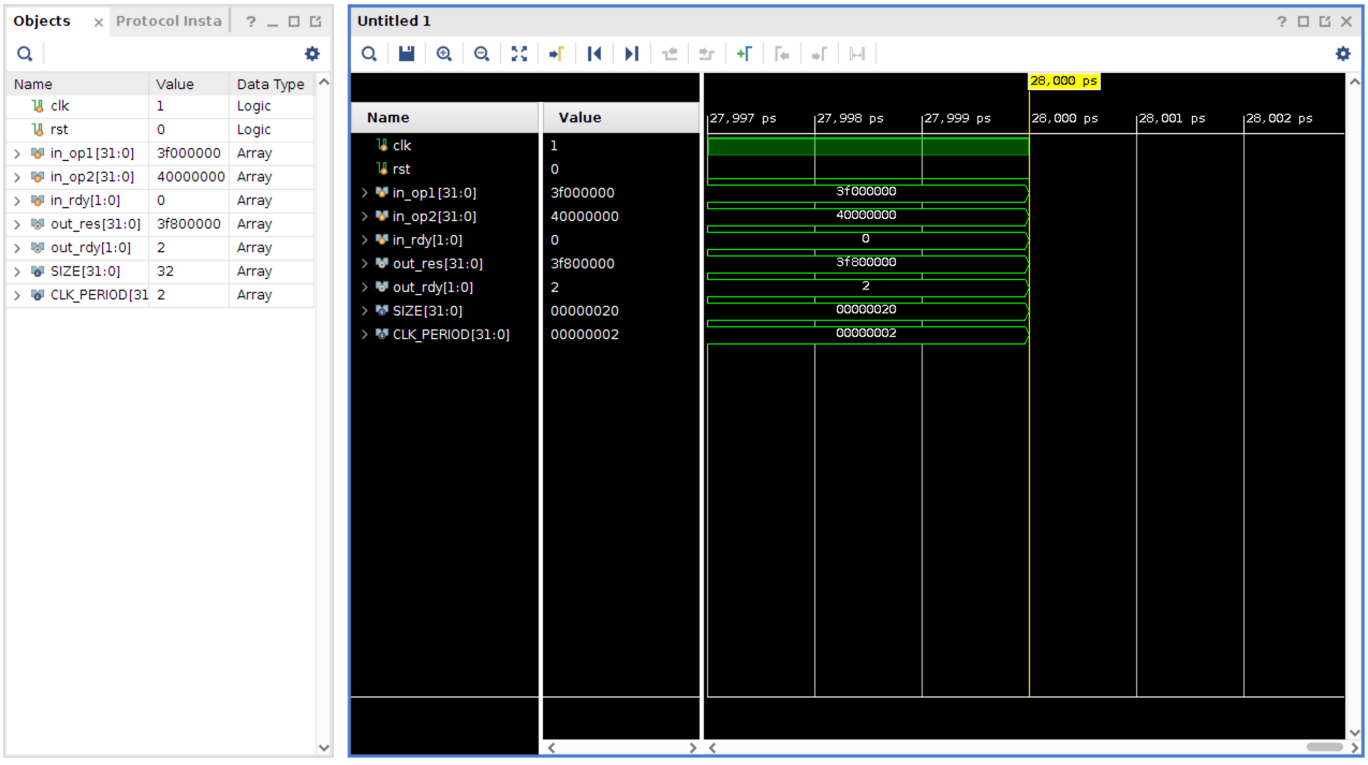

Figura 7. Simulazione del modulo `top_level`.

- **Timing**
  - **Summary**

Clock	Target	Estimated	Uncertainty
ap_clk	10.00 ns	5.702 ns	1.25 ns

- **Latency**
  - **Summary**

Latency (cycles)		Latency (absolute)		Interval (cycles)		Type
min	max	min	max	min	max	
1	4	10.000 ns	40.000 ns	1	4	none

Figura 8. Risultato dell'HLS del file `multiplier754_systemc.cpp`.

Design Timing Summary

Setup		Hold	
Worst Negative Slack (WNS):	0.778 ns	Worst Hold Slack (WHS):	0.488 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	10	Total Number of Endpoints:	10

Figura 9. Risultato della sintesi del modulo `multiplier754_vhd`.

Design Timing Summary

Setup		Hold	
Worst Negative Slack (WNS):	0.306 ns	Worst Hold Slack (WHS):	0.353 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	10	Total Number of Endpoints:	10

Figura 10. Risultato della sintesi del modulo `multiplier754_v`.

Design Timing Summary

Setup		Hold	
Worst Negative Slack (WNS):	0.455 ns	Worst Hold Slack (WHS):	0.212 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	24	Total Number of Endpoints:	24

Figura 11. Risultato della sintesi del modulo `top_level`.

- **Timing**
  - **Summary**

Clock	Target	Estimated	Uncertainty
ap_clk	10.00 ns	5.702 ns	1.25 ns

- **Latency**
  - **Summary**

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

Figura 12. Risultato dell'HLS del file `multiplier754_algorithmic.cpp`.