Modellazione e Sintesi di un Moltiplicatore Floating-point Single Precision

Nicola Serlonghi - VR445270

***Sommario*—Questo documento presenta l'implementazione di un sistema per il calcolo della moltiplicazione in virgola mobile a singola precisione realizzato con l'ausilio di due linguaggi HDL: System Verilog e VHDL. Il risultato è stato poi comparato con la High Level Synthesis di un moltiplicatore floating point in C.**

I. Introduzione

Il sistema che ho realizzato permette di eseguire il calcolo di due moltiplicazioni in virgola mobile secondo lo standard IEEE754.

In figura 1 si può vedere una rappresentazione astratta del sistema, che è diviso in due parti:

- Top level: si occupa di gestire gli input e gli output e di sincronizzare i due moltiplicatori. Esso è stato testato attraverso una testbench progettata per simulare un ambiente esterno.
- Moltiplicatori: si occupano di eseguire il prodotto in virgola mobile.

Per poter scoprire e toccare con mano le differenze dei diversi livelli di astrazione, il sistema è stato sviluppato utilizzando diversi linguaggi, nello specifico troviamo una versione che prevede un moltiplicatore scritto in VHDL ed uno in Verilog con un top level ed una testbench sempre scritti con quest'ultimo. Tutto quanto è stato scritto anche in SystemC, come già detto in precedenza, per avere un indice di paragone delle prestazioni e delle difficoltà di implementazione ai vari livelli di astrazione.

II. Background

Nella sezione precedente sono stati nominati VHDL e Verilog, entrambi sono linguaggi di descrizione dell'hardware (HDL), cioè sono dei linguaggi specifici usati per descrivere in modo formale la struttura e il comportamento di un circuito elettrico o logico digitale. Grazie a quest'ultima caratteristica è possibile eseguire un'analisi ed una sintesi automatica del circuito, inoltre è anche possibile simulare processi concorrenti.

Questi due linguaggi, insieme a SystemC che è un insieme di librerie di C++, operano nel nostro caso ad un livello di descrizione RTL (register transfer level). L'altra versione implementata, che permette invece di effettuare la HLS (high level synthesis), è stata scritta in C++ e poi sintetizzata utilizzando Vivado HLS. Questo tipo di sintetizzazione interpreta una descrizione algoritmica creando l'implementazione in hardware digitale.

III. Metodologia applicata

Dopo aver analizzato minuziosamente le specifiche richieste, ho deciso di disegnare il sistema partendo da una EFSM. Ho preferito questo tipo di sviluppo in quanto offre la possibilità di capire molto bene i flussi di calcolo che il sistema dovrà compiere ed in una seconda fase di revisione, apportare dei miglioramenti come la minimizzazione del numero di stati o di transazioni.

Una volta ottenuta la EFSM definitiva, ho iniziato lo sviluppo vero e proprio dando priorità alla parte riguardante il livello RTL, in particolare il moltiplicatore in VHDL. Dunque sono andato a scrivere quest'ultimo proseguendo poi con la creazione di una testbench minimale per sincerarmi della corretta operatività del componente e quindi della EFSM creata in precedenza. Solo dopo aver ottenuto dei risultati positivi e aver svolto i dovuti raffinamenti, ho implementato il moltiplicatore speculare scritto in Verilog, seguito poi da un top level e da una testbench stavolta completa e conforme alle specifiche date, scritte anch'esse col medesimo linguaggio. Come ultima cosa prima di spostare la mia attenzione sulle le altre parti dell'elaborato, ho testato il sistema nella sua interezza svolgendo dei test più esaustivi con l'ausilio di file contenenti degli input più vari possibili in modo da non lasciare zone d'ombra.

Sono passato dunque a sviluppare la parte RTL, andando a sfruttare nuovamente la EFSM già definita, raffinata e testata al punto precedente creando però questa volta una implementazione basata su SystemC RTL, trovandomi anche in questo caso dopo aver svolto nuovamente vari test, con una grammatica sintetizzabile.

Infine, ma non per importanza, ho rivolto le mie attenzioni alla parte HLS del progetto, implementando in C++ un algoritmo che svolge la moltiplicazione in virgola mobile.

In relazione a quanto detto in precedenza, si può osservare che l'architettura del progetto è rimasta invariata per VHDL, Verilog e SystemC e verrà descritta in dettaglio:

A. Moltiplicatore

Questo componente si occupa di svolgere la moltiplicazione in virgola mobile rispettando lo standard IEEE754. L'interfaccia è visibile in figura 2 ed è composta da 5 inputs e 2 outputs:

- $[op1]$ è il primo operando.
- $[op2]$ è il secondo operando.
- $[in_rdy]$ dice al modulo quando $op1$ e $op2$ sono pronti per eseguire la moltiplicazione.

- $[res]$ è il risultato della moltiplicazione.
- $[res_rdy]$ dice quando il (res) è pronto.
- $[rst]$ resetta il modulo.
- $[clk]$ è il segnale di clock.

Gli inputs devono essere già dei valori normalizzati, altrimenti verrà ritornato res con tutti i valori posti a 0. All'interno del modulo Verilog troviamo 9 registri per gestire i dati al meglio durante la moltiplicazione, mentre in VHDL troviamo 9 segnali con il medesimo scopo.

Entrambi i moduli sono al loro interno modellati su due processi: fsm e datapath. Il primo gestisce lo stato corrente ed il prossimo stato ed è guidato dallo stato, mentre il secondo gestisce la parte di calcolo ed è guidato dal clock.

La combinazione di entrambe le parti va a creare la EFSM che è possibile visionare in figura 3 ed è qui descritta brevemente:

- [ST_0] Stato di inizializzazione e reset.
- [ST_1] La macchina sta in questo stato finche gli input non sono pronti.
- [ST_2] Smista in base al tipo di input.
- [ST_ZERO] Gestisce il valore zero in input.
- [ST_INF] Gestisce il valore infinito in input.
- [ST_NAN] Gestisce gli input non validi.
- [ST_3] Gestisce la normalizzazione e la denormalizzazione degli input.
- [ST_4] Controlla se la mantissa è normalizzata e agisce di conseguenza
- [ST_5] Normalizza il valore.
- [ST_6] Controlla l'overflow dell'esponente dopo aver fatto shift normalization.
- [ST_7] Gestisce l'overflow durante la moltiplicazione.
- [ST_8] Gestisce l'underflow.
- [ST_9] Salva il nuovo esponente calcolato.
- [ST_10] Controlla se c'è bisogno di arrotondare.
- [ST_11] Salva la nuova mantissa calcolata.
- [ST_ROUND] Fa l'arrotondamento.
- [ST_12] Salva il nuovo esponente calcolato.
- [ST_NORM] Viene raggiunto quando ho un input normalizzato.
- [ST_DENORM] Viene raggiunto quando ho un input denormalizzato.
- [ST_OUT] Restituisce in output il risultato e ritorna allo stato ST_0.
- [ST_ERR] Restituisce in output tutti zero per far vedere che c'è stato un errore con i valori di input.

B. Top level

Il top level, scritto in Verilog, ha il compito di sincronizzare i due moltiplicatori (VHDL e Verilog). Esso accetta in input due valori alla volta, che possono essere poi inviati ad entrambi i moltiplicatori, andando a settare in_rdy a 11, oppure possono essere inviati prima al moltiplicatore Verilog e successivamente al moltiplicatore VHDL, settando in_rdy a 01 o 10.

Avendo dunque due moltiplicatori avremo anche due risultati; ho deciso quindi una sola uscita (res) per l'output del risultato, affiancata da un bit (res_rdy) che mi indica se il risultato proviene dalla prima o dalla seconda macchina.

Per gestire tutto ciò anche il top level è suddiviso in due processi: FSM e Datapath; insieme vanno a formare la EFSM visionabile in figura 4 e qui brevemente descritta:

- [ST_0] Stato di inizializzazione e reset.
- [ST_1] Smista in base al tipo di input.
- [ST_2] Serializza in uscita il risultato di VHDL.
- [ST_3] Serializza in uscita il risultato di Verilog
- [ST_4] Entrambi i risultati sono pronti, salva quindi il risultato di VHDL e restituisce in output il risultato di Verilog.
- [ST_5] Restituisce in output il risultato di VHDL e ritorna allo stato ST_0 per accettare nuovi valori.

C. Testbench

La testbench è scritto in Verilog, la scelta è stata di dare in input gli stessi operandi ad i due i moltiplicatori, in questo modo vengono testati entrambi.

Il sistema sviluppato in SystemC è identico a quello in VHDL e Verilog quindi rimangono valide la EFSM e la suddivisione in 2 processi, FSM e Datapath, indicata in precedenza.

IV. Risultati

Il sistema VHDL/Verilog è stato testato utilizzando una serie di operandi scritti in un file di testo, la testbench si occupa di caricare i vari valori da file e darli in pasto ai due moduli e di stampare a video i due risultati ottenuti ed anche il risultato atteso, in modo da poter avere un riscontro immediato.

Per quanto riguarda i test della parte di SystemC, sono stati inseriti a mano gli operandi che si erano evidenziati, nei vari test precedenti, più problematici, lasciando poi l'onere al progettista di verificare la correttezza dei risultati ottenuti.

Riporto di seguito in forma tabellare i vari risultati ottenuti.

Resource	Estimation	Available	Utilization
LUT	224	53200	0.42%
FF	174	106400	0.16%
DSP	2	220	0.91%
BUFG	1	32	3.13%

Tabella I
SINTESI MOLTIPLICATORE VERILOG.

Resource	Estimation	Available	Utilization
LUT	149	53200	0.28%
FF	152	106400	0.14%
DSP	2	220	0.91%
BUFG	1	32	3.13%

Tabella II
SINTESI MOLTIPLICATORE VHDL.

La sintesi del moltiplicatore Verilog produce i valori presenti in tabella I mentre nella tabella II sono presenti i valori del moltiplicatore VHDL. Come si può notare nel modulo scritto in Verilog c'è stato un incremento di LUT e FF.

Resource	Estimation	Available	Utilization
LUT	444	53200	0.82%
FF	395	106400	0.37%
DSP	4	220	1.82%
IO	102	125	81.60%
BUFG	1	32	3.13%

Tabella III
SINTESI DEL TOP LEVEL.

La tabella III riporta invece i valori di sintesi del top level. Si può vedere un incremento di tutti i valori e si può anche dedurre che il top level ha 174 netlist e 356 nets, il modulo Verilog ha 770 nets e 468 leaf cells contro i 737 e 437 del modulo VHDL.

Dopo aver eseguito l'implementazione più volte, abbassando di volta in volta il vincolo di clock, impostato in precedenza tramite Vivado, il numero più basso che ho ottenuto, su cui è possibile lavorare, è 9ns. Una volta effettuata l'implementazione è stato possibile analizzare i valori timing e power ottenuti. Per quanto riguarda il primo è possibile visualizzare i dati nella tabella IV, mentre per il secondo ho ottenuto una stima di implementazione di: 0.112 W come total power su chip con 26.4 C come temperatura di giunzione. L'energia utilizzata è del 14% dinamico con 0.018 W e 86% statico con 0.105 W.

Timing	WNS	TNS	WHS	THS	Tot. Endpoints
Setup	0.585ns	0.000	-	-	818
Hold	-	-	0.066ns	0.000ns	818

Tabella IV
SINTESI TIMING DEL MOLTIPLICATORE SYSTEMC.

A. *Confronto con HLS*

La sintesi HLS produce un interessante rapporto mostrato nella tabella V. L'utilizzo in Istance di FF e LUT sono la metà rispetto al top level sintetizzato. I 3 registri FF sono a 3 bit e la LUT a 4 bit.

Name	BRAM_18K	DSP48E	FF	LUT
Istance	-	3	128	138
Multiplexer	-	-	-	21
Register	-	-	3	-
Total	0	3	131	159
Available	270	240	82000	41000
Utilization (%)	0	1	∼0	∼0

Tabella V
TIMING DELL'ALGORITMO HLS IN C.

Clock	Target	Estimated	Uncertainty
ap_clk	10	8.286	1.25

Tabella VI
CLOCK DELL'ALGORITMO HLS IN C.

V. CONCLUSIONI

Valutando i risultati ottenuti da questo elaborato, posso dire che in linea generale la sintesi della semplice funzione in C++ risulta migliore di quella implementata in Verilog e VHDL. Sono molto soddisfatto di essere riuscito a cogliere, in modo molto più marcato, le differenze che ci sono nell'utilizzo dei vari linguaggi di descrizione dell'hardware.

APPENDICE

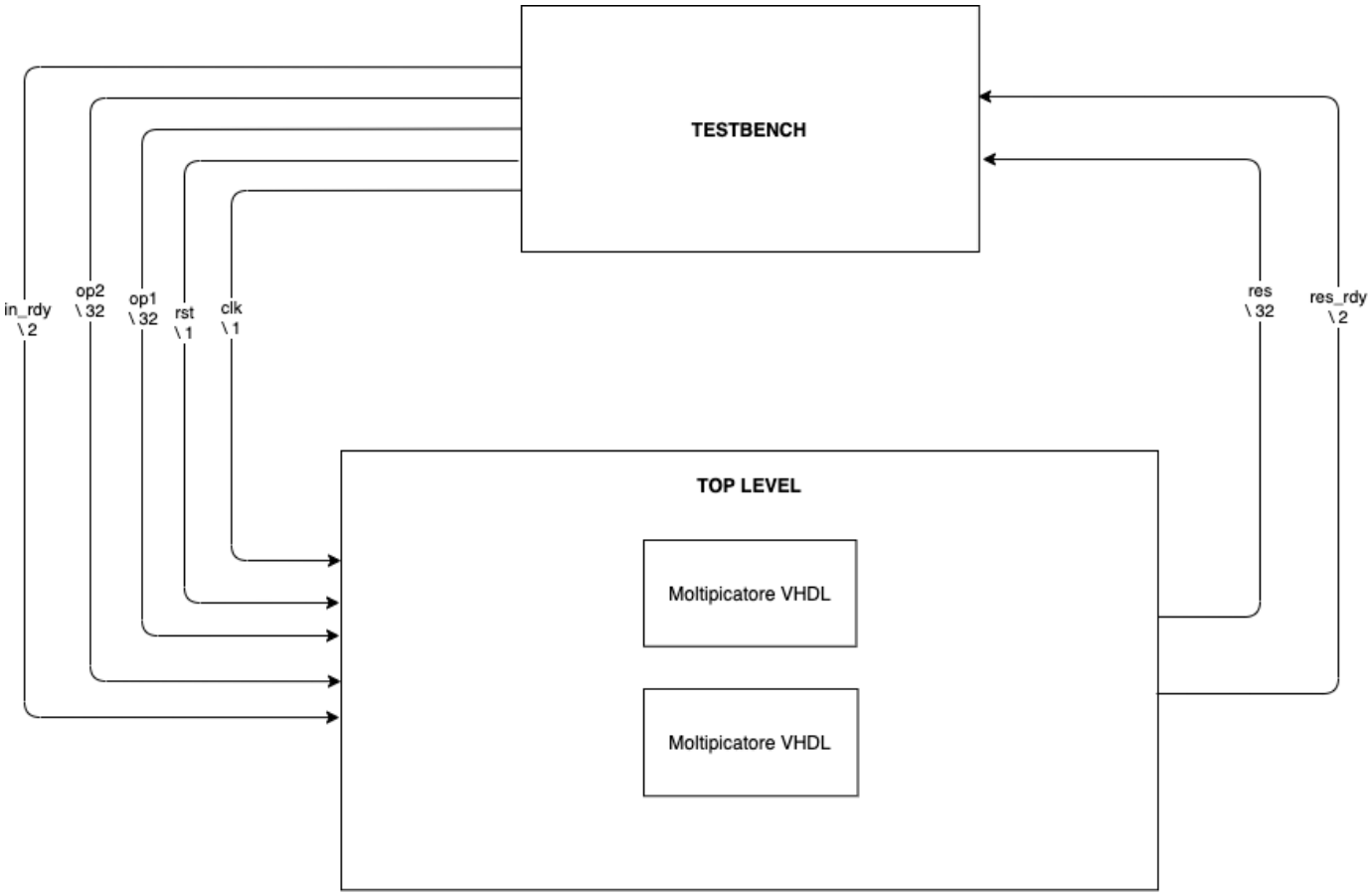

Figura 1. Sistema a livello astratto.

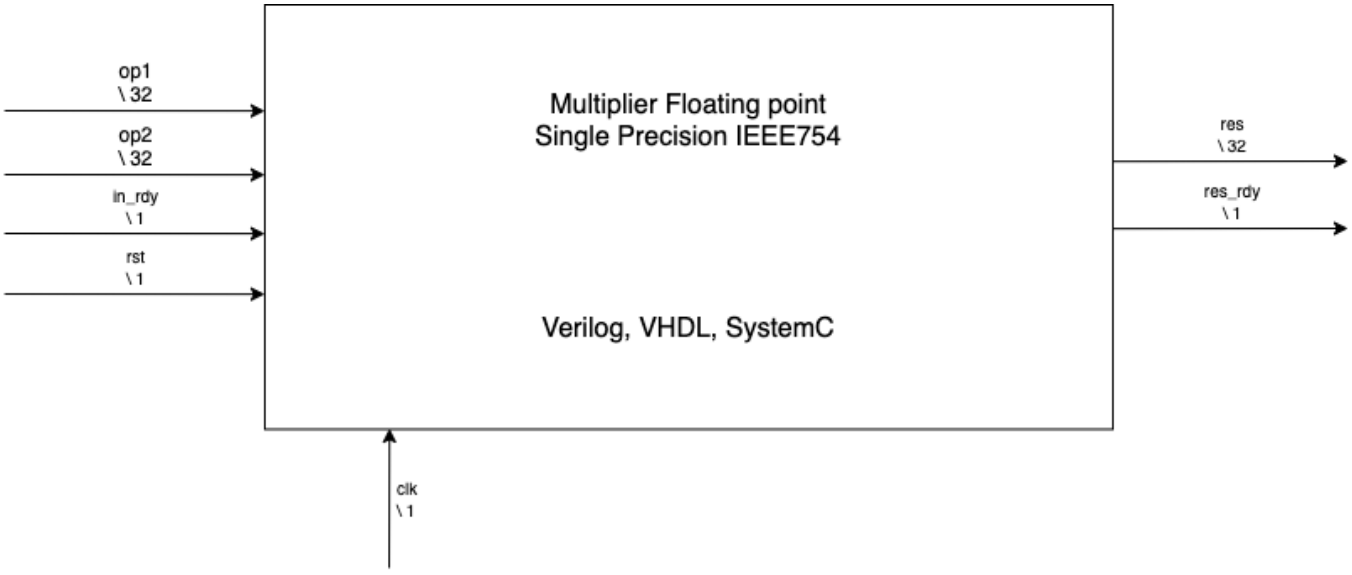

Figura 2. Schema dell'interfaccia del moltiplicatore.

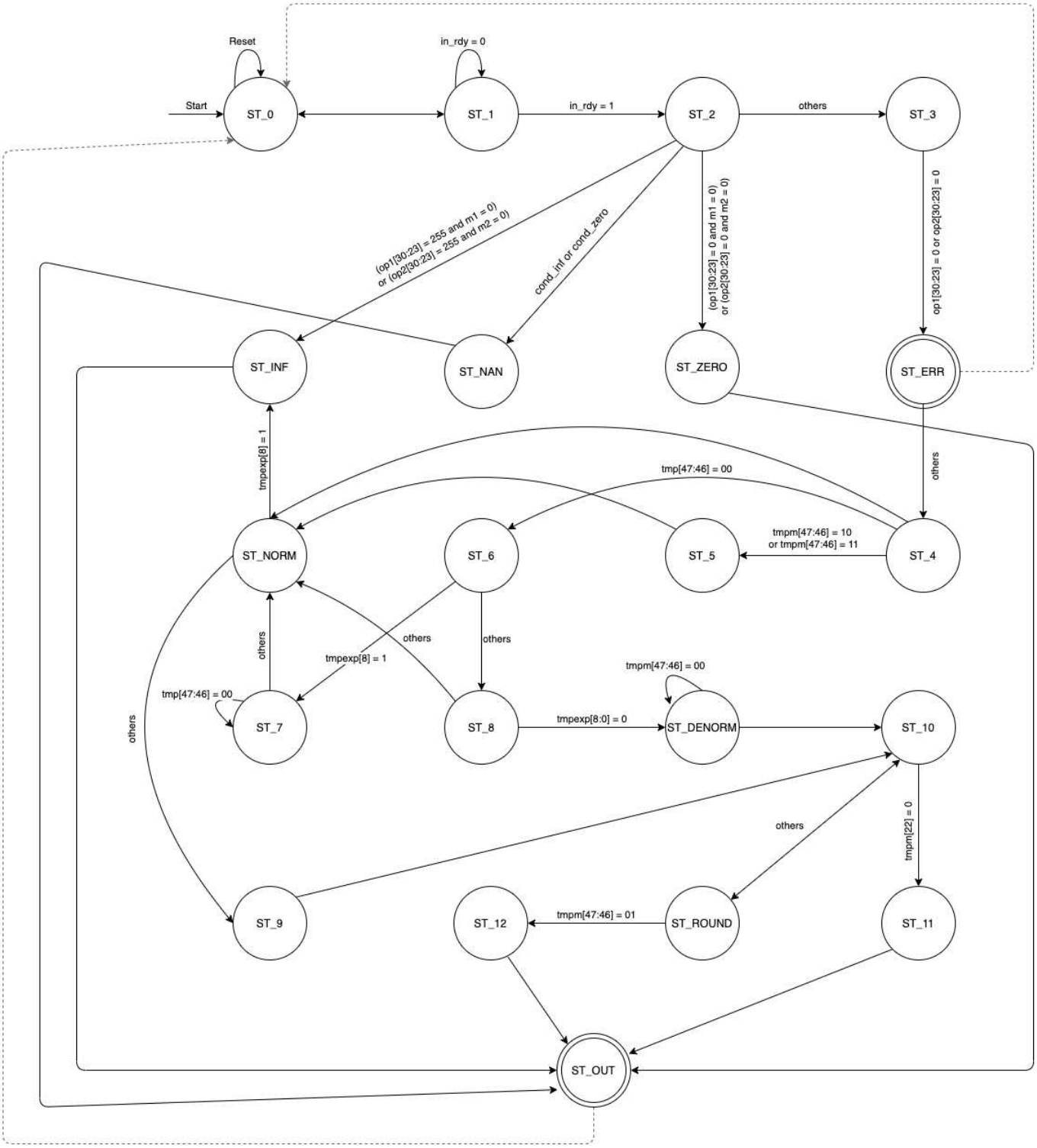

Figura 3. EFSM del multiplier.

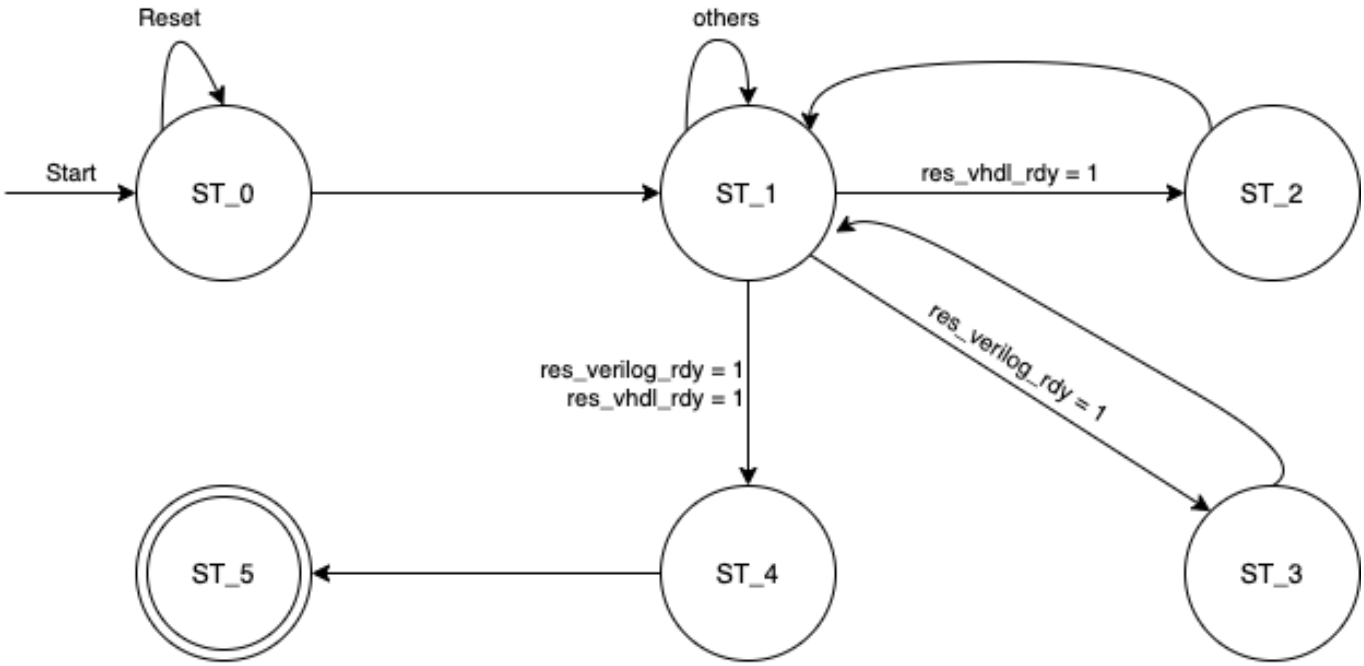

Figura 4. EFSM del top level.