Modellazione e Sintesi di un Moltiplicatore Floating-point Single Precision

Mirco De Marchi - VR445319

***Sommario*—Gli obiettivi del progetto sono modellare con diversi linguaggi di descrizione dell'hardware delle implementazioni di un moltiplicatore floating-point a singola precisione IEEE754 a diversi livelli di descrizione per essere sintetizzabile ed implementabile sulla *PYNQ-Z1 FPGA*. Modellare con un top level che gestisca l'input a due di questi moltiplicatori e serializzi l'output di questi, mantenendo come limite di porte di ingresso e uscita del modulo a 125. Eseguire high level synthesis di un moltiplicatore e confrontare i risultati di utilizzo delle risorse e di latenza. Gli obiettivi di questo report sono descrivere ed argomentare i risultati ottenuti dalla modellazione in hardware di un moltiplicatore floating-point.**

I. Introduzione

La modellazione in hardware di un moltiplicatore in floating-point singola precisione è stata descritta in 3 linguaggi: Verilog, VHDL e SystemC.

Per poter utilizzare il moltiplicatore floating-point (figura 2) bisogna inserire i due operandi e comunicare al modulo che l'input è pronto. Con l'output avviene al contrario: il modulo comunicherà quando il risultato della moltiplicazione floating-point è pronto e solo durante quel ciclo di clock sarà possibile recuperare tale valore.

Per quanto riguarda invece il top level (figura 1) è stata implementata la possibilità di inserire solo due operatori alla volta, per il primo e il secondo moltiplicatore, che dovranno essere inseriti su due cicli di clock differenti. Tuttavia c'è la possibilità di inserire nello stesso ciclo di clock gli stessi due operatori per ognuno dei due moltiplicatori floating-point. L'output è previsto serializzato su un'unica uscita. Ciò significa che il modello comunicherà quando è pronto un risultato della moltiplicazione floating-point e quale dei due moltiplicatori ha prodotto il corrispettivo risultato.

Le tecnologie utilizzate di sono rese utili per poter modellare hardware su più livelli, ovvero a livello RT e algoritmico, così da poter confrontare le diverse soluzioni, sia da un punto di vista simulativo, sia da un punto di vista del risultato finale.

L'attività che ha reso possibile la realizzazione di questo assignment si è svolta nei seguenti passi:

- Progettazione su carta delle EFSM che descrivono il flusso di operazioni del moltiplicatore floating-point e del top-level;
- Scrittura in SystemC del modello rappresentato a livello RT del moltiplicatore floating-point;
- Scrittura in SystemC a livello algoritmico di un moltiplicatore floating-point;
- Scrittura di un top level che gestisse i due moduli precedentemente fatti secondo la EFSM progettata;
- Scrittura di un testbench che simulasse contemporaneamente i due moltiplicatori, confrontandone il risultato e facendo simulazione di corner case;
- Una volta soddisfatto della simulazione, scrittura in VHDL e Verilog del moltiplicatore floating-point secondo la EFSM progettata;
- Scrittura in VHDL di un top level che gestisse i due moduli precedentemente sviluppati;
- Scrittura di un testbench in Verilog che seguisse lo stesso flusso di testing del testbench scritto in SystemC;
- Dopo opportuni test, sintesi e aggiunta di constraints sul periodo del clock;
- Implementazione del modello sintetizzato su FPGA mantenendo il constraits ottimale sul clock;
- Scrittura di codice in C++ di un moltiplicatore floating-point descritto a livello algoritmico;
- High level synthesis del modello algoritmico in SystemC, C++ e del modulo RTL in SystemC;
- Confronto dei report di utilizzo delle risorse e di latenza prodotti dalla sintesi ed implementazione a livello RT in Verilog e VHDl e dalla high level synthesis fatta invece dei progetti RTL e a livello algoritmico scritti in SystemC;
- Scrittura della relazione.

I risultati prodotti sono stati ottimo in temini di tempo. Inoltre SystemC si è dimostrato il sistema migliore per simulare un modello hardware, ma meno specifico per quando c'è invece da portarsi alla sintesi ed implementazione. Invece Verilog e VHDL si sono dimostrati ottimi per ottenere una sintesi hardware, ma meno efficienti per fare simulazione.

II. Background

Modellare e sviluppare hardware può essere un'operazione molto complicata. Pensare di descrivere hardware a collegamenti e porte logiche è improponibile, soprattutto se si parla di progetti di grosso calibro. Tuttavia già pensare di descrivere hardware come relazione fra una final state machine, un datapath e una gerarchia di moduli è già molto meglio. Però esistono dei metodi di sviluppo di hardware molto più veloci, che permettono testing e sintesi automatica, riducendo così il time to market.

L'HDL (Hardware Description Language) ovvero linguaggio di descrizione di hardware, permette di modellare secondo alcuni stili del codice detto behavioural che poi può essere trasformato in hardware. In particolare con un HDL si può descrivere a livello RT, cioè con una EFSM, ovvero FSM+datapath, oppure a livello algoritmico. Esistono diversi linguaggi di descrizione di hardware. In questo progetto è stato

utilizzato VHDL e Verilog per descrivere hardware a livello RT, e SystemC per fare modellazione di hardware sia a livello RT sia a livello algoritmico.

Quando si scrive codice di descrizione hardware è possibile poi sintetizzare, poichè si rispettino gli stili di definizione dei processi. In particolare quando si scrive correttamente una descrizione a livello RT, ovvero andando a progettare una EFSM, definita da 1 a 3 processi, è possibile sintetizzare, poichè vengono utilizzati una combinazione degli stili di processo. Tool di sintesi come Vivado, prodotto da Xilinx, permettono di sintetizzare ed arrivare a generare l'implementazione del proprio codice HDL su FPGA.

Inoltre tool come Vivado, permettono anche di testare, quindi simulare la propria descrizione in HDL. Questo processo di simulazione viene fatta per mezzo di un opportuno testbench che genera dei valori di input per il modulo implementato e controlla gli output.

Mentre quando viene scritto codice a livello algoritmico, come viene comunemente fatto in C++, ma che può essere fatto con apposite librerie come SystemC, ha bisogno di un High Level Synthesis (HLS). Questo processo può essere fatto su codice algoritmico scritto in C++ o SystemC, ma anche scritto in Verilog o VHDL. Il processo di High Level Synthesis esegue una serie di operazioni statiche sul codice, che prevede 3 fasi:

1) Ottimizzazione di codice;
2) Scheduling delle operazioni fatte nel codice;
3) Allocazione degli operatori;

In SystemC, in particolare, è possibile modellare e simulare assieme descrizioni a livello RT e descrizioni algoritmiche, così da permettere di abbassarsi a un livello di descrizione hardware, ma mantiene le flessibilità del C++ e tutti i suoi vantaggi, compreso il testing e debugging.

III. Metodologia Applicata

Per la modellazione a livello RT è stato utilizzato Verilog e VHDL, che tramite il tool di sintesi ed implementazione su FPGA fornito da Vivado, è stato possibile simulare l'implementazione RTL e realizzarla, ottenendo così informazioni di utilizzo delle risorse della FPGA selezionata, ovvero la *PYNQ-Z1*. In particolare in Verilog e VHDL sono stati fatti due moltiplicatori, definiti da un'unica EFSM, mentre il top level è stato fatto solo in VHDL.

In SystemC sono state modellate un'implementazione del moltiplicatore floating-point a livello RT, una a livello algoritmico e il relativo top level dei due moltiplicatore. In fine è stata scritta in C++ un'implementazione a livello algoritmico di un moltiplicatore floating-point con i tipi originali.

Sia in SystemC, che in VHDL e Verilog si è data come struttura quella rappresentata in figura 1, per poter simulare in modo efficiente tutti i moltiplicatori progettati. Per capire il processo e l'idea di base che c'è nel flusso delle operazioni nel moltiplicatore floating-point è utile seguire i passi descritti nella figura **??**. In seguito viene data una descrizione dettagliata su come funziona il moltiplicatore e come vengono gestiti da un top level due di questi.

Per gestire la comunicazione degli input e degli output è stato utilizzato l'handshake protocol. Nel caso del moltiplicatore flaoating-point vengono inseriti in input i due operandi a 32 bit e con un bit di input ready si comunica quando effettivamente gli operandi sono pronti per essere processati dalla macchina (a 1 vengono processati). L'output sarà pronto quando il bit di result ready lo comunicherà (ovvero quando è a 1) e solo in quel ciclo di clock si potrà acquisire il risultato della moltiplicazione degli operandi in floating-point.

La EFSM del moltiplicatore floating-point (figura 8) prima di tutto deve gestire la lettura dell'input, partendo da uno stato iniziale che resetta tutti i segnali e le porte di uscita, su cui ci rimane finchè l'input ready è a 0. Nell'ultimo stato della EFSM, quando il risultato è pronto, viene alzato ad 1 l'output di result ready, per poi ritornare nello stato iniziale. Il risultato pronto durerà un solo ciclo di clock, mentre per inserire l'input è necessario tenere a 1 l'input ready per un solo ciclo di clock, poi può tornare a 0.

La EFSM deve gestire vari rappresentazioni in floating-point, secondo lo standard IEEE754: normalizzati, denormalizzati, zero con segno, infinito con segno e due implementazioni di NaN. Le due implementazioni di NaN si verificano nel caso della moltiplicazione infinito e zero, e nel caso di NaN e un valore qualsiasi. Nel caso infinito moltiplicato zero, il risultato in uscita dovrà essere un NaN con i primi 10 bit più significativi a 1 e i rimanenti a 0, mentre nel caso NaN moltiplicato per un valore qualsiasi viene riportato nell'output esattamente il NaN in ingresso. Nella EFSM subito dopo che l'input è pronto vengono suddivisi gli operandi, partendo dai bit più significativi, in segno (1 bit)), esponente (8 bit) e mantissa (23 bit). Qui viene eseguito un controllo sugli input:

- NaN moltiplicato a qualsiasi valore fa NaN;
- Infinito moltiplicato a uno zero fa NaN;
- Zero moltiplicato a un qualsiasi valore fa zero;
- Infinito moltiplicato a un qualsiasi valore fa infinito;
- In tutti gli altri casi gli operatori sono floating-point normalizzati o denormalizzati e la computazione continua.

A questo punto se il rispettivo esponente dell'operando è tutto a 0 vuol dire che l'operando è un floating-point denormalizzato. Viene preso un segnale mantissa interno di grandezza 24 bit, così da poter mettere il bit più significativo a 0 se l'operando è denormalizzato, altrimenti se è normalizzato viene messo a 1. Dopodichè viene eseguita la moltiplicazione della mantissa e la somma degli esponenti ad eccesso 127 (ovvero sottraendo poi 127 dal risultato della somma). L'esponente risultante potrebbe essere andato in overflow, come in underflow. Per capirlo basterà vedere il carry bit dato dalla somma e se è a 1 allora sicuramente c'è stato un overflow o underflow. In tal caso se $exp1+exp2 < 127$ allora è in underflow, altrimenti è in overflow. La mantissa risultante invece sarà dal doppio dei bit e avrà come due bit più significativi la parte intera e tutto il resto la parte frazionaria. Ora viene fatta la normalizzazione della mantissa, ovvero se la parte intera della mantissa risultante è 01 allora è già normalizzata e non c'è da fare alcun calcolo, se è 10 oppure 11 basta shiftare di 1 a destra una volta per normalizzarla e se è a 00 bisogna shiftare a sinistra finchè non

è normalizzata. Ogni volta che si esegue uno shift a destra, l'esponente va incrementato, ogni volta che si esegue lo shift a sinistra l'esponente va decrementato. Tuttavia se quando viene fatto lo shift a sinistra l'esponente viene azzerato allora la fase di normalizzazione viene terminata poichè in questo caso il risultato sarà un valore floating-point denormalizzato.

Finita questa fase di normalizzazione si verifica se l'esponente è in overflow o underflow. Se è in underflow allora si fa lo shift a destra fino a che il carry bit dell'esponente torna a zero. Questa operazione viene fatta nella speranza di riuscire a trovare una forma denormalizzata del risultato. Nel caso peggiore il risultato sarà 0. Se invece è in overflow allora il risultato può solo che essere infinito.

Per tutti gli altri casi viene eseguito un'approssimazione. In questo caso viene fatta in modo molto semplice, guardando il primo bit che verrebbe scartato dalla mantissa risultato e se è a 1 allora viene sommato 1 a tutta la parte sinistra a partire da quel bit. Se viene fatta l'approssimazione bisogna riverificare se la mantissa è normalizzata, poichè nel caso limite in cui il risultato della moltiplicazione generasse tutti 1 nella parte della mantissa che bisognasse riportare nell'uscita, nel momento in cui si va a sommare 1, la mantissa va in overflow e va rinormalizzata. In fine viene riportato nell'uscita il segno che è uno XOR dei bit più significati degli operandi, l'esponente e la mantissa risultanti da tutte queste operazioni.

Per quanto riguarda invece il top level, viene sempre usato l'handshake protocol, tuttavia è necessario serializzare l'output in differenti cicli di clock e poter partizionare l'input nei rispettivi operandi dei due moltiplicatori floating-point. Per fare ciò il top level prende in input due operandi a 32 bit e un input ready da due bit per cui se abbiamo:

- 00: significa che i due operandi in input non sono pronti;
- 01: significa che i due operandi in input sono pronti per il primo moltiplicatore floating-point e che questo può iniziare a processare;
- 10: significa che i due operandi in input sono pronti per il secondo moltiplicatore floating-point e che questo può iniziare a processare;
- 11: significa che i due operandi in input vengono usati per entrambi i moltiplicatori floating-point e che dunque inizieranno a processare operandi uguali.

Per ottenere questo comportamento per gli input del top level non è necessario progettare una EFSM, ma è sufficiente collegare i due operandi in ingresso del top level agli operandi di input di entrambi i moltiplicatori floating-point, collegare il primo bit del input ready del top level con l'input ready del primo moltiplicatore e il secondo bit del input ready del top level con l'input ready del secondo moltiplicatore, facendo quindi un port map delle porte.

Mentre per l'uscita ci sarò un solo output come risultato da 32 bit e un result ready da due bit per capire quale moltiplicatore floating point ha prodotto il risultato:

- 00: non è pronto alcun risultato;
- 01: è pronto il risultato del primo moltiplicatore;
- 10: è pronto il risultato del secondo moltiplicatore;
- 11: caso mai verificabile;

In questo caso invece è necessario gestire questo comportamento con una EFSM. In pratica il top level (figura 3)non fa altro che aspettare nello stato iniziale fino a quando il risultato di uno dei due moltiplicatori floating-point notifica con il rispettivo result ready che il risultato è pronto, mettendo tutti gli output a zero. Dopodichè si muove in uno stato quando il primo moltiplicatore ha il risultato pronto oppure in un altro stato se il secondo moltiplicatore ha il risultato pronto, riportando il rispettivo risultato e in base allo stato mettere result ready a 01 o 10. In entrambi i casi poi ritorna nello stato iniziale. Nel momento in cui invece i risultati dei due moltiplicatori floating-point dovessero arrivare nello stesso istante allora il top level dovrà serializzarli. In uno stato mette il risultato del primo moltiplicatore in output e a 01 il result ready, e nello stato successivo, dunque al ciclo di clock successivo, riporta in output il risultato del secondo moltiplicatore e mette a 10 il result ready. Fatto questo torna sempre allo stato iniziale.

I collegamenti dei segnali interni tra FSM e DATAPATH del moltiplicatore si possono vedere nella figura 6, mentre i collegamenti dei segnali interni del top level sono nella figura 5.

Tutta questa progettazione è stata testata con un testbench (strutturato come in figura 4) che legge da file gli operandi da passare al top level con input ready pari a 11 così da avviare entrambi i moltiplicatori con gli stessi operandi. Successivamente a questi test vengono generati altri operandi random così da completare in totale 100 test. In particolare questa simulazione si è rivelata molto utile in SystemC poichè ha permesso di confrontare il risultato del moltiplicatore floating-point a livello RT, che dopo sarebbe diventato anche il moltiplicatore descritto in VHDL e Verilog, con un moltiplicatore floating-point dato dalla moltiplicazione di due float come in C++ classico. Questo mi ha permesso di verificare in modo puntiglioso se la mia progettazione a livello RT facesse effettivamente le operazioni previste dallo standard IEEE754 e il risultato si è dimostrato ottimo. A meno di piccole imperfezioni sull'arrotondamento il risultato è identico.

Il progetto in SystemC potrà essere testato andando nella directory del progetto, creando una cartella build e compilando dentro build con CMake. Eseguendo poi da terminale `make run` il programma verrà eseguito.

IV. Risultati

I seguenti sono i risultati in termini di tempo della sintesi ed implementazione RT del modulo *tl*, ovvero del modulo top level formato da due moduli moltiplicatori floating-point.

Clock period minimo	4ns
WNS	0,597ns
WHS	0,285ns
TNS	0,0ns
THS	0,0ns
LUT	0,98% (520)
FF	0,01% (455)
DSP	1,82%
IO	80,8%
BUFG	21,88%

I seguenti sono i risultati in termini di tempo ed allocazione delle risorse della sintesi ed implementazione RT del modulo *multiplierIEEE754_verilog*, ovvero del modulo che esegue la moltiplicazione floating-point.

Clock period minimo	5ns
WNS	0,889ns
WHS	0,350ns
TNS	0,0ns
THS	0,0ns
LUT	0,44% (233)
FF	0,01% (5)
DSP	0,91%
IO	80%
BUFG	9,38%

I seguenti sono i risultati in termini di tempo ed allocazione delle risorse della sintesi ed implementazione RT del modulo *multiplierIEEE754_vhdl*, ovvero del modulo che esegue la moltiplicazione floating-point.

Clock period minimo	4ns
WNS	0,732ns
WHS	0,312ns
TNS	0,0ns
THS	0,0ns
LUT	0,44% (234)
FF	0,01% (5)
DSP	0,91%
IO	80%
BUFG	9,38%

I seguenti sono i risultati in termini di tempo ed allocazione delle risorse della high level synthesis del modulo RT scritto in SystemC *multiplierIEEE754_fsmd*, ovvero del modulo che esegue la moltiplicazione floating-point in termini di EFSM, usando la stessa progettazione fatta con Verilog e VHDL.

Clock period minimo	6,540ns
Incertezza	1,25ns
LUT	947
FF	0

I seguenti sono i risultati in termini di tempo ed allocazione delle risorse della high level synthesis del modulo *multiplierIEEE754_float* di livello algoritmico scritto in SystemC, ovvero del modulo che esegue la moltiplicazione floating-point in termini algoritmici, ovvero eseguendo banalmente una moltiplicazione tra float ma mantenendo i poi i segnali e i tipi di SystemC.

Clock period minimo	5,702ns
Incertezza	1,25ns
LUT	399
FF	177

I seguenti sono i risultati in termini di tempo ed allocazione delle risorse della high level synthesis della funzione *multiplierIEEE754* C++, che semplicemente fa la moltiplicazione di due float e ne ritorna il risultato usando il tipo float di base di C++.

Clock period minimo	5,702ns
Incertezza	1,25ns
LUT	348
FF	147

I risultati ottenuti di allocazione delle risorse e di latenza prodotti dalla sintesi dei modelli RTL descritti in Verilog e VHDL, e l'high level synthesis dei modelli a livello algoritmico e a livello RT in SystemC e C++, sono coerenti tra loro in termini di tempo, anche se un po' divergenti in termini di allocazione delle risorse.

In VHDL e Verilog ho ottenuto un'implementazione con una frequenza di clock di circa 200/250MHz. Questo risultato è sicuramente consistente poichè anche con l'high level synthesis si ottiene un'implementazione con il clock period vicino ai 5ns e con un'incertezza di 1,25ns, il che giustifica benissimo il risultato di 4/5ns raggiunto in VHDL e Verilog. Tra VHDL e Verilog si è ottenuto un risultato di clock period diverso, questo mi fa capire che anche se due descrizioni sono a un livello di astrazione simile e una progettazione identica, il risultato dell'implementazione potrebbe essere diversa.

Infatti in termini di allocazione delle risorse VHDL e Verilog sono praticamente identici. Mentre rispetto alla high level synthesis l'allocazione delle risorse risultano diverse. Questo è perfettamente giustificabile dal fatto che una descrizione dell'hardware fatta a livello RT, come quella fatta in VHDL e Verilog, è molto più precisa prodotta da un sorgente algoritmico con un processo di high level synthesis.

Questo mi fa capire che l'high level synthesis è molto utile per ricevere un risultato veloce e avere l'idea della fattibilità

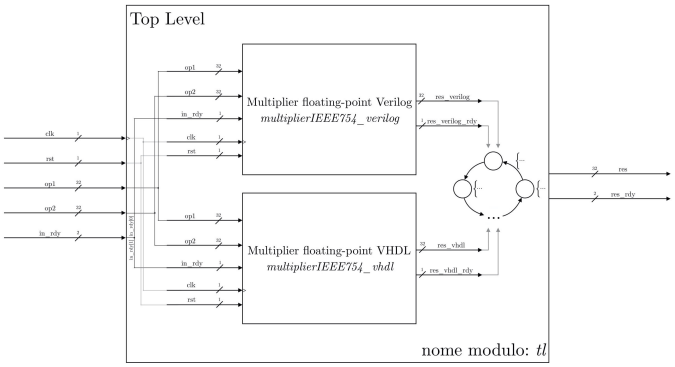

Figura 1. Interfaccia Top Level

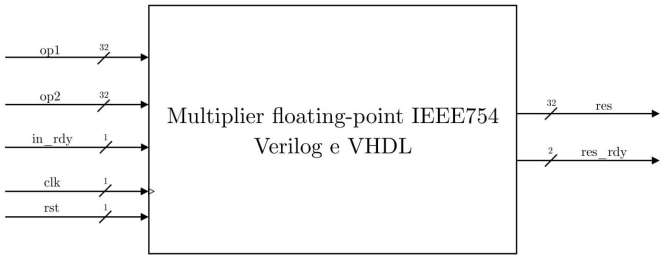

Figura 2. Interfaccia Moltiplicatore Floating-Point

del progetto, ma la vera e propria sintesi ed implementazione dovrebbe sempre essere fatta a livello RT in Verilog o VHDL.

V. Conclusione

Come risultato di questo assignment posso dirmi soddifatto dei risultati ottenuti nella sintesi in termini di tempo e spazio. Le simulazioni, soprattutto in SystemC, si sono rivelate molto efficaci per capire se la mia progettazione a livello RT fosse corretta. Paragonata alla moltiplicazione algoritmica tra due float fatta in C++ il mio modello di moltiplicatore floating-point risulta molto vicino al vero standard IEEE754, a meno solo di alcuni arrotondamenti particolari della mantissa e del esponente.

In fine ho imparato che SystemC è un linguaggio davvero ottimo da ogni punto di vista: offre una simulazione comoda e versatile, e permette velocemente di sintetizzare tramite un tool di high level synthesis, così da avere subito un feedback chiaro sulla fattibilità della realizzazione del modello. Tuttavia quando c'è da scrivere la descrizione del hardware vera e propria, allora risulta sicuramente molto più affidabile uno sviluppo in Verilog o VHDL. In particolare ho avuto modo di rivalutare VHDL rispetto a Verilog, al contrario dell'opinione mondiale.

Appendice

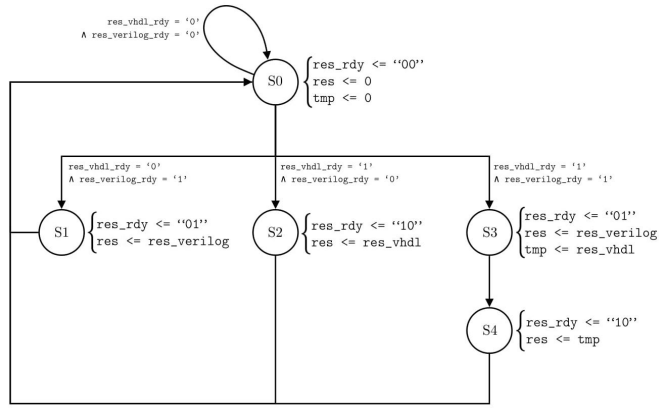

Figura 3. EFSM Top Level

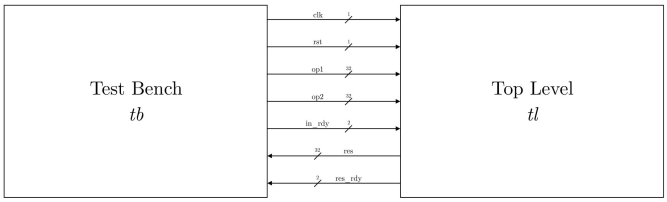

Figura 4. Modello di simulazione

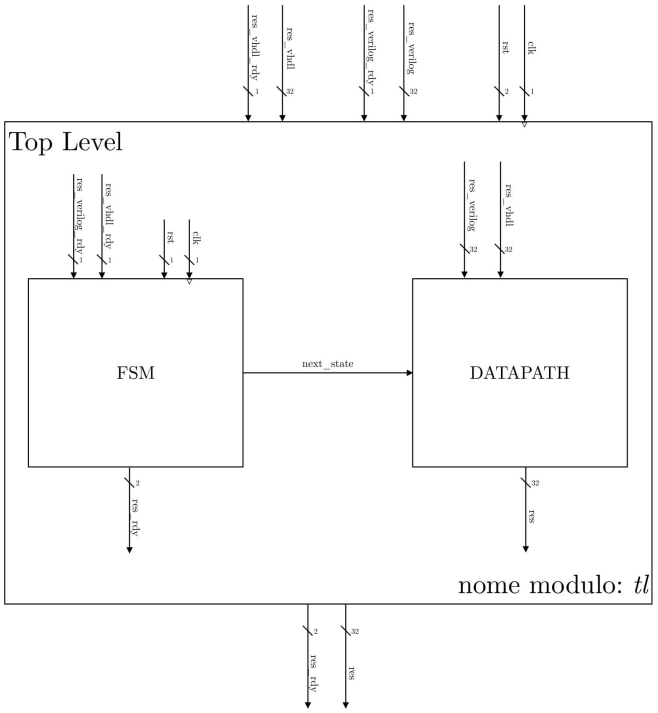

Figura 5. Relazione FSM e DATAPATH Top Level

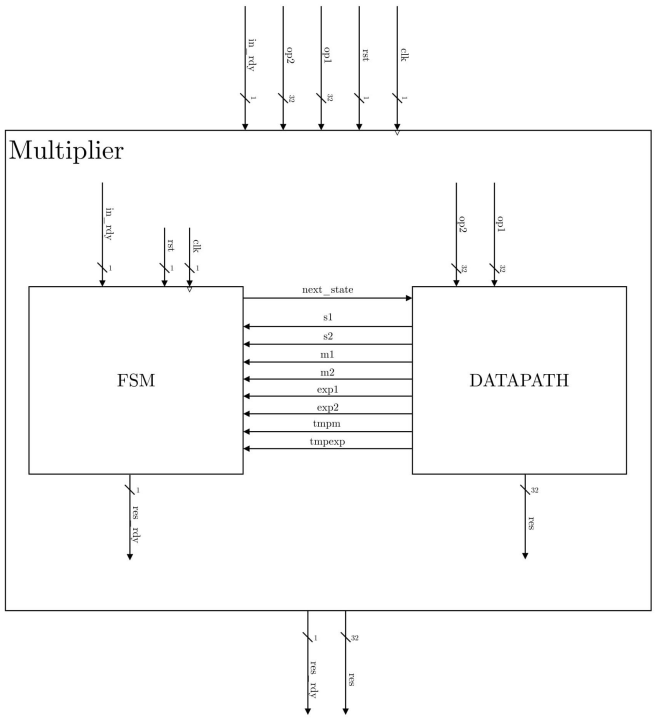

Figura 6. Relazione FSM e DATAPATH Moltiplicatore Floating-point

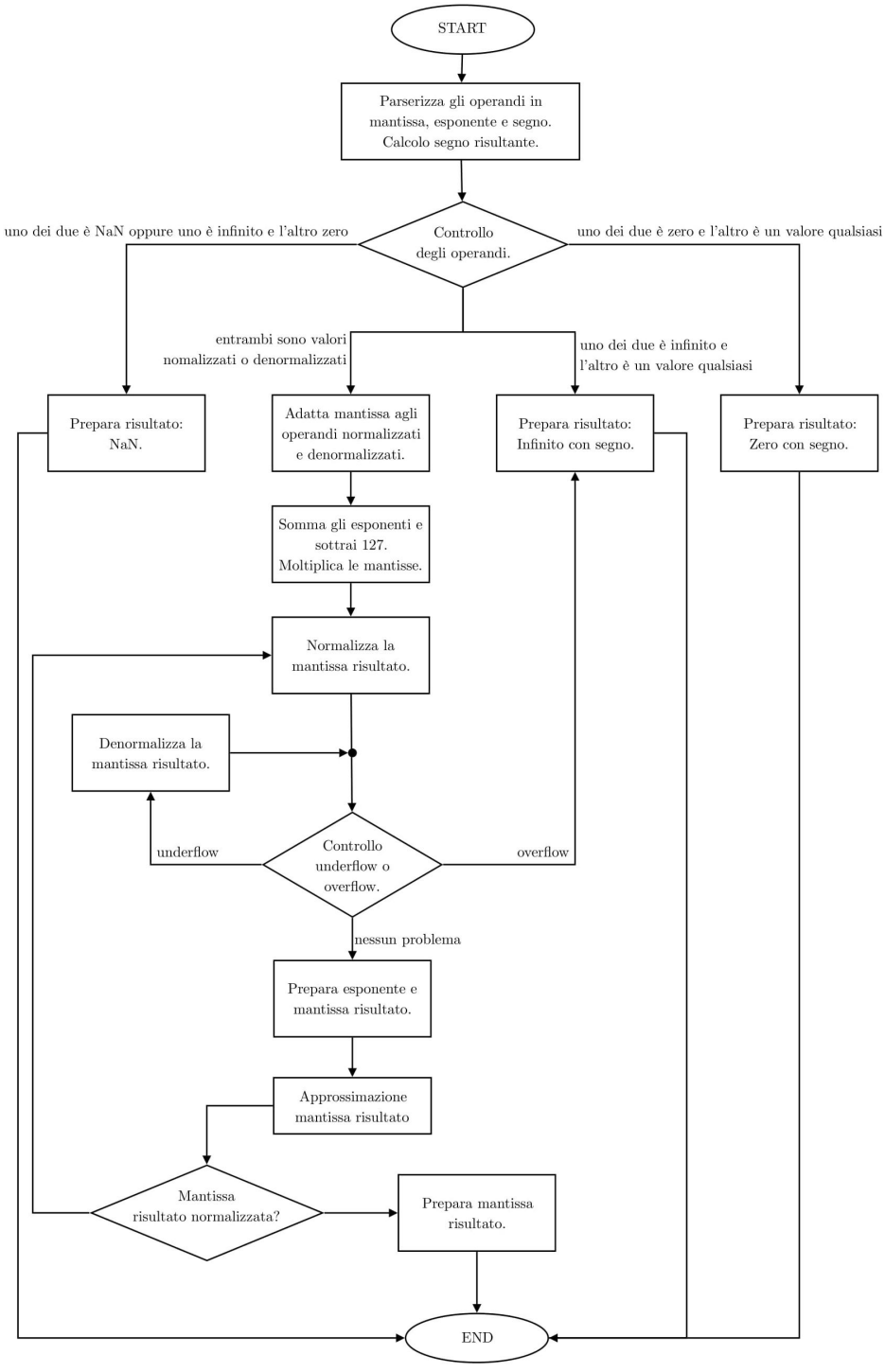

Figura 7. Diagramma Moltiplicatore Floating-Point

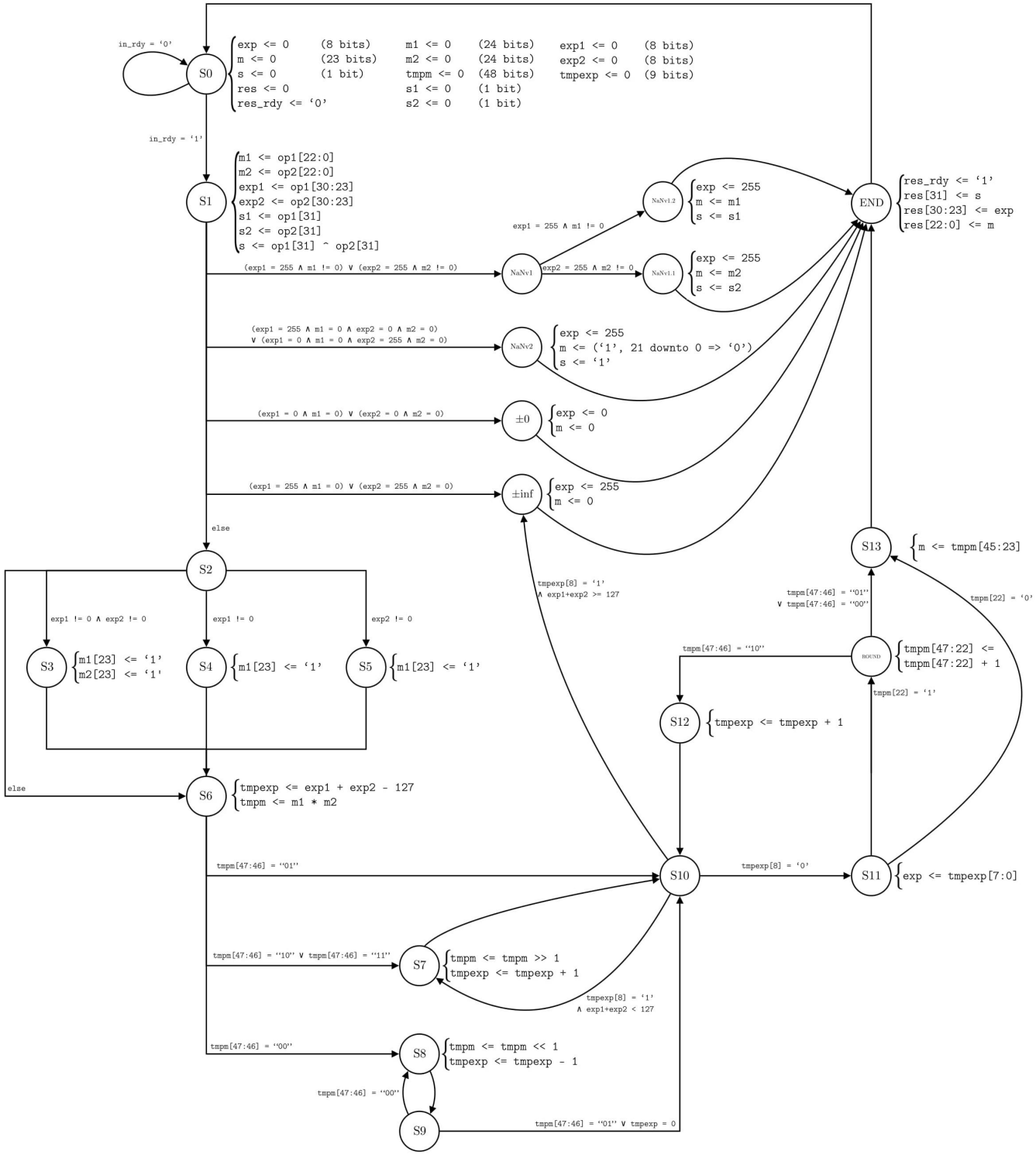

Figura 8. EFSM Moltiplicatore Floating-Point