ELABORATI DI PROGETTAZIONE SISTEMI EMBEDDED

Enrico Giordano VR386687

Progetto 1: implementazione moltiplicatore in SystemC RTL

# Descrizione del progetto

Il progetto consisteva nella realizzazione di un moltiplicatore in virgola mobile, doppia precisione, tra due numeri con virgola.

Il risultato finale è formato da due moduli SystemC: un moltiplicatore implementato come FSMD (controllore + datapath) e un testbench per stimolare e verificare il progetto; tutto questo è stato diviso in più file:

* main.cc : descrive la modalità di collegamento tra il modulo mul\_RTL e il testbench;
* mul\_RTL.cc: descrive il modulo che esegue la moltiplicazione tra due numeri e genera in uscita il risultato;
* mul\_RTL\_testbench.cc : descrive il modulo testbench.

# Interfaccia

L’interfaccia è descritta nel file main.cc, che si occupa di collegare il modulo che esegue la moltiplicazione e il testbench per testarlo.

L’interfaccia si occupa di :

* creare le istanze del modulo del moltiplicatore e del modulo di testing;
* dichiarare le porte di ingresso e uscita del sistema e i segnali che permettono la comunicazione dei moduli precedentemente istanziati;
* eseguire il binding delle porte, ovvero di associare le porte ai rispettivi moduli garantendone la comunicazione.

I segnali istanziati nell’interfaccia sono i seguenti:

* clock: <sc\_logic> rappresenta il segnale sincrono di clock che gestisce lo scorrere degli istanti di tempo del sistema;
* reset\_signal: <bool> viene utilizzato per resettare il sistema.
* p\_In\_data1: <sc\_lv<64>> rappresenta un numero da moltiplicare;
* p\_In\_data2: <sc\_lv<64>> rappresenta un numero da moltiplicare;
* p\_In\_enable: <sc\_uint<1>> indica se può iniziare la computazione;
* p\_Out\_enable: <sc\_uint<1>> indica se è pronto un dato in uscita;
* p\_result: <sc\_lv<64>> rappresenta il risultato della moltiplicazione;

p\_In\_enable e p\_Out\_enable sono segnali che determinano rispettivamente l’inizio e la fine dell’esecuzione dell’algoritmo; di conseguenza vengono utilizzati come flag di collegamento e sincronizzazione tra il modulo di esecuzione e quello di test.

Per effettuare il collegamento, è stato necessario istanziare nel main i due moduli (mul\_RTL\_testbench e mul\_RTL) e collegare ogni loro segnale, input e output con i segnali sopra descritti. Dopo averli collegati, per far partire la simulazione si esegue la funzione sc\_start().

# Descrizione EFSM

L’EFSM, descritta nel file mul\_RTL.cc, è il modulo di elaborazione del sistema e rappresenta l’unione di datapath e macchina a stati finiti dell’algoritmo della moltiplicazione.

L’algoritmo che rappresenta questa macchina a stati è il seguente:

1. SR: reset di tutti i segnali e uscite;
2. S0: preparazione del sistema e inizializzazione delle porte di uscita e di ingresso (si scrive 0 in queste porte in modo da eliminare ogni segnale “sporco”);
3. SoI: lettura dei numeri ed estrapolazione delle singole parti di questi (segno, esponente e mantissa in variabili interne per facilitare i calcoli);
4. S1: somma degli esponenti per ottenere l’esponente del risultato;
5. S2: polarizzazione dell’esponente risultante (togliere 1023 da esso);
6. S3: prodotto tra mantisse in interazione (parte di controllo), ovvero controllo se si devono fare altri passi per la moltiplicazione (in tutto si fanno 52 controlli perché si deve controllare ogni bit della mantissa del secondo numero);
7. S31: prodotto tra mantisse in interazione (parte di esecuzione), ovvero se la mantissa del secondo numero all’indice i-esimo vale ‘1’, si somma alla mantissa risultante la mantissa del primo numero shiftandola di i volte;
8. S4: normalizzazione del risultato in interazione (parte di esecuzione), ovvero si controlla che non ci siano numeri dopo il 104-esimo bit, se ce ne sono si shifta la mantissa risultante a destra di 1 e si incrementa di 1 l’esponente risultante, altrimenti si shifta di 1 a sinistra la mantissa e si decrementa l’esponente;
9. S5: controllo di overflow, ovvero se l’esponente vale 0 o se la mantissa vale 0;
10. S7: normalizzazione del risultato in interazione (parte di controllo), ovvero si controlla se bisogna normalizzare ancora;
11. S8: calcolo del segno;
12. S9: preparazione del risultato da mandare in output.

Il file che contiene l’ESFSM contiene 2 metodi:

* void mul\_RTL :: elaborate\_mul\_FSM(void), che contiene la parte datapath (di calcolo);
* void mul\_RTL :: elaborate\_mul(void), che contiene la parte di fsm (cambio di stato).

# Descrizione dei processi

La procedura di esecuzione del sistema prevede l’istanziazione di due metodi descritti attraverso il comando “SC\_METHOD”, che sono rispettivamente i due processi sopradescritti. Questi processi sono dichiarati di tipo method poiché la loro esecuzione viene eseguita totalmente e può essere richiamata dal sistema più volte, senza mai essere interrotta.

I metodi sono sensibili ai cambiamenti di valore di una serie di segnali che compongono, per ogni metodo, una sensitivity list; al variare di valore di ogni segnale appartenente alla sensitivity list di un metodo, il metodo viene eseguito nuovamente. Un ulteriore procedura di istanziazione dei processi è quella che fa uso delle thread attraverso il comando “SC\_THREAD”. Il processo di testing è istanziato come thread, questo vuol dire che può essere lanciato una volta sola e interrotto attraverso il comando wait(), questo perché il testbench deve chiamare ed attendere il termine dell’esecuzione del modulo mul\_RTL. Tutti i processi sono sensibili al variare del clock.

mul\_RTL ha la sensitivity list descritta in questo modo:

sensitive << STATUS << isready << number\_a << number\_b;

mentre elaborate\_mul\_FSM ha questa sensitivity list:

sensitive << reset.neg();

sensitive << clk.pos();

Progetto 2: implementazione moltiplicatore in SystemC TLM

Descrizione del progetto

Si vuole implementare l’algoritmo della moltiplicazione tra due numeri in virgola mobile doppia precisione a livello TLM, utilizzando le tre tipologie di implementazione del livello TLM: UT, LT e AT4.

In realtà, poichè a livello TLM è più importante osservare come comunicano i differenti moduli tra di loro, senza dare enfasi all’effettivo funzionamento algoritmico già descritto a livello RTL, è stata implementata la moltiplicazione come una normale moltiplicazione tra due numeri in C++, mentre si è implementato il protocollo di comunicazione tra moduli secondo lo standard TLM.

# TLM

Il livello TLM prevede l’utilizzo di due tipologie di modulo per il funzionamento di un sistema.

Queste due tipologie sono rispettivamente: **Initiator** e **Target**. Initiator e Target comunicano utilizzando specifiche primitive e sincronizzandosi tra di loro attraverso socket di comunicazione. Di seguito si vedranno le differenti modalità di comunicazione, ovvero Untimed (senza la nozione di tempo), Loosely Time e Aproximately Time.

# Untimed (UT)

Nella tipologia di TLM il modulo Initiator è implementato nel file “mul\_UT\_testbench.cc”, mentre il modulo Target nel file “mul\_UT.cc”. Il modulo Target rappresenta nel nostro sistema, l’elaborazione vera e propria della moltiplicazione. Il modulo Initiator richiama l’elaborazione per eseguire l’algoritmo. L’Untimed utilizza la primitiva “b\_transport()”, differenziata in modalità Read e Write per la comunicazione tra i due moduli.

La b\_transport è implementata nel modulo Target e viene richiamata dal modulo Initiator per effettuare una simulazione (dopo aver preparato adeguatamente i segnali di ingresso al modulo Target) in modalità Write, successivamente il Target comincerà ad elaborare i segnali in ingresso per calcolarne il risultato della moltiplicazione. Siccome Target e Initiator sono concorrenti, l’Initiator richiama una seconda volta la primitiva b\_transport in modalità Read per poter leggere il risultato del modulo Target. Questo darà la disponibilità alla lettura del payload, (pacchetto di dati che trasferisce la b\_transport), solo successivamente al “TLM\_OK\_RESPONSE” che rappresenta la fine dell’elaborazione del Target.

# Loosely Time (LT)

Anche in questo caso l’elaborazione si divide in modulo Target e Initiator, i quali comunicano tra loro attraverso i rispettivi socket. Come per l’UT l’Initiator richiama la primitiva b\_transport, implementata nel modulo Target, per richiedere al Target l’esecuzione dell’elaborazione. Questo modello si differenzia dall’Untimed in quanto è necessario considerare una sincronizzazione a livello temporale tra i due moduli Target e Initiator. Anche qui l’Initiator richiama la b\_transport in modalità Write per inviare il payload contenente gli input per l’elaborazione del MUL al Target. Il Target, ricevuto il payload, comincia la sua elaborazione e una volta terminata risponde all’Initiator con il messaggio “TLM\_OK\_RESPONSE”. Siccome i moduli sono concorrenti l’Initiator richiama una seconda volta la b\_transport in modalità Read per leggere i risultati dell’esecuzione che saranno disponibili solo dopo il “TLM\_OK\_RESPONSE”. Il tempo viene gestito in modalità di sincronizzazione dei due moduli e visualizzato in fase di esecuzione.

# Comparazione di tempi tra TLM e RTL

Le tempistiche di esecuzione del sistema a livello RTL e TLM consentono di vedere come il sistema sia ottimizzato in termini di ritardo maggiormente a livello RTL rispetto TLM-UT. Si nota infatti, che l’esecuzione del sistema a livello RTL impiega circa XXXXXX, mentre a livello TLM-UT XXXXX. Questo perché, il livello UT non è ottimizzato in termini di ritardo contenendo una wait che blocca l’Initiator durante tutta l’esecuzione del Target. Analogamente, lo stesso tempo lo riporta il TLM-LT. Diversamente il più vicino alle prestazioni di esecuzione dell’ RTL è il TLM-AT4che sfruttando le sue quattro fasi permette un’esecuzione concorrenziale “non completamente bloccante” dei due moduli Target e Initiator, portando il ritardo di esecuzione a XXXXXXX.