Integrazione su Virtual Platform e modellazione con SystemC-TLM di un Moltiplicatore Floating-point Single Precision

Enrico Sgarbanti - VR446095

***Sommario*—Questo documento mostra l'integrazione di un modulo, che realizza due moltiplicazioni a virgola mobile singola precisione secondo lo standard IEEE754[1], nella virtual platform COM6502-Splatters e sua modellazione in SystemC[2] TLM nei diversi stili.**

I. Introduzione

Il primo obiettivo consiste nell'integrare nella virtual platform COM6502-Splatters il modulo double_multiplier creato nel primo progetto. L'idea è fare un piccolo software da caricare nella ROM che richiede l'esecuzione di due moltiplicazioni con quattro operandi, scelti arbitrariamente, di cui uno letto da input. Per fare ciò, utilizzando il protocllo AMBA APB, è necessario realizzare un wrapper del double_multiplier per collegare il componente al bus e un driver per farsì che esso possa essere invocato dal software.
Il secondo obiettivo è implementare il componente double_multiplier nei vari stili di SystemC-TLM e fare con confronto con l'implementazione in SystemC-RTL. Ci si aspetta che con l'aumentare dell'accuratezza della descrizione temporale, la simulazione richiederà più tempo.

II. Background

Nel classico flusso di progettazione di un sistema embedded si parte a sviluppare software solo dopo aver finito la progettazione hardware. Manca però una visione concretamente utilizzabile all'interno del sistema, prima della fase di tape-out, e ciò porta spesso a dover modificare il codice, aumentando così il time-to-market. La **progettazione basata su piattaforma** è la creazione di un'architettura stabile basata su microprocessore che può essere rapidamente estesa, personalizzata per diverse applicazioni e consegnata ai clienti per una rapida implementazione. (J.M. Chateau-STMicroelectornics). Essa permette di fare verifica funzionale, stime di tempo per analisi di perfomance, partizionare hardware e software, porta ad un incremento della velocità e permette modularità e riuso. La modellazione a livello di transazione (**TLM**) è un tipo di progettazione che sta tra il livello algoritmico e quello RT. I dettagli di implementazione vengono astratti preservando però gli aspetti comportamentali del sistema, permettendo quindi una simulazione più veloce, ma meno accurata di quella RTL. Fornisce quindi una piattaforma dove si può iniziare rapidamente a sviluppare software, molto prima rispetto al classico flusso di sviluppo.
In SystemC-TLM la comunicazione tra componenti si ottiene dallo scambio di pacchetti tra un modulo **initiator** e un modulo **target** attraverso 0 o più componenti intermedi. Il trasferimento di dati da un modulo ad un altro è detto **transazione** e avviene attraverso una **socket**. Il percorso che compiono i dati dal initiator al target è detto **forward-path**, invece quello dal target all'initiator è detto **backward-path** e lo si utilizza solo se l'interfaccia è non bloccante. Ci sono tre principali stili che definiscono la relazione tra tempo e dati e permettono al progettista di descrivere il sistema con livello più o meno astratto:

- **Approximately timed:** le transazioni sono divise in quattro fasi: inizio richiesta, fine richiesta, inizio risposta, fine risposta. L'interfaccia è non bloccante quindi viene usato sia il forward-path che il backward-path. Esso è indicato per l'esplorazione architetturale e l'analisi delle performance.
- **Loosely timed:** le transazioni sono divise in due fasi: inizio transizione, fine transizione. L'interfaccia è bloccante quindi viene usato solo il forward-path poichè l'initiator aspetta la risposta del target. Esso rappresenta i dettagli di temporizzazione sufficienti per avviare un sistema operativo ed eseguire sistemi multi-core.
- **Untimed:** la nozione di tempo non è necessaria e quindi non viene presa in considerazione.

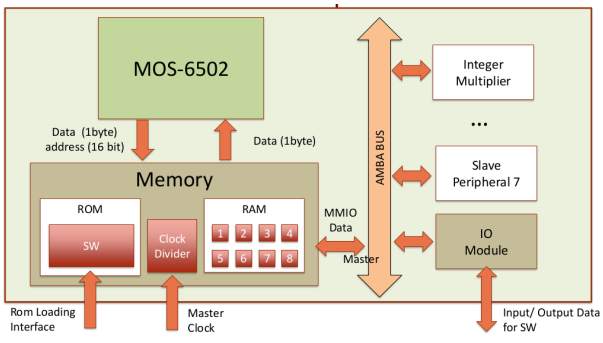

Figura 1: COM6502-Splatters

La virtual platform utilizzata è COM6502-Splatters [Figura 1] che include:

- **CPU** MOS 6502 (1975) con indirizzamento a 16 bit e gestione di dati in 8 bit.
- **ROM** da 16KB in un singolo blocco.
- **RAM** da 16KB divisa in 8 blocchi per permettere operazioni multiple di lettura/scrittura.

- **BUS** ARM APB (advanced peripherical Bus) che supporta fino a 8 periferiche.
- **IO Module** usato per richiedere e inviare dati alla piattaforma.
- **Multiplier** usato per eseguire moltiplicazioni fra interi.

Essa deve essere compilata col cross-compilatore cc65[3].
MBA (Advanced Microcontroller Bus Architecture) è uno standard open-source di ARM per la connessione e la gestione di blocchi funzionali nei progetti di system-on-a-chip. Nella versione APB (Advanced peripherical bus) ci sono due attori: **Master** che controlla le perifiriche; **Slave** periferica controllata dal master. I segnali utilizzati in questo protocollo sono:

- **pclk:** segnale di clock della periferica.
- **preset:** segnale di reset della periferica.
- **paddr:** indirizzo.
- **psel:** segnale che indica se la periferica è stata selezionata.
- **penable:** segnale che indica se la periferica è stata abilitata.
- **pwrite:** segnale che indica operazioni di scrittura (1) o lettura (0) sulla periferica.
- **pwdata:** dati sulla periferica da parte del Master.
- **pready:** segnale che indica che i dati per il Master sono pronti.
- **prdata:** dati sulla periferica per il Master.

III. Metodologia applicata

A. Struttura progetto

- **Virtual_Platform/**
  - **application/** cartella contenente il codice sorgente dell'software e dei driver.
  - **platform/** cartella contenente il codice sorgente dell'hardware e dei vari wrapper.
  - **cc65/** cross-compilatore scaricabile dalla pagina github[3] con commit di riferimento: 582aa41f2a702ff477a00a5d69a794390a13b544).
- **TLM/**
  - **UT/** progetto con modellazione TLM Untimed.
  - **LT/** progetto con modellazione TLM Loosely Timed.
  - **AT4/** progetto con modellazione Approximately Timed.
  - **RTL/** progetto con modellazione a livello RT. Questa versione è funzionalmente equivalente a quella dell'altro progetto, ma col testbench adattato per essere coerente con quello usato per le modellazioni TLM.
  - **script.sh** piccolo script per eseguire in automatico in tutte le cartelle i comandi make, make clean e l'esecuzione con time.
  - Ognuna di queste cartella presenta la seguente struttura:
    - **Makefile:** tool per la compilazione automatica del progetto. Richiede che la variabile d'ambiente SYSTEMC_HOME contenga il path alla libreria di SystemC.
    - **include:** contiene gli headers del progetto.
    - **src:** contiene i file sorgenti del progetto.
    - **bin:** contiene l'eseguibile generato dopo la compilazione.
    - **obj:** contiene i file oggetto generati dopo la compilazione.

B. Virtual Platform

1) Procedimento: Per prendere dimestichezza con la piattaforma è stato prima integrato il modulo di moltiplicazione IEEE754 scritto in verilog sulla periferica 3. Per fare ciò è stato creato un wrapper in hardware con l'interfaccia APB slave per poterlo fare comunicare con il resto della piattaforma e un driver per poterlo utilizzare a livello software. In seguito è stato integrato il modulo d'interesse cioè double_multiplier sulla periferica 4. Entrambi i codici sono stati testati eseguendo due semplici moltiplicazioni dove un operando è stato viene da input.

2) Wrapper double_multiplier: I segnali del bus APB sono stati collegati nel seguente modo al double_multiplier:

- **pclk:** collegato a **clk**.
- **preset:** collegato a **reset**.
- **paddr:** non utilizzato.
- **psel:** non utilizzato.
- **penable:** utilizzato nella EFSM.
- **pwrite:** non utilizzato.
- **pwdata:** utilizzato nella EFSM per prelevare gli operandi.
- **pready:** utilizzato nella EFSM per indicare che su **prdata** è presente un risultato.
- **prdata:** utilizzato nella EFSM per inviare il risultato al master.

Sono stati inoltre usati i seguenti segnali intermedi:

- **op1, op2:** collegati alle porte **op1** e **op2** del double_multiplier e utilizzati per inviare gli operandi.
- **res:** collegato alla porta **res** del double_multiplier e utilizzato per ricevere il risultato delle moltiplicazioni.
- **op1_tmp, op2_tmp, op3_tmp, op4_tmp:** utilizzati per memorizzare i valori degli operandi letti dal bus e poi inviarli a **op1** e **op2**.
- **res_tmp:** utilizzato per memorizzare il valore del secondo risultato da **res** e inviarlo al momento giusto sul bus.
- **ready, done:** utilizzati per il protocollo di handshake col double_multiplier
- **STATE, NEXT_STATE:** utilizzati per rappresentare lo stato presente e lo stato prossimo della FSMD.

Avendo scelto di leggere gli operandi (e scrivere i risultati) su cicli di clock consecutivi si è stati costretti ad utilizzare molti registri per memorizzare i valori temporanei. Si può migliorare questo aspetto utilizzando *ready* e *done* diversi per le due moltiplicazioni all'interno di double_multiplier.
Il wrapper è descritto grazie alla EFSM [Figura 7] la quale è formata da 14 stati:

- **ST_WAIT1:** stato di partenza. Qui vengono resettati i segnali interni e gli output a zero. In caso di segnale *preset* a 1 si torna in questo stato. In caso di segnale *penable* a 1, il master avrà pubblicato il valore del primo input in *pwdata* e quindi si passa a *ST_READ1*.

- **ST_READ1:** qui si salva il valore di *pwdata* in *op1_tmp*. In caso di segnale *penable* a 0 si passa a *ST_WAIT2*.
- **ST_WAIT2:** qui si attende che venga inviato l'operando successivo. In caso di segnale *penable* a 1, il master avrà pubblicato il valore del secondo input in *pwdata* e quindi si passa a *ST_READ2*.
- **ST_READ2:** qui si salva il valore di *pwdata* in *op2_tmp*. In caso di segnale *penable* a 0 si passa a *ST_WAIT3*.
- **ST_WAIT3:** qui si attende che venga inviato l'operando successivo. In caso di segnale *penable* a 1, il master avrà pubblicato il valore del terzo input in *pwdata* e quindi si passa a *ST_READ3*.
- **ST_READ3:** qui si salva il valore di *pwdata* in *op3_tmp*. In caso di segnale *penable* a 0 si passa a *ST_WAIT4*.
- **ST_WAIT4** qui si attende che venga inviato l'operando successivo. In caso di segnale *penable* a 1, il master avrà pubblicato il valore del quarto input in *pwdata* e quindi si passa a *ST_READ4*.
- **ST_READ4:** qui si salva il valore di *pwdata* in *op4_tmp*. Ora sono stati raccolti tutti gli operandi per *double_multiplier* quindi si passa direttamente a *ST_ELAB1*.
- **ST_ELAB1:** qui si passano i primi due operandi a *double_multiplier* e poi si passa a *ST_ELAB2*.
- **ST_ELAB2:** qui si passano gli altri due operandi a *double_multiplier* e si rimane in attena che *done* diventi 1 per poi passare a *ST_RET0*.
- **ST_RET0:** qui si inserisce su *prdata* il valore di *res* e si pone *pready* a 1, per indicare al Master che è pronto il primo risultato. Poi si passa direttamente a *ST_RET1*.
- **ST_RET1:** qui si salva in *res_tmp* il risultato della seconda moltiplicazione ottenuto da *double_multiplier* e si resta in attesa che il master abbia letto il valore del primo risultato. Quando *penable* diventa 0 allora il Master avrà letto il valore e si passa in *ST_WAIT5*.
- **ST_WAIT5:** qui si pone *pready* a 0 per indicare al Master che si è pronti a trasmettere il secondo risultato e si rimane in attesa che il Master richieda quel valore. Quando *penable* diventa 1 allora si passa in *ST_RET2*.
- **ST_RET2:** qui si inserisce su *prdata* il secondo risultato cioè *res_tmp* e si resta in attesa che il master abbia letto il valore. Quando *penable* diventa 0 allora il Master avrà letto il valore e si ritorna in *ST_WAIT1*.

3) Driver double_multiplier: Per utilizzare il double_multiplier è stata aggiunta una routine all'interno del file /application/src/routines.c chiamata *double_multiplier*.
La comunicazione tra master e slave è descritta dal sequence diagram in figura 8. Sostanzialmente il master invia uno alla volta gli operandi di 32 bit e poi resta in attesa che *pready* diventi 1. Lo slave nel frattempo salva gli operandi in registri, dopodichè li invia nel giusto ordine a *double_multiplier* e attende che *done* diventi 1. A questo punto invia al master il primo risultato, imposta *pready* a 1 e poi si salva il secondo risultato in un registro. Il master si salva il valore del primo risultato e poi pone *penable* a 0 per dire allo slave che ha ricevuto il dato, il quale di conseguenza imposta *pready* a 0. Dopodichè il master imposta *penable* a 1 per dire allo slave che è pronto a ricevere il secondo risultato e si mette in attesa che *pready* diventi 1. Lo slave analogamente a prima inviarà il risultato e porrà *pready* a 1 sbloccando il master che si salverà il risultato e metterà *pready* a 0 permettendo così allo slave di ritornare allo stato iniziale.

C. SystemC TLM

L'obiettivo è realizzare funzionalmente il double_multiplier in modo da avere una piattaforma su cui poter sviluppare software parallelamente alla realizzazione dell'hardware.

```
└─> $ ./UT/bin/double_multiplier_UT.x
[TB:] Want to performe double_multiplication
[TB:] Results are:
        1): 01000000110000000000000000000000 => 6
        2): 01000000001000000000000000000000 => 2.5


Info: /OSCI/SystemC: Simulation stopped by user.
```

Figura 2: Simulazione con SystemC TLM untimed

Nello stile **untimed** il testbench è l'initiator che chiama il target cioè il double_multiplier, il quale elabora le moltiplicazioni e restituisce i risultati, sbloccando l'initiator. Nella figura 2 si vedono i risultati ottenuti.

```
└─> $ ./LT/bin/double_multiplier_LT.x
[TB:] Want to performe double_multiplication
[TB:] Invoking the b_transport primitive
                [DM:] Received invocation of the b_transport primitive [WRITE]
                [DM:] Invoking the dm_function to calculate the mults
                [DM:] Calculating dm_function ...
[TB:] Results are:
        1): 01000000110000000000000000000000 => 6
        2): 01000000001000000000000000000000 => 2.5

Time: 0 s + 100 ns
--------------------------------------------------

Info: /OSCI/SystemC: Simulation stopped by user.
```

Figura 3: Simulazione con SystemC TLM loosely-timed

Nello stile **loosely-timed**, analogamente all'untimed, il testbench chiama il double_multiplier, il quale elabora le moltiplicazioni e restituisce i risultati con l'informazione di tempo trascorso. Come valore di **timing_annotation** è stato utilizzato 100ns, valore ricavato dalla moltiplicazione di 10ns (cioè il periodo minimo a cui il componente sintezzato può funzionare) per 10 (cioè la lunghezza di cicli di clock media che sono necessari al componente per eseguire le due moltiplicazioni). Il numero di cicli di clock necessari all'esecuzione di una moltiplicazione può variare molto, come si può osservare lanciando la simulazione col **full_target_test** della descrizione in SystemC RTL. Come lunghezza del quanto per il loosely-timed è stato utilizzato il periodo di clock minimo cioè 10ns. Nella figura 2 si vede bene che l'initiator invoca il target, resta in attesa della risposta e poi stampa i risultati. Si nota anche che dal tempo 0 sono passati 100ns.

Nello stile **approximately-timed** ci sono quattro fasi per la comunicazione osservabili in figura 4:

- Fase **BEGIN REQUEST**: l'initiator invoca il target, mandandogli gli operandi. Inizio forward-path
- Fase **END REQUEST**: il target riceve gli operandi e attiva IOPROCESS. Fine forward-path

```
└─> $ ./AT4/bin/double_multiplier_AT4.x
[TB:] Want to performe double_multiplication
[TB:] <<<BEGIN REQ>>>
[TB:] Invoking the nb_transport_fw primitive [WRITE]
                [DM:] Received invocation of the nb_transport_fw primitive
                [DM:] <<<END REQ>>>
                [DM:] Activating the IOPROCESS
                [DM:] End of the nb_transport_fw primitive
[TB:] Waiting for nb_transport_bw to be invoked
                [DM:] IOPROCESS has been activated
                [DM:] Invoking the dm_function to calculate the mults
                [DM:] Calculating dm_function ...
                [DM:] <<<BEGIN RESP>>>
                [DM:] Invoking the nb_transport_bw primitive [WRITE]
[TB:] Performing nb_transport_bw primitive
[TB:] <<<END RESP>>>
[TB:] Stop to waiting for nb_transport_bw invocation
[TB:] <<<BEGIN REQ>>>
[TB:] Invoking the nb_transport_fw primitive [READ]
                [DM:] Received invocation of the nb_transport_fw primitive
                [DM:] <<<END REQ>>>
                [DM:] Activating the IOPROCESS
                [DM:] End of the nb_transport_fw primitive
[TB:] Waiting for nb_transport_bw to be invoked
                [DM:] IOPROCESS has been activated
                [DM:] Returning results
                [DM:] <<<BEGIN RESP>>>
                [DM:] Invoking the nb_transport_bw primitive [WRITE]
[TB:] Performing nb_transport_bw primitive
[TB:] <<<END RESP>>>
[TB:] Stop to waiting for nb_transport_bw invocation
[TB:] Results are:
        1): 01000000110000000000000000000000 => 6
        2): 01000000001000000000000000000000 => 2.5


Info: /OSCI/SystemC: Simulation stopped by user.
```

Figura 4: Simulazione con SystemC TLM approximately-timed

- Fase **BEGIN RESPONSE**: vengono calcolate le moltiplicazioni e viene notificato l'initiato. Inizio backward-path
- Fase **END RESPONSE**: Viene ricevuta la notifica. Fine backward-paths

Analogamente si eseguendo le quattro fasi per ottenere il risultati.

```
└─> $ ./RTL/bin/double_multiplier_RTL.x

        SystemC 2.3.2-Accellera --- May 21 2020 15:39:11
        Copyright (c) 1996-2017 by all Contributors,
        ALL RIGHTS RESERVED
         1): 01000000110000000000000000000000 => 6
         2): 01000000001000000000000000000000 => 2.5

Info: /OSCI/SystemC: Simulation stopped by user.
```

Figura 5: Simulazione con SystemC RTL

Per rendere più significativo il confronto è stato riportato anche il progetto **RTL**, ma con con testbench analagoto a quello usato per gli stili del TLM. In figura 5 si vedono i risultati del test.
In ogni progetto SystemC, dentro il file "define.hh" si può attivare la modalità debug in cui viene testato il double_multiplier con degli operandi scelti arbitrariamente e stampati dei messaggi per controllare il corretto funzionamento (figure 2 3 4 5). Se la modalità di debug è disattiva tutti i progetti eseguirano TESTNUM volte double_multiplier con operandi generati randomicamente e non stamperanno nulla a video.
È stato poi messo a disposizione uno script dove è possibile eseguire con l'argomento:

- **clean** il comando make clean in ogni directory, per eliminare i file sorgenti ed eseguibili.
- **make** il comando make in ogni directory, per eseguire la compilazione.
- **time** per eseguire sequenzialmente gli eseguibili con il comando *time* per ricavare il tempo di esecuzione delle simulazioni.

IV. RISULTATI

A. Simulazione e testbench sulla VirtualPlatform

Il main del software legge un valore dal modulo I/O e chiama il driver di *double_multiplier* per eseguire la moltiplicazione con l'operando letto e altri 3 scelti arbitrariamente. Una volta ottenuto i risultati vengono poi trasmessi per essere letti in simulazione del testbench. (Nel main è anche presente la possibilità di utilizzare gli stessi operandi per eseguire due moltiplicazioni separate col driver *float_multiplier*).
Nel testbench scritto in verilog viene caricato il codice del software nella ROM, inviato un valore sul bus, che verrà poi utilizzato come operando e infine stampati i due risultati ottenuti.
Nelle figure 9 10 è possibile guardare la simulazione. In particolare si nota il corretto comportamento descritto in precedenza dalla EFSM (figura7) e il sequence diagram (figura8) e l'elaborazione delle moltiplicazioni $1.0 \times 2.0 = 2.0$ e $3.0 \times 2.5 = 7.5$.

B. TLM

```
└─> $ ./script.sh time
COMMAND: time

 >> time of UT

Info: /OSCI/SystemC: Simulation stopped by user.

real    0m0,717s
user    0m0,716s
sys     0m0,000s

 >> time of LT

Info: /OSCI/SystemC: Simulation stopped by user.

real    0m0,805s
user    0m0,805s
sys     0m0,000s

 >> time of AT4

Info: /OSCI/SystemC: Simulation stopped by user.

real    0m1,463s
user    0m1,459s
sys     0m0,004s

 >> time of RTL

        SystemC 2.3.2-Accellera --- May 21 2020 15:39:11
        Copyright (c) 1996-2017 by all Contributors,
        ALL RIGHTS RESERVED

Info: /OSCI/SystemC: Simulation stopped by user.

real    0m30,373s
user    0m30,256s
sys     0m0,072s
```

Figura 6: Confronto timing simulazioni TLM e RTL con 1000000 esecuzioni di double_multiplier

In figura 6 si vede che, come previsto, con l'aumentare dell'accuratezza temporale aumenta anche il tempo necessario per la simulazione. Si nota che fra lo stile untimed e loosely-timed non cambia molto, ma tra tra la versione più accurata

TLM cioè approximately-timed e quella RTL c'è una grossa differenza.

V. Conclusioni

Sono rimasto particolarmente colpito dalla velocità delle simulazioni con SystemC-TLM rispetto a SystemC-RTL, ma soprattutto alla velocità con cui si riesce a scrivere il codice per descrivere il sistema funzionalmente. Purtroppo la versione approximately-timed risulta molto distante dalla versione RTL a livello di accuratezza, infatti viene usato un tempo medio di esecuzione del double_multipler, il quale però varia molto in base agli operandi.
Sono soddisfatto dei risultati ottenuti e sicuramente questo progetto mi ha aiutato a comprendere meglio cos'è la progettazione basata su piattaforma e la sua utilità.

Riferimenti bibliografici

[1] I. C. Society, "Ieee standard 754 for binary floating-point arithmetic," *Online*, 1985.
[2] Accellera Systems Initiative *et al.*, "Systemc," *Online, December*, 2013.
[3] "Cc65," https://github.com/cc65/cc65.
[4] "Vivado," https://www.xilinx.com/products/design-tools/vivado.html.

Appendice

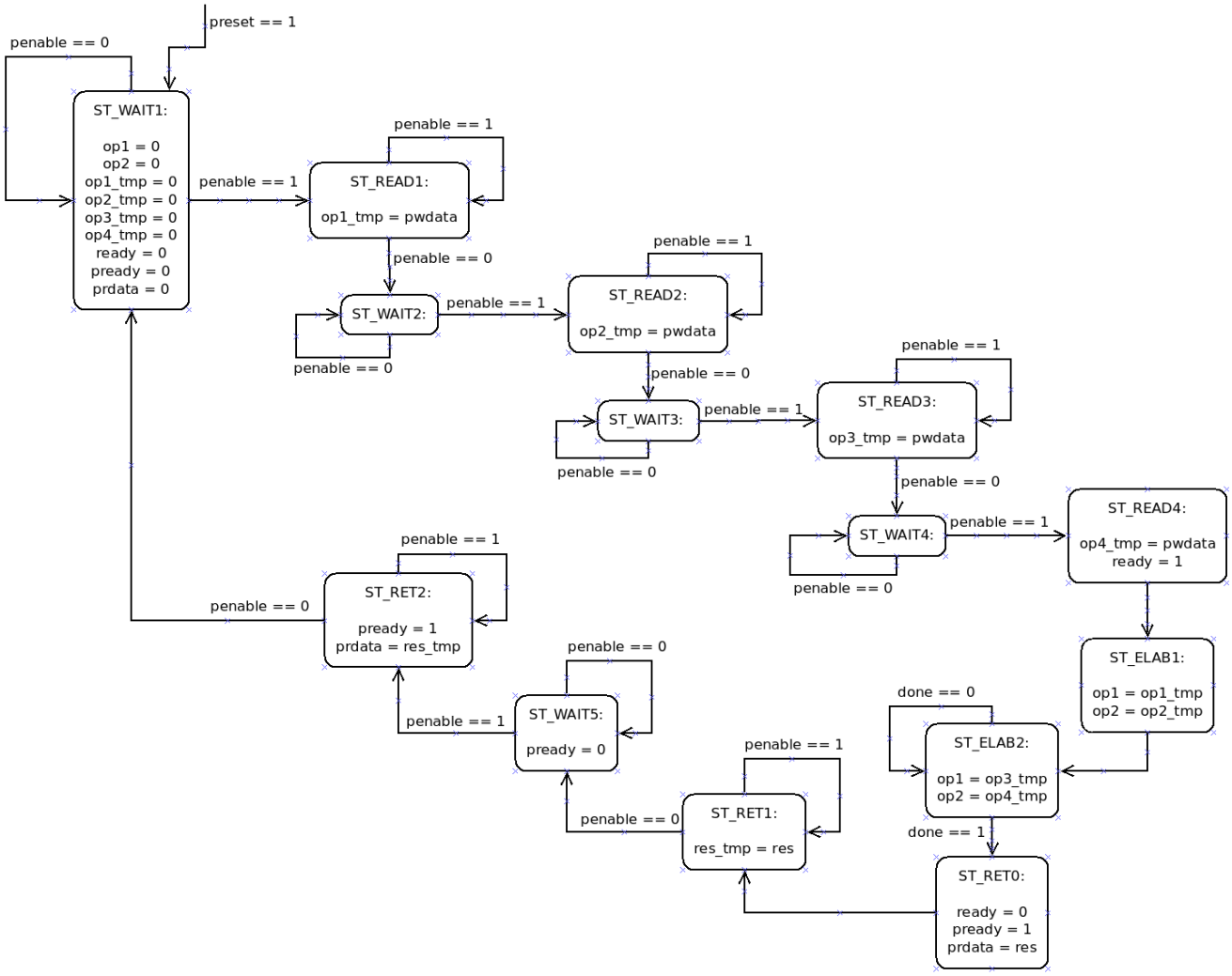

Figura 7: EFSM del wrapper di double_multiplier

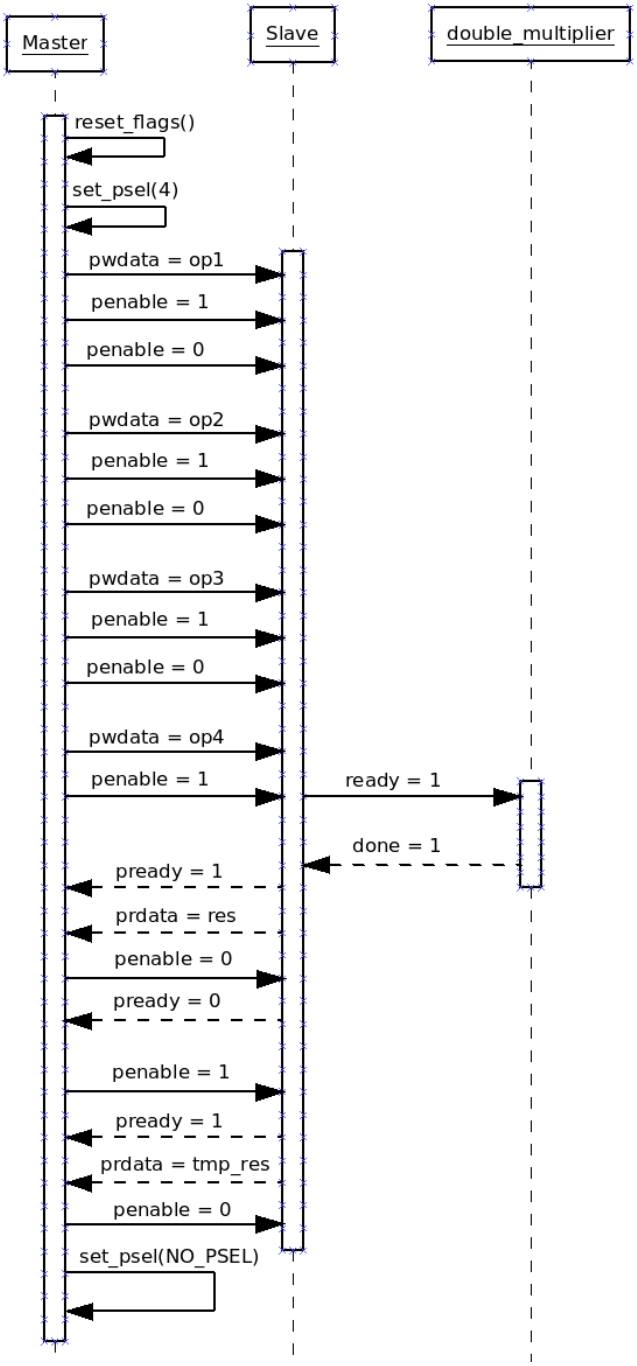

Figura 8: Sequence diagram della comunicazione tra Master Slave e double_multiplier

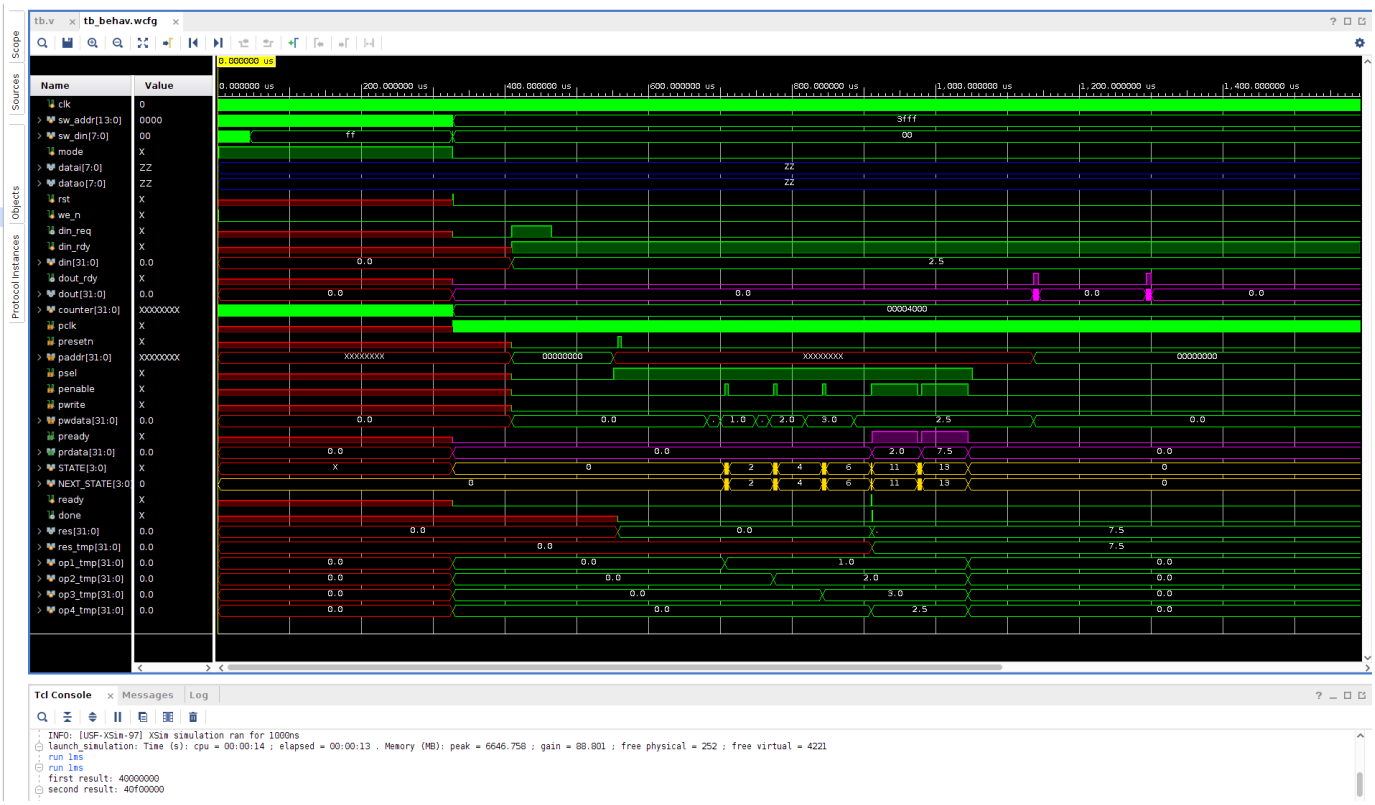

Figura 9: Simulazione virtual platform

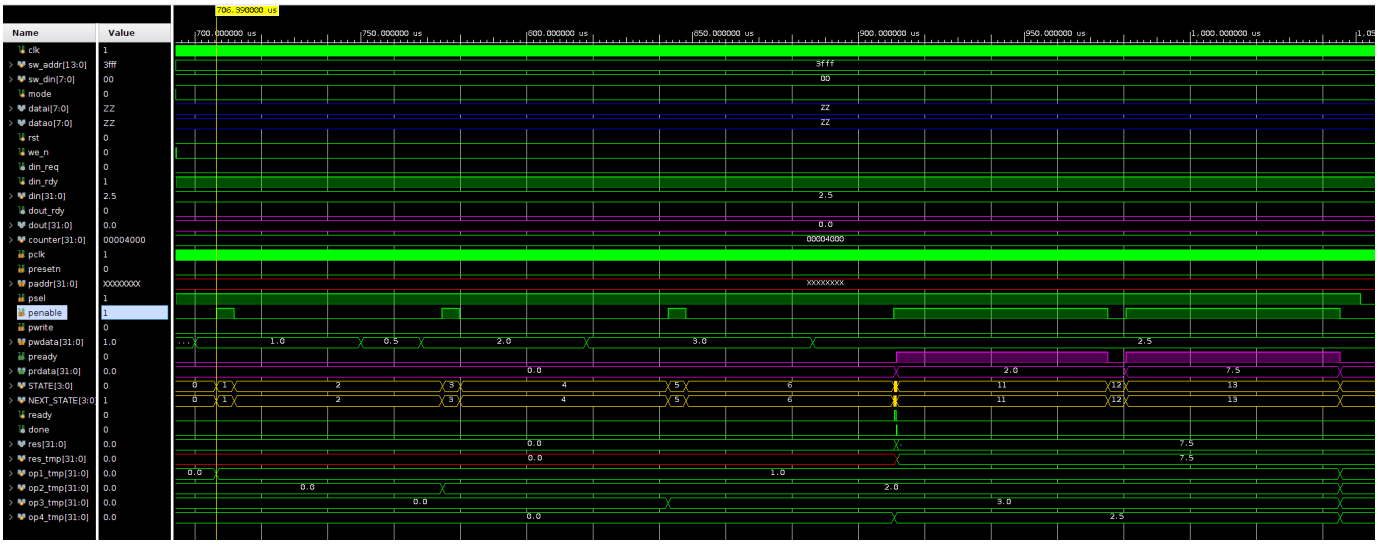

Figura 10: Simulazione con zoom virtual platform