Modellazione in VHDL di un cifrario basato su algoritmo XTEA

Massimiliano Incudini - VR433300

***Sommario*—Il seguente documento documenta il lavoro svolto durante la seconda parte del modulo di laboratorio del corso Progettazione di Sistemi Embedded (aa 2018/2019). Il suo obiettivo è illustrare le scelte progettuali effettuate durante lo sviluppo del modulo XTEA nel linguaggio VHDL.**

I. Introduzione

Il progetto consiste nello sviluppare un cifrario il cui funzionamento è basato sull'algoritmo eXtended TEA fornito insieme alla consegna.

1) Il cifrario dovrà essere implementato come modulo VHDL, poi testato automaticamente tramite uno script `stimuli.do` per il software proprietario Modelsim.
2) Il modulo deve essere importato in Vivado, simulato tramite un testbench e sintetizzato per la piattaforma Xilinx PYNQ.
3) I sorgenti SystemC del cifrario dovranno essere sintetizzati tramite Vivado HLS, ed il risultato della sintesi confrontato con quello del modulo VHDL.

II. Architettura del modulo

Il modulo cifratore presenta gli ingressi e le uscite visibili in Figura 1.

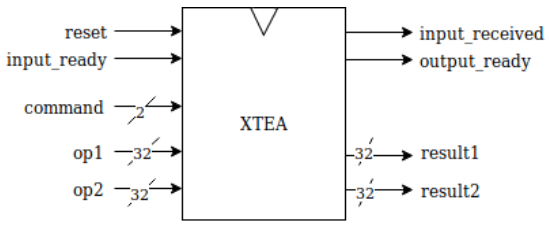

Figura 1. Interfaccia del modulo VHDL

In input abbiamo:

- `clock` (1bit): segnale di sincronia;
- `reset` (1bit): quando alto resetta il modulo alle condizioni iniziali;
- `input_ready` (1bit): quando alto il modulo può leggere i valori sulle porte `command`, `op1` e `op2`;
- `command` (2bit): operazione da eseguire, deve essere uno tra i valori `CONFIGURE_KEYS_0_1`, `CONFIGURE_KEYS_2_3`, `RUN_ENCRYPT`, `RUN_DECRYPT`;
- `op1` (32bit): primo operando;
- `op2` (32bit): secondo operando.

In output abbiamo:

- `input_received` (1bit): quando alto il modulo segnala di aver letto i valori in input;
- `output_ready` (1bit): quando alto il modulo segnala completato l'operazione ed è in grado di ricevere nuovi comandi;
- `result1`, `result2` (32bit ciascuno): prima e seconda parte del risultato.

Il modulo per poter criptare o decriptare ha bisogno in input di 6 parole a 32bit: quattro appartenenti alla chiave di cifratura, e due appartenenti al messaggio. Abbiamo quindi una prima fase di configurazione, nel quale impostiamo la chiave. Non occorre ripetere questa prima fase se non per cambiare la chiave o dopo un reset. Successivamente possiamo passare il messaggio, due word alla volta, al modulo per effettuare l'operazione richiesta. Questo meccanismo è schematizzato in Figura 2.

Figura 2. Operazioni del modulo

Ognuna delle operazioni segue un preciso protocollo illustrato in Figura 3. Per prima cosa l'utilizzatore scrive sulle porte dei dati e alza `input_ready`. Il modulo legge i dati ed abbassa `output_ready`. Il ciclo di clock successivo il modulo alza `input_received`. A questo punto l'utilizzatore abbassa `input_ready`. Una volta che il modulo ha completato l'operazione scrive sulle porte di output i risultati, alza `output_ready` ed abbassa `input_received`.

III. Implementazione

Il modulo è implementato nel file `xtea.vhd`. La sua struttura è quella di una FSMD, e divisa in tre processi.

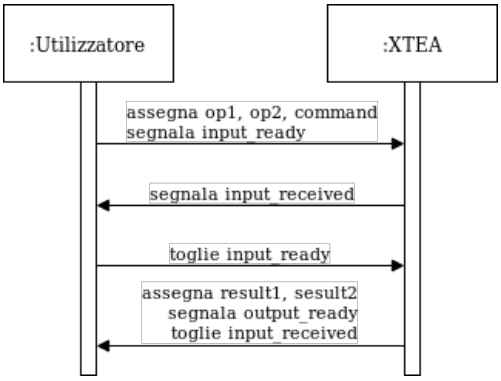

Figura 3. Protocollo di esecuzione di una qualsiasi operazione

- Il primo processo scandisce lo step di computazione, ha nella sua sensitivity list il `clock` e serve per aggiornare il segnale `current_state`.
- Il secondo processo implementa la FSM, aggiorna i segnali di controllo tra cui `next_state`, `input_received` e `output_ready`, nonchè legge e immagazzina i dati in input.
- Il terzo processo implementa il datapath, effettua i calcoli sui dati in input e scrive i dati in output.

Lo schema della E-FSM associata al modulo è visibile in Figura 4. Inoltre, le costanti rappresentanti i comandi del modulo sono presenti nel file `xtea_helper.vhd`.

IV. Simulazione con Modelsim

Per poter simulare il progetto col software Mentor Modelsim creiamo un nuovo progetto all'interno del quale importiamo i file `xtea.vhd` e `xtea_helper.vhd`. Nella console digitiamo i comandi:

```
# Compilazione del file e visualizzazione report
vcom -reportprogress 300 -work work C:/percorso/xtea.vhd
# Apertura software di simulazione
vsim [-wlf output_waveform_file.wlf] work.xtea
do stimuli.do
# Chiusura simulazione
quit -sim
```

Segue una parte del file `stimuli.do` che automatizza in parte il setup della simulazione. Il comando `force <porta> <valore> <tempo di assegnazione> ns` assegna al tempo voluto il valore alla porta. Così facendo possiamo comandare in modo molto immediato il nostro modulo. L'alternativa a questo sarebbe scrivere un file VHDL di testbench.

Il metodo qui utilizzato ha una limitazione. Quando eseguiamo un'operazione, mettiamo `input_ready` alto, e poi lo abbassiamo quando viene segnalato `input_received`. Non avendo un'istruzione del tipo "esegui fino a che `input_received = 1`" decidiamo di aspettare due cicli di clock, dopo i quali sappiamo dall'implementazione che tale segnale viene messo alto. Se però l'implementazione cambiasse allora questo sistema non funzionerebbe più.

Per questo nelle sezioni successive utilizziamo un modulo testbench, che permette una simulazione più "robusta".

```
# fai ripartire la simulazione (se non lo fai,
# continua dal punto nel quale si era fermato)
```

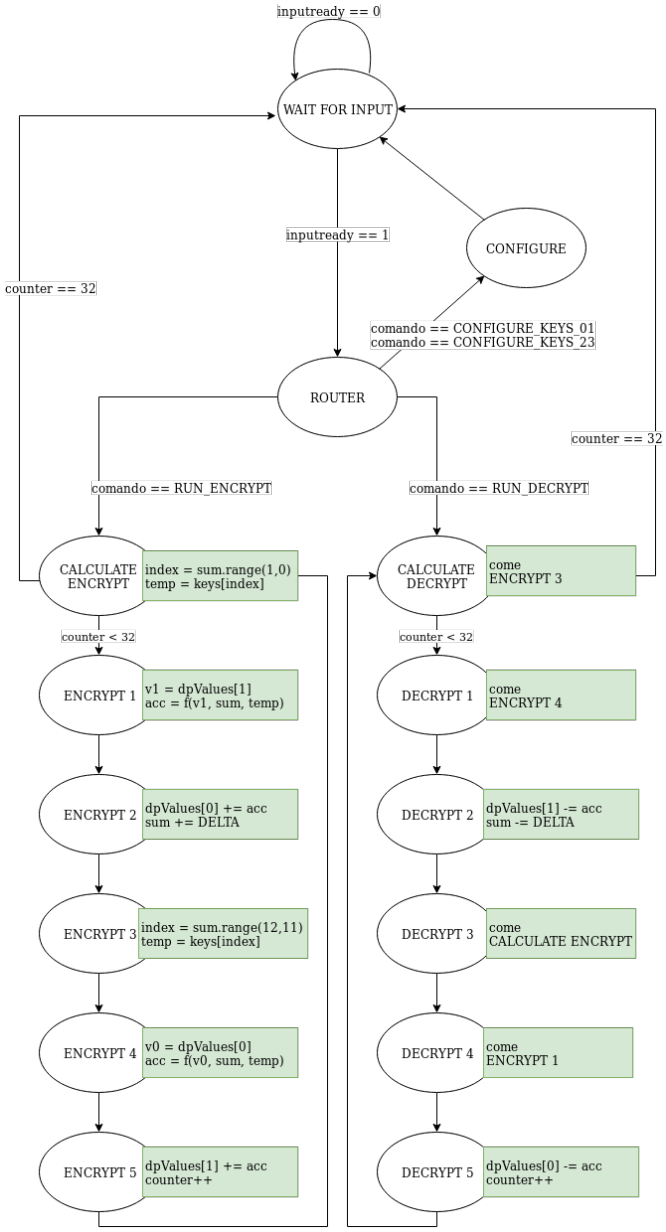

Figura 4. EFSM

```
restart -f

# tolgo e ri-aggiungo i segnali del modulo alla schermata
delete wave *
add wave *

# RESET DEL MODULO
# il clock parte con 0, resta basso 1ns, viene alzato,
# resta alto 1ns, e ripeto il processo ogni 2ns
force clock 0 0 ns, 1 1 ns -repeat 2
force reset 1 0 ns, 0 2 ns
force input_ready 0 0 ns
run 10ns

# CONFIGURAZIONE PRIMA PARTE DELLE CHIAVI
# i segnali che dico partire da 0ns non ripartono davvero
# dell'inizio ma dal punto in cui si era fermata la sim.
force op1 16#AAAABBBB 0 ns
force op2 16#CCCCDDDD 0 ns
force command 2#00    0 ns
```

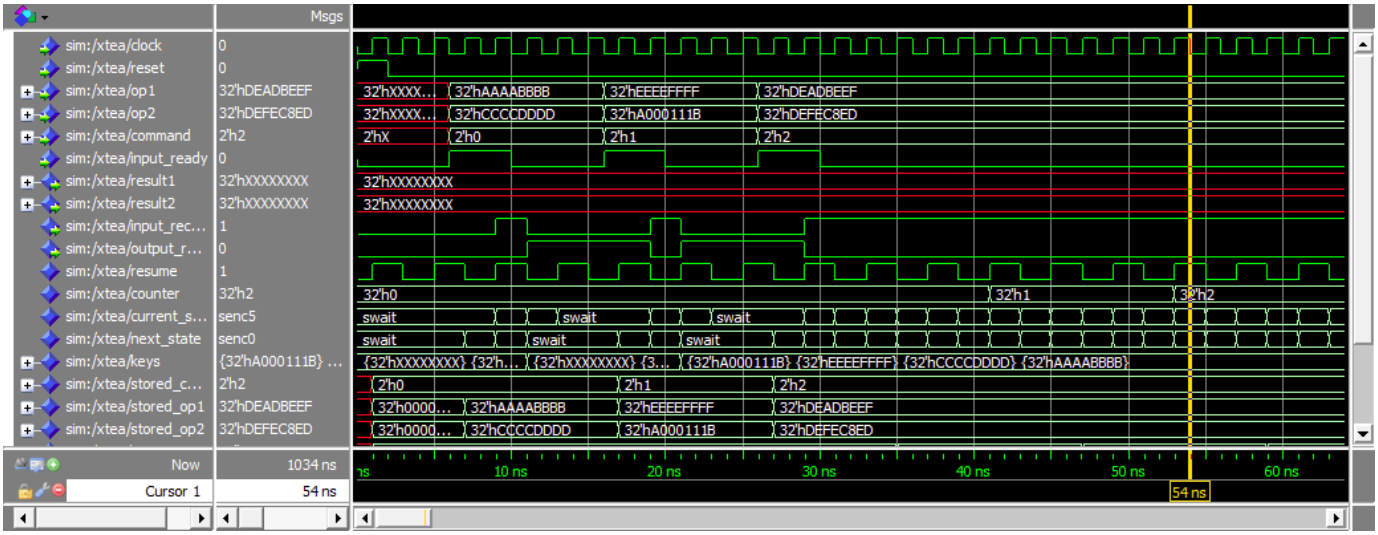

Figura 5. Porzione della simulazione. La fase di reset parte a 0ns. Le fasi di configurazione della prima e della seconda parte delle chiavi partono a 6ns e 16ns. La fase di criptazione del messaggio parte a 21ns.

```
force input_ready 1   0 ns
run 4 ns
force input_ready 0
run 6 ns
```

Il software permette di salvare le forme d'onda nel formato proprietario `wlf`. Una porzione di questa è visibile in Figura 5. All'occorrenza, questo può essere convertito nel formato aperto `vcd` con il tool `wlf2vcd`. Il risultato può essere visto con il software Gtkwave.

V. Sintesi in Vivado

I sorgenti utilizzati all'interno di Modelsim vengono importati in Xilinx Vivado.

Il modulo viene testato ora attraverso un modulo testbench `xtea_tb.vhd`. Questo mima il comportamento del file di stimoli ma in modo più agevole, poichè il linguaggio VHDL permette istruzioni `wait port'event and port = '1'` che permettono di aspettare indefinitamente l'evento su una porta prima di proseguire con il resto del codice, anzichè obbligarci a fare i calcoli della temporizzazione a mano.

Una volta che abbiamo appurato il corretto funzionamento del modulo, procediamo con la sintesi. Il modulo viene sintetizzato per la piattaforma PYNQ con codice `xc7z020clg400-1`. Il modulo risulta sintetizzabile, e le risorse utilizzate risultano quelle di Tabella I.

Site Type	Used	Fixed	Available	Util (%)
Slice LUTs*	486	0	53 200	0.91
LUT as Logic	486	0	53 200	0.91
LUT as Memory	0	0	17 400	0.00
Slice Registers	441	0	106 400	0.41
Register as Flip Flop	15	0	106 400	0.01
Register as Latch	426	0	106 400	0.40
F7 Muxes	0	0	26 600	0.00
F8 Muxes	0	0	13 300	0.00

Tabella I
Utilizzo delle componenti di `xtea`

L'implementazione su FPGA risulta invece un problema col design attuale. Infatti la scheda seleziona permette di poter solamente 125 porte di input/output mentre per il nostro modello ne servono 135. Se volessimo utilizzare il modulo come sistema standalone sulla scheda, dobbiamo trovare il modo di ridurre il numero di IO. Una soluzione semplice è implementare un componente wrapper di `xtea`, con porte dati più piccole, che passano i valori al nostro modulo in più cicli di clock (`input_ready` viene messo a 1 solo quando il valore è stato salvato completamente).

Viene fatta l'implementazione di questo componente all'interno del file `xtea_s2p.vhd`. L'implementazione avviene correttamente, e le porte di IO usate sono 38.

In ogni caso, si presuppone che il cifratore sia inserito all'interno di un sistema embedded più ampio, quindi il problema del numero di porte in realtà è meno grave di quanto ipotizzato inizialmente.

VI. Sintesi ad alto livello con Vivado HLS

Proviamo ad importare il file di specifiche del modulo all'interno di Vivado HLS. L'unico file fornito viene diviso in un sorgente ed in un testbench (quest'ultimo contiene la sola funzione `main`).

Per testare la correttezza delle specifiche avviamo la simulazione del codice C. Il risultato è corretto e ne segue il log (stampa del logo omessa):

```
INFO: [SIM 2] *************** CSIM start ***************
INFO: [SIM 4] CSIM will launch GCC as the compiler.
   Compiling ../../xtea_tb.cpp in debug mode
   Compiling ../../xtea.cpp in debug mode
   Generating csim.exe
First invocation:
   - the encryption of 12345678 and 9abcdeff
   - with key 6a1d78c88c86d67f2a65bfbeb4bd6e46
   is: 99bbb92b, 3ebd1644
Second invocation:
   - the decryption of 99bbb92b and 3ebd1644
   - with key 6a1d78c88c86d67f2a65bfbeb4bd6e46
   is: 12345678, 9abcdeff
Done!!
INFO: [SIM 1] CSim done with 0 errors.
INFO: [SIM 3] *************** CSIM finish ***************
```

Infine avviamo la sintesi. In output abbiamo un file `xtea.vhd`. Questo viene importato in Vivado e sintetizzato. La sintesi ha esito positivo. L'implementazione ha esito negativo sempre a causa del numero di IO.

Site Type	VHDL RTL	Specifiche C++
Slice LUTs	486	696
LUT as Logic	486	696
LUT as Memory	0	0
Slice Registers	441	606
Register as Flip Flop	15	606
Register as Latch	426	0
F7 Muxes	0	0
F8 Muxes	0	0

Tabella II
CONFRONTO

VII. CONCLUSIONI

La Tabella II confronta l'utilizzo di componenti delle varie versioni del modulo.

Vediamo come la versione VHDL occupa circa il 15% delle risorse (porte logiche, memoria) rispetto alla versione sintetizzata a partire dalle specifiche. La perdita di spazio è in realtà irrisoria rispetto al tempo perso per implementare il VHDL, quindi è abbastanza ovvio che questa prima versione è da preferirsi.

Infine notiamo come la versione VHDL utilizzi un certo numero di latch invece che flip-flops, sintomo esiste un processo in cui uno o più segnali vengono scritti in alcuni branch mentre in altri no. Questo difetto è prodotto da una svista del programmatore. Nella versione sintetizzata questi problemi non compaiono.