Modellazione in VHDL di un modulo hardware basato sull'algoritmo XTEA

Vladislav Bragoi - VR436747

***Sommario*—In questo documento vengono descritte le scelte progettuali adottate per implementare l'algoritmo eXtended TEA (XTEA) in un modulo hardware, utilizzando il linguaggio di descrizione VHDL.**

I. Introduzione

Il progetto svolto ha l'obiettivo di sviluppare in VHDL sintetizzabile un modulo hw che implementi l'algoritmo XTEA per la cifratura/decifratura di due parole a 32 bit ciascuna, a partire da una versione già sviluppata precedentemente in SystemC a diversi livelli di astrazione. Inoltre, tramite i principali tool di sintesi in commercio, eseguire la sintesi su una FPGA di riferimento quale la Xilinx PYNQ Z1 per verificare la corretta funzionalità del modulo prodotto e toccare con mano i principali motivi per i quali le tecniche di high level synthesis non vengono ancora oggi ampiamente impiegate.

II. Background

In questo progetto è stato utilizzato il linguaggio di descrizione hardware VHDL[1] per il design e la progettazione del modulo hardware, mentre i concetti di High Level Synthesis (HLS) [2] sono stati utilizzati come confronto con una versione generata automaticamente a partire da una descrizione più algoritmica. In particolare, i software utilizzati sono:

- Mentor Graphics Modelsim, per il design e la simulazione del modulo hw,
- Xilinx Vivado per verificare che il modulo progettato sia sintetizzabile,
- Xilinx Vivado HLS per la generazione automatica dell'hardware digitale a partire da una descrizione ad alto livello (ad esempio in C/C++) da confrontare con il risultato della sintesi logica.

III. Metodologia applicata

A. Implementazione

L'implementazione in VHDL segue perfettamente la versione già implementata in SystemC, rispettando l'interfaccia del modulo (in figura 1) e la scomposizione in FSM e DATAPATH già studiata precedentemente (visibile in figura 2). In particolare il componente presenta 6 ingressi a 32 bit ciascuno, di cui 4 per le chiavi da utilizzare durante la cifratura/decifratura e 2 per ricevere in ingresso le parole da cifrare/decifrare. Sono presenti inoltre i segnali di *din_rdy*, *mode* per selezionare la modalità di esecuzione del modulo, *rst* per il reset e *clk* per il clock. In output invece, il modulo restituisce il segnale di *dout_rdy* e 2 segnali a 32 bit per le parole cifrate/decifrate. Di seguito l'interfaccia in VHDL:

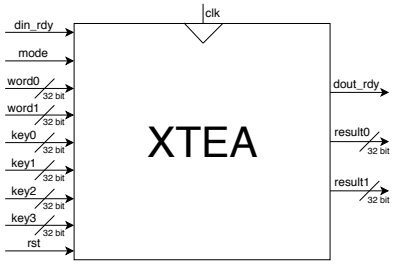

Figura 1: Interfaccia del modulo

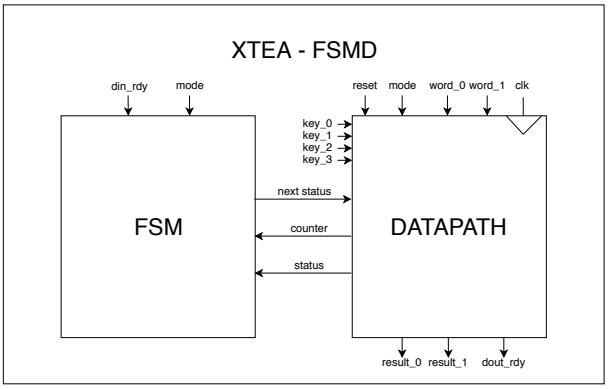

Figura 2: Scomposizione del modulo in fsm e datapath

```
entity xtea is
    port (
        clk         : in    bit;
        rst         : in    bit;
        mode        : in    bit;
        din_rdy     : in    bit;
        dout_rdy    : out   bit;
        key0        : in    unsigned (31 downto 0);
        key1        : in    unsigned (31 downto 0);
        key2        : in    unsigned (31 downto 0);
        key3        : in    unsigned (31 downto 0);
        word0       : in    unsigned (31 downto 0);
        word1       : in    unsigned (31 downto 0);
        result0     : out   unsigned (31 downto 0);
        result1     : out   unsigned (31 downto 0));
end xtea;
```

Per quanto riguarda invece la fase di progettazione, i due processi *fsm* e *datapath* sono implementati per rispettare lo schema rappresentato nella EFSM in figura 6. In particolare il primo processo è sensibile ai segnali *STATUS* e *din_rdy* e si occupa di aggiornare lo stato prossimo del sistema, mentre il secondo processo, implementato rispettando lo stile 4 dei processi in VHDL, si occupa di aggiornare lo stato corrente del sistema, eseguire le operazioni logico-aritmetiche definite dall'algoritmo e infine scrivere il risultato sulle porte di output.

B. Simulazione

Per riuscire a simulare il modello, è necessario utilizzare il software Mentor Graphics Modelsim citato precedentemente. I comandi da utilizzare, una volta creato un nuovo progetto e importati i due file `xtea.vhd` e `stimuli.do`, per lanciare la simulazione sono:

```
# carica il modello in sessione
$ vsim work.xtea
# lancia lo script di simulazione
$ do stimuli.do
```

Da notare che nello script di simulazione, oltre ad una prima fase di configurazione dei segnali su cui fare il tracing grafico, viene eseguita una fase di soft reset del modulo. Questo per assicurarsi che il sistema riparta dallo stato iniziale previsto. Successivamente vengono assegnati alle varie porte del componente hw i valori desiderati. Ad esempio, per la cifratura vengono eseguiti i seguenti comandi per cifrare le parole esadecimali *12345678* e *9abcdeff*:

```
force word0 16#12345678 0 ns
force word1 16#9abcdeff 0 ns
force key0 2#01101010000111010111100011001000 0 ns
force key1 2#10001100100001101101011001111111 0 ns
force key2 2#00101010011001011011111110111110 0 ns
force key3 2#10110100101111010110111001000110 0 ns
force din_rdy 1 0 ns
force rst 1 0 ns
...
run 600 ns
```

e la simulazione viene fatta durare 600 nanosecondi. Dopo circa 550 nanosecondi infatti, sul grafico (vedi figura 5) è possibile vedere che il segnale *dout_rdy* è impostato a 1 e nelle porte di output *result_0* e *result_1* si trovano le due parole cifrate. La simulazione completa è tracciata nel file `vsim.wlf`.

C. Sintesi

Per verificare che il modulo progettato fosse sintetizzabile è stato necessario utilizzare il tool Xilinx Vivado. Una volta caricato il file `xtea.vhd` e impostata la piattaforma PYNQ *xc7z020clg400-1* è stato possibile lanciare la sintesi, di cui ne viene riportato il risultato in figura 3. Dalla figura è possibile notare come la scelta di mantenere l'interfaccia già definita per la versione in SystemC ci richieda un utilizzo di 261 porte di I/O, risultando dunque non implementabile direttamente sulla FPGA scelta. In realtà questo limite può essere superato abbastanza tranquillamente se si utilizzano i registri della board, su cui mappare le porte del circuito e sfruttando il bus AXI[3] per una comunicazione diretta, oppure costruendo una versione diversa del modello, avente un terzo processo che si occupa di caricare i valori sfruttando più cicli di clock e solo al completamento dell'operazione sbloccare gli altri due processi aggiornando i segnali a cui questi due sono sensibili.

D. Sintesi ad alto livello

Per confrontare il codice sviluppato con uno generato automaticamente grazie alla sintesi ad alto livello, è stato necessario utilizzare il tool Xilinx Vivado HLS già citato nelle sezioni sopra. Questo tool ha permesso di generare

Resource	Estimation	Available	Utilization %
LUT	285	53200	0.54
FF	205	106400	0.19
IO	261	125	208.80
BUFG	1	32	3.13

Figura 3: Utilizzo delle risorse su piattaforma PYNQ

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	2451	-
FIFO	-	-	-	-	-
Instance	-	-	-	-	-
Memory	-	-	-	-	-
Multiplexer	-	-	-	134	-
Register	-	-	1332	-	-
Total	0	0	1332	2585	0
Available	280	220	106400	53200	0
Utilization (%)	0	0	1	4	0

Figura 4: Risultato della sintesi ad alto livello su piattaforma PYNQ

automaticamente del codice VHDL a partire da un sorgente in C++ in cui l'algoritmo veniva descritto ad alto livello. Il risultato della sintesi è visibile in figura 4. Questa versione soffre comunque del problema relativo al sovrautilizzo delle porte di I/O, risultando non implementabile direttamente sulla PYNQ. Si rivelano necessarie anche qui dunque le accortezze citate sopra.

IV. Risultati

I risultati relativi all'implementazione del modulo hw in VHDL sono già stati in parte trattati precedentemente. La scelta di restare fedeli alla versione SystemC ha permesso un porting molto più rapido nel nuovo linguaggio ma ha richiesto qualche accortezza in più nella gestione dei vincoli legati alla piattaforma hardware di riferimento (ad esempio l'utilizzo delle risorse hardware disponibili). Con la simulazione inoltre, di cui rimando nuovamente alla figura 5 per un'analisi più dettagliata, è stato possibile verificare la correttezza dell'implementazione, passaggio fondamentale per un successivo deploy su una board effettiva quale la PYNQ Z1 utilizzata nel tool di sintesi.

V. Conclusioni

Il confronto tra la versione del modulo XTEA sviluppato in VHDL e la versione derivante da una sintesi ad alto livello, ha permesso di capire come un codice automatico generalmente si presenta essere molto più complesso, impegnativo da gestire e mantenere, e soprattutto molto più oneroso dal punto di vista dell'utilizzo delle risorse hardware, dato che partendo da un codice ad alto livello, si hanno informazioni più generiche sul modello da costruire. Questo è uno dei motivi che fanno sì che ancora oggi, nonostante i diversi tool in commercio, lo sviluppo diretto venga preferito ad una versione automatizzata. Il modo migliore dunque per costruire un circuito digitale risulta essere quello percorso in questo documento, ovvero passare da una versione in C++/SystemC ad alto livello, ad una versione in VHDL/Verilog più dettagliata, verificando la correttezza del

componente prodotto con simulazioni software e/o test diretti su hardware programmabile prima di un eventuale impiego su larga scala costruendo hardware specifico, almeno fino a quando le tecniche di high level synthesis non permettano di raggiungere la versatilità di un codice costruito a mano.

RIFERIMENTI BIBLIOGRAFICI

[1] "Ieee standard vhdl language reference manual," *IEEE Std 1076-2008 (Revision of IEEE Std 1076-2002)*, pp. c1–626, Jan 2009.

[2] A. M. Philippe Coussy, *High-Level Synthesis*. Springer, Dordrecht, 2008.

[3] Xilinx, "Axi reference guide," https://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/v13_4/ug761_axi_reference_guide.pdf, 2012.

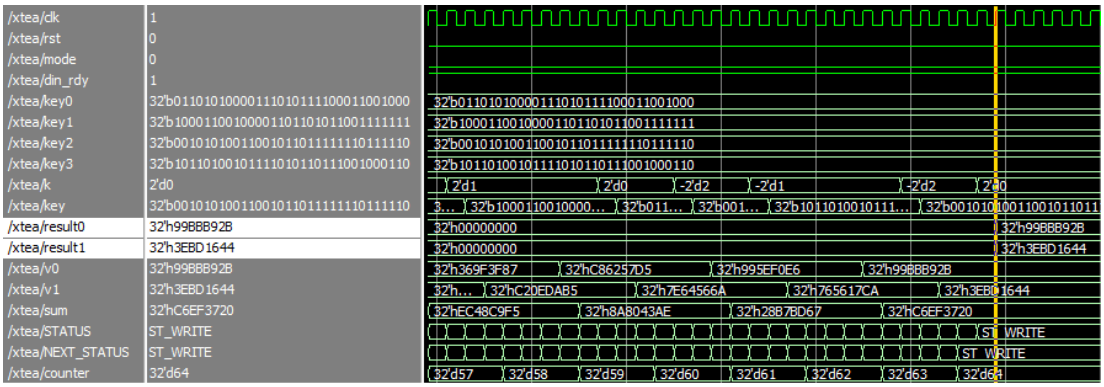

Figura 5: Esecuzione della cifratura delle parole *12345678* e *9abcdeff*

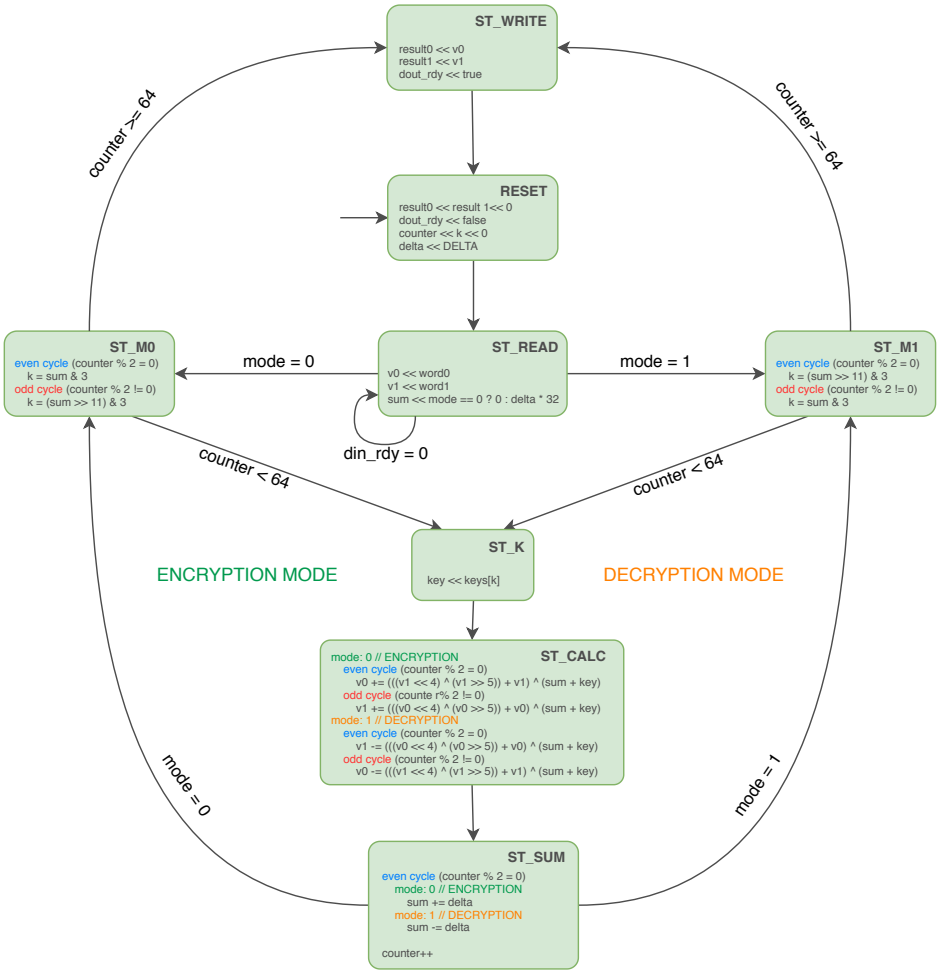

Figura 6: EFSM: rappresenta la completa funzionalità del modulo