POLITECNICO DI TORINO

Politecnico di Torino

Telsy

Corso di Laurea Magistrale in Ingegneria Elettronica

Tesi di Laurea Magistrale

Implementazione hardware di SHA-3 e HMAC su FPGA

Relatori
prof. Maurizio Martina
ing. Andrea Molino
ing. Gabriele Coppolino

Candidato
Giulio Pecoraro

Aprile 2021

Sommario

Nell'era moderna caratterizzata da bitcoin e da trasmissioni di migliaia di dati sensibili, un ruolo importante per la sicurezza delle informazioni viene ricoperto dalla crittografia che si fonda su algoritmi di cifratura e autenticazione. Fanno parte della seconda famiglia le funzioni hash che lavorano su dati in binario di lunghezza variabile producendo una rappresentazione condensata del messaggio iniziale.
Lo scopo di questa tesi, sviluppata in Telsy, azienda leader nell'ambito della crittografia e della Cyber Security, è quello di analizzare e sviluppare su FPGA un algoritmo di hash di recente pubblicazione noto come SHA-3.
Il primo passo per arrivare ad un'implementazione efficace dell'algoritmo è stato studiare le basi matematiche dello SHA-3 per acquisire una conoscenza approfondita del suo funzionamento. La fase di sviluppo, in particolare dello SHA3-512, è proseguita su due strade differenti, una di implementazione software dell'algoritmo da eseguire sul processore NIOS II presente sull'FPGA e l'altra di implementazione hardware da sintetizzare sulla medesima scheda in modo da poter dedurre la soluzione migliore in termini di prestazioni.
Per la prima implementazione è stato utilizzato un codice in C creato dagli stessi sviluppatori dell'algoritmo a cui è stata aggiunta un'interfaccia esterna per ricevere il messaggio in ingresso da elaborare e per eseguire il codice sul processore. Determinate funzioni per la gestione del timer presente sulla scheda hanno permesso di misurare il tempo di esecuzione della funzione. Per l'implementazione hardware, un codice VHDL opportunamente adattato all'obiettivo ha consentito di ricevere gli stessi ingressi utilizzati nell'implementazione precedente, in modo da poter confrontare le due implementazioni. Un'apposita interfaccia ha permesso il passaggio di dati e di controlli tra il processore e il blocco hardware dello SHA-3. Dopo aver testato il corretto funzionamento del blocco sono stati misurati i tempi di esecuzione per differenti messaggi in ingresso in modo da poter confrontare le due soluzioni.
Nella seconda parte del lavoro è stato sviluppato, seguendo gli stessi passi dello SHA-3, un algoritmo per l'autenticazione dei messaggi noto come HMAC. Una caratteristica di questo metodo, che permette di creare un codice di autenticazione del messaggio in ingresso basandosi su una chiave segreta e su una funzione di hash qualunque, ha permesso il riutilizzo dello stesso blocco SHA3-512.

Abstract

In the modern era characterized by bitcoin and the transmissions of thousands of sensitive data, an important role for information security is played by cryptography, which is based on encryption and authentication algorithms. Hash functions are part of the second family and, starting from a variable-length input data, they produce a condensed representation.
The purpose of this thesis, developed in Telsy, a leading company in the field of cryptography and Cybers Security, is to analyze and develop a recently published hash algorithm known as SHA-3 on FPGA.
The first step in arriving at an effective implementation of algorithm was to study the mathematical foundations of SHA-3 to acquire a thorough understanding of its functioning. The development phase, in particular of SHA3-512, continued on two different paths, one of software implementation of the algorithm to be performed on the NIOS II processor present on the FPGA and the other of hardware implementation to be synthesized on the same board to be able to deduce the best solution in terms of performance.
For the first implementation, a C code created by the developers of the algorithm was used to which an external interface was added to receive the incoming message to be processed and to execute the code on the processor. Certain functions for managing the timer on the board have made it possible to measure the execution time of the function.
For the hardware implementation, a suitably tailored VHDL code made it possible to receive the same inputs used in the previous implementation, so that the two implementations could be compared. A special interface allowed the passage of data and controls between the processor and the hardware block of the SHA-3. After testing the correct functioning of the block, the execution times for different incoming messages were measured in order to compare the two solutions. In the second part of the work, an algorithm for message authentication known as HMAC was developed following the same steps as SHA-3. A feature of this method, which uses a secret key and any hash function to create an authentication code of the incoming message, has allowed the reuse of the same SHA3-512 block.

Ringraziamenti

Vorrei esprimere tutta la mia gratitudine all'azienda Telsy che, nonostante l'emergenza epidemiologica in corso, ha permesso l'utilizzo dei suoi spazi per lo sviluppo della tesi.
Un ringraziamento particolare per le enormi conoscenze che sono stati in grado di trasmettermi va ai miei tutor Gabriele Coppolino e Andrea Molino che hanno creduto in me scegliendomi come tesista per il loro progetto e che con la loro pazienza e disponibilità mi hanno supportato anche a distanza e aiutato a superare le difficoltà dovute alla situazione pandemica.
Ringrazio sentitamente anche il professore Maurizio Martina del Politecnico di Torino che ha supervisionato il mio progetto fornendomi indicazioni e correzioni preziose.

Indice

Elenco delle tabelle 1

Elenco delle figure 2

Glossario 4
Acronimi . . . 4
Operazioni di base . . . 4

1 Introduzione 5

2 Funzioni hash 6
2.1 Definizione e utilizzi . . . 6

3 Algoritmo SHA-3 8
3.1 Definizione . . . 8
3.2 Permutazione *Keccak-p* . . . 9
3.2.1 State mapping θ . . . 12
3.2.2 State mapping ρ . . . 12
3.2.3 State mapping π . . . 13
3.2.4 State mapping χ . . . 13
3.2.5 State mapping ι . . . 13
3.3 Costruzione spugna . . . 14
3.3.1 Forma esadecimale del padding . . . 16
3.4 Risorse utilizzate . . . 17
3.5 Implementazione software . . . 18
3.5.1 Risultati ottenuti . . . 23
3.6 Implementazione hardware . . . 26
3.6.1 Top_entity . . . 26
3.6.2 Padder . . . 28
3.6.3 F_permutation . . . 30
3.6.4 Simulazione iniziale dell'architettura . . . 31

3.6.5 Interfacciamento con il processore NIOS II 32
3.6.5.1 Avalon Memory Mapped Interface 33
3.6.5.2 Avalon Streaming Interface 34
3.6.5.3 FSM 36
3.6.5.4 Sintesi del sistema 37
3.6.5.5 Simulazione dell'interfacciamento 38
3.6.5.6 Misurazione dei tempi di esecuzione 40
3.6.6 Risultati ottenuti 42
3.7 Confronto tra le due implementazioni 45

4 Algoritmo HMAC 47
4.1 Definizione 47
4.2 Implementazione software 49
4.2.1 Risultati ottenuti 50
4.3 Implementazione hardware 53
4.3.1 Datapath 53
4.3.2 Interfacciamento con il processore NIOS II 55
4.3.2.1 FSM 56
4.3.2.2 Sintesi del sistema 58
4.3.2.3 Simulazione dell'interfacciamento 59
4.3.2.4 Misurazione dei tempi di esecuzione 60
4.4 Risultati ottenuti 62
4.5 Confronto tra le due implementazioni 64

5 Conclusioni 66

Bibliografia 67

Elenco delle tabelle

3.1 *Lunghezza della permutazione Keccak-p e relative quantità* 10
3.2 *Offset di* ρ 13
3.3 $RC[n_r]$ *in formato esadecimale con* $0 < n_r < 24$ 14
3.4 *Forma esadecimale del padding per lo SHA-3* 16
3.5 *Dimensione del codice in kB per la funzione SHA3-512* 23
3.6 *Tempi di esecuzione in versione software per lo SHA3-512* 24
3.7 *Blocco PAD* 29
3.8 *Area occupata e bit di memoria per lo SHA3-512* 42
3.9 *Tempi di esecuzione in versione hardware per lo SHA3-512* 43
3.10 *Confronto tra le risorse utilizzate per le due implementazioni dello SHA3-512* 45
3.11 *Confronto tra i tempi di esecuzione per le due implementazioni dello SHA3-512* 45
4.1 *Dimensione del codice in kB per la funzione HMAC* 51
4.2 *Tempi di esecuzione in software per l'algoritmo HMAC* 51
4.3 *Area occupata e bit di memoria per l'algoritmo HMAC* 62
4.4 *Tempi di esecuzione in versione hardware per l'algoritmo HMAC* . 62
4.5 *Confronto tra le risorse utilizzate per le due implementazioni dell' HMAC* 64
4.6 *Confronto tra i tempi di esecuzione per le due implementazioni dell' HMAC* 64

Elenco delle figure

3.1 *Rappresentazione dello state array [5]* 10
3.2 *Rappresentazione delle parti dello state array [5]* 11
3.3 *Costruzione spugna [5]* 15
3.4 *Qsys per lo SHA3-512 versione software* 21
3.5 *Messaggio stampato nella console di Eclipse per lo SHA3-512 versione SW* 23
3.6 *Confronto tra i tempi di esecuzione dello SHA3-512 per le due differenti ottimizzazioni* 24
3.7 *Schema a blocchi del Padder* 28
3.8 *Schema a blocchi della F_permutation* 30
3.9 *Simulazione per la parte di acquisizione dati del blocco SHA3-512* . 31
3.10 *Simulazione per la parte di generazione del risultato del blocco SHA3-512* 31
3.11 *Timing dei segnali dell'interfaccia Avalon-MM [2]* 34
3.12 *Timing dei segnali dell'interfaccia Avalon-ST [2]* 35
3.13 *FSM dello SHA3-512* 36
3.14 *Qsys per lo SHA3-512 versione hardware* 37
3.15 *Address Map per lo SHA3-512 versione hardware* 38
3.16 *Forme d'onda per il processo di acquisizione dati in SignalTap per lo SHA3-512* 39
3.17 *Forme d'onda per il processo di rilascio dei dati in SignalTap per lo SHA3-512* 39
3.18 *Forme d'onda per lo SHA3-512 visualizzate in Modelsim utilizzando il testbench del Qsys* 40
3.19 *Esempio di stampa del risultato nella console di Eclipse per la versione hardware dello SHA3-512* 41
3.20 *Esempio di ALM della scheda FPGA Cyclone V [3]* 42
3.21 *Tempi di esecuzione dello SHA3-512 in versione hardware* 43
3.22 *Confronto tra i tempi di esecuzione dello SHA3-512 per le due implementazioni* 46
4.1 *Messaggio stampato nella console di Eclipse per HMAC versione SW* 50

4.2 *Confronto tra i tempi di esecuzione dell' HMAC per le due differenti ottimizzazioni* 52
4.3 *Schema a blocchi HMAC* 53
4.4 *FSM HMAC* 56
4.5 *Qsys per HMAC versione hardware* 58
4.6 *Address Map per l' HMAC versione hardware* 59
4.7 *Forme d'onda per il processo di acquisizione dati in SignalTap per l' HMAC* 59
4.8 *Forme d'onda per il processo di rilascio dei dati in SignalTap per lo SHA3-512* 59
4.9 *Forme d'onda per l'HMAC visualizzate in Modelsim utilizzando il testbench del Qsys* 60
4.10 *Esempio di stampa del risultato nella console di Eclipse per la versione hardware dell' HMAC* 61
4.11 *Tempi di esecuzione dell' HMAC in versione hardware* 63
4.12 *Confronto tra i tempi di esecuzione dell' HMAC per le due implementazioni* 65

Glossario

Acronimi

ALM	Adaptive logic module.
HMAC	Keyed-Hash Message Authentication Code.
LUT	Look-up table.
NIST	National Institute of Standards and Technology.
SHA-3	Secure Hash Algorithm-3.
SHAKE	Secure Hash Algorithm KECCAK.
XOF	Extendable-output function.
XOR	Exclusive-OR.

Operazioni di base

$\boldsymbol{X \oplus Y}$	Stringa risultante dall'applicazione bit a bit dell'operazione exclusive-OR a X e Y.
$\boldsymbol{X \mid\mid Y}$	Concatenazione di X e Y.
$\boldsymbol{m \bmod n}$	Per m e n interi, m mod n è un intero r tale che $0 \leq r < n$ e $m - r$ è multiplo di n.

Capitolo 1

Introduzione

La crittografia, nel mondo dell'informazione, ricopre un ruolo fondamentale per le sue capacità di proteggere le informazioni e di garantire l'autenticità dei messaggi inviati e ricevuti. I dati vengono convertiti da un formato leggibile ad uno codificato e potranno essere riletti solo dopo la decrittazione.
In crittografia vengono applicati due processi fondamentali, la cifratura e la codifica. La codifica lavora su parole o frasi mentre la cifratura opera sulle lettere individuali di un alfabeto o per trasposizione, chiamata anche permutazione, mescolando i caratteri di un messaggio in un nuovo ordine, o per sostituzione, scambiando un carattere con un altro. Un ruolo fondamentale viene ricoperto dagli algoritmi di cifratura e di autenticazione e fanno parte della seconda famiglia, le funzioni hash che lavorano su dati in binario per generare una rappresentazione condensata del messaggio.
L'algoritmo di hash esaminato nel capitolo 3, è lo SHA-3, in particolare lo SHA3-512, l'ultimo membro della famiglia di standard *"Secure Hash Algorithm"* pubblicata dal NIST. La famiglia dello SHA-3 impiega la permutazione che in questo Standard è chiamata Keccak-p, progettata da un team tra cui l'italiano Bertoni.
L'algoritmo SHA3-512 è stato analizzato prima dal punto di vista matematico ed in seguito, è stato implementato a livello software e hardware su FPGA per valutare le sue prestazioni.
Nel capitolo 4 è stato sviluppato anche un metodo di autenticazione del messaggio basato su una funzione di hash generica chiamato HMAC che utilizza una combinazione del messaggio originale e una chiave segreta per la generazione del codice. La funzione di hash scelta per questo algoritmo è nuovamente lo SHA3-512.
Anche HMAC è stato prima studiato e successivamente implementato in versione software e hardware su FPGA per valutarne le prestazioni.

Capitolo 2

Funzioni hash

2.1 Definizione e utilizzi

Le funzioni hash lavorano su dati in binario e vengono utilizzate per generare una rappresentazione condensata del messaggio. L'input è chiamato *message* e l'output è chiamato *digest*. La lunghezza del messaggio in ingresso è arbitraria mentre la lunghezza del *digest* è fissa.
Le funzioni hash hanno un ruolo molto importante in crittografia per le loro particolari proprietà ritenute fondamentali. Innanzitutto, il *digest* è strettamente legato al messaggio in entrata poiché ogni messaggio dovrebbe generare un *digest* univoco. Infatti, se si considerano due messaggi che differiscono solo per un carattere, le loro immagini hash saranno diverse. Inoltre, le funzioni hash sono unidirezionali, cioè dal *digest* non si può, dal punto di vista computazionale, risalire al messaggio originale [1]. L'unico modo per visualizzare l'informazione originaria è quello di applicare un attacco di forza bruta *(brute force)* ma poiché il numero di possibili casi da testare è elevatissimo, è semplice comprendere come questo sistema non sia ragionevolmente eseguibile a livello pratico. Inoltre, anche se si trovasse una combinazione di dati in ingresso in grado di generare lo stesso valore di hash, non si avrebbe la certezza che si tratti dei dati in ingresso originali [13]. Infine, una terza proprietà garantisce la bassissima probabilità di trovare una collisione, ovvero due messaggi diversi che producono il medesimo *digest*. Quando due *message* producono lo stesso hash, si parla di collisione. In generale la qualità di una funzione hash è misurata direttamente in base alla difficoltà di individuare due ingressi che generino una collisione. Per affermare la sicurezza di un determinato algoritmo di hashing, è sufficiente verificare la quasi impossibilità di ottenere una collisione. In caso contrario si dimostra la sua vulnerabilità e di conseguenza l'algoritmo non può considerarsi sicuro. Alcuni esempi di algoritmi vulnerabili sono SNEFRU, MD2, MD4 ed MD5 [1].

Le funzioni hash crittografiche sono componenti fondamentali in una varietà di applicazioni per la sicurezza delle informazioni, come la generazione e la verifica della firma digitale o la produzione di bit pseudocasuali.
La firma digitale è un sistema per garantire la sicurezza di una transazione effettuata tra due parti e ne definisce il rapporto di fiducia. Utilizza un algoritmo a chiave pubblica che si basa su una coppia di chiavi e permette di effettuare in modo certo e sicuro la transazione garantendone sia l'integrità e sia l'autenticazione del mittente che l'ha firmata.
La sicurezza di tutti gli algoritmi come SHA-1, SHA-2 e quelli a chiave pubblica che si basano sulle funzioni unidirezionali che sono semplici da calcolare in modo diretto ma praticamente impossibile in modo inverso potrà essere compromessa dall'avvento dei computer quantistici che permetteranno invece di eseguire in modo immediato il calcolo inverso [12].

Capitolo 3

Algoritmo SHA-3

3.1 Definizione

SHA-3 è l'ultimo membro della famiglia di standard *"Secure Hash Algorithm"* pubblicata dal NIST nell'agosto del 2015 e si differenzia in modo significativo rispetto alle precedenti versioni di algoritmi della stessa famiglia, lo SHA-1 e lo SHA-2. Ciascuna delle funzioni SHA-3 si basa su un'istanza dell'algoritmo KECCAK, progettato da Guido Bertoni, Joan Daemen, Michaël Peeters e Gilles Van Assche, che il 2 ottobre 2012 fu selezionato dall'ente governativo americano NIST come migliore implementazione del nuovo e futuro standard per la crittografia SHA-3 al termine della *SHA-3 Cryptographic Hash Algorithm Competition*.
Il processo che ha portato alla scelta di un nuovo standard per le funzioni di hash ebbe inizio nel 2007 poiché furono registrati un certo numero di attacchi e individuati punti di debolezza nei predecessori dello standard SHA-2. Il rischio era che le implementazioni maggiormente utilizzate dello standard SHA-2 prima o poi sarebbero diventate vulnerabili. Poiché la scelta e la definizione di una primitiva per la crittografia richiede parecchio tempo, il NIST avviò il processo di creazione dello SHA-3 con il chiaro intento di trovare un valido sostituto per lo SHA-2 che in realtà si dimostrò più robusto di quanto inizialmente ipotizzato. Con l'obiettivo dichiarato di anteporre a ogni cosa la robustezza, altri aspetti come prestazioni, velocità di esecuzione, numero di risorse necessarie, possibilità di parallelizzare o serializzare la struttura e facilità di implementazione a livello software passarono in secondo piano. Alla fine, la scelta del NIST è ricaduta sull'algoritmo Keccak, che presentava profonde differenze rispetto allo SHA-2. In questo modo si riteneva di ridurre i rischi di violare simultaneamente più algoritmi [13].
La famiglia dello SHA-3 è costituita da quattro funzioni hash criptografiche e da

due funzioni di uscite estendibili (*XOF*). Queste sei funzioni condividono la struttura definita *costruzione spugna*, e per questo sono chiamate funzioni spugna.
Le quattro funzioni hash SHA-3 sono denominate SHA3-224, SHA3-256, SHA3-384 e SHA3-512. In generale, il suffisso dopo il trattino indica la lunghezza fissa del *digest*, ad esempio SHA3-256 produce un *digest* di 256 bit.
Una funzione di uscita estendibile (*XOF*) è una funzione su stringhe di bit in cui l'uscita può essere estesa a qualsiasi lunghezza desiderata. I due SHA-3 *XOF* che il NIST ha standardizzato, si chiamano SHAKE128 e SHAKE256. I suffissi "128" e "256" indicano il numero di bit da cui deriva la robustezza in termini di sicurezza che queste due funzioni possono generalmente supportare.
Lo SHA-3 può sostituire direttamente lo SHA-2 nelle applicazioni correnti, in caso di necessità. Attualmente però, lo SHA-3 non è ancora subentrato allo SHA-2, poiché allo stato attuale nessun attacco è ufficialmente riuscito a produrre una collisione completa con questo algoritmo. A causa però degli attacchi riusciti a MD5, SHA-0 e SHA-1, il NIST ha percepito la necessità di un hash crittografico alternativo e dissimile, che è diventato SHA-3.
L'algoritmo studiato durante lo sviluppo della tesi è lo **SHA3-512**. Dato che per definizione il *digest* ha lunghezza fissa e per convenzione il numero dopo il trattino nel nome dell'algoritmo indica il numero di bit del *digest*, 512 è il numero di bit del *digest* preso in considerazione.
Ciascuna delle sei funzioni SHA-3 impiega la stessa permutazione sottostante, infatti le funzioni SHA-3 sono le *modalità di funzionamento* (*modi*) della permutazione che in questo Standard è chiamata Keccak-p la cui descrizione è tratta dalla pubblicazione [5].

3.2 Permutazione *Keccak-p*

La permutazione Keccak-p è indicata come Keccak-p$[b, n_r]$, dove b è la lunghezza fissa della stringa che è permutata e il parametro n_r è il numero di iterazioni della trasformazione interna chiamata *round*.
La permutazione è definita per ogni $b \in \{25{,}50{,}100{,}200{,}400{,}800{,}1600\}$ e per ogni n_r intero positivo. Un altro parametro importante è ℓ, definito come:

$$\ell = log_2(b/25)$$

Quindi, quando $n_r = 12 + 2\ell$ si definisce

$$\text{Keccak-f}[b] = \text{Keccak-p}[b, 12 + 2\ell].$$

Per sviluppare lo SHA-3 il team propose di utilizzare la permutazione più grande, Keccak-f[1600] e di conseguenza $n_r = 24$. Ogni permutazione consiste nell'iterazione di una funzione semplice senza l'utilizzo di una chiave.
Il *round* della permutazione Keccak-p indicato come *Rnd* consiste in una sequenza

di cinque trasformazioni chiamate *state mapping*. La permutazione riceve in ingresso stringhe di b bit chiamate *states* e produrrà gli *states* di uscita, anche essi stringhe di b bit. Gli *states* di ingresso e di uscita dello *state mapping*, invece, sono organizzati come vettori di bit di ampiezza $5 \times 5 \times w$, dove $w = (b/25)$. I sette possibili valori per le variabili ℓ e b definiti per la permutazione Keccak-p sono riportati nella tabella 3.1:

b	25	50	100	200	400	800	1600
w	1	2	4	8	16	32	64
ℓ	0	1	2	3	4	5	6

Tabella 3.1: *Lunghezza della permutazione Keccak-p e relative quantità*

La stringa che rappresenta lo *state* si indica con S, i cui bit sono indicizzati da $0\ a\ b-1$, così che:

$$S = S[0] \mid\mid S[1] \mid\mid \ldots \mid\mid S[b-2] \mid\mid S[b-1].$$

Si definisce A un vettore di bit $5 \times 5 \times w$, che rappresenta lo *state*, i cui indici sono (x,y,z) con $0 \leq x < 5,\ 0 \leq y < 5,\ e\ 0 \leq z < w$, e $A[x, y, z]$ uno *state array*, cioè una rappresentazione dello stato tramite un vettore tridimensionale.
Dal punto di vista rappresentativo lo *state array* può essere considerato un parallelepipedo costituito da tanti piccoli cubi. Le sue dimensioni dipendono dai parametri b e w come mostrato in figura 3.1:

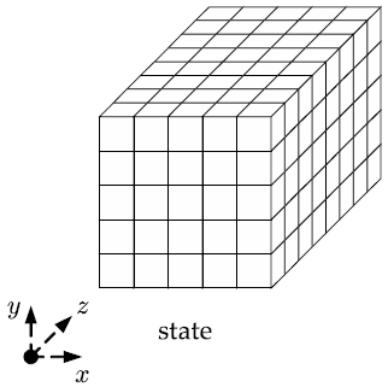

Figura 3.1: *Rappresentazione dello state array [5]*

Nel caso rappresentato $b = 200$ e di conseguenza $w = 8$. I sotto-vettori in due dimensioni sono chiamati *sheets, planes e slices*, mentre i sotto-vettori in singola dimensione sono chiamati *rows, columns e lanes*, rappresentati in figura 3.2:

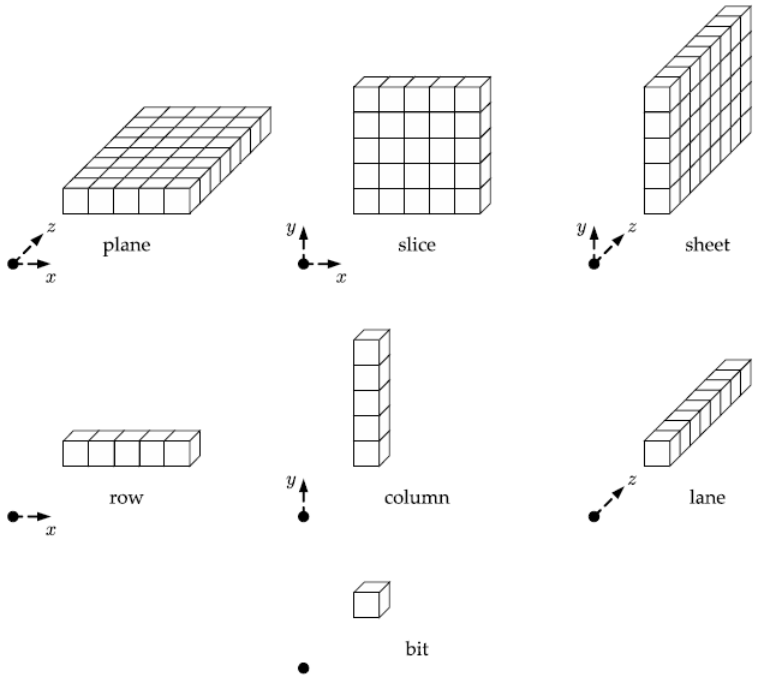

Figura 3.2: *Rappresentazione delle parti dello state array [5]*

Per esempio, si indica con *lane* un sotto-vettore di $b/25$ bit che ha le coordinate x e y costanti.
Dato che l'ingresso della permutazione è S, cioè una stringa di b bit, ma l'ingresso dello *state mapping* è un vettore tridimensionale, prima di eseguire il *round* la stringa S deve essere trasformata in uno *state array*, quindi, per tutte le coordinate (x,y,z) tali che $0 \leq x < 5,\ 0 \leq y < 5,\ e\ 0 \leq z < w$, si ha che:

$$A[x, y, z] = S[w(5y + x) + z].$$

Al termine delle operazioni, lo *state array* finale dovrà essere convertito nella stringa S che rappresenta il risultato definitivo della permutazione. Per eseguire l'operazione inversa, S potrà essere costruita dai *lanes* e *planes* poiché per ogni coppia di interi (i,j) tale che $0 \leq i < 5\ e\ 0 \leq j < 5$, si definisce la stringa:

$$Lane(i, j) = A[i, j, 0] \mid\mid A[i, j, 1] \mid\mid A[i, j, 2] \mid\mid \ldots \mid\mid A[i, j, w - 2] \mid\mid A[i, j, w - 1].$$

dove *Lane(i,j)* indica una stringa di tutti i bit della *lane* le cui coordinate x e y sono i e j. Per ogni intero $0 \leq j < 5$ si definisce la stringa:

$$Plane(j) = Lane(0, j) \mid\mid Lane(1, j) \mid\mid Lane(2, j) \mid\mid Lane(3, j) \mid\mid Lane(4, j).$$

Infine, per calcolare S si ha che:

$$S = Plane(0) \mid\mid Plane(1) \mid\mid Plane(2) \mid\mid Plane(3) \mid\mid Plane(4).$$

Nel mezzo della permutazione si hanno i cinque *state mapping* che compongono il *round* indicati con le lettere greche θ, ρ, π, χ *e* ι. Ogni algoritmo che costituisce questi *state mapping* prende uno *state array* A come ingresso e restituisce un A' come uscita. Dunque la funzione di *round* è una trasformazione che applica i cinque *state mapping*:

$$Rnd(A, i_r) = \iota(\chi(\pi(\rho(\theta(A)))), i_r).$$

Lo *state mapping* ι ha come secondo ingresso oltre ad A anche un intero i_r, chiamato *indice di round*. Si può dunque concludere che lo scopo della permutazione Keccak-p$[b, n_r]$ è quello di iterare la funzione *round* un numero di volte pari a n_r, che nel caso esaminato è uguale a 24.

3.2.1 State mapping θ

L'algoritmo prevede tre fasi:

1. per tutte le coppie (x,z) tali che $0 \leq x < 5$ *e* $0 \leq z < w$ si ha che:

$$C[x, z] = A[x, 0, z] \oplus A[x, 1, z] \oplus A[x, 2, z] \oplus A[x, 3, z] \oplus A[x, 4, z];$$

2. per tutte le coppie (x,z) tali che $0 \leq x < 5$ *e* $0 \leq z < w$ si ha che:

$$D[x, z] = C[(x - 1)\ mod\ 5, z] \oplus C[(x + 1)\ mod\ 5, (z - 1)\ mod\ w];$$

3. per tutte le coordinate (x,y,z) tali che $0 \leq x < 5,\ 0 \leq y < 5$ *e* $0 \leq z < w$ si ha che:

$$A'[x, y, z] = A[x, y, z] \oplus D[x, z].$$

3.2.2 State mapping ρ

L'algoritmo prevede quattro fasi:

1. per tutte le z per cui $0 \leq z < w$ si ha che:

$$A'[0,0, z] = A[0, 0, z];$$

2. siano $(x, y) = (1,0)$;
3. per t da 0 a $(n_r - 1)$ si ha che:
   - per tutte le z per cui $0 \leq z < w$ si ha che:

     $A'[x, y, z] = A[x, y, (z - (t + 1)(t + 2)/2)\ mod\ w];$
   - siano $(x, y) = (y, (2x + 3y)\ mod\ 5)$;
4. ritorna A'.

Per riassumere, l'effetto di ρ è quello di ruotare i bit di ciascuna *lane* di una lunghezza chiamata *offset* che dipende dalle coordinate x e y della *lane*, come evidenziato nella tabella 3.2:

	$x = 3$	$x = 4$	$x = 0$	$x = 1$	$x = 2$
$y = 2$	153	231	3	10	171
$y = 1$	55	276	36	300	6
$y = 0$	28	91	0	1	190
$y = 4$	120	78	210	66	253
$y = 3$	21	136	105	45	15

Tabella 3.2: *Offset di* ρ

3.2.3 State mapping π

L'algoritmo prevede due fasi:

1. per tutte le coordinate (x,y,z) tali che $0 \leq x < 5,\ 0 \leq y < 5\ e\ 0 \leq z < w$ si ha che:

$$A'[x, y, z] = A[(x + 3y)\ mod\ 5, x, z];$$

2. ritorna *A'*.

3.2.4 State mapping χ

L'algoritmo prevede due fasi:

1. per tutte le coordinate (x,y,z) tali che $0 \leq x < 5,\ 0 \leq y < 5,\ e\ 0 \leq z < w$ si ha che:

$$A'[x, y, z] = A[x, y, z] \oplus ((A[(x + 1)\ mod\ 5, y, z] \oplus 1) \cdot A[(x + 2)\ mod\ 5, y, z]);$$

2. ritorna *A'*.

Il punto a destra dell'assegnazione per la fase 1 indica la moltiplicazione di numeri interi, che in questo caso è equivalente all'operazione booleana *AND*.

3.2.5 State mapping ι

Lo *state mapping* ι è caratterizzato dal ricevere come ingresso anche l'indice di arrotondamento i_r. L'algoritmo prevede tre fasi:

1. per tutte le coordinate (x,y,z) tali che $0 \leq x < 5,\ 0 \leq y < 5\ e\ 0 \leq z < w$ si ha che:

$$A'[x, y, z] = A[x, y, z];$$

2. per le coordinate z tali che $0 \leq z < w$ si ha che:

$$A'[0,0, z] = A[0, 0, z] \ \oplus RC[z];$$

3. ritorna *A'*.

Il parametro $RC[z]$ presente nella 2 indica il singolo bit della *costante di round*. Le costanti sono differenti a seconda del numero di *round* come mostrato in tabella 3.3:

$RC[0] = 0x0000000000000001$	$RC[12] = 0x000000008000808B$
$RC[1] = 0x0000000000008082$	$RC[13] = 0x800000000000008b$
$RC[2] = 0x800000000000808A$	$RC[14] = 0x8000000000008089$
$RC[3] = 0x8000000080008000$	$RC[15] = 0x8000000000008003$
$RC[4] = 0x000000000000808B$	$RC[16] = 0x8000000000008002$
$RC[5] = 0x0000000080000001$	$RC[17] = 0x8000000000000080$
$RC[6] = 0x8000000080008081$	$RC[18] = 0x000000000000800A$
$RC[7] = 0x8000000000008009$	$RC[19] = 0x800000008000000A$
$RC[8] = 0x000000000000008A$	$RC[20] = 0x8000000080008081$
$RC[9] = 0x0000000000000088$	$RC[21] = 0x8000000000008080$
$RC[10] = 0x0000000080008009$	$RC[22] = 0x0000000080000001$
$RC[11] = 0x000000008000000A$	$RC[23] = 0x8000000080008008$

Tabella 3.3: $RC[n_r]$ *in formato esadecimale con* $0 < n_r < 24$

3.3 Costruzione spugna

Come già accennato, le sei funzioni che costituiscono lo SHA-3 sono chiamate *funzioni spugna*, cioè basate su permutazioni fisse che possono essere regolate a piacimento, e che condividono la struttura chiamata *costruzione spugna* rappresentata nella figura 3.3:

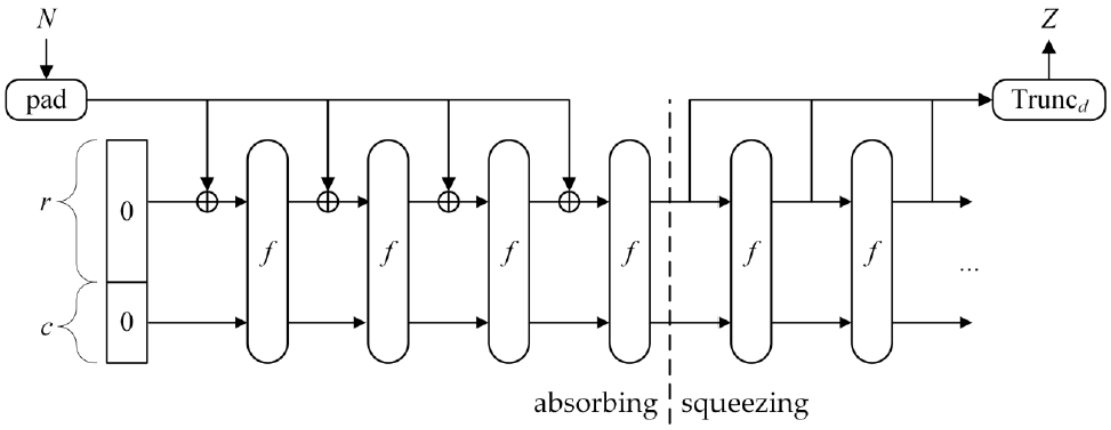

Figura 3.3: *Costruzione spugna [5]*

La costruzione impiega i seguenti tre componenti:

1. una funzione sottostante su stringhe di lunghezza fissa, indicata con f;
2. un parametro chiamato *rate*, indicato con r, e un parametro c che indica la capacità;
3. una regola di riempimento, indicata da pad.

La funzione che la costruzione produce da questi componenti è chiamata *funzione spugna* ed è indicata come $SPONGE[f, pad, r]$. Questa funzione richiede due input: una stringa di bit, indicata con N, e la lunghezza d in bit dell'uscita, $SPONGE[f, pad, r](N, d)$. La particolarità è che l'ingresso viene "assorbito", fase di *absorbing*, ad una velocità fissata e l'output viene rilasciato *("squeezed")* alla stessa velocità. La *funzione spugna* ha una struttura più generale di una funzione hash, con input di lunghezza variabile e output di lunghezza arbitraria basata su una permutazione (f) di lunghezza fissa che opera su un numero stabilito di bit (b), che è anche l'ampiezza dello *state*. Questa permutazione viene eseguita bit a bit e opera su $b = r + c$, dove r indica la velocità e c la capacità. [1]
Le funzioni SHA-3 sono istanze della costruzione spugna in cui la funzione sottostante f, una permutazione, è invertibile. In generale, però, in una costruzione spugna generica, f è una funzione qualsiasi che non per forza deve essere invertibile. Il concetto di invertibilità della funzione hash di un algoritmo criptografico è strettamente legato a quello di vulnerabilità. I precedenti algoritmi utilizzavano funzioni di cui era difficile risalire alle loro funzioni inverse e quindi l'algoritmo veniva violato. Anche lo SHA-3 ricorre a una funzione invertibile ma l'inversione della funzione non comporta rischi per la sicurezza, poiché per costruzione l'inversione della permutazione è nota e non legata alla vulnerabilità.
Il messaggio in input viene suddiviso in blocchi da r bit (se necessario vengono completati con degli 0 se il numero di bit dell'ultimo blocco è minore di r), poi la

funzione spugna assorbe i blocchi e rilascia l'*output*. Nella fase di assorbimento, ad ogni blocco dato in input, viene applicata l'operazione di XOR bit a bit ai primi r bit dello *state* e poi l'intero vettore viene permutato da f. Una volta processati tutti i blocchi del messaggio, la funzione spugna passa alla fase di "rilascio". In questa fase i primi r bit dello *state* vengono rilasciati come blocchi di output e tra un rilascio e l'altro viene nuovamente applicata f all'intero vettore di *state*. Il numero di blocchi in output viene scelto arbitrariamente. Gli ultimi c bit dello *state* non dipendono mai direttamente dai blocchi di *input* e non vengono mai rilasciati come output nella fase di *squeezing*, ma vengono modificati solo dalla permutazione f [1]. Keccak appartiene alla famiglia delle *funzioni spugna*, e in particolare quando $b = 1600$ si parla di $Keccak[c]$ dove:

$$Keccak[c] = SPONGE[KECCAK - p[1600,24], pad10^*1, 1600 - c].$$

Quindi, data una stringa in ingresso N e un'uscita su d bit si ha che:

$$Keccak[c](N, d) = SPONGE[KECCAK - p[1600,24], pad10^*1, 1600 - c](N, d).$$

L'espressione $pad10^*1$ indica la regola di *padding*, dove il simbolo * indica una sequenza di zeri omessa che permette di ottenere un'uscita di lunghezza desiderata. Dato un messaggio in ingresso M si definisce:

$$SHA3 - 512(M) = KECCAK[1024](M||01, 512).$$

La funzione di hash SHA-3 si definisce, dunque, partendo dalla definizione della permutazione $Keccak[c]$, aggiungendo un suffisso di due bit al messaggio M e specificando la lunghezza dell'uscita. Infatti, il parametro N risulta essere $N = M||01$ e la capacità c sarà sempre uguale al doppio del parametro d [5].

3.3.1 Forma esadecimale del padding

In molte applicazioni il *padding* viene rappresentato in forma esadecimale e il messaggio in ingresso ha una lunghezza in bit pari a 8m, dove m è un numero intero positivo. Conoscendo m è possibile calcolare il numero totale di byte da aggiungere al messaggio M indicato con q:

$$q = (r/8) - (m \ mod \ (r/8)).$$

Il valore di q determina la forma esadecimale di questi byte secondo la tabella 3.4:

Numero di byte del padding	**Padding finale del messaggio**
$\boldsymbol{q = 1}$	$M\|\|0x86$
$\boldsymbol{q = 2}$	$M\|\|0x0680$
$\boldsymbol{q > 2}$	$M\|\|0x06\|\|0x00\ldots\|\|0x80$

Tabella 3.4: *Forma esadecimale del padding per lo SHA-3*

3.4 Risorse utilizzate

Per le implementazioni software e hardware dello SHA3-512, e poi per quelle dell'HMAC, la scheda utilizzata è la scheda di sviluppo *Opal Kelly ZEM5310* dotata di un *Altera Cyclone V E FPGA (5CEFA4F23)*, con frequenza del clock pari a 100 MHz [9].
Per ottenere misure accurate sul tempo di esecuzione delle varie implementazioni, gli ingressi forniti allo SHA-3 e al blocco HMAC sono costituiti da un numero di byte che spazia da un minimo di 64 byte a un massimo di 9000 byte. Tutti questi ingressi sono valori randomici, la cui casualità deriva dal rumore atmosferico, generati grazie all'ausilio del sito [11].

3.5 Implementazione software

Nell'implementazione software è stato eseguito l'algoritmo SHA3-512 direttamente sul processore *NIOS II* dell'FPGA. Il codice C utilizzato in questa sezione reperibile al sito [6], è stato sviluppato dagli stessi creatori del Keccak. Per comprendere meglio il suo funzionamento viene riportato lo pseudocodice dell'algoritmo:

```
int blockSize

// Fase di assorbimento

while(inputByteLen > 0) {
        blockSize = MIN(inputByteLen, rate/8);
        for i in 0 to blockSize
            state[i] ^= input[i];
        inputByteLen -= blockSize;

        if (blockSize == rate/8) {
            Keccak-f[1600](state);
            blockSize = 0;
        }
    }


// Applicazione della regola di padding

state[blockSize] ^= delimitedSuffix;
state[rateInBytes-1] ^= 0x80;

// Fase di permutazione

Keccak-f[1600](state) {
  for i in 0 to 23
    state = Round[1600](state, RC[i])
  return state
}

Round[1600](state,RC) {

// Theta step

for x in 0 to 4
 C[x] = state[x,0] xor state[x,1] xor state[x,2]
        xor state[x,3] xor state[x,4]

 D[x] = C[x-1] xor rot(C[x+1],1)

for (x,y) in (0 to 4,0 to 4)
 state[x,y] = state[x,y] xor D[x],
```

```
 // Rho and Pi steps

 for (x,y) in (0 to 4,0 to 4)
B[y,2*x+3*y] = rot(state[x,y], r[x,y]),

 // Chi step

for (x,y) in (0 to 4,0 to 4)
state[x,y] = state[x,y] xor ((not B[x+1,y]) and B[x+2,y]),

// Iota step

 state[0,0] = state[0,0] xor RC

 return state
}

// Fase di rilascio

while(outputByteLen > 0) {
        blockSize = MIN(outputByteLen, rate/8);
        memcpy(output, state, blockSize);

        output += blockSize;

        outputByteLen -= blockSize;

        if (outputByteLen > 0)
            Keccak-f[1600](state);
    }
```

Il codice reperito in rete è costituito da due funzioni:

1. void *FIPS202_SHA3_512*(*const unsigned char *input*, *unsigned int inputByteLen*, *unsigned char *output*);
2. Keccak(576, 1024, input, inputByteLen, 0x06, output, 64).

La funzione 1 ha il solo obiettivo di ricevere i parametri in ingresso e di restituire alla fine il risultato delle operazioni. I parametri ricevuti sono:

a. const unsigned char *input;

b. unsigned int inputByteLen;

c. unsigned char *output.

Il parametro *a* è un puntatore ad un array di *unsigned char* (8 bit, senza segno) che contiene il messaggio da elaborare di qualsiasi lunghezza. Il parametro *b* è un intero senza segno e identifica il numero di byte dell'ingresso, variabile importante richiesta dall'algoritmo. Infine, anche il terzo parametro *c* è un puntatore ad una stringa che conterrà il risultato finale dell'elaborazione. In questo caso la sua lunghezza è fissa ed equivale a 64 byte.
La funzione 2 viene richiamata dalla prima ed è la funzione chiave dell'algoritmo, poiché esegue le tre fasi che compongono lo SHA-3, la fase di *assorbimento*, di permutazione dello *state* richiamando a sua volta la funzione *KeccakF1600_StatePermute(void *state)* e infine la fase di rilascio dell'uscita.
I parametri che la funzione 2 riceve sono:

a. il *rate* dello SHA-3 uguale a 576;

b. la capacità uguale a 1024;

c. il messaggio in ingresso;

d. il numero di byte dell'ingresso;

e. il *delimited Suffix Bits* da aggiungere all'uscita del messaggio in ingresso;

f. la stringa di uscita;

g. il numero di byte dell'uscita.

A questo codice nel *main* è stato aggiunto il richiamo alla funzione 1 e la dichiarazione degli ingressi. Inoltre, in un primo momento è stata utilizzata anche una funzione che riceve l'uscita prodotta dallo SHA-3 e una stringa contenente il risultato corretto, ottenuto dal sito [14] che è un calcolatore di SHA-3 online, per confrontare le due stringhe e verificare la correttezza dell'esecuzione.
Successivamente è stato eseguito il codice descritto in precedenza sul processore NIOS II utilizzando i seguenti tools:

1. *Quartus Prime 18.1;*
2. *Platform Designer*, tool di *Quartus* che permette di integrare i componenti IP all'interno di una rappresentazione grafica del sistema;
3. *Nios II Software Build Tools for Eclipse*, usato per comunicare con il processore e per effettuare le simulazioni.

Dopo aver creato un nuovo progetto in *Quartus* è stato costruito il sistema nel *Platform Designer* come mostrato in figura 3.4:

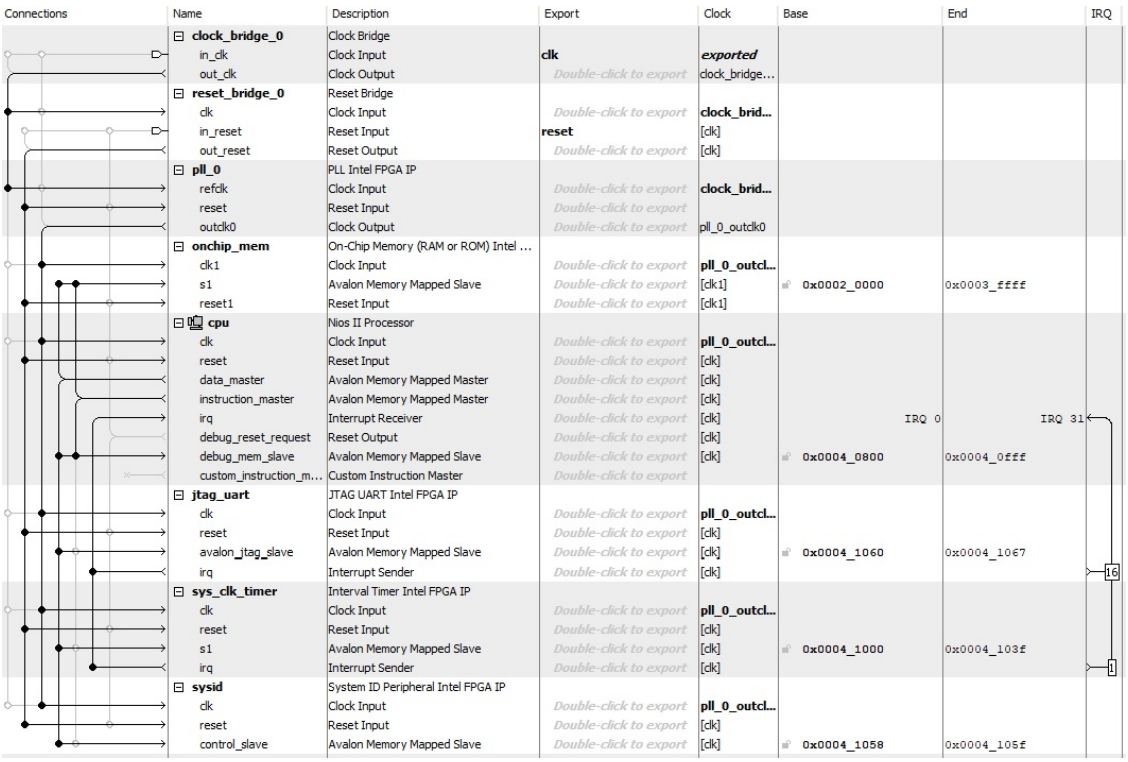

Figura 3.4: *Qsys per lo SHA3-512 versione software*

All'interno del sistema rappresentato si hanno differenti blocchi:

1. *clock e reset bridge*, per ricevere il clock e il reset dall'esterno e distribuirlo ai vari componenti;
2. *PLL*, per incrementare la frequenza del sistema a 140 MHz;
3. memoria RAM da 2^{17} byte;
4. microprocessore (NIOS II) di tipo f;
5. jtag, timer e *System ID*.

Per il microprocessore 4 vanno testati due parametri importanti che permettono l'incremento delle prestazioni, l'*Instrunction set size* settato a 4 Kbyte, e la *Data cache size* impostata a 32 Kbyte. Dopo aver completato tutte le connessioni nel sistema, con il comando *Generate*, il *Platform Designer* crea il file *.sopcinfo*, un file XML con codifica *ASCII* che contiene in maniera sintetica tutte le informazioni sul sistema e sui parametri impostati. Per la parte di simulazione è stato utilizzato *Eclipse*, che permette di compilare e poi eseguire il codice C. Come in *Quartus*, anche in *Eclipse* è stato sviluppato un nuovo progetto costituito da due parti principali: la prima parte utilizza il file *.sopcinfo* generato precedentemente per creare il *BSP (Board Support Package)*, cioè un codice di supporto che permette l'utilizzo delle periferiche e del processore della scheda FPGA, mentre la seconda si serve del *BSP* per creare una nuova applicazione, che si avvale del file C che deve essere eseguito e di altri file necessari per la compilazione. Dopo aver creato il *BSP* è possibile modificare alcuni suoi parametri e nel caso specifico utilizzando il *BSP editor* è stato impostato alla sezione *timestamp_timer* il timer inserito nel *Qsys* in modo da poter misurare dei tempi di esecuzione.
È possibile variare alcuni parametri per impostare il livello di ottimizzazione che il compilatore può adottare: il *debug level* e l' *optimization level*. Il primo è stato impostato su *OFF*, perché in questo caso non si ha la necessità di effettuare il debug, e per la sola esecuzione del codice quel parametro non è utile. Il secondo parametro stabilisce quale livello di ottimizzazione il compilatore può adottare e nel caso analizzato può scegliere o l'ottimizzazione per ridurre le dimensioni occupate in memoria per il codice oppure l'ottimizzazione di livello 3, che permette il massimo delle prestazioni durante l'esecuzione.
Il codice C presente nell'applicazione creata contiene dei vettori di test, che definiscono l'uscita corretta per quel dato messaggio in ingresso computata con l'aiuto del calcolatore online dello SHA3-512 [14]. Dopo averla stampata a video nella *console* di *Eclipse*, viene comparata con il valore del vettore test e se coincide viene stampato un messaggio per segnalare la correttezza del test.
Per calcolare le dimensioni occupate dal codice in termini di kbyte, è stato sufficiente analizzare le informazioni all'interno della *console* di *Eclipse* durante la compilazione. Per misurare i tempi di esecuzione, all'interno del codice C è stata aggiunta la libreria *alt_timestamp.h* che contiene tutte le funzioni utili per poter gestire il timer presente sulla scheda FPGA. In particolare, le due funzioni utilizzate sono:

1. *void alt_timestamp_start()*;

2. *int alt_timestamp()*.

La funzione 1 ha l'obiettivo di resettare il timer al valor iniziale, mentre la funzione 2 ritorna il numero di *clock ticks*. Il tempo di esecuzione è stato misurato da quando viene richiamata la funzione *Keccak(576, 1024, input, inputByteLen, 0x06,*

output, 64) a quando la funzione termina per poi ritornare nel *main*. Quindi, sono state definite due variabili chiamate *start_time* e *end_time*. La variabile *start_time* indica il numero di *clock ticks* intercorsi dal reset del contatore fino al richiamo della funzione, mentre la variabile *end_time* specifica il numero di *clock ticks* trascorsi da *start_time* fino al termine delle operazioni della funzione *Keccak*. Per calcolare il tempo in *ms*, è stata applicata la seguente operazione:

$$Tempo = \frac{end\_time - start\_time}{frequency} \cdot 1000.$$

Il parametro *frequency* presente al denominatore indica la frequenza di clock utilizzata nel sistema che per l'implementazione software è impostata a 140 MHz.
Un esempio del messaggio visualizzato nella *console di Eclipse* è mostrato in figura 3.5:

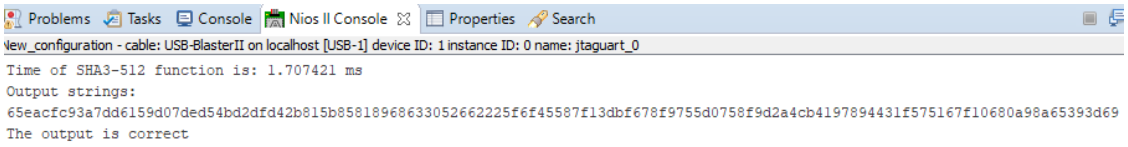

Figura 3.5: *Messaggio stampato nella console di Eclipse per lo SHA3-512 versione SW*

Bisogna precisare che all'interno del periodo temporale analizzato, non sono presenti funzioni *printf*, infatti, per esempio, la stampa del risultato non viene conteggiata. Questo è fondamentale poiché in generale, le funzioni di stampa richiamano una serie di procedure che porterebbero ad incrementare inutilmente il tempo misurato.

3.5.1 Risultati ottenuti

Il primo risultato ottenuto riguarda la dimensione del codice della sola funzione SHA-3, escludendo quindi tutta l'interfaccia esterna. I valori sono indicati nella tabella 3.5:

	Dimensione del codice (kB)
Ottimizzato per prestazioni	22
Ottimizzato per dimensioni	17

Tabella 3.5: *Dimensione del codice in kB per la funzione SHA3-512*

Analizzando la tabella, è possibile constatare che l'ottimizzazione per dimensione permette una riduzione di circa il 23% del numero di byte occupati dal codice. Nel

caso in questione la riduzione non è significativa, poiché lo spazio occupato è di molto inferiore alle dimensioni della memoria, ma in generale per codici più onerosi questa riduzione potrebbe essere significativa.
Per quanto riguarda l'analisi del tempo di esecuzione, i risultati ottenuti sono sintetizzati nella tabella 3.6:

		DIMENSIONI INPUT (B)										
		64	128	256	512	1024	1400	2048	4096	5028	8192	9000
Tempo di esecuzione (ms)	Ottimizzato per prestazioni	1.71	3.40	6.80	13.57	25.44	33.91	49.17	96.63	118.66	193.60	214.17
	Ottimizzato per dimensioni	3.15	6.30	12.60	25.17	47.19	62.93	91.24	179.33	220.23	358.85	396.40

Tabella 3.6: *Tempi di esecuzione in versione software per lo SHA3-512*

La tabella ci mostra come i tempi di esecuzione con un'ottimizzazione per prestazioni siano minori rispetto al caso di ottimizzazione per dimensioni. La riduzione è circa del 50%, in linea con le aspettative. Per una migliore visualizzazione, di seguito viene riportato un grafico con la comparazione dei due risultati ottenuti 3.6:

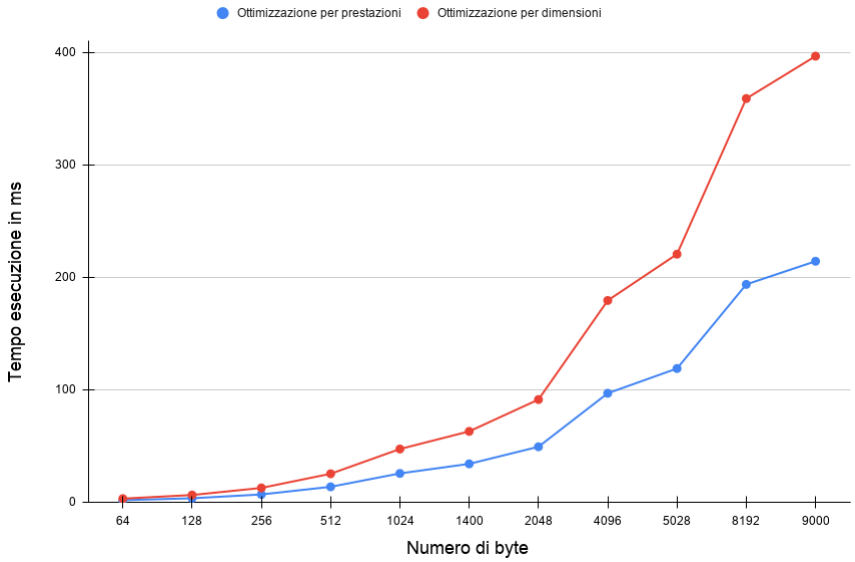

Figura 3.6: *Confronto tra i tempi di esecuzione dello SHA3-512 per le due differenti ottimizzazioni*

I differenti numeri di byte degli ingressi scelti per le varie misurazioni sono valori tipici utilizzati nelle trasmissioni di dati. Una considerazione che si può trarre analizzando le diverse misurazioni, è che il tempo di esecuzione in corrispondenza di un ingresso pari a 64 byte è il tempo minimo che la funzione SHA-3 impiega per il

calcolo del risultato. Infatti, durante le differenti misurazioni, sono stati utilizzati anche numeri di byte inferiori a 64, ma il tempo ricavato è identico.
Infine, si può concludere che è più vantaggioso utilizzare l'ottimizzazione per prestazioni quando l'interesse riguarda il tempo di esecuzione e per questo motivo successivamente, nel confronto tra i risultati dell'hardware, saranno usati solo i valori rappresentati nella prima riga della tabella 3.6. Per l'analisi delle tempistiche come detto in precedenza la frequenza di clock è pari a 140 MHz. Questo valore è stato adottato in seguito alla *Timing analyzer* realizzata con *Quartus*, che ha permesso lo studio del percorso critico e di trovare la massima frequenza con cui il sistema può lavorare.

3.6 Implementazione hardware

Per l'implementazione hardware è stato adattato e modificato il codice VHDL realizzato da *Hsing e Homer*, trovato in rete [8]. Il blocco SHA3-512 è composto principalmente da due blocchi chiamati *Padder* e *F_permutation* e da una *Top_entity* che unisce tramite il classico *port map* questi due componenti e inoltre permette la fase di rilascio riordinando semplicemente i byte dell'uscita.

3.6.1 Top_entity

Gli ingressi ricevuti dal blocco SHA3-512 in VHDL sono:

1. *clock e reset*;
2. *input* su 32 bit;
3. *in_ready e is_last*;
4. *byte_num* su 2 bit.

I segnali *in_ready e is_last* sono due segnali di controllo che il blocco riceve dall'esterno, il primo indica che l'ingresso è valido e quindi deve essere acquisito, il secondo ha lo scopo di indicare che l'ingresso appena ricevuto che compone il messaggio da elaborare è l'ultimo. Il segnale *byte_num* invece, è indispensabile per applicare il *padding*, è su due bit e rappresenta il numero di byte che compongono l'ultimo ingresso. Dato che il numero di bit dell'ingresso è 32, il valore di *byte_num* può spaziare tra 0 e 3. Quando l'ultimo dato è composto da due soli byte, AABB per esempio, *byte_num* sarà "10". Il valore di questo segnale sarà utilizzato solo in presenza dell'ultimo dato cioè quando il segnale *is_last* varrà 1. Per quanto riguarda l'ingresso, il segnale *input* è di 32 bit, mentre nel blocco originario trovato in rete l'ingresso era su 64 bit. La riduzione del numero di bit è stata necessaria perché il processore NIOS II può inviare o ricevere solo dati a 32 bit. Il blocco hardware così come quello software è in grado di elaborare messaggi di qualsiasi lunghezza, non solo con un numero pari di byte ma anche dispari.
Le uscite del blocco SHA-3, invece, sono:

1. *output* su 512 bit;
2. *out_ready*;
3. *buffer_full*.

Il segnale *out_ready* è un segnale di controllo che indica quando l'uscita del blocco SHA-3 è valida, mentre *buffer_full* deriva dal blocco *Padder* ed indica quando il buffer in uscita allo stesso blocco è pieno. Il segnale *output* rappresenta il risultato finale su 512 bit per definizione dello stesso SHA3-512 e rimane valido fino a quando non viene resettato il blocco per poter ricevere nuovi dati.

L'uscita viene calcolata troncando l'*output* del blocco *f_permutation*, e utilizzando due cicli *for* annidati, i byte vengono riordinati:

```
for w in 0 to 7 generate
   for b in 0 to 7 generate

        output(((((w)*64 + (b)*8))+7) downto ((w)*64 + (b)*8) ) <=
    f_out((((w)*64 + (7-b)*8)+7) downto ((w)*64 + (7-b)*8));

end generate;
end generate;
```

Il blocco originario [8] era in grado di eseguire due permutazioni nello stesso instante di clock, di conseguenza, poiché il numero di permutazioni per questo algoritmo è 24, il blocco impiegava solo 12 colpi di clock per ogni permutazione. Lo studio del percorso critico ha evidenziato come la parte della permutazione avesse una forte influenza sulla frequenza perché una frequenza di clock pari a 100 MHz risultava troppo elevata per ottenere risultati corretti. È stato necessario scegliere se utilizzare il blocco originario ma con frequenza inferiore rispetto al resto del sistema oppure eliminare dei blocchi logici dal percorso critico per avere lo stesso regime di clock in tutto il sistema. Alla fine ha prevalso la seconda alternativa ed è stata eliminata la possibilità di eseguire due permutazioni nello stesso colpo di clock, aumentando di conseguenza i colpi totali per ogni permutazione a 24, ma consentendo al sistema di lavorare a 100 MHz.

3.6.2 Padder

Lo schema a blocchi del *Padder* è mostrato nella figura 3.7:

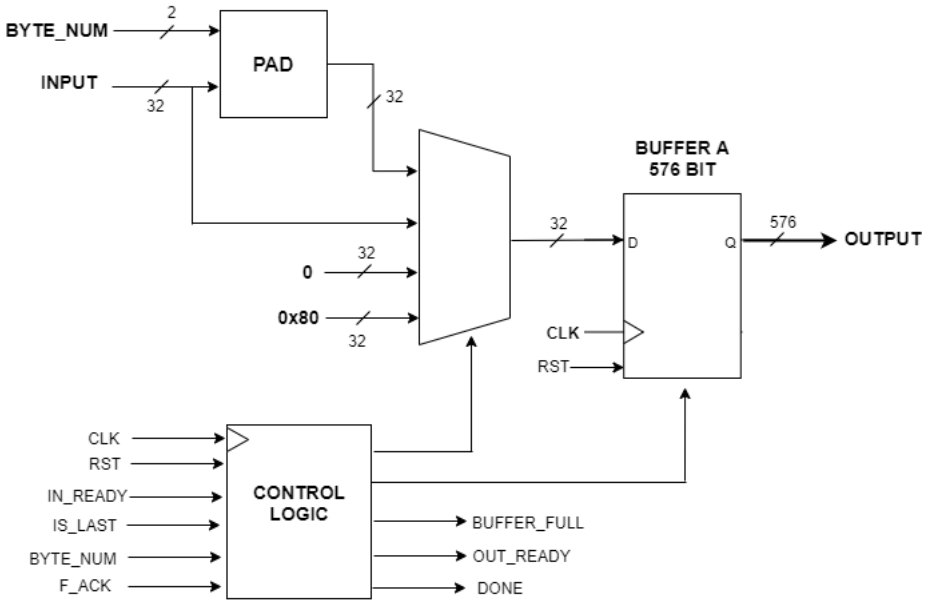

Figura 3.7: *Schema a blocchi del Padder*

Il *Padder* è costituito all'ingresso da un multiplexer pilotato da una logica di controllo. Per la maggior parte del tempo il *MUX* preleva l'ingresso dall'esterno e lo carica all'interno del buffer. Il buffer ad ogni colpo di clock esegue una traslazione dei dati al suo interno per accumularne uno nuovo quando il segnale proveniente dalla logica di controllo ha valore 1. In questo modo, viene acquisito ogni dato valido all'ingresso in modo da poterlo poi processare. La logica di controllo al suo interno ha un contatore che permette di conteggiare il numero di dati che il buffer ha accumulato e, quando il valore del contatore è pari a 16, il segnale di *buffer_full* che indica che il buffer è pieno, varrà 1. In questo caso il blocco non può accettare nuovi dati, quindi il segnale *in_ready* dovrà essere zero fino a quando il buffer non sarà svuotato per poi riprendere l'eventuale acquisizione dei dati. Quando l'uscita del buffer è valida, il segnale uscente dalla logica di controllo *out_ready* ha valore 1 e di conseguenza il blocco di permutazione riceverà i 576 bit e inizierà nuovamente le sue operazioni.
Un'altra uscita del blocco è il segnale *done*, che indica qual è l'ultima acquisizione che il *Padder* ha effettuato e viene utilizzato nella *top_entity* per la generazione del segnale *out_ready*.
Il segnale *f_ack* in ingresso alla logica di controllo proviene dal blocco di permutazione per indicare che il *Padder* può acquisire nuovi ingressi mentre la permutazione è in corso. Questa struttura permette di ottenere un alto *throughput* poiché il blocco

Padder non deve aspettare che la permutazione sia completata per acquisire nuovi ingressi, ma può prepararli durante il lavoro della *F_permutation*.
I restanti ingressi del multiplexer sono utilizzati quando il segnale di *is_last* è 1, ovvero quando è stato ricevuto l'ultimo *input* valido del messaggio, e precisamente l'uscita del blocco *PAD*, un blocco combinatorio che riceve il segnale *byte_num*, e l'ingresso per applicare il *padding* come mostrato nella tabella 3.7:

Byte_num	**Output**
00	X"06000000"
01	*input*(31 *downto* 24) & X"060000"
10	*input*(31 *downto* 16) & X"0600"
11	*input*(31 *downto* 8) & X"06"

Tabella 3.7: *Blocco PAD*

E' importante fare una precisazione sul primo caso della tabella. Come detto in precedenza, il segnale *byte_num* indica il numero di byte che compongono l'ultimo dato del messaggio. Nel caso in cui questo segnale vale "00", il dato in ingresso è completo, cioè composto da tutti e quattro byte, quindi il padding va applicato a partire dal colpo di clock successivo. Nel caso analizzato, il segnale *is_last* dovrà essere asserito al colpo di clock successivo rispetto a quello in cui il dato è stato inviato, mentre nei rimanenti tre casi il segnale dovrà essere asserito subito.
Gli altri due ingressi del multiplexer vengono utilizzati dopo l'uscita del *PAD*, poiché come stabilito dalla regola di pad, bisogna riempire di zeri i rimanenti bit e infine gli ultimi bit saranno costituiti dal valore esadecimale *80*.

3.6.3 F_permutation

F_permutation ha uno schema a blocchi rappresentato nella figura 3.8:

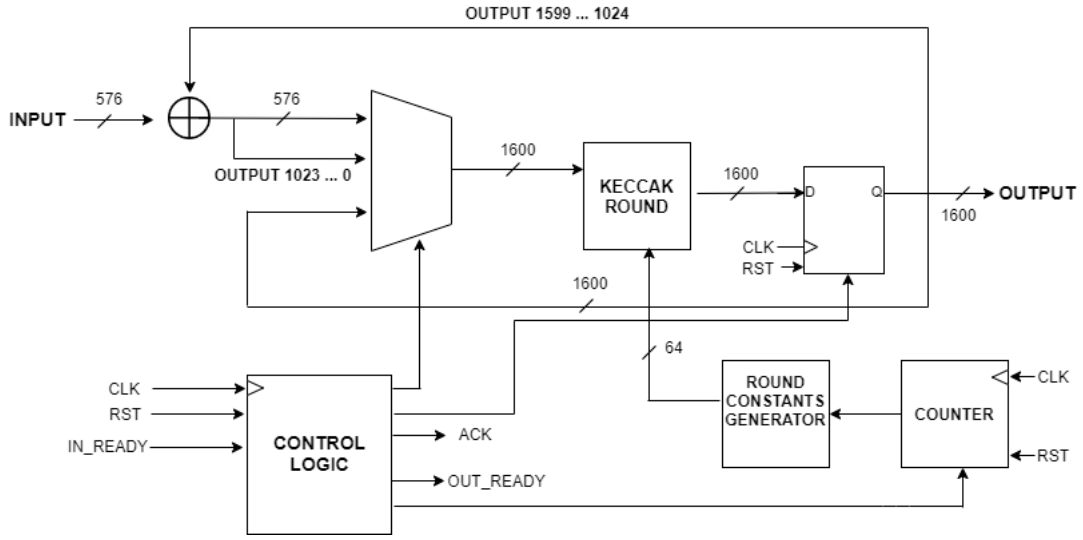

Figura 3.8: *Schema a blocchi della F_permutation*

Il blocco di permutazione riceve l'ingresso dal *Padder* ed esegue lo XOR bit a bit con i primi 576 bit dell'uscita precedente. Il risultato di questa operazione viene esteso con gli ultimi 1024 bit dell'uscita e diventa il primo ingresso del multiplexer, mentre il secondo è costituito direttamente dall'uscita. Il multiplexer è gestito da una logica di controllo. Il segnale che pilota il MUX durante la prima permutazione farà passare il segnale di XOR, mentre per tutte le altre 23 permutazioni rimanenti seleziona l'*output* precedente. L'uscita del multiplexer diventa l'ingresso del blocco, chiamato *Keccak round*, che è di 1600 bit come richiesto dalla permutazione *Keccak-f*[*b*]. Il *Keccak round* è combinatorio e ha il compito di eseguire i cinque *state mapping* come descritto in 3.2. Per calcolare anche lo *state mapping iota*, riceve dal blocco indicato come *Round constants generator* le 24 costanti di *round*, indicate in tabella 3.3, che sono generate di volta in volta tramite un contatore gestito dalla logica di controllo. In altre architetture dello SHA-3, che si possono visionare su internet, le costanti sono memorizzate, per esempio, in memorie RAM, ma un'analisi dettagliata ha evidenziato che i loro valori contengono molti zeri che sprecherebbero inutilmente spazi per cui sarebbe inutile sprecare locazioni di memoria per salvarli.
Infine, all'uscita della *F_permutation* è presente un registro che memorizza l'uscita del *Keccak round* pilotato dalla logica di controllo.
Gli ingressi del blocco di permutazione sono oltre al clock e al reset, l'input su 576 bit e il segnale di *in_ready*, che proviene dal *Padder* per indicare che l'uscita è

valida e la permutazione può iniziare ad operare. La logica di controllo in uscita genera il segnale di *ack* che sarà poi l'ingresso del *Padder* per indicare l'inizio delle operazioni e il segnale *out_ready* che permette la generazione del segnale finale *out_ready* nella *Top_entity*.

3.6.4 Simulazione iniziale dell'architettura

Per testare il corretto funzionamento del blocco SHA3-512 e per approfondire il timing utile poi per l'interfacciamento con il processore NIOS II, è stato creato un testbench per fornire gli ingressi e i comandi. Dopo averlo compilato è stato utilizzato *Modelsim* per effettuare la simulazione e alcuni risultati sono mostrati nelle figure 3.9 e 3.10:

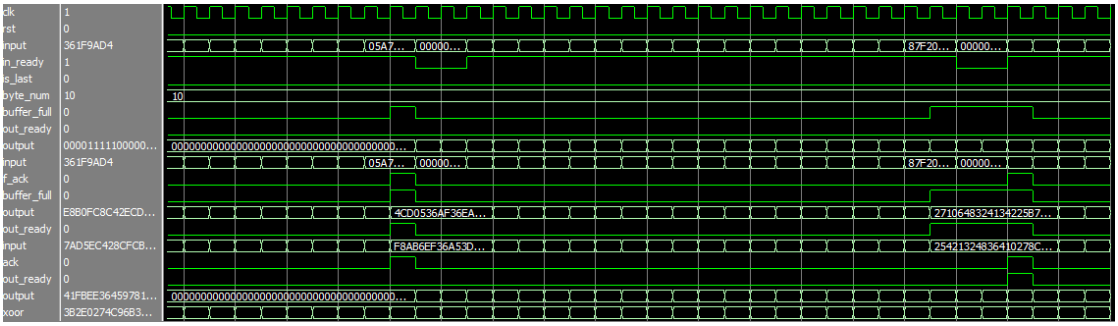

Figura 3.9: *Simulazione per la parte di acquisizione dati del blocco SHA3-512*

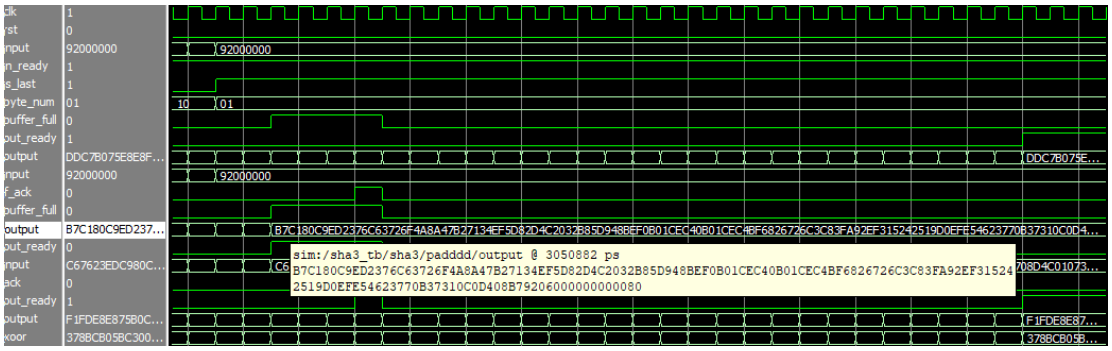

Figura 3.10: *Simulazione per la parte di generazione del risultato del blocco SHA3-512*

Nella prima figura è evidenziata la parte di acquisizione dati in cui ad ogni colpo di clock viene inviato un nuovo dato e il segnale *in_ready* è 1 per ogni dato valido. Il blocco *Padder* trasla i valori nel buffer e quando l'uscita è completa il segnale *out_ready* andrà a 1 e la permutazione comincia ad elaborare l'ingresso

ricevuto. La parte destra dell'immagine evidenzia che il buffer è pieno, il segnale *buffer_full* vale 1, e di conseguenza un nuovo ingresso non può essere prelevato un nuovo ingresso. Quindi, il segnale *in_ready* vale 0 e il dato presente in ingresso, nell'esempio il dato con 32 bit a zero, non viene acquisito fino a quando il buffer non si è svuotato.
La seconda immagine, invece, evidenzia la parte finale del processo di elaborazione del messaggio. Nell'istante in cui viene ricevuto l'ultimo dato, costituito nell'esempio da due soli byte, il segnale *is_last* viene asserito e dunque viene applicato il *padding*, come si vede nel riquadro bianco. Nell'esempio il segnale deve essere subito portato a 1 poiché, come descritto nel *Padder*, il valore di *byte_num* è diverso da "00". Dopo aver completato l'ultimo ciclo di permutazioni il segnale di *out_ready* va a 1 e il risultato è visualizzato all'uscita.

3.6.5 Interfacciamento con il processore NIOS II

Dopo una prima simulazione dell'architettura, è stato necessario studiare una possibile soluzione per interfacciare il blocco con il processore NIOS II presente sull'FPGA. Le informazioni scambiate tra il processore e il blocco hardware precedentemente descritto sono i dati, sia di entrata e sia di uscita, e i diversi segnali di controllo.
Per collegare il NIOS e l'hardware è possibile utilizzare le interfacce *Avalon* che semplificano la progettazione del sistema consentendo di collegare facilmente i componenti nell'FPGA [2]. La famiglia di interfacce Avalon include le interfacce appropriate per lo streaming di dati ad alta velocità, la lettura e la scrittura di registri e memorie e il controllo di dispositivi off-chip. Molti componenti disponibili in Platform Designer incorporano queste interfacce standard ma è possibile annetterle anche in componenti personalizzati. La famiglia Avalon è costituita da differenti interfacce tra le quali:

1. *Avalon Streaming Interface (Avalon-ST)*, che permette il flusso unidirezionale di dati con un'elevata frequenza;
2. *Avalon Memory Mapped Interface (Avalon-MM)*, che utilizza gli indirizzi per leggere e scrivere permettendo la comunicazione tra *master e slave*;
3. *Avalon Conduit Interface*;
4. *Avalon Tri-State Conduit Interface (Avalon-TC)*, per le connessioni con le periferiche esterne;
5. *Avalon Interrupt Interface*;
6. *Avalon Clock Interface* per pilotare o ricevere il clock;
7. *Avalon Reset Interface* per le connessioni del reset.

Oltre alle interfacce *Avalon Clock Interface* e *Avalon Reset Interface* per la gestione del clock e del reset, è stata utilizzata, per gli ingressi da elaborare, l'interfaccia di tipo *Avalon-ST*, per inviare un grande numero di vettori a 32 bit con una frequenza elevata, mentre per leggere il risultato ottenuto, i cui 512 bit sono determinati tutti insieme e quindi devono essere salvati in 16 registri, e per trasmettere dei segnali di controllo è stata adottata l'interfaccia di tipo *Avalon-MM*. Per trasmettere il risultato al processore non è necessaria un'elevata frequenza e di conseguenza sarebbe stato inutile l'uso dello streaming.
L'interfaccia *Avalon-ST* non richiede un indirizzo per la comunicazione contrariamente al processore NIOS II che è in grado di comunicare solo tramite il protocollo *Avalon-MM*, che invece utilizza indirizzi per leggere e scrivere. Per permettere una corretta comunicazione tra il processore NIOS II e il blocco hardware è stata interposta una memoria *FIFO single-clock* per trasformare una comunicazione di tipo *Avalon-MM* in una di tipo *Avalon-ST*, accumulando i dati per poi trasmetterli all'hardware.

3.6.5.1 Avalon Memory Mapped Interface

L'*Avalon-MM* è caratterizzata da un gran numero di segnali in particolare sono stati utilizzati i seguenti:

1. *read*, che viene asserito dal processore quando deve leggere il dato;
2. *readdata*, pilotato dallo *slave* in risposta al segnale di *read*;
3. *write*, che viene asserito dal processore quando deve inviare un dato;
4. *writedata*, il dato da scrivere;
5. *waitrequest*, un segnale asserito dallo *slave* per indicare che non può leggere o scrivere.

Il segnale *readdata* è un vettore a 32 bit utilizzato per trasmettere al processore il risultato finale prodotto dallo SHA-3 e sono necessari 16 cicli di lettura in quanto il valore finale è costituito da 512 bit. Questa comunicazione è controllata dai due segnali *read* e *waitrequest*. Il primo viene governato dal processore, che quando il valore del *read* è 1, acquisisce il dato presente in *readdata*. Il secondo è gestito dal blocco hardware che segnala 1 se il risultato non è ancora disponibile mentre indica 0 se il dato è pronto per essere letto dal processore.
Il *writedata* è un vettore a 32 bit, utilizzato per inviare al blocco dei segnali di controllo, mentre il *write* è inviato dal processore quando deve scrivere. Il segnale di *writedata* è stato così suddiviso:

1. *writedata(31)*, utilizzato come segnale di start;

2. *writedata(30 ... 0)*, indica il numero totale di byte che compongono il messaggio.

Il segnale start viene utilizzato per dare il via alle operazioni del blocco SHA-3, mentre i restanti bit vengono utilizzati per conoscere il numero totale di byte che il processore intende inviare. Questa informazione equivale alla variabile *inputByteLen* del codice C ed è stata utilizzata per individuare l'ultimo ingresso del messaggio per poter asserire il segnale *is_last* e di conseguenza applicare il *padding*. Anche per la scrittura viene utilizzato il segnale *waitrequest*, nella stessa modalità della lettura. La figura 3.11 mostra il *timing* dei segnali precedentemente descritti:

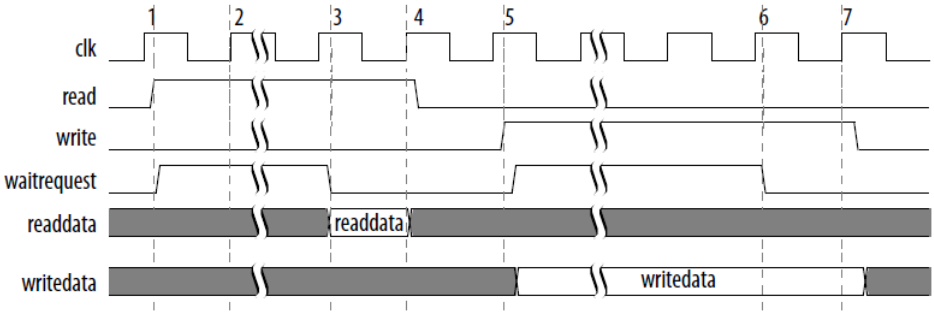

Figura 3.11: *Timing dei segnali dell'interfaccia Avalon-MM [2]*

3.6.5.2 Avalon Streaming Interface

L'*Avalon-ST* è stata utilizzata per gli ingressi e in particolare i segnali:

1. *valid*, inviato dalla FIFO per indicare un dato pronto da inviare;
2. *data*, vettore a 32 bit che contiene il dato;
3. *ready*, inviato dal blocco per segnalare di essere pronto per ricevere un nuovo dato.

L'uso del segnale *ready* permette di sfruttare il concetto di *backpressure* che permette al ricevente di segnalare alla sorgente di non poter più ricevere dati e dunque di interrompere la trasmissione. Quando il *backpressure* è attivo, il ricevente asserisce il segnale *ready* per un solo colpo di clock per indicare che è pronto a ricevere un dato. A questo punto inizia il cosiddetto *ready cycle*. Durante questo ciclo la sorgente può asserire il *valid* e inviare un dato se disponibile, altrimenti riporta il valid a 0. Il parametro che scandisce il timing tra questi segnali viene indicato come *readyLatency*, che può assumere un valore tra 0 e 8. Nel caso esaminato è stato impostato a 1, per indicare che il segnale *valid* può essere asserito almeno un colpo di clock dopo l'asserzione del *ready*.
La figura 3.12 mostra il *timing* dei segnali precedentemente descritti:

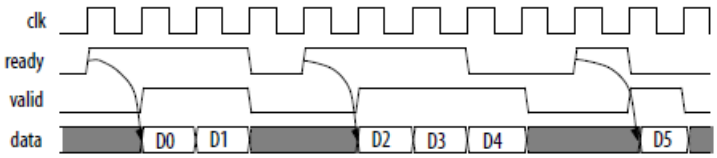

Figura 3.12: *Timing dei segnali dell'interfaccia Avalon-ST [2]*

Analizzando la figura si nota come quando il *ready* viene asserito la sorgente risponde uno o due cicli dopo fornendo l'asserzione del *valid* e il dato. La sorgente, inoltre deve portare il valid a 0 su un ciclo di *non-ready*.

3.6.5.3 FSM

Per associare il blocco SHA-3 con le due interfacce precedentemente descritte, è stata creata, sempre in VHDL, una *top_entity* che possiede come ingressi e uscite i segnali dell'*Avalon-MM* e dell'*Avalon-ST* e mette insieme lo SHA-3 con una macchina a stati realizzata appositamente per la gestione dei diversi controlli e dati. La macchina a stati è mostrata nella figura 3.13:

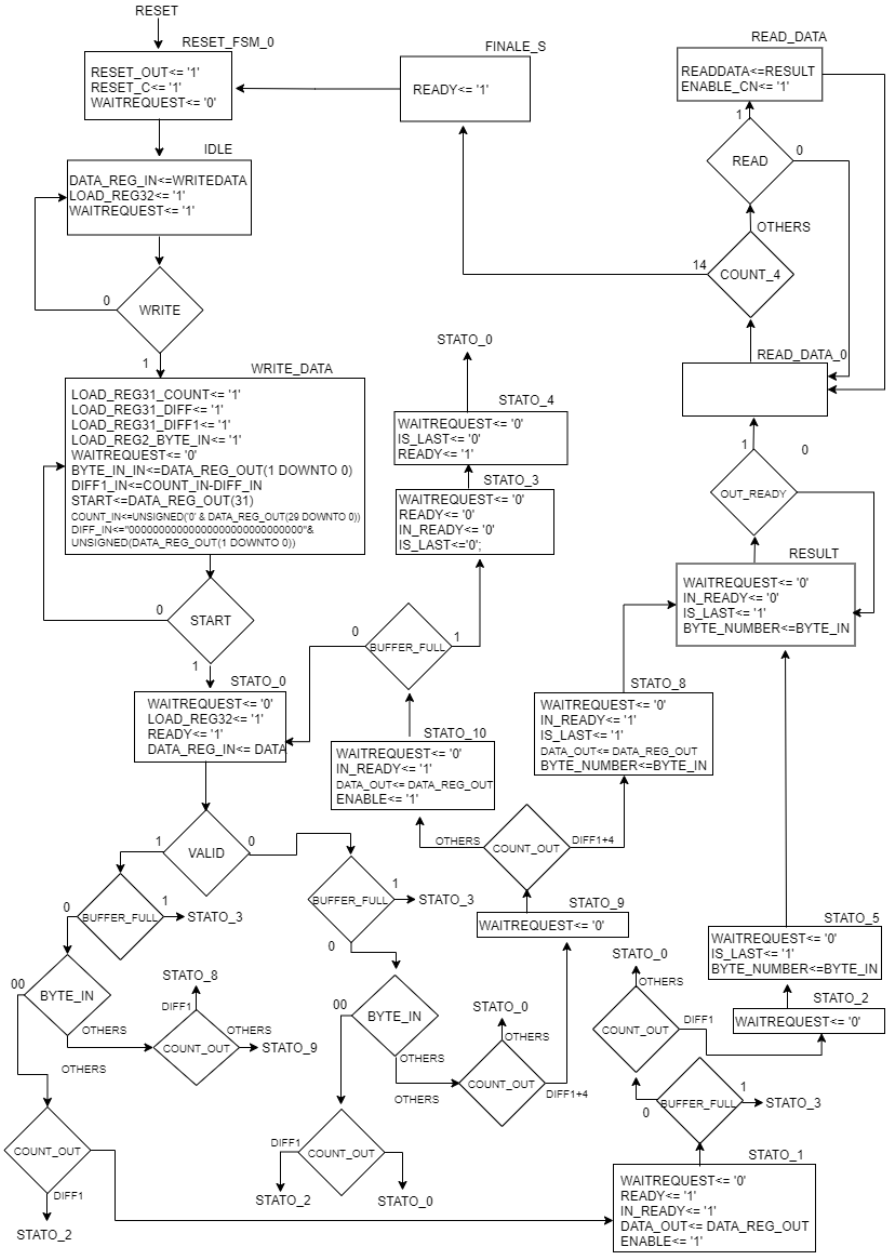

Figura 3.13: *FSM dello SHA3-512*

All'interno della FSM, è stato utilizzato un contatore il cui valore viene confrontato con la variabile che contiene il numero totale di byte del messaggio di ingresso, per poter poi applicare il corretto *padding* come già ampiamente descritto. Altri segnali di controllo utilizzati per la transazione tra i differenti stati sono *write*, *read* e *valid* che derivano dalle interfacce esterne, *byte_in* e *buffer_full* quando il buffer è pieno e non si può ricevere un nuovo dato e quindi il segnale *ready* deve essere riportato a 0, e infine il segnale *out_ready* per segnalare che il risultato finale è disponibile.

3.6.5.4 Sintesi del sistema

Per integrare il componente hardware con i restanti componenti IP è stato riutilizzato nuovamente *Quartus* e il suo tool *Platform Designer*. Gli step seguiti in questa fase sono in generale gli stessi della versione software anche se ci sono delle differenze.
Il sistema costruito nel *Platform Designer* è mostrato nella figura 3.14:

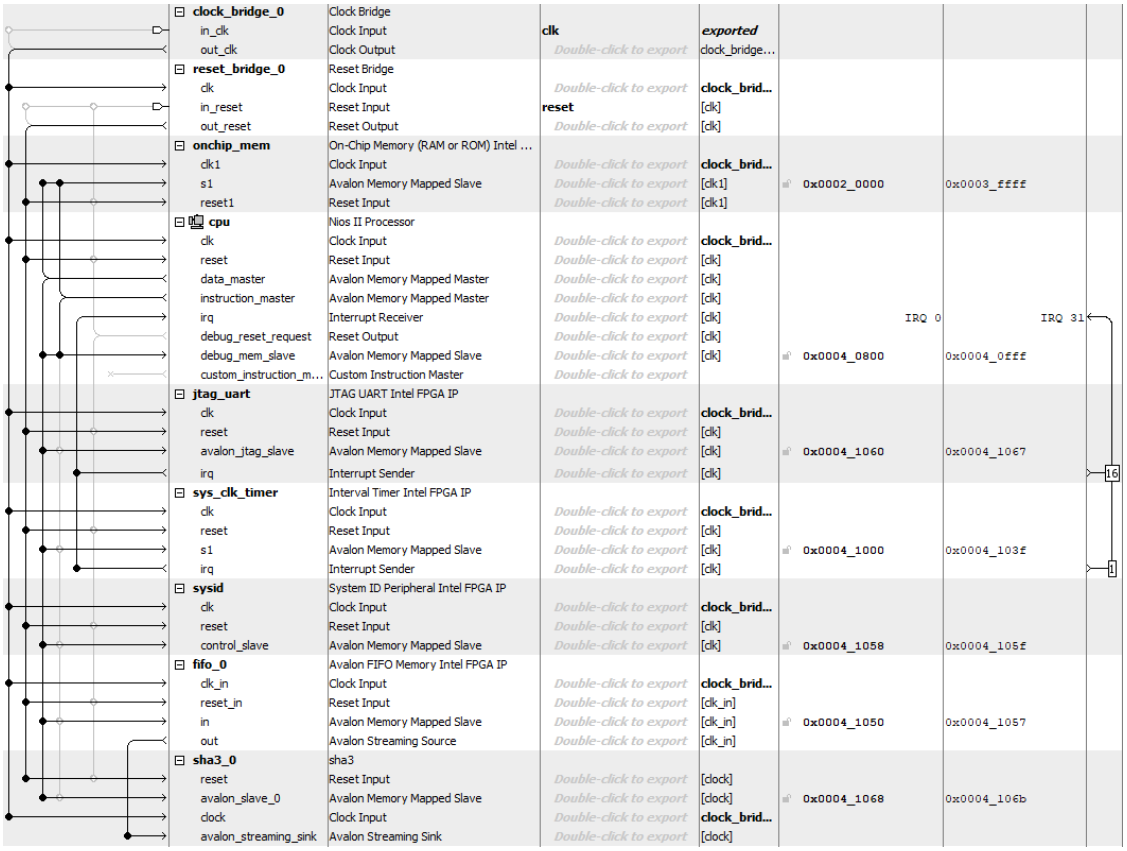

Figura 3.14: *Qsys per lo SHA3-512 versione hardware*

I blocchi all'interno del sistema rappresentato sono gli stessi della variante software,

ma in questo caso è stato eliminato il PLL, perché tutto il sistema lavora con una frequenza di clock pari a 100 MHz ed è stata aggiunta una memoria FIFO per le motivazioni descritte in precedenza. Ovviamente, nel sistema è anche presente il blocco chiamato *sha3*, che descrive il blocco hardware creato che include le interfacce per comunicare con il processore. Per aggiungere lo SHA-3 è stato utilizzato il comando *new component*, in cui è possibile includere i file VHDL che compongono il blocco, assegnare un nome al componente e una descrizione e infine definire tutti i segnali di interfacciamento. Prima di generare il sistema, sono stati assegnati gli indirizzi mediante il comando *Assign Base Addresses*, e si ottiene il seguente *Address Map* in figura 3.15:

	cpu.data_master	cpu.instruction_master
cpu.debug_mem_slave	0x0004_0800 - 0x0004_0fff	0x0004_0800 - 0x0004_0fff
fifo_0.in	0x0004_1050 - 0x0004_1057	
jtag_uart.avalon_jtag_slave	0x0004_1060 - 0x0004_1067	
led_pio.s1	0x0004_1040 - 0x0004_104f	
onchip_mem.s1	0x0002_0000 - 0x0003_ffff	0x0002_0000 - 0x0003_ffff
sha3_0.avalon_slave_0	0x0004_1068 - 0x0004_106b	
sys_clk_timer.s1	0x0004_1000 - 0x0004_103f	
sysid.control_slave	0x0004_1058 - 0x0004_105f	

Figura 3.15: *Address Map per lo SHA3-512 versione hardware*

Questi indirizzi saranno poi utilizzati per lo scambio dei dati tra il processore e il blocco SHA-3 e tra la FIFO e il processore.
Dopo aver generato il file *.sopcinfo*, utilizzando un nuovo progetto *Quartus* è stata effettuata la sintesi dell'architettura andando anche ad assegnare i PIN della scheda FPGA per il clock e il reset.

3.6.5.5 Simulazione dell'interfacciamento

Per simulare l'intero sistema in modo da poter anche visualizzare il timing effettivo dei segnali di controllo inviati dal processore e dalla FIFO, sono stati utilizzati due diversi software, il tool di *Quartus* chiamato *SignalTap Logical Analyzer* e *Modelsim*.
SignalTap è uno strumento di debug presente in *Quartus* ed è fondamentalmente un analizzatore logico digitale integrato di IP usato per guardare i segnali interni nel progetto mentre l'applicazione viene eseguita in tempo reale. *SignalTap* comunica attraverso la stessa connessione JTAG USB utilizzata per programmare il dispositivo ed eseguire il debug della memoria sul chip. Per visualizzare il segnale desiderato è possibile impostare una condizione di *trigger*, e quando viene soddisfatta le forme d'onda memorizzate sono quindi visualizzabili tramite l'interfaccia dell'analizzatore logico *SignalTap* all'interno di *Quartus*. Le forme d'onda visualizzate sono mostrate nelle figure 3.16 e 3.17:

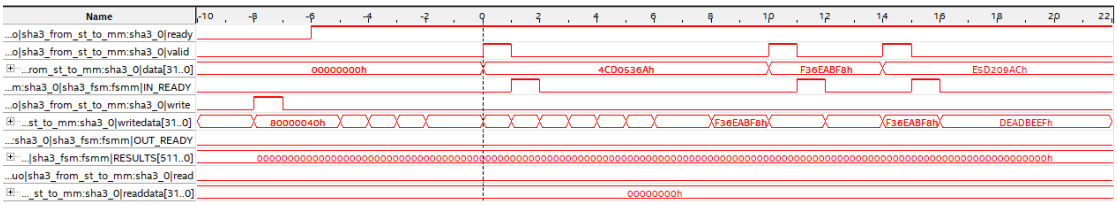

Figura 3.16: *Forme d'onda per il processo di acquisizione dati in SignalTap per lo SHA3-512*

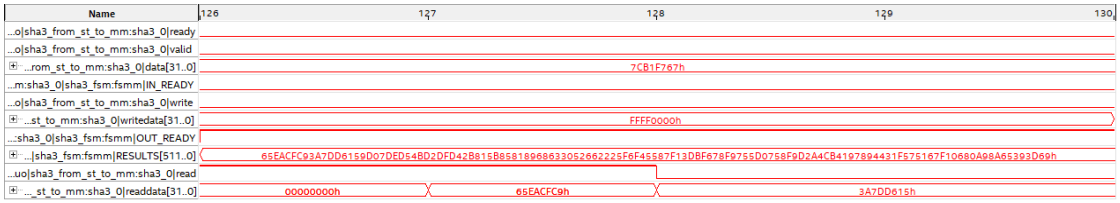

Figura 3.17: *Forme d'onda per il processo di rilascio dei dati in SignalTap per lo SHA3-512*

Nella prima immagine viene raffigurato l'inizio delle operazioni: quando pervengono i segnali di *write* e *writedata*, il blocco riceve lo *start* e aspetta i dati proveniente dalla FIFO tramite i segnali *valid* e *data* per poi iniziare con le operazioni.
Nella seconda figura è possibile visualizzare l'istante in cui il risultato finale è valido, e il blocco aspetta il segnale di *read* dal NIOS per intraprendere la lettura. Quando il segnale è a 1, il *readdata* contenente parte del risultato viene inviato al processore.
Per la simulazione con *Modelsim* è stata sfruttata la capacità del *Platform Designer* di generare un *testbench* dell'intero sistema. Dopo aver creato un piccolo codice in C per il passaggio e la lettura dei dati, il codice viene eseguito su *Modelsim* tramite la funzionalità del programma *Eclipse*. In questo modo è possibile visualizzare le diverse forme d'onda per una corretta simulazione del sistema.
Un esempio delle forme d'onda visualizzate è visibile in figura:

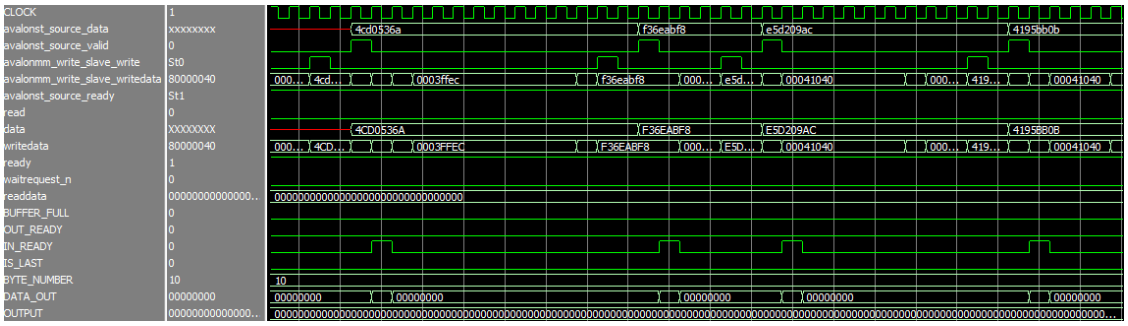

Figura 3.18: *Forme d'onda per lo SHA3-512 visualizzate in Modelsim utilizzando il testbench del Qsys*

3.6.5.6 Misurazione dei tempi di esecuzione

Per poter calcolare il tempo di esecuzione del blocco SHA3-512 in variante hardware, è stato utilizzato nuovamente *Eclipse* con gli stessi passaggi della versione software. Dopo aver generato il BSP e una nuova applicazione, è stato scritto un breve codice C per scrivere i dati all'interno della FIFO e i comandi nello SHA-3 e per leggere i risultati ottenuti dal blocco hardware. Per poter scrivere o leggere da differenti componenti IP presenti all'interno del *Platform Designer*, sono state utilizzate due diverse funzioni, chiamate *IOWR* e *IORD* definite nella libreria *io.h*. La prima viene utilizzata per la compilazione e riceve come parametri l'indirizzo presente nell'*address map* della figura 3.15, a seconda del componente in cui si vuole scrivere, un *offset*, in questo caso sempre 0, e infine il dato che si vuole elaborare. Questa funzione è anche stata prescelta sia per inviare i vettori contenenti i dati da elaborare alla FIFO, che dopo averli ricevuti li invia tramite l'interfaccia *Avalon-ST* allo SHA-3 e sia per inviare lo *start* e il numero totale di byte, in formato esadecimale, al blocco che costituiranno il *writedata*. I vettori dei dati sono definiti *unsigned int* e i loro valori sono espressi in base esadecimale in numero variabile a seconda dei byte da inviare. I dati utilizzati per il test in hardware sono esattamente gli stessi della versione software.
IORD, invece, riceve l'indirizzo su cui si vuole leggere il dato e un *offset*, sempre pari a 0. La funzione ritorna un intero che corrisponde al valore letto, e nel caso del sistema sviluppato, rappresenta i blocchi da 32 bit che costituiscono il risultato finale. Tutti i 512 bit letti sono stati salvati all'interno di un vettore, che viene comparato con un vettore test che contiene il risultato esatto ricavato utilizzando il sito di calcolo online dello SHA3-512 [14], in modo tale che se i due vettori sono identici viene stampato a video un messaggio che avvisa della correttezza delle operazioni.
Per l'ottimizzazione impostata in *Eclipse*, è stato utilizzato il *Level 3* perché il codice C in versione hardware occupa pochi kbyte e di conseguenza non è stato necessario ottimizzarlo per dimensioni.

La misurazione del tempo di esecuzione è stata effettuata con l'ausilio di un contatore che permette di ottenere il numero totale di colpi di clock a partire dallo stato successivo alla ricezione del segnale *start* fino al completamento delle operazioni, cioè quando il risultato finale è pronto per essere letto. Il contatore è stato inserito all'interno della macchina a stati e il suo valore all'esterno viene letto dal segnale *readdata*. Dopo la lettura del risultato, all'interno del codice C al termine della lettura del risultato la funzione *IORD* viene nuovamente richiamata per poter stampare la variabile chiamata *ticks* e di conseguenza calcolare il tempo di esecuzione moltiplicandola con il periodo del clock, che nella versione hardware equivale a 10 ns.
Un esempio di stampa del risultato nella console di *Eclipse* è visualizzato nella figura 3.19:

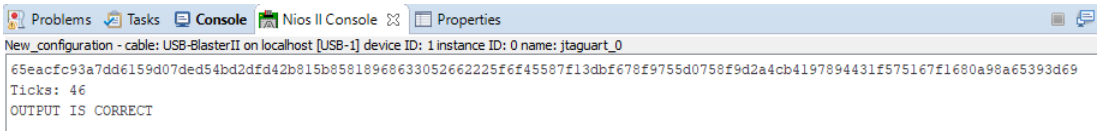

Figura 3.19: *Esempio di stampa del risultato nella console di Eclipse per la versione hardware dello SHA3-512*

3.6.6 Risultati ottenuti

Nel caso di un blocco hardware il primo risultato da analizzare riguarda l'area, in termini di ALM, occupata dal blocco e il numero totale di bit di memoria [3]. L'ALM è l'elemento costitutivo di base delle famiglie di dispositivi supportate, come la serie Cyclone V, ed è progettato per massimizzare le prestazioni e l'utilizzo delle risorse. Ogni ALM può supportare fino a otto ingressi e otto uscite, contiene due o quattro registri e due logiche combinatorie, due sommatori dedicati e una LUT a 64 bit. La scheda FPGA *Cyclone V* utilizza un nuovo ALM a 28 nm.
Un esempio di ALM con una LUT a otto ingressi e con quattro registri dedicati è schematizzato nella figura 3.20:

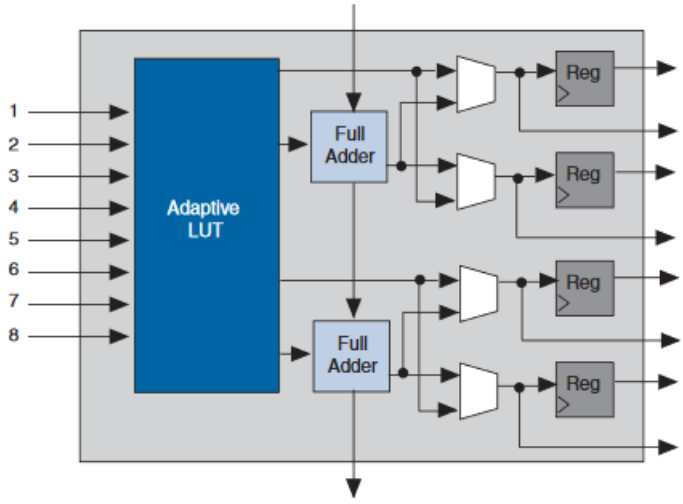

Figura 3.20: *Esempio di ALM della scheda FPGA Cyclone V [3]*

Questi dati sono stati ottenuti tramite la sintesi eseguita con *Quartus* e sono riassunti nella tabella 3.8:

ALM (SHA3-512)	4862.4/18 480 = 26%
ALM (CPU)	1874.2/18 480 = 10%
ALM (TOT)	6736.6
Bit di memoria (TOT)	1 344 000/3 153 920 = 43%

Tabella 3.8: *Area occupata e bit di memoria per lo SHA3-512*

L'area non risulta particolarmente elevata in quanto occupa solo il 26% dello spazio disponibile sulla scheda. Si deve, però, tener presente che, se si vuole integrare il blocco hardware con altre risorse, il dato dell'area potrebbe risultare effettivamente troppo alto.

I risultati dei tempi di esecuzione espressi in μs sono sintetizzati nella tabella 3.9:

	DIMENSIONI INPUT (B)										
	64	128	256	512	1024	1400	2048	4096	5028	8192	9000
TEMPO DI ESECUZIONE (μs)	0.46	0.69	1.16	1.90	3.56	4.76	6.92	13.40	16.28	26.36	30.32

Tabella 3.9: *Tempi di esecuzione in versione hardware per lo SHA3-512*

Per una migliore visualizzazione dei risultati ottenuti viene riportato un grafico in figura 3.21 :

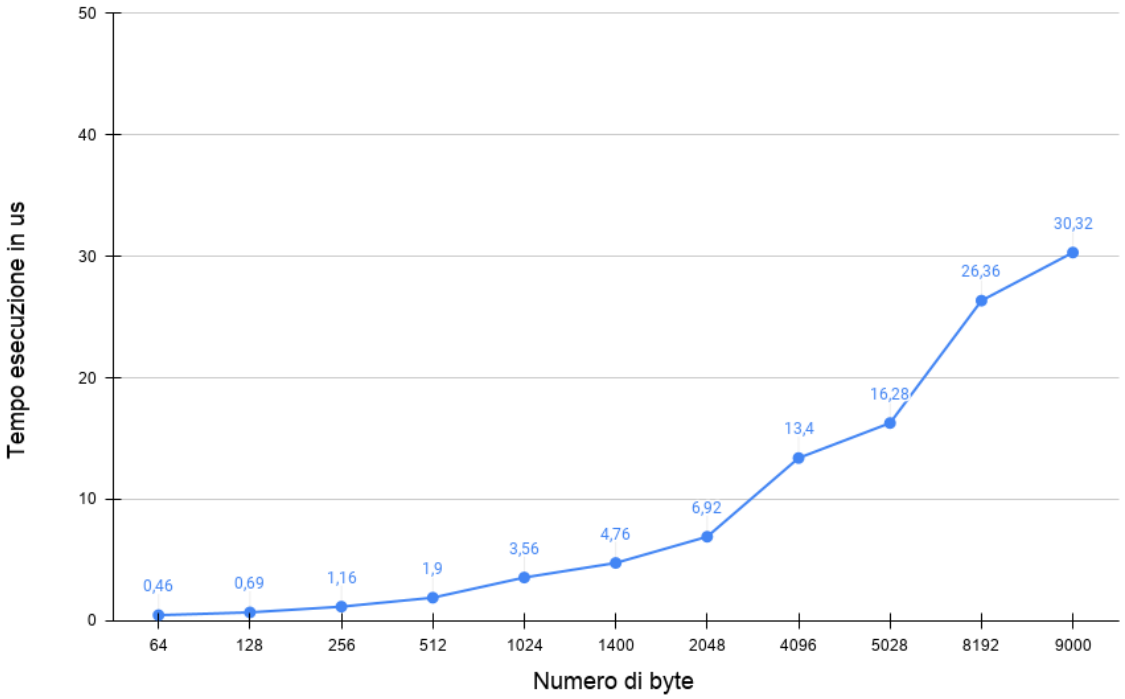

Figura 3.21: *Tempi di esecuzione dello SHA3-512 in versione hardware*

I tempi di esecuzione, come già spiegato, sono stati calcolati da quando il blocco SHA-3 riceve il primo dato valido a quando il risultato è disponibile, in modo da poter escludere il tempo necessario al riempimento della FIFO che risulterebbe molto lungo e porterebbe ad un incremento degli esiti. Anche in questo caso il valore ottenuto in corrispondenza dei 64 byte, rappresenta il minimo tempo di esecuzione che il blocco SHA3-512 impiega per il calcolo del risultato finale. Questa considerazione è in linea con quanto descritto nel *datapath*, poiché ricevendo un dato da 32 bit ogni colpo di clock, per riempire tutti i 576 bit del padder, il tempo

impiegato sarà di 18 colpi di clock a cui vanno sommati i 24 del calcolo della permutazione, per un totale di 42 colpi di clock che con un periodo di 10 ns equivale a 0.42 μs. In tabella viene riportato 0.46 μs tenendo conto di stati aggiuntivi nella FSM.

3.7 Confronto tra le due implementazioni

Per concludere l'analisi delle due diverse implementazioni dello SHA3-512, sono state realizzate due tabelle riassuntive che comparano i risultati ottenuti in modo da poter poi dedurre la soluzione migliore.
La tabella 3.10 presenta il resoconto delle risorse utilizzate:

SOFTWARE	Dimensioni codice (kB)	Ottimizzato per prestazioni	22
		Ottimizzato per dimensioni	17
HARDWARE	Dimensioni hardware	ALM (SHA3-512)	4862.4 /18 480 = 26%
		ALM (CPU)	1874.2/18 480 = 10%
		ALM (TOT)	6736.6
		Bit di memoria(TOT)	1 344 000/3 153 920 = 43%

Tabella 3.10: *Confronto tra le risorse utilizzate per le due implementazioni dello SHA3-512*

La tabella 3.11 confronta i tempi ottenuti:

		DIMENSIONI INPUT (B)										
		64	128	256	512	1024	1400	2048	4096	5028	8192	9000
SOFTWARE	Tempo esecuzione (ms)	1.71	3.40	6.80	13.57	25.44	33.91	49.17	96.63	118.66	193.60	214.17
HARDWARE	Tempo esecuzione (μs)	0.46	0.69	1.16	1.90	3.56	4.76	6.92	13.40	16.28	26.36	30.32

Tabella 3.11: *Confronto tra i tempi di esecuzione per le due implementazioni dello SHA3-512*

Per una comparazione più dettagliata dei dati, è stato realizzato nella figura 3.22 un grafico in scala logaritmica con i due diversi risultati ottenuti.
Analizzando la tabella 3.11, è possibile evidenziare una significativa riduzione dei tempi di esecuzione quando viene utilizzata la versione hardware. La diminuzione dei valori, in media del 99% rispetto a quelli della versione software, soddisfa ampiamente le aspettative, poiché l'algoritmo SHA-3, nei vari studi effettuati, è stato sempre valutato un algoritmo sicuro e resistente agli attacchi, ma lento quando viene eseguito in software. Questa considerazione, già confermata da diversi ricercatori, è stata riaffermata da queste analisi. Per quanto riguarda l'aspetto delle risorse, si può affermare che a livello software il codice dello SHA-3 risulta molto piccolo in termini di dimensioni, il che potrebbe essere un vantaggio se questa funzione dovesse essere inserita in un contesto più ampio dove magari le dimensioni del codice sono significative. Si può decisamente concludere che si ha un trade-off tra dimensioni e prestazioni nelle due implementazioni e che la versione hardware è sicuramente la più efficiente.

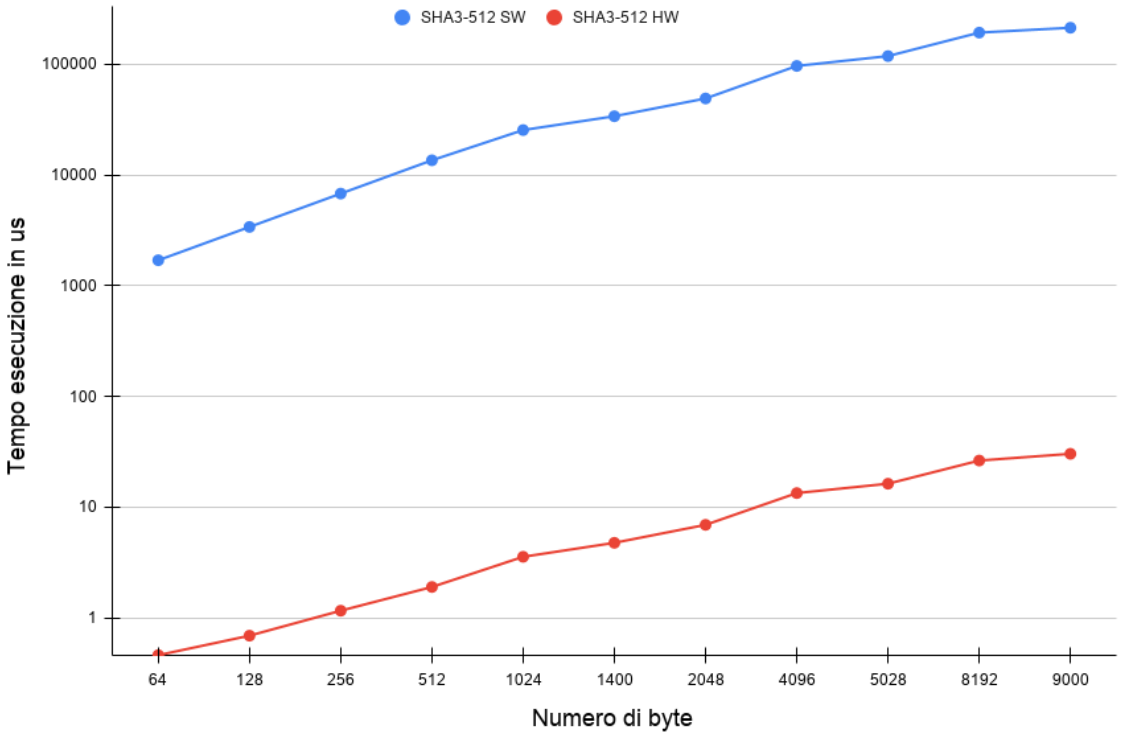

Figura 3.22: *Confronto tra i tempi di esecuzione dello SHA3-512 per le due implementazioni*

Capitolo 4

Algoritmo HMAC

4.1 Definizione

L'algoritmo HMAC è una modalità per l'autenticazione di messaggi basata su una funzione di hash, utilizzata in diverse applicazioni legate alla sicurezza informatica come, per esempio, verificare l'integrità delle informazioni trasmesse o la loro autenticità. In generale i codici di autenticazione dei messaggi vengono utilizzati tra le due parti che condividono una chiave segreta per convalidare le informazioni trasmesse tra loro.
HMAC si avvale di una combinazione del messaggio originale e di una chiave segreta per la generazione del codice.
Una caratteristica peculiare di HMAC è la sua indipendenza da qualunque funzione di hash che rende possibile la sostituzione della funzione se non dovesse dimostrarsi abbastanza sicura.
Anche per questa parte dello sviluppo di tesi, la funzione hash SHA-3, precedentemente analizzata e sviluppata, ha prevalso sulle altre.
Alcuni vantaggi di questo metodo di autenticazione sono:

1. può utilizzare senza modifiche tutte le funzioni hash disponibili, per esempio quelle funzioni che in software sono efficienti e che possono essere reperite facilmente;
2. preserva le prestazioni originali della funzione hash senza incorrere in un degrado significativo;
3. utilizza e gestisce le chiavi in modo semplice;
4. consente una facile sostituibilità della funzione hash sottostante nel caso in cui vengano trovate o richieste funzioni hash più veloci o più sicure.

La definizione generale di HMAC richiede una funzione hash crittografica, indicata in generale con H e una chiave segreta indicata con K. Il messaggio in ingresso viene diviso in j bit e poi si seleziona una chiave K che può essere formata da un numero di bit superiore, uguale o inferiore a j. Si determina il valore K' che in funzione dei tre casi assume valori diversi:

1. $K' = K \ se \ |K| = j \ bit$;
2. $K' = K + padding \ di \ zeri \ se \ |K| < j \ bit$;
3. $K' = H(K) \ se \ |K| > j \ bit$.

Si definiscono, inoltre, altre due quantità chiamate *ipad* e *opad*:

$$ipad = 00110110 \ ripetuta \ j/8 \ volte;$$
$$opad = 01011100 \ ripetuta \ j/8 \ volte.$$

Infine, è possibile definire la funzione HMAC come:

$$HMAC_K(M) = H((K' \oplus opad)||H((K' \oplus ipad)||M)).$$

Il simbolo || indica una semplice concatenazione posizionale [7].
Nel caso che è stato analizzato e implementato, H è la funzione SHA3-512, di conseguenza il parametro j è pari a 576 bit che corrisponde al *rate*. Di conseguenza i parametri *ipad* e *opad* saranno formati da 72 byte. La lunghezza della chiave scelta è di 64 byte, poiché è consigliabile utilizzare una lunghezza della chiave che sia almeno pari al numero di byte dell'uscita che per lo SHA3-512 è di 64 byte. Per calcolare K' si deve ricorrere al caso 2.
I passi da seguire per svolgere l'algoritmo HMAC sono:

1. eseguire il padding di zeri alla chiave per ottenere una stringa da 72 byte;
2. eseguire l'operazione di XOR tra la stringa ricavata in 1 e la stringa *ipad*;
3. concatenare alla stringa ricavata in 2 il messaggio in ingresso;
4. applicare lo SHA3-512 alla stringa risultante;
5. eseguire l'operazione di XOR tra la stringa ricavata in 1 e la stringa *opad*;
6. concatenare alla stringa ricavata in 5 l'uscita precedente dello SHA3-512;
7. applicare lo SHA3-512 alla stringa risultante per ricavare il risultato finale.

4.2 Implementazione software

Nell'implementazione software è stato eseguito l'algoritmo HMAC direttamente sul processore della scheda FPGA. Il codice C è stato sviluppato ampliando quello dello SHA-3, poiché in HMAC la funzione di hash viene richiamata due volte. Seguendo i sette passaggi descritti nel paragrafo precedente, è stata implementata una nuova funzione, così definita:

void HMAC(*const unsigned char *input*, *const unsigned char *key*,
unsigned int inputByteLen, unsigned char **output*);

I parametri ricevuti sono:

a. const unsigned char *input;

b. const unsigned char *key;

c. unsigned int inputByteLen;

d. unsigned char *output.

I parametri *a* e *b* sono puntatori ad un array di *unsigned char*(8 bit, senza segno) espressi in base esadecimale: *a* contiene il messaggio in ingresso mentre *b* la chiave a cui è stato già effettuato il padding di zeri.
Il parametro *c* è un intero senza segno che indica il numero di byte del messaggio in ingresso e infine il parametro *d* è un puntatore ad una stringa che salva il risultato finale dell'algoritmo HMAC.
All'interno di questa funzione, sono dichiarati i parametri *ipad* e *opad* come stringhe da 72 byte, che vengono definiti come costanti poiché sono valori fissi.
Quando la funzione HMAC viene richiamata dal main, riceve la chiave e calcola l'operazione di XOR bit a bit tra la chiave e la stringa *ipad* e richiama la funzione *FIPS202_SHA3_512* che riceve come parametri la concatenazione tra lo XOR calcolato e la chiave di ingresso, il numero di byte dell'ingresso, ricevuto dal main, sommato ai 72 byte dello XOR. Il risultato calcolato dallo SHA3-512 viene salvato in una stringa intermedia. A questo punto viene eseguito l'altra operazione di XOR, tra la chiave e la stringa *opad*, e viene nuovamente richiamata la funzione *FIPS202_SHA3_512*, che questa volta riceve come parametri la concatenazione tra la stringa dello XOR e il risultato calcolato in precedenza, il numero totale di byte che questa volta è fisso e pari a 136, che corrisponde alla somma tra i 72 byte della stringa XOR e i 64 byte del risultato finale dello SHA3-512. Quando lo SHA3-512 ha finito l'elaborazione, la stringa contenente il risultato viene ricevuta dal main che ha il compito di dichiarare gli ingressi e la chiave e passarli alla funzione.
Dopo aver testato in un primo momento il codice C elaborato, il passo successivo è quello di eseguirlo sul processore NIOS II della scheda. I tools utilizzati sono gli stessi dello SHA-3 e in particolar modo il Qsys adottato è quello della figura 3.4, poiché permette di usare un PLL per aumentare la frequenza in modo da poter

lavorare a 140 MHz.
Il file *.sopcinfo* creato dal Platform Designer nello sviluppo dell'algoritmo SHA-3, può essere usato in *Eclipse* anche per HMAC, di conseguenza anche il BSP. Per poter inserire però un nuovo codice C, è possibile creare una nuova applicazione che contiene il codice descritto in precedenza, e anche in questo caso sono stati modificati i parametri di ottimizzazione, impostando dapprima un'ottimizzazione per dimensione e dopo una di tipo livello 3 per ottenere il massimo delle prestazioni.
Il codice C all'interno dell'applicazione creata contiene una serie di vettori di test, che permettono il confronto e la verifica della correttezza del risultato sfruttando il calcolatore online di HMAC [10]. Anche in questo caso l'uscita viene stampata a video nella *console* di *Eclipse* e poi confrontata con il valore contenuto nel vettore di test, per poi stampare a video un messaggio che indica la correttezza del test quando i due vettori coincidono.
Per calcolare le dimensioni che il codice occupa in memoria sono state analizzate le informazioni contenute nella *console* di *Eclipse* durante la compilazione, mentre per misurare il tempo di esecuzione sono state riutilizzare le due funzioni *void alt_timestamp_start()* per il reset del contatore e *int alt_timestamp()* per il conteggio dei *clock ticks*. Il tempo di esecuzione è stato misurato da quando la funzione HMAC viene richiamata nel main fino a quando termina per poi ritornare nel main. Quindi, sono state definite le due variabili *start_time* e *end_time*, *start_time* contiene il numero di *clock ticks* intercorsi tra il reset del contatore e il richiamo della funzione HMAC, mentre *end_time* indica il numero di *clock ticks* intercorsi tra *start_time* e il termine delle operazioni della funzione HMAC. Il tempo è stato misurato in ms con la solita espressione:

$$Tempo = \frac{end\_time - start\_time}{frequency} \cdot 1000.$$

Anche in questo caso la misura del tempo di esecuzione non include la presenza di funzioni *printf*. Un esempio del messaggio visualizzato nella *console* di *Eclipse* è mostrato in figura 4.1:

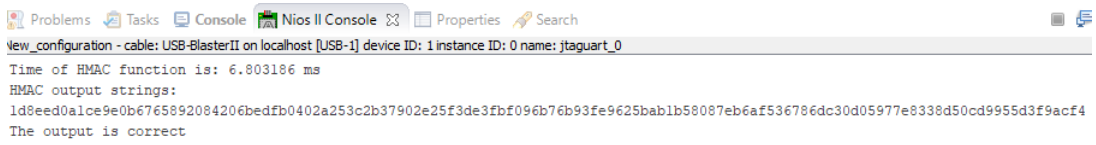

Figura 4.1: *Messaggio stampato nella console di Eclipse per HMAC versione SW*

4.2.1 Risultati ottenuti

Il primo risultato ottenuto riguarda la dimensione del codice della sola funzione HMAC, escludendo quindi tutta l'interfaccia esterna e il codice della funzione

FIPS202_SHA3_512 che viene richiamata.
I valori sono indicati nella tabella 4.1:

	Dimensione del codice (kB)
Ottimizzato per prestazioni	54
Ottimizzato per dimensioni	49

Tabella 4.1: *Dimensione del codice in kB per la funzione HMAC*

Analizzando la tabella, anche per la funzione HMAC è possibile osservare una riduzione di circa il 23% del numero di byte occupati dal codice con l'ottimizzazione per dimensione. Il codice, in entrambi i casi, occupa più byte rispetto a quello dello SHA-3, questo deriva anche dal fatto che all'interno della funzione HMAC sono dichiarate delle costanti come *ipad* e *opad* a 72 byte ciascuna che comporta un aumento di memoria allocata.
I risultati ottenuti dall'analisi dei tempi di esecuzione sono riportati nella tabella 4.2:

		DIMENSIONI INPUT (B)										
		64	128	256	512	1024	1400	2048	4096	5028	8192	9000
Tempo di esecuzione (ms)	Ottimizzato per prestazioni	6.80	8.50	11.88	18.65	30.51	38.97	54.21	101.62	123.63	198.12	218.45
	Ottimizzato per dimensioni	12.63	15.78	22.08	34.68	56.73	72.48	100.84	189.04	230.03	368.61	406.41

Tabella 4.2: *Tempi di esecuzione in software per l'algoritmo HMAC*

La tabella evidenzia una notevole riduzione dei tempi di esecuzione con un'ottimizzazione per prestazioni e precisamente i valori risultano ridotti circa del 50% rispetto alle misure ricavate con un'ottimizzazione per dimensioni.
Per una migliore visualizzazione, viene riportato un grafico con la comparazione dei due differenti risultati nella figura 4.2:

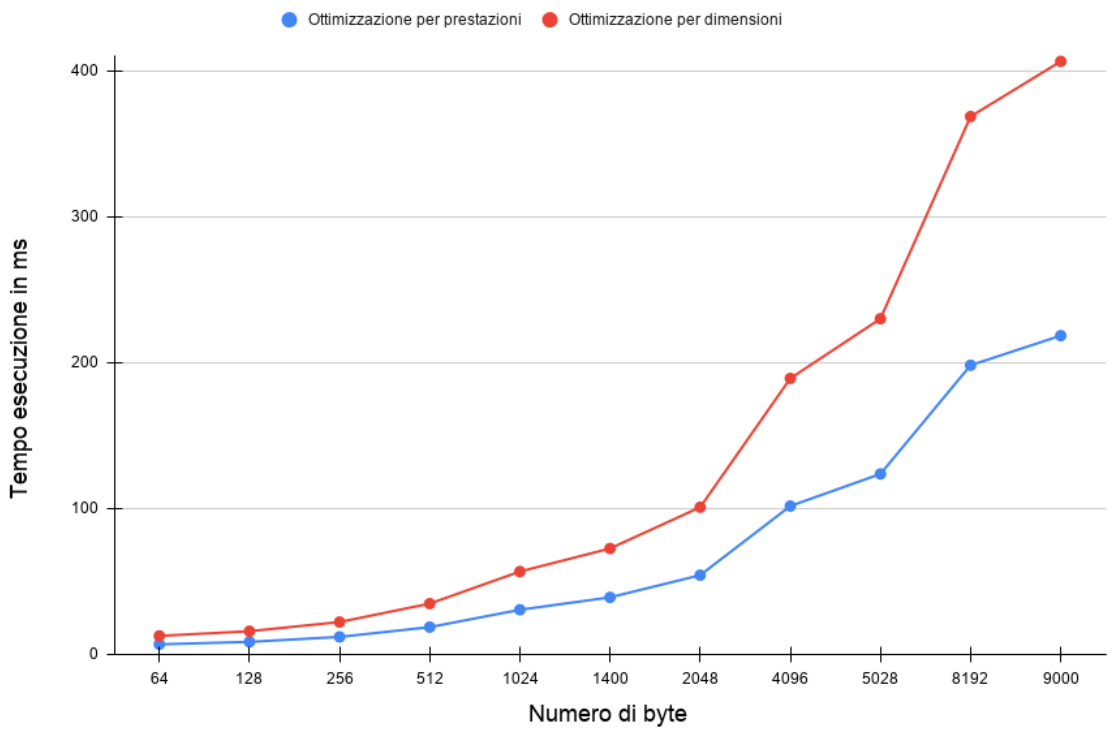

Figura 4.2: *Confronto tra i tempi di esecuzione dell' HMAC per le due differenti ottimizzazioni*

I diversi numeri di byte degli ingressi scelti per le varie misurazioni sono gli stessi utilizzati nello SHA-3, mentre la chiave è stata selezionata in maniera casuale. Il tempo di esecuzione in corrispondenza di un ingresso pari a 64 byte è il tempo minimo che la funzione HMAC impiega per il calcolo del risultato, come già evidenziato nello SHA-3. Lo stesso risultato è stato anche ottenuto adottando ingressi con un minor numero di byte.
Dall' analisi del grafico si può certamente concludere che anche per l'HMAC come per l'algoritmo SHA-3 è più vantaggioso utilizzare l'ottimizzazione per prestazioni quando si vogliono avere tempi ridotti di esecuzione e per questo, successivamente, nel confronto tra i risultati dell'hardware, saranno presi in considerazione solo i valori rappresentati nella prima riga della tabella 3.6. La frequenza utilizzata, come spiegato precedentemente è di 140 MHz, valore ottenuto anche in questo caso eseguendo la *Timing analyzer* realizzata con *Quartus*, che ha permesso lo studio del percorso critico e di trovare la massima frequenza con cui il sistema può lavorare.

4.3 Implementazione hardware

Per l'implementazione hardware, sono stati presi in considerazione sia il codice VHDL dello SHA3-512 descritto nel capitolo precedente e sia nuovi blocchi hardware utilizzati dall'algoritmo HMAC.

4.3.1 Datapath

Il datapath dell'architettura sviluppata è schematizzato nella figura 4.3:

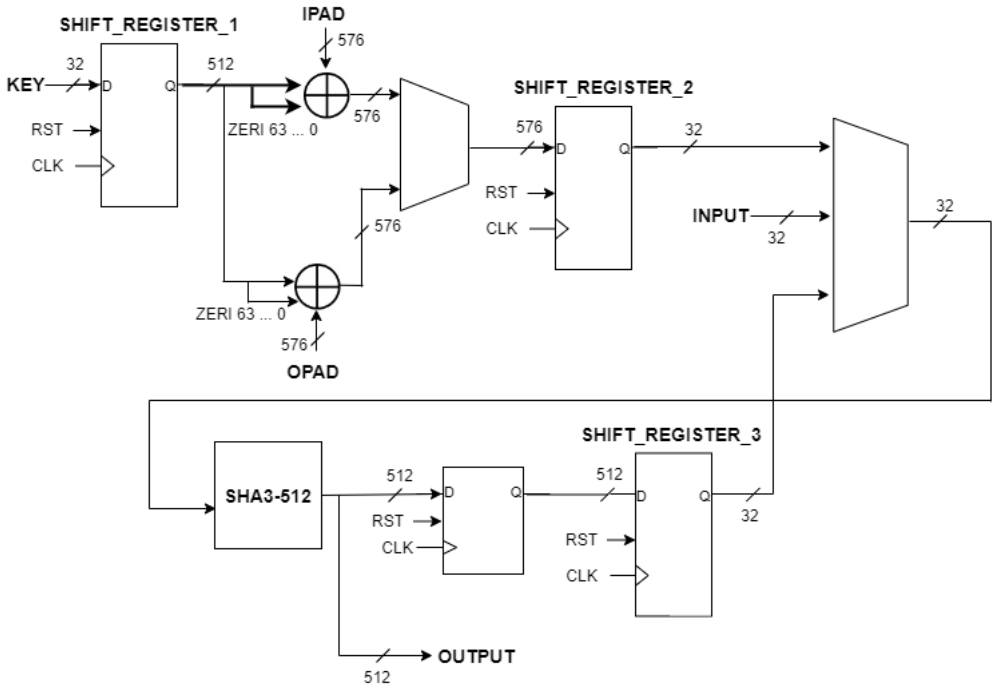

Figura 4.3: *Schema a blocchi HMAC*

Gli ingressi dell'architettura sono:

1. *clock e reset*;
2. *input* a 32 bit;
3. *key* a 32 bit;
4. *in_ready e is_last*;
5. *byte_num* su 2 bit.

I segnali *in_ready, is_last* e *byte_num* sono gli stessi descritti nello sviluppo dello SHA3-512 e per questo saranno nuovamente considerati come ingressi del blocco.

Il segnale *key*, un vettore di 32 bit, ha il compito di inviare la chiave da utilizzare durante l'elaborazione. I 32 bit ricevuti gradatamente dal blocco, vengono salvati in uno *shift_register*, in figura chiamato *shift_register_1*, che è in grado di ricevere i 32 bit in ingresso, accumularli e alla fine restituirà un vettore di 512 bit, cioè 64 byte, dello stesso valore definito per la lunghezza della chiave. Quando la chiave è disponibile, viene effettuato prima in maniera combinatoria il *padding* con degli zeri in modo da ottenere una lunghezza finale di 576 bit e poi viene eseguita l'operazione di XOR con vettori che contengono i segnali *opad* e *ipad* che non sono ricevuti dall'esterno, ma sono definiti costanti poiché il loro valore è fisso. Dopo aver eseguito l'operazione di XOR il risultato viene trasmesso tramite un *multiplexer* ad un secondo *shift_register*, in figura chiamato *shift_register_2*, che riceve un dato a 576 bit e rilascia ad ogni colpo di clock, i primi 32 bit. Il multiplexer è gestito dalla macchina a stati e prima fa passare i 576 bit derivanti dallo XOR tra *ipad* e la chiave, mentre, successivamente, avrà in uscita i 576 bit derivanti dallo XOR tra *opad* e la chiave.
Il secondo multiplexer è a quattro entrate, ma una non viene utilizzata. Il primo ingresso rappresenta l'uscita dello *shift_register_2*, il secondo, l'ingresso esterno che contiene il messaggio da elaborare e il terzo, l'uscita dello *shift_register_3*. Anche questo multiplexer viene gestito dalla macchina a stati che all'inizio seleziona il primo ingresso, che viene acquisito dal blocco SHA3-512 che parte con l'elaborazione, come descritto ampiamente nel capitolo precedente. Dopo che tutti i 576 bit dello XOR sono stati inviati allo SHA-3, il multiplexer cambia ingresso e seleziona il dato esterno inviato 32 bit alla volta. Dopo che l'intero messaggio è stato ricevuto dallo SHA-3, quando il risultato intermedio è pronto, viene salvato all'interno di un registro perché per poter eseguire una nuova elaborazione, il blocco SHA-3 deve essere resettato, e questo comporterebbe la perdita del risultato. L'uscita del registro diviene l'ingresso del terzo *shift_register*, in figura indicato come *shift_register_3*, che riceve i 512 bit del risultato e ad ogni colpo di clock ne rilascia 32, che diventano il terzo ingresso del multiplexer. Dopo che lo SHA3-512 ha terminato l'elaborazione, viene applicato il reset, e poi il primo multiplexer pilota questa volta il secondo ingresso, cioè lo *XOR* tra *opad* e la chiave, il secondo MUX gestisce nuovamente il primo ingresso, che viene ricevuto dal blocco SHA-3, e dopo aver terminato l'invio del vettore di XOR il secondo multiplexer governa il terzo ingresso in modo da poter concatenare il risultato precedente. Quando lo SHA3-512 produce il risultato finale, viene direttamente inviato all'esterno, senza essere salvato nel registro.
Le uscite del blocco HMAC sono:

1. *output* su 512 bit;
2. *out_ready*;
3. *buffer_full*.

Out_ready e *buffer_full* sono i segnali di controllo che il blocco SHA-3 produce e

sono inviati alla macchina a stati per gestire le operazioni, mentre il segnale *output* rappresenta il risultato finale dell'elaborazione.
Anche per l'algoritmo HMAC, l'ingresso può essere di qualsiasi lunghezza e con un numero di byte sia pari che dispari.

4.3.2 Interfacciamento con il processore NIOS II

Dopo aver eseguito una simulazione del sistema creato utilizzando un *testbench* e il programma *Modelsim*, è stata studiata nuovamente una possibile soluzione per poter interfacciare il blocco HMAC con il processore. Per lo scambio di dati e controlli sono state nuovamente utilizzate le interfacce *Avalon-ST* e *Avalon-MM*, in particolar modo la prima è utile per l'invio della chiave e poi del messaggio, in questo modo è possibile trasmettere dati a 32 bit con un'elevata frequenza che non richiedono l'uso di indirizzi. Come già precedentemente spiegato, la mancanza di indirizzi comporta nuovamente l'uso di una memoria FIFO *single-clock*, per trasformare una comunicazione di tipo *Avalon-MM* in una di tipo *Avalon-ST*. La seconda interfaccia, come nello SHA-3 è stata utilizzata per l'invio dei segnali di controllo e per trasmettere il risultato al processore. Ovviamente per la gestione del clock e del reset, sono state adottate nuovamente le interfacce *Avalon Clock Interface* e *Avalon Reset Interface*.
L'*Avalon-MM*, ancora una volta, ha operato con i segnali già presentati nel capitolo precedente, in particolare i segnali *read, readdata, write, writedata e waitrequest*. Il segnale *readdata*, gestito dai segnali *read e waitrequest* contiene il risultato finale delle operazioni, diviso su 32 bit, che viene inviato al processore mediante 16 cicli di lettura. Il *writedata*, coordinato da *write e waitrequest*, è un vettore a 32 bit utilizzato nuovamente per l'invio dei segnali di controllo tra il processore NIOS II e il blocco HMAC, con la stessa suddivisione dei bit già spiegata, il bit *writedata(31)* è il segnale di *start*, mentre i rimanenti bit costituiscono il numero totale di byte che compongono il messaggio in ingresso. Nell'implementazione software, il parametro *inputByteLen* corrisponde alla somma tra i byte del messaggio e quelli della chiave, mentre in hardware equivale solamente al numero di byte del messaggio perché la chiave e il messaggio sono gestiti in maniera differente, e quindi sfruttando nuovamente un contatore è possibile asserire il segnale *is_last* quando è stato individuato l'ultimo ingresso.
Il *timing* dei segnali di questa interfaccia è lo stesso descritto e poi mostrato in figura 3.11. L'interfaccia *Avalon-ST* è caratterizzata allo stesso modo dai segnali di *valid, data e ready*, con le stesse considerazioni espresse nello sviluppo dello SHA-3, e con lo stesso *timing* mostrato in figura 3.12. Il segnale *data*, gestito dai segnali *valid e ready*, ha il compito di ricevere, 32 bit alla volta, la chiave, di salvarla nello *shift_register_1* e di ricevere il messaggio.

4.3.2.1 FSM

Per associare il blocco HMAC con le due interfacce descritte, così come nello SHA-3, è stata creata in VHDL una *top_entity* che possiede come ingressi e uscite, oltre al *clock* e al *reset*, i segnali dell'*Avalon-MM* e dell'*Avalon-ST* e mette insieme il *datapath* descritto con la macchina a stati.
La macchina a stati è schematizzata in figura 4.4:

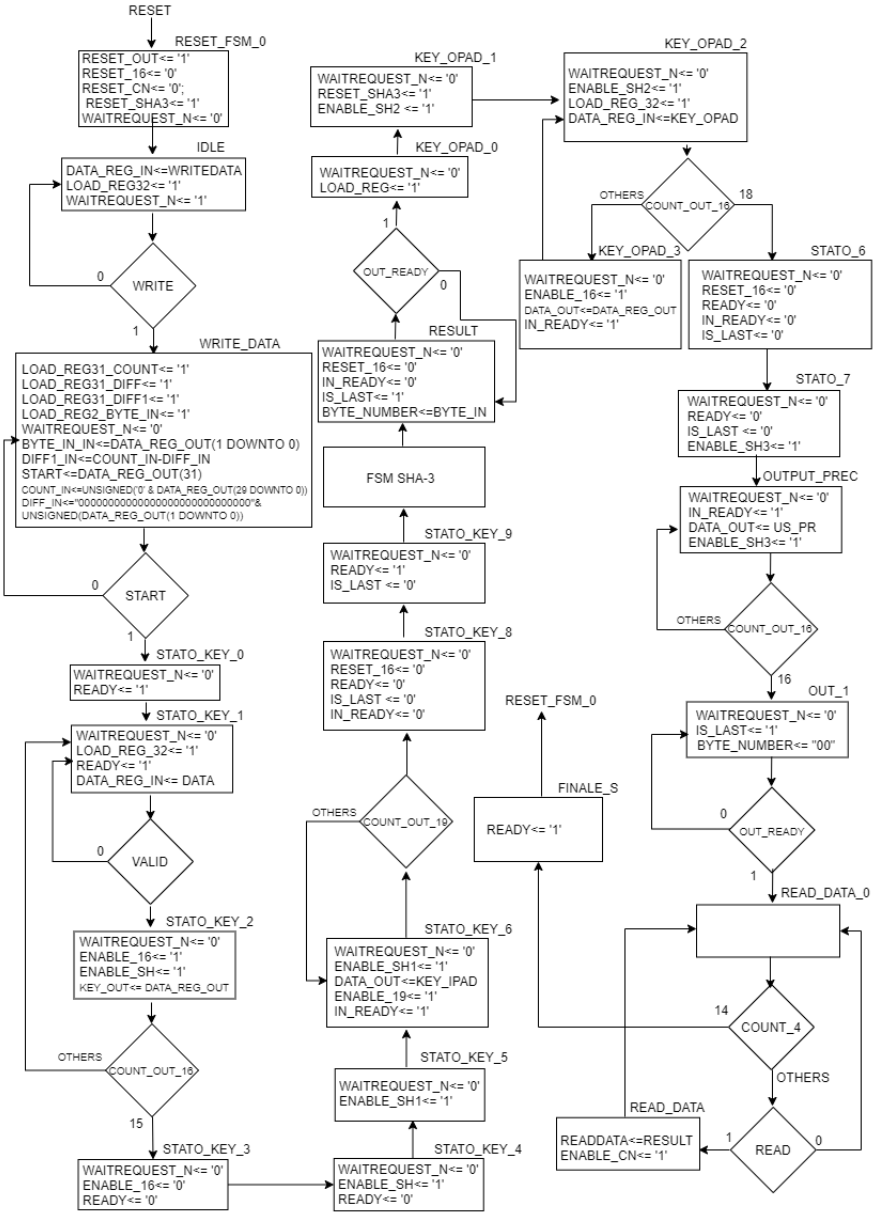

Figura 4.4: *FSM HMAC*

La macchina a stati può essere suddivisa in tre stadi, nel primo si ha l'acquisizione della chiave dall'esterno tramite un contatore che verifica se tutte le 16 parti che costituiscono la *key* sono state inviate e dopo aver applicato il *padding* di zeri ed eseguito lo XOR ha inizio la parte intermedia. In questa parte all'inizio viene inviato al blocco SHA3-512 lo XOR della chiave con il segnale *ipad*, e viene utilizzato un secondo contatore, che questa volta conta fino al valore 18, poiché la lunghezza dopo lo XOR è di 72 byte, per verificare che tutto il vettore di XOR è stato inviato. Durante tutta questa operazione il segnale *ready* è ovviamente 0, poiché il messaggio non può ancora essere ricevuto e per questo è necessario segnalare alla FIFO che non può inviare nuovi dati. Dopo aver terminato l'invio dello XOR è possibile ricevere i dati che costituiscono l'ingresso, in questo caso la successione degli stati è identica a quella mostrata nella FSM dello SHA-3 e per questo nell'immagine viene riportata solamente la dicitura *FSM SHA-3*. Per la gestione del messaggio in ingresso, come descritto nella FSM dello SHA-3, viene utilizzato un contatore il cui valore è comparato con il numero totale di byte dell'ingresso contenuto all'interno del segnale *writedata*. Dopo che il risultato intermedio è stato salvato nel registro, il blocco SHA-3 viene resettato e ha inizio l'ultima fase in cui viene prima inviato lo XOR tra la chiave e il segnale *opad*, con il riutilizzo del primo contatore e infine il risultato precedente con l'ausilio dello stesso contatore. Infine, quando il valore finale è disponibile viene inviato al processore mediante il segnale *readdata*. All'interno della macchina a stati, per la transizione degli stati oltre ai valori dei contatori sono stati utilizzati anche i segnali delle varie interfacce ma anche del blocco SHA-3.

4.3.2.2 Sintesi del sistema

Il passo successivo è stato quello di integrare il componente hardware con i restanti componenti IP, utilizzando *Quartus* e il tool *Platform Designer*. Il sistema ottenuto è descritto nella figura 4.5:

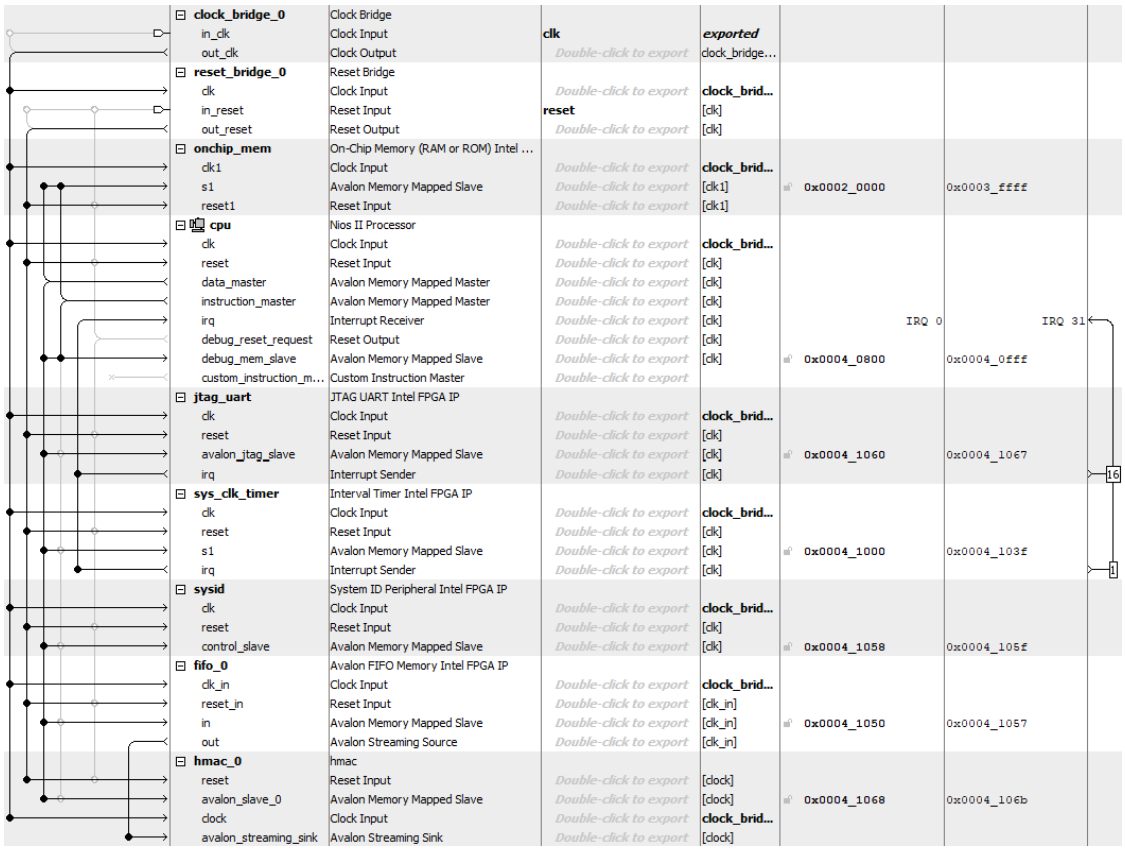

Figura 4.5: *Qsys per HMAC versione hardware*

Il sistema rappresentato in figura 4.5 è simile a quello dell'algoritmo SHA-3 e differisce solamente per l'ultimo componente, chiamato *hmac*, che ovviamente è costituito dal blocco HMAC, generato sempre con il comando *new component*, includendo tutti i file VHDL e definendo tutti i segnali di interfacciamento. Anche nella versione hardware dell'algoritmo HMAC, la frequenza di clock è pari a 100 MHz, a differenza della variante software che vale 140 MHz. Prima di generare il sistema, sono stati assegnati gli indirizzi che costituiscono il seguente *Address Map* elencati nella figura 4.6:

	cpu.data_master	cpu.instruction_master
cpu.debug_mem_slave	0x0004_0800 - 0x0004_0fff	0x0004_0800 - 0x0004_0fff
fifo_0.in	0x0004_1050 - 0x0004_1057	
hmac_0.avalon_slave_0	0x0004_1068 - 0x0004_106b	
jtag_uart.avalon_jtag_slave	0x0004_1060 - 0x0004_1067	
led_pio.s1	0x0004_1040 - 0x0004_104f	
onchip_mem.s1	0x0002_0000 - 0x0003_ffff	0x0002_0000 - 0x0003_ffff
sys_clk_timer.s1	0x0004_1000 - 0x0004_103f	
sysid.control_slave	0x0004_1058 - 0x0004_105f	

Figura 4.6: *Address Map per l' HMAC versione hardware*

Anche in questo caso, gli indirizzi sono stati utilizzati per lo scambio dei dati tra il processore e il blocco HMAC e tra la FIFO e il processore. Dopo la generazione del file *.sopcinfo*, e dopo aver assegnato i PIN della scheda per il clock e il reset, è stata effettuata la sintesi dell'architettura con *Quartus*.

4.3.2.3 Simulazione dell'interfacciamento

Per simulare l'intero sistema sono stati adottati *SignalTap logical Analyzer* e *Modelsim*.
Le forme d'onda visualizzate sono mostrate nelle figure 4.7 e 4.8:

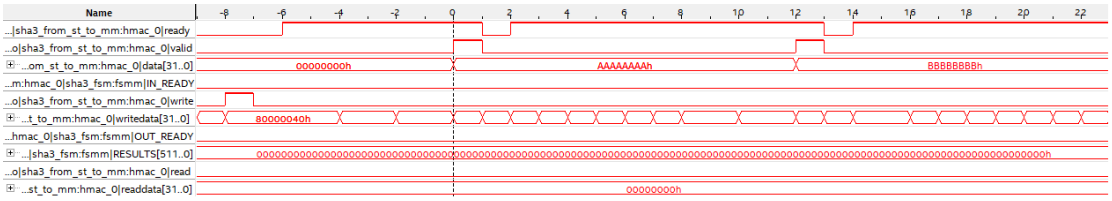

Figura 4.7: *Forme d'onda per il processo di acquisizione dati in SignalTap per l' HMAC*

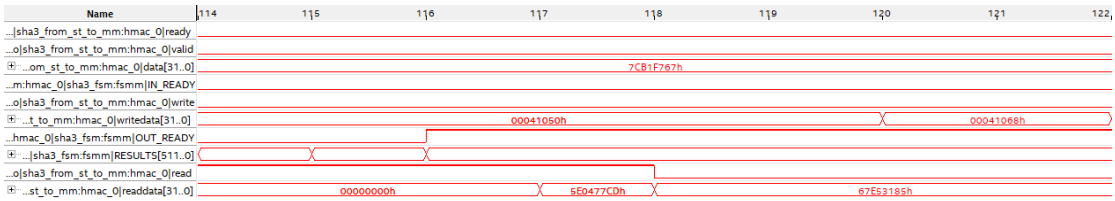

Figura 4.8: *Forme d'onda per il processo di rilascio dei dati in SignalTap per lo SHA3-512*

Nella prima immagine è possibile osservare il primo step delle operazioni: con l'arrivo dei segnali *write* e *writedata*, il blocco riceve lo start ed attende i dati che in questo caso sono i vettori di bit che costituiscono la chiave.

Nella seconda figura, invece, è possibile analizzare l'istante in cui il risultato finale è disponibile sull'uscita e il blocco aspetta il segnale *read* per inviare tramite il *readdata* parte dell'uscita al processore.
Per usare *Modelsim* è stato nuovamente generato un *testbench* del sistema dal *Platform Designer*. Anche in questo caso, per far partire la simulazione è stato creato un breve codice C per il passaggio e la lettura dei dati e con l'ausilio di *Eclipse* è possibile avviare la simulazione.
Un esempio di forme d'onda visualizzate in *Modelsim* è visibile in figura 4.9:

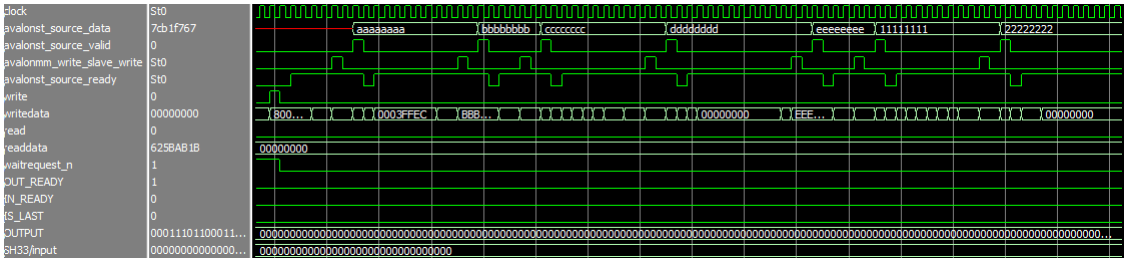

Figura 4.9: *Forme d'onda per l'HMAC visualizzate in Modelsim utilizzando il testbench del Qsys*

4.3.2.4 Misurazione dei tempi di esecuzione

Per poter misurare il tempo di esecuzione nella versione totalmente hardware dell'algoritmo HMAC è stato preferito nuovamente *Eclipse*. E' stato concepito un nuovo BSP sulla base del file *.sopcinfo* generato in precedenza, ed è stata poi creata una nuova applicazione contenente un breve codice C per la scrittura dei dati nella FIFO e dei comandi nel blocco HMAC. Le funzioni scelte per la scrittura e la lettura sono nuovamente *IOWR* e *IORD* definite con la libreria *io.h*.
La funzione *IOWR* è stata utilizzata insieme a due cicli *for*, il primo per poter scrivere all'interno della FIFO il vettore dei byte che costituisce la chiave, definito come *unsigned int*, il secondo per poter creare i vettori dei dati sempre definiti *unsigned int*, che contengono i byte del messaggio che si vuole elaborare. *IOWR* è stata anche prescelta per inviare il segnale di controllo, costituito dallo *start* e dal numero totale di byte, all'indirizzo del blocco HMAC presente nell'*address map* in figura 4.6. La chiave e i differenti byte che costituiscono l'ingresso sono gli stessi della versione software.
La funzione *IORD*, invece, che ritorna un valore intero corrispondente al dato letto, ha il compito di leggere 32 bit alla volta il risultato finale dell'operazione. I 512 bit totali, dopo essere stati letti, questi vengono salvati all'interno di un vettore che viene poi comparato con un vettore contenente il risultato esatto ricavato dal sito di calcolo online di HMAC [10], in modo tale che se i due vettori coincidono viene stampato un messaggio che avvisa della correttezza del test.
L'ottimizzazione per prestazioni su *Eclipse* è di tipo *Level 3* poiché il codice C è

molto piccolo.
Per il calcolo dei tempi di esecuzione è stato utilizzato nuovamente un contatore per conseguire il numero totale di colpi di clock a partire dallo stato successivo alla ricezione del segnale *start* fino all'ottenimento del risultato. Il contatore è stato inserito all'interno della FSM e il suo valore finale viene letto all'esterno tramite il segnale *readdata*. Come nello SHA-3, tramite la funzione *IORD*, è possibile leggere la variabile *ticks* e stamparne a video il suo valore e, di conseguenza il tempo di esecuzione sarà il suo valore moltiplicato il periodo di clock che nel caso hardware è pari a 10 *ns*.
Un esempio di quanto viene stampato nella *console* di *Eclipse* è sintetizzato nella figura 4.10:

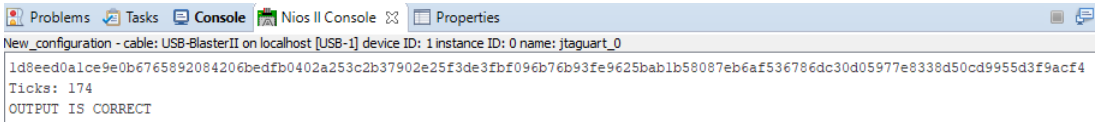

Figura 4.10: *Esempio di stampa del risultato nella console di Eclipse per la versione hardware dell' HMAC*

4.4 Risultati ottenuti

Il primo risultato analizzato riguarda l'area in termini di ALM occupata dal blocco e il numero totale di bit di memoria. Un esempio di ALM per la scheda FPGA *Cyclone V* è quello mostrato in figura 3.20.
I dati ottenuti tramite la sintesi eseguita con *Quartus* sono riussunti nella tabella 4.3:

ALM (HMAC)	5402/18 480 = 29%
ALM (CPU)	1858.2/18 480 = 10%
ALM (TOT)	7260.2
Bit di memoria (TOT)	1 344 000/3 153 920 = 43%

Tabella 4.3: *Area occupata e bit di memoria per l'algoritmo HMAC*

L'area che il blocco HMAC occupa equivale al 29% dello spazio disponibile sulla scheda, percentuale ovviamente più alta rispetto a quella dello SHA-3, ma in linea con le previsioni in quanto sono stati aggiunti dei componenti, anche se in generale non risulta troppo elevata, se considerata, come già spiegato, in un contesto isolato.
I risultati del tempo di esecuzione espresso in μs sono presentati nella tabella 4.4:

	DIMENSIONI INPUT (B)										
	64	128	256	512	1024	1400	2048	4096	5028	8192	9000
TEMPO DI ESECUZIONE (μs)	1.74	1.98	2.46	3.42	5.10	6.30	8.46	15.18	18.30	28.86	31.50

Tabella 4.4: *Tempi di esecuzione in versione hardware per l'algoritmo HMAC*

Per una migliore visualizzazione dei risultati, di seguito viene riportato un grafico nella figura 4.11:

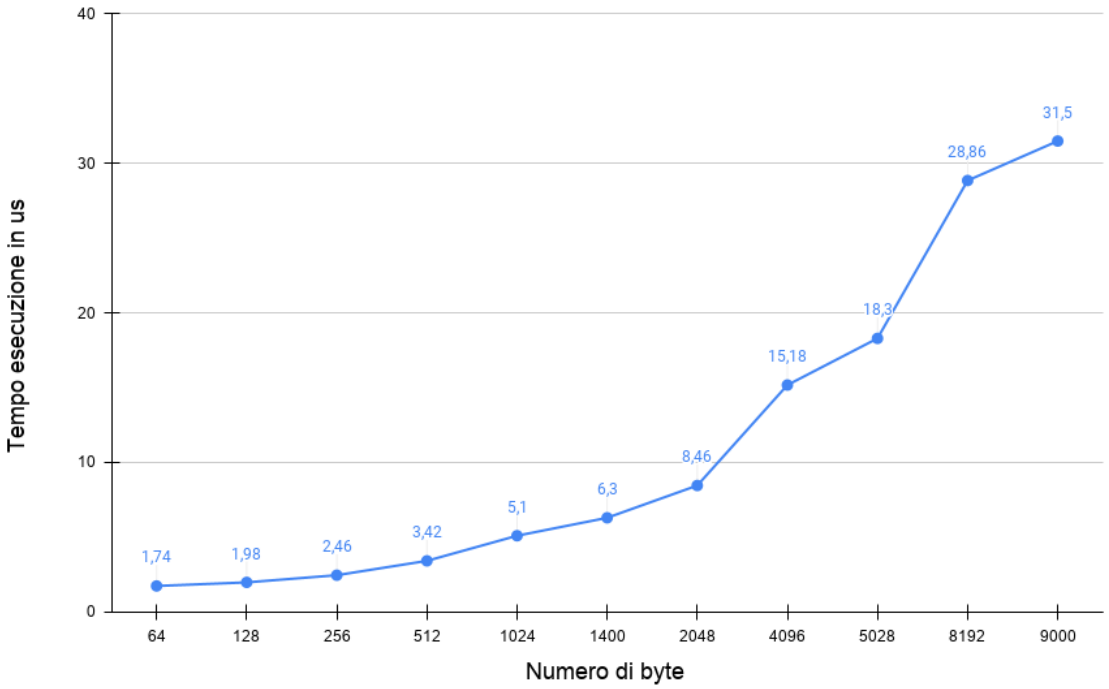

Figura 4.11: *Tempi di esecuzione dell' HMAC in versione hardware*

I risultati, come chiarito in precedenza, sono stati ottenuti escludendo il riempimento della FIFO e considerando solamente l'esecuzione di tutto il blocco HMAC. Analizzando i dati, è possibile evidenziare un aumento in generale del tempo di esecuzione per l'algoritmo HMAC rispetto al caso SHA-3, in linea con le aspettative, perché durante l'esecuzione dell'HMAC il blocco SHA3-512 compie due cicli di elaborazione e inoltre tra i due vi sono dei colpi di clock aggiuntivi. Si può sicuramente dedurre che il tempo di esecuzione dell'algoritmo HMAC dipende sostanzialmente da quello dello SHA-3, come evidenziato anche dalla parte teorica.
Anche per la versione hardware di questo algoritmo, il valore ottenuto in corrispondenza dei 64 byte rappresenta il minimo tempo di esecuzione che il blocco HMAC impiega per il calcolo del risultato finale.

4.5 Confronto tra le due implementazioni

Per concludere l'analisi dell'algoritmo HMAC sono state realizzate due tabelle riassuntive con i risultati ottenuti per le due diverse implementazioni.
La tabella 4.5 presenta il resoconto delle risorse utilizzate:

SOFTWARE	Dimensioni codice (kB)	Ottimizzato per prestazioni	54
		Ottimizzato per dimensioni	49
HARDWARE	Dimensioni hardware	ALM (HMAC)	5402 /18 480 = 29%
		ALM (CPU)	1852.8/18 480 = 10%
		ALM (TOT)	7260.2
		Bit di memoria(TOT)	1 344 000/3 153 920 = 43%

Tabella 4.5: *Confronto tra le risorse utilizzate per le due implementazioni dell' HMAC*

La tabella 4.6 confronta i tempi ottenuti:

		DIMENSIONI INPUT (B)										
		64	128	256	512	1024	1400	2048	4096	5028	8192	9000
SOFTWARE	Tempo esecuzione (ms)	6.80	8.50	11.88	18.65	30.51	38.97	54.21	101.62	123.63	198.12	218.45
HARDWARE	Tempo esecuzione (μs)	1.74	1.98	2.46	3.42	5.10	6.30	8.46	15.18	18.30	28.86	31.50

Tabella 4.6: *Confronto tra i tempi di esecuzione per le due implementazioni dell' HMAC*

Per una comparazione più dettagliata dei dati, è anche stato tracciato un grafico in scala logaritmica in figura 4.12.
Analizzando la tabella 4.6, è possibile evidenziare una significativa riduzione dei tempi di esecuzione con la versione hardware. La diminuzione dei valori, in media del 99% rispetto a quelli della versione software, soddisfa ampiamente le aspettative, poiché i tempi di esecuzione dell'algoritmo HMAC dipendono dai tempi dello SHA-3 che, come già ribadito, è valutato lento quando viene eseguito in software. Ovviamente l'algoritmo HMAC potrebbe essere velocizzato in software se, per esempio, si prendesse in considerazione funzione di hash, più efficiente in software.
Per quanto riguarda l'aspetto delle risorse, si può affermare che a livello software il codice risulta piccolo in termini di dimensioni, ma più grande di quello dello SHA-3. Questo come detto anche nel precedente capitolo, potrebbe essere un vantaggio se questa funzione dovesse essere inserita in un contesto più ampio dove magari le dimensioni del codice sono significative.
Anche nell'algoritmo HMAC si ha un trade-off tra dimensioni e prestazioni nelle due implementazioni seppur in maniera ridotta poiché il codice diventa più oneroso

in termini di dimensioni, ma sicuramente la versione più efficiente è quella hardware.

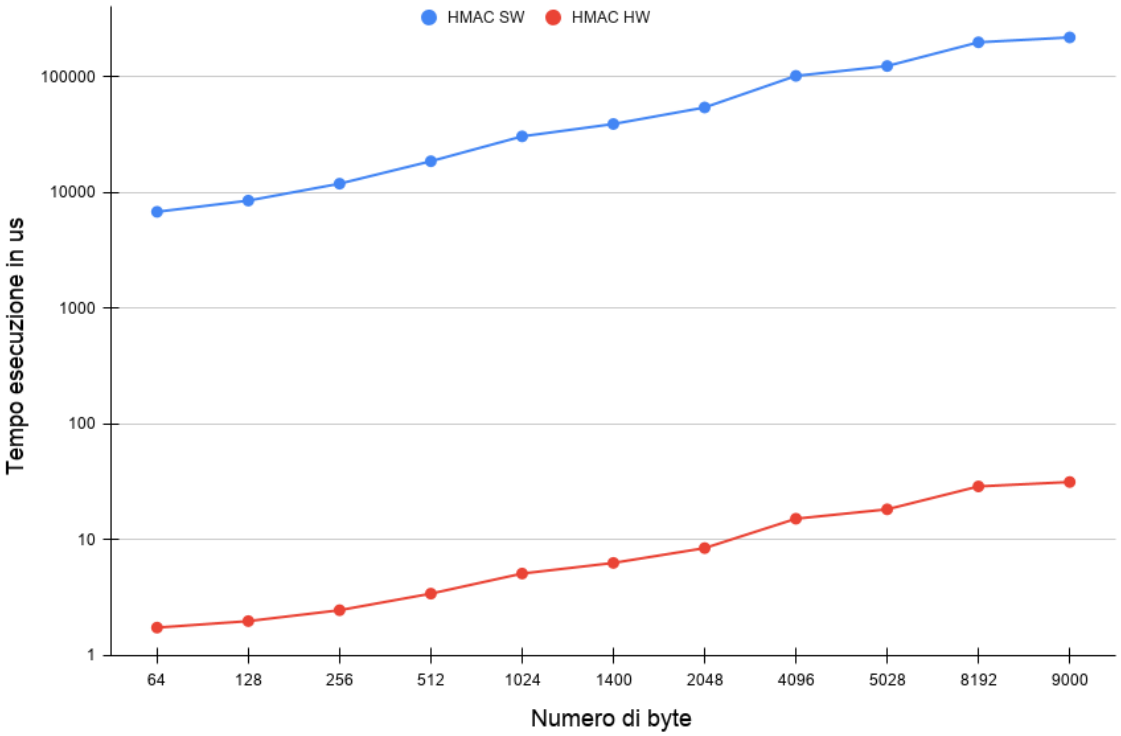

Figura 4.12: *Confronto tra i tempi di esecuzione dell' HMAC per le due implementazioni*

Capitolo 5

Conclusioni

Il lavoro presentato si è concentrato sull'analisi di un algoritmo criptografico di recente pubblicazione noto come SHA-3, in particolare sullo SHA3-512, e su un algoritmo di autenticazione dei messaggi indicato come HMAC. Per entrambi è stata effettuata un'attenta analisi che ha permesso la conoscenza delle basi matematiche su cui si fondano i due algoritmi e in seguito sono stati implementati sia in software e sia in hardware per valutarne le prestazioni.
I risultati ottenuti in termini di prestazioni evidenziano che, per entrambi gli algoritmi analizzati, la versione hardware è la più efficiente, con tempi di esecuzione che risultano inferiori, in media, del 99% rispetto a quelli software. Le misurazioni effettuate permettono di avvalorare un risultato già noto in letteratura, che afferma che l'algoritmo SHA-3, e di conseguenza l'HMAC è lento a livello software. Grazie a questi risultati, è possibile riutilizzare i blocchi dello SHA-3 o dell'HMAC sviluppati, all'interno di applicazioni criptografiche che richiedono un algoritmo efficiente a nella versione hardware. Una prima possibile prosecuzione del lavoro è incentrata sullo studio e implementazione degli altri tre SHA-3, lo SHA3-224, lo SHA3-256 e lo SHA3-384 in modo da valutare la dipendenza del tempo di esecuzione dal numero di bit del *digest*. Infine, è possibile modificare l'HMAC realizzato sostituendo lo SHA3-512 con gli altri membri della famiglia.

Bibliografia

[1] Valmorra Camilla. "Funzioni hash e sicurezza crittografica". Tesi di laurea. Alma Mater Studiorum - Università di Bologna, 2012/2013.

[2] Altera Corporation. *Avalon Interface Specifications*. URL: https://www.intel.com/content/www/us/en/programmable/documentation/nik1412467993397.html.

[3] Altera Corporation. *Cyclone V Device Family Advance*. URL: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/cyclone-v/cv_51001.pdf.

[4] Altera Corporation. *Embedded Design Handbook*. URL: https://www.intel.com/content/www/us/en/programmable/documentation/iga1446487888057.html.

[5] NIST Computer Security Division. *SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions*. FIPS Publication 202. National Institute of Standards e Technology, U.S. Department of Commerce, ago. 2015. URL: http://csrc.nist.gov/publications/drafts/fips-202/fips_202_draft.pdf.

[6] M. Peeters G. Bertoni J. Daemen e G. Van Assche. *Software resources*. URL: https://keccak.team/software.html.

[7] R. Canetti H. Krawczyk M. Bellare. *HMAC: Keyed-Hashing for Message Authentication*. Informational. Network Working Group, feb. 1997. URL: https://tools.ietf.org/html/rfc2104.

[8] Hsing, Homer. *SHA3 (KECCAK)*. 2012. URL: https://opencores.org/projects/sha3.

[9] Opal Kelly. *ZEM5310*. URL: https://opalkelly.com/products/zem5310/.

[10] Liavaag.org. *Online HMAC Generator*. URL: https://www.liavaag.org/English/SHA-Generator/HMAC/.

[11] RANDOM.ORG. *True Random Number Service*. URL: https://www.random.org/company/.

[12] Marioni Silvano. *Blockchain a rischio con i computer quantistici: quali soluzioni*. 2018. URL: https://www.agendadigitale.eu/sicurezza/blockchain-a-rischio-con-i-computer-quantistici-quali-soluzioni/.

[13] Lovati Stefano. *Implementazione degli algoritmi SHA-3 nei sistemi embedded*. 2020. URL: https://it.emcelettronica.com/implementazione-degli-algoritmi-sha-3-nei-sistemi-embedded.

[14] Online Tools. *SHA3-512*. URL: https://emn178.github.io/online-tools/sha3_512.html.