

# Zynq-7000: Sviluppo HDL di una periferica (SHA-256) custom in due versioni, con interfaccia AXI full ed AXI lite - Valutazione e confronto dei tempi di trasferimento PS-PL nei due casi

## Protocollo AXI4 (Full e Lite)

---

AMBA AXI4 è la specifica di un bus standard per la comunicazione tra due endpoint in un design hardware. Xilinx ha largamente adottato questa specifica, riutilizzando interfacce AXI4 fornite da venditori di IP Core (come ARM). Xilinx mette a disposizione una propria implementazione della specifica per i progettisti digitali, consentendo di istanziare IP Core equipaggiati di tale implementazione. Le implementazioni proprietarie di AXI da parte di Xilinx sono ampiamente documentate [1]. Per una descrizione approfondita della specifica AXI4 è possibile consultare il documento ufficiale rilasciato da AMBA [2].

### AXI4 Full

In questa sezione verrà fornita una descrizione sintetica della specifica del protocollo AXI Full. Per approfondimenti è possibile consultare la specifica ufficiale rilasciata da AMBA [2].

### Architettura di AXI4

AXI4, nella versione Full e Lite, prevede un'architettura semplice nel caso due soli dispositivi vengano interconnessi. L'architettura prevede di assegnare due ruoli distinti ai due dispositivi: il ruolo di Master e il ruolo di Slave. Il dispositivo Master è il dispositivo che avvia le transazioni di *lettura* e *scrittura*, mentre il dispositivo Slave è obbligato a rispondere a queste transazioni. In Figura 1 è rappresentata la vista strutturale dell'architettura AXI4 con un Master e uno Slave.

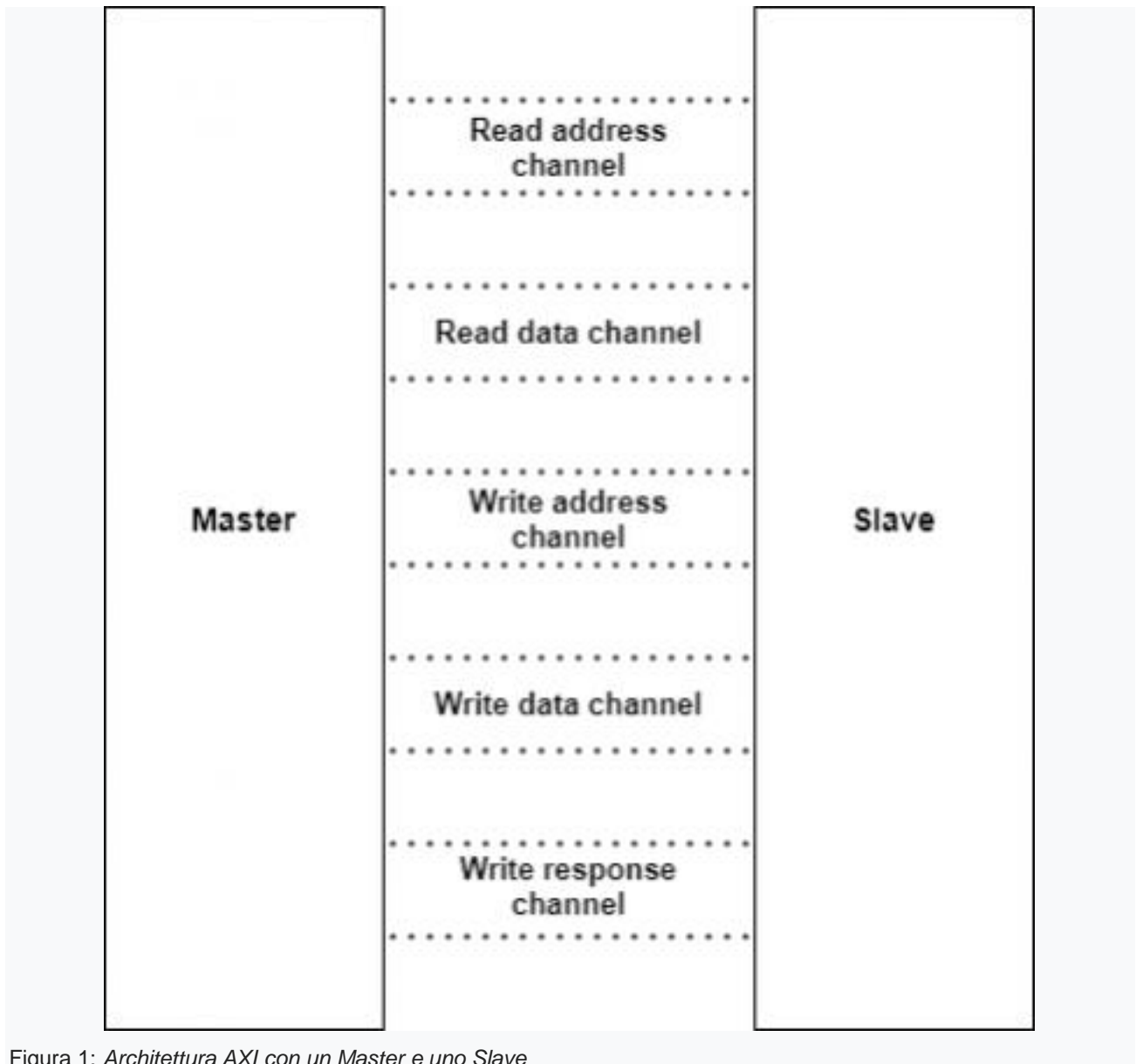


Figura 1: Architettura AXI con un Master e uno Slave

In questo modello sono catturati i cinque canali descritti dalla specifica ufficiale. Questi cinque canali saranno parzialmente trattati nelle successive sezioni.

L'architettura AXI4 si complica notevolmente quando c'è la necessità di interconnettere più Master e più Slave in configurazioni variabili del tipo 1 a N, N a 1 oppure N a M. In questo caso sarà necessario prevedere una interconnessione più complicata che consenta di poter veicolare la comunicazione tra un Master e uno specifico Slave. In genere, tuttavia, le interconnessioni non hanno solamente la capacità di veicolare la comunicazione allo Slave corretto. Le interconnessioni hanno le responsabilità, ad esempio, di adattare interfacce più "larghe" a interfacce più "strette" in termini di linee dato, oppure di adattare la frequenza di funzionamento dei due dispositivi comunicanti. In Figura 2 è rappresentata la vista strutturale dell'architettura AXI4 con più Master e più Slave.

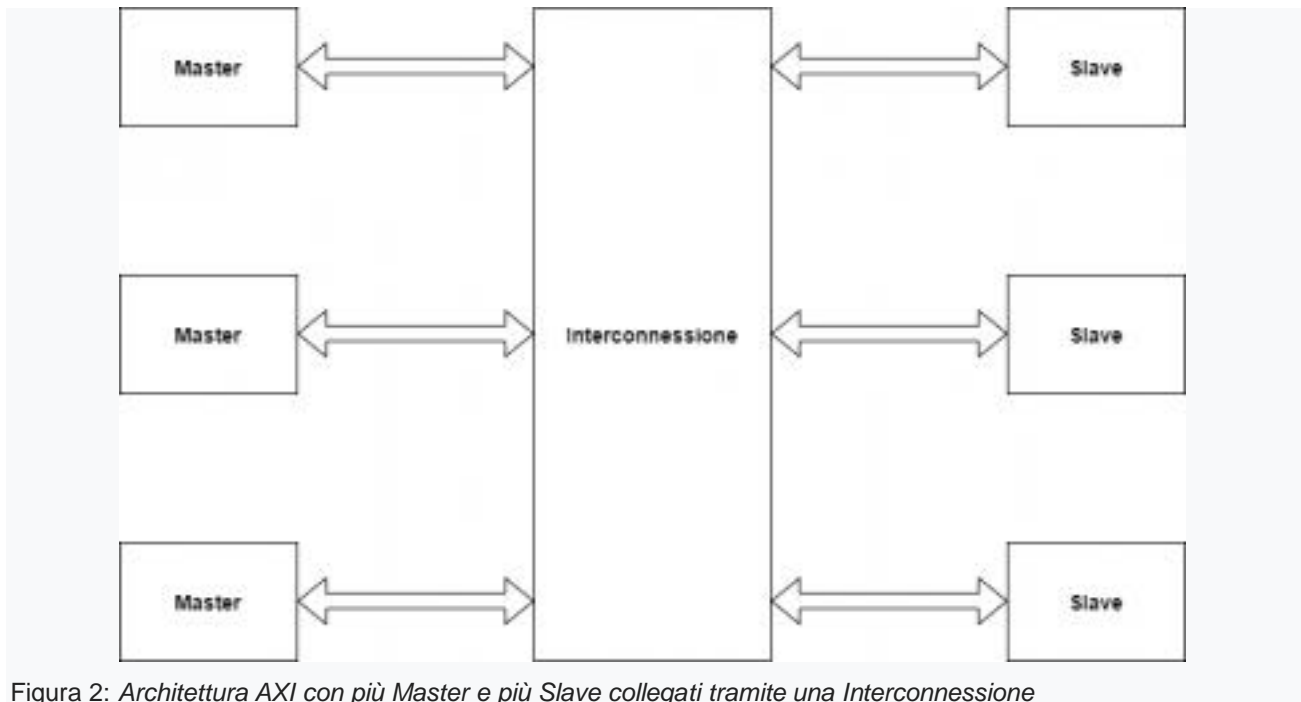


Figura 2: Architettura AXI con più Master e più Slave collegati tramite una Interconnessione

Il modo in cui la interconnessione è implementata dipende dal particolare IP Core utilizzato. Xilinx mette a disposizione diverse implementazioni della interconnessione, per ciascuna delle configurazioni (1 a N, N a 1, N a M). Per configurazioni semplici (1 a N e N a 1), la logica di interconnessione risulta essere piuttosto semplice. Per configurazioni più complicate (N a M), Xilinx mette a disposizione implementazioni che consentono, laddove possibile, la comunicazione parallela tra più dispositivi Master e Slave, mediante un'interconnessione Crossbar. Per maggiori dettagli, consultare la specifica [1].

### Canali di comunicazione

Il protocollo AXI4 si articola in una comunicazione basata su scambio di informazioni su cinque canali logicamente separabili. A ogni canale è associato un gruppo di fili per la comunicazione di informazioni di controllo, di indirizzi, e di dati. I canali sono cinque:

- Read address channel
- Read data channel
- Write address channel
- Write data channel
- Write response channel

Ciascun canale istanzia la comunicazione pilotando una serie di segnali di controllo a cui seguono, possibilmente, informazioni a riguardo di indirizzi e/o dati. La comunicazione è sempre istanziata tra una coppia di Master e Slave. Ciascuno dei segnali dei canali viene pilotato o dal Master o dallo Slave. Tuttavia, se un segnale è pilotato dal Master, esso non può essere pilotato dallo Slave. Descrivere in dettaglio ciascun canale non è l'obiettivo di questo articolo. Tuttavia è bene notare che AXI4 Full specifica la possibilità di poter aumentare il throughput del trasferimento dati implementando la modalità *burst*. Questa modalità prevede, per una singola transazione, di poter trasferire più dati a fronte della specifica di un singolo indirizzo.

### Transazione di lettura

Verrà esplorata, a titolo di esempio, una transazione di lettura istanziata tra un Master e uno Slave in un'architettura 1 a 1. Una transazione di lettura, così come di scrittura, prevede di effettuare un handshaking sia per la lettura dell'indirizzo, sia per la lettura dei dati. Per quanto riguarda l'handshaking per la lettura dell'indirizzo, verranno usati i segnali appartenenti al canale Read address. I segnali di questo canale sono molti, ed esplorarli tutti vorrebbe dire avere una vista completa di tutta la specifica del protocollo. Per quanto interesserà la trattazione a seguire, considereremo solamente i segnali

ARADDR, ARLEN, ARSIZE, ARBURST, ARVALID, ARREADY. L'unico segnale a essere pilotato dallo Slave è il segnale ARREADY, mentre gli altri sono pilotati dal Master. Per quanto riguarda, invece, l'handshaking per la lettura dei dati, verranno usati i segnali appartenenti al canale Read data. I segnali di interesse di questo canale sono RDATA, RRESP, RVALID, RREADY e RLAST. L'unico segnale a essere pilotato dal Master è RREAD, mentre gli altri sono pilotati dallo Slave. Prima di presentare la dinamica dei segnali in una possibile transazione di lettura, in Figura 3 è recuperato dalla specifica AMBA [2] e qui ripresentato un diagramma che modella le dipendenze causali tra i segnali di una transazione di lettura.

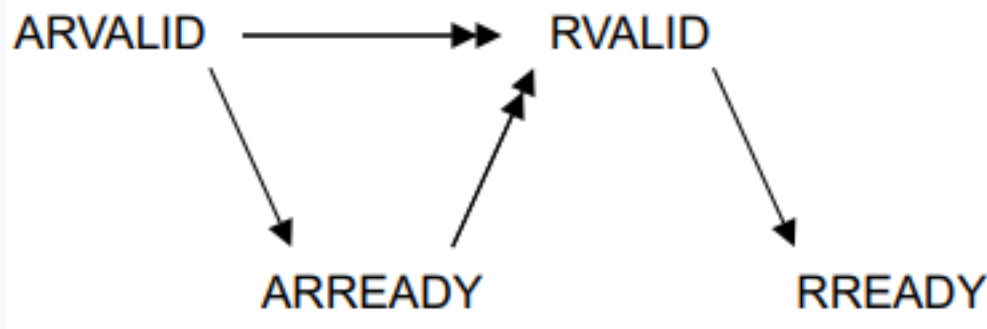


Figura 3: Dipendenze causali tra i segnali di AXI4

Questo diagramma specifica che ARVALID può precedere ARREADY (ma non viceversa). Tuttavia ARVALID e ARREADY *devono* precedere RVALID. Infine, RVALID può precedere RREADY, ma non viceversa. In Figura 4 è riportato il diagramma temporale che descrive come i segnali di AXI4 Full evolvono in una tipica transazione di lettura

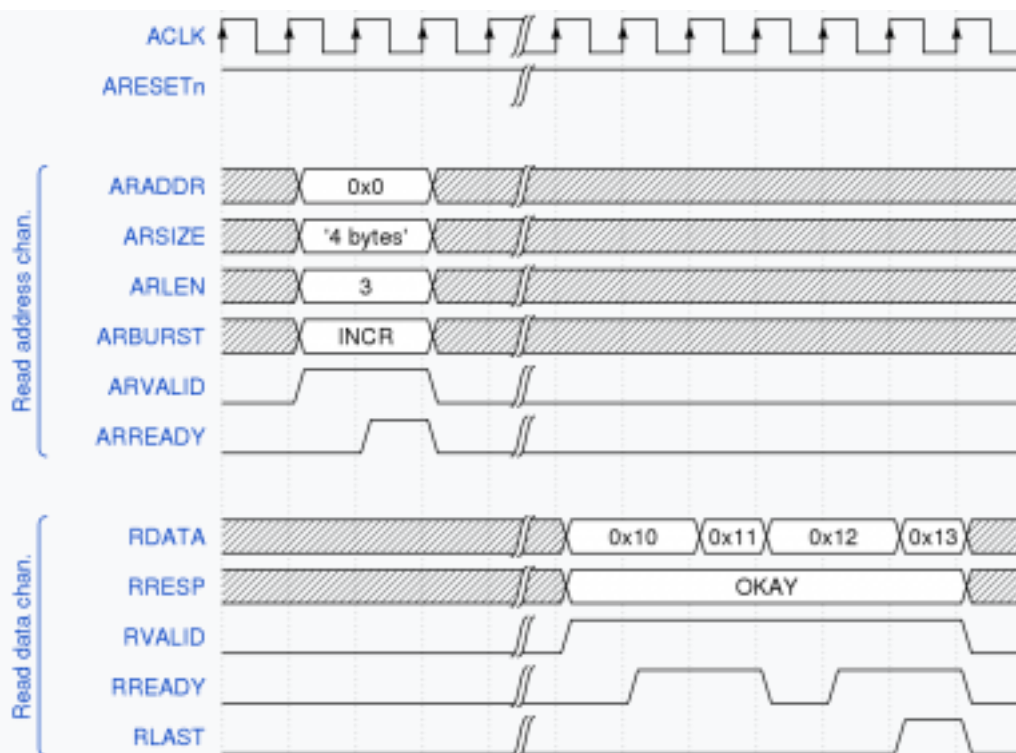


Figura 4: Evoluzione temporale dei segnali di AXI4 Full durante una transazione di lettura

Da notare che l'evoluzione dei segnali di handshaking rispetta quanto specificato dal diagramma delle dipendenze causali in Figura 3. Interessante è anche notare che per AXI4 Full vengono specificati, nella fase di lettura dell'indirizzo, informazioni a riguardo della transazione stessa, come il numero di trasferimenti per burst (ARLEN), la dimensione delle parole (ARSIZE) e la tipologia di burst (ARBURST).

### Tipologia di burst

Sia nelle transazioni di lettura che di scrittura, AXI4 Full consente di specificare la tipologia di burst tramite il segnale ARBURST. Questo segnale codifica in binario la tipologia di burst selezionata. In Figura 4 è presentata una descrizione sintetica delle possibili tipologie di burst adottabili in una transazione di lettura.

AxBURST[1:0]	Burst type
0b00	FIXED
0b01	INCR
0b10	WRAP
0b11	Reserved

Figura 5: Codifica delle tipologie di Burst

La modalità di interesse è la modalità INCR. Questa modalità consente di specificare che gli indirizzi devono essere incrementati durante una transazione AXI4 Full in burst mode per la lettura, o scrittura, di dati da/su indirizzi successivi.

### AXI4 Lite

La versione Lite di AXI4 non differisce molto, in termini di architettura e tipologie di comunicazione, da AXI4 Full. Come il nome suggerisce, questa versione di AXI4 è una versione più leggera della versione Full, necessitando di meno logica per la gestione delle transazioni di lettura e scrittura. Ad esempio, AXI4 Lite non prevede la possibilità di poter avere transazioni in burst-mode. Nella specifica ufficiale AMBA [2], i progettisti si concentrano per lo più sulla conversione protocollare tra AXI4 Full, ACE, Stream ecc. e AXI4 Lite.

## Applicazione SHA256

L'obiettivo di questa trattazione è quella di presentare i risultati ottenuti e il flusso di sviluppo relativo ad una periferica AXI compliant, sia nella versione Full che in quella Lite. Al fine di mettere in luce la differenza tra le due implementazioni, si è cercato di realizzare una periferica il cui funzionamento dipendesse da una notevole quantità di dati in ingresso, e la cui accelerazione potesse essere ragionevole.

La scelta è ricaduta quindi sugli algoritmi di Hashing, ed in particolare sullo sviluppo di una periferica che accelerasse parte dell'algoritmo di SHA256. In questa sezione approfondiremo quindi le basi teoriche che si celano dietro l'hashing e l'algoritmo implementato, passando poi per una architettura di alto livello sia software che hardware.

### Preambolo Teorico

Nel mondo della crittografia e della sicurezza, fondamentale è il concetto di **Funzione crittografica di Hash**, ovvero una classe di funzioni che data una stringa di lunghezza arbitraria, che definiremo *messaggio*, genera una stringa di lunghezza fissa definita valore di Hash, o *digest*. Le funzioni di hash sono progettate in maniera tale da non essere invertibili, tali per cui sia quindi molto semplice passare da un Messaggio ad un Digest, ma sia impossibile ricostruire il messaggio a partire dal Digest.

In linea di principio l'unico modo di ricostruire il messaggio è quello di tentare ricerche a forza bruta. L'algoritmo di hashing deve essere infatti deterministico, in modo tale che un messaggio passato per una funzione di hash ritorni sempre lo stesso valore.

Dato un alfabeto Alpha, comune a tutti gli insiemi, sia M l'insieme di tutti i possibili messaggi di lunghezza variabile e sia D l'insieme di tutti i possibili digest a lunghezza fissata. Sia  $|m|$  la cardinalità del messaggio, e quindi K il sottoinsieme di M contenente tutti i messaggi di dimensione  $|m|$ . se  $|d|$  è la cardinalità del digest, appartenente allo spazio dei digest D, fissa, può verificarsi la circostanza per cui  $|K| > |D|$ . in questo caso il tentativo di mappaggio tra uno spazio a dimensione maggiore verso uno a

dimensione minore, può creare fenomeni di aliasing, tali per cui due stringhe differenti appartenenti allo spazio dei messaggi  $K$ , vadano a generare uno stesso digest appartenente allo spazio  $D$ . In questo caso si parla di Collisioni, che sono il vero tallone d'achille dell'hashing.

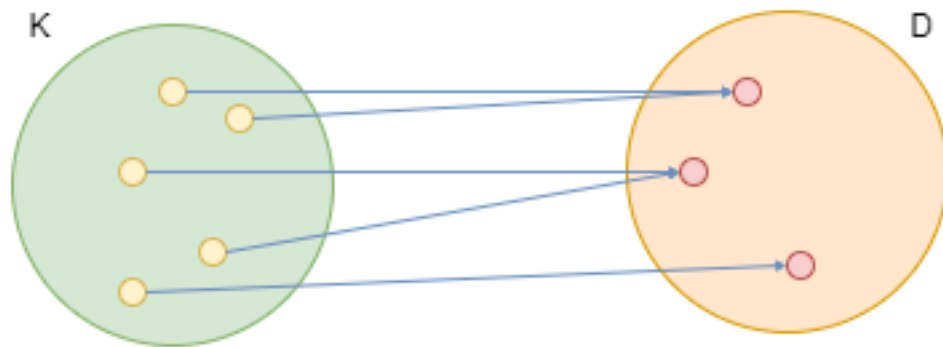


Figura 6: Associazione tra l'insieme dei Messaggi e l'insieme dei Digest

Data però questa natura di compattazione, l'hashing è anche applicato per realizzare strutture di ricerca efficienti, come le *tabelle di Hash*. Non solo, come già accennato, le funzioni di hash sono ampiamente utilizzate in ambito sicurezza e anche criptovalute. La principale funzionalità dell'hashing è relativa alla integrità;

Ipotizziamo che un nodo A voglia comunicare con un nodo B attraverso un canale non sicuro. In questo scenario A invia un messaggio  $M_A$  verso B, e decide di inviarlo in chiaro. Un nodo malevolo C potrebbe intercettare tale comunicazione e modificare il messaggio  $M_A$ . Tuttavia se A decide di applicare una funzione di hash, può concatenare il digest ottenuto  $D_A$  al messaggio originale. In questa maniera qualunque manipolazione effettuata da C sul messaggio originale, porterebbe B a decidere che il messaggio sia stato modificato, in quanto l'hashing di  $M_C$  non coinciderebbe con  $D_A$ . Chiaramente esistono scenari più completi che garantiscano anche Confidenzialità ed Autenticazione

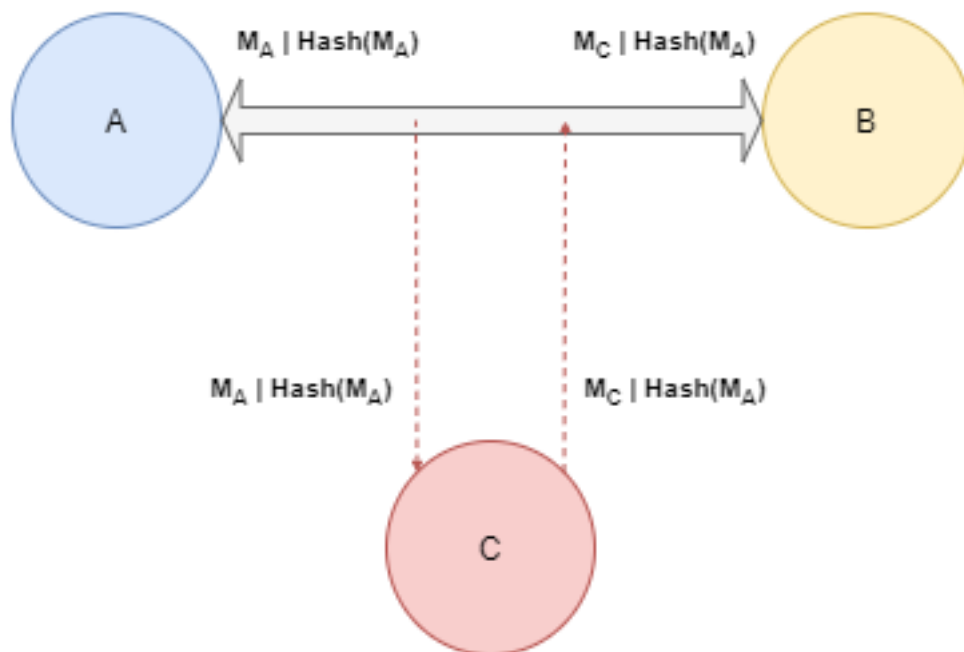


Figura 7: Esempio di protezione d'integrità nella comunicazione non sicura

## SHA256

Arrivati a questo punto diventa imperativo decidere quale funzione di hash implementare; nel nostro caso abbiamo scelto di concentrarci sulla famiglia di **Secure Hashing Algorithm – SHA**, ed in particolare sulla versione due dell'algoritmo.

SHA-256 appartiene infatti alla seconda iterazione dell'algoritmo, e dato un messaggio di dimensione arbitraria è in grado di produrre un digest a 256 bit. Tale algoritmo presenta anche una versione con digest a 512 bit, e al giorno d'oggi anche la famiglia SHA3 è disponibile.

L'algoritmo fa uso di parole a 32 bit ed una serie di operazioni di rotazioni e di addizioni di costanti. Poiché nella nostra applicazione abbiamo deciso di implementare questo algoritmo, vale la pena analizzarne gli step in maniera concisa.

L'algoritmo fa anzitutto uso di due **Matrici di Costanti**, che chiameremo H e K. sia X un sottoinsieme discreto dei numeri Naturali con cardinalità  $2^{32}$ , X incluso in  $\mathbb{R} : |X| = 2^{32}$ , allora H appartiene all'insieme  $X^8$ , definendo quindi un vettore di 8 elementi, mentre K appartiene  $X^{64}$ .

Per quanto ne concerne i valori di questi due vettori, H rappresenta i primi 32 bit della parte frazionaria della radice quadrata dei primi 8 numeri primi. E dunque il valore H(0) conterrà la parte frazionaria della radice di due, H(1) quella della radice di tre e così via. Discorso analogo lo si applica per K, dove però il generico elemento K(i) conterrà i primi 32 bit della parte frazionaria della radice cubica dell'i-esimo numero primo. (da 2 a 311).

```
h0 := 0x6a09e667
h1 := 0xbb67ae85
h2 := 0x3c6ef372
h3 := 0xa54ff53a
h4 := 0x510e527f
h5 := 0x9b05688c
h6 := 0x1f83d9ab
h7 := 0x5be0cd19
k[0..63] :=
    0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5, 0x3956c25b, 0x59f111f1,
    0x923f82a4, 0xab1c5ed5,
    0xd807aa98, 0x12835b01, 0x243185be, 0x550c7dc3, 0x72be5d74, 0x80deb1fe,
    0x9bdc06a7, 0xc19bf174,
    0xe49b69c1, 0xefbe4786, 0x0fc19dc6, 0x240ca1cc, 0x2de92c6f, 0x4a7484aa,
    0x5cb0a9dc, 0x76f988da,
    0x983e5152, 0xa831c66d, 0xb00327c8, 0xbf597fc7, 0xc6e00bf3, 0xd5a79147,
    0x06ca6351, 0x14292967,
    0x27b70a85, 0x2e1b2138, 0x4d2c6dfc, 0x53380d13, 0x650a7354, 0x766a0abb,
    0x81c2c92e, 0x92722c85,
    0xa2bfe8a1, 0xa81a664b, 0xc24b8b70, 0xc76c51a3, 0xd192e819, 0xd6990624,
    0xf40e3585, 0x106aa070,
    0x19a4c116, 0x1e376c08, 0x2748774c, 0x34b0bcb5, 0x391c0cb3, 0x4ed8aa4a,
    0x5b9cca4f, 0x682e6ff3,
    0x748f82ee, 0x78a5636f, 0x84c87814, 0x8cc70208, 0x90befffa, 0xa4506ceb,
    0xbef9a3f7, 0xc67178f2
```

A questo punto, dato il *messaggio*, l'algoritmo procede per i seguenti passaggi:

- Viene effettuato l'append di un bit 1 al messaggio, e poi m bits pari a 0, dove m è il minimo numero maggiore di zero tale per cui la lunghezza del messaggio, in bits, diventi pari a 448 modulo 512. Alla fine viene aggiunta la lunghezza in bit del messaggio originale sugli ultimi 64 bit, in codifica big-endian.
- Il messaggio viene spezzato in chunk da 512 bit se necessario, e ogni chunk viene diviso in 16 parole da 32 bit, che chiameremo W.
- Per ogni Chunk, le 16 parole vengono estese a 64 con degli zeri; successivamente gli elementi da 16 a 63 sono calcolati con precisione per mezzo di rotazioni, shift, xor e somme, così come di seguito:

```
for i from 16 to 63
```



```

    s0 := (w[i-15] rightrotate 7) xor (w[i-15] rightrotate 18) xor (w[i-15]
rightshift 3)
    s1 := (w[i-2] rightrotate 17) xor (w[i-2] rightrotate 19) xor (w[i-2]
rightshift 10)
    w[i] := w[i-16] + s0 + w[i-7] + s1

```

- L'algoritmo passa nella fase di compressione del chunk, andando ad effettuare un'altra serie di somme, xor, and e rotazioni, prelevando valori dal vettore H, dal vettore K e dal messaggio W :

```

for i from 0 to 63
    s0 := (a rightrotate 2) xor (a rightrotate 13) xor (a rightrotate 22)
    maj := (a and b) xor (a and c) xor (b and c)
    t2 := s0 + maj
    s1 := (e rightrotate 6) xor (e rightrotate 11) xor (e rightrotate 25)
    ch := (e and f) xor ((not e) and g)
    t1 := h + s1 + ch + k[i] + w[i]

```

- Il digest viene prodotto come l'append dei valori h, calcolati e sovrascritti per ogni Chunk :

```

digest = hash = h0 append h1 append h2 append h3 append h4 append h5 append h6
append h7

```

## Ratio Applicazione

Come descritto nella sezione precedente, una delle principali differenze tra AXI full e AXI lite sta nel fatto che il primo implementa trasferimenti in burst, mentre il secondo no. Al fine di poter notare delle differenze tra i due protocolli, si è scelto di implementare una periferica in hardware che potesse realizzare almeno la parte di compressione dell'algoritmo prima citato.

Infatti, per quanto detto, è vero che l'intero algoritmo prenda in carico una stringa, definita messaggio, di dimensione arbitraria, ma il Chunk prodotto è sempre di dimensione pari a 256Byte. Lavorando quindi a questo livello, una periferica che implementi l'algoritmo di compressione può essere tranquillamente adattata per prelevare il Chunk sia in 64 singole letture, che in un solo burst da 64 letture da 32 bit, oppure in due burst da 32 letture da 32 bit, e così via.

Non solo, anche in lettura è possibile applicare il burst per poter leggere gli 8 byte del Digest. Ciò naturalmente non velocizza l'algoritmo di per se, che comunque viene accelerato nelle funzioni che altrimenti verrebbero realizzate sequenzialmente da un processore general purpose, ma fornisce delle migliorie solo nell'interfacciamento con la memoria, sia in lettura che in scrittura.

L'obiettivo che ci siamo posti per questo caso di studio è stato quindi quello di descrivere una periferica custom che svolgesse l'algoritmo di compressione e che fosse riutilizzabile con le due differenti interfacce AXI full e lite; successivamente programmare un Driver in linguaggio C per implementare l'intero algoritmo, e usufruire della periferica, con il suo modello di programmazione, per giungere al Digest.

## Architettura Software / Hardware

Fatto il preambolo teorico e descritto il concept di base dell'applicazione, spendiamo ora due parole sull'architettura ad alto livello del sistema, partendo dall'Hardware e arrivando fino al Software.

A livello Hardware la periferica è stata progettata in VHDL e implementata sul PL di una scheda Zybo. La periferica necessita esporre una interfaccia AXI, sia essa Lite o Full, con una certa memoria interna alla interfaccia AXI stessa (BRAM per AXI Full e Registri per AXI Lite). La periferica di per se espone una interfaccia verso l'AXI che gli permetta di interagire in maniera minimale con la memoria, così da poter leggere il messaggio W quando pronto e scrivere il Digest alla fine dell'esecuzione.



Internamente la periferica si compone di una parte di Memoria, una unità operativa ed una unità di controllo. La prima si compone di due ROM, una contenente il vettore H, e che quindi chiameremo ROM-H, di 32 byte, ed una contenente il vettore K, che quindi chiameremo ROM-K, di dimensione 256 Byte,

L'unità di controllo genera i segnali di controllo necessari alla manipolazione dell'unità operativa, che invece si occupa di realizzare nella pratica l'algoritmo di compressione, prevedendo quindi una serie di operazioni combinatorie, scorrimenti, somme e assegnazioni.

A livello Software invece, il core A9 è programmato per eseguire un layer software strutturato a 3 livelli. Il livello 1 comprende un Driver per la periferica, che implementa delle API per accedere in lettura ed in scrittura alla periferica stessa, astratta per mezzo di una struttura e delle MACRO. Tale livello funge da Hardware Abstraction Layer, e consente l'interfacciamento trasparente con il compressore. Al livello immediatamente più alto viene implementato l'intero algoritmo SHA256, che prevederà un insieme di funzionalità atte a realizzare l'algoritmo, che farà poi uso delle API dell'HAL per completare l'algoritmo.

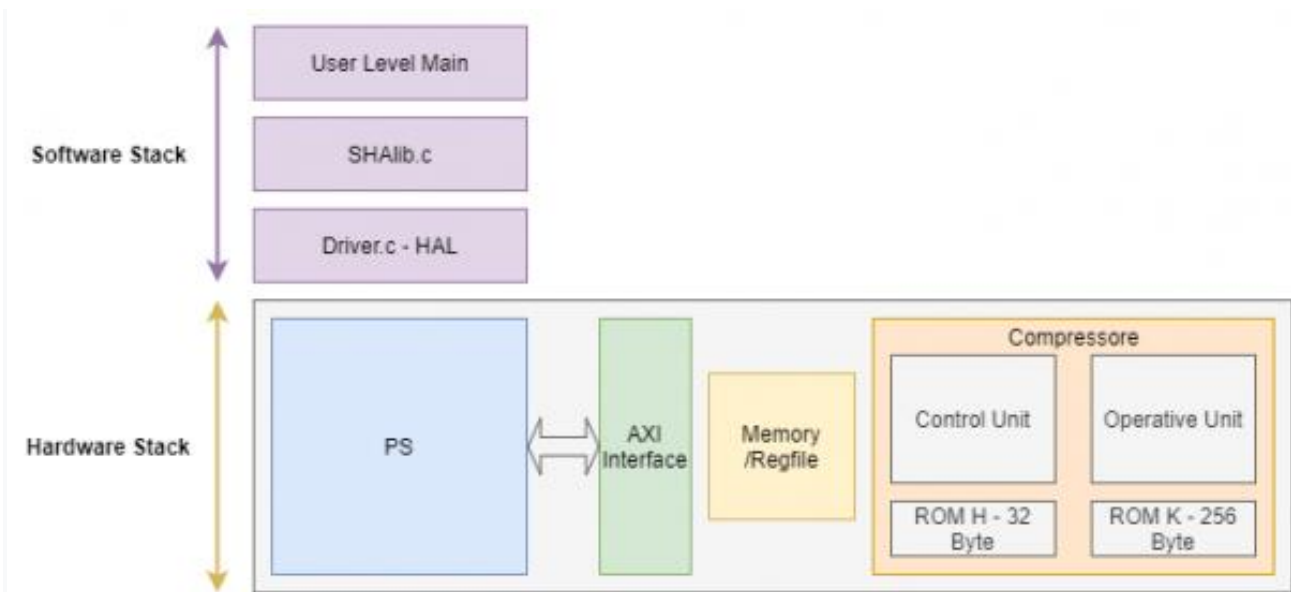


Figura 8: Architettura ad alto livello del sistema Software/Hardware

## Architettura Periferica

In questa sezione discuteremo nel dettaglio dell'architettura della periferica, iniziando con una descrizione strutturale dei componenti utilizzati, passando poi per una chiarificazione della mappa di memoria così come vista dalla periferica e così come vista dal PS, ed infine discuteremo degli aspetti comportamentali del dispositivo, fornendo una visione completa del funzionamento della periferica.

### Descrizione Strutturale

Così come visto nell'architettura ad alto livello presentata in precedenza, la periferica si compone di 4 unità fondamentali : La control Unit, l'Operative Unit, o Compressore, la ROM H e la ROM K.

Anzitutto cerchiamo di descrivere in maniera più dettagliata come tali componenti siano collegati tra loro e come si interfaccino verso l'esterno. La periferica espone all'esterno una interfaccia molto minimale, composta da segnali di Dati, come il Data\_in ed il Digest\_out. Il primo viene utilizzato per prelevare dati dalla memoria AXI, il secondo per immetterne : entrambi sono canali a 32 bit ed il loro scopo è relativo alla lettura del messaggio W e alla produzione del Digest.

Al contempo sono presenti dei segnali di controllo; affinché la periferica venga avviata, il segnale di Start deve essere asserito, mentre i segnali di controllo relativi alla memoria sono necessari per implementare un semplice protocollo di lettura/scrittura qualora una transazione AXI non sia in gioco. Tuttavia l'intero sistema è pensato in maniera tale che una volta avviato, nessuna scrittura o lettura dai registri/memoria AXI possa essere effettuata fino alla produzione del Digest.

Internamente invece, i quattro componenti sono collegati tra loro per mezzo di una serie di segnali interni. Il compressore espone delle interfacce verso le ROM K ed H e verso la memoria AXI. I dati da leggere e scrivere esternamente infatti passano per il Compressore, essendo l'unica unità di elaborazione presente. Al contempo, i dati provenienti dalle ROM sono usate dal ciclo di compressione, e devono quindi essere poste in ingresso all'unità operativa.

Da un punto di vista di controllo invece, il compressore espone una interfaccia verso l'unità di controllo, tramite la quale può sapere quando partire tramite un segnale di Start, e quale sia il round attualmente in corso: così facendo il compressore sa se asserire il segnale di End Round a fine di un round, oppure End Compression, qualora i 64 round siano stati completati, così da fornire il digest in uscita sulla memoria AXI. L'unità di controllo è quella che cattura il segnale di Start della periferica, così da dar via alle elaborazioni. Possiede anch'essa una interfaccia di lettura verso la memoria AXI, così da poter leggere il registro di controllo, di cui parleremo a breve. Il segnale di Round viene utilizzato sia per informare il compressore della corrente iterazione, ma anche per indirizzare la ROM K.

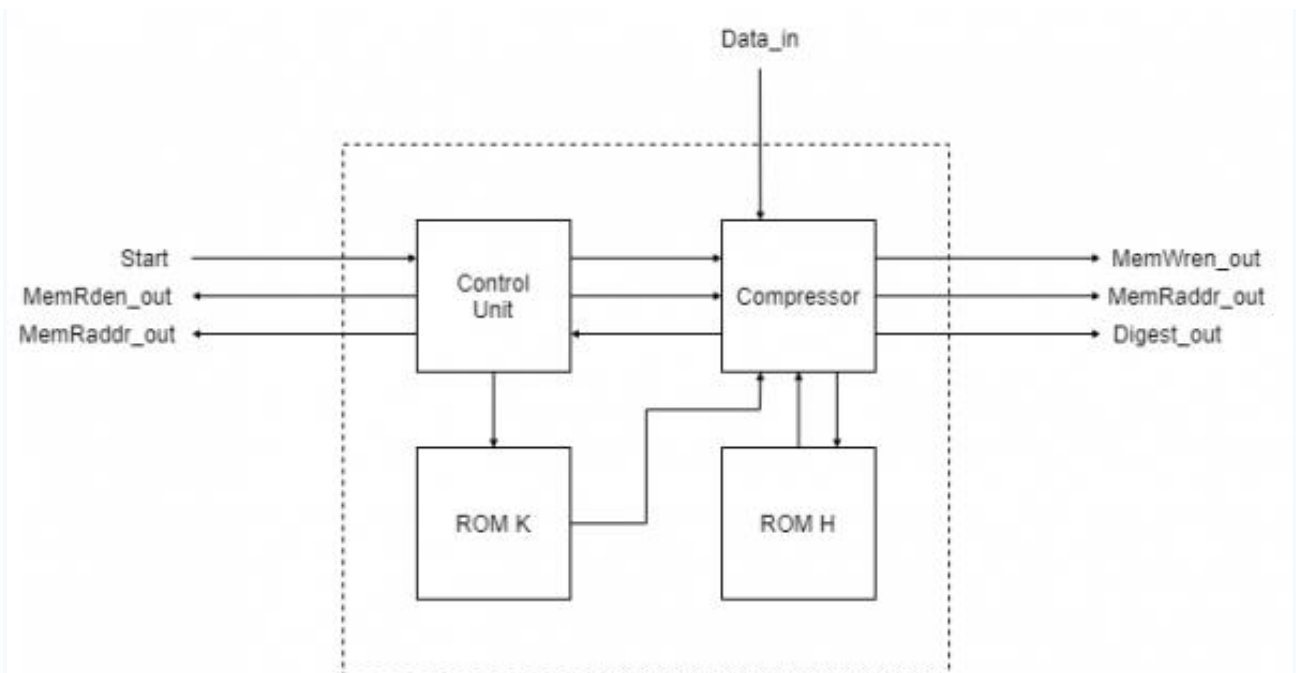


Figura 1: Descrizione strutturale della periferica

## Compressor

Il compressore è un componente abbastanza articolato, ed incapsula la logica operativa della periferica. Il componente prevede al suo interno 3 contatori: il *ReadRomCounter* che consente di leggere sequenzialmente tutte le entry della ROM, l'*AssignCounter* che tiene conto delle assegnazioni sequenziali da fare alla fine di ogni round e il *WriteRamCounter*, che tiene conto delle scritture da fare verso la memoria AXI una volta prodotto il Digest.

Il compressore deve fare inoltre uso di 8 registri ad uso generico e 6 *Rotate Register* per effettuare specifiche operazioni di shifting e rotazione. I primi 8 sono utilizzati per archiviare i valori prelevati dalla ROM, e vengono sovrascritti ad ogni Round, per poi venire alla fine concatenati nella produzione dell'Hash.

I *Rotate register* vengono usati quindi per effettuare le omologhe operazioni sui registri E e A; successivamente vengono utilizzati altri registri d'appoggio S1, S0, CH e NAJ, nei quali vengono mantenuti i risultati prodotti da operazioni logiche tra i registri R e i registri a uso generico. Infine un addizionatore si occupa di sommare i valori prelevati dalla ROM K, il messaggio W relativo alla iterazione i-esima e i registri d'appoggio.

Una macchina a stati interni regola correttamente l'esecuzione e l'attivazione di questi componenti, così che una volta completato il round, l'unità di controllo venga notificata. La compressione comincia quando il Flip Flop Start interno cattura il segnale in arrivo, e termina quando l'ultimo round viene completato, così da attivare la scrittura del Digest in Memoria AXI; il tutto viene coordinato per mezzo dei contatori prima descritti.



L'unità di controllo è il componente del sistema delegato alla generazione dei segnali di controllo per gli altri componenti interni. Non solo si occupa di prelevare il segnale di start dall'interfaccia AXI e abilitare il compressore, ma funge anche da generatore di indirizzi per la ROM K e per la Memoria AXI, durante le operazioni di lettura di W.

## ROM

Le ROM, sia H che K, sono dei componenti puramente combinatori, che dato un indirizzo in ingresso, restituiscono un valore in uscita. Essendo puramente di lettura, e venendo realizzate come delle LUT, sono in grado di fornire l'uscita nello stesso colpo di clock in cui varia l'indirizzo. Internamente la ROM H contiene il vettore H delle radici quadrate dei primi 8 numeri primi descritto nella sezione precedente hardcoded su 32 byte, mentre la ROM K contiene il vettore K delle radici cubiche dei primi 64 numeri primi.

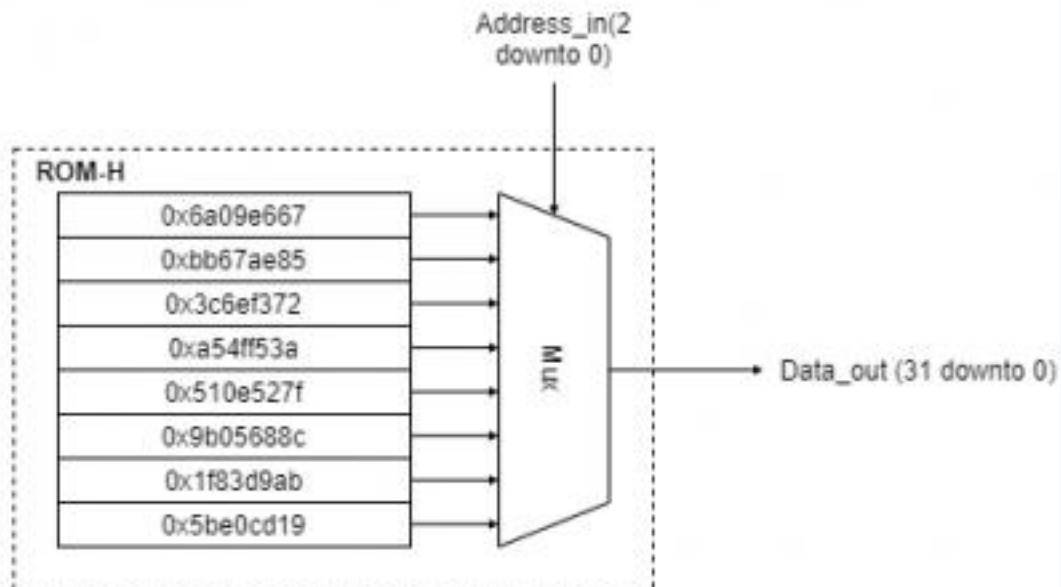


Figura 10: Descrizione strutturale della ROM H

## Mappa di Memoria

In questa sottosezione discuteremo la mappa di memoria adottata per interagire con la periferica tramite Interfaccia AXI. Ricordiamo che le periferiche AXI sono accessibili secondo approccio Memory Mapped, ragion per cui, scrivendo ad uno specifico indirizzo (0x7AA00000 per AXI FULL e 0x43C00000 per AXI LITE) è possibile accedere al primo registro, o alla prima locazione di memoria, previsto per la periferica.

AXI Lite e AXI Full, così come già spiegato in precedenza, si differenziano anche per la scelta di memoria adottata. AXI Full implementa delle Block Ram, con un loro protocollo d'accesso, che ha richiesto l'introduzione di specifici segnali per indirizzare e abilitare questa memoria lato periferica. Dall'altro lato, AXI Lite implementa un semplice Regfile; la mappa di memoria che presentiamo è applicabile ad entrambe le varianti, in quanto fa riferimento unicamente a degli offset rispetto ad un indirizzo base, assegnato diversamente nei due casi.

Facendo riferimento al fatto che ogni parola in memoria sia di 32 bit, e quindi copra 4 indirizzi, i primi  $64 \cdot 4 = 256$  indirizzi della memoria vengono utilizzati per archiviare il messaggio esteso W. Notiamo che mentre AXI veda alla granularità del Byte, e quindi debba moltiplicare gli indirizzi per 4, la periferica ha la visibilità a livello di Word, e quindi internamente si riferirà a questi indirizzi nel range 0...63 e non 0...255.

A seguire, le successive 8 Locazioni sono dedicate al Digest, scritto da parte del Compressore e letto unicamente dal PS; ne consegue che questa regione di memoria sarà di sola lettura a parte del PS, ma di lettura e scrittura per la periferica.

All'indirizzo 72 è presente il registro di controllo, il cui unico scopo è quello di segnalare l'inizio di una compressione scrivendovi un qualunque valore, mentre all'indirizzo 73 ritroviamo il registro di stato, che semplicemente tornerà un valore diverso da zero fintanto che la compressione non sarà terminata, inibendo la scrittura di un nuovo messaggio nei registri dedicati a W.

Un modello di programmazione derivante quindi imporrebbe di scrivere, a livello software, tutte le 64 parole ai primi indirizzi, effettuare poi una scrittura qualsiasi sul registro di controllo, e porsi infine in polling sul registro di stato. Una volta variato il valore interno a tale registro, si potrà andare effettivamente a leggere il dato dalle locazioni dedicate al Digest, per poi avere di nuovo la possibilità di scrivere un altro chunk.

Vale la pena citare anche l'indirizzo 74, il Version Register, utilizzato per lo più come debug, e contenente la versione attuale della periferica. Tutti gli indirizzi da 75 fino a 65535 sono riservati, ma ai fini del testing sono stati lasciati come si lettura e scrittura, così da poter testare correttamente il funzionamento dell'interfacciamento AXI.

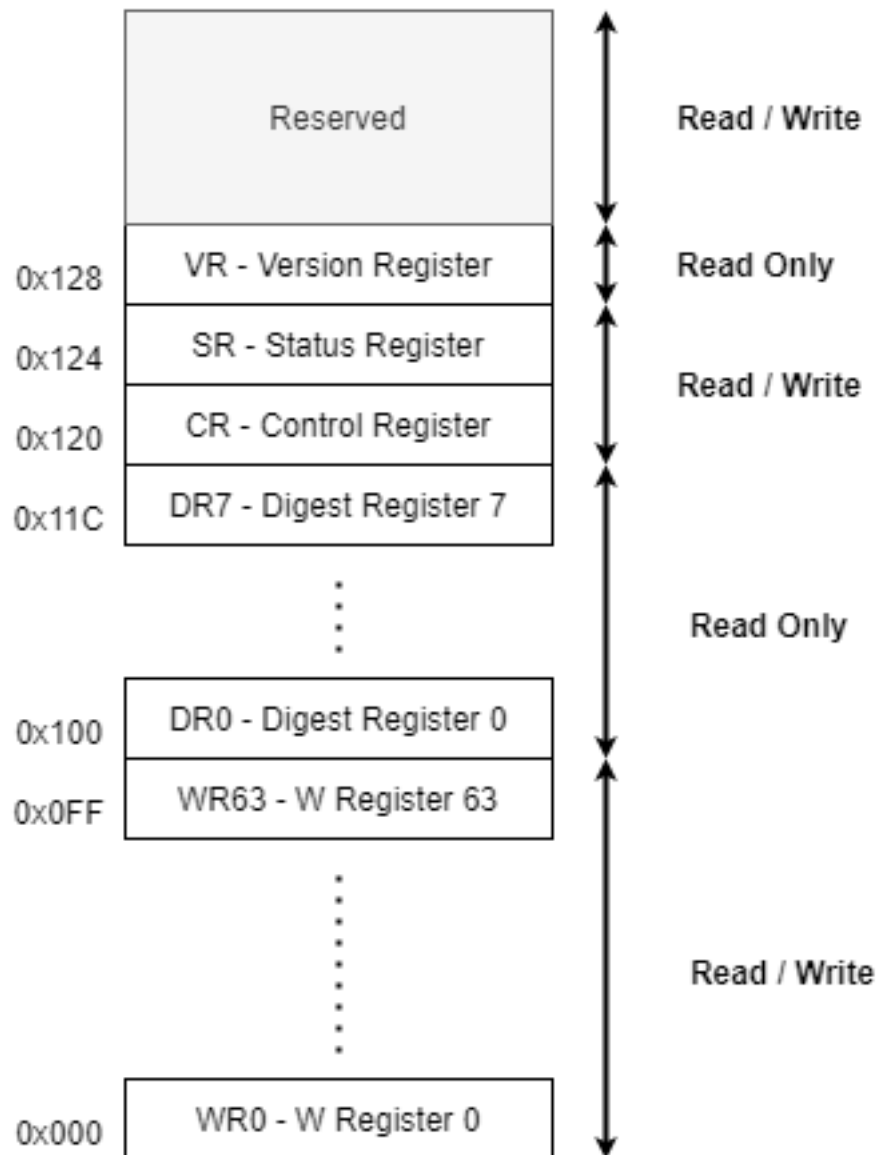


Figura 11: Mappa di memoria della periferica

## Descrizione Comportamentale

La periferica sviluppata dev'essere descritta dal punto di vista comportamentale per poter comprendere come i requisiti funzionali dell'applicazione vengono soddisfatti. In questa descrizione prescindiamo dall'esplorare la dinamica seguita dal protocollo AXI4 nello scambio di informazioni. Faremo bensì ricorso a diagrammi UML di sequenza per la descrizione dei messaggi scambiati dai vari componenti nelle varie fasi previste per l'interlocuzione con la periferica. Le fasi sono tre: Inizializzazione, Esecuzione e Terminazione. Nella fase di Esecuzione verranno forniti anche dei diagrammi a stati per descrivere il comportamento della unità di controllo che coordina la parte operativa della periferica.

### Inizializzazione

La fase di inizializzazione prevede che il Processing System scambi con la periferica, mediante l'uso di un'interfaccia AXI, le informazioni di controllo e i dati necessari a innescare la fase di *compressione*. In Figura 12 è presentato il diagramma UML di sequenza che modella la fase di inizializzazione.

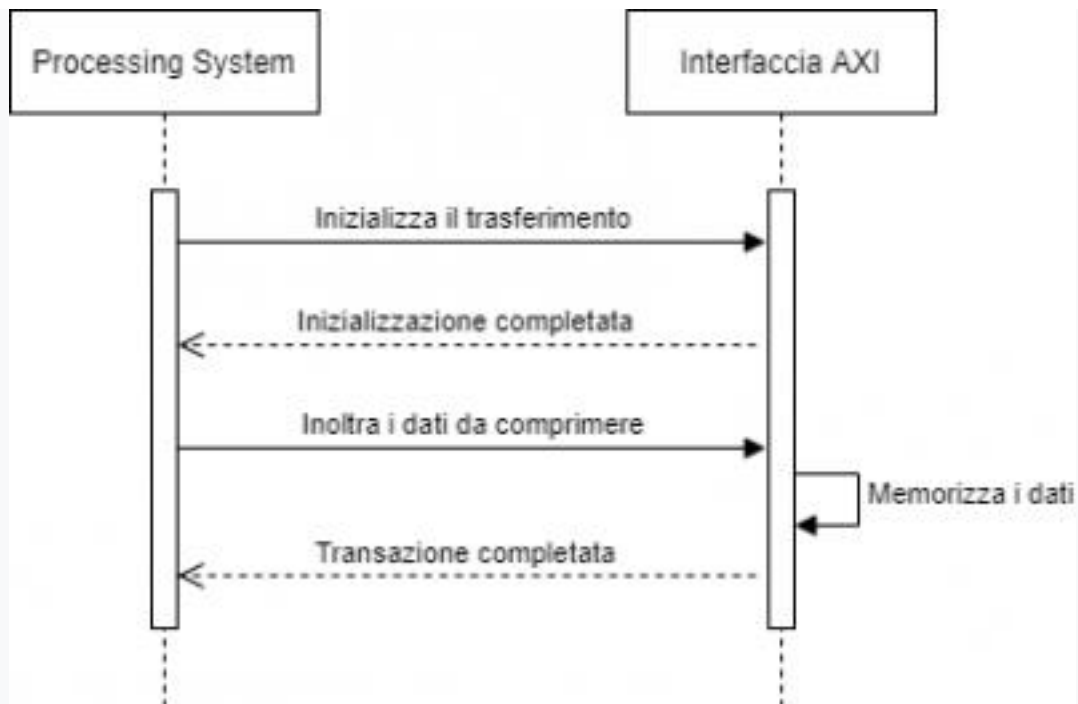


Figura 12: *Diagramma UML di sequenza modellante la fase di Inizializzazione*

Chiaramente, così come è stato esposto nella parte riguardante AXI4 Full e Lite, le fasi di trasferimento indirizzo e dati vengono gestite dalle interfacce AXI Master e Slave del Processing System e della periferica.

### **Esecuzione**

La fase di esecuzione prevede che il Processing System segnali, tramite un apposito registro di controllo, l'avvio della compressione. Quando l'interfaccia rileva questa scrittura, la periferica viene notificata della segnalazione, e avvia l'algoritmo di compressione. In Figura 13 è presentato il diagramma UML di sequenza che modella la fase di esecuzione.

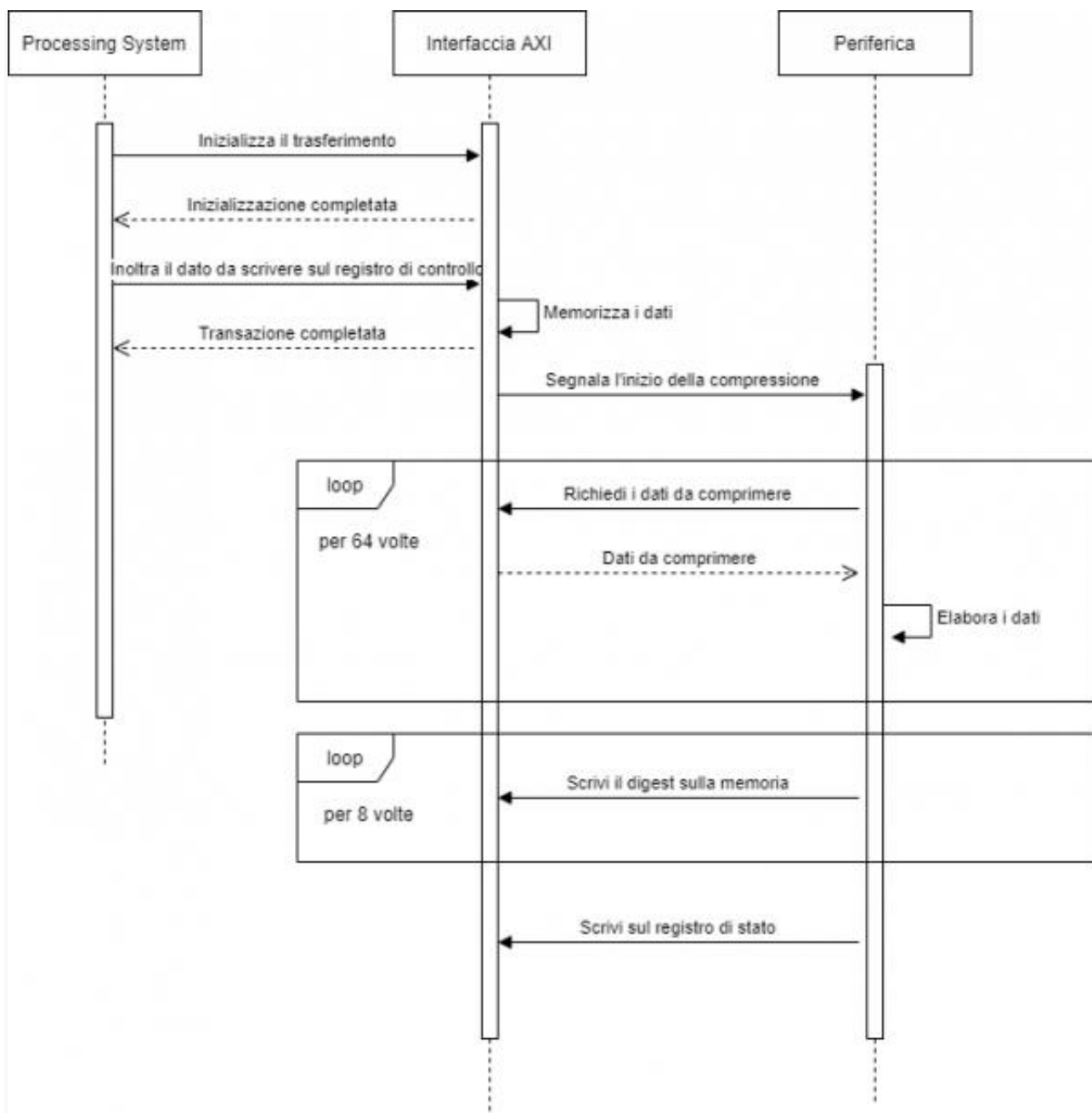


Figura 13: Diagramma UML di sequenza modellante la fase di Esecuzione

L'evoluzione comportamentale dei componenti presenti all'interno dell'Interfaccia AXI e della Periferica è gestita da delle unità di controllo che supervisionano il processo. Saranno tali unità di controllo a gestire il trasferimento coordinato dei dati da elaborare verso la periferica, ma anche per l'accettazione del *digest*, che è, in ultima istanza, il risultato dell'intero algoritmo SHA-256. L'unità di controllo dell'Interfaccia AXI gestisce l'inoltro verso la Periferica dei dati da elaborare nei 64 Round da svolgere. I Round sono 64 poichè la dimensione dei dati da elaborare è pari a 256 Byte, che sono, di fatto, 64 parole da 32 bit. Al termine del sessantaquattresimo Round, mediante la segnalazione di stato della Periferica, l'unità di controllo entrerà nello stato idle. In Figura 14 è rappresentato il diagramma degli stati che modella il comportamento dell'unità di controllo dell'Interfaccia AXI.



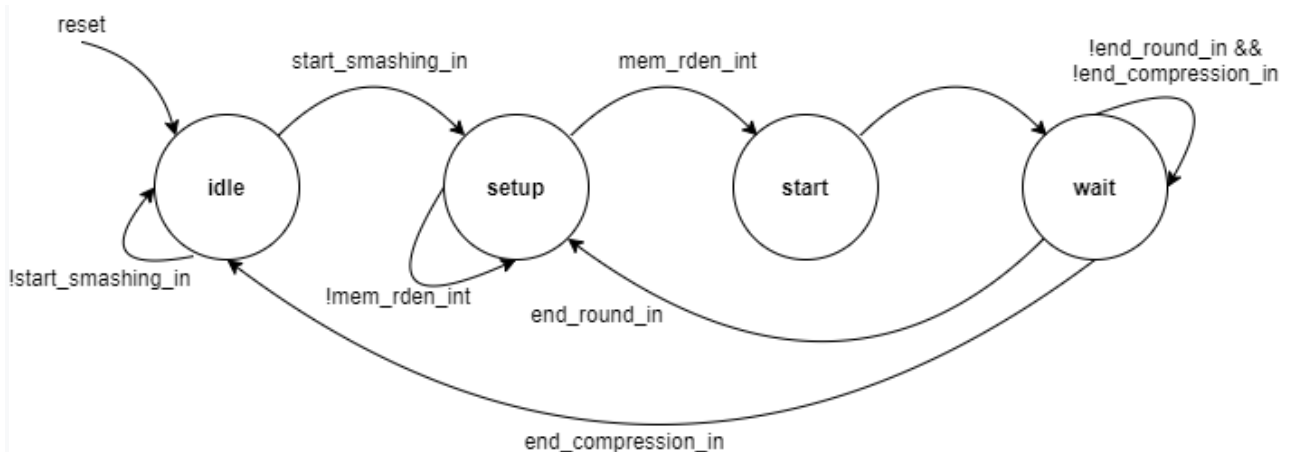


Figura 14: Diagramma di stato modellante la unità di controllo dell'Interfaccia AXI

In VHDL l'unità di controllo è stata descritta con la seguente interfaccia:

```

entity ControlUnit is
Port (
    clk : in std_logic;
    reset : in std_logic;

    start_smashing_in : in std_logic;
    end_round_in : in std_logic;
    end_compression_in : in std_logic;

    Mem_rden_out : out std_logic;

    Mem_rAddress_out : out std_logic_vector(6 downto 0);

    start_compression_out : out std_logic;
    round_out : out std_logic_vector(5 downto 0)
);
end ControlUnit;
  
```

Il corpo prevede la descrizione diretta della macchina a stati precedentemente descritta:

```

architecture Behavioral of ControlUnit is
    type Control_State is (Idle,Setup,Start,Waiting); -- Define the states
    signal State : Control_State;
    signal round_int : std_logic_vector(5 downto 0);
    signal start_compression_int : std_logic;
    signal Mem_wren_int : std_logic;
    signal Mem_rden_int : std_logic;
    signal Mem_rAddress_int : std_logic_vector(6 downto 0);
begin
    round_out <= round_int;
    start_compression_out <= start_compression_int;
    Mem_rden_out <= Mem_rden_int;
    Mem_rAddress_out <= Mem_rAddress_int;
    process(clk)
    begin
        if rising_edge(clk) then
            if reset = '0' then
                --Outputs
                round_int <= (others => '0');
            end if;
        end if;
    end process;
  
```

```

        start_compression_int <= '0';
        Mem_wren_int <= '0';
        Mem_rden_int <= '0';
        Mem_rAddress_int <= (others => '0');
        --State
        State <= Idle;
    else
        case State is
            when Idle =>
                round_int <= "000000";
                if( start_smashing_in = '1' ) then
                    State <= Setup;
                else
                    State <= Idle;
                end if;
            when Setup =>
                --Read AXI mem Protocol
                if( Mem_rden_int = '0' ) then
                    Mem_rden_int <= '1';
                    Mem_rAddress_int(5 downto 0) <= round_int;
                    State <= Setup;
                else
                    Mem_rden_int <= '0';
                    Mem_rAddress_int(5 downto 0) <= "000000";
                    State <= Start;
                end if;
            when Start =>
                start_compression_int <= '1';
                State <= Waiting;
            when Waiting =>
                start_compression_int <= '0';
                if( end_round_in = '1' and end_compression_in = '0' )
then
                    round_int <= round_int + 1;
                    State <= Setup;
                elsif( end_round_in = '0' and end_compression_in = '1' ) then
                    round_int <= "000000";
                    State <= Idle;
                end if;
            when others =>
                State <= Idle;
        end case;
    end if;
end if;
end process;
end Behavioral;

```

L'unità di controllo della Periferica risulta essere più raffinata poichè necessita di coordinare non solo il trasferimento dei dati da comprimere, ma anche la compressione dei dati stessa e anche il trasferimento del digest. A causa di ciò, la macchina a stati prevede di utilizzare svariati Contatori e Flip-Flop interni per scandire i tempi di commutazione da uno stato a un altro. L'unità di controllo della Periferica deve accertarsi, all'inizio di un Round, che i Byte corretti dell'input siano caricati dalla RAM interna all'interfaccia AXI. Oltretutto, deve accertarsi che al primo Round vengano caricate le otto costanti, ciascuna da 4 Byte, da una delle ROM preposte a contenere tali costanti. Queste due condizioni complicano lo stato di "idle" della unità di controllo. A seguire, a valle delle operazioni di somma, ci sono altri stati che prevedono l'utilizzo di contatori per scandire operazioni sequenziali, tra cui "assign" e "finish". In particolare, "finish" è lo stato innescato al termine di tutti i Round, e prevede il caricamento, nella RAM dell'interfaccia AXI, del *digest* da fornire in uscita come risultato ultimo della compressione. In Figura 15 è rappresentato il diagramma a stati dell'unità di controllo della Periferica.

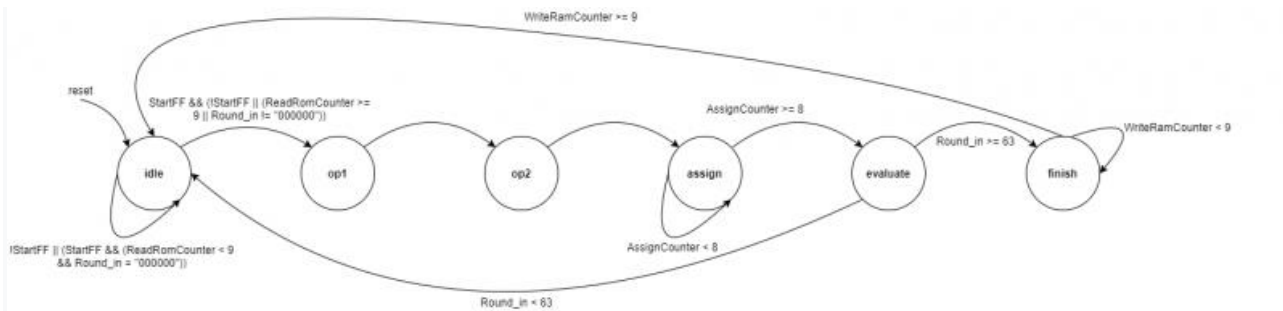


Figura 15: Diagramma di stato modellante la unità di controllo della Periferica

Di seguito l'interfaccia della Periferica (in VHDL nominata Compressore):

```

entity Compressor is
Port (
    clk : in std_logic;
    reset : in std_logic;
    MemH_in : in std_logic_vector(31 downto 0);
    --Mem Interfaces
    MemW_in : in std_logic_vector(31 downto 0);
    MemK_in : in std_logic_vector(31 downto 0);
    ROM_addr_out : out std_logic_vector(3 downto 0);
    Mem_wren_out : out std_logic;
    Mem_wAddress_out : out std_logic_vector(6 downto 0);
    --Control Signals
    Start_in : in std_logic;
    Round_in : in std_logic_vector(5 downto 0);
    End_Round_out : out std_logic;
    End_Compression_out : out std_logic;
    --Data out
    RAM_Data_out : out std_logic_vector(31 downto 0)
);
end Compressor;

```

A seguire, vi è l'implementazione del corpo del Compressore:

```

architecture Behavioral of Compressor is
    type Compressor_State is (Idle,Op1,Op2,Assign,Evaluate,Finish); -- Define
the states
    signal State : Compressor_State;
    --OUT signal mapping
    signal End_Round_int : std_logic;
    signal End_Compression_int : std_logic;
    signal ROM_addr_int : std_logic_vector(3 downto 0);
    signal RAM_Data_int : std_logic_vector(31 downto 0);
    signal Mem_wren_int : std_logic;
    signal Mem_wAddress_int : std_logic_vector(6 downto 0);
    --Counters
    signal WriteRamCounter : std_logic_vector(3 downto 0);
    signal ReadRomCounter : std_logic_vector(3 downto 0);
    signal AssignCounter : std_logic_vector(3 downto 0);
    --Flip Flops
    signal StartFF : std_logic;
    --Internal Register
    signal A_int : std_logic_vector(31 downto 0);
    signal B_int : std_logic_vector(31 downto 0);
    signal C_int : std_logic_vector(31 downto 0);
    signal D_int : std_logic_vector(31 downto 0);
    signal E_int : std_logic_vector(31 downto 0);

```

```

signal F_int : std_logic_vector(31 downto 0);
signal G_int : std_logic_vector(31 downto 0);
signal H_int : std_logic_vector(31 downto 0);
signal R0 : std_logic_vector(31 downto 0); --Rotate Register 0
signal R1 : std_logic_vector(31 downto 0); --Rotate Register 1
signal R2 : std_logic_vector(31 downto 0); --Rotate Register 2
signal R3 : std_logic_vector(31 downto 0); --Rotate Register 3
signal R4 : std_logic_vector(31 downto 0); --Rotate Register 4
signal R5 : std_logic_vector(31 downto 0); --Rotate Register 5
signal S0 : std_logic_vector(31 downto 0); --Rotated and Xored Reg 0
signal S1 : std_logic_vector(31 downto 0); --Rotated and Xored Reg 1
signal ch : std_logic_vector(31 downto 0); --Xored Reg 0
signal naj : std_logic_vector(31 downto 0); --Xored Reg 1
signal temp0 : std_logic_vector(31 downto 0); --Temp Reg 0
signal temp1 : std_logic_vector(31 downto 0); --Temp Reg 1
begin
    End_Round_out <= End_Round_int;
    End_Compression_out <= End_Compression_int;
    RAM_Data_out <= RAM_Data_int;
    Mem_wAddress_out <= Mem_wAddress_int;
    Mem_wren_out <= Mem_wren_int;
    ROM_addr_out <= ROM_addr_int;
    process(clk)
    begin
        if rising_edge(clk) then
            if reset = '0' then
                --Outputs
                End_Round_int <= '0';
                End_Compression_int <= '0';
                Mem_wren_int <= '0';
                Mem_wAddress_int <= "00000000";
                A_int <= x"00000000";
                B_int <= x"00000000";
                C_int <= x"00000000";
                D_int <= x"00000000";
                E_int <= x"00000000";
                F_int <= x"00000000";
                G_int <= x"00000000";
                H_int <= x"00000000";
                R0 <= (others => '0');
                R1 <= (others => '0');
                R2 <= (others => '0');
                R3 <= (others => '0');
                R4 <= (others => '0');
                R5 <= (others => '0');
                S0 <= (others => '0');
                S1 <= (others => '0');
                naj <= (others => '0');
                ch <= (others => '0');
                ROM_addr_int <= (others => '0');
                RAM_Data_int <= (others => '0');
                --Counters
                WriteRamCounter <= (others => '0');
                ReadRomCounter <= (others => '0');
                AssignCounter <= (others => '0');
                --FlipFlops
                StartFF <= '0';
                --State
                State <= Idle;
            else
                case State is
                    when Idle =>
                        End_Round_int <= '0';

```

```

End_Compression_int <= '0';
if( Start_in = '1' and StartFF = '0' ) then --Catch Start
Event
    StartFF <= '1';
    ROM_addr_int <= "0000";
    elsif( StartFF = '1' ) then
    if( ReadRomCounter < 8+1 and Round_in = "000000" ) then
        ReadRomCounter <= ReadRomCounter + 1;
        case ReadRomCounter is
            when "0000" => A_int <= MemH_in; ROM_addr_int <=
"0001";
            when "0001" => B_int <= MemH_in; ROM_addr_int <=
"0010";
            when "0010" => C_int <= MemH_in; ROM_addr_int <=
"0011";
            when "0011" => D_int <= MemH_in; ROM_addr_int <=
"0100";
            when "0100" => E_int <= MemH_in; ROM_addr_int <=
"0101";
            when "0101" => F_int <= MemH_in; ROM_addr_int <=
"0110";
            when "0110" => G_int <= MemH_in; ROM_addr_int <=
"0111";
            when "0111" => H_int <= MemH_in; ROM_addr_int <=
"0000";
            when others => ROM_addr_int <= "0000";
        end case;
    else
        ReadRomCounter <= "0000";
        --Rx(31-N downto 0) <= y_in(31 downto N);
        --Rx(31 downto 31-(N-1)) <= y_in(N-1 downto 0);
        --Init the Rotate Registers
        R0(25 downto 0) <= E_int(31 downto 6);
        R0(31 downto 26) <= E_int(5 downto 0);
        R1(20 downto 0) <= E_int(31 downto 11);
        R1(31 downto 21) <= E_int(10 downto 0);
        R2(31-25 downto 0) <= E_int(31 downto 25);
        R2(31 downto 31-(25-1)) <= E_int(25-1 downto 0);
        R3(31-2 downto 0) <= A_int(31 downto 2);
        R3(31 downto 31-(2-1)) <= A_int(2-1 downto 0);
        R4(31-13 downto 0) <= A_int(31 downto 13);
        R4(31 downto 31-(13-1)) <= A_int(13-1 downto 0);
        R5(31-22 downto 0) <= A_int(31 downto 22);
        R5(31 downto 31-(22-1)) <= A_int(22-1 downto 0);
        StartFF <= '0';
        State <= Op1;
    end if;
    else
        State <= Idle;
    end if;
    when Op1 =>
        for i in 0 to 31 loop
            S1(i) <= R0(i) xor R1(i) xor R2(i);
            S0(i) <= R3(i) xor R4(i) xor R5(i);
            ch(i) <= (E_int(i) and F_int(i)) xor ((not E_int(i)) and
G_int(i));
            naj(i) <= (A_int(i) and B_int(i)) xor (A_int(i) and
C_int(i)) xor (B_int(i) and C_int(i));
        end loop;
        State <= Op2;
    when Op2 =>
        Temp0 <= H_int + S1 + ch + MemK_in + MemW_in;
        Temp1 <= S0 + naj;

```

```

        State <= Assign;
    when Assign =>
        if( AssignCounter < (8) ) then
            AssignCounter <= AssignCounter + 1;
            case AssignCounter is
                when "0000" => H_int <= G_int;
                when "0001" => G_int <= F_int;
                when "0010" => F_int <= E_int;
                when "0011" => E_int <= D_int + Temp0;
                when "0100" => D_int <= C_int;
                when "0101" => C_int <= B_int;
                when "0110" => B_int <= A_int;
                when "0111" => A_int <= Temp0 + Temp1;
                when others => AssignCounter <= "0000";
            end case;
            State <= Assign;
        else
            AssignCounter <= "0000";
            State <= Evaluate;
        end if;
    when Evaluate =>
        if( Round_in < 63 ) then
            State <= Idle;
            End_Round_int <= '1';
        else
            State <= Finish;
        end if;
    when Finish =>
        if( WriteRamCounter < (9) ) then
            Mem_wren_int <= '1';
            WriteRamCounter <= WriteRamCounter + 1;
            case WriteRamCounter is
                when "0000" => Mem_wAddress_int <= "1000000";
                RAM_Data_int <= A_int + MemH_in; ROM_addr_int <= "0000" ; --0x40
                when "0001" => Mem_wAddress_int <= "1000001";
                RAM_Data_int <= B_int + MemH_in; ROM_addr_int <= "0001" ; --0x41
                when "0010" => Mem_wAddress_int <= "1000010";
                RAM_Data_int <= C_int + MemH_in; ROM_addr_int <= "0010" ; --0x42
                when "0011" => Mem_wAddress_int <= "1000011";
                RAM_Data_int <= D_int + MemH_in; ROM_addr_int <= "0011" ; --0x43
                when "0100" => Mem_wAddress_int <= "1000100";
                RAM_Data_int <= E_int + MemH_in; ROM_addr_int <= "0100" ; --0x44
                when "0101" => Mem_wAddress_int <= "1000101";
                RAM_Data_int <= F_int + MemH_in; ROM_addr_int <= "0101" ; --0x45
                when "0110" => Mem_wAddress_int <= "1000110";
                RAM_Data_int <= G_int + MemH_in; ROM_addr_int <= "0110" ; --0x46
                when "0111" => Mem_wAddress_int <= "1000111";
                RAM_Data_int <= H_int + MemH_in; ROM_addr_int <= "0111" ; --0x47
                when "1000" => Mem_wAddress_int <= "1001001";
                RAM_Data_int <= x"00000001"; --0x49 Status Register
                when others => Mem_wAddress_int <= "1111111";
            end case;
            State <= Finish;
        else
            Mem_wren_int <= '0';
            End_Compression_int <= '1';
            State <= Idle;
        end if;
    when others =>
        State <= Idle;
    end case;
end if;
end if;

```

```
end process;  
end Behavioral;
```

### Terminazione

A valle dell'esecuzione della compressione, ci dev'essere una fase di terminazione che consente al Processing System di poter recuperare, senza ambiguità, il digest risultato dalla compressione svolta dalla Periferica. Per far ciò, ci si avvale di uno speciale registro, il registro di Stato, all'interno dell'Interfaccia AXI. Questo registro viene scritto dalla Periferica a valle del termine della compressione. Il Processing System, in polling su questo registro, al momento in cui rileverà che il registro di Stato è stato "sporcato" dalla Periferica, provvederà a pulire tale registro e recuperare i dati dalla RAM dell'Interfaccia AXI. In Figura 16 viene fornito il diagramma che modella la Terminazione.

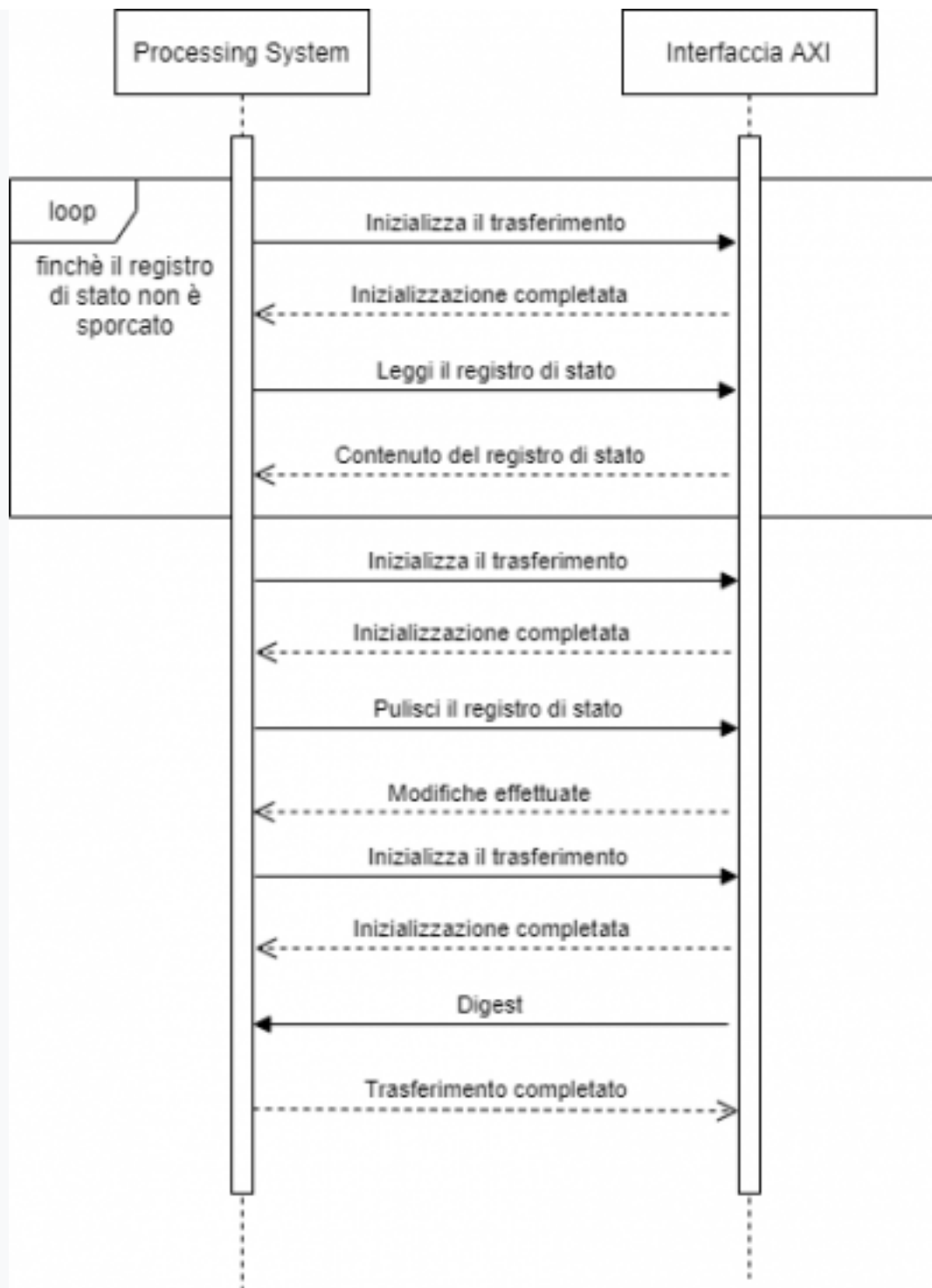




Figura 16: Diagramma UML di sequenza modellante la fase di Terminazione

## Risultati

In questa sezione si andranno a riportare i risultati ottenuti nei test della nostra periferica con AXI lite e full sia in simulazione che tramite l'utilizzo di un driver. Si cercherà in questo modo di verificare sperimentalmente quanto l'utilizzo del burst nella trasmissione e ricezione dei dati tra la parte PS e la nostra periferica possa modificare i tempi di esecuzione dell'algoritmo di SHA-256 o almeno della parte che è stata implementata in hardware. Dopodiché si andrà a descrivere come è stato sviluppato il driver per l'utilizzo della periferica e come è stata sviluppata in software la restante parte dell'algoritmo SHA-256.

### Tempi Hardware

Per calcolare la differenza nei tempi di esecuzione al cambiare dell'interfaccia AXI tra lite e full sono stati sviluppati due test bench praticamente identici tra loro. L'unica differenza sta nel fatto che nel caso di AXI full viene eseguito un burst nella trasmissione e nella ricezione dei dati. Il test consiste dei seguenti passi sequenziali:

1. Reset dell'AXI e della periferica
2. Scrittura dell'array W di 256 byte nella memoria della periferica
3. Scrittura sul registro di controllo RC della periferica (start della compressione)
4. Attesa per il calcolo del Digest da parte della periferica
5. Lettura del Digest di 32 byte dalla periferica

La differenza nei due test bench è ricercabile nei passi 2 e 5, in quanto la scrittura in caso di AXI full avverrà con un burst di 64 trasferimenti con un unico hand shake, mentre nel caso lite ci saranno 64 hand shake per la scrittura dell'array W. Allo stesso modo, per la lettura, avremo un burst di 8 letture con un unico hand shake in caso di AXI full, mentre con AXI lite dovranno avvenire 8 hand shake per la lettura del Digest. Verifichiamo quanto detto andando a simulare il flusso di esecuzione tramite il tool Vivado. Il codice del test bench segue i passi descritti precedentemente realizzando un protocollo AXI e nei due casi ci darà i seguenti risultati:

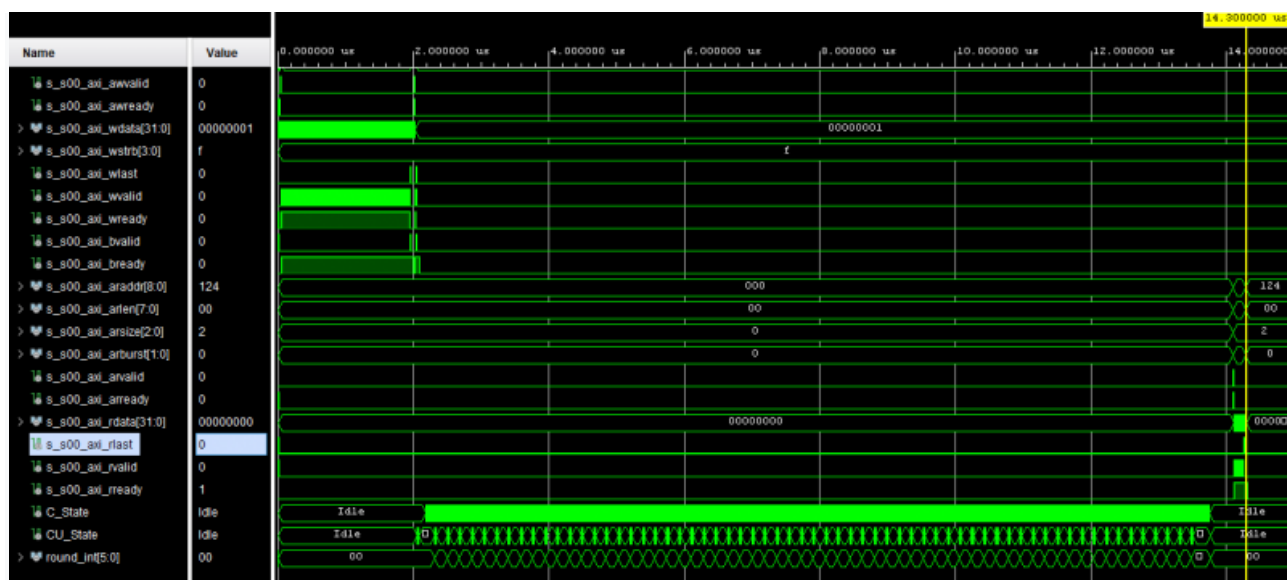


Figura 17: AXI-FULL

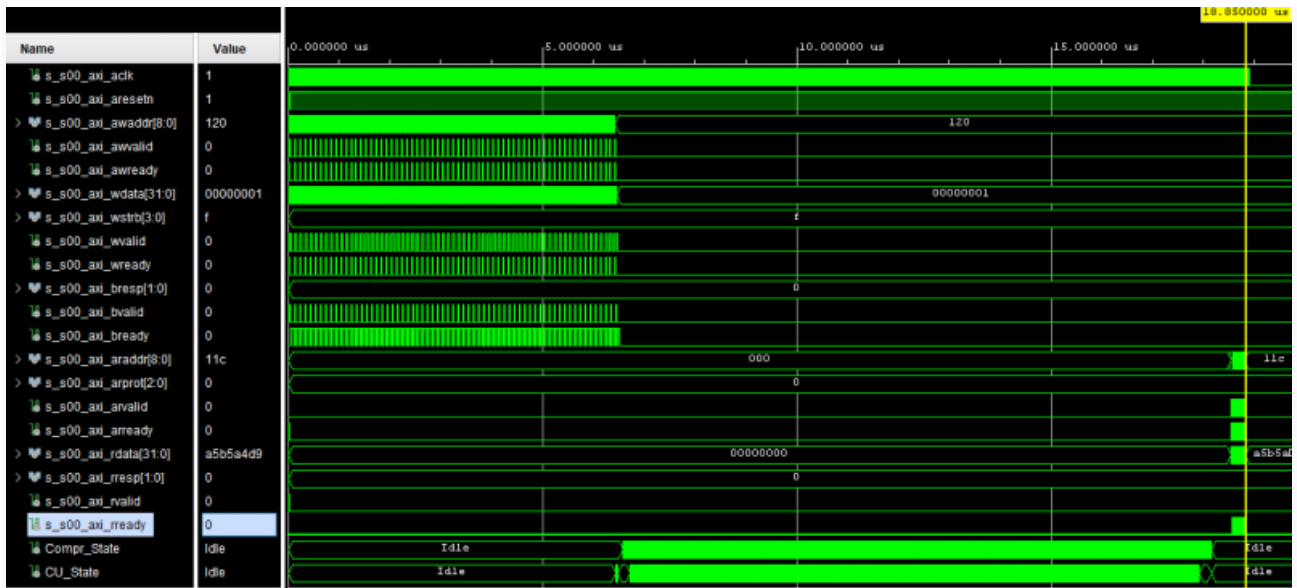


Figura 18: AXI-LITE

A prima vista è possibile subito notare che il tempo di esecuzione in caso di AXI-Full è di circa 14,30 us mentre in caso di AXI-Lite è di circa 18,85 us per cui c'è un'accelerazione di più di 4 us per tutto il flusso. In realtà però come già accennato i tempi di esecuzione dei calcoli all'interno della periferica è lo stesso l'unica differenza sta nei tempi di trasmissione. Per cui possiamo andare ad approfondire meglio il tempo in ognuno dei 5 step precedentemente descritti con la seguente tabella:

*Tabella 1: Tempi di esecuzione AXI-FULL/AXI-LITE*

Protocollo	Reset	Scrittura W (64 Byte)	Scrittura RC (1 Byte)	Attesa Digest	Lettura Digest (8 Byte)
AXI-FULL	30 ns	1930 ns	110 ns	12050 ns	180 ns
AXI-LITE	30 ns	6740 ns	110 ns	12050 ns	320 ns

L'accelerazione in scrittura con un burst di 64 elementi permette di risparmiare 4410ns su 6340ns cioè circa il 70% dei tempi di scrittura. Mentre la lettura con un burst di 8 elementi permette di risparmiare 140ns su 320ns cioè circa il 45% dei tempi di lettura. Vediamo queste differenze in simulazione:

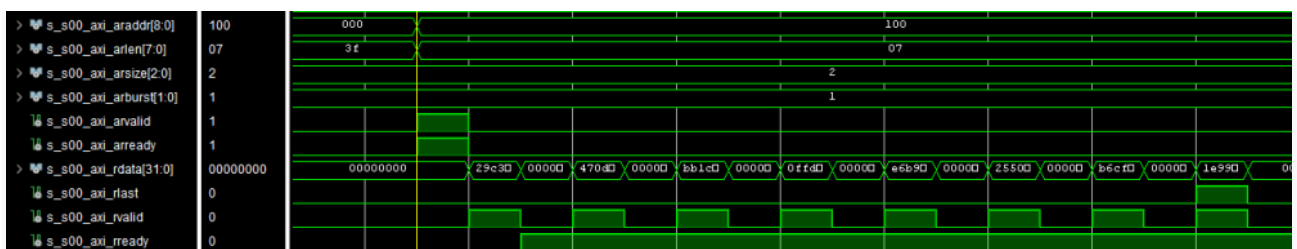


Figura 19: read with burst

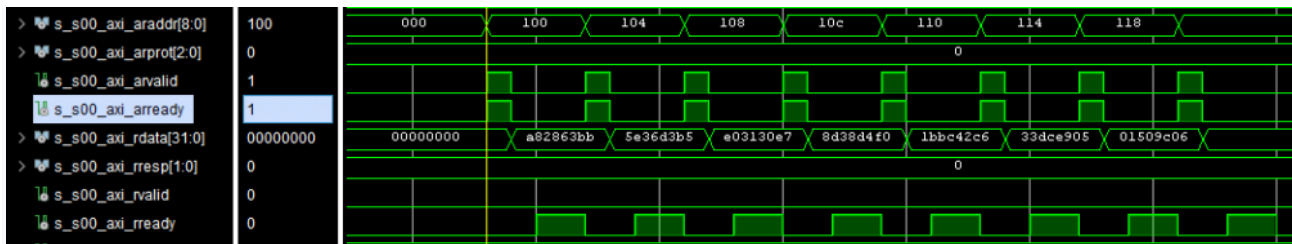


Figura 20: read without burst

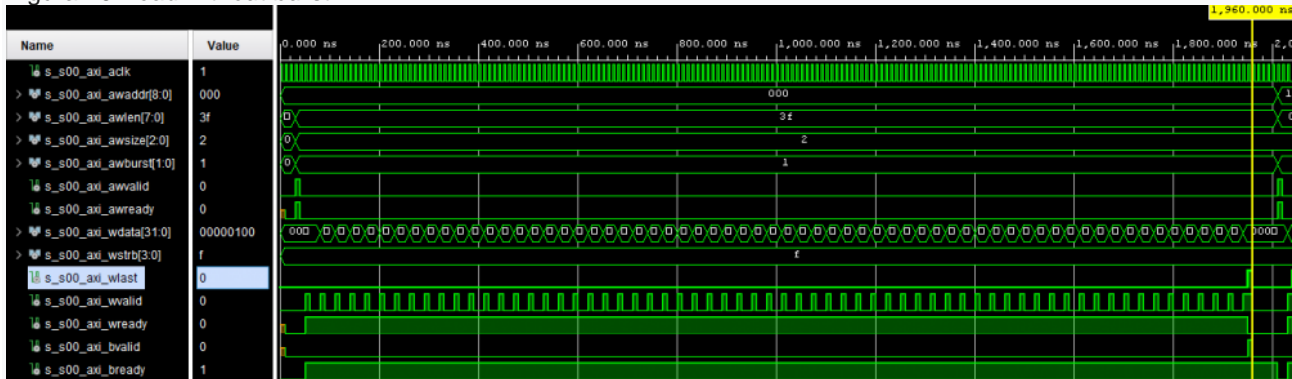


Figura 21: write with burst

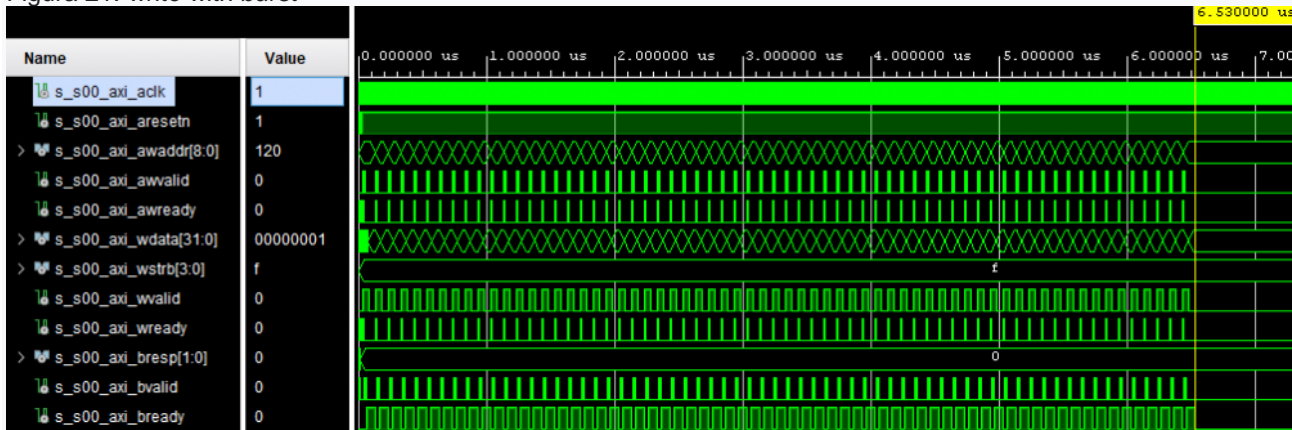


Figura 22: write without burst

Ci si potrebbe chiedere quindi se fosse possibile quantizzare il miglioramento ottenuto tramite AXI-FULL rispetto ad AXI-LITE al variare del numero di elementi trasmessi in burst. Ovviamente minore è il numero degli elementi in burst, maggiore è il numero degli handshake che bisogna fare per la comunicazione, per cui il tempo di esecuzione andrà ad aumentare. Come caso di test è possibile eseguire la scrittura del protocollo dimezzando la dimensione degli elementi in burst ad ogni test passo per passo per verificare se esiste una sorta di legge lineare che descriva il ritardo causato dagli handshake. Svolgendo il test con dei burst di 64, 32, 16 e 8 elementi i risultati sono i seguenti:

**Tabella 2: Tempi di esecuzione AXI-FULL al variare degli elementi in burst**

AXI-FULL	elementi in burst	tempo	overhead handshake
	64	1930 ns	60 ns
	32	1990 ns	120 ns

	16	2110 ns	240 ns
	8	2350 ns	480 ns
	4	2830 ns	960 ns

Dato "t" il tempo di trasmissione in nanosecondi, "n" il numero di handshake, "b" il tempo di trasmissione base senza handshake (1870ns nel nostro esempio) e h il tempo per ogni handshake(60ns) dagli esperimenti risulta semplice esprimere una legge sul ritardo ottenuto:

$$t = b + h \cdot n \quad \text{con } h = 60\text{ns}$$

Dato quindi il numero di elementi  $2^k$  e il numero di elementi in burst  $2^l$ , dove nel nostro esempio un elemento è una parola di 32bit, è possibile calcolare il numero di handshake come:

$$n = 2^{(k-l)}$$

Per cui la formula del tempo in funzione del numero di elementi e il numero di elementi in burst è la seguente:

$$t = b + h \cdot 2^{(k-l)} \quad \text{con } h = 60\text{ns}$$

## Driver Software

Una volta sviluppata e testata la periferica è opportuno fornire un driver per l'utilizzo di tale periferica il cui comportamento e protocollo è stato ampiamente discusso nei precedenti paragrafi. Per fare ciò è stato utilizzato il tool Vitis che, preso in ingresso il bitstream generato tramite Vivado, è in grado di fornire i file per la periferica e per la programmazione su scheda target automaticamente. Per cui l'unica cosa che ci resta da fare è semplicemente sviluppare un driver che possa utilizzare le funzionalità della nostra periferica, vediamo quindi come prima cosa il file Driver.h:

```

1 #include <stdio.h>
2
3 #define MS_BASE_FULL_ADDRESS 0x7AA00000 //Base Address for the peripheral, Axi Full version - MS W MEMORY REGION - 256 BYTE
4 #define MS_DIGEST_FULL_ADDRESS 0x7AA00100 //MS Digest Memory region - 32 BYTE
5 #define MS_CR_FULL_ADDRESS 0x7AA00120 //Control register memory region - 4 BYTE
6 #define MS_SR_FULL_ADDRESS 0x7AA00124 //Status register memory region - 4 BYTE
7 #define MS_VR_FULL_ADDRESS 0x7AA00128 //Version register memory region - 4 BYTE
8 #define MS_RESERVED_FULL_ADDRESS 0x7AA00132 //RESERVED
9
10 #define MS_BASE_LITE_ADDRESS 0x43C00000 //Base Address for the peripheral, Axi Lite version - MS W MEMORY REGION - 256 BYTE
11 #define MS_DIGEST_LITE_ADDRESS 0x43C00100 //MS Digest Memory region - 32 BYTE
12 #define MS_CR_LITE_ADDRESS 0x43C00120 //Control register memory region - 4 BYTE
13 #define MS_SR_LITE_ADDRESS 0x43C00124 //Status register memory region - 4 BYTE
14 #define MS_VR_LITE_ADDRESS 0x43C00128 //Version register memory region - 4 BYTE
15 #define MS_RESERVED_LITE_ADDRESS 0x43C00132 //RESERVED
16
17 #define AXI_LITE 0
18 #define AXI_FULL 1
19
20 typedef struct{
21     uint32_t W[64];
22     uint32_t Digest[8];
23     uint32_t CR;
24     uint32_t SR;
25     uint32_t VR;
26     uint32_t Reserved[0xFFFF-0x0136+0x0001];
27 }MS_Axi_t;
28
29 void Initialize(volatile MS_Axi_t **, const uint8_t *);
30
31 void Write(volatile uint32_t *, const uint32_t *);
32
33 uint32_t Read(const volatile uint32_t *);
34
35 void Deinitialize(volatile MS_Axi_t **);
36
37 void SHA_Compression(volatile MS_Axi_t *, const uint32_t *, uint32_t *);
38

```

Figura 27: *Driver.h*

Il seguente file utilizza le direttive del precompilatore per fornire all'utente dei nomi simbolici che rappresentano gli indirizzi ai vari registri presenti nella nostra periferica. Inoltre, sono fornite le funzionalità per l'inizializzazione, la de-inizializzazione, la lettura e la scrittura su registri. Infine, nell'interfaccia è fornita una funzione `SHA_Compression()` che permette di svolgere il protocollo completo per l'esecuzione della parte di SHA implementata in hardware. La maggior parte delle funzioni risultano essere banali ma possiamo approfondire quest'ultima funzione vedendo il codice:

```

void SHA_Compression(volatile MS_Axi_t * MS, const uint32_t * W, uint32_t * result){

    //Step 1 : Write the Matrix into the Peripheral Memory

    for(int i = 0; i < 64; i++){
        Write(&MS->W[i], W[i]);
    }

    //Step 2 : Start the compression by writing to the CR

    Write(&MS->CR, 0x00000001);

    //Step 3 : Check the Status Register in Polling
    while(Read(&MS->SR) == 0x00000000);

    //Step 4 : Clean the SR and get the Digest

    for(int i = 0; i < 8; i++){
        result[i] = MS->Digest[i];
    }

    Write(&MS->SR, 0x00000000);

}

```

Figura 28: *SHA\_Compression*

La funzione mostrata, utilizzando la struttura rappresentante la nostra mappa di memoria presente nel file Driver.h, risulta essere molto semplice: per prima cosa si andrà a scrivere l'array W nella memoria della periferica, dopodiché verrà scritto il registro di controllo per fare iniziare la compressione. Una volta iniziata la compressione il driver attenderà in polling che il registro di stato venga modificato ad indicare la fine della compressione e infine si andrà a leggere il Digest e rimodificare il registro di stato riassegnandogli il valore 0.

Per testare l'algoritmo completo è stata sviluppata anche la prima parte di SHA-256 in software. La prima parte dell'algoritmo dovrà prendere in ingresso una qualsiasi parola (Per l'attuale livello di implementazione saranno accettate solo parole con al più 32 caratteri) e calcolare l'array W con una serie di passaggi. Infine verrà utilizzata la periferica da noi creata che a partire dall'array W restituirà il Digest che corrisponderà alla versione hashata della parola iniziale. L'algoritmo completo è il seguente:

```
void SHA256(char * text, uint32_t * Digest){
    uint32_t TextSize;
    uint32_t dim = 0;
    uint8_t Data[64];
    uint32_t W[64];
    uint32_t s0;
    uint32_t s1;
    MS_Axi_t * MS;

    while(text[dim] != '\0') dim++;
    TextSize = 8*dim;

    if(TextSize > 512-65 || dim > 32) printf("To be added....\n");
    else{
        for(int i = 0; i < 64; i++){
            if(i < dim+1){
                Data[i] = text[i];
            }
            else if (i == dim+1){
                Data[i] = 0x80;
            }
            else if (i > dim +1 && i < 64 - 1){
                Data[i] = 0x00;
            }
            else{
                Data[63] = TextSize;
            }
        }

        for(int i = 0; i < 64; i++){
            if(i < 16){
                W[i] = ((uint32_t)Data[i*4+3]) | ((uint32_t)Data[i*4+2] << 8) | ((uint32_t)Data[i*4+1] << 16) | ((uint32_t)Data[i*4] << 24);
            }
            else{
                W[i] = 0;
            }
        }

        for(int i = 16; i < 64; i++){
            s0 = (rightRotate(W[i-15],7)) ^ (rightRotate(W[i-15],18)) ^ (W[i-15] >> 3);
            s1 = (rightRotate(W[i-2],17)) ^ (rightRotate(W[i-2],19)) ^ (W[i-2] >> 10);
            W[i] = (W[i-16] + s0 + W[i-7] + s1) ;
        }

        Initialize(&MS,AXI_FULL);
        SHA_Compression(MS,W,Digest);
        Deinitialize(&MS);
    }
}
```




Figura 29: SHA-256

Infine vediamo un semplice main che va ad utilizzare la nostra periferica sia in versione AXI\_FULL che in versione AXI\_LITE per generare l'hash della stringa "hello world":

```

56 int main()
57 {
58     init_platform();
59
60     //Variables
61     uint32_t Digest[8];
62     XTime start;
63     XTime end;
64
65     //Starting string
66     char a[12];
67     strcpy(a, "hello world");
68
69     //Hash function with AXI_LITE
70     XTime_GetTime(&start);
71     SHA256(a, Digest, AXI_LITE);
72     XTime_GetTime(&end);
73
74     printf("Digest LITE: ");
75     for(int i = 0; i < 8; i++) printf("%08x", Digest[i]);
76     printf("\nTime LITE: %lld\n\n", 2*(end-start));
77
78     //Hash function with AXI_FULL
79     XTime_GetTime(&start);
80     SHA256(a, Digest, AXI_FULL);
81     XTime_GetTime(&end);
82
83     printf("Digest FULL: ");
84     for(int i = 0; i < 8; i++) printf("%08x", Digest[i]);
85     printf("\nTime FULL: %lld\n\n", 2*(end-start));
86
87     cleanup_platform();
88
89     return 0;
90 }
91

```

 Console
  Vitis Serial Terminal
  Debugger Console
  Executables
  Vitis

Connected to: Serial ( COM5, 115200, 0, 8 )

Digest LITE: b94d27b9934d3e08a52e52d7da7dabfac484efe37a5380ee9088f7ace2efcde9  
 Time LITE: 48140

Digest FULL: b94d27b9934d3e08a52e52d7da7dabfac484efe37a5380ee9088f7ace2efcde9  
 Time FULL: 45814

Figura 29: *main*



## Riferimenti

---

[1] AXI Reference Guide. [Online] Disponibile

presso: [https://www.xilinx.com/support/documentation/ip\\_documentation/ug761\\_axi\\_reference\\_guide.pdf](https://www.xilinx.com/support/documentation/ip_documentation/ug761_axi_reference_guide.pdf)

[2] AMBA AXI and ACE Protocol Specification. [Online] Disponibile

presso: <https://developer.arm.com/documentation/ih0022/hc>