Implementazione Hardware di SHA-256 per Mining Bitcoin su FPGA. Prova pratica per il corso di Sicurezza dell'Informazione M

Arsal-Hanif Livoroi

3 settembre 2025

1 Introduzione

Questo progetto è stato realizzato per unire le conoscenze apprese da Sistemi Digitale e Sicurezza dell'Informazione.

L'Obiettivo di questo progetto è l'implementazione del algoritmo SHA-256 su FPGA (PYNQ-Z2), e cercare di ottimizzaro per il mining di Bitcoin.

- PYNQ, il cui sistema operativo è basato su Linux, e quindi da la possibilità di gestire il file system in maniera più agevole rispetto alle altre board, nelle quali la procedura di installazione di un sistema operativo è più complessa. Questo induce a cercare di utilizzare metodi complessi per la gestione di file. Ho trovato diversi articoli che cercano di creare un file system direttamente su bare metal, ma sono soluzioni complesse da adottare e servirebbe uno studio specifico ed approfondito.
- SHA-256 è alla base del mining di diverse criptovalute e quindi esistono molte versioni con cui potersi confrontare, anche se questo progetto ha uno scopo finale differente poiché prevede di elaborare un messaggio unico molto lungo, invece che molti messaggi brevi.
- è possibile gestire i messaggi in ingresso come un flusso di dati, quindi c'è la possibilità di utilizzare gli stream e la direttiva dataflow. Questi strumenti sono molto utilizzati in ambito di Computer Vision ed affini, o in generale in ogni ambito che preveda un flusso di informazioni in ingresso/uscita. Quindi sono strumenti che si addicono ad un vasto uso applicativo.

1.1 SHA-256

SHA-256 è una funzione crittografica di hash, e come ogni algoritmo della stessa famiglia produce un digest (o hash) di lunghezza fissa, in questo caso 256 bit, partendo da un messaggio di lunghezza variabile.

Di seguito è riportata il semplice pseudo-codice dell'algoritmo, preso direttamente da Wikipedia, ma facendo le dovute verifiche confrontandolo con fonti più affidabili per determinarne la correttezza.

```
Note: All variables are unsigned 32 bits and wrap modulo 232 when calculating

Initialize variables
(first 32 bits of the fractional parts of the square roots of the first 8 primes 2..19):
h0 := 0x6a09e667
h1 := 0xbb67ae85
h2 := 0x3c6ef372
h3 := 0xa54ff53a
h4 := 0x510e527f
h5 := 0x9b05688c
h6 := 0x1f83d9ab
h7 := 0x5be0cd19

Initialize table of round constants
(first 32 bits of the fractional parts of the cube roots of the first 64 primes 2..311):
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

#Pre-processing:
append the bit '1' to the message
append k bits '0', where k is the minimum number >= 0 such that the resulting message
    length (in bits) is congruent to 448 (mod 512)
append length of message (before pre-processing), in bits, as 64-bit big-endian integer

#Process the message in successive 512-bit chunks:
break message into 512-bit chunks
for each chunk
    break chunk into sixteen 32-bit big-endian words w[0..15]

    #Extend the sixteen 32-bit words into sixty-four 32-bit words:
    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

    #Initialize hash value for this chunk:
    a := h0
    b := h1
    c := h2
    d := h3
    e := h4
    f := h5
    g := h6
    h := h7

    #Main loop:
    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]

        h := g
        g := f
        f := e
        e := d + t1
        d := c
        c := b
        b := a
        a := t1 + t2

    #Add this chunk's hash to result so far:
    h0 := h0 + a
    h1 := h1 + b
    h2 := h2 + c
    h3 := h3 + d
    h4 := h4 + e
    h5 := h5 + f
    h6 := h6 + g
    h7 := h7 + h

#Produce the final hash value (big-endian):
digest = hash = h0 append h1 append h2 append h3 append h4 append h5 append h6 append h7
```

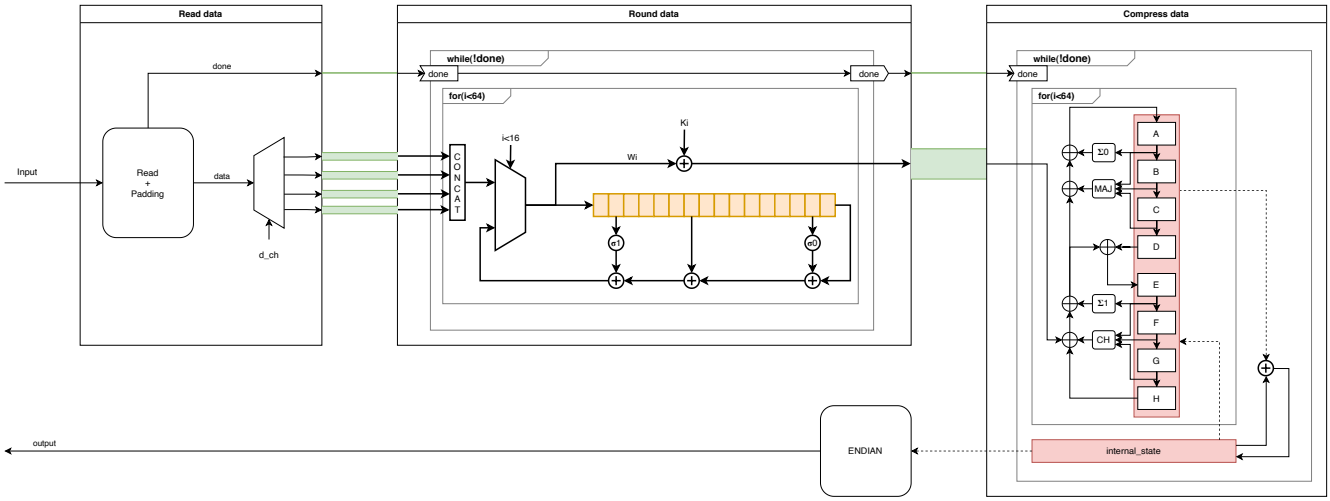


Figura 1: Diagramma dell'architettura SHA-256 adottata

2 Implementazione di SHA-256 su FPGA

Come visto nello pseudocodice, l'algoritmo è abbastanza semplice. Le implementazioni più diffuse seguono sostanzialmente la stessa struttura, infatti consiste nel prendere in ingresso il messaggio, dividerlo in chunk da 64 byte e per ogni chunk eseguire sequenzialmente l'espansione del messaggio e poi la compressione, fino ad arrivare all'ultimo chunk, dove viene eseguito il padding ed un ultima espansione con successiva compressione. L'unica differenza sostanziale che si può notare nelle varie implementazioni è il fatto che il padding viene eseguito solo alla fine all'ultimo blocco.

Le implementazione HLS più diffuse, invece, si limitano a mettere in pipeline i vari loop oppure farne l'unrolling completo o parziale, strategie che non sfruttano il pieno potenziale del FPGA, dato che mentre viene eseguita la compressione il modulo che si occupa dell'espansione è in idle, e se si volesse mantenerlo in funzione dando la direttiva di pipeline all'intera funzione, si dovrebbero usare molte più risorse, ammesso che la compilazione vada a buon fine, data la natura dell'algoritmo, generando errori di "memory dependencies" difficili da correggere.

Invece, le implementazioni VHDL o Verilog più utilizzate nella comunità del minig sono spesso indirizzate alla realizzazione di prototipi per ASIC, sfruttando alcune semplificazioni per fare direttamente lo SHA-256 doppio e sfruttando delle ottimizzazioni note.

2.1 Progettoazione IP tramite Vitis HLS

Per progettare l'IP ho utilizzato Vitis HLS 2021.1, in figura 1 ho rappresentato in un diagramma l'idea del algoritmo che ho poi implementato.

I vari container rappresentano i task che vengono creati dal compilatore di Vitis HLS che interpreta la direttiva DATAFLOW, mentre in verde sono rappresentati i vari stream utilizzati per il flusso di dati tra i vari task.

Andando più nel dettaglio, vengono sintetizzati 4 task: Read data; Round data; Compress data; Endian. Quest'ultimo in realtà potrebbe essere implementato anche come semplice funzione che verrà eseguita una sola volta alla fine.

Read data

- Legge e reindirizza lo stream di input composto da *ap_axiu*. Come primo valore viene riportato se c'è altro da leggere. In caso affermativo viene letto e reindirizzato il contenuto fino alla ricezione del TLAST e si ripete la verifica. In caso negativo si procede con gli step successive.
- Esegue il padding del messaggio generando nello stream d'uscita la sequenza di byte necessaria a rispettare le specifiche dello SHA-256.

- Gli ultimi 8 byte compongono la lunghezza del messaggio quindi vengono generati nello stream d'uscita del task.

NB: Ogni 64 caratteri viene generato un segnale di done corrispondente a "0" se non ha ancora finito ed "1" al termine. Questo serve a sincronizzare i vari Task.

Questa fase ha un grosso problema, non sono riuscito a trova un modo per abbassare la latenza, questo è dovuto al modo in cui viene letto l'input dall'interfaccia axis. Servirebbe un alto metodo di trasferimento dei caratteri per poter incrementare ulteriormente le prestazioni. Ho provato varie interfacce di input, ma senza alcun successo.

Round data

- Ad ogni iterazione del loop più esterno reindirizza il segnale di done al prossimo task finché il segnale non è "1", cioè ogni 64 *parole* generate
- Crea le prime 16 *parole* concatenando 4 caratteri alla volta provenienti dagli stream di ingresso, e reindirizzo le *parole* sullo stream in uscita, inserendole inoltre in uno shift register per il loop successivo.
- Genera in uscita le restanti 48 *parole* utilizzando lo shift register inizializzato nella fase precedente seguendo le specifiche SHA-256

Questa parte è molto efficiente grazie allo shift register, ma comunque presenta una limitazione di velocità nel primo loop per via della velocità con cui vengono prodotto i caratteri in ingresso dovuto al Task precedente.

Ho provato anche una versione utilizzando un buffer, ma ha prestazioni inferiori ed utilizza più risorse.

Un'altra variante prevede che il concatenamento dei 4 caratteri venga eseguito nel Task precedente, cioè nella fase di lettura dell'input, ma non ci sono differenze rispetto a questa versione in termini di prestazioni e risorse.

Nella versione finale, in uscita, invece di un singolo stream ho adottato una soluzione multicanale, questo per poter utilizzare UNROLL e la versione con Sympy (analizzata più avanti) senza problemi.

Compress data

- Inizializza lo stato interno del output. (nel diagramma è rappresentato in basso per questioni di comodità, ma è la prima operazione che viene eseguita)
- Ad ogni iterazione del loop più esterno verifica il segnale di done finché il segnale non è *vero*, cioè ogni 64 *parole* generate.
- Ad ogni iterazione copia lo stato interno in un buffer temporaneo che servirà a comprimere 64 *parole* alla volta modificando il buffer temporaneo. Una volta compresse, il buffer temporaneo verrà sommato allo stato interno, cioè il valore della compressione precedente.
- Una volta concluso, lo stato interno rappresenterà l'hash a cui servirà un ultima elaborazione prima di ottenere il risultato finale

Questo task è il più oneroso di tutti, per questo ho cercato di adottare qualche semplificazione simile all'Unrolling che ha permesso di risparmiare qualche operazione (tramite il CSE ricavato grazie a Sympy).

Per esempio ho utilizzato una semplificazione di fattore 4, per questo tra il Task *Round data* e il *Compress Data* ho utilizzato 4 canali, ma si potrebbe fare la stessa cosa con più canali.

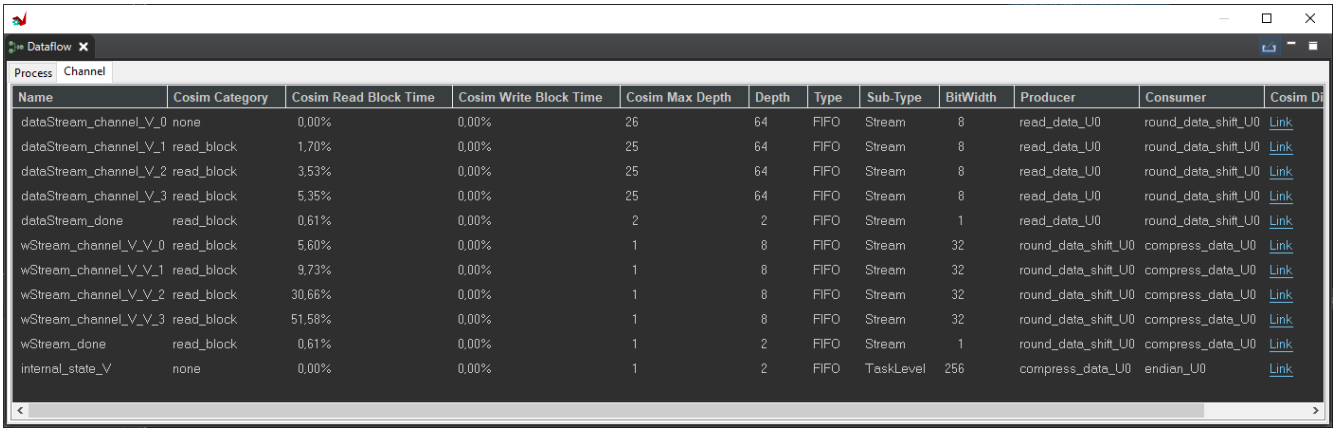
In questa fase normalmente si esegue la somma con la costante K_i, ma ho ritenuto fosse meglio calcolare il valore nel task precedente, per suddividere le operazioni in modo equo.

Endian

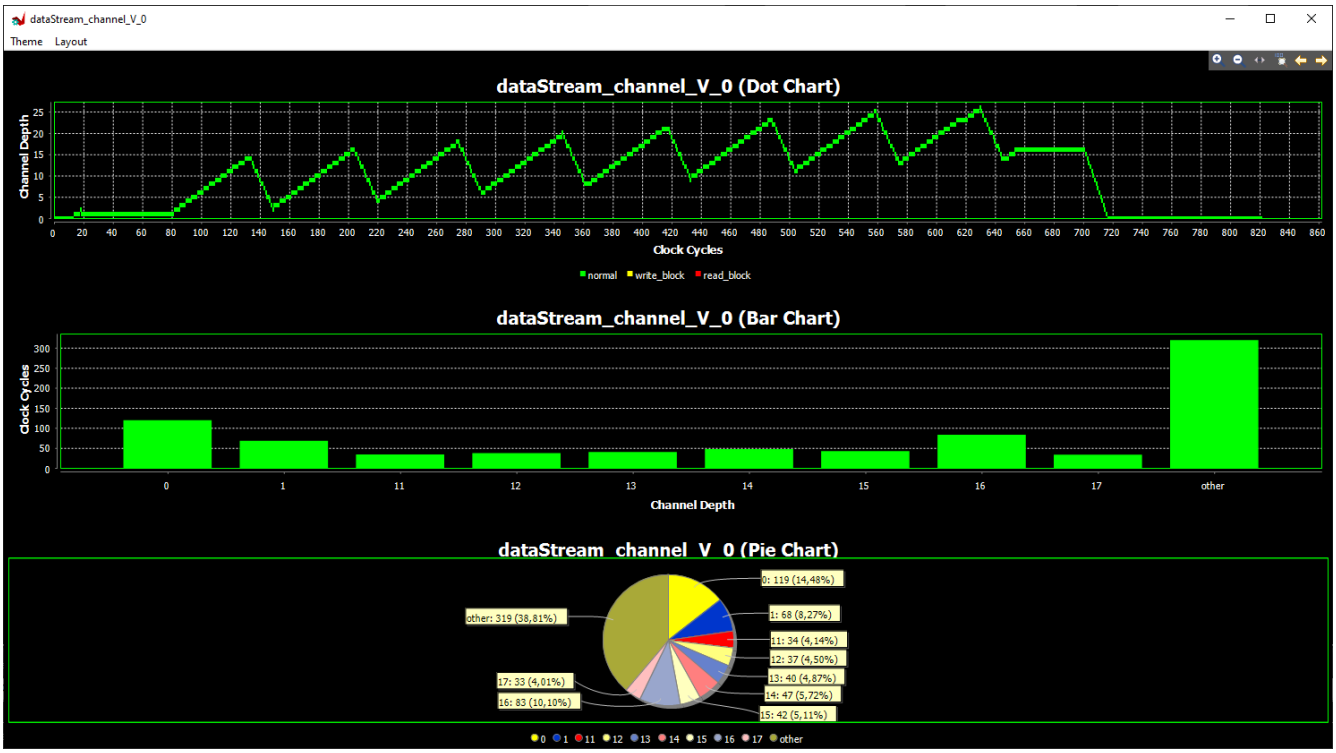
Converte da Little Endian a Big Endian.

Quest'operazione viene eseguita una sola volta, alla fine.

Possiamo osservare l'andamento dell'algoritmo tramite il Wave Viewer, dove si nota come questa fase sia ininfluente rispetto al resto.



Name	Cosim Category	Cosim Read Block Time	Cosim Write Block Time	Cosim Max Depth	Depth	Type	Sub-Type	BitWidth	Producer	Consumer	Cosim Di
dataStream_channel_V_0	none	0,00%	0,00%	26	64	FIFO	Stream	8	read_data_U0	round_data_shift_U0	Link
dataStream_channel_V_1	read_block	1,70%	0,00%	25	64	FIFO	Stream	8	read_data_U0	round_data_shift_U0	Link
dataStream_channel_V_2	read_block	3,53%	0,00%	25	64	FIFO	Stream	8	read_data_U0	round_data_shift_U0	Link
dataStream_channel_V_3	read_block	5,35%	0,00%	25	64	FIFO	Stream	8	read_data_U0	round_data_shift_U0	Link
dataStream_done	read_block	0,61%	0,00%	2	2	FIFO	Stream	1	read_data_U0	round_data_shift_U0	Link
wStream_channel_V_V_0	read_block	5,60%	0,00%	1	8	FIFO	Stream	32	round_data_shift_U0	compress_data_U0	Link
wStream_channel_V_V_1	read_block	9,73%	0,00%	1	8	FIFO	Stream	32	round_data_shift_U0	compress_data_U0	Link
wStream_channel_V_V_2	read_block	30,66%	0,00%	1	8	FIFO	Stream	32	round_data_shift_U0	compress_data_U0	Link
wStream_channel_V_V_3	read_block	51,58%	0,00%	1	8	FIFO	Stream	32	round_data_shift_U0	compress_data_U0	Link
wStream_done	read_block	0,61%	0,00%	1	2	FIFO	Stream	1	round_data_shift_U0	compress_data_U0	Link
internal_state_V	none	0,00%	0,00%	1	2	FIFO	TaskLevel	256	compress_data_U0	endian_U0	Link



Canali stream

Parte essenziale di questo progetto. Gli stream sono strumenti molto potenti, e per gestirgli al meglio è necessario utilizzare i tool forniti dal ambiente di sviluppo. Infatti, l'efficienza dei canali si può studiare dopo aver eseguito la co-simulazione, questo permette di analizzare il congestionamento delle code, permettendo così un bilanciamento adeguato della profondità degli stream e della velocità dei vari Task.

Per esempio se c'è una congestione in scrittura, si può aumentare la profondità del canale. Mentre se c'è un rallentamento in lettura, si può aumentare il numero di canali e/o rendere più veloce il Task che produce il dato.

Un esempio dei tool messi a disposizione da Vitis HLS sono riportati nelle figure sopra.

I canali implementati sono suddivisi in due gruppi la cui struttura è definita nel file *sha256.hpp* e sono *dataChannels* e *wordChannels*.

- *dataChannels*: Collega il task *Read data* e *Round data* per inviare i caratteri senza segno (8 bit) ed è composto da 4 canali di profondità pari a 64 più un canale da un bit per il segnale di *done*
- *wordChannels*: Collega il task *Round data* e *Compress data* per inviare le parole (32 bit) ed è composto da 4 canali di profondità pari a 8 più un canale da un bit per il segnale di *done*

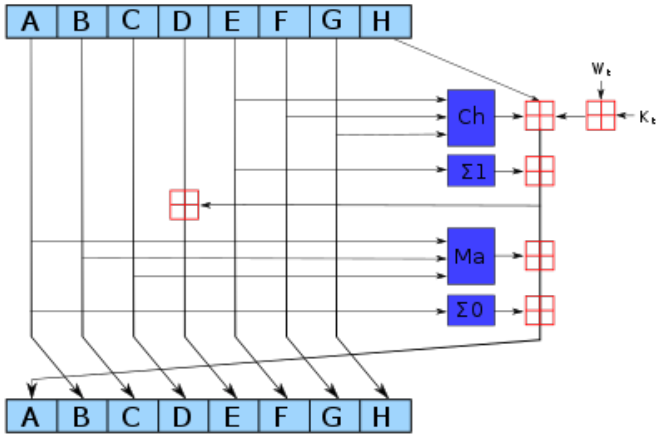


Figura 2: Rappresentazione di un iterazione della fase di compressione

2.2 Sympy

Libreria Python utile, tra le altre cose, a manipolare simbolicamente le espressioni.

Ho utilizzato questa libreria per cercare di ottimizzare la fase più costosa in termini di tempo e risorse, cioè la compressione. In figura 2 viene rappresentata una delle 64 iterazioni della compressione.

Mentre la funzione *compress_1*(), qui sotto, è la sua implementazione.

```
void compress_1(wordChannels &w, uint s[], ap_uint<BIT_WCH> &ch)
{
    uint w0=w.channel[ch++].read();
    uint a=s[0];
    uint b=s[1];
    uint c=s[2];
    uint d=s[3];
    uint e=s[4];
    uint f=s[5];
    uint g=s[6];
    uint h=s[7];
    uint x0=h + w0 + CH(e, f, g) + ep1(e);
    s[7]=g;
    s[6]=f;
    s[5]=e;
    s[4]=d + x0;
    s[3]=c;
    s[2]=b;
    s[1]=a;
    s[0]=x0 + MAJ(a, b, c) + ep0(a);
}
```

Per cercare di ottimizzare questa fase ho implementato un generatore di codice C tramite Python, il cui codice è riportato nel file *"cse_sha256.py"*. Questa tecnica è analoga alla direttiva UNROLL, infatti il numero di semplificazioni corrisponde al *unroll factor*, ma con la sostanziale differenza che alcune espressioni vengono semplificate tramite la CSE (Common Substitution Expression) evitando alcuni passaggi.

Tramite questa procedura ho generato diversi blocchi che si possono trovare nel file *"compress.hpp"*, ogni funzione ha nel nome il numero di iterazioni che sono state semplificate nel blocco. Per esempio *compress_4*() rappresenta la semplificazione di 4 iterazioni e per una compressione di 64 iterazioni dovrà essere eseguite 16 volte. Di seguito ho riportato il codice.

```
void compress_4(wordChannels &w, uint s[], ap_uint<BIT_WCH> &ch)
{
    uint w0=w.channel[ch++].read();
    uint w1=w.channel[ch++].read();
    uint w2=w.channel[ch++].read();
    uint w3=w.channel[ch++].read();
    uint a=s[0];
    uint b=s[1];
    uint c=s[2];
    uint d=s[3];
    uint e=s[4];
    uint f=s[5];
    uint g=s[6];
    uint h=s[7];
    uint x0=h + w0 + CH(e, f, g) + ep1(e);
    uint x1=d + x0;
    uint x2=g + w1 + CH(x1, e, f) + ep1(x1);
    uint x3=c + x2;
    uint x4=f + w2 + CH(x3, x1, e) + ep1(x3);
    uint x5=b + x4;
    uint x6=e + w3 + CH(x5, x3, x1) + ep1(x5);
    uint x7=x0 + MAJ(a, b, c) + ep0(a);
    uint x8=x2 + MAJ(x7, a, b) + ep0(x7);
    uint x9=x4 + MAJ(x8, x7, a) + ep0(x8);
    s[7]=x1;
    s[6]=x3;
    s[5]=x5;
    s[4]=a + x6;
    s[3]=x7;
    s[2]=x8;
    s[1]=x9;
    s[0]=x6 + MAJ(x9, x8, x7) + ep0(x9);
}
```

Nel file *"cse_sha256.py"* si può notare che gli step principali per l'esecuzione della CSE in questo contesto sono:

- Creare un vettore simbolico [A:H] corrispondente allo stato
- Iterare eseguendo le funzioni simboliche per il numero di volte desiderato
- Eseguire la CSE del vettore per ricavare così le operazioni necessarie per la creazione di sub-espressioni.

Aumentando il numero di semplificazioni si può ridurre la frequenza di lavoro mantenendo delle prestazioni simili. Questo è un grosso vantaggio se si dovesse riconvertire quest'architettura per il mining.

Si potrebbe eseguire la semplificazione di tutte e 64 le iterazioni, ma il tempo di calcolo per fare la CSE aumenta esponenzialmente, come riportato nella tabella 1. Sono riuscito ad eseguire al massimo 25 iterazioni. Perciò il valore massimo utilizzabile è 16, poiché i valori ideali sono potenze di 2.

In base al fattore di semplificazione selezionato ho ricavato il numero di canali tra il Task "Round" e "Compress". La tabella 2 rappresenta il numero di LUT ed FF necessari per eseguire le 64 iterazioni, inoltre è riportata la frequenza ideale e il tempo per ottenere un hash di una stringa con meno di 57 caratteri.

Non ho utilizzato fattori di semplificazione superiore al 4 poiché ha un costo in termini di risorse abbastanza elevato

Numero Semplificazioni (unroll factor)	CSE time [s]
1	0,091662884
2	0,004992247
3	0,003996849
4	0,004997969
5	0,009993792
6	0,010993958
7	0,012992382
8	0,017979622
9	0,019988775
10	0,021996975
11	0,028974771
12	0,032980204
13	0,04697299
14	0,070961475
15	0,144915342
16	0,312820911
17	0,82754159
18	2,09978461
19	5,72773146
20	15,8809120
21	47,63327336
22	129,8626561
23	359,2172771
24	1011,7365338
25	2802,8143937

Tabella 1: Tempo necessario per ricavare le semplificazioni in base al fattore desiderato. Da notare la crescita esponenziale

Numero Semplificazioni	Clock period [ns]	Fmax [MHz]	T cosim [ns]	FF	LUT
1	20	80.80	3990	3819	6804
2	10	137.59	2885	5865	8576
2	30	51.41	2045	3925	7772
2	25	57.55	2105	4214	7780
2	20	81.49	2105	4214	7780
4	100	20.93	2045	3820	9309
4	50	33.6	2045	4158	9377
4	25	55.32	2105	4816	9447

Tabella 2: Esempio di valori delle risorse impiegate (FF e LUT), frequenza e tempo impiegato nella cosimulazione in base al numero di semplificazioni usato nella fase di compressione

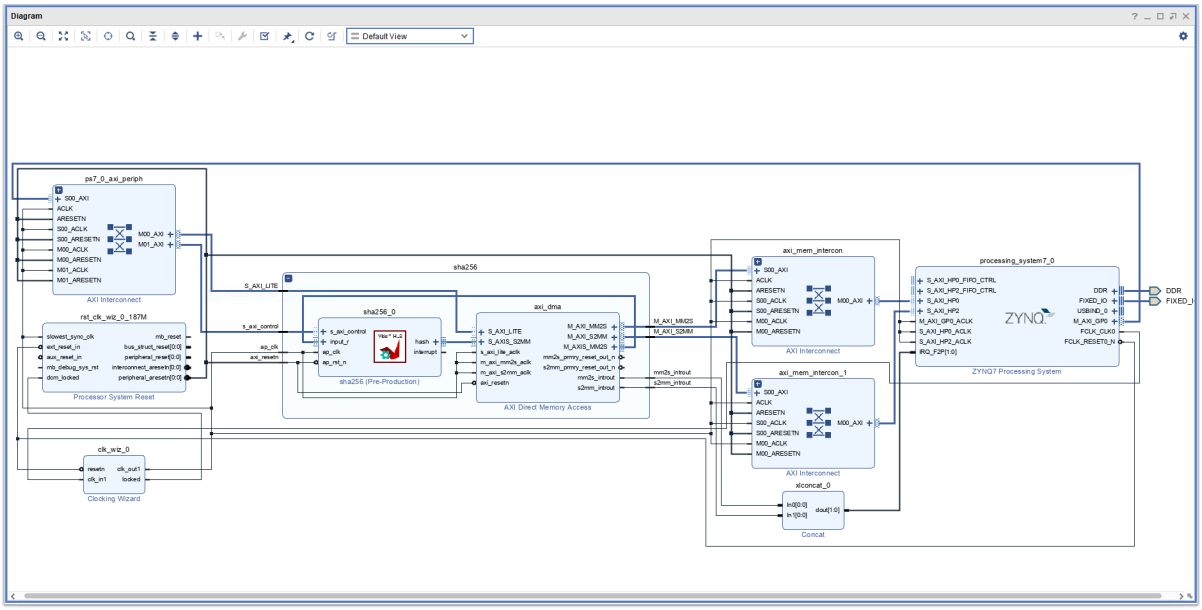


Figura 3: Diagramma per la generazione del bitstream in Vivado

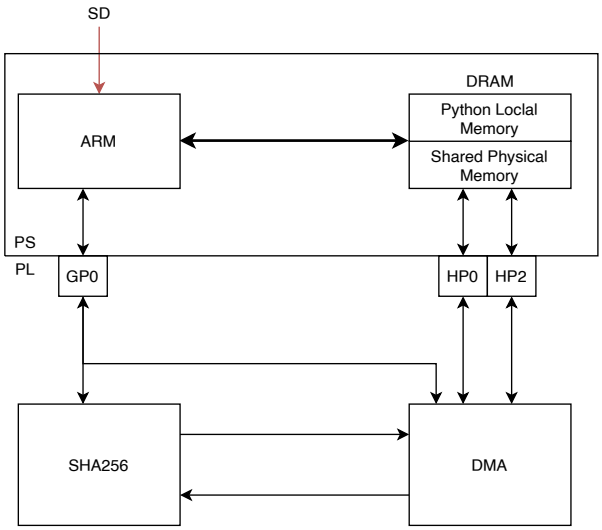


Figura 4: Semplificazione del diagramma precedente

2.3 Driver (PYNQ-Z2)

Utilizzando Vivado ho generato il bitstream ed il file hwh, che verranno poi caricati sul PYNQ per la configurazione dell'FPGA. Il diagramma a blocchi in figura 3 è la rappresentazione dei moduli utilizzati, dove, oltre al IP che ho sintetizzato tramite Vitis HLS, ho utilizzato anche il modulo DMA (Direct Memory Access) per una comunicazione più efficiente tra PS e PL. Una rappresentazione più semplice è riportata in figura 4 dove ho riportato gli elementi più significativi di questa architettura.

Il meccanismo del driver è molto basilare:

- Il file, che si trova sulla microSD, viene letto a blocchi (di dimensione gestibile dalla DRAM del PYNQ) che vengono caricati nella memoria locale utilizzata dall'applicazione
- Ogni blocco viene copiato dalla memoria locale in uno spazio di memoria contigua allocata sempre nella DRAM, questa memoria sarà condivisa con il modulo DMA

- Ogni blocco viene trasferito dalla DRAM alla DMA tramite le High Performance Interfaces (HP)
- Il modulo DMA reindirizza i blocchi verso l'IP SHA-256 dove verranno elaborati
- Una volta letto l'ultimo blocco, l'IP restituirà l'hash al DMA
- Infine, dal DMA l'output verrà trasferito nell'apposita memoria contigua, allocata preventivamente.

Per la vera e propria implementazione del driver ho cercato di seguire le linee guida della documentazione ufficiale di PYNQ, che viene fatta integralmente in Python. Ho realizzato anche un versione del driver in C utilizzando una libreria di terze parti per interagire con il PL.

2.3.1 Driver Python

Il codice viene riportato direttamente sul notebook di jupyter nel file *sha256.ipynb*.

Ho creato diverse versioni di Driver in python:

- versione che segue le linee guida della documentazione PYNQ
- versione che utilizza direttamente una funzione, senza utilizzare la classe alleggerendo lo stack
- versione che sfrutta 2 buffer nella lettura, in modo che mentre un buffer è impegnato nell'invio dei dati da DRAM a DMA l'altro può leggere un secondo blocco, alternandosi fino al completamento della lettura del file.

Le prestazione di tutte le varie versioni sono molto simili, con un leggero miglioramento della versione che sfrutta il buffer.

Il risultato di ogni versione è stato confrontato con la libreria ufficiale per python, hashlib, per testare la correttezza e verificare eventuali miglioramenti dal punto di vista delle prestazioni.

Nome	Time [s]	Hash
Hashlib	2.8261	00A36691822E1309F95DF1283C4C26E351661943BC4F7E83323E53248776E9F6
mySha256 class	2.4248	00A36691822E1309F95DF1283C4C26E351661943BC4F7E83323E53248776E9F6
mySha256 fun	2.4081	00A36691822E1309F95DF1283C4C26E351661943BC4F7E83323E53248776E9F6
mySha256 2buf	1.7823	00A36691822E1309F95DF1283C4C26E351661943BC4F7E83323E53248776E9F6

Tabella 3: Test su un file da 99 MB

Nome	Time [s]	Hash
Hashlib	24.7823	21BE956416A1669FA79B498827FB4D3F24F3CA4DA611523459C5166518459B4F
mySha256 class	25.5942	21BE956416A1669FA79B498827FB4D3F24F3CA4DA611523459C5166518459B4F
mySha256 fun	25.0967	21BE956416A1669FA79B498827FB4D3F24F3CA4DA611523459C5166518459B4F
mySha256 2buf	23.1919	21BE956416A1669FA79B498827FB4D3F24F3CA4DA611523459C5166518459B4F

Tabella 4: Test su un file da 511 MB

Purtroppo, come si può notare dalle tabelle, non ci sono differenze significative in termini di velocità di completamento, ma almeno la frequenza di lavoro del FPGA e nettamente inferiore a quella del processore ARM. Per indagare meglio ho implementato un ultima versione del driver sfruttando il multithreading tramite C.

2.3.2 Driver C

Per interfacciarmi al PL ho utilizzato delle API di terze parti.

Il motivo principale per cui ho creato una versione del driver in C è per la maggior confidenza che ho nella gestione di applicazione multithreading con questo linguaggio. Infatti in questa versione ho utilizzato un architettura produttore consumatore molto semplice. Il produttore legge da file ed invia i blocchi, tramite un ring buffer, al consumatore che a sua volta reindirizza i blocchi direttamente nella Shared Memory, da cui passa alla DMA, risparmiando il passaggio per la memoria locale. In questo modo è possibile misurare le prestazione del produttore e del consumatore separatamente.

Così facendo è facile capire che il collo di bottiglia principale è la lettura da micro SD. Infatti, ad esempio, per un file da 511 MB il produttore impiega 22 secondi per leggere il file, mentre il consumatore ne impiega soltanto 6.

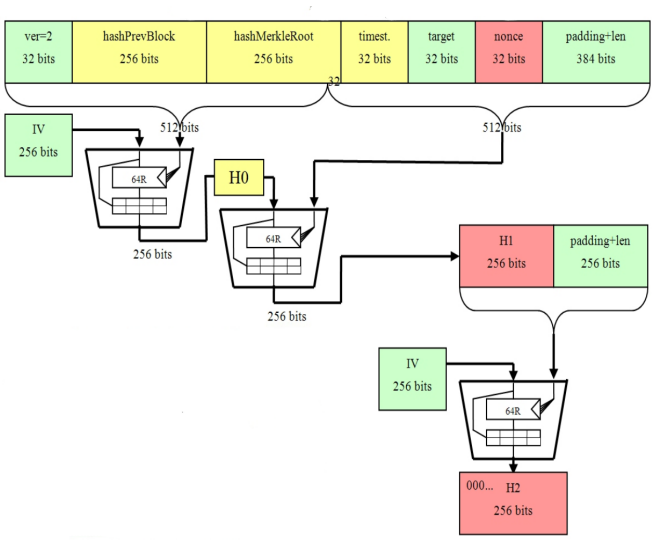


Figura 5: Ottimizzazione SHA256d. (verde: frequenza variazione rara; giallo: frequenza variazione media (ad ogni nuovo blocco); rosso: frequenza variazione elevata;)

Field	Size	Description
version	32 bits	Version of the Bitcoin software version creating this block
hashPrevBlock	256 bits	Hash of the previous block considered as valid in the Bitcoin network (most of the time there is only one candidate)
hashMerkleRoot	256 bits	Here a set of recent yet unconfirmed Bitcoin transactions are hashed into one single value on 256 bits = the Merkle Root
timestamp	32 bits	Current timestamp in seconds since 1970-01-01 00:00 UTC
target	32 bits	The current Target represented in a compact 32 bit format
nonce	32 bits	Nonce chosen by the miner, typically goes from 0x00000000 to 0xFFFFFFFF until the CISO puzzle is solved
padding + len	384 bits	standard fixed SHA256 padding on 384 bits for Len=640 bits

Figura 6: Descrizione delle varie parti

3 Mining con FPGA

Ora, utilizzando le ottimmizzazioni di SHA256 per FPGA fatta, ho cercato di creare un architettura per il mining di criptovalute.

3.1 SHA-256 double (SHA256d)

Come accennato nell'introduzione esistono diverse ottimizzazioni per il mining, la mia implementazione è basata sulle ottimizzazioni di questo articolo.

Come si può notare dal diagramma 5 l'hash H0 dei primi 512 bit rimane invariato durante il mining del blocco, mentre la parte restante dovrà essere ricalcolata ad ogni nuovo Nonce.

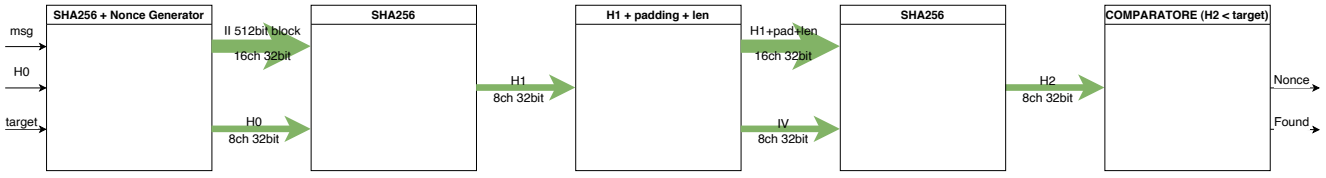


Figura 7: Catena di task per SHA256d

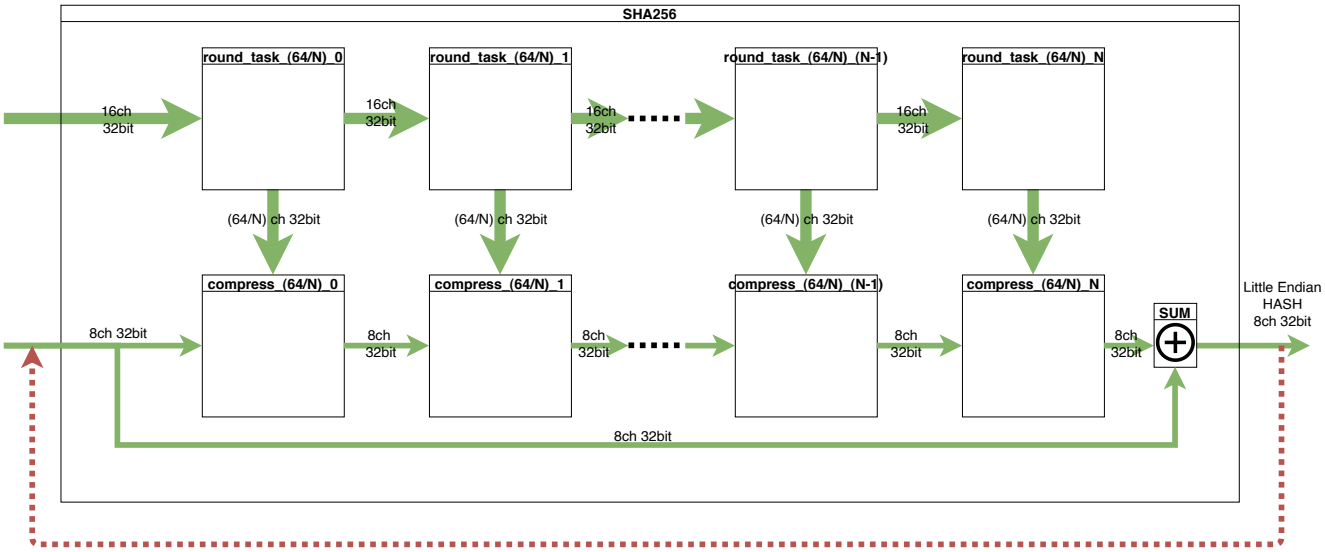


Figura 8: Architettura SHA256 flow per ottenere $hashrate = f_{clk}$.

La strategia è quella di produrre e confrontare un nuovo H2 con il target ad ogni ciclo di clock, perciò ogni blocco della catena in figura 7 deve essere in grado di produrre e consumare il mesaggio alla stessa frequenza della frequenza di funzionamento. ($f_{clk} = hashrate$)

Tutti i task, tranne lo SHA-256, occupano poche risorse ed impiegano poco tempo per essere eseguiti e sono facili da implementare, quindi non molto interessanti.

Perciò si deve trovare un architettura per il blocco SHA-256 che ad ogni clock consumi i 512 bit del messaggio suddiviso in 16 canali da 32 bit ciascuno ed i 256 dello stato in ingresso del hash suddiviso in 8 canali a 32 bit, ed allo stesso tempo produca in uscita l'hash, sempre da 256 bit, corrispondente.

La soluzione adottata è quella di suddividere le 64 iterazioni della compressione in N sub-task, che consuma 64/N parole da 32bit suddivise su altrettanti canali. Le parole sono generate dai round task, che non sono altro che degli shift register da 16 posizioni con uno scorrimento di 64/N ad ogni clock, struttura simile al corrispondente task del diagramma 1. Il diagramma 8 dovrebbe dare un idea dell'architettura utilizzata.

Normalmente, nello SHA-256 ci dovrebbe essere una retroazione, dato che i messaggi potrebbero essere più lunghi di 512 bit, ma in questo caso non è necessario quindi è stato omesso nell'implementazione finale.

Un idea più concreta dovrebbe essere rappresentata dal diagramma 9. In questo caso, ogni blocco si occupa di 16 word alla volta, perciò bastano 4 task per la compressione e 4 task per lo shift.

Quest'architettura rispetta l'obiettivo preposto, ma ad un costo in area molto elevato. Inoltre, se si volesse diminuire il periodo di clock aumentandone di conseguenza l'hashrate, si potrebbe scomporre in task più piccoli, permettendo di ottenere prestazioni più elevate, però il costo è di impiegare più Flip-Flop per la realizzazione di un numero maggiore di stream.

La soluzione finale proposta utilizza 16 task ognuno dei quali si occupano di 4 word alla volta.

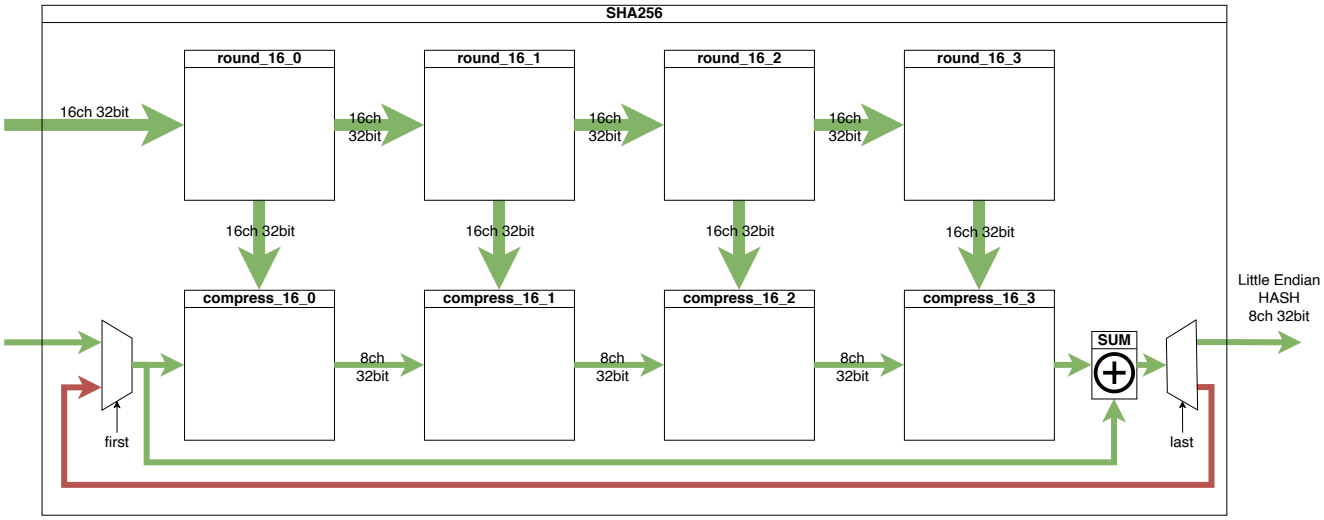


Figura 9: Implementazione concreta dell'architettura SHA256 flow.

```
#define N_TASK 16
#define N_CH 64/N_TASK
#define BIT_N_CH 2

typedef hls::stream<uint> channel_8[8];
typedef hls::stream<uint> channel_16[16];
typedef hls::stream<uint> channel_r2c[N_CH];

void compress_task(channel_r2c &w_in, channel_8 &s_in, channel_8 &s_out, ap_uint<BIT_WCH> &ch ){
  uint in[8], out[8];
  uint w_channel[N_CH];

// UTILE fare ARRAY_PARTITION di in, out e w_channel

  in[0]=s_in[0].read();
  in[1]=s_in[1].read();
  in[2]=s_in[2].read();
  in[3]=s_in[3].read();
  in[4]=s_in[4].read();
  in[5]=s_in[5].read();
  in[6]=s_in[6].read();
  in[7]=s_in[7].read();

  for(int i=0;i<N_CH;i++){
#pragma HLS UNROLL
    w_channel[i]=w_in[i].read();
  }

  compress_4(w_channel, in, out, ch);

  s_out[0]<<out[0];
  s_out[1]<<out[1];
  s_out[2]<<out[2];
  s_out[3]<<out[3];
  s_out[4]<<out[4];
  s_out[5]<<out[5];
  s_out[6]<<out[6];
  s_out[7]<<out[7];
}

void round(hls::stream<uint> w_in[], hls::stream<uint> w_r2c[N_CH], hls::stream<uint> w_r2r[16],
    uint n_task)
{
  uint w_shift[16];
  uint w_channel[16];
  uint w[16];
```

```
  ap_uint<4> i=0;
  uint tmp_w;
#pragma HLS ARRAY_PARTITION variable=w_shift type=complete

  for(int i=0; i<16;i++){
#pragma HLS UNROLL
    w_shift[i]=w_in[i].read();
  }

  shift_register:
  for(int i=0;i<N_CH;i++){
#pragma HLS UNROLL
    tmp_w = sig1(w_shift[14]) + w_shift[9] + sig0(w_shift[1]) + w_shift[0];
    w_r2c[i]<<w_shift[0]+Ki[(N_CH*n_task)+i];
    w_shift[0]=w_shift[1];
    w_shift[1]=w_shift[2];
    w_shift[2]=w_shift[3];
    w_shift[3]=w_shift[4];
    w_shift[4]=w_shift[5];
    w_shift[5]=w_shift[6];
    w_shift[6]=w_shift[7];
    w_shift[7]=w_shift[8];
    w_shift[8]=w_shift[9];
    w_shift[9]=w_shift[10];
    w_shift[10]=w_shift[11];
    w_shift[11]=w_shift[12];
    w_shift[12]=w_shift[13];
    w_shift[13]=w_shift[14];
    w_shift[14]=w_shift[15];
    w_shift[15]=tmp_w;
  }

  if(n_task<(N_TASK)){
    for(int i=0; i<16;i++){
#pragma HLS UNROLL
      w_r2r[i]<<w_shift[i];
    }
  }

}

void sha256_task(hls::stream<uint> w_in[], hls::stream<uint> s_init[], hls::stream<uint> hash[] )
{
  channel_r2c w_r2c[N_CH_R2C];
  channel_16  w_r2r[N_CH_R2R];
  channel_8   s[N_CH_S];
  channel_8   s_prev;

  ap_uint<BIT_WCH> ch=0;
  uint blk=1;

// UTILE fare ARRAY_PARTITION di w_r2c, w_r2r, Ki, s

#pragma HLS DATAFLOW

  split(s_init, s[0], s_prev);
  round(w_in,w_r2c[0], w_r2r[0],blk++);
  compress_task(w_r2c[0], s[0], s[1],ch);

  uint i=1;
  for(i=0; i<N_TASK-1; i++){
#pragma HLS UNROLL
    round(w_r2r[i],w_r2c[i+1], w_r2r[i+1],blk++);
    ch=N_CH*(i+1);
    compress_task(w_r2c[i+1], s[i+1], s[i+2],ch);
  }
  sum_state(hash, s[i+1], s_prev);

}

void sha256_top(uint msg[16], uint target, uint nonce, boolean found){
```

```
117
118   uint message[16];
119
120   for(int i=0; i<16;i++){
121 #pragma HLS UNROLL
122     message[i]=msg[i];
123   }
124
125   hls::stream<uint> dataStream[16];
126   hls::stream<uint> s[8];
127   hls::stream<uint> hash[8];
128
129 //#pragma HLS DATAFLOW
130   generate_init(message, dataStream, s);
131   sha256_task(dataStream, s,hash);
132   compare(hash,target, found, nonce);
133 }
```

Nel funzione top, per questioni di comodità, viene preso in esame il caso dell'applicazione di SHA-256 una sola volta.

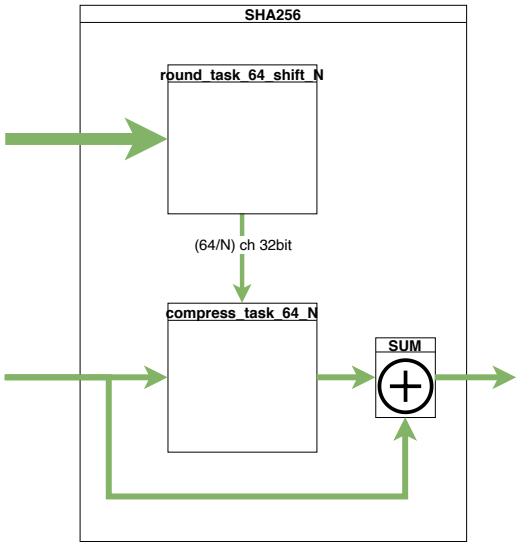


Figura 10: Architettura di SHA256 senza feedback ad area minore

3.2 Riduzione dell'area

Volendo occupare un area minore si potrebbe utilizzare un architettura simile a quella della prima parte con ogni task che si occupa di $64/N$ word alla volta (dove 64/N è il numero di canali impiegato) in un loop da 64 iterazioni con un UNROLL factor=$64/N$. Questo, in termini di prestazioni, si dovrebbe tradurre in un $hashrate = f_{clk}/N$

```
void round_all(hls::stream<uint> w_in[], hls::stream<uint> w_r2c[N_CH])
{
  uint w_shift[16];
  ap_uint<8> i=0;

  uint tmp_w;

  for(int i=0; i<16;i++){
#pragma HLS UNROLL
    w_shift[i]=w_in[i].read();
  }

  for(int j=0;j<N_TASK;j++){
#pragma HLS PIPELINE off
    for(int i=0;i<N_CH;i++){
#pragma HLS UNROLL
      tmp_w = sig1(w_shift[14]) + w_shift[9] + sig0(w_shift[1]) + w_shift[0];
      w_r2c[ch++]<<w_shift[0]+Ki[j*N_CH+i];
      w_shift[0]=w_shift[1];
      w_shift[1]=w_shift[2];
      w_shift[2]=w_shift[3];
      w_shift[3]=w_shift[4];
      w_shift[4]=w_shift[5];
      w_shift[5]=w_shift[6];
      w_shift[6]=w_shift[7];
      w_shift[7]=w_shift[8];
      w_shift[8]=w_shift[9];
      w_shift[9]=w_shift[10];
      w_shift[10]=w_shift[11];
      w_shift[11]=w_shift[12];
      w_shift[12]=w_shift[13];
      w_shift[13]=w_shift[14];
      w_shift[14]=w_shift[15];
      w_shift[15]=tmp_w;
    }
  }
}

void compress_task(channel_r2c &w_in, channel_8 &s_in, channel_8 &s_out ){
```

```
  uint in[8], out[8];
  uint w_channel[N_CH];
  ap_uint<BIT_WCH> ch=0;

  in[0]=s_in[0].read();
  in[1]=s_in[1].read();
  in[2]=s_in[2].read();
  in[3]=s_in[3].read();
  in[4]=s_in[4].read();
  in[5]=s_in[5].read();
  in[6]=s_in[6].read();
  in[7]=s_in[7].read();

  for(int i=0;i<N_CH;i++){
#pragma HLS UNROLL
    w_channel[i]=w_in[i].read();
  }
  for(int i=0; i<N_TASK; i++){
#pragma HLS PIPELINE off
    ch=i*N_CH;
    compress_4(w_channel, in, out, ch);
    in[0]=out[0];
    in[1]=out[1];
    in[2]=out[2];
    in[3]=out[3];
    in[4]=out[4];
    in[5]=out[5];
    in[6]=out[6];
    in[7]=out[7];
  }

  s_out[0]<<out[0];
  s_out[1]<<out[1];
  s_out[2]<<out[2];
  s_out[3]<<out[3];
  s_out[4]<<out[4];
  s_out[5]<<out[5];
  s_out[6]<<out[6];
  s_out[7]<<out[7];
}

void sha256_task(hls::stream<uint> w_in[], hls::stream<uint> s_init[], hls::stream<uint> hash[] ){

  channel_r2c w_r2c[N_CH_R2C];
  channel_16  w_r2r[N_CH_R2R];
  channel_8   s[N_CH_S];
  channel_8   s_prev;

#pragma HLS DATAFLOW

  split(s_init, s[0], s_prev);
  round_all(w_in,w_r2c[0]);
  compress_task_all(w_r2c[0], s[0], s[1]);
  sum_state(hash, s[1], s_prev);

}
```

3.3 SHA-256: schema finale

Nel diagramma 11 sono presenti molte retroazioni, questo elemento ha creato molti problemi durante l'implementazione poiché l'approccio adottato era basato su una documentazione parziale.

3.4 Conclusione

Questa parte è stata prevalentemente teorica, dato che ho potuto implementare e testare solo una parte del modulo HLS per via del limite della board Pynq. In ogni caso, anche se si potesse implementare sul FPGA non avrebbe molto senso impiegarlo per il minig vero e proprio dato che esistono schede ASIC che sono nettamente più efficienti, ma a livello di progettazione è stato un ottimo caso di studio.

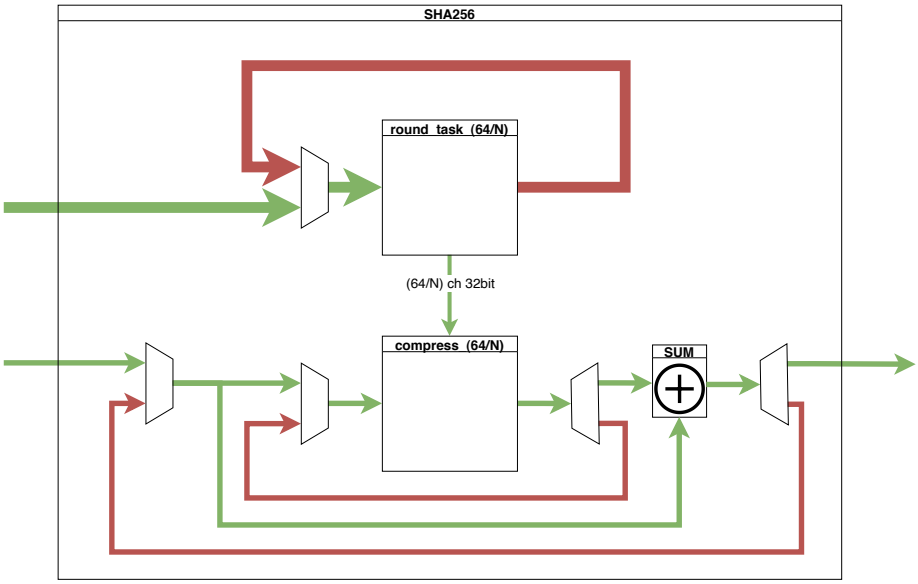


Figura 11: Architettura completa di SHA256 ad area minore

Infatti, quest'attività progettuale mi ha permesso di ragionare più nel dettaglio sulle possibili varianti implementative, apprendendo in questo modo alcune nozioni utili anche in progetti di natura molto diversa.

Inoltre, ha fatto emergere l'importanza nel dare il giusto spazio alla lettura approfondita della manualistica e delle fonti principali.