Algoritmo SHA-2

Implementazione in VHDL

Oscar Locatelli
Politecnico di Milano

Ingegneria - V Facoltà
Laurea Specialistica
Ingegneria Informatica

Progetto per il corso di Laboratorio Software
m.745939

Sommario

Descrizione del problema e scelte di progetto

Problema

Obbiettivo del progetto è l'implementazione in VHDL dell'algoritmo crittografico di hashing Secure Hash Algorithm versione 2 (SHA-2), del quale fornirò solo informazioni parziali e utili alla comprensione del mio operato. La trattazione completa è disponibile sul portale del NIST [1].

Da un messaggio di lunghezza arbitraria ($<2^{64}$) l'algoritmo genera una *hash* di lunghezza fissata, pari in bit al numero presente nel nome delle quattro varianti dell'algoritmo SHA-224, SHA-256, SHA-384, SHA-512.

E' prevista una fase di *preprocessing* (o *padding*) nella quale al messaggio viene accodato un bit a `1` e tanti bit a `0` fino a portarne la lunghezza a un valore che diviso per 512 dia resto 448 (cioè 512 – 64) e poi accodata la lunghezza del messaggio originale (64 bit).

Il messaggio viene diviso in blocchi da 512 bit e l'algoritmo viene applicato a cascata sui blocchi sequenzialmente, utilizzando il risultato ottenuto dal blocco precedente come dato di ingresso per il calcolo successivo. Ogni blocco passa per tre fasi: l'*espansione del dato*, il *ciclo di compressione* e l'*elaborazione della hash* (o del *digest* intermedio).

La fase più critica per l'implementazione è quella di compressione che consiste in un loop di 64 passi (80 per SHA-384/512) dipendenti tra loro e quindi *non parallelizzabili*. Anche l'espansione nella stesura ufficiale dell'algoritmo è un ciclo, tuttavia la si può effettuare senza perdita di cicli di clock durante il caricamento del blocco tramite un buffer e un puntatore su di esso.

Ho applicato altre modifiche e ottimizzazioni (*rischeduling, pipeline tra le fasi dei diversi blocchi, ridondanze*) per ridurre il ritardo totale, alcune tratte dai molti documenti disponibili in rete, cui riporto i principali nella bibliografia [2] [3], altre introdotte da me su misura del mio progetto. Ho esaminato e scartato numerose altre possibilità (*unrolling, quasi-pipeline, addizionatori speciali*) poiché non adatte alla mia architettura. [4] [5] [6] [7]

Scelte di Progetto

L'idea era quella di creare un componente il più *semplice* possibile da utilizzare, al quale passare il messaggio un *chunk* (porzione) alla volta senza attendere nessun segnale di sincronizzazione. Una volta giunto alla fine del messaggio monitorare un segnale d'uscita in attesa della hash. Tutto senza preoccuparsi di "preparare" il messaggio tramite padding manuale (nei documenti in rete questa fase è sempre lasciata al processo dell'utilizzatore).

Inizialmente, dopo aver studiato le specifiche del NIST, pensavo di poter creare un'interfaccia a 512 bit a cui passare un intero blocco ogni ciclo di clock ed elaborarlo in *tempo reale*. Avrei così ottenuto il risultato con un solo ciclo di clock ritardo dalla fine del messaggio. Il problema della compressione (non parallelizzabile) mi ha però costretto ad abbandonare la strategia a singolo clock a favore di quella a "un passo ogni ciclo di clock", pena l'esplosione del numero di componenti da utilizzare e conseguente blocco dello strumento di sintesi.

C'erano tre possibilità: Passare tutto il messaggio al componente e salvarlo in un buffer con successiva elaborazione e attesa del risultato. Facile ma con ovvi limiti imposti sulla lunghezza del messaggio ben inferiori alla capacità dell'algoritmo (2^{64} bit).

Seconda ipotesi, introdurre un tempo d'attesa tra il passaggio di due blocchi consecutivi, cioè dopo aver fornito 512 bit, bloccare il processo utilizzatore per diversi cicli di clock (o un segnale *ready*) in attesa di ottenere il *digest* parziale, per poi procedere con il prossimo treno di bit. Nessun limite né di memoria, né di altri tipi, ma cade l'idea principale, la facilità d'utilizzo.

La strada che ho scelto prevede l'elaborazione di un blocco in contemporanea alla lettura del successivo, in una sorta di pipeline tra le fasi, tramite l'utilizzo di un buffer a scorrimento da 512 bit per la memorizzazione dei dati da elaborare e quelli nuovi appena letti. Le istruzioni per l'utilizzatore così non cambiano, deve solo passare il messaggio senza interruzioni e attendere il risultato finale.

C'è un limite intrinseco a questo tipo di architettura, il sincronismo tra le fasi, o almeno un numero di cicli di lettura pari o superiore a quello necessario alle fasi interne, condizione necessaria per poter mantenere fissa la dimensione del buffer così da non avere vincoli di memoria (e quindi di lunghezza del messaggio) e per non costringere l'utilizzatore a fare delle pause tra un blocco e l'altro.

Tramite alcune ridondanze (segnali `a_2-h_2`) sono riuscito a "inglobare" l'elaborazione finale del digest nella fase di compressione, quindi il numero di cicli della fase di lettura deve essere pari a solo quelli necessari alla compressione che sono rispettivamente 64 e 80 per SHA-224/256 e SHA-384/512.

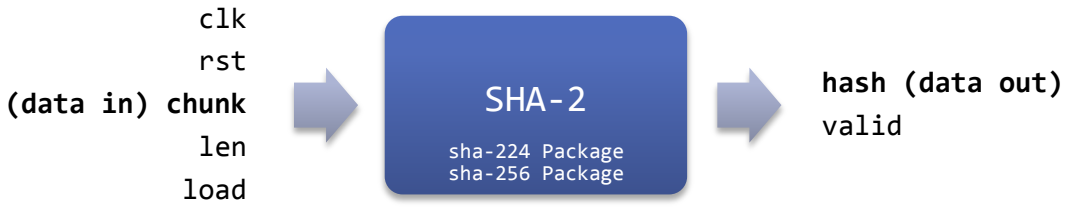
La dimensione dell'interfaccia di ingresso (il chunk di blocco passato ad ogni ciclo di clock) dipende da dagli step e più precisamente segue questa formula: *Dimensione blocco / step = Dimensione chunk*.

Nel caso di SHA-224/256: $512_{bit} / 64_{cicli} = 8_{bit/ciclo}$, ovvero 8 bit di interfaccia.

Per SHA-384/512: $512_{bit} / 80_{cicli} \geq 6_{bit/ciclo}$, ovvero massimo 6 bit di interfaccia.

Ed ecco il vincolo di questa soluzione: una implementazione unica non può valere per tutte le sottofamiglie dell'SHA-2, tuttavia l'idea non cambia, vanno solo fatte delle modifiche marginali, ad esempio dimezzare la dimensione del chunk di input (128 cicli di lettura e 80 di compressione) o fare un loop unrolling 2x della compressione (64 e 40) e usare cicli di attesa interna per "aspettare" la lettura per cui, come scelta di progetto, ho deciso di sviluppare solo la soluzione per l'SHA-224/256.

Interfacce



Input

`clk`: segnale di clock

`rst`: segnale di reset

`chunk`: prossimi 8 bit del messaggio

`len`: lunghezza del chunk, sempre 8 tranne eventualmente per l'ultimo chunk

`load`: segnale di caricamento, alto fintantoché si carica il messaggio, basso dopo l'ultimo chunk

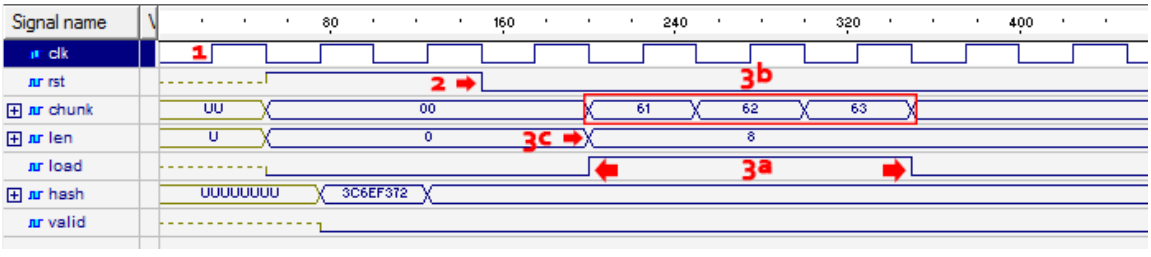
Output

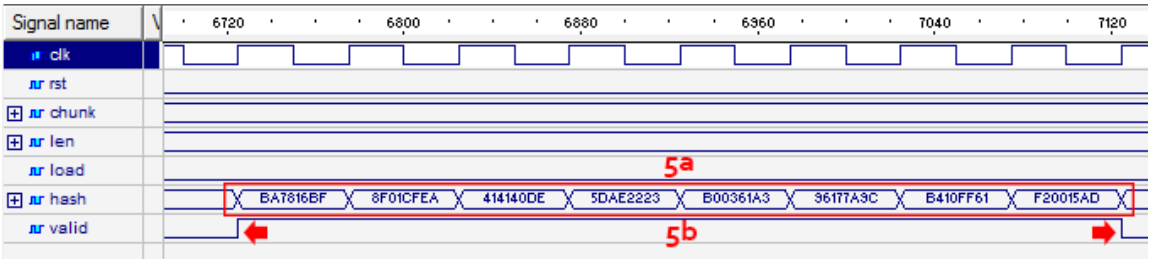
`hash`: alla fine dell'elaborazione contiene in successione le parole da 32 bit della hash calcolata (7 per l'SHA-224, hash da 224 bit e 8 per l'SHA-256, hash da 256 bit)

`valid`: segnale di validità dell'uscita, alto nei cicli di clock in cui le parole della hash sono presenti sul segnale d'uscita (`hash`)

Procedura di utilizzo

1- Applicare un segnale di clock (testato a 50 Mhz) al segnale `clk`, il componente è sincronizzato sul fronte di salita.
2- Resettare il componente ponendo il segnale `rst` alto per almeno un ciclo di clock, poi abbassarlo.
3- A questo punto alzare il segnale load e scrivere byte per byte (in sincrono con il clock) il messaggio sul segnale `chunk` e su `len` la lunghezza effettiva del chunk. L'architettura non richiede ne permette pause nel caricamento e i chunk devono sempre essere pieni (len = 8) ad eccezione dell'ultimo, che può avere lunghezza minore, per supportare messaggi binari o codifiche non allineate al byte.
4- Finito il caricamento abbassare il segnale `load` e attendere il completamento dell'elaborazione.
5- Una volta calcolata la hash verrà scritta sull'omonimo segnale (`hash`) una word da 32 bit per ciclo. Nel mentre il segnale `valid` sarà alto, per indicare al processo utilizzatore la presenza del risultato, dopodiché varrà riportato a stato logico basso e il componente tornerà in stato *idle*.
6- Se si vuole procedere con un altro messaggio basta riprendere il caricamento tornando al punto 3, senza effettuare alcun reset.





Architettura

I componenti

- `rom0`, la rom contenente la tabella costante K
- `step_count`, contatore degli step di caricamento e compressione (64)
- `out_count`, contatore delle word della hash da caricare sul segnale d'uscita `hash` (7 per sha-224, 8 per sha-256)

I segnali

- `clk`, segnale di clock
- `rst`, segnale di reset
- `step`, valore corrente contatore `step_count`
- `state`, variabile contenente lo stato corrente:
  - `s_idle`, in attesa
  - `s_load`, sta caricando il messaggio
  - `s_add_block`, eventuale aggiunta di un blocco se (messaggio + 1) modulo 512 > 448
  - `s_padding`, aggiunge gli zeri e la lunghezza per chiudere il messaggio da elaborare
  - `s_compute`, sta finendo di calcolare la hash del blocco
- `load`, indica che sto caricando un nuovo chunk
- `load_pad`, indica se sto caricando il buffer, alto se sto caricando o lo stato è `s_load`, `s_add_block` o `s_padding`
- `loading`, load_pad ritardato di 1 ciclo (per generare `loaded`)
- `loaded`, load_pad ritardato di 2 cicli (per generare `load_rst`)
- `load_rst`, indica l'inizio del caricamento di un nuovo messaggio, `(loaded xor load_pad) and load_pad`
- `step_rst`, reset del conteggio degli step, `load_rst` or `rst`
- `first`, indica se è il primo blocco del messaggio
- `first2`, first ritardato di 1 ciclo
- `blk_num`, numero di blocchi di cui è composto il messaggio più padding
- `blk_ok`, numero di blocchi elaborati
- `blk_vld`, blocco caricato
- `chunk`, nuovo chunk del messaggio in input
- `len`, lunghezza del chunk (sempre 8 tranne eventualmente l'ultimo)
- `msg_len`, lunghezza totale del messaggio originario (calcolata sommando le len)
- `buf`, buffer a scorrimento utilizzato per salvare i chunk gli ultimi 512 bit di messaggio + padding caricati, necessario per il pipelining
- `new_w`, nuova word presa dal buffer da usare per l'espansione
- `cal_w`, nuova word calcolata in uno step precedente e da usare per gli step successivi dell'espansione
- `w`, word da usare per l'espansione (da `new_w` o `cal_w`)
- `wnd`, buffer interno contenente i dati del blocco espanso
- `k`, variabile K[step] caricata dalla `rom0`
- `h_vld`, hash elaborata segnale interno
- `valid`, hash elaborata segnale di output
- `w_out`, valore corrente contatore `out_count`
- `hash`, contiene una parola da 32 bit alla volta della hash, quando `valid` è alto
- `a-h`, variabili interne per la compressione
- `a_2-h_2`, ridondanza del digest del blocco precedente da sommare a `a-h` solo all'ultimo step per diminuire di 1 ciclo il ritardo di ogni elaborazione blocco e mantenere il sincronismo con il caricamento
- `s0,s1,maj,ch,th0,th1`, funzioni cablate interne per la compressione
- `h0-7`, digest risultato di ogni elaborazione blocco e word della hash finale a fine elaborazione

I processi

- `first_blk`, imposta i segnali di primo blocco (`first` e `first2`), la conversione è bloccata per il riempimento buffer
- `blk_counter`, conta i blocchi e imposta i segnali `blk_num`, `blk_ok`, `blk_vld` e `h_vld`
- `blk_validity`, imposta il segnale di fine elaborazione blocco (`blk_vld`) quando `step` è al valore massimo (63)
- `p0_fill_buffer_and_padding`, fase di caricamento del prossimo blocco e di preprocessing (padding) a fine messaggio
- `fill_word`, carica la prossima word da usare nella fase di espansione (`w`) prendendola da new_w (che arriva dal buffer e quindi è una word del blocco caricato) nel primi 16 step e da cal_w (parola precalcolata in fase di espansione) negli altri 48 step
- `p1_expansion`, fase di espansione: espande il blocco nella finestra di espansione `wnd` usando la word `w` caricata nel process `fill_word` al ciclo precedente
- `p2_compression`, fase di compressione: calcola i segnali `a-h` e i segnali `a_2-h_2` (validi solo all'ultimo step di compressione e sommati ad `a-h` per ridurre il ritardo)
- `p3_digest_computation`, fase finale di calcolo del nuovo digest (dal quale all'ultimo blocco si estrae la hash)

Data flow

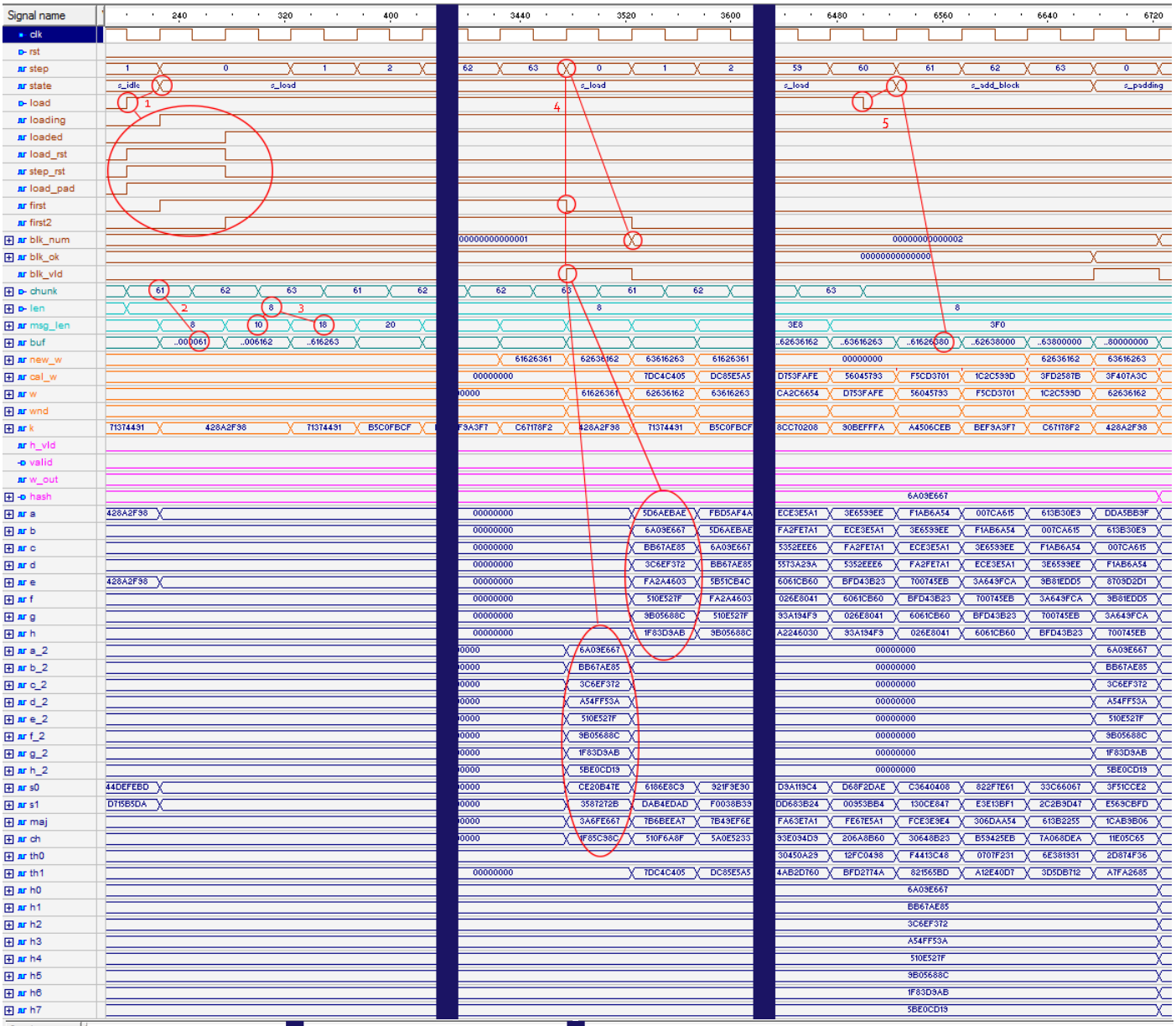
Analizzando il path "percorso" dai dati, dal messaggio originale alla hash risultato, si può distinguere queste catene:

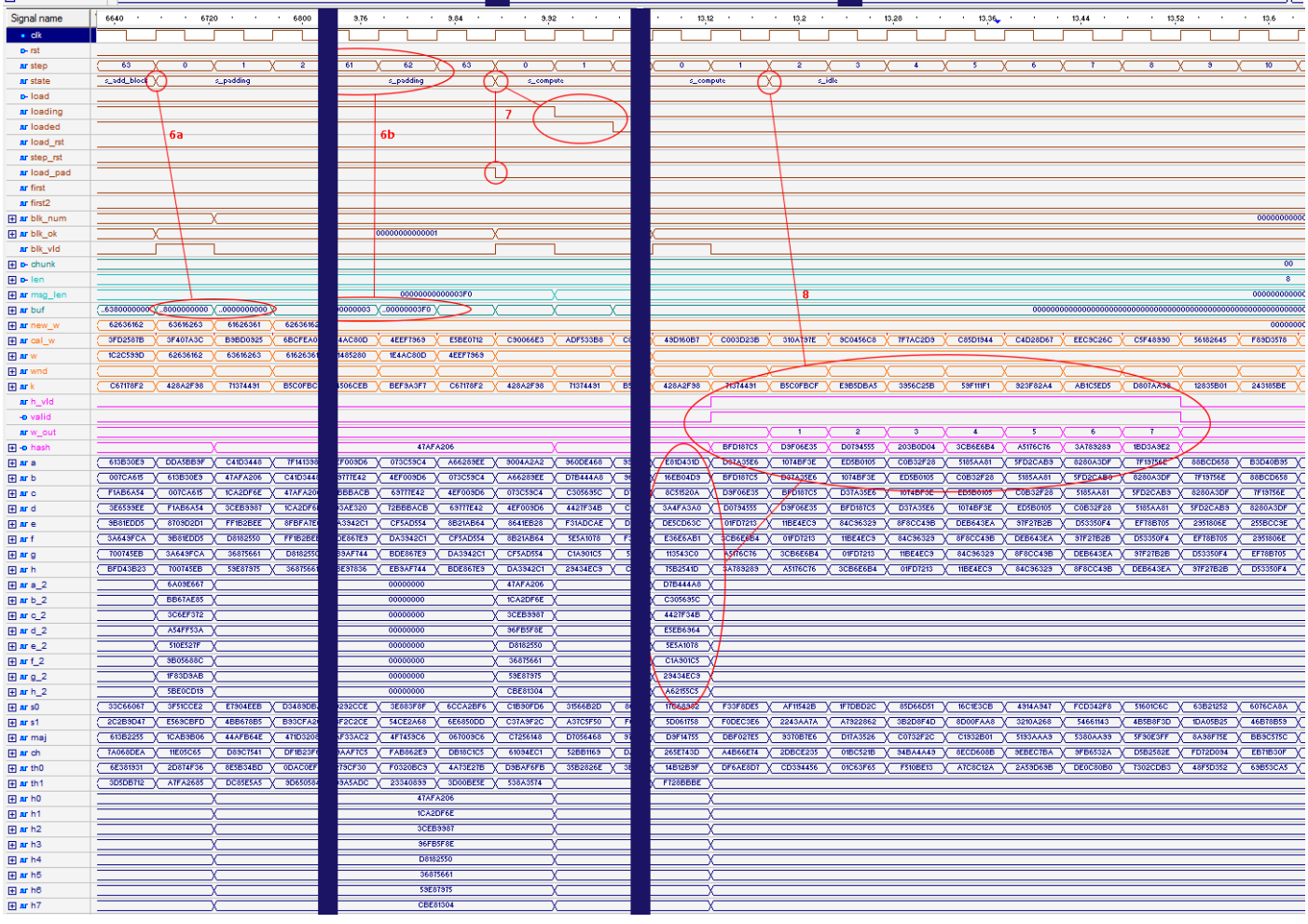
- caricamento: `chunk -> buf -> new_w -> w -> wnd`
- espansione: `wnd -> cal_w -> w -> wnd`
- compressione: `wnd -> a-e,s0,s1,maj,ch,th0,th1 -> h1-h7 -> hash`

Con riferimento le figure qui sotto, che mostrano spezzoni di "waveform" di uno dei miei test (il messaggio è la stringa di 126 caratteri con pattern "abcabc…") ora mostrerò le relazioni fondamentali tra i segnali interni del componente.

1- Il segnale load influenza vari segnali interni che servono per la sincronizzazione e abilitazioni di vari process:
   - `state` = `s_load` (sto caricando) [process po_fill_buffer_and_padding, riga 285]
   - `loading`, `loaded`, `load_rst`, `step_rst`, `load_pad`, `first` e `first2`, vengono inizializzati (vedere la tabella segnali) [process fill_word, righe 364-365; connessioni persistenti, righe 186-187; process first_block]
2- Il messaggio viene scritto nel segnale `chunk` 8 bit a ciclo e caricato nel buffer a scorrimento `buf` [process po_fill_buffer_and_padding, righe 345-346]
3- La lunghezza del chunk corrente presente nel segnale `len` viene sommata nell'accumulatore `msg_len`, che a fine caricamento conterrà la lunghezza totale del messaggio [process po_fill_buffer_and_padding, righe 289, 294]
4- Dopo 64 cicli ho il primo blocco (e così i successivi) è stato caricato (segnalato da `blk_vld`), il numero di clocchi viene aumentato di uno (`blk_num`), i segnali `first` e `first2` vengono abbassati in 2 cicli e può cominciare l'elaborazione. I segnali a-h verranno aggiornati di continuo fino alla fine dell'elaborazione [process blk_validity; process blk_counter; process first_block; process p2_compression]
5- A messaggio caricato il segnale di load verrà posto a `0` dall'utilizzatore e viene aggiunto un bit a `1` nel buffer e viene calcolato il resto della divisione della lunghezza del messaggio (in msg_len) per 512. Se questo risulta minore di 448, si passa direttamente allo stato `s_padding`. Altrimenti è necessario aggiungere un ulteriore blocco per accodare la lunghezza (64 bit). Quindi lo stato viene posto a `s_add_block` e si cominciano ad accodare un bit a `0` al buffer fino alla fine del ciclo di 64 step di caricamento del blocco, dopo di che si passa allo stato `s_padding`. [process po_fill_buffer_and_padding, righe 291-316]
6- La fase di padding prevede l'accodamento di zeri nel buffer fino agli ultimi 4 step (in realtà tra lo step 59 e 62), nei quali viene scritta nel buffer la lunghezza totale. La fase di caricamento è conclusa [process po_fill_buffer_and_padding, righe 321-328]
7- Finito il padding si passa allo stato `s_compute` e vengono abbassati i segnali `load_pad`, `loading` e `loaded`. In questa fase vengono portati a termine i calcoli sull'ultimo blocco. [process po_fill_buffer_and_padding, riga 322; process fill_word, righe 364,365; connessione persistente, riga 181; process p2_compression]
8- Infine dopo 64 (o 128 in caso di necessità di aggiunta di un blocco nel punto 5) cicli dall'ultimo blocco caricato la hash sarà pronta, calcolata commando i segnali `a-h`, che contengono i risultati della compressione dell'ultimo blocco e il digest `h0-h7` del blocco precedente. Il segnale `h_vld`, e quindi il segnale `valid` di output, passano a stato alto e sul segnale di output hash vengono scritte le 8 (o 7 per sha-224) word della hash calcolata. Dopodiché il segnale `h_vld` e `valid` tornano a zero. Lo stato viene posto a `s_idle`. [process p2_compression; process p3_digest_computation; process blk_counter, righe 236-246; process po_fill_buffer_and_padding, riga 338]

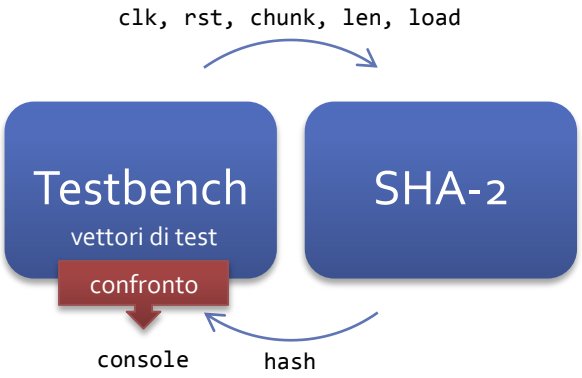
La conversione è finita.


Signal name
clk
rst
step
state
load
loading
loaded
load_rst
step_rst
load_pad
first
first2
blk_num
blk_ok
blk_vld
chunk
len
msg_len
buf
new_w
cal_w
w
wnd
k
h_vld
valid
w_out
hash
a
b
c
d
e
f
g
h
a_2
b_2
c_2
d_2
e_2
f_2
g_2
h_2
s0
s1
maj
ch
th0
th1
h0
h1
h2
h3
h4
h5
h6
h7
1
2
3
4
5


Signal name
clk
rst
step
state
load
loading
loaded
load_rst
step_rst
load_pad
first
first2
blk_num
blk_ok
blk_vld
chunk
len
msg_len
buf
new_w
cal_w
w
wnd
k
h_vld
valid
w_out
hash
a
b
c
d
e
f
g
h
a_2
b_2
c_2
d_2
e_2
f_2
g_2
h_2
s0
s1
maj
ch
th0
th1
h0
h1
h2
h3
h4
h5
h6
h7
6a
6b
7
8

Test e verifica

Per verificare il corretto funzionamento del componente ho creato un testbench direttamente nel progetto VHDL che simula il processo utilizzatore.



Utilizzando il tool di sviluppo Active-HDL si può avere un riscontro sia automatico che visivo.

Nel testbench ho riportato una serie di vettori di test con la relativa lunghezza e hash di controllo precalcolata da me con generatori trovati sul web [8] [9]. Basta scegliere quale testare (facendo attenzione a includere nel progetto solo il package di costanti per sha-256 o sha-224) e lanciare la simulazione.

Il controllo manuale/visivo lo si può fare creando una waveform e aggiungendo i segnali utili (come nelle figure riportate nella sezione precedente).

Inoltre viene appunto fatto un controllo automatico e nella console di output a fine elaborazione (o ciclicamente se attivata l'opzione doLoop) compare un messaggio di conferma se il risultato equivale alla hash precalcolata o di errore in caso contrario.

Esempi dei messaggi nella console:

OK:

```
# EXECUTION:: NOTE    : --------- COMPUTE HASH OK --------
# EXECUTION:: Time: 192350 ns,  Iteration: 0,  TOP instance,  Process: test.
# EXECUTION:: NOTE    : BFD187C5D9F06E35D0794555203B0D043CB6E6B4A5176C763A7892891BD3A9E2
# EXECUTION:: Time: 192350 ns,  Iteration: 0,  TOP instance,  Process: test.
```

ERRORE:

```
# EXECUTION:: NOTE    : --------- COMPUTED: --------
# EXECUTION:: Time: 13550 ns,  Iteration: 0,  TOP instance,  Process: test.
# EXECUTION:: NOTE    : BFD187C5D9F06E35D0794555203B0D043CB6E6B4A5176C763A7892891BD3A9E3
# EXECUTION:: Time: 13550 ns,  Iteration: 0,  TOP instance,  Process: test.
# EXECUTION:: NOTE    : --------- EXPECTED: --------
# EXECUTION:: Time: 13550 ns,  Iteration: 0,  TOP instance,  Process: test.
# EXECUTION:: NOTE    : BFD187C5D9F06E35D0794555203B0D043CB6E6B4A5176C763A7892891BD3A9E2
# EXECUTION:: Time: 13550 ns,  Iteration: 0,  TOP instance,  Process: test.
# EXECUTION:: FAILURE: --------- COMPUTE HASH FAILED --------
# EXECUTION:: Time: 13550 ns,  Iteration: 0,  TOP instance,  Process: test.
```

Nel testbench vi sono anche altre possibili configurazioni, ad esempio doLoop per effettuare conversioni cicliche, o in caso la lunghezza del vettore fosse zero, viene passata al componente una stringa composta si soli caratteri 'a' di lunghezza pari alla variabile num_a.

Bibliografia *(seguire anche i documenti collegati agli articoli citati)*

1. **NIST.** Federal Information Processing Standards Publication 180-3, Secure Hash Standards (SHS). [Online] October 2008. FIPS PUB 180-3.

2. **R. Chaves, G. Kuzmanov, L. A. Sousa, and S. Vassiliadis.** Improving SHA-2 hardware implementations. *Workshop Cryptograph. Hardw. Embedded Syst. (CHES).* October 2006. p. 298–310.

3. **V. H. Hanumakumar, et. al.** Cost-Efficient SHA Hardware Accelerators Using VHDL. *(IJAEST) International Journal of Advanced Engineering Sciences and Technologies.* Vol. 5, 1, p. 37-52.

4. **R. P. McEvoy, F. M. Crowe, C. C. Murphy and W. P. Marnane.** Optimisation of the SHA-2 family of hash functions on FPGAs. *IEEE Computer Society Annual Sym- posium on Emerging VLSI Technologies and Architectures (ISVLSI'06).* 2006. p. 317–322.

5. **I. Ahmad, A. S. Das.** Hardware implementation analysis of SHA-256 and SHA-512 algorithms on FPGAs. *Computers and Electrical Engineering.* 2005. Vol. 31, 6, p. 345-360.

6. **L. Dadda, M. Macchetti, J. Owen.** The Design of a High Speed ASIC Unit for the Hash Function SHA-256 (384, 512). *DATE.* s.l. : IEEE Computer Society, 2004. p. 70-75.

7. **M. Macchetti, L. Dadda.** Quasi-pipelined hash circuits. *IEEE Symposium on Computer Arithmetic.* s.l. : IEEE Computer Society, 2005. p. 222–229.

8. http://www.fileformat.info/tool/hash.htm. *Online SHA-256 Converter from ASCII and HEX string.* [Online]

9. Online SHA-224 Converter from ASCII string. *http://www.miniwebtool.com/sha224-hash-generator.* [Online]