# ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΑΤΡΩΝ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΤΕΧΝΟΛΟΓΙΑΣ ΥΠΟΛΟΓΙΣΤΩΝ

ΕΡΓΑΣΤΗΡΙΟ VLSI ΙΙΙ


## Secure Hash Algorithm – 256
## SHA-256


ΟΜΑΔΑ

| | |
|---|---|
| ΜΑΪΣ ΜΥΡΙΑΜ | 1083745 |
| ΜΠΟΥΡΤΣΟΥΚΛΗΣ ΘΕΟΔΟΣΙΟΣ | 1083891 |


ΠΑΤΡΑ, 24/02/2025

# Περιεχομενα

# 1. SHA-256 Βασικες Αρχες

Ο SHA -256 είναι ένας Hash αλγόριθμος ο οποίος δέχεται ως είσοδο μία λέξη μήκους l από 1 έως $2^{64}$ bits και παράγει στην έξοδο μία λέξη των 256 bits η οποία για ίδιο input word παράγει το ίδιο Hash. Η πιθανότητα 2 διαφορετικές λέξεις εισόδου να παράγουν ίδιο hash είναι πολύ μικρή.

Ο αλγόριθμος αυτός χρησιμοποιεί 6 διαφορετικές λογικές συναντήσεις:

- $Ch(x,y,z) = (x \wedge y) \oplus (\neg x \wedge z)$
- $Maj(x,y,z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z)$
- $\sum_0^{\{256\}}(x) = ROTR^2(x) \oplus ROTR^{13}(x) \oplus ROTR^{22}(x)$
- $\sum_1^{\{256\}}(x) = ROTR^6(x) \oplus ROTR^{11}(x) \oplus ROTR^{25}(x)$
- $sigma_0^{\{256\}}(x) = ROTR^7(x) \oplus ROTR^{18}(x) \oplus SHR^3(x)$
- $sigma_1^{\{256\}}(x) = ROTR^{17}(x) \oplus ROTR^{19}(x) \oplus SHR^{10}(x)$

Όπου $ROTR^n(x)$ είναι κυκλική δεξιά ολίσθηση κατά n θέσεις και $SHR^n(x)$ απλή δεξιά ολίσθηση n θέσεων.

Επίσης ο αλγόριθμος περιλαμβάνει 2 σετ σταθερών. Το 1ο έχει 64 σταθερές των 32 bit η καθεμία και το 2ο 8 σταθερές των 32 bit η μία.

## 1.1 Διαδικασια υπολογισμου

### 1ο Βήμα (Padding)

Το μήνυμα με μήκος l, το σπάμε σε block τα οποία τα ονομάζουμε Messages ($M^{(n)}$), στο τελευταίο message κολλάμε στο τέλος ένα 1 και τόσα 0 τέτοια ώστε $M^{(n)} + 1 + k*0 + 64 = 512$. Όπου τα τελευταία 64 bits δηλώνουν το μήκος του l.

### 2ο Βήμα (Parsing)

Αφού η λέξη εισόδου έχει χωριστεί σε n Messages το κάθε Message σπάει σε 16 λέξεις των 32 bits τα οποία συμβολίζονται ως $M_i^{(n)}$ , με i = 0,1,2,…,15 .

## 3ο Βήμα (Message Schedule)

Με κάθε πίνακα των 16 32bit λέξεων υπολογίζουμε μέσω της σχέσης

$$W_t = \begin{cases} M_t^{(i)} & 0 \le t \le 15 \\ \sigma_1^{\{256\}}(W_{t-2}) + W_{t-7} + \sigma_0^{\{256\}}(W_{t-15}) + W_{t-16} & 16 \le t \le 63 \end{cases}$$

Τα πρώτα 16 messages μένουν ως έχουν και τα υπόλοιπα 48 δημιουργούνται με την παραπάνω σχέση.

## 4ο Βήμα (SHA compute)

Αφού έχουμε το message schedule αρχικοποιούμε τα σήματα abcdefgh με το 2ο σετ σταθερών ($H^0$) και τρέχουμε για 64 φορές την παρακάτω πράξη.

$$T_1 = h + \sum_1^{\{256\}}(e) + Ch(e,f,g) + K_t^{\{256\}} + W_t$$

$$T_2 = \sum_0^{\{256\}}(a) + Maj(a,b,c)$$

$$h = g$$

$$g = f$$

$$f = e$$

$$e = d + T_1$$

$$d = c$$

$$c = b$$

$$b = a$$

$$a = T_1 + T_2$$

Παρατηρούμε ότι σε κάθε λούπα χρησιμοποιούμε 1 Wt και 1 Kt , όπου Kt η t-οστη σταθερά του 1ου σετ σταθερών.

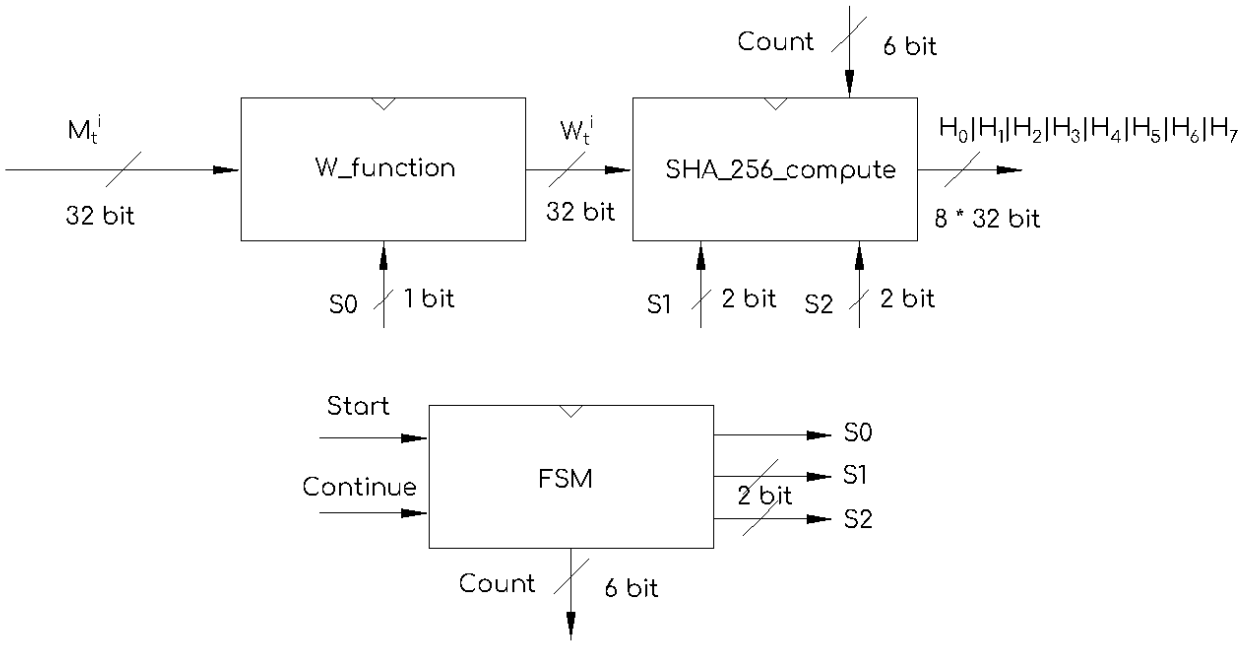
Αφού ολοκληρωθούν όλες οι επαναλήψεις ορίζουμε το $H^i$ = abcdefgh + $H^{i-1}$ . Αν έχουμε και άλλο message συνεχίζουμε την λούπα με αρχικοποιημένα abcdefgh όμως με το προηγούμενο H που φτιάξαμε, αλλιώς το Hash είναι έτοιμο και ορίζεται ως η συνένωση των H : **$H_0|H_1|H_2|H_3|H_4|H_5|H_6|H_7$**

# 2. Λογικη Υλοποιησης

Με βάση την αρχή λειτουργίας του αλγορίθμου δημιουργήσαμε την δική μας αρχιτεκτονική.

Η υλοποίηση ξεκίνησε από το **βήμα 3** καθώς θεωρήσαμε ότι το padding και το parsing γίνονται από εξωτερικό επεξεργαστή ο οποίος μας στέλνει έτοιμο το parsed μήνυμα. Η αποστολή γίνεται σειριακά με μία 32bit λέξη ( $M_i^{(n)}$ ) να έρχεται στην είσοδο ανά clock. Την λέξη αυτή παίρνει το κύκλωμα W_function το οποίο σε κάθε clock βγάζει και ένα Wt στην έξοδό του το οποίο μπαίνει ως είσοδος στην επόμενη μονάδα SHA256_compute. Η οποία υπολογίζει όπως προαναφέρθηκε με τις λούπες το hash (Περισσότερες λεπτομερείς στα επόμενα κεφάλαια).

Το Block Diagram της υλοποίησης παρουσιάζεται παρακάτω.




*Σχήμα 1. Block Diagram SHA-256*

# 3. FSM

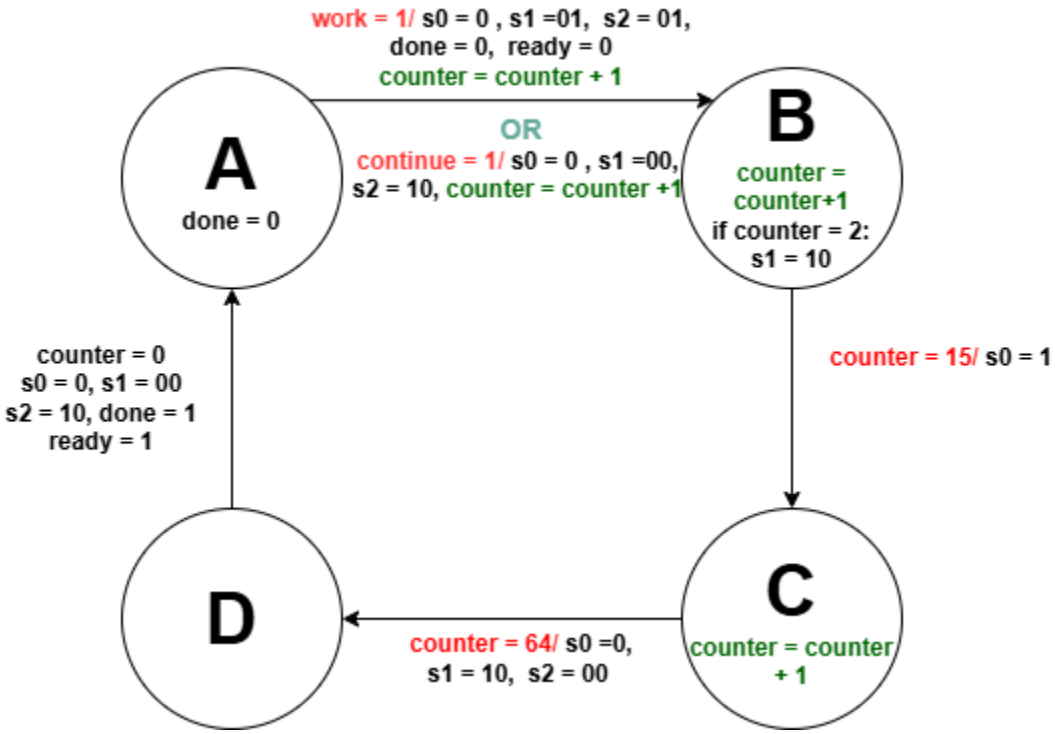
Ο controller του κυκλώματος, η FSM αποτελείται από 4 καταστάσεις και 4 σήματα ελέγχου.

| Κατάσταση | Λειτουργία |
|---|---|
| A | Αναμονή, το κύκλωμα αρχικοποιημένο και περιμένει σήμα start η Continue |
| B | Κύκλωμα σε λειτουργία, σειριακή αποστολή εισόδου W στο SHA compute (0≤t≤15) |
| C | Κύκλωμα σε λειτουργία, σειριακή αποστολή με επεξεργασία W στο SHA compute (16≤t≤63) |
| D | Έτοιμο hash αποθήκευση και επιστροφή στην αναμονή (κατάσταση A) |

| S0 | Στέλνει στην έξοδο του W_function : |
|---|---|
| 0 | την είσοδο του Wt |
| 1 | επεξεργασμένη τιμή Wt |

| S1 | Αρχικοποίηση abcdefgh με : |
|---|---|
| 00 | Hash i-1 |
| 01 | Με το 1ο σετ σταθερών |
| 10 | Τα abcdefgh που παράχθηκαν στον προηγούμενο κύκλο |

| S2 | Αρχικοποίηση $H^{i-1}$ με : |
|---|---|
| 00 | abcdefgh + H i-1 |
| 01 | Με το 1ο σετ σταθερών |
| 10 | Hold $H^{i-1}$ |




*Σχήμα 2. Σχηματικό FSM.*

# 4. Message Scheduler ( W Function )

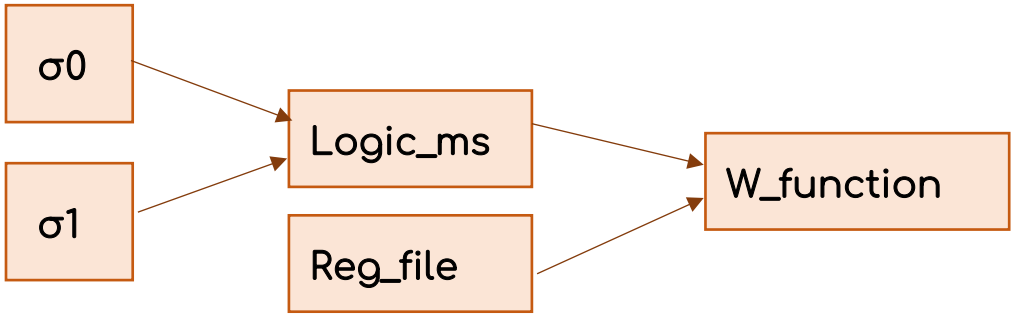



*Figure 2 Bottom - up schematic for message scheduler*

| | |
|---|---|
| Sigma_0_function | $\sigma_0^{(256)}(x) = ROTR^7(x) \oplus ROTR^{18}(x) \oplus SHR^3(x)$ |
| Sigma_1_function | $\sigma_1^{(256)}(x) = ROTR^{17}(x) \oplus ROTR^{19}(x) \oplus SHR^{10}(x)$ |
| Logic_ms | Υπολογισμός της τελευταίας θέσης του πίνακα για τους κύκλους t = 0 εώς 63 |
| Reg_file | Επιλογή ποια τιμή θα γραφτεί στην τελευταία θέση του πίνακα σύμφωνα με το σήμα ελέγχου της FSM και shift όλου του πίνακα προς τα πάνω |
| W_function | Ανάθεση εξόδου message scheduler |

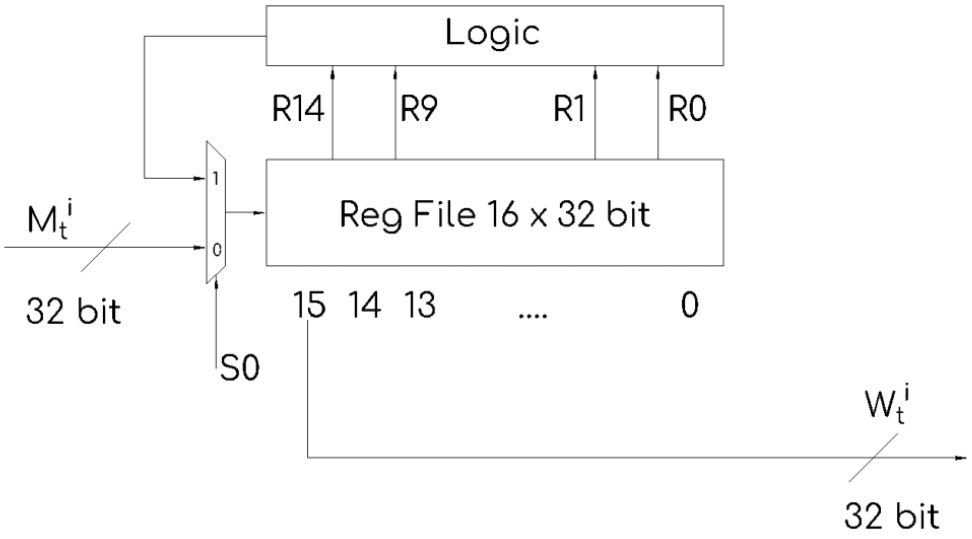



*Figure 1 Logic schematic of message scheduler*

Ο message scheduler παραλαμβάνει ένα message στην είσοδο του σε κάθε κύκλο για τους 16 πρώτους κύκλους του αλγορίθμου. Το message είναι μεγέθους 32-bits και τοποθετείται στην τελευταία θέση του παρακάτω πίνακα:

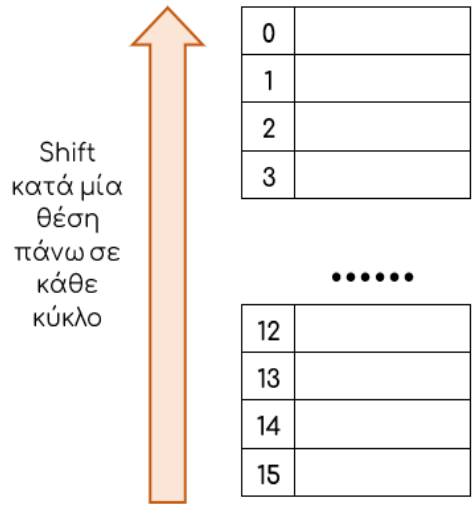



*Figure 3 message scheduler register file*

Σε κάθε κύκλο όλες οι θέσεις γίνονται shift προς τα πάνω όπως φαίνεται στο σχήμα. Αυτή η διαδικασία φαίνεται στην παρακάτω εικόνα μέσω της προσομοίωσης. Όπου παρατηρούμε το σταδιακό γέμισμα του register file.

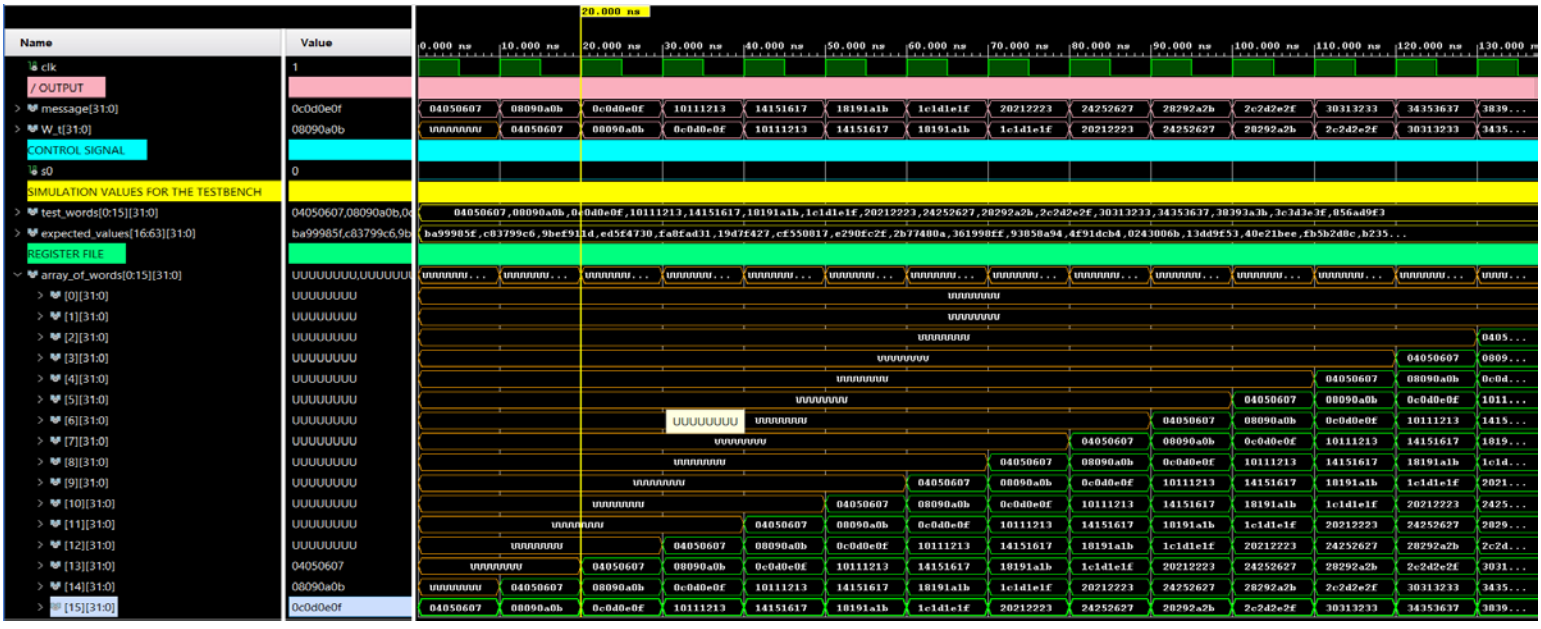



*Figure 4 Προσομοίωση όταν αρχίζει να γεμίζει για πρώτη φορά το register file*

Έτσι μετά από 16 κύκλους ο πίνακας καταχωρητών έχει γεμίσει και αρχίζει να εκτελείται το δεύτερο κομμάτι του αλγορίθμου. Στο οποίο δεν παραλαμβάνουμε πλέον νέα messages, αλλά εκτελούμε την εξής πράξη: $W_t = \sigma_1(W_{t-2}) + W_{t-7} + \sigma_0(W_{t-15}) + W_{t-16}$

η οποία όπως φαίνεται χρειάζεται πάντα τα στοιχεία που βρίσκονται στις θέσεις του πίνακα: 14, 9, 1 και 0. Και σ' αυτό το βήμα θα κάνουμε κατά μία θέση shift τον πίνακα των καταχωρητών.


*Figure 5 Γέμισμα καταχωρητή και αλλαγή λειτουργίας μόλις έρθει σήμα από την FSM*

Όπως φαίνεται και στην παραπάνω προσομοίωση, μόλις το σήμα s0 (σήμα ελέγχου από την FSM) γίνει 1 τότε ξεκινάμε να ανανεώνουμε την τελευταία θέση του πίνακα σύμφωνα με την πράξη όπου εκτελέσαμε και είναι και η έξοδος του κυκλώματος στον επόμενο κύκλο. Αυτή η διαδικασία εκτελείται για t = 16 εώς 63.

Ο κύκλος 64 συμπίπτει με τον κύκλο 0 διότι βγάζουμε το τελευταίο W και ταυτόχρονα ανανεώνουμε την τελευταία θέση του πίνακα με τον νέο message της επόμενης επανάληψης. Όπως φαίνεται στην παρακάτω εικόνα.

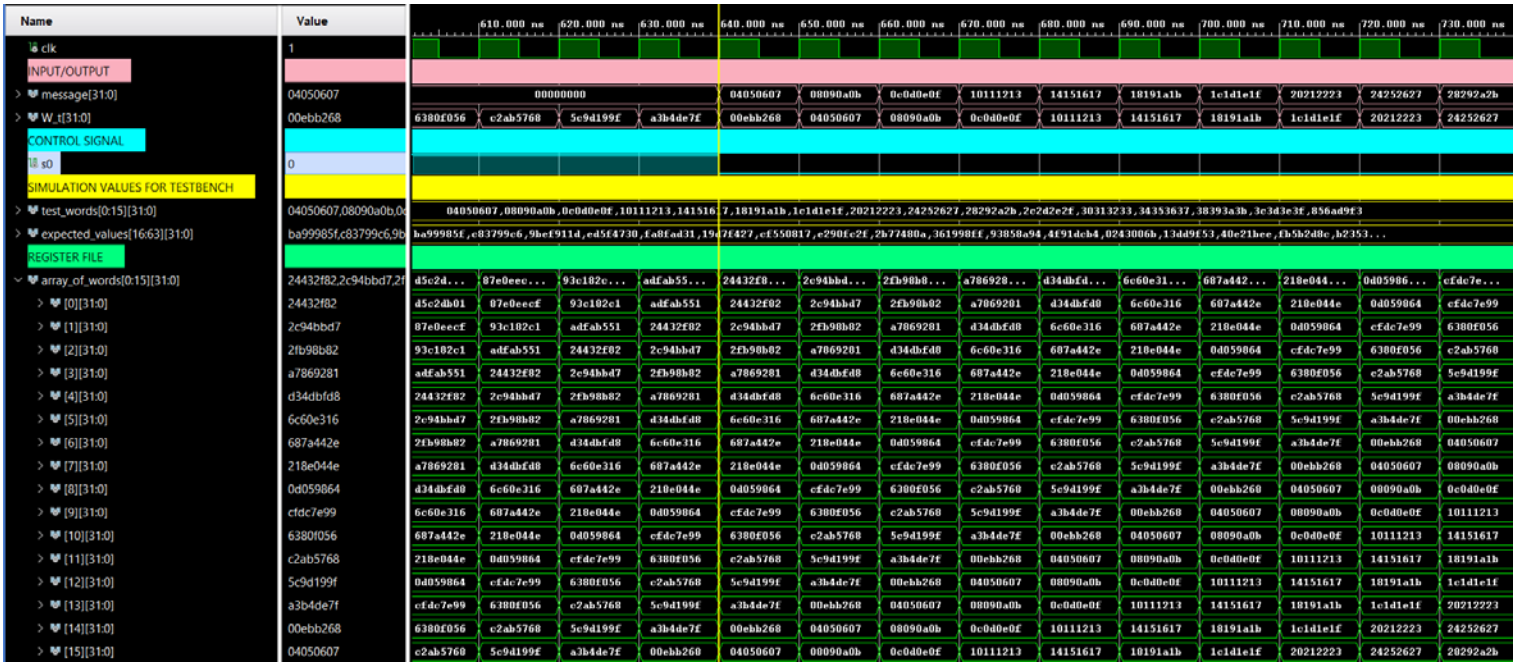



*Figure 6 Ο κέρσορας δείχνει το τέλος της πρώτης λούπας και την αρχή της δεύτερης λούπας*

Οι σημειώσεις τρέξανε μέσω του testbench που δημιουργήθηκε για το συγκεκριμένο κομμάτι. Έχουμε εξάγει τα σωστά αποτελέσματα για το συγκεκριμένο κομμάτι από πρόγραμμα στην python και μέσω του testbench επιβεβαιώνουμε την ορθή λειτουργεία του κυκλώματος. Βλέπουμε στην παρακάτω εικόνα ότι όλα τα αποτελέσματα κάνουνε matching.

```
Note: Match at cycle 16
Time: 180 ns  Iteration: 0  Process: /W_testbench/stimulus_process  File: D:/my_projects_in_vivado/W_presentation/W_presentation.srcs/sim_1/new/W_testbench.vhd
Note: Match at cycle 17
Time: 190 ns  Iteration: 0  Process: /W_testbench/stimulus_process  File: D:/my_projects_in_vivado/W_presentation/W_presentation.srcs/sim_1/new/W_testbench.vhd
Note: Match at cycle 18
Time: 200 ns  Iteration: 0  Process: /W_testbench/stimulus_process  File: D:/my_projects_in_vivado/W_presentation/W_presentation.srcs/sim_1/new/W_testbench.vhd
Note: Match at cycle 19
Time: 210 ns  Iteration: 0  Process: /W_testbench/stimulus_process  File: D:/my_projects_in_vivado/W_presentation/W_presentation.srcs/sim_1/new/W_testbench.vhd
Note: Match at cycle 20
Time: 220 ns  Iteration: 0  Process: /W_testbench/stimulus_process  File: D:/my_projects_in_vivado/W_presentation/W_presentation.srcs/sim_1/new/W_testbench.vhd
Note: Match at cycle 21
Time: 230 ns  Iteration: 0  Process: /W_testbench/stimulus_process  File: D:/my_projects_in_vivado/W_presentation/W_presentation.srcs/sim_1/new/W_testbench.vhd
Note: Match at cycle 22
Time: 240 ns  Iteration: 0  Process: /W_testbench/stimulus_process  File: D:/my_projects_in_vivado/W_presentation/W_presentation.srcs/sim_1/new/W_testbench.vhd
Note: Match at cycle 23
Time: 250 ns  Iteration: 0  Process: /W_testbench/stimulus_process  File: D:/my_projects_in_vivado/W_presentation/W_presentation.srcs/sim_1/new/W_testbench.vhd
Note: Match at cycle 24
Time: 260 ns  Iteration: 0  Process: /W_testbench/stimulus_process  File: D:/my_projects_in_vivado/W_presentation/W_presentation.srcs/sim_1/new/W_testbench.vhd
Note: Match at cycle 25
Time: 270 ns  Iteration: 0  Process: /W_testbench/stimulus_process  File: D:/my_projects_in_vivado/W_presentation/W_presentation.srcs/sim_1/new/W_testbench.vhd
Note: Match at cycle 26
Time: 280 ns  Iteration: 0  Process: /W_testbench/stimulus_process  File: D:/my_projects_in_vivado/W_presentation/W_presentation.srcs/sim_1/new/W_testbench.vhd
Note: Match at cycle 27
Time: 290 ns  Iteration: 0  Process: /W_testbench/stimulus_process  File: D:/my_projects_in_vivado/W_presentation/W_presentation.srcs/sim_1/new/W_testbench.vhd
```

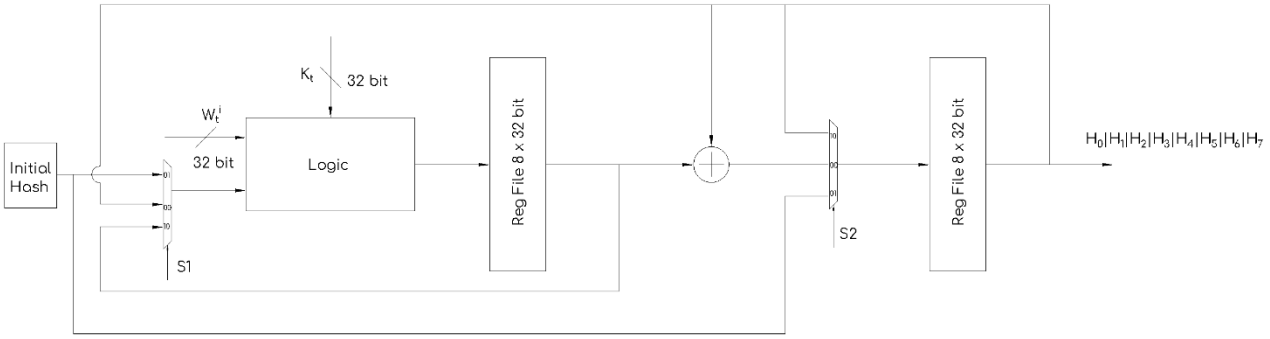
*Figure 7 Διαπίστωση ορθής λειτουργίας του κυκλώματος*

# 5. SHA – 256 COMPUTE

Τα Wt εισέρχονται σειριακά στο κύκλωμα με κάθε clock μαζί με τις σταθερές Kt. Την χρονική στιγμή 0 ως abcdefgh έχουμε το initial και στην συνέχεια έχουμε ανατροφοδότηση από το reg file με τα παλιά abcdefgh.

Αν το μήνυμα έχει πάνω από 1 message τότε για την συνέχεια του υπολογισμού με τα επόμενα Wt, ως αρχική συνθήκη των abcdefgh έχουμε το αποτέλεσμα του hash της προηγούμενης επανάληψης.

Αφού ολοκληρωθούν όλοι οι κύκλοι το hash είναι πλέον έτοιμο στην έξοδο του κυκλώματος.




*Σχήμα 3. Μονάδα υπολογισμού του HASH.*

Με λέξη εισόδου το "abc" παρατηρούμε μεμονωμένα την ορθή λειτουργία της μονάδας όπως φαίνεται παρακάτω.

Η μονάδα αυτή για τον υπολογισμό ενός hash θέλει 65 κύκλους ρολογιού.

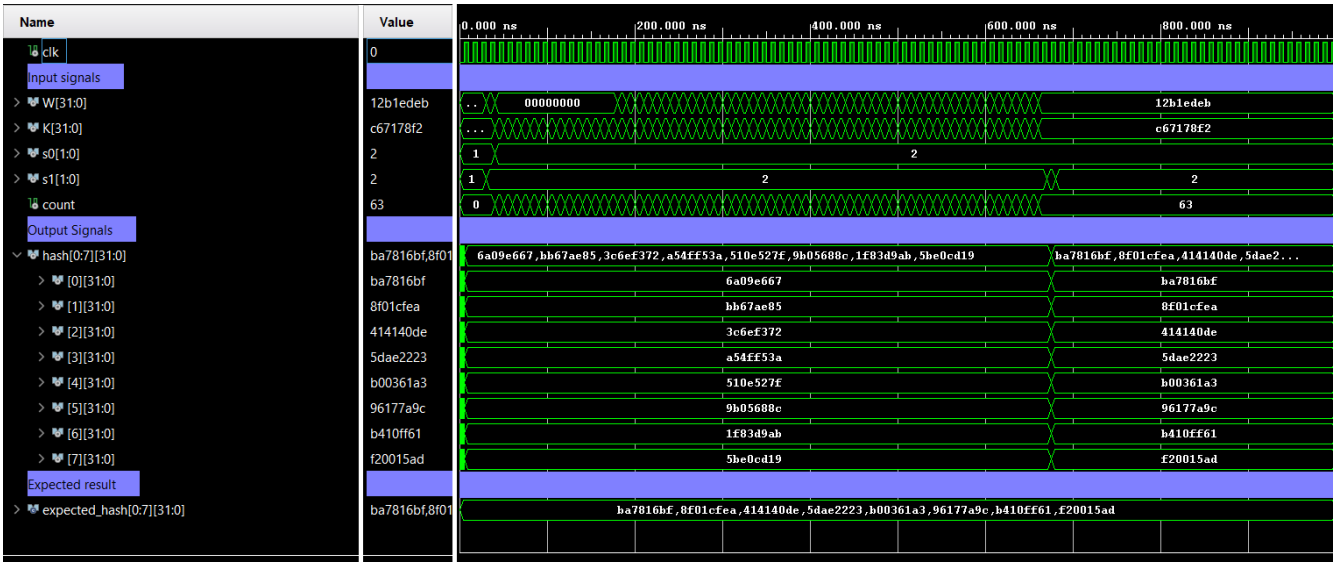
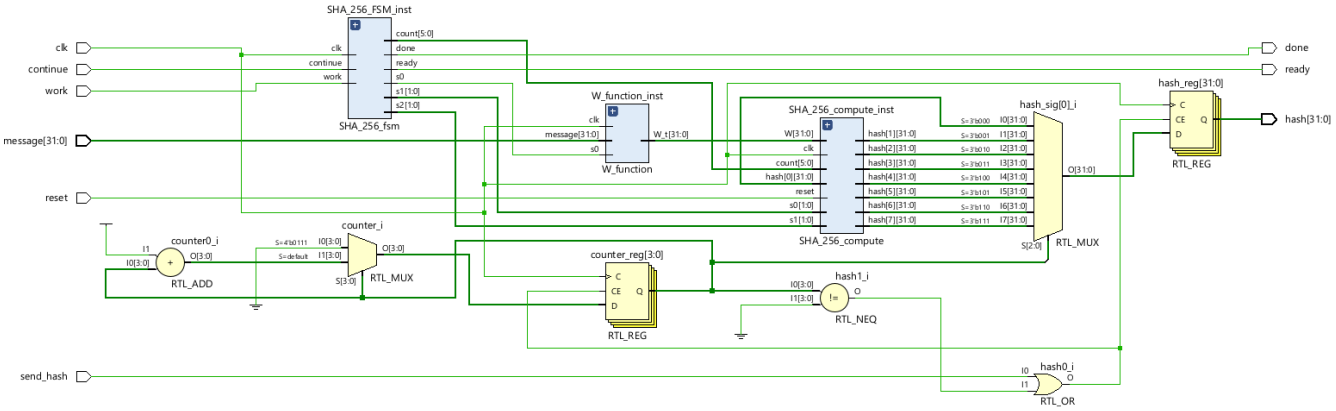



*Figure 8. Προσομοίωση μονάδας Hash Compute για είσοδο "abc".*

# 6. ΣΥΝΟΛΙΚΗ ΥΛΟΠΟΙΗΣΗ

Με χρήση των παραπάνω κυκλωμάτων w function, sha compute και fsm δημιουργήσαμε το συνολικό κύκλωμα υπολογισμού του hash. Λόγω των περιορισμένων io ports του FPGA αποφασίστηκε το αποτέλεσμα του hash να βγαίνει ανά 32 bit στην έξοδο. Με βάση τα παραπάνω και την χρήση του Vivado το κύκλωμα που προκύπτει είναι το παρακάτω.




*Σχήμα 4. Τελικό σχηματικό του κυκλώματος.*

Το κύκλωμα εκτός των components είναι για την σειριακή φόρτωση της εξόδου με το αποτέλεσμα του hash. Αφού το σχηματικό του Vivado ταίριαζε με την δική μας υλοποίηση συνεχίσαμε στις προσομοιώσεις του συνολικού κυκλώματος .

Στην περίπτωση αυτή επιλέχθηκε ένα μήνυμα το οποίο θα δημιουργούσε 2 Messages. Συγκεκριμένα ένα τυχαίο string «theodosissaaefergtrgrtyhtyhyhtht5t45y56y4rrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrry» το οποίο έχει ως αποτέλεσμα το «ba7816bf8f01cfea414140de5dae2223b00361a396177a9cb410ff61f20015ad» σε hex αναπαράσταση.

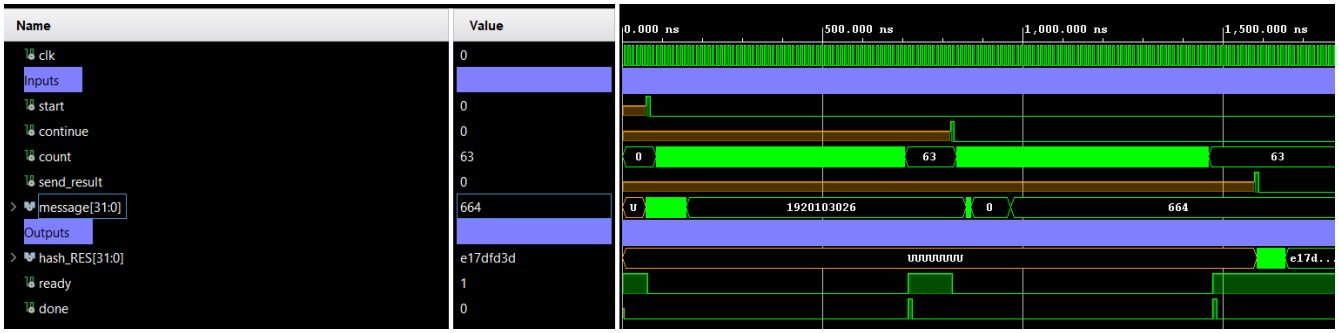



*Figure 9. Αποτέλεσμα προσομοίωσης.*

Το κύκλωμα βγάζει αποτέλεσμα κάθε 66 clocks. Παρατηρούμε ότι όταν τελειώσει ο υπολογισμός σηκώνεται το flag done και ready το οποίο σηματοδοτεί ότι το κύκλωμα τελείωσε τον υπολογισμό  και είναι έτοιμο για τον επόμενο. Όσο το ready είναι 1 ο εξωτερικός επεξεργαστής είτε μπορεί να στείλει start είτε continue το start σηματοδοτεί ότι ο επόμενος υπολογισμός είναι ασυσχέτιστος με τον προηγούμενο και το continue ότι ο ένας είναι συνέχεια του άλλου για το επόμενο message.

Όταν έρθει το done ο επεξεργαστής επίσης μπορεί να στείλει 1 στο send_hash το οποίο σηματοδοτεί την σειριακή εκφόρτωση του αποτελέσματος.

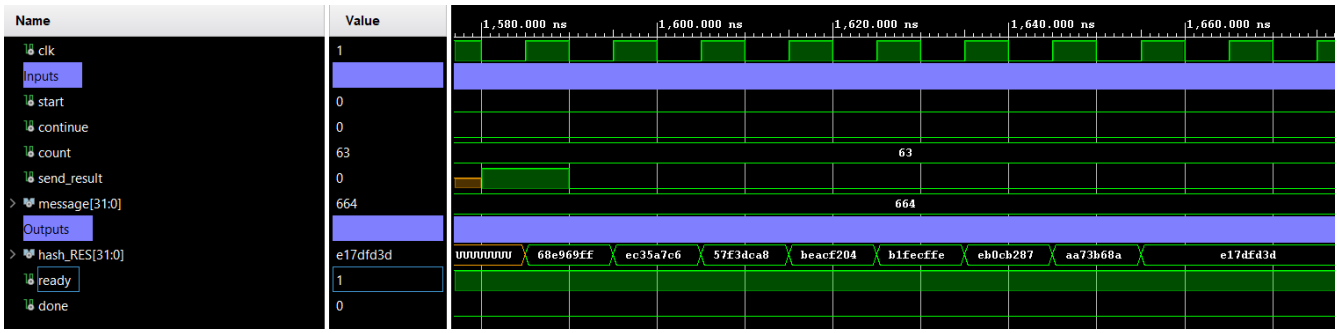



*Figure 10. Προσομοίωση συνολικού κυκλώματος (zoom).*

Σύμφωνα με την υλοποίηση σε python του αλγορίθμου το αποτέλεσμα είναι σωστό.

Εν συνεχεία αφού επιβεβαιώθηκε η ορθή λειτουργία του κυκλώματος συνεχίσαμε στην σύνθεση και στο implementation στο FPGA.

# 7. Synthesis – Implementation

Στο στάδιο αυτό επιλέχθηκαν τα pins του FPGA για τις εισόδους και τις εξόδους. Και έπειτα μέσω πειραματισμού στο Vivado για την ικανοποίηση των setup και hold constrains θέταμε διάφορα ρολόγια έτσι ώστε να μην υπάρχει Violation στον χρονισμό του κυκλώματος. Πολύ βοηθητικές στην διαδικασία αυτή ήταν οι ρυθμίσεις κατά την σύνθεση και το implementation για την ελαχιστοποίηση του χώρου αλλά και του χρόνου του κυκλώματος. Τελικά αφού ικανοποιήθηκαν όλοι οι περιορισμοί το τελικό implementation στο board φαίνεται παρακάτω.

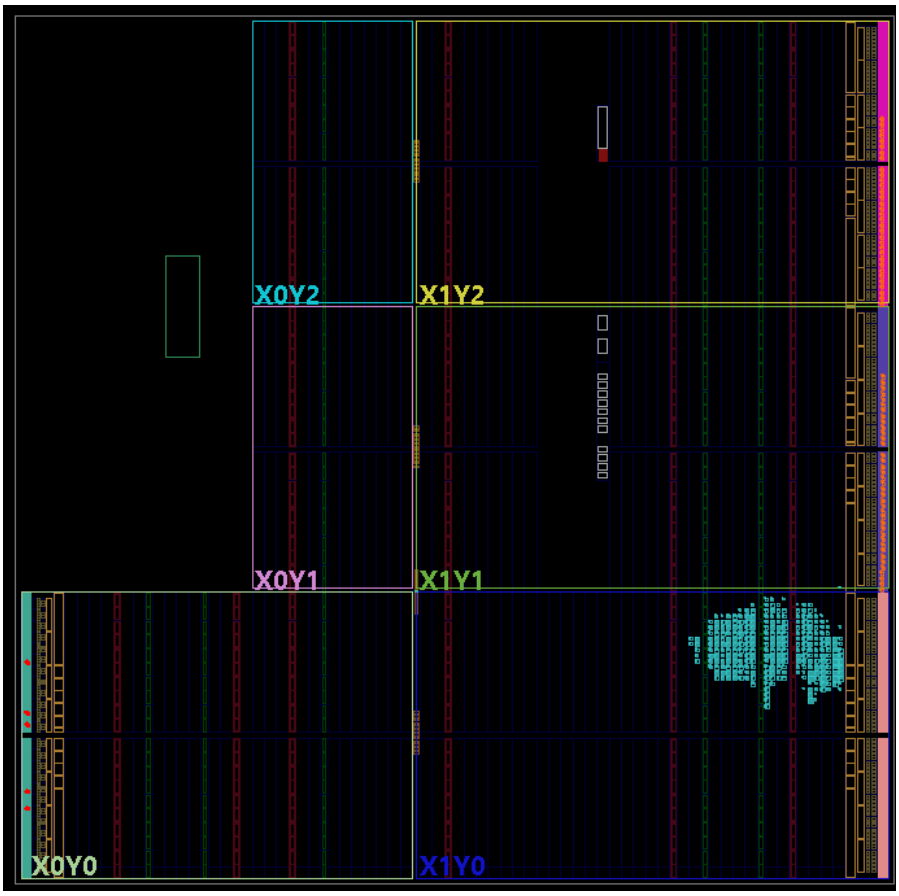



*Figure 11. Υλοποιημένο κύκλωμα στο FPGA.*

Τέλος τρέξαμε και post implementation simulation δηλαδή προσομοίωση η οποία είναι σχεδόν ίδια με το να έτρεχε ο κώδικας στο FPGA. Εκεί το κύκλωμα θέλει λίγο χρόνο για να αρχικοποιηθεί όμως μετά από μερικά iterations τα αποτελέσματα βγαίνουν σωστά.

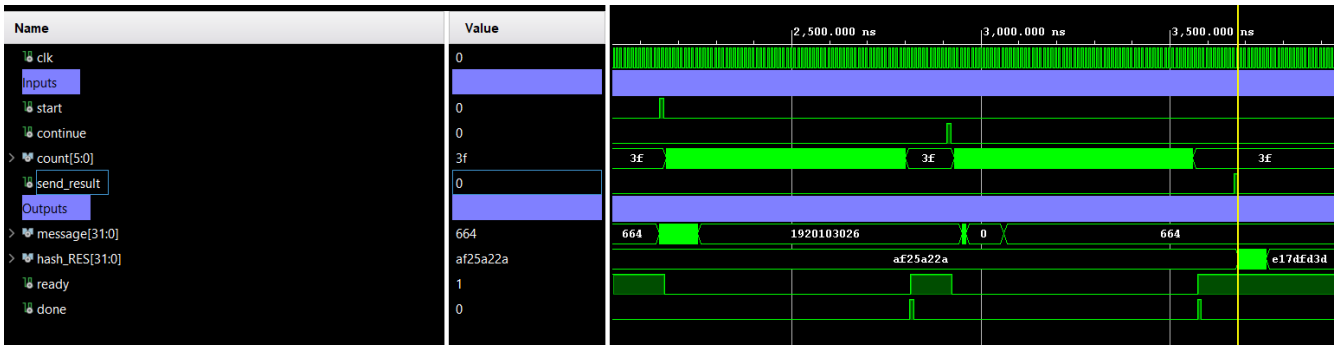



*Figure 13. Post implementation Simulation.*

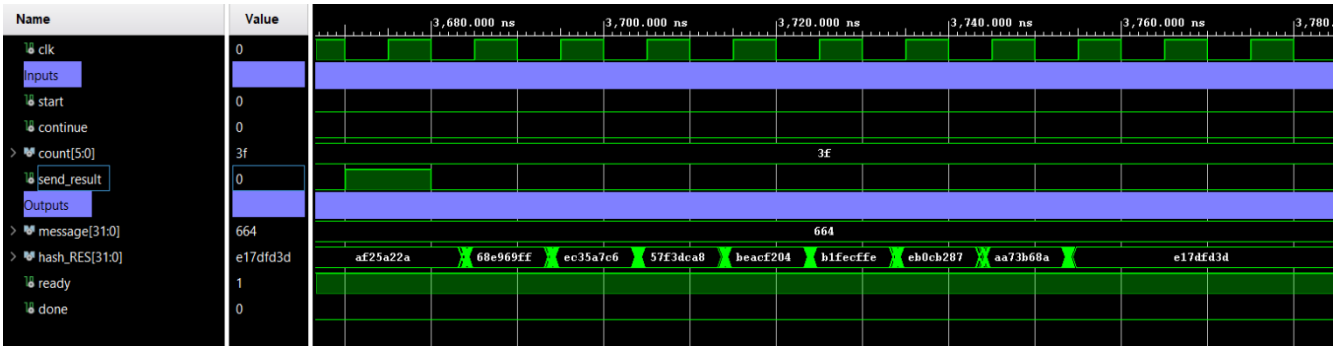



*Figure 12. Post implementation Simulation.(zoom).*

Τα σωστά αποτελέσματα είναι εμφανή αφού έρθει το σήμα send_result τα οποία ταυτίζονται και με την απλή behavioral προσομοίωση.

Μετά και το implementation έχουμε πλήρη εικόνα του κόστος της υλοποίησης σε LUT'S Register's καταναλώσεις ισχύος κ.α. Μέσω των logs του implementation έχουμε τα παρακάτω :

| Site Type | Used | Fixed | Prohibited | Available | Util% |
|---|---|---|---|---|---|
| Slice | 280 | 0 | 0 | 13300 | 2.11 |
| SLICEL | 171 | 0 | | | |
| SLICEM | 109 | 0 | | | |
| LUT as Logic | 967 | 0 | 0 | 53200 | 1.82 |
| using O5 output only | 0 | | | | |
| using O6 output only | 879 | | | | |
| using O5 and O6 | 88 | | | | |
| LUT as Memory | 32 | 0 | 0 | 17400 | 0.18 |
| LUT as Distributed RAM | 0 | 0 | | | |
| LUT as Shift Register | 32 | 0 | | | |
| using O5 output only | 0 | | | | |
| using O6 output only | 0 | | | | |
| using O5 and O6 | 32 | | | | |
| Slice Registers | 736 | 0 | 0 | 106400 | 0.69 |
| Register driven from within the Slice | 544 | | | | |
| Register driven from outside the Slice | 192 | | | | |
| LUT in front of the register is unused | 27 | | | | |
| LUT in front of the register is used | 165 | | | | |
| Unique Control Sets | 13 | | 0 | 13300 | 0.10 |

*Figure 14. Χρήση Slices - FF - LUT.*

Παρατηρούμε ότι η υλοποίηση μας χρησιμοποιεί πολύ λίγους πόρους του FPGA πράγμα λογικό και θεμιτό καθώς θέλουμε πολλές λειτουργίες εντός του FPGA να τρέχουν παράλληλά.

Η κατανάλωση ισχύος του κυκλώματος φαίνεται παρακάτω.

| | |
|---|---|
| Total On-Chip Power (mW) | 164 |
| Dynamic (mW) | 58 |
| Device Static (mW) | 105 |

Δηλαδή το κύκλωμα καταναλώνει ελάχιστη ισχύ.

Υπολογίζουμε Throughput = 787 Mbps

# 8. Πιθανη υλοποιηση για μειωση Critical Path

Η αρχιτεκτονική μας λειτουργεί επιτυχώς με 100MHz ρολόι, πώς θα μπορούσαμε όμως να την αλλάξουμε προκειμένου να αυξήσουμε τη συχνότητα του ρολογιού.

Παρατηρούμε ότι οι μεταβλητές a,b,c,d,e,f,g,h έχουνε χρονική εξάρτηση μεταξύ τους όπως φαίνεται στην παρακάτω εικόνα. Αυτό σημαίνει ότι μπορώ να ξέρω το μελλοντικό h για παράδειγμα, αφού μπορώ να διαπιστώσω ότι σε 3 κύκλους θα έχει την τιμή e. Παρόμοιες παρατηρήσεις μπορώ να κάνω και για τις άλλες μεταβλητές.

Επειδή όμως οι μεταβλητές T1 και T2 εξαρτώνται από τις μεταβλητές a,b,c,h,e,f,g και Wt μπορώ να τα προϋπολογίσω 1 με 3 κύκλους πριν τα χρειαστώ. Το πώς θα πραγματοποιηθεί αυτό θα φανεί από το εάν εφαρμόσουμε στον αλγόριθμό μας unrolling και βρούμε τις πραγματικές χρονικές εξαρτήσεις ανάμεσα σε όλες τις μεταβλητές μας. Αυτή η τεχνική θα αυξήσει το area του κυκλώματος μας, αλλά θα επιτύχει αρκετά καλύτερα αποτελέσματα όσο αναφορά την ταχύτητα του κυκλώματος.

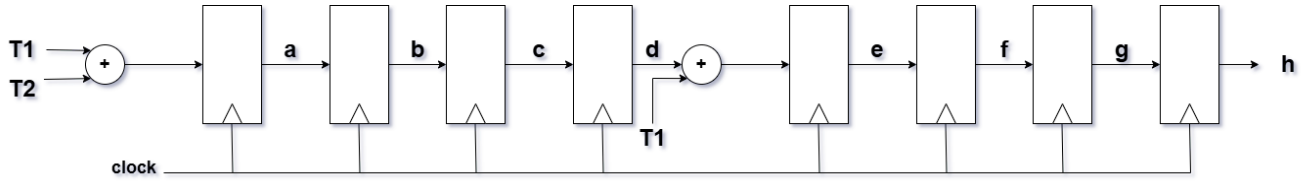



*Figure 15 Χρονική εξάρτηση ανάμεσα στις μεταβλητές*

# 9. Παραρτημα

Παρατίθονται όλοι οι VHDL κώδικες με top – down λογική

```
entity SHA_256 is
    Port (
        clk, reset  : in std_logic;
        message     : in std_logic_vector(31 downto 0);
        work    : in  std_logic;
        send_hash : in std_logic;
        continue : in std_logic;
        hash   : out std_logic_vector(31 downto 0);
        done      : out std_logic;
        ready : out std_logic:='0'
    );
end SHA_256;

architecture rtl of SHA_256 is

    component SHA_256_compute is
        Port (
            clk, reset  : in std_logic;
            W      : in std_logic_vector(31 downto 0);
            s0, s1 : in std_logic_vector(1 downto 0);
            count  : in integer range 0 to 63;
            hash   : inout reg_file_8_32
        );
    end component;

    component W_function is
        generic(number_of_bits_per_word: natural := 32);
        port(
            message: in std_logic_vector(number_of_bits_per_word-1 downto 0);
            clk: in std_logic;
            s0 : in std_logic;
            W_t: out std_logic_vector(number_of_bits_per_word-1 downto 0)
        );
    end component;

    component SHA_256_fsm is
        port (
        clk     : in  std_logic;
         work    : in  std_logic;
         continue : in std_logic;
         done      : out std_logic;
         s0       : out std_logic :='0';
         s1       : out std_logic_vector(1 downto 0) := "01";
         s2       : out std_logic_vector(1 downto 0) := "01";
         count    : out integer range 0 to 63;
         ready    : out std_logic:='1'
     );
     end component;

     signal s0_sig : std_logic :='0';
     signal s1_sig : std_logic_vector(1 downto 0) := "01";
     signal s2_sig : std_logic_vector(1 downto 0) := "01";
     signal W_sig : std_logic_vector(31 downto 0) := (others => '0');
     signal count : integer range 0 to 63;
     signal hash_sig : reg_file_8_32;
     signal counter : integer range 0 to 15:=0;


begin
```

```
67  SHA_256_FSM_inst : SHA_256_fsm
68      port map (
69          clk     => clk,
70          work    => work,
71          continue => continue,
72          done      => done,
73          s0      => s0_sig,
74          s1      => s1_sig,
75          s2      => s2_sig,
76          count   => count,
77          ready   => ready
78      );
79
80      W_function_inst : W_function
81      port map (
82          message => message,
83          clk => clk,
84          s0 => s0_sig,
85          W_t => W_sig
86      );
87
88      SHA_256_compute_inst : SHA_256_compute
89      port map (
90          clk => clk,
91          reset => reset,
92          W => W_sig,
93          s0 => s1_sig,
94          s1 => s2_sig,
95          count => count,
96          hash => hash_sig
97      );
98
99      process(clk) -- process to send the hash to the output
100
101
102     begin
103
104       if clk'event and clk='1' then
105           if (send_hash = '1') or (counter /= 0) then
106
107
108               hash <= hash_sig(counter);
109
110               if counter = 7 then
111                   counter <=0;
112               else
113                   counter<=counter+1;
114               end if;
115
116           end if;
117
118       end if;
119
120       end process;
121
122
123
124 end rtl;
```

*Figure 16 SHA_256*

```vhdl
entity SHA_256_fsm is
    port (
        clk      : in  std_logic;
        work     : in  std_logic;
        continue : in std_logic;
        done      : out std_logic;
        s0       : out std_logic :='0';
        s1       : out std_logic_vector(1 downto 0) := "01";
        s2       : out std_logic_vector(1 downto 0) := "01";
        count    : out integer range 0 to 63;
        ready    : out std_logic:='1'
    );
end SHA_256_fsm;

architecture rtl of SHA_256_fsm is

type state_type is (A, B, C, D);
signal state: state_type:= A;
signal counter: integer range 0 to 65 := 0;


begin

process(clk)
begin

    if clk'event and clk='1' then

     if (counter>=1) and (counter < 65) then
         count <= counter-1;
     end if;


      case state is
          when A =>
               done <= '0';
               s2 <= "10"; -- hold previus hash
               if work = '1' then
                  state <= B;
                  s0 <= '0'; -- keep mesaage in intact
                  s1 <= "01"; -- initialize abc... with initial hash
                  s2 <= "01"; -- initialize Hash i-1 with initial hash
                  done <= '0';
                  ready <= '0';
                  counter <= counter + 1;

               elsif continue = '1' then
                  state <= B;
                  S0 <='0';
                  s1 <= "00"; -- abcde... = Hash i-1
                  s2 <= "10"; -- hold previus hash
                  ready <= '0';
                  counter <= counter + 1;
                end if;
```

```
64              when B =>
65                  if counter = 2 then
66                      s1 <= "10"; -- feed stage 1 with previous abcde.. resul
67                  end if;
68
69                  if counter = 15 then
70                      state <= C;
71                      s0 <= '1'; -- calculate with existing messages W17-W63
72                  end if;
73                  counter <= counter + 1;
74              when C =>
75                  if counter = 64 then
76                      state <= D;
77                      s0 <= '0';
78                      s1 <= "10";
79
80
81                  end if;
82                  counter <= counter + 1;
83              when D =>
84                  state <=A;
85                  counter <=0;
86                  s0 <= '0'; -- keep mesaage in intact
87                  s1 <= "00"; -- load abcde... with H i-1
88                  s2 <= "00"; -- add abcde... + H i-1
89                  done <= '1';
90                  ready <= '1';
91
92              end case;
93
94      end if;
95
96  end process;
97
98
99  end rtl;
00
```

*Figure 17 SHA 256 FSM*

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.Data_pack.all;


entity W_function is
    generic(number_of_bits_per_word: natural := 32);
    port(
    message: in std_logic_vector(number_of_bits_per_word-1 downto 0);
    clk: in std_logic;
    s0 : in std_logic;
    W_t: out std_logic_vector(number_of_bits_per_word-1 downto 0)
    );
end W_function;


architecture message_scheduler_W_function of W_function is

    component Logic_ms is
        port (
            W_T_2 : in std_logic_vector(31 downto 0);
            W_T_7 : in std_logic_vector(31 downto 0);
            W_T_15 : in std_logic_vector(31 downto 0);
            W_T_16 : in std_logic_vector(31 downto 0);
            W_T : out std_logic_vector(31 downto 0)
        );
    end component;
    component Reg_file is
        port (
        clk : in std_logic;
        func : in std_logic;
        message : in std_logic_vector(M-1 downto 0);
        W : in std_logic_vector(M-1 downto 0);
        array_of_words : inout array_of_16_recent_words
    );
    end component;

    signal W_T_2, W_T_7, W_T_15, W_T_16, W: std_logic_vector(number_of_bits_per_word-1 downto 0);
    signal array_of_words: array_of_16_recent_words;
begin


Reg_file_1 : Reg_file port map (clk, s0, message, W, array_of_words);

logic : Logic_ms port map (W_T_2, W_T_7, W_T_15, W_T_16, W);

-- W(t-2) is always in the same position of the arreay because once in every cycle
-- we shift all the words one position
W_T_2 <= array_of_words(N-2);
W_T_7 <= array_of_words(N-7);
W_T_15 <= array_of_words(N-15);
W_T_16 <= array_of_words(N-16);

processs:
process(clk)
begin


if clk'event and clk='1' then
-- the output is always the last position of the array
-- and the output gets it's value one cycle later
    W_t <= array_of_words(N-1);

end if;

end process;
```

*Figure 18 W_Function*

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.Data_pack.all;

entity Reg_file is
    port (
        clk : in std_logic;
        func : in std_logic;
        message : in std_logic_vector(M-1 downto 0);
        W : in std_logic_vector(M-1 downto 0);
        array_of_words : inout array_of_16_recent_words
    );
end Reg_file;

architecture Behavioral of Reg_file is

begin

process(clk)
begin
-- func is the control signal coming from the fsm
-- func chooses the message in the input if we are in the first 16 cycles
-- or it chooses the word output we have computed in Logic_ms vhdl file
if clk'event and clk='1' then

    if func = '1' then
        array_of_words(N-1) <= W;
    else
        array_of_words(N-1) <= message;
    end if;

-- we have created a register file of 16 positions where we keep the 16 most recent
-- W the rest of the W's are no longer useful for the computation of the fututre W's
-- and that is why we don't save them in registers
-- each position in the array shows how recent the W is and in each cycle
-- we shift the W's in the reg file and update the last position according to func
    for i in 0 to array_of_words'length - 2 loop
        array_of_words(i) <= array_of_words(i+1);
    end loop;

end if;

end process;

end Behavioral;
```

*Figure 19 Reg_File*

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.Data_pack.all;

entity Logic_ms is
    port (
        W_T_2 : in std_logic_vector(31 downto 0);
        W_T_7 : in std_logic_vector(31 downto 0);
        W_T_15 : in std_logic_vector(31 downto 0);
        W_T_16 : in std_logic_vector(31 downto 0);
        W_T : out std_logic_vector(31 downto 0)
    );
end Logic_ms;

--architecture Behavioral of Logic_ms is

component sigma_1 is
    generic(number_of_bits_per_word: natural := 32);
    port(word_input: in std_logic_vector(number_of_bits_per_word-1 downto 0);
     word_output: out std_logic_vector(number_of_bits_per_word-1 downto 0));
end component;

component sigma_0 is
    generic(number_of_bits_per_word: natural := 32);
    port(word_input: in std_logic_vector(number_of_bits_per_word-1 downto 0);
     word_output: out std_logic_vector(number_of_bits_per_word-1 downto 0));
end component;


    signal s0 : std_logic_vector(31 downto 0);
    signal s1 : std_logic_vector(31 downto 0);


begin
    Sig1 : sigma_1 port map(W_T_2,s0);
    Sig0 : sigma_0 port map(W_T_15,s1);
    -- outcome for cycles 16 to 63, we get the outcomes from sigma_0 and sigma_1 and
    -- we compute the word output according to the algorithm
    W_T <= std_logic_vector((unsigned(s0) + unsigned(s1))+(unsigned(W_T_16)+unsigned(W_T_7)));
end Behavioral;
```

*Figure 20 Logic_ms*

```vhdl
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;

entity sigma_1 is
    generic(number_of_bits_per_word: natural := 32);
    port(word_input: in std_logic_vector(number_of_bits_per_word-1 downto 0);
     word_output: out std_logic_vector(number_of_bits_per_word-1 downto 0));
end sigma_1;


architecture behavior_of_sigma_1 of sigma_1 is

signal term_1, term_2, term_3: std_logic_vector(number_of_bits_per_word-1 downto 0);

begin
-- rotate right 17 positions
term_1 <= word_input(16 downto 0)& word_input(number_of_bits_per_word-1 downto 17);
--rotate right 19 positions
term_2 <= word_input(18 downto 0)& word_input(number_of_bits_per_word-1 downto 19);
-- shift right 10 positions
term_3 <= "0000000000" & word_input(number_of_bits_per_word-1 downto 10);

word_output <= term_1 xor term_2 xor term_3;


end behavior_of_sigma_1;
```

*Figure 22 sigma_1*

```vhdl
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;


entity sigma_0 is
    generic(number_of_bits_per_word: natural := 32);
    port(word_input: in std_logic_vector(number_of_bits_per_word-1 downto 0);
     word_output: out std_logic_vector(number_of_bits_per_word-1 downto 0));
end sigma_0;


architecture behavior_of_sigma_0 of sigma_0 is

signal term_1, term_2, term_3: std_logic_vector(number_of_bits_per_word-1 downto 0);

begin

-- rotate right 7 positions
term_1 <= word_input(6 downto 0)& word_input(number_of_bits_per_word-1 downto 7);
-- rotate right 18 positions
term_2 <= word_input(17 downto 0)& word_input(number_of_bits_per_word-1 downto 18);
-- shift right 10 positions
term_3 <= "000" & word_input(number_of_bits_per_word-1 downto 3);
-- xor all the above terms
word_output <= term_1 xor term_2 xor term_3;

end behavior_of_sigma_0;
```

*Figure 21 sigma_0*

```vhdl
entity SHA_256_compute is
    Port (
        clk, reset  : in std_logic;
        W      : in std_logic_vector(31 downto 0);
        s0, s1 : in std_logic_vector(1 downto 0);
        count  : in integer range 0 to 63;
        hash   : inout reg_file_8_32
    );
end SHA_256_compute;

architecture rtl of SHA_256_compute is

    component Stage1 is
        Port (
            input  : in  reg_file_8_32;
            K      : in std_logic_vector(31 downto 0);
            W      : in std_logic_vector(31 downto 0);
            output : out reg_file_8_32
        );
    end component;


    component Constant_reg is
     port (
        t : in integer range 0 to 63;
        K : out std_logic_vector(31 downto 0)
    );
    component Initial_hash_reg is
       port (
           initial_hash : out reg_file_8_32
       );
    end component;

    component mux is
    port (
        initial_hash : in reg_file_8_32;
        stage_1_output : in reg_file_8_32;
        Hash_i_1 : in reg_file_8_32;
        s0 : in std_logic_vector(1 downto 0);
        output : out reg_file_8_32
    );
    end component;

    component Final_stage is
    port (
        clk : in std_logic;
        abcdefg : in reg_file_8_32;
        initial_hash : in reg_file_8_32;
        s2 : in std_logic_vector(1 downto 0);
        hash : out reg_file_8_32
    );
end component;
```

```
end component;

    signal K : std_logic_vector(31 downto 0);
    signal inital_hash : reg_file_8_32;
    signal stage_1_output, stage_1_reg, output_mux, final_abcdefg: reg_file_8_32;


begin

    IH1: Initial_hash_reg port map(inital_hash); --Contains H0 Constant
    C1 : Constant_reg port map(count, K);            --K constants select based on count (signal from fsm)
    St1 : Stage1 port map(output_mux, K, W, stage_1_output);  --Logic of SHA 256 compute
    M1 : mux port map(inital_hash,stage_1_reg, hash, s0, output_mux); --Selects the output based on s0 (from FSM) S0=1 initial hash, S0=0 stage 1 out

    Final : Final_stage port map(clk, final_abcdefg, inital_hash, s1, hash); -- Does the a+H and stores the new hash value

    process(clk, reset)

    begin

        if reset = '1' then



        elsif rising_edge(clk) then
            stage_1_reg <= stage_1_output; --abcdefgh reg for each round
            final_abcdefg <= stage_1_output; -- abcdefg reg for final stage



        end if;

    end process;



end rtl;
```

*Figure 24 SHA-256 COMPUTE*

```
entity Initial_hash_reg is
    port (
        initial_hash : out reg_file_8_32
    );
end Initial_hash_reg;

architecture rtl of Initial_hash_reg is
begin
    initial_hash(0) <= X"6a09e667";  -- Initial hash values for SHA-256
    initial_hash(1) <= X"bb67ae85";
    initial_hash(2) <= X"3c6ef372";
    initial_hash(3) <= X"a54ff53a";
    initial_hash(4) <= X"510e527f";
    initial_hash(5) <= X"9b05688c";
    initial_hash(6) <= X"1f83d9ab";
    initial_hash(7) <= X"5be0cd19";
end rtl;
```

*Figure 23 Initial_hash_reg*

```
entity Constant_reg is
    port (
        t : in integer range 0 to 63;
        K : out std_logic_vector(31 downto 0)
    );
end Constant_reg;

architecture rtl of Constant_reg is
    signal K_array : K_array := (
        X"428a2f98", X"71374491", X"b5c0fbcf", X"e9b5dba5",
        X"3956c25b", X"59f111f1", X"923f82a4", X"ab1c5ed5",
        X"d807aa98", X"12835b01", X"243185be", X"550c7dc3",
        X"72be5d74", X"80deb1fe", X"9bdc06a7", X"c19bf174",
        X"e49b69c1", X"efbe4786", X"0fc19dc6", X"240ca1cc",
        X"2de92c6f", X"4a7484aa", X"5cb0a9dc", X"76f988da",
        X"983e5152", X"a831c66d", X"b00327c8", X"bf597fc7",
        X"c6e00bf3", X"d5a79147", X"06ca6351", X"14292967",
        X"27b70a85", X"2e1b2138", X"4d2c6dfc", X"53380d13",
        X"650a7354", X"766a0abb", X"81c2c92e", X"92722c85",
        X"a2bfe8a1", X"a81a664b", X"c24b8b70", X"c76c51a3",
        X"d192e819", X"d6990624", X"f40e3585", X"106aa070",
        X"19a4c116", X"1e376c08", X"2748774c", X"34b0bcb5",
        X"391c0cb3", X"4ed8aa4a", X"5b9cca4f", X"682e6ff3",
        X"748f82ee", X"78a5636f", X"84c87814", X"8cc70208",
        X"90befffa", X"a4506ceb", X"bef9a3f7", X"c67178f2"
    );

    begin
        K <= K_array(t);
end rtl;
```

*Figure 25 Constant_reg*

```vhdl
entity Stage1 is
    Port (
        input  : in  reg_file_8_32;
        K      : in std_logic_vector(31 downto 0);
        W      : in std_logic_vector(31 downto 0);
        output : out reg_file_8_32
    );
end Stage1;


architecture rtl of Stage1 is

    component Sigma0 is
        Port (
            input  : in  std_logic_vector(31 downto 0);
            output : out std_logic_vector(31 downto 0)
        );
    end component;

    component Sigma1
        Port (
            input  : in  std_logic_vector(31 downto 0);
            output : out std_logic_vector(31 downto 0)
        );
    end component;

    component Ch
        Port (
            x  : in  std_logic_vector(31 downto 0);
            y  : in  std_logic_vector(31 downto 0);
            z  : in  std_logic_vector(31 downto 0);
            output : out std_logic_vector(31 downto 0)
        );
    end component;

    component Maj
        Port (
            x  : in  std_logic_vector(31 downto 0);
            y  : in  std_logic_vector(31 downto 0);
            z  : in  std_logic_vector(31 downto 0);
            output : out std_logic_vector(31 downto 0)
        );
    end component;

    component Adder_5
        Port (
            a  : in  std_logic_vector(31 downto 0);
            b  : in  std_logic_vector(31 downto 0);
            c  : in  std_logic_vector(31 downto 0);
            d  : in  std_logic_vector(31 downto 0);
            e  : in  std_logic_vector(31 downto 0);
            output : out std_logic_vector(31 downto 0)
        );
    end component;
signal T1, T2 : std_logic_vector(31 downto 0);
    signal temp1, temp2, temp3, temp4 : std_logic_vector(31 downto 0);

begin

    Sigma0_0 : Sigma0 Port map (input => input(0), output => temp1);
    Maj_0    : Maj Port map (x => input(0), y => input(1), z => input(2), output => temp2);
```

```
    T2 <= std_logic_vector(unsigned(temp1) + unsigned(temp2));
    Sigma1_0 : Sigma1 Port map (input => input(4), output => temp3);
    Ch_0     : Ch Port map (x => input(4), y => input(5), z => input(6), output => temp4);
    Adder_5_0 : Adder_5 Port map (a => temp3, b => temp4, c => input(7), d => K, e => W, output => T1); -- tree adder of 5 inputs (32 bit)


    output(0) <= std_logic_vector(unsigned(T1) + unsigned(T2));
    output(1) <= input(0);
    output(2) <= input(1);
    output(3) <= input(2);
    output(4) <= std_logic_vector(unsigned(input(3)) + unsigned(T1));
    output(5) <= input(4);
    output(6) <= input(5);
    output(7) <= input(6);


end rtl;
```

*Figure 28 Stage_1*

```
entity Sigma0 is
    Port (
        input  : in  std_logic_vector(31 downto 0);
        output : out std_logic_vector(31 downto 0)
    );
end Sigma0;

architecture rtl of Sigma0 is
begin
    output <= std_logic_vector(rotate_right(unsigned(input), 2) xor
                               rotate_right(unsigned(input), 13) xor
                               rotate_right(unsigned(input), 22));

end rtl;
```

*Figure 27 Sigma_0*

```
entity Sigma1 is
    Port (
        input  : in  std_logic_vector(31 downto 0);
        output : out std_logic_vector(31 downto 0)
    );
end Sigma1;

architecture rtl of Sigma1 is
begin
    output <= std_logic_vector(rotate_right(unsigned(input), 6) xor
                               rotate_right(unsigned(input), 11) xor
                               rotate_right(unsigned(input), 25));

end rtl;
```

*Figure 26 Sigma_1*

```
5  entity Maj is
6      Port (
7          x  : in  std_logic_vector(31 downto 0);
8          y  : in  std_logic_vector(31 downto 0);
9          z  : in  std_logic_vector(31 downto 0);
10         output : out std_logic_vector(31 downto 0)
11     );
12 end Maj;
13
14 architecture rtl of Maj is
15 begin
16     output <= (x and y) xor (x and z) xor (y and z);
17 end rtl;
```

*Figure 30 Maj*

```
5  entity Ch is
6      Port (
7          x  : in  std_logic_vector(31 downto 0);
8          y  : in  std_logic_vector(31 downto 0);
9          z  : in  std_logic_vector(31 downto 0);
10         output : out std_logic_vector(31 downto 0)
11     );
12 end Ch;
13
14 architecture rtl of Ch is
15 begin
16     output <= (x and y) xor ((not x) and z);
17 end rtl;
```

*Figure 29 Ch*

```
entity Adder_5 is
    Port (
        a  : in  std_logic_vector(31 downto 0);
        b  : in  std_logic_vector(31 downto 0);
        c  : in  std_logic_vector(31 downto 0);
        d  : in  std_logic_vector(31 downto 0);
        e  : in  std_logic_vector(31 downto 0);
        output : out std_logic_vector(31 downto 0)
    );
end Adder_5;

architecture rtl of Adder_5 is
    signal s1,s2,s3 : std_logic_vector(31 downto 0);


begin
    s1 <= std_logic_vector(unsigned(a) + unsigned(b));
    s2 <= std_logic_vector(unsigned(s1) + unsigned(c));
    s3 <= std_logic_vector(unsigned(d) + unsigned(e));
    output <= std_logic_vector(unsigned(s2) + unsigned(s3));
end rtl;
```

*Figure 32 Adder_5*

```
entity mux is
    port (
        initial_hash : in reg_file_8_32;
        stage_1_output : in reg_file_8_32;
        Hash_i_1 : in reg_file_8_32;
        s0 : in std_logic_vector(1 downto 0);
        output : out reg_file_8_32
    );
end mux;

architecture rtl of mux is
begin
    with s0 select
        output <= initial_hash when "01",
                  stage_1_output when "10",
                  Hash_i_1 when others;
end rtl;
```

*Figure 31 mux*

```
entity Final_stage is
    port (
        clk : in std_logic;
        abcdefg : in reg_file_8_32;
        initial_hash : in reg_file_8_32;
        s2 : in std_logic_vector(1 downto 0);
        hash : out reg_file_8_32
    );
end Final_stage;


architecture rtl of Final_stage is

    signal H_i_1,add : reg_file_8_32;
    signal H_i : reg_file_8_32;

begin

gen:
for i in 0 to 7 generate
    add(i) <= std_logic_vector( unsigned(H_i(i)) + unsigned(abcdefg(i)) ); -- add the a,b,c,d,e,f,g,h with previous hash to get the new hash
end generate;

with s2 select
H_i_1 <= add when "00", -- add the a,b,c,d,e,f,g,h with previous hash to get the new hash
      initial_hash when "01", --H_i_1 = initial_hash
      H_i when others; --H_i_1 = H_i
  hash <= H_i;

  proccess:
  process(clk)
  begin

  if clk'event and clk='1' then


      H_i <= H_i_1;    -- register to store the new hash

  end if;

  end process;


  end rtl;
```

*Figure 33 Final_stage*