Princess Sumaya University for Technology

King Abdullah II School for Electrical Engineering

Computer Engineering Department

Computer architecture (2)

(22321)

Hash Functions

Sha256 pipeline

Name: Ahmad Alzoul-20190492

Name: Motasem refati-20170101

Abstract

The pipelined implementation of SHA-256 is a way to make the widely used cryptographic hash function run faster and do more work. Using a pipeline design, the performance lets multiple input blocks be processed simultaneously, cutting down on processing time and making the most of the hardware. This abstract talks about the main features, benefits, and problems of the pipelined method, focusing on the need for balanced pipeline stages and the management of dependencies. Performance tests show that throughput has improved significantly, which makes pipelined SHA-256 a promising way to speed up cryptographic processes in applications like blockchain and secure communications.

TABLE OF CONTENTS

1 INTRODUCTION

SHA-256 (Secure Hash Algorithm 256-bit) is a cryptographic hash function that produces a 256-bit hash value from an input message. It uses multiple parts and a specific design methodology:

1. Message Input: The data or message to be hashed.

2.Padding: Adding extra bits to the message being sent to meet the algorithm's block size standards.

3. State variables: The hash values that were used to start the hashing process and the hash values that were used in between.

Constants: Values that have already been set and are used in the algorithm's steps.

5. Round Operations: Logical and bitwise operations are done on the message and state variables to change the hash values.

6. Message Schedule: A list of words that come from the input message and are used in the round processes.

Hash Output: The final value of the hash, which is usually shown as a 256-bit digest.

In the design method, the message is broken up into blocks, variables are set up, the message is made bigger, rounds of operations are done, and the hash value is set.

2 PROPOSED DESIGN

This is our description of each module and its functionality:

1-'**sha256_block**': The SHA-256 block processor is represented by this top-level module. It manages control signals and coordinates the data flow among many submodules. It refreshes the internal hash values while maintaining the status of the current round.

2-'**sha256_round**': This module represents a single round of the SHA-256 algorithm. It takes input values (Kj, Wj, and the current values of a to h) and computes the output values for the next round (a_e to h_e) based on various SHA-256 operations like Ch, Maj, and circular shifts.

3-'sha256_s0': This module computes the Σ_0 (Sigma-0) operation of SHA-256. It takes a 32-bit input x and performs bitwise operations to generate a 32-bit output S0.

4-**'sha256_s1':** same as s0 to give us the output S1.

5-**'sha256_k_machine**': This module generates the round constants **Kj** used in the SHA-256 algorithm. It contains a ROM (Read-Only Memory) that stores the initial values of the round. The module provides the current round **K** based on the clock signal.

6-'**sha256_H_i**': This module provides the initial hash values that used in sha algorithm.

7-'**sha2_round**': this module does the round computation for sha256 algorithm, include the privies modules.

8-helpful functions: '**Ch'** and **'Maj**' just module that I used to simple wirte the code and not need to write full equation.
9- '**W_machine': This** module represents the message scheduling; it maintains a stack (**W_stack_c**) to store the input message (**M**) and provides the current **W** value based on the clock signal.

3 FIGURES AND BLOCK DIAGRAMS

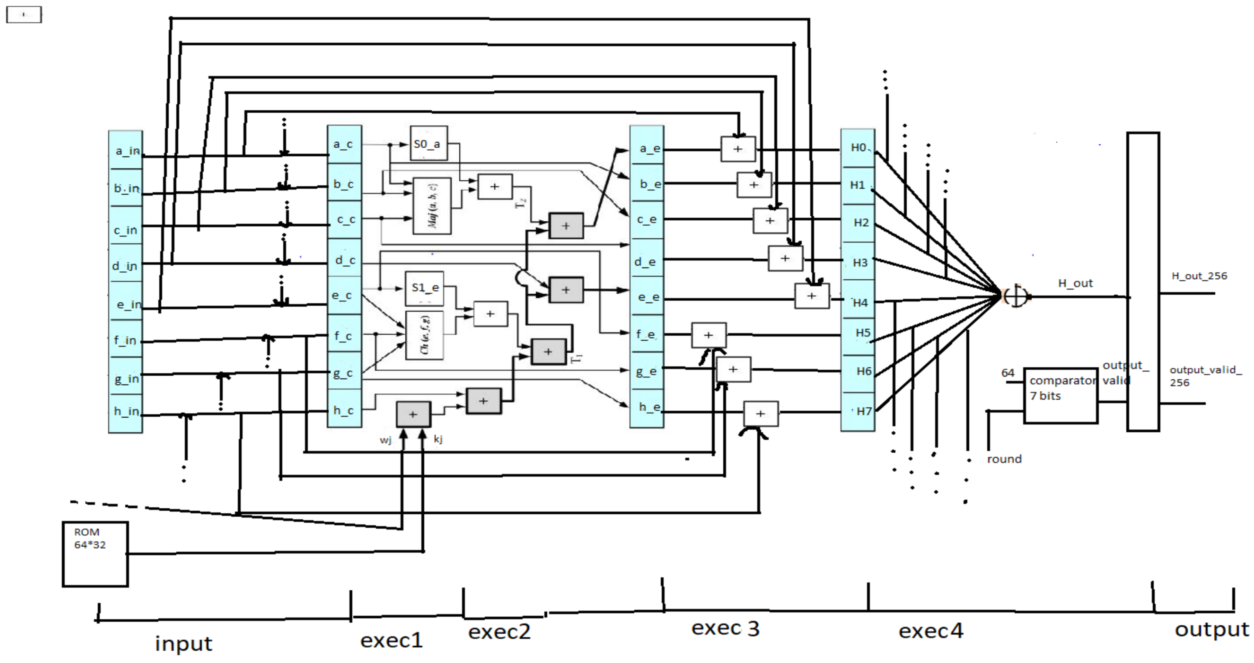


figure [1] sha256 design with stages

We've got a 6-stage pipeline, as shown in the figure [1], so let's talk about every stage:

1-**inpu:** here we have the initial value of a_in to h_in and also the ROM for kj.

2: **exec1:** here is the start of the compression function, and here are the initial values kj and wj for the next stage. Also do the equations for Ch and S1 and S0 and maj, and here are their equations:

Ch(x,y,z) = (x and y) XOR (not E and G)

Maj(x,y,z) = (x AND y) XOR (x and c) XOR (y and z)

S0(a)= (a>>>2) XOR (a>>> 13) XOR (

S1(a) = (a>>> 6) XOR (a>>> 11) XOR (a>>> 25)

3-exec2: get the values for T1 and T2 and update the value of a_e to h_e.

T1 = h + S1 + ch + k [i] + w [i].

T2 = S0 + maj.

4: **exec3: here we update the hash value for H0 to H7 by adding a_in to a_e and h_in to h_e.**

5-exec4: Here we combine the H0 to H7 to get our hash output for each round and also start the comparator so we know if we get 64 rounds complete.

6-output: here we show our hash output H_out_256 and comparator result 1, which means that we finish the 64 rounds; 0 means it's not finished.

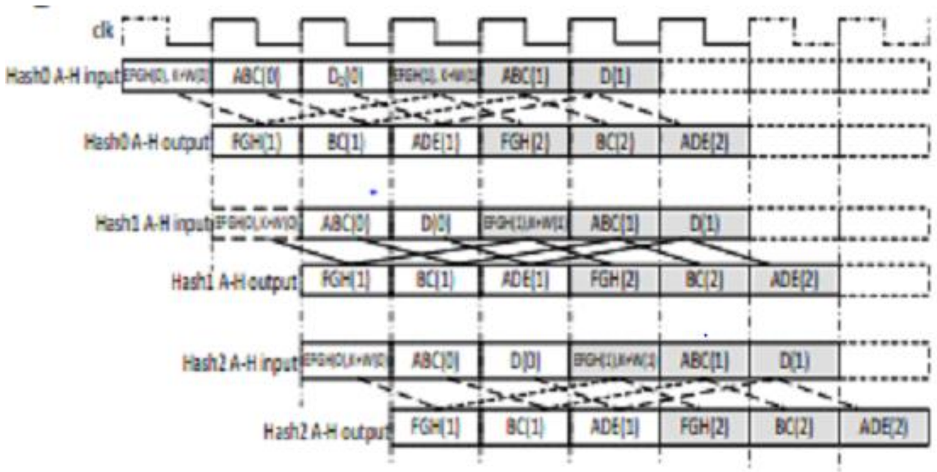


FIGURE (2) parallel pipeline SHA_256

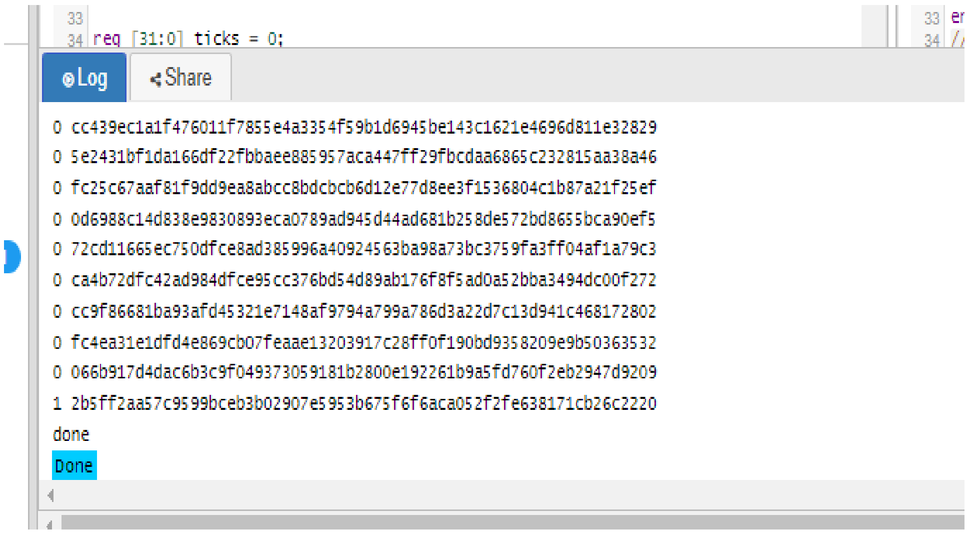
So, we can see in Figure 2 the parallel pipeline that can be calculated together at the same time.

And show the input and output stages overlapped between the stag

4: Design Analysis

In our proposal, we were only going to do the round computation part, but now we do a fully pipelined sha256 except for the padding part, which was not simple to do. Some of our problems were that we didn't make 'a_i to h_i," "a_c to h_c," and "a_e to h_e"; we made them all in the same wire, "a to h," and that made a lot of errors, like some time we needed the initial value or current value, so make it all in the same wire, and mistake also, we used to do for loop, and that made a lot of errors in the design, like non-deterministic iteration, sequential execution, and this made us understand the design and the algorithm of sha256 more.

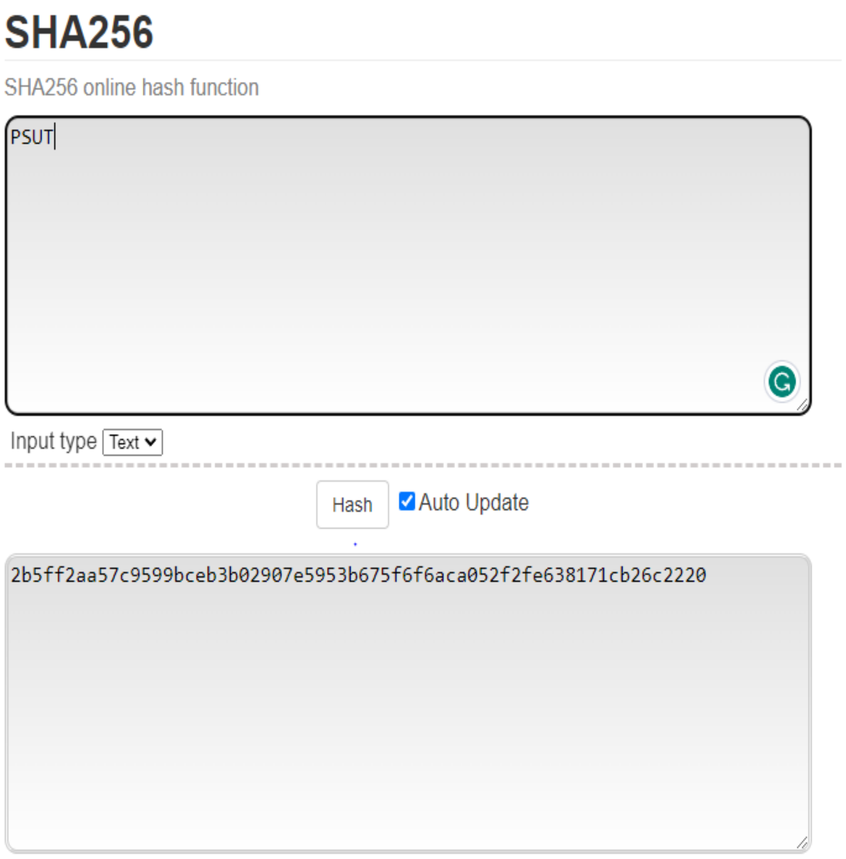
4 Result and Discussion



```
reg [31:0] ticks = 0;
```

```
0 cc439ec1a1f476011f7855e4a3354f59b1d6945be143c1621e4696d811e32829
0 5e2431bf1da166df22fbbaee885957aca447ff29fbcdaa6865c232815aa38a46
0 fc25c67aaf81f9dd9ea8abcc8bdcbcb6d12e77d8ee3f1536804c1b87a21f25ef
0 0d6988c14d838e9830893eca0789ad945d44ad681b258de572bd8655bca90ef5
0 72cd11665ec750dfce8ad385996a40924563ba98a73bc3759fa3ff04af1a79c3
0 ca4b72dfc42ad984dfce95cc376bd54d89ab176f8f5ad0a52bba3494dc00f272
0 cc9f86681ba93afd45321e7148af9794a799a786d3a22d7c13d941c468172802
0 fc4ea31e1dfd4e869cb07feaae13203917c28ff0f190bd9358209e9b50363532
0 066b917d4dac6b3c9f04937305918 1b2800e192261b9a5fd760f2eb2947d9209
1 2b5ff2aa57c9599bceb3b02907e5953b675f6f6aca052f2fe638171cb26c2220
done
Done
```

figure [3] Hash output for PSUT massage


SHA256

SHA256 online hash function

PSUT

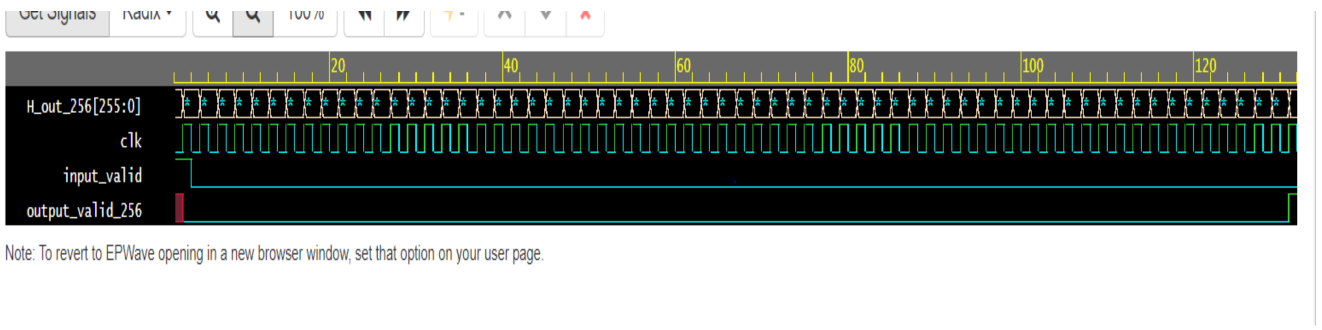
Input type Text

Hash ✓ Auto Update

```
2b5ff2aa57c9599bceb3b02907e5953b675f6f6aca052f2fe638171cb26c2220
```

figure [4] The output from a website [4]

As we can see from the website, we get same result after 64 rounds so our hash design is correct.



Note: To revert to EPWave opening in a new browser window, set that option on your user page.

figure [5] Wpwave showing cycle by cycle

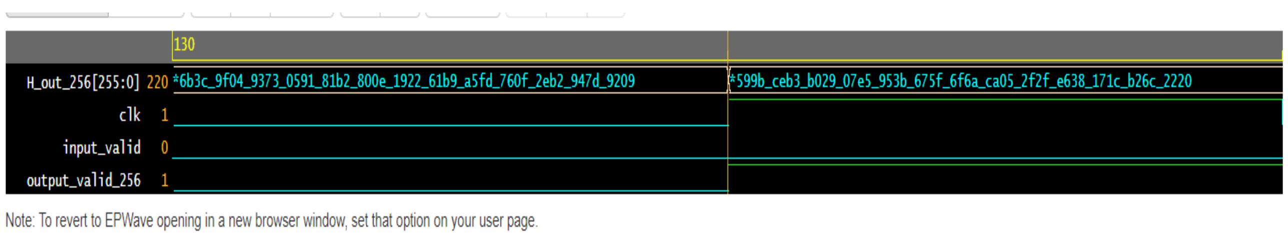

```
                         130
H_out_256[255:0] 220 *6b3c_9f04_9373_0591_81b2_800e_1922_61b9_a5fd_760f_2eb2_947d_9209   *599b_ceb3_b029_07e5_953b_675f_6f6a_ca05_2f2f_e638_171c_b26c_2220
             clk   1
     input_valid   0
output_valid_256   1
```
Note: To revert to EPWave opening in a new browser window, set that option on your user page.

figure [6] Wpwave for the hash output

As we can see in the figure [3], that our hash design gives hash output for every cycle by cycle.

5 Conclusions

Our project fully pipelined design for SHA-256 , although excluding the padding part, was successful in generating the correct hash output. The design process helped us gain a deeper understanding of the SHA-256 algorithm and the challenges involved in implementing it in hardware now we can do some improve to our design by breaking the critical path [1].

References

1- https://ieeexplore.ieee.org/document/8693229

2- https://ieeexplore.ieee.org/document/9349095

3- https://www.scitepress.org/papers/2010/29914/29914.pdf

4- https://emn178.github.io/online-tools/sha256.html