SHA-256

Name: Zhiping Wang
Unityid: zwang79
StudentID: 200265085

Delay (ns to run provided provided example). Clock period: 7.5ns # cycles”: 87 for message that length = 1. Number of cycles = msg_length + 86	Logic Area: 17796.464 (um^2) Memory: N/A	1/(delay.area) ($ns^{-1}.um^{-2}$) 8.61×10^{-8}
Delay (TA provided example. TA to complete)		1/(delay.area) (TA)

Abstract

SHA-2 (**Secure Hash Algorithm 2**) is a set of cryptographic hash functions designed by the United States National Security Agency (NSA). They are built using the Merkle–Damgård structure, from a one-way compression function itself built using the Davies–Meyer structure from a (classified) specialized block cipher. **SHA-256** limits the word length to 32 bits. This project is a simplified version in which the massage length is limited within 55 instead of 2^{64}. The algorithm is fairly straight forward and has a linear structure. This report introduces the hardware implementation based on Verilog 2001 and proposes several methods to optimize the algorithm to increase working frequency and reduce chip area.

SHA-256

Name: Zhiping Wang Unityid: zwang79 StudentID: 200265085

Abstract

SHA-2 (**Secure Hash Algorithm 2**) is a set of cryptographic hash functions designed by the United States National Security Agency (NSA). They are built using the Merkle–Damgård structure, from a one-way compression function itself built using the Davies–Meyer structure from a (classified) specialized block cipher. **SHA-256** limits the word length to 32 bits. This project is a simplified version in which the massage length is limited within 55 instead of 2^{64}. The algorithm is fairly straight forward and has a linear structure. This report introduces the hardware implementation based on Verilog 2001 and proposes several methods to optimize the algorithm to increase working frequency and reduce chip area.

1. Introduction

Basic algorithm

The objective of this project is to implement the full SHA-256 operation and write the final hash array, H[0] through H[7] to the output memory. The message will be contained as ascii in an SRAM that is outside of my verilog module. The length of the message is specified using a 6-bit number representing the number of ascii characters in the message. The message is read and a 512-bit block/vector, M is constructed using the message and the bit length of the message. The vector M is then separated into an array of sixteen 32-bit words, M_1(0) ... M_1(15).

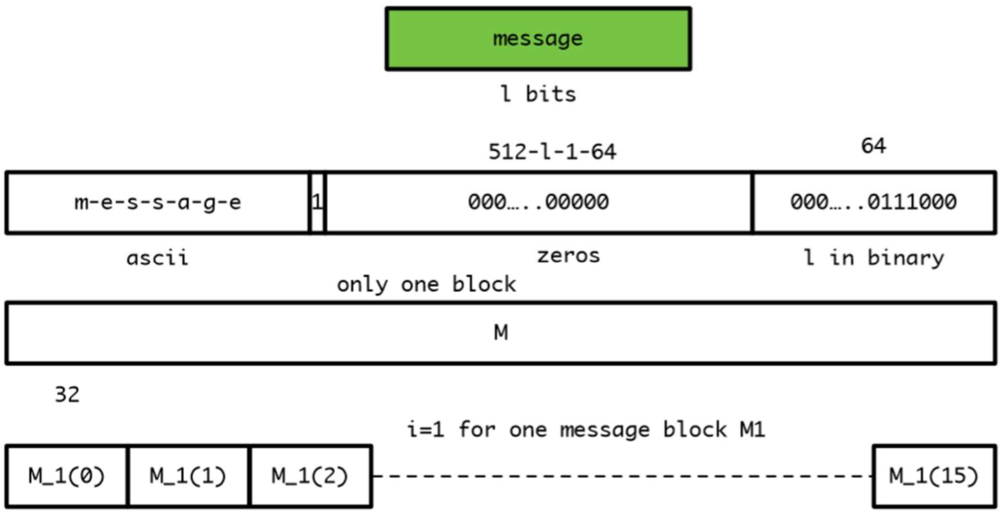


Fig.1: M vector

The array M_1 is copied into the first 16 elements of a 64 32-bit word array, W. The elements 16 through 63 of W are processed using a combination of XOR and shift/rotate. Each element W[i] is a function of lower order elements e.g. W[i] = fn(W[i-2], W[i-7], W[i-15], W[i-16]) shown below

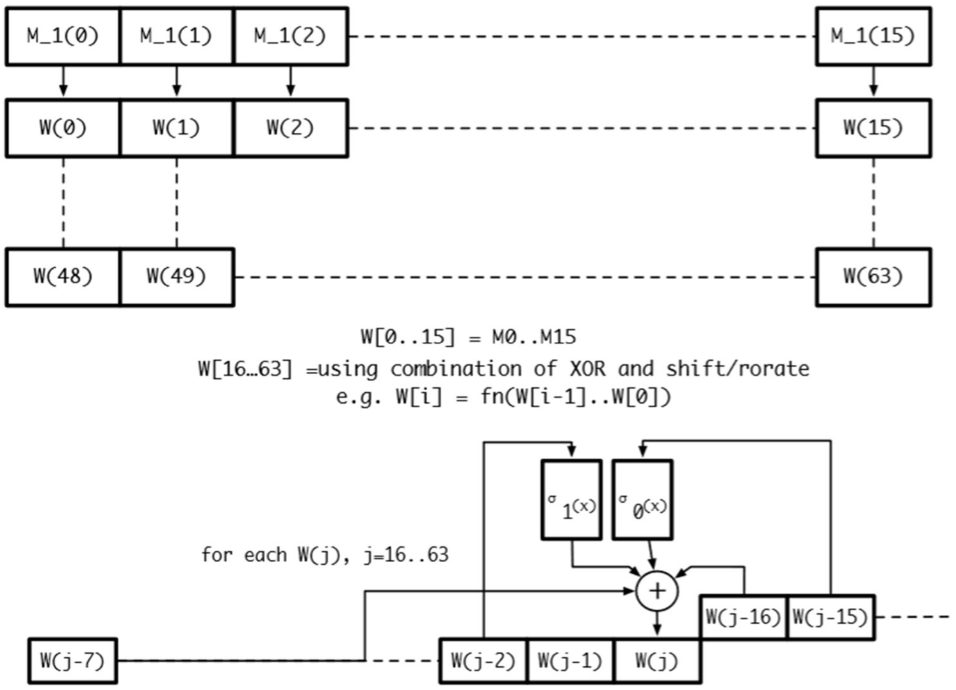


Fig.2: Generate W array

At this point we have constructed the 64 element W array. We then construct another 64-element array, K by reading 64 values from the K SRAM.

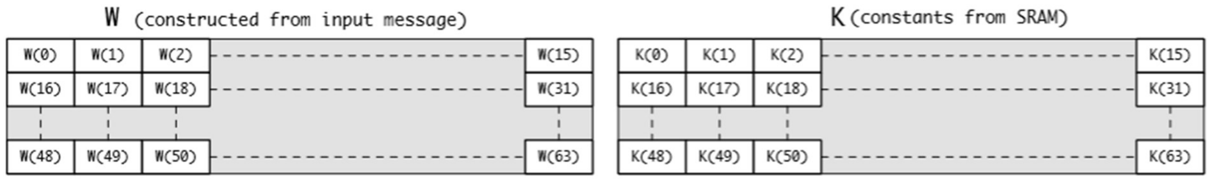


Fig.3: W and K array

Once we have W and K, we load an eight 32-bit element array H from the contents of the H SRAM. We then initialize eight 32-bit registers, a,b,c,d,e,f,g,h from the contents of the H vector. We then iterate 64 times, j=0...63 on the a-h registers using element j from W and K and calculating a-h as shown below.

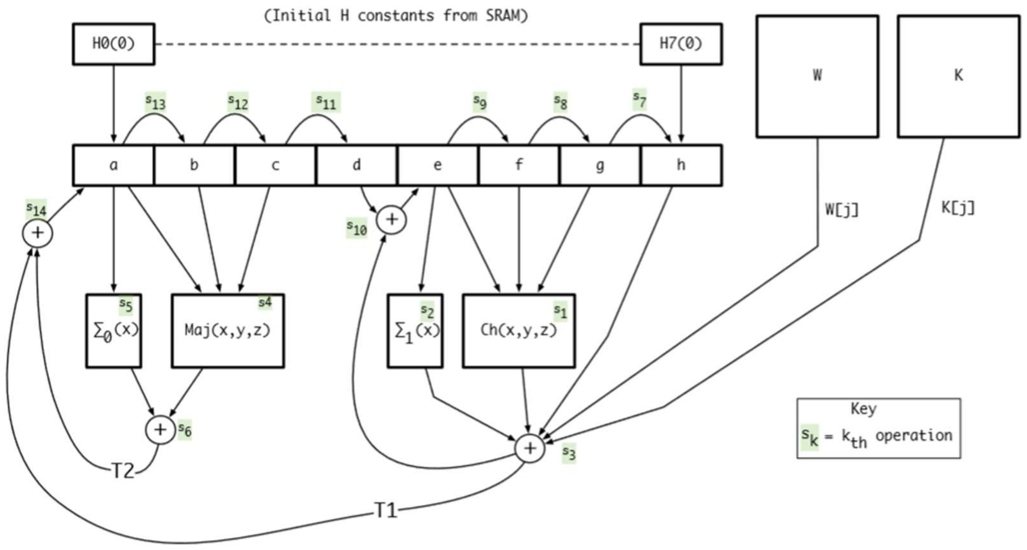
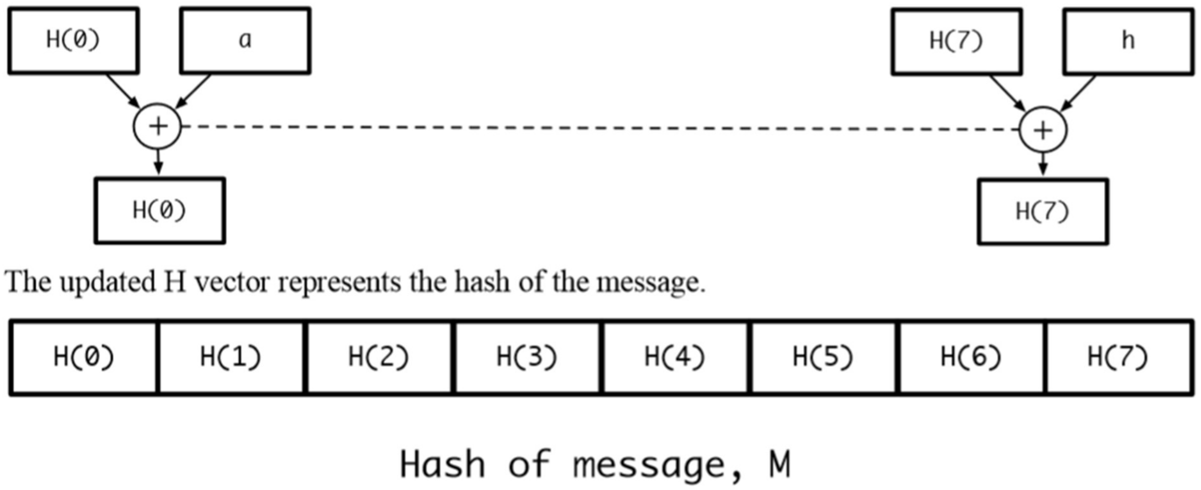


Fig.4: Iterations

Once we have completed all 64 steps on a-h, the H vector is updated as shown below.



The updated H vector represents the hash of the message.

Fig.5: Output

The eight 32-bit values from H are then written to an output SRAM.

Innovations

- We do not need to store the values of K and H array as long as we use them right after they were read from the SRAM. Plus, we do not need a ~ h registers.
- The scale of W array can be reduced to 16 blocks. This can save us a lot of area.
- We can use pipeline technique to produce the values of W array.
- The values of H array can be read and pre-treated while M and W module is working.

Results achieved

The function of SHA-256 algorithm is fully achieved. The clock period is 7.5 ns, and the number of clock cycles that is taken in every calculation is msg_length + 86, e.g. if the message length is 1, the number of clock cycles will be 87. The area is 17796.464 um^2.

Content abstract

- Micro-Architecture
- Interface Description
- Hierarchy Design
- Verification
- Results Achieved
- Summary

2. Micro-Architecture

As I mentioned before, the overall structure is fairly straightforward, so the method I use is basically according to the algorithm.

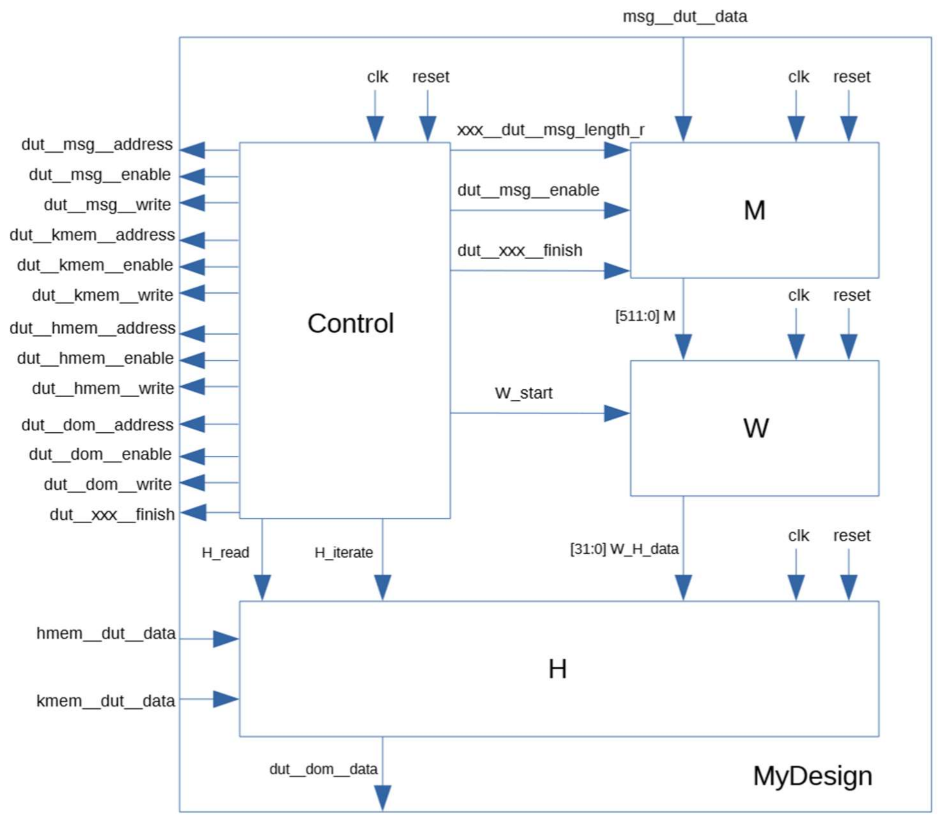


Fig.6: Micro-Architecture

Innovation details

1. We do not need to store the values of K and H array as long as we use them right after they were read from the SRAM. Plus, we do not need a ~ h registers.

   For K:

   In my design, the read operation of K array takes 1 clock cycle per value. It also takes 1 clock cycle for every iteration in H module. This gives me an idea that I don't really need to store the values of K in an array, instead, I only need to read K right before I use it.

   For H:

   Same thing goes with H, I don't need to store its values. The value of H vector is twice in the iteration operation (first for the initial value of a ~ h, then for summation after 64 times of iterations), which means all I need to do is to read from H SRAM twice. In addition, we can conduct the iteration steps with H[0] ~ H[7], that is to say we don't need a ~ h registers.

2. The scale of W array can be reduced to 16 blocks. This can largely cut down the total area.

   As is depicted in the algorithm, to generate the j th block of W, I need (j-16) th, (j-15) th, (j-7) th, (j-2) th previous values. This provides us two optimization potentials.

   First one is that I only need to store 16 previous blocks of W array to generate a new one. That means the scale of W array can be reduced to 16 blocks. And thanks to the nature of clock edge assignment, the value of new W block can be store in the first one of the 16 blocks, e.g. if we already have W[0] … W[15], the next value can be store in W[0] once again and does not affect the following process.

   The second opportunity is that (j-1) th value is not required to generate the j th value, so I can design a pipeline here. I will discuss this in the following section.

3. We can use pipeline technique to produce the values of W array.

   To implement pipeline in W module, I choose to insert intermediate registers sig0 and sig1.

$$\mathrm{W}[\mathrm{j}] = \mathrm{W}[\mathrm{j}-7] + \mathrm{W}[\mathrm{j}-16] + \mathrm{sig0} + \mathrm{sig1}$$

   In this case, we can generate W[j] as well as sig0 and sig1 (for W[j+1]) in one clock cycle. This technique can reduce the clock period to some degree.

4. The values of H array can be read and pre-treated while M and W module is working.

   Unlike the K values, both H values and W values that are applied in the iteration process need to be pre-treated. This provides us an opportunity to arrange the read operation of H flexibly. For example, we can read M and H at the same time to save some clock cycles. If one is finished earlier, it can "wait" for the other. The "wait" operation will be carried out by H_iterate signal.

3. Interface Specification

Chart.1: Interface I/O

Name	Source Module	Object Module	Length	Function
Extenal Signals				
clk	Global	Global	1	Clock
reset	Global	Global	1	Reset
xxx__dut__go	Input	Control	1	Flag signal indicate the start of operation.
xxx__dut__msg_length	Input	Control, M	$clog2(MAX_MESSAGE_LENGTH)+1	The length of massage. Affects total number of clock cycles. It is also an important factor in generating M vector.
dut__msg__address	Control	M SRAM	$clog2(MAX_MESSAGE_LENGTH)	The address of massage when massage is read from M SRAM. Generated by Control Module.
dut__msg__enable	Control	M SRAM	1	High when data is read from SRAM. Generated by Control Module.
dut__msg__write	Control	M SRAM	1	Low when data is read from SRAM. Generated by Control Module.
dut__kmem__address	Control	K SRAM	$clog2(NUMBER_OF_Ks)	The address of K when K is read from K SRAM. Generated by Control Module.
dut__kmem__enable	Control	K SRAM	1	High when data is read from SRAM. Generated by Control Module.
dut__kmem__write	Control	K SRAM	1	Low when data is read from SRAM. Generated by Control Module.
dut__hmem__address	Control	H SRAM	$clog2(NUMBER_OF_Hs)	The address of H when H is read from H SRAM. Generated by Control Module.
dut__hmem__enable	Control	H SRAM	1	High when data is read from SRAM. Generated by Control Module.
dut__hmem__write	Control	H SRAM	1	Low when data is read from SRAM. Generated by Control Module.
dut__dom__address	Control	Output SRAM	$clog2(OUTPUT_LENGTH)	The address of output data when output data is written into ouput SRAM. Generated by Control Module.

dut__dom__enable	Control	Output SRAM	1	High when data is written into SRAM. Generated by Control Module.
dut__dom__write	Control	Output SRAM	1	High when data is written into SRAM. Generated by Control Module.
dut__xxx__finish	Control	Output	1	High when all operations are done, ready for another “go” signal.
Internal Signals				
W_start	Control	W	1	Indicate that M vector is ready, and the values of W can be generated.
H_read	Control	H	1	Indicate that H is being read. This will be high twice for the whole operation.
H_iterate	Control	H	1	Indicate that the iteration process begins.
[511:0] M	M	W	512	M vector
[31:0] W_H_data	W	H	32	W vector used in iteration

4. Hierarchy Design

MyDesign.v

My design demonstrates a hierarchy structure. MyDesign module is the top (without memory) module. It consists of instantiations only.

M.v

The function of M module is to gather data read from M SRAM and build the 512-bit M vector. Then transmit the vector to W module.

W.v

After receiving the M vector, W module divides it into 16 blocks and distributes them into the W[0] ~ W[15], then produce new values of W blocks with them. Meanwhile, W module also transmit the values stored in the array to H vector for iteration.

H.v

The function of H module is a little bit complicated. First, it reads the initial values of H twice from SRAM. Secondly, H module reads the values of K from SRAM right before the K values participate in the iteration calculation, so that there is no need to store them. Thirdly, H module conducts the 64-loop iteration. Lastly, it adds the iteration result to the initial values read for the second time and send the summation to top module.

Control.v

The Control module generate all flag signals, including addresses, enables, writes, finish signals. As well as inter-module flag signals, for example, W_start signal that tells W module M vector is ready for division, etc. Control module basically arranges all the timing in the entire operation.

5. Verification

My verification can be separated into two parts: functional verification based on Modelsim, synthesis verification based on synopsys.

Functional verification: the basic idea is to exam the waveform generated by Modelsim according to a certain input and check if the output is correct. If not, then check the waveform module by module to find out if there is a logic mistake in the design. Moreover, the test vector (message) needs to be diverse, for example, the message length of my test vectors varies from 1 to 55. The multiple inputs ensure that the function is valid in all kinds of situations.

Synthesis verification: this is rather straight forward. The basic idea is to synthesis my design with dc_shell and check if there are any warnings or errors that need to be eliminated. Some common errors I had are listed below.

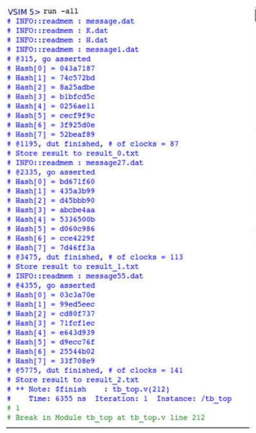
Chart.2 Errors

description	reason	solution
Interesting syntax error. Synopsys can detect this kind of syntax error while Modelsim cannot.	Code style like M[511-8*counter:504-8*counter] <= data_in; Is not synthesizable, variable counter cannot appear on both bit sides.	M [511-8*counter-:8] <= data_in;
Latches.	Did not cover all circumstances in conditional statement.	Cover all circumstances or use default.
Certain net is driven by more than one source, and not all drivers are three-state.	Register is changed in different "always block"	A certain register can only be changed in one procedural block.
Setup/hold time violations	Clock period was too tense; critical path needs to be optimized.	Set the clock period looser; optimize critical path.

6. Results Achieved

Results

The function of SHA-256 algorithm is fully achieved. The clock period is 7.5 ns, and the number of clock cycles that is taken in every calculation is msg_length + 86, e.g. if the message length is 1, the number of clock cycles will be 87. The area is 17796.464 um^2.



```
VSIM 5> run -all
# INFO::readmem : message.dat
# INFO::readmem : K.dat
# INFO::readmem : H.dat
# INFO::readmem : message1.dat
# @315, go asserted
# Hash[0] = 043a7187
# Hash[1] = 74c572bd
# Hash[2] = 8a25adbe
# Hash[3] = b1bfcd5c
# Hash[4] = 0256ae11
# Hash[5] = cecf9f9c
# Hash[6] = 3f925d0e
# Hash[7] = 52beaf89
# @1195, dut finished, # of clocks = 87
# Store result to result_0.txt
# INFO::readmem : message27.dat
# @2335, go asserted
# Hash[0] = bd671f60
# Hash[1] = 435a3b99
# Hash[2] = d45bbb90
# Hash[3] = abcbe4aa
# Hash[4] = 5336500b
# Hash[5] = d060c986
# Hash[6] = cce4229f
# Hash[7] = 7d46ff3a
# @3475, dut finished, # of clocks = 113
# Store result to result_1.txt
# INFO::readmem : message55.dat
# @4355, go asserted
# Hash[0] = 03c3a70e
# Hash[1] = 99ed5eec
# Hash[2] = cd80f737
# Hash[3] = 71fcf1ec
# Hash[4] = e643d939
# Hash[5] = d9ecc76f
# Hash[6] = 25544b02
# Hash[7] = 33f708e9
# @5775, dut finished, # of clocks = 141
# Store result to result_2.txt
# ** Note: $finish    : tb_top.v(212)
#    Time: 6355 ns  Iteration: 1  Instance: /tb_top
# 1
# Break in Module tb_top at tb_top.v line 212
```

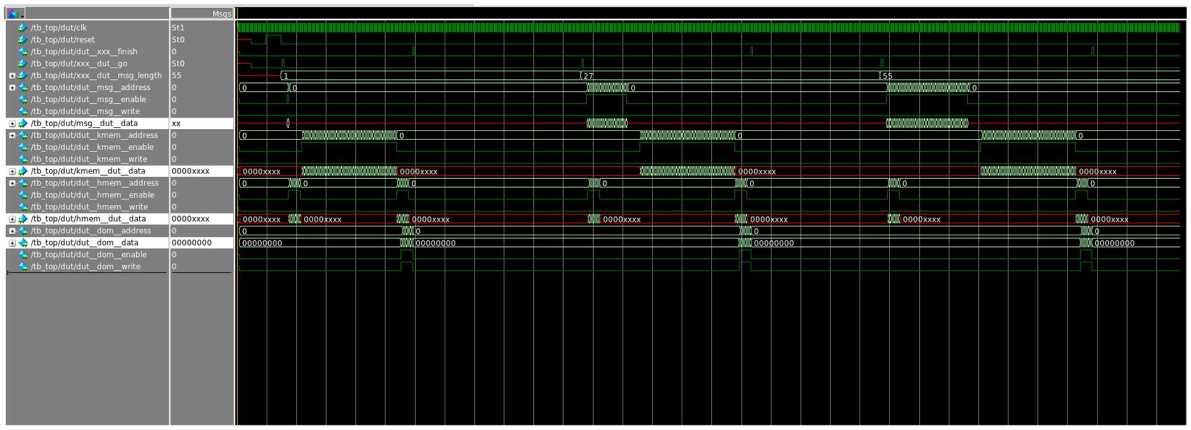

Msgs
/tb_top/dut/clk St1
/tb_top/dut/reset St0
/tb_top/dut/dut__xxx__finish 0
/tb_top/dut/xxx__dut__go St0
/tb_top/dut/xxx__dut__msg_length 55
/tb_top/dut/dut__msg__address 0
/tb_top/dut/dut__msg__enable 0
/tb_top/dut/dut__msg__write 0
/tb_top/dut/msg__dut__data xx
/tb_top/dut/dut__kmem__address 0
/tb_top/dut/dut__kmem__enable 0
/tb_top/dut/dut__kmem__write 0
/tb_top/dut/kmem__dut__data 0000xxxx
/tb_top/dut/dut__hmem__address 0
/tb_top/dut/dut__hmem__enable 0
/tb_top/dut/dut__hmem__write 0
/tb_top/dut/hmem__dut__data 0000xxxx
/tb_top/dut/dut__dom__address 0
/tb_top/dut/dut__dom__data 00000000
/tb_top/dut/dut__dom__enable 0
/tb_top/dut/dut__dom__write 0
1
127
155

Fig.7: Modelsim results and waveform of 3 different-message-length inputs

As is shown in Fig.7, I import three different test vectors in my testbench file, they maintain different message length of 1, 27 and 55 respectively. The output of testbench file is listed on the left side, and on the right side are the waveforms of I/Os in my top module.

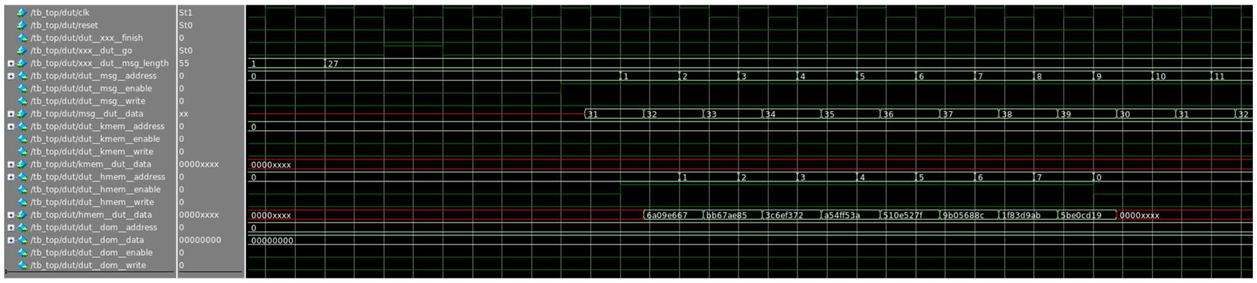

/tb_top/dut/clk
/tb_top/dut/reset
/tb_top/dut/dut__xxx__finish
/tb_top/dut/xxx__dut__go
/tb_top/dut/xxx__dut__msg_length
/tb_top/dut/dut__msg__address
/tb_top/dut/dut__msg__enable
/tb_top/dut/dut__msg__write
/tb_top/dut/msg__dut__data
/tb_top/dut/dut__kmem__address
/tb_top/dut/dut__kmem__enable
/tb_top/dut/dut__kmem__write
/tb_top/dut/kmem__dut__data
/tb_top/dut/dut__hmem__address
/tb_top/dut/dut__hmem__enable
/tb_top/dut/dut__hmem__write
/tb_top/dut/hmem__dut__data
/tb_top/dut/dut__dom__address
/tb_top/dut/dut__dom__data
/tb_top/dut/dut__dom__enable
/tb_top/dut/dut__dom__write

Fig.8: 27-length input waveform beginning part

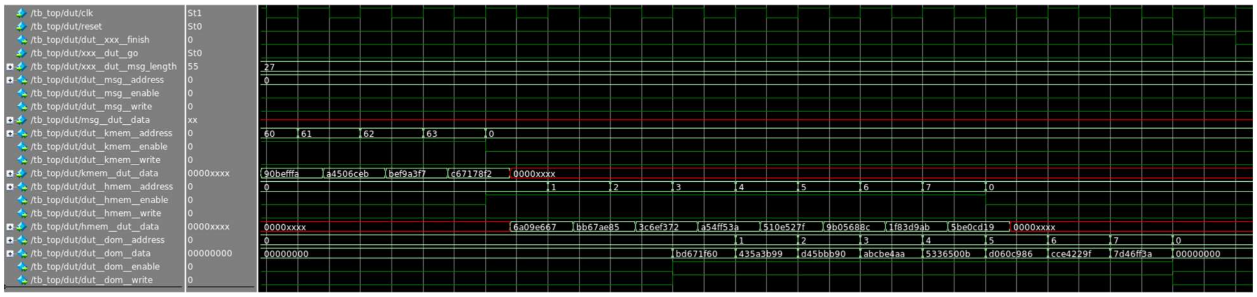

/tb_top/dut/clk
/tb_top/dut/reset
/tb_top/dut/dut__xxx__finish
/tb_top/dut/xxx__dut__go
/tb_top/dut/xxx__dut__msg_length
/tb_top/dut/dut__msg__address
/tb_top/dut/dut__msg__enable
/tb_top/dut/dut__msg__write
/tb_top/dut/msg__dut__data
/tb_top/dut/dut__kmem__address
/tb_top/dut/dut__kmem__enable
/tb_top/dut/dut__kmem__write
/tb_top/dut/kmem__dut__data
/tb_top/dut/dut__hmem__address
/tb_top/dut/dut__hmem__enable
/tb_top/dut/dut__hmem__write
/tb_top/dut/hmem__dut__data
/tb_top/dut/dut__dom__address
/tb_top/dut/dut__dom__data
/tb_top/dut/dut__dom__enable
/tb_top/dut/dut__dom__write

Fig.9: 27-length input waveform ending part

Fig.8 and Fig.9 focus on the 27-message-length input and demonstrate some details of the waveform.

In Fig.8, we can see that after xxx__dut__go goes high, the operation begins. Message and H started to read in as the addresses accumulate.

In Fig.9, we can see that after kmem__dut__data stops to read in, the iteration process is completed. The hmem__dut__date reads in for the second time, and dut__dom_data starts to output. After the output is done, dut__xxx_finish goes high for one clock period.

Optimization

Most of time I spent on this project goes into optimization. The initial cell area I got after my first synthesis was more than 30k um^2 and the clock period was 24 ns.

(1) Algorithm optimization

I have already discussed about the optimization I made on algorithm level in the innovation section. Most of the improvement in my design is based on this part. For example, reducing the size of W array from 64 blocks to 16 block results in a 13k um^2 area reduction, pipeline implementation reduces the clock period for about 5 ns. Also, I believe the updated version of Constraints.tcl file makes the clock period to reduce rapidly. I'm not sure about it but I think it allows Synopsys to synthesis a way faster clock.

(2) Circuit Sharing

I added some intermedia "nodes" in long calculation chains so that these "nodes" can be shared in different calculations.

For example,

```
assign T1 = sum1 + ch + H[7] + W_H_data + kmem__dut__data_r;
assign T2 = sum0 + maj;
assign a1 = T1 + T2;
assign e1 = H[3] + T1;
```

T1 and T2 are sharable "nodes".

This part does not contribute very much to area reduction. I believe one reason is that we have few sharing opportunities in this design.

7. Conclusions

This project provides me an opportunity to have a deeper acknowledgement for front-end ASIC design, especially for hierarchy structure. My Verilog coding is now more organized and methodical. However, my greatest achievement in this project is that I have a better understanding of optimization.

As I mentioned before, I spent most of the time on optimizing my design and the trade-off between area and clock period is the most common challenge. On algorithm level, for example, the pipeline technique increases area and reduce clock period at the same time. In synthesis, if I set the clock period to 25 ns, the cell area would be around 16k um^2. When the clock period was set to 7.5 ns, the cell area synthesized would increase to nearly 18k um^2.

In conclusion, my design can fulfill the function of simplified SHA-256 algorithm described in the project. The cell area and clock period demonstrated in the result section are the best-optimized results that I can get.