University of Cincinnati

Capstone Project:

Hardware implementation of SHA-256 Hash Function

Advising Faculty: Dr. Ranga Vemuri

Team member:

Duy Tran

Date: 11/06/2020

**I/ Overview**

The goal of this project is to design a digital circuit that is specialized in performing the SHA-256 Hash algorithm. SHA-256 Hash algorithm’s pseudocode is shown as below:

*Note 1: All variables are 32 bit unsigned integers and addition is calculated modulo 232*

*Note 2: For each round, there is one round constant k[i] and one entry in the message schedule array w[i], 0 ≤ i ≤ 63*

*Note 3: The compression function uses 8 working variables, a through h*

*Note 4: Big-endian convention is used when expressing the constants in this pseudocode,*

*and when parsing message block data from bytes to words, for example,*

*the first word of the input message "abc" after padding is 0x61626380*

*Initialize hash values:*

(first 32 bits of the *fractional parts* of the square roots of the first 8 primes 2..19):

h0 := 0x6a09e667

h1 := 0xbb67ae85

h2 := 0x3c6ef372

h3 := 0xa54ff53a

h4 := 0x510e527f

h5 := 0x9b05688c

h6 := 0x1f83d9ab

h7 := 0x5be0cd19

*Initialize array of round constants:*

(first 32 bits of the *fractional parts* of the cube roots of the first 64 primes 2..311):

k[0..63] :=

0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5, 0x3956c25b, 0x59f111f1, 0x923f82a4, 0xab1c5ed5,

0xd807aa98, 0x12835b01, 0x243185be, 0x550c7dc3, 0x72be5d74, 0x80deb1fe, 0x9bdc06a7, 0xc19bf174,

0xe49b69c1, 0xefbe4786, 0x0fc19dc6, 0x240ca1cc, 0x2de92c6f, 0x4a7484aa, 0x5cb0a9dc, 0x76f988da,

0x983e5152, 0xa831c66d, 0xb00327c8, 0xbf597fc7, 0xc6e00bf3, 0xd5a79147, 0x06ca6351, 0x14292967,

0x27b70a85, 0x2e1b2138, 0x4d2c6dfc, 0x53380d13, 0x650a7354, 0x766a0abb, 0x81c2c92e, 0x92722c85,

0xa2bfe8a1, 0xa81a664b, 0xc24b8b70, 0xc76c51a3, 0xd192e819, 0xd6990624, 0xf40e3585, 0x106aa070,

0x19a4c116, 0x1e376c08, 0x2748774c, 0x34b0bcb5, 0x391c0cb3, 0x4ed8aa4a, 0x5b9cca4f, 0x682e6ff3,

0x748f82ee, 0x78a5636f, 0x84c87814, 0x8cc70208, 0x90befffa, 0xa4506ceb, 0xbef9a3f7, 0xc67178f2

*Pre-processing (Padding):*

begin with the original message of length L bits

append a single '1' bit

append K '0' bits, where K is the minimum number >= 0 such that L + 1 + K + 64 is a multiple of 512

append L as a 64-bit big-endian integer, making the total post-processed length a multiple of 512 bits

*Process the message in successive 512-bit chunks:*

break message into 512-bit chunks

**for** each chunk

create a 64-entry message schedule array w[0..63] of 32-bit words

*(The initial values in w[0..63] don't matter, so many implementations zero them here)*

copy chunk into first 16 words w[0..15] of the message schedule array

*Extend the first 16 words into the remaining 48 words w[16..63] of the message schedule array:*

**for** i **from** 16 to 63

s0 := (w[i-15] **rightrotate** 7) **xor** (w[i-15] **rightrotate** 18) **xor** (w[i-15] **rightshift** 3)

s1 := (w[i- 2] **rightrotate** 17) **xor** (w[i- 2] **rightrotate** 19) **xor** (w[i- 2] **rightshift** 10)

w[i] := w[i-16] **+** s0 **+** w[i-7] **+** s1

*Initialize working variables to current hash value:*

a := h0

b := h1

c := h2

d := h3

e := h4

f := h5

g := h6

h := h7

*Compression function main loop:*

**for** i **from** 0 to 63

S1 := (e **rightrotate** 6) **xor** (e **rightrotate** 11) **xor** (e **rightrotate** 25)

ch := (e **and** f) **xor** ((**not** e) **and** g)

temp1 := h **+** S1 **+** ch **+** k[i] **+** w[i]

S0 := (a **rightrotate** 2) **xor** (a **rightrotate** 13) **xor** (a **rightrotate** 22)

maj := (a **and** b) **xor** (a **and** c) **xor** (b **and** c)

temp2 := S0 **+** maj

h := g

g := f

f := e

e := d **+** temp1

d := c

c := b

b := a

a := temp1 **+** temp2

*Add the compressed chunk to the current hash value:*

h0 := h0 **+** a

h1 := h1 **+** b

h2 := h2 **+** c

h3 := h3 **+** d

h4 := h4 **+** e

h5 := h5 **+** f

h6 := h6 **+** g

h7 := h7 **+** h

*Produce the final hash value (big-endian):*

digest := hash := h0 **append** h1 **append** h2 **append** h3 **append** h4 **append** h5 **append** h6 **append** h7

*Source:* [*https://en.wikipedia.org/wiki/SHA-2*](https://en.wikipedia.org/wiki/SHA-2)

Attached to this report are VHDL descriptions of the register-level design of the circuit. The VHDL are written in a way that should be able to be synthesized and loaded onto an FPGA for prototyping. There are also testbenches files for some of the critical components. The hireachy of the top most level files are:

Sha256.vhd

Sha256\_controller.vhd

State\_m0.vhd

State\_m1.vhd

State\_m2.vhd

LoopControl.vhd

sm1sm2\_overlapped\_signals.vhd

Mux\_123\_control.vhd

Mux\_5\_control.vhd

Mux\_7\_control.vhd

Sha256\_data.vhd

MainCir1.vhd

MainCir2.vhd

w\_reg.vhd

h\_reg.vhd

atoh\_reg.vhd

k\_reg.vhd

ch.vhd

maj.vhd

The circuit was simulated in ModelSim, with clock cycle with a period of 10ns (5 on/5 off). Hash rate was calculated as

1/t

(Hash per second)

Where t is the time (in seconds) it takes to complete a 512 bit-block hash. t is the time between the *start* pulse is sent and the *ready* signal is received.

**II/ Pseudocode for driving the circuit**

//Upon Circuit Startup//

For 5580\*half\_clock

test\_mode = ‘1’

test\_in = ‘0’

//////////////////////////////

bit\_array I(*message length*) = *Preprocessed and padded message*

For z = 0 to (l.length() / 512) do

bit\_array circuit\_input(512) = I[z\*512 to z\*512 + 511]

If ( z == 0)

For 2\* half\_clock

start = ‘1’

initial = ‘1’

else

For 2\* half\_clock

start = ‘1’

initial = ‘0’

wait for 2\* half\_clock

while (ready == ‘0’)

wait

bit\_array O(512) = o0 append o1 append o2 append o3 append o4 append o5 append o6 append o7

**III/ Components**

1. Basic components

These are the components that are built behaviorally in VHDL. They are the building blocks for the circuit. All other components (except state machines) are built structurally based on these components:

AND2x1

AND3x1

AND4x1

DFF\_POS (postive edge triggered flipflops)

NOT1x1

OR2x1

OR3x1

OR4x1

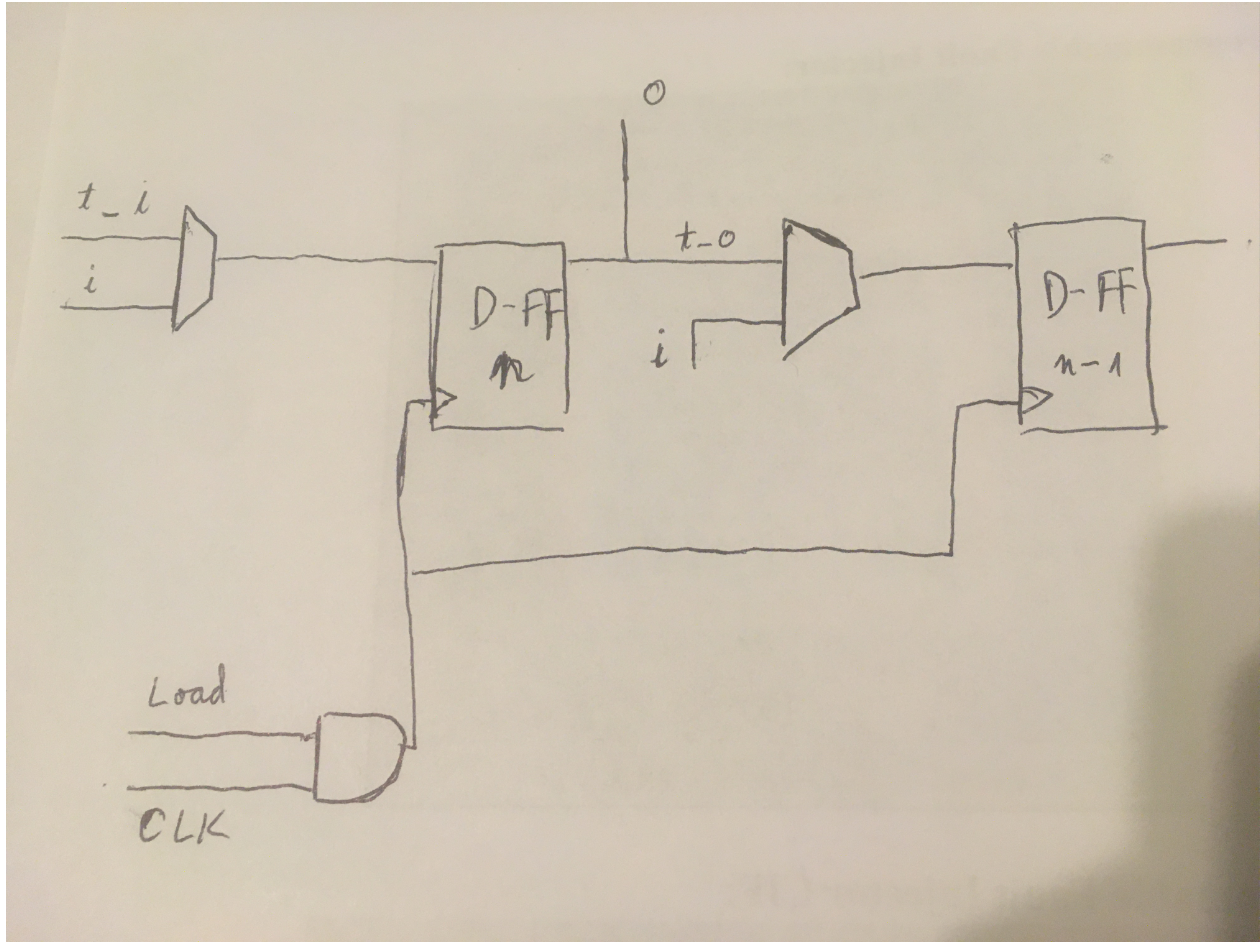
XNOR2x1

XOR2x1

XOR3x1

1. Register

The registers are built from the pos-edge D flip flop. The flip flops are grouped into 32 bits block for the data circuit and 6 bits block for the loop control circuit in the shift register configuration for purpose of testing. The schematic for a register block is shown as below:



The t\_c signal overide the input from the main circuit and force the register to take in single shift bit t\_i. When load signal is 0, the register will hold the previous value despite the clock signal. When load signal is 1, the register will take the value from upon clock’s rising edge.

1. Rotate\_shift

The rotate shift circuit are built from Rro.vhd, which contains 32 Rro\_mod.vhd modules. Each Rro\_mod modules is basically a 1x4 demultiplexer which will route the bit to its right rotated position. The control signals for said demux is as follow:

rc1 rc2 = ‘00’: Right rotate 1

= ‘01’ Right rotate 2

= ‘10’ Right rotate 6

= ‘11’ Right rotate 11

The sen signal enables bit masking for right shift operations when it is ‘1’

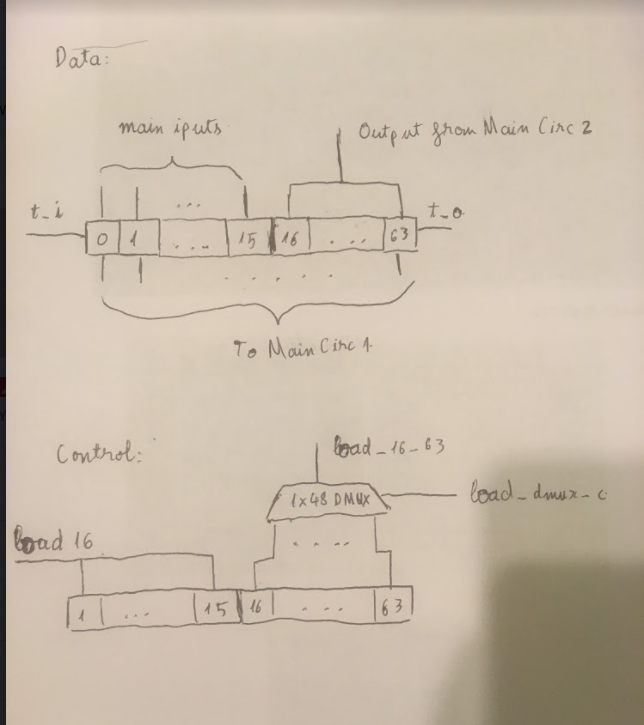
The sc signal control the masking mode: ‘0’ for masking 3 MSBs and ‘1’ for masking 10 MSBs

Ex: To perform right shift 3 right rotate the input 2 then right rotate the result 1 then mask 3 MSBs

1. Main 64 bits register (w\_reg)

This register is where the message 512 bits chunks are received (main input)

The diagram is shown as below. Note that each block is a 32 bits register block, each line is a 32 bits bus. *\*Exception: The control signals are all 1 bit line.* Clock, t\_c is implied to be the same for all registers.



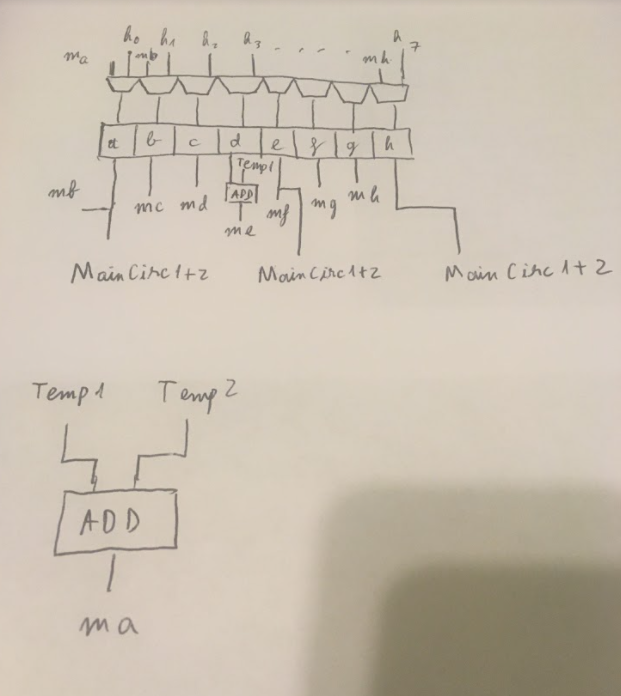
+ 2

The 1x48 Demux is built in a way such that the control signal starts at 16 and goes to 63 to switch the routes 0 to 47.

1. a to h reg

These are registers that holds the values of variables a to h in the pseudocode.

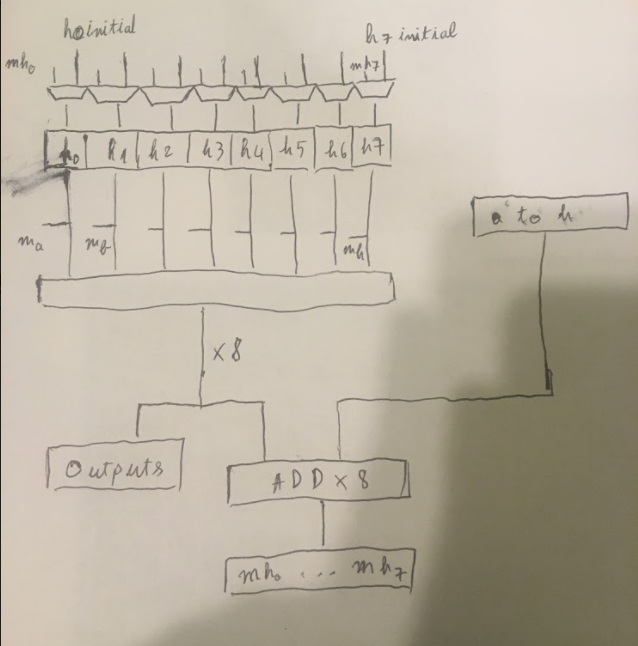
The diagram is shown below. Note that t\_c, clock, load\_atoh, and atoh\_mux\_c are shared between all blocks.



1. h\_reg

These are registers that hold the h0 to h7 values in the pseudo code. These values are also the hash digests.

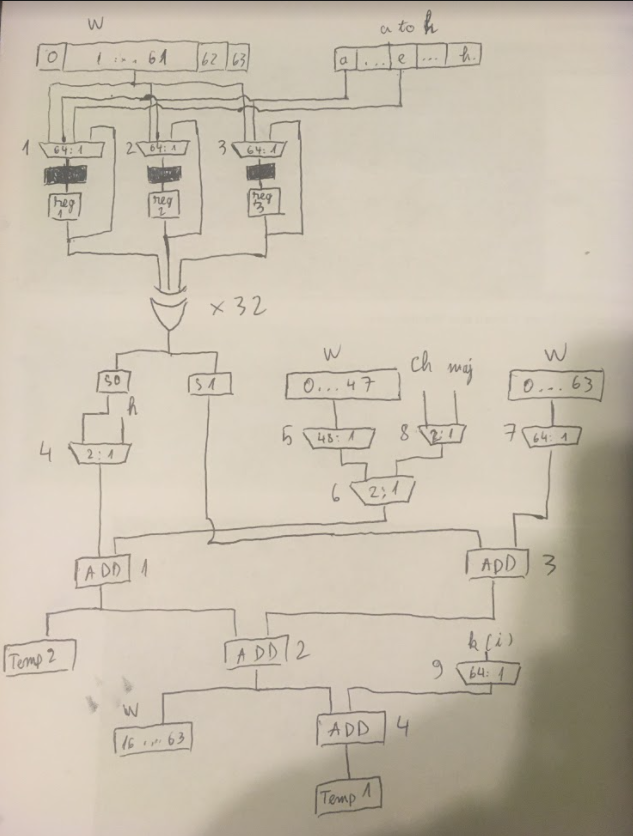
The diagram are shown below. Note that t\_c, clk, load\_h, h\_mux\_c are shared between all blocks. h0\_initial to h7\_initial are hardwired to the initial values in the pseudo code.



x 8

1. MainCirc

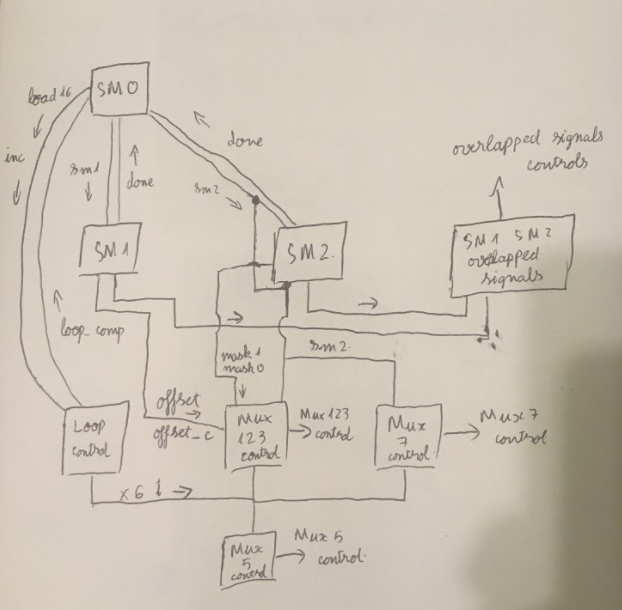
The Sha\_256 data circuit is shown in the diagram below. The squares (except for ADD) are all 32 bits registers. The black squares are the Rotate\_shift components. ch and maj are built in separate files. k(i) are hardwired as following the pseudocode guide. MainCirc1 is from the top to the register S0 and S1. MainCirc2 continues from mux4 to the end.



1. Controller

The controlling of muxes and registers load are splitted into 3 stages, each cotrolled by a finite state machine. State machine 1 and 2 are controlled by **state machine 0**. The state tables are in the attached excel file. Sheet 1 is **state machine 0**, sheet 2 is **state machine 1**, sheet 3 is **state machine 2**. The tables above are for control flow and the tables below are for the actual outputs of each state machine.

The diagram for controller is shown below:



SM0, SM1, SM2 are state machines

Loop control is a simple counter that starts at ‘0’ or ‘16’ depending on the load\_16 signal. The counter increment by 1 if signal incre is ‘1’ at clock rising edge. Loop control output signal loop\_comp to ‘1’ if the count reaches 63. Loop control output 6 bits signal **i** used by muxes that have their control values changes depending and the pseudocode’s for loop values.

SM1SM2 Overlapped signals is an array of 2:1 multiplexers that route the overlapped signals of **state machine 1** and **state machine 2**. Those signals are:

rc1\_1, rc2\_1, sen\_1, sc\_1

rc1\_2, rc2\_2, sen\_2, sc\_2

rc1\_3, rc2\_3, sen\_3, sc\_3

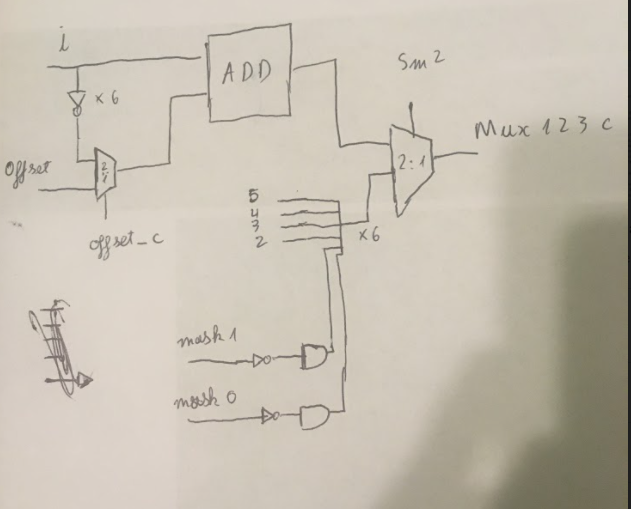
loadrg1, loadrg2, loadrg3

loads0, loads1

mux\_4\_c, mux\_6\_c

The muxes are control by signal sm2, meaning they will switch to **state machine 2**’s signals if sm2 is ‘1’. Note: **state machine 1**’s mux\_4\_c and mux\_6\_c are hardwired to 0.

Mux123 control diagram is shown as below:



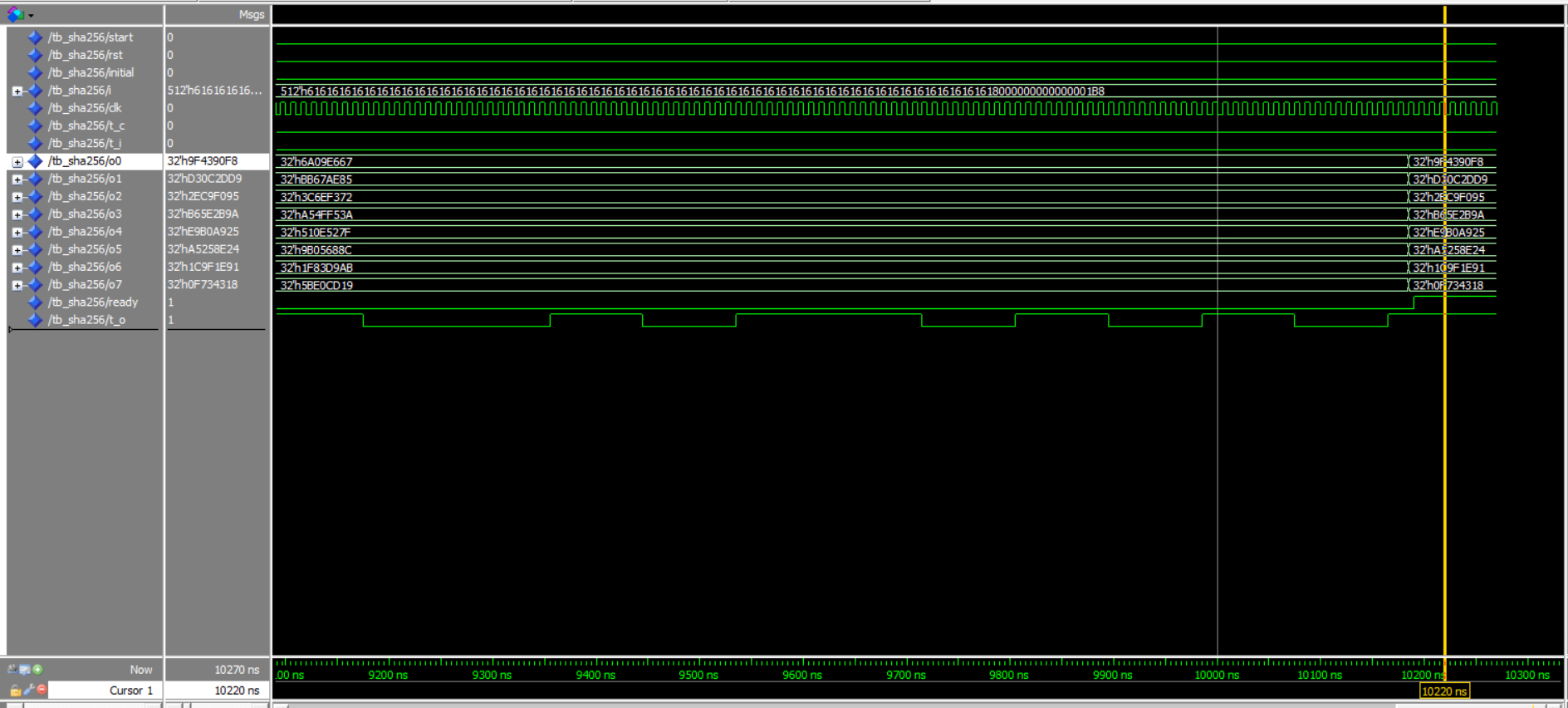
When sm2 is ‘0’, muxes 1,2,3 control is taken from the adder (controlled by **state machine 0**). When sm2 is ‘1’, muxes 1,2,3 cotrol takes a hardwired 63 (‘111111’) value, mask 1 active will make the control value become 61 (‘111101’), mask 0 active will make the control value become 62 (‘111110’)

Mux 5 control is an addition between the i value and hardwired -16 (‘110000’)

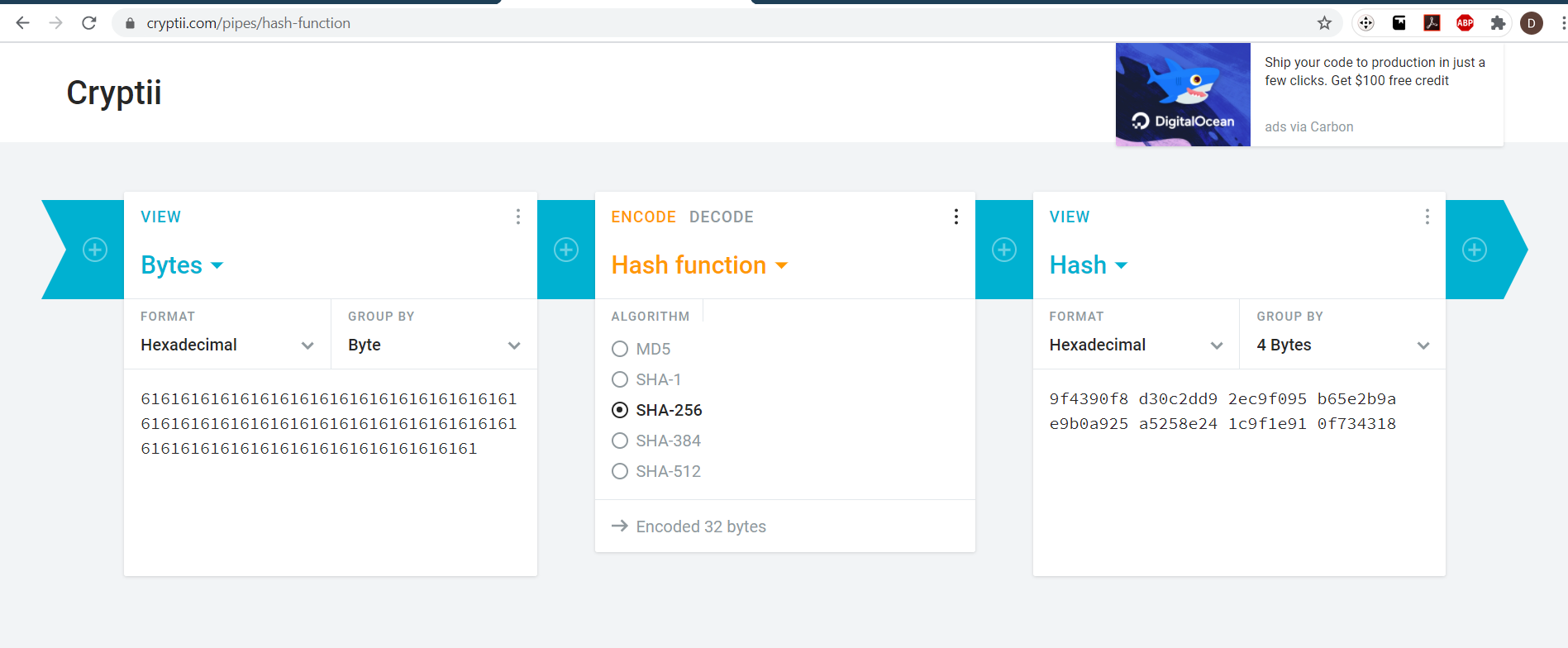
Mux 7 control is an addition between the i value and hardwired -7 (‘111001’) when sm2 is ‘0’, a hardwired 63 (‘111111’) when sm2 is ‘1’

**IV/ Results and discussion**

tb\_Sha256.vhd contains the testbench for 1 hash of the circuit. The message is 55 lowercase letter ‘a’, which is 110 number 61 hexadecimal. After padding, the block is exactly 512 bits



An online SHA256 hash for comparison:



The start signal rose at 70ns and the ready signal rose again at 10190ns. The clock has 10ns period. That means to complete 1 hash the circuit requires

(10190 - 70) / 10 = 1012 clock cycles.

The Nexys 4 DDR FPGA has 450 MHz clock frequency, which means a clock period of about 2.22 ns. Assuming such frequency, the potential hash rate of the circuit is:

1 / (2.22e-9 \* 1012) = 445 109 H/s = 0.445 MH/s

Assuming the circuit can be synchronized and work at a moderately powerful computer, which has average clock frequency at 3 GHz (clock period of 0.33 ns), the potential hash rate is:

1 / (0.33e-9 \* 1012) = 2994370 H/s = 2.99 MH/s

There are potential areas for improvement: at the cost for space we can use 6 Rotate\_shift components to calculate S0 and S1 simultaneously; we can also modify the Rotate shift component to have 12 modes of rotate or shift bits to avoid having to loop many time to rotate large step; the state machines are primarily Moore machines, perhaps Mealy machines can help reduce the number of states.

Power consumption testing is to be done in the future.