ECE 464/564 Projects : Fall 2018

***Students work on these projects individually**. Reminder: Collaboration is encouraged but the sharing of Verilog code is strictly forbidden. Also, group creation of one set of common code is forbidden. We will be running code comparison tools to look for even partial sharing of code.*

General information for all projects

A top level module will be provided that instantiates:

1. Message memory.
2. K constant memory
3. Initial H memory
4. Output memory
5. Your DUT

A testbench will be provided that instantiates the top level module and:

6. Provides a one-cycle wide go signal to start your design.
7. Waits for your design to send the finish flag high.
8. Counts the number of clock cycles it took to finish.
9. It will check your output for correctness

Note, after completing one full calculation, your design should be ready for the go flag to go high again and start another calculation.

Your design is implemented in a separate file MyDesign.v. Please make sure to synthesize only MyDesign.v and NOT the test fixture, nor the SRAM.

ECE464 and ECE 564-601 (EOL) Project

Your project is to design and implement hardware that is a subset of the ECE564 project. The objective is to implement up to the creation of the M_1 array and write each element, M_1[0] thru M_1[15] to the output memory.

ECE 564-001 Project

The objective is to implement the full SHA-256 operation and write the final hash array, H[0] through H[7] to the output memory.

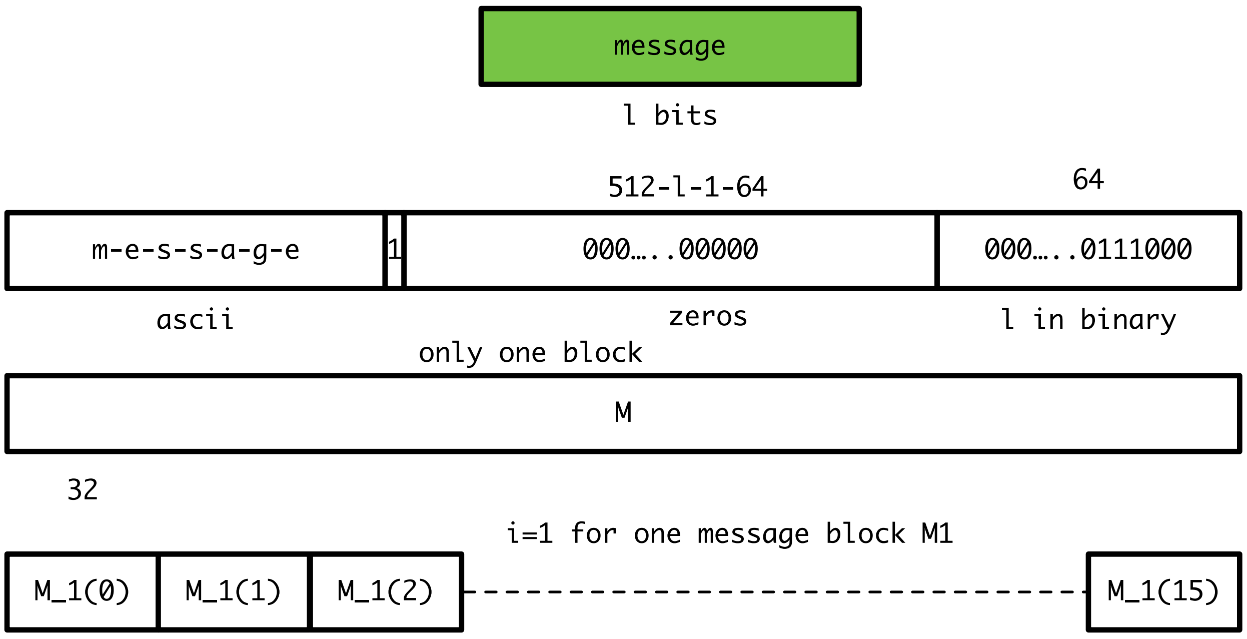
Project Description

This is a simplified version of the SHA-256 hash function. The project limits the length of the message to 55 characters thus ensuring there is only one 512-bit message block

The message will be contained as ascii in an SRAM that is outside of your verilog module. The length of the message is specified using a 6-bit number representing the number of ascii characters in the message. For example, if the message is ‘hello’ the length will be 6’d5. The message SRAM will contain 0x68, 0x65, 0x6C, 0x6C, 0x6F.

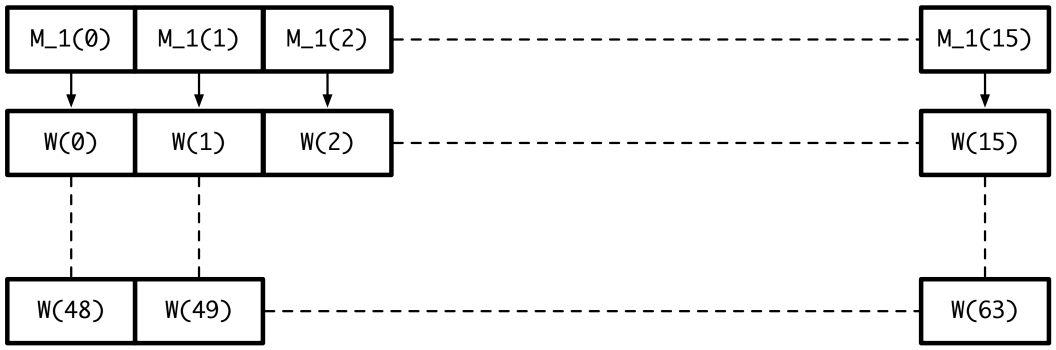
The SHA-256 process are shown below:

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). At this point, the ECE464 and ECE564-601 students will write the 16 words from M_1 into the output SRAM.

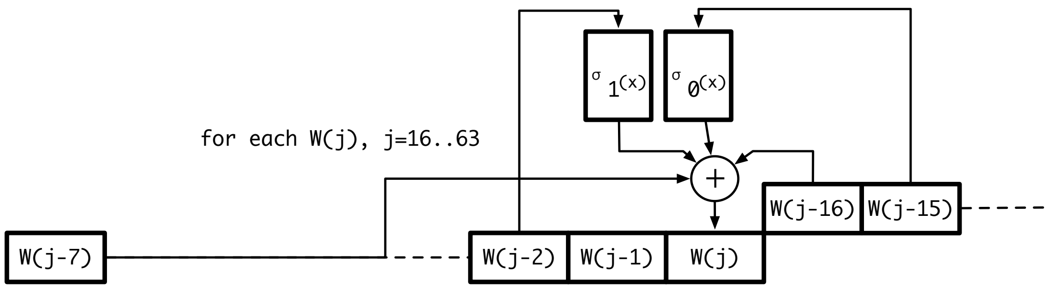


An example of the construction of the M vector and M_1 array is shown in the appendix.

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.

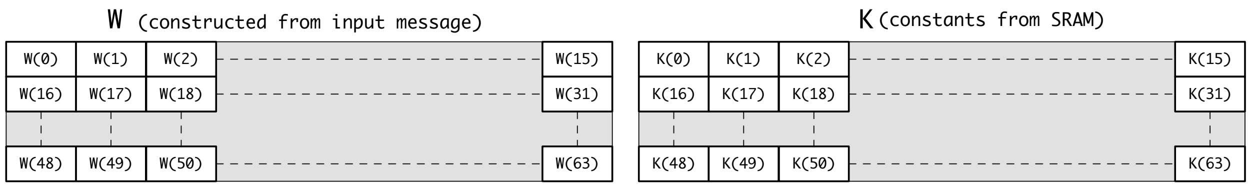


W[0..15] = M0..M15
W[16…63] =using combination of XOR and shift/rorate
e.g. W[i] = fn(W[i-1]..W[0])

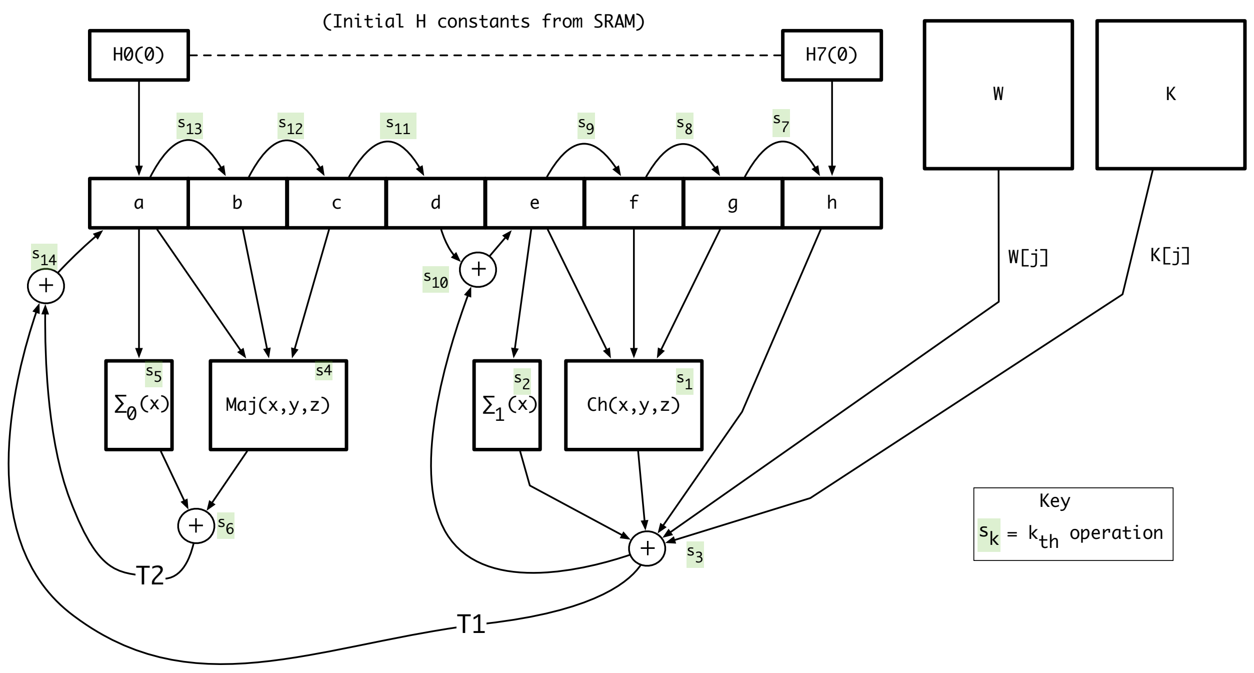


At this point we have constructed the 64 element W array.

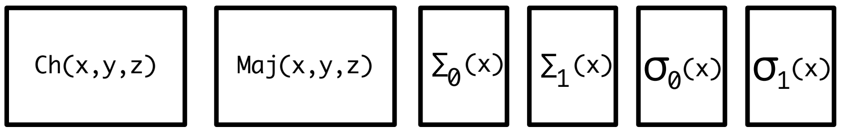
We then construct another 64 element array, K by reading 64 values from the K SRAM.



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.



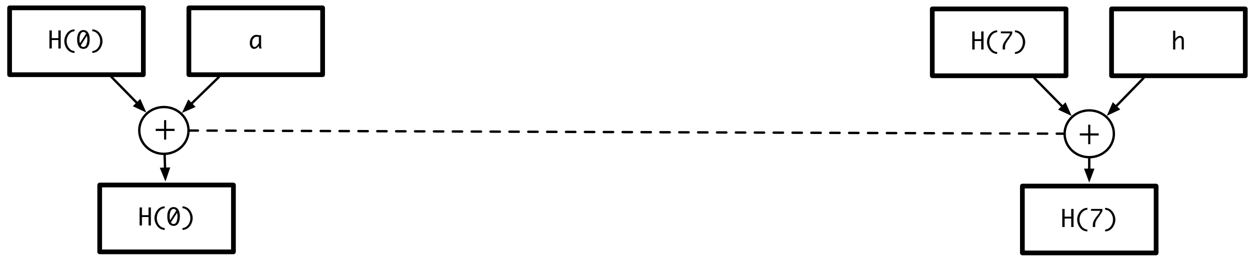
The various operations shown in the diagrams, such as, Maj, Ch, **σ , Σ** , S, R are shown below.



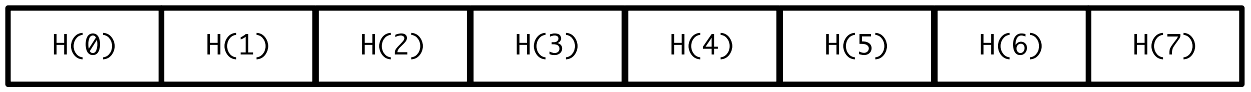
$$
\begin{aligned}
Ch(x,y,z) &= (x \wedge y) \oplus (\neg x \wedge z) \\
Maj(x,y,z) &= (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z) \\
\Sigma_0(x) &= S^2(x) \oplus S^{13}(x) \oplus S^{22}(x) \\
\Sigma_1(x) &= S^6(x) \oplus S^{11}(x) \oplus S^{25}(x) \\
\sigma_0(x) &= S^7(x) \oplus S^{18}(x) \oplus R^3(x) \\
\sigma_1(x) &= S^{17}(x) \oplus S^{19}(x) \oplus R^{10}(x)
\end{aligned}
$$

$\oplus$	bitwise XOR
$\wedge$	bitwise AND
$\vee$	bitwise OR
$\neg$	bitwise complement
$+$	mod 2^{32} addition
R^n	right shift by n bits
S^n	right rotation by n bits

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.



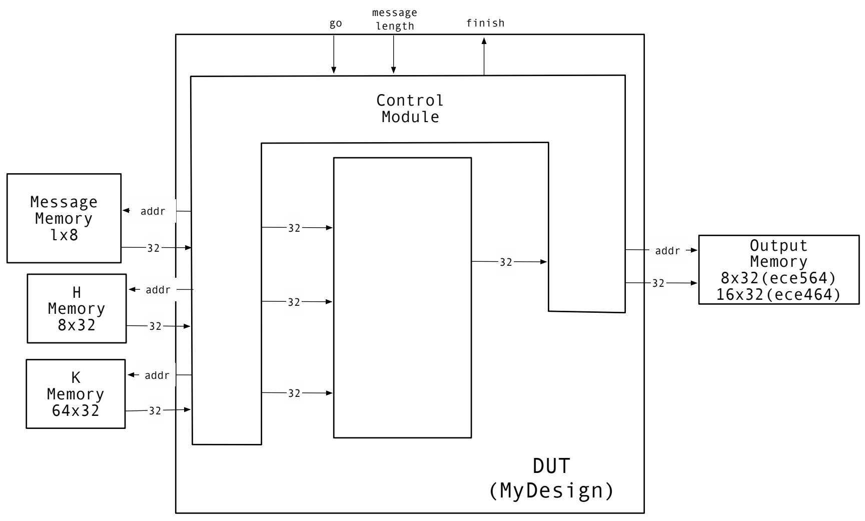
Hash of message, M

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

You will be provided with python code that demonstrates all the steps to help you debug your code.

Summary

Below is a block diagram.



The length of the message will be input on the “message length” signal, xxx__dut__msg_length. The “go” input will pulse high to start the design. The message is stored as a verilog dat file and will be preloaded into the message memory. The H memory and K memory will also be loaded by the testbench. Your design will need to read the contents of the message memory based on the message length. e.g. if the message is “hello”, the message length will be 6’d5 and you will read

address location 0 through 4 of the message memory. You will read the contents of the H memory and K memory starting at address 0.
The order in which you perform memory accesses or SHA-256 operations is entirely up to you.
Once you have written the result to the output memory, your design will assert finish and keep it asserted until another "go" pulse arrives. Your design will deassert finish and only reassert once you have written to the output memory.
When your design is reset, it will deassert finish and all other memory control signals.

Comments

Additional specifications:

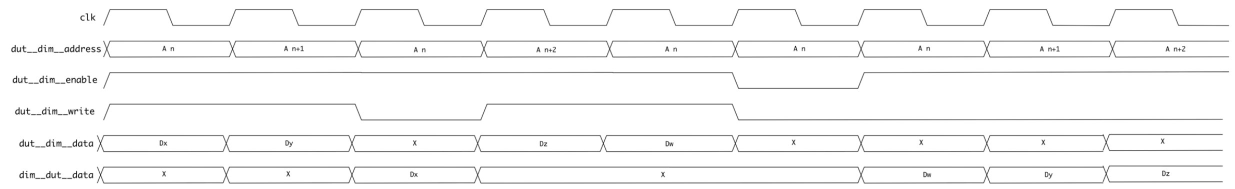
- You can use Synopsys DesignWare or design your own arithmetic units.
- A test fixture and SRAMs will be provided.

Interfaces

Signals

Direction	**Type**	**Width (Bus)**	**Name**	**Comment**
Control				
output	reg		dut__xxx__finish	High when DUT is ready for a ‘go’. Deassert after ‘go’
input	wire		xxx__dut__go	Pulsed
input	wire	[5:0]	xxx__dut__msg_length	Length of message in bytes
K constant Memory				
output	reg	[5:0]	dut__kmem__address	
output	reg		dut__kmem__enable	High for Read
output	reg		dut__kmem__write	Not used, Low for Read
output	reg	[31:0]	dut__kmem__data	Not used. Write Data
input	wire	[31:0]	kmem__dut__data	Read Data
H initial Data memory				
output	reg	[8:0]	dut__hmem__address	
output	reg		dut__hmem__enable	High for Read and Write
output	reg		dut__hmem__write	High for write
output	reg	[31:0]	dut__hmem__data	Write Data
input	wire	[31:0]	hmem__dut__data	Read Data
Message memory				
output	reg	[8:0]	dut__msg__address	
output	reg		dut__msg__enable	High for Read and Write
output	reg		dut__msg__write	Not used, High for write
output	reg	[7:0]	dut__msg__data	Not used, Write Data
input	wire	[7:0]	msg__dut__data	Read Data
Output Data Memory				
output	reg	[3:0]	dut__dom__address	
output	reg	[31:0]	dut__dom__data	Write Data
output	reg		dut__dom__enable	High for Write
output	reg		dut__dom__write	High for write
General				
input	wire		clk	
input	wire		reset	Active high

Example RAM Signaling



Compile and Simulation

```
vlog -sv ece564_project_tb_top.v
```

```
vsim -c -do "run -all; quit" tb_top
```

```
vlog -sv ece564_project_tb_top.v
```

Example M vector and M_1 array

```
Assume the message is “Hello”
```

```
The message.dat file will contain:
```

```
@0 48
@1 65
@2 6c
@3 6c
@4 6f
```

```
The 5-character signal 'Hello' converted to binary forms is:
```

```
0100100001100101011011000110110001101111
```

```
After appending a ‘1’, padding with zeroes and adding the length in bits (64’d40) the 512-bit M
vector will be:
```

```
0100100001100101011011000110110001101111100000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000101000
```

```
The M_1 array will be as follows:
```

```
M_1[1] = 32’h48656c6c
M_1[2] = 32’h6f800000
M_1[3] = 32’h00000000
M_1[4] = 32’h00000000
M_1[5] = 32’h00000000
M_1[6] = 32’h00000000
M_1[7] = 32’h00000000
M_1[8] = 32’h00000000
M_1[9] = 32’h00000000
M_1[10] = 32’h00000000
M_1[11] = 32’h00000000
M_1[12] = 32’h00000000
M_1[13] = 32’h00000000
M_1[14] = 32’h00000000
M_1[15] = 32’h00000000
M_1[16] = 32’h00000028
```