ECE564 ASIC & FPGA Design
Final Project

Name: Soumil Krishnanand Heble
UnityID: sheble
StudentID: 200207079

Delay (ns to run provided example). **Clock period:** 8 ns **# Clock Cycles:** 104 (Message: Hello)	**Logic Area (um^2):** 15852.5362 **Memory:** N/A	**1/(delay.area) ($ns^{-1}.um^{-2}$)** 7.58E-8
Delay (TA provided example. TA to complete)		**1/(delay.area) (TA)**

Abstract: Designed a simplified SHA256 hashing module using synthesizable Verilog RTL and achieved a cell area of 15852.5362 um^2 using 45 nm Nandgate OpenCell library. The designed module reads in the message (ASCII) to be hashed, H and K values from an SRAM and writes the result to an output SRAM. The module starts its computation when a go signal is asserted and after writing the result to the output SRAM a finish signal is asserted indicating the hash value has been computed and written to memory. The module is also supplied with the number of characters in the message before the go signal is asserted. The simplification in this implementation of the SHA256 comes from the fact that the message length is restricted to 55 characters leading to the creation of only one 512-bit block. On an average the design takes 121 cycles @ 125 MHz and all the inputs and outputs are registered leading to glitch free IO.

Simplified SHA256 Hashing Algorithm

Soumil Krishnanand Heble

Introduction

Hardware being designed:

- A hardware implementation of the SHA256 algorithm that takes in a message of maximum 55 characters (ASCII) from an SRAM and writes the computed has to an output SRAM. The message being limited to 55 characters results in the creation of only one 512-bit message block.
- The hardware module reads the H and K values from SRAMs and starts the hash computation after a go signal is asserted. The module also takes in the number of characters in the message SRAM.
- Upon completion of the hash value write to the output SRAM a finish signal is asserted.

Summary of Key Innovations:

- Since the message is limited to 55 characters, only 447 flip-flops are required to create the initial padded block as per SHA2 specifications.
    - Assuming a message of 55 characters
        - 55 ASCII characters = 440 bits
        - Append 0x80 = 1 bit (Only requires the MSB rest can be logic 0)
        - Zero Padding = 0 bit (logic 0 – no flip flops required)
        - Message Length = 6 bits (Length = 440, of the 16 bits used for length only requires 6 bits can be either 1 or 0 rest can be logic 0)
        - Total Flip-Flops = 447
- Since the W values are required for the message digest calculation sequentially the values are calculated and supplied on the fly to the hash computation module. This saves the need to use 64 32-bit wide registers. This requirement also means that as soon as the 512 bit padded message block is ready the hash computation can be started and can run parallel to the W value calculation.
- The W computation is pipelined and the usage of parallel prefix adders from the Designware library allows the clock period to be squeezed down to 6 ns.
- The sequence of operations required for H computation is parallelized and uses parallel prefix adders from the Designware library.
- The above design choices result in a cell area of 15852.5362 um^2.

Summary of Results Achieved:

- Area
  - 15852.5362 um^2
- Number of Cycles
  - Message_1 = 100, Length = 1
  - Message_5 = 104, Length = 5
  - Message_27 = 126, Length = 27
  - Message_55 = 154, Length = 55
  - *** Average = 121
- Clock Period
  - 8 ns
- Performance (Area * # Cycles * Clock Period, units = um^2 ns)
  - Message_1 = 12682028.96
  - Message_5 = 13189310.12
  - Message_27 = 15979356.49
  - Message_55 = 19530324.59
  - *** Average = 15345255.04

Micro-Architecture

Hardware "Algorithmic" Approach Used:

While the synthesizable Verilog RTL was being written it was developed as three separate modules namely gen_padded, gen_w and gen_h but for the final submission the code from all the three modules was combined into a single synthesizable Verilog RTL file named MyDesign.v. This was done so that the interfaces as specified in the project description can be maintained.

The three modules if chained can perform the same operation and result in the same area, performance and their IO's are registered as well. The code fragments from the respective modules start with the comment "/*>>>>> <module_name> ******/" with the module names being either of gen_padded, gen_w or gen_h. Since the IOs are registered the delay in the signal/data/address arrival is handled by using state machines and delay using register chains in all the three modules.

The algorithmic approach explained is split in three as per the above modules:
gen_padded: Creates the padded message required by SHA2 algorithm by reading in data from the message SRAM.

- **State Machine:**
  - **State P0 (IDLE):** Wait in this state until the go signal is asserted.
  - **State P1 (LATCH_MESSAGE_LEN):** Reads in the message length provided and clear the required bits in the 512 bit padded message register then move to the next state.

  - **State P2 (READ_SRAM):** Present address to SRAM and write message length to the required position in the 512 bit padded message register. Move to the next state once the SRAM request address equals the message length.
  - **State P3 (DELAY):** Delay state so that the data from SRAM is written (This state is required due to registering IOs). Move to the next state unconditionally.
  - **State P4 (WRITE_0x80):** Append 0x80 after the message is read and move to the next state unconditionally.
  - **State P5 (FINISH_WAIT):** Assert a pad register ready signal to the gen_w module and wait for the next go and finish to be high then start the cycle again from State P1.
- The address presented to the SRAM and the enable signal is delayed internally by using chained registers and used to index into the pad register. The delayed enable is used as a write enable for the pad register. This saves from using additional counter and a larger FSM for handling the delay of registered IO.

gen_w: Reads in the 512 bit padded message from gen_padded module and computes and presents the W values upon request from gen_h module (hash computation module). The gen_h module has an SRAM like interface to the gen_w module to request for W values. It has to present the gen_w module with an address and enable signal to read out the values.

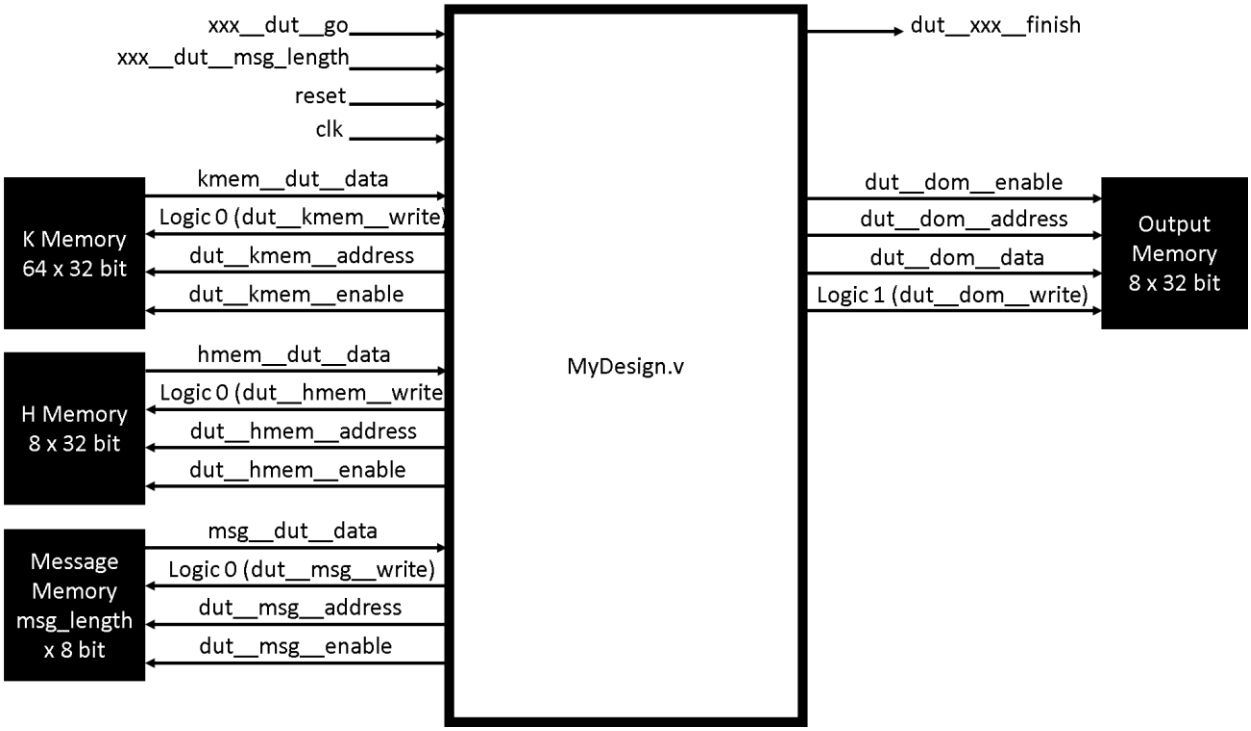
- **State Machine:**
  - **State W0 (IDLE):** Wait in this state until the pad message ready signal is asserted by the gen_padded module.
  - **State W1 (READ_IN_PAD):** Read in the padded message in to the 16 32-bit register for W value computation then move to next state.
  - **State W2 (SERVICE_REQ):** Wait for W value request from gen_h module and service the value if the requested address matched the current value to be serviced count. Move to State W3 if the current value to be serviced counter overflows.
  - **State W3 (WAIT_!PAD_RDY):** Wait for the pad ready signal to be de-asserted then move to State W0.
- The W value calculation is pipelined and happens on the fly. First 16 W values are ready as soon as the padded message is read in to the 16 32-bit registers. The W values to be read for the next value to be computed (initially the 17^{th} value) is picked from the 1^{st}, 2^{nd}, 9^{th} and 15^{th} locations from the 16 32-bit registers. The 16 32-bit registers act as a FIFO queue upon receiving a read request from the gen_h module the value in the top or 1^{st} register is placed on the data line and the data in the following registers are moved up one position.
- The W computation is pipelined and the results are registered into the base of the 16 32-bit registers.

gen_h: Reads a-h values from H SRAM, W from gen_w module and constants from the K SRAM and computes the hash of the message and writes it to the output SRAM.

- **State Machine:**
  - **State M0 (IDLE):** Wait in this state until the go signal is asserted.
  - **State M1 (READ_AH):** Copy the a-h values from H SRAM into the 8 32-bit registers for has computation. When the address counter feeding address to the SRAM overflows move to the next state.
  - **State M2 (WAIT_FOR_W):** Wait for the W module to read in the padded message and get ready to service the address request. Move to the next state once W module is ready.
  - **State M3 (READ_WK_0):** Start reading the W values from the W module and the K values from the K SRAM. Move to the next state unconditionally.
  - **State M4 (WAIT_0):** Wait for the request to propagate and W and K data to arrive. Move to the next state unconditionally.
  - **State M5 (WAIT_1):** Wait for the request to propagate and W and K data to arrive. Move to the next state unconditionally.
  - **State M6 (WAIT_2):** Wait for the request to propagate and W and K data to arrive. Move to the next state unconditionally.
  - **State M7 (H_COMPUTE):** Perform the H operations. Increment the current iteration counter. Move on to State M8 if the current iteration counter overflows else go to State M7.
  - **State M8 (ADD_AH):** Read in the a-h values from H SRAM and add them to the a-h hash computation registers. Move to the next state once the SRAM request address counter overflows.
  - **State M9 (WAIT_ADD_0):** Wait for the pending additions to complete. Move to the next state unconditionally.
  - **State M10 (WAIT_ADD_1):** Wait for the pending additions to complete. Move to the next state unconditionally.
  - **State M11 (WRITE_OP):** Write the computed hash values to the output SRAM. Move to the next state once the output SRAM write address overflows.
  - **State M12 (WAIT):** Assert finish signal. Move to State M1 if go signal is asserted again.
- The H SRAM enable signals and address are passed to a chained register delay chain and used as write enable and index address to the a-h registers.

Interface Specification

Top Level Module Interfaces



Top Level Module Interface Table and Description

Direction	Type	Width (Bus)	Name	Description
Input	Wire	1 bit	clk	Clock Signal
Input	Wire	1 bit	reset	Synchronous Reset Signal
Input	Wire	1 bit	xxx__dut__go	Go Pulse To Start SHA256 Computation
Input	Wire	6 bit	xxx__dut__msg_length	Number of Characters in Message SRAM
Output	Reg	1 bit	dut__xxx__finish	Hash Written to Output SRAM Signal
Output	Reg	1 bit	dut__msg__enable	Enable SRAM
Output	Reg	1 bit	dut__msg__write	SRAM !R/W Select (Always Logic 0)
Output	Reg	6 bit	dut__msg__address	SRAM Access Address
Input	Wire	8 bit	msg__dut__data	Data from SRAM Read Port
Output	Reg	1 bit	dut__kmem__enable	Enable SRAM
Output	Reg	1 bit	dut__kmem__write	SRAM !R/W Select (Always Logic 0)
Output	Reg	6 bit	dut__kmem__address	SRAM Access Address
Input	Wire	32 bit	kmem__dut__data	Data from SRAM Read Port
Output	Reg	1 bit	dut__hmem__enable	Enable SRAM
Output	Reg	1 bit	dut__hmem__write	SRAM !R/W Select (Always Logic 0)
Output	Reg	3 bit	dut__hmem__address	SRAM Access Address
Input	Wire	32 bit	hmem__dut__data	Data from SRAM Read Port
Output	Reg	1 bit	dut__dom__enable	Enable SRAM
Output	Reg	1 bit	dut__dom__write	SRAM !R/W Select (Always Logic 1)
Output	Reg	3 bit	dut__dom__address	SRAM Access Address
Output	Reg	32 bit	dut__dom__data	Data to SRAM Write Port

Verification

The synthesizable Verilog RTL and the synthesized netlist were verified rigorously for maximum possible operating conditions using the provided sample test bench and a custom test bench. The obtained hash values were validated by using the results from a high level language implementation of the module in python for the same input messages.

The provided sample test bench applies a go signal and waits for the finish line to be asserted then repeats the process again one more time. Meanwhile it snoops the output memory write line when the output memory enable line is asserted and writes the data to a result text file for each time the go signal is asserted. Using this test script the hash of four different messages of were computed.

The four different messages tested are:

- Message_1 = a
- Message_5 = Hello
- Message_27 = abcdefghijklmnopqrstuvwxyza
- Message_55 = abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabc

The second custom test bench tested the synthesizable Verilog RTL for two test cases.

- The first case tests the design by pulsing the go signal while a hash computation is running. This test makes sure that the design does not respond to rogue go pulses while a computation is running.
- The second case asserts the go line indefinitely and lets the computation run twice. This test makes sure that the module is ready for the next computation immediately after it finishes a run.

Verification Result: The synthesizable Verilog RTL and synthesized netlist passed all the tests successfully.

Following are some screenshots of the signals of the top module when the sample test bench provided is run with the message “Hello” of length 5:

```
VSIM 2> run -all
# INFO::readmem : message.dat
# INFO::readmem : K.dat
# INFO::readmem : H.dat
# @315, go asserted
# Hash[0] = 185f8db3
# Hash[1] = 2271fe25
# Hash[2] = f561a6fc
# Hash[3] = 938b2e26
# Hash[4] = 4306ec30
# Hash[5] = 4eda5180
# Hash[6] = 07d17648
# Hash[7] = 26381969
# @1365, dut finished, # of clocks = 104
# Store result to result_0.txt
# @5335, go asserted
# Hash[0] = 185f8db3
# Hash[1] = 2271fe25
# Hash[2] = f561a6fc
# Hash[3] = 938b2e26
# Hash[4] = 4306ec30
# Hash[5] = 4eda5180
# Hash[6] = 07d17648
# Hash[7] = 26381969
# @6385, dut finished, # of clocks = 104
# Store result to result_1.txt
# ** Note: $finish    : ece564_project_tb_top.v(186)
#    Time: 10335 ns  Iteration: 1  Instance: /tb_top
# 1
# Break in Module tb_top at ece564_project_tb_top.v line 186
```

Figure 1 - Modelsim Test bench Run Transcript

```
$ python sha256Verification.py Hello | tail -13
Finished the 64-iteration loop
H1: 0x185f8db3
H2: 0x2271fe25
H3: 0xf561a6fc
H4: 0x938b2e26
H5: 0x4306ec30
H6: 0x4eda5180
H7: 0x07d17648
H8: 0x26381969

sha256 result: 185f8db32271fe25f561a6fc938b2e264306ec304eda518007d1764826381969
reference result: 185f8db32271fe25f561a6fc938b2e264306ec304eda518007d1764826381969
The results match
```

Figure 2 - Hash of the message "Hello" using a High Level Language Implementation of the Module in Python

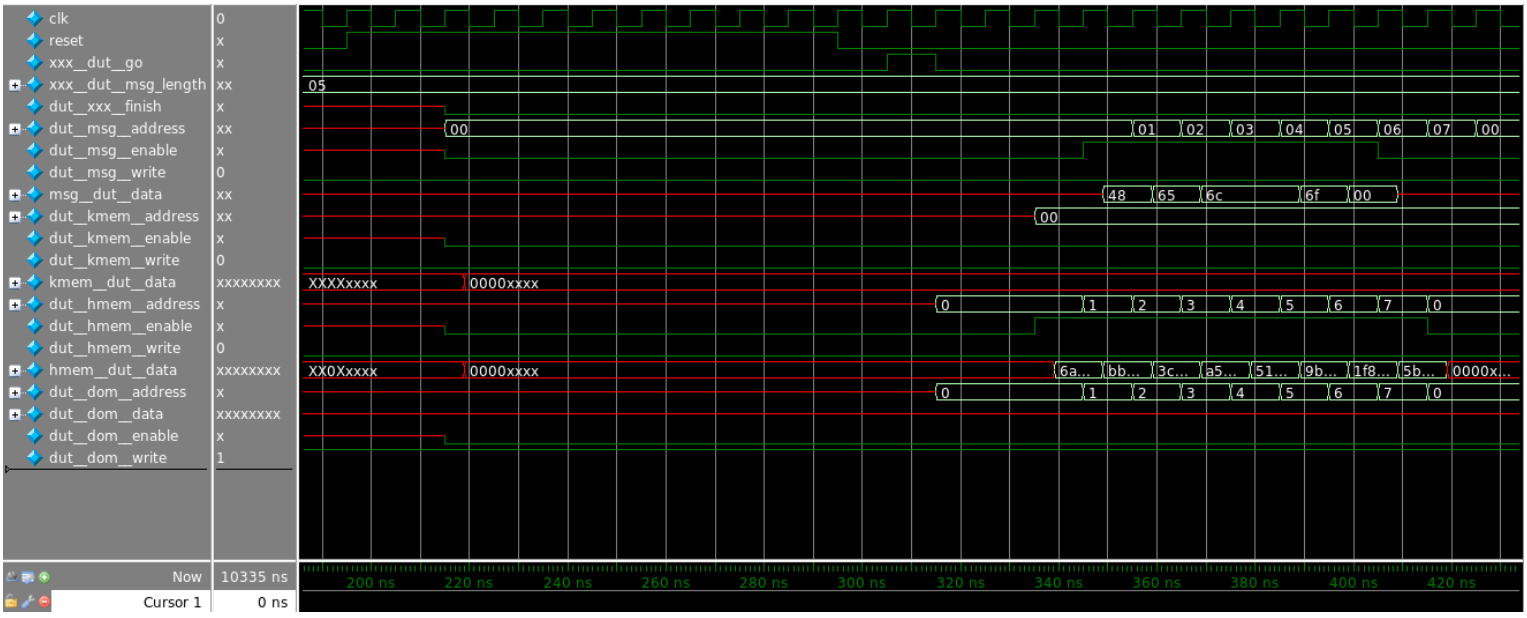


Figure 4 - Top Level Module Signal Waveform I

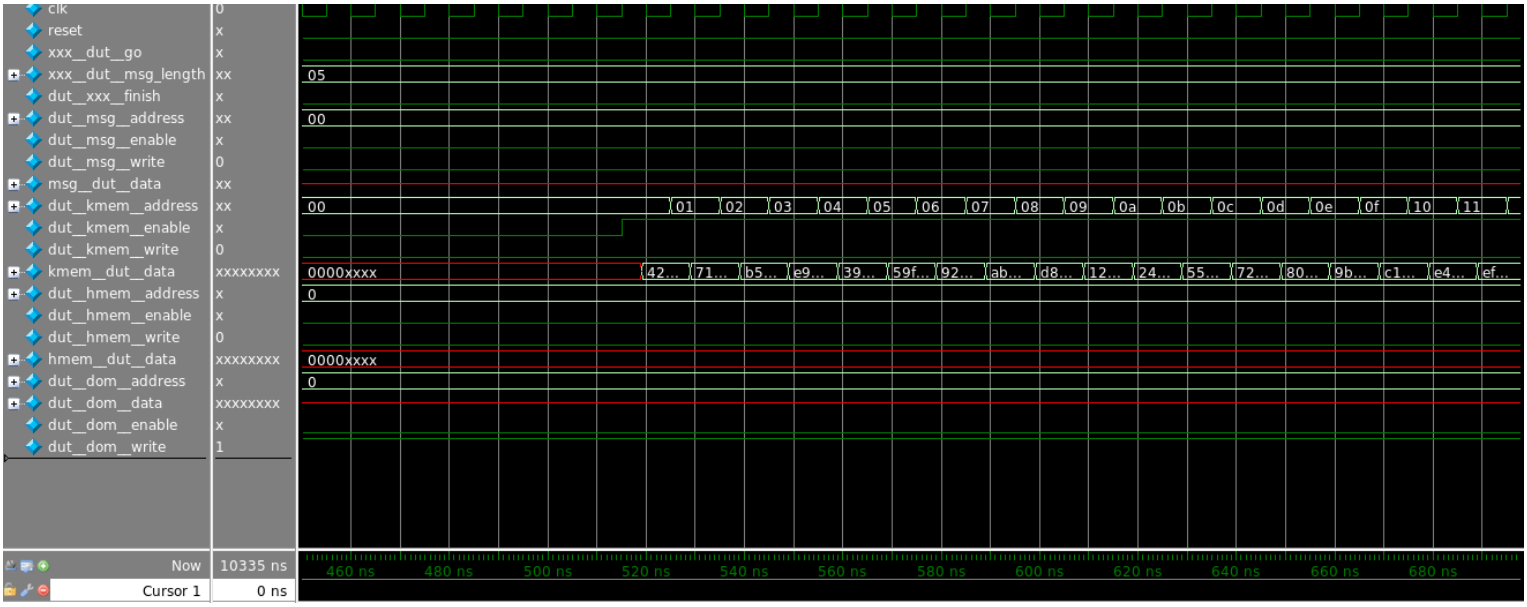


Figure 3 - Top Level Module Signal Waveform II

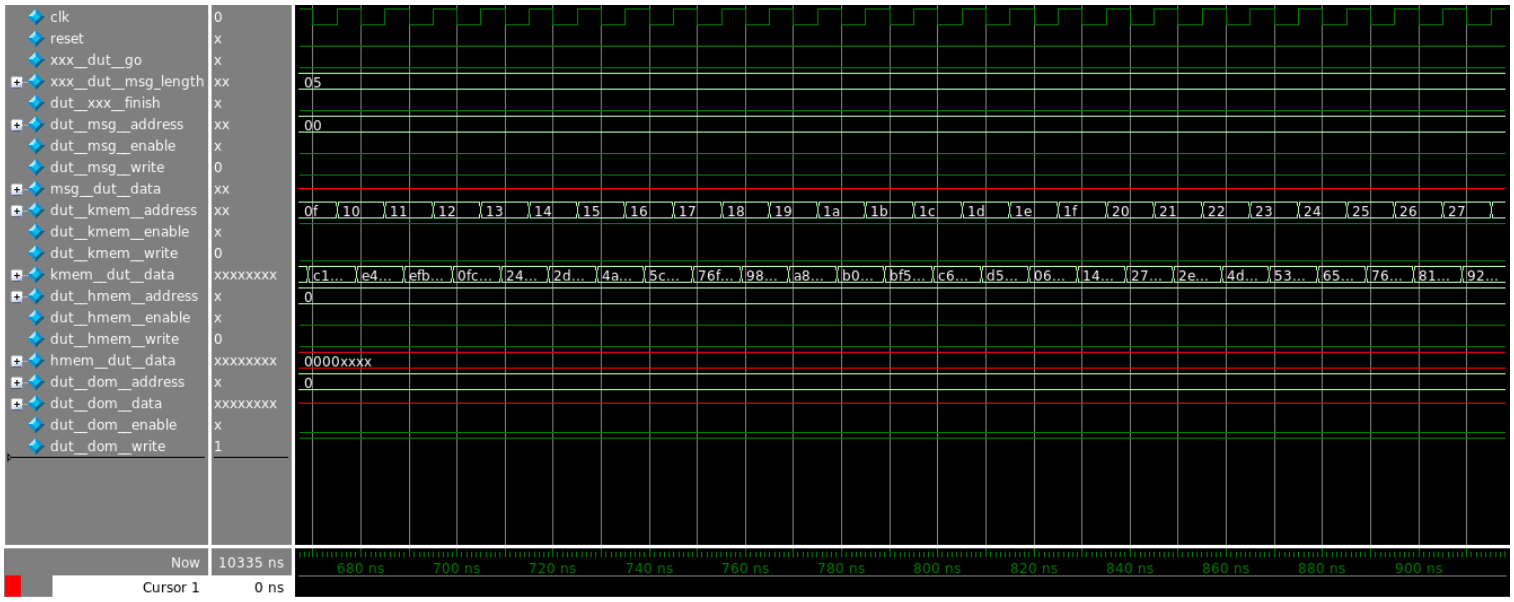


Figure 6 - Top Level Module Signal Waveform III

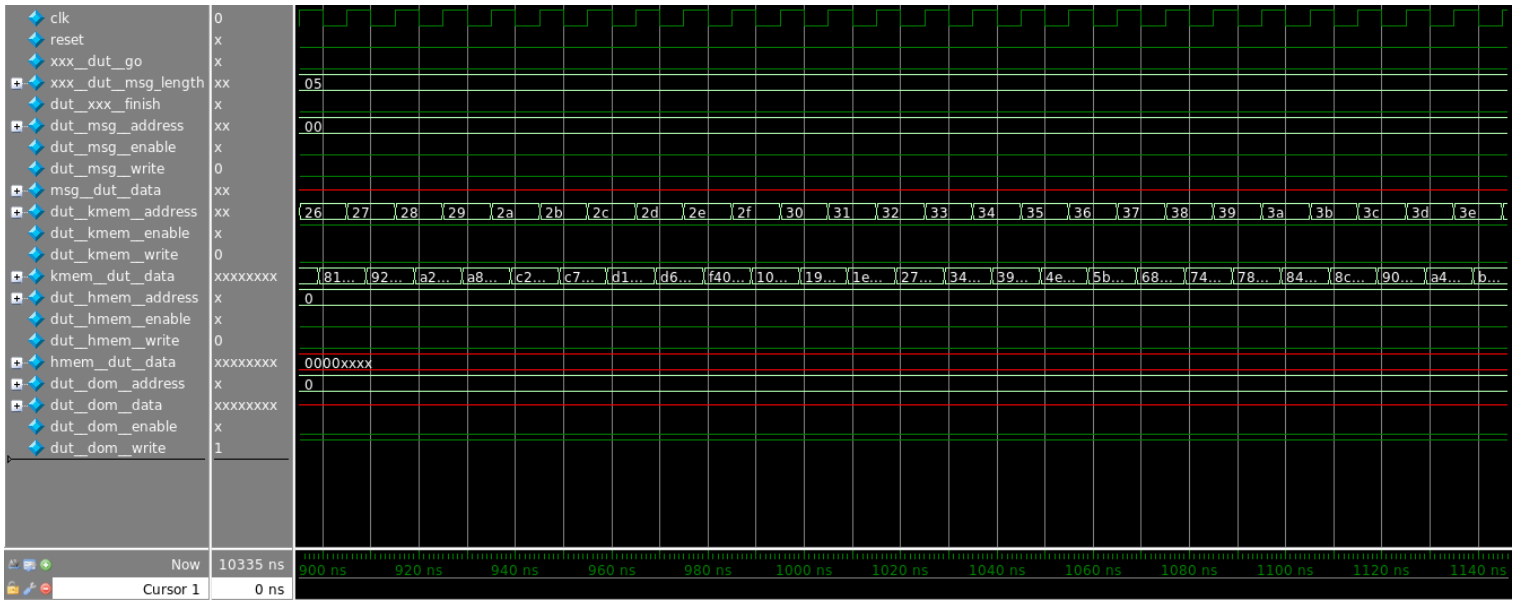


Figure 5 - Top Level Module Signal Waveform IV

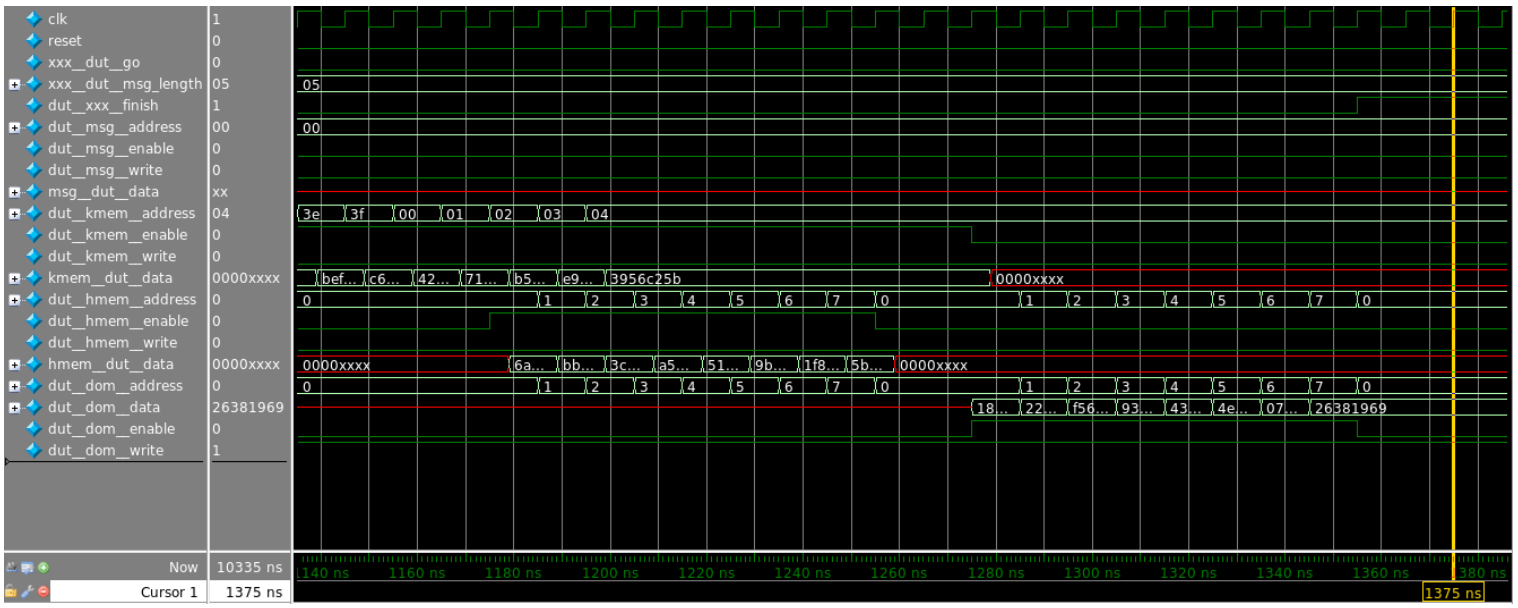


Figure 7 - Top Level Module Signal Waveform V

Results Achieved

The synthesizable Verilog RTL was compiled using Synopsys 2015 Design Compiler and used 45nm Nandgate OpenCell Library along with some IP from the Designware library (for blocks such as parallel prefix adders and multiplexers).

The final **area** of MyDesign.v is **15852.5362 um^2** (Combinational Logic: um^2, Non-Combinational Logic: um^2) at a clock period of 8 ns (125 MHz).

- Design Compiler version: Synopsys 2015
- Medium compile effort
- Minimize area while meeting timing requirements
- Clock Skew: 0.05 ns
- The inputs are driven by a DFF: T_{cq}: 0.2 ns, IP Delay: 0.04 ns
- The outputs drive four DFF: T_{su}: 0.25 ns, OP Delay: 0.45 ns
- **Setup Violation Check:** 0.0007 ns (Slack **MET**)
- **Hold Violation Check:** 0.0188 ns (Slack **MET**)
- **Hold fixed Setup Violation Check:** 0.0009 ns (Slack **MET**)

Message Length	# Clock Cycles	Clock Period (ns)	Area (um^2)	Performance (# Clock Cycles * Area * Clock Period, unit um^2 ns)
1	100	8	15852.5362	12682028.96
5	104	8	15852.5362	13189310.12
27	126	8	15852.5362	15979356.49
55	154	8	15852.5362	19530324.59
Average	**121**	**8**	**15852.5362**	**15345255.04**

Conclusion

The project goal was successfully achieved and the resulting synthesizable Verilog RTL was verified with various messages and the hash values produced were compared with a high level language implementation of the module in python. The RTL was synthesized using Synopsys 2015 Design Compiler and was able to achieve a clock period of 8 ns with a cell area of 15852.5362 um^2. The design compiler used parallel prefix adders from the Designware library and 45 nm standard cells from Nandgate OpenCell library. Upon inspection of the synthesized design it was found that the critical path in the design is the hash computation iteration which consists of several chained adders and other logic gates. It is difficult to pipeline the H computation operations since the next computation is dependent on the result of the current iteration.

The secure hashing algorithm (SHA) is used for many applications like authentication, message signing, data integrity check and cryptography. Due to the nature of operations in the algorithm an application specific hardware for SHA computation benefits from the performance boost offered by the intrinsic parallelism of the hardware compared to the execution of the algorithm in a general purpose CPU. Its widely used in many mainstream applications like SSH, TLS etc. hence many modern processors usually have some hardware implementation for SHA operations for e.g. Many of the Intel CPUs have a dedicated cryptoprocessor core for SHA computation and with the Internet of Things gaining popularity among the masses security is becoming paramount hence low power hardware implementations of these cryptographic algorithms are important.