The Design Document for CSEE 4840 Embedded System Design

Bitcoin Miner

GoldMiners:
Jules Comte
Zhe Mo
Tianyu Qin
Mingyang Song
Xueji Zhao

1 Introduction

In this project, we aim to develop a Bitcoin miner based on the DE1-SoC board. The SHA256 hashing algorithm which is used in Bitcoin mining will be performed based on the FPGA. The ARM-based Hard Processor System (HPS) will be responsible for the communication between the Bitcoin miner and the Bitcoin pool by implementing and running the Stratum v1 protocol.

The Stratum protocol is a popular mining protocol used in the Bitcoin network, and it allows the miner to communicate with the mining pool, receive work units and submit solutions. The HPS will be responsible for handling the network communication, parsing the received data and sending it to the FPGA for processing.

The FPGA implementation of the SHA256 algorithm is highly efficient and can perform faster calculations than a traditional CPU. By using an FPGA, the hash rate and the possibility of successfully mining a block can be potentially improved.

2 System Block Diagram and Algorithms

- **Block Diagram of Overall Mining Process**

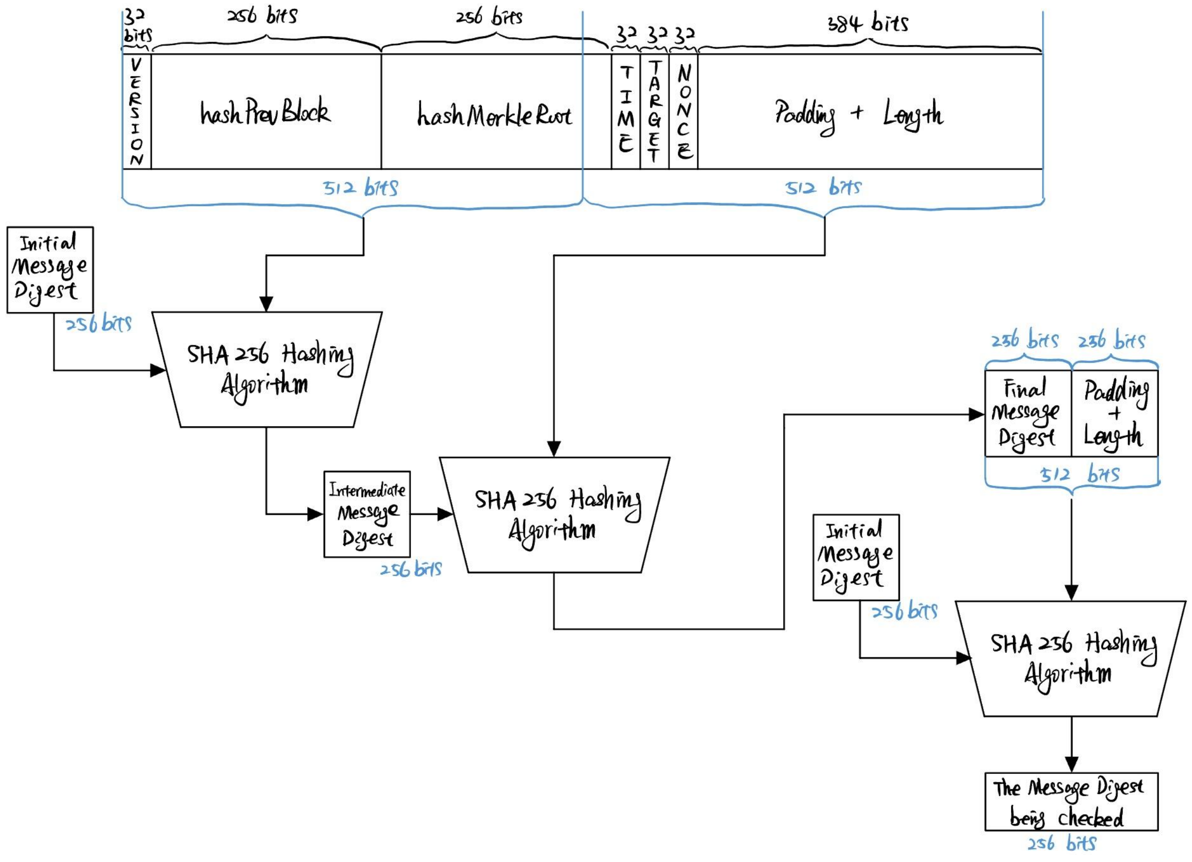


Figure 1

- **Zoom in on the SHA256 Algorithm Module**

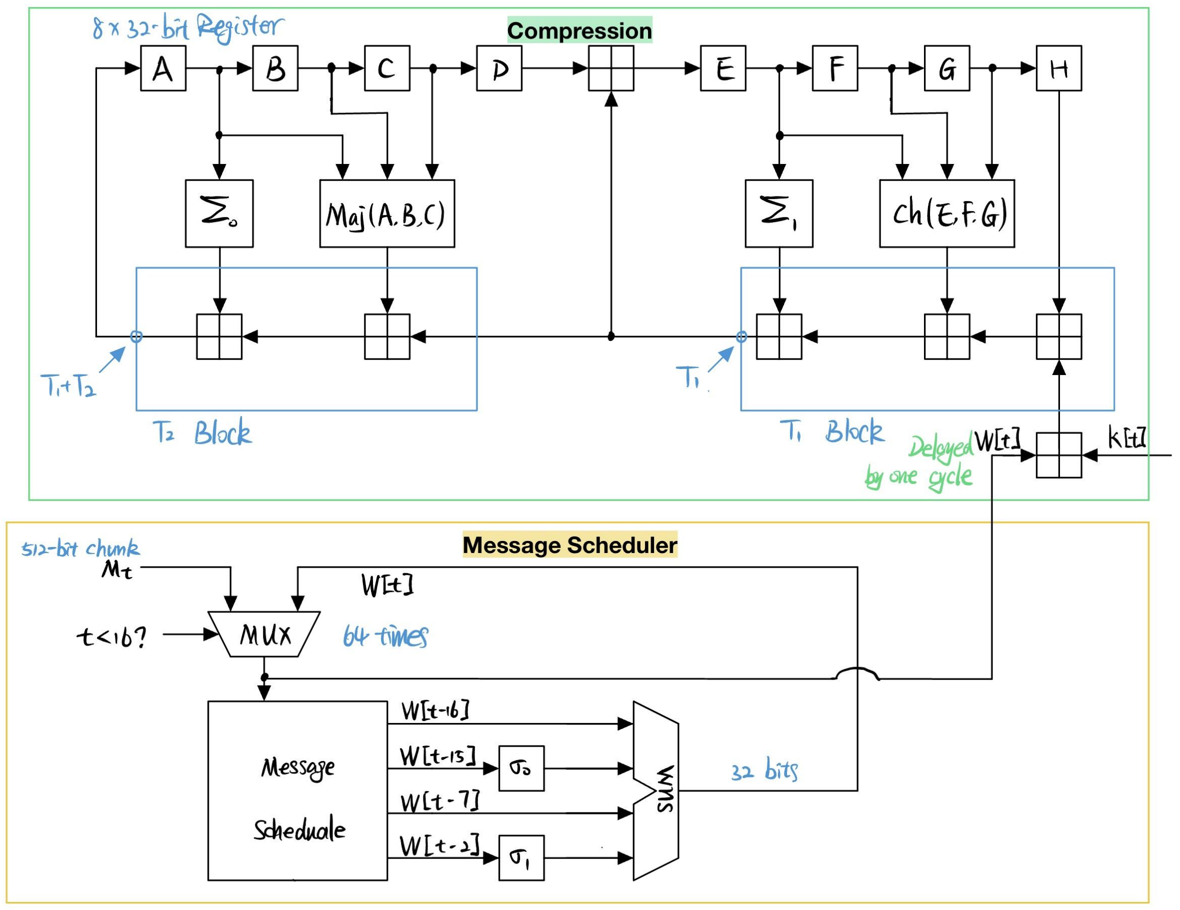


Figure 2

- **Register Saving SHA256 Algorithm Module**

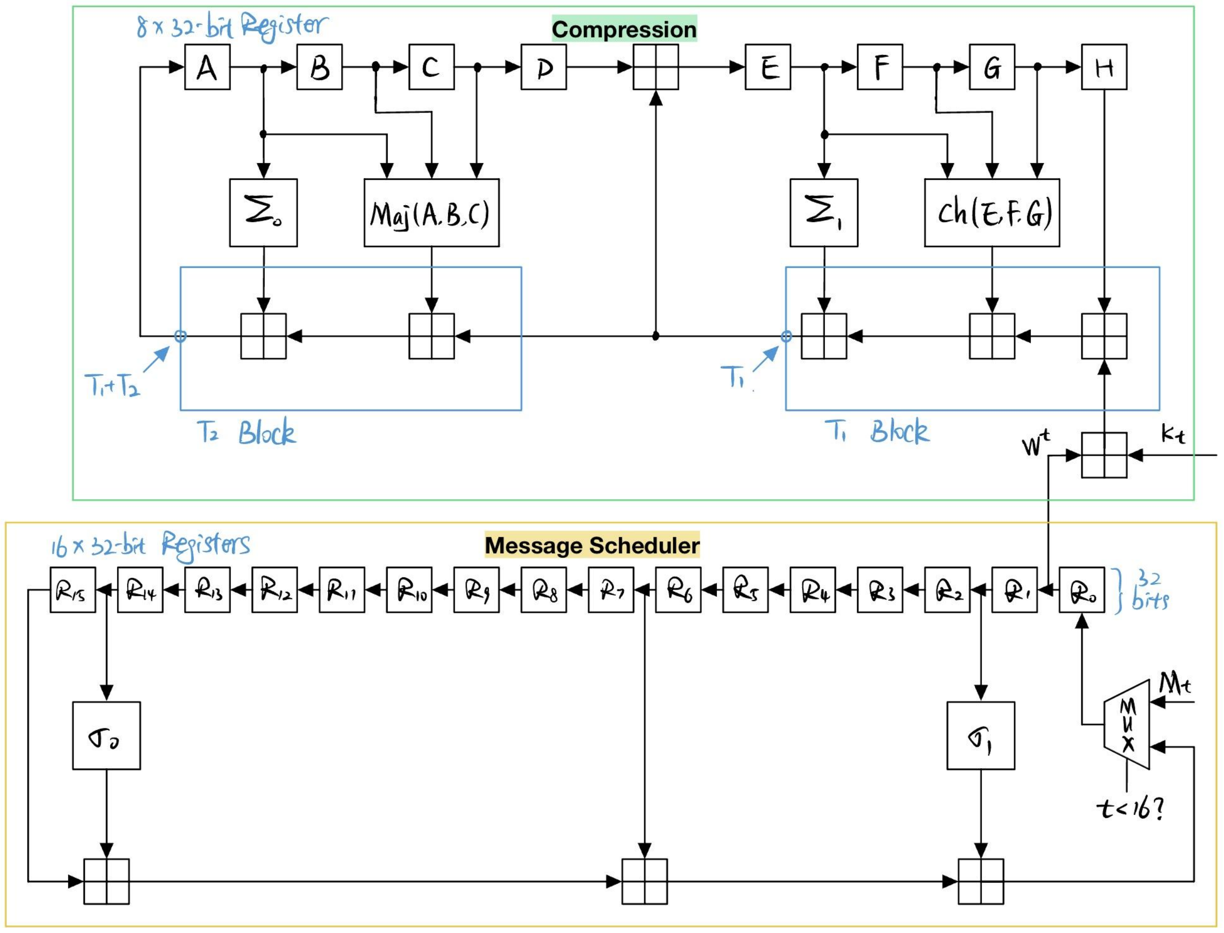


Figure 3

3 Algorithms

- Overall Mining Process

Bitcoin mining uses the SHA256 Algorithm. As shown in Figure 1, we implement three SHA256 Algorithm modules. To hash the 1024-bit input, we implement the SHA256 Algorithm module twice and get a 256-bit output hash value. This hash value is padded and fed into the input of the last SHA256 Algorithm module to get the final hash value. Thus, this process is called **double SHA256 hash** (SHA256(SHA256(Block Header))). Once getting the final hash value, we compare it with the expectation, and then get the award (unlikely :)) or do the double SHA256 hash again for a different nonce.

- SHA256 Algorithm

Fundamentally, the SHA256 Algorithm has two parts: **Message scheduler** and **Compression**. In the Message scheduler, we parse the 512-bit input into the first 16 32-bit Words (W[0] to W[15]) and use them to generate the rest 48 32-bit Words. In the Compression, with the 64 32-bit Words, we calculate the hash value composed of 8 32-bit Message Digests: A, B, C, D, E, F, G, and H. Ultimately, we add the new Message Digests to the initial Message Digests (H0, H1, H2, H3, H4, H5, H6, and H7) to get the final hash value.

To be more efficient, we "pipeline" these two parts as shown in Figure 2. Because the calculation of Message Digests only depends on the current Word, we can calculate the Message Digests once getting the current Word instead of waiting for all 64 Words to be done. This operation can save roughly 64 clock cycles in one round of the SHA256 Algorithm.

Furthermore, since the distance between the current Word and the previous Words needed is fixed (at most 16), we can only use 16 registers to save necessary Words instead of all 64, as shown in Figure 3. Therefore, we can release 48 32-bit registers.

- **Functions included in Figure 3**

```
function logic [31:0] rotL(input [31:0] a, input [5:0] b);
   rotL = (a << b) | (a >> (32 - b));
endfunction
function logic [31:0] rotR(input [31:0] a, input [5:0] b);
   rotR = (a >> b) | (a << (32 - b));
endfunction
module ch (input [31:0] x,y,z, output [31:0] ch);
   assign ch = (x & y) ^ ((~x) & z);
endmodule
module maj (input [31:0] x,y,z, output [31:0] maj);
   assign maj = (z & y) ^ (x & z) ^ (y & z);
endmodule
module ep0 (input [31:0] x, output [31:0] ep0);
   assign ep0 = rotR(x,2) ^ rotR(x,13) ^ rotR(x,22);
endmodule
module ep1 (input [31:0] x, output [31:0] ep1);
   assign ep1 = rotR(x,6) ^ rotR(x,11) ^ rotR(x,25);
endmodule
module sig0(input [31:0] x, output [31:0] sig0);
   assign sig0 = rotR(x,7) ^ rotR(x,18) ^ ({3{1'b0},x[31:3]});
endmodule
module sig1(input [31:0] x, output [31:0] sig1);
   assign sig1 = rotR(x,17) ^ rotR(x,19) ^ ({10{1'b0},x[31:10]});
endmodule
```

- **Stratum V1 Protocol**

  The Stratum allows miners to communicate with the mining pool and receive work units to solve.

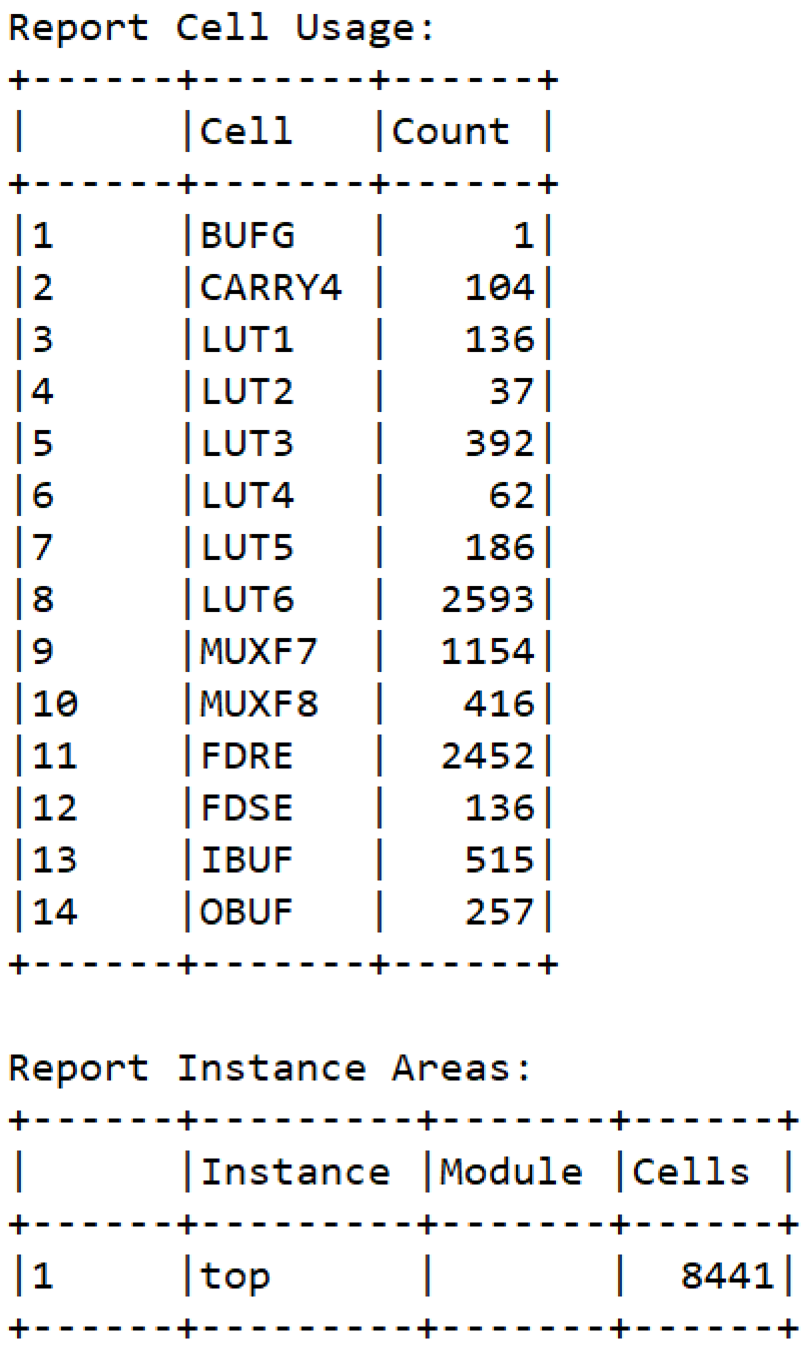
  This protocol defines 5 different messages (mining.subscribe, mining.authorize, mining.notify, mining.submit, mining.set_difficulty) to complete the mining collaboration process between the mining pool and the miners, as follows:

  First, the miner sends a subscription request to the mining pool. The mining pool replies with a unique identifier, which is labeled as "exnonce1". This identifier must be included in all mining activities of the miner in order to differentiate it from other miners. The reply from the mining pool also includes "exnonce2_size", which specifies the length limitation of "exnonce2". When the miner uses "exnonce2" later on, the length is restricted and cannot be arbitrarily changed.

  After that, the miner needs to be authorized. This is because some mining pools adopt a registration-based mining method, which requires users to register their username and password (while most mining pools use anonymous mining). Therefore, the miner needs to send its wallet address and machine ID to the mining pool.

  Once the miner is authorized, the mining pool will issue a task difficulty to the miner. In subsequent mining activities, the miner must achieve the task difficulty before getting rewarded. The mining pool will continually issue tasks to the miner. When the miner receives a task, it can determine whether to continue mining the previous task or to immediately start mining the new task based on the task identifier. The miner continuously performs these calculations, incrementing the nonce value in the block header until it finds a solution that meets the target difficulty. Once a solution is found, the miner sends the solution to the pool.

4 Resource Budgets



```
Report Cell Usage:
+------+-------+------+
|      |Cell   |Count |
+------+-------+------+
|1     |BUFG   |     1|
|2     |CARRY4 |   104|
|3     |LUT1   |   136|
|4     |LUT2   |    37|
|5     |LUT3   |   392|
|6     |LUT4   |    62|
|7     |LUT5   |   186|
|8     |LUT6   |  2593|
|9     |MUXF7  |  1154|
|10    |MUXF8  |   416|
|11    |FDRE   |  2452|
|12    |FDSE   |   136|
|13    |IBUF   |   515|
|14    |OBUF   |   257|
+------+-------+------+

Report Instance Areas:
+------+---------+-------+------+
|      |Instance |Module |Cells |
+------+---------+-------+------+
|1     |top      |       |  8441|
+------+---------+-------+------+
1. Slice Logic
--------------

+-------------------------+------+-------+-----------+-------+
|        Site Type        | Used | Fixed | Available | Util% |
+-------------------------+------+-------+-----------+-------+
| Slice LUTs*             | 3188 |     0 |    134600 |  2.37 |
|   LUT as Logic          | 3188 |     0 |    134600 |  2.37 |
|   LUT as Memory         |    0 |     0 |     46200 |  0.00 |
| Slice Registers         | 2588 |     0 |    269200 |  0.96 |
|   Register as Flip Flop | 2588 |     0 |    269200 |  0.96 |
|   Register as Latch     |    0 |     0 |    269200 |  0.00 |
| F7 Muxes                | 1154 |     0 |     67300 |  1.71 |
| F8 Muxes                |  416 |     0 |     33650 |  1.24 |
+-------------------------+------+-------+-----------+-------+
```

5 Hardware Software Interface

We propose for the sha256 accelerator module to have the following interface:

```
module sha256(

      input logic         clk,            // Clock

      input logic         reset,

      input logic         go,             // Start sha256 round

      input logic  [31:0] writedata,

      input logic         write,

      input               chipselect,

      input logic  [3:0]  address,

      output logic [31:0] h0,             // h0

      output logic [31:0] h1,             // h1

      output logic [31:0] h2,             // h2

      output logic [31:0] h3,             // h3

      output logic [31:0] h4,             // h4

      output logic [31:0] h5,             // h5

      output logic [31:0] h6,             // h6

      output logic [31:0] h7,             // h7

      output logic      done);            // 1 when module done
```

Overview of the interface:

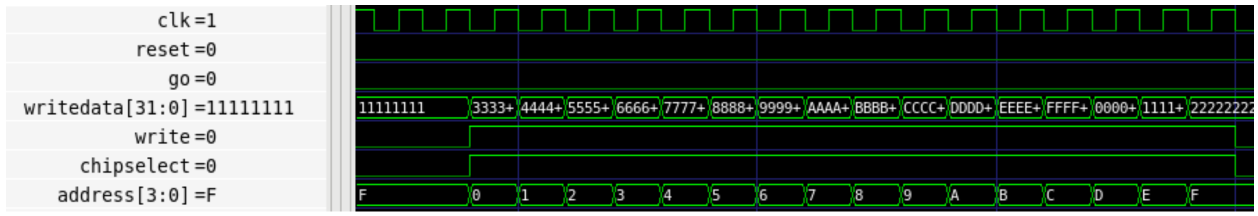
1. **clk**: Provides a clock signal to the hardware.
2. **reset:** Resets the hardware to its initial state.
3. **go:** Tells the hardware to start a round of sha256.
4. **writedata:** 4 bytes-wide, allows the software to send input data to the hardware.

5. **write:** Must be set to high when the software wants to write data to the hardware.
6. **chipselect:** Must be set to high when the software wants to write to the hardware.
7. **address:** Allows the software to send data to the 16 memory locations of the hardware.
8. **h0-h7:** Output registers of the hardware, they contain the message digest when the sha256 round is done.
9. **done:** Goes low-to-high when the hardware is done with a round of sha256.

How to send input data to the hardware:

- The accelerator takes in a fixed-width input of 64 bytes.
- To do so, the software must write the 64 bytes in 16 chunks of 4 bytes to the 16 memory locations of the accelerator, in a little-endian manner where the most significant chunk of the input goes to address 0.
- To do so, the software must raise the write and chipselect signals, and place the input data on the writedata register.

See below for an example write of a 64-byte chunk, for 1 round of the sha256 algorithm:

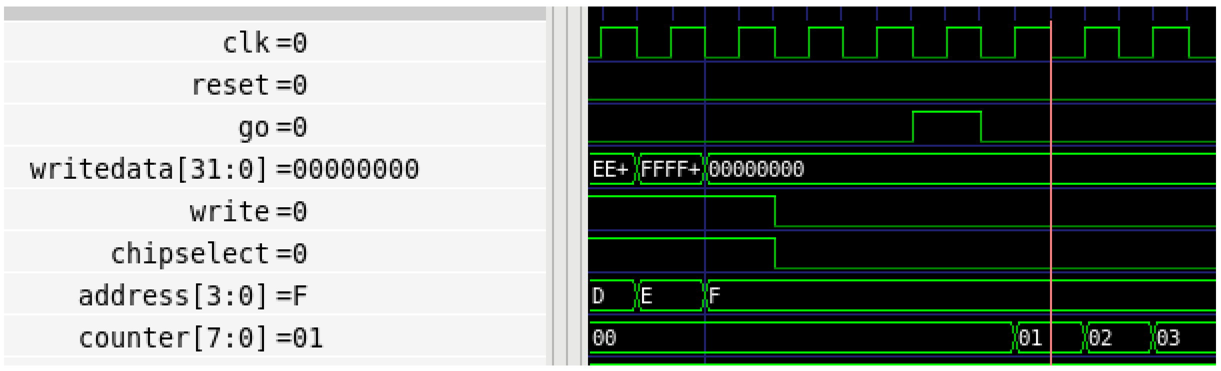


How to launch the SHA256round:

- When the software has written the input at the appropriate addressed memory locations, the software must raise the go signal for one cycle. The hardware will read the input registers, and begin a round of sha256.

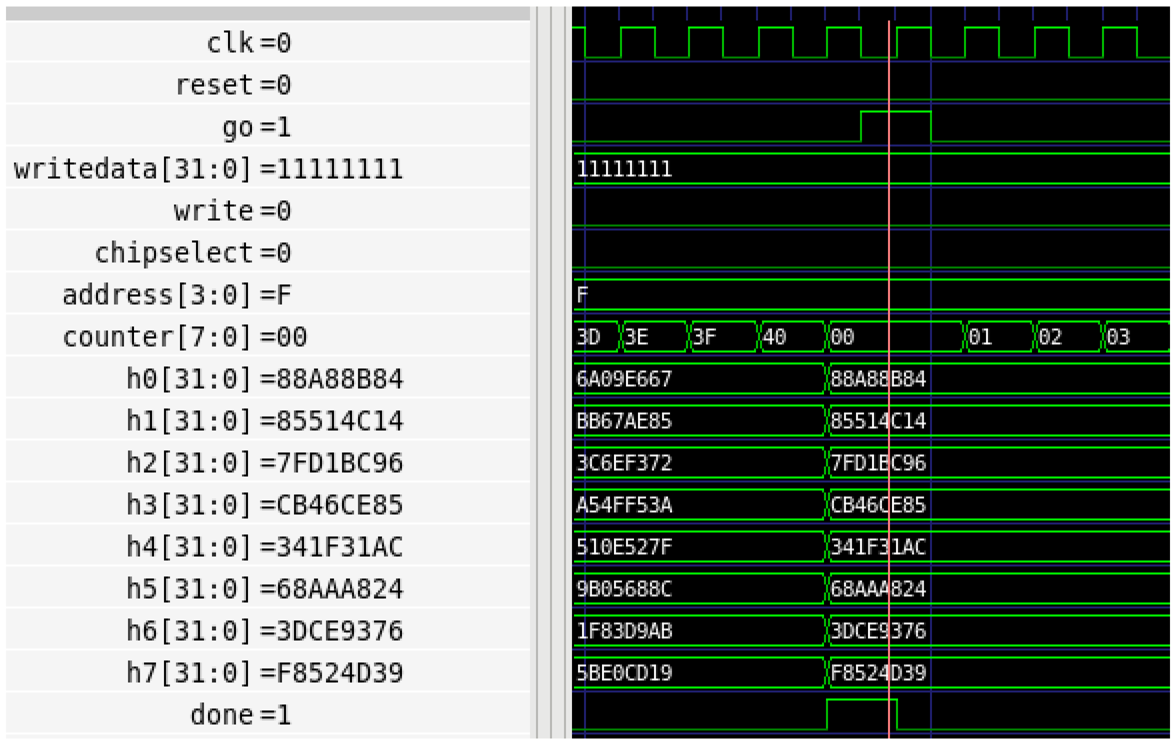
- When the done signal goes high after a round of sha256 is complete, the go signal can be raised high before the next positive edge of the clock to start the following round of sha256 for the next 64 byte chunk.

See below for an example of a launch of the first round of sha256. The left part of the waveform shows the final part of the write of the input data. A few cycles later, the go signal is raised for one clock cycle, and the accelerator begins working, as shown by the increment of the counter register at the very bottom.



How to know when the hardware is done with the current round, and launch the next round as required:

See below an example of a launch of a round of sha256 following the completion of a previous round. While the hardware is working on the previous round, it is best practice to use the write and chipselect signals to input the data for the following round.

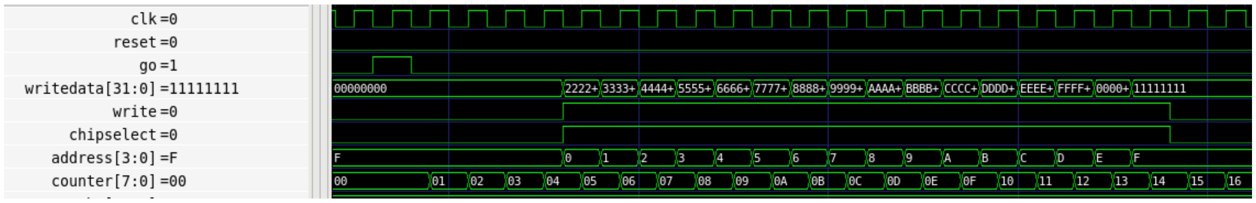


Notice the following:

- When the hardware is done with a sha256 round, it raises the done signal for one clock cycle. This also means that the h0-h7 registers now contain the message digest, which can be read from if the software determines that no more rounds are needed.
- In case the software requires more rounds, it raises the go signal for one clock cycle, and the hardware immediately proceeds with working on the following round, as can be seen by the counter register in the picture.

How to write the next chunk of 64 bytes while the hardware is working on the current 64 byte chunk:

See the picture below for an example of how data can be written to the input registers of the hardware while it is working on a round of sha256.



We can see that we first launch a round of sha256 by raising the go signal, and the counter starts running. A few cycles later, we raise the write and chipselect signals, and start writing new input data to the hardware. The hardware will not read this new input data until it is done with the current sha256 round, and we raise the go signal once again for it to begin a new round of sha256, as we described in the previous section.