SHA256 is an algorithm that takes an input message bit string that has a maximum size of 2^64 bits and transforms it into an output value digest bit string that has a fixed length of 256 bit. In other words, it turns data that comes from things like a message or a file, and turns it into some other data in a seemingly random way.

SHA256 is used in a wide range of applications that require security against hackers that can reverse engineer systems. For example, it can be used to securely transfer files and prevent tampering – any tampering to the file, even the slightest bit change, would significantly alter the digest which is a feature. Moreover, although there are multiple input strings that map to the same 256 bit string output, it is infeasible to brute force such solutions.

In the context of Bitcoin, SHA256 is used to generate a digest that conditionally links the blocks together in a manner such that the entire data of a block is converted to a hash signature that is present in the proceeding block. As discussed, any tampering to one block (changing of a transaction) would break the entire block because of the starkly changed hash signature, causing a mismatch to the next block, breaking the chain and invalidating that block.

SHA256 Implementation

The implementation of the SHA256 algorithm is a finite state machine with states: idle, read, read wait, block, compute, write, and write wait.

Before the actual algorithm is actually ran, we need to read from memory the message that is to be hashed. In this case, the message will be fixed to 20 words – since a word is 32 bits, this is a total of 640 bits to be read.

The FSM first enters IDLE state, where we initialize variables such as the given start hash values, indexing variables, and variables for memory reading – we enable reading of the memory by setting cur\_we to 0, and also set the starting offset to 0 and cur\_addr to message\_addr to begin reading at the start location of the message. Offset is in units of words, thus by adding 1 to it, we are advancing by 32 bits in terms of bits in memory. Since there are 20 words of message, we will want to increment offset and read until offset is 19. This is what happens in the READ state.

READ state increments the offset, and reads the returned word length chunk from memory. We store what’s read in the message variable that will eventually become the whole message. It is worth noting that since memory reads and writes take 1 cycle to process, we have a WRITE\_WAIT state that acts as a 1-cycle buffer for this action. Hence, after incrementing the offset, we move on to the READ\_WAIT state, that will inevitably point back to the READ state, where mem\_read\_data, the returned word, is stored.

Once all 20 words are read and stored into the message variable, we move on to the BLOCK variable where the actual SHA256 algorithm occurs.

The w variable stores 20 words, representing a message block. The algorithm can only process 512 bits at a time. The BLOCK state checks if we are currently on the last block or not. If the former is true, then we pad the block with a 1, followed by 0’s, until there are 64 bits left to hold the message length in binary.

For each message block stored in w, the COMPUTE state occurs 64 times over 64 clock cycles. This is where the rounds of compression happen that use the 64 32-bit constants in k, the current eight 32-bit hashes, and the message block words to compute eight new 32-bit additions (variables a through h) to be added to the current hashes.

For a cycle of the COMPUTE state, the core SHA256 operation is called as sha256\_op(). This constitutes right rotations, XORs, and additions of the ith k-constant, and message block word. Note that since i goes up to 63, we can not index the message block word above the 15th. The SHA256 algorithm originally does preprocessing of the 16 word message block and expands it into 64 words. However, the core algorithm for word expansion to generate the ith word above 15 relies on a window of size 16 before the ith word. Phrased differently, this means we can keep a word message variable w to a fixed length of 16 for reduced use of registers.

To avoid using a w length of 64, the w variable every cycle of COMPUTE is left shifted, and the new w, computed with the wtnew() function, is inserted from the right. Thus, sha256\_op can take in the ith message word, up to the 64th one, by always indexing the first word at w[0].

Once 64 rounds of compression is done, the state moves to the BLOCK state to check whether the entire message has been processed – if not, then repeat the same compression to the next 16-word message block. Else, we move to the WRITE state.

The WRITE state has a complementary WRITE\_WAIT state for the same reasons READ has READ\_WAIT – writing data to memory takes 1 clock cycle for the data to actually be written, and for a new piece of data to be loaded in to the writer.

The final values of the eight 32-bit hashes (h0 to h7) represent the message digest. They are to be written word by word to the memory, starting at the address specified in output\_addr. We can do this by first setting the start address in cur\_address as output\_address, enabling the write enable by setting cur\_we to high, and transitioning to the WRITE\_WAIT state which will go back to the WRITE state once the data has actually been written. Now we can increment the offset to repeat the writing process for the next hash.

We do this until the offset is 7, meaning that all eight hashes have been written to memory. We finally move to the IDLE\_STATE where the done flag is also set high to signify the input message has been digested by SHA256.

A picture containing timeline

Description automatically generated

The wave form shows the SHA256 module going through the explained states to digest the 20-word length input message. Once the hashes have been written at around 3.9 ns, the done flag is raised to high, and the simulation stops.

Text

Description automatically generated with medium confidence

Graphical user interface

Description automatically generated with medium confidence

Table

Description automatically generated

Table

Description automatically generated with medium confidence