ESE 461: Design Automation for Integrated Circuit Systems

Final Project

Option 2: Bitcoin Hashing

---

1. Bitcoin Introduction

Bitcoin is a novel cryptocurrency that was introduced in 2008[1] and has quickly gained tremendous popularity over the past few years. One key feature of Bitcoin is that instead of relying on any trusted central authority, it is maintained purely by a peer-to-peer (P2P) network of pseudo-anonymous users via a fully-distributed public ledger, known as the blockchain. The blockchain is a linked list of hash tree data structures that records every Bitcoin transaction ever existed, and it is updated by the network of P2P communicating nodes that continuously verify and record Bitcoin transactions. In order to add legitimate extensions to the blockchain, the node has to solve an extremely difficult math puzzle, and this mechanism is known as proof-of-work. The virtuous cycle of cryptography-based transaction and proof-of-work, distributed consensus, and the financial incentives of mining is at the heart of the success of Bitcoin.

2. Bitcoin Hash Algorithm [2][3]

Bitcoin mining uses the hashcash proof of work function. The hashcash algorithm requires the following parameters: a service string, a nonce, and a counter. In bitcoin the service string is encoded in the block header data structure, and includes a version field, the hash of the previous block, the root hash of the merkle tree of all transactions in the block, the current time, and the difficulty. Bitcoin stores the nonce in the extraNonce field which is part of the coinbase transaction. Also stores in the extraNounce field is the left most leaf node in the merkle tree (the coinbase is the special first transaction in the block).

The basics of the hashcash algorithm are described in more detail here. When mining bitcoin, the hashcash algorithm repeatedly hashes the block header while incrementing the counter & extraNonce fields. Incrementing the extraNonce field entails recomputing the merkle tree, as the coinbase transaction is the left most leaf node. The block is also occasionally updated as you are working on it.

A block header contains these fields:

Field	Purpose	Updated when...	Size (Bytes)
Version	Block version number	You upgrade the software and it	4

		specifies a new version	
hashPrevBlock	256-bit hash of the previous block header	A new block comes in	32
hashMerkleRoot	256-bit hash based on all of the transactions in the block	A transaction is accepted	32
Time	Current timestamp as seconds since 1970-01-01T00:00 UTC	Every few seconds	4
Bits	Current target in compact format	The difficulty is adjusted	4
Nonce	32-bit number (starts at 0)	A hash is tried (increments)	4

Most of these fields will be the same for all users. There might be some minor variation in the timestamps. The nonce will usually be different, but it increases in a strictly linear way. "Nonce" starts at 0 and is incremented for each hash. Whenever Nonce overflows (which it does frequently), the extraNonce portion of the generation transaction is incremented, which changes the Merkle root.

Moreover, it is extremely unlikely for two people to have the same Merkle root because the first transaction in your block is a generation "sent" to one of your unique Bitcoin addresses. Since your block is different from everyone else's blocks, you are (nearly) guaranteed to produce different hashes. Every hash you calculate has the same chance of winning as every other hash calculated by the network.

Bitcoin uses: SHA256(SHA256(Block_Header)) but you have to be careful about byte-order.

For example, this python code will calculate the hash of the block with the smallest hash as of June 2011, Block 125552. The header is built from the six fields described above, concatenated together as little-endian values in hex notation:

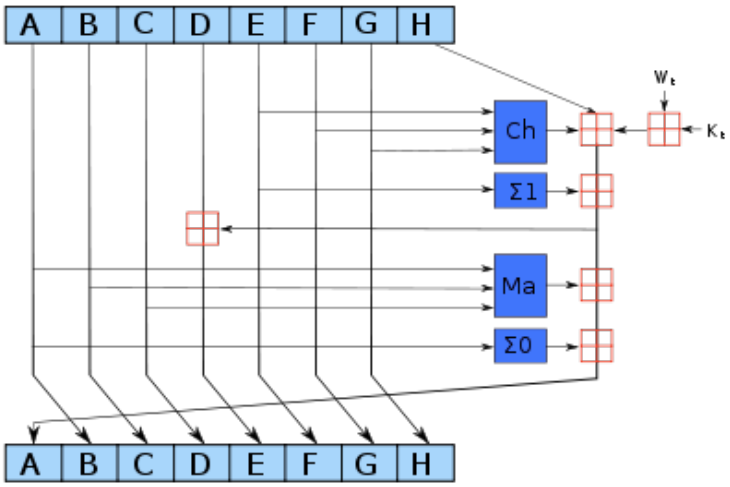
```
>>> import hashlib
>>> header_hex = ("01000000" +
 "81cd02ab7e569e8bcd9317e2fe99f2de44d49ab2b8851ba4a308000000000000" +
 "e320b6c2fffc8d750423db8b1eb942ae710e951ed797f7affc8892b0f1fc122b" +
 "c7f5d74d" +
 "f2b9441a" +
 "42a14695")
>>> header_bin = header_hex.decode('hex')
>>> hash = hashlib.sha256(hashlib.sha256(header_bin).digest()).digest()
```

```
>>> hash.encode('hex_codec')
'1dbd981fe6985776b644b173a4d0385ddc1aa2a829688d1e0000000000000000'
>>> hash[::-1].encode('hex_codec')
'0000000000000000e1d8d6829a8a21adc5d38d0a473b144b6765798e61f98bd1d'
```

3. SHA-256 Algorithm [4][5][6]

The SHA (Secure Hash Algorithm) is one of a number of cryptographic hash functions designed by the National Security Agency (NSA). A cryptographic hash is like a signature for a text or a data file. SHA-256 algorithm generates an almost-unique, fixed size 256-bit (32-byte) hash. Hash is a one way function – it cannot be decrypted back. This makes it suitable for password validation, challenge hash authentication, anti-tamper, digital signatures.

Simply put, here is how SHA-256 works: after the initialization step, which may include padding if the message does not fit the standard size, the main loop of SHA-256 consists of 64 iterations and each iteration performs the computation illustrated in the figure below. In this way, the original message is compressed into a 256-bit value, known as the hash of the original message.



The detailed calculation of SHA-256 is presented in reference [3]. And you can find sample code here.

3. Project Requirement

For this project, we will focus on implementing the basic bitcoin block hashing algorithm, which equals to double SHA-256, i.e. performing the SHA-256 algorithm twice.

In the testbench you create, please fix all fields except the *Nonce*(The orange part in the above table) and iterate through the *Nonce* values until you find the first *Nonce* that allows the hashed block header to have at least 1 byte (8-bit) of leading 0s. You should do this Nonce finding for two different test block headers as follows (C code):

```
BYTE bitcoin_test_block1[] = {

0x01, 0x00, 0x00, 0x00,// ver

0x81, 0xcd, 0x02, 0xab, 0x7e, 0x56, 0x9e, 0x8b, 0xcd, 0x93, 0x17, 0xe2,
0xfe, 0x99, 0xf2, 0xde, 0x44, 0xd4, 0x9a, 0xb2, 0xb8, 0x85, 0x1b, 0xa4,
0xa3, 0x08, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,// hashPrevBlock

0xe3, 0x20, 0xb6, 0xc2, 0xff, 0xfc, 0x8d, 0x75, 0x04, 0x23, 0xdb, 0x8b,
0x1e, 0xb9, 0x42, 0xae, 0x71, 0x0e, 0x95, 0x1e, 0xd7, 0x97, 0xf7, 0xaf,
0xfc, 0x88, 0x92, 0xb0, 0xf1, 0xfc, 0x12, 0x2b,// hashMerkleRoot

0xc7, 0xf5, 0xd7, 0x4d,// Time

0xf2, 0xb9, 0x44, 0x1a,// Bits (Target)

0x00, 0x00, 0x00, 0x00// Nonce(You need find the proper value of Nonce)

};

BYTE bitcoin_test_block2[] = {

0x02, 0x00, 0x00, 0x00,// ver

0x7e, 0xf0, 0x55, 0xe1, 0x67, 0x4d, 0x2e, 0x65, 0x51, 0xdb, 0xa4, 0x1c,
0xd2, 0x14, 0xde, 0xbb, 0xee, 0x34, 0xae, 0xb5, 0x44, 0xc7, 0xec, 0x67,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,// hashPrevBlock

0xd3, 0x99, 0x89, 0x63, 0xf8, 0x0c, 0x5b, 0xab, 0x43, 0xfe, 0x8c, 0x26,
0x22, 0x8e, 0x98, 0xd0, 0x30, 0xed, 0xf4, 0xdc, 0xbe, 0x48, 0xa6, 0x66,
0xf5, 0xc3, 0x9e, 0x2d, 0x7a, 0x88, 0x5c, 0x91,// hashMerkleRoot

0x02, 0xc8, 0x6d, 0x53,// Time

0x6c, 0x89, 0x00, 0x19,// Bits (Target)

0x00, 0x00, 0x00, 0x00// Nonce(You need find the proper value of Nonce)

};
```

4. Design Specifications

Your custom-designed bitcoin hashing accelerator should be able to correctly perform the bitcoin hashing algorithm, and achieve the minimum performance listed below using the 180nm standard cell library we provide:

Clock frequency: higher than 80MHz

Power consumption: lower than 100 mW

Total chip area: lower than 4.2 mm^2

Reference and useful links

[1] Nakamoto, Satoshi. "Bitcoin: A peer-to-peer electronic cash system." (2008).

[2] https://en.bitcoin.it/wiki/Block_hashing_algorithm

[3] http://www.righto.com/2014/02/bitcoin-mining-hard-way-algorithms.html

[4] http://www.iwar.org.uk/comsec/resources/cipher/sha256-384-512.pdf

[5] https://en.wikipedia.org/wiki/SHA-2

[6] http://classes.engineering.wustl.edu/ese461/FinalProject/Bitcoin/BitcoinCCodeSample.zip