BITCOIN AND BLOCKCHAIN TECHNOLOGY WITH FOCUS ON FPGA MINERS

MOHAMMAD NIKNAM

2021-2022

Abstract:

Bitcoin is decentralized payment system and the most famous cryptocurrency which has a peer-to-peer network that operates on a cryptographic protocol. Bitcoin does not rely on any centralized entity like banks or governments to ensure the system reliability. Users send and receive bitcoins (the units of currency), directly from person to person, by broadcasting digitally signed messages to the network. Transactions are recorded into a distributed, replicated public database known as the blockchain, with consensus achieved by a proof-of-work system called mining, where miners are rearward for computing power spent to support the network. In this paper we discuss the bitcoin technology, process of Bitcoin mining, and mining hardware with focus on FPGA miners.

Research Aim:

The main objectives of this project are as follows.

- Learn about the bitcoin network and payment system
- Study cryptography basics, Cryptographic Hash Functions properties, data structures based on cryptography (Blockchain), and digital signature.
- Analyze bitcoin mining algorithm and SHA-256
- Study mining concept and compare the advantages of implementing bitcoin mining in different hardware.
- Learn about what FPGA is, describe the architecture of it, and how is the usage and applications of it in bitcoin mining compare to other hardware.

Keywords:

Cryptocurrency, Bitcoin, FPGA, Cryptography, Hash function, SHA256, Blockchain, Digital signature, Mining.

Table of Contents

1-INTRODUCTION

Bitcoin, "A Peer-to-Peer Electronic Cash System"[1], is unlike anything the world has seen before. By providing fast, inexpensive, international money transfer, it has the potential to revolutionize both the modern-day concept of money and commerce.

Bitcoin is a peer-to-peer[2] digital currency based on public key cryptography. It started as a free software project and a paper published by Satoshi Nakamoto[3] in 2009. Nakamoto, designed a system of online value transfer that supports a promising Internet currency. He introduced bitcoin as a version of electronic cash that would allow payments to be Sent from one party to another without going through a financial institution.

Traditionally, financial institutions, such as banks, are trusted to store and protect a customer's currency. The bank will handle the transfer of money between its customers and clients, but there are several disadvantages in this trust-based model. Electronic transfers between banks can be costly since there is usually a transaction fee, they can be slow taking several days to complete, and transfers cannot be made anonymously and completely non-reversible transactions are not really possible, since financial institutions cannot avoid mediating disputes. The cost of mediation increases transaction costs. But no mechanism exists to make payments over a communications channel without a trusted party.

What is needed is an electronic payment system based on cryptographic proof instead of trust. Bitcoin is a system of owning and transferring currency that omits these trusted third parties, allowing any two willing parties to transact directly with each other without the need for a trusted third party. And instead, Satoshi propose a solution to the double-spending[4] problem using a peer-to-peer distributed Timestamp server to generate computational proof of the chronological order of transactions.

Bitcoin is made possible by a combination of software and network technologies. A program called the Bitcoin client simultaneously manages and helps you spend bitcoins. This program maintains a long ledger called the blockchain that holds every transaction confirmed by the Bitcoin network.

The Bitcoin network, consisting of thousands of machines running the Bitcoin software, has two main tasks to accomplish. One is relaying transaction information and the second is verifying those transactions to ensure the same bitcoins cannot be spent twice.

The first task is accomplished easily due to the fact that the Bitcoin network is operated as a peer-to-peer network. After all, sharing data is easy. By operating on many nodes across the globe, the network ensures it will operate as long as it provides a useful service, and the system is secure as long as honest nodes collectively control more CPU power than any cooperating group of attacker nodes.

The second task is a bit more complicated and is solved through what I consider to be Bitcoin's key innovation. This development, a process called mining, is carried out by computers running mining software.

---

[1] This phrase is come from title of original bitcoin whitepaper by Nakamoto in 2009.

[2] P2P - Peer-to-peer refers to systems that work like an organized collective by allowing each individual to interact directly with the others. In the case of Bitcoin, the network is built in such a way that each user is broadcasting the transactions of other users. And, crucially, no bank is required as a third party.

[3] Satoshi Nakamoto – the creator of Bitcoin and the author of the original Bitcoin whitepaper and code. His real identity is unknown to the world.

[4] Double Spend - If a malicious user tries to spend their bitcoins with two different recipients at the same time, this is double spending. Bitcoin mining and the block chain are there to create a consensus on the network about which of the two transactions will confirm and be considered valid.

2-CRYPTOGRAPHY

Cryptography is a deep academic research field using many advanced mathematical techniques that are notoriously subtle and complicated. Fortunately, Bitcoin relies on only a handful of relatively simple and well-known cryptographic constructions. In this section, we specifically study cryptographic hashes and digital signatures, two primitives that prove to be useful for building cryptocurrencies.

2.1-Cryptographic Hash Functions

In this section, we've talked about hash functions, cryptographic hash functions with special properties, applications of those properties, and a specific hash function that we use in Bitcoin.
The first cryptographic primitive that we need to understand is a *cryptographic hash function*. A *hash function* is a mathematical function with the following three properties:

- Its input can be any string of any size.
- It produces a fixed-sized output. For the purpose of making the discussion in this chapter concrete, we will assume a 256-bit output size. However, our discussion holds true for any output size, as long as it is sufficiently large.
- It is efficiently computable. Intuitively this means that for a given input string, you can figure out what the output of the hash function is in a reasonable amount of time. More technically, computing the hash of an n-bit string should have a running time that is $O(n)$.

These properties define a general hash function, one that could be used to build a data structure, such as a hash table. We're going to focus exclusively on *cryptographic* hash functions. For a hash function to be cryptographically secure, we require that it has the following three additional properties: (1) collision resistance, (2) hiding, and (3) puzzle friendliness.

Property 1: Collision Resistance

The first property that we need from a cryptographic hash function is that it's collision-resistant. A collision occurs when two distinct inputs produce the same output. A hash function *H(.)* is collision-resistant if nobody can find a collision.
Formally:

> **Collision-resistance:** A hash function *H* is said to be collision resistant if it is infeasible to find two values, ***x*** and ***y***, such that [$x \neq y$], yet [***H(x) = H(y)***].

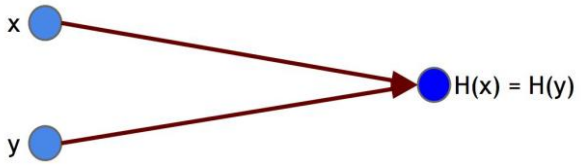


Figure 1 - A hash collision. x and y are distinct values, yet when input into hash function H, they produce the same output.

Notice that we said "nobody can find" a collision, but we did not say that no collisions exist. Actually, collisions exist for any hash function, and we can prove this by a simple counting argument. The input space to the hash function contains all strings of all lengths, yet the output space contains only strings of a specific fixed length. Because the input space is larger than the output space (indeed, the input space is infinite, while the output space is finite), there must be input strings that map to the same

output string. In fact, there will be some outputs to which an infinite number of possible inputs will map.

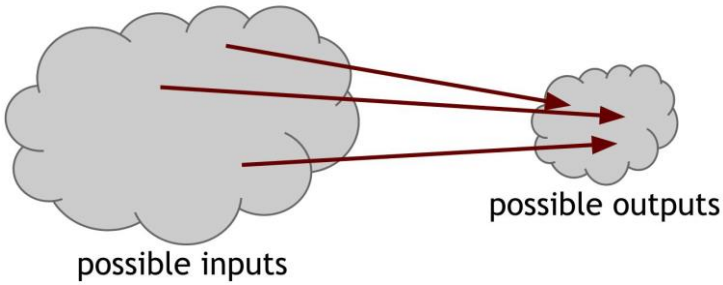


Figure 2 - Inevitability of collisions. Because the number of inputs exceeds the number of outputs, we are guaranteed that there must be at least one output to which the hash function maps more than one input.

Now, to make things even worse, we said that it has to be impossible to find a collision. Yet, there are methods that are guaranteed to find a collision. Consider the following simple method for finding a collision for a hash function with a 256-bit output size: pick $2^{256} + 1$ distinct values, compute the hashes of each of them, and check if there are any two outputs are equal. Since we picked more inputs than possible outputs, some pair of them must collide when you apply the hash function.[5]

The method above is guaranteed to find a collision. But if we pick random inputs and compute the hash values, we'll find a collision with high probability long before examining $2^{256} + 1$ inputs. In fact, if we randomly choose just $2^{130} + 1$ inputs, it turns out there's a 99.8% chance that at least two of them are going to collide.[6]

This collision-detection algorithm works for every hash function. But, of course, the problem is that it takes a very long time to do. For a hash function with a 256-bit output, you would have to compute the hash function $2^{256} + 1$ times in the worst case, and about 2^{128} times on average. That's of course an astronomically large number—if a computer calculates 10,000 hashes per second, it would take more than one octillion (10^{27}) years to calculate 2^{128} hashes! For another way of thinking about this, we can say that if every computer ever made by humanity had been computing since the beginning of the universe, the odds that they would have found a collision by now are still infinitesimally small. So small that it's far less than the odds that the Earth will be destroyed by a giant meteor in the next two seconds.

We have thus found a general but impractical algorithm to find a collision for *any* hash function. A more difficult question is: Is there some other method that could be used on a particular hash function to find a collision? In other words, although the generic collision detection algorithm is not feasible to use, there may be some other algorithm that can efficiently find a collision for a specific hash function. Yet for other hash functions, we don't know if such methods exist. We suspect that they are collision resistant. However, there are no hash functions *proven* to be collision-resistant. The cryptographic hash functions that we rely on in practice are just functions for which people have tried really, really hard to find collisions and haven't yet succeeded. In some cases, such as the old MD5 hash function, collisions were eventually found after years of work, leading the function to be deprecated and phased out of practical use. And so, we choose to believe that those are collision resistant.

---

[5] ***Brute-force attack*** *tries to find a collision by testing every possible input - by trial and error - until it finds two that match in hash results. in other words, find the original input that produced a known hash (called a preimage attack). It takes 2^n attempts for an n-bit hash. After that, in 2^n+1 attempt, collision is guaranteed.*

[6] *The fact that we can find a collision by only examining roughly the square root of the number of possible outputs results from a phenomenon in probability known as the birthday paradox. For further elaboration check Appendices.*

Application: Message digests

If we can assume that we have a hash function that is Collision Resistance, then we can use that hash function as message digest. That means if we know that x and y have the same hash, then it's safe to assume that x and y are the same. Because of Collision Resistance Property, since there's not a collision that we know of, then knowing the hashes are the same, we can assume that the values are the same. And this let us use the hash as a kind of message digest. Suppose, for example, that we had a file, a really big file. And we wanted to be able to recognize later whether another file was the same as the file we saw the first time, right? So, one way to do that would be to save the whole big file. And then when we saw another file later, just compare them. But because we have hashes that we believe are collision free, it's more efficient to just remember the hash of the original file. Then if someone shows us a new file, and claims that it's the same, we can compute the hash of that new file and compare the hashes. If the hashes are the same, then we conclude that the files must have been the same. And that gives us a very efficient way to remember things we've seen before and recognize them again. And, of course, this is useful because the hash is small, it's only 256 bits, while the original file might be really big. So, hash is useful as a message digest.

Property 2: Hiding - Preimage resistance (one wayness)

The second property that we want from our hash functions is that it's hiding. The hiding property asserts that if we're given the output of the hash function y = H(x), there's no feasible way to figure out what the input, x, was.

In cases where the number of possible input (X) values is limited and specified (in fact it's predictable), the adversary can hash the possible inputs one by one and compare them with the desired output. Thus, he finds the input X that results in the output Y = H (X). So, in such cases, our hash function practically does not hide the input.

In order to be able to achieve the hiding property, it needs to be the case that there's no value of *x* which is particularly likely. That is, *x* has to be chosen from a set that's, in some sense, very spread out. If *x* is chosen from such a set, this method of trying a few values of x that are especially likely will not work. Also, we want method that we can hide even an input that's not spread out.

We can now be slightly more precise about what we mean by hiding (the double vertical bar $\|$ denotes concatenation).

> **Hiding:** A hash function H is hiding if: when a secret value *r* is chosen from a probability distribution that has *high min-entropy,* then given H *(r* $\|$ *x)* it is infeasible to find *x.*

High min-entropy means that the distribution is very spread out, and it's not predictable, so that no particular value is chosen with more than negligible probability. for a concrete example, if r is chosen uniformly from among all of the strings that are 256 bits long, then any particular string was chosen with probability $1/2^{256}$, which is an infinitesimally small value. So, as long as r was chosen that way, then the hash of r concatenated with x is going to hide x. And that's the hiding property that the hash function will be deemed to have.

> **r:** A random and hidden value that is called "Key" or "Nonce" in cryptography. The term "Nonce" refers to the fact that its value is used only once. And Each time the hash function is called, in order to hide the input (X), it is combined with a new random value and then hashed.

Application

Let's look at the applications of "collision resistance" and "hiding" by an example. Suppose we write a message on a letter and put it in a sealed envelope. In this example, the message is like X, and the envelope plays the hash function role, and the usage of nonce, is likened to sealing. Suppose we put this message (X) in a sealed (combination of X and nonce) envelope (to hash) and put it on a table in

front of everyone. Now, no one but us, who have sealed it (we know the message and nonce), can find out the message (X value) by looking at the envelope (hash function output). So, the message remains a secret from everyone else, which states **"hiding"** property.
Also, obviously, after sealing the envelope, even if we change our mind, we cannot change the message already written in the envelope or replace it with another message (it is infeasible to find any two messages, one to replace by each other, which after combining with nonce, and get hashing, has brought us to the same output). Which states **"collision-resistance"** property.

Property 3: Puzzle Friendliness - Target Collision Resistance

This property is more complicated. If somebody expects a certain output (like Y) from the hash function and consider it as the purpose, and if a part of the input is chosen from satisfying randomness. Then, it is infeasible to find another input that results considered purpose (Y).

> **Puzzle friendliness:** A hash function *H* is said to be puzzle friendly if for every possible *n*-bit output value *y*, if *k* is chosen from a distribution with high min-entropy, then it is infeasible to find *x* such that $H(k \parallel x) = y$ in time significantly less than 2^n.

A good puzzle shouldn't have to be solved fast. And there is no shortcut to solve the puzzle, but looking for pieces one by one randomly, to find the right piece. As the same way, for an appropriate hash function, it should be time and energy consuming to check inputs to find the one which reaches the certain output. Also, no other shortcut strategy or algorithm can be found that is more efficient than randomly checking each input. As a result, in the situation that the number of inputs is very large, it is practically impossible to find the input X that results in a certain output Y.
About the relation between Property 1 (Collision Resistance) and Property 3 (Puzzle Friendliness), It can be said that: If the first property is hold by a hash function, then the third property is also true. Note that the third property is different from the first property. For the considered certain input and output (such as x and h (x)), to be able to find another input(x') whose hash is h (x') = h (x), is enough to break the third property. While the first property states that the hash function should be such that practically no random pairs of input and output can be found where the inputs are different but the function results the same output.
Consider a hash function with n-bit output (which can take any of 2^n values). Solving the puzzle requires finding an input so that the output falls within the set Y (can be selected and considered), which is typically much smaller than the set of all outputs. The size of Y determines how hard the puzzle is. If Y is the set of all n-bit strings (it means all outputs are acceptable) the puzzle is trivial, whereas if Y has only 1 element (it means only one output is acceptable) the puzzle is maximally hard. The application of this concept and the idea to have a sort of computational puzzle which gives the ability to determine the difficulty of the puzzle, plays an important role in "mining" that we will talk about later.

SHA-256

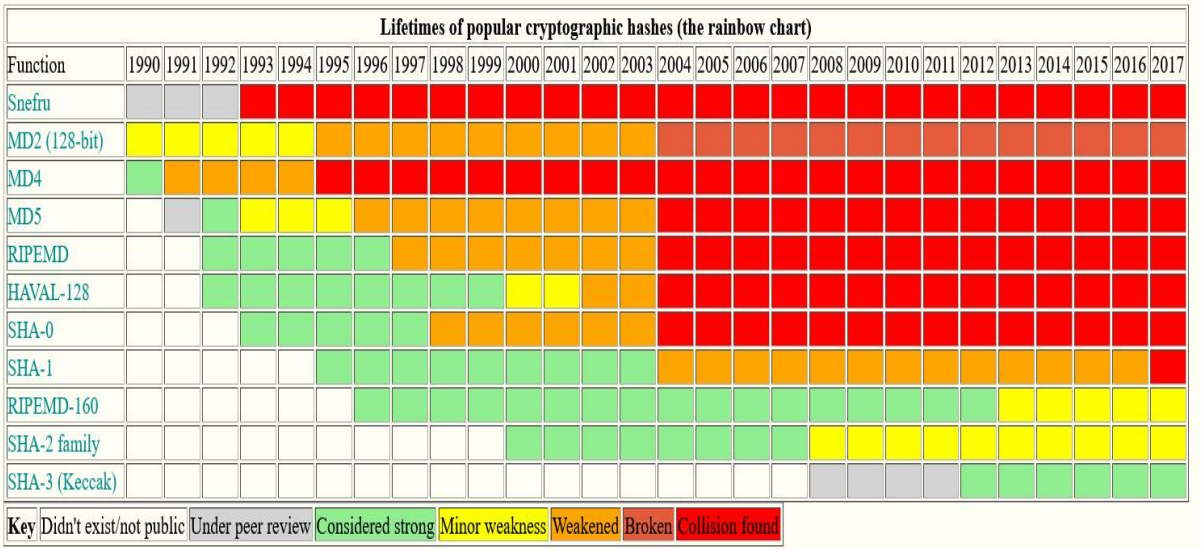


Figure 3 - Lifetimes and states of most common cryptographic hash algorithms. 1990-2017

Many hash functions exist, but this is the one Bitcoin uses primarily, and it's a pretty good one to use. It's called SHA-256 from SHA-2 family. Recall that we require that our hash functions work on inputs of arbitrary length. Luckily, as long as we can build a hash function that works on fixed-length inputs, there's a generic method to convert it into a hash function that works on arbitrary length inputs. It's called the ***Merkle-Damgard transform***. SHA-256 is one of a number of commonly used hash functions that make use of this method. In common terminology, the underlying fixed-length collision-resistant hash function is called the ***compression function***. It has been proven that if the underlying compression function is collision resistant, then the overall hash function is collision resistant as well. The Merkle-Damgard transform is quite simple. Say the compression function takes inputs of length *m* and produces an output of a smaller length *n.* The input to the hash function, which can be of any size, is divided into ***blocks*** of length *m-n.* The construction works as follows: pass each block together with the output of the previous block into the compression function. Notice that input length will then be (m-*n)* + *n* = *m,* which is the input length to the compression function. For the first block, to which there is no previous block output, we instead use an ***Initialization Vector (IV)***. This number is reused for every call to the hash function, and in practice you can just look it up in a standards document. The last block's output is the result that you return.

SHA-256 uses a compression function that takes 768-bit input and produces 256-bit outputs. The block size is 512 bits. See Figure 4 for a graphical depiction of how SHA-256 works.

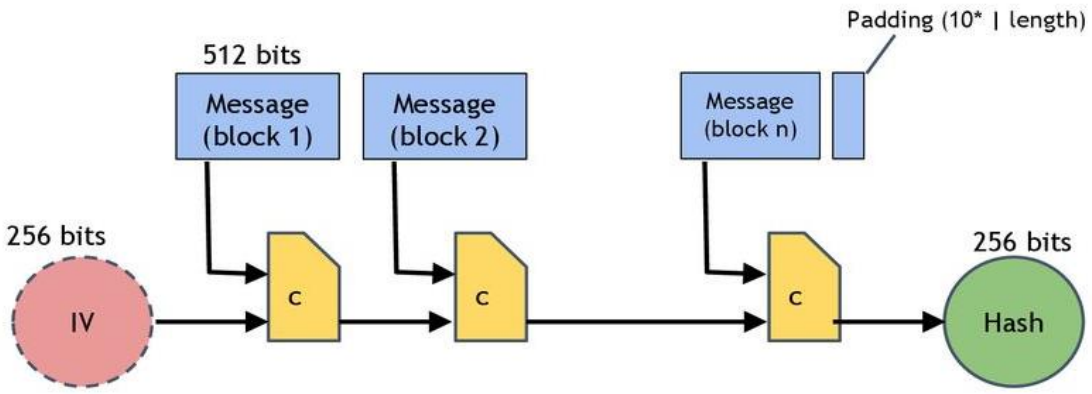


Figure 4 - SHA-256 Hash Function (simplified). SHA-256 uses the Merkle-Damgard transform to turn a fixed-length collision-resistant compression function into a hash function that accepts arbitrary-length inputs. The input is "padded" so that its length is a multiple of 512 bits.

Compression Function - One-Way Function

one-way compression function is a function that transforms and mixes two fixed length inputs and produces a single fixed length output of the same size as one of the inputs. This can also be seen as that the compression function transforms one large fixed-length input into a shorter, fixed-length output. The transformation is "one-way", meaning that it is difficult given a particular output to compute inputs which compress to that output. One-way compression functions are not related to conventional data compression algorithms, which instead can be inverted exactly (lossless compression) or approximately (lossy compression) to the original data.
The mixing is done in such a way that full "avalanche effect[7]" is achieved.

Initialization Vector (IV), (Initial Hash Value)

Initialization Vector (IV), is a 256-bit value, as the initial hash value for SHA-256 hash function. "IV" with the first 512-bit block of message, are inputs of the first compression function. "IV", as a constant value, consists of eight 32-bit word, which are reused for every call to the SHA-256 hash function. figure below shows "IV" value in Hexadecimal-representation.

$$H_0^{(0)} = \texttt{6a09e667} \qquad H_4^{(0)} = \texttt{510e527f}$$
$$H_1^{(0)} = \texttt{bb67ae85} \qquad H_5^{(0)} = \texttt{9b05688c}$$
$$H_2^{(0)} = \texttt{3c6ef372} \qquad H_6^{(0)} = \texttt{1f83d9ab}$$
$$H_3^{(0)} = \texttt{a54ff53a} \qquad H_7^{(0)} = \texttt{5be0cd19}$$

Figure 5 - "Initialization Vector", a constant value, consist of eight 32-bit word (in Hex). Recorded in SHA-256 standard.

These words were obtained by taking the first thirty-two bits of the fractional parts of the square roots of the first eight prime numbers.

---

[7] Avalanche effect, is a term associated with a specific behavior of mathematical cryptographic functions. Which is: all the bits of function's output depend on every bit of input. In other word, A slight change in input (even change in one bit) should result a significant random change in output. Indeed, this effect provides "Preimage resistance" and "hiding" property.

Merkle-Damgard Strengthening & Padding

In order to make the construction secure, Merkle and Damgard proposed that messages be padded with a padding that encodes the length of the original message. This is called *length padding* or *Merkle-Damgard strengthening*.

The message isn't going to be, in general, necessarily exactly a multiple of the block size, so we're going to add some padding at the end of last block. And the padding is going to consist of, a 64-bit length field at the end of the padding, which is the length of the message in bits(binary). And then before that, a one bit, followed by some number of zero bits.

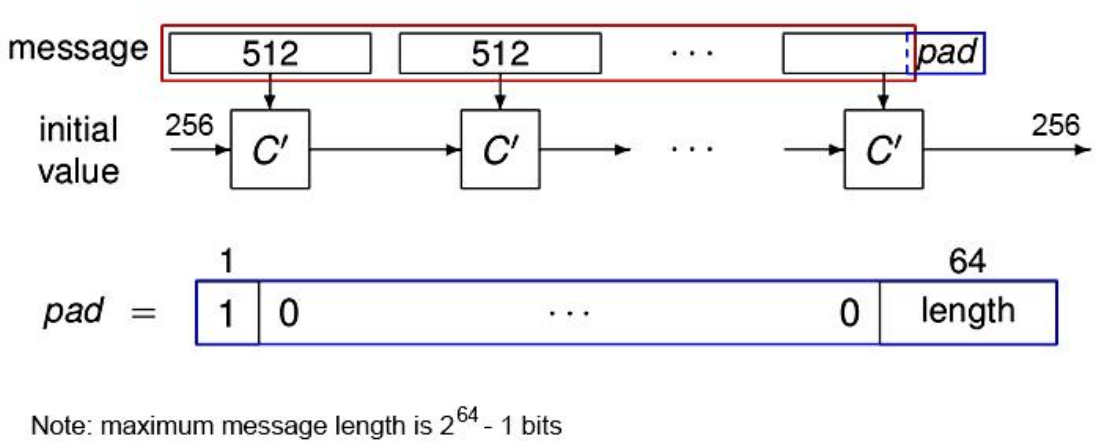


Figure 6 - Merkle-Damgard Strengthening & Padding to SHA-256 algorithm.

First, the message is padded with a binary '1' then it is cut into blocks of 512 bits. If the length of the last block does not exceed 448 bits, as many zeros as necessary are appended to fill 448 bits and the binary length of the original message (before padding) is appended in the last 64 bits of the block to form a 512bit block. Else, the block is filled with zeros up to a length of 512 bits, and an extra block is appended filled with 448 zeros; again, the binary length of the original message is appended in the last 64 bits to form a complete 512-bit block. So, once you've padded the message such that, its length is exactly a multiple of the 512-bit block size. This form of padding is non-ambiguous and is an example of a valid Merkle-Damgard strengthening.

padding is always done, even if the message happens to be a multiple of the input block size. If padding is not always done, there is an easy hash collision, a message will have the same hash as that message with the pad appended.

2.2-HASH POINTERS AND DATA STRUCTURES

In this section, we'll discuss ways of using hash functions to build more complicated data structures that are used in distributed systems like Bitcoin. we're going to discuss ***hash pointers*** and their applications.

Hash pointer

A hash pointer is a data structure that is simply a pointer to where some information is stored together with a cryptographic hash of the value of that data at some fixed point in time. Whereas a regular pointer gives you a way to retrieve the information, a hash pointer also gives you a way to verify that the information hasn't changed.

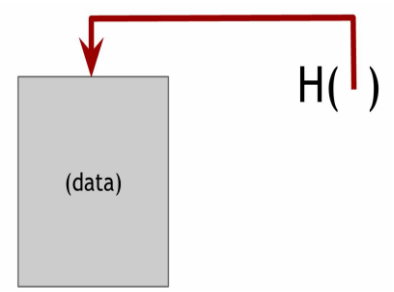


Figure 7 - A hash pointer.

Block chain

In Figure below, we built a linked list using hash pointers. We're going to call this data structure a ***blockchain.*** in a block chain, each block has data as well as a hash pointer to the previous block. So, each block not only tells us where the value of the previous block was, but it also contains a digest of that value that allows us to verify that the value hasn't changed. We store the head of the list, which is just a regular hash-pointer that points to the most recent data block.

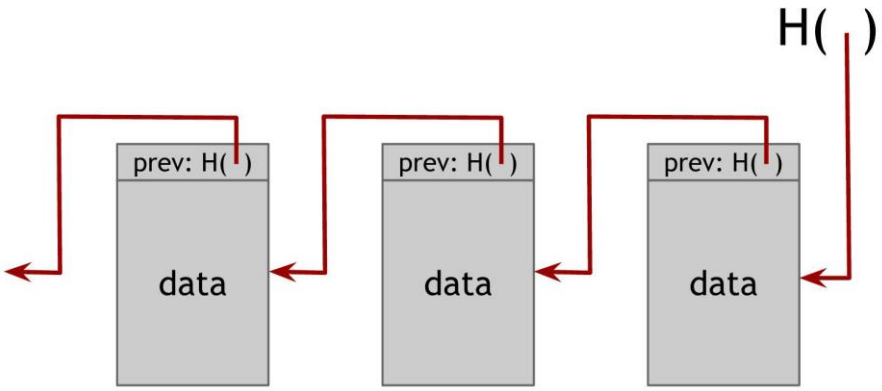


Figure 8 - A block chain is a linked list that is built with blocks of data and hash pointers

Application: tamper-evident log

A use case for a block chain is a tamper-evident log. That is, we want to build a log data structure that stores a bunch of data, and allows us to append data onto the end of the log. But if somebody alters data that is earlier in the log, we're going to detect it.

To understand why a block chain achieves this tamper-evident property, let's ask what happens if an adversary wants to tamper with data that's in the middle of the chain. the adversary changes the data of some block k. Since the data has been changed, the hash in block k + 1, which is a hash of the entire block k, is not going to match up. Remember that we are statistically guaranteed that the new hash will not match the altered content since the hash function is collision resistant. And so, we will detect

the inconsistency between the new data in block k and the hash pointer in block k + 1. the adversary can continue to try and cover up this change by changing the next block's hash as well. but this strategy will fail when he reaches the head of the list, and because he won't be able to tamper with that, as long as we store the hash pointer at the head of the list. Thus, by just remembering this single hash pointer, we've essentially remembered a tamper-evident hash of the entire list. and the adversary will be unable to change any block without being detected. So, we can build a block chain like this containing as many blocks as we want.

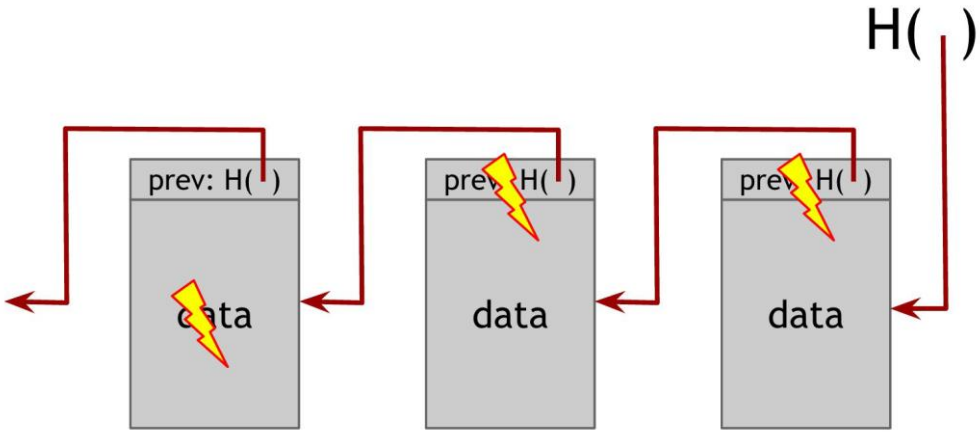


Figure 9 - Tamper-evident log. If an adversary modifies data anywhere in the block chain, it will result in the hash pointer in the following block being incorrect. If we store the head of the list, then even if the adversary modifies all of the pointers to be consistent with the modified data, the head pointer will be incorrect, and we will detect the tampering.

Merkle tree

Another useful data structure that we can build using hash pointers is a binary tree. A binary tree with hash pointers is known as a ***Merkle tree***, after its inventor Ralph Merkle. Suppose we have a number of blocks containing data. These blocks comprise the leaves of our tree. We group these data blocks into pairs of two, and then for each pair, we build a data structure that has two hash pointers, one to each of these blocks. These data structures make the next level up of the tree. We in turn group these into groups of two, and for each pair, create a new data structure that contains the hash of each. We continue doing this until we reach a single block, the root of the tree.

As before, we remember just the hash pointer at the head of the tree. We now have the ability to follow paths through the hash pointers to any point in the list. This allows us make sure that the data hasn't been tampered with because, just like we saw with the block chain, if an adversary tampers with some data block at the bottom of the tree, that will cause the hash pointer that's one level up to not match, and even if he continues to tamper with this block, the change will eventually propagate to the top of the tree where he won't be able to tamper with the hash pointer that we've stored. So again, any attempt to tamper with any piece of data will be detected by just remembering the hash pointer at the top.

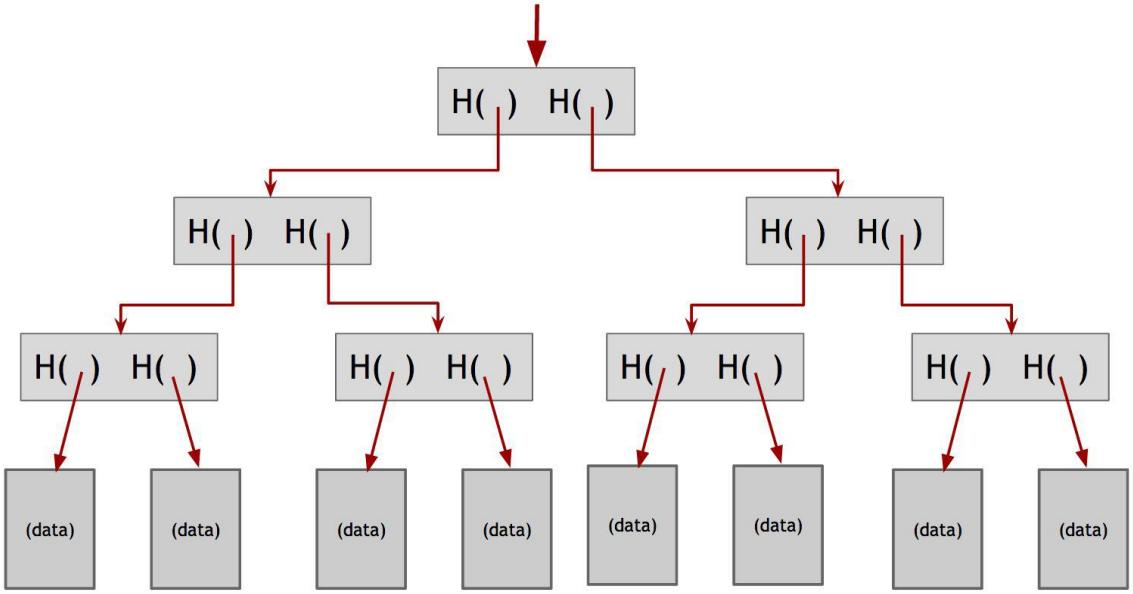


*Figure 10 - **Merkle tree.** In a Merkle tree, data blocks are grouped in pairs and the hash of each of these blocks is stored in a parent node. The parent nodes are in turn grouped in pairs and their hashes stored one level up the tree. This continues all the way up the tree until we reach the root node.*

Application1: proof of membership

Another nice feature of Merkle trees is that, unlike the block chain that we built before, it allows a concise proof of membership. Say that someone wants to prove that a certain data block is a member of the Merkle Tree. As usual, we remember just the root. Then they need to show us this data block, and the blocks on the path from the data block to the root. We can ignore the rest of the tree, as the blocks on this path are enough to allow us to verify the hashes all the way up to the root of the tree. See Figure below for a graphical depiction of how proof of membership in Merkle Tree works.

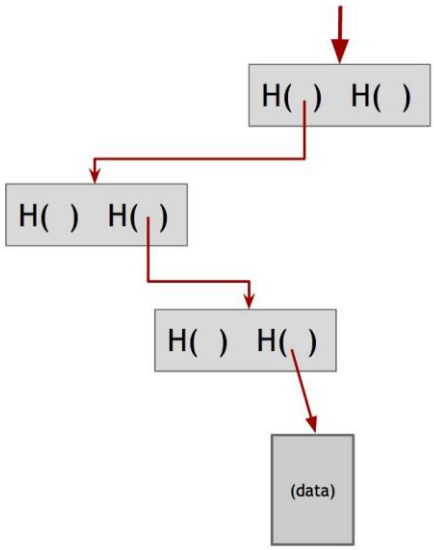


Figure 11 - Proof of membership. To prove that a data block is included in the tree, one only needs to show the blocks in the path from that data block to the root.

If there are *n* nodes in the tree, only about *log(n)* items need to be shown. And since each step just requires computing the hash of the child block, it takes about *log(n)* time for us to verify it. And so even if the Merkle tree contains a very large number of blocks, we can still prove membership in a

relatively short time. Verification thus runs in time and space that's logarithmic in the number of nodes in the tree.
while, in the blockchain data structure, in order to check the membership of a certain data in the blocks, we need to have: other complementary data of same block, all the data from later blocks, and at least hash-pointer of the previous block. Also, all the calculation on the blocks most be repeated. if we concluded the hash-pointer of the latest block, as same as the hash-pointer that we stored before, then we find the proof of membership.

Application2: proof of non-membership - Variant: sorted Merkle tree

A ***sorted Merkle tree*** is just a Merkle tree where we take the blocks at the bottom, and we sort them using some ordering function. This can be alphabetical, lexicographical order, numerical order, or some other agreed upon ordering.
With a sorted Merkle tree, it becomes possible to verify non-membership in a logarithmic time and space. That is, we can prove that a particular block is not in the Merkle tree. And the way we do that is simply by showing a path to the item that's just before where the item in question would be and showing the path to the item that is just after where it would be. If these two items are consecutive in the tree, then this serves as a proof that the item in question is not included. For if it was included, it would need to be between the two items shown, but there is no space between them as they are consecutive.

2.3-DIGITAL SIGNATURES

In this section, we'll look at ***digital signatures.*** This is the second cryptographic primitive, along with hash functions. A digital signature is an electronic analogue of a written signature. We desire two properties from digital signatures, Firstly, the digital signature can be used to provide assurance that the claimed signatory signed the information and only you can make your signature, but anyone who sees it can verify that it's valid. Secondly, a digital signature should be used to detect whether or not the information was modified after it was signed (i.e., to detect the integrity of the signed data). In other word, we want the signature to be tied to a particular document so that the signature cannot be used to indicate your agreement or endorsement of a different document. A properly implemented digital signature algorithm that meets the requirements of this Standard can provide these services.

Digital signature scheme

A digital signature scheme consists of the following three algorithms:

- **keys generation: (sk, pk) := generatekeys(*keysize*)** The generatekeys method takes a key size and generates a key pair. The private key or secret key *sk* is kept privately and used to sign messages. *Pk* is the public verification key that you give to everybody. Anyone with this key can verify your signature.
- **Signature generation: sig := sign(*sk* , *message*)** The sign method takes a message and a secret key, *sk*, as input and outputs a signature for *message* under *sk.*
- **Signature verification: isvalid := verify(*pk* , *message* , *sig*)** The verify method takes a message, a signature, and a public key as input. It returns a boolean value, *isvalid,* that will be ***true*** if *sig* is a valid signature for *message* under public key *pk*, and ***false*** otherwise.

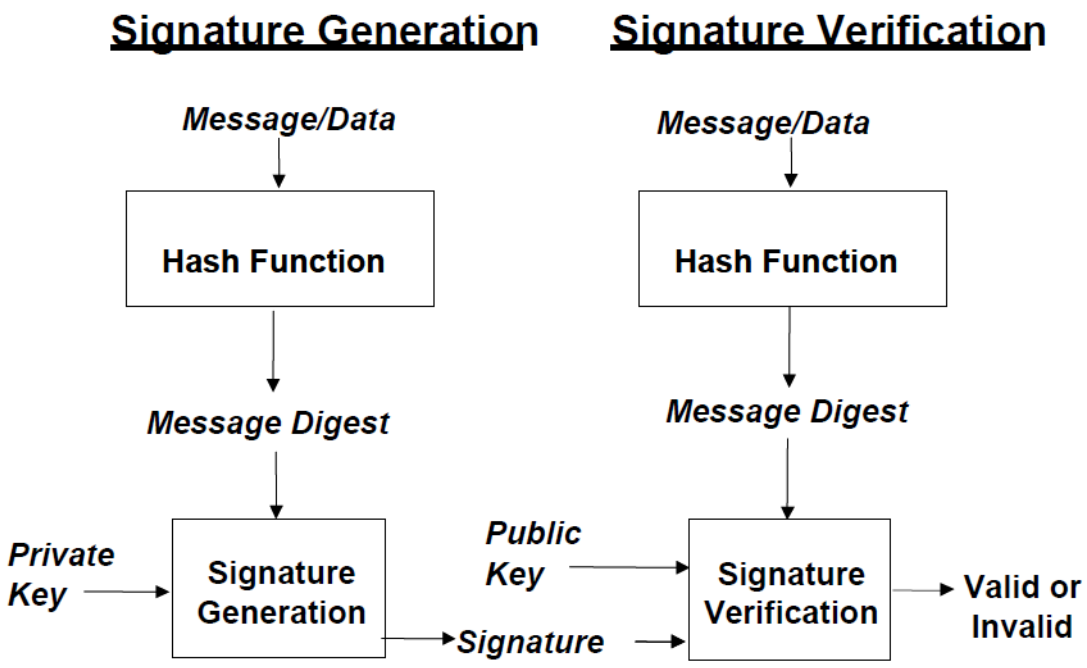


Figure 12 - Digital signature processes

A digital signature algorithm includes a signature generation process and a signature verification process. A signatory uses the generation process to generate a digital signature on data; a verifier uses the verification process to verify the authenticity of the signature. Each signatory has a public and

private key and is the owner of that key pair. the private key is used in the signature generation process. The key pair owner is the only entity that is authorized to use the private key to generate digital signatures. In order to prevent other entities from using the private key to generate fraudulent signatures, the private key must remain secret. The approved digital signature algorithms are designed to prevent an adversary who does not know the signatory's private key from generating the same signature as the signatory on a different message. The public key is used in the signature verification process. The public key need not be kept secret, anyone can verify a correctly signed message using the public key.

keys generation (Generatekeys) and **Signature generation (sign)** algorithms can be randomized algorithms. Indeed, generatekeys had better be randomized because it ought to be generating different keys for different people. **Signature verification**, on the other hand, will always be deterministic. For both the signature generation and verification processes, the message (i.e., the signed data) is converted to a fixed-length representation of the message by means of an approved hash function. Both the original message and the digital signature are made available to a verifier. A verifier also requires assurance that the key pair owner actually possesses the private key associated with the public key, and that the public key is a mathematically correct key.

Briefly, we require from these algorithms that the following two properties hold:

1.Valid signatures must verify: verify (pk , message , sign (sk , message)) == **true**

If I sign a message with sk , my secret key, and someone later tries to validate that signature over that same message using my public key, pk , the signature must validate correctly. This property is a basic requirement for signatures to be useful at all.

2.Signatures are existentially unforgeable:

The second requirement is that it's computationally infeasible to forge signatures. That is, an adversary who knows your public key and gets to see your signatures on some other messages can't forge your signature on some message for which he has not seen your signature. Signature scheme is unforgeable if and only if, no matter what algorithm the adversary is using, his chance of successfully forging a message is extremely small, so small that we can assume it will never happen in practice.

ECDSA

Bitcoin uses a particular digital signature scheme that's called the Elliptic Curve Digital Signature Algorithm (ECDSA). ECDSA is a U.S. government standard, an update of the earlier DSA algorithm adapted to use elliptic curves. These algorithms have received considerable cryptographic analysis over the years and are generally believed to be secure.

More specifically, Bitcoin uses ECDSA over the standard elliptic curve "secp256k1" which is estimated to provide 128 bits of security (that is, it is as difficult to break this algorithm as performing 2^{128} symmetric-key cryptographic operations such as invoking a hash function).

It might be useful to have an idea of the sizes of various quantities in ECDSA:

Private key:	256 bits
Public key, uncompressed:	512 bits
Public key, compressed:	257 bits
Message to be signed:	256 bits
Signature:	512 bits

Figure 13 - sizes of various quantities in ECDSA

Note that while ECDSA can technically only sign messages 256 bits long, this is not a problem: messages are always hashed before being signed, so effectively any size message can be efficiently signed. Also, you can sign a hash pointer. If you sign a hash pointer, then the signature covers, or protects, the whole structure and everything the chain of hash pointers points to. For example, if you were to sign the hash pointer that was at the end of a block chain, the result is that you would effectively be digitally signing the that entire block chain.

Public Keys as Identities

The idea is to take a public key and equate that to an identity of a person or an actor in a system. If you see a message with a signature that verifies correctly under a public key, *pk,* then you can think of this as *pk* is saying the message. via the **generateKeys** operation in our digital signature scheme. pk is the new public identity that you can use, and sk is the corresponding secret key that only you know and lets you speak for on behalf of the identity pk. In practice, you may use the hash of pk as your identity since public keys are large. If you do that, then in order to verify that a message comes from your identity, one will have to check (1) that pk indeed hashes to your identity, and (2) the message verifies under public key pk. Moreover, by default, your public key *pk* will basically look random, and nobody will be able to uncover your real-world identity by examining *pk.* But of course, once you start making statements using this identity, these statements may leak information that allows one to connect *pk* to your real-world identity.

Application: Decentralized identity management.

This brings us to the idea of decentralized identity management. Rather than having a central authority that you have to go to in order to register as a user in a system, you can register as a user all by yourself. You don't need to be issued a username nor do you need to inform someone that you're going to be using a particular name. If you want to be somewhat anonymous for a while, you can make a new identity, use it just for a little while, and then throw it away. All of these things are possible with decentralized identity management, and this is the way Bitcoin, in fact, does identity. These identities are called **addresses**, in Bitcoin jargon. And that's really just a hash of a public key.

Pseudo-anonymity: in Bitcoin you don't need to explicitly register or reveal your real-world identity, but the pattern of your behavior might itself be identifying, this feature called Pseudo-anonymity.

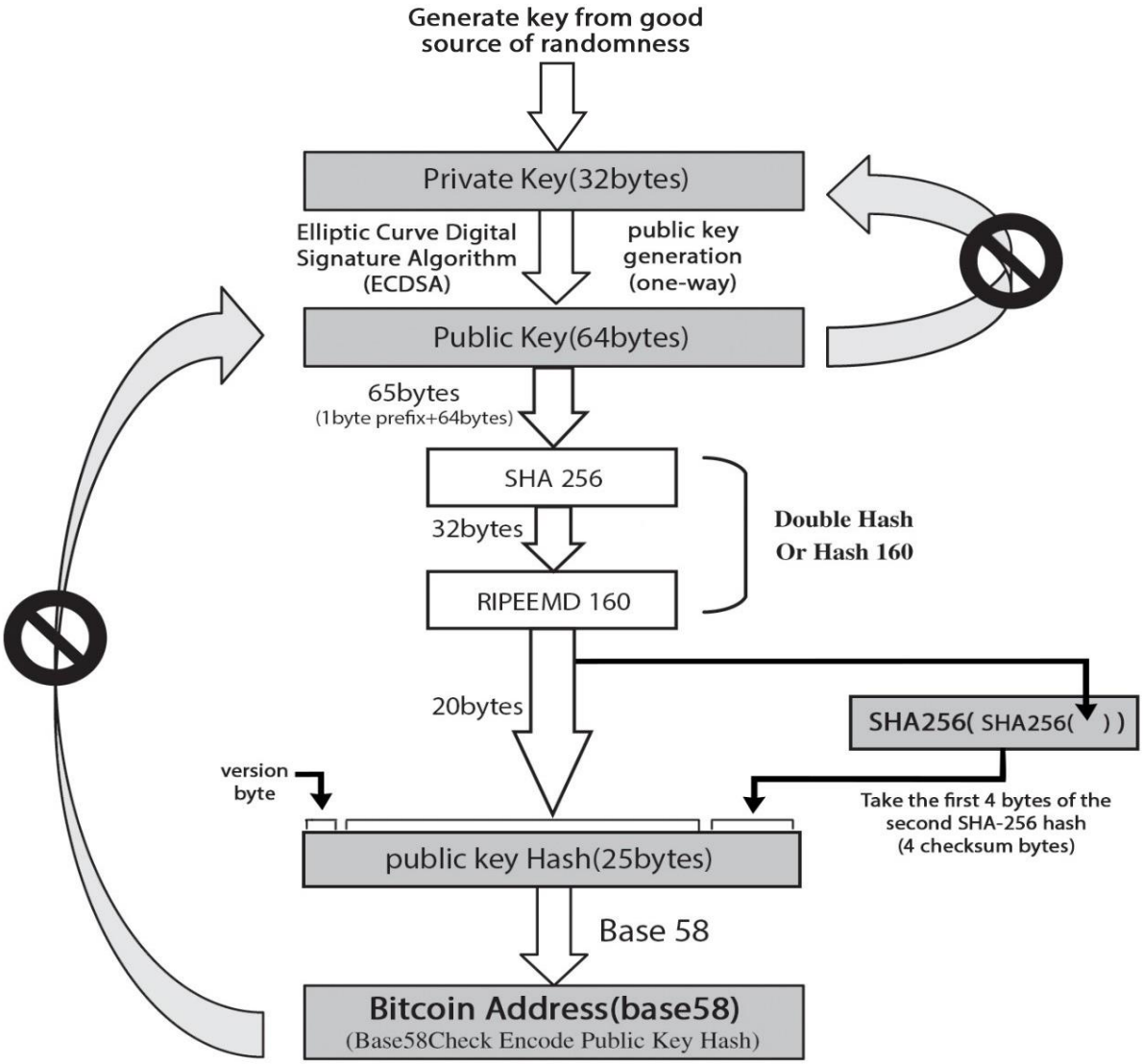


Figure 14 - Conversion from private key to Bitcoin Address

3-BITCOIN MECHANISM & NETWORK

3.1-Bitcoin in use

As said before, Bitcoin relies on cryptographic proof instead of trusted third parties. Public key cryptography is used to make and verify digital signatures that users use to send payments. Let's suppose Alice and Bob are two users in the bitcoin network. Alice and Bob each have an address which is similar to a bank account number and tracks the number of bitcoins they have. The address is also associated with a public and private key (see fig. 14). The private key is used to sign transactions when sending bitcoins while the public key can be used by anyone to validate the transaction signature.
Now, suppose Alice wants to send bitcoins to Bob:

1. Bob sends his address to Alice.
2. Alice adds Bob's address and the amount of bitcoins to a 'transaction' message.
3. Alice then signs the transaction message with her private key and announces her public key for signature verification.
4. Alice broadcast the transaction on the bitcoin network where all users can see the message.

All users on the Bitcoin network that know the transaction addresses belong to Alice and Bob can see that Alice has transferred bitcoins to Bob.
Later, Bob decides to transfer the same bitcoins to Charlie. Bob now repeats the steps Alice performed to send her bitcoin to Bob.
Another user, Eve cannot try to steal these bitcoins by replacing Bob or Charlie's address with her own. The transfers were signed with Alice and Bob's private key instructing that the coins were transferred from Alice to Bob and then Bob to Charlie. Once Charlie accepts the coins, he also accepts that the coins were first passed from Alice to Bob, and then from Bob to him.
This record of transactions between Alice, Bob, and Charlie is added to a constantly growing chain of blocks that contains the record of all transactions on the bitcoin network. The record of transactions is maintained by the bitcoin network, and each block is validated with proof of work before it is accepted into the chain. Valid blocks are chained together so that the transfer of bitcoins can be tracked. Once the block containing Alice's transaction to Bob is added to the block chain, Bob can be confident that the transaction has been accepted by other computers in the network and permanently recorded. This prevents Alice from trying to send the same coins to another user and double spending her coins. The bitcoin network generates blocks every 10 minutes which would require Bob to wait at least this amount of time to be able to verify the transaction.

3.2-Bitcoin Concepts

Transactions

Satoshi Nakamoto, bitcoin creator, defined an electronic coin as a chain of digital signatures. Each owner transfers the coin to the next by digitally signing a hash of the previous transaction and the public key of the next owner and adding these to the end of the coin. A payee can verify the signatures to verify the chain of ownership.

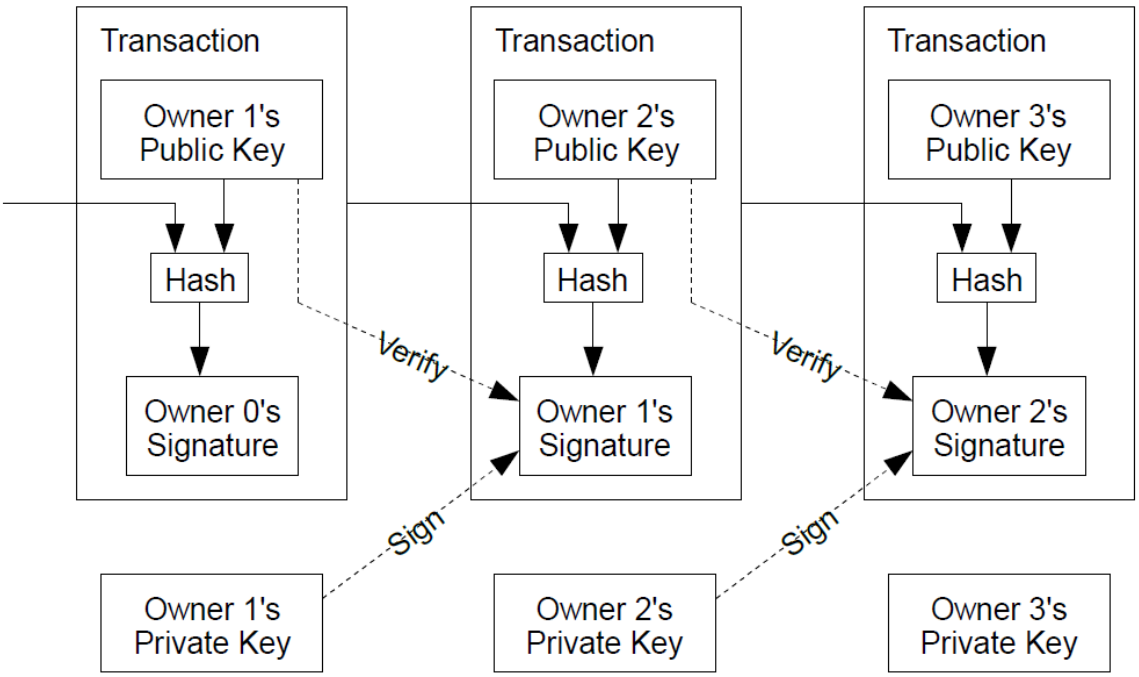


Figure 15 - Electronic coin as a chain of digital signatures.

The problem of course is the payee can't verify that one of the owners did not double-spend the coin. A common solution is to introduce a trusted central authority, or mint, that checks every transaction for double spending. After each transaction, the coin must be returned to the mint to issue a new coin, and only coins issued directly from the mint are trusted not to be double-spent. The problem with this solution is that the fate of the entire money system depends on the company running the mint, with every transaction having to go through them, just like a bank.

We need a way for the payee to know that the previous owners did not sign any earlier transactions. For our purposes, the earliest transaction is the one that counts, so we don't care about later attempts to double-spend. The only way to confirm the absence of a transaction is to be aware of all transactions. In the mint-based model, the mint was aware of all transactions and decided which arrived first. To accomplish this without a trusted party, transactions must be publicly announced, and we need a system for participants to agree on a single history of the order in which they were received. The payee needs proof that at the time of each transaction, the majority of nodes agreed it was the first received.

Timestamp Server

The solution which Nakamoto proposed begins with a timestamp server. A timestamp server works by taking a hash of a block of items to be timestamped and widely publishing the hash. Each timestamp includes the previous timestamp in its hash, forming a chain, with each additional timestamp reinforcing the ones before it.

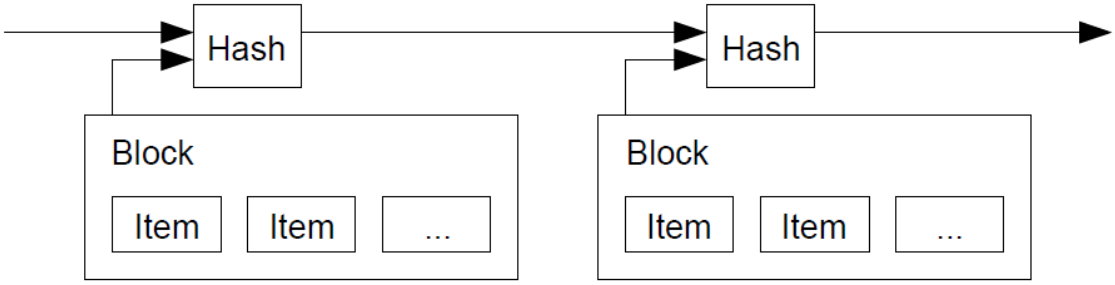


Figure 16 - Taking a hash of a block of items and hash of previous timestamp to be timestamped.

Proof-of-Work

To implement a distributed timestamp server on a peer-to-peer basis, we will need to use a proof-of-work system. The proof-of-work involves scanning for a value that when hashed, such as with SHA-256, the hash begins with a number of zero bits. The average work required is exponential in the number of zero bits required and can be verified by executing a single hash.

For our timestamp network, we implement the proof-of-work by incrementing a nonce in the block until a value is found that gives the block's hash the required zero bits.[8] Once the CPU effort has been expended to make it satisfy the proof-of-work, the block cannot be changed without redoing the work. As later blocks are chained after it, the work to change the block would include redoing all the blocks after it.

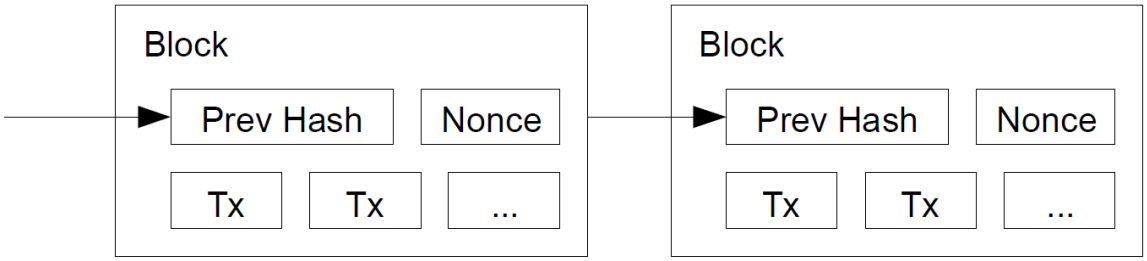


Figure 17 - Find a Nonce which satisfy proof-of-work target.

Incentive

By convention, the first transaction in a block is a special transaction that starts a new coin owned by the creator of the block. This adds an incentive for nodes to support the network, and provides a way to initially distribute coins into circulation, since there is no central authority to issue them. The steady addition of a constant of amount of new coins is analogous to gold miners expending resources to add gold to circulation. In our case, it is CPU time and electricity that is expended.

The incentive can also be funded with transaction fees. If the output value of a transaction is less than its input value, the difference is a transaction fee that is added to the incentive value of the block containing the transaction. Once a predetermined number of coins[9] have entered circulation, the incentive can transition entirely to transaction fees and be completely inflation free.

The incentive may help encourage nodes to stay honest. If a greedy attacker is able to assemble more CPU power than all the honest nodes, he would have to choose between using it to defraud people by stealing back his payments, or using it to generate new coins. He ought to find it more profitable

---

[8] *This concept will discuss with more details in next section, titled by "Difficulty".*

[9] *The total supply of BTC is limited and pre-defined in the Bitcoin protocol at 21-million. Since 2009 to early 2022, approximately 18.9-milion of it mined (have entered circulation with the mining reward).*

to play by the rules, such rules that favor him with more new coins than everyone else combined, than to undermine the system and the validity of his own wealth.

Bitcoin Mining in summary

Bitcoin mining[10] is essentially the process of cryptographic hashing a block header looking for a hash that, when interpreted as big number, has the very unlikely property of being bellow some small target value. Such target is automatically adjusted by the Bitcoin network to keep the generation of Bitcoins in a constant pace. When the computing power of the network increases, the target decreases causing the mining process to become more difficult. On the other hand, if the computing power of the network decreases, the target is increased also increasing the likelihood of finding a hash bellow this target.

Inside each block header, a special field called nonce is reserved for mining purposes only. Miners tries different nonce values looking for a valid hash (a hash bellow the target value). Each time a new nonce is tried, a new hash is obtained. If a nonce generates a valid hash, it is called a golden nonce.

For any good hashing algorithm (like SHA-256), there is no known way to find the golden nonce other than by sweeping all possible nonce values. This ensures that in order to find a golden nonce miners have to spend certain amount of computer power bringing out the proof-of-work concept which backs the reliability of the Bitcoin ecosystem.

Consensus mechanism

The proof-of-work also solves the problem of determining representation in majority decision making. If the majority were based on one-IP-address-one-vote, it could be subverted by anyone able to allocate many IPs. Proof-of-work is essentially one-CPU-one-vote. The majority decision is represented by the longest chain, which has the greatest proof-of-work effort invested in it. If a majority of CPU power is controlled by honest nodes, the honest chain will grow the fastest and outpace any competing chains. To modify a past block, an attacker would have to redo the proof-of-work of the block and all blocks after it and then catch up with and surpass the work of the honest nodes. The probability of a slower attacker catching up diminishes exponentially as subsequent blocks are added. Nodes always consider the longest chain to be the correct one and will keep working on extending it. If two nodes broadcast different versions of the next block simultaneously, some nodes may receive one or the other first. In that case, they work on the first one they received, but save the other branch in case it becomes longer. The tie will be broken when the next proof-of- work is found and one branch becomes longer; the nodes that were working on the other branch will then switch to the longer one.

Bitcoin consensus algorithm (simplified)

The steps to run the network are as follows:

1. New transactions are broadcast to all nodes.
2. Each node collects new transactions into a block.
3. Each node works on finding a difficult proof-of-work for its block.
4. When a node finds a proof-of-work, it broadcasts the block to all nodes.
5. Nodes accept the block only if all transactions in it are valid and not already spent.
6. Nodes express their acceptance of the block by working on creating the next block in the chain, using the hash of the accepted block as the previous hash.

---

[10] This subject will discuss in next section, with more details.

New transaction broadcasts do not necessarily need to reach all nodes. As long as they reach many nodes, they will get into a block before long. Block broadcasts are also tolerant of dropped messages. If a node does not receive a block, it will request it when it receives the next block and realizes it missed one.

Privacy

The traditional banking model achieves a level of privacy by limiting access to information to the parties involved and the trusted third party. The necessity to announce all transactions publicly precludes this method, but privacy can still be maintained by breaking the flow of information in another place: by keeping public keys anonymous. The public can see that someone is sending an amount to someone else, but without information linking the transaction to anyone. This is similar to the level of information released by stock exchanges, where the time and size of individual trades, the "tape", is made public, but without telling who the parties were.

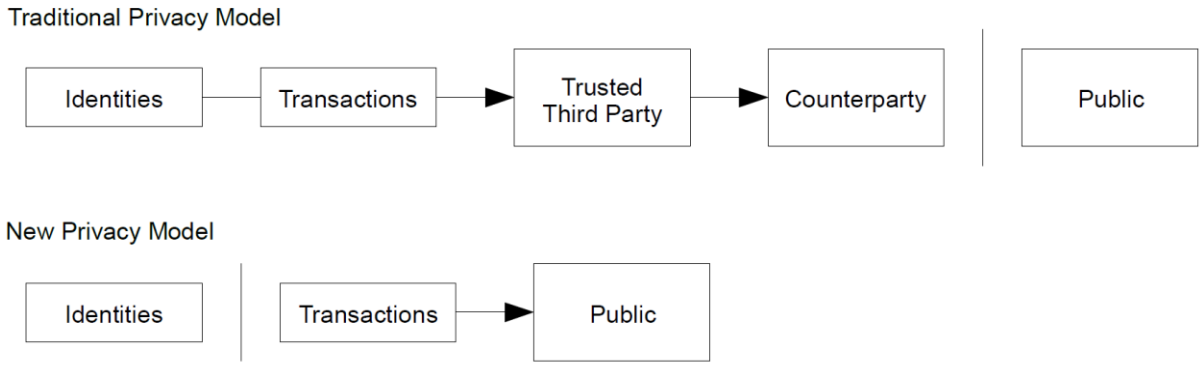


Figure 18 - Traditional-Privacy-Model Vs New-Privacy-Model

3.3-Bitcoin blockchain data structure

An overview of the Bitcoin blockchain data structure is depicted in Figure below. To be stored in the Bitcoin ledger, every individual transaction should be embedded in a Bitcoin block data structure. the Bitcoin blockchain is a sequence of blocks linked with hash values. Each block consists of a block header and a block body. Transactions are stored in the block body, and digest information and other identifiers are recorded in the block header. A blockchain is maintained by the nodes participating in the network, and the data consistency among the nodes is ensured according to predetermined rules of Consensus.

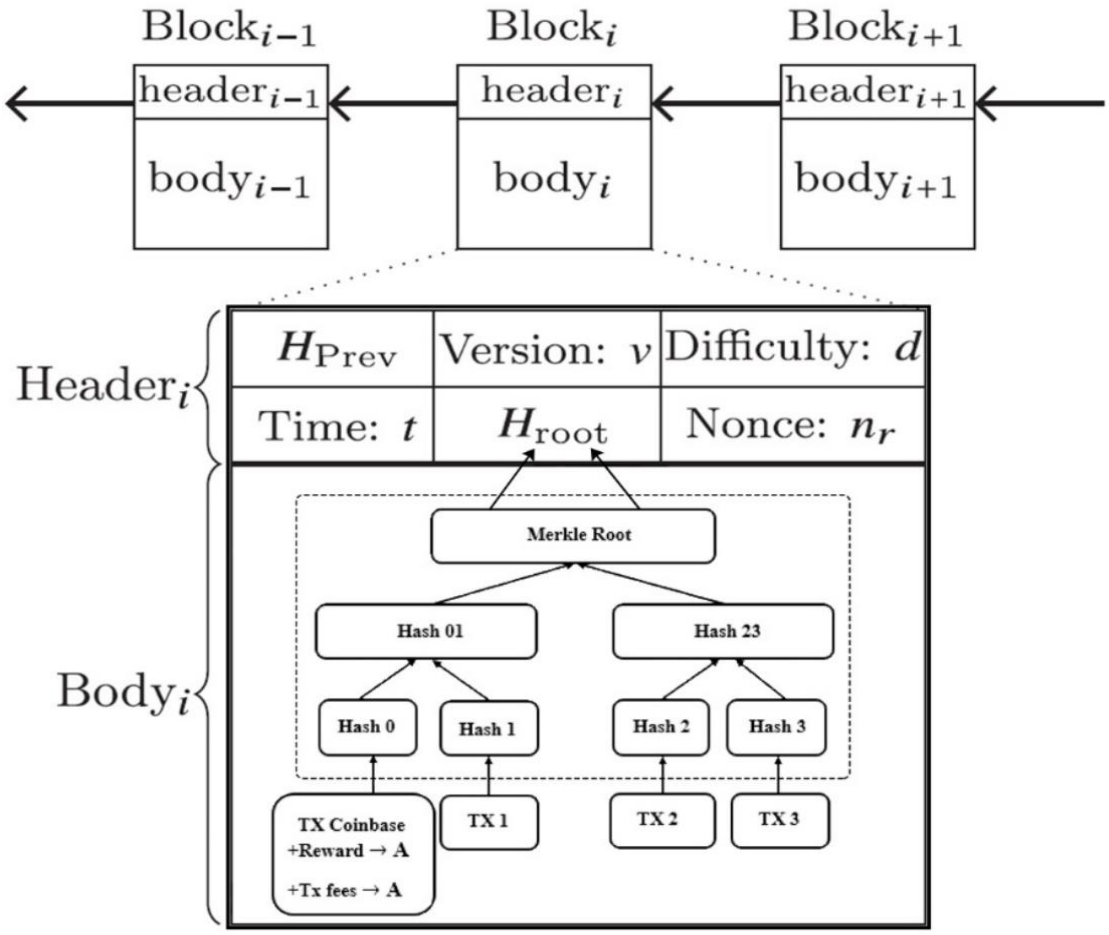
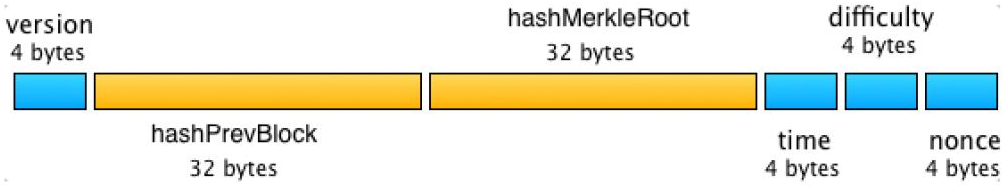


Figure 19 - A overview of the Bitcoin blockchain data structure.

Block Header

A block is “solved” (published and considered valid by peers) when the hash of the block header is below the current target. The block header consists of 640 bits (80 bytes) as shown in figure below. Most of fields are constants, but miners can play with one of them: nonce.



Field	Purpose	Updated When	Size (Bytes)
Version	A version number to track software/ protocol upgrades	When software is upgraded, a new version is specified	4
Previous Block Hash	256-bit hash of the previous block header	A new block comes in	32
Merkle Root	A hash of the root of the merkle tree of this block's transactions	A transaction is accepted	32
Timestamp	The approximate creation time of this block	Every few seconds	4
Bits(Target)	Current target in compact format	The difficulty is adjusted	4
Nonce	A counter used for the Proof-of-Work algorithm	A hash is tried (increments)	4

Figure 20 - The Block header fields and structure, The block header is an 80-byte value.

4-BITCOIN MINING

Bitcoin, at a simple glance, is three things. First it is a protocol (or set of rules) that defines how the network should operate. Second it is a software project that implements that protocol. Third it is a network of computers and devices running software that uses the protocol to create and manage the Bitcoin currency. There's no central authority for Bitcoins, similar to a central bank which controls currencies. Instead, programmers solve complex puzzles to endorse Bitcoin transactions and get Bitcoins as a reward. This activity is called Bitcoin mining, Mining is defined in the protocol, implemented in software, and is an essential function in managing the Bitcoin network. Miners validate every transaction, they build and store all the blocks, and they reach a consensus on which blocks to include in the block chain.

4.1-The task of Bitcoin miners

To be a Bitcoin miner, you have to join the Bitcoin network and connect to other nodes. Once you're connected, there are six tasks to perform:

1. ***Listen for transactions.*** First, you listen for transactions on the network and validate them by checking that signatures are correct and that the outputs being spent haven't been spent before.
2. ***Maintain block chain and listen for new blocks.*** You must maintain the block chain. You start by asking other nodes to give you all of the historical blocks that are already part of the block chain before you joined the network. You then listen for new blocks that are being broadcast to the network. You must validate each block that you receive by validating each transaction in the block and checking that the block contains a valid nonce. We'll return to the details of nonce checking later in this section.
3. ***Assemble a candidate block.*** Once you have an up-to-date copy of the block chain, you can begin building your own blocks. To do this, you group transactions that you heard about into a new block that extends the latest block you know about. You must make sure that each transaction included in your block is valid.
4. ***Find a nonce that makes your block valid.*** This step requires the most work and it's where all the real difficulty happens for miners.
5. ***Hope your block is accepted.*** Even if you find a block, there's no guarantee that your block will become part of the consensus chain. There's bit of luck here; you have to hope that other miners accept your block and start mining on top of it, instead of some competitor's block.
6. ***Profit.*** If all other miners do accept your block, then you profit! At the time of this writing in early 2022, the block reward is 6.25 bitcoins which is currently worth over $312,500. In addition, if any of the transactions in the block contained transaction fees, the miner collects those too. Mathematically, transaction fees are the difference between the amount of bitcoin sent and the amount received. Conceptually, transaction fees are a reflection of the speed with which a user wants their transaction validated on the blockchain. Transaction fees are based on the data volume of a transaction and the congestion of the network. through the Segregated Witness (SegWit) protocol upgrade in 2017, now A block can contain a maximum theoretical limit of 4 MB of data and a more realistic limit of 2 MB. The default limit before the upgrade was 1MB. so, there is a limit to how many transactions can be processed in one block. A larger transaction will take up more block data. Thus, larger transactions typically pay fees on a per-byte basis. If you wish to have your transaction confirmed immediately, your optimal fee rate may vary significantly.

We can classify the steps that a miner must take into two categories.

1. ***Validating transactions and blocks.*** Some tasks help the Bitcoin network and are fundamental to its existence. These tasks are the reason that the Bitcoin protocol requires miners in the first place
2. **The race to find blocks and profit.** Some tasks aren't necessary for the Bitcoin network itself but are intended to incentivize miners to perform the essential steps. Of course, both of these are necessary for Bitcoin to function as a currency, since miners need an incentive to perform the critical steps.

Finding a valid block

In previous sections, we saw that there are two main hash-based structures. There's the block chain where each block header points to the previous block header in the chain, and then within each block there's a Merkle tree of all of the transactions included in that block. The first thing that you do as a miner is to compile a set of valid transactions that you have from your pending transaction pool into a Merkle tree. Of course, you may choose how many transactions to include up to the limit on the total size of the block. You then create a block with a header that points to the previous block. In the block header, there's a 32-bit nonce field, and you keep trying different nonces looking for one that causes the block's hash to be under the target. For practical simple example, to begin with the required number of zeros. A miner may begin with a nonce of 0 and successively increment it by one in search of a nonce that makes the block valid.

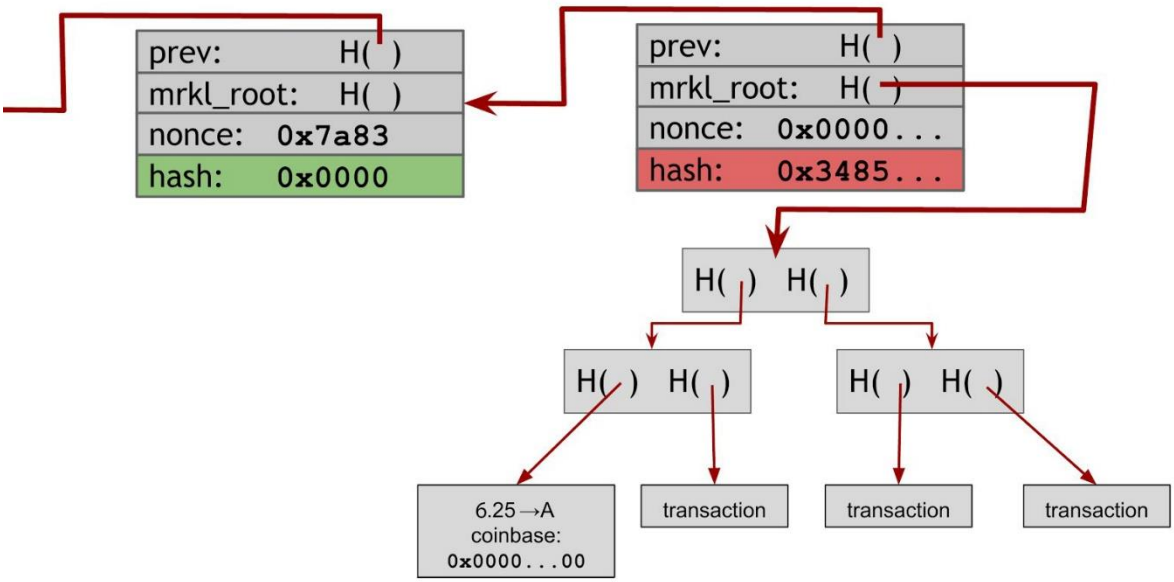


Figure 21 - Finding a valid block. In this example, the miner tries a nonce of all 0s. It does not produce a valid hash output, so the miner would then proceed to try a different nonce.

In most cases you'll try every single possible 32-bit value for the nonce and none of them will produce a valid hash. At this point you're going to have to make further changes. Notice in Figure above that there's an additional nonce in the Coinbase transaction that you can change as well. After you've exhausted all possible nonces for the block header, you'll change the extra nonce in the Coinbase transaction, for example, by incrementing it by one, and then you'll start searching nonces in the block header once again.

When you change the nonce parameter in the Coinbase transaction, the entire Merkle tree of transactions has to change (See Figure below). Since the change of the Coinbase nonce will propagate all the way up the tree, changing the extra nonce in the Coinbase transaction is much more expensive operation than changing the nonce in the block header. For this reason, miners spend most of their time changing the nonce in the block header and only change the Coinbase nonce when they have exhausted all of the 2^{32} possible nonces in the block header without finding a valid block.

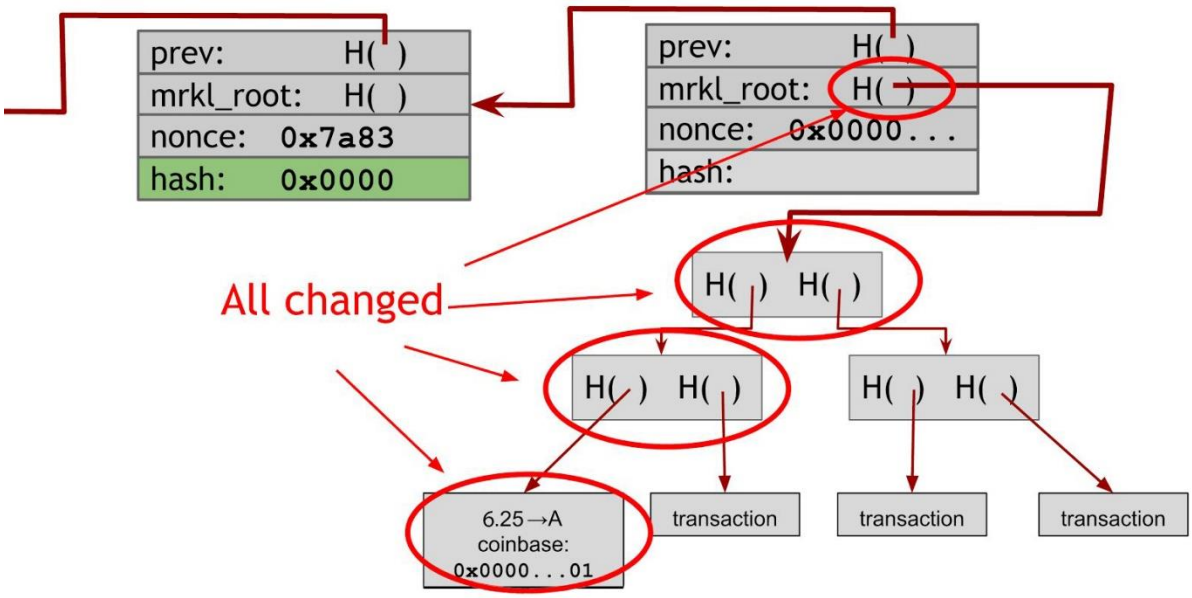


Figure 22 - Changing a nonce in the Coinbase transaction propagates all the way up the Merkle tree.

The vast, vast majority of nonces that you try aren't going to work, but if you stay at it long enough, you'll eventually find the right combination of the extra nonce in the Coinbase transaction and the nonce in the block header that produce a block with a hash under the target. When you find this, you want to announce it as quickly as you can and hope that you can profit from it.

Is everyone solving the same puzzle?

if every miner just increments the nonces as described, aren't all miners solving the exact same puzzle? Won't the fastest miner always win? The answer is no! Firstly, it's unlikely that miners will be working on the exact same block as each miner will likely include a somewhat different set of transactions and in a different order. But more importantly, even if two different miners were working on a block with identical transactions, the blocks would still differ. Recall that in the Coinbase transaction, miners specify their own address as the owner of the newly minted coins. This address by itself will cause changes which propagate up to the root of the Merkle tree, ensuring that no two miners are working on exactly the same puzzle unless they share a public key. This would only happen if the two miners are part of the same mining pool (which we'll discuss shortly), in which case they'll communicate to ensure they include a distinct nonce in the Coinbase transaction to avoid duplicating work.

Difficulty

The Bitcoin mining difficulty defines how hard it is to mine a new block. In other word, Difficulty is a measure of how difficult it is to find a hash below a given target. The Bitcoin network has a global block difficulty. Valid blocks must have a hash below this target.
There are two values we need to consider:

Target ("Bits"): The Bitcoin difficulty target is a 256-bit number. It is stored as an encoded packed representation (called "Bits") for its actual hexadecimal target. The target states how high a block hash can be in order to fulfill the mining condition. The higher the target the easier it is to find a value below it and the lower the difficulty.
Difficulty: Is a numerical expression of how difficult it is to find a suitable hash compared to the easiest difficulty of 1. That means, Floating point representation of difficulty shows how much current target is harder than the one used in the genesis block.
Figure below shows summary information of first block mined in 2022:

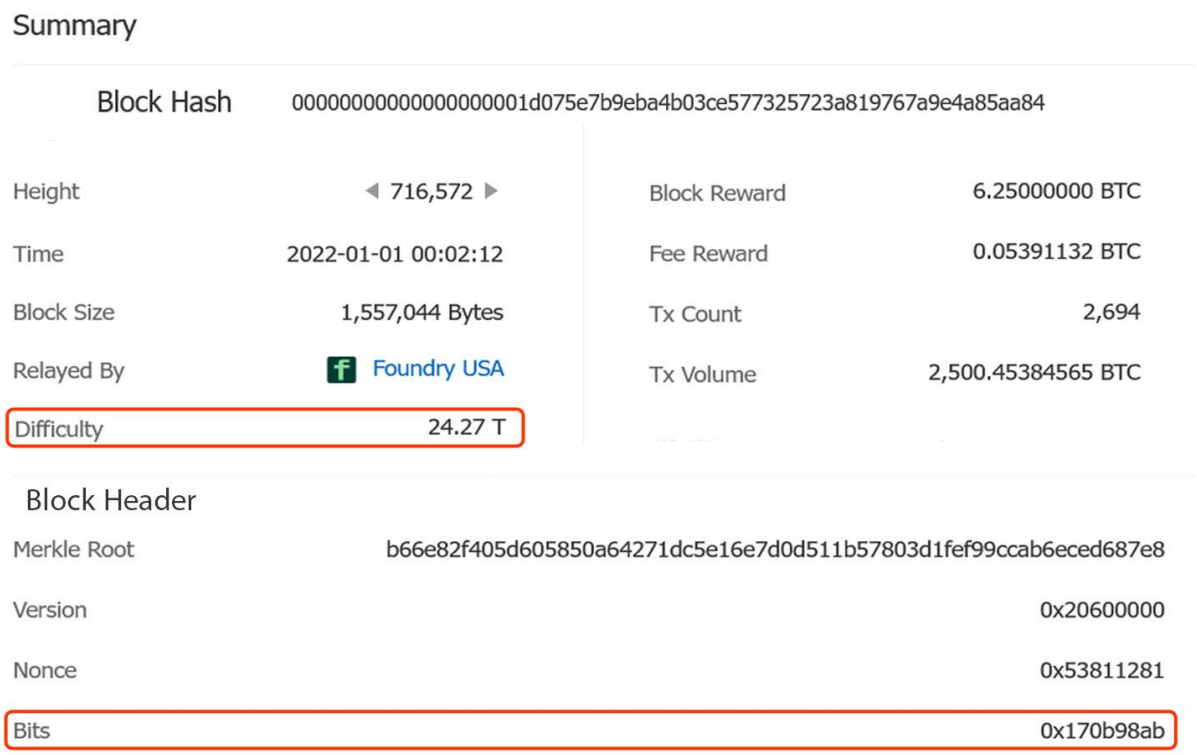


Figure 23 - summary information of first block mined in 2022.

For this block, as of January 2022, figure below shows how to decode the target "bits" and determine the mining difficulty.

Bit as hexadecimal number:

0x170b98ab

Coefficient: 0x17 **Index:** 0x0b98ab

Target as hexadecimal with padding:

0x00000000000000000b98ab0000000000000000000000000000000000000000

Target as hexadecimal:

0x0b98ab0000000000000000000000000000000000000000

Target as decimal number:

1110710552837102268873518195482888052995095958078357504

Difficulty as decimal number:

24272331996979 R 107788522606265018933066343414008033637362371316311654 4

$$\log_2\left(\frac{2^{256}}{1110710552837102268873518195482888052995095958078357504}\right) = 76.4643999715$$

Figure 24 - decode the target "bits" and determining prosses for the mining difficulty and target

So, the hash of any valid block has to be below this value. In other words, one in about 2^{76} nonces (Based on calculation shows in figure above) that you try will work, which is a really huge number. One approximation is that it's 1680-times greater than the human population of earth squared. So, if every person on earth was themselves their own planet earth with 7.8 billion people on it, the total number of people multiply by 1680, would be close to 2^{76}.

Determining the difficulty

Difficulty gets adjusted according to the hash rate in order to maintain a constant block finding time and to dispense the block reward not too fast or too slow. The mining difficulty changes every 2016 blocks, which are found about once every 2 weeks. It is adjusted based on how efficient the miners were over the period of the previous 2016 blocks according to this formula:

next_difficulty = (previous_difficulty * 2016 * 10 minutes) / (time to mine last 2016 blocks)

Note that 2016*10 minutes is exactly two weeks, so 2016 blocks would take two weeks to mine 2016 blocks if a block were created exactly every 10 minutes. So, the effect of this formula is to scale the difficulty to maintain the property that blocks should be found by the network on average about once every ten minutes. There's nothing special about 2 weeks, but it's a good trade-off. If the period were much shorter, the difficulty might fluctuate due to random variations in the number of blocks found in each period. If the period were much higher, the network's hash power might get too far out of balance with the difficulty.

You can see in Figure below that over time the mining difficulty keeps increasing. It's not necessarily a steady linear increase or an exponential increase, but it depends on activity in the market. Mining difficulty is affected by factors like how many new miners are joining, which in turn may be affected by the current exchange rate of Bitcoin. Generally, as more miners come online and mining hardware gets more efficient, blocks are found faster and the difficulty is increased so that it always takes about ten minutes to find a block.

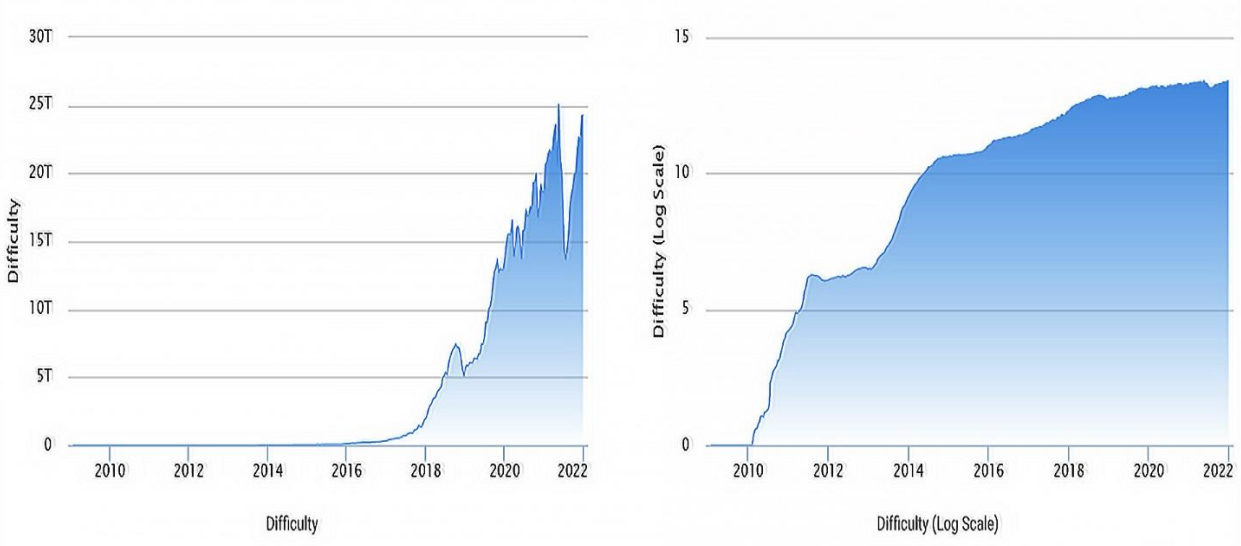


Figure 25 - Difficulty over time - left: over actual scale time - Right: over Log scale time

Each Bitcoin miner independently computes the difficulty and will only accept blocks that meet the difficulty that they computed. Miners who are on different branches might not compute the same difficulty value, but any two miners mining on top of the same block will agree on what the difficulty should be. This allows consensus to be reached.

4.2-Mining Hardware

Notice that the computation that miners have to do is very difficult. In this section, we'll discuss why it is so computationally difficult and take a look at the hardware that miners use to perform this computation.

The core of the difficult computation which miners are working on is the SHA-256 hash function. We discussed hash functions and properties of those in previous sections. SHA-256 is a general-purpose cryptographic hash function that's part of a bigger family of functions that was standardized in 2001 (SHA stands for Secure Hash Algorithm). SHA-256 was a reasonable choice as this was strongest cryptographic hash function available at the time when Bitcoin was designed. It is possible that it will become less secure over the lifetime of Bitcoin, but for now it remains secure.[11]

A closer look at SHA-256

Figure below shows more detail about what actually goes on in a SHA-256 computation. SHA-256 maintains 256 bits of state(variable). The state(variable) is split into eight 32-bit words(a-b-c-d-e-f-g-h) which makes it highly optimized for 32-bit hardware. for first 512-bit block of message these eight 32-bit words are Initialization Vectors (IV). and for subsequent blocks of message, 256 bits of the state(variable) are 256-bit hash of previous block.

At a very simple glance, for each 512-bit block of message (or for each compression function), message prepare and schedule of sixty-four 32-bit words. in each round of 64 round, a number of words in the state are taken (some with bitwise tweaks related to words of message applied) and added together mod 32. The entire state is then shifted[12] over with the result of the addition becoming the new left most word of the state (the result value placed as new eight 32-bit words of state).

this process repeated 64-time for each block of message. And the latest result of eight 32-bit words of state, forms the hash of the block.

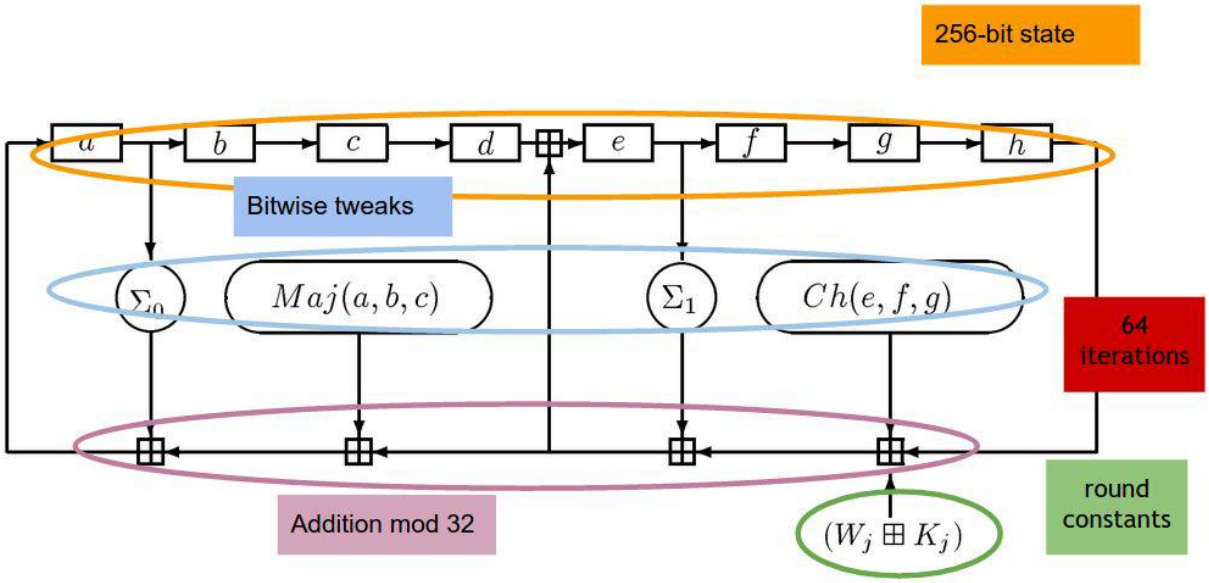


Figure 26 - The structure of SHA-256. This is one round of the compression function.

---

[11] Although the SHA-2 family, including SHA-256, are still considered to be cryptographically secure, the next generation SHA-3 family has now been picked by a contest. SHA-3 is in the final stages of standardization today, but it wasn't available at the time Bitcoin was designed.

[12] This part of design is inspired by bitwise Linear Feedback Shift Registers (LFSR).

Figure above shows just one round of the SHA-256 compression function. A complete computation of SHA-256 does this for 64 iterations. During each round, there are slightly different constants applied so that no iteration is exactly the same.
The task for miners is to compute this function as quickly as possible. Remember that miners are racing each other so the faster they do this, the more they earn. Bitcoin actually requires SHA-256 to be applied twice to a block in order to get the hash that is used by the nodes. The reasons for the double computation are not fully specified, but at this point, it's just something that miners have to deal with.

CPU mining

The first generation of mining was all done on general purpose computers that is general purpose central processing units (CPUs). In fact, CPU mining was running a pseudocode like the code shown in Figure below. That is, miners searched over nonces in a linear procedure, computed SHA 256 in software and checked if the result was a valid block.

```
TARGET = (65535 << 208) / DIFFICULTY;
coinbase_nonce = 0;
while (1) {
      header = makeBlockHeader(transactions, coinbase_nonce);
      for (header_nonce = 0; header_nonce < (1 << 32); header_nonce++){
            if (SHA256(SHA256(makeBlock(header, header_nonce))) <
      TARGET)
                  break; //block found!
      }
      coinbase_nonce++;
}
```

Figure 27 - CPU mining pseudocode.[13]

How fast will this run on a general-purpose computer? On a good desktop PC, you might expect to compute about on scale by 50 million hashes per second (MH/s).
If you're mining on a general-purpose PC today, CPU mining is no longer profitable with the current difficulty. For the last few years, anyone trying to mine on a CPU probably doesn't understand how Bitcoin works and was probably pretty disappointed that they never made any money doing it.

GPU mining

The second generation began when people started to get frustrated with how slow their CPUs were and instead used their graphics card, or graphics processing unit (GPU).
Bitcoin mining can be parallelized easily because you can try computing multiple hashes at the same time with different nonces. In 2010, a language called OpenCL was released. OpenCL is a general-purpose language to do things other than graphics on a GPU. It's a high level-language and over time people have used it to run many types of computation more quickly on graphics cards. This paved the way for Bitcoin mining on GPUs.
Mining with graphics cards had several attractive properties at the time. For one thing, they're easily available and easy for amateurs to set up. They're the most accessible high-end hardware that's available to the general public. They also have some properties that make them specifically good for Bitcoin mining. They're designed for parallelism so they have many Arithmetic Logic Units (ALUs) that

[13] *Notice in the code that as mentioned, SHA-256 is applied twice.*

can be used for simultaneous SHA-256 computations. Some GPUs also have specific instructions to do bitwise operations that are quite useful for SHA-256.
Most graphics cards can also be ***overclocked***, meaning you can run them faster than they're actually designed for if you want to take on the risk that they might overheat or malfunction. With Bitcoin mining, it might be profitable to run the chip much faster than it was designed for even if you induce a few errors by doing so. For example, consider you can run your graphics card 50 percent faster but doing so will cause errors in the SHA-256 computation to 30 percent of the time. In the above example, the throughput is 1.5x compared to not overclocking, whereas the success rate is 0.7x. The product is 1.05, which means overclocking increases your expected profits by 5%.
Finally, you can drive many graphics cards from one motherboard and CPU. So, you can take your computer, which will be running your actual Bitcoin node which gathers transactions from the network and assembles blocks, and attach multiple graphics cards to it to try to find the right nonces to make the SHA-256 of the block valid. Figure below shows a setup to drive many, many GPUs from a single CPU. This was still in the early days of Bitcoin when miners were still mostly hobbyists without much experience running servers, but they came up with some quite ingenious designs for how to pack many graphics cards into a small place and keep them cool enough to operate.

Figure 28 - A home-built rack of GPUs used for Bitcoin mining. You can also see the fans that they used to build a primitive cooling system.

On a good graphics card with aggressive tuning, you might get on scale by 500 MH/s, or 500 million hashes per second.

Disadvantages of GPU mining

GPU mining has some disadvantages. GPUs have a lot of hardware built into them for doing video processing that can't be utilized for mining. Specifically, they have a large number of floating-point units that aren't used at all in SHA-256. GPUs also don't have the greatest cooling characteristics when you put a lot of them next to one another. They're not designed to run side by side as they are in the picture; they're designed to be in a single box doing graphics for one computer.

GPUs can also draw a fairly large amount of power, so a lot of electricity is used relative to a computer. Another disadvantage initially was that you had to either build your own board or buy expensive boards to house multiple graphics cards.

FPGA mining

Around 2011 some miners started switching from GPUs to FPGAs, or Field Programmable Gate Arrays, after the first implementation of Bitcoin mining came out in Verilog, a hardware design language that's used to program FPGAs. The general rationale behind FPGAs is to try to get close as possible to the performance of custom hardware while also allowing the owner of the card to customize it or reconfigure it "in the field." By contrast, custom hardware chips are designed in a factory and do the same thing forever.

FPGAs offer better performance than graphics cards, particularly on "bit fiddling" operations which are trivial to specify on an FPGA. Cooling is also easier with FPGAs and, unlike GPUs, you can theoretically use nearly all of the transistors on the card for mining. Like with GPUs, you can pack many FPGAs together and drive them from one central unit, which is exactly what people began to do (see Figure below). Overall, it was possible to build a big array of FPGAs more neatly and cleanly than you could with graphics cards.

Figure 29 - A home-built rack of FPGAs

Using an FPGA with a careful implementation, you might get up to several GH/s billion hashes per second. This is certainly a large performance gain over CPUs and GPUs.

Despite the performance gain, the days of FPGA mining were quite limited. Firstly, they were being driven harder for Bitcoin mining — by being on all the time and overclocked — than consumer grade FPGAs were really designed for. Because of this, many people saw errors and malfunctions in their FPGAs as they were mining. It also turned out to be difficult to optimize the 32-bit addition step which is critical in doing SHA-256. Secondly, FPGAs are also less accessible-you can't buy them at most stores and there are fewer people who know how to program and set up an FPGA than a GPU.

Most importantly though, even though FPGAs improved performance the cost-per-performance was only marginally improved over GPUs. This made FPGA mining was a rather short-lived phenomenon. Whereas GPU mining dominated for about a year.

ASIC[14] mining

Mining today is dominated by Bitcoin ASICs, or **application-specific integrated circuits**. These are chips that were designed, built, and optimized for the sole purpose of mining Bitcoins. There are a few big vendors that sell these to consumers with a good deal of variety: you can choose between slightly bigger and more expensive models, more compact models, as well as models with varying performance and energy consumption claims.

The performance of today's miners is about above of 1,000GH/s one trillion hashes per second. This is certainly a large performance gain over CPUs and GPUs and FPGAs.

Designing ASICs requires considerable expertise and their lead-time is also quite long. Nevertheless, Bitcoin ASICs were designed and produced surprisingly quickly. In fact, analysts have said that this may be the fastest turnaround time in the history of integrated circuits from specifying a problem and to have a working chip in people's hands. Partially as a result of this, the first few generations of Bitcoin ASICs were quite buggy and most of them didn't quite deliver the promised performance numbers. Bitcoin ASICs have since matured and there are now fairly reliable ASICs available.

Up until 2014, the lifetime of ASICs was quite short due to the rapidly increasing network hash rate, with most boards in the early ASIC era growing obsolete in about six months. Within this time, the bulk of the profits are made up front. Often, miners will make half of the expected profits for the lifetime of the ASIC during just the first six weeks. This meant, even shipping speed can become a crucial factor in making a profit. As the growth rate of Bitcoin's hash power has stabilized, mining equipment has a longer life time.

Figure 30 - A commercial ASIC bitcoin miner

[14] ASIC stands for application specific integrated circuit, which is a specialized silicon chip that performs just one task. In the digital currency space, these chips process SHA-256 in order to mine bitcoin and validate transactions.

5-FPGA

5.1-Introduction

Field programmable gate array (FPGA) is a type of integrated circuit (IC) that can be programmed for different algorithms after fabrication. FPGA first introduced in 1985. Modern FPGA devices consist of up to four million logic cells that can be configured to implement a variety of software algorithms. an FPGA provides significant cost advantages in comparison to an IC development effort and offers the same level of performance in most cases. FPGA is developed version of PLD[15]. In PLD'S programmed after manufacturing in field, it has limited programmability. Field programmable gate is capable of implementing any digital circuit. This provides developer of creating wide array of logical structure minimum low cost.

Figure 31 - A sample FPGA - Xilinx spartan series

Requirement of FPGAs

By the early 1980's large scale integrated circuits (LSI) formed the back bone of most of the logic circuits in major systems. Microprocessors, bus/IO controllers, system timers etc. were implemented using integrated circuit fabrication technology. custom "glue logic"[16] or interconnects were still required to help connect the large integrated circuits in order to:

- Generate global control signals (for resets etc.)
- Data signals from one subsystem to another sub system.

First attempt to this problem was developing custom ICs, but it was not a solution for the problem. custom ICs have their own disadvantages. They are relatively very expensive to develop, and delay introduced for product to market (time to market) because of increased design time. There are two kinds of costs involved in development of custom ICs

1. Cost of development and design
2. Cost of manufacture

---

*[15] PLDs are array-oriented devices that typically have an AND-OR structure with wide-input AND gates feeding a narrower OR gate. The difference between FPGA and PLD is that **FPGA incorporates logic blocks instead of fixed AND-OR gates**. FPGAs are designed for having higher gate count whereas, PLDs are used for lesser gate counts.*

[16] Glue logic is a special form of digital circuitry that allows different types of logic chips or circuits to work together by acting as an interface between them.

FPGAs were introduced as an alternative to custom ICs for implementing entire system on one chip and to provide flexibility of reporogramability to the user. One of the major advantages of FPGA over custom ICs is circuit implementation time is very less because physical layout, masking etc. is absent in this process. Circuit implementation is done with help of the advanced CAD[17] tools.

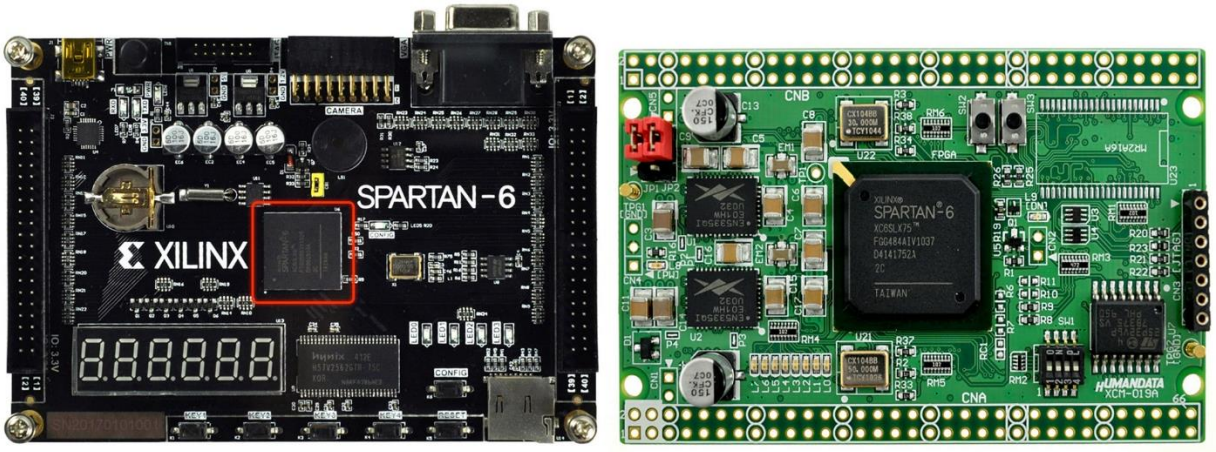


Figure 32 – FPGA Development Boards -Xilinx Spartan 6

[17] *Computer-aided design (CAD) is the use of computers to aid in the creation, modification, analysis, or optimization of a design.*

5.2-Architecture and Structure of FPGA

FPGA is a device that contains a matrix of reconfigurable gate array logic circuitry. When a FPGA is configured, the internal circuitry is connected in a way that creates a hardware implementation of the software application. Unlike processors, FPGAs use dedicated hardware for processing logic and do not have an operating system. FPGAs are truly parallel in nature so different processing operations do not have to compete for the same resources. As a result, the performance of one part of the application is not affected when additional processing is added. Unlike ASICs and hard-wired printed circuit board (PCB) designs which have fixed hardware resources, FPGA-based systems can literally rewire their internal circuitry to allow reconfiguration after the control system is deployed to the field.

A single FPGA can replace thousands of discrete components by incorporating millions of logic gates in a single integrated circuit (IC) chip. The internal resources of an FPGA chip consist of a matrix of configurable logic blocks (CLBs) surrounded by a periphery of I/O blocks. Signals are routed within the FPGA matrix by programmable interconnect switches and wire routes.

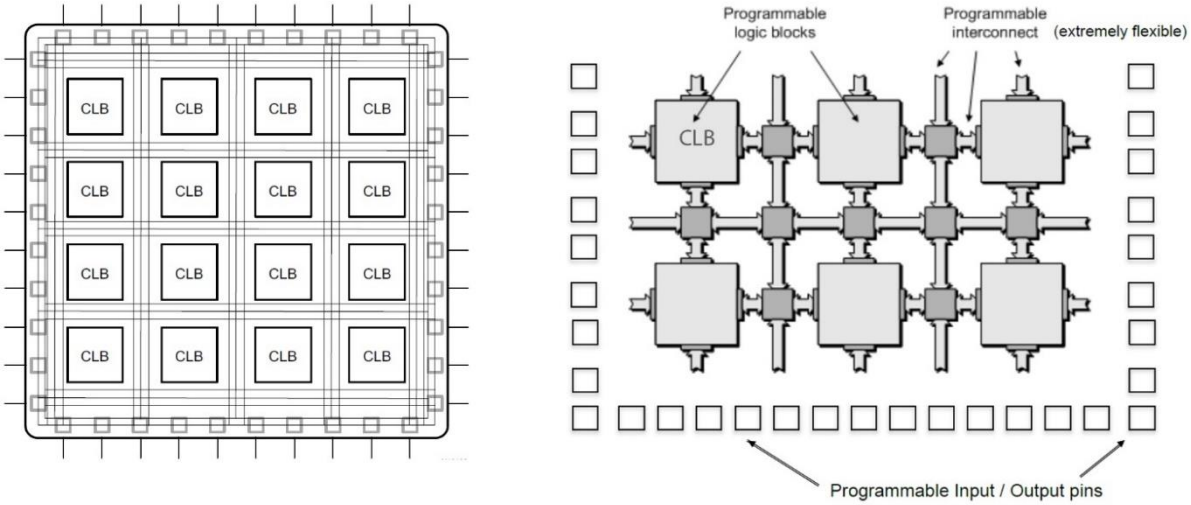


Figure 33 - Basic FPGA Architecture

The basic structure of an FPGA is composed of the following elements:

- **Look-up table (LUT):** This element performs logic operations.
- **Flip-Flop (FF):** This register element stores the result of the LUT.
- **Multiplexer:** a device that can receive multiple input signals and synthesize a single output signal.
- **Input/Output (I/O) pads:** These physically available ports get data in and out of the FPGA.

The combination of these elements results in the basic FPGA architecture shown in figure below.

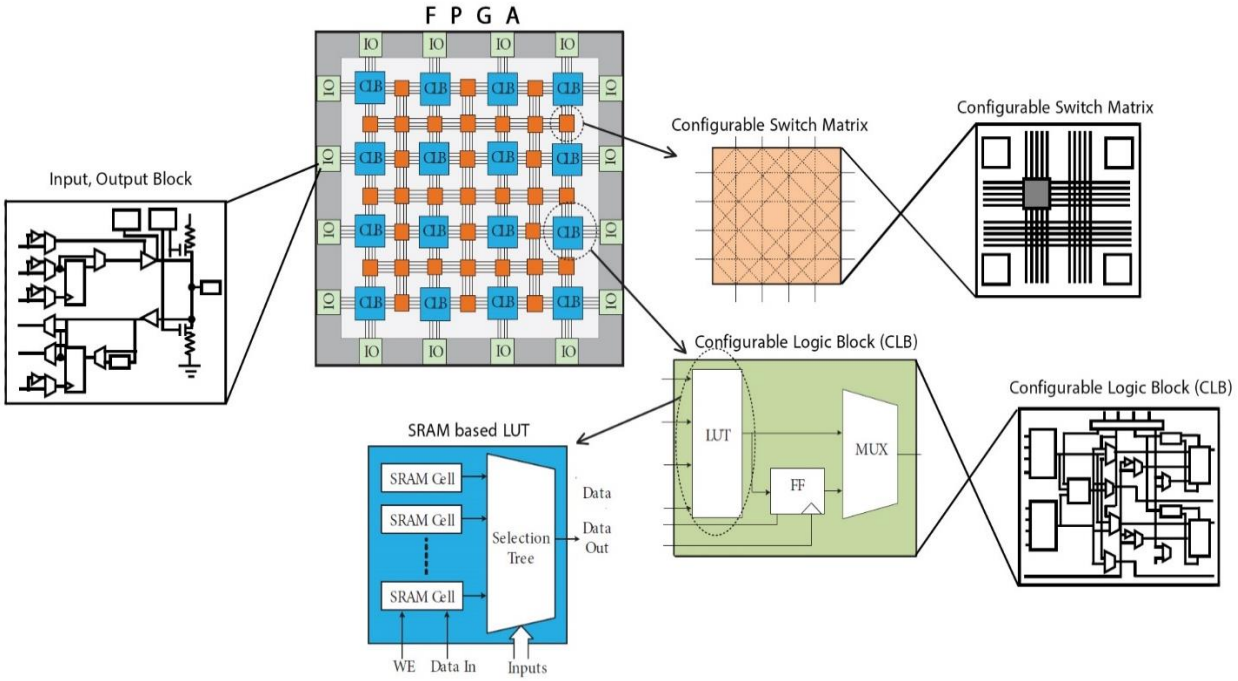


Figure 34 - FPGA Architecture

Internal Architecture of LUT

A combinational logic is made up of basic logic gate connected through wire. In FPGA these gates are simulated using look-up table (LUT). A n-input LUT requires 2^n bits of SRAM to store lookup table and $2^n - 1$ multiplexer to read individual bit. In general, 4-input look-up table is used but nowadays 6 input look-up table is used.

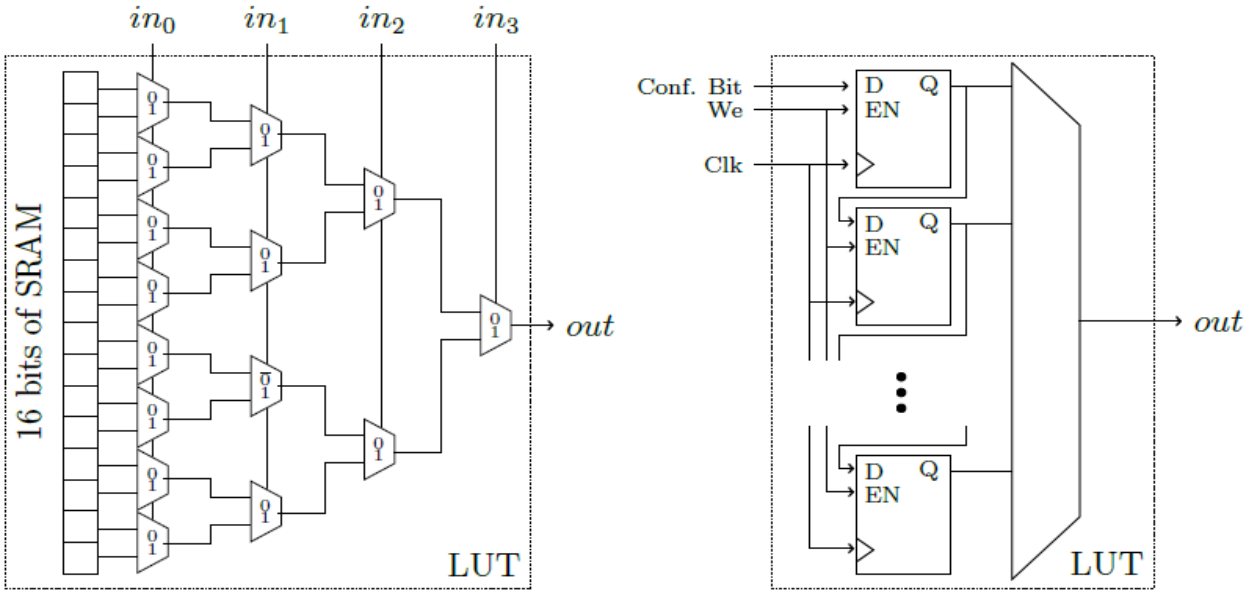


Figure 35 - Four input LUT - (left) circuitry for read - (right) circuitry for write.

The value in0 to in3 are used to determine which SRAM bit is given at output. Boolean function is stored in SRAM cell. SRAM cell is simply a shift register with one-bit width and 2^n bit depth. The Bit is shifted bit-by-bit into LUT when FPGA is programmed. LUT can also be used as memory element on

FPGA. When FPGA is programmed LUT is used as distributed RAM. The multiple LUTs are combined to make wider or deeper memories.

Routing Architecture - switch matrix

Interconnect is used for communicate between different logic island (LIs) these interconnects are configurable. It consists of horizontal and vertical channels (bundle of wire). At interconnect of routing channel there is a programmable links which determine how wire is connected, how input output is routed in particular logic island. All wires are connected to additional three wires at interconnection point but which connection is active is determine by programmable switch.

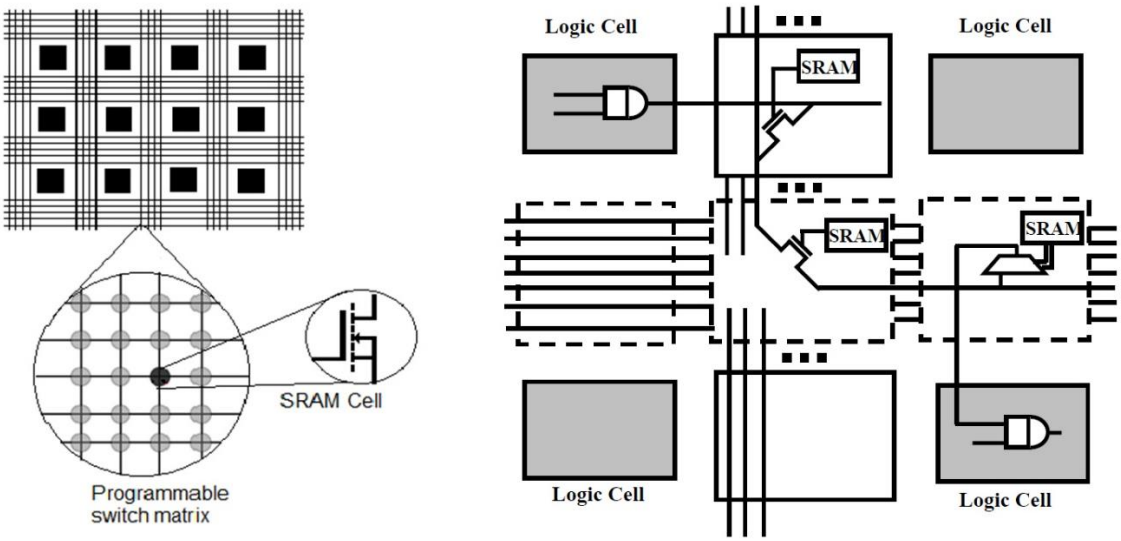


Figure 36 - (left) Routing architecture with switch matrix - (right) SRAM-controlled Programmable Switches

figure above, shows two applications of SRAM cells: for controlling the gate nodes of pass-transistor switches and to control the select lines of multiplexers that drive logic block inputs. The figure(right) gives an example of the connection of one logic block (represented by the AND-gate in the upper left corner) to another through two pass-transistor switches, and then a multiplexer, all controlled by SRAM cells.

Additional Elements In Contemporary FPGA Architectures

Contemporary FPGA architectures incorporate the basic elements along with additional computational and data storage blocks that increase the computational density and efficiency of the device. These additional elements are:

- Embedded memories for distributed data storage
- Phase-locked loops (PLLs) for driving the FPGA fabric at different clock rates
- High-speed serial transceivers
- Off-chip memory controllers
- Multiply-accumulate blocks (DSP Blocks[18])

[18] *The most complex computational block available in a common FPGA is the DSP block which is an arithmetic logic unit (ALU).*

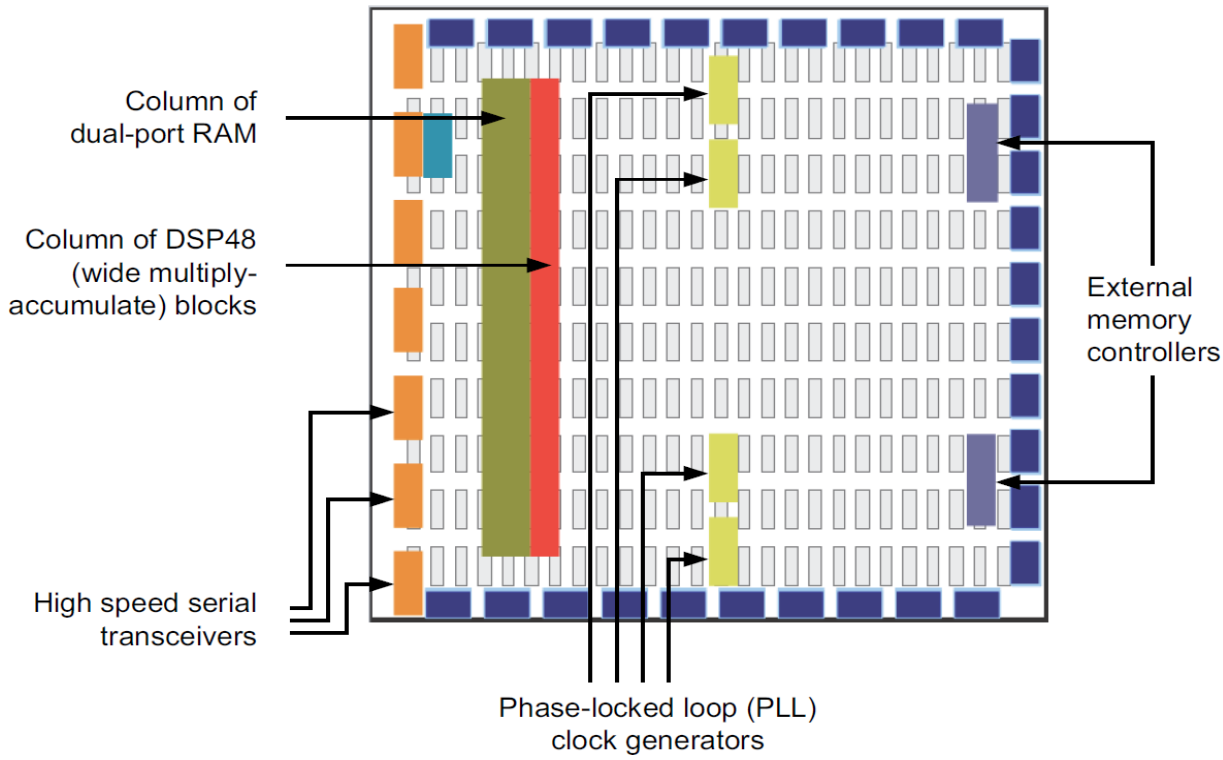


Figure 37 - Contemporary FPGA Architecture

The combination of these elements provides the FPGA with the flexibility to implement any software algorithm running on a processor.

5.3-FPGA Programming

The most important steps and tools of the design flow to produce an FPGA-circuit are depicted in figure below.

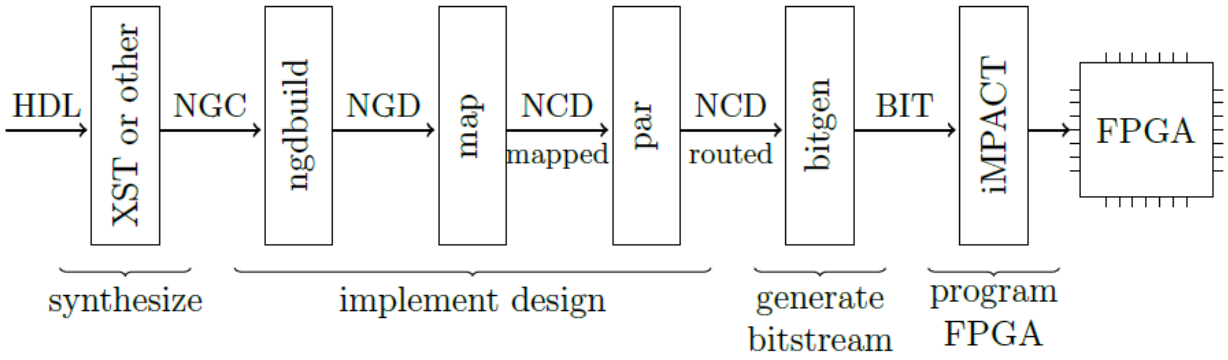


Figure 38 - FPGA design flow.

HDL & Synthesize

Programming on FPGA is same as describe the structure and behavior of electronic circuits and connecting wires in circuit, this is done by using a hardware description language (HDL) such as **VHDL** or **Verilog**. Synthesizer converts HDL into gate level netlist. A gate level netlist is basically your fitted design, before its converter to a programming file. It contains all of the logic and delays of the final system. It allows you to use your testbench from the simulation testing to test the final design. So, synthesis process generates netlist for each design element. The resulting netlists is saved to an ***NGC (Native Generic Circuit)*** file[19].

Translate process

Translate process combines all the input netlists and constraints to a logic design file. This information is saved as a ***NGD (Native Generic Database)*** file. This can be done using NGD Build program. Here, defining manufacturing constraints is assigning the ports in the design to the physical elements (e.g., I/O pins, clock, switches, buttons etc.) Of the targeted device and specifying time requirements of the design. This information is stored in a file named ***UCF (User Constraints File).*** Generally, NGC allow behavioral simulation and NGD allow timing simulation.

Map process

MAP process divides the whole circuit with logical elements into sub blocks such that they can be fit into the FPGA logic blocks. That means map process fits the logic defined by the NGD file into the targeted FPGA elements (configurable Logic Blocks (CLB), Input Output Blocks (IOB)) and generates an ***NCD (Native Circuit Description)*** file which physically represents the design mapped to the components of FPGA.

Place and Route

PAR program is used for this process. The place and route process places the sub blocks from the map process into logic blocks according to the constraints and connects the logic blocks. A tradeoff between all the constraints is taken account by the place and route process. The PAR tool takes the

---

[19] *For Xilinx Synthesis Technology (XST)*

mapped NCD file as input and produces a completely routed NCD file as output. Output NCD file consists of the routing information.

Device Programming - Bitstream Generation

Now the design must be loaded on the FPGA. But the design must be converted to a format so that the FPGA can accept it. BITGEN program deals with the conversion. it encodes design in binary known as **Bitstream *(a .BIT file)***. The routed NCD file is given to the BITGEN program to generate a bitstream which can be used to configure the target FPGA device. Inside FPGA a finite state machine control by Bitstream which extract configuration data from Bitstream. This can be done using a cable.

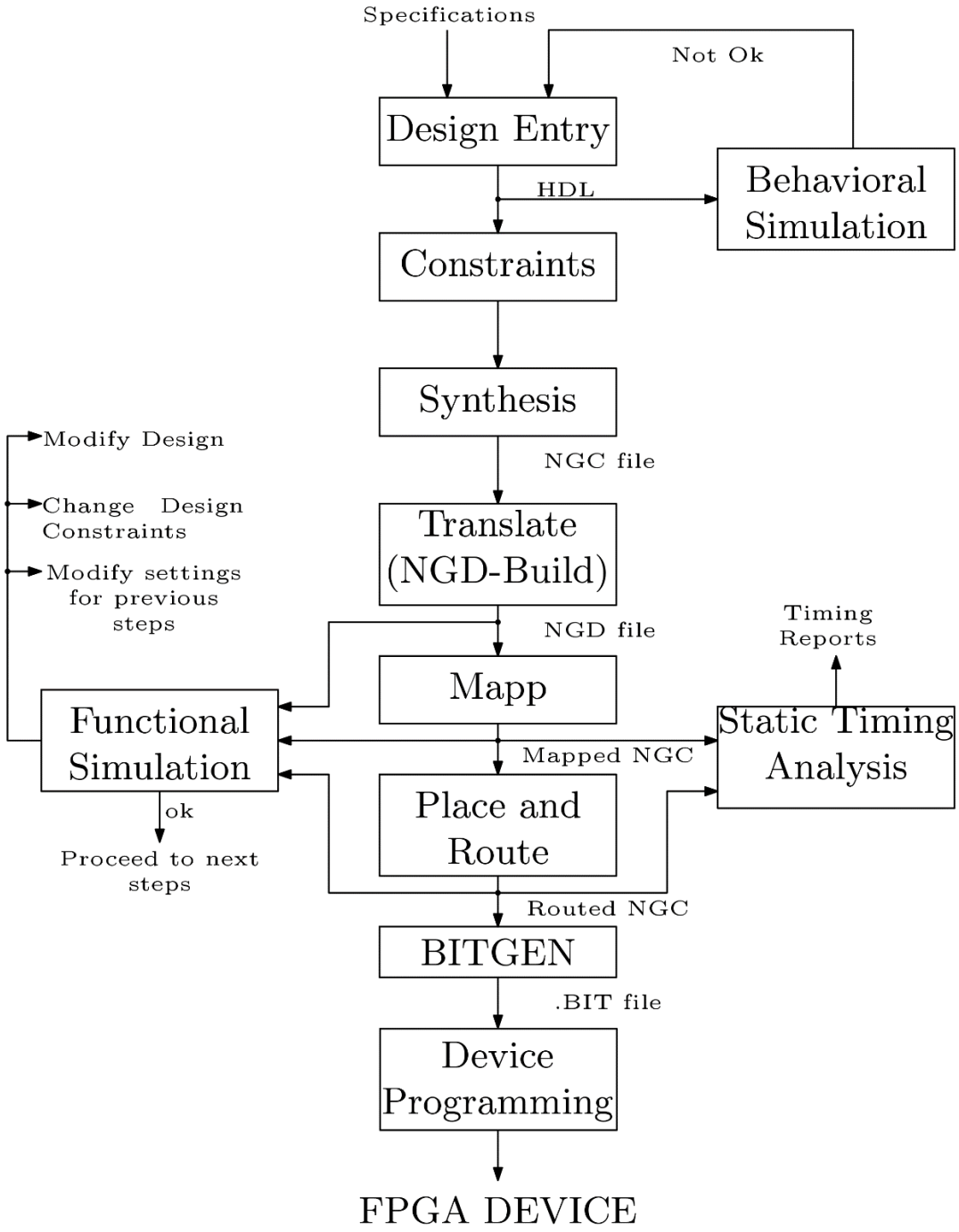


Figure 39 - FPGA implementation - Step by Step

5.4-FPGAs Vs ASICs

ASIC microchip is designed and manufactured for one specific application and does not allow you to reprogram or modify it after it is produced. This means ASICs are not intended for general use.
For chose technology to design or develop a chip, there are several factors to consider:

- Design Flow
- Flexibility
- Performance and Efficiency
- Cost and Time to Market
- Power Consumption

The main properties and applications of ASICs and FPGAs are listed below:

FPGAs

- Off-the-shelf, that is available immediately and does not need to be specially made to suit a particular purpose.
- Reconfigurability, may be reprogrammed in the field (gateware upgrade) – (New features - Bug fixes)
- Rapid development cycle (minutes / hours) - Short time to market
- Low development cost

ASICs

- Higher performance - Low power
- Analog designs possible
- Long development cycle (weeks /months)
- Design cannot be changed once it is produced
- Better radiation hardness
- Extremely high development cost (ASICs are produced at a semiconductor fabrication facility ("fab") according to your design)
- Lower cost per device compared to FPGA, when large quantities are needed

	ASIC	FPGA
Design Flow	Complex	Simplistic
Flexibility		✓
Performance	✓	
Development Cost (NRE)		✓
Production Unit Cost	✓	
Power Consumption	✓	

Figure 40 - FPGAs Vs ASICs

5.5-FPGA As Bitcoin Miner

FPGAs have advantages in bitcoin mining due to their lower power usage and higher levels of hardware customization. As you can recall, from previous sections, mining is all about hashing a candidate block and seeing if the hash meets certain difficulty criteria. When bitcoin was first introduced, central processing units (CPUs) from Intel and AMD were used as miners, but they were quickly replaced by graphics processing units (GPUs) from Nvidia and AMD. CPUs have relatively few arithmetic logic units (ALUs) and are designed to run more general executive and decision-making software. GPUs have the ability to perform lots of repetitive work because they contain large numbers of ALUs designed to increase their ability to calculate the mathematical formulas to drive pixels on a screen. These same ALUs can be repurposed to repeatedly try different hashes, and the number of ALUs has a direct effect on the hash output. However, both the CPU and GPU are designed for multipurpose, universal computing. This means that both can execute any valid code; in computing terms this is called "Turing-completeness". On the other hand, FPGAs are programmed for a specific purpose. FPGAs can be configured to compute the SHA-256 algorithm with even more efficiency since their hardware is developed for this task. This is the reason why FPGAs are faster than the best GPUs at performing hashing calculations.

FPGA can be connected to a computer with USB cable (for many FPGAs, USB-HUB may be needed). When mining with an FPGA, the mining software creates the necessary work to bitcoin system and connect to network. The software primarily retrieves work from the bitcoin network and sends it to the FPGA miner. the FPGA hashes away to find a possible solution. The mining software monitors all this work to make sure that a valid solution taken. when a valid solution found, the mining software reports to the bitcoin network.

FPGA mining versus GPU and CPU and ASIC mining

As you can see, from a comparison shows in figure below, FPGAs are faster at performing hashing calculations than both CPUs and GPUs. They are also more efficient as measured by the use of electricity per hashing unit. The increase in hashing speed in FPGAs is a significant improvement over GPUs and even more so over CPUs. But ASICs makes the game changing. and other older generations of hardware mining (CPUs, GPUs, FPGAs) can no longer compete with ASICs.

Hardware	Mining speed (MH/s)	Power used (Watts)
CPU: Core i7-3930K	66 - 98	170 - 200
GPU: NVIDIA Tesla S2070	750	900
FPGA: Butterflylabs Mini Rig	25,200	1250
ASIC: Bitmain Antminer S9j	14,500,000	1500

Figure 41 - Comparison of typical examples of hardware mining generations. Mentioned Mining speed is at the high performance of hardware, and each of examples are a good one for mining in their type.

One of the biggest reasons CPUs and GPUs are no longer used as miners is because the electricity to run them often cost more than the amount of bitcoins received from mining. ASICs offer the best performance per watt.

Figure 42 - ButterflyLabsMiniRig miner 25GH/s - A commercial high performance package device contains several FPGAs.

Profitability of FPGA mining

As we have mentioned earlier, the Bitcoin network hash rate is really high now, and even mining with FPGAs and some ASICs does not guarantee profits. This is due to the fact that during the mining process you are competing with other miners to try to solve a block. If those other miners are running a larger percentage of the total mining power, you will be at a disadvantage, as they are more likely to solve a block.

6-CONCLUSION

Bitcoin is a new Internet currency that anyone can get started mining. There are a number of reasons you might mine: for profit, to help secure the network, to help found a new Internet currency, or just to gain technical experience.

Professional mining

Today mining has mostly moved away from individuals and toward professional mining centers. Exact details about how these centers operate are not very well known because companies want to protect their setups to maintain a competitive advantage. Presumably, these operations maintain profitability by buying slightly newer and more efficient ASICs than are available for general sale at a bulk discount. Figure below shows a picture of a professional mining center.

Figure 43 - BitFury mining center, a professional mining center in the republic of Georgia

Mining hardware evolution: Similarities to gold mining

Currently, ASIC mining is the only realistic means to be profitable in Bitcoin and it's not very friendly to small miners. Whereas with Bitcoin mining we've seen an evolution from CPUs to GPUs to FPGAs, to now ASICs, gold mining saw an evolution from individuals with gold pans to small groups of people with sluice boxes, to placer mining[20] to modern gold mining which often utilizes gigantic open pit mines to extract tons of raw material from the earth (See Figure below).

[20] *Consisting of large mining groups blasting away hillsides with water*

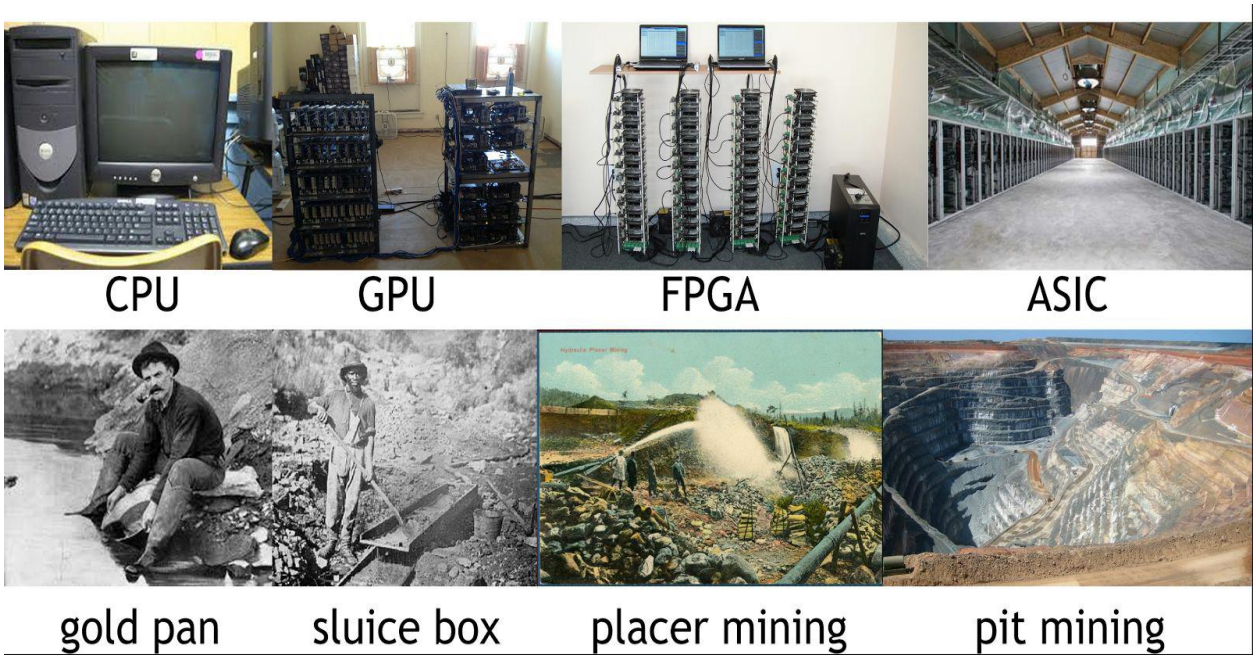


Figure 44 - Evolution of mining. We can see a clear parallel between the evolution of Bitcoin mining and the evolution of gold mining. Both were initially friendly to individuals and over time became massive operations controlled by large companies.

Both with Bitcoin and with gold, the friendliness and accessibility to individuals has gone down over time and large companies have eventually consolidated most of the operations (and profits).

A look to the future - The cycle repeats itself

today, there are several smaller Altcoins[21] have indeed used a different puzzle than SHA-256, but have seen a similar trajectory in mining(hardware) as Bitcoin. recall that for ASICs there is still a long lead time between designing a chip and shipping it, so if a new altcoin uses a new puzzle (even just a modified version of SHA-256), this will buy some time in which ASICs are not yet available. Typically, mining will proceed just at Bitcoin did from CPUs to GPUs and/or FPGAs to ASICs (if the altcoin will be very successful, like Litecoin[22]).

Thus, one strategy for smaller miners may be to try to pioneer new altcoins which aren't yet valuable enough for large mining groups to invest in.[23] in this case, FPGA technology may be the smart choice for development. So, FPGAs may still have a place in the future of cryptocurrency as a platform to test develop new mining algorithms. The biggest advantage of FPGAs in cryptocurrency mining is that they are not limited to one currency such as bitcoin. Other competing cryptocurrencies such as Litecoin could be mined with an FPGA because it could be reprogramed to run Scrypt[24] instead of SHA-256.

---

[21] Altcoins are alternatives to the cryptocurrency king, Bitcoin (BTC). These coins differentiate themselves from Bitcoin by extending their capabilities and plugging their shortcomings.

[22] Litecoin is a peer-to-peer cryptocurrency and open-source software project. it was an early bitcoin spinoff or altcoin, starting in 2011. In technical details, Litecoin is nearly identical to Bitcoin but uses Scrypt as cryptographic algorithm.

[23] just like small gold miners who have been driven out of proven goldfields might try prospecting unproven new areas. Of course, this means the pioneers are facing a significant risk that the altcoin will never succeed.

[24] Scrypt – cryptographic algorithm used in Litecoin (instead of SHA-256 used in btc)

Appendices

A1-the Birthday paradox problem

The Birthday Paradox is a famous problem in probability theory that reveals something surprising: in a group of just 23 people, there is a more than 50% chance that two people share the same birthday. At first, this seems unlikely and Most people expect the number of people needed for a likely match to be much higher.

The math behind the Birthday paradox

Let's define:

- N = 365 (number of possible birthdays)
- k = number of people

We want to find the smallest k such that the probability of at least one collision (shared birthday) is greater than 50%. To understand it intuitively, consider this:
Step1: calculate the probability of **no one sharing a birthday** (i.e., all birthdays are unique) is easier. and then subtract that from 1 gives the probability of at least one shared birthday.

$$P(\text{at least one match}) = 1 - P(\text{no matches})$$

$$P(\text{no match}) = \frac{365}{365} \times \frac{364}{365} \times \frac{363}{365} \times \cdots \times \frac{365 - k + 1}{365}$$

This simplifies to:

$$P(\text{no match}) = \prod_{i=0}^{k-1} \left(1 - \frac{i}{365}\right)$$

Figure 45 - calculate the probability of no one sharing a birthday.

For probability of no one sharing a birthday, the first person can have any birthday (365/365). The second person then must have a different birthday (364/365), the third must avoid the first two (363/365), and so on.
Step2: finding k in the way that satisfy our desire to probability of 50%. When you multiply these decreasing probabilities for 23 people, the result is less than 50%. That means the chance of at least one shared birthday is greater than 50%.

K (people)	P (no match)	P (at least one match)
19	0.592	0.408
20	0.588	0.412
21	0.556	0.444
22	0.524	0.476
23	**0.493**	**0.507**
24	0.461	0.539

Figure 46 – calculation results of probability of no one sharing a birthday (middle column) and probability of at least one collision (right column) for some examples of K values (number of peoples). K=23 is the minimum number of peoples that is needed to satisfy the probability of at least one collision > 50%

This counterintuitive result is why it's called a paradox which implies our instincts about probability don't always match the math.

The Birthday attack

Now, let's connect this to cryptography, where the *"Birthday Paradox"* inspires a technique known as *"the Birthday Attack"*. In cryptography, Hash functions are designed to avoid collisions, which occur when two different inputs produce the same hash. But because of Birthday Paradox, we know that finding such collisions may require fewer attempts than expected, roughly estimated about $\sqrt{N}$ tries[25], where N is the number of possible hash outputs. This is called the birthday bound. Use of a n-bit hash function gives n/2-bit collision resistance.

A Birthday Attack exploits the mathematical insight, which we get from birthday paradox to find collisions in hash functions faster than brute-force methods. Having the same birthday is the analogue of a "collision" in a hash function. If a hacker wants to create two different messages that produce the same hash (for example, one innocent-looking contract and one malicious version), they can use a birthday attack to find such a pair. Once found, they can show the safe message for verification and then secretly switch to the harmful one-because the hash is the same, no one suspects the switch. This kind of attack is particularly dangerous for digital signatures and systems that rely on hash integrity for verification. To defend against such attacks, modern cryptographic systems use strong hash functions (like SHA-256) with large output sizes, making it impractical to find collisions even with the birthday attack.

---

[25] This is an approximation method. For accurate calculation replace N=365 by N=number of possible hash outputs (for instance SHA-256: $N = 2^{256}$) and do the calculation explained before. Because the exact formula is mathematically heavy for very large number of hash outputs, $\sqrt{N}$ method is the common approximation used in cryptographic realm, that gives fast estimation for the value of K (i.e. how many hash outputs we need to generate before there's a good chance of a collision, and so how much the crypto-system is collision resistant in comparison.)

References

1. S. Nakamoto. "Bitcoin A Peer-to-Peer Electronic Cash System". November 2008
2. A. Narayanan, J. Bonneau, E. Felten, A. Miller, and S. Goldfeder. "Bitcoin And Cryptocurrency Technologies". Princeton and Oxford. Princeton University Press. 2016
3. A. M. Antonopoulos. "Mastering Bitcoin-Programming the Open Blockchain". USA. O'Reilly Media.2017
4. Federal Information Processing Standards Publication. "FIPS PUB 180-4: Secure Hash Standard (SHS)". USA. National Institute of Standards and Technology. 2015
5. Federal Information Processing Standards Publication. "FIPS PUB 186-4: Digital Signature Standard (DSS)". USA. National Institute of Standards and Technology. 2013
6. R. C. Merkle. "One Way Hash Functions and DES". Xerox PARC-Palo Alto. Springer-Verlag. 1998
7. A. Szmigielski. "Bitcoin Essentials". UK. Packt Publishing. 2016
8. H. Handschuh, and H. Gilbert. "Security Level of Cryptography - SHA-256". FR. Issy-les-Moulineaux. 2002
9. S. Oliveira, F. Soares, G. Flach, M. Johann, and R. Reis. "Building a Bitcoin Miner on an FPGA". Universidade Federal do Rio Grande do Sul. BR
10. P. Dotemoto. "FPGA Based Bitcoin Mining". California Polytechnic State University. USA. 2014
11. Xilinx Inc. "Introduction to FPGA Design with Vivado High-Level Synthesis". Xilinx Inc. 2019
12. Version 2 EE IIT, Kharagpur. "Module4-lesson20 - Design of Embedded Processors-FPGA"
13. G. Krishna. and S. Roy. "Fundamentals of FPGA Architecture". Technical and Scientific Publisher. 2017
14. V. Jamshidi." NVRH-LUT: A nonvolatile radiation-hardened hybrid MTJ/CMOS-basedlook-up table for ultralow power and highly reliable FPGA designs". Turk J Elec Eng & Comp Sci. 2019
15. D. Johnson, A. Menezes, and S. Vanstone. "The Elliptic Curve Digital Signature Algorithm (ECDSA)". University of Waterloo. CAN. 2001
16. P. Rogaway, and T. Shrimpton. " Cryptographic Hash-Function Basics: Definitions, Implications, and Separations for Preimage Resistance, Second-Preimage Resistance, and Collision Resistance". Springer-Verlag. 2004
17. D. R. Sterry. "Introduction to Bitcoin Mining A Guide For Gamers, Geeks, and Everyone Else". 2012
18. B. Hu, Z. Zhang, and J. Liu. "A comprehensive survey on smart contract construction and execution: paradigms, tools, and systems". Patterns 2. 2021
19. Y. Shahsavari, K. Zhang, and C. Talhi. "A Theoretical Model for Fork Analysis in the Bitcoin Network". University of Quebec. 2019
20. D. R. L. Brown. "Standards for Efficient Cryptography: SEC 2: Recommended Elliptic Curve Domain Parameters". Certicom Corp. 2010
21. H. Sakulin. " Introduction to Field Programmable Gate Arrays". 8th International School for Trigger and Data Acquisition Amsterdam. NL. 2017

Courses

1. A. Narayanan, J. Bonneau, E. Felten, A. Miller, and S. Goldfeder. "Bitcoin And Cryptocurrency Technologies". Princeton University: www.coursera.org
2. E. Demaine, S. Devadas, and N. Lynch. "*6.046J Design and Analysis of Algorithms".* Massachusetts Institute of Technology: MIT. 2015. OpenCourseWare: https://ocw.mit.edu
3. G. Gensler. "*15.S12 Blockchain and Money".* Massachusetts Institute of Technology: MIT. 2018. OpenCourseWare: https://ocw.mit.edu