Litecoin mining project

0811539 馮迺茵

概念介紹：

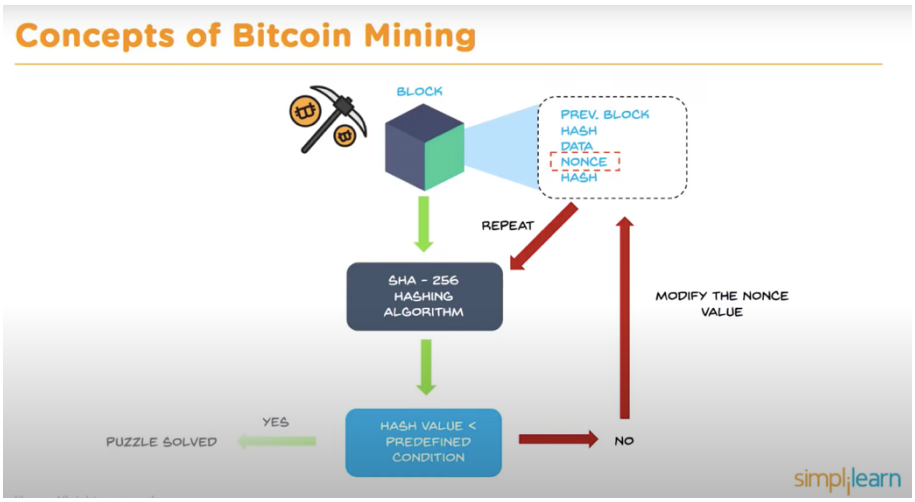
因為比特幣資料較多，而萊特幣的概念基本上跟比特幣相同，差異在於萊特幣使用 Scrypt 演算法，而比特幣則是使用 SHA-256。因此概念介紹會以比特幣為主，演算法部分再切換至本次 project 的主題萊特幣。

由於每個區塊中 Block Header 會包含許多固定的值，其中只有 Nonce 值為一隨機值，因此每個節點進行 POW（proof of work）計算時要算的就是，藉由不斷替換這個 Nonce 值，來讓這個區塊的 Block Header Hash 值，小於一個被設定好的難度目標值（Difficulty Target）。難度值（Difficulty）是指，節點要運算出低於困難度目標值的 Hash 值，平均需花多久時間，也就是平均要完成一次 POW 的時間。而比特幣區塊鏈目前設定為，大約每 10 分鐘會有節點成功算出新的區塊，不過這 10 分鐘只是基於理論值，實際每個新區塊產生的時間會有極大差異，有可能只需要 17 秒（第 407062 個區塊的實際產生時間），也有可能需要 20 分鐘以上（第 407068 個區塊的實際產生時間）。

使用比特幣或萊特幣交易的安全來自於區塊鏈，一個 block 包含了 prev. hash, hash, data, nonce，表示前後 block 互相有著關聯，假設駭客嘗試駭進其中一個 block 並改變其 data，將會造成其 hash value 被改變，如此一來便會與下一個 block 的 prev. hash 不一樣，導致接下來的區塊鏈失效。

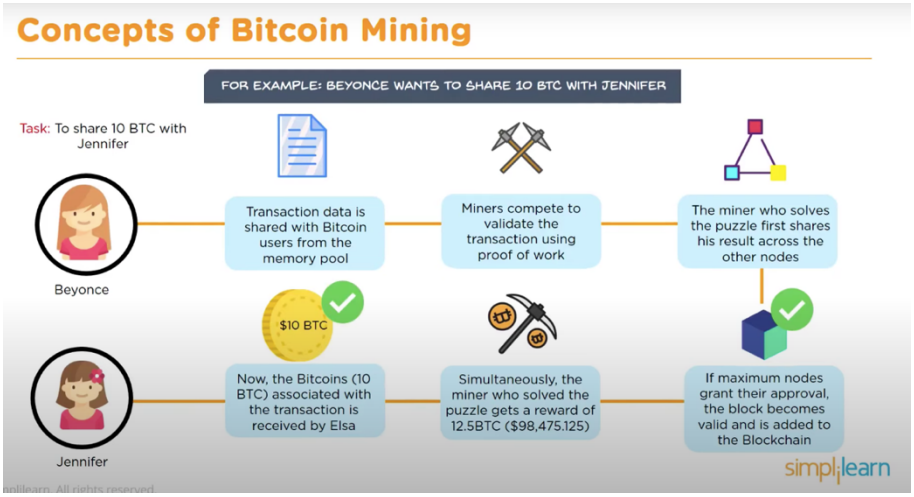
至於交易驗證的公正性來自於 Difficulty 可動態調整，目前每產生 2016 個區塊會調整一次難度，以每 10 分鐘產生一區塊估算，大約是每兩周會調整一次 Difficulty。 由於 POW 具有一定的難度，因此無法預期哪個運算節點可以最快算出新區塊。

比特幣區塊鏈採用 Hashcash 演算法來進行工作量證明，Hashcash 可將任意長度的資料經由 Hash 函數轉換為一組固定長度的代碼，原理是基於一種密碼學上的單向雜湊函數 (One Way Hash Function)，這種函數很容易被驗證，但是卻很難破解，還回推出原本的值。先前 Hashcash 演算法也被用來做阻擋垃圾郵件的機制。SHA-1 的 Hash 值長度有 160 位元，雖比 MD5 好但仍然不夠安全，因此美國國家安全局（NSA）又提出多種更複雜的 SHA-2 演算法，包括 224、256、384、512 位元長度的 Hash 值算法。比特幣區塊鏈採用 Hashcash 來建立一套幾乎無法被竄改的電子現金系統，每個區塊的 Block Header 都會被 Hash 成一串很難被回推的代碼後，放進下一個區塊中，來確保區塊的正確性。



挖礦簡易流程

礦工即是代表不斷在嘗試找出 Nonce 的人們，透過不斷改變 nonce 以達到 hash value 小於 difficulty，因為需要大量 computing 所以耗費相當大的能源，挖礦資源也幾乎被掌握在專業礦機上，如此一來去中心化的訴求漸漸消失，導致萊特幣的崛起。也因為如此，萊特幣才會選擇使用 Scrypt algorithm，希望避免重蹈覆徹。儘管如此，現今萊特幣的挖礦也仰賴 ASIC，因此一般人想要使用 PC 來挖礦也越來越不划算。

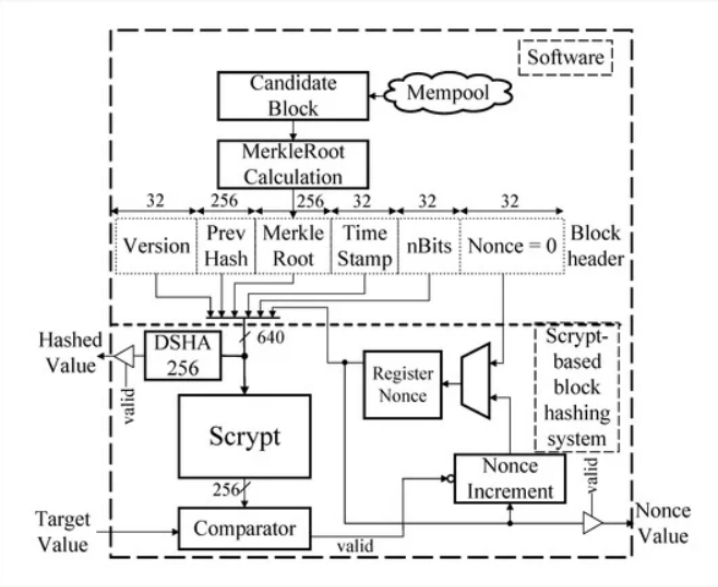


虛擬貨幣交易示意圖

萊特幣為何崛起？

Litecoin 改進的三大點在於確認交易的快速性、低成本交易和加密演算法改進。萊特幣每個區塊生成需要 2.5 分鐘，相較於比特幣的 10 分鐘更為簡短。而演算法的部分則是為了避免專業礦機削弱一般用戶的競爭力。Scrypt 具備 memory intensive 的特性 (可以想成礦工需要快速製造隨機數，而這些隨機數會被儲存於 RAM)。這算是 Scrypt 最大的特點之一，因為其他演算法如 SHA-256 等皆強調 computation intensive，容易導致專業礦機的出現而導致中心化。一開始 ASIC 並不適用於 scrypt-based 的 protocol，因此可確保使用 CPU 和 GPU 挖礦的人能擁有相同競爭力。

Scrypt 演算法：



Scrypt-based Block Hashing for the Blockchain Network

Algorithm 1 (block_hash, nonce) = Scrypt-based_block_hashing(block_header)

```
1: nonce = 0
2: while nonce < 2^32 do
3:     block_hash = SHA256(SHA256(block_header))
4:     scrypt_hash = Scrypt(block_header)
5:     if scrypt_hash < target then return (block_hash, nonce)
6:     else
7:         nonce = nonce + 1
8:     end if
9: end while
```

Scrypt Algorithm

Algorithm 2 Out = Scrypt (blockheader)

Scrypt variables and parameters for hashed block mining:

message = blockheader (640 bits)

salt = blockheader, padding (1024 bits)

Block size factor (r) = 1

Number of iterations (c) = 1

Parallelization parameter (p) = 1

CPU/Memory cost parameter (N) = 1024

Length of DK in bits (dklen) = 256

Algorithm's steps:

1: p_out = PBKDF2(message, salt, c, 1024 × r × p)

2: r_out = ROMix(p_out, N, r)

3: scrypt_out = PBKDF2(message, r_out, c, dklen)

4: **return** scrypt_out

PBKDF2

Algorithm 3 Out = PBKDF2(message, salt, c, dklen)

1: Out = ""

2: **for** i ← 1 to (dklen/256) **do**

3: DK_i = HMAC({salt, i}, message)

4: Out = {DK, DK_i}

5: **end for**

6: **return** Out

Algorithm 4 Out = HMAC(salti, message)

1: IPAD = $36363636...36_{16}$ (256 bits)

2: OPAD = $5C5C5C5C...5C_{16}$ (256 bits)

3: KHASH = SHA256(message)

4: IXOR = {(KHASH ⊕ IPAD), IPAD}

5: OXOR = {(KHASH ⊕ OPAD), OPAD}

6: IHASH = SHA256({IXOR, salti})

7: OHASH = SHA256({OXOR, IHASH})

8: Out = OHASH

9: **return** Out

Algorithm 5 digest = SHA256(message_in)

1: $M^{[0:N-1]}$, N = Padding_Splitting (message_in)

2: $H^{(0)}$ = H_Constants

3: **for** t ← 0 to (N-1) **do**

4: W = BlockDecomposition($M^{(t)}$)

5: $H^{(t+1)}$ = HashComputation($H^{(t)}$, K_Constants, W)

6: **end for**

7: **return** digest = {$H_1^{(N)}$, $H_2^{(N)}$, $H_3^{(N)}$, $H_4^{(N)}$, $H_5^{(N)}$, $H_6^{(N)}$, $H_7^{(N)}$, $H_8^{(N)}$}

ROMix

Algorithm 6 block = ROMix (block, N, r)

```
1: for i ← 0 to (N-1) do
2:     Mem_i = block
3:     block = BlockMix(block, r)
4: end for
5: for j ← 0 to (N-1) do
6:     j = block[489:480] (block's 10-bit from 480 to 489)
7:     block = BlockMix(block ⊕ Mem_j, r)
8: end for
9: return block
```

Algorithm 7 block = BlockMix(block, r)

```
1: X = block[1023:512] (block's 512 high bits)
2: for i ← 0 to (2 × r − 1) do
3:     X = X ⊕ block[511:0] (block's 512 low bits)
4:     X = X + Salsa20/8(X)
5:     if (i == 0) then
6:         OutH = X
7:     else
8:         OutL = X
9:     end if
10: end for
11: block = {OutH, OutL}
12: return block
```

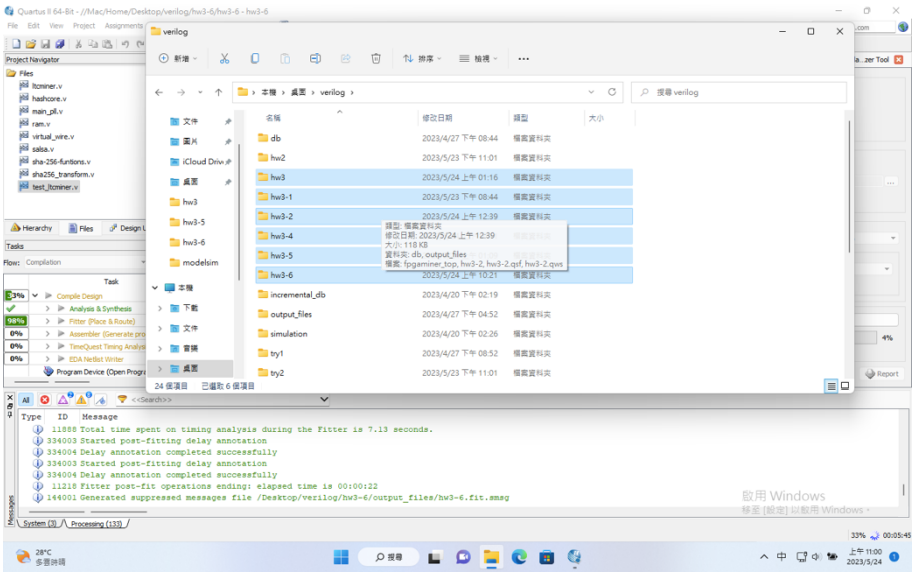
Algorithm 8 Out = Salsa20/8(message)

```
1: {x0, x1, ..., x15} = message
2: for i ← 0 to 3 do
3:     {y0, ..., y15} = ColumnRound({x0, ..., x15})
4:     {x0, ..., x15} = RowRound({y0, ..., y15})
5: end for
6: Out = {x0, x1, ..., x15}
7: return Out
```

Optimization 方向

1. Reduce clock cycles per hashing computation
   ⇨ Less clock cycles 相當於 more calculation done in a fixed amount of time
2. Reduce area
   ⇨ Less area per hashing module 相當於 more hashing modules that can fit on the board（就像晶片越做越小一樣）
3. **Try on different FPGA to see if there is any difference regarding power dissipation.**

Verilog compiling



總共試了六個檔案，一開始找的程式碼較為完整，同時也用到了 BUFG, IBUFG 等 global clock，但在 compile 時一直顯示找不到這兩個的 entity，所以我就嘗試在網路上尋找這兩個的 module，也是因為這樣才知道這兩個東西是什麼。然而，找不到太多相關資訊，下載了 xilinx 的使用手冊也沒有更多進一步的結果。最後的結論是不確定是不是要在 prime edition 才能正常使用，所以我換了另一個程式碼，較為簡單的版本。Compile 也都很順利，幾乎沒有 error，跟前一個比起來真是容易太多了，直接進到 modelsim-altera，卻也在 powerplay power analysis 花了一段時間，首先因為檔案較大，要完整 compile 本來時間就長。加上第一次使用所以有很多不太確定的步驟。 必須直接按 compilation 的箭頭符號才能跑到 fitter 和 assembler 的部分，這樣之後進行 powerplay power analysis 才不會有問題！

Powerplay power analysis

FPGA：EP4CGX50CF23C6

power dissipation = 132.92 mW

PowerPlay Power Analyzer Summary	
PowerPlay Power Analyzer Status	Successful - Wed May 24 14:11:01 2023
Quartus II 64-Bit Version	13.0.1 Build 232 06/12/2013 SP 1 SJ Web Edition
Revision Name	hw3-6
Top-level Entity Name	ltcminer
Family	Cyclone IV GX
Device	EP4CGX50CF23C6
Power Models	Final
Total Thermal Power Dissipation	138.92 mW
Core Dynamic Thermal Power Dissipation	0.00 mW
Core Static Thermal Power Dissipation	126.80 mW
I/O Thermal Power Dissipation	12.12 mW
Power Estimation Confidence	High: user provided sufficient toggle rate data

Slow 1200mV 85C Model Fmax Summary

	Fmax	Restricted Fmax	Clock Name	Note
1	100.32 MHz	100.32 MHz	altera_reserved_tck	

FPGA：EP4CGX30BF14C6

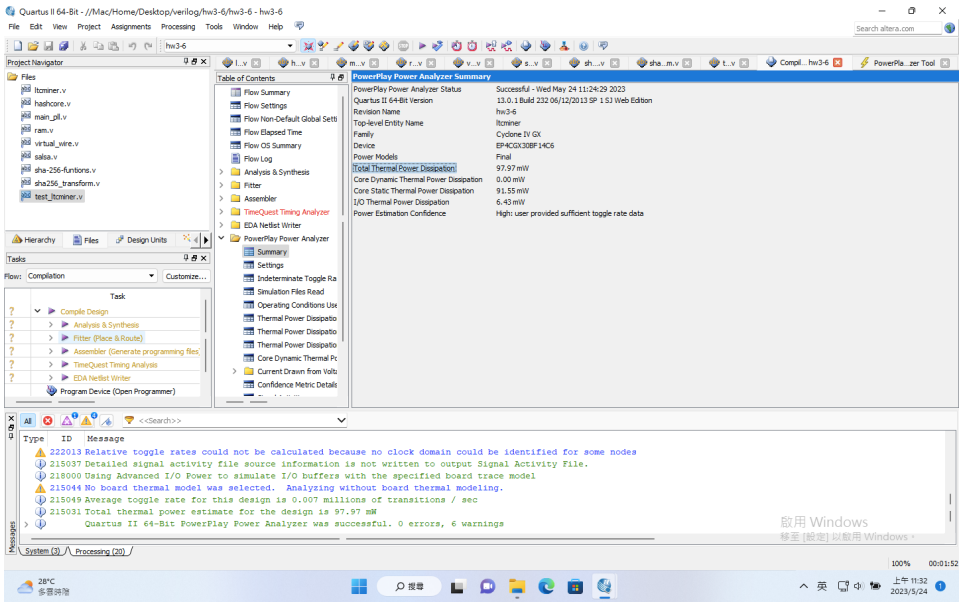
power dissipation = 97.97 mW

PowerPlay Power Analyzer Summary

PowerPlay Power Analyzer Status	Successful - Wed May 24 11:24:29 2023
Quartus II 64-Bit Version	13.0.1 Build 232 06/12/2013 SP 1 SJ Web Edition
Revision Name	hw3-6
Top-level Entity Name	ltcminer
Family	Cyclone IV GX
Device	EP4CGX30BF14C6
Power Models	Final
Total Thermal Power Dissipation	97.97 mW
Core Dynamic Thermal Power Dissipation	0.00 mW
Core Static Thermal Power Dissipation	91.55 mW
I/O Thermal Power Dissipation	6.43 mW
Power Estimation Confidence	High: user provided sufficient toggle rate data

Slow 1200mV 85C Model Fmax Summary

	Fmax	Restricted Fmax	Clock Name	Note
1	101.32 MHz	101.32 MHz	altera_reserved_tck	



FPGA：EP4CGX30BF14C8

power dissipation = 97.95 mW

PowerPlay Power Analyzer Summary

PowerPlay Power Analyzer Status	Successful - Wed May 24 13:57:42 2023
Quartus II 64-Bit Version	13.0.1 Build 232 06/12/2013 SP 1 SJ Web Edition
Revision Name	hw3-6
Top-level Entity Name	ltcminer
Family	Cyclone IV GX
Device	EP4CGX30BF14C8
Power Models	Final
Total Thermal Power Dissipation	97.95 mW
Core Dynamic Thermal Power Dissipation	0.00 mW
Core Static Thermal Power Dissipation	91.52 mW
I/O Thermal Power Dissipation	6.43 mW
Power Estimation Confidence	High: user provided sufficient toggle rate data

Slow 1200mV 85C Model Fmax Summary

	Fmax	Restricted Fmax	Clock Name	Note
1	84.36 MHz	84.36 MHz	altera_reserved_tck	

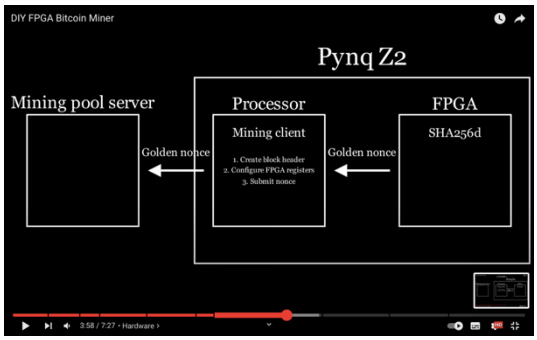
結論：

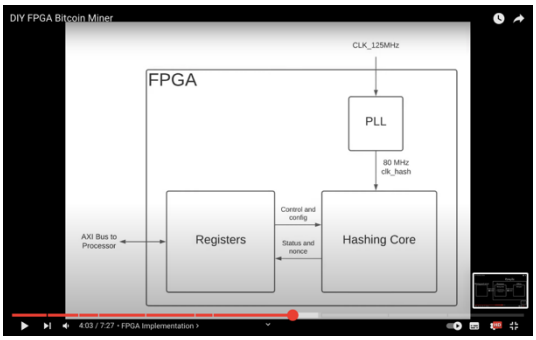
1. 演算法：

因為 SHA-256 只是 Scrypt 的一部分，而 ROM 應該是計算量最龐大的區域，所以從那邊著手改進應該會提升整體速度。

2. 硬體：

嘗試三種 FPGA 型號，其中第一個跟後面兩個相差最明顯，更換後 power dissipation 減少了 34.95 mW，相當於減少 27%左右的能量損耗。





FPGA 簡易圖示

Reference:

https://www.youtube.com/watch?v=3c7aGdeMfjw&t=1s
https://www.youtube.com/watch?v=M_AocgfvTIs
https://www.ithome.com.tw/news/105374
https://www.esat.kuleuven.be/cosic/publications/thesis-260.pdf