PERANCANGAN DAN IMPLEMENTASI PROSESOR SCRYPT UNTUK CRYPTOCURRENCY DENGAN ARSITEKTUR PIPELINE BERBASIS FPGA

DESIGN AND IMPLEMENTATION SCRYPT PROCESSOR FOR CRYPTOCURRENCY WITH PIPELINE ARCHITECTURE BASED ON FPGA

Muhammad Abdillah, Iswahyudi Hidayat, Rizki Ardianto Prodi S1 Teknik Elektro, Fakultas Teknik Elektro, Universitas Telkom
muhammadabdillah@students.telkomuniversity.ac.id rizkia@telkomuniversity.ac.id
iswahyudihidayat@telkomuniversity.ac.id

Abstrak

Dewasa ini cryptocurrency sudah terbukti menjadi salah satu pilihan terbaik untuk proses transaksi uang. Hal tersebut dikarenakan setiap cryptocurrency memiliki proses kriptografi didalamnya, apalagi cryptocurrency yang menggunakan algoritma kriptografi yang kuat, contohnya scrypt. Maka pada penelitian kali ini didesain sebuah prosesor scrypt untuk proses mining salah satu cryptocurrency yaitu litecoin. Prosesor yang dibuat ditambahkan arsitektur pipeline didalamnya dengan tujuan mengurangi jumlah clock yang dibutuhkan. Jumlah clock setelah penambahan arsitektur berkurang 7,92% dari jumlah clock sebelumnya. Implementasi sistem secara perangkat keras menggunakan board FPGA ATLYS yang menghabiskan 50% memori internal FPGA. Berdasarkan delay yang dihasilkan dari proses sintesis, frekuensi maksimum yang dapat digunakan pada implementasi tersebut adalah 26,968MHz. Hash rate yang didapat sebesar 610Hash/detik.

Kata kunci : **Kriptografi, Cryptocurrency, Scrypt, Litecoin, HDL, FPGA**

Abstract

Cryptocurrencies has already became the better choice because it is proven to be saver for money transaction these days. That is because every cryptocurrency has a cryptography process in it, even more cryptocurrencies which using one of the most secure cryptography algorithm these days, scrypt. So a scrypt processor is designed in this research for mining process cryptocurrency litecoin. Pipeline architecture is added in the processor so it will use less clock than before. Number of clock after adding the pipeline architecture is decreased by 7,92%. Than a hardware based implementation of the system on ATLYS FPGA board spent 50% og it's internal memory. The maximum usable frequency is 26,968Mhz based on minimum delay achieved after synthesis process. Hash rate achieved is 610Hash/second.

Keywords : **Cryptography, Ceyptocurrency, Scrypt, Litecoin, HDL, FPGA**

---

1. Pendahuluan

Istilah *cryptocurrency* akhir-akhir ini (setelah kemunculan bitcoin pada tahun 2009) secara terus menerus menjadi lebih banyak dibicarakan oleh banyak kalangan. Dimulai dari kalangan jurnalis dan juga mulai masuk ke bidang socio-ekonomi[1] karena sifatnya yang tidak dikelola secara utuh oleh 1 bank sentral. Karena sifatnya yang tidak memiliki bank sentral, keamanan dari seluruh aktifitas diawasi oleh seluruh *miner* dari *cryptocurrency* tersebut. Selain itu setiap keping uang dari sebuah *cryptocurrency* memiliki kode unik yang dihasilkan dari proses kriptografi. Kode unik tersebut memiliki panjang yang mungkin saja berbeda untuk setiap *cryptocurrency* dan menjadi salah satu jaminan tingkat kesulitan untuk dipalsukan. Dilain hal, panjang dari kode unik ini memastikan bahwa uang yang beredar pada sebuah *cryptocurrency* tidak akan melebihi jumlah maksimal dari kombinasi bitnya. Hal ini yang kemudian menjamin sebuah *cryptocurrency* tidak mengalami *hyper inflasi* karena pembuatan uang yang terlalu banyak oleh bank sentral.

2. Kriptografi Dalam *Cryptocurrency* Berbasis Scrypt

A. SHA-256

Standard Hash Algortihm[2] mempunyai beberapa sub algoritma yang dibedakan berdasarkan panjang data (key) yang akan diproses, salah satunya merupakan SHA-256 yang kemudian juga digunakaan dalam sistem ini. Jika panjang key user tidak sesuai dengan panjang key standar SHA yang digunakan maka perlu ada proses padding yang menyesuaikan panjang key user dengan panjang key standar. Selain key, SHA harus memiliki

masukan lain yaitu nilai awal (direpresentasikan dalam bit strings) dari SHA (yang kemudian menjadi state dari SHA), beberapa konstanta serta sebuah scheduler yang mengatur skema kerja dari SHA. Nilai awal SHA-256 diambil dari dengan 32 bit pertama dari the fractional parts of the square roots dari 8 angka prima pertama. Setiap SHA dikerjakan dalam beberapa kali proses *hash*; keluaran saat ini kemudian bisa menjadi masukan pada proses berikutnya, satu buah key akan diproses oleh SHA-256 dalam 1 buah *cycle* yang terdiri dari 64 kali proses *hash* (64 round).

Setiap sistem dari *Standard Hash Algorithm* mempunyai fungsi yang akan mengkases nilai konstanta yang berbeda pada setiap *round* dari proses *hash*. Konstanta pada SHA-256 merupakan 64 buah words (32 bit) yang akan diakses setiap wordnya (per-32 bit) secara bergantian dari awal *round* sampai akhir. Nilai konstanta tersebut diambil dari 32 bit pertama dari pecahan dari akar pangkat tiga dari 64 angka prima pertama. Fungsi-fungsi yang digunakan merupakan operasi logika dan penggeseran dengan masukan berupa state atau nilai awal dari SHA, konstanta yang sedang digunakan pada *round* saat ini serta *key* yang dimasukkan, fungsi-fungsi tersebut selalu sama pada semua *round*.

B. PBKDF2

Public Key Cryptography Standard (PKCS) adalah beberapa algoritma yang dirancang dan dipublikasi oleh RSA Security LLC sebagai standar dari beberapa proses kriptografi seperti KDF. PKCS #5[3] secara umum membahas tentang algoritma standar dari proses *password based cryptography* seperti *Password Based Key Derivation Function* (PBKDF1), PBKDF2, *Password Based Encryption Standard* 1 (PBES1), PBES2 dan *Password Based Message Authentication Standard* 1 (PBMAC1). Seperti yang sudah disebutkan sebelumnya bahwa *cryptocurrency* mnenggunakan KDF dalam prosesnya dan yang kemudian digunakan sebagai algoritma penurunan keynya adalah PBKDF2 dengan SHA-256 sebagai PRFnya. Beberapa faktor yang menentukan proses dari PBKDF2 adalah PRF, *password/key*, salt, jumlah iterasi dan panjang *key* turunannya yang kemudian di fungsikan sevagai :

DK (*derived key*) = PRF (P , S, c, dkLen).

Password merupakan masukan key dari user yang tidak diketahui sedangkan salt meruapakan parameter dalam sistem yang dapat diturunkan dari *password* yang dimasukan. *Iteration count* digunakan untuk mempersulit penurunan dari sebuah kunci, dalam PBKDF2 jumlah iterasi minimum adalah 1000. Semakin banyak jumlah iterasi maka semakin sulit untuk melakukan proses coba-coba *password*, karena setiap mencoba *password* maka *password* tersebut harus melewati serangkaian proses iterasi yang panjang.

Proses PBKDF2 dimulai dengan penentuan jumlah iterasi yang diinginkan serta panjang *key* turunan yang diinginkan sehingga didapatkan berapa blok proses dengan cara membagi panjang *key* turunan yang dinginkan dengan panjang keluaran PRF. Setelah itu proses dari setiap blok dilakukan secara terpisah dengan setiap bloknya dilakukan dengan fungsi PBKDF2 diatas dengan jumlah iterasi yang sudah ditentukan sebelumnya. Pada iterasi pertama keluaran didapatkan dari proses U1 = PRF (P, S), kemudian pada iterasi kedua U1 digunakan sebagai salt menjadi U2 = PRF (P, U1) sedemikian selanjutnya hingga iterasi terakhir selesai. Keluaran pada satu blok diambil dari hasil *bit-wise* xor seluruh keluaran PRF pada semua iterasi. Kemudian *key* turunan merupakan gabungan dari keluaran semua blok.

C. Salsa

Salsa[4] adalah sebuah fungsi *hash* yang didesain cukup cepat dengan tidak bisa dikerjakan secara pararel sehingga merupakan salah satu *sequential function*. Pada dasarnya salsa pada kriptografi digunakan untuk proses enkripsi-dekripsi data dan bukan merupakan KDF seperti PBKDF2. Operasi-operasi dasar yang digunakan dalam salsa adalah bit-wise xor, penjumlahan serta rotasi kiri dari data yang disusun per-word (32 bit).

Fungsi dasar dari salsa disebut *the quarter round function* yang mengubah masukan 4 word menjadi keluaran 4 word dengan menggunakan operasi-operasi dasar. Sedangkan untuk mengoperasikan masukan 16 word maka *the row round function* dan *the column round function* digunakan dengan dasar fungsi sebelumnya sehingga juga menghasilkan keluaran sepanjang 16 word. Perbedaan dari *columnround* dan *rowround* terdapat pada pengurutan masukan dan keluaran dari fungsi quarterround secara kolomatau baris jika masukan 16 word dibentuk sebagai matriks :

$$y = \begin{matrix} y_{00} & y_{01} & y_{02} & y_{03} \\ y_{04} & y_{05} & y_{06} & y_{07} \\ y_{08} & y_{09} & y_{10} & y_{11} \\ y_{12} & y_{13} & y_{14} & y_{15} \end{matrix} \quad (1)$$

Kedua *round* diatas dikerjakan secara berturut-turut *columnround* terebih dahulu kemudian *rowround*

$$z = \text{doubleround}(y) = \text{rowround}(\text{columnround}(y)) \quad (2)$$

dan disebut sebagai *the doubleround function*. Akhirnya *the littleendian function* adalah fungsi dengan masuk 4 byte (8 bit) dan keluaran 1 word yaitu jika b = (b0,b1,b2,b3) maka :

$$\text{littleendian}(b) = 2^0(b0) + 2^8(b1) + 2^{16}(b2) + 2^{24}(b3) \quad (3)$$

Proses salsa20 dimulai dengan inisialisasi masukan 16 word (x0,x1,..,x15) untuk setiap word terdiri dari 4 byte (x0 = (x0(0),(x0(1),(x0(2),(x0(3))) yang masing-masing word dimasukkan kedalam *the littleendian function*. Dari fungsi *littleendian* didaptkan 16 word data yang akan dimasukkan ke fungsi *doubleround* dan akan menghasilkan keluaran 16 word yaitu data terenkripsi.

D. Scrypt

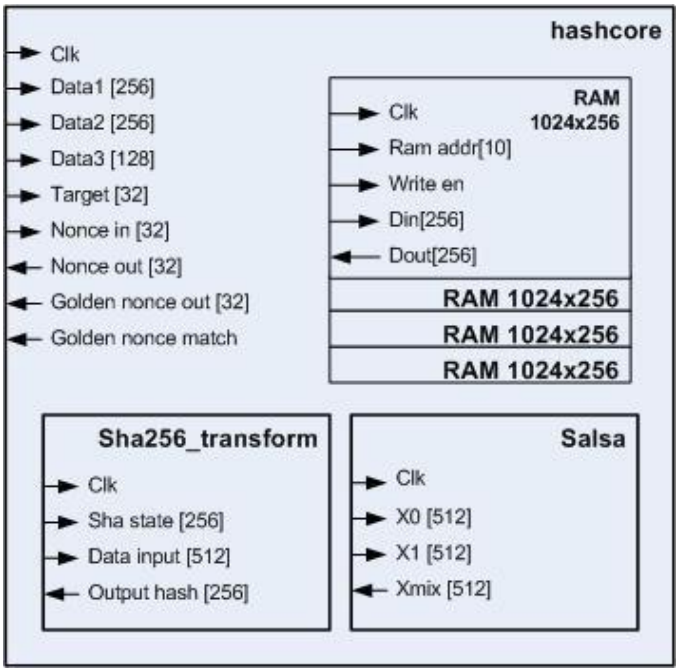
Ide dari scrypt[5] pada awalnya adalah membuat sebuah KDF dengan menggunakan *resource* yang sebanyak-banyaknya sehingga jauh lebih mempermahal proses *brute-force attack*. Ide tersebut kemudian di realisasikan dengan cara mengikutsertakan elemen memori sebanyak-banyaknya pada proses *hash* selain juga harus dengan jumlah iterasi yang sebanyak-banyaknya dan sesuai dengan jumlah memori yang digunakan. Algoritma dengan karakteristik seperti itu dinamakan *memory-hard algorithm*; yang dampaknya lainnya adalah perangkat keras untuk *brute force attack* menjadi lebih besar (perangkat keras lebih besar maka biaya lebih mahal). Selain itu sebuah KDF dengan *resource* sebesar-besarnya tidak boleh bisa dikerjakan secara pararel sehingga jumlah iterasi yang digunakan akan sama percis dengan jumlah memori yang digunakan. Sehingga muncul algoritma baru yang disebut dengan *sequential memori function* yang akan menyebabkan brute force dengan perangkat keras juga semakin sulit dilakukan.

Sebuah algoritma ROMix dipublikasikan dalam paper scrypt yang juga sudah dibuktikan merupakan sebuah *sequential memory hard function*. Algoritma ROMix diawali dengan inisialisasi *hash function* (H) yang akan digunakan dan berapa jumlah iterasi yang juga menentukan jumlah memori yang akan digunakan. Kemudian semua memori diisi dengan keluaran dari fungsi *hash* dari setiap iterasi sampai memori tersebut penuh (setiap memori dengan alamat ke-i diisi dengan keluaran iterasi ke-i). Setelah itu sebuah fungsi yang dinamakan *integrify* digunakan sebagai penunjuk alamat memori yang akan diakses selanjutnya setelah proses pengisian memori selesai yang alamatnya diambil dari keluaran terakhir saat ini. Proses selanjutnya adalah *mixing* (diiterasi dengan jumlah yang sesuai dengan jumlah iterasi pengisian memori) yaitu keluaran terakhir dari H setelah proses pengisian memori selesai, di masukkan kembali kedalam fungsi H sebagai masukan. Namun sebelumnya dilakukan *bit-wise* xor dengan data pada memori pada alamat yang ditunjuk oleh fungsi *integrify*.

Pada proses scrypt masukan dari ROMix harus melewati proses PBKDF2 terlebih dahulu untuk merubah masukan scrypt menjadi bitstream yang sesuai dengan algoritma ROMix. Kemudian hasil dari algoritma ROMix dikembalikan lagi kedalam PBKDF2 untuk kemudian menghasilkan *key* turunan yang dicari. PRF yang digunakan pada PBKDF2 adalah SHA256 dan *hash function* yang digunakan pada algoritma ROMix adalah salsa *core*.

E. Hashcore

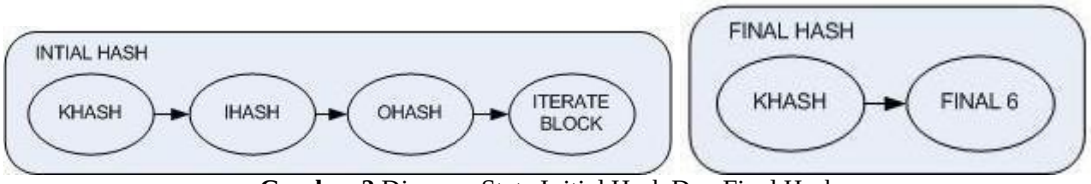
Sistem *mining* litecoin seperti yang sudah dijelaskan sebelumnya merupakan sistem kriptografi yang terdiri dari 2 *core* blok yaitu salsa dan SHA-256 yang tergabung dalam sebuah blok besar hashcore. Salsa *core* memiliki 2 masukan masing-masing 512 bit (x1, x0) dan 1 keluaran 512 bit (xmix) sedangkan SHA-256 memiliki 1 masukan 256 bit (SHA-state), 1 masukan 512 bit (data masukan) serta 1 keluaran 256 bit (keluaran *hash*). Setiap satu paket data yang diterima sistem terdiri dari 1 data 128 bit (data 3), 2 data 256 bit (data 2 dan data 1) dan 1 target 32 bit yang kemudian menjadi masukan dari algoritma kriptografi awal yaitu PBKDF2. Proses pada PBKDF2 terdiri dari beberapa state yaitu KHASH, IHASH, OHASH, ITERATE BLOCK dan FINAL 6 yang setiap prosesnya dieksekusi *core* SHA-256. State-state tersebut kemudian dikelompokkan menjadi blok-blok state yaitu INITIAL HASH (KHASH, IHASH, OHASH dan ITERATE BLOCK) dan FINAL HASH (KHASH dan FINAL 6).



Gambar 1 Blok Diagram Hashcore

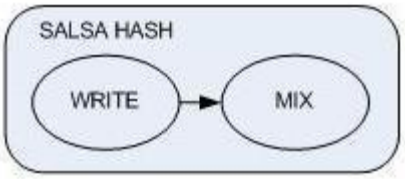
State KHASH merupakan state awal yang memiliki data masukan ketiga data diatas serta sebuah masukan lain yaitu nonce (akan dijelaskan pada bagian 3) dan akan diproses dalam 2 buah *cycle* SHA-256 (karena satu paket data lebih panjang dari masukan SHA-256 yang memiliki masukan sepanjang 512 bit). SHA-State yang digunakan adalah initial value dari SHA-256 sedangkan keluaran *hash* dari KHASH disimpan didalam sebuah variable KH untuk kemudian digunakan pada state selanjutnya. Selanjutnya state kedua adalah IHASH dengan data masukan KH xor "36 hexa" sepanjang 512 bit dan data2, data1 yang juga di proses dalam 2 *cycle*. SHA-State yang digunakan dalam SHA-256 merupakan initial value dari SHA-256 dan keluaran dimasukkan kedalam variable IH. State ketiga merupakan OHASH dengan data masukan KH xor "5c hexa" sepanjang 512 bit dan diproses dalam 1 *cycle*. SHA-State yang digunakan merupakan initial value dari SHA-256 dan keluaran dimasukkan kedalam variable OH.

ITERATE BLOCK merupakan state yang digunakan untuk membuat data bit stream sepanjang 1024 bit (disimpan dalam variable xbuf) yang kemudian dimasukan kedalam x1 dan x0 dari salsa *core*. Setiap proses iterasi ditandai dengan sebuah variable penghitung blockcnt dengan jumlah iterasi sebanyak 4 kali agar keluaran hash dari SHA-256 sepanjang 256 bit. Setiap iterasi akan dilakukan dalam 2 *cycle* dengan setiap iterasi menggunakan data masukan blockcnt, nonce dan data 3. SHA-state yang digunakan adalah IH pada *cycle* pertama dan OH pada *cycle* kedua. State terakhir yaitu FINAL 6 menggunakan 6 *cycle* dengan SHA-state pada setiap *cycle*-nya merupakan keluaran *hash cycle* sebelumnya kecuali pada *cycle* pertama dan kelima menggunakan initial value dari SHA-256. Pada state ini juga keluaran dari salsa diproses kembali sebagai data masukan dari SHA-256 dalam 2 *cycle* yaitu *cycle* kedua dan *cycle* ketiga. Pada akhir state ini, jika sebuah parameter yang diambil dari keluaran *hash* terakhir SHA-256 kurang dari target yang diterima sebelumnya maka nonce yang digunakan dalam proses dikeluarkan.



Gambar 2 Diagram State Initial Hash Dan Final Hash

Salsa *core* digunakan dalam 2 SALSA HASH state yaitu state WRITE dan MIX yang setiap statenya menggunakan memori dengan 1024 alamat dengan setiap alamatnya berisi 1024 bit dalam prosesnya. Satu buah *cycle* pada kedua state tersebut adalah full proses dari salsa *core* untuk data x1, x0 yang terdiri dari 2 kali proses salsa *core*. Setiap proses dari salsa *core* adalah pengulangan sebanyak 4 kali dimana pada pengulangan pertama x1, x0 dari salsa *core* adalah x1, x0 dari state, sedangkan pada pengulangan kedua, tiga dan empat x1, x0 adalah xmix dari pengulangan sebelumnya. Pada setiap akhir proses isi x0 diganti dengan x1 sedangkan x1 diganti oleh xmix.



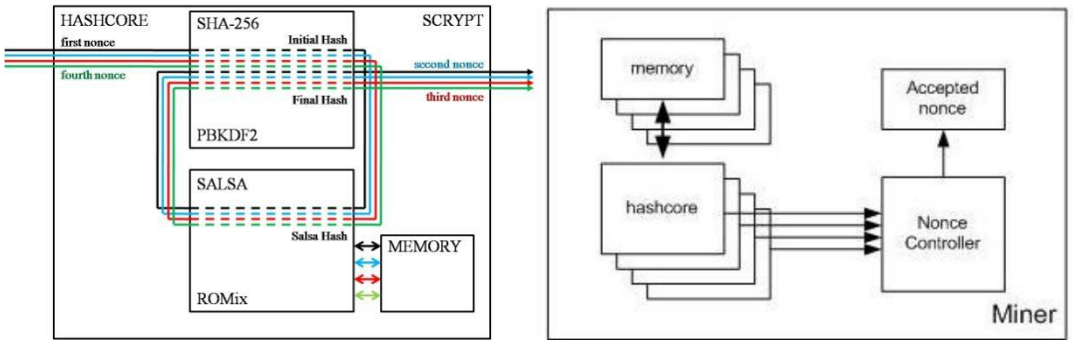
Gambar 3 Diagram State Salsa Hash

State WRITE merupakan state yang digunakan untuk mengisi semua seluruh memori dalam 1024 kali *cycle*. Pengisian data dilakukan secara berurut dari alamat pertama sampai alamat terakhir pada setiap awal *cycle* dan data yang diisikan dalam memori adalah x1, x0. Kemudian dilanjutkan dengan proses MIX yaitu proses pencampuran data dalam ROM juga sebanyak 1024 kali *cycle*. Pada state ini alamat memori yang diakses ditunjuk dari xmix pada *cycle* sebelumnya kemudian di-xor-kan dengan x1, x0 pada awal dari setiap *cycle*. Secara umum setiap paket data akan menlewati proses INITIAL HASH, SALSA HASH dan FINAL HASH secara beturut-turut.

Gambar 4 Diagram State Sistem Mining Litecoin

3. Sistem *Mining Cryptocurrency* Berbasis Scrypt

Dalam sebuah implementasi dari proses *mining* diperlukan setidaknya 1 buah hashcore yang melakukan proses ketiga blok state diatas secara berulang-ulang sampai targetnya tercapai atau sudah ada data baru. Nonce merupakan salah satu masukan dari KHASH pada state PBKDF2 yang akan diincrement disetiap awal dari pengulangan hashcore selanjutnya. Hal ini yang mengakibatkan data yang diproses serta keluaran yang dihasilkan hashcore pada setiap pengulangan akan selalu berbeda. Nonce juga bisa dikeluarkan untuk mengetahui berapa kali jumlah pengulangan yang sudah dilakukan.



Gambar 5 Diaram Alur Data Dalam Hashcore Dan Diagram Blok Sistem Mining Multicore

Selain itu nonce juga dibedakan untuk setiap hashcore pada desain sistem *mining* multicore yang bekerja secara pararel dengan noncenya masing-masing. Sebuah sistem multicore tentunya membutuhkan kontroller yang mampu memilih keluaran hashcore mana yang sudah memiliki hasil kurang dari target. Untuk mendesain kontroller ini dibutuhkan variable "nonce match" yang akan mengeluarkan nilai 1 jika ketentuan diatas sudah terpenuhi. Variable tersebut ada disetiap hashcore sehingga nonce yang dikeluarkan oleh sistem *mining* dipilih dari hashcore yang mempunya golden match sama dengan 1. Karena semua hashcore bekerja secara pararel maka nonce dari hashcore pertama sudah cukup untuk menentukan jumlah pengulangan yang sudah dilakukan.

Pada pengimplementasiannya semua proses tentunya memiliki jumlah latency clocknya masing-masing yang menunjukkan berapa lama setiap blok state dikerjakan. Setiap pengerjaan 1 *cycle* SHA-256 dibutuhkan 2 buah clock tambahan untuk sinkronasi dikarenakan desain SHA-256 membutuhkan 2 buah buffer didalamnya. Blok state INITIAL HASH memerlukan 13 *cycle* SHA-256 sehingga memakan total waktu 858 clock sedangkan blok state FINAL HASH membutuhkan 8 *cycle* yang berarti memakan waktu total 528 clock. Sementara itu blok state SALSA HASH membutuhan total 2048 *cycle* salsa *core* yang berarti memakan waktu total 16384 clock. Sehingga satu kali pengulangan hashcore akan memakan waktu total 17770 clock (35540 clock untuk 2 kali pengulangan hashcore).

Sebuah arsitekture dengan pipeline merupakan architecture dengan kemampuan membuat langkah overloop dari sebuah sistem. Sehingga jika architecture pipeline diimplementasikan maka proses hashcore kedua dan seterusnya akan lebih cepat dari proses hashcore pertama. Proses pipelining digunakan pada blok state SALSA HASH dari pengulangan hashcore pertama saat SHA-256 tidak digunakan blok state apapun. Nonce yang digunakan dipindahkan terlebih dahulu ke variabel baru nonce 1 sehingga SHA-256 bisa melakukan INITIAL HASH untuk pengulangan hashcore kedua dengan nonce baru yang sudah diincrement.

Setelah blok state SALSA HASH untuk pengulangan hashcore pertama selesai maka SHA-256 melakukan blok state FINAL HASH untuk pengulangan hashcore pertama.

Tabel 1 Diagram Proses Mining Litecoin Dengan Arsitektur Pipeline

Nonce Number	State						
1	IH	SH	FH				
2		IH	SH	FH			
3			IH	SH	FH		
4				IH	SH	FH	
5					IH	SH	FH
Process	1	2	3	4	5	6	7

Karena proses SALSA HASH yang membutuhkan jumlah clock yang jauh lebih banyak dibandingkan INITIAL HASH dan FINAL HASH maka proses FINAL HASH dari pengulangan hashcore pertama akan selesai terlebih dahulu. Sehingga nonce bisa diincrement kembali dan SHA-256 melakukan INITIAL HASH untuk pengulangan hashcore ketiga bersamaan dengan SALSA HASH untuk pengulangan hashcore kedua.

Seperti sudah dijelaskan sebelumnya bahwa pembandingan nilai hasil proses *mining* dengan target hanya dilakukan pada akhir proses FINAL HASH maka nonce match pertama hanya akan keluar paling cepat pada clock ke 17770. Namun karena pipelining maka mulai dari pengulangan hashcore kedua hasil hash akan bisa dibandingkan lebih cepat yaitu hanya selama blok state SALSA HASH. Sehingga pembandingan hasil *hash* kedua akan dilakukan pada clock ke 34154 yaitu 1386 clock lebih cepat dari sistem *mining* tanpa pipelining.

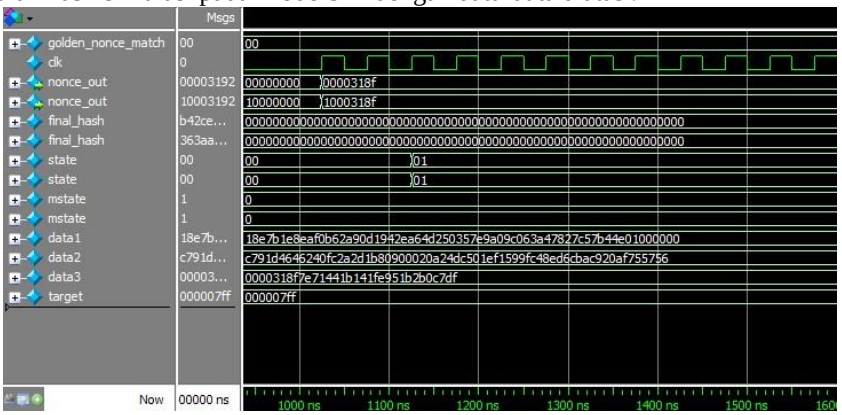
4. Simulasi Dan Implementasi FPGA

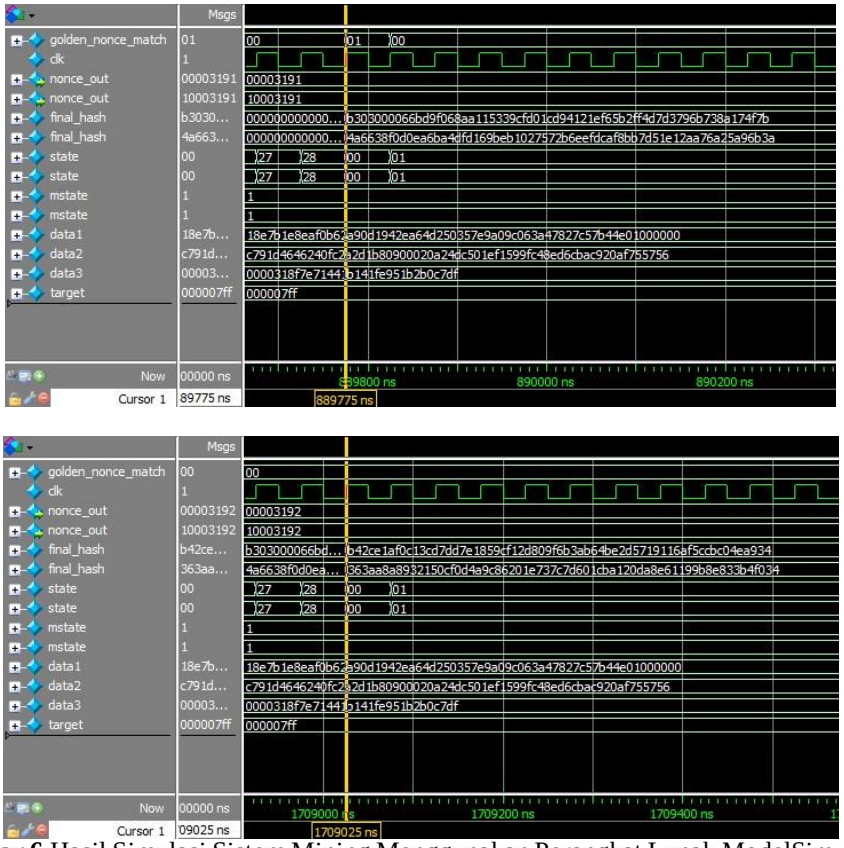
Simulasi dilakukan dengan menggunakan data, target dan hasil seharusnya yang sudah disediakan dari awal. Hasil simulasi harus menghasilkan golden match pada hashcore 1 pada pengulangan pertama dan juga hasil *hash* yang sama persis. Berikut nilai awal dan nilai akhir untuk simulasi dalam hexadesimal :

Tabel 2 Nilai Awal Yang Digunakan Serta Nilai Akhir Yang Harus Dicapai Pada Simulasi

Data 1	18e7b1e8eaf0b62a90d1942ea64d250357e9a09c063a47827c57b44e01000000
Data 2	c791d4646240fc2a2d1b80900020a24dc501ef1599fc48ed6cbac920af755756
Data 3	0000318f7e71441b141fe951b2b0c7df
Target	000007ff
Final Hash	b303000066bd9f068aa115339cfd01cd94121ef65b2ff4d7d3796b738a174f7b

Berikut adalah hasil simulasi pada ModelSim dengan data-data diatas :





Gambar 6 Hasil Simulasi Sistem Mining Menggunakan Perangkat Lunak ModelSim-Altera

Sebuah proses sintesis menggunakan perangkat lunak Xilinx ISE 14.5 sudah dilakukan untuk mengetahui berapa jumlah *resource* yang dibutuhkan untuk mengimplementasikan sistem yang sudah dirancang. Implementasi menggunakan board ATLYS Spartan-6 XC6SLX45 CSG324C. Berikut tersedia hasil implementasi dari sistem *mining* litecoin :

Tabel 3 Hasil Implementasi Sistem Mmining Litecoin Di FPGA

Slice Logic	**Digunakan**	**Tersedia**	**Persentase**
Slice Registers	6.524	54.576	11%
Sebagai Flip-flop	6.499		
Sebagai Latch	0		
Sebagai Latch-thrus	0		
Sebagai gerbang AND/OR	58		
Slice LUT	9.459	27.288	34%
Sebagai gerbang logika	9.358	27.288	34%
Sebagai Memori	0	6.408	0%
Occupied Slice	3.351	6.822	49%
MUXCY	2.624	23.644	19%
LUT Flip Flop pairs	11.099		
Filp Flop Tidak Digunakan	5.051	11.099	45%
LUT Tidak Digunakan	1.640	11.099	14%
Flip Flop Dan LUT digunakan	4.408	11.099	39%
Bonded I/O Blok	15	218	6%
RAMB16BWER	56	116	48%
RAMB8BWER	4	232	1%
BUFIO2/BUFIO2_2CLK	1	32	3%
BUFIO2FB /BUFIO2FB_2CLK	1	32	3%
BUFG/BUFGMUX	3	16	18%
DCM/DCM_CLKGEN	1	8	12%

Selain *resource* yang dibutuhkan, hasil sintesis yang lain adalah *timing constraint* dari desain yang sudah ada. Berikut hasil *timing constraint* dari sistem yang sudah disintesis :

Tabel 4 *Timing Constraints* Implementasi FPGA Dari Desain

Constraint	***Best Case Achievable***
hash_clk	37,080 ns
uart_clk	3,891 ns

5. Kesimpulan

Berdasarkan hasil perancangan, implementasi dan pengujian keseluruhan pada tugas akhir ini, pemrosesan data dan nonce singlecore ataupun multicore sudah dapat berjalan dengan baik. Hal tersebut dapat dilihat dari final hash yang sudah sesuai dengan data yang seharusnya. Sementara itu pengimplementasian arsitektur pipeline mengurangi total clock yang dibutuhkan sebesar 7,92%. Frekuensi maksimum yang dapat dipakai adalah 26,968MHz sesuai dengan *time constraint* dari sintesis, namun pada implementasinya frekuensi clock yang digunakan adalah 10MHz. Kemudian nilai hash rate didapat dari frekuensi maksimum yang dapat digunakan dibagi jumlah clock yang digunakan yaitu 610H/s.

Daftar Pustaka:

[1] Dostov, Victor. Shust, Pavel. (2013) Cryptocurrencies : an unconventional challenge to the AML/CFT regulators?. *Jurnal of Financial Crime*. [Online] 21, 3, 249-263. Tersedia di : http://dx.doi.org/10.1108/JFC-06-2013-0043 [diakses 5 September 2014].

[2] Compuer Security Resource Center. (2002) *Secure Hash Standard*. [Online] National Isntitute of Standard and Technology. Federal Information Precessing Standard (FIPS) Publication 180-2. Tersedia di : http://csrc.nist.gov/publication/fips/fips180-2/fips180-2withchangenotice.pdf [diakses 7 Mei 2015].

[3] Kaliski, Burt. (2000) PKCS #5 : *Password-Based Cryptography Sepcification Version 2.0*. [Online] RSA Laboratories. Tersedia di : https://ietf.org/rfc/rfc2898.txt [diakses 12 Mei 2015].

[4] Bernstein, Daniel Julius. (2007) *Salsa20 Specification*. [Online] Depatment of Mathematics, Statistics and Computer Science. The University of Illionis. Tersedia di : https://cr.yp.to.snuffle/salsafamily-20071225.pdf [diakses 25 Mei 2015]

[5] Percival, Colin. (2009) *Stronger Key Derivation Function Via Sequential Memory-Hard Function*. [Online] BSDCan'09 conference. Mei 2009. Tersedia di : https://www.tarsnap.com/scrypt/scrypt.pdf [diakses 7 Mei 2015].