به نام خدا

مستند پروژه‌ی طراحی سیستم‌های دیجیتال

Echo Hashing Algorithm

استاد: دکتر بهاروند

اعضای گروه:

امیر افسری

سید علی‌رضا حسینی

نیما جمالی

علی‌رضا دقیق فرسوده

سینا کاظمی

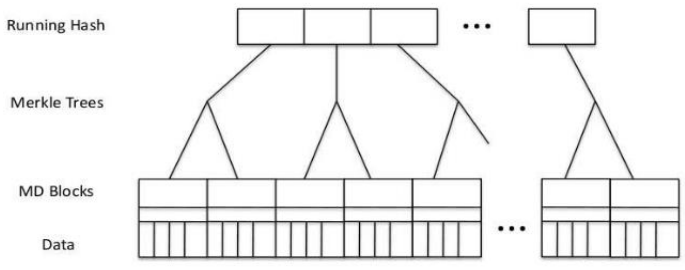
کیارش گل‌زاده

تابستان ۱۳۹۸

فهرست مطالب

مقدمه

بیت‌کوین یک رمز ارز و نظام پرداخت جهانی مبتنی بر تکنولوژی بلاک‌چین می باشد .
رمز نگاری یا الگوریتم‌های هش همان چیزی است که بلاک‌چین بیت کوین را امن نگه می‌دارد. آنها بلاک های سازنده‌ی صنعت رمزنگاری امروز هستند .
برای شروع مهم است که یک دید کلی از عملکرد هش و آنچه انجام می شود، داشت .
الگوریتم هش اطلاعات را در هر اندازه ای (عدد، حروف، فایل‌های رسانه ای) دریافت کرده و آنها را به یک رشته از اعداد و حروف تبدیل می‌کند. این اندازه بیت ثابت می تواند متفاوت باشد (بستگی به تابع هش مورد استفاده دارد)
بلاک‌چین بیت‌کوین از الگوریتم (SHA256 - Secure Hash algorithm) استفاده می‌کند.
هر بلاک دارای یک هدر یا سرصفحه منحصر به فرد است و هر بلاک توسط هش هدر بلاک شناسایی می‌شود. برای موفقیت در استخراج بلاک، لازم است که یک استخراج کننده هدر بلاک هش را به‌گونه ای تنظیم کند که هدف مورد نظر به دست آید. این هدف در حال حاضر این است که هش SHA-256 از هدر بلاک باید یک رشته الفبایی و عددی ۲۵۶ بیتی باشد و باید با ۱۸ صفر شروع شود. هدف به عنوان سختی، در هر ۲۰۱۶ بلاک تغییر می‌کند.
الگوریتم SHA-256 عضو الگوریتم های SHA-2 می باشد که جانشین SHA-2 شده است و برای عملیات هش امن طراحی شده است.
با گذشت زمان حملات اینترنتی به میزان قابل توجهی افزایش می یابد، زیرا هزینه پردازش کامپیوتر کاهش می‌یابد، در نتیجه همواره باید به دنبال الگوریتم های امن‌تر بود (هیچ الگوریتم هش کردن قادر به حفظ سطح بالایی از امنیت حتی برای یک دهه نیز نمی باشد) این به این معنی است که رمزنگاران برای جانشین SHA-2 الگوریتم SHA-3 را در نظر گرفته‌اند.
خانواده‌ی SHA-3 آخرین عضو خانواده الگوریتم های امنیت هش است و همچنین زیرمجموعه ای از خانواده ابتدایی رمزنگاری گسترده Keccak است. الگوریتم ECHO، زیرمجموعه ای از SHA-3 می‌باشد که بر پایه‌ی الگوریتم‌های AES است (Advance Encryption Standard) که یک پیغام و SALT را به عنوان ورودی دریافت می‌کند و هش هر طول از ۱۲۸ تا ۵۱۲ را تولید می‌کند.
الگوریتم ECHO همچنین از روش کار Mekle-Damgard در عملیات فشرده‌سازی استفاده می‌نماید؛ بدین شکل که پیغام ابتدایی به چند زیرمجموعه شکسته می‌شود و هرکدام جداگانه و به صورت ترتیبی هش می‌شوند.
الگوریتم ECHO به طور بسیار ساده طراحی شده است زیرا تنها واحد فشرده سازی، عنصر اصلی سازنده ی ECHO است ولی در عین سادگی چون از دسته الگوریتم های AES است امنیت بالایی در آن نهفته است.
در ادامه به طور مبسوط و مستند به این الگوریتم پرداخته شده است.



شکل ۱ : شمای الگوریتم

شرح کلی الگوریتم

هدف اصلی در این الگوریتم، پیدا کردن یک عدد به عنوان Hash برای یک پیغام (رشته) ورودی است. این روش هش کردن، انواع مختلفی دارد، که عبارت است از ECHO-224، ECHO-256، ECHO-384 و ECHO-512. برای اجرای هر یک از این الگوریتم‌ها، رشته‌ی ورودی باید مشخصات به خصوصی داشته باشد.

روش کار الگوریتم ECHO به این شرح است که ابتدا رشته‌ی ورودی ($\mathscr{M}$) دریافت می‌شود. این رشته باید به چند قسمت تقسیم شود، سپس به ترتیب روی هر یک از این قسمت‌ها، تابع مشخصی صدا زده شود و از خروجی آن برای صدا زدن این تابع روی قسمت بعدی ورودی استفاده شود. در نتیجه، در ابتدای امر باید $\mathscr{M}$ با انجام تغییراتی، به رشته‌ی ثانویه‌ی $\mathscr{M}'$ تبدیل شود.

پارامترهایی که برای اجرای این الگوریتم وجود دارد، به شرح زیر است:

V_{i-1}: خروجی تابع ذکر شده در مرحله قبل (که طول آن CSIZE نامیده می‌شود)

M_i: هر قسمت از پیام (که قسمت بندی شد و مرحله به مرحله از آن‌ها استفاده می‌شود) و طول آن MSIZE نامیده می‌شود ($MSIZE = 2048 - CSIZE$)

C_i: تعداد بیت‌هایی از پیام اولیه ($\mathscr{M}$) که هنوز هش نشده است.

مقدار دلخواهی به نام SALT.

همچنین تعداد بیت‌هایی که به عنوان خروجی هش اعلام می‌شود، HSIZE نامیده می‌شود.

برای اجرای الگوریتم برای HSIZE های تا ۲۵۶ بیت، از تابعی به نام $COMPRESS_{512}$ استفاده می‌شود، به این صورت که داریم:

$$V_i = COMPRESS_{512}(V_{i-1}, M_i, C_i, SALT)$$

و برای HSIZE های بیت ۲۵۷ تا ۵۱۲ بیت، از تابعی به نام $COMPRESS_{1024}$ استفاده می‌شود.

$$V_i = COMPRESS_{1024}(V_{i-1}, M_i, C_i, SALT)$$

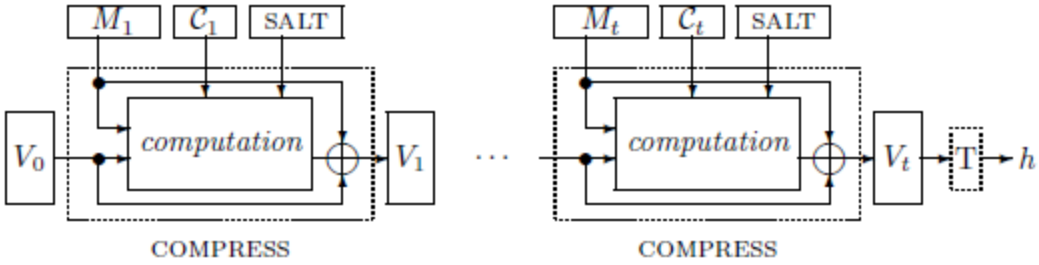
این توابع در ادامه توضیح داده می‌شود.

پارامتر های ذکر شده، برای هر یک از انواع الگوریتم‌های ECHO به شکل زیر است:

hash length (HSIZE)	*uses compression function*	*chaining variable* (CSIZE)	*message block* (MSIZE)	*counter length*	*salt length*
224	COMPRESS_{512}	512	1536	64 or 128	128
256	COMPRESS_{512}	512	1536	64 or 128	128
384	COMPRESS_{1024}	1024	1024	64 or 128	128
512	COMPRESS_{1024}	1024	1024	64 or 128	128

جدول ۱ : پارامترهای مربوط به انواع مختلف *echohash*

شمای کلی الگوریتم به این شکل است:



شکل ۲: شمای کلی الگوریتم *echohash*

که در این شکل کاملاً مشخص است که خروجی هر قسمت، به عنوان ورودی قسمت بعدی استفاده می‌شود.

شروع کار

مقدار اولیه C_0 را برابر ۰ قرار می‌دهیم. همچنین V_0 به ازای انواع مختلف الگوریتم ECHO به شرح زیر مقداردهی می‌شود. مثلا در ECHO-512، V_0 را به هشت قسمت ۱۲۸ بیتی تقسیم می‌کنیم.

$$V_0^i = 00020000\ 00000000\ 00000000\ 00000000$$

سپس پردازش‌هایی روی $\mathscr{M}$ انجام می‌شود، تا طول آن مضربی از MSIZE شود. اگر طول کل پیام برابر با L باشد، آن‌گاه این عملیات روی $\mathscr{M}$ انجام می‌شود تا به $\mathscr{M}'$ تبدیل شود.

یک بیت ۱ به انتهای $\mathscr{M}$ اضافه می‌شود.

به تعداد x تا بیت ۰ به انتهای $\mathscr{M}$ اضافه می‌شود، که x برابر است با:

$$x = MSIZE - \big((L + 144) mod MSIZE\big) - 1$$

نمایش ۱۶ بیتی دودویی HSIZE به رشته اضافه می‌شود.

نمایش ۱۲۸ بیتی L (طول رشته) به رشته اضافه می‌شود.

و رشته نهایی $\mathscr{M}$ نامیده می‌شود.

مراحل اصلی

در پروژه، از ECHO-512 استفاده می‌شود، که این الگوریتم از $COMPRESS_{1024}$ استفاده می‌کند. در ادامه، این تابع شرح داده می‌شود و از آن به عنوان مولد هش استفاده می‌شود. در این الگوریتم داریم:

$$V_i = COMPRESS_{1024}(V_{i-1}, M_i, C_i, SALT)$$

$\mathscr{M}$ به قسمت‌های ۱۰۲۴ بیتی تقسیم بندی می‌شود (M_i). همچنین V_i برابر با ۸ کلمه‌ی ۱۲۸ بیتی قرار داده می‌شود $(v_i^0 \dots v_i^7)$. هر یک از M_i ها نیز به ۸ کلمه‌ی ۱۲۸ بیتی $(m_i^0 \dots m_i^7)$ تقسیم می‌شود. یک ماتریس به شکل زیر تولید می‌شود و اعمالی موسوم به AES ده مرتبه روی آن‌ها اجرا می‌شود.

v_{i-1}^0	v_{i-1}^4	m_i^0	m_i^4
v_{i-1}^1	v_{i-1}^5	m_i^1	m_i^5
v_{i-1}^2	v_{i-1}^6	m_i^2	m_i^6
v_{i-1}^3	v_{i-1}^7	m_i^3	m_i^7

جدول ۲: ماتریس تشکیل داده شده در *Compress*

اعمال AES

اعمال AES (Advanced Encryption Standard)، اعمالی استاندارد به منظور رمزنگاری و هش کردن داده‌ها هستند که در الگوریتم ECHO (در واقع در تابع Compress) به کار گرفته می‌شوند.

در تابع $COMPRESS_{1024}$، ده مرتبه به ترتیب سه تابع زیر اجرا می‌شوند: (S همان ماتریس تشکیل داده شده است)

$$SubWords(S, SALT, \kappa)$$
$$ShiftRows(S)$$
$$MixColumns(S)$$

و سپس تابع $Final$ اجرا می‌شود.

تابع SubWords

نحوه کار این تابع، به این صورت است:

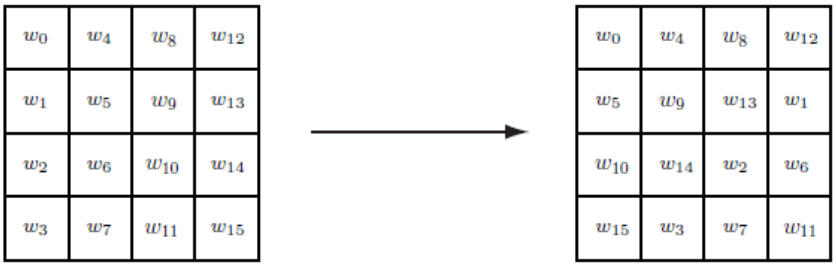
فرض کنید κ اولیه، همان مقدار C_i باشد. تابع ابتدا دو پارامتر k1 و k2 را تشکیل می‌دهد، به این شکل که k1 برابر است با κ که جلوی آن ۶۴ بیت صفر قرار گرفته است، و k2 نیز برابر با SALT است. یک تابع کمکی استاندارد به نام $AES(w, k)$ موجود است که با استفاده از یک Look-Up Table و مقدار ورودی k، ورودی w را به یک داده‌ی دیگر map می‌کند. تابع SubWords، به ازای هر خانه w_i از ماتریس S، آن خانه را با مقدار w_i' جایگزین می‌کند، که

$$w_i' = AES(AES(w_i, k_1), k_2)$$

و سپس κ را یک واحد افزایش می‌دهد (که مطابق با آن، k_1 و k_2 نیز به‌روزرسانی می‌شوند).

تابع ShiftRows

این تابع به این صورت عمل می‌کند که سطر اول ماتریس S را بدون تغییر باقی می‌گذارد، سطر دوم را یک واحد، سطر سوم را دو واحد، و سطر چهارم را سه واحد به چپ شیفت می‌دهد. این عملیات در شکل زیر قابل مشاهده است.



شکل ۳: تابع *ShiftRows*

تابع MixColumns

این تابع، یک سری عملیات را روی هر ستون از ماتریس S انجام می‌دهد. به ازای یک ستون از این ماتریس، فرض کنید اعضای آن از بالا به پایین w_0 الی w_3 باشند. ابتدا هر یک از این کلمه‌ها را به صورت بایت‌بایت جدا می‌کند.

$$w_0 = (B_0, B_1, \dots, B_{15})$$
$$w_1 = (B_{16}, B_{17}, \dots, B_{31})$$
$$w_2 = (B_{32}, B_{33}, \dots, B_{47})$$
$$w_3 = (B_{48}, B_{49}, \dots, B_{63})$$

حال به ازای $0 \le j \le 15$، بایت‌های این کلمات به این صورت جایگزین می‌شوند:

$$\begin{pmatrix} B'_j \\ B'_{16+j} \\ B'_{32+j} \\ B'_{48+j} \end{pmatrix} = \begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix} \cdot \begin{pmatrix} B_j \\ B_{16+j} \\ B_{32+j} \\ B_{48+j} \end{pmatrix}$$

تابع Final

در این تابع، مقادیر جدید v_i محاسبه می‌شوند تا در سری بعدی اجرای $Compress$، استفاده شوند. این مقادیر به این صورت محاسبه می‌شوند:

$$v_i^0 = v_{i-1}^0 \oplus m_i^0 \oplus w_0 \oplus w_8$$
$$v_i^1 = v_{i-1}^1 \oplus m_i^1 \oplus w_1 \oplus w_9$$
$$v_i^2 = v_{i-1}^2 \oplus m_i^2 \oplus w_2 \oplus w_{10}$$
$$v_i^3 = v_{i-1}^3 \oplus m_i^3 \oplus w_3 \oplus w_{11}$$
$$v_i^4 = v_{i-1}^4 \oplus m_i^4 \oplus w_4 \oplus w_{12}$$
$$v_i^5 = v_{i-1}^5 \oplus m_i^5 \oplus w_5 \oplus w_{13}$$
$$v_i^6 = v_{i-1}^6 \oplus m_i^6 \oplus w_6 \oplus w_{14}$$
$$v_i^7 = v_{i-1}^7 \oplus m_i^7 \oplus w_7 \oplus w_{15}$$

محاسبه مقدار نهایی Hash

پس از اتمام اعمال توابع روی تمام قسمت‌های پیام ورودی، هشِ نهایی با کنار هم قرار دادن بیت‌های v_iها به دست می‌آید.

$$Hash = v_t^0 \parallel v_t^2 \parallel \mathrm{v}_\mathrm{t}^3 \parallel \mathrm{v}_\mathrm{t}^4 \parallel \mathrm{v}_\mathrm{t}^5 \parallel \mathrm{v}_\mathrm{t}^6 \parallel \mathrm{v}_\mathrm{t}^7$$

که $\parallel$ نماد Concatenation است.

در صورتی که یک مقدار هش با طول HSIZE مورد انتظار باشد، می‌توان از این مقدار $Hash$ محاسبه شده، به اندازه‌ی دلخواه، بیت‌های مورد نظر را برداشت. برای مثال، اگر $HSIZE = 512$ باشد، هشِ نهایی برابر می‌شود با:

$$h = v_t^0 \parallel \mathrm{v}_\mathrm{t}^1 \parallel \mathrm{v}_\mathrm{t}^2 \parallel \mathrm{v}_\mathrm{t}^3$$

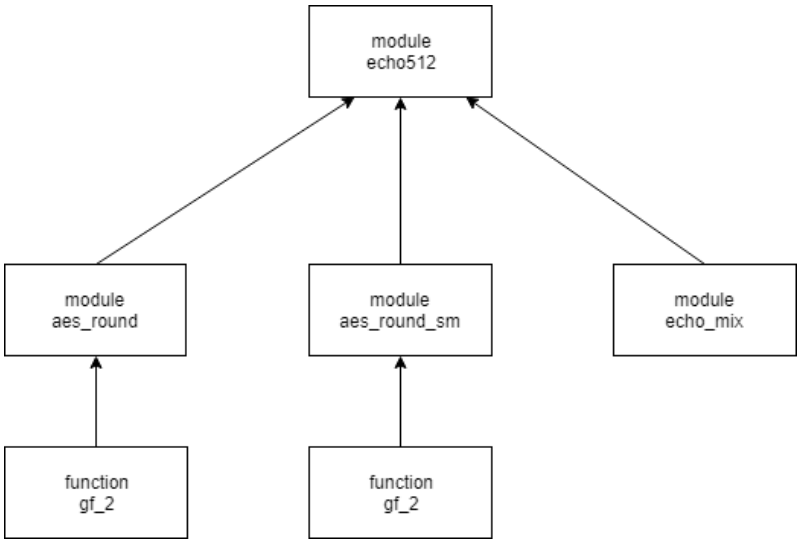
توصیف معماری سیستم

معماری کلی سیستم

در کد سخت‌افزاری، وظیفه عملیات hash بر عهده‌ی ماژول echo512 است که در سلسله مراتب معماری، ماژول Top است. درون این ماژول، ماژول‌های aes_round و echo_mix تعریف شده اند که وظیفه انجام قسمت‌های مختلف hash را دارند. این الگوریتم به دسته الگوریتم‌های اسفنجی تعلق دارد که در آن خروجی‌ها به صورت زنجیره به عنوان ورودی به مرحله‌ی بعدی داده می‌شوند.
کلاک ماژول‌های درونی توسط کلاک اصلی تغذیه می‌شوند.

نمودار درختی اجزا

ساختار سلسله مراتبی اجزا در کد سخت افزاری در شکل زیر آورده شده است.



شکل ٤: نمودار درختی اجزا

ماژول echo512

```
module echo512 (
    input clk,
    input [511:0] data,
    output [31:0] hash
);
```

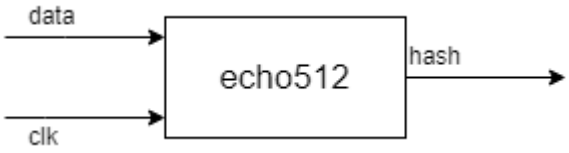
رابط کاربری

ماژول echo512 دارای سه مؤلفه می‌باشد:

۱. ورودی data که به عنوان داده ای که قرار است hash شود.

۲. ورودی clk که کلاک سیستم است.

۳. خروجی hash که مقداری است که برنامه به عنوان مقدار hash شده به ما برمی‌گرداند.



شکل ۵: بلوک دیاگرام ماژول *echo512*

مقدمه

ماژول echo512 با کمک ماژول های aes_round_sm و aes_round و echo_mix داده ورودی را به چندین بخش تقسیم می‌کند و هربار طی مراحلی با پاس دادن بخشی از داده ورودی به ماژول‌ها، خروجی مرحله بعد را پیدا می‌کند و در نهایت مقدار نهایی hash را پیدا می‌کند.

عملکرد

ابتدا یک vector از جنس reg تعریف و نام آن msg گذاشته می‌شود که حجمی برابر ۵۱۲ بیت دارد.

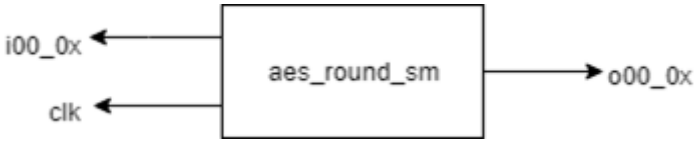
برای پر کردن آن هشت بیت هشت بیت از پرارزش‌ترین بخش data جدا می‌شود و در هشت بیت کم ارزش msg قرار داده می‌شود و برعکس به این معنا که هشت بیت کم ارزش data در هشت بیت پر ارزش msg قرار داده می‌شود. بدان معنی که هر هشت بیت data یک واحد در نظر گرفته شده و data به صورت معکوس در msg ریخته می‌شود. به مثال زیر توجه شود:

```
msg[  7:  0] <= data[511:504];
msg[511:504] <= data[  7:  0];
```

اکنون ابتدا Round 0 بررسی خواهد شد و پس از توضیح روابط آن با Round 1، نتایج به همه ی round ها تعمیم داده خواهد شد.

- Round 0

ماژول aes_round_sm دو ورودی می‌گیرد و در نهایت خروجی o00_0x را خروجی می‌دهد. حال که ورودی ها بخش اول round 0 بررسی می‌شود.(i00_0x)



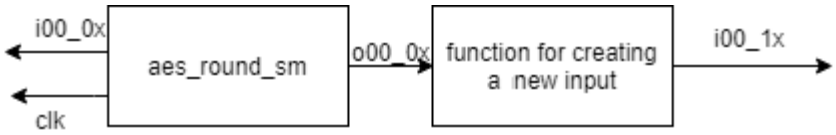
شکل ٦: بلوک دیاگرام ماژول *aes_round_sm*

i00_03 , i00_01 , i00_00 به صورت مشابه مقدار دهی می‌شوند. به طول مثال، برای i00_02:

```
i00_02 ={msg[ 8*32,9*32], msg[ 9*32,10*32], msg[ 10*32,11*32], msg[ 11*32,12*32]}
```

حال i00_1x بررسی خواهد شد.

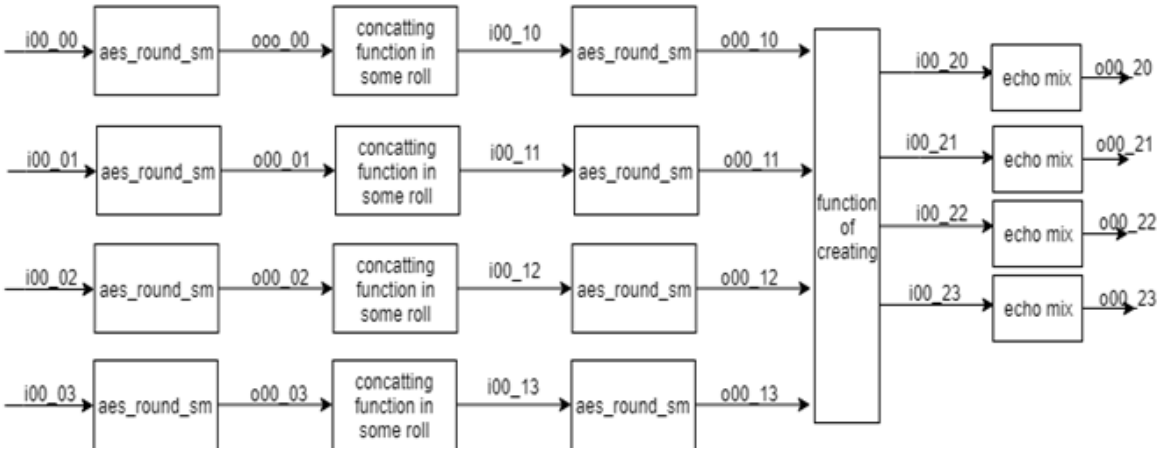
```
i00_1x = {o00_0x[127:108], (o00_0x[107:96] ^ 12'h020(x*5+0)), (o00_0x[95:0])}
```



شکل ۷ : دیاگرام درونی *round 0*

این اعداد نیز به مانند i00_0x به ماژول aes_round_sm فرستاده می‌شود تا خروجی o00_1x مشاهده شود.
حال بخش سوم از مرحله اول از round 0 بررسی می‌شود .
در این مرحله به چگونگی ساخته شدن i00_2x پرداخته خواهد شد.

برای ساختن i00_20 هشت بیت اول تمام خروجی های o00_1x با هم concat می‌شود و سپس روی هشت بیت دوم همین کار تکرار می‌شود و سپس هشت بیت سوم و در انتها هشت بیت چهارم و در پایان همه ی این موارد با یک‌دیگر concat می‌شوند.
برای i00_21 از هشت بیت پنجم تا هشت بیت هشتم این روند وجود دارد و برای i00_22 از هشت بیت نهم تا هشت بیت دوازدهم و در پایان از هشت بیت سیزدهم تا هشت بیت شانزدهم این روند وجود خواهد داشت.
باید توجه کرد که برای یافتن o00_2x باید i00_2x را به ماژول echo_mix پاس داد، برخلاف دو مورد قبلی که i00_xy را به ماژول aes_round_sm پاس داده می‌شد.



شکل ۸: دیاگرام ساخت *i00_2x*

در مرحله دوم از round 0، به مانند مرحله یک ابتدا i01_0x ها initialize می‌شوند.

```
i01_00 <= 128'h00000200000000000000000000000000
i01_01 <= {msg[4*32:5*32], msg[5*32:6*32], msg[6*32 :7*32], msg[7*32 : 8*32] }
```

```
i01_02 <= 128'h00000000000000000000000002000000
i01_03 <= 128'h00000200000000000000000000000000
```
مانند مرحله قبل با دادن هر ورودی به ماژول‌های aes_round_sm برای موارد i01_0x,i01_1x و همچنین دادن i01_2x به ماژول echo_mix خروجی یافت می‌شود.

ورودی به این‌گونه دریافت می‌شود:

```
i01_1x = {o01_00[127:108], o01_00[107:96] ^ 12'h020((5*x+4) mod 16), o01_00[95:0]}
```
همچنین i01_2x دقیقاً به مانند مرحله اول از خروجی‌های o01_1z حساب می‌شود.

در مرحله سوم از round 0، خروجی هر بخش به مانند دو مرحله قبلی یافت می‌شود. ورودی ها به اختصار بررسی خواهند شد.

```
i02_00 <= { msg[((0*4)+0)*32 +: 32], msg[((0*4)+1)*32 +: 32], msg[((0*4)+2)*32 +:
32], msg[((0*4)+3)*32 +: 32] };
i02_01 <= 128'h00000000000000000000000000000000
i02_02 <= 128'h00000200000000000000000000000000
i02_03 <= 128'h00000200000000000000000000000000
i02_1x <= { o02_00[127:108], o02_00[107:96] ^ 12'h020((5x+8)mod16), o02_00[95:0]
};
```
i02_2x هم به مانند مراحل قبل به دست می‌آید.

در مرحله چهارم از round 0، خروجی هر بخش به مانند سه مرحله قبلی یافت می‌شود. ورودی ها به اختصار بررسی خواهند شد.

```
i03_00 <= 128'h00000080000000000000000000000000;
i03_01 <= 128'h00000200000000000000000000000000;
i03_02 <= 128'h00000200000000000000000000000000;
i03_03 <= { msg[12*32 :13 *  32], msg[13*32 :14 *32], msg[15*32: 16 *32],
msg[16*32+:17*32]};
i03_1x = {o03_00[127:108], o03_00[107:96] ^ 12'h020((5x+12)mod 16), o03_00[95:0]};
```
i03_3x هم به مانند مرحله یک محاسبه خواهد شد.

- Round 1

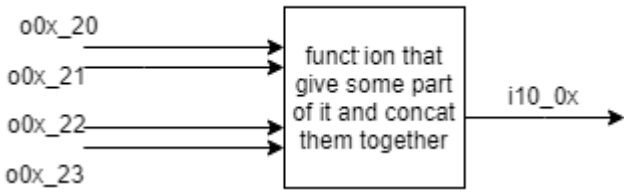
در این round برای تعریف i10_0x بر خلاف مرحله قبل که i00_0x به صورت مقدار دهی اولیه، مقدار دهی می‌شدند، در این مرحله i10_0x از نتیجه خروجی مرحله آخر در round 0 بدست می‌آیند. به نمونه زیر توجه شود.

```
i10_00 <= { o00_20[127:120], o00_20[95:88], o00_20[63:56], o00_20[31:24],
o00_21[127:120], o00_21[95:88], o00_21[63:56], o00_21[31:24], o00_22[127:120],
o00_22[95:88], o00_22[63:56], o00_22[31:24], o00_23[127:120], o00_23[95:88],
o00_23[63:56], o00_23[31:24] };

i10_01 <= { o01_20[119:112], o01_20[87:80], o01_20[55:48], o01_20[23:16],
o01_21[119:112], o01_21[87:80], o01_21[55:48], o01_21[23:16], o01_22[119:112],
o01_22[87:80], o01_22[55:48], o01_22[23:16], o01_23[119:112], o01_23[87:80],
o01_23[55:48], o01_23[23:16] };

i10_02 <= { o02_20[111:104], o02_20[79:72], o02_20[47:40], o02_20[15: 8],
o02_21[111:104], o02_21[79:72], o02_21[47:40], o02_21[15: 8], o02_22[111:104],
o02_22[79:72], o02_22[47:40], o02_22[15: 8], o02_23[111:104], o02_23[79:72],
o02_23[47:40], o02_23[15: 8] };
```

```
i10_03 <= { o03_20[103: 96], o03_20[71:64], o03_20[39:32], o03_20[ 7: 0],
o03_21[103: 96], o03_21[71:64], o03_21[39:32], o03_21[ 7: 0], o03_22[103: 96],
o03_22[71:64], o03_22[39:32], o03_22[ 7: 0], o03_23[103: 96], o03_23[71:64],
o03_23[39:32], o03_23[ 7: 0] };
```



شکل ۹: به دست آوردن ورودی از خروجی‌های مرحله قبل

از این مرحله به بعد تمام کار ها به صورت قبل انجام می‌شود و هر ورودی از خروجی مرحله قبل به دست می‌آید و همین روال تا round 9 انجام می‌شود.

پس از آن‌که تمام خروجی های round 9 محاسبه شد، ماژول echo مقدار hash را به طور کامل محاسبه خواهد کرد.

برای اینکار ۹۵ تا vector تعریف می‌شود که همه آنها از جنس reg می‌باشند و اندازه هرکدام از آن ها 32 بیت است.

همچنین دو vector به نام‌های A0 و A2 تعریف می‌شود.

با توجه به اینکه در انتهای کار round 9 به طور کامل انجام شده و مقادیر تعریف شده برای آن از حالت x خارج شده است , مقدار A0 , A2 به صورت زیر تعریف می شود.

```
A0 <= { o90_23[119:112], o90_23[87:80], o90_23[55:48], o90_23[23:16] };
A2 <= { o92_23[119:112], o92_23[87:80], o92_23[55:48], o92_23[23:16] };
```

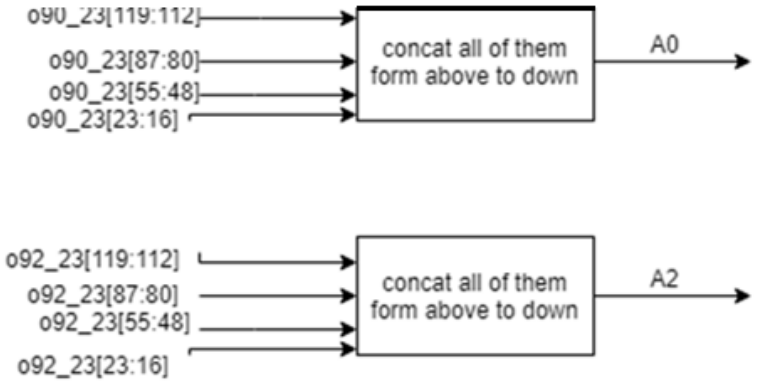
برای به دست آوردن مقدار نهایی hash علاوه بر دو مقدار بالا، به قسمتی از ورودی هم نیاز می باشد؛ که این قسمت در واقع قسمتی از msg به صورت زیر خواهد بود .

```
msg[255:224] = {msg[255:248], msg[247:240], msg[239:232], msg[231:224]}
msg[231:224] <= data[287:280];
msg[239:232] <= data[279:272];
msg[247:240] <= data[271:264];
msg[255:248] <= data[263:256];
```

طی ۹۴ بار شیفت دادن مقدار ذکر شده در بالا در m94 ریخته می‌شود. یعنی ابتدا مقدار بالا در m00 ریخته می‌شود سپس در کلاک بعدی به m01 و به همین شکل انتقال داده می‌شود تا به m94 برسد.

مقدار نهایی hash به صورت زیر به دست خواهد آمد.

```
h <= A0 ^ A2 ^ m94
```

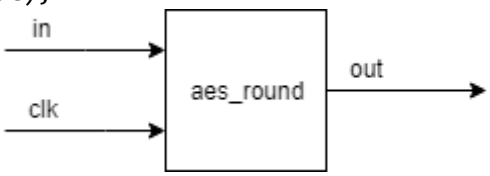


شکل ۱۰ : نحوه ساخت $A0$ و $A2$

شکل ۱۱: نحوه محاسبه $hash$ از روی $A0$، $A2$ و $m94$

ماژول aes_round

```
module aes_round (
    input clk,
    input [127:0] in,
    output reg [127:0] out);
```



شکل ۱۲ : بلوک دیاگرام ماژول *aes_round*

رابط کاربری

۱. ورودی clk: کلاک سیستم است.
۲. ورودی in: یک بردار ۱۲۸ تایی که داده‌ای است که عملیات روی آن صورت می‌پذیرد.
۳. خروجی out: برداری ۱۲۸ تایی که حاوی نتیجه عملیات است.

مقدمه

این ماژول داده‌ها را دریافت می‌کند و با انجام یک سری عملیات خروجی نتیجه می‌دهد. این ماژول از تابع s_box استفاده می‌کند و خروجی آن را به ازای مقادیر مختلف ذخیره می‌کند و با انجام XOR بر روی آن ها و دیتای اصلی، قسمت های ۳۲تایی از خروجی را محاسبه می‌کند.

عملکرد

این ماژول حاوی جدول ثابت است که از آن به عنوان تابع استفاده می‌کند.
جدول s_box یک آرایه ۲۵۶ خانه‌ای از بردارهایی ۸ بیتی است:

```
wire [7:0] s_box[0:255];
```

در ادامه این جدول مقداردهی شده است:

```
assign s_box[0] = 8'h63;
assign s_box[1] = 8'h7C;
assign s_box[2] = 8'h77;
```

سپس مقادیر متناسب با تکه های ۸ بیتی از ورودی (in) را از جدول پیدا کرده و در خانه های ماتریس sb قرار می‌دهد:

```
assign sbF = s_box[in[63:56]];
assign sbE = s_box[in[95:88]];
assign sbD = s_box[in[127:120]];
assign sbC = s_box[in[31:24]];
assign sbB = s_box[in[87:80]];
assign sbA = s_box[in[119:112]];
assign sb9 = s_box[in[23:16]];
assign sb8 = s_box[in[55:48]];
assign sb7 = s_box[in[111:104]];
assign sb6 = s_box[in[15:8]];
assign sb5 = s_box[in[47:40]];
assign sb4 = s_box[in[79:72]];
```

```
assign sb3 = s_box[in[7:0]];
assign sb2 = s_box[in[39:32]];
assign sb1 = s_box[in[71:64]];
assign sb0 = s_box[in[103:96]];
```

سپس ماتریس‌های 16 تایی g1 و g2 تعریف می شوند:

```
wire [7:0] g1_0, g1_1, g1_2, g1_3, g1_4, g1_5, g1_6, g1_7, g1_8, g1_9, g1_A, g1_B, g1_C,
g1_D, g1_E, g1_F;
wire [7:0] g2_0, g2_1, g2_2, g2_3, g2_4, g2_5, g2_6, g2_7, g2_8, g2_9, g2_A, g2_B,
g2_C, g2_D, g2_E, g2_F;
```

sb ها در g1 ها کپی می‌شوند و خروجی تابع gf_2 (کاهیده دو برابر آن ها در فضای گالوای $GF[2^8]$) را در g2 ها ذخیره می‌کند:

```
assign g2_0 = gf_2(sb0);
assign g1_0 = sb0;
```

در اخر با عملیات XOR بین خانه‌های معین این ماتریس‌ها، ماتریس خروجی(o) را می‌سازد:

```
reg[7:0] o0,o1,o2,o3,o4,o5,o6,o7,o8,o9,oA,oB,oC,oD,oE,oF;
always @ ( * ) begin
        o0 <= g2_C ^ g2_0 ^ g1_0 ^ g1_4 ^ g1_8;
        o1 <= g2_8 ^ g2_C ^ g1_C ^ g1_0 ^ g1_4;
       ...
end
```

- ماژول aes_round_sm مشابه این ماژول است. با این تفاوت که نتایج آن‌ها در دو مرحله خروجی داده می‌شود. ابتدا در یک رجیستر میانی و سپس در خروجی ریخته می‌شود.

```
sbFx <= s_box[in[63:56]];
sbEx <= s_box[in[95:88]];
sbDx <= s_box[in[127:120]];
…

// Extra FF To Allow Use Of Block Ram Output Register
sbF <= sbFx;
sbE <= sbEx;
sbD <= sbDx;
…
```

ماژول echo_mix

```
module echo_mix (
    input clk,
    input [127:0] in,
    output reg [127:0] out);
```

شکل ۱۳: بلوک دیاگرام ماژول *echo_mix*

رابط کاربری

۱. ورودی clk: کلاک سیستم است.
۲. ورودی in: داده ورودی به صورت برداری ۱۲۸ تایی می باشد.
۳. خروجی out: خروجی ماژول که به صورت برداری ۱۲۸ تایی است.

مقدمه

این ماژول با دریافت ورودی، آن را به هم می‌ریزد .به این صورت که جدولی از روی XOR خانه های مجاور هر سطر ورودی می‌سازد. در نهایت با جابجایی آنها خروجی را تولید می‌کند.

عملکرد

این ماژول ورودی و خروجی ۱۲۸ بیتی دارد و به کلاک متصل می‌شود.

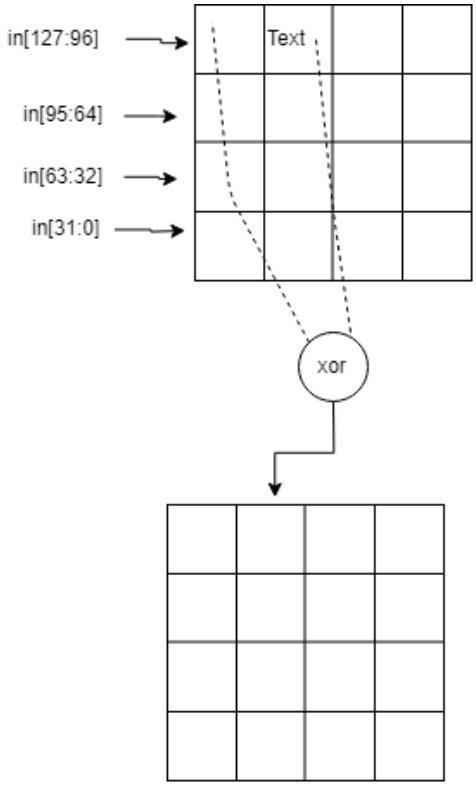
```
wire [7:0] ax00, ax01, ax02, ax03
wire [7:0] ax10, ax11, ax12, ax13
wire [7:0] ax20, ax21, ax22, ax23
wire [7:0] ax30, ax31, ax32, ax33
```

سپس ماتریس ax را با خانه های ۸ بیتی می‌سازد و ورودی را در آن مرتب می‌کند.

```
assign { ax03, ax02, ax01, ax00 } = in[127:96]
```

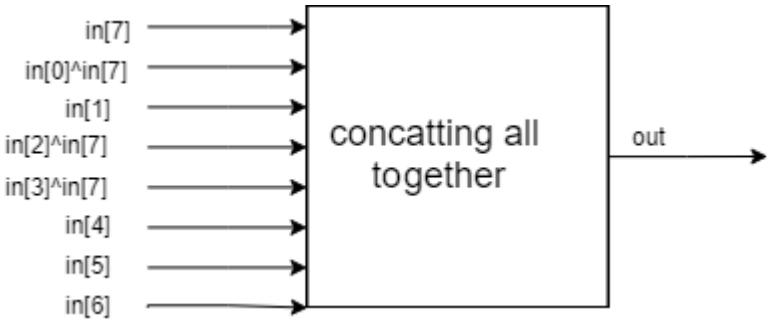
یعنی ۳۲ بیت آخر را در سطر اول و ۳۲ بیت دوم در سطر دوم و به همین ترتیب در ax قرار می‌دهد.
سپس دو ماتریس a , b را می سازد. ماتریس ax را در a کپی می‌کند و ماتریس b را هم از ترکیب خانه های ax می‌سازد. به این ترتیب که:

$$b_{i,j} = ax_{i,j-1} \oplus ax_{i,j}$$



شکل ۱٤ :نحوه XOR بین خانه ها

سپس برای ساخت ۳۲ بیت چهارم خروجی (out) مقادیر سطر اول، ۳۲ بیت سوم از مقادیر سطر دوم و ... استفاده می‌کند.



شکل ۱۵: دیاگرام ساخت خروجی ماژول

نحوه‌ی اجرای کد سخت‌افزاری

۱ ـ ابتدا در مرحله 0 داده ها initial می‌شوند و داده ها با استفاده از یک سری مقدار از پیش تعریف شده (و همچنین با استفاده از بخشی از داده ورودی که به تفصیل توضیح داده خواهد شد) مقداردهی خواهندشد و آماده پردازش می‌شوند.

۲ ـ ۱۰ بار ماژول aes_round و echo_mix استفاده می‌شود و با هر I به یک o رسیده می‌شود. هر کدام از این I ها در واقع خروجی مرحله قبل می‌باشد که با پاس دادن آن به ماژول های مربوطه خروجی آن مرحله به دست می‌آید که با انجام یک سری عملیات بر روی آنها، تبدیل به ورودی مرحله بعد می‌شوند.

۳ـ در آخر از آنجایی که هر کدام از این مقادیر ۱۲۸بیتی هستند؛ در صورتی که خروجی ۳۲ بیتی باشد، لازم است بخشی از ورودی طی قاعده ای با بخشی از خروجی ها ترکیب شود پس به صورت مقابل عمل می‌شود. مقدار نهایی خروجی در round 9 و همچنین بخشی از msg که توضیح داده خواهد شد با یکدیگر xor می‌شوند و خروجی نهایی شکل می‌گیرد. به این صورت که طی یک عملیات، ۳۲ بیت از مرحله اول خروجی در round آخر برداشته می‌شود و با ۳۲ بیت از مرحله آخر خروجی در round آخر و همچنین با ۳۲ بیت از داده ورودی xor می‌شوند و خروجی نهایی حاصل می‌شود.

روند شبیه سازی و نتایج آن

تحلیل کد مدل طلایی

کد مدل طلایی (که به زبان C نوشته شده است) در مواردی جزئی با الگوریتم معیار EchoHash تفاوت‌هایی دارد. برای استفاده از توابع echohash.c، یک فایل exec.c نوشته شد که با دادن ورودی به تابع echohash، خروجی لازم را دریافت و با تابع printBits، بیت‌های آن را چاپ می‌کند. در کد نرم‌افزاری C، یک پوینتر به یک رشته ورودی که می‌خواهیم هش شود به تابع echohash داده می‌شود. این تابع، یک آرایه به نام hash دارد که تمام بیت‌های خروجی الگوریتم echo در آن ریخته می‌شود. یک شیء به نام ctx وجود دارد که تمام مقادیر مورد نیاز برای هش کردن (از جمله شمارنده های C_i، متغیرهای زنجیره‌ای (Vb)، و یک buffer که پیام‌های ورودی در آن ریخته می‌شود و ۱۰۲۴ تا ۱۰۲۴ تا با هم hash می‌شود، قرار دارد. مقادیر اولیه V_iها در کد C، برابر با ۵۱۲ قرار داده می‌شود. در کدِ C، خبری از وجود padding برای پیام ورودی نیست. و اگر کلّ ورودی در بافر جا شود، دیگر عملیات Compress انجام نمی‌شود و خروجی به نحوی دیگر محاسبه می‌شود. در غیر این صورت، پیام به صورت تکه تکه در بافر قرار گرفته و ضمن افزایش شمارنده C_i، عملیات Compress انجام می‌شود. در عملیات Compress (ماکروی COMPRESS_BIG) آرایه‌ای به نام K وجود دارد که قرار است مقدار ورودی به تابع AES را در خود نگه دارد. در این کد، چون C_i ها و K اعدادی ۱۲۸ بیتی هستند، برای نگه‌داری آن‌ها از یک آرایه استفاده می‌کنیم.

سپس در INPUT_BLOCK_BIG، ماتریس S از روی ورودی و مقادیر V_{i-1} ساخته می‌شود. در ادامه، ۱۰ بار عملیات Round اعمال شده و در نهایت، یک بار عملیات FINAL روی داده‌ها انجام می‌گیرد.

در ماکروی BIG_ROUND، به ترتیب ماکروهای BIG_SUB_WORDS، BIG_SHIFT_ROWS و BIG_MIX_COLUMNS اجرا می‌شود. در BIG_SUB_WORDS، روی هر خانه از ماتریس، دو بار تابع AES صدا می‌شود؛ بار اول با استفاده از شمارنده‌ی K و بار دوم با استفاده از SALT (که SALT در کد سی برابر با صفر قرار داده شده است). لازم به ذکر است که جزئیات مربوط به این قسمت از برنامه از جمله Look-Up Table ها، در فایل aes_helper.c موجود است.

در ماکروی BIG_SHIFT_ROWS، عملیات مربوط به شیفت دادن سطرها و در ماکروی BIG_MIX_COLUMNS، عملیات مربوط به تغییر دادن ستون ها اجرا می‌شود.

در تابع FINAL_BIG، مقدارهای V_i طبق فرمول‌های آورده شده در بالا، آپدیت می‌شود.

در انتهای هش کردن کل پیام، مقدار نهایی Hash (در تابع Close)، در آرایه hash ریخته می‌شود. در انتها، هر تعداد بیت که برای مقدار هش نهایی از آن استفاده شود، با استفاده از memset در مکانی از حافظه (که پوینتر به آن در تابع echohash به عنوان output وجود دارد)، قرار می‌گیرد. در نهایت، مقدار ورودی و هشِ محاسبه شده به صورت باینری چاپ می‌شود.

مشاهده ورودی‌ها و خروجی‌های اصلی و مقادیر میانی

در تست‌های زیر، خروجی‌های میانی نظیر A0، A2 و m94 که خروجی hash از xor آنها به دست می‌آید، قابل مشاهده می‌باشند. (نتایج شبیه سازی ۹ رشته‌ی متفاوت و خروجی آنها در فایل ضمیمه (tests.txt) قابل مشاهده است)

تست ۱

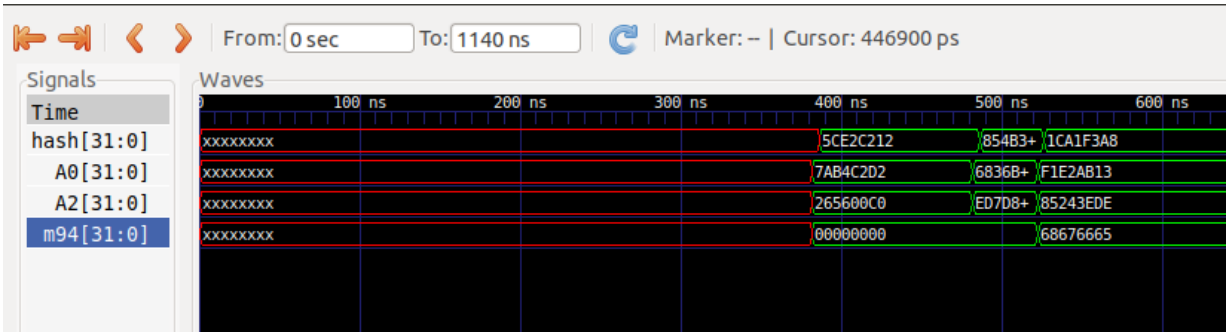
- ورودی (در مبنای ۱۶)

```
61626364656667686162636465666768616263646566676861626364656667686162636465666768616263646566676861626364656667686162636465666768
```

- خروجی

خروجی کد c	0x46D03A01
خروجی کد Verilog	0x1CA1F3AB

- شکل موج



شکل ۱۶: شکل موج تست ۱

تست ۲

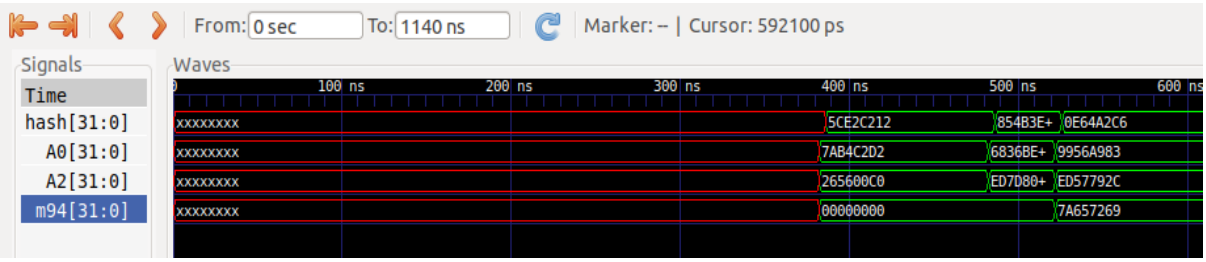
- ورودی (در مبنای ۱۶)

6469676974616C73797374656D64657369676E70726F6A656374616673617269646967697461 6C73797374656D64657369676E70726F6A656374616673617269

- خروجی

خروجی کد c	0xA780AF4C
خروجی کد Verilog	0xE56B9DB7

- شکل موج



شکل ۱۷: شکل موج تست ۲

نحوه عملکرد Testbench

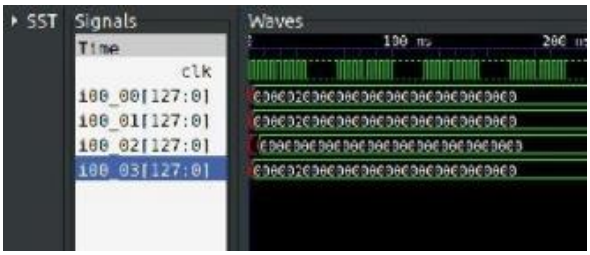
ماژول echo_tribus_tb که همان تست بنچ استفاده شده است، همانند کد اصلی دارای یک پیام ۵۱۲ بیتی به نام data است و مقادیر کلاک نیز در آن اعمال شده است. همچنین در یک initial block، مقادیر ورودی که به کد طلایی داده شده، بر روی data ذخیره می‌گردند. در این ماژول، یک نمونه از ماژول echo512 با نام uut قرار داده شده که ورودی آن data و clock و خروجی آن، hash تولیدی در آن ماژول است که در موج خروجی بررسی و مشاهده گردید.

بررسی نحوه عملکرد کد Verilog

ورودی زیر به عنوان data به کد داده می‌شود.

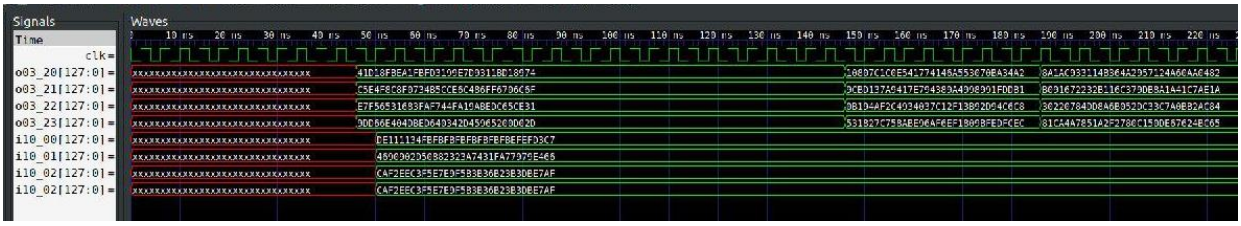
```
data = 512’b0;
always #2 clk = ~clk;
```

به شکل موج زیر توجه شود:



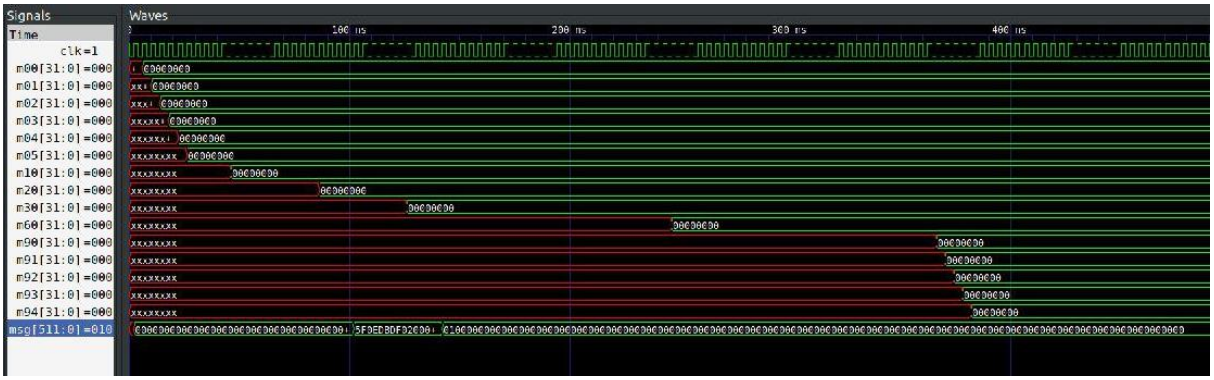
شکل ۱۸: شکل موج داده‌های *initialize* شده

همان طور که در شکل مشخص است ابتدا مقادیر i00_00 تا i00_03 را براساس کد initialize می‌کند فلذا از آنجایی که مقدار آنها به هیچ خروجی قبلی وابسته نیست مقدار x در این رجیستر ها قرار نمی‌گیرد.



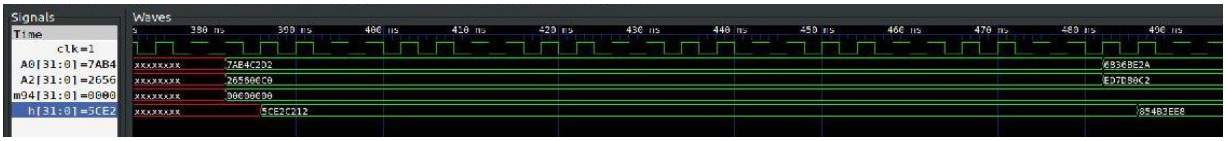
شکل ۱۹: شکل موج ورودی و خروجی‌های *Round 1*

این شکل موج نحوه ورودی گرفتن هر مرحله با توجه به مرحله قبل را توجیه می‌کند. بدین صورت که تا وقتی که خروجی آخرین مرحله قبل از آن پیدا نشده است و به عبارتی بهتر مقدار x را درون خود جای داده است، مقدار موجود در آن x خواهد بود. با بدست آمدن خروجی مرحله قبل و با تاخیر اندک، عملیاتی روی خروجی مرحله قبل اعمال می‌شود و مقدار موجود در ورودی این مرحله از حالت x خارج شده و مقداری جدید و معتبر می‌پذیرد.



شکل ۲۰: شکل موج تاخیر مقادیر m

همان طور که در توضیح کد Verilog ذکر شد، خروجی نهایی ما علاوه در مرحله آخر به بخشی از ورودی بستگی دارد که به صورت msg تبدیل شده و در رجیستری هایی به نام mxx ریخته می‌شود بدین صورت که ابتدا در m00 ریخته می‌شود و در مرحله بعدی به m01 انتقال داده می‌شود و در انتها به m94 منتقل شده و خروجی نهایی بدست می‌آید. شکل بالا این موضوع را توجیه می‌کند بدین گونه که ابتدا مقدار موجود در تمامی m ها به جز m00 که با بخشی از ورودی پر شده x می‌باشد. در کلاک بعدی m01 مقدار موجود در m00 را می‌پذیرد و از حالت x خارج می‌شود اما بقیه رجیسترها همچنان مقدار x را دارند. این روند ادامه می‌یابد تا زمانی که بعد از 94 کلاک مقدار موجود در m93 به m94 انتقال می‌یابد و آن را از حالت x خارج و آماده پردازش برای یافتن مقدار نهایی hash کنند.



شکل ۲۱: شکل موج مقادیر نهایی $hash$

همان طور که در توضیح کد وریلاگ گفته شد، هدف از ۹۴ بار انتقال دادن داده در بین m ها این است که همزمان با دوعامل دیگر در تولید hash یعنی A0 و A2 از مقدار x خارج شود . شکل موج بالا این مساله را نیز توجیه می‌کند بدین‌صورت که هر سه این عوامل که A0 و A2 از نتیجه خروجی های round 9 بدست می‌آیند و m بخشی از داده ورودی ما با قاعده ذکر شده است، همزمان از حالت x خارج شده و آماده پردازش برای یافتن مقدار hash می‌شوند. حال که هر سه این مقادیر یافت شد بعد از تاخیر اندکی خروجی نهایی حاصل می‌شود.

مقایسه مقدار خروجی در کد C و کد Verilog

همان طور که مشاهده می‌شود، مقادیر دو خروجی با هم برابر نیستند و بین آنها اختلاف وجود دارد. از جمله مهم‌ترین این دلایل، تفاوت در اندازه‌ی ورودی در کد طلایی و کد Verilog است. سایز ورودی در کد Verilog برابر ۵۱۲ بیت است در حالی که اندازه‌ی ورودی در کد C برابر ۱۰۲۴ بیت است که خود این امر باعث اختلاف در خروجی این دو کد شده است. پس برای این مقایسه باید یکی از ۲ کد به گونه‌ای اصلاح گردد که اندازه‌ی ورودی در هر ۲ کد یکسان باشد. در این صورت جدول به دست آمده از ۲ کد یکسان خواهد بود اما باز هم در خروجی اختلاف خواهیم دید که ناشی از دلایل دیگری است.

همچنین نکته‌ی دیگری که وجود دارد، این است که به علت ۵۱۲ بیتی بودن ورودی Verilog، شمارنده C_i نیز ۵۱۲ تا ۵۱۲ تا اضافه می‌شود و باعث ایجاد تفاوت در حاصل تابع SubBytes در هر مرحله می‌شود. و متفاوت بودن مقدار اولیه متغیر زنجیره‌ای (V_0) در کد Verilog و C نیز دلیلی برای متفاوت بودن خروجی است.

در کد C ، خروجی hash یک رشته‌ی باینری ۵۱۲ بیتی است که ۴ بایت اول آن به عنوان خروجی ۳۲ بیتی داده شده و با کد Verilog مقایسه شده‌اند. در کد اولیه مدل طلایی، ۸ بایت از کلّ hash محاسبه شده برگشت داده می‌شد که به دلیل ایجاد شباهت با کد Verilog، این تعداد به ۴ بایت کاهش داده شد. باید توجه داشت که هیچ ۴ بایت متوالی در کد c یافت نمی‌شود که مقادیر آنها با مقادیر خروجی کد Verilog برابر باشد.

حل مشکلات اجرایی نرم‌افزاری و سخت‌افزاری

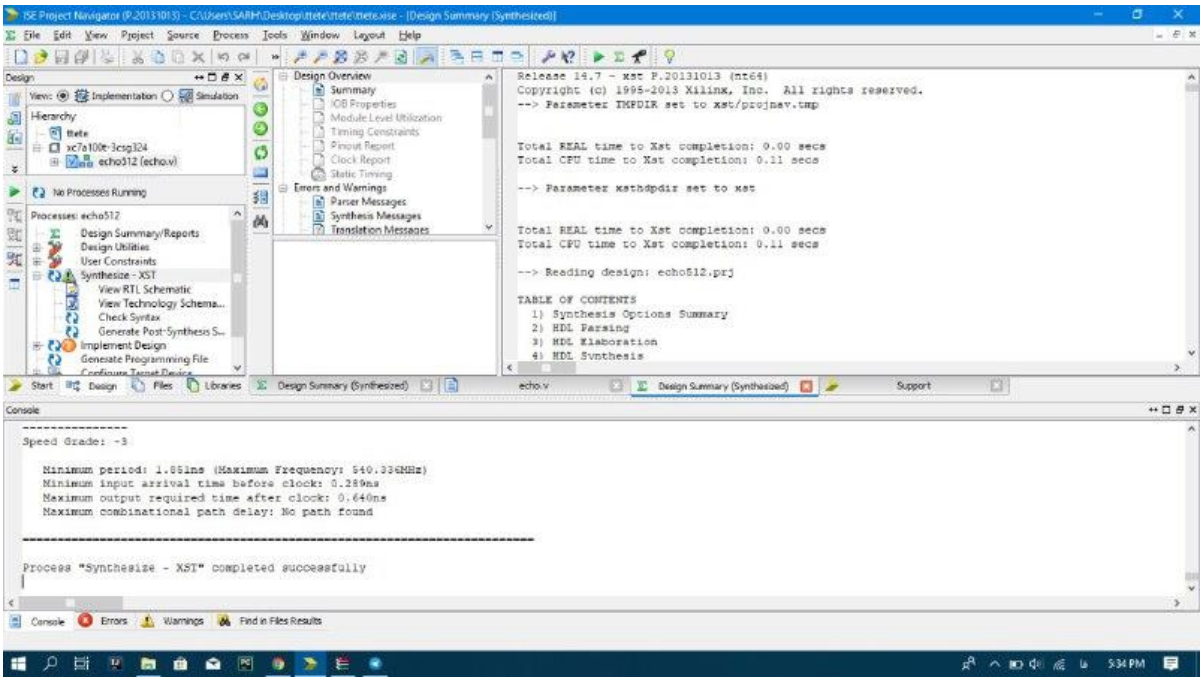
برای پیاده سازی نرم افزار نیاز به اضافه کردن کتاب‌خانه‌ی jansson.h وجود داشت که ضمیمه‌ی این مستند گردیده است. برای اجرای برنامه بر روی سیستم عامل linux نیاز به نصب کتاب‌خانه‌ی curl.h هم بود. هم‌چنین فایل قابل اجرا برای مشاهده‌ی خروجی کد طلایی طراحی گردید که در آن همه‌ی ۵۱۲ بیت خروجی نیز قابل مشاهده می‌باشند. (فایل exec.c)

در اجرای کد Verilog خطای can't assign packed to unpacked type مشاهده شد که برای اجتناب از این خطا (که در خطوط ۱۸۸۷ و ۲۰۰۷ کد اصلی Verilog وجود داشت) هر کدام از مقادیر آرایه به صورت جداگانه assign گردید:

```
assign s_box[0] = 8'h63;
assign s_box[1] = 8'h7C;
assign s_box[2] = 8'h77;
assign s_box[3] = 8'h7B;
assign s_box[4] = 8'hF2;
```

سنتز

برای سنتز کد سخت‌افزاری از ابزار Xilinx استفاده شد.



شکل ۲۲: محیط نرم‌افزار *Xilinx*

پس از نصب و اجرای سنتز، نتایج به دست آمد که در ادامه آورده شده است.

جدول فلیپ‌فلاپ ها

#RAMs	3072
#8x256-bit single-port distributed Read Only RAM	3072
#Registers	203840
#Flip-Flops	203840
#Xors	40634
#1-bit xor2	14592
#1-bit xor3	11680
#1-bit xor4	7008
#12-bit xor2	152
#32-bit xor2	2
#8-bit xor2	2336
#8-bit xor5	4864

جدول ۳: تعداد فلیپ‌فلاپ‌ها در سنتز

جدول Primitive ها

#BELS	52235
#GND	17
#INV	537
#LUT2	9538
#LUT3	24944

#LUT4	2038
#LUT5	8360
#LUT6	6784
#VCC	17
#FlipFlops/Latches	81276
#FD	80993
#FDE	283
#Shift Registers	347
#SRLC16E	251
#SRLC32E	96
#Clock Buffers	1
#BUFGP	1
#IO Buffers	244
#IBUF	212
#OBUF	32

جدول ٤ : تعداد *Primitive* ها در سنتز

جدول معیارهای زمانی

Timing constraint	Default period analysis for Clock 'clk'
Clock period	1.851ns (frequency: 540.336MHz)
Total number of paths / destination ports	245051 / 81411
Delay	1.851ns (Levels of Logic = 3)
Source	i92_13_0 (FF)
Destination	r92_13/out_23 (FF)
Source Clock	clk rising
Destination Clock	clk rising

جدول ٥: معیارهای زمانی در سنتز

گزارش نهایی

برای انجام بهتر پروژه کار به چند مرحله تقسیم شد.

- شناخت الگوریتم
- فهم کد های نرم افزاری و سخت افزاری
- اجرا تست ها و ثبت نتایج

در شروع، برای درک الگوریتم، طی جلسات بین تمام اعضای گروه، منابع مختلفی مطالعه شد. همزمان با این روند، تلاش شد روابط بین کد نرم‌افزاری و مستند فهمیده شود.

در ادامه به دو گروه تقسیم شده و کدهای نرم‌افزاری و سخت‌افزاری به صورت جدا مورد بررسی قرار گرفت. در طی این روند پس از فهم شیوه پیاده سازی الگوریتم در هر دو زبان، شباهت‌ها و تفاوت‌های آن‌ها با هم مقایسه شد. همچنین هر گروه موظف به نوشتن فایل تست برای کد مربوط به خود گردید.

به منظور اجرای برنامه ها نیاز به تغییر در نقاطی از کد سخت افزاری (مطابق آنچه گزارش شد) پیش آمد که انجام شد. در کد نرم‌افزاری نیز مشکل فراهم کردن هدر فایل های (header file) مورد نیاز برای کد C وجود داشت که با قرار دادن فایل های مورد نیاز در فولدر پروژه برطرف شد.

در نهایت، اجرا تست ها و ثبت نتایج از یکسو و نیز ساخت مستند و تصاویر مرتبط با کد ها از سویی دیگر و به شکل موازی صورت گرفت و خروجی های این دو مسیر در پایان، پس از سنتز کد سخت‌افزاری، در قالب این مستند جمع آوری شد.

نتیجه گیری گروه از کار بر روی این پروژه پس از مطالعه و تست کد های سخت‌افزاری و نرم‌افزاری، این است که علی‌رغم یکسان بودن کلی روند الگوریتم‌ها تا رسیدن به یک جدول نهایی از داده های هش شده (اعم از عملیات aes، مقادیر جدول s_box و...) تفاوت در نحوه گرفتن ورودی و همچنین ساخت خروجی از روی این جدول باعث تفاوت نتیجه خروجی این دو برنامه می‌شود. همچنین خروجی و اطلاعات سنتز کد سخت افزاری نیز مطابق آنچه آورده شد حاصل آمد.

فهرست منابع

Benadjila, R., Billet, O., Gueron, S., & Robshaw, M. J. (2009). The Intel AES Instructions Set and the SHA-3 Candidates. *ASIACRYPT '09 Proceedings of the 15th International Conference on the Theory and Application of Cryptology and Information Security: Advances in Cryptology*, 162-178.

Beuchat, J. L., Okamoto, E., & Yamazaki, T. (2010). A Compact FPGA Implementation of the SHA-3 Candidate ECHO. *IACR Cryptology ePrint Archive*.

Kinsy, M., & Uhler, R. (2019). SHA-3: FPGA Implementation of ESSENCE and ECHO Hash Algorithm Candidates Using Bluespec.

Schläffer, M. (2011). Subspace Distinguisher for 5/8 Rounds of the ECHO-256 Hash Function. *Selected Areas in Cryptography. SAC*.