

دانشکده مهندسی کامپیوتر  
طراحی سیستم‌های دیجیتال  
مستند پروژه

---

## بررسی الگوریتم درهم‌سازی skein

---

نگارندگان:  
حسن سندانی  
محمد صالح سعیدی  
مریم حاک  
محمد مهدی عرفانیان  
علی جندقی

۱۶ تیر ۱۳۹۸



# فهرست مطالب

|    |   |
|----|---|
| ۳  | ۱ مقدمه   |
| ۴  | ۱.۱ توضیح الگوریتم  |
| ۴  | ۱.۱.۱ مثال‌هایی از درهم‌سازی                              |
| ۴  | ۲.۱ مختصری درباره الگوریتم‌های درهم‌سازی امنیتی           |
| ۵  | ۳.۱ هدف الگوریتم درهم‌سازی skein                          |
| ۶  | ۴.۱ نحوه کلی عملکرد الگوریتم                              |
| ۶  | ۱.۴.۱ The Threefish block cipher                          |
| ۶  | ۲.۴.۱ Unique Block Iteration                              |
| ۸  | ۳.۴.۱ تابع درهم‌سازی Skein                                |
| ۸  | ۴.۴.۱ Optional Arguments                                  |
| ۱۰ | ۵.۱ کاربردهای الگوریتم درهم‌سازی Skein                    |
| ۱۲ | ۲ توصیف معماری سیستم                                      |
| ۱۳ | ۱.۲ اینترفیس‌های سیستم                                    |
| ۱۳ | ۱.۱.۲ ورودی‌ها  |
| ۱۳ | ۲.۱.۲ خروجی   |
| ۱۳ | ۲.۲ کلاک‌ها و نحوه راه‌اندازی سیستم                       |
| ۱۴ | ۳.۲ دیگرام بلوکی سخت‌افزار                                |
| ۱۶ | ۴.۲ توصیف ماژول‌های سخت‌افزار                             |
| ۱۶ | ۱.۴.۲ Skein-512   |
| ۱۹ | ۲.۴.۲ skein round   |
| ۲۱ | ۳.۴.۲ skein_round_1,2,3,4                                 |
| ۲۴ | ۳ شبیه‌سازی   |
| ۲۵ | ۱.۳ توضیح روند شبیه‌سازی سخت‌افزار و گام‌های اجرایی       |
| ۲۵ | ۲.۳ مشاهده ورودی‌ها و خروجی‌های اصلی و میانی              |
| ۲۵ | ۱.۲.۳ توضیح نحوه عملکرد Testbench                         |
| ۲۸ | ۲.۲.۳ شکل موج حاصل از Testbench                           |
| ۲۸ | ۳.۲.۳ جدول ورودی‌ها و خروجی‌های Testbench                 |
| ۳۰ | ۳.۳ اجرا و تحلیل کد نرم‌افزاری (مدل طلایی)                |
| ۳۰ | ۱.۳.۳ تحلیل کد C  |
| ۳۲ | ۲.۳.۳ مشاهده خروجی‌های کد C                               |
| ۳۴ | ۳.۳.۳ مقایسه کد verilog با مدل طلایی و مرجع اصلی الگوریتم |
| ۳۴ | ۴.۳.۳ مقایسه خروجی کد اصلی با داکيومنت مرجع               |
| ۳۴ | ۵.۳.۳ اصلاحات موردنیاز کد verilog                         |

|    |                                   |     |
|----|-----------------------------------|-----|
| ۳۶ | پیاده‌سازی سخت‌افزاری             | ۴   |
| ۳۷ | مقدمه‌ای بر پیاده‌سازی سخت‌افزاری | ۱.۴ |
| ۳۷ | ایرادات سنتز و ارائه راهکار       | ۲.۴ |
| ۳۸ | گزارش پیاده‌سازی                  | ۳.۴ |

# فصل ۱

## مقدمه

توضیحی اولیه مشتمل بر تعریف الگوریتم، نحوه کلی عملکرد الگوریتم، پایه‌های ریاضی، کاربردها و استانداردها

## ۱.۱ توضیح الگوریتم

الگوریتمی که در ادامه این مستند شرح و توضیح آن آمده است الگوریتم درهم سازی skein یا skein hash function است. این الگوریتم از سری الگوریتم‌های درهم‌سازی امنیتی یا cryptographic hash function و یکی از نامزدهای نهایی مسابقه انتخاب بهترین تابع درهم‌سازی NIST می‌باشد. این مسابقه برای انتخاب بهترین الگوریتم درهم‌سازی برای استاندارد جدید SHA-3 برگزار شد. طبق ادعای طراحان الگوریتم این الگوریتم می‌تواند در 6.1 کلاک در بایت داده‌ها را هش کند، که به این معنیست که در پردازنده دوهسته‌ای 64 بیتی با فرکانس پردازشی 3.1 GHz می‌تواند با سرعت 500 مگابایت بر ثانیه داده‌ها را هش کند. این مقدار سرعت تقریباً دو برابر سرعت هش کردن داده الگوریتم SHA-512 است. همچنین با گزینه درخت درهم‌سازی که می‌تواند به صورت اختیاری در الگوریتم پیاده‌سازی شود می‌توان در پیاده‌سازی موازی الگوریتم سرعت را به بیش از این هم رساند. نکته دیگری که در مورد الگوریتم skein لازم به ذکر است این است که این الگوریتم پیاده‌سازی آسان و ساده‌ای دارد و فقط از سه عملگر اصلی برای محاسبه هش استفاده می‌کند و نحوه عملکرد الگوریتم به راحتی قابل به خاطر سپاری و یادگیری است.

الگوریتم درهم‌سازی skein برای حالت‌های ورودی ۲۵۶، ۵۱۲ و ۱۰۲۴ بیتی و هر مقداری خروجی پیاده‌سازی شده است که این خاصیت در انعطاف الگوریتم در حالت‌های مختلف بسیار حیاتی است. در پیاده‌سازی سخت‌افزاری نیز این الگوریتم قوی عمل می‌کند، برای پیاده‌سازی skein-512 بر سخت‌افزار به حدود ۲۰۰ بایت فضای مموری نیاز داریم، برای skein-256 این مقدار به حدود ۱۰۰ بایت کاهش پیاده می‌کند که این الگوریتم را به یک الگوریتم مناسب برای پیاده‌سازی‌های روی قطعات کوچک سخت‌افزاری تبدیل می‌کند، مثلاً می‌توان از skein-256 در پیاده‌سازی smart card استفاده کرد. [۱]

### ۱.۱.۱ مثال‌هایی از درهم‌سازی

• Skein-256-256(“”)

*c8877087da56e072870daa843f176e9453115929094c3a40c463a196c29bf7ba*

• Skein-512-256(“”)

*39ccc4554a8b31853b9de7a1fe638a24cce6b35a55f2431009e18780335d2621*

• Skein-512-512(“”)

*bc5b4c50925519c290cc634277ae3d6257212395cba733bbad37a4af0fa06af4*

*1fca7903d06564fea7a2d3730dbdb80c1f85562dfcc070334ea4d1d9e72cba7a*

## ۲.۱ مختصری درباره الگوریتم‌های درهم‌سازی امنیتی

در دنیای امروز الگوریتم‌های درهم‌سازی امنیتی تقریباً در تمامی نقاط مختلفی که با اینترنت سر و کار دارند پیدا می‌شوند، بزرگ‌ترین کاربرد این الگوریتم‌ها ایجاد امضای دیجیتالی یا digital signature است که در ذخیره رمزهای عبور، اتصالات امنیتی به سرورها، مدیریت رمزنگاری‌ها و اسکن ویروس‌ها و بدافزارها به کار می‌رود، تقریباً تمامی پروتکل‌های امنیتی در دنیای اینترنت امروز بدون الگوریتم‌های درهم‌سازی امنیتی به سختی قابل پیاده‌سازی خواهند بود.

بزرگترین الگوریتم‌های درهم‌سازی امنیتی فعلی الگوریتم‌های خانواده SHA می‌باشند، الگوریتم‌های خانواده SHA به اختصار و فقط ذکر نام موارد زیر اند.

• SHA-0

• SHA-1

• SHA-256

• SHA-512

تمامی موارد بالا از روی الگوریتم‌های MD4 و MD5 اقتباس شده اند. در سال‌های اخیر کاستی‌ها و مشکلات امنیتی زیادی در الگوریتم‌های MD4, MD5, SHA-0, SHA-1 یافت شده‌اند اما هنوز باگ امنیتی بزرگی برای الگوریتم‌های SHA-256, SHA-512 یافت نشده است اما به دلیل وابستگی زیاد صنعت و امنیت فعلی اطلاعات به الگوریتم‌های درهم‌سازی در سال ۲۰۱۲ تصمیم بر این شد تا جایگزین مناسب و جدیدی برای الگوریتم‌های SHA-256, SHA-512 نیز انتخاب شود تا در صورتی که این الگوریتم‌ها شکسته شدند به سرعت الگوریتم‌های جدید در قالب نام SHA-3 جایگزین شوند.

### ۳.۱ هدف الگوریتم درهم‌سازی skein

هدف الگوریتم درهم‌سازی skein مانند دیگر الگوریتم‌های درهم‌سازی امنیتی ایجاد یک تابع برای درهم‌سازی داده‌های مختلف است به شکلی که ویژگی‌ها زیر برای آنان برقرار باشند.

- **قطعی بودن:** به شکلی که به ازای ورودی یکسان مقدار درهم‌سازی با تکرار الگوریتم برابر باشد، مثلاً با دادن ورودی "salam" به صورت متوالی به تابع مقدار هش تغییر نکند.
- **یک طرفه بودن:** نتوان از مقدار خروجی مقدار ورودی را یافت.
- **یک به یک بودن:** نتوان دو ورودی پیدا کرد به شکلی که به ازای این دو ورودی مقدار خروجی مساوی شود.
- **حساس بودن:** با تغییر اندک در ورودی خروجی به شکل قابل ملاحظه‌ای تغییر کند تا مقدار هش قابل حدس زدن نباشد.
- **سریع بودن:** الگوریتم باید بتواند هش را در مدت زمانی کوتاهی حساب کند تا به کاربردی بودن برسد.

## ۴.۱ نحوه کلی عملکرد الگوریتم

ایده اصلی الگوریتم بر ایجاد بلوک‌های رمزگذاری قابل تنظیم یا به زبان نویسندگان الگوریتم tweakable block cipher بنا نهاده شده است؛ به صورت دقیق‌تر می‌توان گفت که Skein از سه قسمت اصلی زیر تشکیل شده است و برای درهم‌سازی از ایشان استفاده می‌کند.

### • Threefish

این قسمت یک بلوک رمزگذاری قابل تنظیم است که در هسته اصلی الگوریتم پیاده‌سازی شده است، این بلوک‌ها در سایزهای ۲۵۶، ۵۱۲، ۱۰۲۴ بیتی تعریف شده اند.

### • Unique Block Iteration (UBI)

UBI یک حالت زنجیریست که با استفاده از بلوک قبلی به عنوان ورودی خود سعی در ایجاد یک الگوریتم فشرده‌سازی مخصوص ورودی می‌کند که بلوک ورودی با سایز دلخواه را به یک خروجی با سایز مشخص تبدیل کند.

### • Optional Argument System

این ویژگی به الگوریتم اجازه می‌دهد تا از تعدادی ویژگی اختیاری بدون تحمیل هزینه بیش از حد اجرایی استفاده کند. [۲]

همراهی سه بخش یادشده باهم ویژگی‌های جالب و کاربردی بسیاری را به الگوریتم درهم‌سازی Skein افزوده است، در ادامه به صورت خلاصه به نحوه عملکرد هر بخش می‌پردازیم.<sup>۱</sup>

## ۱.۴.۱ The Threefish block cipher

Threefish یک بلوک رمزگذاری قابل تنظیم است که برای سه سایز بلوک مختلف تعریف شده است، ۲۵۶، ۵۱۲ و ۱۰۲۴ بیت. اصل اساسی در طراحی Threefish توجه به این مورد است که تعداد زیادی از مراحل ساده امن‌تر از تعداد کمی مراحل پیچیده است. Threefish فقط از سه عملگر اصلی XOR، جمع کردن و دوران به اندازه یک عدد ثابت<sup>۲</sup> استفاده می‌کند. شکل ۱.۱ نحوه عملکرد تابع غیرخطی استفاده شده در Threefish را نشان می‌دهد، این تابع در زبان طراحان الگوریتم MIX نامیده می‌شود و بر روی دو کلمه ۶۴ بیتی اجرا می‌شود. هر تابع MIX شامل یک جمع، یک دوران و یک XOR است. ۱.۱ نحوه عملکرد Threefish-512 را نشان می‌دهد، هر یک از مراحل هفتاد و دو گانه الگوریتم Skein-512 از چهار تابع MIX به همراه ضرب در یک کلمه ۶۴ بیتی انجام می‌شوند. ثابت‌های چرخش به شکلی انتخاب می‌شوند تا پخش‌شدگی را در هشت به حداکثر خود برسانند. برای به دست آوردن مقدار Threefish-512 بار الگوریتم شکل ۲.۱ تکرار می‌شود.<sup>۳</sup>

## ۲.۴.۱ Unique Block Iteration

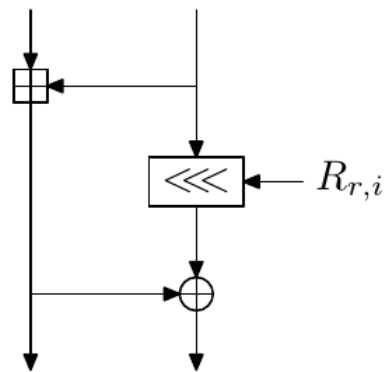
Unique Block Iteration یا به اختصار UBI زنجیره‌ای از ورودی‌ها را با یک رشته با طول دلخواه تلفیق می‌کند تا یک خروجی با اندازه مورد نظر و ثابت به دست آورد، در حقیقت UBI مقدار The Threefish block cipher را که مقداری با اندازه نامشخص و تعیین‌نشده‌ست را به خروجی با مقداری با اندازه ثابت تبدیل می‌کند، شکل ۳.۱ نحوه محاسبه UBI برای الگوریتم Skein-512 را نشان می‌دهد، اندازه ورودی ۱۶۶ بایت است که در سه بلوک ریخته شده است، بلوک‌های  $M_0$  و  $M_1$  هر کدام ۶۴ بایت دارند و  $M_2$  که برچسب آخرین بلوک<sup>۴</sup> را دارد باقی‌مانده اندازه یعنی ۳۸ بایت دارد. با استفاده از tweak بلوک که قلب اصلی UBI را تشکیل می‌دهد UBI متوجه می‌شود که آیا تمامی بلوک‌ها برای ایجاد خروجی پردازش شده اند یا خیر و

<sup>۱</sup> برای مطالعه بیشتر می‌توانید به بخش سوم [۲] مراجعه کنید.

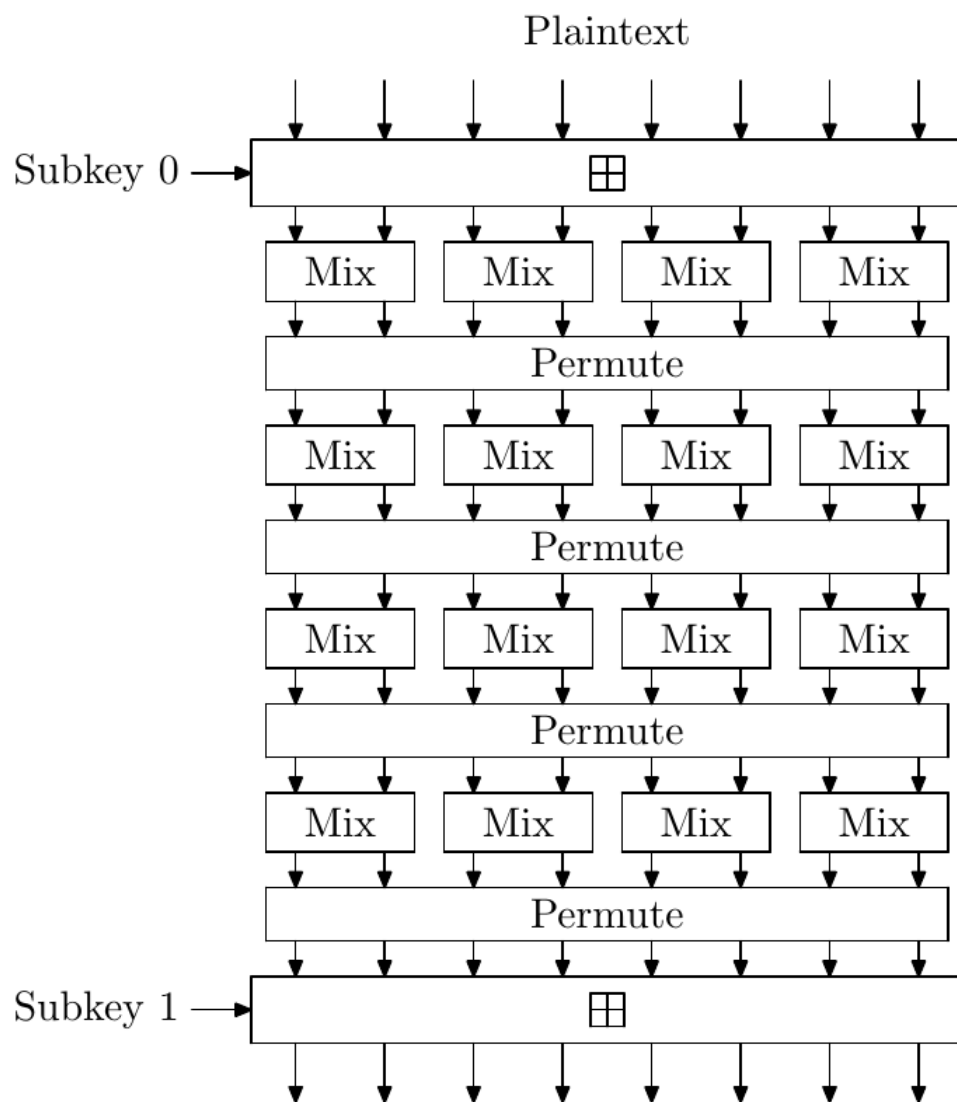
<sup>۲</sup> Rotation Constant

<sup>۳</sup> برای مطالعه جزئی‌تر می‌توانید به [۲] مراجعه کنید.

<sup>۴</sup> final block

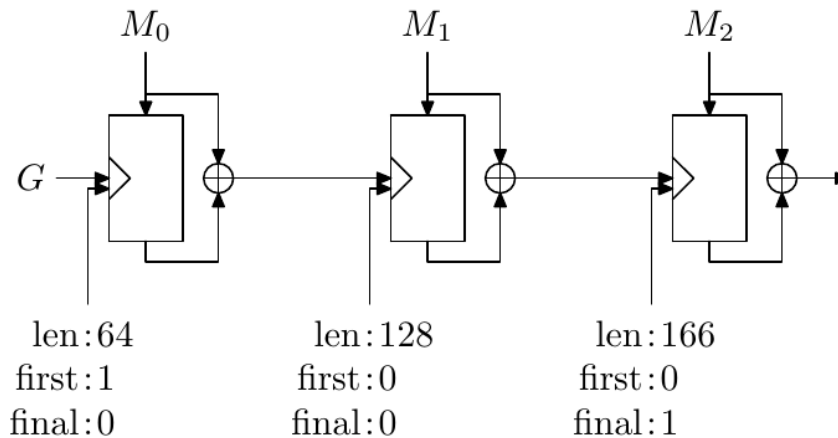


شکل ۱.۱: تابع MIX



شکل ۲.۱: چهار مرحله از ۷۲ مرحله Threefish-512 block cipher





شکل ۳.۱: درهم‌سازی پیام سه بلوکه با UBI

این که آیا به بلوک پایانی (پایان زنجیره) رسیده است یا خیر. UBI یکی از انواع Matyas-Meyer-Oseas ها است. [۳]

### ۳.۴.۱ تابع درهم‌سازی Skein

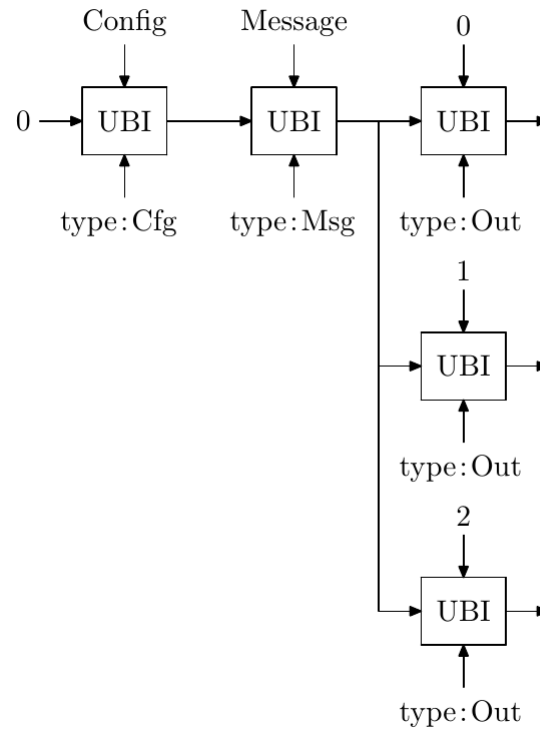
تابع اصلی درهم‌سازی در حالت نرمال که مد نظر این نوشتار است برای ایجاد هش از چندین درخواست از UBI و بالتبع از Threefish block cipher هش یک داده ورودی را حساب می‌کند، برای محاسبه هش سه بار UBI با ورودی‌های مختلف زیر صدا زده می‌شود، شکل ۴.۱ توضیحات زیر را به صورت شماتیک نشان می‌دهد.

- **Config** این ورودی مقدار اندازه خروجی و تعدادی از پارامترها برای Tree-hashing را فراهم می‌کند، در صورتی که از حالت استاندارد و نرمال Skein برایش درهم‌سازی استفاده شود این مقدار قابل پیش‌پردازش است.
- **Message** مقدار داده ورودی است.
- **Counter** شمارنده‌ای برای نشان دادن تعداد بار تکرار الگوریتم ایجاد خروجی برای رسیدن به خروجی با اندازه مورد نظر است، در صورتی که خروجی بیش از اندازه‌ای مورد انتظار باشد، دوباره تابع ایجاد خروجی فراخوانی می‌شود.

### ۴.۴.۱ Optional Arguments

در راستای افزایش انعطاف‌پذیری الگوریتم درهم‌سازی skein برای کاربردهای مختلف تعدادی ورودی به صورت اختیاری به الگوریتم افزوده شده‌اند، در ادامه مختصراً به توضیح ایشان می‌پردازیم.

- **Key** (اختیاری) کلیدی برای تبدیل skein به تابع MAC یا KDF.
- **Configuration** (اجباری) همان مقدار Config که پیش‌تر توضیح داده شد.
- **Personalization** (اختیاری) رشته‌ای که برنامه می‌تواند با استفاده از آن تابع‌های مختلفی برای کاربردهای مختلفی بسازد.
- **Nonce** (اختیاری) مقدار Nonce برای استفاده در حالت stream cipher و حالت درهم‌سازی تصادفی.



شکل ۴.۱: تابع ایجاد هش با خروجی بزرگ‌تر از اندازه مورد انتظار

- **Message** (اختیاری)  
ورودی نرمال تابع درهم‌سازی.
- **Output** (اجباری) مقدار خروجی الگوریتم.  
در محاسبه هش تابع درهم‌سازی *Skein* به ترتیب ذکر شده در بالا *UBI* ورودی‌ها محاسبه می‌شود.

## ۵.۱ کاربردهای الگوریتم درهم‌سازی Skein

- **Skein به عنوان تابع درهم‌سازی ساده‌ترین راه استفاده از الگوریتم Skein** استفاده به عنوان تابعی برای به دست آوردن هش ورودی‌ست، در این حالت Skein مانند تمام الگوریتم‌های دیگر درهم‌سازی عمل می‌کند و رشته‌ای را به عنوان هش با اندازه از پیش تعیین شده خروجی می‌دهد.
- **Skein به عنوان MAC** از تابع درهم‌سازی Skein می‌توان برای تولید MAC<sup>۵</sup> استفاده کرد، از MAC برای واریسی این که یک پیام از یک فرستنده معتبر بدون تغییر ارسال شده یا که در طی مسیر دست‌کاری شده است استفاده می‌شود.
- **HMAC**
- **Randomized Hashing**
- **Digital Signatures**
- **Key Derivation Function (KDF)**
- **Password-Based Key Derivation Function (PBKDF)**
- **PRNG**
- **Stream Cipher**

---

<sup>۵</sup>Message authentication code

# کتاب نامه

<http://www.skein-hash.info/about> [۱]

The Skein Hash Function Family [۲]  
Version 1.3 — 1 Oct 2010  
<http://www.skein-hash.info/sites/default/files/skein1.3.pdf>

S.M. Matyas, C.H. Meyer, and J. Oseas, “Generating strong one-way functions with [۳]  
crypto- graphic algorithms  
” IBM Technical Disclosure Bulletin, Vol. 27, No. 10A, 1985, pp. 5658–5659.

## فصل ۲

# توصیف معماری سیستم

تشریح اینترفیس‌های سیستم، کلاک‌ها و نحوه راه‌اندازی سیستم، دیاگرام بلوکی سخت‌افزار، ساختار درختی سیستم و توصیف ماژول‌های سخت‌افزار

## ۱.۲ اینترفیس‌های سیستم

در ابتدا به صورت خلاصه اینترفیس‌های سیستم سخت‌افزاری الگوریتم Skein بیان می‌شود، اینترفیس یک سیستم شامل ورودی‌ها و خروجی‌ها و مشخصات ایشان است.

### ۱.۱.۲ ورودی‌ها

ورودی‌ها کد verilog الگوریتم Skein به شرح زیر اند.

#### • clk

ورودی کلاک سیستم است که با آن سیستم کار خود را به صورت ترتیبی<sup>۱</sup> انجام می‌دهد، فرکانس کلاک با توجه به نحوه پیاده‌سازی سخت‌افزاری و نتایج حاصل از سنتز تعیین می‌شود. در Testbench داده شده کلاک هر ۱۰ نانوثانیه تغییر می‌کند.

#### • midstate

ورودی ۵۱۲ بیتی برای الگوریتم Skein-512 است که حالت میانی در هش را معلوم می‌کند.

#### • nonce

nonce مقداری دلخواه است که برای به حداکثر رساندن تصادفی و غیرقابل شکستن بودن هش در محاسبه هش استفاده می‌شود، این مقدار می‌تواند عددی دلخواه باشد. در الگوریتم Skein-512 اندازه این ورودی ۳۲ بیت به اندازه طول عدد در Integer گرفته شده است.

#### • data

ورودی اصلی ست که باید هش آن محاسبه شود، در کد verilog داده شده اندازه این ورودی ۹۶ بیت در نظر گرفته شده است.

### ۲.۱.۲ خروجی

تنها خروجی سیستم مقدار هش در output است که ۵۱۲ بیت طول دارد. (الگوریتم مورد بحث Skein-512 است)

## ۲.۲ کلاک‌ها و نحوه راه‌اندازی سیستم

این سیستم فقط از یک کلاک استفاده می‌کند و برای راه‌اندازی سیستم انجام کارهای زیر ضروری ست.

۱. وصل کردن کلاک با فرکانس مناسب به سیستم

۲. اعمال ریست کلی بر سیستم<sup>۲</sup>

۳. تعیین ورودی‌های اولیه

• ورودی‌های سیستم در ادامه به طور کامل تشریح شده‌اند، هیچ یک از ورودی‌ها نسبت به دیگری تقدم زمانی هنگام assignment ندارد.

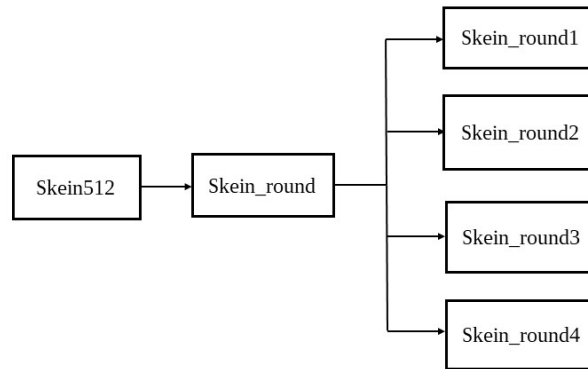
۴. راه‌اندازی سیستم

<sup>۱</sup> Sequential

<sup>۲</sup> Global Reset

## ۳.۲ دیاگرام بلوکی سخت‌افزار

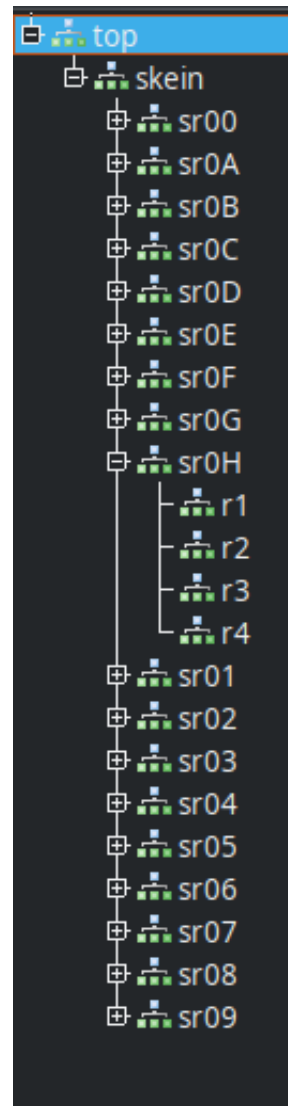
دیاگرام بلوکی کلی سخت‌افزار در شکل ۱.۲ آمده است.



شکل ۱.۲: دیاگرام بلوکی سخت‌افزار

| Type | Signals     |
|------|-------------|
| wire | clk         |
| wire | h[575:0]    |
| wire | ho[575:0]   |
| reg  | hx0[575:0]  |
| reg  | hx1[575:0]  |
| reg  | hx2[575:0]  |
| reg  | hx3[575:0]  |
| reg  | hx4[575:0]  |
| reg  | p0[63:0]    |
| reg  | p1[63:0]    |
| reg  | p2[63:0]    |
| reg  | p3[63:0]    |
| reg  | p4[63:0]    |
| reg  | p5[63:0]    |
| reg  | p6[63:0]    |
| reg  | p7[63:0]    |
| wire | p[511:0]    |
| wire | po0[511:0]  |
| wire | po1[511:0]  |
| wire | po2[511:0]  |
| wire | po3[511:0]  |
| wire | po4[511:0]  |
| wire | po[511:0]   |
| wire | round[31:0] |
| wire | t0[63:0]    |
| wire | t1[63:0]    |

(ب) ساختار سیگنال‌ها هنگام شبیه‌سازی



(آ) ساختار ماژول‌ها هنگام شبیه‌سازی

شکل ۲.۲: ساختار درختی سیستم



## ۴.۲ توصیف ماژول‌های سخت‌افزار

### Skein-512 ۱.۴.۲

#### اطلاعات کلی

|          |   |
|----------|---|
| skein512 |   |
| inputs   | $clk, [511 : 0]midstate, [95 : 0]data, [31 : 0]nonce$ |
| outputs  | $[511 : 0]hash$                                       |

جدول ۱.۲: اطلاعات کلی ماژول skein512

در خطوط اولیه تعداد reg و wire تعریف شده است. دو reg به نام های  $phase\_d$  و  $phase\_q$  تعریف شده اند که یک بیتی اند و مقدار صفر به آنها داده شده است. دو assignment یک خطی دیده می شود.

۱. در reg ۳۲ بیتی با نام  $nonce\_le$  که در خطوط بالاتر تعریف شده است مقادیر  $nonce$  (که ورودی ۳۲ بیتی ماژول هستند) به صورت ۸ بیت - ۸ بیت و به صورت برعکس ذخیره می شوند. یعنی به طور مثال ۸ بیت کم ارزش  $nonce$  در ۸ بیت پر ارزش  $nonce\_le$  ذخیره شده اند.

۲. در reg ۳۲ بیتی با نام  $nonce2\_le$  که در خطوط بالاتر تعریف شده است مقادیر  $nonce2$  (که برعکس  $nonce$ ، ورودی ماژول نیست و خود در خطوط بالاتر به صورت یک reg ۳۲ بیتی تعریف شده است و در واقع در حال حاضر مقداری را به خود اختصاص نداده است) به صورت ۸ بیت - ۸ بیت و به صورت برعکس ذخیره می شوند. یعنی به طور مثال ۸ بیت کم ارزش  $nonce2$  در ۸ بیت پر ارزش  $nonce2\_le$  ذخیره شده اند. (خط ۵۶)

یک عبارت assign طویل مربوط به hash دیده می شود:

- در این عبارت بیت های رجیستری به نام  $h\_q$  که ۵۱۲ بیت دارد و در خطوط بالاتر تعریف شده است، به بیت های خروجی hash اساین می شود.
- ۶۴ مجموعه ۸ بیتی از  $h\_q$  به بیت های hash اساین می شود که نظم این مقداردهی در زیر توضیح داده می شود. در این توضیحات hash را به ترتیب از پر ارزش ترین ۸ بیت شروع به پر کردن می کنیم.
- پر ارزش ترین بیت های hash با بیت های ۴۶۳ تا ۴۵۶ پر شده است. (یعنی پر ارزش ترین بیت hash با بیت ۴۶۳ ام  $h\_q$  پر شده است و به همین ترتیب)
- مجموعه بعدی ۸ تایی از ۴۶۴ تا ۴۷۱ هستند که در دومین ۸ تایی با ارزش hash قرار می گیرند.
- این روند تا هشتمین ۸ بیت ارزشمند hash ادامه پیدا می کند جایی که در این جایگاه مجموعه : [511 : 504] از  $h\_q$  جای می گیرد. (تا اینجا نظم داشتیم)
- نهمین ۸ بیت ارزشمند hash توسط بیت های [384 : 391] از  $h\_q$  پر می شوند.
- این روند ادامه پیدا می کند (یعنی دهمین ۸ بیت ارزشمند با [392 : 399] پر می شوند)، تا ۱۶ امین ۸ بیت ارزشمند hash که با مجموعه [440 : 447] پر شده اند.
- ۱۷ امین ۸ بیت ارزشمند با مجموعه [320 : 327] پر می شود.

- این روند مانند قبل به صورت صعودی ادامه پیدا خواهد کرد تا به ۲۵امین مجموعه ۸بیتی برسیم.

**درواقع هر ۸ بار که مجموعه بیت‌های ۸بیتی را assign می‌کنیم، یک بی‌نظمی داریم.**

- ۲۵امین ۸بیتی hash با بیت‌های [256 : 263] پر می‌شود.
- دوباره روند سابق و صعودی را داریم تا به ۳۳امین assignment برسیم.
- ۳۳امین ۸بیتی hash با بیت‌های [192 : 199]

**هر بار بی‌نظمی داریم بازه جدید بعد از بی‌نظمی ۱۲۰ واحد کمتر از بازه قبلی خواهد بود مثلاً ۳۲امین ۸بیت پرازش hash با بیت‌های [312 : 319] پر شده‌اند که ۱۲۰ واحد از بازه‌ای که برای ۳۳امین ۸ بیت ارزشمند hash اختصاص داده می‌شود بیشتر است. (در بالا ۳۳امین نوشته شده است)**

- ۸ مجموعه که به صورت صعودی پیش برویم به ۴۰امین ۸بیت می‌رسیم که طبق نظم با بیت‌های [248 : 255] پر شده است و ۴۱امین ۸بیتی با بازه [128 : 135] پر شده است.
- ۸ مجموعه که به صورت صعودی پیش برویم به ۴۸امین ۸بیت می‌رسیم که طبق نظم با بیت‌های [184 : 191] پر شده است و ۴۹امین ۸بیتی با بازه [64 : 71] پر شده است.
- ۸ مجموعه که به صورت صعودی پیش برویم به ۵۶امین ۸بیت می‌رسیم که طبق نظم با بیت‌های [120 : 127] پر شده است و ۵۷امین ۸بیتی با بازه [0 : 7] پر شده است.
- از ۵۷امین مجموعه ۸ تایی با ارزش hash تا آخرین مجموعه با ارزش hash (۶۴امین) نیز به صورت صعودی و طبق نظم پیش می‌رود. (خط ۱۲۱)

بعد از خطوط assignment ۱۸ instance از ماژول skein\_round گرفته شده است. این - in stance ها را از 00 تا 0H نام‌گذاری کردیم (نام‌گذاری در مبنای بالاتر از ۱۰ شده است)

### ورودی skein\_round ها

- کلاک که همه به کلاک سیستم متصل‌اند.
- **Round** رجیستر ۳۲بیتی که به ترتیب ورودی ۰ تا ۱۷ به هر اینستنس داده شده است.
- **p** رجیستر ۵۱۲بیتی - که به اینستنس شماره 01 تا 0H به ترتیب p01 تا p0H وصل شده است. به اینستنس شماره 00 هم p00\_q وصل شده است.
- **H** رجیستر ۵۷۶بیتی - که به اینستنس شماره 01 تا 0H به ترتیب h01 تا h0H وصل شده است. به اینستنس شماره 00 هم h00\_q وصل شده است.
- **T0** رجیستر ۶۴بیتی - که به اینستنس شماره 00 تا 0H به ترتیب این دنباله سه‌تایی وصل شده است:  $t0\_q, t1\_q, t2\_q$  این دنباله سه‌جمله‌ای به ترتیب تکرار می‌شود.
- **T1** رجیستر ۶۴بیتی - دقیقاً مثل T0 با این تفاوت که دنباله سه‌تایی  $t1\_q, t2\_q, t0\_q$  به این شکل است.
- **P0** رجیستر ۵۱۲بیتی - که اینستنس شماره 00 تا 0H به ترتیب o00 تا o0H وصل شده است.
- **H0** رجیستر ۵۷۶بیتی - که اینستنس شماره 00 تا 0H به ترتیب ho00 تا ho0H وصل شده است. خط (۱۴۱)

در ادامه یک *always* بلاک داریم که حساس به تغییرات همه چیز است. (خط ۱۴۳) در این بلاک متغیرهایی که در انتهایشان *\_d* دارند مقداردهی می‌شوند.

ابتدا *phase\_d* مقدار *not* متغیر *phase\_q* را به خود اختصاص می‌دهد.

#### • اگر *phase\_q* یک باشد

– مقداردهی به *p00\_d* (۵۱۲ بیتی): ۶۴ بیت کم‌ارزش ( [63 : 0] ) از *data* در ۶۴ بیت پرارزش *p00\_d* قرار می‌گیرد. سپس در ۳۲ بیت بعدی *p00\_d* (از چپ) عیناً *nonce le* قرار داده می‌شود. سپس ۳۲ بیت باقی‌مانده از *data* ([95 : 64]) طبق روند قرار داده می‌شود. باقی بیت‌های این رجیستر هم با صفر پر می‌شوند (*384'd*) – (خط ۱۴۸)

– مقداردهی به *h00\_d* (۵۷۶ بیتی): در ۶۴ بیت کم‌ارزش این *reg* مقدار صفر قرار داده می‌شود و باقی بیت‌ها دقیقاً به *midstate* (ورودی ۵۱۲ بیتی) متصل می‌شوند.

– مقداردهی به *t0\_d* (۶۴ بیتی) : *h000000000000000050*

– مقداردهی به *t1\_d* (۶۴ بیتی) : *hb000000000000000000*

– مقداردهی به *t2\_d* (۶۴ بیتی) : *hb000000000000000050*

– *h\_d* هم مقدار *h\_q* را به خود می‌گیرد.

#### • اگر *phase\_q* صفر باشد

– مقداردهی به *p00\_d* (۵۱۲ بیتی): این *reg* با صفر پر می‌شود.

– مقداردهی به *h00\_d* (۵۷۶ بیتی):

\* بیت‌های [575 : 512] :  $data[63 : 0] \wedge (oH[511 : 448] + hH[575 : 512])$

\* بیت‌های [511 : 448] :  $\{ data[95 : 64, nonce2\_le] \} \wedge (oH[447 : 384] + hH[511 : 448])$

\* بیت‌های [447 : 384] :  $oH[383 : 320] + hH[447 : 384]$

\* بیت‌های [383 : 320] :  $oH[319 : 256] + hH[383 : 320]$

\* بیت‌های [319 : 256] :  $oH[255 : 192] + hH[319 : 256]$

\* بیت‌های [255 : 192] :  $oH[191 : 128] + hH[255 : 192] + 64'h000000000000000050$

\* بیت‌های [191 : 128] :  $oH[127 : 64] + hH[191 : 128] + 64'hb0000000000000000$

\* بیت‌های [127 : 64] :  $oH[63 : 0] + hH[127 : 64] + 18$

– مقداردهی به *t0\_d* (۶۴ بیتی) : *h000000000000000008*

– مقداردهی به *t1\_d* (۶۴ بیتی) : *hFF0000000000000000*

– مقداردهی به *t2\_d* (۶۴ بیتی) : *hFF0000000000000008*

– مقداردهی به *h\_d* (۵۱۲ بیتی):

\* بیت‌های [511 : 448] :  $o0H[511 : 448] + ho0H[575 : 512]$

\* بیت‌های [447 : 384] :  $o0H[447 : 384] + ho0H[511 : 448]$

\* بیت‌های [383 : 320] :  $o0H[383 : 320] + ho0H[447 : 384]$

\* بیت‌های [319 : 256] :  $o0H[319 : 256] + ho0H[383 : 320]$

\* بیت‌های [255 : 192] :  $o0H[255 : 192] + ho0H[319 : 256]$

\* بیت‌های [191 : 128] :  $o0H[191 : 128] + ho0H[255 : 192] + 64'h000000000000000008$

\* بیت‌های [127 : 64] :  $o0H[127 : 64] + ho0H[191 : 128] + 64'hFF0000000000000000$

\* بیت‌های [63 : 0] :  $o0H[63 : 0] + ho0H[127 : 64] + 18$

نظم مناسبی دیده می‌شود به این شکل که به ترتیب ۶۴ بیت پردازش  $h$   $d$  با مجموع ۶۴ بیت پردازش  $o0H$  و ۶۴ بیت پردازش  $ho0H$  پر می‌شود. به جز ۳ مورد آخر که با اعدادی ثابت هم جمع می‌شوند.

**Always** بلاک دوم فقط به لبه مثبت کلاک حساس است. (خط ۲۱۱) (عموماً متغیرهای  $q$  مقادیر متناظر  $d$  را به خود می‌گیرند)

- $hH$  مقدار  $ho0H$  را به خود می‌گیرد.
- $oH$  مقدار  $o0H$  را به خود می‌گیرد.
- $phase\_q$  مقدار  $phase\_d$  را به خود می‌گیرد.
- $h\_q$  مقدار  $h\_d$  را به خود می‌گیرد.
- $t0\_q$  مقدار  $t0\_d$  را به خود می‌گیرد.
- $t1\_q$  مقدار  $t1\_d$  را به خود می‌گیرد.
- $t2\_q$  مقدار  $t2\_d$  را به خود می‌گیرد.
- در ادامه مجموعه ای از مقداردهی ها را مربوط به  $reg$  های  $h0x$  و  $p0x$  داریم. (خط ۲۲۶ تا ۲۶۱) ( $x$  منظور از ۱ تا  $H$ )
- $h0x$  ها: مقدار  $ho0y$  را می‌گیرند با این تفاوت که  $y$  از  $x$  یک واحد کمتر است. (به طور مثال  $h01$  مقدار  $ho00$  را به خود می‌گیرد)
- $p0x$  ها: مقدار  $o0y$  را می‌گیرند با این تفاوت که  $y$  از  $x$  یک واحد کمتر است. (به طور مثال  $h01$  مقدار  $o00$  را به خود می‌گیرد)
- $p00\_q$  مقدار  $p00\_d$  را می‌گیرد.
- $h00\_q$  مقدار  $h00\_d$  را می‌گیرد.
- $nonce2$  هم که در ابتدای فایل مقداری مجهول داشت اینجا مقدار  $nonce$  (ورودی) منهای  $32d54$  را می‌گیرد.

## ۲.۴.۲ skein\_round

در خط ۱۲۴ از فایل وریلاگ، ۱۸ اینستنس از ماژول  $skein\_round$  گرفته شده است.

### اطلاعات کلی

| skein_round |  |
|-------------|--|
| inputs      | $clk, [31 : 0]round, [511 : 0]p, [575 : 0]h, [63 : 0]t0, [63 : 0]t1$ |
| outputs     | $[511 : 0]p0, [575 : 0]h0$   |

جدول ۲.۲: اطلاعات کلی ماژول  $skein\_round$

- در این ماژول چهار ماژول دیگر زیر ایجاد شده اند.
  - skein\_round\_1
  - skein\_round\_2
  - skein\_round\_3
  - skein\_round\_4
- در این ماژول یک always block و تعدادی assignment وجود دارد.
- دو مجموعه reg تعریف شده:
  - ۶۴ بیتی:  $p0, p1, p2, p3, p4, p5, p6, p7$
  - ۵۷۶ بیتی:  $hx0, hx1, hx2, hx3, hx4$
- یک مجموعه wire تعریف شده:
  - ۵۱۲ بیتی:  $po0, po1, po2, po3, po4$
- ۳ عدد assignment داریم:
  - $ho$  (یکی از خروجی‌ها) از  $hx4$  مقدار می‌گیرد.
  - $po$  (دیگر خروجی) از  $po4$  مقدار می‌گیرد.
  - $po0$  به ترتیب ۸ بیت-۸ بیت (ازپرازش به کم‌ارزش) از reg های  $p0, p1, \dots, p7$  مقدار می‌گیرد.
- از هر ۴ ماژول باقی مانده (skein\_round\_1,2,3,4) در کد یک اینستنس گرفته شده است. (خط ۳۱۰)
- ورودی کلاک به کلاک سیستم متصل شده است. ورودی  $even$  ماژول‌ها همگی به  $!round[0]$  متصل اند. (یکی از بیت‌های ورودی)
- به عنوان  $in$  و  $out$  هم به هر ماژول  $po(x)$  و  $po(x+1)$  داده می‌شود که  $x+1$  شماره  $round$  ماژول است. به طور مثال به ماژول  $skein\_round\_1$  برای ورودی  $po0$  و برای خروجی  $po1$  داده می‌شود.
- نکته مهم این است که خروجی ماژول ۱ ورودی ماژول ۲ است و به همین ترتیب تا ماژول ۴.**
- یک always-block حساس به لبه مثبت کلاک داریم. (خط ۳۱۵)
  - ورودی های  $h$  و  $p$  را به صورت ۶۴ بیت – ۶۴ بیت جمع می‌زند و در  $p0$  تا  $p7$  نگه‌داری می‌کند.
  - به این شکل که جمع پرازش‌ترین ۶۴ بیت  $h$  و  $p$  در  $p0$  ریخته می‌شود. (و به همین ترتیب پیش می‌رود)
  - از  $p0$  تا  $p4$  کاملاً طبق نظم گفته‌شده انجام می‌شود.
  - در مورد  $p5$  علاوه بر دو مجموعه ۶۴ بیتی با  $t0$  (یکی از ورودی‌ها) هم جمع می‌شود.
  - در مورد  $p6$  علاوه بر دو مجموعه ۶۴ بیتی با  $t1$  (یکی از ورودی‌ها) هم جمع می‌شود.
  - در مورد  $p7$  علاوه بر دو مجموعه ۶۴ بیتی با  $round$  (یکی از ورودی‌ها) هم جمع می‌شود.
- برای reg های  $hx$  (۵۷۶ بیتی) هم یک جابه‌جایی اتفاق می‌افتد:
  - به این شکل که  $hx4$ ، مقدار  $hx3$  را می‌گیرد.
  - به این شکل که  $hx3$ ، مقدار  $hx2$  را می‌گیرد.
  - به این شکل که  $hx2$ ، مقدار  $hx1$  را می‌گیرد.

– به این شکل که  $hx1$ ، مقدار  $hx0$  را می‌گیرد.

– برای  $Hx0$  اتفاق نسبتاً پیچیده‌ای می‌افتد:

\* ۶۴ بیت کم‌ارزشش با ۶۴ بیت پرارزش  $h$  (ورودی) پر می‌شود.

\* ۴۴۸ بیت پرارزشش با بیت‌های  $[511 : 64]$  از  $h$  پر می‌شود.

\* ۶۴ بیت باقی‌مانده وسط  $hx0$  ( $[64 : 127]$ ) با نتیجه زیر پر می‌شود: (خط ۳۴۱)

```
1 ((h[575:512] ^ h[511:448]) ^ (h[447:384] ^ h[383:320])) ^ ((h[319:256] ^ h[255:192]) ^ (h[191:128] ^ h[127: 64])) ^ 64'h1BD11BDAA9FC1A22
```

در واقع در توضیح خط بالا می‌توان گفت  $xor$  تمام مجموعه‌های ۶۴ بیت‌های ورودی  $h$  به جز کم‌ارزش‌ترین مجموعه ( $h[64 : 0]$ ) است که در نهایت با یک عدد ثابت ۶۴ بیتی  $xor$  شده است.

### ۳.۴.۲ skein\_round\_1,2,3,4

چهار ماژول باقی‌مانده با نام‌های  $skein\_round\_1$ ,  $skein\_round\_2$ ,  $skein\_round\_3$ ,  $skein\_round\_4$  را به دلیل شباهت ساختاری با هم بررسی می‌کنیم.

#### اطلاعات کلی

| skein_round_1,2,3,4 |                          |
|---------------------|--------------------------|
| inputs              | $clk, even, [511 : 0]in$ |
| outputs             | $[511 : 0]out$           |

جدول ۳.۲: اطلاعات کلی ماژول‌های  $skein\_round\_1,2,3,4$

• در هر ۴ ماژول دو مجموعه  $wire$  گرفته شده است:

– ۶۴ بیتی:  $p0, p1, p2, p3, p4, p5, p6, p7$

– ۶۴ بیتی:  $p0x, p1x, p2x, p3x, p4x, p5x, p6x, p7x$

• مجموعه‌ای از assignment ها داریم:

–  $P0$  تا  $p7$  به ترتیب به ۶۴ بیت‌های پرارزش تا کم‌ارزش  $in$  متصل اند. (به کم‌ارزش‌ترین  $p7$ )

• Assignment های مربوط به  $p0x$  تا  $p7x$  برای ۴ ماژول متفاوت است در نتیجه جداگانه بررسی می‌کنیم:

– ماژول ۱

$Pkx$  ها با  $k$  های زوج مقدار  $pk + p(k + 1)$  را به خود می‌گیرند. مثلاً  $p2x = p2 + p3$   
 $P1x$  با توجه به  $even$  مقدار می‌گیرد:

\* اگر  $even$  یک باشد،  $p1[17 : 0], p1[63 : 18]$  (خط شکاف بین بیت ۱۷ و ۱۸)

\* اگر  $even$  صفر باشد،  $p1[24 : 0], p1[63 : 25]$  (خط شکاف بین بیت ۲۴ و ۲۵)

$P3x$  با توجه به  $even$  مقدار می‌گیرد:

\* اگر  $even$  یک باشد،  $p3[27 : 0], p3[63 : 28]$  (خط شکاف بین بیت ۲۷ و ۲۸)

- \* اگر  $even$  صفر باشد،  $p3[33:0], p3[63:34]$  (خط شکاف بین بیت ۳۳ و ۳۴)  $P5x$  با توجه به مقدار  $even$  می‌گیرد:
- \* اگر  $even$  یک باشد،  $p5[44:0], p5[63:45]$  (خط شکاف بین بیت ۴۴ و ۴۵)
- \* اگر  $even$  صفر باشد،  $p5[29:0], p5[63:30]$  (خط شکاف بین بیت ۲۹ و ۳۰)
- \*  $P7x$  با توجه به مقدار  $even$  می‌گیرد:
- \* اگر  $even$  یک باشد،  $p7[26:0], p7[63:27]$  (خط شکاف بین بیت ۲۶ و ۲۷)
- \* اگر  $even$  صفر باشد،  $p7[39:0], p7[63:40]$  (خط شکاف بین بیت ۳۹ و ۴۰)

- ماژول ۲

```

1 //module 2
2 assign p0x = p0 + p3
3 assign p1x = (even) ? { p1[30:0], p1[63:31] } : { p1[50:0], p1[63:51] }
4 assign p2x = p2 + p1
5 assign p3x = (even) ? { p3[21:0], p3[63:22] } : { p3[46:0], p3[63:47] }
6 assign p4x = p4 + p7
7 assign p5x = (even) ? { p5[49:0], p5[63:50] } : { p5[53:0], p5[63:54] }
8 assign p6x = p6 + p5
9 assign p7x = (even) ? { p7[36:0], p7[63:37] } : { p7[13:0], p7[63:14] }

```

با توجه به توضیحات در مورد ماژول ۱، در ماژول ۲ با حفظ کلیات جزئیات تغییر می‌کند.

- ماژول ۳

```

1 //module 3
2 assign p0x = p0 + p5
3 assign p1x = (even) ? { p1[46:0], p1[63:47] } : { p1[38:0], p1[63:39] }
4 assign p2x = p2 + p7
5 assign p3x = (even) ? { p3[14:0], p3[63:15] } : { p3[34:0], p3[63:35] }
6 assign p4x = p4 + p1
7 assign p5x = (even) ? { p5[27:0], p5[63:28] } : { p5[24:0], p5[63:25] }
8 assign p6x = p6 + p3
9 assign p7x = (even) ? { p7[24:0], p7[63:25] } : { p7[20:0], p7[63:21] }

```

- ماژول ۴

```

1 //module 4
2 assign p0x = p0 + p7
3 assign p1x = (even) ? { p1[19:0], p1[63:20] } : { p1[55:0], p1[63:56] }
4 assign p2x = p2 + p5
5 assign p3x = (even) ? { p3[ 7:0], p3[63: 8] } : { p3[41:0], p3[63:42] }
6 assign p4x = p4 + p3
7 assign p5x = (even) ? { p5[ 9:0], p5[63:10] } : { p5[ 7:0], p5[63: 8] }
8 assign p6x = p6 + p1
9 assign p7x = (even) ? { p7[54:0], p7[63:55] } : { p7[28:0], p7[63:29] }

```

ماژول‌های ۳ و ۴ هم در این مورد مشابهت دارند با توضیحات داده شده در مورد ماژول ۱،  $pkx$  های  $k$  زوج مجموع دو  $p$  هستند. اگر  $k$  فرد باشد،  $pkx$  برابر همان  $pk$  خواهد بود با این تفاوت که با توجه به صفر یا یک بودن  $even$  بین یکی از بیت های  $pk$  شکاف می‌اندازد و سمت چپ شکاف را در قسمت کم‌ارزش خود و سمت راست شکاف را در قسمت پرارزش پر می‌کند. (در واقع همان rotation که در توصیف الگوریتم بیان شد این‌جا دیده می‌شود).

- تنها always-block در ماژول (حساس به لبه مثبت کلاک) :

در این بلاک خروجی  $out$ ، به صورت مجموعه های ۶۴ بیتی مقداردهی می‌شود.  $Out$  ۵۱۲ بیتی است پس ۸ بار مقداردهی لازم است. برای هر ۴ ماژول مقدارهای نسبت داده شده به ۶۴ بیتی های  $out$  را (به ترتیب از پرارزش ترین ۶۴ بیت تا کم‌ارزش ترین آن) داریم :

## - ماژول ۱

| Index | 7        | 6        | 5                      | 4                      | 3        | 2                      | 1        | 0                      |
|-------|----------|----------|------------------------|------------------------|----------|------------------------|----------|------------------------|
| Value | $p_{0x}$ | $p_{2x}$ | $p_{1x} \wedge p_{0x}$ | $p_{3x} \wedge p_{2x}$ | $p_{4x}$ | $p_{5x} \wedge p_{4x}$ | $p_{6x}$ | $p_{7x} \wedge p_{6x}$ |

## - ماژول ۲

| Index | 7        | 6                      | 5        | 4                      | 3        | 2                      | 1        | 0                      |
|-------|----------|------------------------|----------|------------------------|----------|------------------------|----------|------------------------|
| Value | $p_{0x}$ | $p_{1x} \wedge p_{2x}$ | $p_{2x}$ | $p_{3x} \wedge p_{0x}$ | $p_{4x}$ | $p_{5x} \wedge p_{6x}$ | $p_{6x}$ | $p_{7x} \wedge p_{4x}$ |

## - ماژول ۳

| Index | 7        | 6                      | 5        | 4                      | 3        | 2                      | 1        | 0                      |
|-------|----------|------------------------|----------|------------------------|----------|------------------------|----------|------------------------|
| Value | $p_{0x}$ | $p_{1x} \wedge p_{4x}$ | $p_{2x}$ | $p_{3x} \wedge p_{6x}$ | $p_{4x}$ | $p_{5x} \wedge p_{0x}$ | $p_{6x}$ | $p_{7x} \wedge p_{2x}$ |

## - ماژول ۴

| Index | 7        | 6                      | 5        | 4                      | 3        | 2                      | 1        | 0                      |
|-------|----------|------------------------|----------|------------------------|----------|------------------------|----------|------------------------|
| Value | $p_{0x}$ | $p_{1x} \wedge p_{6x}$ | $p_{2x}$ | $p_{3x} \wedge p_{4x}$ | $p_{4x}$ | $p_{5x} \wedge p_{2x}$ | $p_{6x}$ | $p_{7x} \wedge p_{0x}$ |



## فصل ۳

# شبیه‌سازی

توصیف روند شبیه‌سازی سخت‌افزار و گام‌های اجرایی، مشاهده ورودی‌ها و خروجی‌های اصلی و میانی، مقایسه با مقادیر حاصل از اجرای کد نرم‌افزاری (مدل طلایی)، توصیف مراحل اجرای الگوریتم به همراه شکل موج‌ها، نحوه عملکرد Testbench

## ۱.۳ توضیح روند شبیه‌سازی سخت‌افزار و گام‌های اجرایی

برای شبیه‌سازی سخت‌افزاری کد verilog الگوریتم Skein را در محیط شبیه‌سازی Modelsim اجرا کردیم. گام‌های اجرایی به صورت کلی برای شبیه‌سازی کد سخت‌افزاری موارد زیر بود.

- مطالعه کد الگوریتم و تعیین ورودی‌ها
- نوشتن Testbench
- اجرای کد در محیط Modelsim با Testbench های مختلف
- گرفتن Waveform و مقادیر خروجی (اصلی و میانی)

## ۲.۳ مشاهده ورودی‌ها و خروجی‌های اصلی و میانی

در ادامه ابتدا کد های Testbench اجرا شده بر الگوریتم و سپس Waveform های حاصله و در انتها خروجی‌ها به صورت متنی آورده می‌شود.

### ۱.۲.۳ توضیح نحوه عملکرد Testbench

در ادامه ابتدا کد verilog نوشته‌شده برای Testbench آورده و سپس توضیحاتی درباره آن ایراد شده است.

## Testbench 1

```
1 //Master Testbench example
2
3 module skein_tb;
4
5 // Inputs
6 reg clk;
7 reg [511:0] midstate;
8 reg [95:0] data;
9 reg [31:0] nonce;
10
11 // Outputs
12 wire [511:0] hash;
13
14 // Instantiate the Unit Under Test (UUT)
15 skein512 uut (
16     .clk(clk),
17     .midstate(midstate),
18     .data(data),
19     .nonce(nonce),
20     .hash(hash)
21 );
22
23 initial begin
24     // Initialize Inputs
25     clk = 0;
26     midstate = 0;
27     data = 0;
28     nonce = 0;
29
30     // Wait 100 ns for global reset to finish
31     #1000
32     data = 512'd12345609823;
33     midstate = 96'd456;
34     nonce = 32'd453;
35     #1000;
36     data = 512'd76594320945555431222976000000000654;
37     midstate = 96'd456;
38     nonce = 32'd453;
39
40 end
41 always
42     #1 clk = clk;endmodule
```

## Testbench 2

```
1 module skein_tb;
2
3   // Inputs
4   reg clk;
5   reg [511:0] midstate;
6   reg [95:0] data;
7   reg [31:0] nonce;
8
9   // Outputs
10  wire [511:0] hash;
11
12  // Instantiate the Unit Under Test (UUT)
13  skein512 uut (
14    .clk(clk),
15    .midstate(midstate),
16    .data(data),
17    .nonce(nonce),
18    .hash(hash)
19  );
20
21  initial begin
22    // Initialize Inputs
23    clk = 0;
24    midstate = 0;
25    data = 0;
26    nonce = 0;
27
28    // Wait 100 ns for global reset to finish
29    #1000
30    data = "FF";
31    #5000;
32  end
33  always
34    #1 clk = clk;endmodule
```

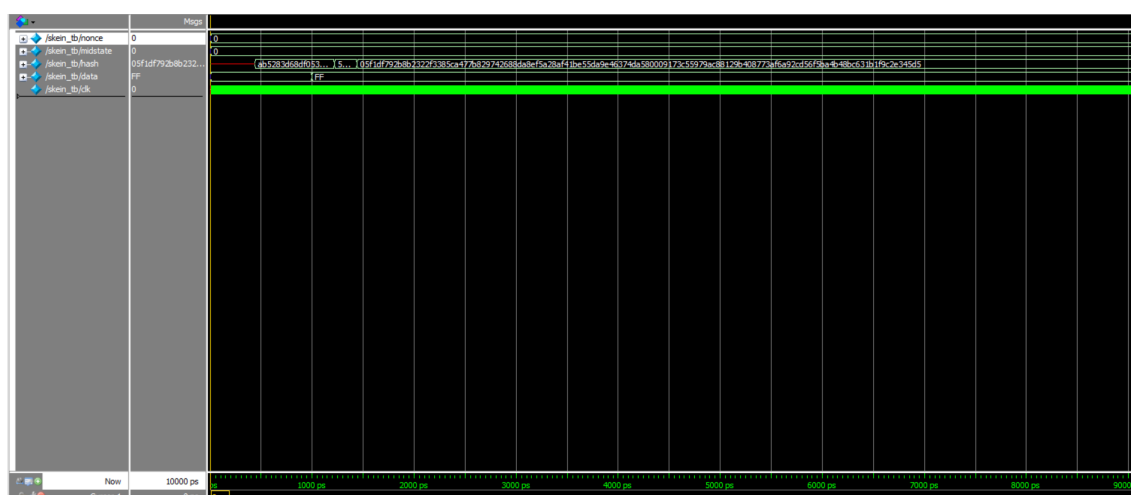
## ۲.۲.۳ شکل موج حاصل از Testbench

Waveform 1



شکل ۱.۳: شبیه‌سازی با Testbench 1

Waveform 2



شکل ۲.۳: شبیه‌سازی با Testbench 2

## ۳.۲.۳ جدول ورودی‌ها و خروجی‌های Testbench

| (clk) Time  | Data                                    | Nonce   | Midstate |
|-------------|---|---------|----------|
| 0 - 1000    | 0                                       | 0       | 0        |
| 1000 - 2000 | 512'd12345609823                        | 32'd453 | 96'd456  |
| 2000 - End  | 512'd7659432094555543122297600000000654 | 32'd453 | 96'd456  |

جدول ۱.۳: مقادیر ورودی‌ها و زمان برای Testbench 1

| (clk) Time | Data | Nonce | Midstate |
|------------|------|-------|----------|
| 0 - 1000   | 0    | 0     | 0        |
| 1000 - End | FF   | 0     | 0        |

جدول ۲.۳: مقادیر ورودی‌ها و زمان برای Testbench 2

| Time        | Hash  |
|-------------|---|
| 433 - 1217  | ab5283d68df053ac62d053789d4b45b81a02c959d7cab97fc43451166351f117f949fe918475f762ba80567046338211461648316d4432e6c505edc3b5ee6ff5  |
| 1217 - 1436 | dd477bfb0f07e299560b050c7aedb947bad77571f9a7d886a06f197a55f7946b8a9cecb948a5478380168f8bfaf8e6d7d828459564973272b18cdf99d0234f20c0dea4dfd9994c6eb97f500589565239347be8a5b2e4ce4832c6cc9095baa51 |
| 1436 - 2217 | bf2bdde45ef619f4086e71e7d86f637314357e6d20632c31612f5424644cc2236d383e0cceb223c20c45b816a165072ad200b8091682e8e5c31295ee62ca3719  |
| 2217 - 2437 | afbd493a4b85859d1cbe08d98bf01e66be18f3d3536987eeef06cc7965851bf86b722c1b1fb150c850e02ee44e03a447401ca4ac3cde4de6eb95b2e853d0d34b  |
| 2437 - End  | 53583685f4b21f9b98229734756d7b835e46c2f589e461ab7c3177fb7e572b64  |

جدول ۳.۳: مقادیر درهم‌سازی و زمان برای Testbench 1

| Time        | Hash   |
|-------------|--|
| 433 - 1217  | ab5283d68df053ac62d053789d4b45b81a02c959d7cab97fc43451166351f117f949fe918475f762ba80567046338211461648316d4432e6c505edc3b5ee6ff5 |
| 1217 - 1436 | 5e63442c883739354bfbf8008368ac0c09c61fa86b430b2864bdfe41bf48dd9b8f653c22f3b712ad81f285ce4e81ca5083a9edcc07f7ddfdb0748e5b8fca57a7 |
| 1436 - End  | 05f1df792b8b2322f3385ca477b829742688da8ef5a28af41be55da9e46374da580009173c55979ac88129b408773af6a92cd56f5ba4b48bc631b1f9c2e345d5 |

جدول ۴.۳: مقادیر درهم‌سازی و زمان برای Testbench 2

### ۳.۳ اجرا و تحلیل کد نرم‌افزاری (مدل طلایی)

به همراه پروژه کد C الگوریتم Skein نیز به عنوان مدل طلایی ارائه شد، در ادامه مختصراً کد C مدل طلایی را تحلیل می‌کنیم.

#### ۱.۳.۳ تحلیل کد C

تابع *skeinhash* با گرفتن ورودی *data* که به صورت آرایه ای از *unsignedchar* و *output* که به صورت آرایه ای از *uint8\_t* می‌باشد شروع می‌کند. با فراخواندن *sph\_skein512\_init* که ورودی از جنس *sph\_skein\_big - contex* می‌گیرد و سپس *sph\_skein512* که ورودی *sph\_skein\_big - contex* و داده و طول داده را می‌گیرد و در آخر *sph\_skein512\_close* که خروجی و *sph\_skein\_big - context* را به عنوان ورودی دارد کار خود را پایان می‌دهد و نتیجه را در آرایه خروجی به طول ۳۲ کپی می‌کند. در ابتدا *sph\_skein\_big - context* بررسی می‌شود: زمینه ای برای محاسبه ی اسکین شامل مقادیر واسطه و بخشی از داده از آخرین بلوک وارد شده است.

```
1 #ifndef DOXYGEN_IGNORE
2     unsigned char buf[64];      /* first field, for alignment */
3     size_t ptr;
4     sph_u64 h0, h1, h2, h3, h4, h5, h6, h7;
5     sph_u64 bcount;
6 #endif
7 } sph_skein_big_context;
```

- *buf*  
آرایه ای به طول ۶۴ که بخش به بخش داده را در خود ذخیره می‌کند و روی آن پردازش انجام می‌شود.

- *ptr*  
سایز بخش اشغال شده بافر

- *h0..h7*  
*h0, ..., h7*: برای محاسبه در *uint64<sub>t</sub>* استفاده می‌شود. UBI برای محاسبه در

- تعداد بلاک های داده  
*bcount*

تابع *skein\_big\_init* مقادیر *sph\_skein\_big - context* که از این به بعد از آن به اختصار *ctx* یاد میشود را مقدار دهی اولیه کرده و تمامی مقادیر آن را به جز بافر صفر می‌گذارد.

تابع *sph\_skein512* بخش اصلی محاسبات را به عهده دارد. اگر سایز بافر *ctx* خالی مانده بیشتر از طول داده باشد داده در آن کپی می‌شود.

سپس مقادیر *ctx* توسط تابع *READ\_STATE\_BIG* به روز رسانی می‌شوند و با مشخص کردن مقدار *first* که بعدتر از آن استفاده میکند و برابر با متغیر *first* در UBI می‌باشد با استفاده از *bcount* زمینه که در ابتدا برابر با ۰ است، وارد لوپ محاسبه می‌شود.

اگر بافر پر شده باشد، *bcount* به علاوه یک شده و *first* و *ptr* صفر می‌شوند. تابع *UBI\_BIG* با ورودی  $96 + first$  و ۰ فراخوانی می‌شود که همان *The Unique Block Iteration (UBI) chaining mode* است که یک مقدار زنجیره ای ورودی را با یک رشته ورودی با طول اختیاری ترکیب می‌کند و خروجی با طول ثابت را تولید می‌کند.

بلوک های پیام *M0* و *M1* و ... و *M7* هرکدام گنجایش ۶۴ بیت داده را دارند که به ترتیب توسط بافر *ctx* پر می‌شوند.

مقدار *p0p7* مربوط به *threefish* که متن ساده، یک رشته از بایت های با طولی برابر با کلید، است برابر

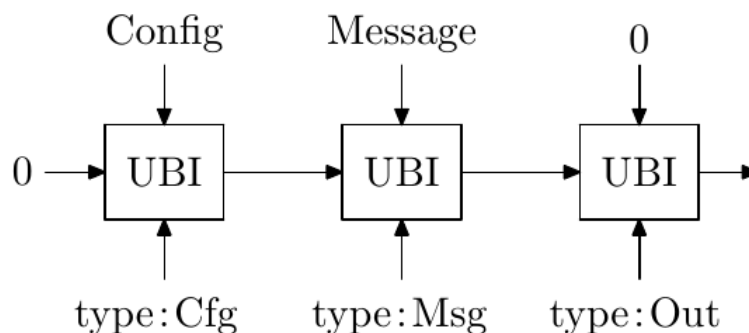
$m0..m7$  قرار می‌گیرد.

دو متغیر  $t0$  و  $t1$  (*threefishtweak*) با استفاده از  $bcount$  که متعلق به  $ctx$  است و ورودی‌های تابع  $UBI\_BIG$  محاسبه میشوند و تابع  $TFBIG\_KINIT$  با ورودی‌های  $h0h7$  مربوط به  $ctx$  و  $t0$  و  $t1$  که قبلتر محاسبه شد فراخوانی میشود که با استفاده از آن و  $TFBIG\_ADDKEY$  که با هم نقش *threefish* اسکین را به عهده دارند  $h0..h7$  مقدار دهی میشوند. به طوری که  $hn = mn \wedge pn$ .

و این مرحله تا وقتی داده‌ای که در بافر نرفته باقی مانده ادامه دارد و سپس تمامی مقادیر  $ctx$  در آن نوشته میشوند. (به جز بافر)

خروجی را در بافر ارائه شده میریزد و به پردازش خاتمه میدهد. زمینه یا  $ctx$  به طور خودکار دوباره مقدار دهی اولیه می‌شود. ورودی این تابع به جز زمینه و آرایه‌ی خروجی، تعداد بیت‌های اضافه  $n$  و خود بیت‌های اضافه  $ub$  میباشد که خود تابع دو ورودی آخر را مشخص می‌کند. اگر  $n$  صفر نباشد مقدار  $x$  ای با استفاده از این دو تولید میشود که  $sph\_skein512$  آن فراخوانی میشود. در این مرحله، اگر  $ptr == 0$  یعنی پیام خالی است؛ در غیر این صورت، بین ۱ تا ۶۴ بیت وجود دارد که هنوز پردازش نشده‌اند. در هر صورت بافر باید به یک بلوک کامل پر شده با صفر تبدیل شود (مشخصه Skein می‌گوید که پیام خالی پوشیده شده است تا حداقل یک بلوک برای پردازش وجود داشته باشد). هنگامی که این بلوک پردازش شده است، این فرآیند دوباره با بلوک پر از صفر، برای خروجی (آن بلوک *encoding* "۰"، بیش از ۸ بیت و سپس با صفر پر شده) انجام می‌شود.

همانطور که در شکل ۳.۳ میبینیم اسکین بر اساس چندین فراخوانی  $UBI$  ساخته شده است. نتیجه محاسبات بلوک پیکربندی  $UBI$  برای تمام پیام‌ها ثابت است و می‌تواند به صورت IV از پیش محاسبه شود.



شکل ۳.۳: نحوه کار Skein

```

1 static const sph_u64 IV512[] = {
2     SPH_C64(0x4903ADFF749C51CE), SPH_C64(0x0D95DE399746DF03),
3     SPH_C64(0x8FD1934127C79BCE), SPH_C64(0x9A255629FF352CB1),
4     SPH_C64(0x5DB62599DF6CA7B0), SPH_C64(0xEABE394CA9D5C3F4),
5     SPH_C64(0x991112C71A75B523), SPH_C64(0xAE18A40B660FCC33)
6 };
  
```



## ۲.۳.۳ مشاهده خروجی‌های کد C

کد C که به همراه پروژه قرار داشت را اجرا کردیم، در ادامه تصویری از کد اجراشده و جدول خروجی‌های آن آمده است.

```

1 #include "miner.h"
2
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <stdint.h>
6 #include <string.h>
7 #include <time.h>
8 #include "sph_skein.h"
9
10
11
12 void skeinhash(void *output, const void *input)
13 {
14     uint8_t hash[64];
15     sph_skein512_context ctx;
16
17     sph_skein512_init(&ctx);
18     sph_skein512(&ctx, input, 80);
19     sph_skein512_close(&ctx, (void*)hash);
20
21
22     memcpy(output, hash, 64);
23 }
24 void delay(unsigned int milliseconds){
25
26     clock_t start = clock();
27
28     while((clock() - start) * 1000 / CLOCKS_PER_SEC < milliseconds);
29 }
30 void print_hex(uint8_t *s, size_t len) {
31     //printf("%c", len);
32     for(int i = 0; i < 64; i++) {
33         printf("%x", s[i]);
34     }
35     printf("\n");
36 }
37
38 int main(int argc, const char * argv[]) {
39     uint8_t dst[64];
40     delay(100);
41     unsigned char buf[80] = "FF";
42     skeinhash(dst, buf);
43     for(int i = 0; i < 64; i++) {
44         printf("%x", dst[i]);
45         printf(" ");
46     }
47     printf("\n");
48
49     return 0;
50 }

```

```

36 }
37
38 int main(int argc, const char * argv[]) {
39     uint8_t dst[64];
40     delay(100);
41     unsigned char buf[80] = "FF";
42     //~29744672478978896854942682582";
43     skeinhash(dst, buf);
44     for(int i = 0; i < 64; i++) {
45         printf("%x", dst[i]);
46         printf(" ");
47     }
48     printf("\n");
49     return 0;
50 }
51
52

```

Output: d1 db 60 9 92 d1 3f f5 e8 18 ce 10 74 33 ad 2d ec 14 8c 32 ad 8 d4 61 68 3d 67 7b 7 dc f9 46 96 ce 76 43 d c7 20 6b f3 a0 f3 f2 12 f6 7b 90 1b bd  
ce 4e e2 e0 1b 8c b2 14 d0 89 30 df 9a b7  
Program ended with exit code: 0

شکل ۴.۳: اجرای کد C

```

13 {
14     uint8_t hash[64];
15     sph_skein512_context ctx;
16
17     sph_skein512_init(&ctx);
18     sph_skein512(&ctx, input, 80);
19     sph_skein512_close(&ctx, (void*)hash);
20
21     memcpy(output, hash, 64);
22 }
23
24 void delay(unsigned int milliseconds){
25
26     clock_t start = clock();
27
28     while((clock() - start) * 1000 / CLOCKS_PER_SEC < milliseconds);
29 }
30 void print_hex(uint8_t *s, size_t len) {
31     //printf("%c", len);
32     for(int i = 0; i < 64; i++) {
33         printf("%x", s[i]);
34     }
35     printf("\n");
36 }
37
38 int main(int argc, const char * argv[]) {
39     uint8_t dst[64];
40     delay(100);
41     unsigned char buf[80] = "29744672478978896854942682582";
42     skeinhash(dst, buf);
43     for(int i = 0; i < 64; i++) {
44         printf("%x", dst[i]);
45         printf(" ");
46     }
47     printf("\n");
48 }
49

```

Output: c7 1a 89 f7 64 27 fb 2 2b bc e4 f3 6f 93 8c 9 35 76 2c 33 92 18 3e 75 61 25 52 a4 34 e9 5a cc 4d ad c2 28 0 ea 7d 3e 7f 5b 74 da e0 98 cf bb 1c e0  
5 11 10 8e 49 7c 6c 67 95 31 c0 cc ec f  
Program ended with exit code: 0

شکل ۵.۳: اجرای کد C

Output 1

| input                          | output   |
|--------------------------------|--|
| -29744673475978895854942682582 | c71a59f76427fb022bbce4f36f938c0935762c3392183e75612552a434e95acc<br>4dad22800ea7d3e7f5b74dae098cfbb1ce00511188e697c6c679531c0cc0f<br>d1db600992d13ff5e818ce107433ad2dec148c32ad08d461683d677b07dcf946<br>FF 96ce76430dc7206bf3a0f37212f67b901bbdce4ee2e01b8cb214d08930df9ab7 |

جدول ۵.۳: جدول ورودی و خروجی‌های مدل طلایی

### ۳.۳.۳ مقایسه کد verilog با مدل طلایی و مرجع اصلی الگوریتم

همان‌طور که در خروجی‌های کد C و verilog مشهود است یکی از دو کد مدل طلایی یا کد سخت‌افزاری دارای اشکالاتی است که منجر به تفاوت خروجی می‌شود، طبق بررسی‌های انجام‌شده کد C با کد اصلی الگوریتم تطابق دارد و نتیجتاً اشکال از کد verilog است. در ادامه مشکلات یافت‌شده در کد verilog ذکر خواهد شد.

### ۴.۳.۳ مقایسه خروجی کد اصلی با داکيومنت مرجع

یکی از ورودی‌ها همانند ورودی مثال در داکيومنت مرجع داده شد اما خروجی‌ها یکی نبود، خروجی داکيومنت مرجع در ادامه آمده است، تیم پروژه موفق به یافت علت تفاوت در پاسخ‌ها نشد.

| input | output   |
|-------|--|
| FF    | 71B7BCE6FE6452227B9CED6014249E5BF9A9754C3AD618CCC4E0AAE16B316CC<br>8CA698D864307ED3E80B6EF1570812AC5272DC409B5A012DF2A579102F340617A |

جدول ۶.۳: جدول ورودی و خروجی‌های مستند مرجع

### ۵.۳.۳ اصلاحات موردنیاز کد verilog

در ابتدا لازم به ذکر است که تلاش شده تا الگوریتم با الگوریتم داکيومنت اصلی که توسط طراحان الگوریتم طراحی شده مطابقت داده شود، تمامی ارجاعات داخل متن به

The Skein Hash Function Family

Version 1.3 — 1 Oct 2010

<http://www.skein-hash.info/sites/default/files/skein1.3.pdf>

خواهد بود.

در مازول‌های چهارگانه  $skein\_round\_x$  در اساینمنت  $pkx$  ها ( $k$  متغیر) عمل  $MIX$  در حال انجام است. به این شکل که برای  $k$  های زوج جمع ساده دو کلمه از ورودی و برای  $k$  های فرد با توجه به مقدار  $even$ ، عمل  $left - rotation$  انجام میشود و سپس با یکی از  $k$  های زوج  $xor$  میشود تا به خروجی متصل شود.  $Left - rotation$  به اندازه  $x$  یعنی همه ی بیت های یک کلمه (۶۴ بیتی) به اندازه  $x$  بیت به سمت چپ منتقل شوند و بیت هایی که از مرز سمت چپ کلمه (بیت ۶۴ ام) بیرون میزنند، از سمت راست با همان ترتیب وارد میشوند.

در مازول‌های  $skein\_round\_2$  در خط ۴۱۶ و  $skein\_round\_4$  در خط ۵۰۲ assignment های مربوط به  $p3x$  و  $p7x$  با توجه به جدول شماره ۴ از صفحه ۱۱ منبع ذکر شده، جابه‌جا نوشته شده اند و نیاز است

تصحیح شود.

عبارت صحیح برای *skein\_round\_2* :

```
1 //Modification of skein_round_2
2 assign p0x = p0 + p3;
3 assign p1x = (even) ? { p1[30:0], p1[63:31] } : { p1[50:0], p1[63:51] };
4 assign p2x = p2 + p1;
5 assign p3x = (even) ? { p3[36:0], p3[63:37] } : { p3[13:0], p3[63:14] };
6 assign p4x = p4 + p7;
7 assign p5x = (even) ? { p5[49:0], p5[63:50] } : { p5[53:0], p5[63:54] };
8 assign p6x = p6 + p5;
9 assign p7x = (even) ? { p7[21:0], p7[63:22] } : { p7[46:0], p7[63:47] };
```

عبارت صحیح برای *skein\_round\_4* :

```
1 //Modification of skein_round_4
2 assign p0x = p0 + p7;
3 assign p1x = (even) ? { p1[19:0], p1[63:20] } : { p1[55:0], p1[63:56] };
4 assign p2x = p2 + p5;
5 assign p3x = (even) ? { p3[54:0], p3[63:55] } : { p3[28:0], p3[63:29] };
6 assign p4x = p4 + p3;
7 assign p5x = (even) ? { p5[ 9:0], p5[63:10] } : { p5[ 7:0], p5[63: 8] };
8 assign p6x = p6 + p1;
9 assign p7x = (even) ? { p7[ 7:0], p7[63: 8] } : { p7[41:0], p7[63:42] };
```

در ماژول‌های *skein\_round\_x* (هر چهار ماژول) حین اتصال کردن *pkx* ها به *out* از روند خاصی که درجدول شماره ۳ صفحه ۱۱ منبع ذکر شده باید پیروی کرد.

با توجه به اعداد مربوط به ماژول‌های ۸ کلمه‌ای، خروجی‌ها را باید به *pkx* های مناسب متصل کرد که در ماژول‌ها رعایت نشده است.

محتویات *always – block* داخل هر چهار ماژول *skein\_round\_x* باید شکل زیر تغییر کند:

```
1 //Modificaiotion of skein_round_x ( x = 1,2,3,4)
2 out [511:448] <= p2x;
3 out [447:384] <= p0x ^ p1x;
4 out [383:320] <= p4x;
5 out [319:256] <= p6x ^ p7x;
6 out [255:192] <= p6x;
7 out [191:128] <= p4x ^ p3x;
8 out [127: 64] <= p0x;
9 out [ 63: 0] <= p2x ^ p3x;
```

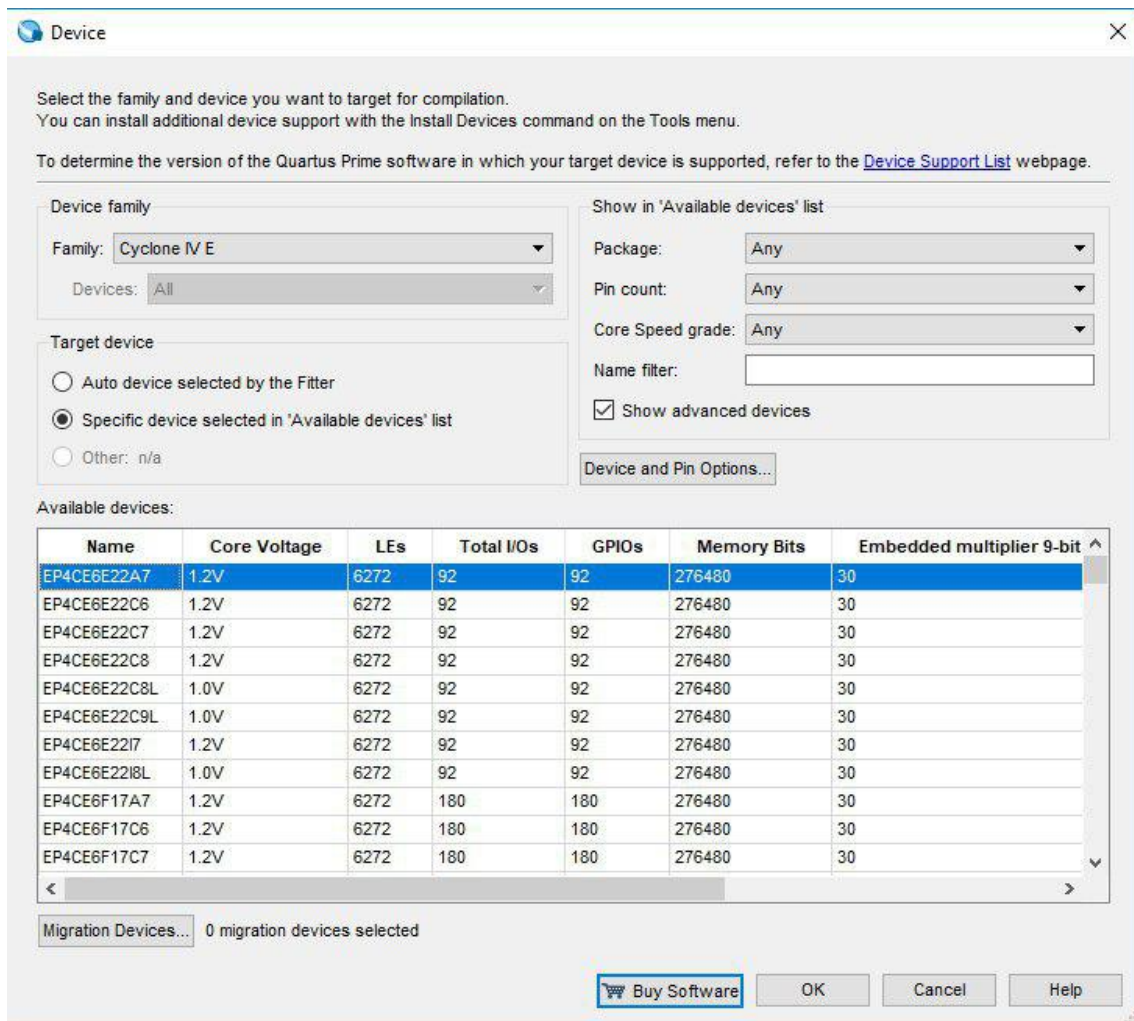
## فصل ۴

# پیاده‌سازی سخت‌افزاری

سنتز سیستم روی *FPGA* ، بررسی ایرادات سنتز و ارائه راهکار، گزارش پیاده‌سازی

## ۱.۴ مقدمه‌ای بر پیاده‌سازی سخت‌افزاری

کد verilog داده‌شده در پروژه با استفاده از ابزار Quartus برای سنتز بر روی FPGA شبیه‌سازی شد، در ادامه مختصراً به شرح ایرادات سنتز و سپس به گزارش آمارهای حاصل از سنتز می‌پردازیم.

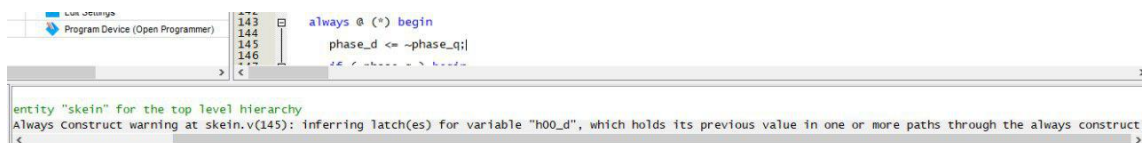


شکل ۱.۴: انتخاب وسیله برای سنتز در محیط Quartus

## ۲.۴ ایرادات سنتز و ارائه راهکار

برای شبیه‌سازی سخت‌افزاری دو ایراد وجود داشت.

- نخست این که نام ماژول و فایل یکسان نبود و ابزار Quartus در سنتز به مشکل می‌خورد.
- پس از رفع ایراد پیشین ابزار هنگام سنتز اخطار زیر را می‌داد.



شکل ۲.۴: اخطار اولیه شبیه‌ساز

برای حل این مشکل خط ۱۴۳ کد وریلاگ از

```
1 always @ (*) begin
```

به

```
1 always @ (posedge clk) begin
```

تغییر کرد.

## ۳.۴ گزارش پیاده‌سازی

پس از سنتز موفق سخت‌افزاری گزارش پیاده‌سازی توسط شبیه‌ساز ارائه شد، تصاویر ۳.۴ و ۴.۴ مقادیر گزارش شبیه‌ساز پس از سنتز اند.

### Slice Logic Distribution:

|                                     |       |        |       |     |
|-------------------------------------|-------|--------|-------|-----|
| Number of LUT Flip Flop pairs used: | 83451 |        |       |     |
| Number with an unused Flip Flop:    | 6231  | out of | 83451 | 7%  |
| Number with an unused LUT:          | 19932 | out of | 83451 | 23% |
| Number of fully used LUT-FF pairs:  | 57288 | out of | 83451 | 68% |
| Number of unique control sets:      | 39    |        |       |     |

شکل ۳.۴: گزارش تعداد flip flop ها و LUT ها

```
Minimum period: 2.806ns (Maximum Frequency: 356.398MHz)
Minimum input arrival time before clock: 1.290ns
Maximum output required time after clock: 0.580ns
Maximum combinational path delay: No path found
```

شکل ۴.۴: گزارش فرکانس و دیگر زمان‌ها در سخت‌افزار