國立臺灣大學電機資訊學院電子工程學研究所

碩士論文

Graduate Institute of Electronics Engineering
College of Electrical Engineering and Computer Science
National Taiwan University

Master Thesis

SHA512-256d 演算法硬體架構實現於 FPGA

Hardware Implementation of SHA512-256d Algorithm on FPGA

趙彥安

Yen-An Chao

指導教授：李致毅 博士

Advisor: Jri Lee, Ph.D.

中華民國 112 年 6 月

June 2023

doi:10.6342/NTU202301593

國立臺灣大學碩士學位論文
口試委員會審定書

MASTER'S THESIS ACCEPTANCE CERTIFICATE
NATIONAL TAIWAN UNIVERSITY

SHA512-256d 演算法硬體架構實現於 FPGA

Hardware Implementation of SHA512-256d Algorithm on FPGA

本論文係趙彥安(R10943028)在國立臺灣大學電機資訊學院電子工程學研究所完成之碩士學位論文，於民國 112 年 06 月 29 日承下列考試委員審查通過及口試及格，特此證明。

The undersigned appointed by the Institute of Electronics Engineering on 29-06-2023 have examined a Master's thesis entitled above presented by Yen-An Chao(R10943028) candidate and hereby certify that it is worthy of acceptance.

口試委員 Oral examination committee:

（指導教授 Advisor）

系主任/所長 Director: 江介宏

doi:10.6342/NTU202301593

國立臺灣大學

學位論文學術倫理暨原創性聲明書

1. 本人已經自我檢核，確認無違反學術倫理情事，論文倘有造假、變造、抄襲、由他人代寫，或涉其他一切有違著作權及學術倫理之情事，及衍生相關民、刑事責任，概由本人負責，概無異議。
2. 本人之學位論文已確實經本校論文內容相似度比對系統檢核，內容比對相似度為 14 %（如有需要得另補充說明），符合系、所、學位學程自訂標準。

聲明人：趙彥安

學號：R10943028

中華民國 112 年 6 月 29 日

指導教授簽章：

共同指導教授簽章（無免）：

系所（學位學程）主管簽章：

備註：研究生應於繳交學位論文前完成論文相似度比對作業，並將本聲明書送交指導教授及系、所、學位學程主管簽章，本聲明書正本由各系、所、學位學程留存備查。

doi:10.6342/NTU202301593

摘要

本論文所提及的 SHA512-256d 演算法為 Radiant 區塊鏈的演算法，區塊鏈是一種分散式的資料庫，可以儲存各種交易和資訊，而且不需要中央機構來驗證或管理，不受政府或銀行的控制。其中最為有名的區塊鏈是比特幣，其發明者是一位化名為中本聰的人物，他在 2008 年發表了一篇論文，介紹了比特幣的設計原理和運作方式。本論文所提及的 Radiant 改良自比特幣，將比特幣使用的 SHA256 演算法更進為 SHA512-256d 演算法，來提供一種安全、透明和更加去中心化的方式來進行網路交易。

Radiant 使用工作量證明機制(Proof of Work)，工作量證明透過消耗大量硬體資源、電力、時間來解碼一個複雜的數學問題，成功解碼的人才能創建新的區塊，並獲得獎勵。為了維持這個機制，必須操作 SHA512-256d 演算法並得到小於特定值的答案，其中單位時間內的計算次數則稱為算力(hash rate)，而論文內容將會針對如何提升 SHA512-256d 演算法的算力來討論，並以 FPGA 實作。

SHA512-256d 演算法的特性是 compute-bound 和 compute-hard，也就是會需要大量運算資源和暫存器，其目的是為了要防止 ASIC 以及 FPGA 利用硬體高運算效率的特性，導致算力被主宰。SHA512-256d 演算法將 SHA512 演算法輸入兩次，並返回前 256 位作為輸出。相較比特幣使用的 SHA256 演算法，這種方法可以提高安全性，提升了用來計算的資料寬度，而這也導致 ASIC 及 FPGA 面積的使用大大提升，成本也隨之上升，使 ASIC 及 FPGA 無法完美發揮其運算快速之優勢。儘管如此，本論文仍將利用現有 FPGA 的有限資源進行分析及實作。

本論文將 SHA512-256d 演算法實作於 Xilinx FPGA VU33P 晶片上，並使用 Xilinx 的開發軟體 Vivado 進行 synthesis、routing 和 routing，藉由分析原版 SHA512-256d 演算法缺點，將架構加以改良，使算力能進一步提升。論文中詳述

doi:10.6342/NTU202301593

SHA512-256d 演算法分析、原版硬體架構缺點之分析、演算法新硬體架構設計、FPGA 硬體資源整理分配、最後實作以及在 FPGA 進行 emulation 的結果。論文最後整理出了原版及新版架構的算力差距、資源使用，以及新版 emulation 的結果。

關鍵字: 現場可程式化邏輯閘陣列、區塊鏈、Radiant、SHA512-256d, SHA512-256, RXD

doi:10.6342/NTU202301593

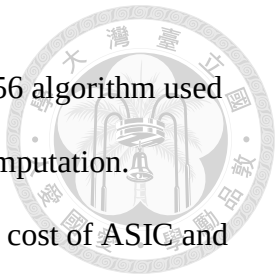
Abstract

The SHA512-256d algorithm discussed in this thesis is the algorithm used by the blockchain Radiant. Blockchain is a decentralized database that can store various transactions and information without the need for a central authority to verify or manage it, making it independent of governments or banks. The most famous blockchain is Bitcoin, which was introduced in a paper by an anonymous person or group named Satoshi Nakamoto in 2008, explaining the design principles and operation of Bitcoin. Instead of using the SHA256 algorithm employed by Bitcoin, Radiant utilizes the SHA512-256d algorithm. This enhancement aims to provide a more secure, transparent, and decentralized way of conducting network transactions.

Radiant utilizes Proof of Work, in which participants significant hardware resources, electricity, and time to solve a complex mathematical problem, and only those who successfully solve the problem can create new blocks and receive rewards. To maintain this mechanism, the SHA512-256d algorithm is operated, and the goal is to obtain an answer that is less than a specific value. The number of calculations performed per unit of time is referred to as the hash rate, and the paper discusses how to enhance the hash rate of the SHA512-256d algorithm, specifically through FPGA implementation.

The SHA512-256d algorithm is characterized as compute-hard, meaning it requires a significant amount of computational resources and registers to prevent ASIC and FPGA devices from dominating the hashing power by their hardware efficiency. The SHA512-256d algorithm takes the SHA512 algorithm as input twice and returns

doi:10.6342/NTU202301593



the first 256 bits as the output. This approach, compared to the SHA256 algorithm used in Bitcoin, enhances security by increasing the data width used for computation. However, this also leads to a significant increase in the utilization and cost of ASIC and FPGA resources, limiting their ability to fully exploit their computational speed advantages. Nonetheless, this paper will analyze and implement the algorithm using the limited resources of existing FPGAs.

This paper implements the SHA512-256d algorithm on the Xilinx FPGA VU33P chip using Xilinx's development software, Vivado, for synthesis, routing, and placement. By analyzing the drawbacks of the original SHA512-256d algorithm, the architecture is improved to further enhance the hash rate. The paper provides a detailed analysis of the SHA512-256d algorithm, an analysis of the drawbacks of the original hardware architecture, the design of the new hardware architecture for the algorithm, allocation of FPGA hardware resources, implementation details, and the results of emulation on the FPGA. The paper concludes by presenting the difference in hash rates and resource utilization between the original and new architectures, as well as the results of the new architecture's emulation on the FPGA.

Keywords: FPGA, blockchain, Radiant, SHA512-256d, SHA512-256, RXD

doi:10.6342/NTU202301593

Contents

doi:10.6342/NTU202301593

doi:10.6342/NTU202301593

doi:10.6342/NTU202301593

List of Figures

Figures of Chapter 2

Figures of Chapter 3

doi:10.6342/NTU202301593

doi:10.6342/NTU202301593

doi:10.6342/NTU202301593

List of Tables

Tables of Chapter 2

Tables of Chapter 3

doi:10.6342/NTU202301593

Tables of Chapter 4

doi:10.6342/NTU202301593

doi:10.6342/NTU202301593

Chapter 1 Introduction

1.1 Motivation

Blockchain is a decentralized network, meaning it operates without a central authority to guarantee the accuracy of new data. To establish consensus within this distributed system, various methods are employed to verify the validity of new transactions. One widely known method is Proof of Work. In a blockchain, participants are required to generate a Proof of Work by performing intricate mathematical calculations before they can finalize a transaction. This process allows participants within the blockchain to ensure data integrity by verifying the correctness of the generated answer.

In the blockchain, all data, including transactions, is organized into blocks. Participants seeking to complete the Proof of Work process must generate a hash. The blockchain algorithm will accept a hash with a value smaller than a predetermined threshold. The first person who calculates and submits a valid hash is granted the authority to create a new block and record transactions. This newly created block, containing the transaction information, is then updated to the blockchain network as the next block in the chain.

To fulfill the requirements of the Proof of Work algorithm, individuals must hash a random value called nonce together with the data from the previous block, resulting in a new hash value. As the data of the previous block remains constant, the objective is to find a valid hash among numerous nonce values. This task, known as

doi:10.6342/NTU202301593

mining, is doing by people referred to as miners, who diligently search for the elusive nonce.

Nevertheless, due to the vast and highly randomized range of nonce values (2^32), locating a suitable nonce can be a challenging task. Miners who possess faster computational capabilities have a greater advantage in achieving success. This has led to the advantage of specialized hardware such as Application-Specific Integrated Circuits (ASICs) and Field-Programmable Gate Arrays (FPGAs), which leverage their superior computing power to outperform other devices like Graphic Processing Units (GPUs) or Central Processing Units (CPUs). Therefore, cryptocurrency algorithms are often designed with measures in place to prevent ASICs and FPGAs from dominating the hash rate, ensuring a more equitable mining environment. As a result, in the present day, there is a preference to utilize GPUs for mining rather than ASICs or FPGAs. This is driven by the fact that GPUs offer a more accessible and widespread option for mining, allowing a larger number of people to participate in the process.

The Radiant blockchain employs the SHA512-256d algorithm, which is computationally intensive, requiring a substantial amount of computing resources. Moreover, in contrast to Bitcoin's SHA256 algorithm, SHA512-256d utilizes double data width, resulting in higher resource consumption. It makes nobody has ever tried to mine Radiant on FPGA yet.

Based on the author's previous experience of implementing the SHA256 algorithm on an FPGA in the Laboratory, it is indeed feasible to consider exploring a similar approach for the SHA512-256d algorithm. The central focus of the thesis revolves around the analysis and subsequent redesign of the architecture's critical path, making the design calculate more hashes in a second. Additionally, the design

doi:10.6342/NTU202301593

methodology employed in this study holds the potential to be adapted for designing architectures on various FPGA devices. Consequently, the author perceives the design of the architecture for SHA512-256d as an excellent opportunity for valuable training and a research topic.

1.2 Organization of the Thesis

The thesis content is presented as follows. Chapter 2 provides an introduction to the SHA512-256d algorithm's characteristics and highlights the shortcomings of the old architecture. In Chapter 3, the improved design is presented, including block diagrams and a discussion of the encountered challenges. The primary focus of the improvement lies in optimizing the critical path to increase the computation of hash in a period. Chapter 4 compares the architectures and presents performance measurements for the improved architecture, including the hash rate achieved. The results of the designs are summarized, leading to a conclusive analysis. In Chapter 5, the potential future work is explored and discussed.

doi:10.6342/NTU202301593

Chapter 2 Preliminary and the problems of original architecture

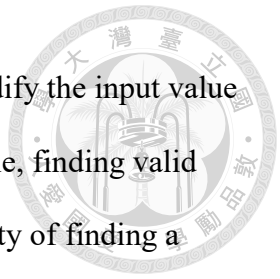
2.1 Introduction to Hash function and Radiant

Before introducing the SHA512-256d algorithm, it is important to first understand the properties of the hash function. The main characteristic of the hash function is its "one-way". This means that it is a function that takes an input string of variable length and converts it into a fixed-length output. Due to the complexity of the hash function, it is computationally difficult to invert the process.

Another key property of the hash function is that two identical input strings will always produce the same outputs, while two different output strings must have been generated from two different inputs. This property ensures the consistency of the hash function, making it easy to verify the integrity of data. Moreover, it also makes it extremely difficult for an attacker to derive the original input from the output alone, adding additional security that makes it become an essential tool in various applications such as data integrity verification and digital signatures. These have made hash function algorithms indispensable in the design and implementation of secure and decentralized blockchain systems.

As mentioned above, a key characteristic of hash functions is that even a slight difference in the input data will result in a completely different output. Due to the inherent difficulty of reversing the hash function process, the only way to generate valid hash values is through trial and error with different input values. This is where the

doi:10.6342/NTU202301593



parameter called a nonce comes into play, as it can be adjusted to modify the input value of the hash function. Since the resulting hash values are not predictable, finding valid hash values is purely a probabilistic endeavor. However, the probability of finding a valid hash value can be increased by the speed of the addition of computations. Therefore, it becomes essential to enhance the computing speed to improve the efficiency of finding valid hash values.

Radiant is a decentralized digital asset system that facilitates direct value exchange without going through a central party. This blockchain network was launched on June 21st, 2022, with a projected operational lifespan of at least two years. The chosen consensus employed by Radiant is Proof of Work, using the SHA512-256d algorithm. This thesis will introduce and delve into the workings of the SHA512-256d algorithm within the context of Radiant.

2.2 The property of SHA512-256d algorithm

2.2.1 Overview of SHA512-256d algorithm

SHA512-256d is a compute-hard algorithm that requires lots of computing resources and registers. Serving as one of the Proof of Work hash functions, the SHA512-256d algorithm takes the header and nonce as inputs and produces a hash as output. The header is 76 bytes long information from the block, comprising a 4-byte version number, a 32-byte previous block hash, a 32-byte Merkle root, a 4-byte timestamp of the block, and a 4-byte long difficulty target.

Since the header is obtained from the blockchain, it remains constant during the calculation process. On the other hand, being a random number, the nonce will

doi:10.6342/NTU202301593

change and calculate continuously until a valid nonce that meets a specific condition is discovered. In Proof of Work, a nonce is deemed valid only if the resulting hash that hashed with it is smaller than the difficulty target set by the blockchain. Therefore, the objective of the thesis is to devise a computing machine that is capable of calculating hashes swiftly with nonces, optimizing for speed and efficiency.

The software architecture of SHA512-256d can be seen in Figure 2.1.

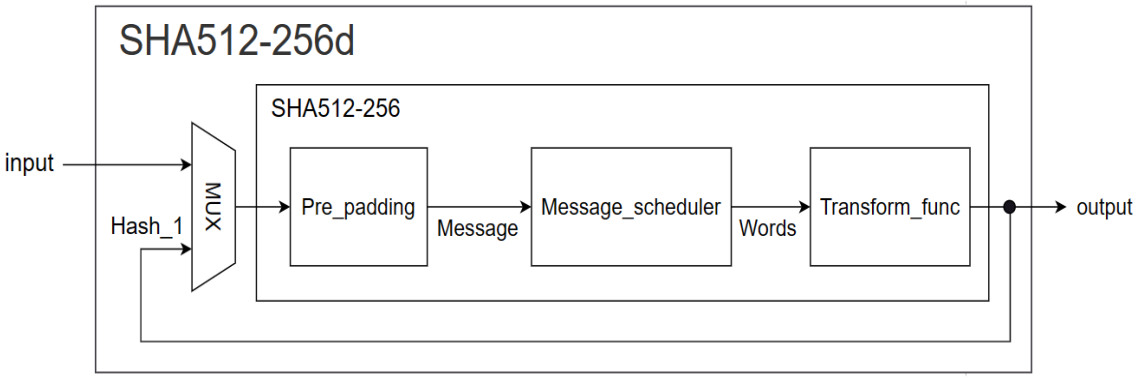


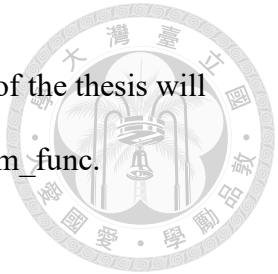
Figure 2.1 The architecture of SHA512-256d

In SHA512-256d, the SHA512-256 algorithm needs to be executed twice. The initial input for the first SHA512-256 iteration is constructed by combining the header and nonce, and then the output hash, named Hash_1, from the first iteration is used as the input for the second SHA512-256 iteration.

The input data for the Pre-padding will be padded to achieve a length of 1024 bits. This padding process guarantees that the input data meets the required length. The resulting 1024 bits of data, referred to as Messagare is then partitioned into 16 message blocks, with each message block consisting of 64 bits. These 16 message blocks are sent into the Message_scheduler for further processing. Finally, the generated 5120 bits of data are sent to the Transform_func and lead to the final value. Given that Pre_padding

doi:10.6342/NTU202301593

simply involves appending additional bits to the input data, the focus of the thesis will be on the introduction and design of Message_scheduler and Transform_func.



2.2.2 Software design of SHA512-256d functions

```
Message_scheduler()
{
    for (i = 0; i < 80; i ++)
    {
        if (i<16)
        {
            Words[i] = Message[i] ;
        }
        else
        {
            Words[i] = σ1 (Words[i-2]) + Words[i-7] + σ0 (Words[i-15]) + Words[i-16] ;
        }
    }
}
```

According to the C code above, the initial 16 words will be assigned values based on the input Message, and then the remaining words, from 16 to 79, will derive their values through three adder computations. As these words depend on previous words, numbers of registers are required to store during the computation process. The operation function of σ_1 and σ_0 will be determined in Table 2.2.

a + b	Addition modulo 2^{64}
a & b	Bitwise AND operation
a ^ b	Bitwise XOR operation (exclusive-OR)
~a	Bitwise complement operation
SHR^n (a) (= a >> n)	Right shift operation, the value is assigned by discarding the rightmost n bits of the word a and then padding the result with n zeros on the left.
$ROTR^n$ (a)	Rotate right operation, the value is equal to (x >> n) or (x << 64 - n).

Table 2.1 The operations used in SHA512-256d

doi:10.6342/NTU202301593

1.	$\sigma_0(x) = ROTR^1(x)\ \hat{}\ ROTR^8(x)\ \hat{}\ SHR^7(x)$
2.	$\sigma_1(x) = ROTR^{19}(x)\ \hat{}\ ROTR^{61}(x)\ \hat{}\ SHR^6(x)$
3.	$\Sigma_0(x) = ROTR^{28}(x)\ \hat{}\ ROTR^{34}(x)\ \hat{}\ ROTR^{39}(x)$
4.	$\Sigma_1(x) = ROTR^{14}(x)\ \hat{}\ ROTR^{18}(x)\ \hat{}\ ROTR^{41}(x)$
5.	$Ch(x,y,z) = (x\ \&\ y)\ \hat{}\ (\sim x\ \&\ z)$
6.	$Maj(x,y,z) = (x\ \&\ y)\ \hat{}\ (x\ \&\ z)\ \hat{}\ (y\ \&\ z)$

Table 2.2 6 special logic functions in SHA512-256d

Transform_func()
{
    $H = H_{init\_7}$;
    $G = H_{init\_6}$;
    $F = H_{init\_5}$;
    $E = H_{init\_4}$;
    $D = H_{init\_3}$;
    $C = H_{init\_2}$;
    $B = H_{init\_1}$;
    $A = H_{init\_0}$;

    for (i = 0; i < 80; i ++)
    {
        $T_1 = H + \Sigma_1(E) + Ch(E, F, G) + K_i + W_i$;
        $T_2 = \Sigma_0(A) + Maj(A, B, C)$;
        $H = G$;
        $G = F$;
        $F = E$;
        $E = D + T_1$;
        $D = C$;
        $C = B$;
        $B = A$;
        $A = T_1 + T_2$;
    }

    $D = D + H_{init\_3}$;
    $C = C + H_{init\_2}$;
    $B = B + H_{init\_1}$;
    $A = A + H_{init\_0}$;

    Hash = A || B || C || D ;
}

doi:10.6342/NTU202301593

H_{init_0}	0x22312194FC2BF72C
H_{init_1}	0x9F555FA3C84C64C2
H_{init_2}	0x2393B86B6F53B151
H_{init_3}	0x963877195940eabd
H_{init_4}	0x96283ee2a88effe3
H_{init_5}	0xbe5e1e2553863992
H_{init_6}	0x2b0199fc2c85b8aa
H_{init_7}	0x0eb72ddc81c52ca2

Table 2.3 The initial hash values in SHA512-256d

The eight initial hash values can be found in Table 2.3, while the values of K_i from 0 to 79 are listed in Table A.1 in Appendix A. In the Transform_func, the first step involves initializing eight working variables, denoted as A, B, C, D, E, F, G, and H, with the corresponding initial hash values. Following this initialization, a loop comprising 80 rounds is executed. At the end of the loop, each working variable is required to be added to its corresponding initial value. Finally, the output hash is obtained by concatenating the values of A, B, C, and D. It is important to note that apart from the output hash, which is 256 bits in length, all other variables involved in the calculations are 64 bits. To summarize, the complete SHA512-256d algorithm has been presented, including the initialization of working variables, the execution of 80 rounds, and the generation of the final output hash.

2.3 Problem Statement

doi:10.6342/NTU202301593

2.3.1 Introduction of FPGA VU33P

In this section, we will introduce the FPGA VU33P, which is the primary device utilized in our laboratory. The VU33P is one of the Vertex Ultrascale+ series manufactured by Xilinx which t comes equipped with 8 GB of High Bandwidth Memory (HBM) and is widely employed in our research. However, since the SHA512-256d algorithm does not require extensive memory usage, the HBM feature will not be utilized in this thesis.

	VU31P	VU33P	VU35P	VU37P	VU45P	VU47P	VU57P
System Logic Cells	961,800	961,800	1,906,800	2,851,800	1,906,800	2,851,800	2,851,800
CLB Flip-Flops	879,360	879,360	1,743,360	2,607,360	1,743,360	2,607,360	2,607,360
CLB LUTs	439,680	439,680	871,680	1,303,680	871,680	1,303,680	1,303,680
Max. Distributed RAM (Mb)	12.5	12.5	24.6	36.7	24.6	36.7	36.7
Block RAM Blocks	672	672	1,344	2,016	1,344	2,016	2,016
Block RAM (Mb)	23.6	23.6	47.3	70.9	47.3	70.9	70.9
UltraRAM Blocks	320	320	640	960	640	960	960
UltraRAM (Mb)	90.0	90.0	180.0	270.0	180.0	270.0	270.0
HBM DRAM (GB)	4	8	8	8	16	16	16
CMTs (1 MMCM and 2 PLLs)	4	4	8	12	8	12	12
Max. HP I/O(1)	208	208	416	624	416	624	624
DSP Slices	2,880	2,880	5,952	9,024	5,952	9,024	9,024

Figure 2.2 Feature summary of the series of Xilinx Virtex Ultrascale+ [8]

As shown in Figure 2.2, VU33P composed of 439,680 Look Up Tables (LUTs) and 879,360 Flip-Flops (FFs), also features amounts of Block RAM (BRAMs) and UltraRAM (URAMs). However, with some routing issues, BRAM and URAM can not be used in this design, and the reason will be discussed in section 3.5.3. Besides, the Digital Signal Processors (DSPs) are also not recommended to be used in this design. The reason is that the SHA512-256d algorithm runs with 64 bits of data, while the input width of DSP is just 48 bits, causing redundant and routing congestion due to requirements in the design.

doi:10.6342/NTU202301593

2.3.2 Original hardware architecture

Since the author is aiming for a higher hash rate, which refers to the number of hashes that can be computed within a given period, it is not suitable to wait until all computations of SHA512-256d are finished before calculating the next input. In the thesis, the design implements full pipelining, allowing hashes to be computed one-by-one in cycles, even though the expense of increased area and circuit congestion.

In this subsection, the original hardware architecture of SHA512-256d and Transform_func are presented with a block diagram. The block diagram of SHA512-256d is depicted in Figure 2.3, and the Transform_func is shown in Figure 2.4.

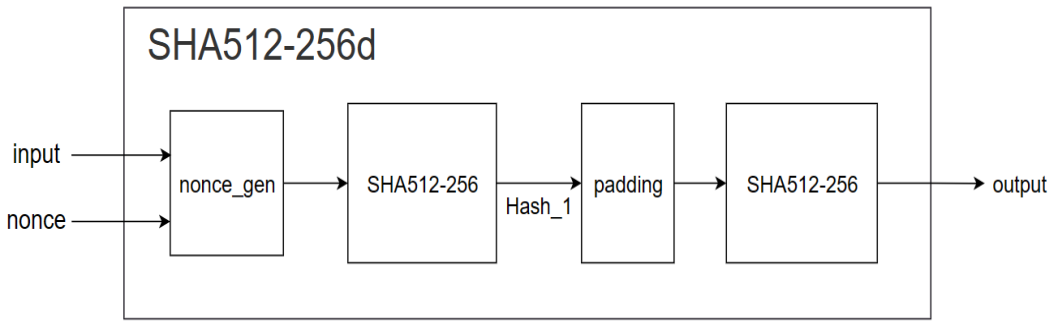


Figure 2.3 The block diagram of the SHA512-256d function

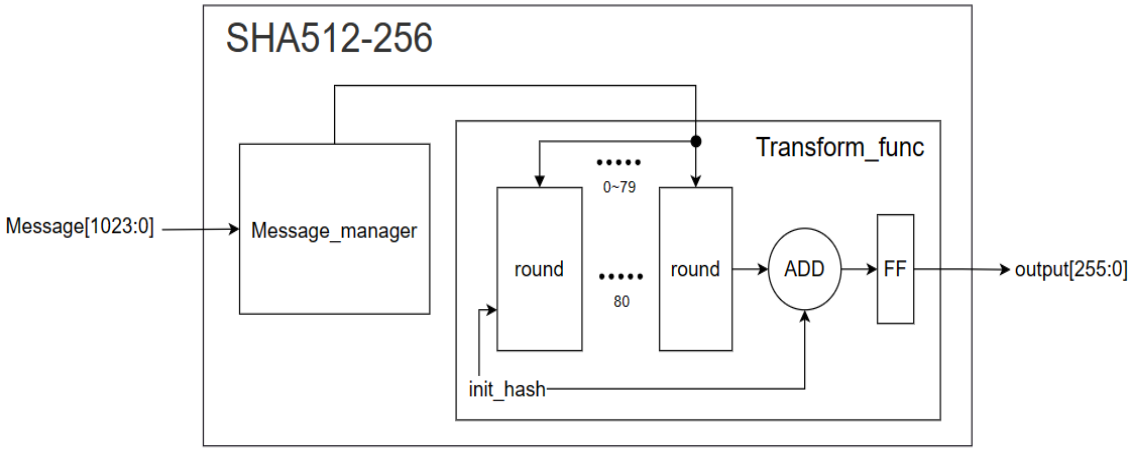


Figure 2.4 The block diagram of SHA512-256 and Transform_func

doi:10.6342/NTU202301593

From the C code of Transform_func provided in section 2.2.2, it is evident that each round within the Transform_func relies on the most complicated and time-consuming function. The critical path can be described by the following equation:

$$T_1 = H + \Sigma_1(E) + Ch(E, F, G) + K_i + W_i$$
$$T_2 = \Sigma_0(A) + Maj(A, B, C)$$
$$A = T_1 + T_2 = H + \Sigma_1(E) + Ch(E, F, G) + K_i + W_i + \Sigma_0(A) + Maj(A, B, C) \quad (2.1)$$

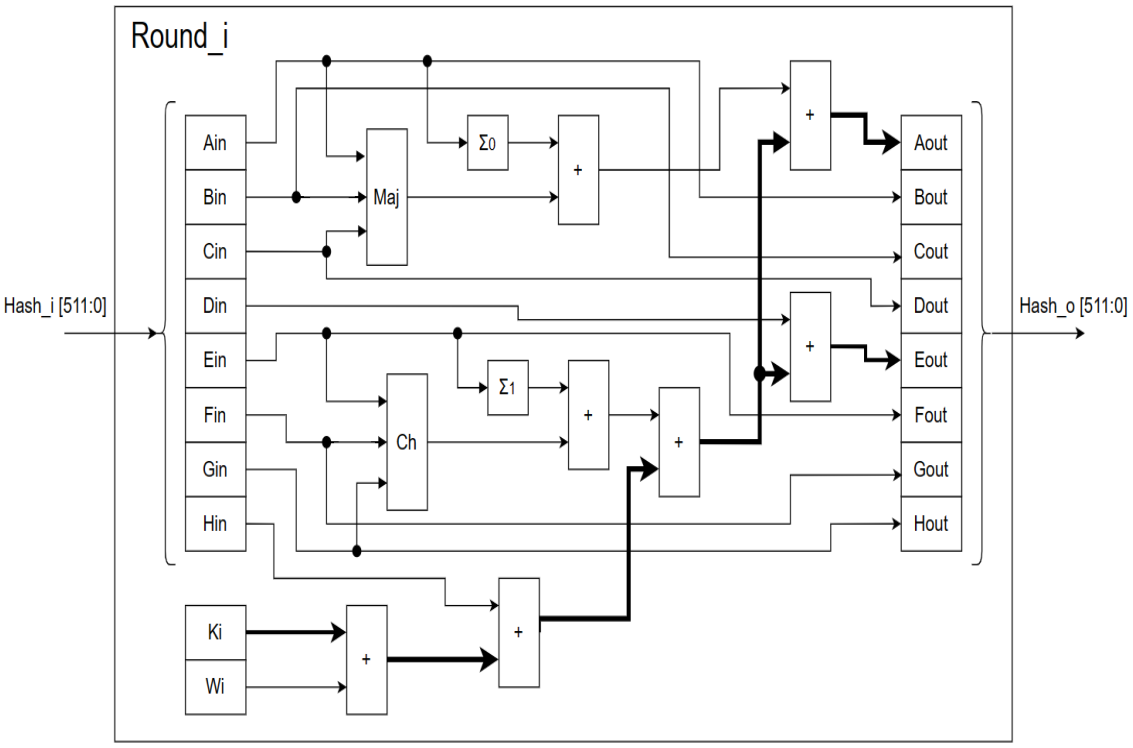


Figure 2.5 The block diagram of the i-th Round function

From the original architecture shown in Figure 2.5, it is designed such that each round takes only one cycle to compute. This implies that the computation of A in each round, which involves six 64 bits adders (excluding the calculation of $\Sigma_1(E)$, $\Sigma_0(A)$, Ch(E, F, G), and Maj(A, B, C)), must be completed within a single cycle. The total latency for the architecture is 2 * 80 = 160 which means the total latency of two for

doi:10.6342/NTU202301593

loops in SHA512-256d. For example, if the design can be executed in 250MHz with all slack of timing path is positive, and run two SHA512-256d in parallel, the hash rate of this design is 500M hash/s.

$$\text{Hash rate} = \text{frequency} * \frac{\text{number of data computed in parallel}}{process\ one\ hash\ value\ (clock\ cycle)} \quad (2.2)$$

$$= 250\text{M} * \frac{2 * 80 * 2}{80 * 2} = 500M\ hash/s$$

The original circuit can achieve a hash rate of 500M hashes per second when operating at a clock rate of 250M and running in two parallel pipelines. To further increase the hash rate, the circuit should maintain full pipelining and focus on shortening the critical path to enhance the clock rate. This will enable the circuit to operate at its maximum speed and achieve an even higher hash rate. We have to note that since the characteristic of compute-hard, it would be difficult to reduce the usage of LUTs.

doi:10.6342/NTU202301593

Chapter 3 Circuit Improvement of SHA512-256d Algorithm

3.1 Design methodologies on FPGA

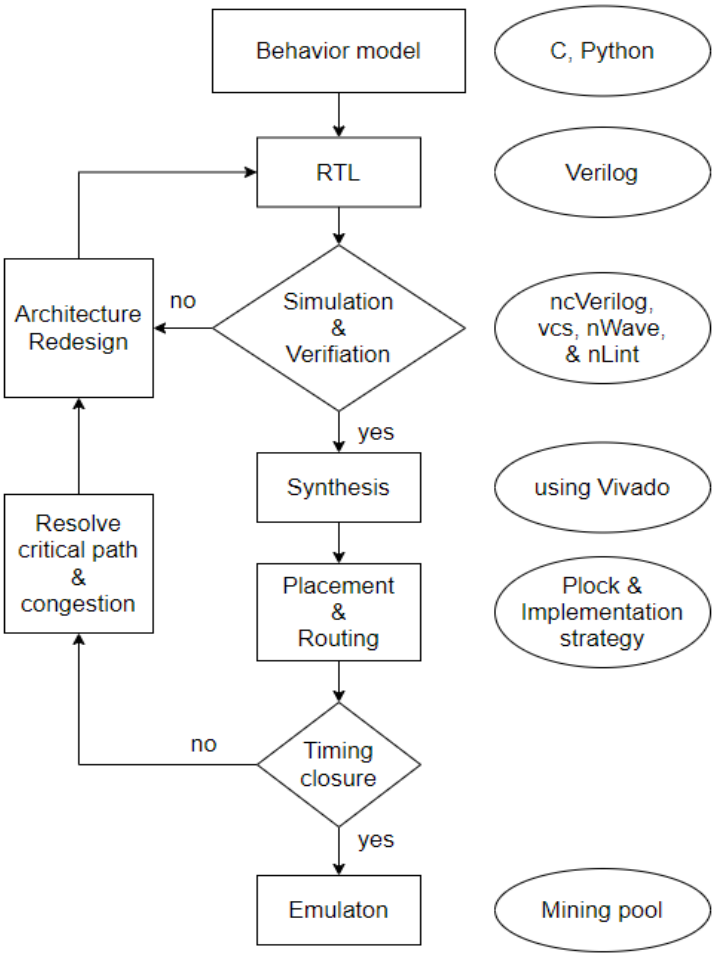
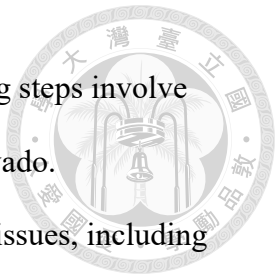


Figure 3.1 The design methodology

Figure 3.1 shows the design flow of the thesis on FPGA. The author employs a behavior model to swiftly design and analyze the resource requirements, such as LUTs and FFs. Then Verilog is employed as the RTL coding language for this project. Once

doi:10.6342/NTU202301593



the RTL design is verified through simulation and nLint, the remaining steps involve synthesis and implementation using the Xilinx-provided software, Vivado.

However, several factors can give rise to significant timing issues, including congestion caused by overutilization or a long critical path. If the timing fails to meet the requirement, the author must address the issue based on the implementation report and repeat the design flow. After achieving timing closure, a bitstream is generated and programmed onto the FPGA, allowing for emulation and the collection of actual performance data.

3.2 Design strategy and constraints

Based on the reports from Vivado and the discussions in subsection 2.3.2, it can be analyzed that a specific part of the original architecture, Transform_func, is facing serious timing problems. In order to solve this issue, the author decides to prioritize the redesign of the Transform_func module to relax the timing. After that, the author focuses on optimizing the Message_scheduler module to reduce the number of Flip-Flops. Finally, the improved architecture is integrated with the other original modules.

According to the information provided on the Xilinx official website, it is generally more challenging to achieve timing closure when the design exceeds 80% utilization of Look-Up Tables. Besides, the hardware architecture of the SHA512-256d algorithm utilizes more resources as it achieves higher throughput (hash rate). To ensure an adequate design margin, the author sets the upper bounds of approximately 60% and 55% LUT utilization for the hardware implementation of the entire design and all SHA512-256d modules, while the upper bound of FF utilization is approximately 70%

doi:10.6342/NTU202301593

and 75%.

3.3 Analysis of Round function

Based on the C code of the SHA512-256d algorithm, as mentioned in section 2.2.2, it is evident that the Round function iterates 2 * 80 = 160 times, involving numerous calculations with 64-bit inputs. Consequently, the Round functions consume a significant portion of the FPGA resources. This indicates that the hardware architecture of the Round functions has a profound impact on the overall design performance. Furthermore, as discussed in subsection 2.3.2, the critical path also lies within the Round function. Therefore, a thorough analysis of the C code for each round in the Transform_func is essential as a starting point.

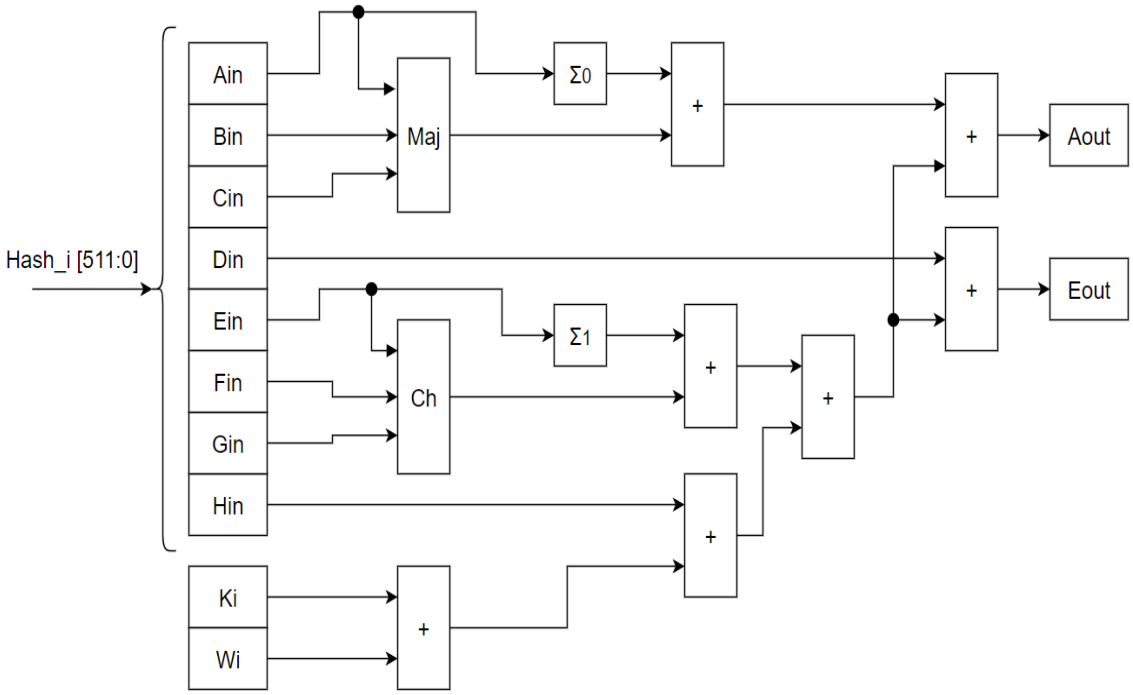


Figure 3.2 The simplification of the Round function involves only the data flow that needs to be computed.

doi:10.6342/NTU202301593

In the Round function, as depicted in Figure 2.5, it can be observed that the output values of B_{out}, C_{out}, D_{out}, F_{out}, G_{out}, and H_{out} directly depend on the input values A_{in}, B_{in}, C_{in}, E_{in}, F_{in}, and G_{in} respectively. Only the calculation of A_{out} and E_{out} needs to be performed separately. Hence, the complex calculation depicted in Figure 2.5 can be simplified to the form illustrated in Figure 3.2.

3.4 New architecture of Round function

3.4.1 Pipeline method in Round function

To reduce the critical path length, one intuitive approach is to implement pipelining. By appending a Flip-Flop in the data flow, as illustrated in Figure 3.3, the critical path can be shortened.

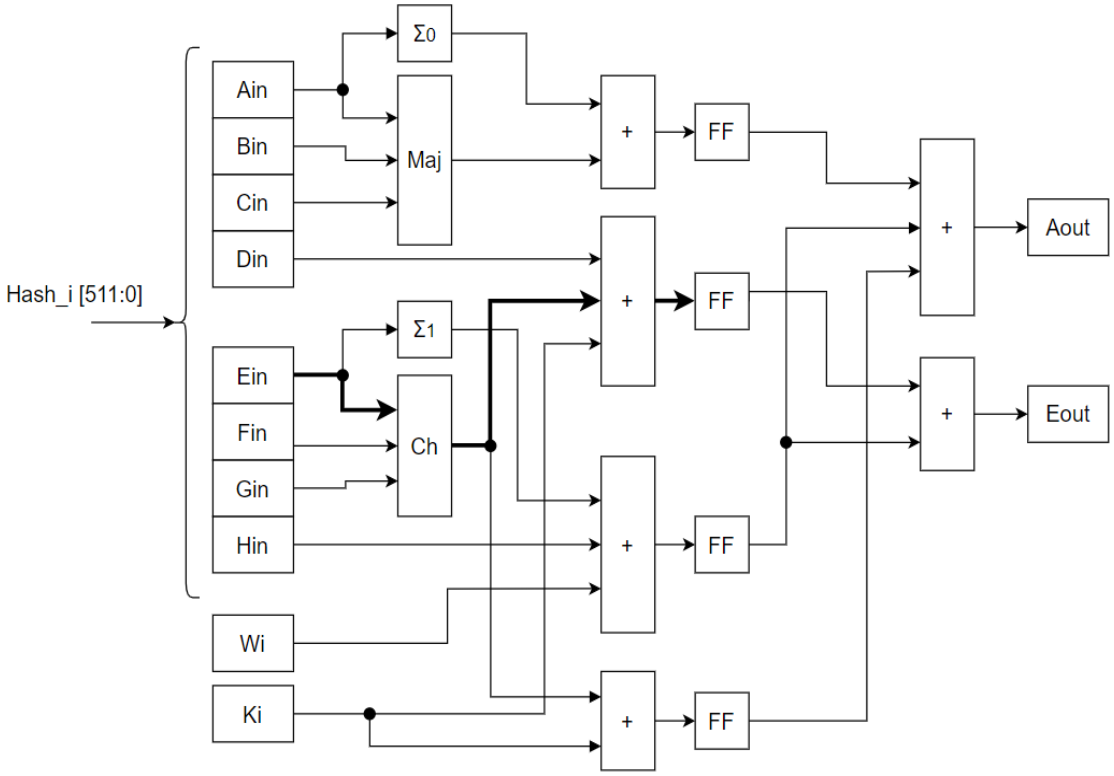
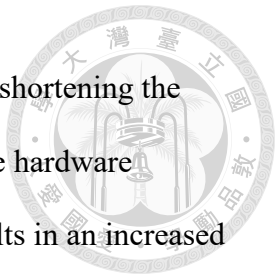


Figure 3.3 Two-stage pipelined of the simplified Round function

doi:10.6342/NTU202301593



Increasing the number of pipeline stages can indeed help in shortening the critical path. However, it is important to consider the limitations of the hardware resources available on the FPGA. The addition of pipeline stages results in an increased usage of Flip-Flops within the design. We can consider the pipelined design's FF utilization will be nearly the multiple of the original design.

During our experiment, we found that applying a three-stage pipeline caused congestion errors during the Vivado synthesis process, leading to implementation failure. As a result, the author had to explore alternative approaches to achieve a shorter critical path while staying within the constraints of the hardware resources.

3.4.2 Precompute in Transform_func of E variable

In the previous subsection, the author concluded that the two-stage pipeline is recommended for implementation. However, upon examining Figure 3.3, we can observe a specific path, indicated in bold, that warrants further analysis. As shown below:

$$E_{in} \rightarrow Ch(E_{in}, F_{in}, G_{in}) \rightarrow 3 \text{ input adder} \rightarrow FF \quad (3.1)$$

This path includes the Ch function and a three-input adder, which still represents a lengthy critical path that offers room for simplification.

Furthermore, the output of the Ch function is connected to multiple other elements, increasing the complexity of routing. This leads to a higher fanout and potential timing violations due to congestion in a high-frequency design. As a result, the author sought to identify a critical path that involves only a two-input adder and a minimal number of logic elements, while minimizing fanout as much as possible.

doi:10.6342/NTU202301593

$$
\begin{cases}
T_1 = H_{i-1} + \Sigma_1(E_{i-1}) + Ch(E_{i-1}, F_{i-1}, G_{i-1}) + K_i + W_i \\
T_2 = \Sigma_0(A_{i-1}) + Maj(A_{i-1}, B_{i-1}, C_{i-1}) \\
H_i = G_{i-1} \\
G_i = F_{i-1} \\
F_i = E_{i-1} \\
E_i = D_{i-1} + T_1 \\
D_i = C_{i-1} \\
C_i = B_{i-1} \\
B_i = A_{i-1} \\
A_i = T_1 + T_2
\end{cases} \tag{3.2}
$$

Based on Equation 3.2, the computation of A and E involves the variable T1, which implies that A_i can be calculated from E_i. Furthermore, the remaining calculations of A and E have limited relations. As a result, it is possible to divide the Round function into two distinct groups: Round_A, consisting of variables A, B, C, and D, and Round_E, comprising variables E, F, G, and H.

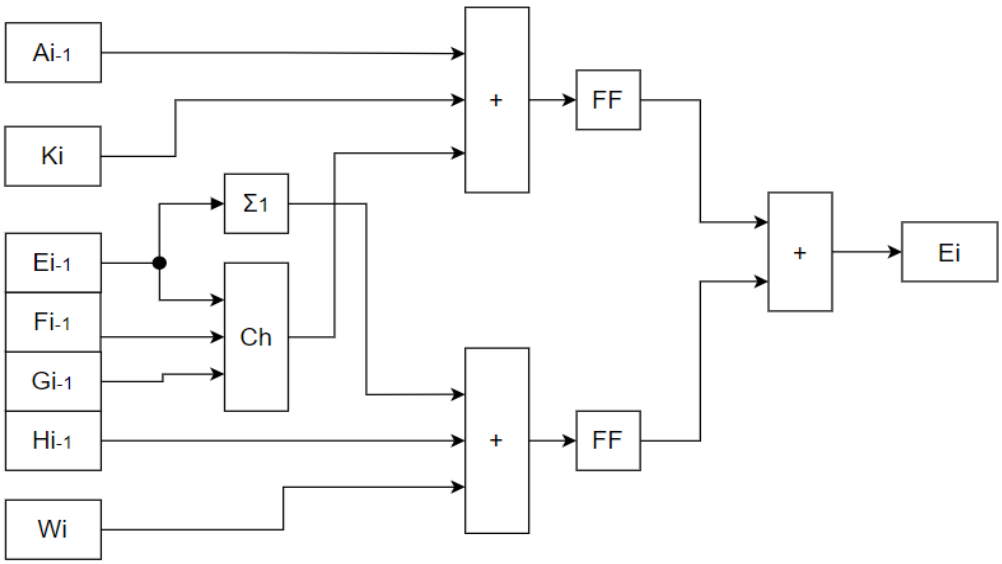


Figure 3.4 The block diagram of the data flow of Ei in the i-th round

doi:10.6342/NTU202301593

The computation of the i-th round of E_i is taken into discussion first:

$$E_i = D_{i-1} + T_1 = D_{i-1} + H_{i-1} + \Sigma_1(E_{i-1}) + Ch(E_{i-1}, F_{i-1}, G_{i-1}) + K_i + W_i \quad (3.3)$$

In Equation 3.3, K_i is constant, and the value of W_i has been calculated in Message_scheduler. Therefore, from Equation 3.2, we have the following equations:

$$\begin{cases} D_{i-1} = C_{i-2} = B_{i-3} = A_{i-4} \\ H_{i-1} = G_{i-2} \end{cases} \quad (3.4)$$

$$\begin{aligned} E_i &= D_{i-1} + H_{i-1} + \Sigma_1(E_{i-1}) + Ch(E_{i-1}, F_{i-1}, G_{i-1}) + K_i + W_i \\ &= A_{i-4} + G_{i-2} + \Sigma_1(E_{i-1}) + Ch(E_{i-1}, F_{i-1}, G_{i-1}) + K_i + W_i \\ &= A_{i-4} + W_i + G_{i-2} + K_i + \Sigma_1(E_{i-1}) + Ch(E_{i-1}, F_{i-1}, G_{i-1}) \end{aligned} \quad (3.5)$$

Since A_{i-4}, W_i, G_{i-2}, K_i can be obtained from the previous round, we can compute the addition of these four 64 bits variables referred to as $Precomputed\_E_{i-1}$ in the (i-1) -th round.

$$\begin{aligned} E_i &= A_{i-4} + W_i + G_{i-2} + K_i + \Sigma_1(E_{i-1}) + Ch(E_{i-1}, F_{i-1}, G_{i-1}) \\ &= Precomputed\_E_{i-1} + \Sigma_1(E_{i-1}) + Ch(E_{i-1}, F_{i-1}, G_{i-1}) \end{aligned} \quad (3.6)$$

$$Precomputed\_E_{i-1} = A_{i-4} + W_i + G_{i-2} + K_i \quad (3.7)$$

Equations 3.6 and 3.7 present the modified data flow for E_I, where the author has successfully reduced the path length in each equation. As a result, now a two-stage pipeline can be applied to these equations, as illustrated in Figure 3.5 and Figure 3.6.

doi:10.6342/NTU202301593

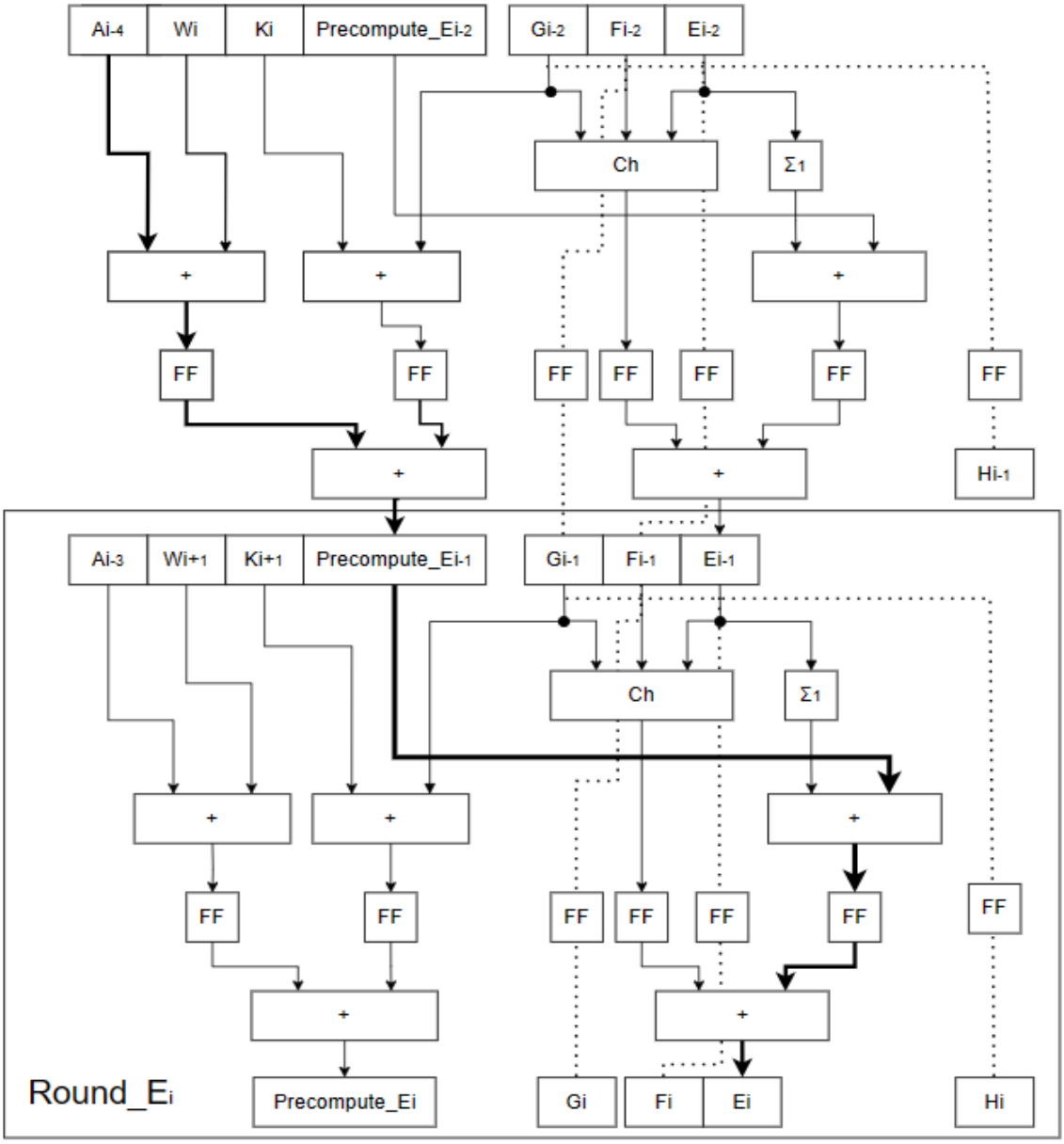


Figure 3.5 The data flow of the i-th Round_Ei

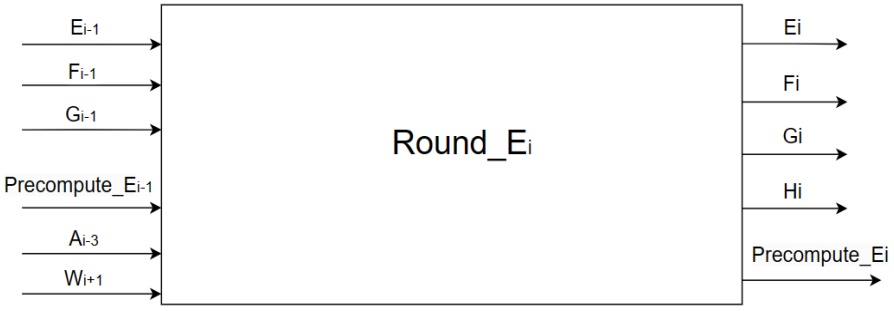


Figure 3.6 The In/Out ports of the i-th Round_Ei

doi:10.6342/NTU202301593

In the block diagrams above, the author makes the critical path of Round_E a logical function and a two inputs adder, also the fanout in this circuit becomes lower than before. The LUT and FF of Round_E are listed in Table 3.1, which is implemented at the frequency of 700M.

Resource / Module	439680 LUT		879360 FF	
Round_E	392	0.089%	640	0.072%

Table 3.1 The LUT and FF utilization of Round_E

3.4.3 Precompute in Transform_func of A variable

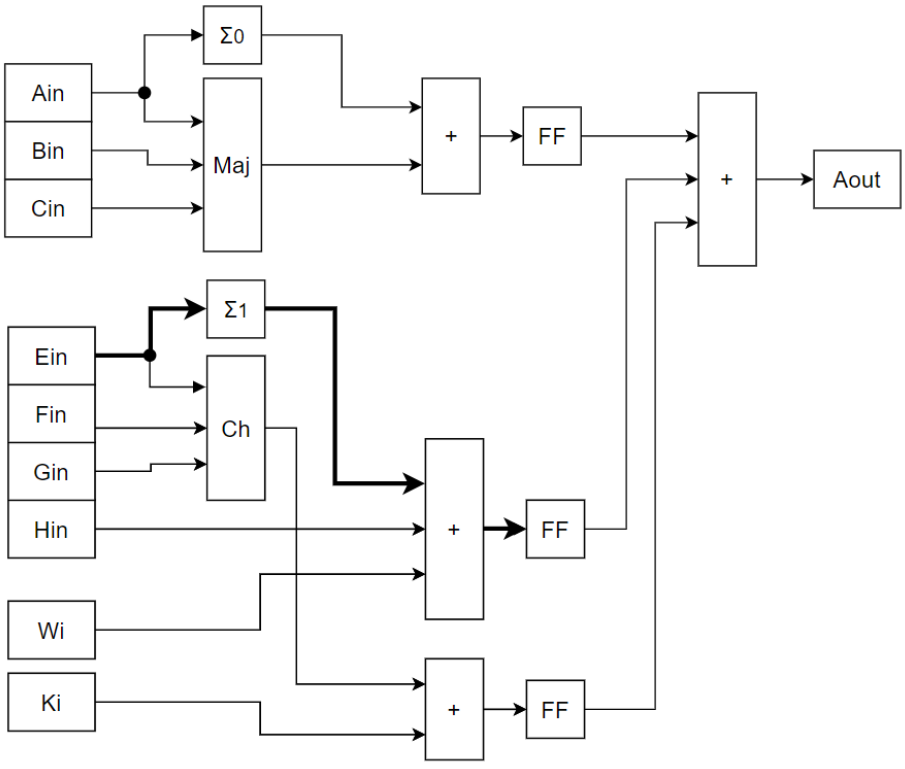


Figure 3.7 The block diagram of the data flow of Ai in the i-th round

doi:10.6342/NTU202301593

Similar to variable E in the Round function, variable A also exhibits the same issue. Figure 3.3 highlights a specific path in bold that is shown below:

$$E_{in} \rightarrow \Sigma_1(E_{in}) \rightarrow 3 \text{ input adder} \rightarrow FF \quad (3.1)$$

This path consists of the Σ_1 function and a three-input adder, indicating the potential for simplification.

As mentioned in subsection 3.4.2, variable A can be computed based on variable E within the Round_A module. By referring to Equation 3.2, we can derive the following equations for the variable A:

$$\begin{cases} T_1 = H_{i-1} + \Sigma_1(E_{i-1}) + Ch(E_{i-1}, F_{i-1}, G_{i-1}) + K_i + W_i \\ T_2 = \Sigma_0(A_{i-1}) + Maj(A_{i-1}, B_{i-1}, C_{i-1}) \\ E_i = D_{i-1} + T_1 \\ D_i = C_{i-2} \\ A_i = T_1 + T_2 \\ \quad = E_i - D_{i-1} + T_2 \\ \quad = E_i - D_{i-1} + \Sigma_0(A_{i-1}) + Maj(A_{i-1}, B_{i-1}, C_{i-1}) \\ \quad = E_i - C_{i-2} + \Sigma_0(A_{i-1}) + Maj(A_{i-1}, B_{i-1}, C_{i-1}) \quad (3.8) \\ \quad = Precomputed\_A_{i-1} + \Sigma_0(A_{i-1}) + Maj(A_{i-1}, B_{i-1}, C_{i-1}) \quad (3.9) \end{cases}$$

$$Precomputed\_A_{i-1} = E_i - C_{i-2} = E_i + \sim C_{i-2} + 1 \quad (3.10)$$

In our design, to satisfy Equation 3.8, the i-th Round_E module is executed 4 cycles earlier than the i-th Round_A module in the two-stage pipeline. Causing the variables E_I and C_{i-2} in Equation 3.8 can be obtained from the previous rounds that are completed. We can compute the addition of this variable referred to as $Precomputed\_A_{i-1}$ in the (i-1) -th round, which is shown in Equation 3.10. Figure 3.8 and Figure 3.9 show the new two-pipeline block diagram.

doi:10.6342/NTU202301593

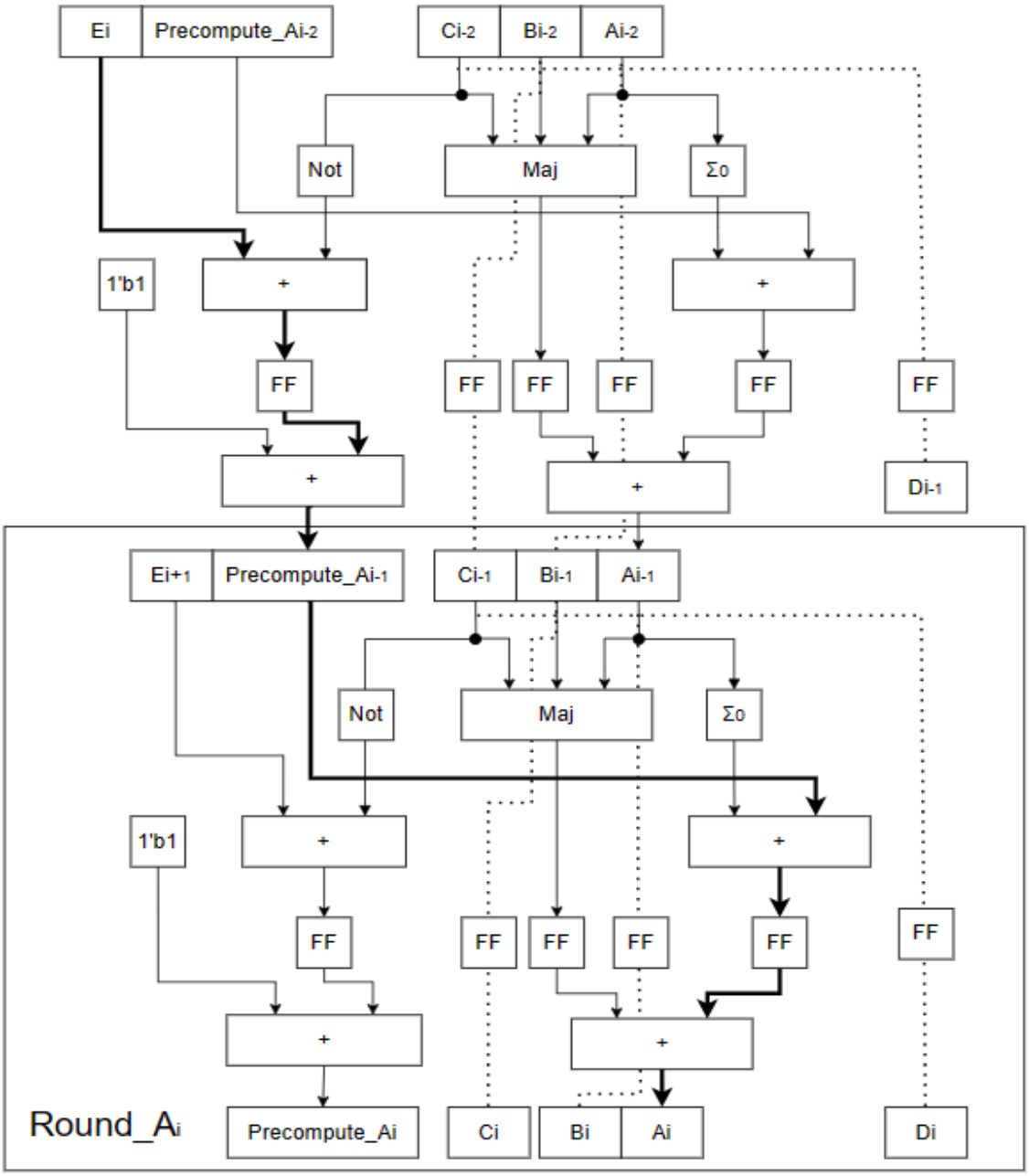


Figure 3.8 The data flow of the i-th Round_Ai

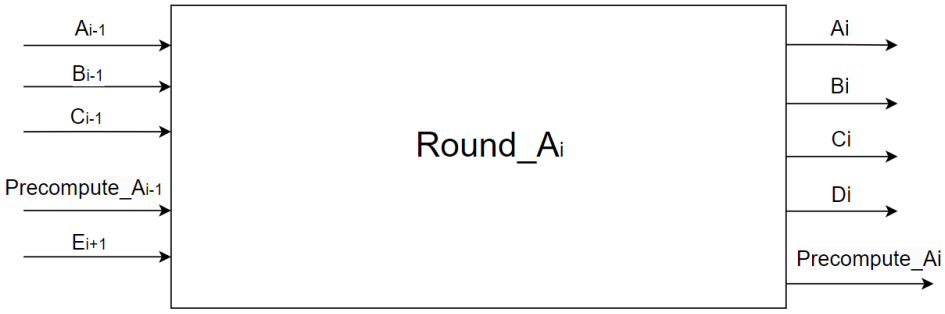


Figure 3.9 The In/Out ports of the i-th Round_Ai

doi:10.6342/NTU202301593

In the block diagrams above, the author makes the critical path of Round_A a logical function and a two inputs adder, while the fanout in this circuit becomes lower than before. The LUT and FF of Round_E are listed in Table 3.1, which is implemented at the frequency of 700M.

Resource / Module	439680 LUT		879360 FF	
Round_A	200	0.045%	576	0.065%

Table 3.2 The LUT and FF utilization of Round_A

3.5 Building architecture for Message_scheduler

3.5.1 Two-stage pipelined of Message_scheduler

Referring to the C code provided in subsection 2.2.2, the equation for the Words in the Message_scheduler module can be represented as follows:

$$\begin{cases} Words[i] = Message[i], & 0 <= i < 16 \\ Words[i] = \sigma_1(Words[i-2]) + Words[i-7] + \sigma_0(Words[i-15]) + Words[i-16], & 16 <= i < 80 \end{cases} \quad (3.11)$$

Given that the preceding 16 64 bits values of the Words are derived from the assignment of the 1024 bits input Message, the author's focus will now shift to the labeled elements ranging from 16 to 79. In Equation 3.11, implemented in the original design, a path utilizing a four-input 64 bits adder is observed, which is considerably longer compared to the new architecture of the Round_A and Round_E modules discussed in section 3.4. To address the timing issue, a two-stage pipelined method is

doi:10.6342/NTU202301593

applied to this module named Word_compute. Thus, the computational logic in the paths within this module is transformed into two input adders, which are shorter than the paths in the Transform_func module.

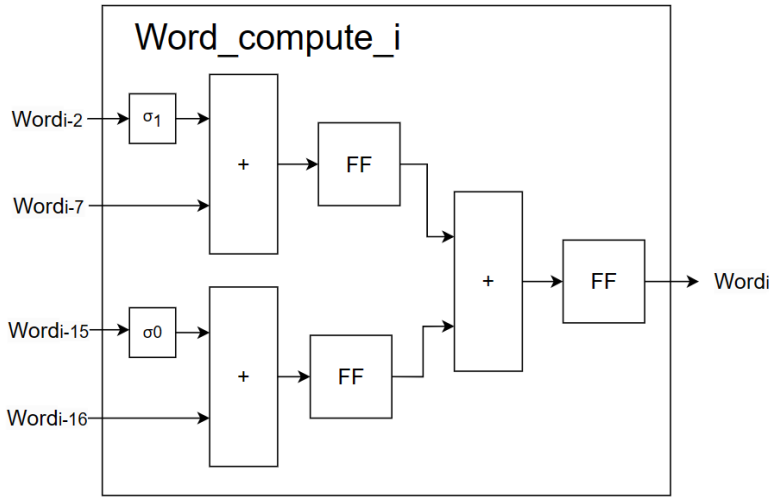


Figure 3.10 The i-th Word_compute module in two-stage pipelined

In the two-stage pipelined design, the Word_shift module is utilized to distribute distinct inputs to the 64 Word_compute modules. To enhance clarity in the assignment process, the Word_shift module can be subdivided into 64 Word_shift_element modules, as depicted in the accompanying illustrations (Figure 3.11 and Figure 3.12). By dividing the Word_shift module into individual Word_shift_element modules, we achieve a more organized and manageable allocation of specific inputs to related Word_compute modules, making it easier to track and manage the input assignments.

Moreover, the utilization of Word_shift_element modules enables the establishment of a pipelined structure, which enables the storage and transmission of different words in each cycle.

doi:10.6342/NTU202301593

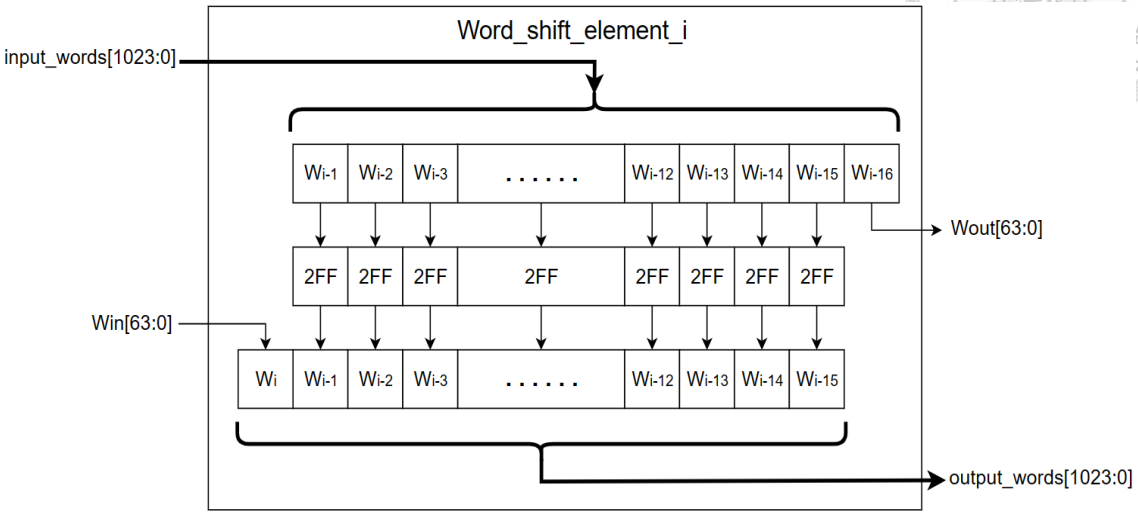


Figure 3.11 The architecture of the i-th two-staged pipelined Word_shift_element

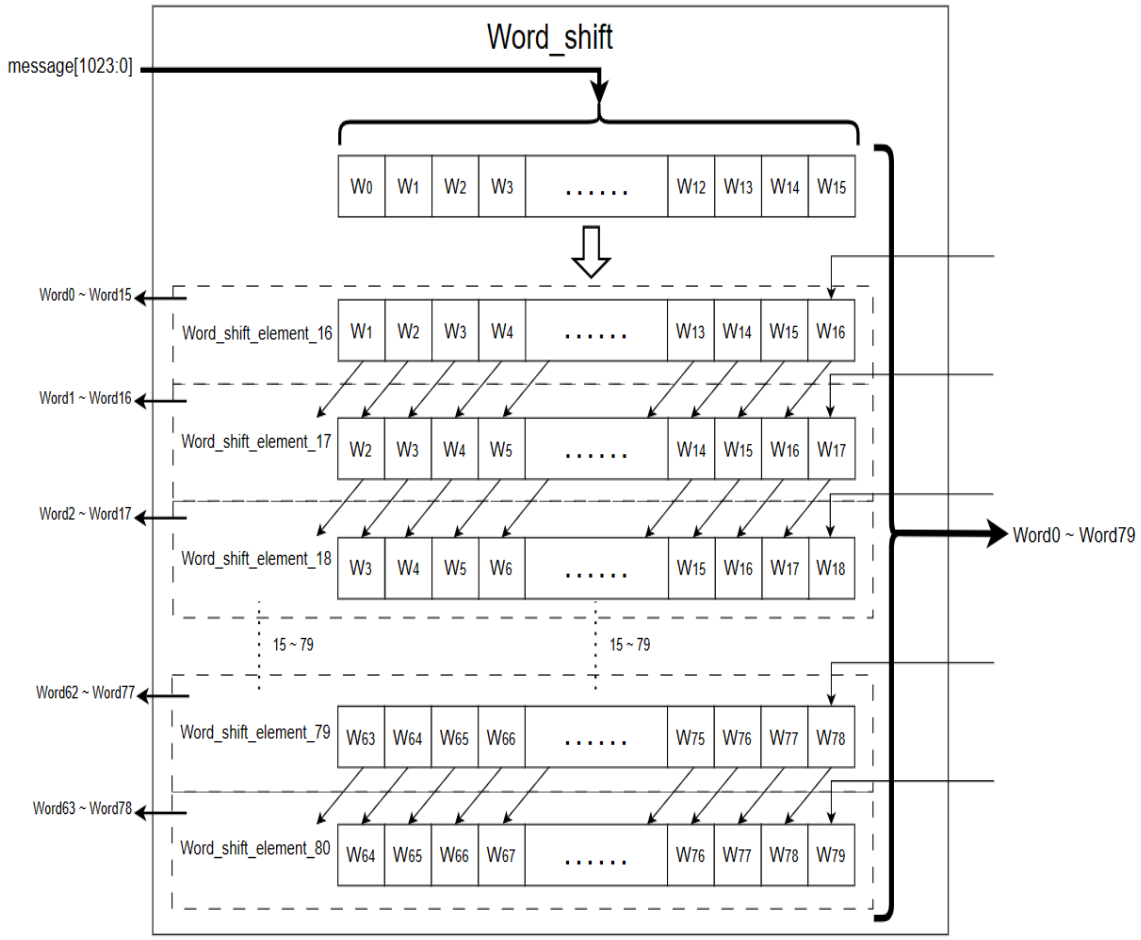


Figure 3.12 The block diagram of the two-staged pipelined Word_shift module

doi:10.6342/NTU202301593

3.5.2 New architecture for Message_scheduler

Furthermore, considering Words[i+1], the equations can be formulated as follows:

$$\begin{aligned} \text{Words}[i+1] &= \sigma_1(\text{Words}[(i+1)-2]) + \text{Words}[(i+1)-7] + \sigma_0(\text{Words}[(i+1)-15]) + \text{Words}[(i+1)-16] \\ &= \sigma_1(\text{Words}[i-1]) + \text{Words}[i-6] + \sigma_0(\text{Words}[i-14]) + \text{Words}[i-15] \end{aligned} \quad (3.12)$$

$$\begin{cases} \text{Words}[i] = \sigma_1(\text{Words}[i-2]) + \text{Words}[i-7] + \sigma_0(\text{Words}[i-15]) + \text{Words}[i-16] \\ \text{Words}[i+1] = \sigma_1(\text{Words}[i-1]) + \text{Words}[i-6] + \sigma_0(\text{Words}[i-14]) + \text{Words}[i-15] \end{cases} \quad (3.13)$$

As illustrated by equation 3.13, the computation of Words[i] and Words[i+1] can be performed concurrently by utilizing the Words indexed i-1 to i-16. This optimization results in a reduction in the number of modules within the Word_compute component from 64 to 32, thereby decreasing the number of registers used in these modules.

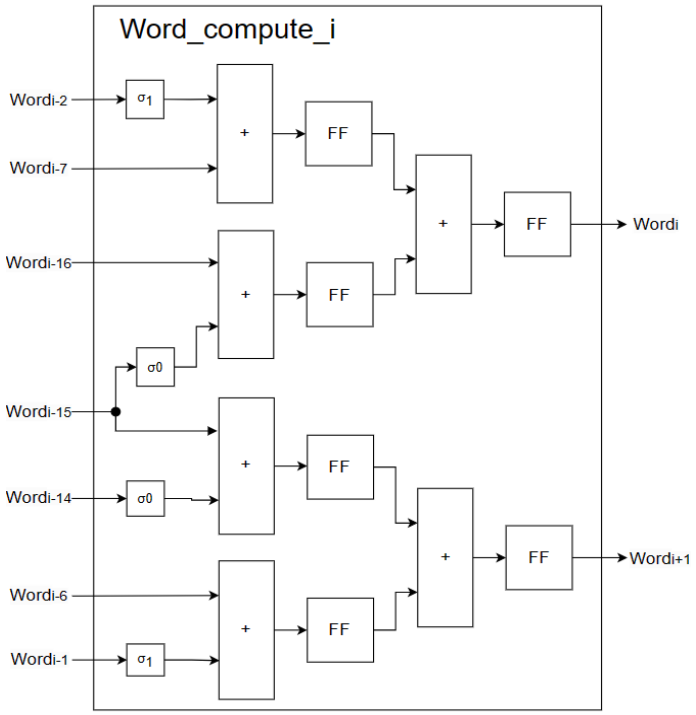


Figure 3.13 The i-th Word_compute module that calculates two words simultaneously

doi:10.6342/NTU202301593

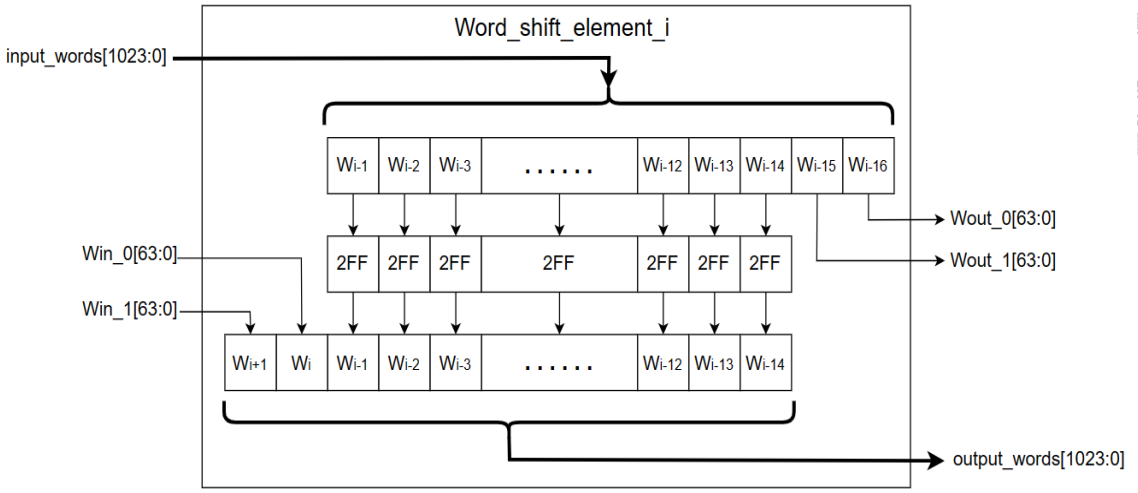


Figure 3.14 New architecture of the i-th Word_shift_element module

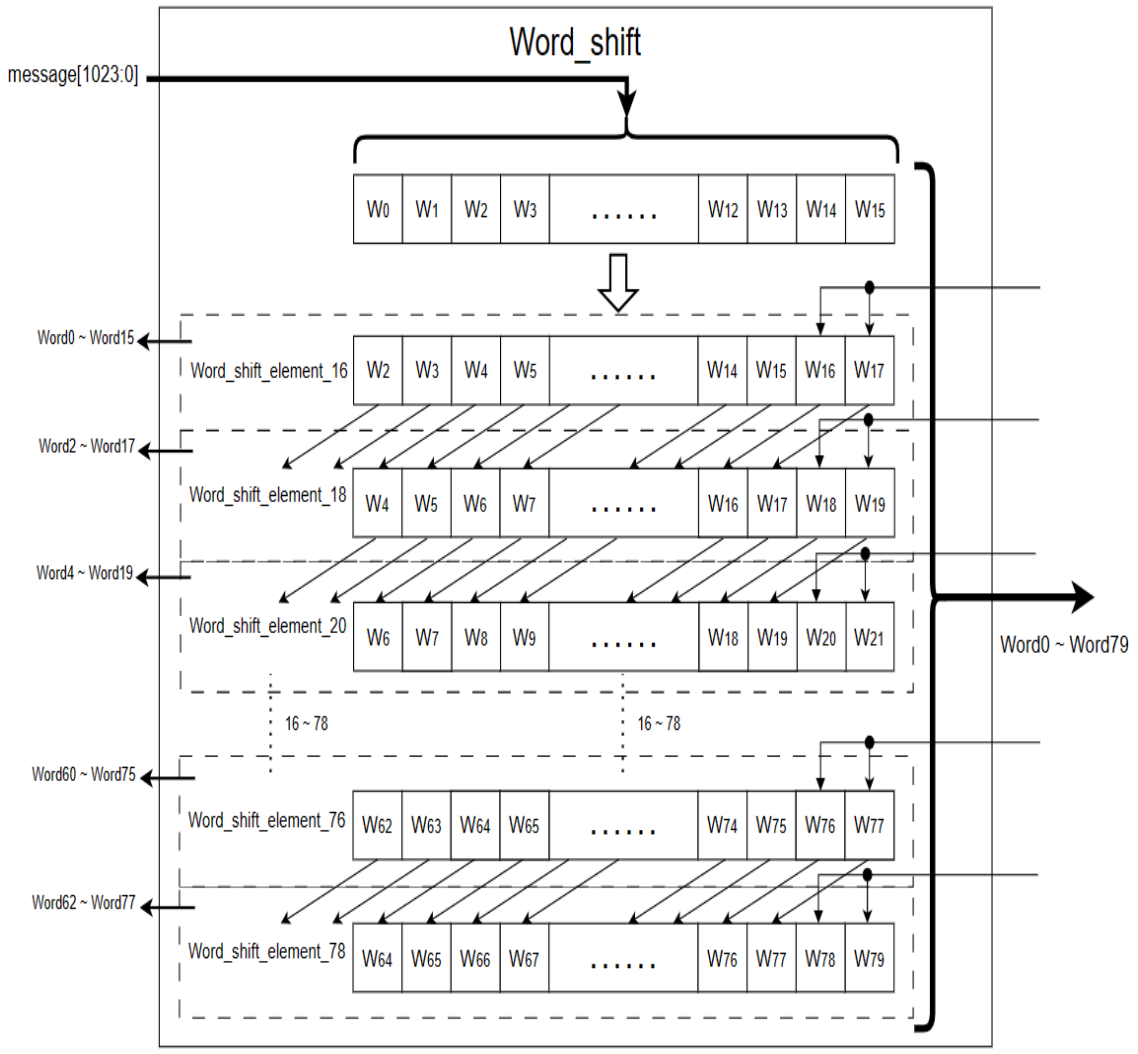


Figure 3.15 New block diagram of the Word_shift_element module

doi:10.6342/NTU202301593

The architecture of both the Word_shift_element and Word_shift modules undergo changes in this structure, as illustrated in Figure 3.14 and Figure 3.15. This modification reduces the number of steps required to construct the complete Words. However, due to the requirements of the Round_E module described in subsection 3.4.2, which necessitates the input word at a specific time, it becomes necessary to store these Words until they are needed.

The overall architecture of the Message_scheduler module, which encompasses these components, is presented in Figure 3.16.

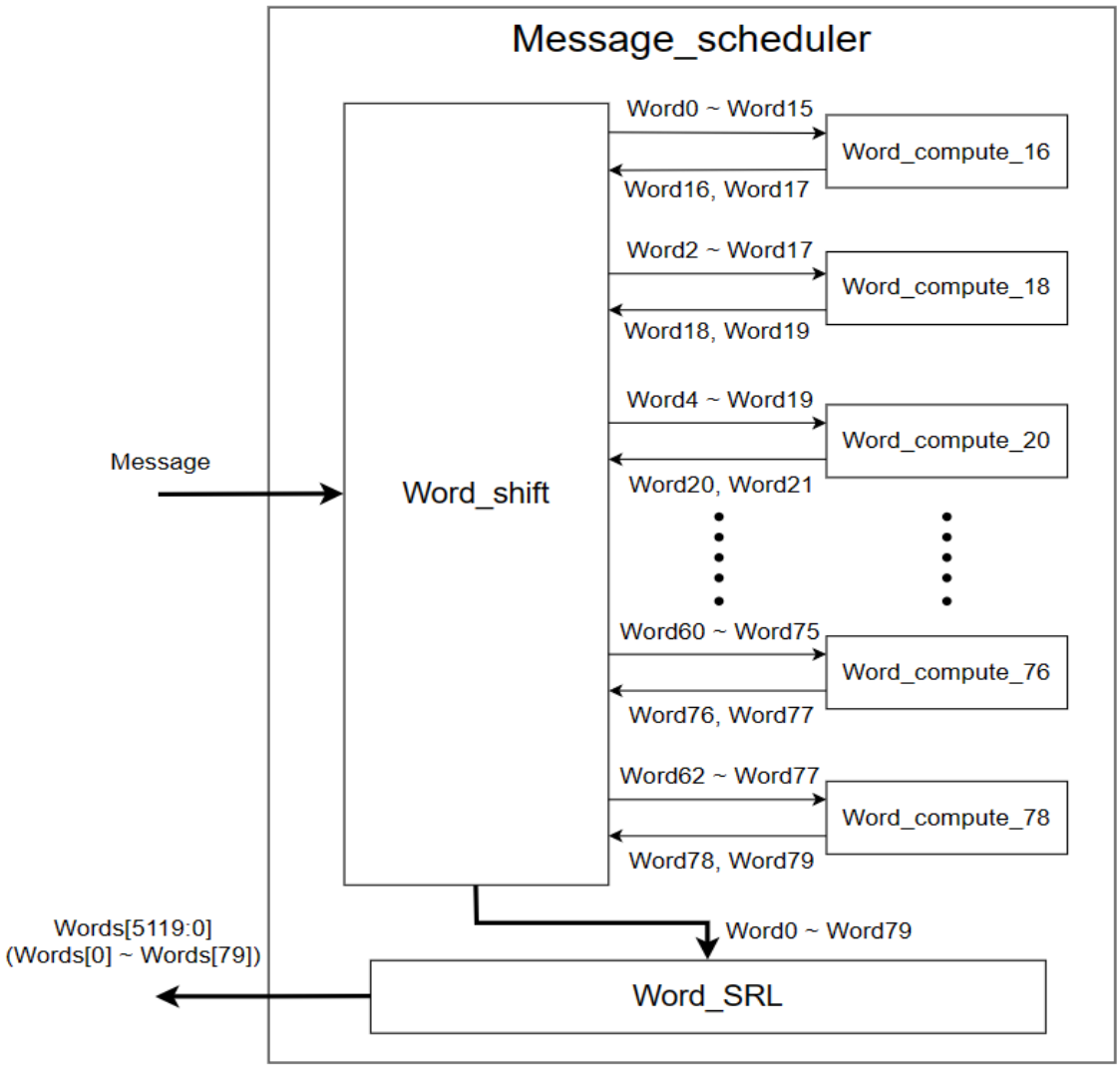
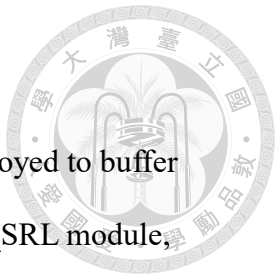


Figure 3.16 Overview of the Message_scheduler module

doi:10.6342/NTU202301593



To address this requirement, the Word_SRL module is employed to buffer these Words and output them only when they are required. The Word_SRL module, which is made of all registers, serves as a temporary storage mechanism for the Words, and this design ensures that the Words are stored and made available for processing as per the needs of the Round_E module in every cycle. The number of registers required to delay each of the 64 Word_i inputs in the Word_SRL module is listed in Table A.2 in Appendix A.

3.5.3 Discussion of the resource distribution in the Message_scheduler module

In addition to optimizing critical paths and computational methods, there is also room for improvement in the data storage method. As described in subsection 3.5.2, the timing at which we finish computing the entire Words affects the number of registers required in the Word_SRL module for delay purposes. Regardless of the pipeline initially, we can compare the registers usage of the first 23 rounds of SHA512-256's Message_scheduler using Table 3.3 and Table 3.4.

Noting that the input header of the first SHA512-256 which is 1024 bits, as mentioned in subsection 2.2.1, are all constant with the exception of the 32 bits nonce which is located at message[639:608], referred to as W9 in the tables below. The constant registers, highlighted in gray, will be bypassed by the Vivado synthesis tools. Therefore, by increasing the frequency of appearance of these constants, we can reduce the number of registers required.

doi:10.6342/NTU202301593

Round	Word registers															
1	W0	W1	W2	W3	W4	W5	W6	W7	W8	W9	W10	W11	W12	W13	W14	W15
2	W2	W3	W4	W5	W6	W7	W8	W9	W10	W11	W12	W13	W14	W15	W16	W17
3	W4	W5	W6	W7	W8	W9	W10	W11	W12	W13	W14	W15	W16	W17	W18	W19
4	W6	W7	W8	W9	W10	W11	W12	W13	W14	W15	W16	W17	W18	W19	W20	W21
5	W8	W9	W10	W11	W12	W13	W14	W15	W16	W17	W18	W19	W20	W21	W22	W23
6	W10	W11	W12	W13	W14	W15	W16	W17	W18	W19	W20	W21	W22	W23	W24	W25
7	W12	W13	W14	W15	W16	W17	W18	W19	W20	W21	W22	W23	W24	W25	W26	W27
8	W14	W15	W16	W17	W18	W19	W20	W21	W22	W23	W24	W25	W26	W27	W28	W29
9	W16	W17	W18	W19	W20	W21	W22	W23	W24	W25	W26	W27	W28	W29	W30	W31
10	W18	W19	W20	W21	W22	W23	W24	W25	W26	W27	W28	W29	W30	W31	W32	W33
11	W20	W21	W22	W23	W24	W25	W26	W27	W28	W29	W30	W31	W32	W33	W34	W35
12	W22	W23	W24	W25	W26	W27	W28	W29	W30	W31	W32	W33	W34	W35	W36	W37
13	W24	W25	W26	W27	W28	W29	W30	W31	W32	W33	W34	W35	W36	W37	W38	W39
14	W26	W27	W28	W29	W30	W31	W32	W33	W34	W35	W36	W37	W38	W39	W40	W41
15	W28	W29	W30	W31	W32	W33	W34	W35	W36	W37	W38	W39	W40	W41	W42	W43
16	W30	W31	W32	W33	W34	W35	W36	W37	W38	W39	W40	W41	W42	W43	W44	W45
17	W32	W33	W34	W35	W36	W37	W38	W39	W40	W41	W42	W43	W44	W45	W46	W47
18	W34	W35	W36	W37	W38	W39	W40	W41	W42	W43	W44	W45	W46	W47	W48	W49
19	W36	W37	W38	W39	W40	W41	W42	W43	W44	W45	W46	W47	W48	W49	W50	W51
20	W38	W39	W40	W41	W42	W43	W44	W45	W46	W47	W48	W49	W50	W51	W52	W53
21	W40	W41	W42	W43	W44	W45	W46	W47	W48	W49	W50	W51	W52	W53	W54	W55
22	W42	W43	W44	W45	W46	W47	W48	W49	W50	W51	W52	W53	W54	W55	W56	W57
23	W44	W45	W46	W47	W48	W49	W50	W51	W52	W53	W54	W55	W56	W57	W58	W59

Table 3.3 Normal structure of Word_SRL regardless of two-staged pipeline

Round	Word registers															
1	W0	W1	W2	W3	W4	W5	W6	W7	W8	W9	W10	W11	W12	W13	W14	W15
2	W0	W1	W2	W3	W4	W5	W6	W7	W8	W9	W10	W11	W12	W13	W14	W15
3	W0	W1	W2	W3	W4	W5	W6	W7	W8	W9	W10	W11	W12	W13	W14	W15
4	W0	W1	W2	W3	W4	W5	W6	W7	W8	W9	W10	W11	W12	W13	W14	W15
5	W0	W1	W2	W3	W4	W5	W6	W7	W8	W9	W10	W11	W12	W13	W14	W15
6	W0	W1	W2	W3	W4	W5	W6	W7	W8	W9	W10	W11	W12	W13	W14	W15
7	W0	W1	W2	W3	W4	W5	W6	W7	W8	W9	W10	W11	W12	W13	W14	W15
8	W0	W1	W2	W3	W4	W5	W6	W7	W8	W9	W10	W11	W12	W13	W14	W15
9	W0	W1	W2	W3	W4	W5	W6	W7	W8	W9	W10	W11	W12	W13	W14	W15
10	W0	W1	W2	W3	W4	W5	W6	W7	W8	W9	W10	W11	W12	W13	W14	W15
11	W0	W1	W2	W3	W4	W5	W6	W7	W8	W9	W10	W11	W12	W13	W14	W15
12	W0	W1	W2	W3	W4	W5	W6	W7	W8	W9	W10	W11	W12	W13	W14	W15
13	W0	W1	W2	W3	W4	W5	W6	W7	W8	W9	W10	W11	W12	W13	W14	W15
14	W0	W1	W2	W3	W4	W5	W6	W7	W8	W9	W10	W11	W12	W13	W14	W15
15	W0	W1	W2	W3	W4	W5	W6	W7	W8	W9	W10	W11	W12	W13	W14	W15
16	W2	W3	W4	W5	W6	W7	W8	W9	W10	W11	W12	W13	W14	W15	W16	W17
17	W4	W5	W6	W7	W8	W9	W10	W11	W12	W13	W14	W15	W16	W17	W18	W19
18	W6	W7	W8	W9	W10	W11	W12	W13	W14	W15	W16	W17	W18	W19	W20	W21
19	W8	W9	W10	W11	W12	W13	W14	W15	W16	W17	W18	W19	W20	W21	W22	W23
20	W10	W11	W12	W13	W14	W15	W16	W17	W18	W19	W20	W21	W22	W23	W24	W25
21	W12	W13	W14	W15	W16	W17	W18	W19	W20	W21	W22	W23	W24	W25	W26	W27
22	W14	W15	W16	W17	W18	W19	W20	W21	W22	W23	W24	W25	W26	W27	W28	W29
23	W16	W17	W18	W19	W20	W21	W22	W23	W24	W25	W26	W27	W28	W29	W30	W31

Table 3.4 Expansion structure of Word_SRL regardless of two-staged pipeline

doi:10.6342/NTU202301593

By carefully examining the tables, we can analyze the expansion registers in the completion of the computation of the entire Words, and the corresponding registers needed in the Word_SRL module for delay as shown in Table A.2 in Appendix A. This evaluation allows us to make informed decisions regarding the optimal balance between computational efficiency and the utilization of storage resources within the system. Table 3.5 shows the comparison of registers utilization of FF and LUT as memory in different Message_scheduler. The LUT as memory is used in Word_SRL as shift right registers.

Resource / Module	879360 FF		205440 LUT as memory		Addition of FF and LUT as memory
Normal two-stage pipelined Message_scheduler	64423	7.33%	7728	3.76%	72151
Expansion two-stage pipelined Message_scheduler	64414	7.33%	3456	1.68%	67870

Table 3.5 The comparison of FF utilization in different Message_scheduler

In addition, however, as shown in Figure 3.14, although Word_shift_element is used for only storing data, the module encounters challenges when utilizing FPGA IP BRAM and URAM for data storage. Table 3.6 reveals that both BRAM and URAM have a width of 72 bits. However, the storage input data width in the Word_shift_element module amounts to 14 * 64 = 896 bits. This necessitates the use of 13 BRAMs to accommodate the module's throughput. Furthermore, since the design

doi:10.6342/NTU202301593

incorporates 4 SHA512-256 modules, the total requirement for BRAMs would be a maximum of 4 * 32 * 13 = 1664, surpassing the available BRAM blocks on the FPGA. Moreover, this high demand for BRAMs would lead to significant dispersion in routing, making it impractical in high frequency. Besides, the Distributed RAM will consume LUTs, due to the congestion caused by the LUY usage, therefore, it is deemed more suitable to utilize FFs in these modules.

RAM IP	BRAM	URAM	Distributed RAM
Features	Width: 72 bits Depth: 512	Width: 72 bits Depth: 4096	Consists of LUTs
Blocks	672	320	flexible

Table 3. 6 The features of RAM on FPGA

3.6 Block building of SHA512-256d

In this section, the author will present the architecture of the complete SHA512-256d module. The final step in constructing the entire SHA512-256d module involves connecting the Round_A, Round_E, Message_scheduler, Pre_round, Final_add, SHA512-256, Nonce_gen, and Nonce_compare modules. It is important to note that Round_A, Round_E, Message_scheduler, and Final_add are submodules of the SHA512-256 module, and they have been designed with full pipelining.

The block diagram of SHA512-256 can be depicted as follows:

doi:10.6342/NTU202301593

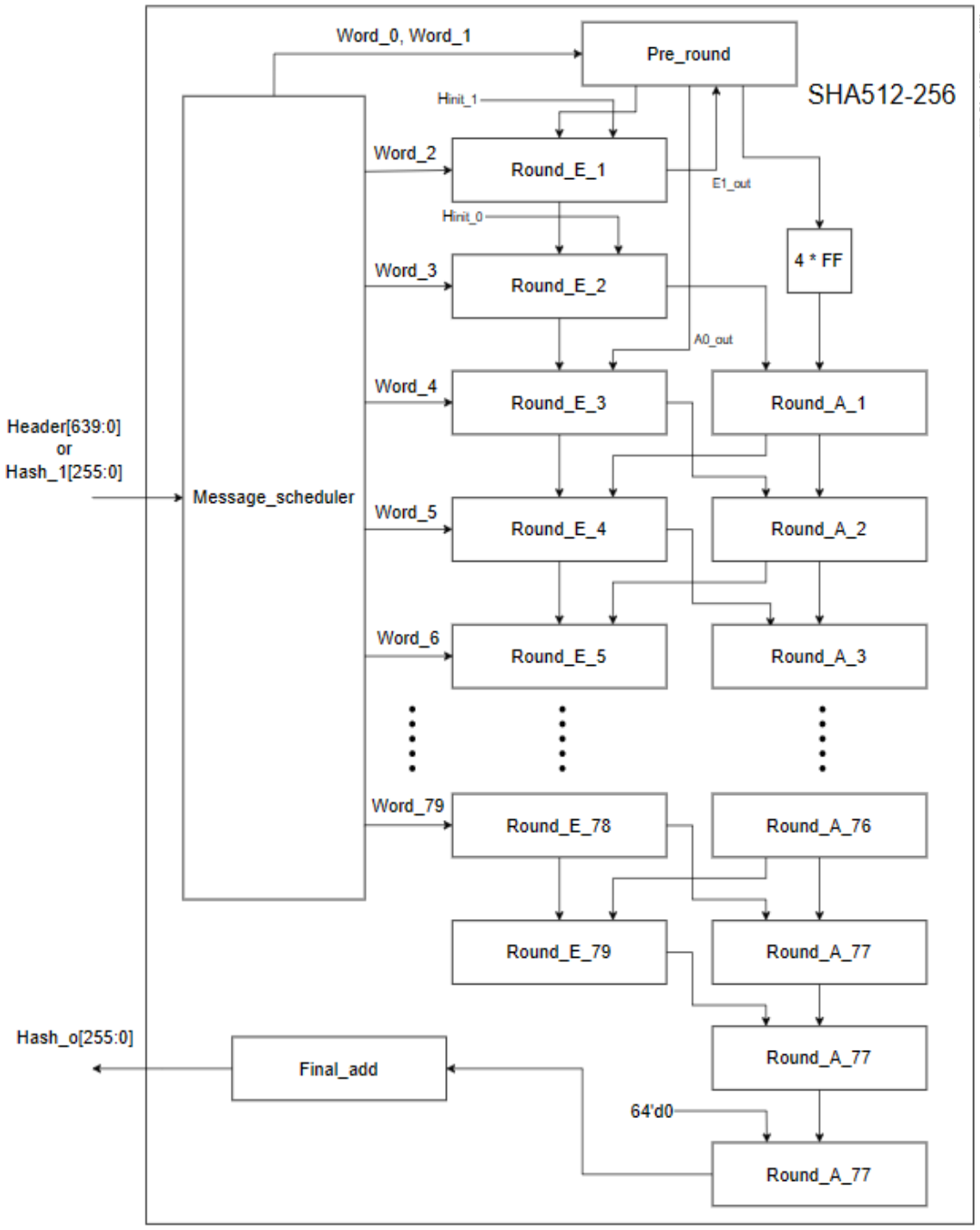


Figure 3.17 The architecture of new SHA512-256 design

According to Equation 3.6 in subsection 3.4.2, the output Words from the Message_scheduler module are distributed as inputs to each Round_E module individually. Furthermore, as mentioned in subsection 3.4.3, the i-th Round_A module

doi:10.6342/NTU202301593

executes with a four-cycle delay compared to the i-th Round_E module, except for the 0_{th} round, which is calculated simultaneously in the module named Pre_round to ensure the fulfillment of the equations, also Precompute_E_0 and Precompute_E_0 will be calculated in Pre_round.

From the equations in subsection 3.4.2, the following equations can be obtained:

$$\begin{cases} E_0 = H_{init\_3} + H_{init\_7} + \Sigma_1(H_{init\_4}) + Ch(H_{init\_4}, H_{init\_5}, H_{init\_6}) + K_0 + W_0 \\ \quad = \mathbf{Pre\_E_0} + W_0 \\ A_0 = H_{init\_7} + K_0 + \Sigma_1(H_{init\_4}) + Ch(H_{init\_4}, H_{init\_5}, H_{init\_6}) + \Sigma_0(H_{init\_0}) \\ \quad + Maj(H_{init\_0}, H_{init\_1}, H_{init\_2}) + W_0 \\ \quad = \mathbf{Pre\_A_0} + W_0 \\ Precomputed\_E_0 = H_{init\_2} + K_1 + H_{init\_6} + W_1 \\ \quad = \mathbf{Precom\_E_0} + W_1 \\ Precomputed\_A_0 = E_1 - H_{init\_2} \end{cases} \quad (3.14)$$

Considering that the variables from H_{init_0} to H_{init_7} in Table 2.3 are constant values, and the value of variable K can be obtained from Table A.1 in Appendix A, it is possible to precompute the results of Pre_E_0, Pre_A_0, and Precom_E_0 in advance and treat them as constants(As shown in Table3.7). This optimization allows us to reduce the number of calculations required during the execution, and the module is depicted in FiguFigure 3.18

doi:10.6342/NTU202301593

Pre_E_0	0xD7A66E1FBCAF1B12
Pre_A_0	0x38CD0991C39458FE
$Precom\_E_0$	0xBFCC96F8BFC8CFC8

Table 3.7 The precompute values in the Pre_round module

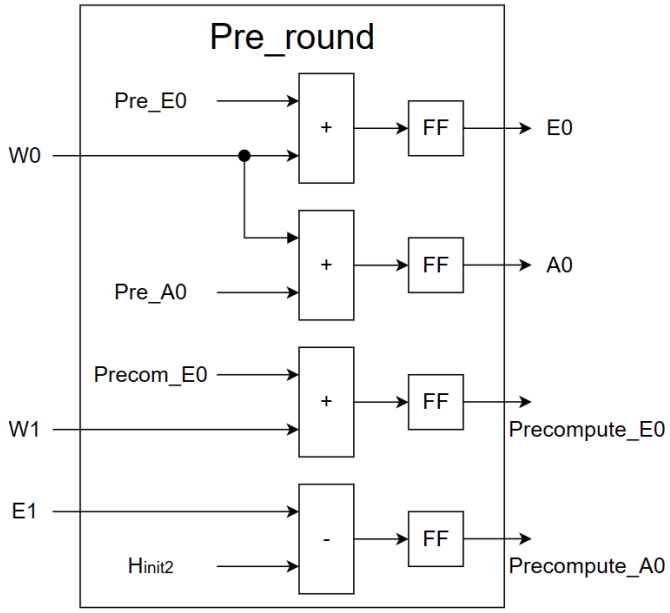


Figure 3.18 The architecture of the Pre_round module

The Final_add module is designed to perform the final step of the Transform_func, as mentioned in the C code provided in section 2.2.2. The equation for this step is as follows:

$$\begin{cases} D_{out} = D_{in} + H_{init\_3}; \\ C_{out} = C_{in} + H_{init\_2}; \\ B_{out} = B_{in} + H_{init\_1}; \\ A_{out} = A_{in} + H_{init\_0}; \end{cases} \quad (3.15)$$

doi:10.6342/NTU202301593

The Nonce_gen module is specifically designed to generate the nonce index for input header data. In our design, this module continuously generates the index in sequential order on each cycle, thereby fulfilling the requirement for full pipelining.

Finally, once the computation is completed, the resulting hash values are sent to the Nonce_compare module, where they undergo comparison with the target value configured by the software. If the hash values are smaller than the 256 bits' target value, they are deemed valid, and the subsequent valid nonce will be sent out.

The whole architecture of the SHA512-256d algorithm is shown in the Figure below.

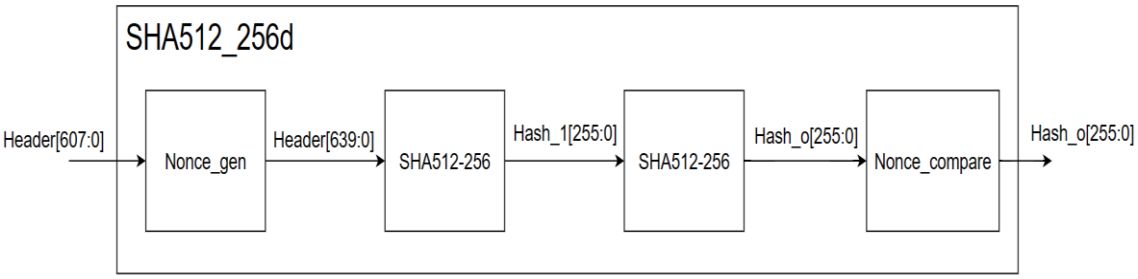


Figure 3.19 Improved architecture of the whole SHA512-256d algorithm

3.7 Overview of the whole system

3.7.1 The overview of the design on FPGA

The overview of the data flow of emulation on the FPGA will be introduced in this section. Initially, the software transmits data, including commands and addresses, to the FPGA using the Universal Asynchronous Receiver/Transmitter (UART) protocol. The data is transmitted as 32 bits values at a frequency of 900KHz. The received data is then processed by the BBBif_top module, which decodes the commands in UART protocol format.

doi:10.6342/NTU202301593

Subsequently, the data is configured into the CSR (Control Status Register), the registers of the Csrslv_top module, for storage, classification, and further processing. Finally, the processed data is sent to the SHA512-256d_top module for hash value computation.

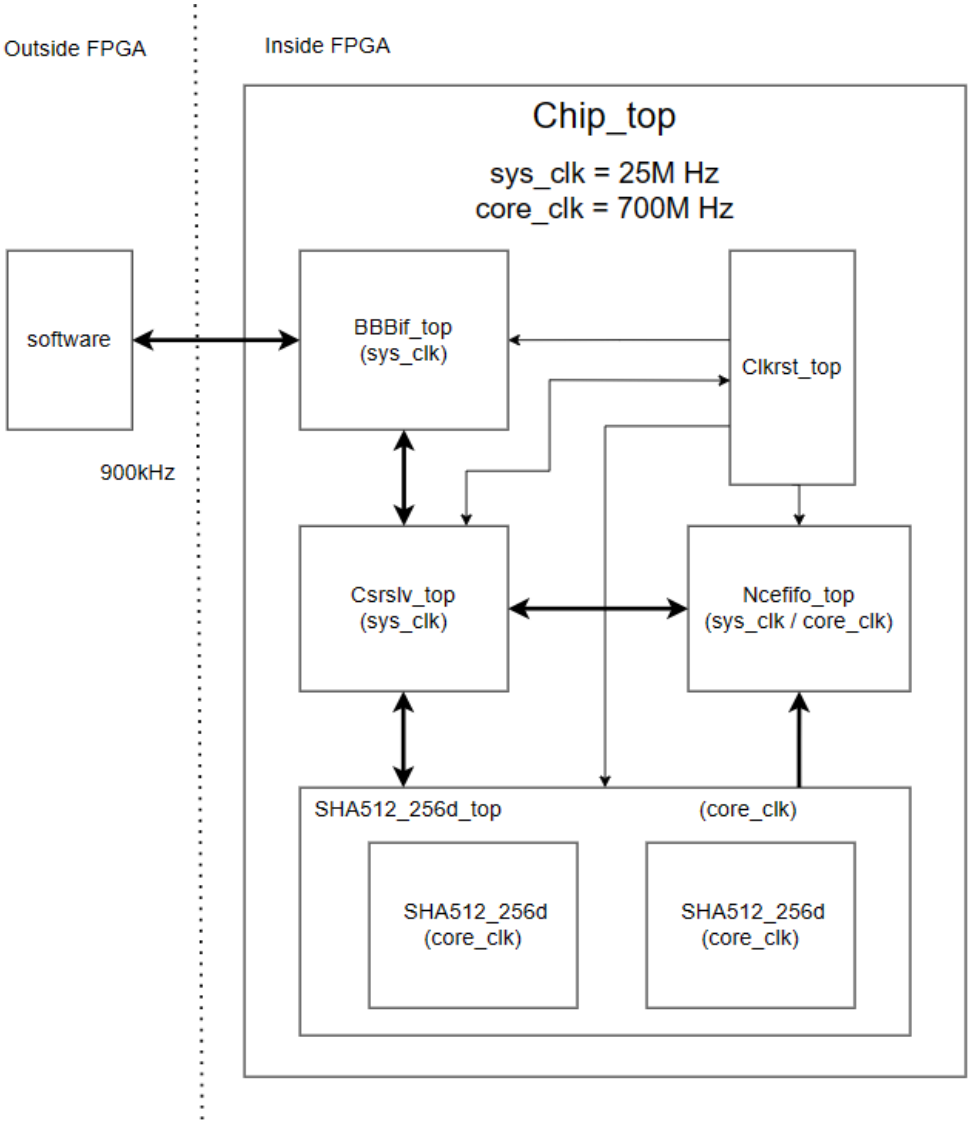


Figure 3.20 The block diagram of the whole system on FPGA
The bold lines are used for the indication of the main data flow

Within the SHA512-256d_top module, by adhering to the FPGA area limit, we can strategically place two SHA512-256d modules to operate in parallel. This

doi:10.6342/NTU202301593

configuration allows for an efficient doubling of the hash rate. After the computation is achieved, the valid hash values will be forwarded to the Ncefifo_top module, where the valid hashes are stored in the First-In-First-Out (FIFO) and await the read data command from the software.

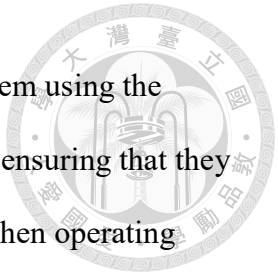
The software will read the address of the nonce_status signal continuously after all the header data has been sent, which means the SHA512-256d modules start executing. Once the FIFO within the Ncefifo_top module is not empty, implying that the valid nonce has been found, the signal named nonce_status is sent to the Csrslv_top module. Since the output of the Ncefifo_top module is directly connected to the Csrslv_top module, when the software receives a value of 1 for the nonce_status signal, it reads the CSR address that is linked to the output of the FIFO. Finally, the valid hash value is transmitted back to the software via the BBBif_top module, and the software verifies this value against the correct value obtained from the C code.

The system employs two clock domains, namely sys_clk and core_clk, which are generated by the MMCM IP within Clkrst_top. The core_clk, by default, operates at a frequency of 700MHz and is exclusively utilized by SHA512-256d_top and Ncefifo_top modules. On the other hand, the sys_clk runs at a frequency of 25MHz.

3.7.2 Ncefifo_top

As shown in Figure 3.20, we can find that Ncefifo_top is the only module that has two different clock frequency control. To ensure efficient and reliable data transfer, the valid nonces are written into the async-FIFO (asynchronous FIFO) using the core_clk, which typically runs at a higher frequency. This allows for faster processing and storage of the nonce values. By utilizing an async-FIFO buffer, we can temporarily

doi:10.6342/NTU202301593



store the valid nonces until they can be read and processed by the system using the sys_clk. This approach enables efficient handling of the valid nonces, ensuring that they are stored and available for processing at the appropriate time, even when operating with different clock frequencies.

To avoid the Clock Domain Crossing (CDC) problem, the address in the asynchronous FIFO is written in Gray code. This coding scheme ensures that when checking the fullness or emptiness of the async-FIFO, the writing and reading pointers do not cause errors due to metastability. By using Gray code for the address, transitions between adjacent states involve only a single bit change, reducing the likelihood of metastability issues during Clock Domain Crossing.

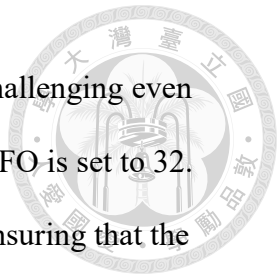
Additionally, certain debug signals, such as the flush signal, are handled using a double flop synchronizer. This means that the signal is operated with two consecutive flip-flops, which helps prevent metastability. The double flop synchronizer adds protection against potential timing issues that may arise when crossing clock domains. These measures, including the use of Gray code for address encoding and double flop synchronizers for critical signals, ensure the robust and reliable operation of the system.

Furthermore, considering the mining difficulty, the Nonce_compare module requires the hash target to have at least 32 bits of zeros in the Most Significant Bit (MSB) of the hash. This criterion ensures that the generated hash value meets the required difficulty level. When operating at a hash rate of 1400M hash/s, the probability of finding a valid nonce that satisfies the given hash target can be calculated as follows:

$$\text{Possibility of finding the valid nonce} = \frac{hash\ rate}{\text{hash target zeros in MSB}}$$

$$=< \frac{1400MH/s}{2^{32}} \sim 0.326\ hash/s \qquad (3.16)$$

doi:10.6342/NTU202301593



According to Equation 3.16, finding a valid nonce can be challenging even within a second. In the Ncefifo_top module, the depth of the async-FIFO is set to 32. This configuration implies that the FIFO is unlikely to become full, ensuring that the FIFO does not reach its maximum capacity quickly, minimizing the chances of overflow or loss of data, and the internal pointer is less prone to frequent changes. Consequently, the reduced frequency of pointer changes within the async FIFO improves the stability of the associated signals. By ensuring that the FIFO operates within a comfortable range of occupancy and minimizing rapid changes in the FIFO pointer, the overall stability and performance of the mining process are enhanced.

doi:10.6342/NTU202301593

Chapter 4 Measurement result

In this chapter, the implementation result of different architectures will be present, including the power measurement and the emulation result.

Resource / Module	439680 LUT		879360 FF		205440 LUT as memory	
Round	572	0.089%	512	0.072%	0	0%
Message_scheduler	14481	3.29%	44825	5.10%	3616	1.76%
SHA512-256	59061	13.43%	84950	9.66%	3616	1.7076%
SHA512-256d	116821	26.57%	167803	19.08%	7040	3.43%
SHA512-256d_top	233196	53.04%	335356	38.14%	14080	6.85%
Chip_top	238131	54.16%	339893	38.65%	14100	6.86%

Table 4.1 The resource utilization of original architecture implemented at 300MHz

Resource / Module	439680 LUT		879360 FF		205440 LUT as memory	
Round_E	392	0.089%	640	0.072%	0	0%
Round_A	200	0.045%	576	0.065%	0	0%
Word_compute	256	0.06%	384	0.040%	0	0%
Word_shift_element	128	0.03%	1792	0.20%	0	0%
Word_shift	3098	0.70%	45968	5.23%	0	0%

doi:10.6342/NTU202301593

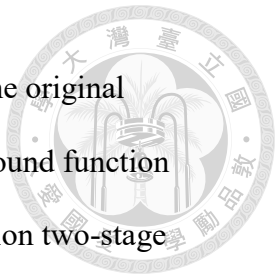
Word_SRL	7758	1.76%	6511	0.74%	7728	3.76%
Message_scheduler	18610	4.23%	64423	7.33%	7728	3.76%
SHA512-256	65981	15.01%	158109	17.98%	7728	3.76%
SHA512-256d	132575	30.15%	316821	36.03%	15488	7.54%
SHA512-256d_top	264689	60.20%	633550	72.05%	30976	15.08%
Chip_top	269638	61.33%	638079	72.56%	30996	15.09%

Table 4.2 The resource utilization of the normal two-stage pipelined with improved Round function architecture implemented at 550MHz

Resource / Module	439680 LUT		879360 FF		205440 LUT as memory	
Round_E	392	0.089%	640	0.072%	0	0%
Round_A	200	0.045%	576	0.065%	0	0%
Word_compute	256	0.06%	384	0.040%	0	0%
Word_shift_element	128	0.03%	1792	0.20%	0	0%
Word_shift	3099	0.70%	46104	5.24%	0	0%
Word_SRL	3456	0.79%	6316	0.72%	3456	1.68%
Message_scheduler	14311	3.25%	64414	7.33%	3456	1.68%
SHA512-256	58385	13.28%	152333	17.32%	3490	1.70%
SHA512-256d	120387	27.38%	312990	35.59%	6882	3.35%
SHA512-256d_top	242715	55.20%	628354	71.46%	13792	6.72%
Chip_top	247598	56.31%	632831	71.96%	13816	6.73%

Table 4.3 The resource utilization of the word expansion two-stage pipelined improved architecture implemented at 700MHz

doi:10.6342/NTU202301593



Tables 4.1, 4.2, and 4.3 lists the implementation results of the original architecture implemented at 250MHz, the two-stage pipelined with Round function improved architecture implemented at 500MHz, and the word expansion two-stage pipelined improved architecture implemented at 700MHz respectively. As shown in Table 4.1 and Table 4.2, after applying the improved Word_compute module which can calculate two words at the same time, the FF usage does not increase double directly due to the two-stage pipeline, causing the reduction of some utilization. Moreover, the FF can reduce more from the comparison of Table 4.1 and 4.3 due to Word expansion.

Tables 4.1, 4.2, and 4.3 present the implementation results for the original architecture implemented at 300MHz, the two-stage pipelined architecture with improved Round function implemented at 550MHz, and the word expansion two-stage pipelined improved architecture implemented at 700MHz, respectively. In Table 4.1 and Table 4.2, it is observed that the utilization of FFs does not directly double after the two-stage pipeline design. The reduction can be attributed to the enhanced Word_compute module, which facilitates the simultaneous calculation of two words. Furthermore, a reduction in FF usage can be observed when comparing Tables 4.2 and 4.3, primarily attributed to the implementation of the Word expansion technique.

Due to the inherent nature of the compute-hard algorithm SHA512-256d, the improvement in LUT utilization for logic optimization is not prominently achievable. However, by the cost of the rising numbers of registers for pipelining and the improved in round architecture, we can effectively prioritize increasing the hash rate by shortening the critical path. As shown in Table 4.4:

doi:10.6342/NTU202301593

	implement frequency	wns	tns
Original architecture	300M	-0.111	-169.208
Normal two-stage pipelined with improved Round function architecture	550M	-0.025	-10.113
Word expansion two-stage pipelined improved architecture	650M	0.0	0.0
	700M	-0.162	-589.046

Table 4.4 The timing results of architectures

The improved architecture demonstrates a significant enhancement in the implementation frequency, achieving a speed that is 2.3 times faster compared to the original architecture. Additionally, during emulation, the improved architecture can effectively operate at 700MHz, resulting in a corresponding hash rate of 1400MH/s. This hash rate is 2.8 times better than the original design's capability of running at 500MH/s. It is worth noting that despite the increase in FF utilization by 1.8 times, the improved architecture showcases substantial improvements in both implementation frequency and hash rate.

Emulation frequency	Hash rate	Power	Efficiency
600MHz	1.2GH/s	86.0w	14.0MH/w
650MHz	1.3GH/s	91.6w	14.2MH/w
700MHz	1.4GH/s	96.9w	14.4MH/w

Table 4.5 Power measurement

doi:10.6342/NTU202301593

Table 4.5 shows the power and efficiency of the improved architecture in different frequency emulations on the FPGA.

doi:10.6342/NTU202301593

Chapter 5 Future work

5.1 Review

The thesis presents a comprehensive design flow for enhancing the hardware architecture of the SHA512-256d algorithm on FPGA to achieve a higher hash rate. The author identifies a critical path that contributes to the low frequency of the original design. To address this issue, a two-staged pipeline is introduced, requiring a redesign of the logic architecture. The goal is to minimize the number of logic elements in the critical path. By analyzing the C code of the SHA512-256d algorithm, the architecture is designed to precompute certain variables, effectively reducing the length of the combinational path. Furthermore, through evaluation of FPGA resources, it is determined that Flip-Flops (FFs) are the primary resource for temporary data storage. As a result of these improvements, the improved design achieves a higher hash rate during emulation that reaches 1400GH/s, which is 2.8 times better than the original design's hash rate of 500MH/s. These enhancements highlight the efficacy of the redesigned architecture and the utilization of FFs for data storage in achieving substantial performance improvements.

5.2 Future work

According to Table 4.3, the improved design utilizes 56.31% of LUTs and 71.96% of FFs, without utilizing DSP or BRAM resources. As previously mentioned, the decision to exclude these resources was due to concerns about routing congestion

doi:10.6342/NTU202301593

and potential timing violations in a high frequency design. However, it is possible to incorporate these IPs into the design at the expense of a lower operating frequency. This may result in a reduction in LUT and FF utilization, allowing for the inclusion of more SHA512-256d modules in parallel, thus compensating for the hash rate. Nonetheless, it should be noted that routing congestion remains a significant challenge to achieving this concept.

Besides, there is room for improvement in the assignment of the module Message_scheduler. As discussed in section 3.5, adopting different methods for word computation and register expansion can result in significant register savings. The number of constant words varies depending on the input message, necessitating different strategies for optimizing register utilization.

doi:10.6342/NTU202301593

Reference

[1] H. L. Pham, T. H. Tran, T. D. Phan, V. T. D. Le, D. K. Lam, and Y. Nakashima, "Double SHA-256 hardware architecture with compact message expander for Bitcoin mining," IEEE Access, vol. 8, pp. 139634–139646, July 28, 2020.

[2] L. Li, S. Lin, S. Shen, K. Wu, X. Li, and Y. Chen, "High-throughput and area-efficient fully-pipelined hashing cores using BRAM in FPGA," Microprocessors Microsyst., vol. 67, pp. 922019–922082, Jun. 2019.

[3] Yin Zhang, Zhangqing He, Meilin Wan, Muwen Zhan, Ming Zhang, Kuang Peng, Min Song, and Haoshuang Gu, "A New Message Expansion Structure for Full Pipeline SHA-2," IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS—I: REGULAR PAPERS, VOL. 68, NO. 4, APRIL 2021

[4] Xilinx, Block Memory Generator v8.3, April 5, 2017

[5] Xilinx, UltraScale Architecture Configurable Logic Block, February 28, 2017

doi:10.6342/NTU202301593

[6] A. R. Zamanov, V. A. Erokhin, and P. S. Fedotov, “Asic-resistant hash functions,” IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus), pages 394–396. IEEE, 2018.

[7] Mohammad Peyraviana, Allen Roginskya, Ajay Kshemkalyanib, “On Probabilities of Hash Value Matches,” March. 1998

[8] Xilinx, UltraScale Architecture and Product Data Sheet: Overview, February 7, 2022

[9] Xilinx, UltraScale Architecture Configurable Logic Block User Guide, February 28, 2017

[10] Radiantblockchain.org, Radiant: A Peer-to-Peer Digital Asset System, August 11, 2022

[11] Xilinx, UltraScale+ FPGAs Product Selection Guide, June 20, 2023

[12] Xilinx, UltraScale Architecture DSP Slice User Guide, August 20, 2021

[13] Quynh H. Dang Gaithersburg, Penny Pritzker, Willie E. May, Charles H. Romine, “SECURE HASH STANDARD,” Federal Inf. Process. Stds. (NIST FIPS) - 180-4, August 4, 2015

doi:10.6342/NTU202301593

[14] Xilinx, UltraFast Design Methodology Guide for FPGAs and SoCs, June 07, 2023

[15] Xilinx, UltraFast Design Methodology Timing Closure Quick Reference Guide, November 30, 2022

[16] ZEYAD A. AL-ODAT, MAZHAR ALI, and ASSAD ABBAS, SAMEE U. KHAN, "Secure Hash Algorithms and the Corresponding FPGA Optimization Techniques," ACM Computing Surveys, Vol. 53, No. 5, Article 97, September 2020.

doi:10.6342/NTU202301593

Appendix

A.1 80 constants K_i for each SHA512-256 operations

index	value in hexadecimal	index	value in hexadecimal
0	0x428a2f98d728ae22	40	0xa2bfe8a14cf10364
1	0x7137449123ef65cd	41	0xa81a664bbc423001
2	0xb5c0fbcfec4d3b2f	42	0xc24b8b70d0f89791
3	0xe9b5dba58189dbbc	43	0xc76c51a30654be30
4	0x3956c25bf348b538	44	0xd192e819d6ef5218
5	0x59f111f1b605d019	45	0xd69906245565a910
6	0x923f82a4af194f9b	46	0xf40e35855771202a
7	0xab1c5ed5da6d8118	47	0x106aa07032bbd1b8
8	0xd807aa98a3030242	48	0x19a4c116b8d2d0c8
9	0x12835b0145706fbe	49	0x1e376c085141ab53
10	0x243185be4ee4b28c	50	0x2748774cdf8eeb99
11	0x550c7dc3d5ffb4e2	51	0x34b0bcb5e19b48a8
12	0x72be5d74f27b896f	52	0x391c0cb3c5c95a63
13	0x80deb1fe3b1696b1	53	0x4ed8aa4ae3418acb
14	0x9bdc06a725c71235	54	0x5b9cca4f7763e373
15	0xc19bf174cf692694	55	0x682e6ff3d6b2b8a3
16	0xe49b69c19ef14ad2	56	0x748f82ee5defb2fc
17	0xefbe4786384f25e3	57	0x78a5636f43172f60
18	0x0fc19dc68b8cd5b5	58	0x84c87814a1f0ab72
19	0x240ca1cc77ac9c65	59	0x8cc702081a6439ec
20	0x2de92c6f592b0275	60	0x90befffa23631e28
21	0x4a7484aa6ea6e483	61	0xa4506cebde82bde9
22	0x5cb0a9dcbd41fbd4	62	0xbef9a3f7b2c67915
23	0x76f988da831153b5	63	0xc67178f2e372532b
24	0x983e5152ee66dfab	64	0xca273eceea26619c
25	0xa831c66d2db43210	65	0xd186b8c721c0c207
26	0xb00327c898fb213f	66	0xeada7dd6cde0eb1e
27	0xbf597fc7beef0ee4	67	0xf57d4f7fee6ed178
28	0xc6e00bf33da88fc2	68	0x06f067aa72176fba
29	0xd5a79147930aa725	69	0x0a637dc5a2c898a6
30	0x06ca6351e003826f	70	0x113f9804bef90dae
31	0x142929670a0e6e70	71	0x1b710b35131c471b
32	0x27b70a8546d22ffc	72	0x28db77f523047d84
33	0x2e1b21385c26c926	73	0x32caab7b40c72493
34	0x4d2c6dfc5ac42aed	74	0x3c9ebe0a15c9bebc
35	0x53380d139d95b3df	75	0x431d67c49c100d4c
36	0x650a73548baf63de	76	0x4cc5d4becb3e42b6
37	0x766a0abb3c77b2a8	77	0x597f299cfc657e2a
38	0x81c2c92e47edaee6	78	0x5fcb6fab3ad6faec
39	0x92722c851482353b	79	0x6c44198c4a475817

doi:10.6342/NTU202301593

A.2 The number of registers required to delay each of the 64 Word_i inputs in the Word_SRL module

index	numbers of registers	index	numbers of registers	index	numbers of registers	index	numbers of registers
16	1	32	17	48	33	64	49
17	3	33	19	49	35	65	51
18	3	34	19	50	35	66	51
19	5	35	21	51	37	67	53
20	5	36	21	52	37	68	53
21	7	37	23	53	39	69	55
22	7	38	23	54	39	70	55
23	9	39	25	55	41	71	57
24	9	40	25	56	41	72	57
25	11	41	27	57	43	73	59
26	11	42	27	58	43	74	59
27	13	43	29	59	45	75	61
28	13	44	29	60	45	76	61
29	15	45	31	61	47	77	63
30	15	46	31	62	47	78	63
31	17	47	33	63	49	79	65

doi:10.6342/NTU202301593