electronics

MDPI

Article

A Symmetric and Multilayer Reconfigurable Architecture for Hash Algorithm

Wang Fan [1], Qinrang Liu [1,*], Xinyi Zhang [1], Yanzhao Gao [2], Xiaofeng Qi [2] and Xuan Wang [3]

[1] Institute of Information Technology, PLA Information Engineering University, Zhengzhou 450001, China; isfanw@163.com (W.F.)
[2] National Digital Switching System Engineering and Technological Research Center, Zhengzhou 450001, China
[3] College of Cyberspace Security, Zhengzhou University, Zhengzhou 450003, China
* Correspondence: lqr@ndsc.com.cn

Abstract: As an essential protection mechanism of information security, hash algorithms are extensively used in various security mechanisms. The diverse application scenarios make the implementation of hash algorithms more challenging regarding flexibility, performance, and resources. Since the existing studies have such issues as wasted resources and few algorithms are supported when implementing hash algorithms, we proposed a new reconfigurable hardware architecture for common hash algorithms in this paper. First, we used the characteristics of symmetry of SM3 (Shang Mi 3) and SHA2 (Secure Hash Algorithm 2) to design an architecture that also supports MD5 (Message Digest 5) and SHA1 (Secure Hash Algorithm 1) on both sides. Then we split this architecture into two layers and eliminated the resource wastes introduced by different word widths through exploiting greater parallelism. Last, we further divided the architecture into four operators and designed an array. The experimental results showed that our architecture can support four types of hash algorithms successfully, and supports 32-bit and 64-bit word widths without wasting resources. Compared with existing designs, our design has a throughput rate improvement of about 56.87–226% and a throughput rate per resource improvement of up to 5.5 times. Furthermore, the resource utilization rose to 80% or above when executing algorithms.

Keywords: hash algorithm; reconfigurable computing; hardware design; parallelism

Citation: Fan, W.; Liu, Q.; Zhang, X.; Gao,Y.; Qi, X.; Wang, X. A Symmetric and Multilayer Reconfigurable Architecture for Hash Algorithm. *Electronics* **2023**, *12*, 2872. https://doi.org/10.3390/electronics12132872

Academic Editor: Paris Kitsos

Received: 2 June 2023
Revised: 24 June 2023
Accepted: 26 June 2023
Published: 29 June 2023

1. Introduction

With the development of network and communication technology, modern society has higher requirements for protection of information security. As a branch of cryptography, hash algorithms have become an important protection mechanism of information security [1]. After years of development, many mature hash algorithms have been proposed and applied [2]. The most widely used hash algorithms are the MD family, SHA family, and national commercial cypher, specifically including MD5, SHA1, SHA2 (SHA224, SHA256, SHA384, SHA512), and SM3.

Hash algorithms convert a plaintext with arbitrary length into a ciphertext with fixed length [3] through a mapping algorithm which must be unidirectional, and collision resistant [4]. They are sensitive to input, such as the chaotic system [5]. The data processed by the hashing algorithms is greatly reduced in size and unrecoverable. As a computational method, hashing algorithms can be applied to many fields, such as encryption, secure authentication, and identification. Combining hash algorithms with cryptographic algorithms can implement digital signatures and identity authentication [6]. The processing or transmitting of messages processed by hash algorithms can reduce computation overheads, improve identification speed and ensure safety. Combining hash algorithms with watermarking techniques can be used for copyright protection, etc. [7]. In blockchain technology, many mechanisms require the involvement of hash algorithms [8]. For example, the miner

responsible for packing transactions needs to find a message digest that satisfies a specific format through hash computing and message digests from the transaction records of blocks should be calculated by hash algorithms, layer by layer, eventually forming a chain table that cannot be tampered with reversibly [9]. Another evident example is HTTPs protocol. HTTPs have become the mainstream Internet protocol replacing HTTP. The certificate required by Https is created by hash calculating [10]. In other words, without the hash algorithm, we barely have access to the Internet. So hash algorithms are a necessary component of modern technology.

In recent years, the scale of information transmitted by networks has increased dramatically. The information transmission speed in 5G networks can be 100 times faster than 4G [11]. The booming data scale and rapid request speed require the computational performance of the hashing algorithms to be improved because otherwise the digest generation speed inevitably lags far behind the service request speed. In addition, the expanding application scenarios also pose challenges. Although a single hash algorithm can perform data digest generation, many security protocols or software applications under the existing security framework often provide multiple hash algorithms for users to negotiate or choose from. For example, the IPSEC (Internet Protocol Security) protocol provides MD5 and SHA1 [12], etc., and the TLS (Transport Layer Security) protocol supports SHA256 and SHA384 [13], etc. These changes require that hash algorithms be implemented with high performance and flexibility.

So how to implement hash algorithms with high performance and flexibility is a problem that must be solved in technology development. Although existing research on hash algorithm acceleration has partially improved performance through dedicated design, resource-for-time, and local optimization means, the balance between performance, flexibility, and resources is not satisfactory and either supports few algorithms or wastes computational resources. To solve this problem, we analyzed the characteristics of common hash algorithms, searched for the primary causes of resource wastage at the structure and module levels, so as to propose a new reconfigurable architecture in this paper. The main contributions of this paper are summarized as follows:

1. We proposed a reconfigurable architecture from the perspective of the left–right symmetry at the structural level to solve the problem of resources unused when executing the algorithms.
2. We divided the computational structure into upper and lower layers, to solve the issue of resources being wasted due to different word widths among algorithms, separated the structure further into left and right parts, and analyzed the array design method through four generated operators.
3. We evaluated the performance of the architecture, and the experimental results showed that our design has a throughput rate improvement of about 56.87–226% and a throughput rate per resource improvement of up to 5.5 times, with a high resource utilization rate on the premise of algorithm reconfiguration.

The rest of this paper is organized as follows. Section 2 introduces the existing work and algorithm information. Section 3 describes our reconfiguration idea and detailed design. Section 4 is the experimental evaluation. Section 5 concludes this paper.

2. Background

2.1. Related Work

The research on hash algorithm acceleration can be divided into three main categories: ASIC (application-specific integrated circuit), reconfigurable design, and ASIP (Application Specific Instruction Set Processor).

The ASIC architecture designs a dedicated circuit for cryptographic algorithms, which mainly targets every single cryptographic algorithm and can achieve very high processing speed through parallel acceleration, pipeline design technology, and so on. Wang Zhendao achieved a processing speed of 81.31 Gb/s for the MD5 algorithm on the Arria10 device by optimizing the critical path and adopting 32-stage pipelining [14]. Fang

Yi [15] improved the SM3 processing speed to 80.43 Gb/s through 64-stage pipelining, but this is clearly a strategy of space for time. Another strategy is to design a multi-in-one architecture that shifts operations from the critical path to the non-critical path [16]. This strategy essentially calculates the critical path in advance, but it is not universal; for example, it is not applicable to MD5, and the effects of its improvements vary among algorithms because the number of operations which can be calculated in advance differ. The most evident advantage of ASIC is the execution speed, but it can only accelerate a single algorithm [17]. When multiple algorithms need to be accelerated, even if they are the same type, it needs to design multiple hardware architectures, which results in enormous hardware overheads and resource waste [18]. Multiple algorithms are rarely accelerated by this means.

Reconfigurable architecture means the overall architecture remains unchanged, but the chip functionality can still be adjusted according to the application requirements. It is as efficient as ASIC and as flexible as CPU through commonality analysis, operator extraction, reconfiguration cell design, interconnection, and configuration information generation [19]. The essential feature is that configuration information determines the data path. Much research has been carried out on reconfigurable architectures for hash algorithms, but there are still many problems, such as, for example, few algorithms are supported and there is severe waste of resources. Liu Heng from Zhejiang University proposed a hash algorithm reconfigurable architecture [20], in which the basic idea is to select different data paths for different algorithms through Mux by reusing the addition unit and implementing other operations separately. It can accelerate four standard algorithms, but the most severe problem is that, when performing 32-bit word width operations, more than 50% of the resources are idle, resulting in serious resource waste. The architecture proposed by Xi Shengxin only supports two algorithms [21], which directly merge the algorithmic data flow graphs, failing to avoid the resource wastage introduced by the data width problem. Yang Xiaohui designed an architecture that only supports SHA1/224/256 [22]. She used CSA (Carry Save Adder) adders to reduce the latency on the critical path but this doed not multiplex computational resources and, thus, cannot significantly improve performance and resource indicators. Zhu Ninglong designed an architecture, supporting SM3 and SHA2 [23], which only multiplexed the assignment circuit, while other computational logic was still implemented separately. Despite it supporting a few algorithms, its resource utilization still remains low. In Reference [24], the author multiplexed the adder through selecting different inputs and optimized the performance by using the CSA adder for SM3 and SHA2. Another type of design reconfigures the hash algorithm with the block cipher, such as the high-performance reconfigurable cryptoprocessors Anole [25] and PVHArray [26]. When running the hash algorithms, the reconfiguration granularity is only at the addition and shift levels, and, thus, the throughput is low, being only 460 Mb/s for SHA256 in Anole. So, even though this branch has been studied for a long time, there are still no efficient architectures for standard hashing algorithms that achieve balance in performance, flexibility, and area.

Hardware acceleration based on extended instruction sets involves designing instructions specifically for cryptographic computing on the CPU (Central Processing Unit) [27–29] and adding hardware acceleration units to the ALU (Arithmetic and logic unit) to execute these instructions [30], which often uses ultra-long instruction word techniques. However, it can never eliminate the constraint of von Neumann architecture, which makes it challenging to design coarse-grained instructions and issues of programmability exist [31]. In existing studies, the throughput rate for classical cryptographic algorithms is only 89.73 Mbps [32].

2.2. Introduction of the Algorithm

Hash algorithms are used to compute data digests, which compress plaintext with arbitrary length to ciphertext with fixed length. The computing can be divided into data padding, compression, and extension. Data compression and expansion are arithmetic-

intensive operations, which are suitable for hardware implementation, while the padding operation is a control-intensive operation, which is more suitable for software implementation, so we do not introduce the padding operation in this paper. Table 1 lists the algorithm information.

Table 1. Algorithm information.

Algorithm	Maximum Input Length	Block Length	Word Width	Output Length	Round
MD5	2^{64}	512	32	128	64
SHA1	2^{64}	512	32	160	80
SHA224	2^{64}	512	32	224	64
SHA256	2^{64}	512	32	256	64
SHA384	2^{128}	1024	64	384	80
SHA512	2^{128}	1024	64	512	80
SM3	2^{64}	512	32	256	64

Most hash algorithms adopt an iterative compression structure with strong similarities. Figure 1 shows the schematic diagram of the compression function [14,33–35], where the letters of a to h are the input/output registers. And Table 2 depicts these in mathematical language. Table 3 shows the data extension rule.

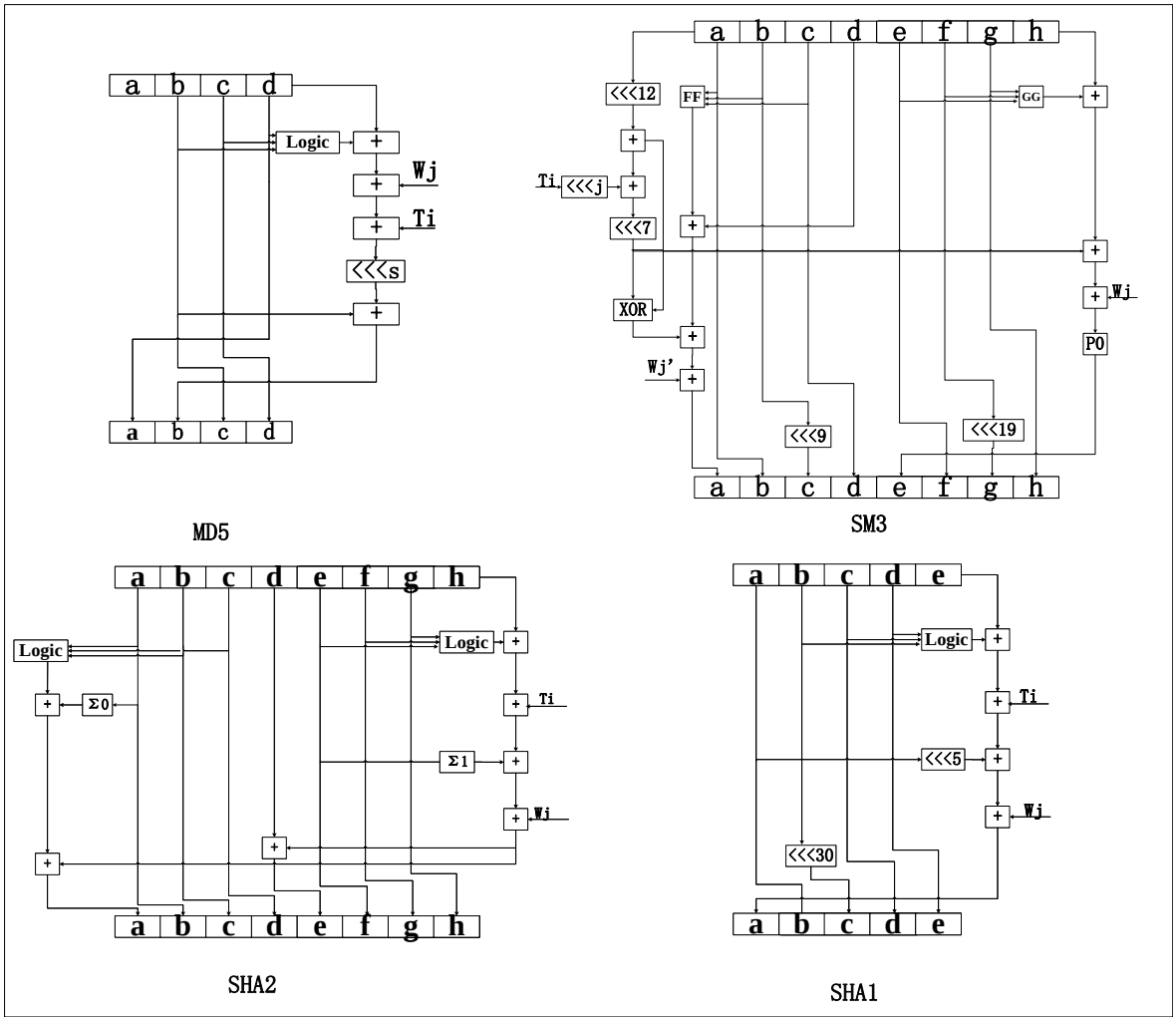


Figure 1. Compression function structure.

Table 2. Compression function in mathematical description.

Algorithm	Mathematical Description
MD5	$(A, B, C, D) \rightarrow (D, ((A + F/G/H/I(B, C, D) + W + T) <<< s) + B, B, C)$
SHA1	$(A, B, C, D, E) \rightarrow (\mathrm{Ft}(B, C, D) + E + (A <<< 5) + W + K, A, (B <<< 30), C, D)$
SHA2	$T1 = \sum_1(E) + H + Ch(E, F, G) + K + W$ $T2 = \sum_0(E) + Maj(A, B, C)$ $(A, B, C, D, E, F, G, H) \rightarrow (T1 + T2, A, B, C, D + T1, E, F, G)$
SM3	$SS_1 = ((A <<< 12) + E + (T <<< j)) <<< 7$ $SS_2 = SS_1 \oplus (A <<< 12)$ $TT_1 = FF_j(A, B, C) + D + SS_2 + W'$ $TT_2 = GG_j(E, F, G) + H + SS_1 + W$ $(A, B, C, D, E, F, G, H) \rightarrow (TT_1, A, (B <<< 9), C, P_0(TT_2), E, (F <<< 19), G)$

Table 3. Data extension rule.

Round(s)	Algorithm	Extension Rules
$0 \leq s < 16$	all	$W_s = M_s$
$s > 16$	MD5	Select one form the plaintext word
	SHA1	$W_s = (W_{s-3} \oplus W_{s-8} \oplus W_{s-14} \oplus W_{s-16}) <<< 1$
	SHA1	$W_s = \sigma_1(W_{s-2}) + W_{s-7} + \sigma_0(W_{s-15}) + W_{s-16}$
	SHA1	$W_s = P_1(W_{s-16} \oplus W_{s-9} \oplus (W_{s-3} <<< 15)) \oplus (W_{s-13} <<< 7) \oplus W_{s-6}$

3. Reconfiguration Architecture

In this section, we analyze the reconfiguration strategy and design the reconfiguration architecture at three levels: the structural level, the module level, and the array level. At the structural level and module level, our main methodology was multiplexing as much as possible, including using the symmetry of the algorithm to design architecture and splitting the architecture to exploit greater parallelism. At the array level, we separated the architecture again to decouple all operators and then analyzed the array design method to select an optimal scheme according to some common metrics.

3.1. Reconfiguration Analysis

It is easy to observe, in Figure 1, that the overall structure of the compression function is similar. They all have a critical path based on addition, and several paths based on simple logic operations, and the assignment logic for them all presents the characteristics of shift assignment. In existing designs, the difficulty of reconfiguration is SM3, which needs more computational resources. The architecture must provide all the resources required by SM3. If an appropriate reconfiguration strategy is not adopted, it results in 50% waste of adder and 5/8 register resources when performing MD5 and SHA1. After careful observation, we found that both the left and right sides of SM3 have a complex path, and the structure of each side can contain MD5 and SHA1. SHA2 also presents such characteristics. So, we concluded that designing an architecture according to SM3 and SHA2, and supporting MD5 and SHA1 on both sides of the architecture, would fully utilize computational resources at the structural level. This was the reconfiguration strategy adopted in this paper.

The above analysis solves the problem of resource waste at the structural level, but serious resource wastage still exists in the phase of algorithmic performance. The fundamental reason is that word widths vary among algorithms. The word widths of SHA384 and SHA512 are 64 bits, while the others are 32 bits. In existing designs, the word width of all computational resources is designed to be 64 bits, resulting in at least 50% of the resource being in an idle state when executing algorithms with 32-bit word width [20–24]. However, if designed to 32 bits, this cannot meet the operational requirements of 64-bit word widths. To address the above problem, we used a processing unit with 32-bit word width to build a unit with 64-bit word width and to select execution modes through control signals. So, after optimizing, the architecture can execute one set operation of 64-bit word

width or two sets of operations of 32-bit word width simultaneously, which means that the architecture can support two sets of SM3 or SHA224/256, etc., or one set of SHA384/512 at the algorithm level. So, the problem of resource waste introduced by word widths is solved.

Data expansion rules among algorithms are simple, and the most crucial feature is 16-level shift registers and simple arithmetic operations. Different computing logic is selected according to the algorithms. The output value of the shift register is passed directly to the iterative compression module to participate in computing.

3.2. Structural Level Design

In this section, we design the compression module and data extension module separately.

3.2.1. Compression Module Design

In the above reconfiguration analysis, the implementation of MD5 and SHA1 should reuse the resources of the left and right sides of SM3 and SHA2 as much as possible. According to this principle, we designed the hardware architecture, shown in Figure 2, which the symbols of A to E and A_n to E_n mean the input and output registers respectively. And it satisfies the following design requirements:

1. Supports SHA2 and SM3;
2. Supports MD5 and SHA1 on both sides.

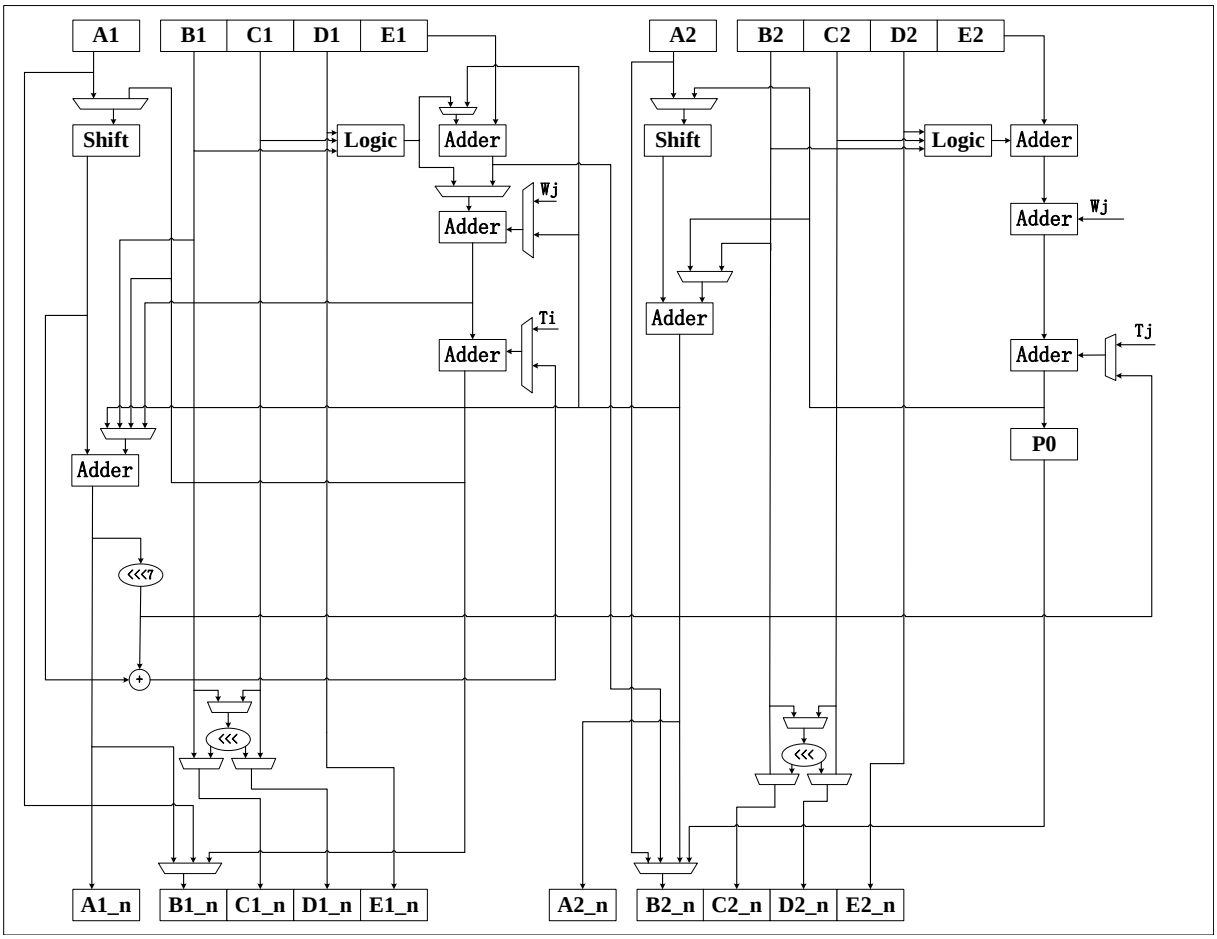


Figure 2. Reconfigurable architecture.

To be compatible with SHA1 on both sides, we set register numbers on each side to 5. The extra registers can execute other computational branches when performing other algorithms. For example, the operation of $(T_i <<< j) + E$ in SM3 can be implemented by A2, the shift unit, and the addition unit. For MD5 and SHA2, the extra registers can copy

the values of B1 and B2 to alleviate their fan-out. Figures A1–A4 in Appendix A provide the data flow diagram when executing the four algorithms.

It can be seen in the data flow diagram, that few wasted resources exist at the whole structural level. When running MD5 and SHA1 algorithms, resources on both the left and right sides are utilized, and the throughput rate for these algorithms is theoretically doubled.

3.2.2. Data Extension Module Design

The data extension module needs to calculate Wi for the compression module. The data extension rules for different algorithms are shown in Table 3. The data path to execute can be selected according to the selected algorithm. According to the reconfiguration analysis, we designed the hardware architecture shown in Figure 3. The 16 registers are W_0–W_{15}, W is the extended data word, and $W'_{\_SM3}$ is W_p is used in SM3.

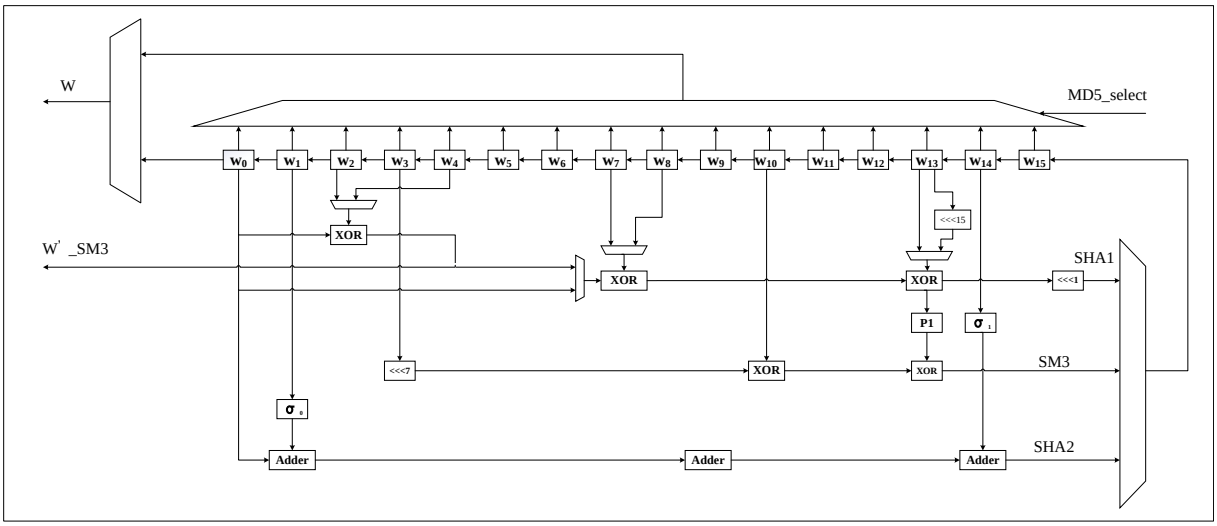


Figure 3. Data extension module.

3.3. Module Design

The architecture above only provides the design at the structural level, but two issues remain unexplained:

1. How to set word width;
2. How to design the sub-module.

To be compatible with SHA512 and SHA384, in existing works, the word width of the computational units and registers are all set to 64, and when executing 32-bit width algorithms, only the high 32 bits are used, while the low 32 bits are set to 0, which leads to at least 50% waste of resources. To solve the problem, we built the processing units with 64-bit word width through units with 32-bit data width. The specific idea was to divide the 64-bit processing unit into high 32-bit and low 32-bit. Each part can handle 32-bit operation independently, while the two can work together to handle 64-bit operation. In other words, we divided the hardware structure shown in Figure 2 into two layers, with the upper layer processing high 32 bits data and the lower layer processing low 32 bits data, and an algorithm with a bit width of 64 bits, such as SHA384/512, is calculated by the two layers together. The data extension module also adopts this strategy.

The logic unit in Figure 2 undertakes the task of calculating the logic functions shown in Table 4. The logic unit must be reconfigurable to meet all the computational requirements. We adopted a three-layer CLB interconnection structure in this paper, with each CLB word width being 32 bits. The detailed structure is shown in Figure 4. The logic function is bit operations and the high and low bits do not disturb each other, so the logic function units in both layers are the same as in Figure 4.

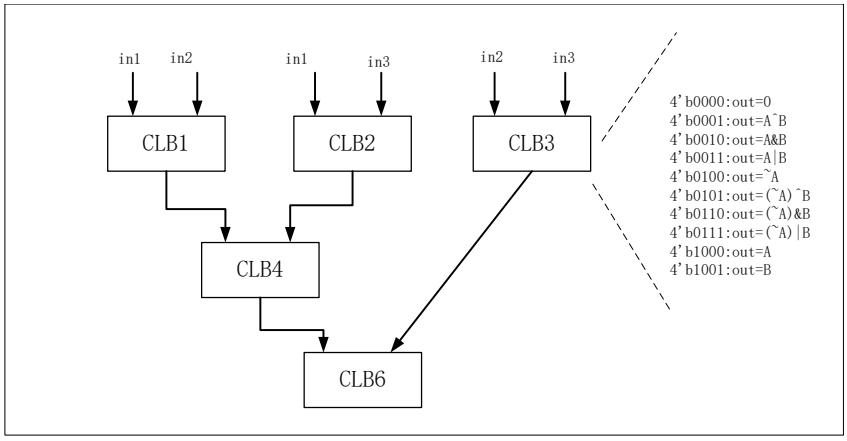


Figure 4. Logic function unit.

Table 4. Logic functions of algorithms.

Algorithm	Function Symbol	Definition
MD5	F(B,C,D)	$(B \wedge C) \vee (\overline{B} \wedge D)$
	G(B,C,D)	$(B \wedge D) \vee (C \wedge \overline{D})$
	H(B,C,D)	$B \oplus C \oplus D$
	I(B,C,D)	$C \oplus (B \vee \overline{D})$
SHA1	F1(B,C,D)	$(B \wedge C) \vee (\overline{B} \wedge D)$
	F2(B,C,D)	$B \oplus C \oplus D$
	F3(B,C,D)	$(B \wedge C) \vee (B \wedge D) \vee (C \wedge D)$
	F4(B,C,D)	$B \oplus C \oplus D$
SHA2	Ch(B,C,D)	$(B \wedge C) \oplus (\overline{B} \wedge D)$
	Maj(B,C,D)	$(B \wedge C) \oplus (B \wedge D) \oplus (C \wedge D)$
SM3	FF_j(B,C,D)	$B \oplus C \oplus D 0 \leq j \leq 15$
		$(B \wedge C) \oplus (B \wedge D) \oplus (C \wedge D) 16 \leq j \leq 63$
	GG_j(B,C,D)	$B \oplus C \oplus D 0 \leq j \leq 15$
		$(B \wedge C) \vee (\overline{B} \wedge D) 16 \leq j \leq 63$

The addition operation was designed differently for high and low layers due to the carry flag, such as the structure shown in Figure 5. Our goal was to construct a 64-bit adder with two 32-bit adders which could work separately or cooperatively. When working separately, they are essentially two independent 32-bit adders, without carrying any flag. When they work cooperatively, they are essentially a 64-bit carry ripple adder consisting of two 32-bit adders. Now, the low bits adder needs to output a carry flag to the high bits adder. The two modes can be selected through the dword signal.

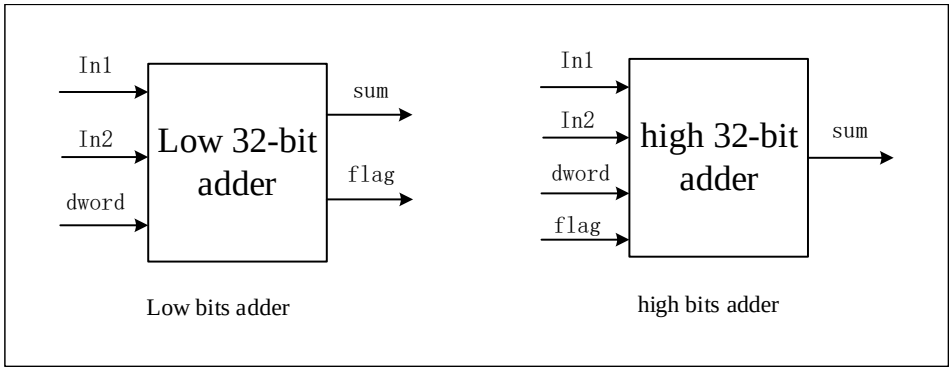


Figure 5. Adder module.

When analyzing algorithms, we found that there are many computational patterns, such as $(A >>> n_1 \oplus \mathrm{A} >>> n_2 \oplus \mathrm{A} >>> n_3)$. So, we designed a dedicated shift unit.

The shift unit includes three shift operations and two xor operations. The unit is designed to perform two functions:first, shift for one input; second, multiple shifts and xor for one input. Since 64-bit shift operations cannot be calculated through 32-bit shift operations, 64-bit shift operations cannot reuse the 32-bit shift operation calculation logic. The structure of the designed shift unit is shown in Figure 6. The lower layer uses the low bits shift unit, and the upper layer uses the high bits shift unit. The σ_0 and σ_1 operations in the data expansion module can also be implemented by the same method but with a fixed shift number.

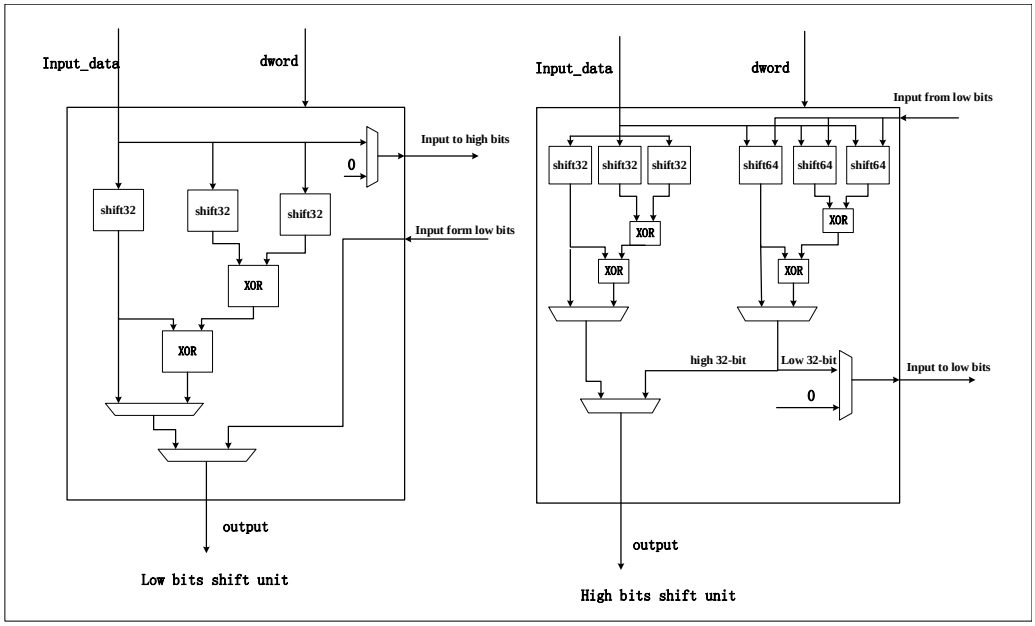


Figure 6. Shift unit.

After the module was designed, as above, the architecture in Figure 2 was divided into two layers. Each layer can perform two sets of MD5 and SHA1 and one set of SHA256, SHA224, and SM3. When executing SHA384 and SHA512 algorithms, the upper and lower layers work together to expand the computing logic and register unit word width to 64 bits. In this way, the waste of resources introduced by the different word widths is eliminated, and the computational capability of all resources can be utilized entirely.

3.4. Array Design

In the above analysis, we optimized the reconfigurable architecture at the structural and module levels. However, due to massive iterations and computation of hash algorithms, a single hardware structure only provides limited computational capability, and, thus, the array design is necessary. The design of the array affects the implementation model of the performing algorithms, and the main difference is whether or not to adopt the pipeline. After analysis, we found that the pipelined implementation was not suitable for the hashing algorithm, mainly for the following reasons: first, if the data extension module pipelines, each data extension module introduces an extra 512 bits interconnection overhead for data words pipelining, as shown by the red line of Mode a in Figure 7; second, if the data expansion module does not pipeline, this means that a set of data is processed in only one data expansion module, as shown in Mode b of Figure 7. At this time, each data expansion module needs to connect with all the compression modules, which introduces enormous interconnection and control overheads and may, ultimately, result in frequent switching of configuration information or failures in data synchronization. Therefore, we did not use the pipelined implementation and processed the whole hash computing in a set of data extension modules and compression modules, as shown in Mode c of Figure 7. From another perspective, pipelined implementation is more suitable for dealing with

programs containing multiple sub-processes. However, our design can complete the whole iteration process just once without dividing into several sub-processes, and, hence, we did not use the pipelined implementation.

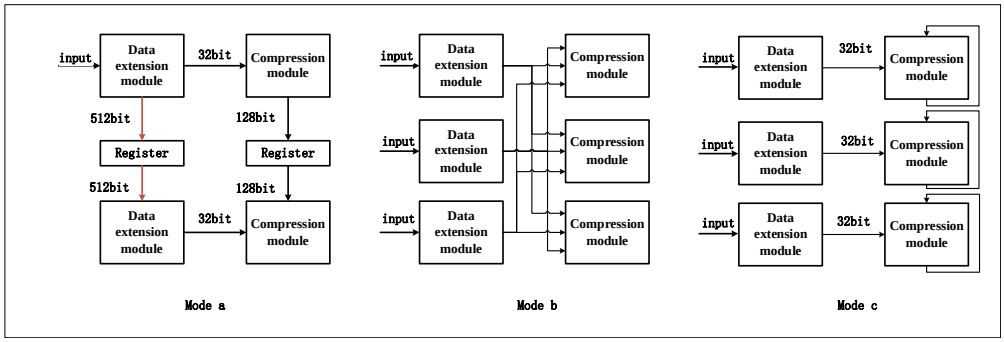


Figure 7. Executing mode.

In array design, function units should be decoupled as much as possible to reduce control difficulties and to increase flexibility. In Section 3.2.1, we designed the architecture according to the principle of left–right symmetry, so, when designing arrays, we could separate the left and right sides from the middle to obtain the two sub-modules. And we denote the left side as A and the right side as B. We connected modules A and B with the data expansion module, deriving A and B operators, as shown in Figure 8. The A and B operators can work together by transmitting three sets of signals. So far, we had obtained four operators according to the high/low and left/right separation principles: A_high, A_low, B_high, and B_low. Table 5 provides the interconnection relations between the operators. Table 6 depicts the requirements of the different algorithms for the operators.

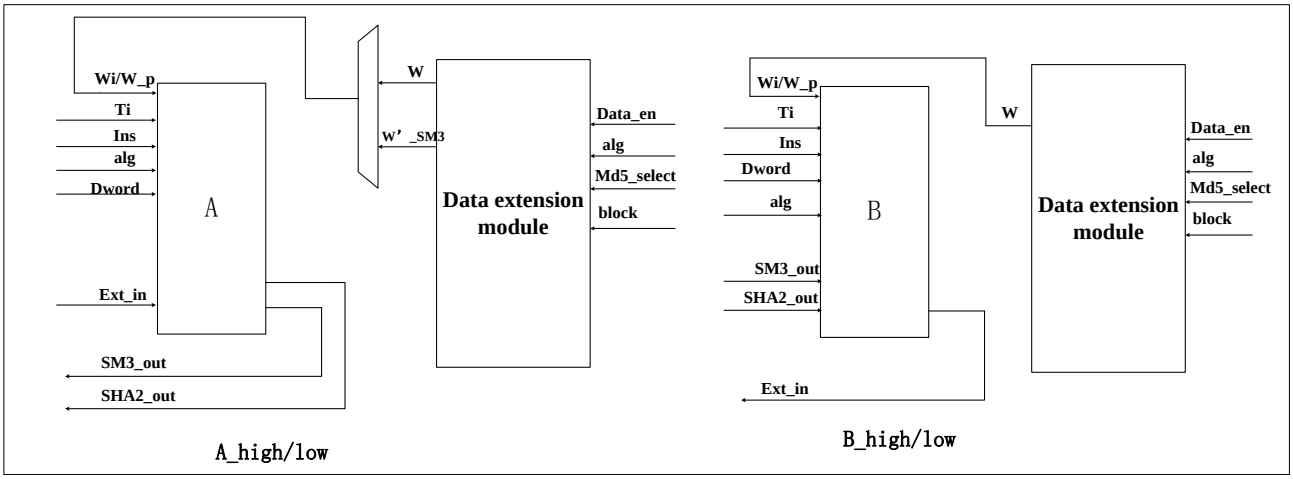


Figure 8. Operators.

Table 5. Interconnection among operators.

	A_high	A_low	B_high	B_low
A_high	×	Low/High	Left/Right	×
A_low	Low/High	×	×	Left/Right
B_high	Left/Right	×	×	Low/High
B_low	×	Left/Right	Low/High	×

Table 6. Operator requirements of algorithms.

Algorithm	Operators
MD5	A_high or A_low or B_high or B_low
SHA1	A_high or A_low or B_high or B_low
SHA224/256	A_high + B_high or A_low + B_low
SHA384/512	A_high + B_high + A_low+B_low
SM3	A_high + B_high or A_low + B_low

The computing capability of the array is positively related to the number of computational resources, so we needed to determine the scale of the array first. Assuming that the array size is M × N, where N is the width of the array, and M is the depth of the array, when the array is not designed in the pipeline, its computational power is proportional to M, so the size of N is the crucial factor affecting the array efficiency. The design of the row structure should consider the parallelism of multiple algorithms and single algorithm. Parallelism of a single algorithm means how many messages belonging to one algorithm are executing simultaneously, while for multiple algorithms it means the number of messages many kinds of algorithms are executing simultaneously. When considering the maximum parallelism of multiple algorithms, the row must satisfy the requirements for computational resources of all algorithms, as shown in Equation (1).

$$\begin{aligned} &R_{SM3} + R_{SHA224/256} + R_{SHA384/512} + R_{MD5} + R_{SHA1} \leq \\ &N_1 * A\_high + N_2 * A\_low + N_3 * B\_high + N_4 * B\_low \end{aligned} \tag{1}$$

where R_{alg} is resources occupied by algorithm, which is shown in Table 6, and N_i is the number of every operator in one row. According to the barrel principle, the parallelism of a single algorithm depends on the smallest amount of resources required by the algorithm. Equation (2) shows the parallelism of a single algorithm.

$$\begin{aligned} &P_{\mathrm{SM3}} = \min\{\mathrm{N1}, \mathrm{N3}\} + \min\{\mathrm{N2}, \mathrm{N4}\}, \\ &P_{\mathrm{SHA224/245}} = \min\{\mathrm{N1}, \mathrm{N3}\} + \min\{\mathrm{N2}, \mathrm{N4}\}, \\ &P_{SHA384/512} = \min\{N1, N2, N3, N4\}, \\ &P_{MD5} = N1 + N2 + N3 + N4, \\ &P_{SHA1} = N1 + N2 + N3 + N4 \end{aligned} \tag{2}$$

After a simple analysis, the minimum design solution meeting the parallelism requirements was $N_1 = N_2 = 2$, $N_3 = N_4 = 3$, or $N_1 = N_2 = 3$, $N_3 = N_4 = 2$. Both solutions provide the same computing capability, and the one that occupied less area was a better choice. The parallelism of the individual algorithms at this time is shown in Table 7.

Table 7. Parallelism of individual algorithm.

	MD5	SHA1	SHA224/256	SHA384/512	SM3
Parallelism	10	10	4	2	4

In this paper, we took $N_1 = N_2 = 2$, $N_3 = N_4 = 3$ as an example. The design could achieve maximum parallelism only when the interconnection relationships in Table 5 were satisfied. At this time, the whole problem became a permutation problem. After excluding part solutions that did not satisfy the interconnection relationship and equivalent solution, the number of solutions satisfying requirements was $(C_5^2 - 2)/2 = 4$, and the detailed solution is shown in Figure 9.

Although the four schemes provide the same maximum parallelism for each algorithm, they do not provide the same mapping flexibility. In this paper, we defined the number of mapping solutions as mapping freedom, and Table 8 gives the mapping freedom of the four schemes.

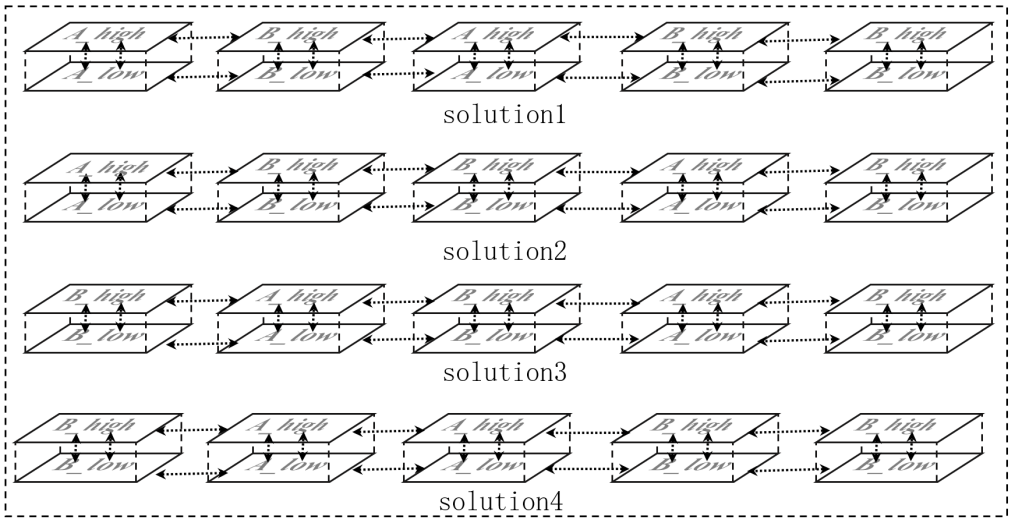


Figure 9. Array design solution.

Table 8. Mapping freedom.

	Mapping Freedom with Parallelism = 1			**Mapping Freedom with Max_Parallelism**		
	MD5 [1]	**SHA224** [2]	**SHA384** [3]	**MD5** [1]	**SHA224** [2]	**SHA384** [3]
so1ution1	10	6	3	1	1	1
solution2	10	6	3	1	2	2
solution3	10	8	4	1	3	3
solution4	10	4	2	1	1	1

[1] SHA1 has the same value as MD5; [2] SHA256 and SM3 have the same value as SHA224; [3] SHA512 has the same value as SHA384.

Therefore, we chose solution 3 as the row design solution in this paper, and the array structure is shown in Figure 10. No interconnection is required between rows, and interconnecting lines within rows and layers are used for the operators to work together. Let us denote the row computing capability as Cap_row, then the array computational capability $Cap_{array} = M * Cap_{row}$. The maximum throughput rate of the array when the computational resources are fully utilized is:

$$\mathrm{Th_{array}} = \sum_{i=0}^{4} \alpha_{\mathrm{i}} * \mathrm{Th\_alg[i]} \tag{3}$$

where $\mathrm{alg} = [MD5, SHA1, SHA224/256, SHA384/512, SM3]$, α_{i} is the number of messages of alg[i], $\mathrm{Th_{array}}$ is the array throughput rate, Th_alg[i] is the throughput rate when running alg[i], and satisfies $\alpha_1 + \alpha_2 + 2\alpha_3 + 4\alpha_4 + 2\alpha_5 = \mathrm{M} * 10$.

The mapping requirements for multiple algorithms can be satisfied according to the following principles: first, do not map MD5 and SHA1 on the same layer; second, search the operator suitable for algorithms from upper to lower and left to right.

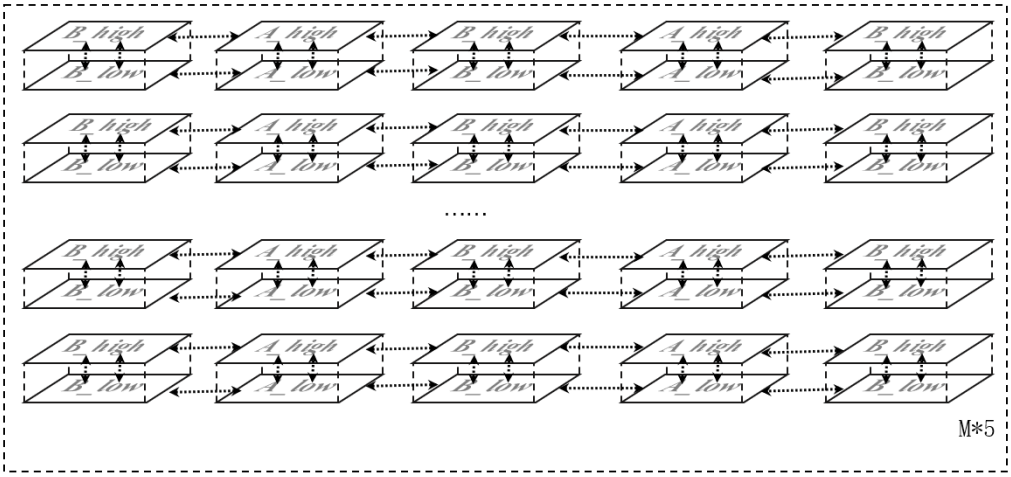


Figure 10. Array.

4. Experiments

In this section, we validate and test the architecture proposed above. Figure 11 shows the verification system platform used. When executing algorithms on our architecture, the users need to input the algorithm type and plaintext and generate the corresponding configuration information. For other information, such as Ti and round in algorithms, the information can be stored in the system in advance. After comparing our computing results with the message digests calculated by software tools, the correctness of our design was validated. To test the performance, we used Quartus II 13.0 to synthesize our design and selected the Stratix II family device to lay out.

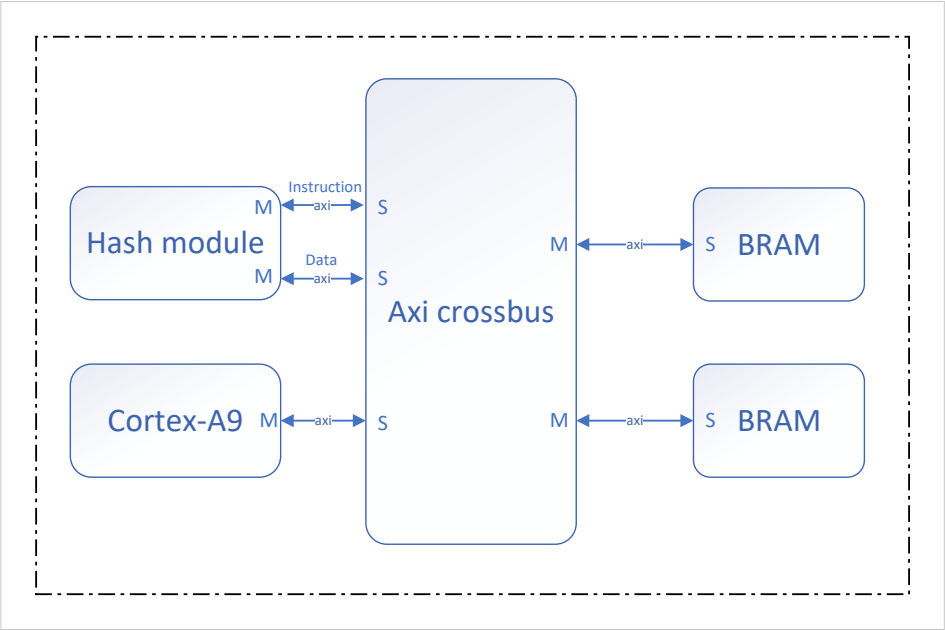


Figure 11. Verification system platform.

4.1. Area Overhead

We could obtain the number of Adaptive Look-Up Tables (ALUTs) and Register easily after laying out our operators and their combinations on Quartus II. The Power included the total of dynamic and static power. However, when performing timing analysis, we could not use the results of the TimeQuest Timing Analyzer as the maximum frequency of our design because Quartus uses the longest path in the design as the critical path by default to perform timing analysis. However, for our design, algorithms supported by

the architecture were fixed, and the computing paths for each algorithm were also fixed. The longest path in the design was not, in fact, used. Therefore, the algorithm's longest computing path should be used as the critical path in timing analysis, rather than the longest path in the whole design. In this paper, we searched the critical path of each operator when algorithms were running through the Report path and Set false path options to calculate the maximum frequency. Then, we used the Frequency parameter to calculate the Latency. The specific information obtained is shown in Table 9.

Table 9. Operator information.

Operator	ALUT	Register	Power (W)	Frequency (MHZ)	Latency (ns)	Algorithm Supported
A_low	1872	772	1.956	125.64	7.959	MD5 and SHA1
A_high	2572	778	2.141	118.69	8.435	MD5 and SHA1
B_low	1793	740	2.208	121.21	8.25	MD5 and SHA1
B_high	2468	778	2.45	131.7	7.593	MD5 and SHA1
A_low+B_low	3711	1512	3.732	115.32	8.671	SM3 and SHA224 and SHA256
A_high+B_high	4907	1556	3.913	111.27	8.987	SM3and SHA224and SHA256
A_high+B_high +A_low+B_low	8882	3068	5.7	108.96	9.177	SH512and SHA384

To compare the computing capability of the architectures with other designs, we selected the A_high + B_high + A_low + B_low scheme. The results of all designs are shown in Table 10, and the target devices of the design in Table 10 were all from the Stratix II family for convenience of comparison.

Table 10. Architecture comparison.

	Algorithms Supported	ALUT	Register	Maximum Frequency/MHz	Throughput Rate/Mbps
Individual implementation	MD5	2334	892	133.6	1068.8
	SHA1	1131	874	171.821	1099.65
	SHA-224/256	2150	1066	136.78	1094.24
	SHA-384/512	4316	2126	127.535	1632.45
	SM3	1936	1161	68.9	526.2
Ref. [24]	SHA1	5887	2124	105.7	727.8
	SHA-224/256				909.8
	SHA-384/512				1455.7
Ref. [20]	MD5	7441	2210	100	800
	SHA1				640
	SHA-224/256				800
	SHA-384/512				1066.6
	SM3				742
ours	MD5	8882	3068	108.96	3486.72
	SHA1				2789.37
	SHA-224/256				1716.54
	SHA-384/512				1394.688
	SM3				1716.54

We easily ascertained that the ALUT number that implemented all algorithms individually was 11,876, and the Register number was 6119, as can be observed from Table 10. The individual implementation did not reuse any computational resources and, thus. seriously wasted resources. Figure 12 provides the resource overheads of the design in Reference [20], our design, and the individual implementation scheme. We could not compare other schemes under such conditions due to the limited algorithms they support.

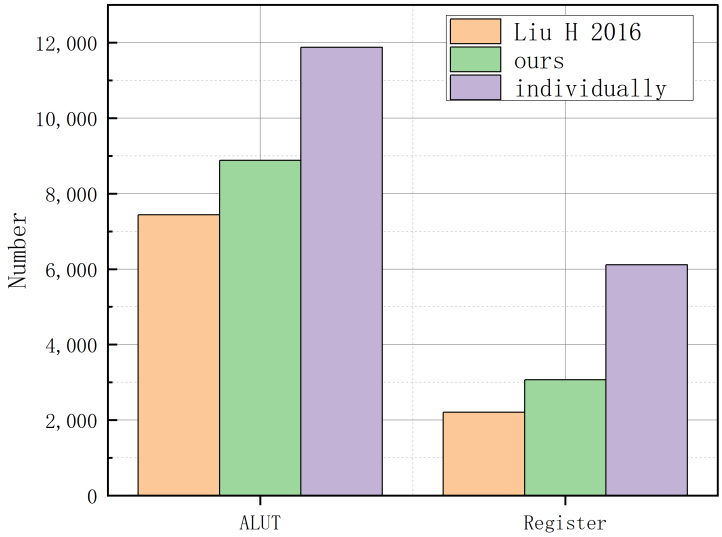


Figure 12. Resources overhead [20].

It was easy to ascertain that, compared with the individual implementation scheme, both the design in Reference [20] and our design could save ALUT and Register to a large extent because both reused computational resources and, thus, the amounts of computational units and registers were reduced. However, the design in Reference [20] achieved more significant resource savings than our design because our architecture provides more powerful reconfiguration capabilities and, therefore, requires more computational resources, providing greater parallelism and more powerful computational capability . Even so, our design also achieved more than 25% ALUT savings and 45% Register savings.

4.2. Resource Utilization Rate

In this paper, our core innovation was to design an architecture and split it into two sides and two layers based on the characteristics of symmetry and word width. By multiplexing, each computing resource can be fully utilized. To validate this feature, we adopted the resource utilization rate indicator to evaluate the resource utilization degree in the compression module. The ALUT numbers sub-modules occupied after synthetization are shown in Table 11.

Table 11. ALUT numbers sub-modules occupy .

	A_high_com [1]	**A_low_com** [1]	**B_high_com** [1]	**B_low_com** [1]
Adder_high1	97	0	64	0
Adder_high2	96	0	96	0
Adder_high3	96	0	97	0
Adder_high4	64	0	96	0
Adder_low1	0	65	0	33
Adder_low2	0	65	0	33
Adder_low3	0	64	0	65
Adder_low4	0	34	0	65
Shift_high	619	0	677	0
Shift_low	0	271	0	336
Logic	162	162	162	162
<<<	175	175	174	175
P0	0	0	32	32
Others [2]	380	412	239	266
total resource	1689	1248	1637	1167
The resources of all compress modules after synthesization: 5802				

[1] com means the compression module of the corresponding operator; [2] Other means other modules, excepting the modules listed in the table.

In Table 11, the same sub0module occupied different amounts of resources in different parent modules, which was mainly because EDA (Electronics Design Automation) tools automatically adopt optimization strategies, including merging logic, removing unused logic, etc. The tactics of layout also affect the result. So even the same modules have different results.

Although the architecture provides many resources, each algorithm uses just a few of them. In Table 12, we calculated the resource that the algorithms would actually use when executing according to the information in the Appendix figures and Table 11. However, these results were approximate, because, the modules in Others were tiny and we could not obtain the detailed ALUT numbers. For these resources, our calculation method was:

(The number of modules that the algorithm occupies in Others/the total number of modules in Others) * the total number of ALUT of Others.

Table 12. ALUT number occupied by algorithms.

	MD5	SHA1	SHA224/256	SHA384/512	SM3
A_high_com	1419	1626	1419	1419	1689
A_low_com	970	1179	970	970	1248
B_high_com	1397	1605	1397	1397	1637
B_low_com	922	1135	922	922	1167

In this paper, we define resource utilization rate as (resources actually occupied by algorithm when executing)/(total resources provided by architecture) * 100%. If the utilization rate is low, it means that the algorithm only uses a small fraction when executing, although the architecture provides more resources, which means the degree of reconfiguration of the architecture is low, and potential waste exists. We adopted the A_high + A_low + B_high + B_low scheme and calculated the ALUT utilization rate of the compression module. The results are shown in Table 13.

Table 13. ALUT utilization rate of our design.

	MD5	SHA1	SHA224/256	SHA384/512	SM3
ALUT utilization rate	81.14%	95.55%	82.85%	82.85%	98.95%

In Reference [20], the designer did not provide information about the resources the inner modules occupied, so we could not calculate the ALUT utilization rate exactly. However, as we said before in Section 3.1, because the design in Reference [20] just uses the low 32 bits of all computational resources when executing MD5, SHA1, SHA224/256 or SM3, even though the word width of all computational resources are 64 bits, at least 50% of the resources are unused. This means that the ALUT utilization rate is lower than 50% when executing these algorithms. For SHA384/512, it uses the whole 64 bits, and the ALUT utilization rate is only related to the architecture of the design. Although we could do an evaluation due to the limited data, we believe our utilization rate would be higher from the multiplexing degree.

The Register utilization rate mainly refers to the input/output register utilization. We could calculate it directly, in accordance with the algorithm and architecture information, and the results are shown in Table 14.

Table 14. Register utilization rates of our design and of the design in Reference [20].

	MD5	SHA1	SHA224/256	SHA384/512	SM3
Our design	80%	100%	80%	80%	80%
Design in ref. [20]	25%	31.25%	50%	100%	50%

It can be seen from the above data that both the utilization rate of ALUT and Register of our design were not lower than 80%, while the utilization rate in Reference [20] was not higher than 50%, except for the SHA384/512 algorithm. The main reason why our design could achieve a higher utilization rate was our use of the symmetry of the SM3 and SHA2 algorithms to design MD5 and SHA1, so the resources occupied by the SM3 and SHA2 algorithms were fully utilized, greatly improving the multiplexing degree. Furthermore, we eliminated the resource waste introduced by word width. Therefore, the ALUT utilization rate in our design was very high, and especially so for the SM3 algorithm, for which it was as high as 98.95%. In regard to Register utilization rate, our architecture failed to reach 100%. That is because, to enhance the parallelism of SHA1, we set two additional registers. Although other algorithms can also use these two registers, it is not necessary. The essential reason for the high resource utilization rate was that we turned the problem of redundant resources into exploiting greater parallelism, which not only reduced the resource waste, but also greatly improved throughput.

4.3. Throughput

In Section 4.2, we mentioned that the throughput rate was greatly improved due to the greater parallelism. Figure 13 depicts the throughput improvement rate of the design in Reference [20] and of our design, compared with the individual implementation scheme.

Figure 13 indicates that the throughput improvement rate generated by our design was pretty evident. The throughput improvement of our architecture reached 56.87–226%, compared with the individual implementation, except for SHA384/512, let alone the design in Reference [20]. The design in Reference [20] is not a dedicated architecture, therefore the throughput rate was lower than that of the individual implementation. Although our design is also not dedicated, we reused computational resources as much as possible to enhance the parallelism of the architecture processing algorithms. For example, our architecture can support four sets of MD5 messages or four sets of SHA1 messages or two sets of SHA224/256 messages or two sets of SM3 messages or one set of SHA384/512 message. So our throughput was much higher than that of the design in Reference [20] and of the individual implementation. This also explains the reason why the improvement effect of SHA384/512 was not satisfactory, while for other algorithms it was very significant. Please note that our improvement ratio was based on the individual implementation scheme rather than the design in Reference [20].

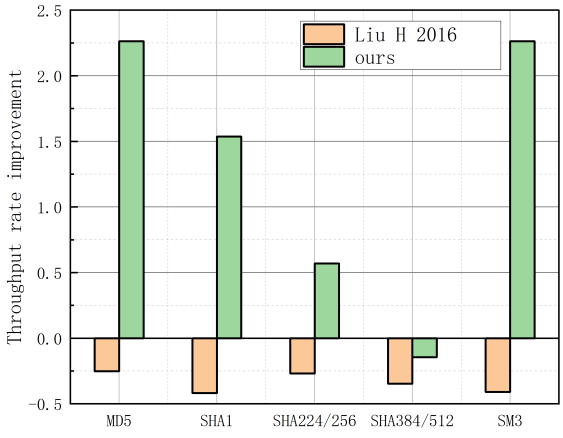
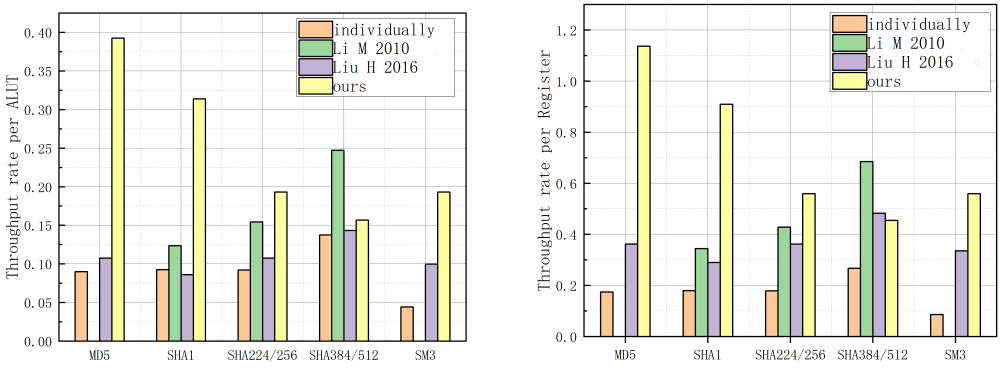


Figure 13. Throughput rate improvement ratio [20].

Computational capability relates to the number of computational resources, so the throughput rate alone cannot objectively evaluate a design. Therefore, we used the throughput rate per resource to measure performance, and the results are shown in Figure 14.



(**a**) (**b**)

Figure 14. (**a**) Throughput rate per ALUT. (**b**) Throughput rate per Register [20,24].

It can be seen that the throughput rate per resource of our design was evidently much higher than those of the other schemes, except SHA384/512. We calculated the improvement of throughput rate per resource of our design compared with other schemes, and the result is shown in Table 15.

The design in Reference [24] had higher resource utilization than ours when executing the SHA384/512 algorithm because it supports fewer algorithms than ours. It is easy to understand that when an architecture supports fewer algorithms, it is closer to a dedicated circuit and, thus, has a higher throughput rate per unit resource. For other algorithms, our architecture achieved better results because our architecture is designed and optimized at both the structural and module levels, so reached up to 5.5 times that of the individual implementation and 2.65 times that of Reference [20] in throughput rate improvement per resource. As we split our architecture into two layers and two sides, each part can run independently and so every computational resource can be fully used without wastage. The architecture had greater parallelism capability, so the improvement was more evident, especially for MD5 and SHA1. For SHA384/512, all the four parts were used cooperatively and, hence, it could not obtain greater parallelism, so the improvement was limited.

Table 15. Register utilization rate.

Comparison Scheme	**MD5**		**SHA1**		**SHA224/256**		**SHA384/512**		**SM3**	
	ALUT	Reg	ALUT	Reg	ALUT	Reg	ALUT	Reg	ALUT	Reg
Ref. [24]	-	-	1.54	1.64	0.25	0.31	−0.36	−0.34	-	-
Ref. [20]	2.65	2.14	2.65	2.14	0.798	0.55	0.09	−0.05	0.94	0.67
individually	3.37	5.5	2.39	4.06	1.1	2.13	0.14	0.7	3.36	5.5

4.4. Array Performance

Maximum parallelism requires three array mapping situations in one row to validate parallelism and mapping principles, as seen in Figure 15. Other mapping cases are similar to this. Since the row structure we designed can meet the resource requirements of all algorithms, and the operator can work separately or cooperatively, all algorithms can be mapped successfully without any operator waste, according to the mapping principle we proposed in Section 3.4, even under the condition of maximum parallelism.

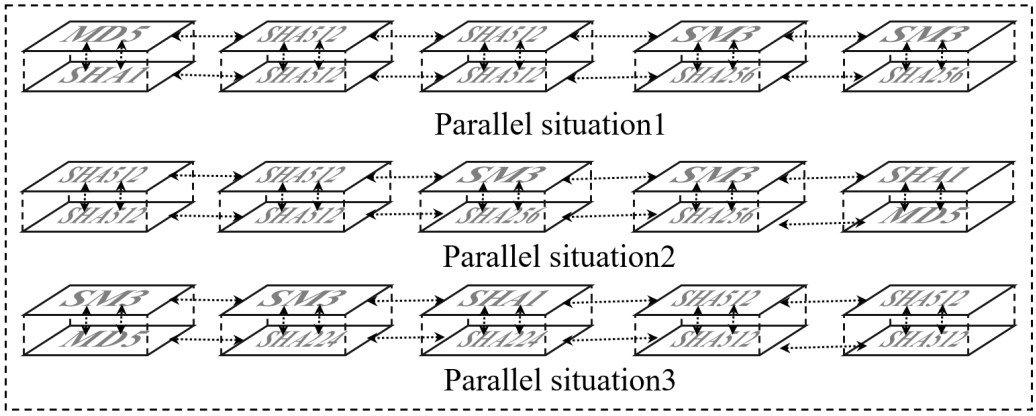


Figure 15. Mapping schemes.

In fact, the pipelined implementation not only results in massive control difficulties and interconnection overheads, but also reduces computing efficiency compared with the non-pipelined implementation, due to the setup and emptying time. Assuming k-level computing stages exist and n computing tasks, the processing time of non-pipelined implementation is:

$$Time\_nonpipeline = \begin{cases} \lfloor n/k \rfloor * k = n & \text{n\%k} == 0 \\ (\lfloor n/k \rfloor * k + 1) * k & \text{n\%k!} = 0 \end{cases} \tag{4}$$

The processing time of pipelined implementation is:

$$Time\_pipeline = n + k - 1 \tag{5}$$

Then

$$\begin{aligned} &Time\_pipeline - Time\_nonpipeline \\ &= \begin{cases} n + k - 1 - n = k - 1, \text{n\%k} == 0 \\ n + k - 1 - (\lfloor n/k \rfloor * k + 1) * k = n - \lfloor n/k \rfloor * k - 1, \text{n\%k!} = 0 \end{cases} \end{aligned} \tag{6}$$

When n%k! $= 0$, $n - \lfloor n/k \rfloor * k \in [1, k-1]$, then $n - \lfloor n/k \rfloor * k - 1 \in [0, k-2]$ and because $k - 1 > 0$, $Time\_pipeline - Time\_nonpipeline \geq 0$ where the equation sign is true when n$-$1%k $==$ 0. Therefore, the processing time with the non-pipelined implementation is shorter. When dealing with the hash algorithms, k = 64 or k = 80. The function curve of the processing time with the number of tasks is shown in Figure 16.

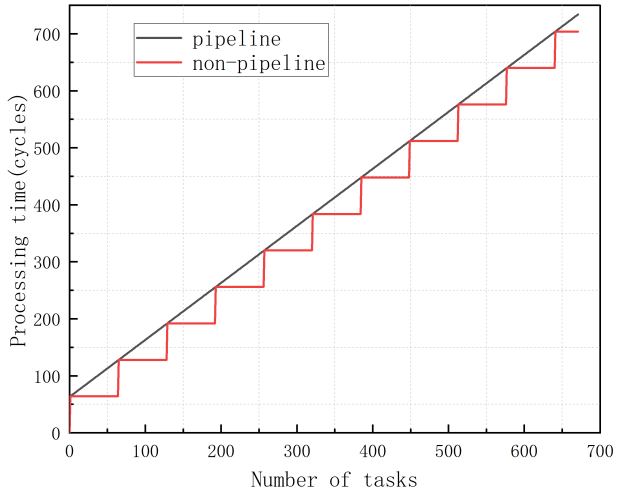


Figure 16. Processing time curve.

The curve when $k = 80$ was similar to $k = 64$. The difference between the two implementation approaches of processing time was due tpo the fact that, when adopting the non-pipelined implementation, the computational resources can be put into work immediately once the computing tasks arrive without waiting for the setup and emptying latency. So, the design approach adopted in our architecture was more efficient.

After the array is designed, the computing capability of the array is enhanced. In this section, we take a 4×5 array as an example. Table 16 gives the maximum throughput rate of a single algorithm. When multiple algorithms run in parallel, the maximum throughput rate can be calculated using Equation (3). When the array provides more computational resources, the computing capability is more powerful.

Table 16. Array computational capability.

Algorithm	Max Throughput Rate (Gbpsz)	Average Time Processing One Message (Cycle)
MD5	34.87	1.6
SHA1	27.9	2
SHA224/256	13.73	4
SHA384/512	11.16	10
SM3	13.73	4

5. Summary

As an important component of information security, improving the processing speed of hashing algorithms to more efficiently deal with booming data scales and rapid data request speeds is a problem that must be solved in modern technology development. To address the problems of imbalance between performance, flexibility, and resources in the existing research on hash function hardware acceleration, we propose a new reconfigurable architecture in this paper. First, we designed hardware architecture based on the characters of the left–right symmetry of SM3 and SHA2, which also supports MD5 and SHA1 on both sides. Secondly, to eliminate the waste of resources due to the different word widths among algorithms, we divided the architecture into two layers to handle high and low data, respectively. Lastly, we decoupled the left and right sides of the architecture, generating four operators, and analyzed the array design method when considering parallelism. The experimental results showed that our design not only supports more algorithms, but also has a throughput rate improvement of about 56.87–226% and a throughput rate per resource improvement of up to 5.5 times. The resource utilization when executing algorithms can reach 80% or above. In future work, we will optimize the timing of our design and explore new methods to generate operators automatically.

Author Contributions: Conceptualization, W.F.; methodology, W.F.; validation, Y.G.; investigation, X.W. and X.Q.; data curation, W.F.; writing—original draft preparation, X.Z.; supervision, X.Q. and Y.G.; project administration, Q.L. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by the National Key R&D Program of China (Grant Nos. 2022YFB4500404).

Data Availability Statement: Not applicable.

Acknowledgments: The authors thank the reviewers for their valuable comments and suggestions.

Conflicts of Interest: The authors declare that they have no known competing financial interests or personal relationships that could have influenced the work reported in this paper.

Appendix A

In this section, we give the data flow diagram when executing the four algorithms.

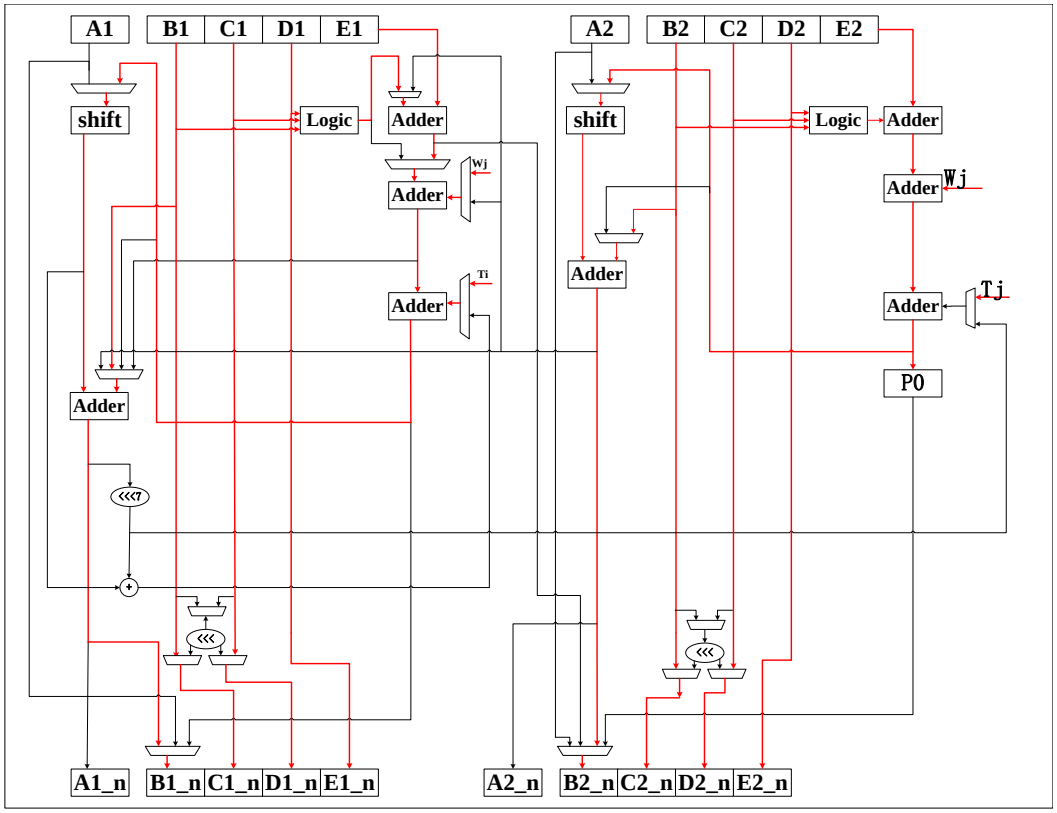


Figure A1. MD 5_flow.

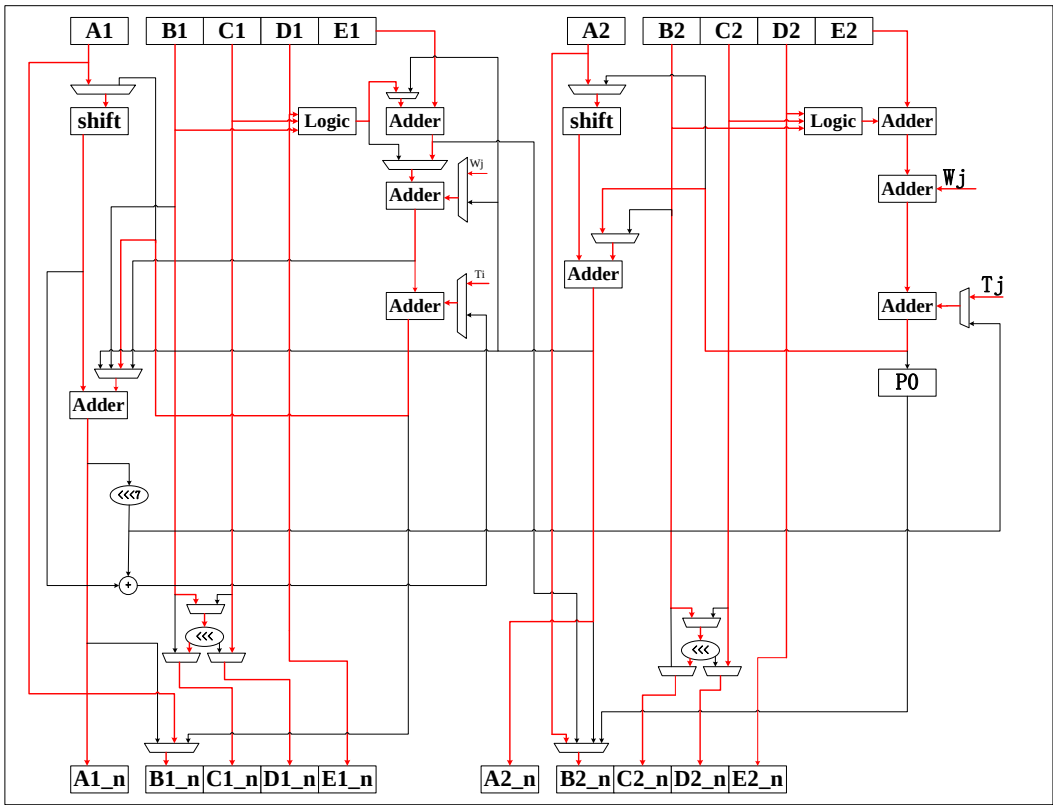


Figure A2. SHA1_flow.

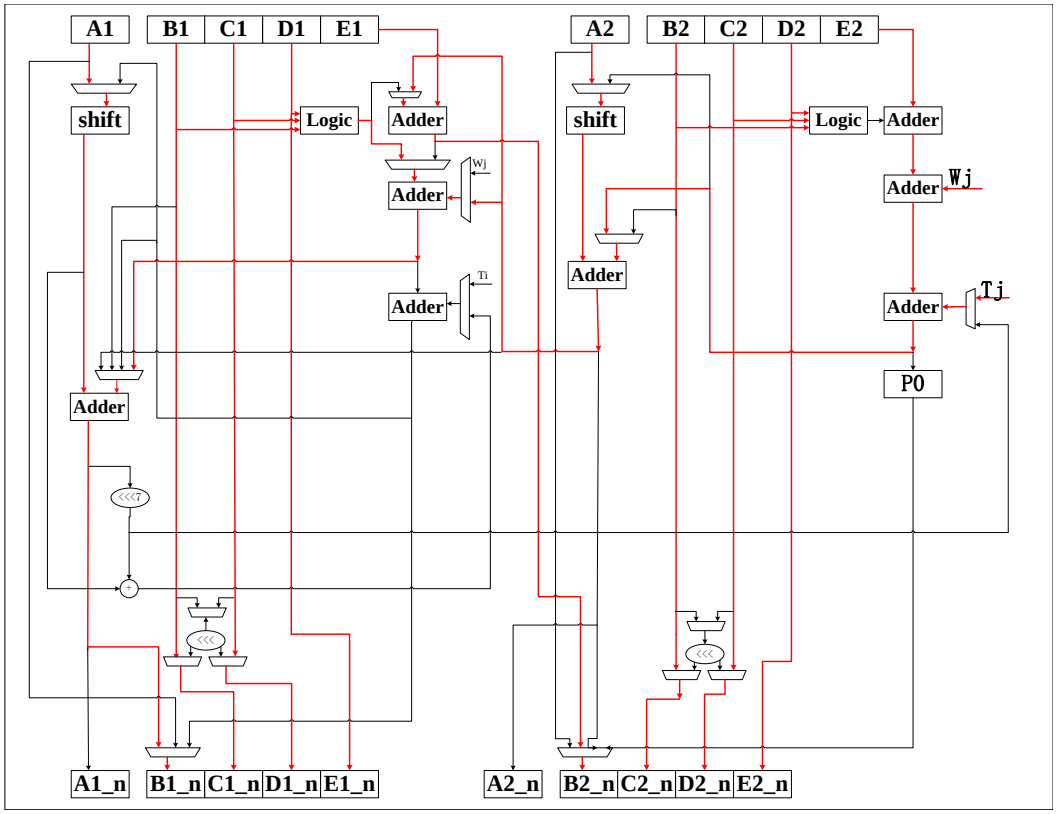


Figure A3. SHA2_flow.

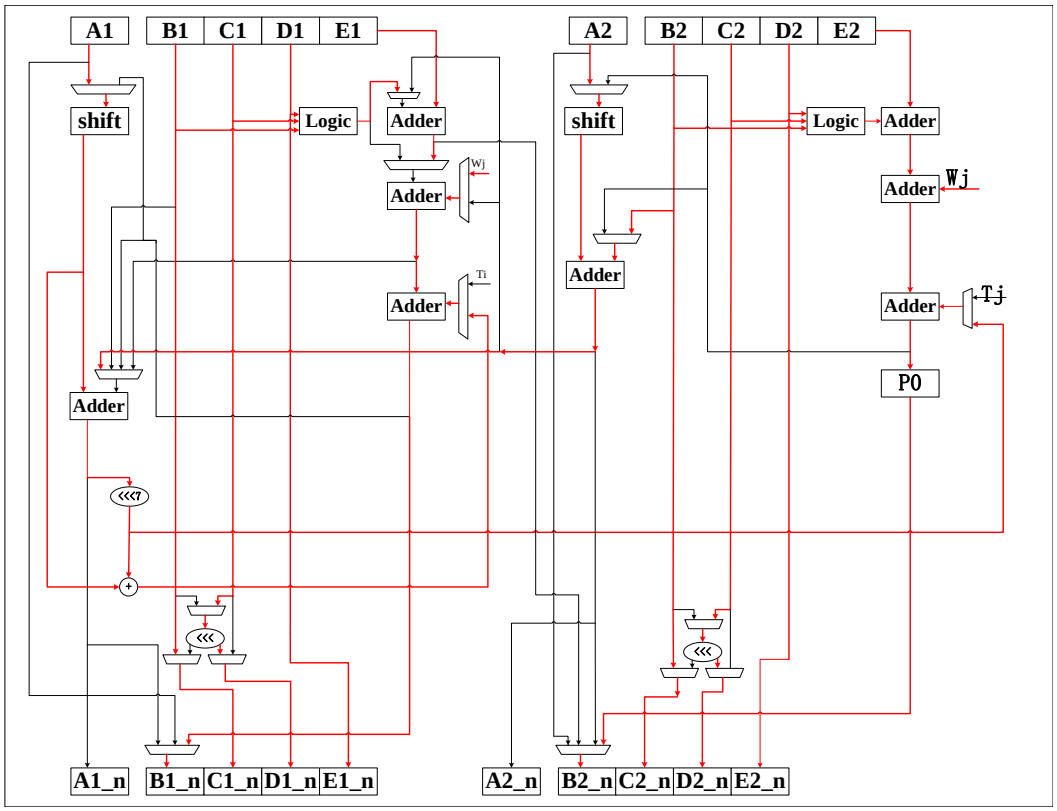


Figure A4. SM3_flow.

References

1. Anwar, M.R.; Apriani, D.; Adianita, I.R. Hash Algorithm in Verification of Certificate Data Integrity and Security. *Aptisi Trans. Technopreneurship* **2021**, *3*, 181–188. [CrossRef]
2. Al-Odat, Z.A.; Ali, M.; Abbas, A. Secure hash algorithms and the corresponding FPGA optimization techniques. *ACM Comput. Surv.* **2020**, *53*, 1–36. [CrossRef]
3. Navamani, T.M. A review on cryptocurrencies security. *J. Appl. Secur. Res.* **2021**, *18*, 49–69. [CrossRef]
4. Lai, Q.Q.;Yang, B.;Chen, Y. Novel construction of identity-based hash proof system based on lattices. *J. Softw.* **2018**, *29*, 1880–1892.
5. Wu, G.C.; Deng, Z.G.; Baleanu, D. New variable-order fractional chaotic systems for fast image encryption. *Chaos Interdiscip. J. Nonlinear Sci.* **2019**, *29*, 083103. [CrossRef] [PubMed]
6. Ma, J.; Huang, X.; Xu, J. Public Accountable Redactable Signature Scheme. *J. Electron. Inf. Technol.* **2020**, *42*, 1079–1086.
7. Abrar, A.; Abdul, W.; Ghouzali, S. Secure image authentication using watermarking and blockchain. *Intell. Autom. Soft Comput.* **2021**, *28*, 577–591. [CrossRef]
8. Xia, Q.; Dou, W.S.; Guo, K.W.; Liang, G.; Zuo, C.; Zhang, F.G. Survey on blockchain consensus protocol. *J. Softw.* **2021**, *32*, 277–299.
9. Wang, L.P.; Guan, Z.; Li, Q.S.; Chen, Z.; Hu, M.S. Survey on Blockchain-based Security Services. *J. Softw.* **2023**, *34*, 1–32.
10. Li, B.; Chu, D.; Lin, J. The Weakest Link of Certificate Transparency: Exploring the TLS/HTTPS Configurations of Third-Party Monitors. In Proceedings of the 2019 18th IEEE International Conference on Trust, Security and Privacy in Computing and Communications on Big Data Science and Engineering (TrustCom/BigDataSE), Rotorua, New Zealand, 5 August 2019.
11. Chen, S.Z.; Kang, S.L. A tutorial on 5G and the progress in China. *Front. Inf. Technol. Electron. Eng.* **2018**, *19*, 309–322. [CrossRef]
12. Kumar, J.; Kumar, M.; Pandey, D.K. Encryption and Authentication of Data Using the IPSEC Protocol. In Proceedings of the Fourth International Conference on Microelectronics, Computing and Communication Systems: MCCS 2019, Ranchi, India, 11 May 2019; pp. 855–862.
13. Yildiz, R.O.; Yilmazer-Metin, A. Design and Implementation of TLS Accelerator. In Proceedings of the 2022 IEEE 15th Dallas Circuit and System Conference, Dallas, TX, USA, 17 June 2022.
14. Wang, Z.D.; Li, N. An Optimized MD5 Algorithm and Hardware Implementation. *J. Hunan Univ. Natural Sci.***2022**, *49*, 106–110.
15. Fang, Y.; Cong, L.H.; Deng, J.Q.; Chen, Z.Y. Fast Implementation of SM3 algorithm based on FPGA. *Comput. Appl. Softw.* **2020**, *37*, 259–262.
16. Miao, J. Hardware Design and Implementation of Secure Hash Algorithms SM3/SHA256/SHA3. Master's Thesis, Tsinghua University, Beijing, China, 2018.
17. Hannig, F.; Derrien, S. Special Issue on Applied Reconfigurable Computing. *J. Signal Process. Syst.* **2022**,*94*, 847–848. [CrossRef]
18. Yuan, H. Research on Key Technologies of Dynamic Reconfigurable Cryptographic Chip. Ph.D. Thesis, Tsinghua University, Beijing, China, 2019.
19. Li, P.J.; Zhang, L.; Xia, Y.F.; Xu, L.M. Architecture design of re-configurable convolutional neural network on software definition. *Chin. J. Netw. Inf. Secur.* **2021**, *7*, 29–36.
20. Liu, H.; Huang, K.; Xiu, S.W.; Li, Y.J.; Yan, X.L. A reconfigurable hardware architecture design for multiple Hash algorithms. *Comput. Eng. Sci.* **2016**, *38*, 411–417.
21. Xi, S.X.; Zhou, Q.L.; Si, X.M.; Li, B.; Tan, J. Optimized implementation of SHA series functions on reconfigurable computing platform. *Appl. Res. Comput.* **2018**, *35*, 2172–2175.
22. Yang, X.H.; Dai, Z.B. Researching and implementation of reconfigurable Hash chip based on FPGA. *J. Syst. Eng. Electron.* **2007**, *18*, 183–187.
23. Zhu, N.L.; Dai, Z.B.; Zhang, L.Z.; Zhan, F. Design and Implementation of Hardware Reconfiguration for SM3 and SHA-2 Hash Function. *Microelectronics* **2015**, *45*, 777–780+784.
24. Li, M.; Xu, J.F.; Dai, Z.B.; Yang, Y.H. Design and Implementation of Reconfigurable Hash Function Cryptographic Chip. *Comput. Eng.* **2010**,*36*, 131–132+136.
25. Liu, L.B.; Wang, B. Anole: A Highly Efficient Dynamically Reconfigurable Crypto-Processor for Symmetric-Key Algorithms. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* **2018**, *37*, 3081–3094. [CrossRef]
26. Du, Y.R.; Li, W.; Dai, Z.B. PVHArray: A Pipeline Variable Hierarchical Reconfigurable Cryptographic Logic Array Structure. *Acta Electron. Sin.* **2020**, *48*, 781–789.
27. Salim, S.I.M.; Soo, Y.; Samsudin, S.I. Application Specific Instruction Set Processor (ASIP) Design in an 8-bit Softcore Microcontroller. *J. Telecommun. Electron. Comput. Eng.* **2018**, *10*, 57–61.
28. Eisenkraemer, G.H.; Moraes, F.G.; Oliveira, L.L.D. Lightweight Cryptographic Instruction Set Extension on Xtensa Processor. In Proceedings of the 2020 IEEE International Symposium on Circuits and Systems, Seville, Spain, 12 October 2020.
29. Yin, J.; Xu, Z.; Fang, X. The design of reconfigurable instruction set processor based on ARM architecture. In Proceedings of the Advanced Computer Architecture: 12th Conference, ACA 2018, Yingkou, China, 10 August 2018; pp. 66–78.
30. Wang, S.S.; Xiao, C.L.; Liu, W.J. Parallel Enumeration of Custom Instructions Based on Multidepth Graph Partitioning. *IEEE Embed. Syst. Lett.* **2018**, *11*, 1–4. [CrossRef]
31. Wei, S.J.; Li, Z.S.; Zhu, J.F.; Liu, L.B. Reconfigurable computing: Toward software defined chips. *Sci. Sin. Informationis* **2020**, *50*, 1407–1426.
32. Hou, P.F. Research and Design of Cipher Specific Instruction Extension on RISC-V Processor. Master's Thesis, PLA Information Engineering University, Zhengzhou, China, 20 June 2018.

33. Wu, D.; Xu, T.G.; Wang, Z.Y.; Liu, J.W. Design of SM3 Hardware Implementation with Integrated Message Padding. *J. Wuhan Univ. Nat. Sci. Ed.* **2019**, *65*, 218–222.
34. Ma, Z.G.; Li, T.T.; Cao, X.X. Design Methodology of SHA2 Hardware Accelerator. *Acta Sci. Nat. Univ. Pekin.* **2022**, *58*, 1007–1014.
35. Ji, Z.X.; Yang, Z.; Sun, Y.; Shan, Y.W. GPU High Speed Implementation of SHA1 in Big Data Environment. *Netinfo Secur.* **2020**, *20*, 75–82.