Quasi-Pipelined Hash Circuits

Marco Macchetti
Politecnico di Milano
Milan, Italy
macchett@elet.polimi.it

Luigi Dadda
ALaRI-USI
Lugano, Switzerland
Politecnico di Milano
Milan, Italy
dadda@alari.ch

Abstract

Hash functions are an important cryptographic primitive. They are used to obtain a fixed-size fingerprint, or hash value, of an arbitrary long message. We focus particularly on the class of dedicated hash functions, whose general construction is presented; the peculiar arrangement of sequential and combinational units makes the application of pipelining techniques to these constructions not trivial. We formalize here an optimization technique called quasi-pipelining, whose goal is to optimize the critical path and thus to increase the clock frequency in dedicated hardware implementations. The SHA-2 algorithm has been previously examined by Dadda et al., with specific versions of quasi-pipelining; here, a full generalization of the technique is presented, along with application to the SHA-1 algorithm. Quasi-pipelining could be as well applied to future hashing algorithms, provided they are designed along the same lines as those of the SHA family.

1 Introduction

1.1 General Considerations

Cryptographic Hash Functions play a crucial role in modern cryptography; a typical usage is to extract hash values, or *fingerprints*, of messages, IP packets, disk files and more in general to condense any binary encoded piece of information into a smaller string. The fingerprint can later be processed more efficiently than the message itself by means of specific algorithms and protocols; for instance, data origin authentication and message integrity can be easily obtained if public-key or symmetric-key encryption is applied to the hash value.

In normal constructions, the source message can have arbitrary length, while the dimension (in bits) of the fingerprint is fixed by the variant of the particular algorithm being used; thus, the fact that different messages can have the same fingerprint is unavoidable. What is qualitatively important for any hash function h is rather that:

1. It is pre-image resistant, i.e. for essentially all specified hash values it is computationally infeasible to find any message that hashes to it.

2. It is second pre-image resistant, i.e. given a message x, it is computationally infeasible to find a different message x' such that $h(x) = h(x')$.

3. It is collision resistant, i.e. it is computationally infeasible to find any two distinct messages that hash to the same value (the hash value being freely choosable).

At the highest level, hash functions can be divided into two classes, *unkeyed* hash functions and *keyed* hash functions. The latter have the same characteristics of the former but take also a secret key k as input parameter; the above three properties are then required to be satisfied for any value of k. Keyed hash functions are also called Message Authentication Codes or MACs. In this paper we will deal only with unkeyed hashing algorithms.

1.2 Known Constructions

Wolfram [22] was the first to suggest the use of cellular automata in cryptography, more precisely to generate a stream of pseudo-random bits starting from a given seed. Daemen et al. have designed *Cellhash* [5], an hash function built from cellular automata. These constructions are generally well suited for hardware implementations because of their regularity and modularity; however, no widely used hash function belongs to this class.

A more common way of designing a cryptographic hash function $h : \{0,1\}^* \rightarrow \{0,1\}^n$ is to first design a compression function $c : \{0,1\}^{m+n} \rightarrow \{0,1\}^n$ that satisfies the given robustness requirements. The compression function

can then be efficiently extended to a robust hash function using the so called Merkle-Dåmgard construction [16].

The message M is first divided into blocks M_i with dimension equal to m bits; at each iteration, each message block M_i is merged with the previous output of the compression function H_{i-1}, which has a dimension of n bits, to obtain a string of $m + n$ bits[1]. The compression function c is then fed with the $m + n$-bit string and calculates the corresponding output H_i. When all message blocks are processed, the last output of the compression function becomes the final hash value, the preceding values being considered *intermediate hash values*.

Since the dimension in bits of the whole message M is not constrained to be a multiple of m, a *padding* mechanism is also defined in order to have an integer number of message blocks. Actually, the importance of padding goes beyond that, because not all padding methods guarantee that the hash function h inherits the robustness of the compression function c; however, a discussion of the different padding methods goes beyond the scope of this paper. The main result of this construction is that the problem of designing good hash functions is reduced to the easier one of designing good compression functions, and for this reason this method is generally adopted.

Different constructions exist for the compression function c, which draw inspiration from basic considerations on how to build a function that is easily computable, but is rather difficult to invert.

A first group is obtained starting from a well-known cryptographic primitive, namely that of a generic symmetric block cipher S; in [15] the 64 most basic ways of constructing a robust compression function from a block cipher were considered. In [2] 20 of the 64 constructions are proven to be secure, all the others leading to weak hash functions. The model used for proving robustness is the so-called *black-box model* [18], in which no detail of the block cipher, apart from the fact that it resembles a random permutation for each value of the key, is used. Hardware and software optimizations for this class of hash functions heavily depend on the details of the employed block cipher S, thus no general design optimization methods have been introduced so far.

Another possible choice is that of designing the compression function from scratch, using operations that are widely available in modern microprocessors such as integer addition and bitwise XOR, and keeping the structure as simple as possible. These constructions are sometimes called *dedicated hash functions*. Several today widely used hash functions, such as MD5, SHA-1 and SHA-2 [1] fall into this category. The common characteristics of these algorithms are:

1. The compression function has an iterated structure, consisting of a loop that updates the value of several internal variables R_i using information contained in the message block M_i being processed. When the total number of internal iterations has been executed, the intermediate hash value H_i is output.

2. Integer sum modulo 2^{32} or 2^{64} is used as an efficient mean to realize non-linearity and diffusion of information among the internal variables, and between the individual bits of each internal variable R_i.

3. Fast, bit-wise logical operators such as XORs, ANDs and cyclic rotations of bits are used and alternated with integer sums during each iteration of the loop.

The alternate application of fast bitwise operators and integer sums ultimately leads to the robustness of the algorithm and to the security of the global hash function, even though no provable security is obtained[2]. The robustness of the SHA-2 algorithm with regards to differential cryptanalysis and other attacking techniques is discussed in [7], where the importance of alternate application of different operators inside the compression function is made evident.

1.3 Existing Design Techniques for Dedicated Hash Functions

Hashing algorithms have been first implemented in software on several platforms. A possible problem is that the speed of computation may not meet the requirements; for this reason focus has recently been put on hardware implementations and specifically on throughput maximization. Most of the efforts concentrate on rapid prototyping platforms, i.e. FPGAs. In [9], [19] and [20] efficient FPGA implementations of MD5, SHA-1 and SHA-2 are presented, and a comparison between the throughput of SHA-1 and SHA-2 is given in [8]. In rapid prototyping platforms, the type of base cells usually puts constraints on the obtainable degree of optimization; thus these techniques somewhat lack general applicability.

Gaj [11] has recently applied loop unrolling to reach high speed of computation in VLSI implementations of the SHA-1 and SHA-2 algorithms. A certain number of iterations of the loop inside the compression function are unrolled and the logic is optimized across the boundaries of different iterations. This has the effect of increasing the area and the speed of the circuit of an amount dependent on the unrolling factor, while it is not always true that the actual area-delay product is lowered.

Dadda, Macchetti, Owen and Chakrabarti [3], [4] have presented high-speed ASIC implementations of the SHA-2 algorithm, obtained through application of *quasi-pipelining*

[1] During the first iteration of the algorithm, a pre-defined constant n-bit value is used as substitute for H_{-1}.

[2] Indeed, the design motivations of the SHA family of algorithms were never even made public.

and *delay balancing*. These techniques could in principle be extended to different algorithms, although no formal generalization of the technique has been given so far.

1.4 Our Contribution

In this paper we concentrate on the class of dedicated hash algorithms; our objective is to define a general design methodology for implementing these functions in dedicated hardware. Our main goal is to reach the highest possible clock operating frequency compatible with a given technology through the application of simple design rules. The effort will lead us to define the *quasi-pipelining* algorithm, a generalization of the pipelining technique motivated by the analysis of hash circuits, but applicable in principle to several other cases (e.g. to feedback registers used in communications and error correcting schemes).

The proposed design algorithm has the benefit of being simple to describe and apply; it takes full advantage of the structural peculiarities of the dedicated hash functions class, and directly outputs the circuital schemes that can be implemented using ASIC libraries with negligible effort. Using the proposed method we obtain the fastest, up to date, implementations of the Secure Hash Algorithms.

We note that the same results may be obtained through alternative methods, for instance by representing one iteration of the compression function under examination as a Data Flow Graph, and applying loop folding techniques. By enumerating all possible scheduling solutions [6], and performing resource allocation, one can obtain solutions similar to the proposed ones.

However, we think this does not affect the validity of the proposed design method, which is of original formulation and extends the circuital schemes presented in [3], [4]. The proposed technique is efficient, and does not imply a design space exploration; it rather directly gives the solution to the implementation problem, and thus scales very well. It may also be useful when future hash algorithms, developed along the same lines that were used for MD5, SHA-1 and SHA-2, will be considered.

In Sect. 2 we present the *quasi-pipelining* algorithm, giving some examples. In Sect. 3 the technique is applied to the SHA-2 and SHA-1 algorithms, the de-facto standards in hashing, and synthesis results are presented. Section 4 concludes the paper.

2 Presentation of the Quasi-Pipelining Technique

2.1 Preliminary Considerations

A common technique for increasing the throughput of electronic circuits is that of pipelining. Pipelining consists of breaking long combinatorial paths by introducing clocked memories; this has the effect of dividing the circuit into *sections*, in which calculations are run independently. The output of a pipeline section becomes the input of the next one at each clock cycle; while generally preserving the global latency, pipelining increases the throughput of a circuit because several instances of the problem are injected at each clock cycle and are processed independently inside each of the sections. The clock pulse is reduced as a result of the breaking of the combinatorial path, thus allowing higher clock frequencies to be used.

Modern microprocessors and DSPs take large advantage of the pipelining technique, which is also applied to arithmetic units. The concept of data-forwarding has also been conveniently adopted [17], consisting of a flow of information between a section of the pipeline and a section which is not immediately the next; as long as the result of a calculation is ready, it can be communicated to all the following sections which may need it to operate efficiently.

The problem of pipelining in presence of feedback paths has been analyzed and treated in the general case by Davidson and Patel [13],[14]; a good survey can be found in [10]. In the following, we will show how the pipelining problem can be tackled in the case of dedicated hash functions; a simple solution, which minimizes the clock period, will be obtained. We stress the fact that we do not apply general scheduling algorithms, nor we perform design space explorations, but rather directly translate the hash specification into the circuit scheme. This results in a more direct and easily applicable design strategy.

Figure 1 is a schematic representation of a generic instance of a compression function belonging to a dedicated hash circuit. In the following we will try, as much as possible, to devise methods applicable to all circuits of this kind, although we will conveniently reference to the circuit of Fig. 1 as a concrete example.

The circuit is essentially made up of a loop that implements the iterative nature of the compression function. The n registers R_i on the left side ($n = 5$ in the Figure) are connected in a shift-register fashion and contain the n internal variables of the loop; at each iteration, or clock cycle, the values of the variables are shifted using a First-In-First-Out (FIFO) approach: the value of R_n is used only as input to some ϕ_i (see explanations below) while the value of R_1 is freshly introduced. The bit-width of the registers is not specified at this point, but usually is assumed to be equal to 32 or 64 bits, since software implementations on modern microprocessors get natural advantage from such configurations.

On the right side we find all the combinatorial elements that calculate the new value to be inserted into the shift register starting from the present values of R_1–R_n. This combinatorial part is made up by:

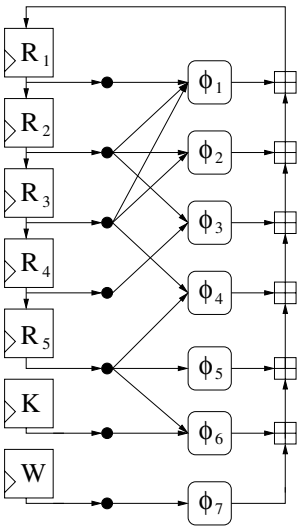

Figure 1. An example of generic compression function.

1. A specified number of arbitrary logical functions ϕ_i, including the identity; each ϕ_i takes a certain number of values from the registers R_i as inputs and produces one output variable.

2. A chain of *combining operators*, each one being any instance of a binary commutative and associative operator.

Modular integer addition is often used as combining operator, so we represent instances of the combining operators with $\boxplus$, the same symbol we use to represent modular integer addition.

On the bottom-left part we also introduce two other variables that may be instantiated multiple times and used by the combinatorial part to calculate the new value for R_1:

1. A table of pre-defined constants K. In the most general configuration, a different constant $K^{(i)}$ is used for each iteration i of the loop, thus the dimension of the table is limited by the total number of iterations inside the compression function. The actual values of the constants depend on the algorithm.

2. A value W that is updated by some other circuit at each iteration of the loop. This part is useful to model some sources of information such as the words of the message block being processed, or values coming from other sub-circuits operating on the message block and that can be usefully separated from the main loop. The value of W at each iteration is denoted with $W^{(i)}$.

Without loss of generality, the values $K^{(i)}$ and $W^{(i)}$ can be thought as coming from registers; this is done in order to clearly define the beginning of the critical path.

Supposing that the complexity of the ϕ_i functions is less than that of a combining operator, the critical path of the circuit in Fig. 1 starts from W and crosses all the combining operators. The goal of pipelining would be to cut this path into smaller ones, separated by the insertion of additional registers. While this is possible to any extent if we consider a combinatorial circuit without feedback loops, the case of dedicated compression functions needs further analysis.

2.2 The Reordering Algorithm

A first task to fulfil is to reorder the combining operators inside the chain. Since every output of the ϕ_i functions is used once in the chain, there is a one-to-one correspondence between the ϕ_i functions and the combining operators; we will refer to the compound of the ϕ_i function and the corresponding combining operator as the ϕ_i *block*.

For every i, we associate an index I_i to the ϕ_i block. The index is nothing but an ordered list of numbers, which are the indexes of the registers R_i whose values are taken as inputs by function ϕ_i; an ascending ordering of numbers is used inside each index[3]. For instance, function ϕ_1 in Fig. 1 is associated with the index $I_1 = \{1, 2, 3\}$.

Since the combining operators obey to the commutative and associative laws, it is always possible to arbitrarily reorder the combining chain. The ϕ_i blocks are moved so that, going top-down in the combining chain, the values of I_i are in ascending order. We say that $I_j > I_i$ if, after a common prefix, the list of numbers of I_j contains a number greater than the corresponding number in I_i; we also say that $I_j > I_i$ if I_i is a prefix of I_j. After this reordering procedure has taken place, we say that the combining chain is *well-ordered*. For instance, the combining chain of Fig. 1 is well-ordered.

2.3 Creation of the Quasi-Pipeline Sections

Every section of the pipelined circuit must be capable of operating independently from the other sections, but in a way to preserve the global functionality of the circuit. We introduce the *quasi-pipeline sections* Q_i; every quasi-pipeline section Q_i contains the set of ϕ_i blocks such that the corresponding indexes I_i have all the same number in the first position of the list. Thanks to the reordering algorithm, all the quasi-pipeline sections are made up by ϕ_i blocks that are consecutive in the combining chain. It is now easy to separate the sections Q_i by introducing additional registers that cut the chain into different sub chains. In Fig. 2 the borders of the quasi-pipeline sections are marked by dashed lines.

[3]The indexes of W and K are always the highest, since they are positioned after all the R_i's.

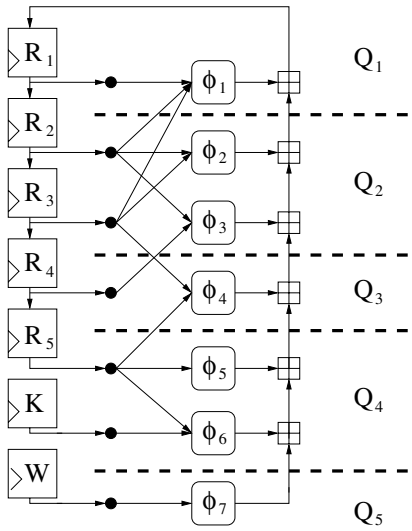

Figure 2. The quasi-pipeline sections of the circuit presented in Fig. 1.

Note that the number of quasi-pipeline sections is upper-bounded by the total number of registers in the left part of the circuit. Additionally, if the delay introduced by ϕ_7 is less or equal to that introduced by ϕ_6, the border between Q_5 and Q_4 can be eliminated without affecting the critical path of the circuit.

2.4 Behaviour of the Quasi-Pipelined Circuit

The quasi-pipeline sections being identified, let us examine the behaviour of the circuit in Fig. 2. We are now in the situation where different iterations of the main loop are executed simultaneously by the circuit, inside the different quasi-pipeline sections; if we suppose to take a snapshot of the circuit at a given instant, the upward crossing of a dashed line in the combinatorial part means now introducing a (positive) delay of one clock cycle, i.e. switching to the immediately preceding iteration of the loop. However, at every iteration, the values in the shift-register are swapped by exactly one position downward. Therefore, it is intuitive that each section must work on a different *time-frame* of $R_1, \ldots, R_n$.

The issue can be made clearer if we reconstruct the first iterations of the circuit in Fig. 2. We will call the border between two quasi-pipeline sections *active* if the border registers are operated, or *inactive* if the registers at the border are kept in a reset state.

The first work-phase of the circuit is the pipeline filling. During this phase the shift register is always maintained in the initial configuration, i.e. registers $R_1, \ldots, R_5$ are not clocked.

Initially, only the border between Q_5 and Q_4 is active; $R_1, \ldots, R_5$ contain their initial values, denoted $R_1^{(0)}, \ldots, R_5^{(0)}$ and W and K output the values $W^{(0)}$ and $K^{(0)}$, respectively.

During the second clock cycle, the border between Q_4 and Q_3 becomes active; W outputs $W^{(1)}$ and K outputs $K^{(0)}$.

During the third clock cycle, the border between Q_3 and Q_2 becomes active; W outputs $W^{(2)}$ and K outputs $K^{(1)}$. Now we face a problem, since function ϕ_4 takes $R_5^{(0)}$ as input, but functions ϕ_5 and ϕ_6, belonging to a different section (i.e. loop iteration), would need to work on the *future* value of R_5, i.e. $R_5^{(1)}$; this value, due to the shift-register configuration[4], is simply $R_4^{(0)}$. Since this is true for all the values coming from the shift register, it is sufficient to swap all the input connections of functions ϕ_5 and ϕ_6 by one position upward to preserve the functionality of the circuit.

The same thing happens for the following iterations, until all the quasi-pipeline sections have become active and the pipeline is full; after that, $R_1, \ldots, R_5$ are clocked at each iteration, and all the input connections of the ϕ_i functions are fixed, i.e. never changed until the last iteration.

One important point is that, given the quasi-pipeline sections defined by the previous steps, the change of input connections for the ϕ_i functions is always possible. In fact, if ϕ_i belongs to the k-th section Q_k, then its input connections must be anticipated by at most $k-1$ positions, since this is the distance between Q_k and Q_1, the first section. Still, since ϕ_i belongs to Q_k, it cannot take as input a value from a register R_i which distances from R_1 less than $k-1$ steps, for otherwise it would belong to a different section.

When all iterations are executed, a phase of *pipeline emptying* is required, to extract the final hash from the shift registers; during this phase, different parts of the shift register are clocked, others being stopped, thus this may be a convenient time to perform the accumulation of the state of the registers into the partial hash register, which is not depicted in the Figures. This may allow for area savings, because a reduced set of adders can operate on the different parts of the shift register individually. The design criteria for this part are similar to what has been said above, and will not be developed in the following discussion due to space limits.

2.5 The Selecting Functions

To efficiently implement the switching of input connections for the ϕ_i functions at different clock cycles, we introduce a layer of *selecting functions* σ_i in the circuit. Figure 3 shows a modified circuit where one σ_i is inserted before

[4] In the most general case of connection of registers, this may be difficult or even impossible to obtain.

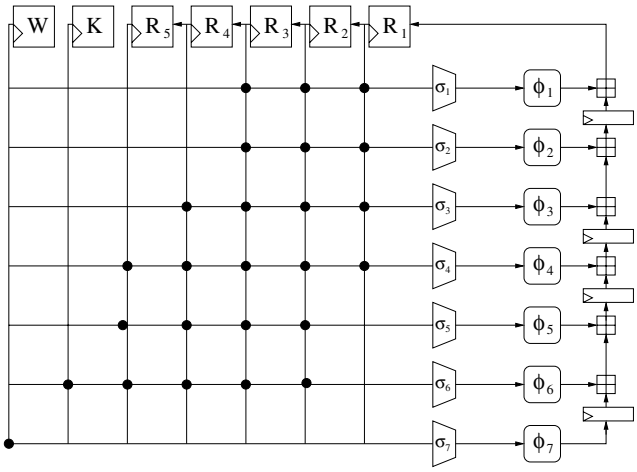

Figure 3. The circuit of Fig. 3 completed with selecting functions.

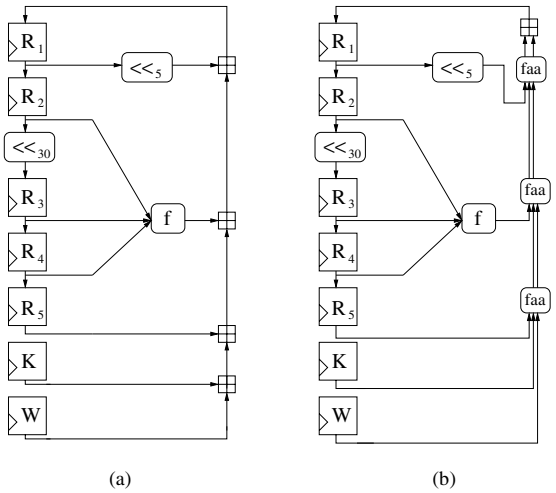

Figure 4. The base circuits for SHA-1.

each ϕ_i. These σ_i select the correct inputs for ϕ_i depending on the iteration number, but in a way absolutely independent from the content of the registers. Each σ_i is composed by standard multiplexers; the select control signals can be internally generated, by means of a counter or similar circuit.

2.6 Concluding Remarks

Once the quasi-pipelined circuit is completed, the critical path is efficiently reduced. We can see from Fig. 3 that no more than 2 combining operators are now inside the critical path of the circuit, instead of the 6 in the initial implementation. Assuming these are the most time-consuming operations, we have an increase of the clock frequency of about 300%. This is obtained at the cost of introducing the additional registers and the multiplexers for the selecting functions. Also, the number of total iterations slightly grows, since 4 more iterations are needed to initially fill the pipeline.

If the combining operators are integer adders, further optimizations are possible since it may be possible, and convenient in terms of the critical path, to substitute the carry-propagating adders in the quasi-pipeline sections with full adder arrays. We will see more about this case in the following Section.

The obtained quasi-pipelined circuit fully implements the initial specification of the hash function, and by no means increases or decreases the security of the algorithm, as this is guaranteed by a careful initial design and specification process.

3 Application to the Secure Hash Standard

3.1 The SHA-1 Algorithm

The Secure Hash Standard is a standard set of dedicated hash algorithms issued by the U.S. National Institute for Standards and Technology (NIST). The first version of the Secure Hash Standard, named SHA-1, produces a hash value of 160 bits, starting from any message with dimension less than 2^{64} bits. The actual security of the scheme, considering birthday attacks [12], is thus equal to 80 bits.

Figure 4a depicts the main loop of the compression function of SHA-1. After the specified number of 80 iterations, the five 32-bit registers $R_1, \ldots, R_5$ contain the final needed value. In the circuit, W represents the output of a message scheduler, or *expander*, whose function is to compute the 80 $W^{(0)}, \ldots, W^{(79)}$ 32-bit words starting from the 16 (32-bit) words composing each message block M_i. The expander does not contain integer adders, and is indeed much simpler than the main loop shown in the Figure; thus, we concentrate on the quasi-pipelining optimization of the latter. The actual values of the constants $K^{(i)}$, along with the initial value of the registers can be found in the specification document [1]. The symbol $<<_n$ denotes cyclic left rotation by an amount of n bit positions. The block f implements different logic functions depending on the iteration number i; more precisely:

$$f(a,b,c) = \begin{cases} (a \wedge b) \oplus (\neg a \wedge c) & 0 \leq i \leq 19 \\ a \oplus b \oplus c & 20 \leq i \leq 39 \\ (a \wedge b) \oplus (a \wedge c) \oplus (b \wedge c) & 40 \leq i \leq 59 \\ a \oplus b \oplus c & 60 \leq i \leq 79 \end{cases}$$

where the symbols $\wedge, \oplus$ denote respectively bit-wise AND and XOR operations and the symbol $\neg$ denotes complementation of all the bits.

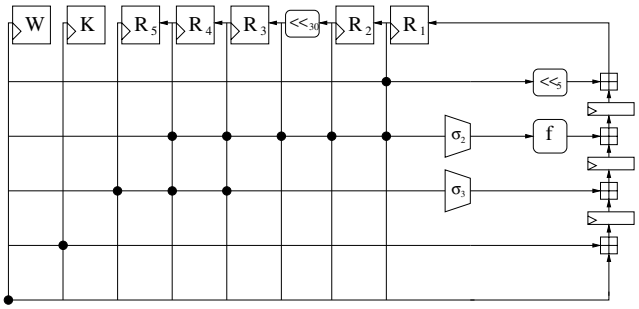

Figure 5. The SHA-1 quasi-pipelined circuit.

Figure 4b shows the fastest possible implementation of the SHA-1 circuit obtained using Full-Adder-Arrays (FAA) to eliminate the unnecessary carry propagations from the chain of adders.

SHA-1 is clearly a particular instance of the hashing circuits examined in the previous sections, the only difference consisting in the cyclic rotation of the bits of R_2 before being assigned to R_3. The straightforward application of the quasi-pipelining optimization technique leads to the following considerations: 1) The chain of adders in Fig. 4a is well-ordered and 2) it is possible to separate all the modular integer adders with quasi-pipeline section borders.

Table 1. Comparison of the different SHA-1 circuits

Circuit	Max clock frequency	Area (μm^2)
Fig. 6a	502 MHz	56529
Fig. 6b	833 MHz	45461
Fig. 7	1.19 GHz	61579

The result is depicted in Fig. 5, where the quasi-pipelined circuit for SHA-1 is shown. The presence of the rotate operation between registers R_2 and R_3 may be a problem since the value of $R_3^{(1)}$ is not equal to $R_2^{(0)}$ but instead is equal to $R_2^{(0)}$ rotated by 30 bit positions to the left; by the way, it is sufficient to introduce connections before and after the rotate operation to produce all the possible inputs for function f. The three circuits of Fig. 4 and Fig. 5 have been described using the VHDL language with the aid of Mentor HDL Designer Pro; they have been also synthesized on the STMicroelectronics HCMOS8 ASIC technology library, featuring 0.18μm silicon process. The results in terms of area (estimated square microns) and maximum clock frequency are summarized in Table 1. It is evident that the introduction of quasi-pipelining leads to clock frequencies not obtainable with standard Carry-Save optimization techniques, at a reasonable area cost.

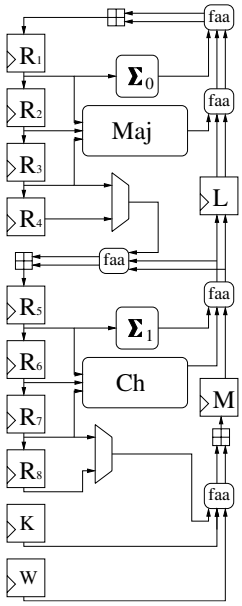

Figure 6. The SHA-2 quasi-pipelined circuit.

3.2 The SHA-2 Algorithm

The SHA-2 algorithm is an evolution of SHA-1, again designed by NIST. The dimension of the final hash value is 256, 384 or 512 bits, depending on the variant of the algorithm being used. A minimum strength of 128 bits is guaranteed when considering birthday attacks [12]. The total number of iterations within the compression function of the SHA-2 algorithms varies from 64 to 80. More details about the SHA-2 family of algorithms, not reported here for space reasons, can be found in [1] and basic circuit schemes can be found in [3] and [4].

In the case of SHA-2, application of the quasi-pipelining technique may not seem straightforward because of the feedback from the addition chain to the shift-register in the middle of the circuit. However, the technique is applicable if we consider that the feedback from the combinational part to the shift-register cuts the circuit into two sections; it is sufficient to apply quasi-pipelining to the two halves of the circuit separately to obtain the scheme in Fig. 6.

This scheme is identical to one reported in [3], where additional considerations on the optimization options are reported. It maybe worthwhile to note that in the scheme of Fig. 6:

1. Carry-propagating adders are substituted with FAAs when this is beneficial from the point of view of the global critical path.

2. The selecting functions have been directly represented with multiplexers and integrated into the scheme, as this is convenient in the specific case.

It is reported [3] that the ratio between the maximum clock frequencies of the circuit in Fig. 6 and a basic scheme obtained with usage of carry-save adders is 1.19 while the ratio between area requirements is 1.13. Again, this confirms that in the case of dedicated hash functions, quasi-pipelining is more powerful than conventional optimization techniques based solely on elimination of redundant propagations of the carries.

4 Conclusions

In this paper we presented a general design optimization method for dedicated hash functions called quasi-pipelining. The method is particularly indicated to enhance the speed of this class of circuits, is of easy and direct formulation, and can be used to effectively translate the function specification into a circuital scheme characterized by the highest clock frequency.

The Secure Hash Algorithms, i.e. SHA-1 and the SHA-2 family, were examined and circuits were given that exceed, in speed of operation, other circuits obtained with standard carry-save optimization techniques.

Further work may include the application of quasi-pipelining to other hash functions, and to other kinds of circuits as well, such as feedback registers used in communication devices and in error detecting schemes.

Acknowledgments

The authors would like to thank the anonymous reviewers for the useful comments.

References

[1] Announcing the secure hash standard. Federal Information Processing Standards Publication 180-2, 2002.

[2] J. Black, P. Rogaway, and T. Shrimpton. Black-box analysis of the block-cipher-based hash-function constructions from pgv. In *Advances in Cryptology: Crypto 2002 Proceedings*, pages 320–335, 2002.

[3] L. Dadda, M. Macchetti, and J. Owen. The design of a high-speed asic unit for the hash function sha-256 (384,512). In *Proceedings of DATE 2004, Vol. 3*, pages 70–75, 2004.

[4] L. Dadda, M. Macchetti, J. Owen, and S. Chakrabarti. An asic design for a high speed implementation of the hash function sha-256 (384, 512). In *Proceedings of GLSVLSI 2004*, pages 421–425, 2004.

[5] J. Daemen, R. Govaerts, and J. Vandewalle. A framework for the design of oneway hash functions including cryptanalysis of damgård's oneway function based on a cellular automaton. In *Advances in Cryptology: Asiacrypt'91 Proceedings*, pages 82–96, 1993.

[6] T. C. Denk and K. K. Parhi. Exhaustive scheduling and retiming of digital signal processing systems. *IEEE Trans. on Circuits and Systems, Part II: Analog and Digital Signal Processing*, 45(7):821–838, 1998.

[7] H. Gilbert and H. Handschuh. Security analysis of sha-256 and sisters. In *Proceedings of SAC 2003*, pages 175–193, 2003.

[8] T. Grembowski, R. Lien, K. Gaj, N. Nguyen, P. Bellows, J. Flidr, T. Lehman, and B. Schott. Comparative analysis of the hardware implementations of hash functions sha-1 and sha-512. In *Proceedings of ISC 2002*, pages 75–89, 2002.

[9] Y. K. Kang, D. W. Kim, T. W. Kwon, and J. R. Choi. An efficient implementation of hash function processor for ipsec. In *Proceedings of the 3rd Asia-Pacific Conference on ASICs*, 2002.

[10] P. M. Kogge. *The Architecture of Pipelined Computers*. McGraw Hill, 1981.

[11] R. Lien, T. Grembowski, and K. Gaj. A 1 gbit/s partially unrolled architecture of hash functions sha-1 and sha-512. In *Topics in Cryptology - CT-RSA 2004 Proceedings*, pages 324–338, 2004.

[12] A. Menezes, P. van Oorschot, and S. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.

[13] J. H. Patel. Improving the throughput of a pipeline by insertion of delays. *ACM SIGARCH Computer Architecture News*, 4(4):159–164, 1976.

[14] J. H. Patel. Pipelines with internal buffers. In *Proceedings of the 5th annual symposium on Computer architecture*, pages 249–254, 1978.

[15] B. Preneel, R. Govaerts, and J. Vandewalle. Hash functions based on block ciphers: a synthetic approach. In *Advances in Cryptology: Crypto '93 Proceedings*, pages 368–378, 1994.

[16] I. D. rd. Design principles for hash functions. In *Advances in Cryptology: Crypto '89 Proceedings*, pages 416–427, 1990.

[17] M. Sami, D. Sciuto, C. Silvano, V. Zaccaria, and R. Zafalon. Exploiting data forwarding to reduce the power budget of vliw embedded processors. In *Proceedings of DATE 2001*, pages 252–257, 2001.

[18] C. Schnorr and S. Vaudenay. The black-box model for cryptographic primitives. *Journal of Cryptology*, 11(2):125–140, 1998.

[19] N. Sklavos and O. Koufopavlou. On the hardware implementations of the sha-2(256,384,512) hash functions. In *Proceedings of ISCAS 2003*, 2003.

[20] K. Ting, S. Yuen, K. Lee, and P. Leong. An fpga based sha-256 processor. In *Proceedings of the FPL 2002 Conference*, pages 577–585, 2002.

[21] X. Wang, D. Feng, X. Lai, and H. Yu. Collisions for hash functions md4, md5, haval-128 and ripemd. In *Rump Session of Crypto 2004*, 2004.

[22] S. Wolfram. Cryptography with cellular automata. In *Advances in Cryptology: Crypto '85 Proceedings*, pages 429–432, 1986.