Multi-Mode Operator for SHA-2 Hash Functions

Ryan Glabb, Laurent Imbert, Graham Jullien, Arnaud Tisserand, Nicolas Veyrat-Charvillon

Available online at www.sciencedirect.com
ScienceDirect
Journal of Systems Architecture 53 (2007) 127–138
JOURNAL OF SYSTEMS ARCHITECTURE
www.elsevier.com/locate/sysarc

Multi-mode operator for SHA-2 hash functions

Ryan Glabb [a], Laurent Imbert [b,a,c], Graham Jullien [a], Arnaud Tisserand [b], Nicolas Veyrat-Charvillon [d,*]

[a] ATIPS Laboratories, Department of Electrical and Computer Engineering, University of Calgary, Calgary, Alberta, Canada T2N 1N4
[b] Arith Group, LIRMM, CNRS – University Montpellier 2, 161 rue Ada, F-34392 Montpellier, France
[c] CISaC, Department of Mathematics and Statistics, University of Calgary, Calgary, Alberta, Canada T2N 1N4
[d] Arénaire Team, LIP (CNRS–ENSL–INRIA–UCBL), ÉNS de Lyon, 46 allée d'Italie, F-69364 Lyon, France

Received 28 April 2006; received in revised form 1 September 2006; accepted 15 September 2006
Available online 1 November 2006

Abstract

We propose an improved implementation of the SHA-2 hash family, with minimal operator latency and reduced hardware requirements. We also propose a high frequency version at the cost of only two cycles of latency per message. Finally we present a multi-mode architecture able to perform either a SHA-384 or SHA-512 hash or to behave as two independent SHA-224 or SHA-256 operators. Such capability adds increased flexibility for applications ranging from a server running multiple streams to independent pseudorandom number generation. We also demonstrate that our architecture achieves a performance comparable to separate implementations while requiring much less hardware.

Keywords: FPGA; Hash function; SHA-2 family; Multi-mode operator

1. Introduction

Cryptographic hash functions [1] are a fundamental tool in modern cryptography, used mainly to ensure the data integrity when transmitting information over insecure channels. Hash functions are also used for the implementation of digital signature algorithms, keyed-hash message authentication codes and in random number generators. Many hash functions exist [2–4], but their actual security level is very difficult to estimate. Whenever weaknesses are found [5], security is compromised and any stand-alone implementations must be phased out leading to costly upgrades toward a new hash function that is deemed secure at that time.

For example, an algorithm has recently been discovered [6] that decreases the resistance to collision of SHA-1 (Secure Hash Algorithm) [7], the most popular hash function so far, reducing the number of necessary computations from 2^{80} to 2^{69} and putting it below the accepted security threshold for high-security operations. Since then, the SHA-2 family of hash functions [8], developed by the National Institute of Standards and Technology (NIST), has become the new standard.

* Corresponding author.
E-mail address: Nicolas.Veyrat-Charvillon@ens-lyon.fr (N. Veyrat-Charvillon).

1383-7621/$ - see front matter
doi:10.1016/j.sysarc.2006.09.006

Due to their complexity and limited lifespan, the cryptographic primitives are generally implemented in software on general purpose processors rather than on specialized hardware architectures. Hardware implementations are also far more expensive and often difficult to realize efficiently. On the other side, software based cryptographic algorithms are much slower than their hardware counterparts by typical factors from 1 to 3 orders of magnitude.

Many secure cryptographic algorithms such as AES (Advanced Encryption Standard) and SHA-1 were designed to be implemented in hardware, and are drastically less efficient when coded in software [1]. In terms of hardware implementations, the two principal approaches are Application-Specific Integrated Circuits (ASIC) technology and Field Programmable Gate Arrays (FPGAs). Due to their ease of use and lower cost, we have chosen FPGAs from the Virtex and Spartan3 Xilinx families for the prototyping phase and synthesis results reported in this paper.

The aim of this work is to show the advantages of using reconfigurable hardware operators to compute various cryptographic primitives associated with the SHA-2 hash functions, using shared resources on a single chip-set.

2. SHA-2 hash standard

Throughout this paper, we will follow the definitions and notations used in the SHA-2 specification [8]. This specification details all steps of the hash algorithms and constants used in the computation. We will only report on the relevant parts useful for the understanding of implementation and optimization issues that are considered in this paper.

The SHA-2 hash standard specifies four secure hash algorithms, SHA-224, SHA-256, SHA-384, and SHA-512. All four of the algorithms are iterative, one-way hash functions that can process a message to produce a hashed representation called a *message digest*. Each algorithm can be described in two stages: preprocessing and hash computation. Preprocessing involves preparing the message through padding, parsing the padded message into m-bit blocks, and setting any initialization values to be used in the hash generation. The hash computation generates a message schedule from the padded message which is used, along with functions, constants and word operations, to iteratively generate a series of hash values. The final hash value generated by the hash computation is used to determine the message digest.

Table 1
Secure hash algorithm characteristics

Algorithm	Word (w)	Message size (l)	Block (m)	Digest	Security
SHA-224	32	$<2^{64}$	512	224	112
SHA-256	32	$<2^{64}$	512	256	128
SHA-384	64	$<2^{128}$	1024	384	192
SHA-512	64	$<2^{128}$	1024	512	256

All sizes are given in bits.

A message M of length l to be hashed is processed by blocks of m bits. Each block is divided in 16 w-bit words for computation, the word-size w depending on the algorithm.

The most important difference between the four algorithms is the size of the message digest. Additionally, the algorithms differ in terms of the size of the blocks and words of data that are used during hashing (Table 1).

2.1. Functions and constants

Each hash function algorithm uses six logical functions, operating on w-bit words, which are represented as x, y, and z. The result of those functions is also a w-bit word. On top of the common $Ch(x,y,z)$ and $Maj(x,y,z)$ functions, four additional functions are defined for SHA-224/256 and SHA-384/512; the XOR operation ($\oplus$) in these functions may be replaced by a bitwise OR and produce identical results. The $ROR_n(x)$ and $SHR_n(x)$ designate rotate right and shift right by n places, respectively

$$Ch(x,y,z) = (x \wedge y) \oplus (\neg x \wedge z)$$

$$Maj(x,y,z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z)$$

2.1.1. Functions

The 32-bit functions used for SHA-224 and SHA-256 are the following:

$$\Sigma_0^{\{256\}}(x) = ROR_2(x) \oplus ROR_{13}(x) \oplus ROR_{22}(x)$$
$$\Sigma_1^{\{256\}}(x) = ROR_6(x) \oplus ROR_{11}(x) \oplus ROR_{25}(x)$$
$$\sigma_0^{\{256\}}(x) = ROR_7(x) \oplus ROR_{18}(x) \oplus SHR_3(x)$$
$$\sigma_1^{\{256\}}(x) = ROR_{17}(x) \oplus ROR_{19}(x) \oplus SHR_{10}(x)$$

The SHA-384 and SHA-512 algorithms use the following 64-bit functions:

$$\Sigma_0^{\{512\}}(x) = ROR_{28}(x) \oplus ROR_{34}(x) \oplus ROR_{39}(x)$$

$$\Sigma_1^{\{512\}}(x) = ROR_{14}(x) \oplus ROR_{18}(x) \oplus ROR_{41}(x)$$

$$\sigma_0^{\{512\}}(x) = ROR_1(x) \oplus ROR_8(x) \oplus SHR_7(x)$$

$$\sigma_1^{\{512\}}(x) = ROR_{19}(x) \oplus ROR_{61}(x) \oplus SHR_6(x)$$

2.1.2. Constants

Sixty-four 32-bit constants are used for the SHA-224 and SHA-256 algorithms, $K_0^{\{256\}}, K_1^{\{256\}}, \ldots, K_{63}^{\{256\}}$. These words represent the first 32 bits of the fractional parts of the cube roots of the first 64 primes.

SHA-384 and SHA-512 use eighty 64-bit constants, $K_0^{\{512\}}, K_1^{\{512\}}, \ldots, K_{79}^{\{512\}}$. These words represent the first 64 bits of the fractional parts of the cube roots of the first eighty primes.

2.2. Preprocessing

This process consists of three steps: (1) padding the message M; (2) cutting the padded message into blocks; and (3) setting the initial hash value, $H^{(0)}$. Its purpose is to ensure that the padded message is a multiple of 512 or 1024 bits.

Let us consider an l-bit message M. A bit '1' is appended, followed by k zero bits. For the SHA-224 and SHA-256 hash functions, k is the smallest, non-negative solution to the equation $l + 1 + k \equiv 448 \pmod{512}$. The 64-bit block representing the total size of the message to be hashed is then appended. The length of the padded message is now a multiple of 512 bits.

For example, the (8-bit ASCII) message "abc" becomes:

$$\underbrace{01100001}_{\text{"a"}}\ \underbrace{01100010}_{\text{"b"}}\ \underbrace{01100011}_{\text{"c"}}\ 1\ \overbrace{0\cdots0}^{423}\ \overbrace{00\cdots011000}^{64}$$

$l = 24$

The operation follows the same scheme for SHA-384 and SHA-512. A 1024-bit padded block is obtained using a 128-bit encoded message length, with k satisfying the equation $l + 1 + k \equiv 896 \pmod{1024}$.

2.3. Parsing the padded message

The padded message is cut into N m-bit blocks, $M^{(1)}, M^{(2)}, \ldots, M^{(N)}$. The m bits of the input block can be expressed as 16 w-bit words for all four hash algorithms. The first w bits of message block i are denoted $M_0^{(i)}$, the next w bits are $M_1^{(i)}$, and so on up to $M_{15}^{(i)}$.

2.4. Setting the initial hash value ($H^{(0)}$)

Before hash computation begins, the initial hash value $H^{(0)}$ is set, which consists of eight w-bit words.

- For SHA-224, $H^{(0)}$ is obtained by taking the 33rd to 64th bits of the fractional parts of the square roots of the ninth through sixteenth prime numbers.
- For SHA-256, we get $H^{(0)}$ by taking the first 32 bits of the fractional parts of the square roots of the first eight prime numbers.
- For SHA-384, the words of $H^{(0)}$ are the first 64 bits of the fractional parts of the square roots of the ninth through sixteenth prime numbers.
- For SHA-512, $H^{(0)}$ consists of eight 64-bit words obtained by taking the first bits of the fractional parts of the square roots of the first eight prime numbers.

2.5. Secure hash algorithms

2.5.1. Hash computation of SHA-256 and SHA-512

We will describe SHA-256 and SHA-512 together, in order to stress their numerous similarities.

We put $Alg = 256$, $w = 32$ and $t_{\max} = 63$ for SHA-256, and $Alg = 512$, $w = 64$ and $t_{\max} = 79$ for SHA-512.

Both algorithms use:

- a message schedule of $t_{\max} + 1$ w-bit words,
- eight working variables of w bits each,
- a hash value of eight w-bit words.

After preprocessing is completed, each message block, $M^{(1)}, \ldots, M^{(N)}$, is processed in order.

Additions (+) are all performed modulo 2^w, and $a \| b$ stands for the concatenation of a and b.

For $i = 1$ to N {

- Prepare the message schedule,

$$W_t = \begin{cases} M_t^{(i)} & 0 \leqslant t \leqslant 15 \\ \sigma_1^{\{Alg\}}(W_{t-2}) + W_{t-7} & \\ +\sigma_0^{\{Alg\}}(W_{t-15}) + W_{t-16} & 16 \leqslant t \leqslant t_{\max} \end{cases}$$

- Initialize the eight working variables, a, b, c, d, e, f, g, and h, with the $(i-1)$st intermediate hash value:(|| denotes concatenation operator)

$$a||b||c||d||e||f||g||h = H^{(i-1)}$$

- For $t = 0$ to $t_{\max}$ rounds, we perform

$$T_1 = h + \Sigma_1^{\{Alg\}}(e) + Ch(e,f,g) + K_t^{\{Alg\}} + W_t$$
$$T_2 = \Sigma_0^{\{Alg\}}(a) + Maj(a,b,c)$$

$$a||\cdots||h = T_1 + T_2||a||b||c||d + T_1||e||f||g$$

- Compute the ith intermediate hash value

$$H^{(i)} = a + H_0^{(i-1)}||\cdots||h + H_7^{(i-1)}$$

}

After repeating those steps a total of N times (after processing $M^{(N)}$), the resulting message digest of M is

$$H_0^{(N)}||H_1^{(N)}||H_2^{(N)}||H_3^{(N)}||H_4^{(N)}||H_5^{(N)}||H_6^{(N)}||H_7^{(N)}$$

The algorithm for SHA-224 is identical to SHA-256, with the exception of using a different initial hash value and truncating the final hash value to its left-most 224 bits

$$H_0^{(N)}||H_1^{(N)}||H_2^{(N)}||H_3^{(N)}||H_4^{(N)}||H_5^{(N)}||H_6^{(N)}$$

Similarly, the SHA-384 algorithm is identical to SHA-512, except for the different initial hash value and truncating of the final hash value to 384 bits

$$H_0^{(N)}||H_1^{(N)}||H_2^{(N)}||H_3^{(N)}||H_4^{(N)}||H_5^{(N)}$$

3. Implementation of the SHA-2 hash functions

In this section, we describe an implementation of the SHA-2 family of algorithms that achieves zero latency; i.e. there are exactly 64 (resp. 80) cycles between the input of the first word in a block and the output of the intermediate hash for a 32-bit (resp. 64-bit) word SHA algorithm. Data is provided along with the first 16 cycles of computation.

This design, besides minimizing the computational overhead, is also very small, mainly because it avoids any unnecessary storage of data. However, its throughput is penalized by having a long critical path. It is, however, possible to achieve competitive results by pre-computing some of the data, and thus reducing the critical path. This improvement requires only a small increase in hardware, and only adds a two cycle latency for the hashing of a whole message.

3.1. General operator architecture

The general architecture, shown in Fig. 1, is a top level representation of the partitioning of the major functional blocks. This architecture can be applied to all hash algorithm modes described in this paper. The operation of each major function is as follows: Fig. 2

- The *control unit* manages all system operations and processes. The control unit's goal is to coordinate new messages and new message blocks in the system and manage relevant functions appropriately.
- The *padder* realizes the message pre-processing, handling all message data to be hashed.
- The *message scheduler* generates the W_t used by the round computation.
- The *round constant unit* holds values of K_t.
- The *round computation unit* updates the a to h variables, given their previous values, K_t and W_t.
- The *intermediate hash* is initialized with each new message and updated at the end of each message block processing.

The operation of the general operator begins when a new message is ready to be hashed. The intermediate hash is then initialized with $H^{(0)}$.

For the first 16 rounds, M_t is transmitted to the message scheduler to provide the first values of W_t. After that, W_t is computed recursively using its previous values W_{t-2}, W_{t-7}, W_{t-15} and W_{t-16}. Along with W_t, the constant K_t is transmitted for each round.

The variables a to h are initialized at the beginning of each new block by the last value of the intermediate hash $H^{(i-1)}$ and updated 64 (resp. 80) times using W_t and K_t. After this, the new intermediate hash value $H^{(i)}$ is produced by adding the a to h variables with each word of $H^{(i-1)}$.

The padder receives its input words via the `WRD_IN` port, and the hash value can be read on port `WRD_OUT` one word at a time using the `H_PART` address port.

3.2. Implementation

In this first implementation of the standalone versions of the SHA-2 hash functions, our goal

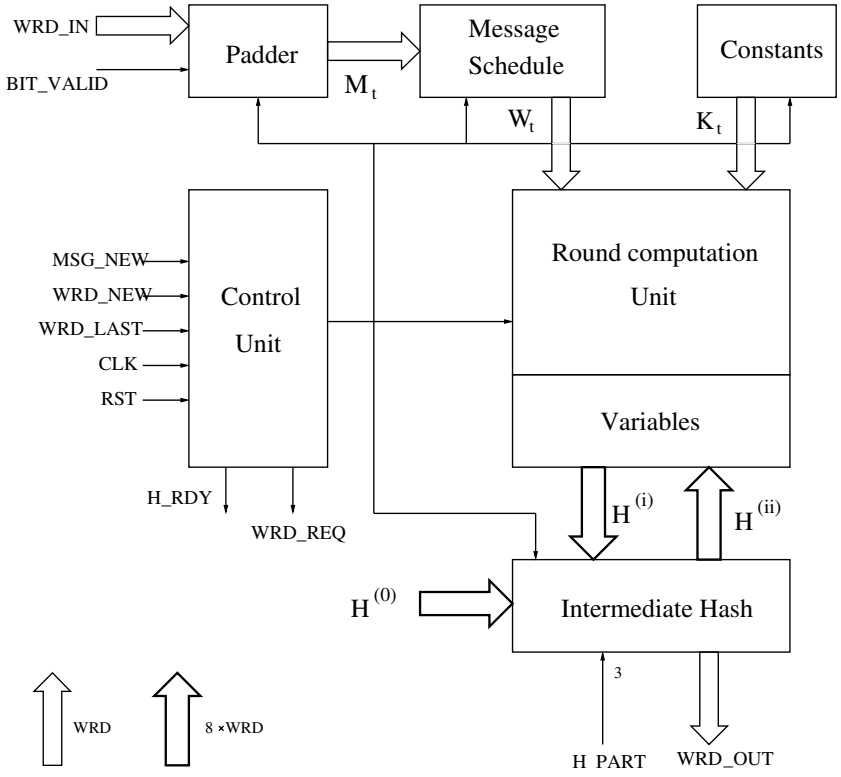

Fig. 1. General structure of the operators.

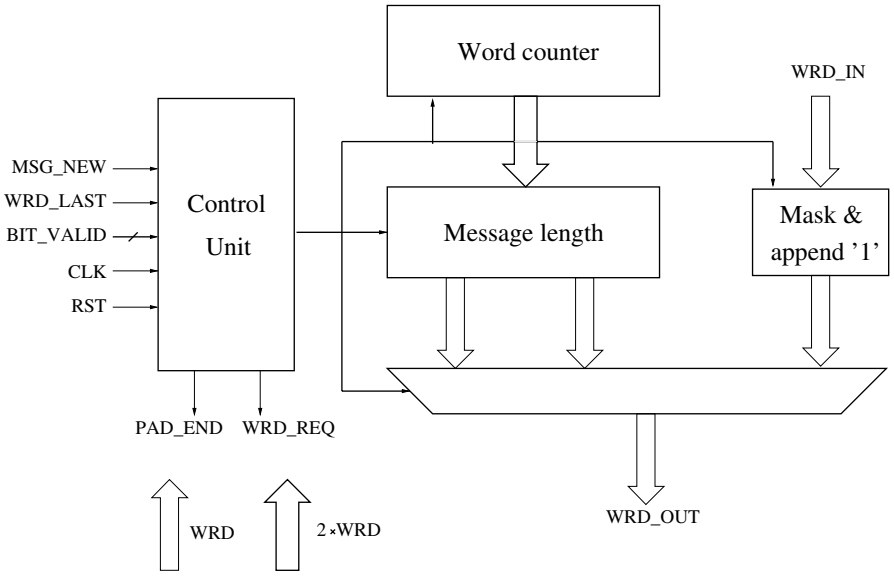

Fig. 2. Implementation of SHA2 padder.

was to minimize the computational overhead, that is to design components with zero latency relative to the hashing of an input message.

As soon as information is available, we begin the computation without waiting for the complete block (or message). While introducing some design issues,

this approach allows us to greatly reduce the hardware cost relative to other existing implementations. This is due to the fact that in exchange for a small control overhead, we remove all unnecessary storage required for buffering.

3.2.1. Padder

In the usual approach, the padder processes the message by blocks, storing a full block, padding as needed and outputting a whole padded block at a time. This is especially true in software where storing and retrieving data are easily handled operations when implementing code on a standard processor architecture. But using such a strategy in hardware implies the use of a block-sized register, and introduces unnecessary latency of, for example, 16 for a serialized input to the padder at each clock cycle.

The reason we can avoid this latency is that, for each computational round, one value of W_t only is required, which is M_t for the 16 first rounds, and does not depend on the padder for subsequent rounds. Thus the padder only has to compute the M_t for the 16 first rounds, and this computation can be done on the fly, as soon as the block words are available.

Our padder processes one word at a time, counting the message length from the first word, given at the same time as the `Message_new` signal, until the `Word_last` signal rises, which indicates the last bit of the actual message (indexed by `Bits_valid`) has been received. Then, the padder appends a bit '1' to the message, followed by as many zeros as needed. The binary-encoded length of the message is finally appended to the message, and the `Message_end` signal is raised.

3.2.2. Message scheduler

From the standard, we know that words W_0 to W_{15} do not have to be processed. Thus, they go directly through the message scheduler (see Fig. 3), and only the subsequent W_t are computed, which depend on 4 out of the 16 preceding values of W_t. Therefore we have to store 16 words containing the previous W_t in order to be able to compute a new message schedule.

3.2.3. Round constants unit

A new round constant must be provided at each round. We implemented this structure using RAM blocks instead of ROM blocks because of the FPGA target architectures we used. The RAM blocks provide 512 32-bit storage locations. Only one of these blocks was needed, including for the SHA-384/512 hashes, because of their dual-port capability.

The 1-cycle latency of the RAM blocks is accounted for in the control part of the architecture.

3.2.4. Round computation unit and intermediate hash

The round computation unit, given W_t, K_t and $a, b, \ldots, h$, computes the next value of the a to h variables using the equations provided in the SHA-2 standard. This computation is performed

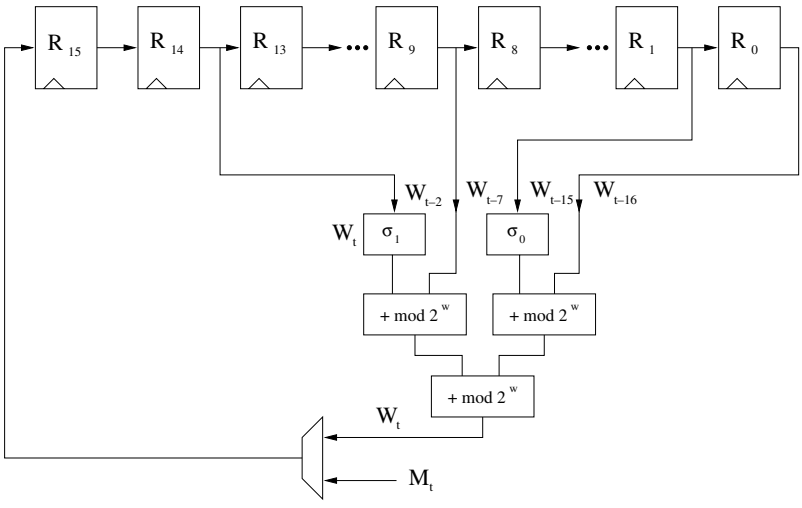

Fig. 3. Implementation of message scheduler.

using a standard tree of carry-propagate adders utilizing the fast-carry adders provided on our target FPGA.

The variables are initialized at the beginning of a new message block with the intermediate hash, and updated every subsequent round with the output of the *round computation unit*.

The intermediate hash is initialized to $H^{(0)}$ with each new message and updated by adding the variables $a, b, \ldots, h$ to its words after each processed block.

3.2.5. Analysis

By computing M_t and W_t on the fly, the hashing operator is able to begin as soon as the message words are provided. It is then possible to achieve zero latency in the computation of intermediate hash results. Therefore, for a SHA-224/256 hashing (resp. SHA-384/512), the hashing of an N blocks message will take $64 \times N$ (resp. $80 \times N$) cycles exactly, including the initialization required by a new message.

The other advantage of this approach is that by computing all data as soon as it is provided, we remove all message buffering, therefore realizing important hardware savings compared to the usual approach where computation only begins following the input of a complete block, adding at least 16 cycles of latency per block and an extra block-wide register to the design.

3.3. Merging of SHA-224/256, and SHA-384/512

The only differences between SHA-224 and SHA-256 are in the values used for $H^{(0)}$ and the number of bits in the digest that are used.

These two differences do not impact on the hardware cost: the values for $H^{(0)}$ are "random" sequences of bits in each case, and the truncating of the output does not reduce the required hardware since the whole hash intermediate value is needed for each block computation.

The same statements hold for SHA-384 and SHA-512 which also require almost the same hardware resources.

We add two extra blocks into our architecture: SHA-224/256 and SHA-384/512 which have an additional input `Alt_Hash` used to choose between the variants. The only hardware overhead in these components, compared to the separate implementations, is a MUX used to select the suitable value of $H^{(0)}$.

4. Optimization

A synthesis of the previously described operators shows that their critical path is quite long, leading to small speed (Fig. 4). In order to increase the operating frequency, the computational process is modified to split the critical path into three clock cycles (Fig. 5). This leads to increased performances in

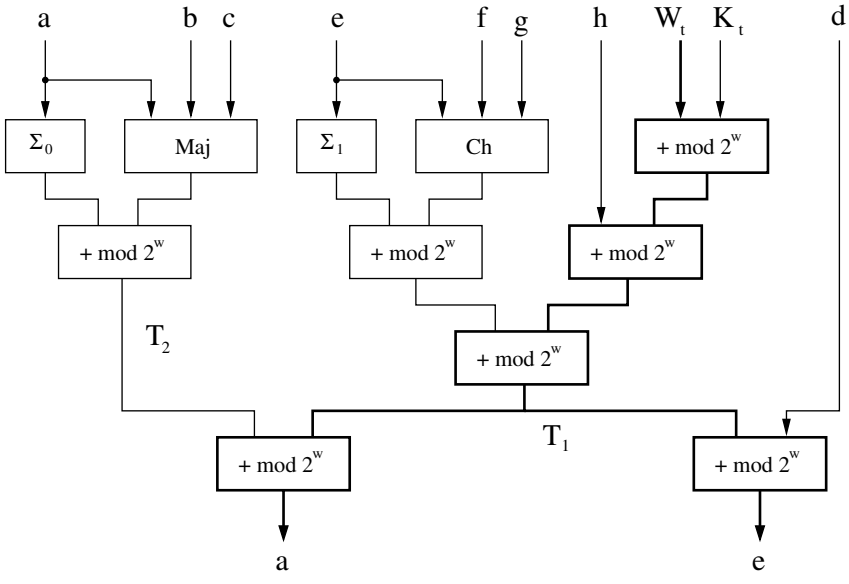

Fig. 4. Original critical path in SHA2 implementation (in thick).

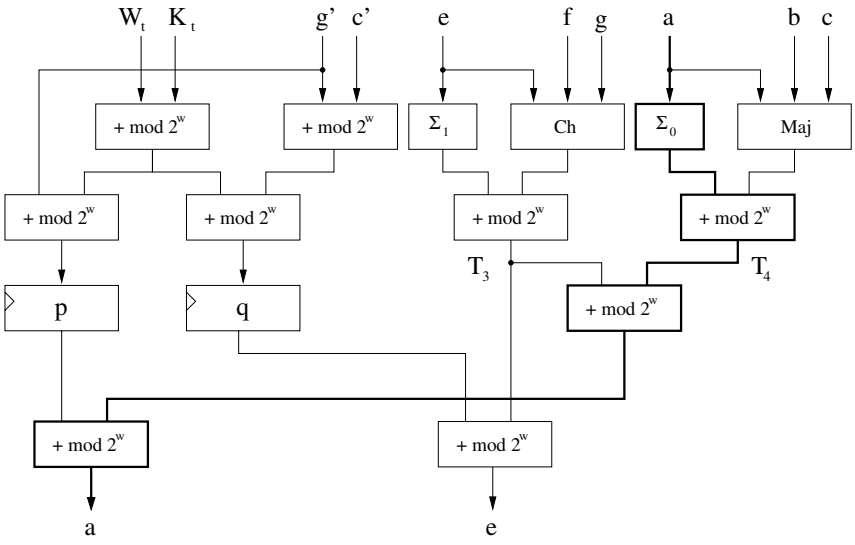

Fig. 5. Critical path for the optimized SHA2 implementation (in thick).

terms of speed at the cost of only a 2 cycles latency per message and a small hardware overhead.

4.1. Determination of the critical path

Synthesis results using Synplify Pro show that the critical path runs from the padder, through the message scheduler and the round computation unit (via the evaluation of T_1) to the storage of the new intermediate hash value. We can trade-off a reduction in critical path, and hence increase the throughput rate, at the cost of an increase in latency, as we discuss next.

4.2. Segmenting the computation delay

We can reduce the length of the critical path by computing some of the intermediate results used for the round computation during the previous rounds. For example, W_t can be stored in a register in order to reduce the critical path after the padder and message scheduler. This results in an increase of one cycle of latency for the message computation, since M_t will be delayed once before being used. This part of the optimization requires no extra hardware since W_{t-1} has to be stored in the message scheduler anyway. The value of K_t can similarly be delayed by one cycle by acting on the address counter feeding the ROM block.

As can be seen from the round computation equations, variables c, d and g, h at round t are equal (except for the initialization round at $t = 0$) to b, c and f, g respectively at round $t - 1$. No precomputation will involve a and e (resp. b and f) since their values result from computations involving the previous values of a and e at round t (resp. $t - 1$).

Then $d + h + K_t^{\{512\}} + W_t$ and $h + K_t^{\{512\}} + W_t$ can be pre-computed efficiently at round $t - 1$.

We introduce

$$c' = \begin{cases} H_3^{(i-1)} & \text{if } t = 0 \\ c & \text{else} \end{cases}, \quad g' = \begin{cases} H_7^{(i-1)} & \text{if } t = 0 \\ g & \text{else} \end{cases}$$

$$p = g' + K_{t+1}^{\{512\}} + W_{t+1}$$

$$q = c' + g' + K_{t+1}^{\{512\}} + W_{t+1}$$

$$T_3 = \Sigma_1^{\{512\}}(e) + Ch(e, f, g)$$

$$T_4 = \Sigma_0^{\{512\}}(a) + Maj(a, b, c)$$

One can then compute: $e = q + T_3$ and $a = p + T_3 + T_4$.

The original critical path is now cut following the padder and message scheduler operations by reading W_{t-1} from the registers of the message scheduler instead of using the combinatorial value W_t at round t. It is also cut by the pre-computation of p and q as they can be used directly with T_3 and T_4 to compute a and e.

4.3. Analysis

The critical path reduction described above introduce two cycles of latency: one is due to the delaying

of W_t, and the other to the computation of p and q. No extra hardware is required for W_t, since W_{t-1} is stored in the message scheduler. The t address used for K_t has to be delayed (requiring 7 flip-flops), and p and q are stored in memory at each round (2 w-bit registers). Some extra logic and routing is also used in the computation of c' and g'.

The hashing of an N-block message will now take $2 + 64 \times N$ (resp. $2 + 80 \times N$) cycles for a SHA-224/256 hashing (resp. SHA-384/512). The synthesis results show that the hardware overhead is low compared to the speed improvement. The shorter critical path actually allowed smaller operators to be synthesized.

5. Merging of the SHA-2 family

Merging the SHA-2 family of functions into a single architecture is more efficient than implementing separate operators for each hash algorithm. For example, in [9], SHA-256, SHA-384 and SHA-512 were each implemented using a separate computational unit. During the computation of SHA-256, that implementation does not use the left half of the 64-bit datapath, and it is held to zero.

Our multi-mode SHA-2 operator has been designed to optimize the hardware efficiency. It is able to run either a hash function working on $w = 64$-bit words (SHA-384 or 512), or two $w = 32$-bit functions (SHA-224 or 256) running concurrently. When running in split mode, the operator can be considered as two separate operators each running a $w = 32$-bit hash.

5.1. Sharing the datapath

5.1.1. Comparison between the hash functions

The hash functions of the SHA-2 family share many similarities. We can classify them into two categories: the $w = 32$ bit functions, SHA-224 and SHA-256, and the $w = 64$ bit functions, SHA-384 and SHA-512. Given their respective word sizes, a large part of the datapaths are identical, and other parts can be shared efficiently:

- The padding is identical with regard to the respective word sizes. A message of length l is processed by blocks of 16 words, and a "1" is appended at the end, followed by as many zeros (k) as necessary in order to have $l + 1 + k \equiv 14w \pmod{16w}$. A 2-word binary representation of l is then appended.
- The message scheduler is identical for all hash functions, except for the σ functions which depend on the word size.
- The definition of the initial hash value $H^{(0)}$ allows its implementation to be shared between the algorithms. That is, the left halves of the SHA-512 words of $H^{(0)}$ are the words of $H^{(0)}$ for SHA-256. Similarly for SHA-384, the right halves of the words of $H^{(0)}$ are the words for $H^{(0)}$ for SHA-224. For example for $H_0^{(0)}$:
  SHA-256 $H_0^{(0)} =$ **6a09e667**
  SHA-512 $H_0^{(0)} =$ **6a09e667**f3bbc908
  and
  SHA-224 $H_0^{(0)} =$ **c1059ed8**
  SHA-384 $H_0^{(0)} =$ cbbb9d5d**c1059ed8**
- In the functions defined by the SHA-2 standard, only *Ch* and *Maj* are identical for all algorithms. The σ and Σ operations are different, although they are based on the same idea, that is a bitwise XOR of three different rotations/shifts of the input value, but the rotate/shift values differ and thus cannot be shared. Since there are only two different sets of functions (one for $w = 32$ and another for $w = 64$), they are both hardwired with selection between the two using a MUX, which is a lower hardware cost solution than the use of a generic structure (barrel rotate/shifter).
- The round constants are the same for equal word sizes, and the value of K_t for $w = 32$ is identical to the left half of the corresponding $w = 64$ constant. For example:
  SHA-224/256 $K_0 =$ **428a2f98**
  SHA-384/512 $K_0 =$ **428a2f98d**728ae22
- The round computation and the intermediate value definitions are the same for all SHA-2 algorithms, although the number of rounds differs depending on the word size. Only 64 rounds are performed for $w = 32$-bit hashes, and 80 for $w = 64$-bit hashes.

5.2. Physical sharing of the hardware

Our multi-mode architecture fits into the same datapath two 32-bit words hash functions and a single 64-bit hash. We note α and β the two 32-bit word hash functions that use the left and right halves of the datapath, respectively, and γ the 64-bit hash that uses the whole datapath.

The physical sharing is accomplished by considering all operations realized for γ on 64-bit words

as two separate 32-bit operations for α and β, by inhibiting dependencies between the two halves in the latter case.For registers and parallel operations, this involves no hardware overhead since the left and right halves are independent regardless of the operator mode. When an addition modulo 2^w is performed, a carry propagation exists between the right and left parts of the 64-bit words for γ that must be inhibited when computing two adjacent modulo 2^{32} additions for α and β. Beside the small logic overhead, control parts are duplicated in order to allow α and β to run concurrently as well as in parallel.

Fig. 6 shows the modifications required to the standard carry propagate adder, available on the FPGA, that allow either one modulo 2^{64} addition or two concurrent modulo 2^{32} additions to be performed.

5.2.1. Padder

In the multi-mode version of the padder, the word counter has been modified in order for it to be used as either two separate 64-bit counters, or as a single 128-bit counter. This implies a rather complicated management of the carry since the last 4-bits (resp. 5-bits) of each message length for a $w = 32$-bit (resp. $w = 64$-bit) hash are given by the input `Bit_valid`. If the operator works in split mode, one carry used in the word counter must be discarded and `Bit_valid` used for the lower bits in the message length of α.

5.2.2. Message scheduler

The message scheduling for SHA-256 and SHA-512 is the same except for: (1) the word size which is doubled; (2) the σ_0 and σ_1 functions which consist of

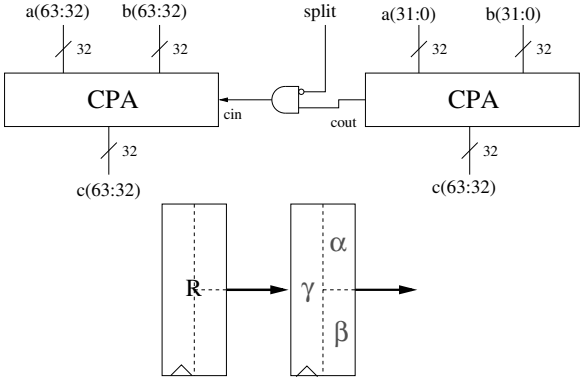

Fig. 6. We introduce a 64-bit/2 × 32-bit selectable modular adder (up). The registers can either be considered as one 64-bit or as two concurrent 32-bit registers (down).

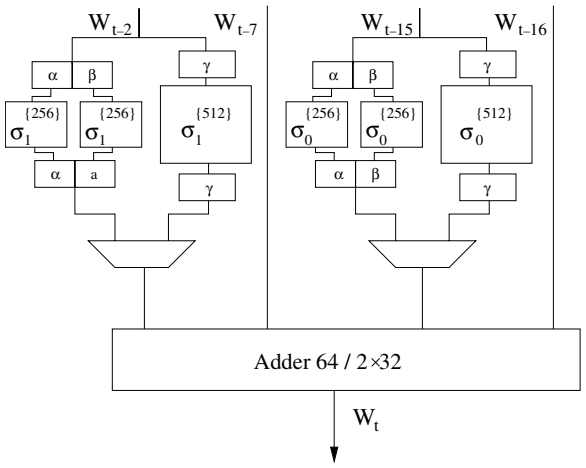

Fig. 7. Multi-mode implementation of the W_t computation.

wiring; and (3) the number of rounds that does not affect the logical structure of the scheduler. Fig. 7 illustrates the multi-mode computation of W_t. MUXes select data paths for each mode, and the previously introduced split adders are used to perform the modulo 2^w additions.

5.2.3. Round constants unit

Since a dual-port 32-bit RAM block is used to compute the SHA-384/512 64-bit round constants, it can also be used, at the cost of some logic overhead in the address input, to provide two different 32-bit round constants as well as two concurrent SHA-244/256 hashes.

In order to ensure the same latency properties as in the separate architectures, some logic has to be added to ensure the correct initialization of the computation when the mode is changed, since the constant unit must output either $K_{0\gamma}$ or $K_{0\alpha} \| K_{0\beta}$ depending on the new mode.

5.2.4. Round computation unit and intermediate hash

The equations for computing the new values of variables $a, b, \ldots, h$ are the same for all hash functions of the SHA-2 family, with appropriate changes relating to the relative word sizes and with the exception of the Σ functions. The only modification of the round computation unit for the multi-mode version therefore consists in using the split adders and implementing both Σ^{512} and Σ^{256} operators for each Σ function, in the same manner that was used for the message scheduler.

The initial hash value, $H^{(0)}$, is selected through additional logic that takes advantage of the similar

values in the different algorithms. The computation of a new intermediate hash is performed using the split adders.

5.2.5. Analysis

The multi-mode architecture shares the same properties as separate architectures for the operators in terms of latency and speed. It is also possible to improve the speed performances with the same segmentation of the critical path.

6. Implementation results

This section summarizes our implementation results using Synplify Pro as the synthesis tool. The criteria considered are FPGA resources (slices), maximum throughput (Mb/s) and their ratio in Mb/s/slice. We synthesized our design using two targets: the Spartan3-XC3S400 which is the most suitable for our architectures and the Virtex 200/400XCV which we used for providing accurate comparisons with existing schemes.

Every hash function of the SHA-2 family was synthesized as a stand-alone operator (224, 256, 384 and 512), or merged by word operating size (224/256 or 384/512), and we also give our results for the multi-mode architecture which is capable of all SHA-2 family modes and can act as two independent 32-bit (SHA-224/256) operators simultaneously.

From a system-level perspective the decision to support the merged or multi-mode operator is made before synthesis. Once implemented, control signals determine the mode of operation. Any of these designs can be synthesized with either two cycles of latency operation for increased clock frequency or zero latency operation with a longer critical path.

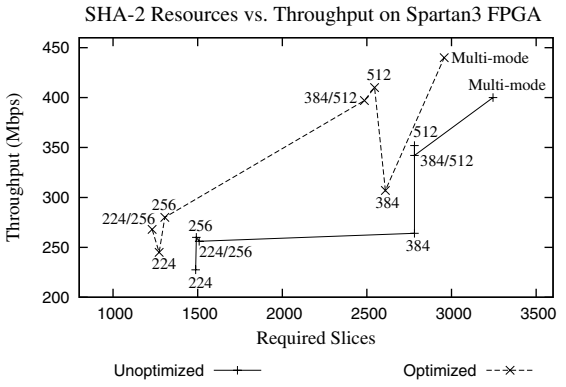

Fig. 8. Throughput vs Area for Spartan3 optimized for speed. All architectures are presented with their zero-latency (continuous) and optimized two-latency (dashed) variants.

6.1. Analysis of proposed architecture

It is apparent in Fig. 8 that the merged versions of our operator provide no overhead in terms of resources for both SHA-224/256 and SHA-384/512. In terms of speed, the optimized critical path provides an average 10% throughput improvement across all modes and also reduces the average number of required slices by 3%. This results in a large improvement for the speed-to-area ratios, especially with regard to our new multi-mode architecture.

Table 2
FPGA Synthesis results and comparison

	Slices	Freq. (MHz)	Cycles per block	Throughput (Mb/s)	Throughput/area (Mb/s/slice)
Reference architecture					
[9] SHA-256	*2120	83	81	262	0.123
[9] SHA-384	*3932	74	97	293	0.075
[9] SHA-512	*4474	75	97	396	0.089
[10] SHA-384/512	*5828	38	Pipelined	479	0.082
[9] SHA-256/384/512	*4768	74	81/97	233.9/390.6	0.049/0.082
Proposed architectures					
SHA-224	1297	77	64(+2)	269.5	0.208
SHA-256	1306	77	64(+2)	308	0.236
SHA-224/256	1260	69	64(+2)	276	0.219
SHA-384	2581	69	80(+2)	331	0.128
SHA-512	2545	69	80(+2)	442	0.174
SHA-384/512	2573	66	80(+2)	422	0.164
**Multi-mode SHA-2	2951	50	64/80(+2)	$2 \times 200/320$	0.136/0.108

* 1 CLB = 2 slices for Virtex, target: Virtex 200/400XCV, ** Max multi-mode throughput is 400/640 Mbps = $2 \times 200/320$ Mbps.

6.2. Comparison with published implementations

We now compare our architectures with previously published stand-alone and multi-mode SHA-2 implementations [9,10] (see Table 2).

The focus of [9] was to implement SHA-256, 384 and 512 in a single operator using the Virtex XCV200 as a target. Ref. [10] discusses a pipelined approach to a single chip SHA-384/512 architecture. Another implementation of the SHA-512 function can be found in [11]. In all of these designs, 16–32 clock cycles are required for the padder to process an input message block before computation begins. This is avoided in our system thanks to an 'on-the-fly' padder that allows a clock cycle reduction of up to 25% compared to [10].

Additionally, due to some pre-computation techniques, we are able to achieve clock frequencies higher than [10] and slightly less than [9] while at the same time significantly reducing the hardware costs.

Our multi-mode operator, in particular, uses considerably fewer resources compared to the multi-mode 256/384/512 implementation [9] with 2951 slices compared to 4768 slices and has a much better throughput to area ratio.

7. Conclusion

In this paper, we have introduced a concurrent SHA-2 operator which optimizes the datapath when a 64-bit SHA-2 hash mode is supported and removes all unnecessary latencies. The proposed multi-mode architecture is able to perform a single SHA-384 or SHA-512 hash function or to behave as two independent computations of SHA-224 or SHA-256 hash functions with minimal hardware overhead. We demonstrated the benefit of integrating a concurrent 32-bit mode when a 64-bit hash is to be supported.

Additionally, the new architecture achieves a performance comparable to previously published separate implementations of these functions while requiring much less hardware. Most importantly, all of the new implementations presented in this paper are more efficient than previously published implementations when considering the throughput-to-area ratio.

Acknowledgements

This work was financially supported through iCORE (Informatics Circle of Research Excellence), NSERC (Natural Sciences and Engineering Research Council of Canada), CMC and an ACI grant from the French ministry of Research and Education.

References

[1] Alfred J. Menezes, Paul C. van Oorschot, Scott A. Vanstone, Handbook of Applied Cryptography, CRC Press, 1997.
[2] Ronald L. Rivest, The MD5 Message-Digest Algorithm, Internet informational RFC 1321, April 1992.
[3] H. Dobbertin, A. Bosselaers, B. Preneel, RIPEMD-160: a strengthened version of RIPEMD, in: IWFSE: International Workshop on Fast Software Encryption, LNCS, 1996.
[4] Vincent Rijmen, Paulo S.L.M. Barreto, The WHIRLPOOL hash function, World-Wide Web document, 2001.
[5] Xiaoyun Wang, Dengguo Feng, Xuejia Lai, Hongbo Yu, Collisions for hash functions MD4, MD5, HAVAL-128 and RIPEMD, Cryptology ePrint Archive, Report 2004/199, 2004, p. 4.
[6] Xiaoyun Wang, Yiqun Lisa Yin, Hongbo Yu, Finding collisions in the full SHA-1, Shandong University, Technical Report, June 2005.
[7] National Institute of Standards and Technology, FIPS PUB 180-1: Secure Hash Standard, Gaithersburg, MD, USA, NIST, April 1995.
[8] National Institute of Standards and Technology, FIPS PUB 180-2: Secure Hash Standard, Gaithersburg, MD, USA, NIST, august 2002.
[9] N. Sklavos, O. Koufopavlou, The Journal of Supercomputing 31 (3) (2005) 227–248.
[10] M. McLoone, J.V. McCanny, Efficient single-chip implementation of SHA-384 & SHA-512, in: IEEE Proceedings of the International Conference on Field-Programmable Technology (FTP), 2002, pp. 311–314.
[11] Grembowski, Lien, Gaj, Nguyen, Bellows, Flidr, Lehman, Schott, Comparative analysis of the hardware implementations of hash functions SHA-1 and SHA-512, in: ISW: International Workshop on Information Security, LNCS 2002.