Design Space Exploration of a Reconfigurable HMAC-Hash Unit

Esam Khan
The Custodian of the Two Holy Mosques Institute for Hajj Research, Umm Al-Qura University, Makkah 6287, Saudi Arabia (e-mail: esamkhan@uqu.edu.sa)

M. Watheq El-Kharashi
Department of Computer and Systems Engineering, Ain Shams University, Cairo 11517, Egypt (e-mail: watheq@ece.uvic.ca)

Fayez Gebali
Department of Electrical and Computer Engineering, University of Victoria, Victoria, BC V8W 3P6, Canada (e-mail: fayez@ece.uvic.ca)

Mostafa Abd-El-Barr
Department of Information Science, College for Women, Kuwait University, Safat 13060, Kuwait (e-mail: mostafa@cfw.kuniv.edu)

In this paper, a design space exploration of a reconfigurable HMAC-hash unit is discussed. This unit implements one of six standard hash algorithms; namely, MD5, SHA-1, RIPEMD-160, HMAC-MD5, HMAC-SHA-1, and HMAC-RIPEMD-160. The design space exploration of this unit is done using the Handel-C language. We propose a key reuse mechanism for successive messages in order to improve the HMAC throughput. In addition, we explore the design space by providing two implementations of the HMAC algorithm, one for a general key size and another for a fixed key size. In each implementation, we incorporate standard key use and the proposed key reuse mechanisms, which results in four different implementations. The performance of these four implementations is analyzed with respect to four design metrics: area, delay, throughput, and power consumption. We found that the proposed key reuse mechanism improves the HMAC throughput significantly, with negligible increase in area and delay, when a large key is reused. In addition, we found that the implementation of HMAC for fixed key size is better in area, delay, throughput, and power consumption than the HMAC implementation for general key size.

Keywords: Authentication, design space exploration, hash functions, HMAC, IPSec.
ACM Classification: B.5.2, C.5, C.0, C.4

1. INTRODUCTION

One of the standard algorithms used for authentication is the Keyed-Hash Message Authentication Code (HMAC) (NIST, 2002a). HMAC is a shared-key algorithm that uses hash functions for authentication. There exists a number of standardized authentication algorithms using HMAC. The most popular ones are HMAC-MD5 (Madson and Glenn, 1998a), HMAC-SHA-1 (Madson and Glenn, 1998b), and HMAC-RIPEMD-160 (Keromytis and Provos, 2000).

Manuscript received: 20 January 2007
Communicating Editor: Zoran Salcic

In a previous work (Khan *et al*, 2007), we designed a reconfigurable unit that implements the HMAC algorithm. This hash unit was designed based on a unified algorithm that implements three hash functions, MD5, SHA-1, and RIPEMD-160. The integration of the HMAC unit and the hash unit resulted in a unified, reconfigurable, HMAC-hash unit that can implement six standard hash functions; namely, MD5, SHA-1, RIPEMD-160, HMAC-MD5, HMAC-SHA-1, and HMAC-RIPEMD-160.

The HMAC unit described by Khan *et al* (2007) implements a general version of the HMAC algorithm, where a key of any length is allowed. In this paper, we implement another version of the HMAC unit that uses fixed key size. In addition, we propose a key reuse mechanism to improve the HMAC performance when a key is reused for successive messages. Each version of the HMAC implementations (general and fixed key size) is implemented with and without the key reuse mechanism. This results in four different implementations of the HMAC unit.

To build the HMAC-hash unit, we used an emerging design methodology that is based on Handel-C (Celoxica Limited, 2003; Aubury *et al*, 1996), which is a high level language based on ANSI-C (Khan *et al*, 2006). Handel-C is designed to enable direct compilation of a design from high level descriptions to FPGA logic or RTL descriptions. Furthermore, it has some constructs that enable design space exploration before a final design decision is made.

There exists a number of works that implemented hardware units for HMAC. Most of these works focus on HMAC-SHA-1 (McLoone and McCanny, 2002; Selimis *et al*, 2003). Some others implemented HMAC-MD5 and HMACSHA-1 (Haa *et al*, 2004; Lu and Lockwood, 2005; Wang *et al*, 2004). However, there is no work that implemented HMAC with three hash functions on the same hardware core as our HMAC-hash unit. In addition, to our knowledge, HMAC-RIPEMD-160 has not been addressed before for FPGA designs. Overall, most of these works used VHDL to build the HMAC units and did not discuss the design space of their units.

In this paper, we discuss the design space exploration we conducted to design the HMAC-hash unit. Most of the design options are explored using available Handel-C constructs. Moreover, the key size and key reuse options are also considered in the design space exploration of the HMAC-hash unit. We present the four different implementations of the HMAC unit mentioned above and compare their performance.

This paper is organized as follows. Section 2 provides some background material. The design space exploration of the HMAC-hash unit conducted using Handel-C is discussed in Section 3. The other two design options, which are the key size and the proposed key reuse, are described in Sections 4 and 5, respectively. In Section 6, the performance of the four different implementations of the HMAC unit is analyzed. Our work is compared to others in Section 7. Finally, we conclude the paper in Section 8.

2. BACKGROUND MATERIAL

In this section, the background material required to understand the work of this paper is given. This includes a brief description of the HMAC algorithm, an overview of the HMAC-hash unit, and a brief discussion of the experimental tools we used.

2.1 The HMAC Algorithm

HMAC is a shared-key security algorithm that uses hash functions for authentication. Its strength is based on the strength of the underlying hash function. The details of the HMAC algorithm are given by NIST (2002a) and Krawczyk *et al* (1997).

HMAC gets a message of arbitrary length `M` and produces a fixed length output `MAC`. It uses a secret key K and an un-keyed hash function h to compute the MAC. The main operation of the HMAC algorithm is given by the following equation:

$$\mathrm{MAC(M)} = \mathrm{Trunc}_t(\mathrm{h}((\mathrm{K}_0 \oplus \mathrm{opad}) \,@\, \mathrm{h}((\mathrm{K}_0 \oplus \mathrm{ipad}) \,@\, \mathrm{M}))).$$

This equation and the symbols used in it are explained in the following algorithm.

Declarations:

- I_b: Number of bytes of the input to the hash function.
- O_b: Number of bytes of the output from the hash function.
- ipad: The constant byte 0x36 repeated I_b times.
- opad: The constant byte 0x5C repeated I_b times.
- K: Shared secret key.
- K_b: Number of bytes of K.
- K_0 : K after being resized to I_b bytes.
- M: The message to be hashed.
- MAC: The hashed value of M (output from the HMAC algorithm).
- h(m): The hash value resulted from hashing a message m using the hash function h.
- t: The number of bits to truncate the MAC to.
- V1 to V6 are temporary variables.

Notation:

- $\oplus$: XOR operation.
- @: Concatenation operation.
- $V\,@\,(0x00)_n$: Padding of n bytes of 0's to the right of variable V.
- $\mathrm{Trunc}_n(V)$: Truncation of variable V to the n leftmost bits.

The algorithm:

1. Resize the key K to K_0:

$$K_0 = \begin{cases} K@(0x00)_{I_b - K_b}, & \text{if } K_b < I_b \\ K, & \text{if } K_b = I_b \\ h(K)@(0x00)_{I_b - O_b}, & \text{if } K_b > I_b \end{cases}$$

2. $V1 = K_0 \oplus \mathrm{ipad}$
3. $V2 = V1 \,@\, M$
4. $V3 = h(V2)$
5. $V4 = K_0 \oplus \mathrm{opad}$
6. $V5 = V4 \,@\, V3$
7. $V6 = h(V5)$
8. $\mathrm{MAC} = \mathrm{Trunc}_t(V6), O_b/2 \le t \le O_b$.

2.2 The HMAC-Hash Unit

The HMAC-hash unit, proposed by Khan *et al* (2007), gets a message, hashes it using one of six hash functions: MD5, SHA-1, RIPEMD-160, HMAC-MD5, HMAC-SHA-1, and HMAC-RIPEMD-160, and produces the resulting hash value. Figure 1 shows a block diagram of this HMAC-hash unit.

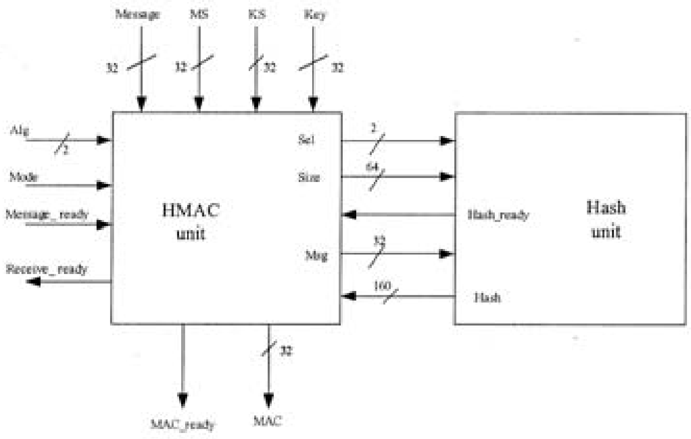


Figure 1: Block diagram of the HMAC-Hash unit.

We designed the HMAC-hash unit as two units: an HMAC unit and a hash unit. The HMAC unit receives the message size (`MS`), key size (`KS`), and then the message and the key in 512-bit blocks, one block at a time. It passes each block to the hash unit. The hash unit hashes the message (or the key, if required) using the unified hash algorithm discussed by Khan *et al* (2007). Signals `Mode` and `Alg` are used to reconfigure the HMAC unit and select one hash function, as shown in Table 1.

The port `MAC` is used to output the result. The output size of the HMAC unit is 160 bits. If the output is to be truncated (as described in the HMAC algorithm in Section 2.1), the most significant bits of the output are selected as required. Since truncation of the output does not affect the performance of the HMAC unit, we assume that the entity communicating with the HMAC-hash unit has the capability of selecting the required number of output bits.

We assume that there is a handshaking mechanism between the HMAC-hash unit and the entity communicating with it. The signals `Message_ready`, `Receive_ready`, `Hash_ready`, and `MAC_ready` are used to facilitate this handshaking process.

The hash unit gets a two-bit input, `Sel`, from the HMAC unit, which is a copy of the signal `Alg`. This signal selects the hash function to be used. The hash unit also gets the message size, `Size`, and

Mode	Alg	Selected hash function
0	00	HMAC-MD5 (RFC 2403 (Madson and Glenn, 1998a))
0	01	HMAC-SHA-1 (RFC 2404 (Madson and Glenn, 1998b))
0	10	HMAC-RIPEMD-160 (RFC 2857 (Keromytis and Provos, 2000))
0	11	Not used
1	00	MD5 (RFC 1321 (Rivest, 1992))
1	01	SHA-1 (RFC 3174 (Eastlake, 2001))
1	10	RIPEMD-160 (Dobbertin *et al*, 1996)
1	11	Not used

Table 1: Reconfigurability of the HMAC unit

the message, `Msg`, one 512-bit block at a time. If the `Mode` bit is zero and the key needs to be hashed, the key is sent to the hash unit as a regular message.

2.3 Handel-C and DK

Handel-C[1] (Celoxica Limited, 2003; Aubury *et al*, 1996) is a high-level language based on ANSI-C. It is designed to enable direct compilation of a design from high-level descriptions to FPGA logic or RTL descriptions. It allows designers to import algorithms written in C and use them to create a rapid implementation and optimization path in hardware (Celoxica Limited, 2002). The advantage of Handel-C over ANSI-C is that it is fully synthesizable and supports parallelism. Most of the constructs available in ANSI-C are also available in Handel-C. In addition, Handel-C adds some constructs that enable hardware design with direct mapping to FPGA (Jussel, 2005).

DK design suite is an Integrated Development Environment (IDE) produced by Celoxica (Celoxica Limited, 2005). It provides a complete design flow, from system specification to hardware implementation. Using DK, one can compile, simulate, and debug a project written in Handel-C and synthesize it directly to FPGA chips. DK includes a number of automatic optimization options that help a designer optimize his/her code before implementing it into hardware. These options include dead-code elimination, common sub-expression elimination, and retiming (Sutcliffe, 2004). DK can also map the code to available ALUs on the target device and estimate device utilization and logic depth using the Logic Estimator. It generates files that show the approximate area consumption of every line of the source code and files that show the lines of code that contribute to the longest path delay. This latter property allows the designer to optimize the code before synthesis.

2.4 Xilinx ISE

To synthesize the designed unit and analyze its performance, we used the Xilinx ISE tools (Xilinx, 2007). These tools acquire HDL or EDIF files as input and generate FPGA-configurable bit files along with some post-place-and-route files that are used for analyzing and optimizing the design. For example, the Timing Analyzer can be used to analyze timing and XPower can be used to analyze power consumption.

3. DESIGN SPACE EXPLORATION USING HANDEL-C

For any hardware design, there are different implementation options, which determine its design space. Design space exploration finds out how a particular design option affects the performance of the design in order to have a trade-off between different performance metrics (Kienhuis, 1999).

Figure 2 depicts the different design options that we considered for our HMAC-hash unit. Available Handel-C constructs enable conducting a design space exploration in order to get the most optimal results. In the following subsections, we discuss the design options implemented using different Handel-C constructs. These options include storage, redundancy, path delay, conditions, loops, and register size. In Sections 4 and 5, we discuss the key size and key reuse options, respectively.

3.1 Storage

Storage can be done in different ways. The default is to use registers. In Handel-C, this is done by declaring a variable or an array of variables. Similarly, declaring a constant or an array of constants maps to register(s) in hardware. Registers can be accessed as much as needed in parallel and they are, in general, faster than memory. However, registers require more FPGA LUTs than memory.

[1] Handel-C is a trademark of Celoxica Limited.

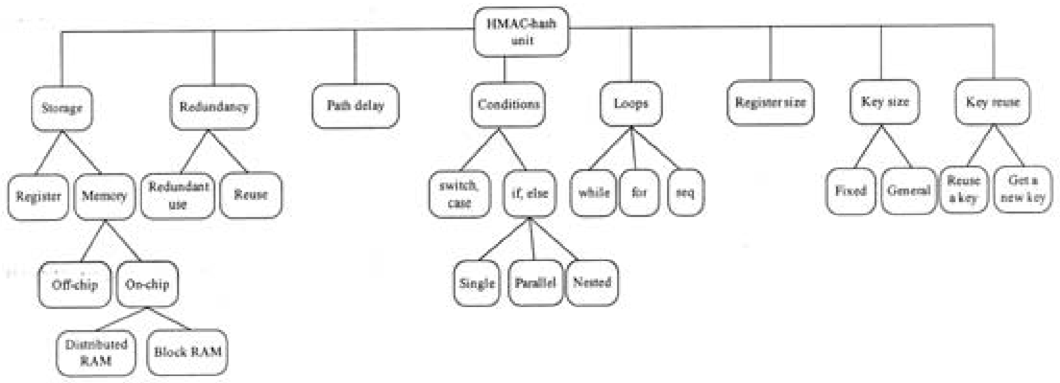


Figure 2: Design space tree for our HMAC-hash unit.

Memory is an alternative option used for storage. Unlike registers, only one element of memory maybe accessed in a single clock cycle. Multi-ported RAMs (MPRAMs) can be used to access more than one memory port in a single clock cycle. In Handel-C, RAM can be used instead of variables and ROM can be used instead of constants. There are two options for memory, off-chip and on-chip. Because on-chip memory is generally faster than off-chip memory, we decided to use on-chip memory.

There are two types of on-chip memory: block RAM and distributed RAM. Block RAM does not consume any LUTs, but it consumes a lot of memory cells. Distributed RAM does not consume memory cells, but consumes a number of LUTs. In addition, using block RAM leads to a performance slower than that resulting from using distributed RAM.

In order to have a balance between area and speed, we decided to use distributed RAM for variables that need to be accessed once in a clock cycle. For example, we used distributed RAM for the constants used for the algorithm initialization. If we need parallelism, we either use dual ported distributed RAM or registers. Dual ported distributed RAM is used when we need to access an element twice in parallel. As an example, the resized key K_0 of the HMAC algorithm is stored in a dual ported distributed RAM. Registers are used when we need more than two parallel accesses to the same element. The `Alg` signal is an example of a variable declared as a register.

3.2 Redundancy

The concept of redundancy applies to functions and registers. For functions, redundancy allows a function to be used more than once at a time, which helps with parallelism. However, this needs more area because each call to a function builds a replicated hardware copy of that function. On the other hand, reusing a function, which is called *shared function* in Handel-C, saves area. However, shared functions cannot be used more than once at a time (Sutcliffe, 2004).

The typical way of implementing redundant functions in Handel-C is to define a function and precede it with the keyword *inline* to have a replicated copy of the function whenever it is invoked. Another, but unusual, way is to distribute the required operation in all statements requiring this functionality instead of having one function of that operation and calling it from these statements. We noticed that sometimes, it is better with respect to area to have the required operation embedded in the same statement rather than defining a function. For example, for converting a variable to the little endian format, we can either declare a function and pass that variable to it, or we can perform the conversion in one statement. We found that the latter implementation option is better with respect to area.

Register reuse results in more multiplexed paths, which increases the delay in accessing the data stored in a certain register. This could be improved by introducing some redundancy. For registers, redundancy is done by defining more than one variable for the same data, which maps to multiple registers. Register redundancy decreases the delay required to access that data since it reduces the logic levels generated by multiplexing, but in general it increases the area required. However, redundancy could also decrease the area required as it might reduce the number of used multiplexors. For example, the `Alg` signal is used extensively in our code. In order to decrease the delay, we used three registers to store the value of `Alg`, and used each copy in different parts of the code. We noticed that this replication of the `Alg` register decreases both delay and area.

3.3 Path Delay

Path delay of an operation is the delay from the source to the destination of that operation. Splitting a certain operation into a number of sub-operations, each to be executed in one clock cycle usually decreases the path delay of the new sub-operations. This implies that the path delay is inversely proportional to the number of clock cycles required to execute an operation.

In general, this method of increasing clock cycles of an operation decreases the longest path delay of a design. However, care must be taken when throughput is considered. Throughput depends on both the longest path delay, which determines the maximum clock frequency of a design, and the number of clock cycles required to process a certain task. So, the trade-off between the two parameters should be studied carefully. For example, a compression step in SHA-1 (Eastlake, 2001; NIST, 2002b) can be executed in one clock cycle or more. Since processing one 512-bit block using SHA-1 is done in 80 compression steps, the total number of clock cycles required to process a block is the result of multiplying 80 by the number of clock cycles required for one compression step. The more clock cycles for a compression step, the more clock cycles are required for processing one 512-bit block, and that will affect the throughput as will be discussed in Section 6.3. On the other hand, executing a compression step in one clock cycle may increase the longest path delay due to the complex operations required for a compression step (addition, logical operations, left rotation, and accessing memory). Therefore, we found that using two clock cycles for each compression step is better for a balance between clock cycles and path delay, which results in a better throughput.

It should be noted that, even if we increase the clock cycles of some complex operations, the longest path delay is still high. This could be explained as the delay of the longest path in that case results from the use of registers in different parts of the code, which requires multi-levels of multiplexing in hardware. For example, using nested *if-conditions* makes the logic depth high. In such cases, the longest path delay results from the long logic depth and not from the decrease in the number of clock cycles needed for a complex operation. Therefore, decreasing the clock cycles in this case is better.

3.4 Conditions

For conditions in Handel-C, we can use either *if-else* or *switch-case*. For cases with two options, such as checking the `Mode` signal, we found that both constructs can be used alternatively and have same effects on both software and hardware. For cases with multi options, one can select between *switch-case*, parallel *if-else*, or nested *if-else*. In hardware, we found that *switch-case* and parallel *if-else* have same performance. For example, checking the `Alg` signal can be implemented using *switch-case* or parallel *if-else*. However, in cases that have complicated numerical conditions, it is more feasible from the implementation point of view to use *if-else*. An example on such cases is checking the padding case in the unified hash algorithm, which depends on the message size. Nested *if-else* has to be used with constructs that cannot be accessed in parallel, such as memory.

3.5 Loops

Loops can be implemented in hardware in different ways. One way is to implement the loop iteration using a logic block and use this logic block for all iterations. This way needs some additional logic for checking the loop condition or checking the loop index. Another way is to implement each iteration in a different logic block, which needs more area for replicating the hardware logic.

In Handel-C, constructs used for loops are *while*, *for*, and *seq* (which is used to replicate a piece of code a number of times and execute the replicated copies in sequence). With respect to area, *while* loops use the same hardware logic for each iteration. They also need some logic and storage for updating and checking the loop condition. Similarly, *for* loops use the same hardware logic for each iteration and need some storage for loop counters and end conditions and some logic to update and increment the index. *seq* loops need more area because each loop iteration is implemented as a different logic block.

With respect to timing, *for* loops require one clock cycle for initializing the loop index and one extra clock cycle for incrementing or decrementing the index for each iteration. These clock cycles can be saved using *while* or *seq* loops.

As an example, the compression steps of MD5 are executed 64 times (Rivest, 1992), and each step takes two clock cycles in our implementation. This loop can be implemented using *for*, *while*, or *seq* loops:

- Using a *for* loop, each iteration takes 3 clock cycles, two for MD5 processing and one for updating the index variable. In addition, one clock cycle is required to initialize the index. So, the number of clock cycles required is $1 + 3 \times 64 = 193$.
- Using a *while* loop, each iteration takes only two clock cycles by updating the index in parallel with MD5 processing. The index can be initialized before the loop in parallel with some statements. Therefore, the number of clock cycles required is $2 \times 64 = 128$.
- Using a *seq* loop requires the same number of clock cycles as the *while* loop. However, the hardware logic used for an MD5 compression step is replicated 64 times.

From the above discussion, we can see that *while* loops are generally better to use than *for* and *seq* loops. However, in some cases, we have to use *seq* loops when the loop index is used with constructs that require compile-time constants. Loop indices used by *seq* loops are compile-time constants, whereas they are not compile-time constants for *while* and *for* loops (Sutcliffe, 2004). An example of a case where we have to use *seq* loops is when the resized key $\mathtt{K}_0$ is XORed with `ipad` or `opad` and sent to the hash unit (Steps 2 and 5 of the HMAC algorithm described in Section 2.1, respectively). Because the size of the `Msg` channel (see Figure 1) is 32, we need 16 clock cycles to send a 512-bit block to the Hash unit. Since $\mathtt{K}_0$ is stored as a 512-bit array, we need to index it 16 times and, in each time, we select 32 bits of $\mathtt{K}_0$ and perform the XOR operation with 32 bits of `ipad` or `opad` and send the result to the hash unit. Compile-time constants are required for bit selection, and hence, this example has to be done using `seq` loops.

In Handel-C, *while* loops can be implemented using either the regular *while* loops, or *do-while* loops. These two constructs have same effects, with respect to both area and timing. The only difference occurs when the loop is embedded inside a condition (e.g., if-condition or another *while* loop). In that case, using *do-while* is preferable in order to avoid combinational cycles caused by two consecutive conditions. It should be noted that *do-while* can only be used if at least one iteration of the loop is to be executed (Celoxica Limited, 2001; Sutcliffe, 2004). For example, word expansion is a step required for SHA-1, and it is executed 64 times. So, we first need to check the `Alg` signal, and if the hash function selected is SHA-1, then we will execute the loop 64 times. In this case, it is better to use *do-while* compared to *while* loops.

3.6 Register Size

Using larger size registers to store message blocks decreases the number of clock cycles required to process them. However, using larger registers increases the area required. For example, we can use a 512-bit register to store a 512-bit block of a message. In this case, one clock cycle is required to receive that block, store it, and pass it to the hash unit. Alternatively, we can use 16 32-bit registers to store one block. In this case, we need 16 clock cycles to receive and store a 512-bit block. The area required for the former case is much more than the latter case. The increase in area is not only for registers, but also for other logic blocks. In other words, using larger registers requires larger buses, adders, and so on.

4. HMAC IMPLEMENTATION FOR FIXED KEY SIZE

The HMAC unit described by Khan *et al* (2007) implements the general HMAC algorithm, which allows keys of any size. For example, HMAC-MD5 specified in RFC 2085 (Oehler and Glenn, 1997) recommends the support of long key lengths. However, in RFC 2403 (Madson and Glenn, 1998a), RFC 2404 (Madson and Glenn, 1998b), and RFC 2857 (Keromytis and Provos, 2000), the key size is restricted to 128 bits for HMAC-MD5 and 160 bits for each of HMAC-SHA-1 and HMAC-RIPEMD-160. Therefore, we implemented another version of the HMAC unit using a key size of 160 bits. If the hash function selected is HMAC-MD5, the least significant 32 bits of the key are filled with 0's. Since the key size is fixed, there is no need to get the key size as an input to the HMAC unit.

The HMAC implementation for a fixed key size eliminates the key resizing step from the HMAC algorithm (Step 1 of the HMAC algorithm described in Section 2.1). Instead, K_0 is created by filling the least significant 352 bits (512–160) with 0's. This feature will improve the performance of the designed unit because of the following:

- The hash unit receives the message in 32-bit words one at a time, starting from the most significant word.
- `ipad` and `opad` can be represented as repeated 32-bit words.
- `V2` and `V5` are formed by XORing K_0 with `ipad` and `opad`, respectively, and then concatenating that to the message at the most significant position.
- Hashing a message is done in a pipelined fashion.

Using the above implementation facts, the HMAC algorithm (Section 2.1) using fixed key size can be implemented as follows. Let `K` be the 160-bit key received such that:

$$K = K1@K2@K3@K4@K5,$$

where `K1` to `K5` are 32-bit words each and @ is a concatenation operation. Let `ipad32` and `opad32` be 32-bit words of `ipad` and `opad`, respectively. Figure 3 shows how words are sent to the hash unit. Figure 3 (a) shows the first block of `V2` (which is `V1`, as in the HMAC algorithm explained in Section 2.1). This block consists of 16 32-bit words. These words are sent to the hash unit in order, one word at a time, starting from the left most word. After this block is sent, the message `M` is sent

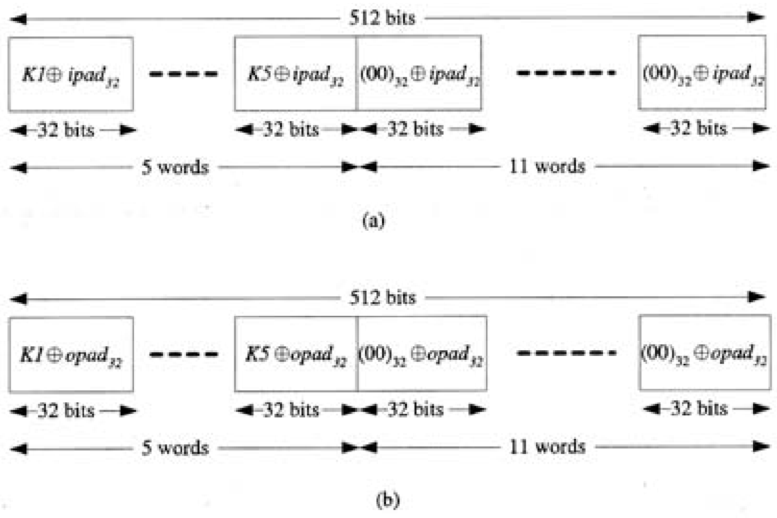


Figure 3: Formatting the first block before sending it to the hash unit in case of using a fixed key size. (a) First block of `V2` (Step 2 of the HMAC algorithm). (b) First block of `V5` (Step 6 of the HMAC algorithm).

to the hash unit, one block at a time. `V5` is sent to the hash unit in a similar way, as shown in Figure 3 (b). `V4`, is sent first to the hash unit, followed by `V3` (as in the HMAC algorithm explained in Section 2.1).

5. KEY REUSE MECHANISM

In order to improve the throughput of the HMAC unit, we propose a key reuse mechanism. This mechanism is useful when the same key is used for successive messages. In that case, there is no need to receive the key and resize it. Instead, the HMAC algorithm uses the available resized key that has been used for the last message. For this purpose, we add a new one-bit input signal called `Same_key` to the designed HMAC-hash unit. When `Same_key` = 1, receiving and resizing the key are deactivated and the old resized key is used. When `Same_key` = 0, a new key has to be received and resized.

We should be careful when using this mechanism. If the key size is greater than 512, the key needs to be hashed for resizing. If the hash function selected for the current message is not the same as that used with the previous message, the resized key for the current message will be different from the resized key used for the previous message, even if the key is the same before resizing.

Table 2 summarizes the different possible cases for the key reuse mechanism. If `Same_key` = 0, the key has to be resized anyway. If `Same_key` = 1, we have to check the hash function to be used.

Same_key	Hash function selected	Key size	Key resizing
0	any	any	Yes
1	same as previous message	any	No
1	different from previous message	≤ 512	No
1	different from previous message	> 512	Yes

Table 2: Different cases for key resizing when using key reuse.

Implementation	Key size	Key reuse
D1	General	No
D2	General	Yes
D3	Fixed	No
D4	Fixed	Yes

Table 3: The four implementations of the HMAC unit with respect to key size and key reuse.

If it is the same, the old key is used. If it is different, the old key is used only if the key size is less than or equal to 512. Otherwise, a new key has to be received and resized.

If we combine the key reuse mechanism with the possible HMAC implementations described in Section 4, we have four possible implementations, which are summarized in Table 3. In the following section, the performance of these four implementations is analyzed and compared.

6. PERFORMANCE ANALYSIS

In this section, the performance analysis of the above four implementations (D1, D2, D3, and D4) is discussed. This analysis considers four design metrics: area, delay, throughput, and power consumption. We synthesized the designed unit to a Virtex II FPGA chip (XC2V4000) (Xilinx, 2005) using the Xilinx ISE tool.

6.1 Area Analysis

Area is measured by the amount of hardware resources occupied by the designed HMAC-hash unit. Table 4 shows detailed figures comparing area requirements for the four implementations.

Implementation	D1	D2	D3	D4
Total number of used LUTs* (Available = 46,080)	14,911	14,913	12,962	12,970
LUTs utilization	32.36%	32.36%	28.13%	28.15%
LUTs used as logic	12,740	12,742	11,623	11,631
LUTs used as a route-thru	647	647	519	519
LUTs used for dual ported RAMs	1,216	1,216	512	512
LUTs used as 16x1 RAMs	32	32	32	32
LUTs used as 16x1 ROMs	263	263	263	263
LUTs used as shift registers	13	13	13	13
Total number of flip flops (Available = 46,080)	3,540	3,552	3,422	3,431
Flip flops utilization	7.68%	7.71%	7.43%	7.45%
Number of bonded IOBs (Available = 824)	169	170	137	138
IOB utilization	20.51%	20.63%	16.63%	16.75%

* Total number of used LUTs includes LUTs used as logic, memory, routing, and shift registers.

Table 4: Details of area requirements of the four implementations of the HMAC-hash unit.

The table shows clearly that implementations with fixed key size (D3 and D4) consume less area than implementations with general key size (D1 and D2). The saving in area using fixed key size implementations is about 13%. This saving occurs in LUTs used as logic, dual ported RAM, and routing, and in flip flops. In addition, the number of IOBs (input/output blocks) of a fixed key size implementation is less than the corresponding general implementation by 32, which is the number of input ports used for key size.

We can notice from Table 5 that the proposed key reuse mechanism (D2 and D4) does not add much area to the implementations that do not include this feature (D1 and D3). It adds few LUTs, few more flip flops, and one IOB (for the `Same_key` signal). However, it increases the throughput for reused keys, especially when key sizes are long, as will be discussed in Section 6.3. Therefore, we recommend including this feature in any HMAC implementation.

6.2 Delay Analysis

Table 5 shows the details of the longest path delay of the proposed unit as determined by the synthesizer (Xilinx ISE place and route tool). The total delay, which determines the maximum frequency, consists of two parts

1. **Data path delay:** which is the delay from a source to a destination. This delay is the sum of two delays: logic delay and routing delay.
2. **Clock path skew:** which is the difference between the time a clock edge arrives at the source of a path and the time it arrives at the destination of that path.

The difference in the total delay when including the key reuse mechanism (D2 and D4) is relatively small (less than 0.15 ns). However, the saving in clock cycles using this mechanism is considerable for reused large keys. Therefore, the overall throughput of HMAC using this feature will be better, as will be shown in Section 6.3.

On the other hand, the key resizing step used for the general HMAC implementations (D1 and D2), adds more delay than key reuse (about 0.4 ns). Therefore, implementing HMAC with fixed key size is preferable as long as security issues regarding key selection are taken into account.

We can see from Table 5 that the routing delay is a very important factor in the total delay. The minimum of routing delay is 40.9%. In our implementation, we relied on automatic routing. Manual routing may be used for further optimization.

Implementation	**D1**	**D2**	**D3**	**D4**
Total delay (ns)	23.005	23.144	22.551	22.680
Maximum frequency (MHz)	43.47	43.21	44.34	44.10
Data path delay (ns)	23.005	23.097	22.551	22.604
Clock path skew (ns)	0	-0.047	0	-0.076
Logic delay (ns)	11.230	13.645	12.050	13.366
Percentage of logic delay to data path delay	48.8%	59.1%	53.4%	59.1%
Routing delay (ns)	11.775	9.452	10.501	9.238
Percentage of routing delay to data path delay	51.2%	40.9%	46.6%	40.9%

Table 5: Details of the longest path delay of the four implementations of the HMAC-hash unit.

6.3 Throughput Analysis

The throughput of processing one message block using hash functions is computed according to the following equation (Khan *et al*, 2007; Wang *et al*, 2004):

$$Throughput_{block} = \frac{BS \times F}{CC_b} \tag{1}$$

where BS is the block size (512 bits in our case), F is the maximum clock frequency ($F = 1/T_c$), T_c is the clock period, and CC_b is the number of clock cycles required to process a block.

The overall throughput of hashing a message is given by the following equation:

$$Throughput_{hash} = \frac{N \times BS \times F}{CC_{pc} + (N \times CC_b)} \tag{2}$$

where CC_{pc} is the number of clock cycles required for preprocessing and completion of hashing a message (two steps of the unified hash algorithm (Khan *et al*, 2007)), and N is the number of message blocks. When N is large enough, we can ignore the term CC_{pc}, and (2) will be equal to (1)

$$Throughput_{hash} = Throughput_{block} = \frac{BS \times F}{CC_b} \tag{3}$$

To find out the throughput of the HMAC-hash unit, we use the following equation:

$$Throughput_{HMAC} = \frac{N \times BS \times F}{3 \times CC_{pc} + (3 + N + k)(CC_b)} \tag{4}$$

where k is the number of the key blocks if its size is more than 512. If its size is less than or equal to 512, it is resized without hashing, and in this case, k equals zero in (4). In the HMAC algorithm (see Section 2.1), there are three additional blocks that need to be hashed. Therefore, we multiply $(3 + N + k)$ by CC_b. We multiply CC_{pc} by 3 because HMAC uses the hash function three times. When N is large enough, we can ignore the term $3 \times CC_{pc}$, and (4) can be rewritten as follows (Wang *et al*, 2004):

$$Throughput_{HMAC} = \frac{N}{3 + N + k} \times Throughput_{hash} \tag{5}$$

Table 6 shows the experimental information required to compute the throughput of the four implementations.

The clock cycles required for the HMAC algorithm are computed per message. This number of clock cycles includes the cycles required to receive the key, prepare the message (and key if required) for hashing, and output the MAC. However, it does not include the time required for hashing, which is included in the next rows of the table. For the implementations with general key size (D1 and D2), the HMAC algorithm takes from 7 to 24 clock cycles, depending on the key size and on the `Same_key` signal (if used). For the implementations with fixed key size (D3 and D4), there is one extra clock cycle required for receiving the key when the selected algorithm is HMAC-SHA-1 or HMAC-RIPEMD-160.

The clock cycles required for hash preprocessing and completion are also computed per message. The preprocessing time required for hashing includes receiving a message block, segmenting it into 16 32-bit blocks, preparing it in the appropriate endian format, and padding the last message block. The completion time includes concatenating the chaining variables to form the hash value, preparing it in the appropriate endian format, and sending it to the HMAC unit (Khan

Implementation	D1	D2	D3	D4
Maximum frequency (MHz)	43.47	43.21	44.34	44.1
Clock cycles for HMAC	10-24	7-24	11-12	7-12
Clock cycles for hash preprocessing and completion for MD5 and RIPEMD-160	23-26			
Clock cycles for hash preprocessing and completion for SHA-1	20-23			
Clock cycles for processing one 512-bit block for MD5	130			
Clock cycles for processing one 512-bit block for SHA-1 and RIPEMD-160	162			
Average MD5 throughput of a 512-bit block (Mbps) (Equation 1)	171.20	170.20	174.63	173.69
Average SHA-1 and RIPEMD-160 throughput of a 512-bit block (Mbps) (Equation 1)	137.40	136.56	140.14	139.38

Table 6: Details of different factors used to compute the throughput of the four implementations of the HMAC-hash unit.

et al, 2007). The number of clock cycles required for all these steps is small compared to the number of cycles required for processing one 512-bit block. Therefore, the larger the message size, the closer the overall throughput to the throughput of processing one 512-bit block. The number of clock cycles required for processing a message block includes a clock cycle for initializing the working variables, clock cycle for updating the chaining variables, and two clock cycles per round (64 rounds for MD5 and 80 rounds for each of SHA-1 and RIPEMD-160).

6.4 Power Consumption Analysis

Table 7 shows the estimated power consumption of the four implementations obtained using ISE XPower tool. The total estimated power consists of two parts:

1. **Dynamic power:** which is the power consumed by switching activities. This includes clock, input, output, logic, and internal signals.
2. **Quiescent power:** which is the power consumed with no signal switching. It is also called "static power".

The dynamic power shown in the table includes only clock and logic signals, because the other types of signals depend heavily on the external loading capacitance of the designed unit. We notice from the table that the quiescent power is the same for all implementations. On the other hand, the

Implementation	D1	D2	D3	D4
Total power consumption (mW)	876	867	805	793
Dynamic power (mW)	201	192	130	118
Quiescent power (mW)	675	675	675	675

Table 7: Details of the estimated power consumption of the four implementations of the HMAC-hash unit.

dynamic power increases with the increase in area as well as in clock frequency. D1 and D2 consume more area than D3 and D4 (see Table 4), and hence D1 and D2 consume more power than D3 and D4. On the other hand, D1 has a higher frequency than D2 (see Table 6), and therefore, D1 consumes more power than D2, although D1 consumes slightly less area than D2 (2 LUTs). Similarly, D3 consumes more power than D4 because it has higher frequency, although it consumes less area (8 LUTs). These results show that frequency has more impact on dynamic power than area, to a certain level.

To optimize the design for power consumption, we have two options. The first is to decrease the area required, by removing some functionality from the design. The second is to decrease the frequency, and in this case, there will be a trade-off between power consumption and throughput.

7. COMPARISON

In this section, we compare the results obtained using our proposed HMAC-hash unit to the results obtained using other designs of hash units. Table 8 summarizes the comparison. The results shown in the table for our proposed design is for D3 (fixed key size with no key reuse) because all other designs used this specification. The table shows that our design outperforms most of the other designs in speed. The only exceptions are the works of Selimis *et al* (2003) and Kitsos *et al* (2002). However, these two designs include only one hash function, whereas ours includes six integrated hash functions. The extra area required by our design is due mainly to the number of incorporated hash functions in our design. Our design incorporates six hash algorithms, whereas at most four hash functions are included in other designs. We can also see from the table that the performance achieved from our design, which is based on the Handel-C methodology, is either comparable to or better than those designs that are based on other methodologies.

Designs	Ng *et al* 2004	Wang *et al* 2004	Kang *et al* 2002	Domnikus 2002	Selimis *et al* 2003	McLoone & McCanny 2002	Kitsos *et al* 2002	Our design
HMAC Included?	No	Yes	No	No	Yes	Yes	No	Yes
Hash Functions implemented*	M, R	M, S	M, S, H	M, S, R, S265	S	S	S	M, S, R
FPGA vendor	Altera	Altera	Altera	Xilinx	Xilinx	Xilinx	Xilinx	Xilinx
FPGA chip	EPF10 K50	EP20 K1000	EP20 K1000	XC V300E	XC V50	XC V1000E	XC V300	XC2 V4000
Area cost (LUTs)	1,964	5,329	10,573	4,493	1,593	14,494**	5,112	12,962
Maximum frequency (MHz)	26.66	21.96	18.00	42.90	82.00	24.20	47.00	44.34

* M = MD5, S = SHA-1, R = RIPEMD-160, H = HAS-160 (The Hash function Algorithm Standard), S256 = SHA-265.

** This number includes area cost of the HMAC-SHA-1 core and an encryption core.

Table 8: Comparison with other design methodologies.

8. DISCUSSION AND CONCLUSION

In this paper, we discussed the design space exploration of the reconfigurable HMAC-hash unit proposed in our previous work (Khan *et al*, 2007). First, we discussed the design options that are implemented using Handel-C. Handel-C and the DK design suite have constructs and options that facilitate design space exploration of any FPGA design. We showed how these options affect the performance of the design and which options we selected.

We also showed two implementations of the HMAC unit with respect to key size. The first implementation uses general key size, where a key of any length is allowed. The second implementation uses fixed key size to conform to some of HMAC specifications in the literature.

We proposed a key reuse mechanism to improve the HMAC throughput. This mechanism eliminates the key resize step of the HMAC algorithm when the same key is used for consecutive messages. We combined this feature with the two implementations of the HMAC units for general and fixed key size, which resulted in four possible implementations of the HMAC unit; namely, HMAC for general key size, with and without key reuse, and HMAC for fixed key size, with and without key reuse.

The performance of the four possible implementations of the HMAC unit with respect to key size and key reuse have been analyzed for four performance metrics, area, delay, throughput, and power consumption. Table 9 summarizes our findings from this analysis. The table shows that the proposed key reuse mechanism improves the HMAC throughput significantly when a large key is reused, with negligible increase in area and delay. It also shows that the implementation of HMAC for fixed key size is better in area, delay, throughput, and power consumption. Therefore, it is better

Design metric	Conclusions
Area	• Implementations with fixed key size consume less area than implementations with general key size. • Including the key reuse mechanism does not add much area to the implementations that do not include this feature.
Delay	• Implementations with fixed key size add less delay than implementations with general key size. • Including the key reuse mechanism does not add much delay to the implementations that do not include this feature. • Routing delay is an important factor of total path delay.
Throughput	• The hash throughput is almost equal to the throughput of processing one message block. • HMAC with fixed key size has a throughput almost equal to the throughput of processing one message block for large message sizes. • For large message sizes, the key reuse mechanism makes the overall throughput of HMAC with general key size almost equal to the throughput of processing one message block.
Power consumption	• HMAC with fixed key size and key reuse consumes the least power. • Frequency has more impact on power consumption than area.

Table 9: Summary of our findings and conclusions

to implement the HMAC algorithm for fixed key size as long as security issues regarding key selection are taken into account, and in this case, it is better not to include the key reuse mechanism. On the other hand, we recommend including the key reuse mechanism in any implementation of the HMAC algorithm with a general key size.

REFERENCES

AUBURY, M., PAGE, I., RANDALL, G., SAUL, J., and WATTS, R. (1996): Handel-C language reference guide. http://www.inf.pucrs.br/~moraes/topicos/hdls/HANDEL-C/HANDELC.PDF. Accessed 13-Feb-2008.

CELOXICA LIMITED (2001): Implementing efficient loops in Handel-C. Application Note 71 v1.1. http://www.celoxica.com/techlib/?les/CEL-W0307171JLS-34.pdf. Accessed 13-Feb-2008.

CELOXICA LIMITED (2002): Handel-C Language Overview. Product Brief. http://www.celoxica.com/techlib/?les/CEL-W0307171KDD-47.pdf. Accessed 13-Feb-2008.

CELOXICA LIMITED (2003): Handel-C language reference manual. http://www.celoxica.com/techlib/?les/CEL-W030811132Q-60.pdf. Accessed 13-Feb-2008.

CELOXICA LIMITED (2005): Software product description for DK Version 4.0. http://www.celoxica.com/support/articles/521/CEL-ENGSPDDKDK 4.0 SP2 SPD-01000.pdf. Accessed 13-Feb-2008.

DOBBERTIN, H., BOSSELAERS, A., and PRENEEL, B. (1996): RIPEMD-160, A strengthened version of RIPEMD. http://www.esat.kuleuven.ac.be/~cosicart/pdf/AB-9601/AB-9601.pdf. Accessed 13-Feb-2008.

DOMINIKUS, S. (2002): A hardware implementation of MD4-Family hash algorithms. In *Proceedings of the 9th International Conference on Electronic, Circuits, and Systems (ICECS 2002)*, Dubrovnik, Croatia: 1143–1146.

EASTLAKE, D. (2001): RFC 3174 - US secure hash algorithm 1 (SHA1). http://www.faqs.org/rfcs/rfc3174.html. Accessed 13-Feb-2008.

HAA, C.-S., LEE, J. H., LEEM, D. S., PARK, M.-S. and CHOI, B.-Y. (2004): ASIC design of IPSec hardware accelerator for network security. In *Proceedings of the 2004 IEEE Asia-Pacific Conference on Advanced System Integrated Circuits (AP-ASIC 2004)*, Fukuoka, Japan: 168–171.

JUSSEL, J. (2005): C to FPGA: An abstract concept for concrete design implementation. http://www.rtcmagazine.com/home/article.php?id=100304. Accessed 13-Feb-2008.

KANG, Y. K., KIM, D. W., KWON, T. W. and CHOI, J. R. (2002): An efficient implementation of hash function processor for IPSec. In *Proceedings of the 2002 IEEE Asia-Pacific Conference on ASICs (AP-ASIC 2002)*, Taipei, Taiwan: 93–96.

KEROMYTIS, A. and PROVOS, N. (2000): RFC 2857 - The use of HMAC-RIPEMD-160-96 within ESP and AH. http://www.faqs.org/rfcs/rfc2857.html. Accessed 13-Feb-2008.

KHAN, E., EL-KHARASHI, M. W., GEBALI, F. and ABD-EL-BARR, M. (2007): Design and performance analysis of a unified, reconfigurable HMAC-Hash unit. *IEEE Transactions on Circuits and Systems - I* 54(12):2683-2695.

KHAN, E., EL-KHARASHI, M. W., GEBALI, F. and ABD-EL-BARR, M. (2006): Applying the Handel-C design flow in designing an HMAC-Hash unit on FPGAs. *IEE Proceedings - Computers & Digital Techniques* 153(5):323 – 334.

KIENHUIS, A. C. J. (1999): Design space exploration of stream-based dataflow architectures: Methods and tools. Ph.D. dissertation. Delft University of Technology, The Netherlands. http://ptolemy.eecs.berkeley.edu/~kienhuis/ftp/thesis.pdf. Accessed 13-Feb-2008.

KITSOS, P., SKLAVOS, N. and KOUFOPAVLOU, O. (2002): An efficient implementation of the digital signature algorithm. In *Proceedings of the 9th International Conference on Electronics, Circuits and Systems (ICECS 2002)*, Dubrovnik, Croatia, 3:1151–1154.

KRAWCZYK, H., BELLARE, M. and CANETTI, R. (1997): RFC 2104 -HMAC: Keyed-hashing for message authentication. http://www.faqs.org/rfcs/rfc2104.html. Accessed 13-Feb-2008.

LU, J. and LOCKWOOD, J. (2005): IPSec implementation on Xilinx Virtex-II Pro FPGA and its application. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05)*, Denver, CO, USA: 158b.

MADSON, C. and GLENN, R. (1998a): RFC 2403 - The use of HMAC-MD5-96 within ESP and AH. http://www.faqs.org/rfcs/rfc2403.html. Accessed 13-Feb-2008.

MADSON, C. and GLENN, R. (1998b): RFC 2404 - The use of HMAC-SHA-1-96 within ESP and AH. http://www.faqs.org/rfcs/rfc2404.html. Accessed 13-Feb-2008.

McLOONE, M. and McCANNY, J. V. (2002): A single-chip IPSec cryptographic processor. In *Proceedings of the IEEE Workshop on Signal Processing Systems (SIPS 2002)*, San Diego, CA, USA: 133–138.

NG, C.-W., NG, T.-S. and YIP, K.-W. (2004): A unified architecture of MD5 and RIPEMD-160 hash algorithms. In *Proceedings of the 2004 International Symposium on Circuits and Systems (ISCAS '04)*, Vancouver, BC, Canada: 23–26.

NIST (2002a): The keyed-hash message authentication code (HMAC). FIPS PUB 198. http://csrc.nist.gov/publications/?ps/?ps198/?ps-198a.pdf. Accessed 13-Feb-2008.

NIST (2002b): Secure hash standards. FIPS PUB 180-2. http://www.csrc.nist.gov/publications/?ps/?ps180-2/?ps180-2withchangenotice.pdf. Accessed 13-Feb-2008.

OEHLER, M. and GLENN, R. (1997): RFC 2085 - HMAC-MD5 IP authentication with replay prevention. http://www.faqs.org/rfcs/rfc2085.html. Accessed 13-Feb-2008.

RIVEST, R. (1992): RFC 1321 - The MD5 message-digest algorithm. http://www.faqs.org/rfcs/rfc1321.html. Accessed 13-Feb-2008.

SELIMIS, G., SKLAVOS, N. and KOUFOPAVLOU, O. (2003): VLSI implementation of the keyed-hash message authentication code for the wireless application protocol. In *Proceedings of the 10th IEEE International Conference on Electronics, Circuits and Systems (ICECS 2003)*, Sharjah, United Arab Emirates, 1:24–27.

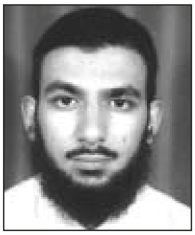
SUTCLIFFE, A. (2004): Using Handel-C with DK. Celoxica Limited. Training Course Manual, Version 1.0.2.

WANG, M.-Y., SU, C.-P., HUANG, C.-T. and WU, C.-W. (2004): An HMAC processor with integrated SHA-1 and MD5 algorithms. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC 2004)*, Yokohama, Japan: 456–458.

XILINX (2007): Xilinx ISE 9.1i Software manuals and help. http://toolbox.xilinx.com/docsan/xilinx9/books/manuals.pdf. Accessed 13-Feb-2008.

XILINX (2005): Virtex-II Platform FPGAs: Complete data sheet. product specification, DS031 (v3.4). http://www.xilinx.com/bvdocs/publications/ds031.pdf. Accessed 13-Feb-2008.

BIOGRAPHICAL NOTES


Esam Khan

Esam A. Khan received the B.Sc. degree in computer engineering (first class honours) and the M.Sc. degree in computer engineering from the Department of Computer Engineering, King Fahd University of Petroleum and Minerals (KFUPM), Dhahran, Saudi Arabia, in 1999 and 2001, respectively, and the Ph.D. degree in electrical and computer engineering from the Department of Electrical and Computer Engineering, University of Victoria, Victoria, BC, Canada, in 2005. He is currently an Assistant Professor at the Custodian of the Two Holy Mosques Institute for Hajj Research, Umm Al-Qura University, Makkah, Saudi Arabia. His research interests include systems-on-a-chip (SoC) and hardware implementations of security and cryptographic algorithms.

M. Watheq El-Kharashi

M. Watheq El-Kharashi received the Ph.D. degree in computer engineering from the University of Victoria, Victoria, BC, Canada, in 2002, and the B.Sc. degree (first class honours) and M.Sc. degrees in computer engineering from Ain Shams University, Cairo, Egypt, in 1992 and 1996, respectively. He is currently an Assistant Professor in the Department of Computer and Systems Engineering, Ain Shams University. His research interests include advanced microprocessor design, simulation, performance evaluation, and testability, systems-on-a-chip (SoC), networks-on-a-chip (NoC), and computer architecture and computer networks education.

Fayez Gebali

Fayez Gebali received the B.Sc. degree in electrical engineering from Cairo University, Cairo, Egypt, the B.Sc. degree in applied mathematics from Ain Shams University, Cairo, Egypt, and the Ph.D. degree in electrical engineering from the University of British Columbia, Vancouver, BC, Canada, in 1972, 1974, and 1979, respectively. He is currently a Professor in the Department of Electrical and Computer Engineering, University of Victoria, Victoria, BC, Canada. His research interests include computer networks and high performance microprocessors. He also works in computer arithmetic, techniques for mapping parallel algorithms onto processor array, and VLSI design for digital signal processing.

Mostafa Abd-El-Barr received the Ph.D. degree in computer engineering from the University of Toronto, Toronto, Canada, in 1986. In July 1986, he joined the Department of Computer Science, University of Saskatchewan, SK, Canada. In September 1996, he joined the Department of Computer Engineering, King Fahd University of Petroleum and Minerals (KFUPM), Dhahran, Saudi Arabia. He is currently a Professor in the Department of Information Science, CFW, Kuwait University, Kuwait. His research interests include fault tolerance of parallel and distributed systems, computer network reliability-based optimization techniques, information security, VLSI design and implementation of algorithms, testing and design for testability, and multiple-valued logic system design.

Mostafa Abd-El-Barr