DESIGN SPACE EXPLORATION OF A RECONFIGURABLE HMAC-HASH UNIT

Esam Khan, M. Watheq El-Kharashi, Fayez Gebali, and Mostafa Abd-El-Barr

Department of Electrical and Computer Engineering, University of Victoria, Victoria, BC, Canada
E-mail: {ekhan, watheq, fayez, mabdelba}@ece.uvic.ca

Abstract: 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 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 use standard key use and the proposed key reuse mechanisms, and that results in four different implementations. The performance of these four implementations is analyzed with respect to three design metrics: area, delay, and throughput. We found that the proposed key reuse mechanism improves the HMAC throughput significantly when a large key is reused, with negligible increase in area and delay. In addition, we found that the implementation of HMAC for fixed key size is better in area, delay, and throughput than the HMAC implementation for general key size.

Keywords: Authentication, Design space exploration, HMAC, Hash functions, IPSec.
1. INTRODUCTION

One of the standard algorithms used for authentication is the Keyed-Hash Message Authentication Code (HMAC) [1]. HMAC is a shared-key algorithm that uses hash functions to perform the authentication process. There exists a number of standardized authentication algorithms using HMAC. The most popular ones are HMAC-MD5 [2], HMAC-SHA-1 [3], and HMAC-RIPEMD-160 [4].

In a previous work [5], we designed a reconfigurable unit that implements the HMAC algorithm. This HMAC unit was integrated to a hash unit, which we described in [6]. 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 [7].

The HMAC unit described in [5], [7] 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 sizes. 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 sizes) is implemented with the key reuse mechanism and without it. That results in four different implementations of the HMAC unit. To build the HMAC-hash unit, we used Handel-C [8].

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.

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 implementations of the HMAC unit is analyzed. Finally, we conclude the paper in Section 7.

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, the HMAC-hash unit, and the design tools 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 in [1].

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 equation:

$$MAC(M) = h((K_0 \oplus \text{opad})@h((K_0 \oplus \text{ipad})@M))$$

(1)

where ^ is the XOR operation, @ is the concatenation operation, $ipad$ and $opad$ are 512-bit constants, and $K_0$ is the shared secret key $K$ resized to 512 bits.

2.2 The HMAC-Hash Unit

The HMAC-hash unit (Figure 1) 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 a hash value.

2.3 Design Tools

We used Handel-C to implement the HMAC-hash unit. Handel-C is a high level language based on ANSIC. It is designed to enable direct compilation of programs to FPGA logic or RTL descriptions. It allows designers to import algorithms written in C and use them to

---

1 Handel-C is a trademark of Celoxica Limited.
create a rapid implementation and optimization path to hardware. The advantage of Handel-C over conventional C is that it is fully synthesizable and that it supports parallelism.

DK design suite is an Integrated Development Environment (IDE) produced by Celoxica [9] and used to compile Handel-C codes. 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.

We used the Xilinx ISE tools[10] to synthesize the designed unit and analyze its performance.

3. DESIGN SPACE EXPLORATION USING HANDEL-C

For any hardware design, there are different implementation options, which determine its design space. Design space exploration intends to find out how a particular design option affects the performance of the design in order to have a tradeoff between different performance metrics [11].

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.

Memory is an alternative option used for storage. Unlike registers, only one element of memory may be 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.

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 \[12\].

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 required area. However, redundancy could also decrease the area required as it might reduce the number of used multiplexors.

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 tradeoff between the two parameters should be studied carefully.

3.4 Conditions

For conditions in Handel-C, we can use either if-else or switch-case. For two options, such as checking the Mode signal, we found that both constructs can be used alternatively and have the same effects on both software and hardware. For 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 the 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 of 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, the 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. In addition, they 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 for index increment and comparison. 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.

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, and that is when the loop index is used with constructs that require compiletime constants. Loop indexes used by seq loops are compile-time constants, whereas they are not compile-time constants for while and for loops [12].

3.6 Register Size

Using larger size registers to store the 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. IMPLEMENTATION FOR FIXED KEY SIZE

The HMAC unit described in [5] implements the general HMAC algorithm, which allows keys of any size. However, in [2]-[4], 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.

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.

However, 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 than the resized key used for the previous message, even if the key is the same before resizing.

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 1. In the following section, the performance of these four implementations is analyzed and compared.

### Table 1: The four implementations of the HMAC unit with respect to key size and key reuse.

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Key size</th>
<th>Key reuse</th>
</tr>
</thead>
<tbody>
<tr>
<td>D1</td>
<td>General</td>
<td>No</td>
</tr>
<tr>
<td>D2</td>
<td>General</td>
<td>Yes</td>
</tr>
<tr>
<td>D3</td>
<td>Fixed</td>
<td>No</td>
</tr>
<tr>
<td>D4</td>
<td>Fixed</td>
<td>Yes</td>
</tr>
</tbody>
</table>

### 6. PERFORMANCE ANALYSIS

In this section, the performance analysis of the above four implementations (D1, D2, D3, and D4) is discussed. This analysis considers three design metrics: area, delay, and throughput.

#### 6.1 Area Analysis

Area is measured by the amount of hardware resources occupied by the designed HMAC-hash unit. Table 2 shows detailed figures comparing area requirements for the four implementations.

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 savings in area 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.

Table 2: Area requirements of the four implementations of the HMAC-hash unit.

<table>
<thead>
<tr>
<th>Implementation</th>
<th>D1</th>
<th>D2</th>
<th>D3</th>
<th>D4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total number of used LUTs*</td>
<td>14,911</td>
<td>14,913</td>
<td>12,962</td>
<td>12,970</td>
</tr>
<tr>
<td>LUTs utilization</td>
<td>32.36%</td>
<td>32.36%</td>
<td>28.13%</td>
<td>28.15%</td>
</tr>
<tr>
<td>Total number of flip flops</td>
<td>3,540</td>
<td>3,552</td>
<td>3,422</td>
<td>3,431</td>
</tr>
<tr>
<td>Flip flops utilization</td>
<td>7.68%</td>
<td>7.71%</td>
<td>7.43%</td>
<td>7.45%</td>
</tr>
<tr>
<td>Number of bonded IOBs</td>
<td>169</td>
<td>170</td>
<td>137</td>
<td>138</td>
</tr>
<tr>
<td>IOB utilization</td>
<td>20.51%</td>
<td>20.63%</td>
<td>16.63%</td>
<td>16.75%</td>
</tr>
</tbody>
</table>

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

6.2 Delay Analysis

Table 3 shows the details of the longest path delay of the proposed unit as determined by the synthesizer (Xilinx ISE place and route tool).

Table 3: Details of longest path delay of the implementations of the HMAC-hash unit.

<table>
<thead>
<tr>
<th>Implementation</th>
<th>D1</th>
<th>D2</th>
<th>D3</th>
<th>D4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data path delay (ns)</td>
<td>23.005</td>
<td>23.097</td>
<td>22.551</td>
<td>22.604</td>
</tr>
<tr>
<td>Logic delay (ns)</td>
<td>11.23</td>
<td>13.645</td>
<td>12.050</td>
<td>13.366</td>
</tr>
<tr>
<td>Percentage of logic delay to data path delay</td>
<td>48.8%</td>
<td>59.1%</td>
<td>53.4%</td>
<td>59.1%</td>
</tr>
<tr>
<td>Routing delay (ns)</td>
<td>11.775</td>
<td>9.452</td>
<td>10.501</td>
<td>9.238</td>
</tr>
<tr>
<td>Percentage of routing delay to data path delay</td>
<td>51.2%</td>
<td>40.9%</td>
<td>46.6%</td>
<td>40.9%</td>
</tr>
</tbody>
</table>

The difference in the data path delay when including the key reuse mechanism (D2 and D4) is 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 4.3.

On the other hand, the key resizing step, which is 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.

6.3 Throughput Analysis

Throughput is defined as data processed in a unit time, and is measured in bps. The throughput of hashing a message is given by the following equation:
\[ \text{Throughput}_{\text{hash}} = \text{Throughput}_{\text{block}} = \frac{BS \times F}{CC_b} \quad (2) \]

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

It is difficult to find an exact figure for the throughput of the general HMAC algorithm because it depends on two parameters, key size and message size. To find out the throughput of the HMAC-hash unit, we use the following equation:

\[ \text{Throughput}_{\text{HMAC}} = \frac{N}{3 + N + k} \times \text{Throughput}_{\text{hash}} \quad (3) \]

where \( k \) is the number of key blocks if its size is more than 512 and and \( N \) is the number of message blocks.

Table 4 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. 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 \textit{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.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|c|}
\hline
\textbf{Implementation} & \textbf{D1} & \textbf{D2} & \textbf{D3} & \textbf{D4} \\
\hline
\textbf{Maximum frequency (MHz)} & 43.47 & 43.21 & 44.34 & 44.1 \\
\textbf{Clock cycles for HMAC} & 10-24 & 7-24 & 11-12 & 7-12 \\
\textbf{Clock cycles for hash preprocessing and completion for MD5 and RIPEMD-160} & & 23-26 & \\
\textbf{Clock cycles for hash preprocessing and completion for SHA-1} & & 20-23 & \\
\textbf{Clock cycles for processing one 512-bit block for MD5} & & 130 & \\
\textbf{Clock cycles for processing one 512-bit block for SHA-1 and RIPEMD-160} & & 162 & \\
\textbf{Average MD5 throughput of a 512-bit block (Mbps)} & 171.2 & 170.2 & 174.63 & 173.69 \\
\textbf{Average SHA-1 and RIPEMD-160 throughput of a 512-bit block (Mbps)} & 137.4 & 136.56 & 140.14 & 139.38 \\
\hline
\end{tabular}
\caption{Details of different factors used to compute the throughput of the four implementations of the HMAC-hash unit.}
\end{table}
The clock cycles required for hash preprocessing and completion (two steps of the hash functions [5], [6]) are also computed per message. The number of clock cycles required for 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 best case for $\text{Throughput}_{\text{HMAC}}$ occurs when the key size is less than 512 and $N$ is large, and in that case the HMAC throughput is almost the same as the hashing throughput. The worst case occurs when $k$ is large compared to $N$. Between these two extremes, HMAC throughput will be a fraction of the hashing throughput, depending on $N$ and $k$.

For the general HMAC implementations when the key size is less than or equal to 512, and for the HMAC implementations with fixed key size, the larger the message size, the closer the overall HMAC throughput to the throughput of processing one message block.

For key sizes greater than 512, key reuse increases the HMAC throughput for reused keys by eliminating the $k$ term from Equation 3, which approaches the throughput of processing one message block for larger message sizes.

7. DISCUSSION AND CONCLUSION

In this paper, we discussed the design space exploration of the reconfigurable HMAC-hash unit proposed in our previous work. First, we discussed the design options that are implemented using Handel-C. 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, and the second implementation uses fixed key size.

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 three
performance metrics, area, delay, and throughput. We conclude that the proposed key reuse mechanism improves the HMAC throughput significantly when a large key is reused, with negligible increase in area and delay. Because the implementation of HMAC for fixed key size is better in area, delay, and throughput, it is better to implement the HMAC algorithm for fixed key size as long as security issues regarding key selection are taken into account. 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 general key size.

REFERENCES