Hardware Implementation Analysis of the MD5 Hash Algorithm

Conference Paper · February 2005

DOI: 10.1109/HICSS.2005.291 · Source: IEEE Xplore

3 authors:

Kimmo Järvinen
Aalto University
65 PUBLICATIONS **1,480** CITATIONS
SEE PROFILE

Matti Tommiska
Xiphera Ltd
20 PUBLICATIONS **563** CITATIONS
SEE PROFILE

Jorma Skyttä
Aalto University
39 PUBLICATIONS **622** CITATIONS
SEE PROFILE

Hardware Implementation Analysis of the MD5 Hash Algorithm

Kimmo Järvinen, Matti Tommiska and Jorma Skyttä
Helsinki University of Technology
Signal Processing Laboratory
Otakaari 5 A, FIN-02150, Finland
{kimmo.jarvinen}{matti.tommiska}{jorma.skytta}@hut.fi

Abstract

Hardware implementation aspects of the MD5 hash algorithm are discussed in this paper. A general architecture for MD5 is proposed and several implementations are presented. An extensive study of effects of pipelining on delay, area requirements and throughput is performed, and finally certain architectures are recommended and compared to other published MD5 designs. The designs were implemented on a Xilinx Virtex-II XC2V4000-6 FPGA and a throughput of 586 Mbps was achieved with logic requirements of only 647 slices and 2 BlockRAMs. Methods to increase the throughput to gigabit-level were also studied and an implementation of parallel MD5 blocks achieving a throughput of over 5.8 Gbps was introduced. At least to the authors' knowledge, MD5 designs presented in this paper are the fastest published FPGA-based architectures at the time of writing.

1. Introduction

Hash algorithms, also called as message digest algorithms, are algorithms which generate a unique message digest for an arbitrary message. The digest can be considered as a fingerprint of the message and it must have the following properties: first, the hash must be easy to compute. Second, it must be very hard to compute the message from the digest and, third, it must be hard to find another message which has the same message digest as the first message [13].

Hash algorithms are used widely in cryptographic protocols and Internet communication in general. Several widely used hash algorithms exist. One of the most famous is the MD5 message digest algorithm developed by Ronald Rivest [12]. Other common algorithms are SHA-1 and its variants [7] and RIPEMD-160 [4], for example.

Commonly, hardware acceleration for hash algorithms is not required, because they are not especially computationally demanding. However in certain applications, chains of thousands of hash algorithm rounds are calculated and hardware acceleration may be required. One of these applications is a micropayment initialization currently under development in the GO-SEC project at Helsinki University of Technology (see `go.cs.hut.fi`) requiring calculation of about 10,000 consecutive MD5 rounds. At the moment, the initialization is performed calculating 10,000 consecutive rounds, but this value is only preliminary and it may change as the project proceeds.

In the GO-SEC micropayment initialization, a chain of 10,000 MD5 rounds is calculated. The input for the chain is a 160-bit key which is used as the input message for the first round. The output message digest of the first round is used as the input message of the second round, etc. When all 10,000 rounds have been calculated, the resulted message digest values are used in reversed order. Because of the one-way nature of the MD5 hash algorithm, the previous message digest cannot be derived from the next one.

In hardware acceleration of chains of hash algorithm rounds, the time required for a single algorithm calculation is more important than throughput. Thus in this paper, concentration is on minimization of the delay of an implementation rather than on throughput maximization. However, two methods to increase the throughput of an MD5 accelerator to Gbps-level are discussed as well.

Only few publications have been published concerning FPGA acceleration of the MD5 algorithm. Dominikus achieved a throughput of 146 Mbps on a Xilinx Virtex 300E FPGA in [5]. Throughputs of 165 and 354 Mbps on Virtex 1000 were achieved with iterative and pipelined designs by Deepakumara et al. in [2]. Implementation by Diez et al. achieved a throughput of 467 Mbps on a Xilinx Virtex-II XC2V3000 FPGA [3].

Field Programmable Gate Arrays (FPGAs) manufactured by Xilinx were chosen as target devices for the architectures presented in this paper. Xilinx Virtex-II device family was chosen, because it provides very fast performance and large logic resources. Virtex-II XC2V4000-6 FPGA was used for the implementations and it includes logic resources of 23,040 slices. A basic element of a slice

0-7695-2268-8/05/$20.00 (C) 2005 IEEE

is a LUT (Look-Up Table) which can implement any 4-to-1-bit operation. A slice includes 2 LUTs, 2 flip-flops and some additional logic. [14]

Designs presented in this paper are called SIG-MD5 designs, where SIG is an acronym for the Signal Processing Laboratory at Helsinki University of Technology, where the research work was performed.

The contributions of the paper are the following: a thorough study of the effects of pipelining on delay, area requirements and throughput of an MD5 design is performed. A general architecture for MD5 hardware implementations is suggested and analyzed. Two methods to increase the throughput of an MD5 implementation to Gbps-level are studied. At least to the authors' knowledge, the fastest and smallest[1] open-literature FPGA-based MD5 implementations are presented.

The paper is organized as follows: a description of the MD5 algorithm is presented in Section 2. The proposed architecture is introduced and analyzed in Section 3 and results of the implementations are presented in Section 4. The results are compared to other published implementations in Section 5 and conclusions are made in Section 6.

2. The MD5 Algorithm

MD5 is a hash algorithm introduced in 1992 by professor Ronald Rivest [12]. It is an enhanced version of its predecessor MD4 [11]. MD5 is widely used in several public-key cryptographic algorithms and Internet communication in general. MD5 calculates a 128-bit digest for an arbitrary b-bit message and it consists of the following steps [12]:

1. **Appending Padding Bits**
   The b-bit message is padded so that a single 1-bit is added into the end of the message. Then, 0-bits are added until the length of the message is congruent to 448, modulo 512.

2. **Appending Length**
   A 64-bit representation of b is appended to the result of the padding. Thus, the resulted message is a multiple of 512 bits. This message is denoted here as M.

3. **Buffer Initialization**
   Let A, B, C and D be 32-bit registers. These registers are used in derivation of the 128-bit message digest. At the beginning, they are initialized as follows:

$$\begin{array}{rcl} A & = & \text{x}''67452301'' \\ B & = & \text{x}''efcdab89'' \\ C & = & \text{x}''98badcfe'' \\ D & = & \text{x}''10325476'' \end{array} \quad (1)$$

4. **Processing of the Message**
   The heart of MD5 is an algorithm which is used for the processing of the message. The message M is divided into 512-bit blocks which are processed separately. Let X_j denote the jth block of M. First, X_0, i.e. the lowest 512-bits of M, is processed with the algorithm, then X_1 etc., until the entire M is processed.

   The algorithm consists of four rounds, each of which comprise 16 steps. Hence, 64 steps are performed in the algorithm. Let i be the index of a step. Let $X_j[k]$ denote the kth 32-bit word of X_j and let $T[i]$ be a table of 64 32-bit constants. Let $\lll s$ denote circular shift left by s bits. Values of k, s and $T[i]$ depend on i and they are presented in Table 1.

   The algorithm is performed as follows: first, values of A, B, C and D are stored into temporary variables AA, BB, CC and DD. Then, the following operations are performed for $i = 0$ to 63:

$$\begin{array}{l} A = B + ((A + \text{Func}(B,C,D) + X_j[k] + T[i]) \lll s) \\ A \leftarrow D, B \leftarrow A, C \leftarrow B, D \leftarrow C \end{array} \quad (2)$$

   where additions are additions of words, i.e. additions modulo-2^{32}. Func(X,Y,Z) is different for every round. Function $F(X,Y,Z)$ is used for the first round ($0 \le i \le 15$), $G(X,Y,Z)$ for the second ($16 \le i \le 31$), $H(X,Y,Z)$ for the third ($32 \le i \le 47$) and $I(X,Y,Z)$ for the final round ($48 \le i \le 63$). The functions are defined as follows:

$$\begin{array}{rcl} F(X,Y,Z) & = & (X \wedge Y) \vee (\neg X \wedge Z) \\ G(X,Y,Z) & = & (X \wedge Z) \vee (Y \wedge \neg Z) \\ H(X,Y,Z) & = & X \oplus Y \oplus Z \\ I(X,Y,Z) & = & Y \oplus (X \vee \neg Z) \end{array} \quad (3)$$

   where $\vee$ is a bitwise or-operation, $\neg$ is a bitwise complement, $\oplus$ is a bitwise exclusive-or-operation (xor) and $\wedge$ is a bitwise and-operation.

   Finally, the values of the temporary variables are added to the values obtained from the algorithm, i.e

$$\begin{array}{rcl} A & = & A + AA \\ B & = & B + BB \\ C & = & C + CC \\ D & = & D + DD. \end{array} \quad (4)$$

5. **Output**
   When all X_j have been processed with the algorithm, the message digest of M is in A, B, C, and D. The low-order byte of A is the first byte of the digest and the high-order byte of D is its last byte.

[1] at the time of writing the paper

Table 1. Values of k, s and $T[i]$

i	k	s	$T[i]$	i	k	s	$T[i]$	i	k	s	$T[i]$	i	k	s	$T[i]$
0	0	7	d76aa478	1	1	12	e8c7b756	2	2	17	242070db	3	3	22	c1bdceee
4	4	7	f57c0faf	5	5	12	4787c62a	6	6	17	a8304613	7	7	22	fd469501
8	8	7	698098d8	9	9	12	8b44f7af	10	10	17	ffff5bb1	11	11	22	895cd7be
12	12	7	6b901122	13	13	12	fd987193	14	14	17	a679438e	15	15	22	49b40821
16	1	5	f61e2562	17	6	9	c040b340	18	11	14	265e5a51	19	0	20	e9b6c7aa
20	5	5	d62f105d	21	10	9	02441453	22	15	14	d8a1e681	23	4	20	e7d3fbc8
24	9	5	21e1cde6	25	14	9	c33707d6	26	3	14	f4d50d87	27	8	20	455a14ed
28	13	5	a9e3e905	29	2	9	fcefa3f8	30	7	14	676f02d9	31	12	20	8d2a4c8a
32	5	4	fffa3942	33	8	11	8771f681	34	11	16	6d9d6122	35	14	23	fde5380c
36	1	4	a4beea44	37	4	11	4bdecfa9	38	7	16	f6bb4b60	39	10	23	bebfbc70
40	13	4	289b7ec6	41	0	11	eaa127fa	42	3	16	d4ef3085	43	6	23	04881d05
44	9	4	d9d4d039	45	12	11	e6db99e5	46	15	16	1fa27cf8	47	2	23	c4ac5665
48	0	6	f4292244	49	7	10	432aff97	50	14	15	ab9423a7	51	5	21	fc93a039
52	12	6	655b59c3	53	3	10	8f0ccc92	54	10	15	ffeff47d	55	1	21	85845dd1
56	8	6	6fa87e4f	57	15	10	fe2ce6e0	58	6	15	a3014314	59	13	21	4e0811a1
60	4	6	f7537e82	61	11	10	bd3af235	62	2	15	2ad7d2bb	63	9	21	eb86d391

3. An Architecture for MD5

The proposed architecture for MD5 is presented in Figure 1. The structure of the MD5 algorithm allows both iterative and pipelined implementations, because the MD5 step of Equation (2) is performed 64 times. With small modifications, the suggested architecture can be used for both iterative and pipelined designs. Pipelining means here that several of the MD5 steps are unrolled and then pipelined by adding registers between them.

Because of the sequential nature of the MD5 algorithm, only limited number of parallelism can be exploited. 64 MD5 steps of Equation (2) must be performed successively, because the output of the last step is used as an input for the next one. However, there are operations inside an MD5 step which can be performed in parallel.

The architecture of Figure 1 implements steps 3 – 5 of the MD5 algorithm presented in Section 2, i.e., the padding and length appending are left outside the design, because they are easy and fast to perform, and hence hardware acceleration is not needed for those steps.

The inputs of the architecture are the following: the message is given for the design with `data_in`, `address` and `load` signals. Width of `data_in` can be chosen freely, e.g. 32 bits were used for implementations presented later in Section 4. `Address` determines which bits of the 512-bit X_j are loaded. The data of `data_in` is loaded into the design when `load` is high. Processing of the algorithm is started with either `start_new` or `continue` signals. `Start_new` is used when a derivation of a new message digest is started, i.e. when X_0 is processed, and `continue` is used when a processing of a new X_j, for which $j \geq 1$, is started. For example, if a 1024-bit message is processed, `start_new` is used for the first 512 bits, and when the second 512 bits are processed, `continue` is utilized.

The counter in the architecture counts from 0 to 63 and it is reset to zero when a new derivation begins. When a derivation of a new message digest is started, A, B, C and D registers are initialized to the values given in Equation (1). Otherwise, values from the previous derivation are used. The second multiplexer is used in iterative architectures, where the same MD5_step block is utilized several times in a processing of the algorithm. The adder (actually four 32-bit adders) performs the additions of Equation (4). The 6-input AND-gate determines when the calculation of the algorithm is finished.

The MD5_step block calculates Equation (2). This block can be implemented also in a pipelined fashion, and then several MD5 steps are calculated in the block. The structure of MD5_step is discussed in Section 3.1. The inputs `T` and `X` of the block are 32 bits wide in an iterative design and in pipelined designs they are $32 \cdot p$ bits wide, where p is the number of pipelined stages. Analysis of the effects of pipelining is presented in Section 3.2.

TXs_register is a storage element for the message X_j and for constants T and s. It can be implemented using simple registers or internal memory of the target device, i.e. BlockRAMs in Xilinx devices. Effects of pipelining on this block are discussed in Section 3.2.

3.1. General MD5 Step

The heart of the architecture is the MD5 step block which calculates Equation (2). A block diagram of the

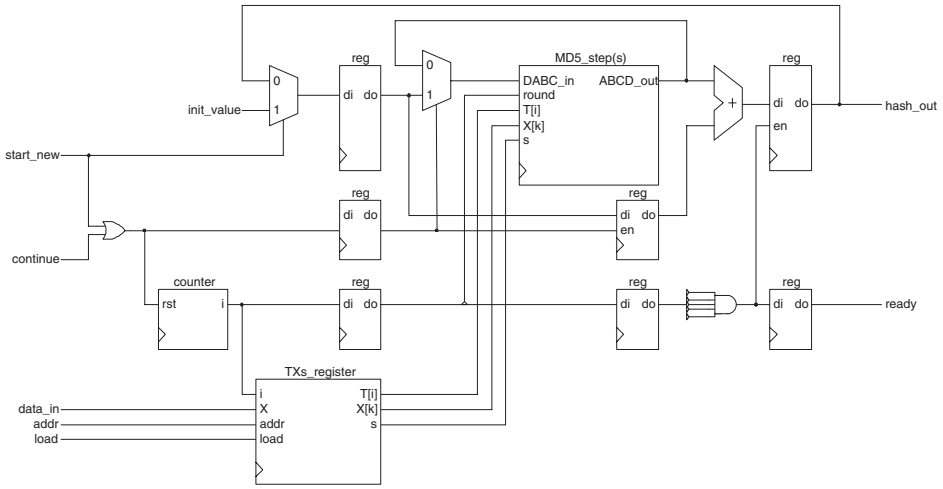


Figure 1. An architecture for MD5

MD5 step block is presented in Figure 2. This block is implemented so that it is performed in one clock cycle and it forms the critical path of the MD5 architecture of Figure 1. As can be seen from Equation (2) four 32-bit additions are required and also all the functions $F(X,Y,Z)$, $G(X,Y,Z)$, $H(X,Y,Z)$ and $I(X,Y,Z)$ must be implemented. However, these functions are easy to implement and fast to perform, because they are simple bitwise logical operations.

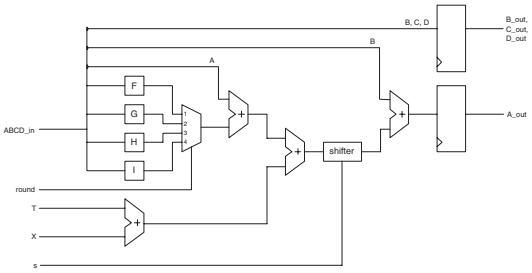


Figure 2. A general MD5 step

Although the shifting may first seem to be trivial, it is actually the most logic resources demanding operation in the MD5 step [2]. Also a significant part of the delay of the block consists of the delay of the shifter.

The MD5 step block was implemented on Xilinx Virtex-II XC2V4000-6 FPGA device and it required 328 slices. An estimated delay after synthesis was 11.574 ns.

3.2. Effects of Pipelining

As can be seen from the specifications of the MD5 algorithm presented in Section 2, the structure of a single MD5 step can be simplified by unrolling and pipelining several steps. Thus, the delay of an MD5 implementation may be reduced by pipelining the calculation of the MD5 steps. An analysis of the effects of pipelining on delay and area requirements of a design is presented in this section. Let p be the number of pipelined steps.

Pipelining simplifies the general MD5 step presented in Section 3.1 in many ways. The most obvious effect is to the functions of Equation (3). If the architecture is fully pipelined, only one function needs to be implemented instead of all four. If 32-stage pipelining is used, only two functions are required in a block.

As can be seen in Table 1, only four values of s per round are used. Thus, the shifter can be simplified by using 2- or 4-stage pipelining. Because most of the logic resources required in an MD5 step implementation are for the shifter, these simplifications have a significant effect on the area requirements of the block. The delay of the block is also slightly reduced when a simpler shifter is utilized.

Because of the structure of the MD5 algorithm, reasonable number of pipelined stages are $p = 1$ (an iterative design), $p = 2$, $p = 4$, $p = 32$ and $p = 64$ (a fully pipelined design). If some other number of pipelined stages, e.g. $p = 8$ or $p = 16$, is used, major enhancement in either area requirements or in delay is not achieved compared to the values given above, as can be witnessed later in this section.

In a fully pipelined architecture, a single MD5 step reduces so that only one of the functions has to be implemented. In that case, the shifter is reduced from the design, because shifting by a constant value can be performed trivially by rearranging the bit vector. The constant $T[i]$ can also be hardwired into the design. The architecture of the MD5 step in a fully pipelined design is presented in Fig-

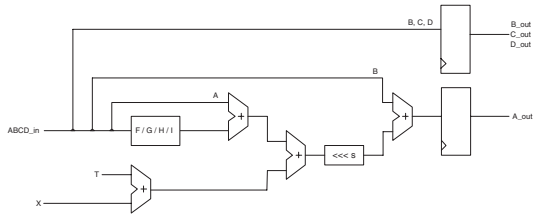


Figure 3. An MD5 step in a fully pipelined architecture

ure 3. In a fully pipelined architecture, there is no need for the multiplexer in front of the MD5_step block in Figure 1 and, hence, it can be reduced from the architecture.

When a single MD5 step is observed, pipelining reduces both delay and area requirements as presented in Figure 4 (a). The values in Figure 4 (a) were obtained using Xilinx Synthesis Tool 6.2 (XST) with high effort level without any constraints. Thus, these values are not exact values of how the architecture performs on a real device, but they give an overview of how pipelining affects on the area requirements and delay. As can be seen, considerable reduction in both area requirements and delay can be achieved in a single MD5 step, if the design is pipelined.

Although the area requirements per MD5 step decrease considerably, the overall area requirements naturally increase as a function of the number of pipelined stages. Approximated area requirements and delay of the calculation of all 64 MD5 steps is presented in Figure 4 (b). The values in Figure 4 (b) were calculated as follows: the slices value is calculated simply by multiplying the number of required slices of a single MD5 step by the number of pipelined stages p. The delay value is computed by multiplying the delay value of a single MD5 step by 64. A delay of 1 ns was added to the delay of a single MD5 step in the calculations, because the multiplexer in front of the MD5_step block in Figure 1 needs to noticed, as well. However, in the calculation of the fully pipelined architecture, this addition was not performed, because the multiplexer was reduced from the design, as discussed earlier. It should also be noticed that the area requirements given in Figure 4 (b) are only for the MD5 steps. In addition to this amount, also other blocks of the architecture in Figure 1 require area.

In Figure 4 (b), it can be seen that, if $p = 8$ or $p = 16$ pipelining does not reduce delay at all. Also the benefits of 32-stage pipelining are almost negligible, because only little faster performance is achieved with considerably larger area requirements. However, 2- and 4-stage pipelining seem to increase the area requirements only moderately, but yet they provide faster performance. The area requirements of the fully pipelined architecture are so high that it can be recommended only for applications, where very high performance is demanded.

Pipelining simplifies also the TXs_register block in Figure 1. However, the effects are not as dramatic as in MD5_step. In a fully pipelined design, the values of $T[i]$ are hardwired into the design and storage element is not needed. Generally, the use of internal memory of the FPGA device, e.g. BlockRAMs, is advantageous in iterative design or in design where only few pipelined stages are used. When many stages are pipelined, register-based implementation should be preferred, because several values must be read simultaneously, which makes the use of internal memory problematic. The register-based approach is advantageous also in ASIC-based (Application Specific Integrated Circuit) implementations, where memory blocks require a significant amount of area.

As a conclusion for the pipelining study, it can be mentioned that pipelining seems to reduce delay of the calculation of the MD5 algorithm at the expense of higher area requirements, as expected. However, final conclusions can be drawn only after the implementation process. These results are presented in Section 4.

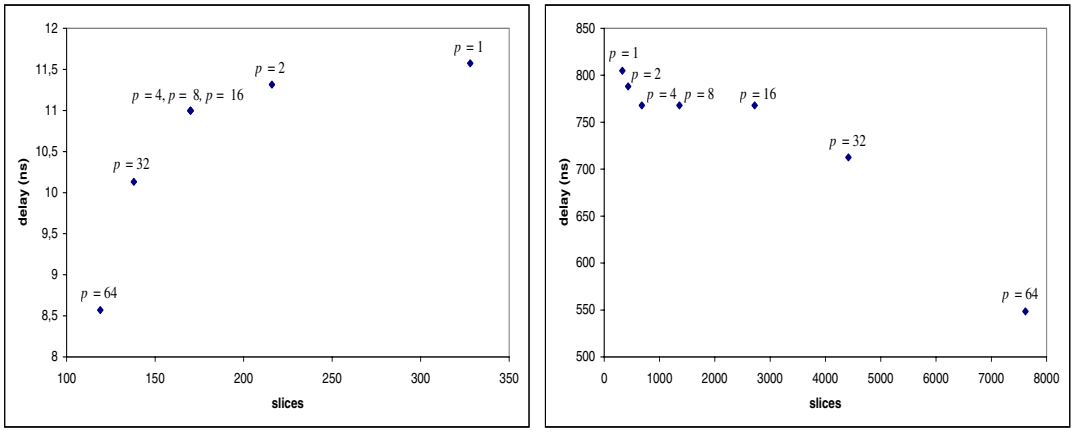
3.3. Methods to Increase Throughput

Although very fast performance can be achieved with the designs presented in the previous sections, their performance may fall short in certain very demanding environments, e.g. in heavily-loaded servers. In such environments, throughput of an implementation becomes a constraint. Thus, methods to increase throughput to gigabit-level are required.

Two different approaches to increase throughput are presented. Two commonly known methods are used for increasing throughput. The first and simpler one is to use parallel MD5 blocks. The second is to take advantage of the pipelined architectures presented in Section 3.2.

In parallel method, several MD5 blocks are used in parallel and each one of them can process MD5 calculations independently. Use of parallel MD5 blocks is a very straightforward way to increase throughput: if n parallel MD5 blocks are used, the achieved throughput is approximately n times the throughput of one MD5 block. The architecture for this method is very simple: several blocks of the architecture of Figure 1 are placed in parallel. Simple controlling logic is required, also.

The utilization of pipelined architectures is more difficult, because the use of the pipeline must be controlled. However, compared to the parallel MD5 block implementations, slightly faster and smaller implementations are resulted at the expense of reduced flexibility. The flexibility is reduced because MD5 calculations cannot be performed independently from each other. The upper limit for concurrent calculations is the number of pipelined stages p. For example, it is possible to alter the 16-stage pipelined



(a) (b)

Figure 4. Approximated area requirements and delay of a single MD5 step (a) and all 64 MD5 steps (b) in iterative and pipelined architectures. Value of p indicates the number of pipelined stages

architecture so that it can process 16 MD5 calculations simultaneously, i.e. one in every pipelined stage. In addition to the reduced flexibility, a problem in this method is that extra storage elements are required for the values of X_j. If one calculation is processed at the time, only 512 bits are needed, but if p calculations are processed, $512 \cdot p$ bits are required. In addition to the extra storage elements, also extra control logic is required. However, the proposed architecture of Figure 1 applies with only small modifications.

The methods presented above do not decrease the delay of a single calculation and, therefore, they do not speed up a single micropayment initialization, where MD5 rounds are calculated consecutively. The parallel or pipelined structures cannot be utilized inside a single initialization chain. However, several initializations can be calculated concurrently which, again, increases throughput.

4. Results of the Implementation

The MD5 architectures presented in Section 3 were implemented in VHDL. Aldec Active-HDL 6.2 was used in project management and simulations. Synthesis was performed with Xilinx Synthesis Tool XST 6.2 and the place & route was performed with Xilinx ISE 6.2. Xilinx Virtex-II XC2V4000-6 was used as a target device.

The VHDL source code for the SIG-MD5 designs was written very carefully in order to guarantee best possible performance. The focus was, especially, on the optimization of the MD5_step block which ultimately defines the performance of an MD5 implementation. The object was to utilize the 4-input LUT-structure as efficiently as possible so that the used area and delay would be minimized.

The synthesis was performed for all designs presented in Section 3, i.e. for iterative, 2, 4, 8, 16 and 32-stage and fully pipelined designs. The synthesis was performed with XST 6.2 with high optimization effort and speed as an optimization goal. The synthesis estimates of area requirements and delay are presented in Figure 5 (a).

Comparing Figures 4 (b) and 5 (a), it can be noticed that the observations made in Section 3 apply also after synthesis. Area requirements are expectedly higher, because only the area requirements of the md5_step(s) were included into the observations in Section 3.

The actual performance of the designs can be found out after the implementation which includes translation, mapping, place & route and timing. The implementation was performed with Xilinx ISE 6.2 and the results of the implementation are presented in Table 2.

Two versions of iterative, 2, 4 and 8-stage pipelined architectures were implemented. The first one exploits internal memory resources of the FPGA, i.e. BlockRAMs. The other one is implemented with simple registers. In FPGA-based implementations, the use of BlockRAMs is beneficial because it significantly reduces the number of required slices. However, in ASIC-based implementations register-based approach is more advantageous, because the gate count is a lot smaller as can be witnessed in Table 2. It should be noticed that, although the gate count is given in Table 2, it is an independent figure, which does not have an effect to other figures in the table. It is only an approximation of the required area on ASICs. The clock frequencies are for Virtex-II XC2V4000 in all cases. The throughput in Table 2 was calculated with the following formula [6]:

$$\text{throughput} = (\text{block size} \times \text{clock frequency})/\text{latency} \quad (5)$$

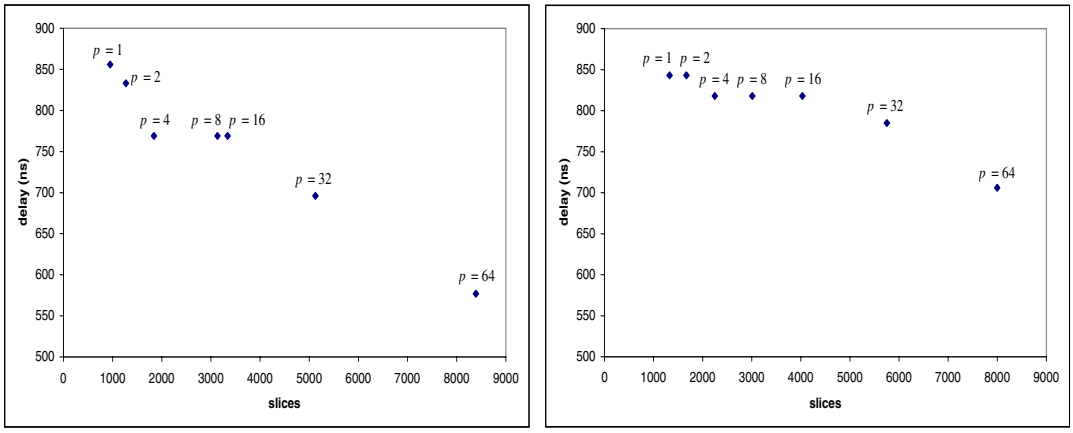


Figure 5. Area requirements and delay after synthesis (a) and final implementation results (b) of SIG-MD5 designs on Virtex-II XC2V4000-6 device. Value of p indicates the number of pipelined stages

Table 2. Results of the MD5 implementations on Xilinx Virtex-II XC2V4000-6

Design	Pipeline	Slices	BRAMs	Gate Count	Clock	Time	Throughput
iterative	1	647	2	143,897	75.5 MHz	875 ns	586 Mbps
iterative	1	1325	0	21,359	78.3 MHz	843 ns	607 Mbps
2-stage	2	826	4	279,296	75.9 MHz	869 ns	589 Mbps
2-stage	2	1670	0	31,200	78.3 MHz	843 ns	607 Mbps
4-stage	4	1057	8	545,869	78.2 MHz	844 ns	607 Mbps
4-stage	4	2248	0	43,051	80.7 MHz	818 ns	626 Mbps
8-stage	8	1893	8	560,999	78.2 MHz	844 ns	607 Mbps
8-stage	8	3011	0	61,727	80.7 MHz	818 ns	626 Mbps
16-stage	16	4031	0	81,191	80.7 MHz	818 ns	626 Mbps
32-stage	32	5752	0	118,803	84.1 MHz	785 ns	652 Mbps
full	64	7997	0	169,715	93.4 MHz	706 ns	725 Mbps

where block size is 512 bits, latency is 66 and clock frequency is given by the implementation.

The area requirements and delay of the final results are presented in Figure 5 (b). The values of the register-based designs were used in Figure 5 (b), because the area requirements are easier to represent if BlockRAMs are not used. Comparing Figures 5 (a) and (b), it can be seen that significant increase in delays occurs after the implementation for the designs where several stages are pipelined. Thus, only a very small enhancement in delay of a single MD5 round can be achieved by pipelining the design. Two, eight, and 16-stage pipelining should be avoided, because similar delays are achieved with smaller slice requirements.

Comparing Figures 5 (a) and (b) it can be seen that, although the post-synthesis results in Figure 5 (a) indicate that serious reduction in delay can be achieved with pipelining, only small enhancement is achieved after place & route. Theoretically, a more efficient place & route would result in better results, i.e. results which are closer to the synthesis results of Figure 5 (a). However, the structure of the design becomes a lot more complicated when many-stage-pipelining is used, thus, making the place & route process much more difficult.

Pipelining considerably increases the area requirements of an MD5 implementation, but only small decrease in delay is achieved. The speed/area ratio is observed by calculating throughput/slice values presented in Table 3. Pipelining decreases significantly the speed/area-ratio.

If the delay of a single MD5 calculation is observed, iterative design offers fast performance with minimal area requirements. Enhancement of only 19.4% can be achieved at the expense of over 500% increase in area requirements if the fully pipelined architecture is used.

The throughputs of Table 2 are calculated by assum-

Table 3. Achieved throughput per slice values for different architectures

Design	kbps / slice
iterative	458
2-stage	363
4-stage	278
8-stage	208
16-stage	155
32-stage	113
full	91

ing that the design can process only one MD5 calculation at once. However, if the throughput of an iterative design is insufficient for an application, pipelined architecture may be used for increasing throughput as described in Section 3.2. The results of the high-throughput designs are discussed in more detail in Section 4.1.

4.1. Results of the High-Throughput Designs

Two different approaches to increase throughput were discussed in Section 3.3. The first and simpler one used parallel MD5 blocks. The second took advantage of pipelined architectures.

Two implementations using parallel MD5 blocks were designed: the first, called SIG-MD5-HT4i, uses four parallel iterative MD5 blocks, and the second, SIG-MD5-HT10i, consists of ten iterative MD5 blocks. HT stands for High Throughput and 4/10 tells the number of parallel blocks and i is for iterative. SIG-MD5-HT4i requires 5845 slices and operates at a clock frequency of 78.3 MHz and SIG-MD5-HT10i occupies 11498 slices and 10 Block-RAMs and operates at 75.5 MHz. High throughputs are achieved with both designs: 2.32 Gbps with SIG-MD5-HT4i and 5.86 Gbps with SIG-MD5-HT10i.

An implementation using 4-stage pipelining was designed. It required 5732 slices and operated at 80.7 MHz achieving a throughput of 2.40 Gbps. This design was named SIG-MD5-HT4p where p stands for pipelined.

Almost similar results are achieved with both strategies. Pipelining can be efficiently utilized for achieving high throughput although the benefits of using it for reducing delay were only small as discussed in Section 4. However, using parallel MD5 blocks is easier to design and use.

5. Performance Comparison

For convenience, iterative and fully pipelined designs presented in Section 3 are renamed to SIG-MD5-I and SIG-MD5-FP, where SIG is an acronym for the Signal Processing Laboratory at Helsinki University of Technology and I stand for iterative and FP for fully pipelined. The high-throughput designs SIG-MD5-HT4p and SIG-MD5-HT10i, presented in Section 4.1, are also included into the comparison, although other similar designs have not been published at least to the authors' knowledge. Other open-literature designs in the comparison are the designs by Deepakumara et al. [2], Diez et al. [3] and Dominikus [5]. Two commercial designs by Amphion [1] and Helion Technology [8] have been also included into the comparison although only limited information of these designs is available. The designs are presented in Table 4.

SIG-MD5 designs perform very well in both required area and performance. SIG-MD5 designs are both fastest and smallest published academical designs, although an exact comparison to certain implementations is difficult, because different FPGA device families are used. The main reason for the good performance of the SIG-MD5 designs is most probably efficient use of design tools and, especially, careful low-level designing and optimization.

The design of Diez et al. is the fastest FPGA-based implementation previously presented in the literature. Because the FPGA device family is the same for both SIG-MD5 and Diez' designs, the comparison between them is straightforward. SIG-MD5-I is slightly smaller than the corresponding design by Diez et al. but it performs nearly 30% faster. Thus, SIG-MD5 designs perform better than any previously published design.

The comparison also shows that high throughput can be easily achieved with methods presented in Section 3.3. SIG-MD5-HT4p achieves over 570% higher throughput with only 20% larger area requirements than Deepakumara's fully pipelined design.

If SIG-MD5 designs are compared to commercial designs by Amphion and Helion Technology, it can be said that SIG-MD5 designs compete well also with these designs. The design by Amphion seems to be slightly smaller but slower, whereas Helion Technology's design is both smaller and faster. However, exact comparison is hard, because only limited information is available.

6. Conclusions

As the use of public-key cryptography and certain micropayment schemes using hash algorithms increase, more performance is required also from hash algorithm implementations. Thus, hardware acceleration may be required also for these algorithms. Especially, micropayment initializations requiring calculations of thousands of MD5 rounds may need to be accelerated.

An architecture for MD5 hash algorithm was presented. The architecture was designed so that it can be easily used for different kinds of MD5 implementations. For example, both very compact and fast designs can be implemented

Table 4. Comparison of published FPGA-based implementations. Commercial designs by Helion Technology and Amphion have been included into the comparison, but comparison to these designs is difficult, because only limited information of these designs is available

Design	Device	Slices	BRAMs	Clock	Latency	Time	Throughput
SIG-MD5-I	Virtex-II 2V4000-6	647	2	75.5 MHz	66	875 ns	586 Mbps
SIG-MD5-I	Virtex-II 2V4000-6	1325	0	78.3 MHz	66	843 ns	607 Mbps
SIG-MD5-FP	Virtex-II 2V4000-6	7997	0	93.4 MHz	66	706 ns	725 Mbps
SIG-MD5-HT4p	Virtex-II 2V4000-6	5732	0	80.7 MHz	66	818 ns	2395 Mbps
SIG-MD5-HT10i	Virtex-II 2V4000-6	11498	10	75.5 MHz	66	875 ns	5857 Mbps
Deepakumara et al. [2]	Virtex V1000-6	880	2	21 MHz	65	3095 ns	165 Mbps
Deepakumara et al. [2]	Virtex V1000-6	4763	0	71.4 MHz	n.a.	n.a.	354 Mbps
Diez et al. [3]	Virtex-II 2V3000	1369	n.a.	60.2 MHz	66	1096 ns	467 Mbps
Dominikus [5]	Virtex-E V300E	$\sim$2008	n.a.	42.9 MHz	206	n.a.	146 Mbps
Amphion [1]	Virtex-II	844	n.a.	60 MHz	n.a.	n.a.	472 Mbps
Helion Technology [8]	Virtex-II -6	613	1	96 MHz	n.a.	n.a.	744 Mbps

with the same architecture. This architecture was used in several designs, which are called the SIG-MD5 designs.

Effects of pipelining were discussed. In an MD5 calculation, 64 MD5 steps need to be calculated and the structure of one step can be significantly simplified by unrolling and pipelining. It was found out that reasonable numbers of pipelined stages are 1 (an iterative design), 2, 4, 32 and 64 (a fully pipelined design). Other numbers of pipelined stages do not decrease the delay of an MD5 calculation.

Although pipelining first seems to reduce delay significantly, only small enhancement in speed at the expense of a lot larger area requirements is achieved at the end. This slowdown originates from the grown area requirements and complexity of the designs, which make the place & route process significantly harder. However, although the delay of a single calculation is not reduced significantly, pipelining may be used efficiently for achieving high throughput.

A delay of 843 ns with logic requirements of only 1325 slices was achieved with an iterative design, SIG-MD5-I, on a Xilinx Virtex-II XC2V4000 FPGA, while for a fully pipelined design, SIG-MD5-FP, the values were 706 ns and 7997 slices, respectively. Corresponding throughput values are 607 Mbps for SIG-MD5-I and 725 Mbps for SIG-MD5-FP. At least to the authors' knowledge, these are the fastest published FPGA-based MD5 designs. The better performance of the SIG-MD5 designs compared to other published FPGA-based MD5 implementations was achieved, most probably, by careful implementation and efficient use of design tools.

Two methods to raise the throughput of an MD5 design were discussed. The first was to simply use parallel MD5 blocks and the second used a pipelined structure. Results of both methods are almost similar, but the first one is easier to design and use. A throughput of 2.40 Gbps was achieved with a design occupying 5732 slices and 5.86 Gbps was attained with 11498 slices and 10 BlockRAMs, respectively.

A study of MD5 implementation was performed and aspects of delay, throughput and area requirements were studied. Although the structure of the MD5 algorithm prevents large-scale parallelization of the operations of the algorithm, very efficient hardware implementations were attained. It was found out that MD5 can be implemented efficiently on hardware with very small logic resources. Because of the small area requirements of an MD5 design, a high-speed MD5 design can be included also into designs where computational resources are limited, e.g. into mobile applications. Very high throughput can be also achieved with moderate logic requirements.

Future Work Future work includes the use of the MD5 architecture presented in this paper for a micropayment initialization designed in the GO-SEC project at HUT. This initialization consists of several consecutive MD5 rounds (about 10,000) and it is slow to perform with software and, thus, hardware acceleration is required at least in heavily-loaded environments. FPGA board described in [9] will be used as an implementation platform for the design.

Hardware implementation aspects of the SHA-1 hash algorithm [7] will be studied and compared to the MD5 implementations presented in this paper. An architecture combining both MD5 and SHA-1 will be designed.

Effects of the recent NIST report [10] are studied. The report discussed flaws in the security of MD5 and several other hash algorithms. However, the security of SHA-1 is not threatened and replacement of MD5 with SHA-1 in the GO-SEC micropayment initialization is considered.

Acknowledgements This paper was written as a part of the GO-SEC project at Helsinki University of Technology. GO-SEC is financed by the National Technology Agency of Finland and several Finnish telecommunication companies.

References

[1] Amphion. CS5315, High Performance Message Digest 5 Algorithm (MD5) Core. Datasheet, URL: http://www.amphion.com/acrobat/DS5315.pdf, (visited September 9, 2004).

[2] J. Deepakumara, H.M. Heys, and R. Venkatesan. FPGA Implementation of MD5 Hash Algorithm. *Proceedings of the Canadian Conference on Electrical and Computer Engineering, CCECE 2001, Toronto, Canada*, Vol. 2:919 – 924, May 13 – 16, 2001.

[3] J.M. Diez, S. Bojanić, Lj. Stanimirovicć, C. Carreras, and O. Nieto-Taladriz. Hash Algorithms for Cryptographic Protocols: FPGA Implementations. *Proceedings of the 10th Telecommunications Forum, TELFOR'2002, Belgrade, Yugoslavia*, November 26 – 28, 2002.

[4] H. Dobbertin, A. Bosselaers, and B. Preneel. RIPEMD-160: A Strengthened Version of RIPEMD. In *Fast Software Encryption*, pages 71–82, 1996.

[5] S. Dominikus. A Hardware Implementation of MD4-Family Hash Algorithms. *Proceedings of the 9th IEEE International Conference on Electronics, Circuits and Systems, ICECS 2002, Dubrovnik, Croatia*, Vol. 3:1143 – 1146, September 15 – 18, 2002.

[6] A.J. Elbirt, W. Yip, B. Chetwynd, and C. Paar. An FPGA Implementation and Performance Evaluation of the AES Block Cipher Candidate Algorithm Finalists. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, Vol. 9:545 – 557, August 2001.

[7] Federal Information Processing Standards. Secure Hash Standard. *FIPS PUB 180-2*, August 1, 2002. With changes, February 25, 2004, URL: http://www.csrc.nist.gov/publications/fips/fips180-2/fips180-2withchangenotice.pdf, (visited September 9, 2004).

[8] Helion Technology. Datasheet - High Performance MD5 Hash Core for Xilinx FPGA. URL: http://www.heliontech.com/downloads/md5_xilinx_helioncore.pdf, (visited September 9, 2004).

[9] E. Korpela. Design of a Generic Reconfigurable Computing Platform. Master's thesis, Helsinki University of Technology, Signal Processing Laboratory, 2004.

[10] National Institure of Standards and Technology. NIST Brief Comments on Recent Cryptanalytic Attacks on Secure Hashing Functions and the Continued Security Provided by SHA-1, August 25, 2004. URL: http://csrc.nist.gov/hash_standards_comments.pdf, (visited September 9, 2004).

[11] R.L. Rivest. The MD4 Message-Digest Algorithm. *RFC 1320, MIT Laboratory for Computer Science and RSA Data Security, Inc.*, April 1992.

[12] R.L. Rivest. The MD5 Message-Digest Algorithm. *RFC 1321, MIT Laboratory for Computer Science and RSA Data Security, Inc.*, April 1992.

[13] B. Schneier. *Applied Cryptography*. John Wiley & Sons, Inc., second edition, 1996.

[14] Xilinx. *Virtex-II Platform FPGAs: Complete Data Sheet*, October 14, 2003. URL: http://direct.xilinx.com/bvdocs/publications/ds031.pdf, (visited September 9, 2004).