IPSJ Transactions on Advanced Computing Systems Vol. 1 No. 3 108–119 (Dec. 2008)

Regular Paper

Multi-stage Pipelining MD5 Implementations on FPGA with Data Forwarding

Hoang Anh Tuan,[†1] Katsuhiro Yamazaki[†1] and Shigeru Oyanagi[†1]

The MD5 (Message Digest 5) hash algorithm is useful for verifying the correctness and integrity of an arbitrary message, but the data dependency in the critical path in its iterations causes a huge computational delay and reduces the system's throughput. This paper describes three-stage and four-stage pipeline MD5 implementations ($3SMD5$ and $4SMD5$) on FPGA, which removes the data dependency in the iteration by the data forwarding method, and breaks that single step computation into 3 or 4 pipeline stages. The four-stage pipeline with both the keys and the constant table located in the BRAM could operate at the highest frequency, because its critical paths are shortened to one adder and some data movements at all stages. The processing of two messages in the alternative form enabled the four-stage pipeline architecture to achieve a higher frequency and throughput than related fine-grained pipelining architectures. Thus, the implementations achieve a good trade-off between the hardware size and the throughput.

1. Introduction

The MD5 algorithm, which is used to verify the integrity of a message, contains many iterations with data dependency inside. The huge computation delay generated in those iterations limits the entire throughput of the system.

MD5 was implemented on FPGA in commercial products [10)–14)] or researches [2)–5)]. They use coarse-grained-pipelining (dealing with multiple messages) [2),3)] or fine-grained (dealing with multiple iterations in one message) architectures [11)–14)]. The coarse-grained-pipelining increases the throughput by duplicating the hardware for simultaneous messages processes, and hence, the hardware size increases significantly. The fine-grained approach of commercial products gives a high throughput with an acceptable hardware size, and achieves a high rate of throughput/hardware size, especially in Helion [13)]. However, the data dependency in the critical path limits the number of pipeline stages (or generates pipeline stalls), therefore it limits the maximum frequency and throughput.

Our aim is to develop a finer-grained-pipeline architecture for hash algorithms implementations by dividing a single algorithm step into 3 or 4 subcomputations and mounting into pipeline stages, which increases the system's throughput for a corresponding small hardware size. The data dependency among operands in continuous steps, which limits the number of pipeline stages in fine-grained architecture, is removed by forwarding the required data from one step to another and by simultaneously processing two messages in an alternative manner. The algorithm contains 64 iterations, 3 or 4 stages each. The higher number of pipeline stages in a four-stage pipeline architecture helps reduce the critical path in each stage to as small as one adder with some data movements, and increases the final frequency, thus increases the throughput in comparison with the fine-grained related work. The hardware size is kept small due to no hardware duplication in comparison with coarse-grained architecture. It gives a good trade-off between the hardware size and the throughput (a high rate of throughput/hardware size).

This paper describes three-stage and four-stage pipeline MD5 implementations on FPGA, called $3SMD5$ [6),7)] and $4SMD5$ [8),9)] with different patterns, in which keys and constants are located in registers or Xilinx Block RAM (BRAM). Then, we show the effectiveness of our method by evaluating the implementations results in terms of hardware size, frequency, max throughput, and trade-off.

2. The MD5 Algorithm

2.1 Algorithm

The MD5 algorithm calculates a 128 bit digest from an arbitrary message through 4 steps of appending padding bits (Padding), appending length (Length), initialization (Init), and message compression (H_{MD5}) [1)] as shown in **Fig. 1**.

(1) Appending padding bits is used to guarantee that the total length of the appended message after adding a 64-bit length can be divided into 512-bit blocks by adding a single 1-bit and multiple 0 s to the original message.

(2) The 64-bit showing the length of the original message is added to the above result to make the final length become a multiple of 512.

†1 Graduate School of Science and Engineering, Ritsumeikan University

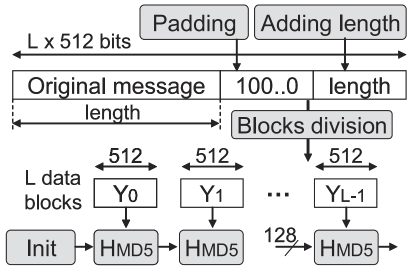

Fig. 1 MD5 algorithm.

(3) The initialization step starts the 128-bit message digest as 4 words of 32-bit, called A, B, C, D which are constants.

(4) Message compression (H_{MD5}) is the heart of the MD5 algorithm, which processes all 512-bit blocks (Y_0 to Y_{L-1}) of the padded message in sequence from the first one, and compresses them into a 128 bit message digest. The H_{MD5} handles the 512-bit input data as 16 keys (X_0 .. X_{15}), 32-bit each.

The compression algorithm consists of 4 rounds, and comprises 16 steps each as shown in **Fig. 2**. Each round uses a special function called F, G, H, and I, respectively.

$$F(B,C,D) = (B \wedge C) \vee (\neg B \wedge D) \quad (1a)$$
$$G(B,C,D) = (B \wedge D) \vee (C \wedge \neg D) \quad (1b)$$
$$H(B,C,D) = B \oplus C \oplus D \quad (1c)$$
$$I(B,C,D) = C \oplus (B \vee \neg D) \quad (1d)$$

T is a 64-element constant table and is used by 64 steps. X is the 16 keys memory (X_0..X_{15}). The key's location relies on the round (k_0, k_1, k_2, and k_3) and the internal step (i)[3].

$$k_0(i) = i \quad (2a)$$
$$k_1(i) = (1 + 5i)\ mod\ 16 \quad (2b)$$
$$k_2(i) = (5 + 3i)\ mod\ 16 \quad (2c)$$
$$k_3(i) = 7i\ mod\ 16 \quad (2d)$$

The main computation is represented by the equations (3)

$$A = B + ((A + Func + X[k] + T[i]) \lll s) \quad (3a)$$
$$A \leftarrow D;\ B \leftarrow A;\ C \leftarrow B;\ D \leftarrow C \quad (3b)$$

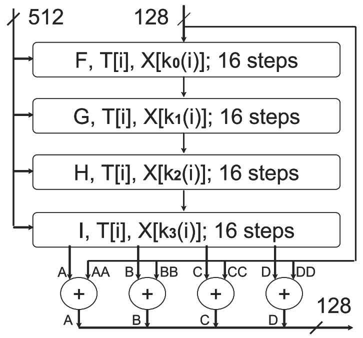
512
128
F, T[i], X[k0(i)]; 16 steps
G, T[i], X[k1(i)]; 16 steps
H, T[i], X[k2(i)]; 16 steps
I, T[i], X[k3(i)]; 16 steps
A
AA
B
BB
C
CC
D
DD
+
A
B
C
D
128

Fig. 2 Compression algorithm.

in which, $\lll$ represents a rotation shift left operation where s relies on the step number, and *Func* means functions F, G, H, and I in equations (1), based on the round number.

The final values A, B, C, D of the H_{MD5} are generated at the finalization stage by adding the results of 64 internal loops to the input values (AA, BB, CC, and DD, respectively) as shown in Fig. 2.

2.2 Pipelining Method

The main computation of the Hash algorithm contains several rounds, which use the same equations for computation. In each round many internal loops are included. So, two pipelining methods have been investigated.

The coarse-grained pipelining method mounts several steps to one stage. The hardware size will increase significantly together with the number of stages as shown in SIG-MD5[2].

The fine-grained pipelining divides a single step into small stages, and helps increase the frequency, and consequently the throughput of the main module. We apply this method in our MD5 implementations with two novelties of data forwarding, and two messages processing in an alternative manner. The data forwarding technique is used to break the critical path in the iterations into 3 stages in the three-stage pipeline design. The data forwarding technique together with the two messages processing in the alternative manner are used to divide the critical path into 4 stages in a four-stage pipeline design. The latter guarantees shorter critical paths to one adder and some data movements at all stages.

Consequently, the architecture achieves a higher operation frequency compared with related work.

3. Related Work

There is some related work on MD5 implementations. In this paper, we focus on hardware implementation in terms of hardware size, throughput, and trade-off of available commercial and research products.

3.1 Keywords definition

The bits/cycle rate is obtained by dividing the size of the data block by the number of clocks needed to process that data. It shows the efficiency of the computation cycle. Without pipelining, this number is 8 (512 bits divided by 64 cycles). In a fine-grained pipeline, some extra clocks are required, consequently there is a reduction of the bits/cycle rate. In the two-stage pipeline for 1 message architecture, this bits/cycle rate is 7.75 (512 bits divided by 66 cycles). In general, the higher the bits/cycle is, the smaller the number of pipeline stages the design has, so the longer the critical paths in the stages are. Thus, in no pipeline architectures such as Yiak[4), Wang[5), and Deep[3), OL[11), and Bisq[10), it is hard to increase the maximum frequency.

The throughput of the design is calculated by ((frequency*block size) / clocks per block).

The trade-off of the design is calculated by dividing maximum throughput by the occupied hardware slices. It expresses the efficiency of hardware used by the design.

3.2 Iterative and Full Loop Unrolling Architecture

The work of Deep[3) represents the iterative and full loop unrolling architectures, which the architectures are implemented based on the original algorithm. The original computations (3) are calculated exactly 64 times to generate the final result, so the architecture achieves a high bits/cycle rate of 8 (512 bits/64cycles). However, the huge computation makes it hard to increase the throughput over 354 Mbps on a Virtex device.

3.3 Coarse-grained Architecture

The coarse grained pipeline architecture[2) groups several steps together to form a pipeline stage. The common procedure is grouping by round, in which the same function, constant, and key's location function are used. Thus, the architectures duplicate the number of adders but keep the number of other function units as small as 1, 2, 4, 8, or 16. The whole algorithm computation for one message is finished after several stages (up to 64 when one repetition is mounted to one stage), and several messages (up to 64) fulfill all the stages to make the computation power. The hardware is duplicated to meet the demand. A high throughput of 5.8 Gbps is recorded for a SIG-MD5 system but the hardware use is also over 10 times higher than others (11,498 hardware slices on Virtex-2). This architecture can be used in extremely high speed security systems, which require a high throughput without any restriction on the hardware size and the power consumption.

3.4 Available Commercial Products

The commercial products[10)–14) come in form of IPs from some companies such as ALMA[12), Bisq[10), Helion[13). The work carried out by Bisq[10) and OL[11) seems to utilize a normal MD5 architecture (no pipeline at all), which requires 64 and 65 cycles to finish one block, and get the bits/cycle rate of 8. Hence, it uses a small hardware size but the throughput is also low. Other commercial products[12)–14) seem to utilize a two-stage pipelining for a single step approach (the algorithm computation contains 64 steps, two pipeline stages in each step) to implement high throughput MD5 cores. They utilize 66 cycles to process a 512-bit data block, which means they can process 7.75 bits/cycle. The pipeline architecture helps them increase the maximum frequency, thus, the maximum throughput with a slight increase in hardware size. The best one, Helion[13), seems to contain two adder-levels in each stage, with a high trade-off of 1.48.

3.5 Integrated Architecture

The work of Yiak[4) and Wang[5) show the integrated architecture for both SHA-1 and MD5. They also achieve the bits/cycle rate of 8 due to no pipeline design. The complexity in the controller, the multiplicity of functions make the work large in hardware size with a low throughput, but it can be used in multi security environments.

4. Data Dependency, Forwarding, and Dependency Removal

4.1 Data Dependency

Data dependency in equations (3) can be easily seen if we rewrite it as follows:

$$tempB = B \tag{4a}$$

$$B = B + ((A + T + X + Func) \ll s) \tag{4b}$$

$$A \leftarrow D;\ C \leftarrow tempB;\ D \leftarrow C \tag{4c}$$

Equations (4) show that the values of A, C, D rely on previous values of D, B, C, respectively. The graph in **Fig. 3** shows the data dependency in computing the new value of B between the two continuous steps. The new value of B, which is calculated by equation (4b) relies on previous values of A, B on current, values of T, X, s and Func. X itself relies on its location denoted by k, which must be calculated from the step number i. Func depends on the previous values of B, C, D and the current step i. Figure 3 also shows that the new value of B (new B) completely depends on the step number and current values of A, B, C, and D. However, the internal values of T, k, X can be pre-computed because they rely on the step number i only. Therefore, we can make a pipeline by pre-computing k, and if we can pre-compute the value of A, the left half of Fig. 3 can also be pre-computed.

4.2 Data Forwarding and Dependency Removal

The locations of operands A, B, C, and D at each step are required for the forwarding operations. **Figure 4** shows the data movement of A, B, C, and D using equations (4) within 4 steps, in which n shows the values that change at each step. In order to compute the new value B (n3) using equation (4b), the current value A is required. Assume the values of operands A, B, C, and D at two steps before are a, b, c, and d, respectively. The values for A at the current and the preceding steps are transferred from operands D, and C respectively, which means the values of current A for the n3 computation are located in D, or C depending on the step number where we want to use it. In short, the values of A used in equation (4b) can be taken as values of the operand D at the preceding step or C at the step before the preceding one.

In this $3SMD5$ and $4SMD5$ implementations, we manage to implement a single step of MD5 into a pipeline based on the data dependency in equations (4). As can be seen in Fig. 3, the computation of B requires a huge sequenced computation of k, X, Func and four 32-bit adders. This generates an enormous latency. However, if we pay attention to the trace of A in Fig. 4, that latency can be divided into smaller stages. The address of the key of the current step can be pre-computed several steps before, because it relies mainly on the step number i. The trace of A in Fig. 4 allows us to define the value of A at the current step as D at the preceding step or C at the step before the preceding one. All that makes it possible to pre-compute A+T+X up to 3 steps before. In other words, the data dependency of the computation of B in the operand A is removed. The new value of B now relies on the value of C at the step before the preceding one. The pre-computation of A+T+X in some pipeline stages by forwarding the value of C or D to A helps ease the delay in the critical path of (4b).

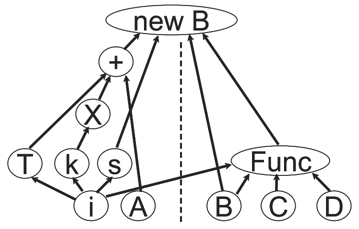

Fig. 3 Data dependency in a single step.

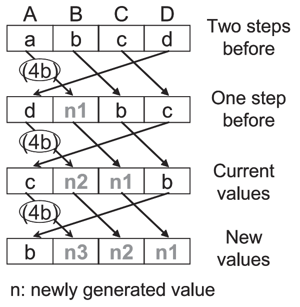

Fig. 4 Trace of A within 4 steps.

5. Three-stage Pipelining Implementation

5.1 Three-stage Pipelining

In order to divide the MD5 algorithm into a three-stage pipeline, equations (4) are re-written into

$$tempB = B \tag{5a}$$
$$AT = A + T[i] \tag{5b}$$
$$ATX = AT + X[k] \tag{5c}$$
$$B = B + ((ATX + Func) \ll s) \tag{5d}$$
$$A \leftarrow D;\ C \leftarrow tempB;\ D \leftarrow C \tag{5e}$$

in which, the current value of A, used for computing the new value of B (n3 in Fig. 4) is forwarded from the step before the preceding one from C. This data forwarding allows (5b) and (5c) to be computed one step earlier and two steps earlier, respectively.

The method to divide the computations of equations (5) into 3 pipeline stages of AT, ATX and Fold requires data forwarding for operand A as shown in **Fig. 5**. The AT stage computes the value of A+T ((5b)) and the address of the key. Then, the ATX stage computes A+T+X[k] value ((5c)). Finally, all values are gathered to form B in the Fold stage ((5d)). At the same time, the data movement among A, C, D occurs ((5a) and (5e)) as shown in Fig. 4. The biggest latency occurs at the Fold stage with the Func, shift and add operations.

Figure 6 shows the operations of those pipeline stages inside 64 steps of the Hmd5 algorithm. When a compression of a key is computed, it first goes to the AT stage for the preparation of A plus a constant. At the ATX stage, the

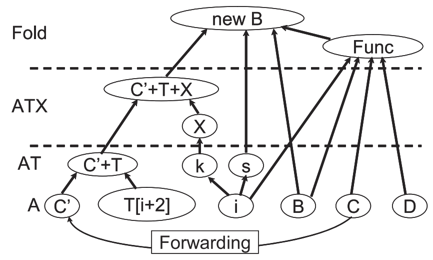

Fig. 5 Data forwarding and pipelining for main computation.

data provided by the AT is added to the required key. At this time, the AT module is used for the next data preparation. The same operation occurs when data move into the Fold stage, and the final result of the required step is given, while the ATX and AT modules are used to prepare values for the compression computations of the next step and two steps later, respectively. In order to complete the compression of one block of data, 66 clocks are required from the starting time.

5.2 Data and Pipeline Stage Implementation

There are 4 different data with different lengths involved in the design of the MD5 algorithm. They include the keys, constant values, shift values, and the message digest data. In the implementations, the digest data is designed as matrices of registers that contain 4 elements, while the keys and 64-element constant table can be located in a BRAM or the registers. The rotation values (shown by s), used for the rotation operations, are specified as a 16-case rotation unit. Other intermediate data are specified as 32-bit registers. The pipeline stages are implemented with two pipeline registers called AT and ATX. **Figure 7** shows the data movement and operations among A, B, C, and D with pipeline registers. The register AT is computed at the AT stage by (5b), in which A is replaced by C (forwarded from C in other words). ATX value is computed at the ATX stage by (5c). So, at the current step, the values of A, B, C, D, and A+T+X are available for the equation (5d).

5.3 The Compression Function Implementation

The three-stage pipeline implementation of the compression module Hmd5 ($3SMD5$) contains 3 stages: AT, ATX, and Fold as shown in **Fig. 8**. Those stages communicate with each other using registers AT, k and ATX. The register AT is used to store the value of C+T value, which means A+T at the Fold stage due to data movements from C to D in AT, then D to A in ATX. The

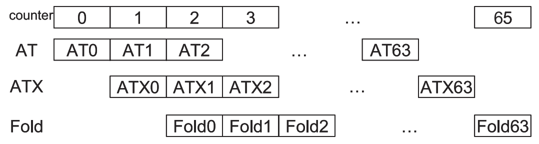

Fig. 6 Three-stage pipeline operations.

location of the key is stored in the register k, which means k(i+2) at the AT stage. ATX register represents the addition of those 3 data for the current step. The registers AT and k are used to connect the AT and ATX stages, while the register ATX is used to connect the ATX and Fold stages.

The AT stage contains two computation units of 32-bit adder and k(i). The adder simply adds two 32-bit data of A and a constant T[i] in equation (5b). The k(i) module is used to generate the key's location in a register or the corresponding key address in the BRAM using equations (2). However, this stage occurs two steps (clocks) before the current Fold stage. Hence, the counter must be i+2 in the T[i] and k(i) module, while A is taken as C following the trace shown in Fig. 4. The multiplexer with the input of A, C, D in Fig. 8 at this stage is used to solve the problem of equation (5b) at the first 3 steps, when the values of A are located in A at the first, D at the second, and C at the third steps. After this stage, the value of A+T and the address of the corresponding key are stored and transferred to the ATX stage. The adder in ATX adds the key at the address shown by k with the value of AT, and stores the result into the ATX register. The Fold stage completes the computation of equation (5d) using $Func(B, C, D)$ module, $\lll s$ module that specifies the rotation shift for s bits by utilizing a 16-case multiplexer, and two 32-bit adders. The result of this is written back to the digest value A, B, C, and D following equation (5e).

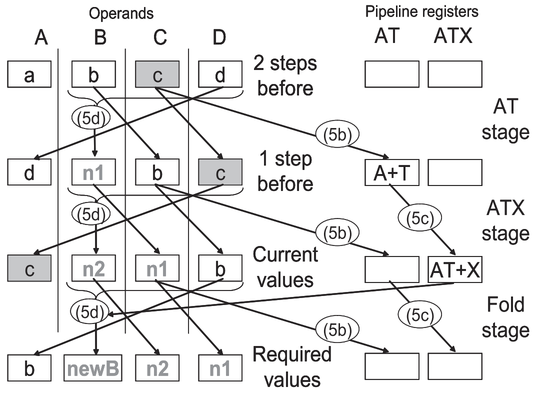

Fig. 7 Data movement with pipeline registers.

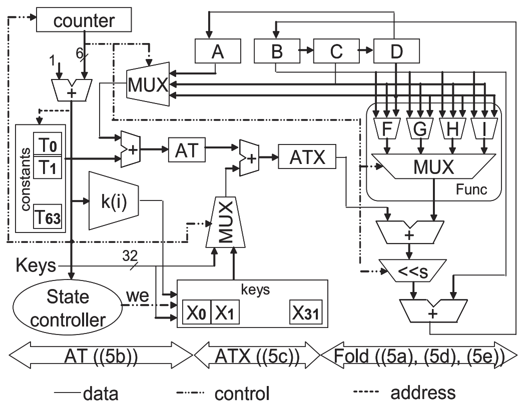

Fig. 8 Three-stage pipeline design of HMD5.

Two implementations are given. The $3SMD5$ locates all keys and constants inside the design in registers, while the $3SMD5^1$ locates the keys in the BRAM. In addition, the Func module are combined with the shift module using 4 parallel adders and a direct wire selection in order to decrease the delay of the Fold stage in the $3SMD5^1$ version.

6. Four-stage Pipelining Implementation

As shown in Fig. 8, the hardware size as well as the number of computation layers in the Fold stage is much larger in comparison with others. It generates an extra delay to the system, and reduces the throughput. Therefore, this stage should be broken into smaller ones to make balance in the computation time, and to increase the throughput of the system.

6.1 Data Dependency in Four-stage Pipelining

In order to break the MD5 main computation into 4 stages, the equations (5) are rewritten:

$$tempB = B \tag{6a}$$
$$AT = A + T[i] \tag{6b}$$
$$ATX = AT + X[k] \tag{6c}$$
$$S = (ATX + Func) \lll s \tag{6d}$$
$$B = B + S \tag{6e}$$

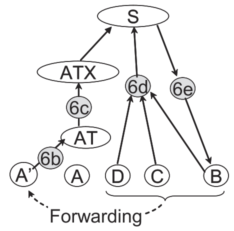

Fig. 9 Data dependency among operands in equations (6).

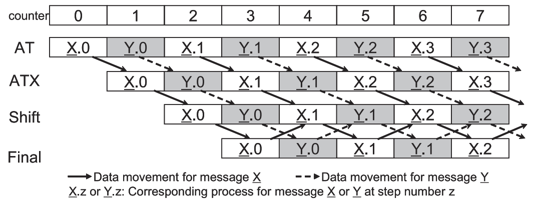
counter
AT
ATX
Shift
Final
Data movement for message X
Data movement for message Y
X.z or Y.z: Corresponding process for message X or Y at step number z

Fig. 10 Four-stage pipelining with two alternate messages.

$$A \leftarrow D;\ C \leftarrow tempB;\ D \leftarrow C \tag{6f}$$

in which equation (5d) is broken into (6d) and (6e). As proved from equations (5) and Fig. 4, equations (6b) and (6c) have very little data dependency on the previous values of the operands A, B, C, and D because we can forward the values of D, C, and B into A for advanced use at different time. However, equations (6d) and (6e), which are used to compute a new value of the operand B, are heavily dependent on the previous value of B. In other words, (6e) has data dependency on (6d) and vice versa. Hence, no pre-computation can be done for these two equations.

Figure 9 shows the data dependency among the operands in equations (6). It clearly shows that AT and ATX can be pre-computed any time by forwarding B, C, and D to A, but S and B ((6d) and (6e)) must be computed one after the other.

6.2 Four-stage Pipelining with Two Alternative Messages

The four-stage pipelining with two messages contains 4 stages of AT, ATX, Shift and Final as shown in **Fig. 10**. The two messages are computed alternatively for each step, in which, X.z and Y.z show the message X and Y at the step number z. The Y message process is added to utilize the spare time generated by the data dependency of S into B. The two messages can be processed after 128 clocks (64 steps for each).

The value AT is calculated by equation (6b) before being transferred to the ATX stage ((6c)). At that time, the AT module turns to compute value AT for the next step. The same thing occurs when computing the value S at the Shift stage ((6d)). However, the Final stage ((6e)) occurs in a different way. It is caused by a mutual data dependency between S and B (equations (6d) and (6e)) as shown in Fig. 9. The Shift and the Final stages for one message must be executed like in the white blocks in Fig. 10. One more clock is required here to complete the computation, consequently it generates a spare time (marked by the dimmed blocks) of all stages in every two clocks. In order to increase the throughput of the main computation module, the spare time generated by a data dependency between S and B is used to calculate the message digest of another message in an alternate form.

6.3 Pipeline Registers and Data Movement

In order to realize the design into hardware, locations of operands, data movement, and operations of pipeline registers are required. **Figure 11** shows the data movement in a four-stage pipeline with two messages, X and Y. In order to process two alternate message digests, the message digest operands A, B, C, and D are extended into 64 bits each. In general, the high 32 bits are used for the message X, while the remaining low bits are used for the message Y.

Since it supports for the computation of equation (6e) at the first step, the computation of (6b) must start 3 steps before by adding a_0 (message digest A of message X) with the constant. The computation starts from 0, counts up and ends at 127 (128 clocks). When the final A (A_{63}) appears at B, the finalization (A+AA in Fig. 2) starts together with the next generation of final B, C, and D.

In order to process the two messages X and Y in an alternative computation manner given in Fig. 10, their operands in low and high parts must be swapped during the normal data movements. Hence, two types of data movements are combined in Fig. 11: the shift from high parts to low ones (to swap the target messages) and the normal movements following equation (6f). The combination

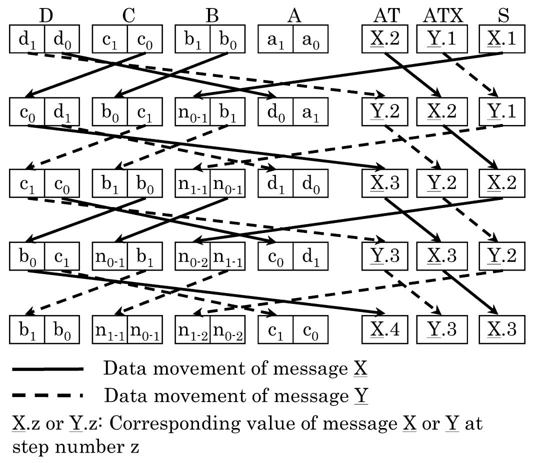
D
C
B
A
AT
ATX
S
Data movement of message X
Data movement of message Y
X.z or Y.z: Corresponding value of message X or Y at step number z

Fig. 11 Data movement in four-stage pipeline with two alternate messages.

makes data move from low parts of B, C, and D to high ones of C, D, and A, respectively. It also defines that the final result of (6e) must be written into the high part of B. The input operands for each pipeline stage are also selected from the low/high parts of operands based on the stages and locations of the corresponding data. Thus, all modules have input and output data at the same locations (registers). It allows us to remove many multiplexers for an efficient implementation.

6.4 Implementation

Figure 12 shows the hardware implementation of the four-stage pipelining MD5 ($4SMD5$) with two messages computed alternatively.

In comparison with the $3SMD5$ version, the message digest operands and the key memory sizes are doubled from four 32-bit operands and sixteen 32-bit keys to four 64-bit operands (A, B, C, and D in Fig. 11) and thirty-two 32-bit keys. The capacity of the counter is doubled by adding one more low-bit for message control which we denote MC-bit. The counter operates by incrementing itself by one every clock (into MC-bit). Then, the MC-bit is extracted for key control, while the remaining 6 bits are used in the same manner with the previous implementation of $3SMD5$.

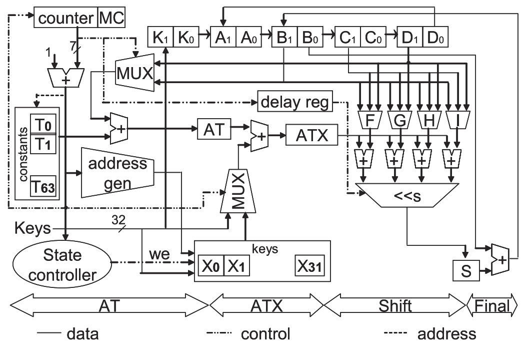

Fig. 12 Four-stage pipeline design with two messages.

The BRAM, used to store the 32 keys, is divided into two parts; the 16-element lower part is used for the first message, while the higher part is used for the second one. The address generation module (address gen) combines the result of k(i) with the MC-bit of the counter to generate the address for the corresponding key.

Besides, the Fold stage is divided into the Shift and Final stages. The Shift stage is in charge of computing functions and the shift value in equation (6d), and the Final stage is used to compute the final addition for B shown by equation (6e). The data movements among the operands follow values given in Fig. 11.

The AT stage contains a 32-bit adder and the key address generation. The key address generation module combines the result of k(i) with the MC-bit to generate the physical address of the corresponding key and message. The adder simply adds the high 32-bit of D, which represents the value of A (for that message two clocks later) and constant T[i] in equation (6b) together. After this stage, the value of A+T is stored and transferred to the ATX stage. The address of the corresponding key is also given for the next stage preparation. The 32-bit adder in ATX adds the value of the key from the external bus (Keys), which is selected by address gen, to the value of AT register before storing the result into the ATX register. The Shift stage completes the computation of equation (6d) using 4 adders in combination with a multiplexer module and 4 logic functional modules F, G, H, and I. The 4 functions work with the input data located in the high half of the message digests registers B, C, and D. The result of the Shift

Table 1 Hardware size and performance.

Name	Devices	Hardware slices	No. of BRAMs	Max Freq (MHz)	MaxThroughput (Gbps)	Bits/cycle	Trade-off
$3SMD5$	Virtex-2	1,010	0	88.0	0.68	7.64	0.67
$3SMD5^1$	Virtex-2	885	1	96.1	0.75	7.64	0.85
$4SMD5^1$	Virtex-2	1,064	1	137.6	1.04	7.75	0.98
$4SMD5^2$	Virtex-2	997	1	157.4	1.26	7.75	1.26
$4SMD5^2$	Virtex-4	1,008	1	182.8	1.44	7.75	1.43

^{1}: keys are located in BRAM
^{2}: keys and constant table are located in BRAM
Trade-off: Max throughput (Mbps) divided by hardware slices

stage is written into pipeline register S for the Final stage, which simply adds the value in register S with the value of B (located at the low part of register B) in equation (6e) to generate the intermediate message digests. The result is written into the high part of register B for the swapping purpose as shown in Fig. 11.

Two implementations are given, $4SMD5^1$ and $4SMD5^2$. In $4SMD5^1$ design, the keys are located in the BRAM, while the constant table is located in the registers. In $4SMD5^2$ design, both the constant table and the keys are located in the BRAM.

7. Implementation Results and Discussions

7.1 Implementation Results

The pipelined MD5 design was implemented on the Virtex-2 and Virtex-4 devices and compiled by Xilinx ISE 8.1 version. **Table 1** shows the implementations results in terms of hardware size, speed, throughput, rate of bits/cycle, and trade-off.

(1) Hardware size and performance

In $3SMD5$, all the keys and constants were stored in the registers inside the hardware. Thus, the hardware size remained large with 1010 hardware slices in use. It reduced significantly to 885 slices if the keys were located in the BRAM in the $3SMD5^1$. Besides, the improvement in the Func module, which utilizes 4 parallel adders and a multiplexer for the shifter, helped increase the speed of the compression module, hence increased the throughput 10.3% up, and achieved 750 Mbps in $3SMD5^1$.

Table 2 Hardware size and performance of related work.

Name		Devices	Hardware slices	No. of BRAMs	Max Freq (MHz)	Max Thpt (Gbps)	Bits/cycle	Trade-off
Researchers' work	SIG [2)]	Virtex-2	11,498	10	75.5	5.85	-	0.51
	SIG [2)]	Virtex-2	5,732	0	80.7	2.40	-	0.42
	Yiak [4)]	Virtex-2	797	-	96.0	0.77	8.00	0.97
	Wang [5)]	ASIC	-	-	-	0.52	8.00	-
	Deep [3)]	Virtex	4,763	0	71.4	0.35	8.00	0.07
Commercial products	Helion [13)]	Virtex-4	641	1	122.0	0.95	7.75	1.48
	Alma [12)]	Virtex-4	683	1	118.0	0.92	7.75	1.35
	OL [11)]	Virtex-2	614	1	62.0	0.49	8.00	0.80
	Bisq [10)]	Virtex	-	-	50.0	0.40	8.00	-
	Media [14)]	-	-	-	-	-	7.75	-

Max Freq: Maximum frequency
Max Thpt: Maximum throughput

In terms of the efficiency of the computation cycle, the 3 extra clocks required in our $3SMD5$ make that rate be 7.64 (512 bits divided by 67 clocks). The 4 extra clocks required to process two messages in our $4SMD5$ make the rate of 7.75 (1024 bits divided by 132 clocks).

(2) Three-stage and four-stage pipeline comparison

The four-stage pipeline with keys and a constant table located in the BRAM achieves the best performance because it has more balanced computation time among the stages. In $4SMD5^1$, the throughput increased nearly 39% compared with $3SMD5^1$, and achieved 1.04 Gbps, while the hardware size increased only 20% due to the hardware complexity of the messages digest registers. The reduction in the number of registers used in $4SMD5^2$ by utilizing the BRAM also helped reduce 6.3% of the hardware size. Thus, the maximum throughput increased 21% in comparison with the $4SMD5^1$ in Virtex-2.

7.2 Comparison with Related Work

The effectiveness of our implementation can be seen by comparing with related work. **Table 2** shows other significant results in both commercial products and researchers.

(1) Work of researchers

Among the system given by researchers, the SIG-MD5 [2)] has the best throughput (2.40 and 5.85 Gbps), but requires a huge hardware size. Hence, the trade-off of this design (0.51) is not good in comparison with ours on the same device (1.26). It can be used in an extremely high throughput system with no restriction on the hardware size. The work of Yiak [4] achieves a high bits/cycle rate due to the absence of any pipeline implementation. Architectures without any pipeline such as those in Yiak [4], Wang [5], and Deep [3] make it hard to increase their maximum frequency. Hence, our $4SMD5$-designs and implementations achieved a better throughput and trade-off than them.

(2) Commercial products

In comparison with related commercial products, our $4SMD5^2$ achieved better results than almost related work, except the Helion [13]. Helion [13] achieves a high throughput (946 Mbps) with a good trade-off (1.48) and an acceptable hardware size. On the same device, our work of $4SMD5^2$ achieved a higher throughput (1.44 Gbps) with nearly the same trade-off of 1.43. Hence, this design can be used with embedded systems that require a high throughput with an acceptable hardware size.

The trade-off in this research focuses on the hardware slices of the design but not the BRAMs and supported modules such as ports and data bus for messages transmission. Two Helion cores can simultaneously process two messages, and achieve 32% higher in throughput than our four-stage pipeline with keys and constant located in BRAM ($4SMD5^2$), while occupying 27% more in hardware size. However, the number of BRAM and supported modules such as the bus, ports are doubled. Based on our experiment, 65 slices are required for 64-element constant table if the table is implemented in a hardware instead of a BRAM. Hence, 1347 hardware slices and one BRAM are required to achieve 1.9 Gbps by using two Helion cores. The total trade-off of using two Helion cores, together with all supported modules, will be smaller than 1.41, so lower than our $4SMD5^2$.

7.3 Mixing Fine-grained and Coarse-grained Pipeline Architecture

The improvement by mixing coarse-grained pipeline in the message level with fine-grained pipeline at the iteration level can help the design achieve a throughput close to that of SIG-MD5 [2] with a reasonable hardware size. Four coarse-grained pipeline stage respond for 4 rounds, 16 iterations each. Each iteration in the coarse-grained stage contains 4 fine-grained pipeline stages as shown above. Each round simultaneously deals with two messages in an alternative manner. The design allows the processing of 8 messages simultaneously. Thus, the throughput is expected to be 4 times higher than that of the fine-grained four-stage pipeline design, and gets 5.76 Gbps. The hardware needed for adders and immediate message digests will be 4 times higher, and occupies about 4000 slices with 4 BRAMs. Thus, this approach can achieve a throughput close to that of SIG-MD5 with a reasonable hardware size, although the design and the verification are complicated.

Table 3 Delay time of each stage on Virtex-2.

Stage	AT (ns)	ATX (ns)	Fold (ns) Shift	Fold (ns) Final	Total delay (ns)
$3SMD5^1$	2.6	2.6	10.76		11.35
$4SMD5^1$	2.6	2.6	1.63	5.85	7.90
Normal MD5	-				17.38

7.4 Delay Time and Discussion

In order to compare the computation time among stages as well as the effectiveness of the new pipeline architecture, other versions for the $3SMD5^1$ and $4SMD5^1$, in which stages were clearly separated by modules, were implemented. **Table 3** shows the delay of each module inside as well as the total delay of the $3SMD5^1$ and $4SMD5^1$ in comparison with the delay of a normal MD5 without any pipeline technology. In the same implementation condition, breaking the single step shown in Fig. 3 into 3 small stages helps decrease the delay of a normal MD5 35% down in the $3SMD5^1$. Then, the big computation time module, Fold, when broken into two in the $4SMD5^1$ also helps decrease the delay a futher 30% down in comparison with the $3SMD5^1$. However, the hardware size slightly increases from 972 slices of the normal MD5 (without the BRAM) to 1,010 and 1,064 slices due to the overhead of pipelining in $3SMD5^1$ and $4SMD5^1$, respectively.

In the $4SMD5^1$, the delay in the Final stage, caused by the final data movement, is much bigger than other delays. Thus, this stage should be improved to achieve a higher throughput, and so, improves the rate of maximum throughput per hardware slice.

8. Conclusion

This paper has described the three and four-stage pipeline designs and implementations of the MD5 algorithm. The four-stage pipeline with both the keys and the constant table located in the BRAM could operate at the highest frequency. This is because its critical paths are shortened to one adder and some data movements at all stages. Processing two messages in an alternative form enabled the four-stage pipeline architecture to fulfill all its stages and achieve a higher frequency and throughput than related fine-grained pipelining architectures. Thus, this fine-grained pipelining architecture helps achieve a good trade-off between the hardware size and the throughput. Furthermore, the architecture allows an improvement in the mixing of coarse-grained and fine-grained pipeline architectures.

Acknowledgments We would like to express our thanks to Professor Izumi for many discussions and much information in this field. We also want to give our thanks to all the reviewers, who gave us many valuable comments to improve this paper. Finally, we want to thank the English checker for his comments to polish up our paper.

References

1) RFC 1321: The MD5 Message-Digest Algorithm.
2) Jarvinen, K., et al.: Hardware Implementation Analysis of the MD5 Hash Algorithm, *Proc 38th IEEE International Conference on System Sciences-2005* (2005).
3) Deepakumara, J., et al.: FPGA Implementation of MD5 Hash Algorithm, Proc of the Canadian Conference on Electrical and Computer Engineering, *CCECE 2001*, Vol.2, pp.919–924 (2001).
4) Yiakoumis, I., et al.: Efficient Small-Sized Implementation of the Keyed-Hash Message Authentication Code, *Eurocon2005*, pp.1875–1878 (2005).
5) Wang, M.-Y., et al.: An HMAC Processor with Integrated SHA-1 and MD5 Algorithms, *Proc ASP-DAC 2004*, pp.456–458 (2004).
6) Tuan, H.A., Yamazaki, K. and Oyanagi, S.: Three stages pipelined MD5 implementation on FPGA, *FIT2007*, LC-008, pp.61–64 (2007).
7) Tuan, H.A., Yamazaki, K. and Oyanagi, S.: Pipeline MD5 Implementations on FPGA with Data Forwarding, *IEICE Technical Report*, RECONF2007-27, pp.71–76 (2007).
8) Tuan, H.A., Yamazaki, K. and Oyanagi, S.: Four-stage Pipelining for Two Messages in MD5 Implementation with Data Forwarding, *70th National Convention of IPSJ*, 1N-6 (2008).
9) Tuan, H.A., Yamazaki, K. and Oyanagi, S.: Multi-stage Pipelining MD5 Implementations on FPGA with Data Forwarding, *The Sixteenth IEEE Symposium on Field-Programmable Custom Computing Machines* (FCCM'08) (2008).
10) Bisquare Systems Private Ltd. Available: http://www.bisquare.com
11) Ocean Logic Ltd. Available: http://www.ocean-logic.com
12) ALMA Technologies. Available: http://www.alma-tech.com
13) Helion Technology Ltd. Available: http://www.heliontech.com
14) Jetstream Media Technologies. Available: http://www.jetsmt.com/us4s/

(Received May 9, 2008)
(Accepted August 26, 2008)

Hoang Anh Tuan is a doctoral student at the Graduate School of Science and Engineering, Ritsumeikan University in Japan. His research interests include reconfigurable computing, security, and hardware/software codesign. He received an MS in computer science from Ritsumeikan University in 2004. He is a member of IPSJ and IEEE.

Katsuhiro Yamazaki is a professor at the Department of VLSI System Design, College of Science and Engineering, Ritsumeikan University in Japan. His research interests include parallel computing, and hardware/software codesign. He received a Ph.D. in computer science from Kyoto University in 1986. He is a member of IEICE, IPSJ, ACM and IEEE.

Shigeru Oyanagi is a professor at the Department of Computer Science, College of Information Science and Engineering, Ritsumeikan University in Japan. His interests include parallel processing, computer architecture, database, and data mining. He graduated Kyoto University in 1972, and received a doctor of engineering from Kyoto University in 1979. He is a member of IEICE, IPSJ, ACM and IEEE.