An Efficient Software Implementation of Fugue

Çağdaş Çalık

Institute of Applied Mathematics, Middle East Technical University, Ankara, Turkey
ccalik@metu.edu.tr

Abstract. We present an efficient software implementation of hash function Fugue using SIMD instructions. By making use of the substitution layer of the recently proposed SIMD implementation of AES by Hamburg, we achieve a fast and constant-time implementation. Combined with architectural optimizations, we observe performance improvements of 42% over optimized C implementation and 9% over existing SIMD implementation on Intel Core 2 Duo processor, the reference platform specified by NIST.
Keywords: Fugue, SHA-3, SIMD, fast implementations.

1 Introduction

Fugue [1] is one of the 14 second round candidates of NIST SHA-3 hash function competition [2]. In addition to the security requirements such as collision resistance, preimage resistance and second preimage resistance, efficiency of the algorithm in hardware and software is also an important measure. A desired property for SHA-3 candidates is that they perform better than SHA-2 in a variety of platforms.

Recently, Hamburg presented a technique to speed-up AES [3] by making use of SIMD instructions. This implementation is not only faster than most of the previous AES implementations in software, but also is resistant to cache-timing attacks due to its being free of memory lookups. The fact that Fugue operations act on 32-bit words and it has the same substitution operation as AES makes it a good candidate for SIMD implementation. In this work, we adapt the technique in [3] and utilize the SIMD architecture to achieve a fast implementation of Fugue which is at the same time a constant time implementation. Even though we are not aware of any cache-timing attacks on hash functions (especially on HMAC construction), our implementation will be resistant to these type of attacks if it happens to be a threat in the future. We compare our work with the optimized C version and two SIMD implementations supplied by authors of Fugue on the reference platform specified by NIST and also on an Intel Core i7-920 processor, which gives better results for AES due to its ability to execute three shuffling operations in parallel.

This paper is organized as follows; in Section 2 we give a brief description of Fugue. Details of the implementation are described in Section 3. In Section 4, optimizations over the baseline implementation are explained. Performance results are presented in Section 5. We discuss future improvements and conclude in Section 6.

2 Description of Fugue

In this section we give a brief description of Fugue. Detailed information can be found in [1]. Fugue supports four output sizes, namely 224, 256, 384 and 512 bits. After appropriate padding of the message, input is processed in blocks of four bytes. After the processing of input is completed, finalization is performed. Fugue has a state consisting of s words (columns), each word being 32 bits. s is 30 for Fugue-224 and Fugue-256, and 36 for Fugue-384 and Fugue-512. We will denote i^{th} word of the state by $\mathrm{S_i}$. Processing of an input block consists of four transformations; **TIX, ROR3, CMIX** and **SMIX**. **TIX** is performed once for each message block, **ROR3, CMIX, SMIX** combination, which is called a **SUBROUND**, is performed twice for Fugue-224 and 256, 3 times for Fugue-384 and 4 times for Fugue-512. Here we give description of each transformation for Fugue-256.

TIX.

Input message word I is merged into the state. Following operations are performed:

$$\begin{aligned} \mathrm{S_{10}} &+= \mathrm{S_0} \\ \mathrm{S_0} &= I \\ \mathrm{S_8} &+= \mathrm{S_0} \\ \mathrm{S_1} &+= \mathrm{S_{24}} \end{aligned}$$

ROR3.

State is rotated right by 3 columns. This corresponds to renaming $\mathrm{S_i}$ as $\mathrm{S_{i+3}}$.

CMIX.

Columns 4, 5 and 6 are added to the columns 0, 1, 2 and $s/2$, $s/2+1$, $s/2+2$.

$$\begin{aligned} \mathrm{S_0} &+= \mathrm{S_4} \\ \mathrm{S_1} &+= \mathrm{S_5} \\ \mathrm{S_2} &+= \mathrm{S_6} \\ \mathrm{S_{s/2}} &+= \mathrm{S_4} \\ \mathrm{S_{s/2+1}} &+= \mathrm{S_5} \\ \mathrm{S_{s/2+2}} &+= \mathrm{S_6} \end{aligned}$$

SMIX.

This is the only nonlinear part of the algorithm. It acts on the first four words of the state and consists of two steps: **Substitution** and **Super-Mix**. Substitution is performed by replacing each byte in these columns using the AES s-box. Super-Mix is the linear diffusion step where the first four words of the state are considered as a 16 byte column vector $X = (x_0, x_1, \ldots, x_{15})$, and is multiplied in $GF(2^8)$ with the following 16×16 matrix N to get $Y = (y_0, y_1, \ldots, y_{15})$ as output.

$$\begin{bmatrix}
1\,4\,7\,1 & 1\,0\,0\,0 & 1\,0\,0\,0 & 1\,0\,0\,0 \\
0\,1\,0\,0 & 1\,1\,4\,7 & 0\,1\,0\,0 & 0\,1\,0\,0 \\
0\,0\,1\,0 & 0\,0\,1\,0 & 7\,1\,1\,4 & 0\,0\,1\,0 \\
0\,0\,0\,1 & 0\,0\,0\,1 & 0\,0\,0\,1 & 4\,7\,1\,1 \\
\\
0\,0\,0\,0 & 0\,4\,7\,1 & 1\,0\,0\,0 & 1\,0\,0\,0 \\
0\,1\,0\,0 & 0\,0\,0\,0 & 1\,0\,4\,7 & 0\,1\,0\,0 \\
0\,0\,1\,0 & 0\,0\,1\,0 & 0\,0\,0\,0 & 7\,1\,0\,4 \\
4\,7\,1\,0 & 0\,0\,0\,1 & 0\,0\,0\,1 & 0\,0\,0\,0 \\
\\
0\,0\,0\,0 & 7\,0\,0\,0 & 6\,4\,7\,1 & 7\,0\,0\,0 \\
0\,7\,0\,0 & 0\,0\,0\,0 & 0\,7\,0\,0 & 1\,6\,4\,7 \\
7\,1\,6\,4 & 0\,0\,7\,0 & 0\,0\,0\,0 & 0\,0\,7\,0 \\
0\,0\,0\,7 & 4\,7\,1\,6 & 0\,0\,0\,7 & 0\,0\,0\,0 \\
\\
0\,0\,0\,0 & 4\,0\,0\,0 & 4\,0\,0\,0 & 5\,4\,7\,1 \\
1\,5\,4\,7 & 0\,0\,0\,0 & 0\,4\,0\,0 & 0\,4\,0\,0 \\
0\,0\,4\,0 & 7\,1\,5\,4 & 0\,0\,0\,0 & 0\,0\,4\,0 \\
0\,0\,0\,4 & 0\,0\,0\,4 & 4\,7\,1\,5 & 0\,0\,0\,0
\end{bmatrix} \cdot \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ \\ x_8 \\ x_9 \\ x_{10} \\ x_{11} \\ \\ x_{12} \\ x_{13} \\ x_{14} \\ x_{15} \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \\ y_4 \\ y_5 \\ y_6 \\ y_7 \\ \\ y_8 \\ y_9 \\ y_{10} \\ y_{11} \\ \\ y_{12} \\ y_{13} \\ y_{14} \\ y_{15} \end{bmatrix}$$

3 Baseline Implementation

We now present the baseline implementation without any optimizations. Afterwards, we will explain how each part can be optimized to get a more efficient result.

3.1 Representation of the state

Representation of the state of the algorithm plays an important role in an efficient implementation. Considering the word oriented structure of Fugue and **ROR3** operation, we chose to store the contents of the state in 10 xmm registers, each containing 3 consecutive columns. The most significant word of each register is kept zero. This representation has various advantages. First of all, **ROR3** operation is completely avoided, we just need to rename register r_i as r_{i+1}. **CMIX** transformation can be performed with two addition instructions once

S_4, S_5 and S_6 is gathered in a register. Also, the most significant word of each register being zero enables us to extract any word(s) from a register to another with a single `pshufd` instruction. Whole state fits in the registers, so we do not need to move data to and from memory. Remaining 6 xmm registers are enough to be used as temporary registers. Indeed, 4 temporary registers are required at most. Fugue-384 and -512 state can be represented in the same manner. This time 12 xmm registers are needed to hold 36 columns and the remaining 4 registers are used as temporary registers. A constraint of this representation is that it only runs on 64-bit mode, because in 32-bit mode the number of available xmm registers is 8.

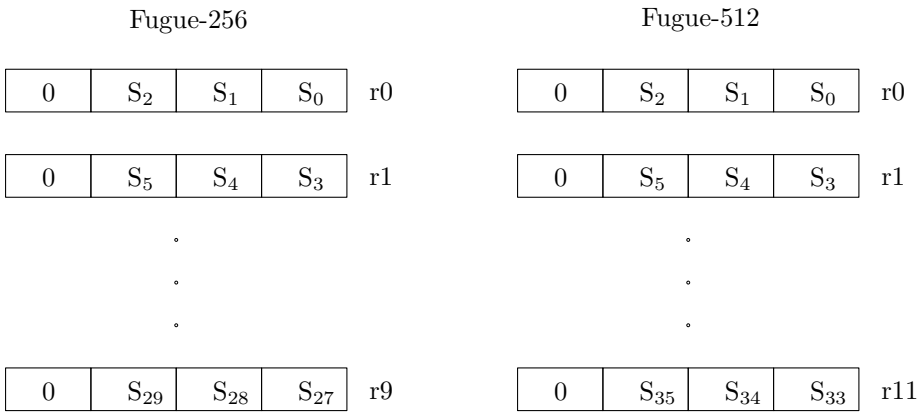

Fig. 1. Representation of Fugue state in xmm registers

3.2 Supplementary Functions

Before explaining how each operation is performed by SIMD instructions, we first describe two auxiliary code fragments that help us organize data in the registers.

PACK This function collects 3 words from one register and combines it with the least significant word from the second register. This operation is required before the **SMIX** transformation in order to gather the first 4 columns of the state into one register. **PACK** function can be performed with a single instruction:

```
insertps r0, r1, 30h
```

UNPACK This function does the inverse of **PACK** function. It copies the most significant word of the first register to the least significant word of the second register and clears the most significant word in the first register.

Listing 1 UNPACK

```
insertps r1, r0, 0c0h
pand     r0, xmmword ptr maskd3n
```

3.3 TIX

This function does the previously described **TIX** transformation.

Listing 2 TIX256

```
pshufd  t1, r0, 0f3h        ; t1 = 0 0 S0 0
xorps   r3, t1              ; S10 += S0
movss   t1, dword ptr [msg] ; t1 = 0 0 0 I
movss   r0, t1              ; S0 = I
pslldq  t1, 8               ; t1 = 0 S0 0 0
xorps   r2, t1              ; S8 += S0
pshufd  t1, r8, 0f3h        ; t1 = 0 0 S24 0
xorps   r0, t1              ; S1 += S24
```

3.4 CMIX

To implement **CMIX**, we first collect S_4, S_5 and S_6 in a temporary register. Then, we add these columns to S_0, S_1, S_2 and $S_{s/2}, S_{s/2+1}, S_{S/2+2}$ in one operation, where s is the number of state columns. In Listing 3, input register $r1$ contains S_3, S_4, S_5 and another input register $r2$ contains S_6, S_7, S_8. After the second instruction, temporary register t contains S_4, S_5, S_6. This register is added to the first and sixth register of the state by adding three columns at once.

Listing 3 CMIX

```
movaps t, r1       ; t = 0 S5 S4 S3
shufps t, r2, 0c9h ; t = 0 S6 S5 S4
xorps  a, t        ; add to columns 0,1,2
xorps  b, t        ; add to columns s/2, s/2+1, S/2+2
```

3.5 ROR3

Our representation of the state completely eliminates this operation. Since each register contains three consecutive words of the state, we simply apply our transformations to the registers with the subsequent index after a **ROR3** operation in the message processing.

3.6 SMIX

SMIX transformation consists of an AES substitution followed by a linear transformation and is the most time consuming part of Fugue. Our implementation of **Substitution** consists of the code taken from implementation of [3], with a few modifications. In the original implementation, the last step of the computation is a lookup operation which gives the output of substitution. In our implementation, we will need 4 and 7 multiples of s-box output in $GF(2^8)$ as well as the original s-box output. Therefore, we will make two more lookups in the end compared to the AES substitution. We will make use of these values while computing the **Super-Mix** operation. Figure 2 shows the steps in the substitution operation, where output of the s-box is: $S(x) = ax^{-1} + b$, a being a 8×8 binary matrix and b is $0x63$ as specified in AES. From the figure, it can be seen that any multiple of the s-box output can be computed by using the intermediate value y produced during the computation.

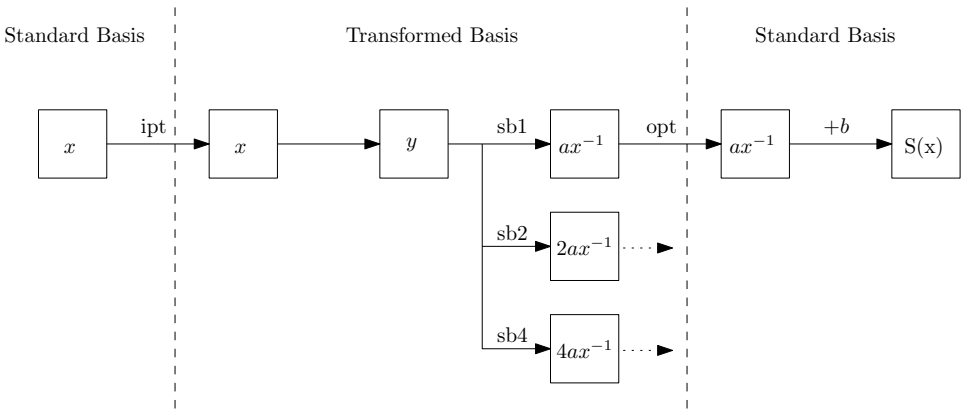

Fig. 2. Substitution

Super-Mix transformation multiplies the 16-byte vector formed by the first four columns of the state with the matrix N specified in the previous section. Here, we give a straightforward implementation of this function without any optimizations. After substitution, we obtain $1x, 4x$ and $7x$, where ix is the 16-byte vector, each element containing the output of substitution multiplied by i in $GF(2^8)$. From these values we can also compute $2x, 5x$ and $6x$, the constants appearing in the matrix N. Then, **Super-Mix** can be performed by xor'ing the permutations of these outputs. Table 1 shows which permutations need to be performed for each multiple of the substitution output. In the table, all numbers indicate indices whereas '*' symbol means a byte value of zero, which can be obtained by setting the most significant bit of the source register for `pshufb` instruction. Sum of all entries in each column gives the result of **Super-Mix**

operation for that output index. For examle, the first output byte can be obtained as a sum of $1.s_0 + 1.s_3 + 1.s_4 + 1.s_8 + 1.s_{12} + 4.s_1 + 7.s_2$, which is equal to the multiplication of the first row of the matrix N with the output of the substitution. The computation of **Super-Mix** in this way requires 5 permutations of $1x$, 3 permutations of $4x$, 1 permutation of $5x$ and $6x$, and 3 permutations of $7x$, yielding a total number of 13 permutations. In the next section, we will describe how the number of permutations can be decreased by taking advantage of high number of zero values used in these permutations.

Table 1. Super-Mix using 13 permutations

1x	0	1	2	3	7	1	2	2	11	12	1	6	15	0	5	10
	3	4	6	7	8	8	6	7	*	*	*	*	*	*	*	*
	4	5	9	11	12	13	13	11	*	*	*	*	*	*	*	*
	8	9	10	14	*	*	*	*	*	*	*	*	*	*	*	*
	12	13	14	15	*	*	*	*	*	*	*	*	*	*	*	*
4x	1	6	11	12	5	10	15	0	9	14	3	4	4	2	2	3
	*	*	*	*	*	*	*	*	*	*	*	*	8	9	7	7
	*	*	*	*	*	*	*	*	*	*	*	*	13	13	14	8
5x	*	*	*	*	*	*	*	*	*	*	*	*	12	1	6	11
6x	*	*	*	*	*	*	*	*	8	13	2	7	*	*	*	*
7x	2	7	8	13	6	11	12	1	4	1	0	3	14	3	4	9
	*	*	*	*	*	*	*	*	10	9	6	5	*	*	*	*
	*	*	*	*	*	*	*	*	12	15	14	11	*	*	*	*

3.7 Putting it all together

Having defined all necessary functions in order to implement Fugue, we can now combine them to form a **SUBROUND** as shown in Listing 4.

Listing 4 SUBROUND

```
CMIX r1, r2, r0, r5, _t0, _t1
PACK r0, r1, _t0
SUBSTITUTE r0, _t1, _t2, _t3, _t0
SUPERMIX _t2, _t3, _t0, _t1, r0
UNPACK r0, r1, _t3
```

3.8 Other Output Sizes

Once we have the core operations implemented for Fugue-256, it is quite straightforward to modify them to get other versions working. The differences are in **TIX** transformation and the number of **SUBROUND** invocations.

Fugue-224. Message processing of Fugue-224 is essentially the same as Fugue-256. Therefore, it has the same performance with Fugue-256.

Fugue-384. Fugue-384 has one more column addition operation in **TIX** transformation and 3 **SUBROUND**s.

Fugue-512. Fugue-512 has two more column addition operations in **TIX** transformation and 4 **SUBROUND**s.

4 Optimizations

We improved our implementation by applying the following optimizations. First three optimizations are mathematical optimizations, reducing the number of operations in order to perform a task. Fourth one is a software optimization and the last one is architectural optimization. Since processors have different architectures, this optimization has to be done for each type of processor separately. Another issue regarding Fugue is, because the number of input bytes it processes per block is small (4 bytes), the effect of each improvement will be high.

4.1 Working in the transformed basis

Substitution function operates in a transformed basis, so that we first need to transform the input value and at the end of the function a transformation to go back to the standard basis is performed. Each Fugue version has at least 2 **SUBROUND**s, meaning that we have to make at least 2 substitutions, and for each substitution switching between standard and transformed basis has a cost. We can eliminate this by keeping the whole state in the transformed basis. This can be done by first transforming the whole state to the transformed basis. The input message block is also transformed before it is used. Only at the and of message processing, i.e., all input is processed, we can transform the whole state to the standard basis. This technique can be optimized further by using the transformed IV values instead of original IV's to avoid the initial state transformation. The final transformation of the state to the standard basis can also be eliminated, however we left this as a future work, because for long messages the effect of these transformations become negligible.

4.2 Optimizing Super-Mix

The baseline implementation of **Super-Mix** required 13 permutations. It is obvious that the less the number of permutations, the higher the performance. In order to explain the optimization in this transformation better, we first present a permutation list better than the original one, and then we will present the

best one we could find. Table 2 is an improved version of Table 1, requiring 11 permutations. In this table, we introduce $2x$ and eliminate $5x$ and $6x$. We also eliminate the $5th$ permutation of $1x$ in Table 1 by adding indices $12, 13, 14, 15$ to $2x, 4x$ and $7x$. Bold values in Table 2 indicate the indices used to eliminate this permutation and also the permutations of $5x$ and $6x$. By adding a permutation of $2x$ and eliminating 3 others, we get a total number of 11 permutations to compute **Super-Mix**. It can be easily checked that this improved permutation list gives exactly the same result as the previous one. For example, the first output byte according to Table 2 this time becomes;

$$\begin{aligned} y_0 &= 1.s_0 + 1.s_3 + 1.s_4 + 1.s_8 + \mathbf{2.s_{12}} + 4.s_1 + \mathbf{4.s_{12}} + 7.s_2 + \mathbf{7.s_{12}} \\ &= 1.s_0 + 1.s_3 + 1.s_4 + 1.s_8 + \mathbf{1.s_{12}} + 4.s_1 + 7.s_2 \end{aligned}$$

which is equal to the desired output.

Table 2. Super-Mix using 11 permutations

1x	0	1	2	3	7	1	2	2	11	12	1	6	15	0	5	10
	3	4	6	7	8	8	6	7	*	*	*	*	*	*	*	*
	4	5	9	11	12	13	13	11	*	*	*	*	*	*	*	*
	8	9	10	14	*	*	*	*	*	*	*	*	*	*	*	*
2x	**12**	**13**	**14**	**15**	*	*	*	*	**8**	**13**	**2**	**7**	**12**	**1**	**6**	**11**
4x	1	6	11	12	5	10	15	0	9	14	3	4	4	2	2	3
	12	**13**	**14**	**15**	*	*	*	*	**8**	**13**	**2**	**7**	8	9	7	7
	*	*	*	*	*	*	*	*	*	*	*	*	13	13	14	8
7x	2	7	8	13	6	11	12	1	4	1	0	3	14	3	4	9
	12	**13**	**14**	**15**	*	*	*	*	10	9	6	5	**12**	**1**	**6**	**11**
	*	*	*	*	*	*	*	*	12	15	14	11	*	*	*	*

Now, we will show how the permutations in Table 2 can be squeezed further, and at the same time by taking advantage of what we call *continuous permutations*, meaning that the result of a permutation can be used as an input to the next permutation without requiring the original input. As an example, if we examine the first two permutations used in Table 2, we can see that the second permutation requires s_4, however the first permutation does not use s_4, so we cannot continue to permute from the output of the first permutation to obtain the second permutation. If we can perform permutations in a continuous way, we save a register to register move operation and get a more efficient result. We also want to note that the indices in a column (for the same multiples) can be swapped due to the commutativity of addition over $GF(2^8)$.

With these observations in mind, we now introduce a better permutation list in Table 3, both having 1 less permutation count, and at the same time enabling us to perform some continuous permutations. There are a few points to clarify about Table 3. The rows with a '+' symbol indicate continuous permutations, i.e., these permutations can be done using the output of the previous permutation. In the permutation list of $1x$, second and third permutations are added first, and the fourth permutations is performed on this sum. Therefore, indices in the last permutation of $1x$ do not represent s_i, but indices of the previously mentioned sum. For instance, the first entry 12 means that the $12th$ entry of the sum, which is $s_8 + s_{12}$ will be copied to this location. Actually, the first four values appearing in the fourth permutation are copied into this location from the temporary place they were added, and the temporary values are deleted by copying them onto the same positions (last four elements) in the fourth permutation. Finally, the last permutation of $4x$ is performed on the sum of $2x$ and the second permutation of $4x$, in order to calculate the required multiples of $6x$ shown in Table 1 and move it to the desired place (one word to the right). The elements in question are shown in boldface for this operation.

Table 3. Super-Mix using 10 permutations

1x	0	1	2	7	8	1	2	2	11	12	1	6	15	0	5	10	
	4	5	6	3	7	8	13	11	10	15	14	5	12	9	10	14	
	3	4	9	11	12	13	6	7	*	*	*	*	8	13	14	15	+
	12	13	14	15	*	*	*	*	*	*	*	*	12	13	14	15	+
2x	*	*	*	*	**8**	**13**	**2**	**7**	10	15	14	5	12	1	6	11	
4x	1	6	11	12	5	10	15	0	9	14	3	4	4	2	2	3	
	13	13	14	8	**8**	**13**	**2**	**7**	10	15	14	5	8	9	7	7	
	13	13	14	8	8	13	2	7	**8**	**13**	**2**	**7**	13	13	14	8	+
7x	2	7	8	13	6	11	12	1	4	1	0	3	14	3	4	9	
	*	*	*	*	*	*	*	*	12	9	6	11	12	1	6	11	+

4.3 Combining SUBROUNDs

Before computing **SMIX**, first four columns of the state are collected in a register. This is accomplished by copying S_3 from the least significant word of the second register of the state to the most significant word of the first register of the state with **PACK** function. At the end of this operation, first register contains S_0, S_1, S_2 and S_3. After the end **SMIX**, new value of S_3 is moved back to the second register with **UNPACK** function. If more than one **SUBROUND** operation is going to be performed one after the other, which is the case for

all Fugue versions, we can embed **CMIX** and **PACK** operations of the second **SUBROUND** into the first one and perform the same transformations with less number of instructions.

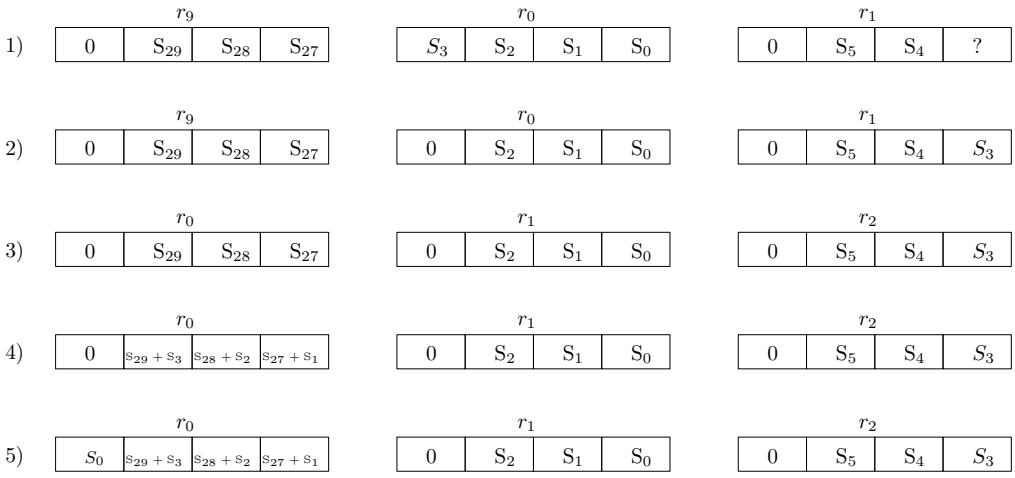

Fig. 3. Operations between two SMIX transformations

Figure 3 shows the operations performed between two consecutive **SMIX** transformation in 5 steps. The first step shows the contents of the three registers after an **SMIX** transformation, with register r_0 containing the output. Second step shows the contents of same registers after an **UNPACK** operation. Here, S_3 in r_0 is moved to the least significant word of r_1. In step 3, **ROR3** operation is performed by renaming r_i as r_{i+1}. Now, first three columns of the state become S_{27}, S_{28} and S_{29}. In step 4, we see the first 3 operations of **CMIX** transformation performed, i.e., S_1, S_2, S_3 are added to the first three words of the state, which are in r_0. Finally in step 5, we see the registers after a **PACK** operation which copies S_0 from r_1 as the fourth word of the state and make r_0 ready for the next **SMIX** transformation. Note that the words added to r_0 for **CMIX** transformation are S_1, S_2 and S_3, and these are already available in r_0 in the first step. If we rotate r_0 in step 1 to the right by one word and add it to r_9, we obtain r_0 of step 5, performing **CMIX** and **PACK** simultaneously. We can perform the other half of **CMIX** by setting the high order word of rotated r_0 in step 1 to zero and adding this register to the sixth register of the state. **UNPACK** operation of second step should also be performed. Consequently, we do less operations this way compared to executing two **SUBROUND**s one after another. The optimized source code for **SUBROUND** is given in Listing 5.

Listing 5 ROUND256

```
CMIX r1, r2, r0, r5, _t0, _t1
PACK r0, r1, _t0
SUBSTITUTE r0, _t1, _t2, _t3, _t0
SUPERMIX _t2, _t3, _t0, _t1, r0

pshufd _t0, r0, 39h           ; _t0 = s0 s3 s2 s1
xorps s0, _t0
pand _t0, xmmword ptr maskd3n ; _t0 =  0 s3 s2 s1
xorps s5, _t0

UNPACK r0, r1, _t3

;CMIX s1, s2, s0, s5, _t0, _t1
;PACK s0, s1, _t0
SUBSTITUTE s0, _t1, _t2, _t3, _t0
SUPERMIX _t2, _t3, _t0, _t1, s0
UNPACK s0, s1, _t3
```

4.4 Inline Functions

Instead of defining Fugue transformations -in particular **Substitution** and **SuperMix**- as functions, we can use inlining to avoid function calling overhead. A disadvantage of this approach is increased code size. On a target platform such as a PC, the code size of inline version of our implementations ranged between 4KB and 5KB, which is tolerable. Therefore, we used inline functions in benchmarking, and verified that they are slightly faster than uninlined versions.

4.5 Architectural optimizations

We focused our effort in optimizing the implementation on two target platforms, Intel Core 2 Duo processor and Intel Core i7-920 processor. The method we used consisted of trying logically equivalent instructions and rearranging instructions that do not depend on each other. Although we made use of optimization guidelines [4], [5], most of our effort involved trial and error.

The optimization in Core 2 Duo processor has been relatively easy. This processor enables the mixed usage of floating point and integer SIMD instructions without a performance penalty. The most notable thing we took care of was using `xorps` instead of `pxor` because the former's machine code is shorter. However, using `pxor` in **Substitution** at two points gave the best results.

The main advantage of Core i7-920 processor over Core 2 Duo concerning our implementation is its ability to execute three shuffling operations in parallel. However, we could not obtain the performance gain we exptected in this processor. This could be due to the number of shufflings per byte being not so high. A major factor affected our style of implementation on Core i7-920 is its constraint on running integer and floating point SIMD instructions together. Our

code consisted of floating point instructions, integer instructions and instructions that can be coded in both ways. Since the most critical instruction `pshufb` is working in the integer domain, we transformed the code to use as much integer instructions as posssible.

5 Benchmarks

There are mainly two options for making an implementation in SIMD architecture. One can use *compiler intrinsics* to make the code portable across different platforms and also take advantage of compiler's optimization features. The other option is writing in assembly language. This gives the programmer more freedom and may result in faster code as was the case in our situation. We implemented Fugue using both intrinsics and assembly language, and compiled it under various compilers. It turned out that hand optimized assembly code gives better results than intrinsics version under all compilers we have tested. However, for the implementations we made comparison, namely Jutla's SSSE3 and SSE4 implementations, different compilers produced different results. We chose the best results observed in these cases.

The benchmarks were calculated by hashing several messages of length 1MB and taking the best observed timing. We would like to note however that the standard deviation of these measurements is quite small. Indeed, the average of the measurements is no more than 1 cycles/byte worse than the best result. On the other hand, for messages shorter than 2^{15} bytes, the performance of our implementation gets worse due to the overhead caused by state transformations made before and after message processing.

Table 4 shows benchmarks on Intel Core 2 Duo processor. Jutla's implementation included only Fugue-256 at the time of this writing, so we were not able to make a comparison for other output lengths. In the table, opt-64 refers to the optimized C implementation written by the authors of Fugue.

Table 4. Benchmarks for Intel Core2 Duo E8400 in cycles/byte

	Implementation	Fugue-224,-256	Fugue-384	Fugue-512
this paper	assembly	15.85	24.87	32.49
jutla-ssse3	intrinsics	18.41	N/A	N/A
jutla-sse4	intrinsics	17.45	N/A	N/A
opt-64	C	27.61	41.53	55.28

Table 5 shows the performance results for Intel Core i7-920 processor. Except Jutla's implementations, the results in this processor are quite close to the ones measured on Core 2 Duo processor. The difference in Jutla's implementations are due to the compiler we had to use on this processor. We didn't have access to an

Intel C/C++ compiler on this machine, therefore the benchmarks were measured with gcc compiler. We expect that Intel C/C++ compiler would produce better results on this processor, especially for SSE4 version.

Table 5. Benchmarks for Intel Core i7-920 in cycles/byte

	Implementation	Fugue-224,-256	Fugue-384	Fugue-512
this paper	assembly	16.57	23.95	31.10
jutla-ssse3	intrinsics	20.97	N/A	N/A
jutla-sse4	intrinsics	38.50	N/A	N/A
opt-64	C	27.66	41.26	55.16

For all output lengths, our implementation is over 40% faster than the optimized C version. We get 9% and 21% better results on Core 2 Duo and Core i7-920 processors respectively, compared to the fastest SIMD implementation of Jutla. The code we used in these benchmarking requires SSE4 support because of the `insertps` instruction. This instruction can be replaced by other instructions to make the code SSSE3 compatible with a slightly worse performance. However, we do not take this version into account since the processors this code is tuned for support SSE4 instruction set.

6 Conclusion and Future Work

In this work, we demonstrated an efficient and constant-time implementation of hash function Fugue using SIMD instructions. We were inspired by a recent technique used to speed-up AES and adapted the substitution part of it to Fugue. We optimized the code for target platforms and achieved better results than the existing ones. Our implementation attains its speed for messages longer than 2^{15} bytes because of the overhead caused by state transformations. We plan to eliminate this drawback as a future work. Implementing Fugue using AES-NI instruction set and see how much it benefits from this architecture is another goal we would like to accomplish.

Acknowledgments

We thank Mike Hamburg for his valuable comments and suggestions. We are also grateful to Onur Özen and Dag Arne Osvik for their help in getting access to an Intel Core i7-920 machine from EPFL for benchmarking.

References

1. Shai Halevi, William E. Hall, Charanjit S. Jutla. The Hash Function Fugue. *Submission to NIST (updated)*, 2009. Available at:

http://domino.research.ibm.com/comm/research_projects.nsf/pages/fugue.index.html/FILE/fugue_09.pdf.
2. National Institute of Standards and Technology. Announcing Request for Candidate Algorithm Nominations for a New Cryptographic Hash Algorithm (SHA-3) Family. *Federal Register*, 27(212):62212–62220, 2007. Available at: http://csrc.nist.gov/groups/ST/hash/documents/FR_Notice_Nov07.pdf.
3. Mike Hamburg. Accelerating aes with vector permute instructions. In *CHES*, pages 18–32, 2009.
4. Agner Fog. The microarchitecture of Intel, AMD and VIA CPUs, 2010-02-16.
5. Intel®64 and IA-32 Architectures Optimization Reference Manual. Order Number: 248966-020, November 2009.