Cipher-Block-Chaining Triple-DES Optimization on a Multi-thread Processor

Chingyi Lin[1]

***Abstract*— Triple DES is one of the most famous decryption algorithm. With OpenSSL, a open source decryption library, it has a great performance in some processor. However, in our target processor, Intel i7-5600U, we can design a kernel which fits the parameter of the target processor. In the end, we can obtain 10% improvement in overall performance.**

I. ALGORITHM DESCRIPTION

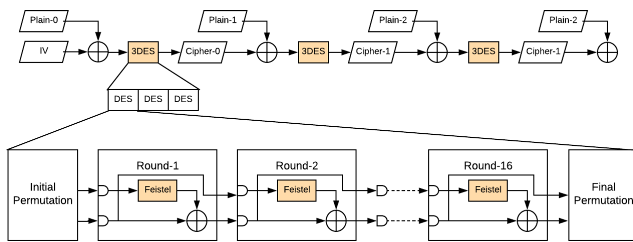

Fig. 1. CBC block diagram

Cipher-block-chaining Triple Data Encryption Standard (CBC-3DES) is a block cipher algorithm. Illustrated in figure 1, a plaintext is divided into chunks and encrypted by yellow block 3DES in sequence. Each input of 3DES is the XOR of last block output and a chunk of plaintext.

Digging into block *3DES* in figure 1, a typical 3DES core consists of three sequential DES blocks. Each DES block contains 2 permutations and 16 Feistel functional blocks connected sequentially. That is, the input of each block is the output of the previous block. The more detail of these functional blocks will be discussed in the following sections.

II. BASELINE IMPLEMENTATION

A 3DES block is mainly composed of two operations, permutation and Feistel functional blocks. In this section, we will explain their behavior and implementation in our baseline library, OpenSSL.

A. Permutation

Permutation reorders all the bits in the input plaintext. OpenSSL implements the permutations in the terms of permutation operation, called PERM_OP. A PERM_OP can swap some specific bits between two number. In 3DES, Initial permutation (IP) and final permutation (FP) are implemented by five PERM_OPs in OpenSSL.

[1]Chingyi Lin is PhD student of Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213 chingyil@andrew.cmu.edu

B. Feistal function

Original Feistal function has four steps: expansion, key mixing, substitution and permutation. It takes a 32-bit half-plaintext and generates a 32-bit half-ciphertext. OpenSSL makes some modification along these four steps.

1) Bit reordering in expansion and key mixing: Figure 2 shows the data flow in expansion and key mixing, in which expansion repeats some bits and increases data size into 48 bits (For example, bit 23 are used by both [28:23] and [24:19]), and key mixing XORs this 48-bit data with a 48-bit pre-defined sub-key.

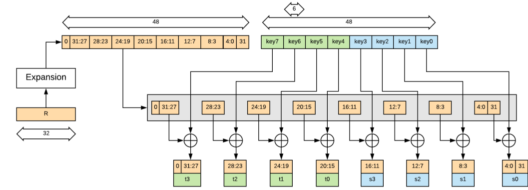

Fig. 2. Original expansion and key binding

Due to the 32-bit granularity of x86 processors, OpenSSL modifies the bit order of those operands and save the number of operation in the following calculation (substitution and permutation). Illustrated in figure 3, you will see that lower part (blue) and higher part (green) of a 48-bit sub-key is interleaved in key-binding. Furthermore, the 32-bit yellow input are duplicated and distributed into the two 32-bit word, and its order is aligned with the bits of this pre-defined order.

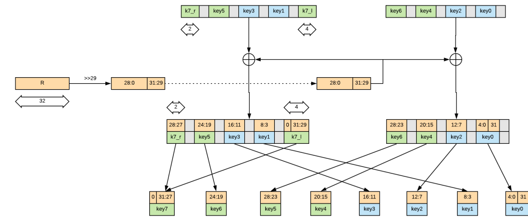

Fig. 3. OpenSSL expansion and key binding

2) Merge substitution and permutation into a single look-up table: Substitution maps eight 6-bit chunks to 4-bit, and the following permutation simply change the bit order of this cascaded 4-bit results in a 32-bit scope.

With the associativity of look-up table, these two operations can be merged simply. Substitution inherently is a look-up table from 6-bit index to a 4-bit value. The permutation can be seen as another look-up table, which

maps a 4-bit index to a 32-bit value with only four 1s inside, specifically, each value in permutation look-up table is 4-weighted 32-bit. That is, the substitution and permutation can be equivalently combined into a single look-up table, called *SPtrans* in OpenSSL, mapping 6-bit index to a 32-bit, 4-weight word. The final result of Feistel function can be obtained by OR/XORing these words from eight different *SPtrans* table.

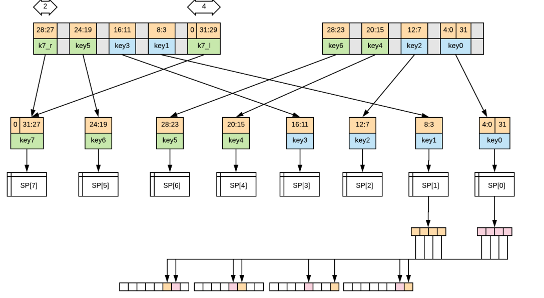

Fig. 4. OpenSSL substitution and permutation

3) Compensating FP with next IP in adjacent DES: A function can be compensated with its inverse intuitively. In DES, initial permutation operation is inverse of final permutation. There are three cascade DES with IP following by last-round FP. So instead of three IP and FP in a 3DES, we just only need to use an IP before three sequentail DES block and a FP in the end of them.

III. Benchmark

Based on the implementation and optimization in the last section, a 3DES consists of 10 PERM_OP, 12 bitwise-shift and 48 D_ENCRYPT. We will detail and evaluate them in this section.

A. Permutation kernel – PERM_OP

PERM_OP is the core of permutations in 3DES. It uses a tricky way to implement permutation under 32-bit granularity. The source code is

The definition of OpenSSL is as following

```
# define PERM_OP(a,b,t,n,m)\
        ((t)=((((a)$\gg$(n))^(b))&(m)),\
        (b)^=(t),\
        (a)^=((t)$\ll$(n)))
```

1) Construction: These PERM_OPs achieve a permutation with shift, XOR and bitwise-AND only. With this macro definition, b can swap half of its bits with a. For example, assuming 32-bit a and b can be expressed in the form of $(a_{31}, a_{30}, ...a_0)$ and $(b_{31}, b_{30}, ...b_0)$ respectively, a statement

```
PERM_OP(r,l,tt,2,0x33333333L)
```

can have a result b = $(a_1, a_0, b_{29}, b_{28}, a_{29}, ..., b_1, b_0)$ and a = $(a_{31}, a_{30}, b_{27}, b_{26}, a_{27}, ..., b_{31}, b_{30})$.

2) Evalutation and theoretical peak: There are no memory effect in this operation. So the measured theoretical peak is same as its execution time, 4.4 cycle.

For calculated theoretical peak, figure 5 shows the data flow in permutation operation kernel. The critical path consists of two shift, two XOR and an AND. From Anger's table [1], the total latency is 5 cycles, which is close to our measurement.

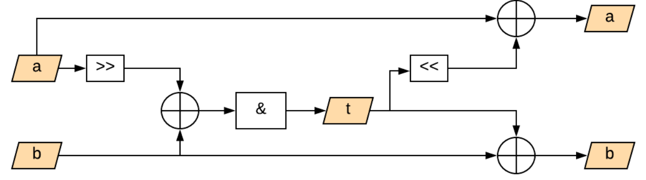

Fig. 5. OpenSSL's permutation kernel

B. Feistel function kernel – D_ENCRYPT

D_ENCRYPT is the core of Feistel function. It does XOR for key binding first, and take the XORed value as the index of the look-up table, *DES_SPtrans*, which is equivalent to substitution and permutation. The source code is like

```
# define D_ENCRYPT(LL,R,S) {
  u=R^s[S  ];
  t=R^s[S+1];
  t=ROTATE(t,4);
  LL^=
      DES_SPtrans[0][(u$\gg$ 2L)&0x3f]^
      DES_SPtrans[2][(u$\gg$10L)&0x3f]^
      DES_SPtrans[4][(u$\gg$18L)&0x3f]^
      DES_SPtrans[6][(u$\gg$26L)&0x3f]^
      DES_SPtrans[1][(t$\gg$ 2L)&0x3f]^
      DES_SPtrans[3][(t$\gg$10L)&0x3f]^
      DES_SPtrans[5][(t$\gg$18L)&0x3f]^
      DES_SPtrans[7][(t$\gg$26L)&0x3f]; }
```

1) Construction: A D_ENCRYPT can be divided into two blocks (ignoring low-overhead ROTATE). The first one is two independent 32-bit XOR followed by two memory load, which read two pre-calculated keys on a consecutive memory space. The second one is eight independent shift-then-mask as the index, and these index randomly access eight individual table. The output of this kernel is a 32-bit word, which is the summation of the loaded results from these eight tables.

2) Evaluation and theoretical peak: There are two parts involved with memory in D_ENCRYPT, key loading and SPtrans table. We measure the original timing and remove the load of these two load respectively. The result is as table I. In this table, the theoretical peak is obtained from the measurement without key load and SPtrans. Shortly, 10 cycles.

	With key load	Without key load
With SPtrans	27.3	20.5
Without SPtrans	16.9	10.0

TABLE I
D_ENCRYPT TIMING RESULT

The 10-cycle theoretical peak is reasonable. Figure 6 shows one of the schedule in Broadwell, our target processor. With four logic operation functional unit in this processor, there are total 31 operations in each Feistel function. Regardless of data dependency, the minimum cycle is still 8 cycles.

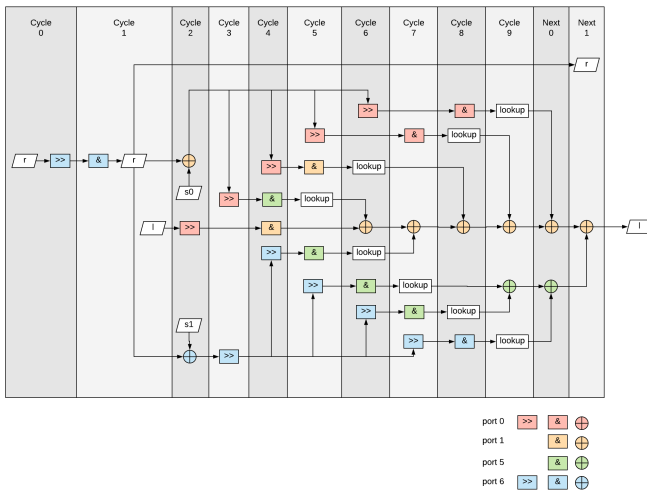

Fig. 6. Theoretical peak of Feistel kernel

From Agner's table[1], each operation is not supported by all the functional unit in Broadwell. This limit is the bottleneck of the theoretical peak. The legend in figure 6 shows the functional unit which can help our Feistel function. The AND and XOR can be handled by four functional unit. However, the shift operation can only be served by port 0 and 5. Considering these constraint and data dependency, the calculated theoretical peak is 10 cycle.

C. Overall performance

Like figure 7, adding all the blocks together, the summation is 1336 cycles, which is close to the measured performance, 1250-cycle. From our block-wise evaluation, a permutation takes 20 cycles and a 16-D_ENCRYPT chain costs 400 cycles. The 20-cycle permutation makes sense because it is close to the sum of five dependent 4.4-latency PERM_OPs.

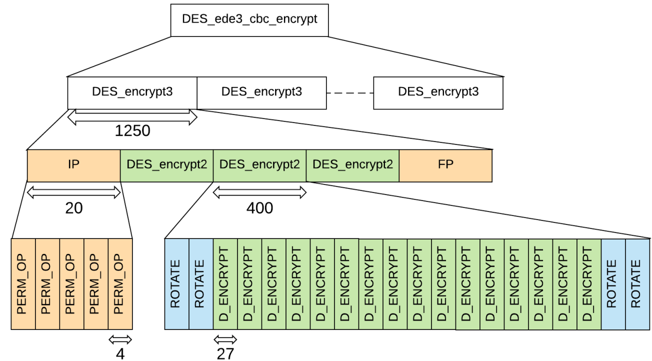

Fig. 7. Overall performance

IV. Optimization

A. Merging two SPtrans table into a bigger table

In each D_ENCRYPT, there are eight random access to eight separate 64-element table. We can reduce the number of memory access with the expense of increased table size. Shown in figure 8, two 6-bit index can be cascaded as a longer 12-bit index. Sacrificing the memory space, we store the pre-computed results and accelerate the execution. So that we can obtain the value with only one memory access. It's a trade-off between execution time and memory space.

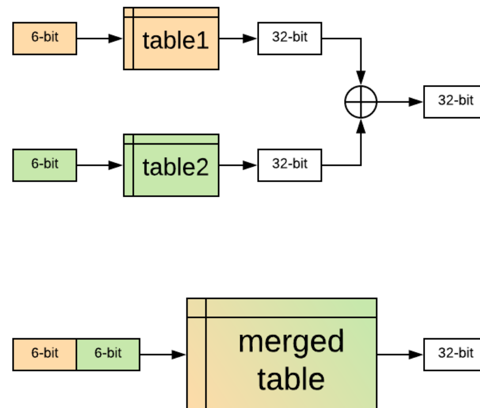

Fig. 8. Table merge

Table contents need to be changed as this modification. The two outputs from each table are originally XORed together. So the merged table is a composite function, in which

$$T(x1, x2) = T1(x1) + T2(x2)$$

, where T1, T2 are two original table and x1, x2 are their inputs.

Table size increases due to the above modification. With 6-bit index, the original tables both have 2^6 elements, but merged table contains 2^12 elements inside. To store all the SPtrans, we need $4(tables) * 2^{12}(element/table) * 4(byte/element) = 64KB$ of storage. This number exceeds the size of L1 cache, suggesting that all of the entry cannot keep in L1 cache.

The measurement is listed on table II, the second and third DES has a lower execution time compared to the OpenSSL implementation. Though the first DES somehow takes longer time than the original, the total time saves about 10% in the end.

	DES-1	DES-2	DES-3
OpenSSL	410	404	402
Optimized	421	332	326

TABLE II
RESULT OF TABLE MERGE

This result shows that the two L1 cache access and the following XOR is slower than a single L2 cache access. From LZMA benchmark, L1 cache access spends 4 cycle, and L2 12 cycles. We can have some basic calculation of the average access time.

Because 3-DES is random accessing these tables, each access is independent to other. The OpenSSL's case is simple, all the tables are stored in L1 cache. The total cost, or more specific, latency, of the 32 bits in figure 8 is $2 * \text{mem}_{L1} + \text{lat}_{XOR} = 2 * 4 + 1 = 9$cycles.

For the merged table, we still have chance to get a data from L1 cache, though we cannot fit all the tables into L1 cache. With 32KB L1 cache in our target processor, the 64KB table can have $\frac{32KB}{64KB} = 50\%$ partition in L1 cache. Hence, the total cost is $\frac{32KB}{64KB} * \text{mem}_{L1} + \frac{64KB-32KB}{64KB} * \text{mem}_{L2} = 0.5 * 4 + 0.5 * 12 = 8$ cycles.

The above calculation implies the theoretical improvement $1 - \frac{8}{9} = 11\%$ of this trick, which matches the result in our evaluation. In table II, the improvement is $1 - \frac{421+332+326}{410+404+402} = 10.5\%$. Though we don't consider the second LOAD unit in our target processor, the effect of this additional unit is just a scalar, which will not greatly affect the ratio between the comparison.

V. FAIL OPTIMIZATION (IMPLEMENTED BUT NOT IMPROVE PERFORMANCE)

A. Create parallel operation from permutation kernel

With multiple functional unit in our target processor, increasing the number of operation might not degrade the performance. The general idea is, we need to make the operation be processed in parallel.

The original kernel has dependency between t to a and b. Furthermore, all of the operation producing a are sequential. That is, the a in next iteration is generated by shift, XOR, AND, shift and XOR in order. These operation are all dependent, implying that we can not take the advantage of multiple functional units in our target processors.

In the figure 9, the optimized block has same functionality with the OpenSSL's. Though it has more functional units, the critical path is only with three blocks, which is less than five blocks from the baseline. The derivation of this block is in the appendix.

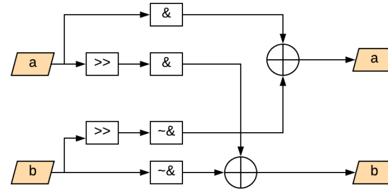

Fig. 9. Merged single permutation kernel

B. Merge two permutation kernel to a bigger one

Besides of optimization on one block, cross-blocks can find the opportunity to reuse the data in a broader scope. Figure 10 is two cascade permutation kernels. There are four ANDs in each kernel. Due to the associativity and commutativity of AND and XOR, we might merge those AND-mask together and save some time.

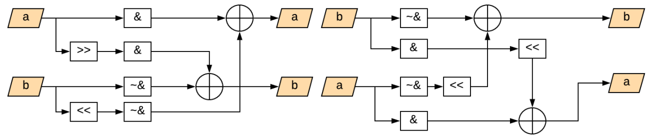

Fig. 10. Two cascade permutation kernels

With some reordering and merge, these two kernels can be combined to a bigger one, like figure 11. The detail will be discussed in appendix. In this merged kernel, two ANDs in different kernel is merged to a single one. Though total number of ANDs is same after this merge, these eight ANDs are parallel in the new kernel and are able to leverage multiple logic functional units in the target processor.

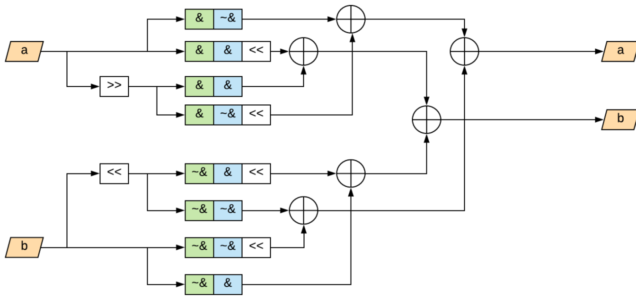

Fig. 11. Merged double permutation kernel

C. Split reduce in D_ENCRYPT

There are eight sequential XOR to reduce the lookup value in D_ENCRYPT. To fully utilize the functional units in our processor, we can separate them into two independent stream and sum them up in the end. Illustrated in figure 12, this trick can shrink the critical path of reduce path.

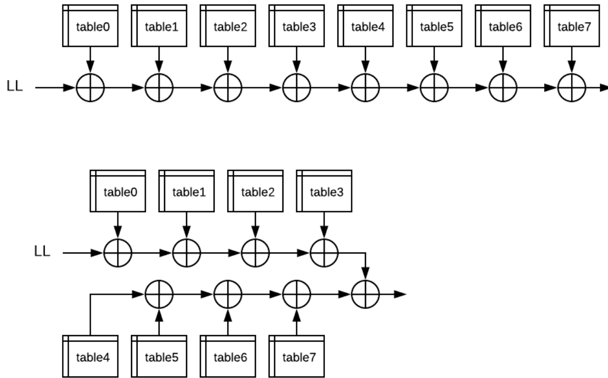

Fig. 12. Split reduce in D_ENCRYPT

VI. Evaluation

	IF	DES-1	DES-2	DES-3	FP	Total
Theoretical peak	20	160	160	160	20	520
OpenSSL	20	410	404	402	20	1256
Optimized	20	421	332	326	20	1119

TABLE III
Result of table merge

In the end, we mitigate the execution time from 1256 to 1119. This 10% improvement is mainly because of the merge table, which successfully sacrifices the storage but gets speed-up.

References

[1] Agner Fog: Intruction tables. https://www.agner.org/optimize/-instruction_tables.pdf

Appendix I – Merged single permutation kernel

Before we start modify this kernel, there are x things to be notified.

1) The second operand of AND, shift are called ANDer and shifter in this appendix. Every ANDer and shifter in each kernel is pre-defined but different. For simplicity, we won't label these operands in the following.
2) Some of the bit-wise can swap without changing the result. For example, the XOR and AND can swap their order.

The OpenSSL's implementation is shown in figure 13. To leverage multiple functional units, the first step of our optimization is to decompose those operation and remove data dependency.

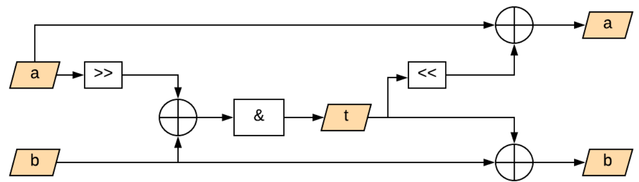

Fig. 13. OpenSSL's permutation kernel

The simplest way of decomposition is duplication. We duplicate the functional unit before t. Furthermore, we swap the order of XOR and AND. After the decomposition, the result can be obtained from the input in parallel, which is shown in 14.

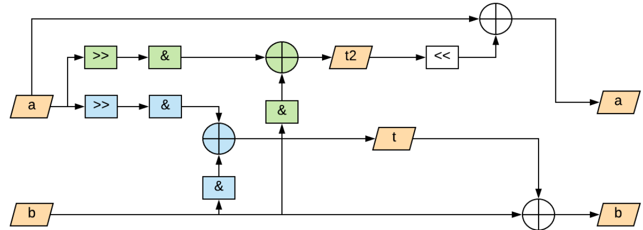

Fig. 14. After decomposition

Here we come the first operation merge

$$(A \otimes B) \ll n = (A \ll n) \otimes (B \ll n)$$

This equation means we can bypass the shift after an XOR. This trick is useful because this bypassed shift can be merged with others. In this case, the bypassed shift can be compensate with another shift.

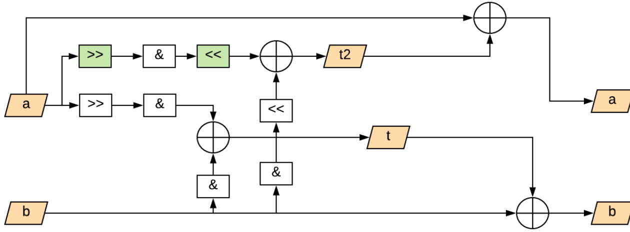

Fig. 15. The chance to compensate shift operator

After bypassing the shift and merge, we shrink the critical path in this kernel. Here, the "≪&" means that the operand of AND is pre-shifted. In permutation kernel, all the ANDer's is pre-defined. So the pre-shifted can be automatically done by compiler or manually, without any cost in runtime.

$$[(A \gg n)\&(m)] \ll n = [(A \gg n) \ll n]\&[m \ll n] = A\&(m \ll n)$$

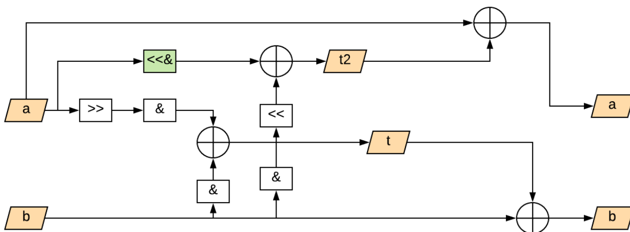

Fig. 16. Compensated shift operator

From the commutativity of XOR, the operand of serial XOR can be swapped. Hence, figure 17 expresses each output as the sum of two single-input functions after the reordering. For example, $b = f(a) + g(b)$, where $f(a) = (a \gg n)\&m, g(b) = b \otimes (b\&m)$. Then the final step is finding a simpler or cheaper operation to replace f and g.

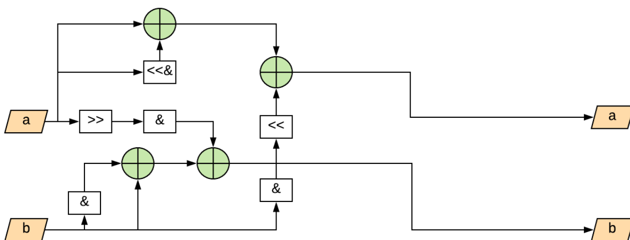

Fig. 17. Operation reordering

The final operation merge is not intuitive, but we can observe that there are similar structure in the diagram, X $\otimes$ (X&Y). Both XOR and AND are bit-independent operation, implying that a single-bit truth table can list all the possible result.

	y=0	y=1
x=0	0	0
x=1	1	0

TABLE IV

TRUTH TABLE OF X $\otimes$ (X&Y)

From the truth table, it turns out that x $\otimes$ (x&y) is equal to x&(y). Though it doesn't save the number of operation, the bitwise-NOT here applied on the key can be pre-calculated and shrink the critical path.

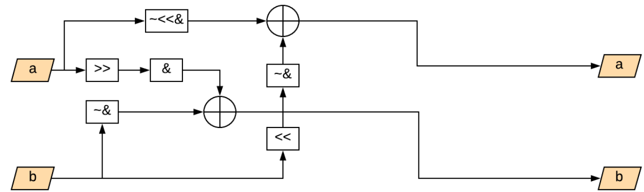

Fig. 18. Operation merge

Figure 18 shows the general merged result. The above tricks can be applied to any case with pre-defined ANDer or shifter.

Fortunately, we have some properties in permutation kernel. We have two input in a kernel, m(32-bit mask) and n(bit length). All of the input follows some rules.

1) m is a 16-weight word. In other words, there are equivalent 0s and 1s in each m.
2) Each m consists of several 2n-bit windows, in which one half is n bits with all 0s, and the other half is all 1s.
3) m≪n = m = m≫n. Because a shift is a swap of all the n-window

From these property, there are some equivalent blocks in this diagram. Simply, "∼≪&" = "≪∼&" = "&" = "≫∼&" = "∼≫&". Then we can derive our final kernel.

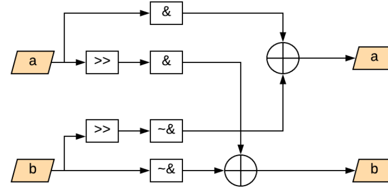

Fig. 19. The optimized permutation kernel

Appendix II – Merged double permutation kernel

Besides operation merge in the single kernel, multiple kernels also have some opportunity to combine operation together.

Simply duplicate our kernel to two, shown in figure 20. The input order will swap after each PERM_OP, so the a of the first kernel is b in the second kernel.

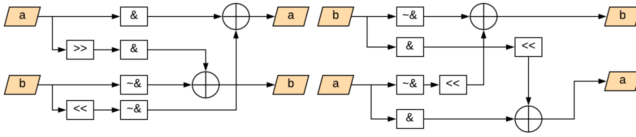

Fig. 20. Two cascade permutation kernels

The first step is always decompose the operations into parallel. Preventing to be too complicated, we decompose the upper part only in this stage.

The basic of decomposition is break the data sharing. Here, the input b in the second kernel is generated by $(a\&m1)\otimes[(b\ll n1)\&(\ m1)]$. The figure 21 duplicates it to two identical functional units chain. After this separation, operation in second kernel can be bypassed independently.

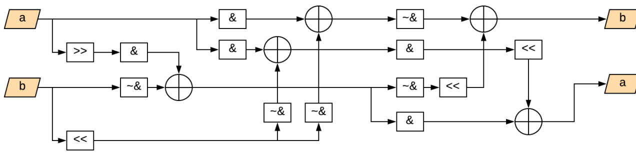

Fig. 21. After decomposition

Bypassing ANDs is the most important trick in this section. The purpose in this section is put as much operation into pre-defined mask as possible. Merging the masks together is a great solution for us.

ANDs in different kernel are painted into two different color. Thanks for the commutativity of AND, with the bypassing, two different ANDs can be merged into one, in which the mask is the AND result of two predefined masks. Figure 22 shows the result after merge.

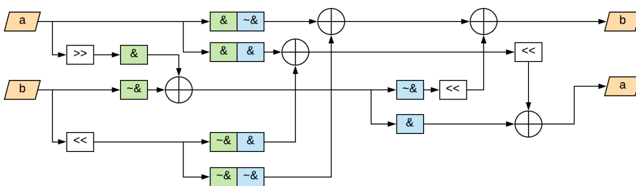

Fig. 22. Partial AND merge

Same tricks can be applied to the lower-part of this bigger kernel. In figure 23, there are total eight parallel ANDs. Although the number of total ANDs is not decreasing, this trick gives it more parallelism in our target processor.

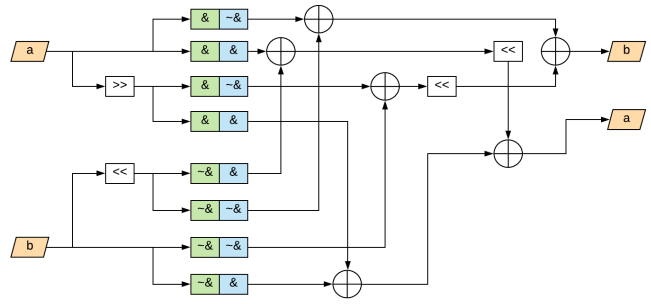

Fig. 23. AND merge in double kernel

Each output is the sum of four 32-bit masked result, except for some annoying shift. The last step is reorder those XOR and bypass the shift in the second kernel. To be tedious, we want the output be the pure summation of a single-input functions.

The figure is the final output of this merged double kernel, each output can be exp

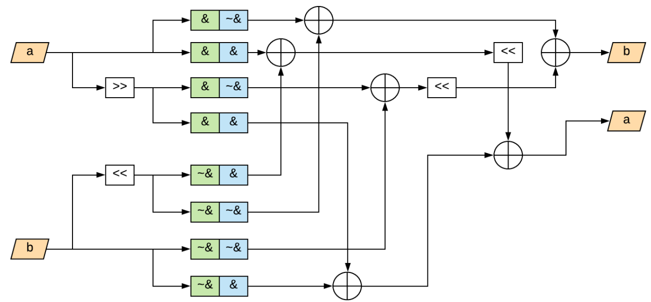

Fig. 24. The optimized double permutation kernel