



## Accelerating Encryption Algorithms using Impulse C

**Scott Thibault, President**

Green Mountain Computing Systems, Inc.

**David Pellerin, CTO**

Impulse Accelerated Technologies, Inc.

Copyright© 2010 Impulse Accelerated Technologies, Inc.

### Overview

Encryption and decryption code that has not been written with hardware compilation in mind can result in less than optimal results when those results are measure in terms of process latencies, data throughput and size of the generated logic. This is because the fundamental target of compilation, in this case an FPGA and its constituent hardware resources, is quite different from a traditional processor, and the C language is by design optimized for processor-based architectures. By using some relatively simple C programming techniques, however, it is possible to dramatically accelerate the performance of many types of algorithms.

This white paper presents some of these techniques, using the 3DES (triple DES encryption) algorithm as a basis for discussion.

*Note: this application note is adapted from **Practical FPGA Programming in C** by David Pellerin and Scott Thibault, published in 2005 by Pearson.*

### Examining the 3DES example

In the *Practical FPGA Programming in C*, an example of data encryption using the triple-DES encryption algorithm was presented, using a legacy C implementation of the algorithm as a basis for the application. The purpose of this example was to demonstrate how to prototype a hardware implementation in an FPGA, using C code that had previously been written for and compiled to an embedded processor. When hardware simulation of the resulting RTL was performed, it was determined that the rate at which the algorithm could generate encrypted blocks of data was approximately one block approximately every 150 cycles. This is rate that is faster in terms of clock cycles than what could be achieved in a software implementation (using an embedded Xilinx MicroBlaze processor for comparison), but given the much higher

clock rates possible in modern processors this is unlikely to be a satisfactory result for FPGA-based hardware acceleration.

To increase the performance of this application, we begin by taking a closer look at the algorithm itself and how it was initially implemented as an Impulse C process. To simplify the example for the purpose of discussion we will focus on only one of the three single-DES passes of this algorithm. The following are excerpts of the original legacy C code that processes one 64-bit block of eight characters:

```
left = ((unsigned long)block[0] << 24)
      | ((unsigned long)block[1] << 16)
      | ((unsigned long)block[2] << 8)
      | (unsigned long)block[3];
right = ((unsigned long)block[4] << 24)
       | ((unsigned long)block[5] << 16)
       | ((unsigned long)block[6] << 8)
       | (unsigned long)block[7];

work = ((left >> 4) ^ right) & 0x0f0f0f0f;
right ^= work;
left ^= work << 4;
work = ((left >> 16) ^ right) & 0xffff;
right ^= work;
left ^= work << 16;
work = ((right >> 2) ^ left) & 0x33333333;
left ^= work;
right ^= (work << 2);
work = ((right >> 8) ^ left) & 0xff00ff;
left ^= work;
right ^= (work << 8);
right = (right << 1) | (right >> 31);
work = (left ^ right) & 0xaaaaaaaa;
left ^= work;
right ^= work;
left = (left << 1) | (left >> 31);
```

The preceding block prepares the data by arranging the 64-bit block into two 32-bit values and performing some initial permutations on the data. Simple bit-level manipulation such as this will be easily converted by the CoDeveloper compiler into hardware logic requiring only a single cycle. Reading the data from the **block** array, however, will require at least eight cycles because only one value per cycle can be read from memory. *Reducing the number of accesses required to memory is one key optimization that we will consider for this algorithm.*

Next, consider the following excerpts of this example, in which a relatively complex macro has been defined as part of the calculation:

```
#define F(l,r,key){\
    work = ((r >> 4) | (r << 28)) ^ key[0];\
    l ^= Spbox[6][work & 0x3f];\
    l ^= Spbox[4][(work >> 8) & 0x3f];\
    l ^= Spbox[2][(work >> 16) & 0x3f];\
    l ^= Spbox[0][(work >> 24) & 0x3f];\
    work = r ^ key[1];\
    l ^= Spbox[7][work & 0x3f];\
    l ^= Spbox[5][(work >> 8) & 0x3f];\
```

```

    l ^= Spbox[3][(work >> 16) & 0x3f];\
    l ^= Spbox[1][(work >> 24) & 0x3f];\
}

```

This macro is subsequently used in the following section of the algorithm:

```

F(left, right, Ks[0]);
F(right, left, Ks[1]);
F(left, right, Ks[2]);
F(right, left, Ks[3]);
F(left, right, Ks[4]);
F(right, left, Ks[5]);
F(left, right, Ks[6]);
F(right, left, Ks[7]);
F(left, right, Ks[8]);
F(right, left, Ks[9]);
F(left, right, Ks[10]);
F(right, left, Ks[11]);
F(left, right, Ks[12]);
F(right, left, Ks[13]);
F(left, right, Ks[14]);
F(right, left, Ks[15]);

```

This code performs the main operation of the encryption/decryption. There are some simple bit manipulations being performed, but there are also many references to both one and two-dimensional array elements. Each instantiation of the macro **F** performs eight loads from the **Spbox** array, and **F** itself is instantiated 16 times for a total of 128 loads. From that we can easily see that this block of code is going to require at least 128 cycles.

The final block of code performs some final permutations on the two 32-bit values and arranges the result in the output array:

```

right = (right << 31) | (right >> 1);
work = (left ^ right) & 0xaaaaaaaa;
left ^= work;
right ^= work;
left = (left >> 1) | (left << 31);
work = ((left >> 8) ^ right) & 0xff00ff;
right ^= work;
left ^= work << 8;
work = ((left >> 2) ^ right) & 0x33333333;
right ^= work;
left ^= work << 2;
work = ((right >> 16) ^ left) & 0xffff;
left ^= work;
right ^= work << 16;
work = ((right >> 4) ^ left) & 0x0f0f0f0f;
left ^= work;
right ^= work << 4;

block[0] = (unsigned char) (right >> 24);
block[1] = (unsigned char) (right >> 16);
block[2] = (unsigned char) (right >> 8);
block[3] = (unsigned char) right;
block[4] = (unsigned char) (left >> 24);
block[5] = (unsigned char) (left >> 16);

```

```
block[6] = (unsigned char) (left >> 8);
block[7] = (unsigned char) left;
```

Again, the simple bit-manipulations being shown here will be easily compiled into a single-cycle operation. Storing the data into the block array will require at least eight cycles, however, because only one data value can be written to memory per cycle.

Using this original, unmodified code the CoDeveloper tools generate a hardware implementation that uses close to 3000 slices in a Xilinx Virtex II FPGA and perform just 10.6 times faster than an equivalent algorithm running in software on an embedded MicroBlaze processor. In the following sections we will demonstrate a number of refinements that can be applied in this example to obtain better results, both in terms of clock cycles and in the size of the resulting logic.

## Refinement 1: reducing size by introducing a loop

As described above, a macro (**F**) was used to repeat the 16 steps of the core processing. In software, this might run a little faster than using a loop. In hardware, though, using the macro in such a way means that we are duplicating that code 16 times creating a potentially much larger implementation. The obvious solution here is to reduce code repetition by introducing a loop.

Without making major modifications we can introduce a loop as follows:

```
for (i=0; i<16; i++) {
    F(left, right, Ks[i]);
    i++;
    F(right, left, Ks[i]);
}
```

Regenerating hardware using the Impulse C tools, we obtain an implementation using about 1500 slices in the Xilinx device. The looping instructions introduce some extra delay, but the performance is still 9.4 times faster than the unmodified software implementation. In short, for a small hit in performance the design size has been cut roughly in half. Refinement four will shed a somewhat different light on this performance difference.

## Refinement 2: array splitting

One of the most significant performance advantages of hardware implementations is the ability to access multiple memories in the same cycle. In the typical software implementation, each CPU is connected to one or more memories by a single bus. In that scenario, the software can only access data from at most one memory per bus transaction (*i.e.*, one or more cycles).

When generating hardware, we have the opportunity to generate any topology necessary including separate connections to multiple memories. The Impulse C tools generate a

separate, individually connected memory for each array used in the process. For example, consider the following code sample:

```
void run()
{
    co_int32 i,A[4],B[4],C[4];

    for (i=0; i<4; i++) {
        A[i]=B[i]+C[i];
    }
}
```

This process will generate three separate memories for the arrays A, B, and C, each with a dedicated connection to the computation hardware. The assignment statement that would require at least three bus transactions to execute in software can now be performed in a single cycle by performing reads from B and C and writing to A simultaneously.

Returning to our DES example, notice that there are two relatively large arrays involved in the core computation, **Spbox** and **Ks**. These are each implemented in their own memory, however they still involve multiple accesses to the same array and thus require multiple cycles when used in this algorithm. Let's look at the memory access involved in the core computation as specified in macro **F** and in the main code loop:

```
#define F(l,r,key){\
    work = ((r >> 4) | (r << 28)) ^ key[0];\
    l ^= Spbox[6][work & 0x3f];\
    l ^= Spbox[4][(work >> 8) & 0x3f];\
    l ^= Spbox[2][(work >> 16) & 0x3f];\
    l ^= Spbox[0][(work >> 24) & 0x3f];\
    work = r ^ key[1];\
    l ^= Spbox[7][work & 0x3f];\
    l ^= Spbox[5][(work >> 8) & 0x3f];\
    l ^= Spbox[3][(work >> 16) & 0x3f];\
    l ^= Spbox[1][(work >> 24) & 0x3f];\
}

. . .

for (i=0; i<16; i++) {
    F(left,right,Ks[i]);
    i++;
    F(right,left,Ks[i]);
}
```

Each iteration of the above loop requires four loads from **Ks**, and eight loads from **Spbox**. The need for eight loads from **Spbox** implies that at least eight cycles will be required because (at most) one value can be read from **Spbox** per cycle. We notice, however, that **Spbox** is a multi-dimensional array and that the row indices are all constants. Because the indices are constant it is possible to actually split this array into individual arrays for each row.

Likewise, we see that **Ks** is a multi-dimensional array and the column indices are all constant. **Ks** can therefore be split into individual arrays for each column. The resulting code looks like this:

```

#define F(l,r,key0,key1){\
    work = ((r >> 4) | (r << 28)) ^ key0;\
    l ^= Spbox6[work & 0x3f];\
    l ^= Spbox4[(work >> 8) & 0x3f];\
    l ^= Spbox2[(work >> 16) & 0x3f];\
    l ^= Spbox0[(work >> 24) & 0x3f];\
    work = r ^ key1;\
    l ^= Spbox7[work & 0x3f];\
    l ^= Spbox5[(work >> 8) & 0x3f];\
    l ^= Spbox3[(work >> 16) & 0x3f];\
    l ^= Spbox1[(work >> 24) & 0x3f];\
}

. . .

i=0;
do {
    F(left,right,Ks0[i],Ks1[i]);
    i++;
    F(right,left,Ks0[i],Ks1[i]);
    i++;
} while (i<16);

```

This is only a minor change to the code, but the result is that each of the new **Spbox** and **Ks** arrays are only accessed twice per iteration, reducing the total cycle count required to execute the process. After regenerating the hardware with CoDeveloper, we obtain an implementation that is slightly smaller in size and 19.5 times faster than the unmodified software implementation running on MicroBlaze. In this case, array splitting has doubled the performance over our first refinement. The result is also smaller in size due to the fact that some address calculation have been eliminated by reducing the dimension of **Spbox** and **Ks**. *Array splitting is therefore a useful technique for reducing both size and cycle counts.*

### Refinement 3: improving communication performance

Sometimes the most significant bottleneck in a hardware/software implementation is the hardware/software communication interface. In the 3DES example, the data is assumed to be both produced and consumed by the connected CPU (a Microblaze processor connected via FSL links). This results in a significant amount of data crossing over the software/hardware interface. When the legacy code was initially ported to Impulse C hardware, no attention was given to the communication overhead and the resulting stream implementation is very inefficient. Consider the following code, which is located at the start of the main processing loop:

```

while (co_stream_read(blocks_in, &block[0], sizeof(uint8)) == co_err_none) {
    for (i = 1; i < BLOCKSIZE; i++) {
        co_stream_read(blocks_in, &block[i], sizeof(uint8));
    }
}

```

Here, each 64-bit block is being transferred eight bits at a time (one character) over an 8-bit stream even though the hardware is connected to the CPU via a 32-bit bus. As with memories,

streams require at least one cycle per read/write operation, so this code will require at least eight cycles. Furthermore, consider the code immediately following the stream reads:

```
left = ((unsigned long)block[0] << 24)
      | ((unsigned long)block[1] << 16)
      | ((unsigned long)block[2] << 8)
      | (unsigned long)block[3];
right = ((unsigned long)block[4] << 24)
       | ((unsigned long)block[5] << 16)
       | ((unsigned long)block[6] << 8)
       | (unsigned long)block[7];
```

After reading the data eight bits at a time, this code rearranges the data into two 32-bit values, which requires eight loads from the block array and therefore at least eight more cycles. The same situation is also present in the output communications at the end of the main processing loop.

Rewriting the streams interface to use 32-bit streams will significantly improve performance. The input and output communication can be rewritten as follows:

```
while (co_stream_read(blocks_in, &left, sizeof(left))==co_err_none) {
    co_stream_read(blocks_in, &right, sizeof(unsigned long));
...
    co_stream_write(blocks_out, &right, sizeof(unsigned long));
    co_stream_write(blocks_out, &left, sizeof(unsigned long));
}
```

Notice that the block array has been eliminated altogether. Obviously this change to the streams specification requires a corresponding change to the producer and consumer processes (which in this example are represented by a single “software test bench” process running on the embedded MicroBlaze processor) but this is a simple change.

Regenerating hardware with this new communication scheme we obtain system performance 48 times faster than the software implementation (which was also modified to access the block data in 32 bit chunks). Thus performance is nearly 2.5 times better than the previous result (refinement 2).

## Refinement 4: loop unrolling

Refinement 1 began by reducing the size of the generated hardware by recoding some repeated code as a loop. Up to refinement 3, we considered performance improvements that kept the size fairly constant and in the end achieved an overall system speedup of 48 over the initial software-based prototype. In the remaining sections we will abandon our attempts to maintain the size and push instead for the maximum performance, as measured in cycle counts.

Refinement 1 introduced a loop and consequently some overhead to the core DES computation in favor of the reduced size. The performance loss was quite modest when considered in isolation; however, that was before optimizing the statements that make up the

body of the new loop. Let's see what impact loop unrolling will have now, after the substantial optimizations of refinements 2 and 3. The following is how the code appears after unrolling the inner loop introduced in refinement 1:

```
F(left, right, Ks0[0], Ks1[0]);
F(right, left, Ks0[1], Ks1[1]);
F(left, right, Ks0[2], Ks1[2]);
F(right, left, Ks0[3], Ks1[3]);
F(left, right, Ks0[4], Ks1[4]);
F(right, left, Ks0[5], Ks1[5]);
F(left, right, Ks0[6], Ks1[6]);
F(right, left, Ks0[7], Ks1[7]);
F(left, right, Ks0[8], Ks1[8]);
F(right, left, Ks0[9], Ks1[9]);
F(left, right, Ks0[10], Ks1[10]);
F(right, left, Ks0[11], Ks1[11]);
F(left, right, Ks0[12], Ks1[12]);
F(right, left, Ks0[13], Ks1[13]);
F(left, right, Ks0[14], Ks1[14]);
F(right, left, Ks0[15], Ks1[15]);
```

Regenerating hardware using the Impulse C tools on this new revision produces a hardware implementation that requires a little over 2000 slices and performs *nearly 80 times faster than the software implementation*. This shows that the overhead introduced by a loop can be significant when the loop body is small.

Note that, rather than duplicating the body eight times and substituting constants for **I** as we have done above, Impulse C also has an UNROLL pragma that essentially does the same thing for loops with constant values for their iteration values. The UNROLL pragma performs unrolling for you as a preprocessing step before other optimizations are performed. For this example, however, it is more convenient to unroll the loop by hand as a prelude to refinement 5.

## Refinement 5: pipelining the main loop

Along with memory optimization, pipelining is also one of the most significant types of optimizations to consider when generating hardware. Pipelining can be applied to loops that require two or more cycles per iteration and that do not have dependencies between iterations. The DES loop processes independent blocks of data with no dependencies between iterations, so it should be possible to apply pipelining.

The goal of pipelining is to execute multiple iterations of a loop in parallel much like the assembly line of a factory. The core DES computation contains 16 instantiations of the F macro, and these instantiations can be called  $F_0, F_1, \dots, F_{16}$ . Iteration  $F_{16}$  is dependent on  $F_{15}$ , which is in turn dependent on  $F_{14}$ , and so on so that  $F_1$  through  $F_{16}$  must be executed sequentially. Imagine that  $F_1$  through  $F_{16}$  are stations in an assembly line processing a 64-bit block ("raw material") and producing a 64-bit block "product" on the other end. At the start of the line,  $F_1$  receives block<sub>1</sub>, does some processing and passes the result on to  $F_2$ . While  $F_2$  is processing block<sub>1</sub>,  $F_1$  is idle so it can go ahead and start processing block<sub>2</sub>. By the time block<sub>1</sub> makes it to station  $F_{16}$ , station  $F_1$  is processing block<sub>16</sub> and all 16 stations are busy operating in

parallel. The result of building such an assembly line is we can generate one block of output every cycle even though it still takes 16 cycles to process one block. The cost (continuing the assembly line metaphor) is that each station needs its own equipment in order for them to operation in parallel.

To generate a pipeline using Impulse C, we simply insert the CO PIPELINE pragma inside the main loop as shown below:

```
while (co_stream_read(blocks_in, &left, sizeof(unsigned long))
      == co_err_none) {
    #pragma CO PIPELINE
    co_stream_read(blocks_in, &right, sizeof(unsigned long));
}
```

Unfortunately, simply adding the pipeline pragma to the refinement 4 code generates a pipeline that only produces one result every 16 cycles. Why did the pipeline fail? The reason is each instance of the **F** macro requires access to the **Spbox** and **Ks** arrays, which prevents them from being executed in parallel.

In order to generate an effective pipeline in this application, each instance of **F** must have its own copy of the **Spbox** arrays. The **Ks** array could also be duplicated, but since the array is small we use another technique, which is to convert the **Ks** array into a register bank. Registers can be read by many sources in the same cycle. To implement **Ks** as a register bank, all indices to **Ks** must be constants. The resulting code appears as follows:

```
#define F(l,r,key0,key1,sp0,sp1,sp2,sp3,sp4,sp5,sp6,sp7){\
    work = ((r >> 4) | (r << 28)) ^ key0;\
    l ^= sp6[work & 0x3f];\
    l ^= sp4[(work >> 8) & 0x3f];\
    l ^= sp2[(work >> 16) & 0x3f];\
    l ^= sp0[(work >> 24) & 0x3f];\
    work = r ^ key1;\
    l ^= sp7[work & 0x3f];\
    l ^= sp5[(work >> 8) & 0x3f];\
    l ^= sp3[(work >> 16) & 0x3f];\
    l ^= sp1[(work >> 24) & 0x3f];\
}

. . .

F(left,right,Ks0[0],Ks1[0],
  Spbox00,Spbox10,Spbox20,Spbox30,Spbox40,Spbox50,Spbox60,Spbox70);
F(right,left,Ks0[1],Ks1[1],
  Spbox01,Spbox11,Spbox21,Spbox31,Spbox41,Spbox51,Spbox61,Spbox71);
F(left,right,Ks0[2],Ks1[2],
  Spbox02,Spbox12,Spbox22,Spbox32,Spbox42,Spbox52,Spbox62,Spbox72);
F(right,left,Ks0[3],Ks1[3],
  Spbox03,Spbox13,Spbox23,Spbox33,Spbox43,Spbox53,Spbox63,Spbox73);
F(left,right,Ks0[4],Ks1[4],
  Spbox04,Spbox14,Spbox24,Spbox34,Spbox44,Spbox54,Spbox64,Spbox74);
F(right,left,Ks0[5],Ks1[5],
  Spbox05,Spbox15,Spbox25,Spbox35,Spbox45,Spbox55,Spbox65,Spbox75);
F(left,right,Ks0[6],Ks1[6],
  Spbox06,Spbox16,Spbox26,Spbox36,Spbox46,Spbox56,Spbox66,Spbox76);
F(right,left,Ks0[7],Ks1[7],
```

```

    Spbox07, Spbox17, Spbox27, Spbox37, Spbox47, Spbox57, Spbox67, Spbox77) ;
F (left, right, Ks0 [8], Ks1 [8],
    Spbox08, Spbox18, Spbox28, Spbox38, Spbox48, Spbox58, Spbox68, Spbox78) ;
F (right, left, Ks0 [9], Ks1 [9],
    Spbox09, Spbox19, Spbox29, Spbox39, Spbox49, Spbox59, Spbox69, Spbox79) ;
F (left, right, Ks0 [10], Ks1 [10],
    Spbox0a, Spbox1a, Spbox2a, Spbox3a, Spbox4a, Spbox5a, Spbox6a, Spbox7a) ;
F (right, left, Ks0 [11], Ks1 [11],
    Spbox0b, Spbox1b, Spbox2b, Spbox3b, Spbox4b, Spbox5b, Spbox6b, Spbox7b) ;
F (left, right, Ks0 [12], Ks1 [12],
    Spbox0c, Spbox1c, Spbox2c, Spbox3c, Spbox4c, Spbox5c, Spbox6c, Spbox7c) ;
F (right, left, Ks0 [13], Ks1 [13],
    Spbox0d, Spbox1d, Spbox2d, Spbox3d, Spbox4d, Spbox5d, Spbox6d, Spbox7d) ;
F (left, right, Ks0 [14], Ks1 [14],
    Spbox0e, Spbox1e, Spbox2e, Spbox3e, Spbox4e, Spbox5e, Spbox6e, Spbox7e) ;
F (right, left, Ks0 [15], Ks1 [15],
    Spbox0f, Spbox1f, Spbox2f, Spbox3f, Spbox4f, Spbox5f, Spbox6f, Spbox7f) ;

```

As a result of this change, each instance of **F** has its own copy of the **Spbox** data, and **Ks0** and **Ks1** will be converted to registers by the compiler.

Running the compiler now results in a 20 stage pipeline that produces one block every two cycles. Two cycles are required because of the 32-bit communication stream requires two cycles to transfer the blocks to and from the CPU. If a 64-bit interface were available then the pipeline would generate a single block every cycle. The overall system performance is now 425 times faster than the software implementation. The size of the hardware has also increased to over 3000 slices due to the duplicated resources required to implement a pipeline.

## Summary

This application note has described a number of C coding techniques that may be used to increase the performance and reduce the size of C code. Although you can in many cases obtain satisfactory, prototype-quality results without considering the tradeoffs of memory access, array specifications, instruction-level dependencies, loop coding styles and instruction pipelines, you should consider these factors carefully when creating applications that require the highest practical performance and throughput. The techniques described in this paper are a good starting point and are applicable to a wide variety of FPGA-based applications.

## About Impulse

Impulse is a world leader in software-to-hardware technologies enabling FPGA-based embedded and accelerated computing. Impulse products and services allow software and hardware development teams to use familiar and productive software programming methods when creating FPGA applications.

Impulse CoDeveloper™ includes the Impulse C™ software-to-hardware compiler, interactive parallel optimizer, and Platform Support Packages supporting a wide range of FPGA-based systems. Impulse tools are compatible with all popular FPGA platforms.