FPGA implementation of the Rijndael algorithm

Benjamin Leperchey, Charles Hymans

June 9, 2000

1 Summary

1.1 Goals

The goal of this project is the implementation of the Rijndael algorithm (one of the AES candidates - see []) on a Pamette. This cipher works on blocks of NBx8 bytes, for NB between 4 and 6. In this implementation, we decided to work with NB=4. The blocks are therefore 128bits long.The encryption is divided in 10 rounds, the last one being slightly different from the others, one step being removed. Data goes by words of 32 bits - 4 bytes actually - so a complete round takes four cycles. This allows us to keep the encoder very small, and to unfold the loop. The division of the blocks in columns also allows different sizes of blocks, with little enlargement of the design (only bigger counters, and a bigger key).

In our basic design, data goes through the design 10 times, one for each round, and a complete encoding takes 40 cycles. We intended to put several copies of the design in the Pamette, in order to reduce the loop length, but we did not have enough time. This could be easily done, though : our designs (the encoder and the decoder) fit in half a chip of a XC4044, so this would only be a matter of counters.

1.2 The Rijndael algorithm

The field used for all arithmetic operations is $GF(2^8)$, which can be seen as the set of polynomials over $GF(2)$ modulo an irreductible polynomial of degree 8 - $x^8+x^4+x^3+x+1$ for example. The addition is a simple XOR, and the multiplication is the usual multplication of polynomials, followed by a modulo.

The rounds of the encoder are composed of :

1. RoundKey : only for the first round, the key is XORed with the block.
2. ShiftRow : the 4 lines of the block are shifted to the left, by 0,1,2 and 3
3. ByteSub : each byte is changed by a non-linear fonction
4. MixColumn : a linear function is applied to each column of the block. It is a multiplication by a fixed polynomial over $GF(2^8)$, modulo x^4+1. This step is skipped in the last round.
5. RoundKey : each byte is XORed with the key. Key length is 128 bits, but a proper computing gives a 1280+128 bits key, 128 bits for the first XOR (it is the original key, and 128 bits by round. The key expansion is not computed by hardware, since it is not changed often. For the time being, it is put into a ROM, and we intend to load it dynamically before any encoding.

Note : in our implementation, ShiftRow comes before ByteSub, which has no consequence on the results, since ByteSub works on the value of the bytes, whereas ShiftRow which works on their position.

The inverse cipher is very similar to the cipher : most of the steps can be switched, and some of them are their own inverse, or they are very similar. The only problem is the MixColumn inverse : the polynomial chosen is easy to use, but its inverse is much more complicated. In the end, we have :

1. RoundKey : only for the first round, the last part of the expanded key is XORed with the block.
2. MixColumn : a linear function is applied to each column of the block. It is a multiplication by a fixed polynomial over $GF(2^8)$, modulo $x^4 + 1$. This step is skipped in the first round.
3. ShiftRow : the 4 columns of the block are shifted to the right, by 0,1,2 and 3
4. ByteSub : each byte is changed by a non-linear fonction
5. RoundKey : each byte is XORed with the key, but the order is reversed.

For more precise information of the Rijndael algorithm, see [].

1.3 Area, timings

Our design fits into of 20x40 array of cells, including the interface with the PCI bus. The Pamette we used (the XC4044) contains four 40x40 chips, which would have allowed us to put several encoders/decoders. If we had put 5 copies of the encoder, there would only be 2 rounds, and the bandwith would be multiplied by 5, so a complete encoding would take only 8 cycles. Since we compute two encryption at the same time - cf ShiftRow -, we would have 16 cycles for two blocks, which is the best we can expect : 8 cycles for reading, 8 cycles for writing.

The inside of our encoder can run at 50 MHz, which would allow a bandwith of 160 Mbit/s, and the unfolded design would run at 800 Mbit/s. The design is actually limited to 40 MHz by the way out of the pamette.

The major problem we encountered was the interface : at this point, the design runs and stops many times during the uploading and downloading, which slows down the entire process. On the contrary, once the 8 words are set, the design runs at full speed. The expected results with the unfolded design are therefore impossible.

1.4 Interface

In the design, data runs in 32bits words, ie one column of the Rijndael block by cycle. This way, with four cycles, a complete round is achieved. Actually, the 32bits words are divided in four 8bits words, since this is the convenient structure for ShiftRow and ByteSub.

In order to implement ShiftRow efficiently, two blocks are encoded at the same time. Therefore, the design reads 8 words in the 8 first cycles, and writes 8 words when all the rounds are complete.

We have put some delay between the output of the design and the writing on the bus : this way, the output comes just after the input. The interface stops the design clock when there is not enough data ready in the input FIFO, or when there is not enough room on the output FIFO. Since the FIFOs can hold only 16 words,

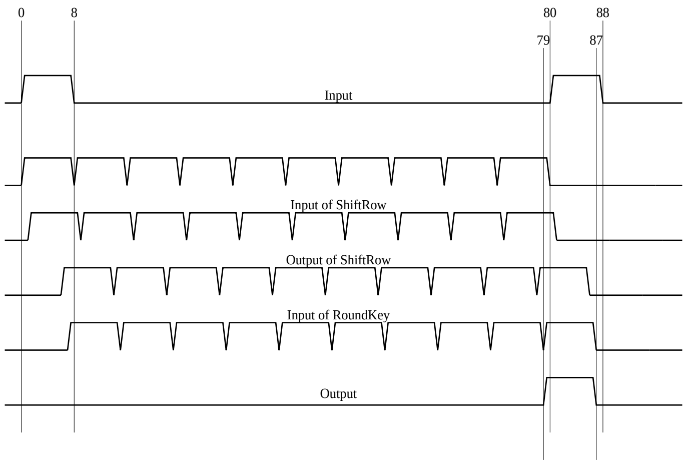

Figure 1: Read and write cycles

and the signal for stopping the clock can take up to 12 cycles, the input FIFO must hold 13 to 16 words and the output FIFO 0 to 3 words for the design to run.

The software interface must therefore write two words, then read two words back, and so on. The design gives data back at the time he takes some, so this order works. The problem is, we have to wait for the design to «eat» the words before we can send another one, and the small range allowed - four - makes the design stop, then start again when the words have arrived. Another problem is, since we can only write two words at the same time, we cannot take advantage of the BUS bursts. The throughput is therefore rather low.

2 Placement in the designs

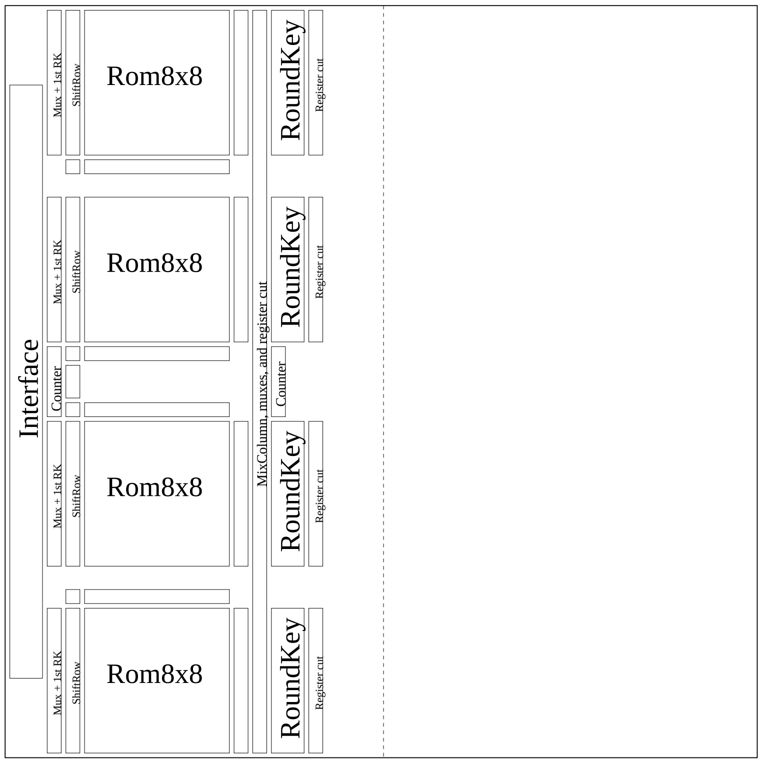

Figure 2: The complete encoder, with correct sizes

3 Details of the parts

3.1 ShiftRow : Ram8x32

ShiftRow shifts the *rows* of the block by 0,1,2,3, so we need all the columns of one block to get the resulting column. Two blocs are computed serially : what comes in ShiftRow is written in one block, what goes out is read from the other one, and the two blocks are exchanged every four cycles.

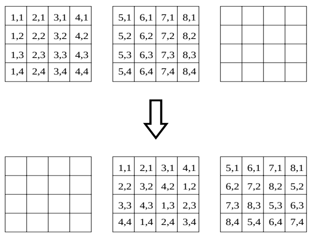

Figure 3: ShiftRow

In this figure, each column represents a cycle. There is a four cycles delay between the input and the output. Since we have put another register, for retiming purposes, there is actually a five cycles delay.

We need a 8x4 words RAM, ie a 8x32 bits, with four distinct addresses for writing and four other for reading (only one read-address is needed for the encoding, but for the decoding, we need four read-addresses and one write address). We use CLB as RAMD, each CLB is used as a 16x1 bit RAM. Addresses are coded on four bits, the last one being set to zero.

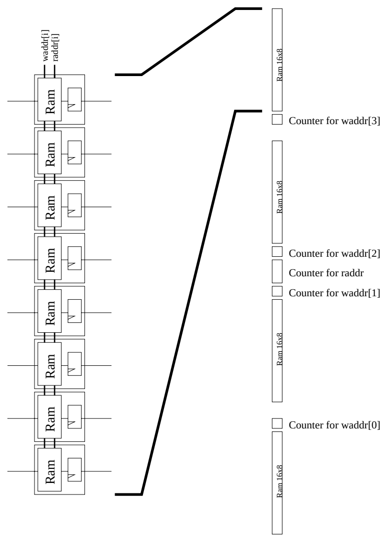

Figure 4: Ram8x32

The size of this structure is 4 blocks of 1x8 CLBs. We must also put 4 counters mod 4 and one counter mod 8 in the same column, in the 8 CLBs left : for the encoding, the only read-address is given by a counter mod 4, plus one bit that indicates which block data is read in, and the four write-addresses are given by four shifted counters mod 4, the last bit being the opposite of the read-address one.

For the decoder, we can either exchange read and write adresses, or write in diagonal the other way (by setting the initial values of the counter to their opposite), and read the same way - column by column.

3.2 ByteSub : Rom8x8

The *S-boxes* are too complex to be implemented directly : we use look-up tables (LUT), initially loaded with the correct values. Each of them takes 8 bit and gives 8 bit back, we need four 8x8bit ROM.

The 8bit input is divided in 3 parts : the bits 0,1,2,3 are used as an adress for 16bit ROMs, the bits 4,5,6 are used to control which ROM is controlling the bus, and the last bit chooses which bus will be read.

Each S-Box is 9x9 CBLs. The main part is 8x8 CLBs : each CLB contains two 16x1 ROMs, connected to a bus by a TBuf. There is two buses by line of CLB. The ninth CLB is used for the decoders which control the TBufs : they take the bits 4,5,6, and the control is set to one iff the value matches the row. The decoders can be put above or under the ROMs, by calling *placement_up* or *placement_down*.

Horizontally, we have 8 CLBs, one for each value of the bits 4,5,6, plus one for the mux between the two buses, controlled by bit 7.

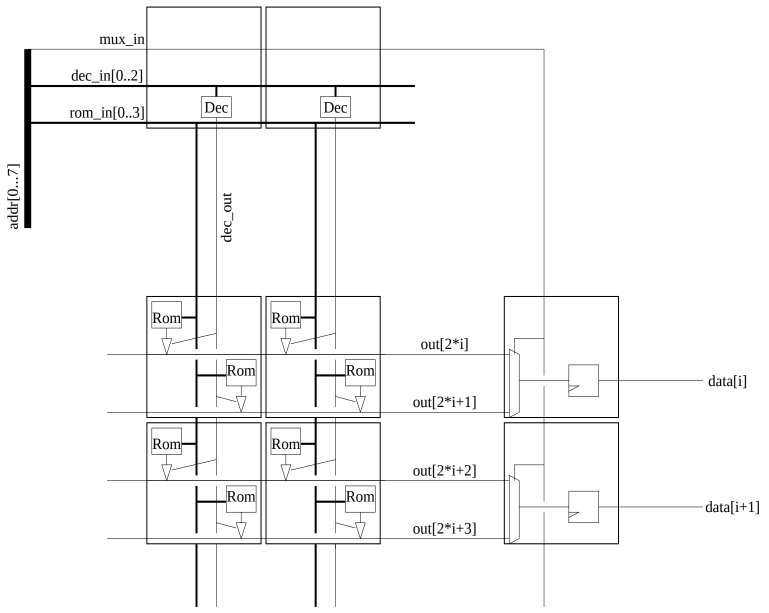

Figure 5: Rom8x8

3.3 MixColumn, MixColumn inverse

MixColumn is very easy to implement : it can be done is 3 XORs. If $A[1..4]$ are the four 8bit words of input, and $B[1...4]$ the four output words, we have

```
tmp = A[1] ^ A[2] ^ A[3] ^ A[4];
tm = A[1] ^ A[2]; tm = xtime(tm); B[1] = tm ^ tmp;
tm = A[2] ^ A[3]; tm = xtime(tm); B[2] = tm ^ tmp;
tm = A[3] ^ A[4]; tm = xtime(tm); B[3] = tm ^ tmp;
tm = A[4] ^ A[1]; tm = xtime(tm); B[4] = tm ^ tmp;
```

where *xtime* is the multiplication by 00000010 in $GF(2^8)$.

We can implement *xtime* as a shift and a XOR against 00011011 if the highest bit is 1 :

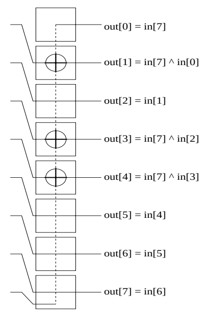

Figure 6: xtime

The matching bits of the four 8 bits words are XORed, we get all_xor. Then, only two are XORed, we get $couple\_xor$. We apply $xtime$ on these $couple\_xor$, and we XOR $xtime(couple\_xor)$ and all_xor.
No placement has been done, since this is mostly a problem of routing.

The inverse is much more difficult to implement : MixColumn is a multiplication by the polynomial $00000011x^3 + 00000001x^2 + 00000001x + 00000010$ modulo $x^4 + 1$. It has of course an inverse, $00001011x^3 + 00001101x^2 + 00001001x + 00001111$, but the constant multiplication is much more complex. We have to compute $00001001.a$, $00000100.a$ and $00000010.a$ for the four a. This can be done by $xtime$ and XORs. Then we must take the right values and XOR them to get the expected result : for

$$b[0] = 00001111.a[0] + 00001001.a[1] + 00001101.a[2] + 00001011.a[3]$$

we have to XOR

- $00001001.a[0]$
- $00000100.a[0]$
- $00000010.a[0]$
- $00001001.a[1]$
- $00001001.a[2]$
- $00000100.a[2]$
- $00001001.a[3]$
- $00000010.a[3]$

We have 8 values, ie 3 XORs by bit. The indexes for other values can be obtained by a circular permutation (for the second 8bit words, 00012233 becomes 11123300).
The size of this structure is two CLBs for the xtimes - the XOR for $00001001.a$ being put with the last one, three for the register cut, but one can be shared with the last xtime and the XOR, and one or two for the three last XORs. This make 5 or 6 CLBs.

3.4 RoundKey

It is only an XOR against the expanded key, so we need four 40x8 ROMs, and a counter. Since we have two blocks at the same time, we must have the columns 0,1,2,3,0,1,2,3 then 4,5,6,7,4,5,6,7... in this order coming from the ROM. We use a special counter, that is a counter mod 80, whose third bit is ignored, the output is bits 654310.

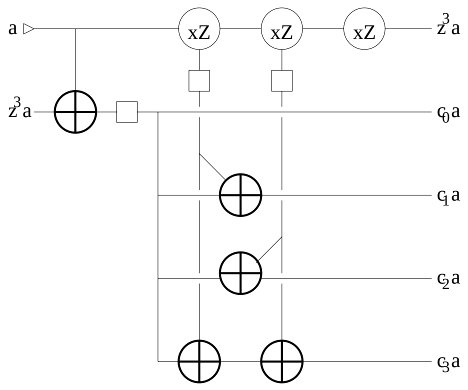

Figure 7: Multipliers in MixColumn inverse

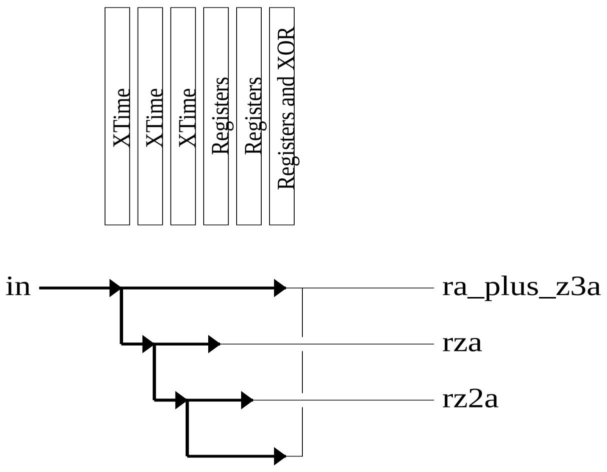

Figure 8: Placement of the multipliers

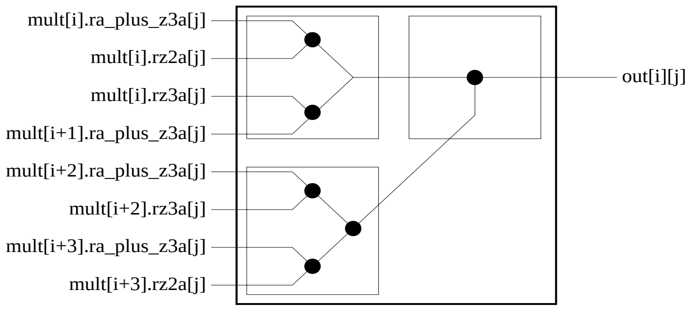

Figure 9: XORs for the MixColumnInverse

4 Pipelining, loop unrolling

4.1 Pipelining

The introduction of registers is easy, and necessary because of the implementation of ShiftRow. Since our design works by groups of 4 cycles, four cuts are needed. With this four cycles delay, and the four cycles delay introduced by ShiftRow, we get a synchronized input : 8 words, then the 8 words after one round, then the 8 words after two rounds...

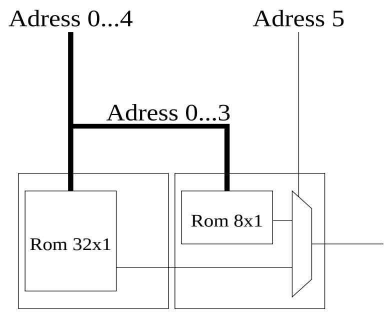

Figure 10: Round Key addition

The operations of Rijndael are very simple, except the ByteSub, which cannot be split easily. We therefore decided to put a register just before and just after the S-Boxes. The two remaining cut were set after after MixColumn, and before the loop.

4.2 Loop unrolling

We wanted to dispose several encoders/decoders on the same FPGA. This way, the number of rounds in each encoder could be reduced, and the design could have a better troughput. If we had put 5 on the whole board, data would go trough each encoder only twice - instead of 10 times - we could therefore treat 5 times more data. We would have a beginning encoder, with the initial round key addition, (it is also the fifth, but there is no initial round key addition in the fifth round), 3 intermediate encoders, with no initial addition, and no test for the MixColumn, and a last encoder, with a test for the MixColumn.

Each encoder would be a bit smaller than the "general purpose" encoder, because some parts could be removed, and we only need a part of the expanded key for each encoder, whereas the unique encoder needs the whole round key in ROM. The counters would also be smaller and simpler (modulo 16 and not 80).

We wanted to dispose the five encoders this way : which is rather easy, since

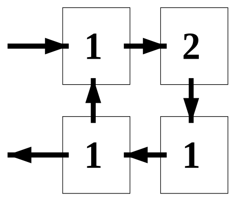

Figure 11: Five encoders on a FPGA

the designs fits (even with the complete key and the interface) in half a chip.