Introduction

In this exercise we first explain digital circuit design for the AES block cipher and the corresponding VHDL source code. Afterwards, we ask you to optimize the code.

Algorithm

To understand digital design the best approach is to do it through a known algorithm, which in our case is the AES block cipher for 128 bits keys. The block cipher is shown in Algorithm 1 (encryption and decryption), and Algorithm 2 (the key schedule). For more details on AES, you can have a look at the standard itself [1], the slides of the Crypto course at RU or other resources available online. To prepare some intermediate values, I suggest using this excel spreadsheet https://www.nayuki.io/page/aes-cipher-internals-in-excel or to find any other code available i.e C, Phyton. For the design to work we need a circuit that can perform all functions: $AddRoundKey$, $SubBytes$, $ShiftRows$, $MixColumns$, $InvSubBytes$, $InvShiftRows$, $InvMixColumns$, $RotWord$, $SubWord$ and some other parts related to the key schedule.

Algorithm 1 AES Encryption and Decryption Algorithms

Require: P is the plaintext.
Require: K is all round keys.
Ensure: C is the ciphertext.
  $C \leftarrow AddRoundKey(P, K_0)$
  for $i \leftarrow 1$ **to** 9 **by** 1 **do**
    $C \leftarrow SubBytes(C)$
    $C \leftarrow ShiftRows(C)$
    $C \leftarrow MixColumns(C)$
    $C \leftarrow AddRoundKey(C, K_i)$
  end for
  $C \leftarrow SubBytes(C)$
  $C \leftarrow ShiftRows(C)$
  $C \leftarrow AddRoundKey(C, K_{10})$
  return C

Require: C is the ciphertext.
Require: K is all round keys.
Ensure: P is the plaintext.
  $P \leftarrow AddRoundKey(C, K_{10})$
  for $i \leftarrow 9$ **to** 1 **by** -1 **do**
    $P \leftarrow InvShiftRows(P)$
    $P \leftarrow InvSubBytes(P)$
    $P \leftarrow AddRoundKey(P, K_i)$
    $P \leftarrow InvMixColumns(P)$
  end for
  $P \leftarrow InvSubBytes(P)$
  $P \leftarrow InvShiftRows(P)$
  $P \leftarrow AddRoundKey(P, K_0)$
  return P

Algorithm 2 AES Key Schedule Algorithm

$K_0 \leftarrow Key$
$RCON \leftarrow 1$
for $i \leftarrow 0$ **to** 10 **by** 1 **do**
  $RoundMask \leftarrow RotWord(SubWord(K_i[127:96])) \oplus RCON$
  $K_{i+1}[31:0] \leftarrow K_i[31:0] \oplus RoundMask$
  $K_{i+1}[63:32] \leftarrow K_i[63:32] \oplus K_{i+1}[31:0]$
  $K_{i+1}[95:64] \leftarrow K_i[95:64] \oplus K_{i+1}[63:32]$
  $K_{i+1}[127:96] \leftarrow K_i[127:96] \oplus K_{i+1}[95:64]$
  $RCON \leftarrow RCON << 1$
  if $RCON \geq 256$ **then**
    $RCON \leftarrow RCON \oplus 283$
  end if
end for
return K

Digital design

To design an AES core it is necessary to first enforce some requirements on our implementation. The requirements are needed to guide us through the design procedure and also to verify if all the steps are correct during the implementation.
The first requirement is to fit the design into one of the lab's FPGAs, the Xilinx® Spartan-6 XC6SLX75-2CSG484C. This model has 11662 Slices, 132 DSPs, 172 18kB BRAMs and 408 user I/O, which should be enough to implement the algorithm in many different ways. As the rule of thumb, block ciphers implemented even in the most naive way would probably consume around one thousands Slices or less. Therefore, the FPGA of choice is not limiting in terms of area.
Second requirement is to have a simple and easy to understand architecture. This will not only help during this tutorial, but also during the future architecture decisions. When you are designing something for the first time, it is imposible to know how much resources the design will need. Therefore, the first version will be evaluated in terms of necessary resources i.e. area, latency, throughput and power. Later with those numbers, the designer will know if this meets the project requirements and will also know how to design better architectures. Just like software, digital design also requires a lot of trials and errors to understand and make progress.
The third requirement is to have a synchronous design. Synchronous designs have only one clock in the entire circuit, therefore all parts are synchronized with each other. This make the design task easier.
With the restrictions of our project as defined above the next step is to define an architecture.

The architecture

The architecture chosen adopts the strategy of combinational and sequential circuit, as shown in Figure 1.

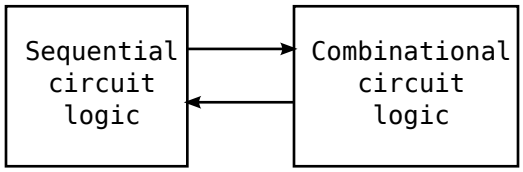


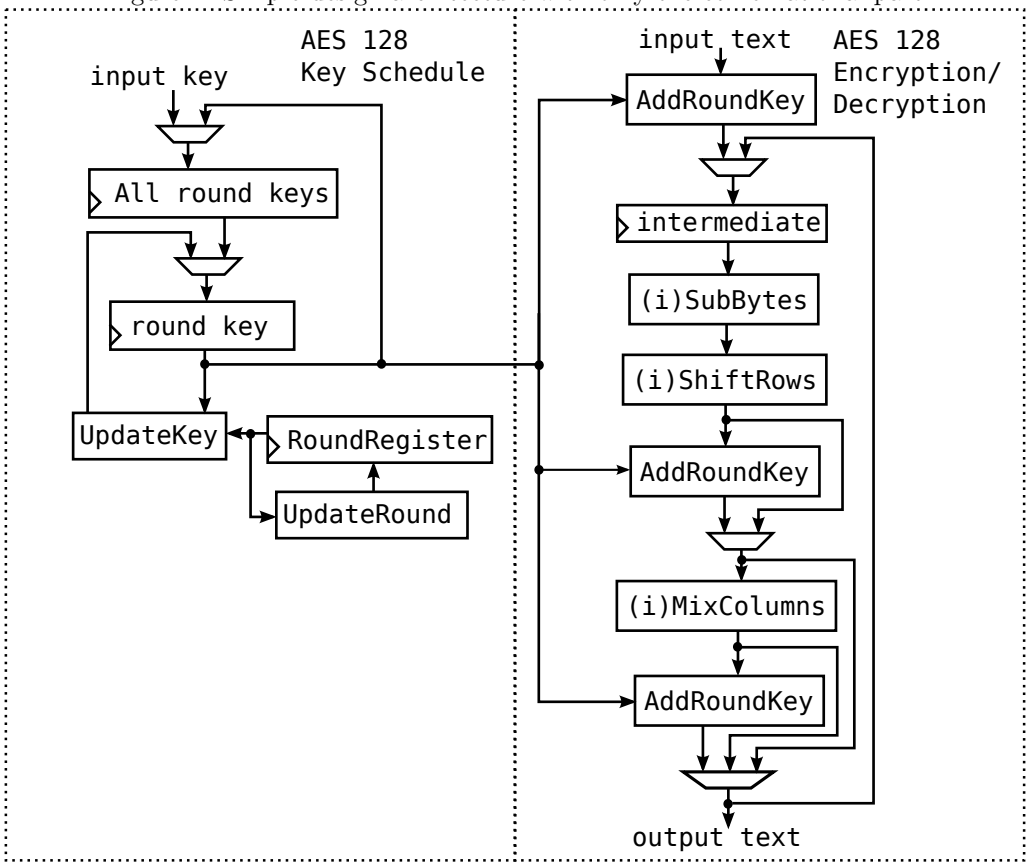
Figure 1: Digital design basic architecture

With this strategy some parts of the algorithm have to be done directly through the combinational circuit, while others are steps in the sequential one. For our case, since the algorithm already has internal rounds that are repeated, the strategy adopted is to perform each round and the inner and out rounds as a combinational circuit. Therefore, the circuit should be able to perform: one loop iteration, the steps before the loop and the steps after the loop. Those have to be done in the encryption and decryption cases. However, the key generation part is still missing, which will be done through a second combinational circuit that works together with the encryption one. With this strategy of splitting the operations, we came up with the architecture in Figure 2. Figure 2 does not have the sequential state machine that will control the circuit. It is nevertheless, possible to understand how the circuit works.

Key derivation process

In the beginning after all registers are clean, the user should feed the key into the circuit through input key and keep it stable, as shown in Figure 2. At the same time with the input key a signal will flag that is necessary to start the key derivation procedure. This signal is handled by the state machine that will control the entire process. In the first cycle it will write the key into the memory where all round keys are stored. Then the same key is fed into the round keys register, which will then be the input for update key circuit, together with the round constant. Then the circuit will write the key into the round key register and update

Figure 2: Simple design architecture with only the combinational part



the round constant in the next step. The new key will serve to generate the next key and will also be stored in the memory. As soon as the circuitry completes the key derivation process, the circuit will return to wait for other commands.

Encryption process

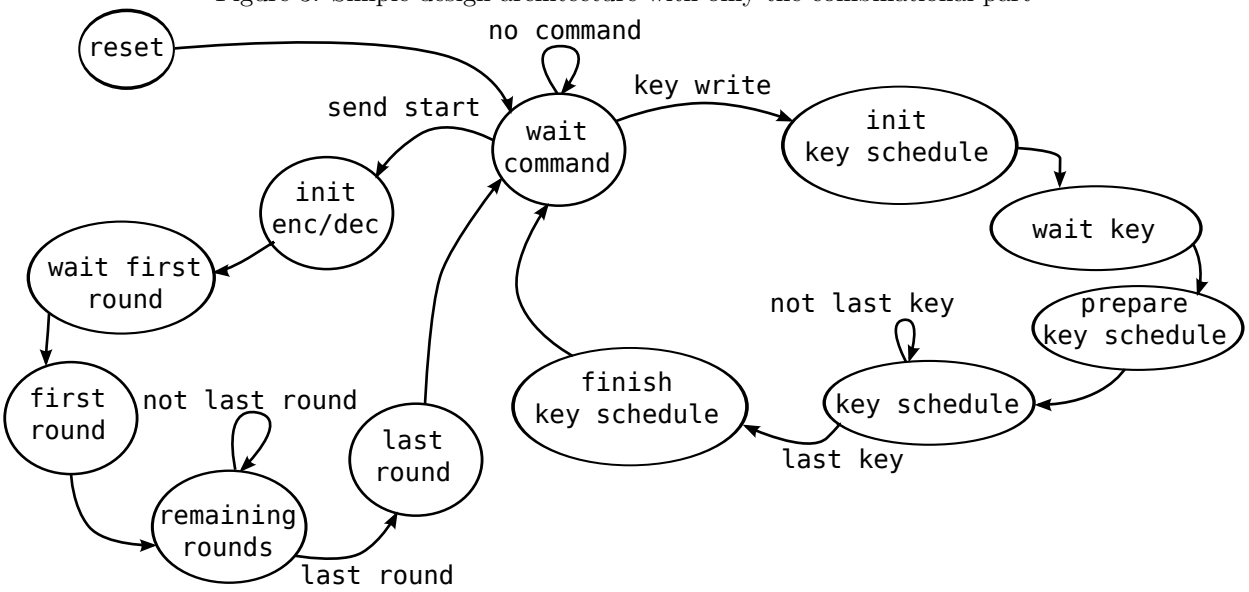
During the encryption/decryption step, the same as for the key derivation, the input is fed into the circuit together with a signal that will indicate the state machine to start. As the first step, the state machine has to make the first key available from the round key register. After the first key is loaded from the memory into the register, the first $AddRoundKey$ can be performed on the input and stored in the intermediate register. In the next cycles each round will be performed and stored in the intermediate register. When most rounds have been finished, the output will be available in the last round for a short period of time in the output text port. To know exactly the moment which the correct output is available, there is an extra signal to indicate when the output is valid.
In case of decryption the process is the same, the only difference is that some internal multiplexers are activated to change the order of the operations, and each function computes its inverse.

State machine

To better understand both encryption/decryption and key derivation process, Figure 3, shows the main state machine. The state machine behaves exactly as described in previous paragraph, where it has a reset stage and later it stays locked in wait state until it receives a new command. The only commands implemented are to generate new round keys and to encrypt/decrypt a block. While in the state of waiting for a command,

the state machine keeps a flag to indicate that is free, and while it is doing any computation this value goes low. To distinguish between finishing a key derivation from an encryption/decryption, there is also another flag controlled by the machine that only goes high when the output port has a valid ciphertext/plaintext. The extra amount of states is to wait for all parts of the circuits to be ready for the next step. In case of encryption/decryption it has to wait for the round number to reset, then with the first round number it is possible to obtain the first key in the memory and store in the round key register. After the round key register has the first key the intermediate register will finally receive the input block after the first $AddRoundKey$. This entire process takes the first 3 states during encryption/decryption process in the state machine. This situation also holds for the key initialization, which also has to reset the round number and then store the first key into the round keys memory. Finally, the first key is stored in the round key register to generate all subsequent keys.

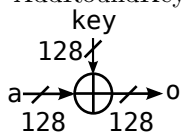
Figure 3: Simple design architecture with only the combinational part



Combinational small functions

Each function was done purely in a combinational manner for both, encryption and decryption mode. Therefore, each one has an input block, a signal to distinguish encryption/decryption mode and an output block except *AddRoundKey* function. This one behaves the same during encryption and decryption and also receives round key as an extra input. This is shown in Figure 4, where the operation is a simple XOR between inputs.

Figure 4: AddRoundKey function



The *SubBytes* function takes one input block and applies to all 16 bytes the S-Box function in case of encryption (or its inverse in case of decryption), as shown in Figures 5 and 6. The S-Box has 8 bits input and 8 bits output, so it can be described by a table of 256 entries. However by giving the table to the tool, the resulting circuit will not be optimized, therefore it will waste a lot of resources. In most hardware implementations in the literature, this function would be done in a very optimized way. However, to understand the circuit and have a working copy, the table approach is reasonable. More on this kind of alternative representations will be told later in the course (e.g. Satoh et al [2]).

Figure 5: SubBytes function and it is inverse

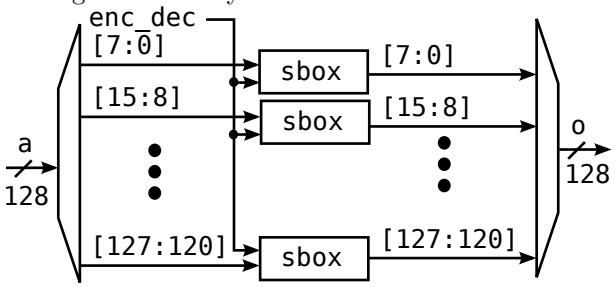
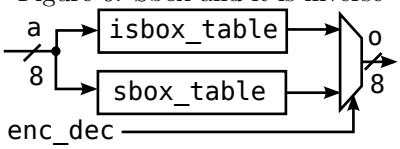


Figure 6: Sbox and it is inverse



The *ShiftRows* function is straightforward, as it just changes some bytes of place. Therefore Figure 7 shows only a mesh of wires that changes place. It should be noted that during encryption values are going to certain positions, and during decryption the values of those positions are going to the previous positions.
MixColumns functions is very complex that is why was split into Figures 9, 10, 11 and 12, and using the circuit from Figure 8. The circuit in Figure 8 multiplies an 8-bit finite field value by a constant x or 2 depending of the notation. This will be used later in the Figure 9 to compute all partial products for all block inputs. This partial product are necessary during the multiplication of a block column by a matrix, which is called *MixColumns*. The partial products of encryption and decryption are different, but since our circuit does both it has to able to compute them at the same time. Therefore, Figure 10 and 11 are operating at same time, and then in Figure 12 depending of the operation one or the other output is chosen.

Figure 7: Shiftrows and it is inverse

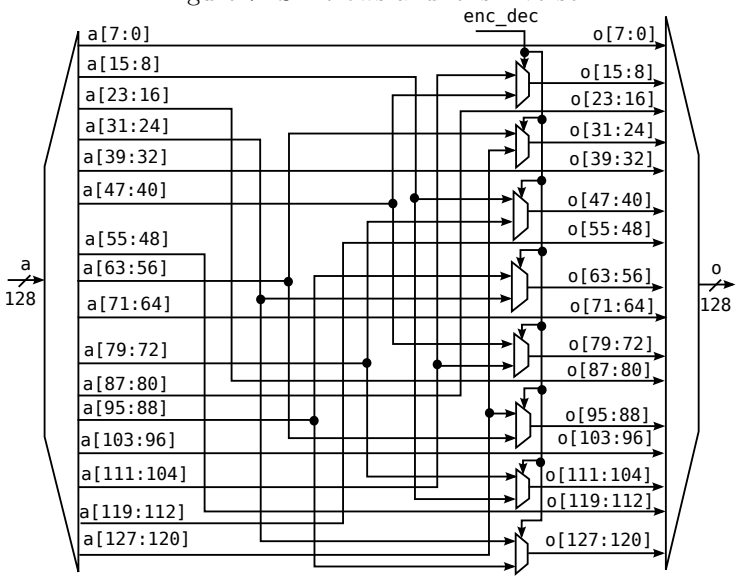
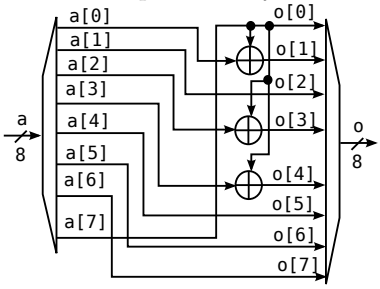


Figure 8: Multiplication by x in GF(2^8)



Finally the last function, $UpdateKey$ is the part which updates one key block into a new key block. Figure 13 shows part of the key passing through the AES S-box in the encryption mode, which then changes positions and one small part is added with a constant. Later all parts are XOR-ed with each other into a cascade manner.

In case we want to be able to generate the previous key, Figure 14 shows the modification in the circuit. As one can see, to generate the previous key, the S-box is used in the encryption mode, the difference is the order of the XOR operations.

Figure 9: MixColumns partial products

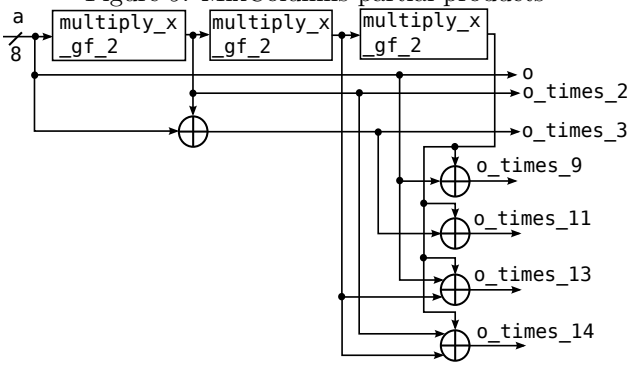


Figure 10: MixColumns encryption part

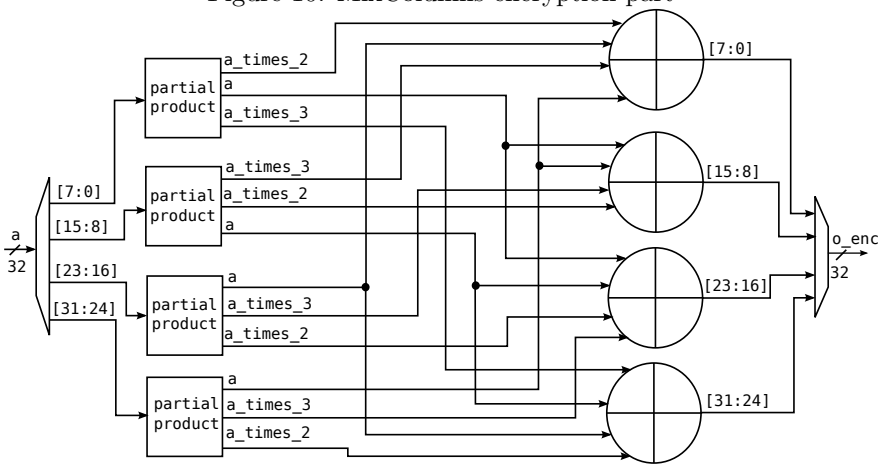


Figure 11: MixColumns decryption part

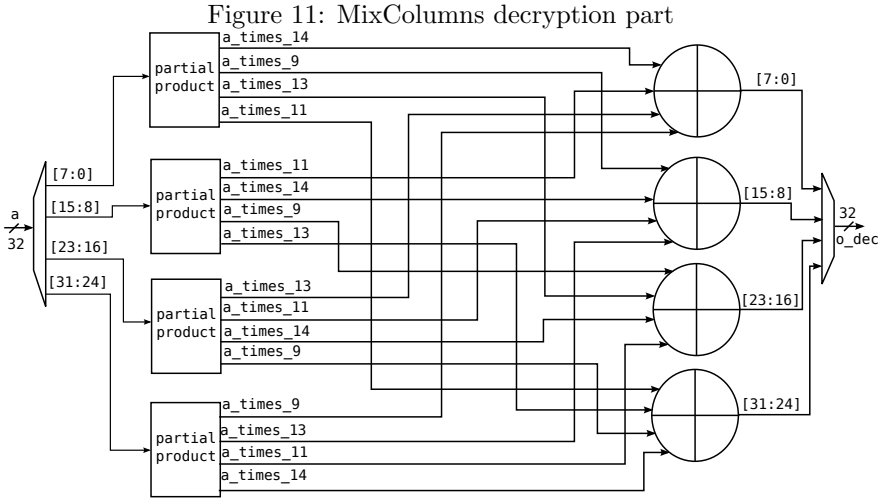


Figure 12: MixColumns function and it is inverse

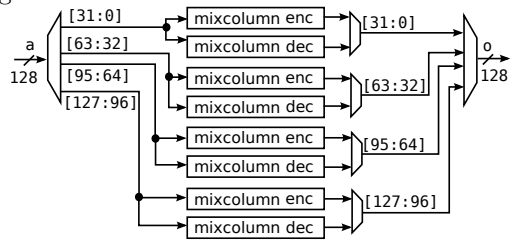


Figure 13: Circuit to update the AES key

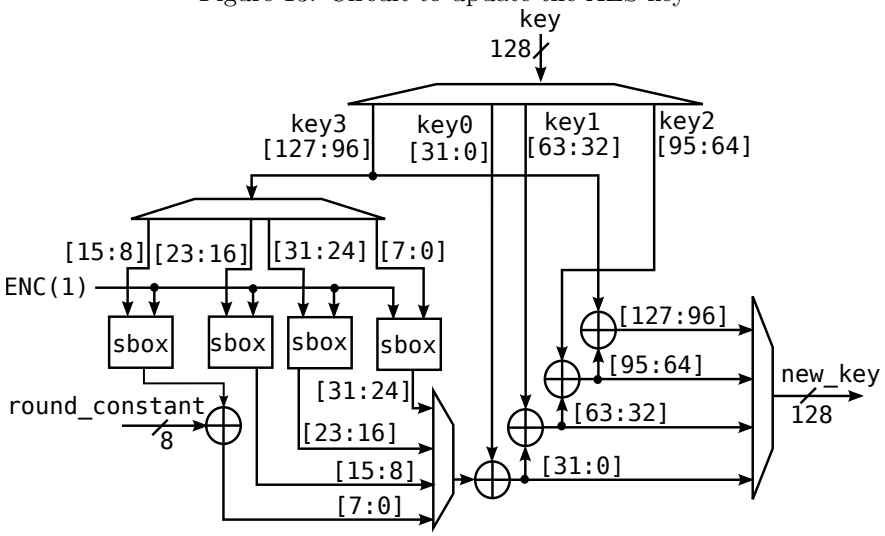
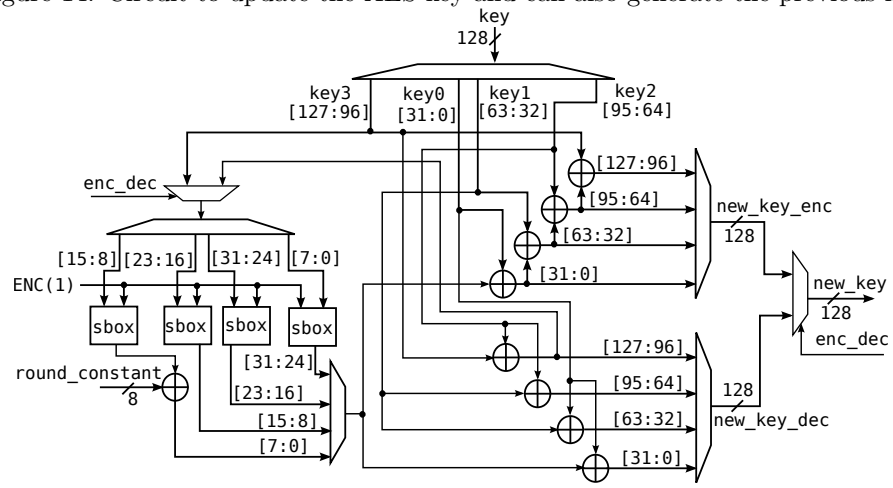


Figure 14: Circuit to update the AES key and can also generate the previous key



VHDL files and simulations testbenchs

All VHDL files needed for this exercise are listed below:

- aes_addroundkeys.vhd
  Perform the $AddRoundKey$ function
- aes_all_rounds_keys.vhd
  Memories which stores all round keys. This memory is addressable, so the round number is used to get the respective round key.
- aes_inverse_multiply_x_gf_2.vhd
  Compute the multiplication by x^{-1} GF(2^8) circuit. This does the inverse of the multiplication circuit. It can be used to generate the inverse of the round constants.
- aes_mixcolumn_multiplication.vhd
  $MixColumns$ operation to just one column.
- aes_mixcolumns.vhd
  $MixColumns$ operation to to all four columns in parallel.
- aes_multiply_x_gf_2.vhd
  Multiply a finite field value by x, sometimes written as '2'. The multiplication is in GF(2^8).
- aes_sbox.vhd
  Compute the AES S-Box operation for one byte.
- aes_shiftrows.vhd
  Entire $ShiftRows$ operation.
- aes_subbytes.vhd
  $SubBytes$ operation for the entire block.
- aes_update_key.vhd
  Compute the new key from the current key.
- aes128_core.vhd
  The AES128 core that is able to do encryption/decryption and generate all round keys.
- aes128_core_state_machine.vhd
  The respective state machine that controls the entire circuit.
- tb_aes_mixcolumns.vhd
  This is the testbench to test the functionality of the $MixColumns$ operation. These are the VHDL files that are tested to see if the device behaves as intended.
- tb_aes_shiftrows.vhd
  This is the testbench to test the functionality of the $ShiftRows$ operation. These are the VHDL files that are tested to see if the device behaves as intended.
- tb_aes128_core.vhd
  This is the testbench to test if the AES128 core works.
- Makefile
  This is a file that is used mainly for simulating the circuit in GHDL. There 3 possible command that you can instantiate:

- make test_core
  Compile and simulates tb_aes128_core.
- make test_mixcolumns
  Compile and simulates tb_aes_mixcolumns.
- make test_shiftrows
  Compile and simulates tb_aes_shiftrows.

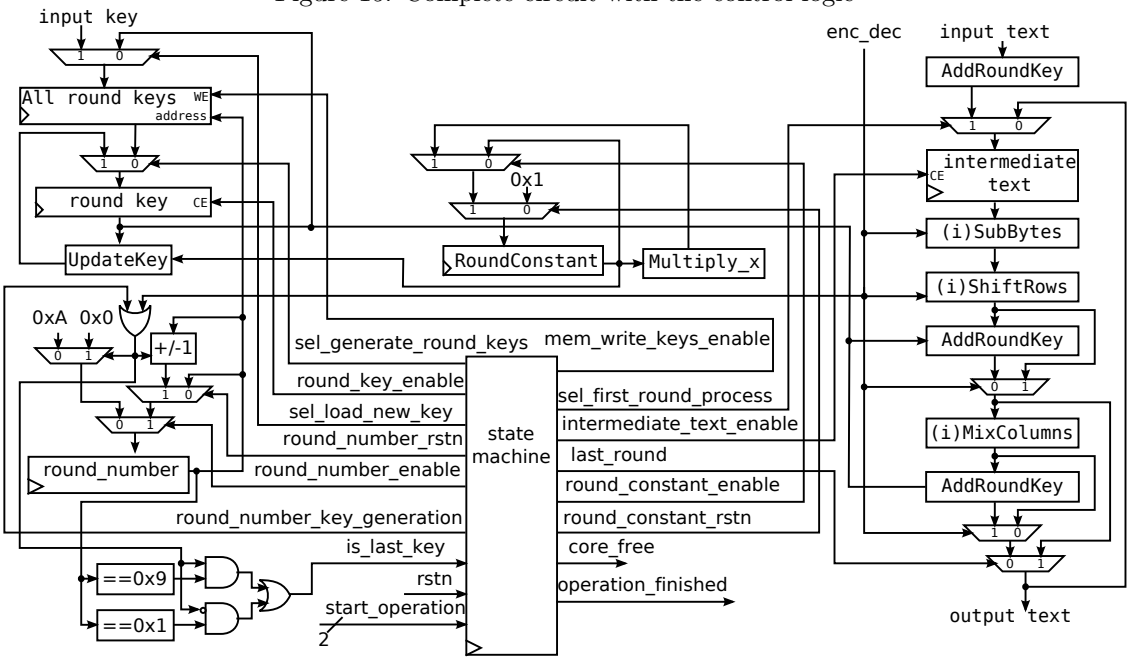
All commands will generate a file that ends with "*.ghw". This file can be opened in the program GTKWave https://sourceforge.net/projects/gtkwave/

Try to run these commands, generate the waveforms and verify how the circuit works and everything around you. In the GTKWave tool you just have to drag the signal that you want to see, and they become available. Running the simulations and seeing how the code works will help you understand the main ideas behind hardware design.

Exercise: Key generation optimization

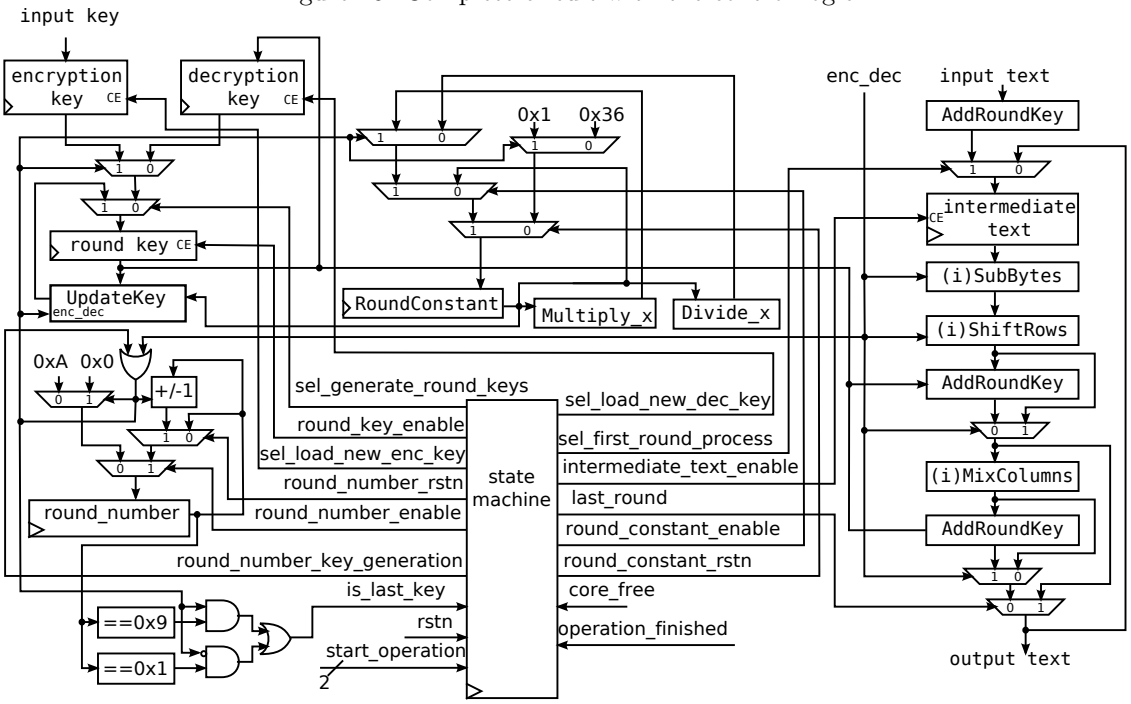
The current architecture with the control logic and all signals is shown in Figure 15. This version is more complete since it has the counter that holds the current round, the round constant generation and the logic to know when the last round happens. One important aspect to notice is that in some cases the encryption/decryption signal "enc_dec" is used together with a logic or with another signal. The reason for this is that the circuit, when receiving a key for key derivation, can have the port "enc_dec" in the decryption mode, and in this case we want our circuit to work in encryption mode.

Figure 15: Complete circuit with the control logic



You should change the current circuit into the one given in Figure 16. The state machine will keep the same amount of states and the configuration as in Figure 3, but the output signal levels have to change. This new circuit, instead of having a memory that holds all round keys, generates the keys on the fly from the first encryption key or the first decryption key. During the decryption mode, the key generation is done backwards, therefore the last key during decryption is in fact the first during encryption. To have decryption first key, the circuit generates all round keys from the encryption first key, but only the last one stays in the decryption register. The encryption first key, is already the input key itself.
Another change is in the round constants, because in the new circuit it has to be possible to generate them backwards. When initializing the circuit, it should start with 0x01 or 0x36, while before it was only 0x01. After initialized with one these options, the content of the register has to be multiplied or divided according to the mode it is running, in the first case it was only necessary to multiply. The division can be made by the "aes_inverse_multiply_x_gf_2.vhd".

Figure 16: Complete circuit with the control logic



The project deliverables are:

- The new code working without the aes_all_rounds_keys.vhd functionality and by doing the encryption/decryption with keys being generated on the fly.
- A one page explaining how it was the exercise. If it was too difficult? Too easy? What were the problems you faced? How did you solve it?

Tools that can be used

You can use these tools during the simulation process:

- Xilinx ISE
  `http://www.xilinx.com/products/design-tools/ise-design-suite.html`
- Model SIM Student Edition
  `https://www.mentor.com/company/higher_ed/modelsim-student-edition`
- GHDL (newest, above 0.33) + GTKWave
  `https://github.com/tgingold/ghdl/releases`
  `https://sourceforge.net/projects/gtkwave/`

For debugging and understanding the values between steps you can use https://www.nayuki.io/page/aes-cipher-internals-in-excel

Virtual Machine

There is also a virtual machine provided with the code that has installed GHDL and GTKWave. The machine has the 32 bits CentOS 5.11 Linux and it is configured to use 512MB of RAM, which is enough for our case. The machine can be used directly or it can also be accessed via SSH in port 22. To make it easier to access the virtual machine, it has been configured to forward port 2222 from the host PC to the virtual machine. You can see more on this on `https://www.virtualbox.org/manual/ch06.html#natforward`. With the port forwarding you can easily access it by: cryptoeng@127.0.0.1:2222 with any SSH program, for Windows user you can use Putty `http://www.chiark.greenend.org.uk/~sgtatham/putty/`.
Accounts on the machine are:

- User: cryptoeng
  Password : cryptoenghw
- User: root
  Password : cryptoenghw

The last user has administrative rights, but in case you need administrative permission with the user cryptoeng just use the command “sudo”.
For file transfers between your host machine and the virtual machine you can try the following:

- SFTP or SCP
  Just as you can use SSH to open a terminal, you can transfer files between host and virtual machine. For Windows users, I can recommend WinSCP `https://winscp.net/eng/download.php`, but other programs that support those protocols should work.
- E-Mail
  In this case you can open your e-mail account or any other website through the virtual machine Firefox and download the file.
- Virtual Box shared folders option
  This option will require to install the virtual box addition on CentOS, which has not be done, consider this option as last resort.

How to simulate and run

1. Open the terminal
2. Go to the folder /home/cryptoeng/aes_project/vhdl_source or just aes_project/vhdl_source in case you are already in the cryptoeng folder.
3. Enter the command "make test_core"
4. It should have compiled and simulated the aes128_core and written the simulation in "tb_aes128_core.ghw"
5. Enter the command "gtkwave tb_aes128_core.ghw"
6. GTKWave will open and show nothing, it is necessary to add the signals you want to see.
7. In the left corner there will be something called "top" with a plus sign if you click it will show tb_aes128_core which is the testbench. Below it will numerate some "Signals" you can drag those from the left to the right.
8. After dragging all of them you will see the screen change. Press the button with the magnifying glass and a square inside, that is your "Zoom to fit". After you press it you should have the vision of the entire simulation.
9. Click on tb_aes128_core to show more options, and keep and try to add all signals there as well, so you see more how is it behaving.
10. The more you click the more deeper you go into the design, to understand what is happening add the "actual_state" from the state machine.

Questions

You can contact me in P.Massolino@cs.ru.nl

References

[1] National Institute of Standards and Technology. FIPS PUB 197, Advanced Encryption Standard (AES), 2001. U.S.Department of Commerce/National Institute of Standards and Technology, `http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf`.

[2] Akashi Satoh, Sumio Morioka, Kohji Takano, and Seiji Munetoh. A compact rijndael hardware architecture with s-box optimization. In Colin Boyd, editor, *Advances in Cryptology — ASIACRYPT 2001: 7th International Conference on the Theory and Application of Cryptology and Information Security Gold Coast, Australia, December 9–13, 2001 Proceedings*, pages 239–254, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.