VLSI Front-End Design Example

Advanced Encryption Standard

Complementary Material for "Top-Down Digital VLSI Design"

Michael Muehlberghuber

November 17, 2014

Document Purpose. To better explain the process of VLSI front-end design, the textbook "Top-Down Digital VLSI Design" by Hubert Kaeslin (Morgan Kaufmann Publishers, 2014) is accompanied by a more substantial circuit example than what can be reproduced in a printed book. The AES cipher is used to demonstrate all design steps from initial specifications through to a gate-level netlist. The source materials, including synthesis code in both VHDL and SystemVerilog plus simulation- and synthesis-specific scripts are available for download from the book's companion website `http://store.elsevier.com/9780128007303`. The present note provides the necessary documentation.

Major Revisions

Revision	Date	Author(s)	Description
0.1	05/06/2014	mbgh	Created initial version
0.2	11/06/2014	mbgh	First full draft based on VHDL sources
0.3	21/06/2014	hk	Adaptations to textbook
0.4	22/10/2014	mbgh	Added SystemVerilog sources and UVM verification environment
0.5	14/11/2014	hk	First draft of the conclusion added
1.0	17/11/2014	mbgh	Finalized first release version

Contents

1 Introduction

Data security is an important field of application for Very Large Scale Integration (VLSI) circuits. One of the reasons for this is that various security-related products need implementing cryptographic algorithms that target specific requirements which can not be met with software running on a general purpose microcomputer. Such products range from low-resource applications such as smart cards or Radio-Frequency Identification (RFID) systems, to high-speed devices for which throughput is of utmost importance.

The present example deals with a block cipher, called Advanced Encryption Standard (AES), which accepts a plaintext and a cipherkey as inputs and computes the corresponding ciphertext. All input and output data to and from the algorithm are provided as a sequence of bits. Those bits may represent any kind of characters, numbers, or symbols in an arbitrary encoding, which is not further discussed here. Once the ciphertext is obtained from the AES block cipher, it can be transmitted over an insecure channel (for instance, the Internet) without compromising the confidentiality of the corresponding plaintext. Only a recipient who owns the same cipherkey will be able to decrypt the ciphertext. Such a setup is called a *symmetric encryption system* and an example,

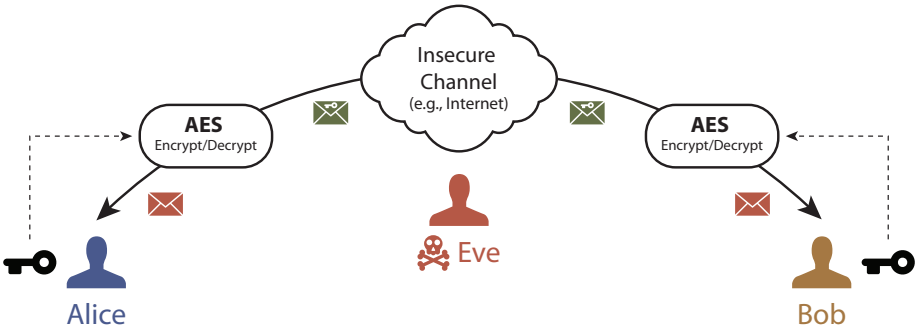

Figure 1: Symmetric encryption system

illustrating the communication between two parties named *Alice* and *Bob*, is shown in Figure 1. The third party, denoted by *Eve*, represents a potential attacker who has access to the insecure communication medium, but cannot derive any plaintext-related information from the transmitted ciphertext.

For simplicity, our example will only cover the encryption part of the block cipher. Moreover, the target application for our design will be in the high-throughput field. Detailed information regarding the design requirements can be found in Section 4.

2 Design Flow and Tools

A number of Electronic Design Automation (EDA) tools are required to turn a behavioral HDL description into a gate-level netlist. Both the tools and the design flow may vary depending on tool availability and on the requirements of your employer. Still, the overall development steps are always quite similar. The following list provides a summary of the tools and key options utilized here.

HDL Language: All design files of the AES block cipher are available as VHDL and SystemVerilog sources. The testbench used for functional verification is available in SystemVerilog exclusively.

Simulator: The HDL description of the AES design is verified using *Questa Sim 10.3a* by Mentor Graphics. Checking for functional correctness starts with directed verification where predetermined stimuli get applied to the model under test. The design is then fed with a large number of random input vectors.

Synthesizer: In order to synthesize the HDL code into a technology-specific gate level netlist, we utilize the 64 bit version of the *Design Compiler Version 2013.12* by Synopsys. The actual synthesis step is accomplished using Tool Command Language (Tcl) scripts, thereby getting reproducible results.

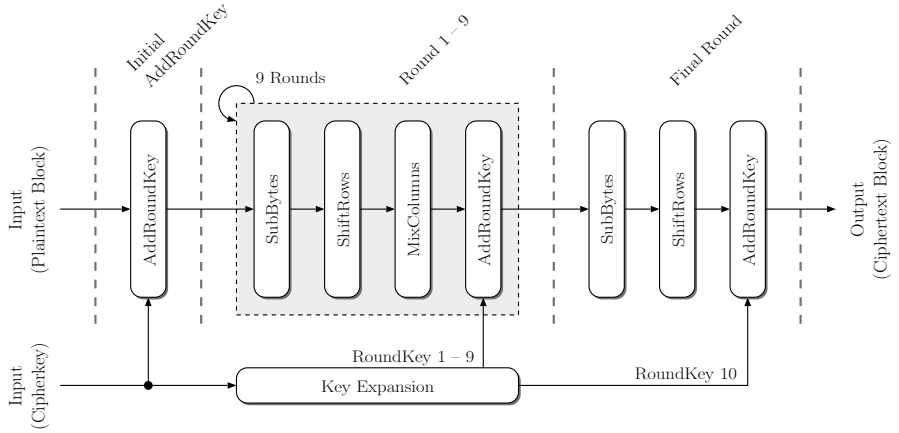

Figure 2: Overview of the AES-128 algorithm including key expansion

Target technology: The results presented here are based on synthesis runs for a mature 65 nm CMOS technology by United Microelectronics Corporation (UMC).

3 The Advanced Encryption Standard

AES is a well-established block cipher standardized by the National Institute of Standards and Technology (NIST) and other institutions [1]. The algorithm operates on data blocks of 128 bit and is available for three different cipherkey sizes, namely 128 bit, 192 bit, and 256 bit. Our example solely deals with the 128 bit version, hereafter referred to as AES-128. More details on the other versions as well as an in-depth explanation of the algorithm can be obtained from [1], for instance.

3.1 Cipher Operations

In general, AES-128 comprises ten identical rounds, each operating on the 128 bit internal state S, that can be represented as a 4×4 matrix of bytes. $S_{i,j}$ denotes the byte of row i and column j with $i, j \in \{0, \ldots, 3\}$. The state matrix gets initialized with the input data (plaintext block). Each cipher round is made up of four different transformations, called *SubBytes*, *ShiftRows*, *MixColumns*, and *AddRoundKey*. The only difference between the ten rounds are the so-called *roundkeys* being used for the AddRoundKey transformation, see Figure 2. Moreover, the MixColumns transformation is omitted in the final round. Prior to the ten rounds, there exists an initial AddRoundKey transformation. The four transforms are defined as follows:

SubBytes: SubBytes performs a byte-wise substitution of the state using a substitution box (S-box). For the actual values of the S-box, we refer the reader to [1].

ShiftRows: Each byte of a row of the state is cyclically shifted to the left by the index of the row (zero-based). Therefore, the first row does not change, the bytes of the second row are rotated one byte to the left, and so on.

MixColumns: This operation can be understood as a column-by-column multiplication modulo $x^4 + 1$ in the finite field GF(2^8), looking at the columns of the state as polynomials of the Galois field. Once again, we refer to the official standard [1] for a detailed description of this transformation.

AddRoundKey: The AddRoundKey transformation is a bitwise XOR operation of the state bits and the bits of the current roundkey.

3.2 Key Expansion

Since AES-128 is made up of ten cipher rounds and an initial AddRoundKey transformation, it requires 10 roundkeys in addition to the original cipherkey provided from externally. All 128-bit wide roundkeys are derived from the main cipherkey using a *Key Expansion* function.

Expanding the cipherkey to the roundkeys works in a column-by-column approach, where the first four columns represent the original cipherkey. Each of the first four columns contains four bytes of the cipherkey and the remaining columns get calculated as shown in Figure 3. The *RotWord* and the *SubWord* operations of the key expansion are defined as follows:

RotWord: Takes one column and performs a circular shift (rotation) of the bytes such that $[k_i, k_{i+1}, k_{i+2}, k_{i+3}]$ becomes $[k_{i+1}, k_{i+2}, k_{i+3}, k_i]$.

SubWord: All four bytes of a column get substituted using the same S-box as utilized throughout the cipher rounds.

The round constants used for the key expansion differ from one round to the next. They can be obtained from [1].

4 Design Requirements

Our aim is to maximize throughput with 5 Gbit/s being a minimum target. In addition, we want to limit our circuit to a maximum of 100 kGE in order to keep die size and manufacturing costs down.[1] To save on overall pin count and package costs, data would very likely get multiplexed over the available I/O pins in a real product. For simplicity, we are not going to burden our example with that. Instead, we assume enough I/Os pins

[1] One GE has the size of a two-input NAND gate. In our 65 nm target technology, 1 GE = 1.44 µm^2.

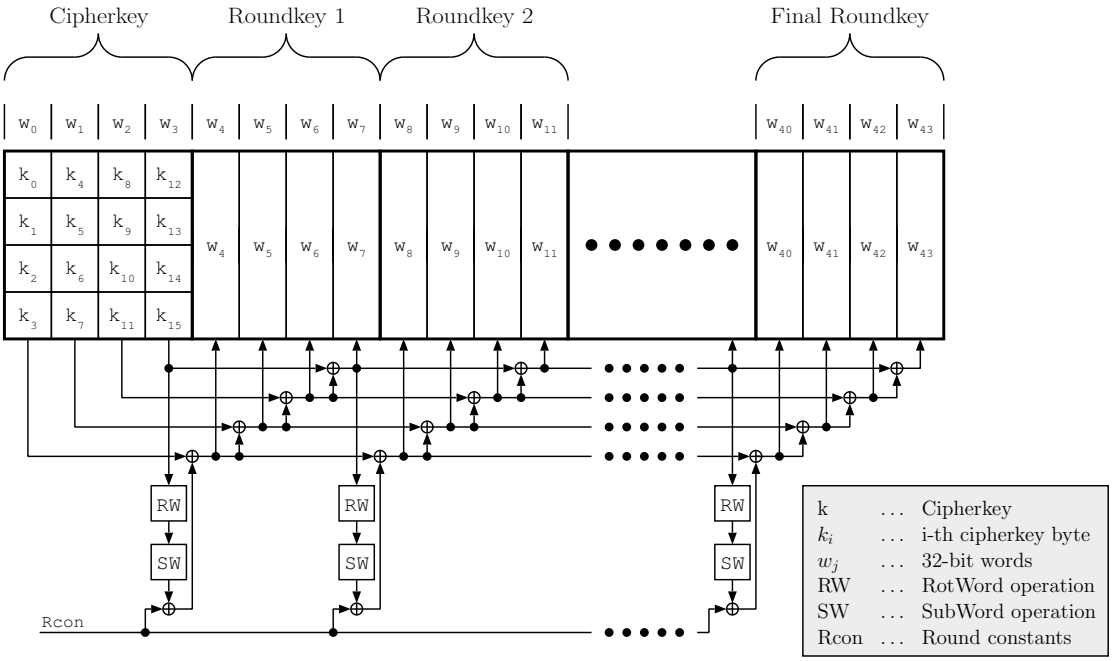

Figure 3: AES-128 key expansion

are available to provide all input data and to retrieve the outputs.[2] Table 2 summarizes the design requirements for our AES-128 architecture.

Table 2: Summary of design requirements

Design Property	**Requirement**
Achievable throughput	maximize, 5 Gbit/s minimum
Available chip area	100 kGE
Chip package	no restrictions on pin count

5 Architecture Design

After sketching, analyzing, and rethinking a number of drafts, we came up with an architecture for AES-128 that relies on pipelining to boost throughput. We decided to store all roundkeys and all intermediate states (between any two consecutive rounds) in registers (denoted by the thick black vertical lines in Figure 4). From the architecture sketch, it is evident that the longest path gets determined either by a single cipher round

[2]This implies that a suitable package should provide at least 384 pins for accepting plaintext and cipherkey and for outputting the ciphertext. Add to that the necessary pins for power/ground, clocking, initialization, and testing.

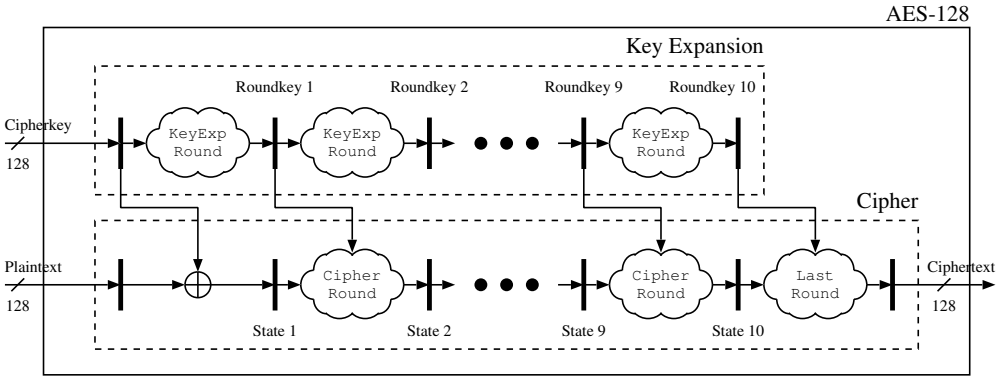

Figure 4: First sketch of the AES-128 circuit architecture

(*Cipher Round*) or by one key expansion round (*KeyExp Round*), whichever is longer. As mentioned in Section 3, each AES cipher round consists of the transformations SubBytes, ShiftRows, MixColumns, and AddRoundKey, whereas one key expansion round is solely made up of the logic required to derive a roundkey from the previous one. Figure 3 reveals that these are just a few XOR operations, the RotWord operation, and the SubWord operation. Hence, the cipher round turns out to be more critical.

With the proposed architecture, it is possible to process 128 bits of data within 12 clock cycles in the worst case. This applies when a new cipherkey is provided to the circuit or when plaintext data is applied for the first time to the inputs of the AES-128 architecture. In the best case, that is once the pipeline of the cipher is full and no key change is required, the architecture provides 128 bits of ciphertext per clock cycle. Using this information, we can determine the longest admissible delay t_{lp} that just satisfies our throughput requirement of $\Theta \geq 5$ Gbit/s:

$$t_{lp} = \frac{128\,\text{bit}}{12\,\text{cycle} \cdot 5\,\text{Gbit/s}} = 2.13\,\text{ns} \tag{1}$$

Thus, our design has to run with a maximum frequency of roughly 500 MHz to yield the desired throughput. Taking the target technology and the operations into account, which most likely will constitute the critical path, this seems to be feasible. Once the pipeline of the cipher is filled with data and the cipherkey does not need to be changed, the maximum achievable throughput (Θ_{500}) with this frequency (500 MHz) results in:

$$\Theta_{500} = \frac{128\,\text{bit} \cdot 500\,\text{MHz}}{1\,\text{cycle}} = 64\,\text{Gbit/s} \tag{2}$$

Similar estimates should also be done for the area requirements of a design. Although it is usually a difficult task to estimate the exact footprint of a final design for a certain technology, it helps to start by analyzing smaller parts of a design and then extrapolating

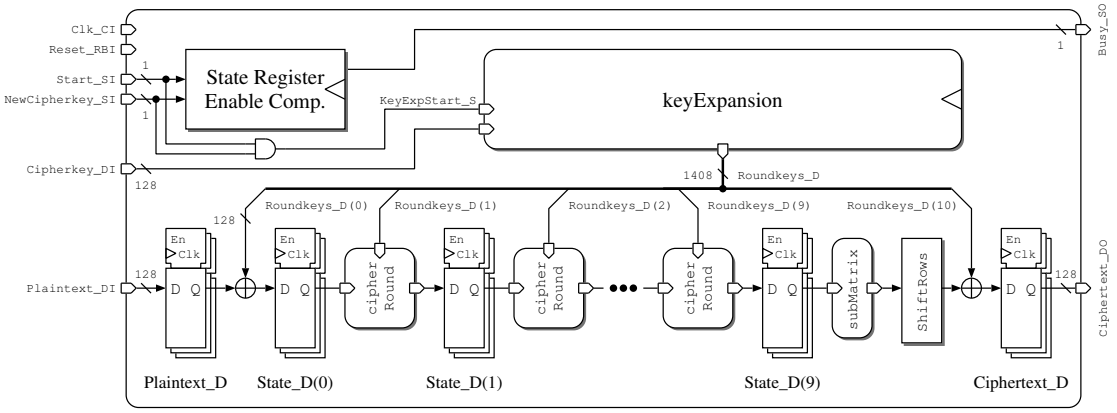

Figure 5: Top-level block diagram of the AES-128 design

those results to the overall architecture. The numbers you obtain from such estimations may still diverge by a small factor from your final results, but when done correctly, they provide a valuable guidance. Thereby, you avoid spending unnecessary development time for designs which will never meet your initial requirements.

5.1 Block Diagram

Figure 5 shows the top-level block diagram of the final AES-128 design. Its major building blocks are the cipher rounds, the state registers or pipeline registers that store the intermediate cipher states, the key expansion block, and a small block responsible for computing the enable signals for the cipher state registers. This enable mechanism is basically a simple one-hot encoded shift register. Each time a new plaintext block has to be encrypted (indicated by the `Start_SI` signal), a `1` gets shifted into this register and is passed on "in parallel" with the plaintext being processed through the pipeline registers of the cipher. A similar approach has been implemented in order to create the enable signals for the roundkey-storing registers within the *keyExpansion* entity.

Since we decided to add registers for both the input and the output data of our AES-128 design, the overall processing time for a plaintext block, after a new cipherkey has been assigned, sums up to 12 cycles. In such a case, the newly derived roundkeys will propagate through the pipeline registers of the key expansion one after another within each clock cycle and will be assigned to their corresponding cipher rounds accordingly. Therefore, during the change of a key, the pipeline of the key expansion contains roundkeys from two different cipherkeys.

We did not implement a standardized I/O interface (this might be part of a follow-up exercise), but solely used a single bit (`Start_SI`) to determine whether the applied data are valid or not. The pinlist of the architecture is described in Table 3.

Table 3: Pinlist of the top-level block diagram

Port Name	**Type**	**Size**	**Description**
`Clk_CI`	Input	1	Input clock.
`Reset_RBI`	Input	1	Asynchronous, active-low reset.
`Start_SI`	Input	1	Indicates that the assigned inputs are valid and starts the encryption process.
`NewCipherkey_SI`	Input	1	Indicates that the cipherkey assigned at the input is a new one. Therefore, the roundkeys have to be derived when starting the encryption.
`Plaintext_DI`	Input	128	The plaintext block, which should be encrypted.
`Cipherkey_DI`	Input	128	The cipherkey, which should be used for encryption.
`Busy_SO`	Output	1	Indicates whether the design is currently processing data or not.
`Ciphertext_DO`	Output	128	The resulting ciphertext block, i.e., the encrypted plaintext block.

6 Simulation

To ensure functional correctness of our AES-128 design, we first developed a software model in C++ that was used for debugging purposes of the HDL design and to create test vectors for the architecture. We provide two different sets of test vectors which are located in the `./sim/tv/` directory of the source code archive and are described in Table 4.

Table 4: Description of the provided test vectors

File	**Description**
`aes128_fips197.tv`	A test vector file containing the stimuli and expected responses as provided in appendix C.1 of the NIST standard [1].
`aes128_random.tv`	Different test vector sets, each containing a random plaintext block, a random cipherkey, and the corresponding ciphertext.

All lines within the test vector files that start with a % are comments and will be ignored by our testbench environment. Each remaining line contains a full test vector set for a single AES-128 run, including plaintext/cipherkey stimuli and the corresponding expected ciphertext responses. All values are provided in hexadecimal format. Figure 6 shows a description of a sample test vector.

Once the test vector files have been generated, they are read by a SystemVerilog testbench environment, which is briefly described in the following.

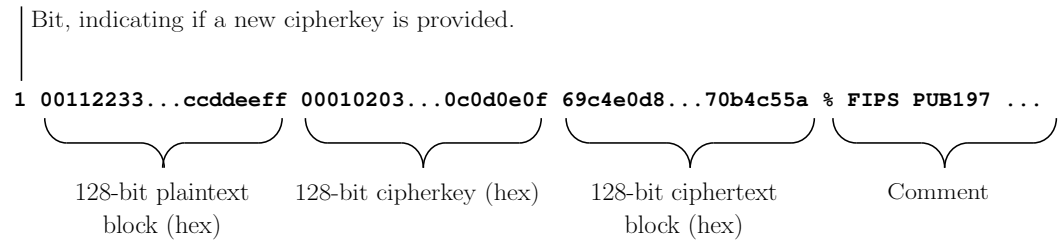

Figure 6: Test vector example (including stimuli and expected responses)

6.1 The Verification Environment

We provide a testbench (verification) environment, which has been developed using SystemVerilog. More specifically, we used the Universal Verification Methodology (UVM) to realize the test environment. UVM represents a "standardized" methodology on how to implement verification environments using SystemVerilog. Figure 7 provides an overview of the verification environment used to test the functional correctness of the AES-128 design. The environment follows a typical UVM setup[3] and works as follows:

- In order to make both the VHDL and the SystemVerilog version of our AES-128 design accessible by the SystemVerilog verification environment, we created a wrapper module which uses a SystemVerilog interface to connect to the Design Under Verification (DUV). Thereby both the VHDL and the SystemVerilog design can be handled equivalently by the test environment.
- Communication between the DUV and the verification environment is established using the SystemVerilog interface `duv_ifc`.
- Only a single test has been developed, which initiates a sequence that reads the stimuli vectors from the provided file, translates them into UVM transactions, and passes the transactions on to the `driver`. The driver then converts the transactions into actual "pin wiggles" and applies them to the DUV.
- The `monitor` on the other hand is responsible for observing the data running to and coming from the DUV. Note that it is only a passive unit and never applies any data to the DUV. Once the monitor observes data transfer to or from the AES-128 design, it provides this information to the `predictor` and the `comparator`.
- Once the predictor receives data from the monitor that has been observed when sending it to the DUV by the driver, it calculates the expected responses for the respective input data. Since in our design the expected responses are available in the test vector file, it solely reads it from the file and passes it on to the comparator.

[3]Readers not familiar with the UVM design paradigm may require to do some further research in order to fully understand the provided environment, which is out of scope of this documentation.

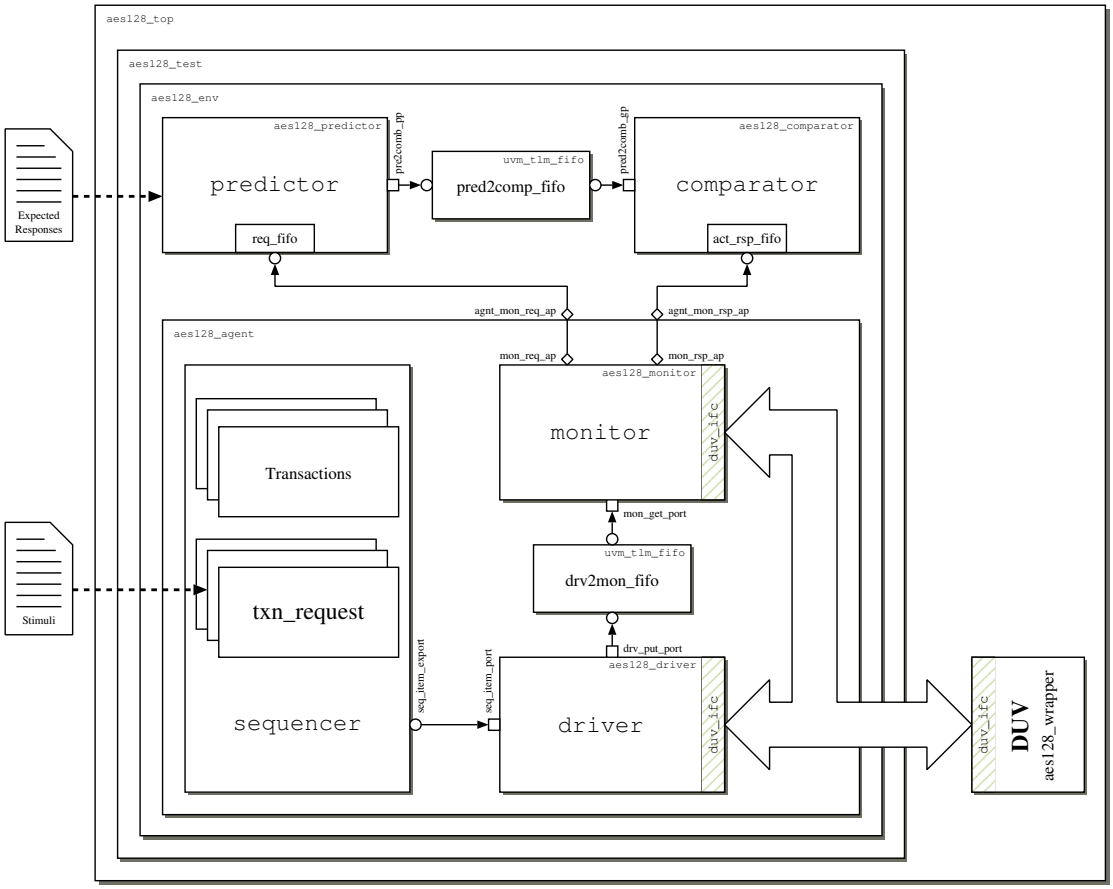

Figure 7: Testbench structure for functional verification

- As soon as the comparator receives the actual responses, observed by the monitor, it compares them against the expected ones and prints some status reports.

Since our verification environment follows common UVM design principles, it can easily be adapted or extended (e.g., add some further tests, change the interface to the DUV).

In order to run the simulation, we use two different shell scripts to first compile all of the required HDL source files (depending on whether you want to simulate the VHDL or the SystemVerilog version of the AES-128 design):

```
> ./sim/scripts/rtl-vhdl-cmp.sh
```

or

```
> ./sim/scripts/rtl-sv-cmp.sh
```

Once the design has been compiled successfully, the GUI-bases simulation can be initiated using another provided shell script:

```
> ./sim/scripts/rtl-sim_ui.sh
```

7 Synthesis[4]

As soon as the functional verification has been completed successfully, the Design Under Test (DUT) can be synthesized for a certain target technology and standard cell library. We used the script located at `./syn/scripts/aes128-synth.tcl` in order to synthesize our architecture. The provided script has been written in Tcl[5] and can be executed within the Synopsys Design Compiler as follows:

```
dc_shell> source ./scripts/aes128-synth.tcl
```

Besides constraining the AES-128 design to a clock period of 2 ns and actually synthesizing the architecture for the target technology, it writes a couple of reports. These reports provide information about the required area (`area.rpt`) as well as some timing information ($\texttt{timing}_{\texttt{xx}}\texttt{.rpt}$). Since the only constraint we have set for our synthesis run is the clock period, the $\texttt{timing}_{\texttt{ss}}\texttt{.rpt}$ is of interest for us. In that file, you should find a line similar to the following:

```
------------------------------------------------------------------------------
  slack (MET)                                                       0.5348
```

The positive slack indicates that the timing constraint for the clock period (2 ns) of our circuit was met by the synthesis tool. Investigating the area report (`area.rpt`) of the synthesis run reveals that the current AES-128 architecture requires approximately 153 kGE, which exceeds our initial area budget of 100 kGE significantly. Therefore, appropriate measures must be taken to meet our goals.

8 Optimizing the Architecture [6]

To reduce circuit complexity to below the acceptable bound of 100 kGE, a couple of potential solutions come into mind:

Relaxing constraints: An easy approach is to relax overly strict timing requirements, thereby allowing the synthesizer to use weaker output drivers for the standard cells being used and hence, reducing the area of the resulting architecture. As we have seen in Section 7, a slack of about 0.5 ns exists in the critical path of our design. Hence, relaxing the clock constraint will definitely not save a lot of area (if any at all). Actually the opposite is the case, which means that a somewhat stricter clock constraint will most likely not increase the overall area significantly.

[4] Note that the results presented in this section are based on the VHDL sources of the AES-128 design. Nevertheless, we verified that using the SystemVerilog sources, we obtain roughly the same results.
[5] Tcl is a programming language, which represents a de-facto standard in many EDA tools.
[6] We do not provide the respective simulation and synthesis scripts for this section. Those should be created by interested readers themselves as part of an exercise.

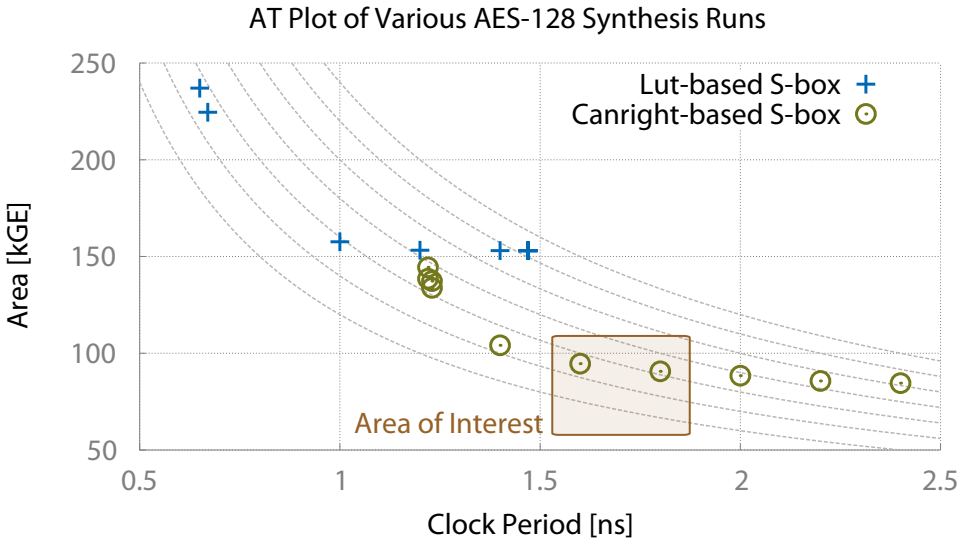

Figure 8: AT plot of the AES-128 design using different S-boxes (synthesis results)

Architecture redesign: A more promising idea to minimize area occupation is to re-evaluate the overall architecture of the AES-128 core. Instead of providing a separate circuit block for each cipher round, iterative decomposition could be applied to cut down on overall circuit size. Such a choice implies major design changes and could significantly delay tapeout, however. Before backing up this far in the design cycle, let us find out how costly the various circuit functions are in terms of logic gates and area occupation.

Detailed analysis of the synthesis results reveals that between 70% and 80% of the overall area is required by the S-boxes alone, that make up the *SubBytes* transformation described in Section 3. This is not surprising as our architecture requires 160 instances of the S-box (16 for each cipher round in order to substitute the 128 bit state) just for the cipher part of the AES-128 design. 40 more instances of the S-box are needed in the key expansion as part of the *SubWord* operations (cf. Figure 3). This sums up to 200 S-boxes in the overall design that together occupy roughly 115 kGE (= 75% of 153 kGE) in our architecture. From this background, we can much better direct our efforts.

Devising a more economic S-box: It is now utterly clear that more efficient S-boxes are highly beneficial. We can argue that the overall design should fit into the 100 kGE envelope if we were able to cut the area requirement of an S-box to between one third and one half of its present value. This appears reasonable as the original implementation of the S-box, which can be found in the file `./src/vhdl/sbox.vhd`, is fairly unsophisticated in that it just defined an array of constants. The task of coming up with an efficient circuit was simply passed over to logic synthesis, with no clues from the specific context.

A more sophisticated and highly area-saving approach is due to D. Canright [2]. The key idea is to implement the $GF(2^8)$ inversion present in the S-box by recurring to smaller subfields[7]. An implementation of the S-box based on this approach can be found in `./src/vhdl/sboxCan.vhd`. Figure 8 provides an AT plot of the AES-128 design once using the original, constants-based S-box approach and once using the Canright S-box implementation. As we can see from the plot, for maximum frequencies of 625 MHz and below, the AES-128 design should fit into the 100 kGE area requirement when using the Canright S-box.

Although the Canright-based S-box suffers from a significant size inflation for clock periods below 1.25 ns or so, meeting our initial goal of 2 ns is easily feasible. Thus, we got a fully functional design that meets our initial goals without redesigning the overall architecture. Adaptations to just one circuit block afforded crucial improvements of the overall architecture, which was only possible since we knew where most of the area was spent. For an actual tape-out of the design, we would most-likely build upon one of the synthesis results lying in the *Area of Interest* shown in Figure 8.

9 Conclusion

Over the past decade, VLSI synthesis and verification technology have evolved to a point where it takes quite substantial examples to explain all their facets. This worked-out example expands upon the materials in the textbook by illustrating various concepts and techniques presented there in a realistic context and on a more substantial circuit. More specifically, we demonstrate:

- Functionally identical RTL synthesis models in both VHDL and SystemVerilog
- Architectural tradeoffs and a small selection of optimization techniques
- Timing constraints and their impact on the synthesis outcome (AT-plot)
- A self-checking testbench that applies directed and random test suites
- Organizing a verification environment into re-usable modules using UVM
- Co-simulating a VHDL model with a SystemVerilog testbench
- Shell and Tcl scripts for governing simulation and synthesis runs

The authors believe that this document and the code files that come with it, along with the background information from the textbook will greatly help newcomers get started with real projects. Best wishes for success!

[7] A more detailed discussion is beyond the scope of this exercise. We refer the interested reader to [2–5] for more information on this topic.

10 Source File Archive Structure

In the following we provide a brief description of the provided archive containing all the resources for the presented AES-128 design. All scripts have been created for the design flow described in Section 2 and therefore, may have to be adapted when the tools being used differ from the ones utilized herein.

```
/
├── sim ..................................... Simulation-specific files
│   ├── scripts ............................. Simulation scripts
│   │   ├── rtl-sim_ui.sh ................... Shell script to start Questa Sim
│   │   ├── rtl-sv-cmp.sh ................... Shell script to compile SystemVerilog sources
│   │   ├── rtl-vhdl-cmp.sh ................. Shell script to compile VHDL sources
│   │   └── rtl_run.tcl ..................... Tcl script to run the simulation
│   ├── tv .................................. Test vectors files
│   │   ├── aes128_fips197.tv ............... Single test vector
│   │   └── aes128_random.tv ................ 1000 random test vectors
│   └── waves ............................... Wave files for simulation
│       └── wave-rtl.do ..................... Simple wave file to show I/Os
├── src ..................................... HDL source files
│   ├── sv .................. SystemVerilog source code files for the AES-128 design
│   ├── tb .......... SystemVerilog source code files for the verification environment
│   └── vhdl ........................... VHDL source code for the AES-128 design
└── syn ..................................... Synthesis-specific files
    └── scripts ............................. Synthesis scripts
        └── aes128-synth.tcl ............... Tcl script to run synthesis in Synopsys
```

List of Acronyms

AES . . . . . Advanced Encryption Standard

CMOS . . . . . Complementary Metal Oxide Semiconductor

DUT . . . . . Design Under Test

DUV . . . . . Design Under Verification

EDA . . . . . Electronic Design Automation

FSM . . . . . Finite State Machine

NIST . . . . . National Institute of Standards and Technology

RFID . . . . . .Radio-Frequency Identification

Tcl . . . . . .Tool Command Language

UMC . . . . . .United Microelectronics Corporation

UVM . . . . . .Universal Verification Methodology

VLSI . . . . . .Very Large Scale Integration

References

[1] NIST, *Advanced Encryption Standard (AES) (FIPS PUB 197)*, National Institute of Standards and Technology, Nov. 2001.

[2] D. Canright, "A Very Compact S-Box for AES," in *Cryptographic Hardware and Embedded Systems – CHES 2005*, ser. Lecture Notes in Computer Science, J. Rao and B. Sunar, Eds. Springer Berlin Heidelberg, 2005, vol. 3659, pp. 441–455. [Online]. Available: http://dx.doi.org/10.1007/11545262_32

[3] V. Rijmen, "Efficient Implementation of the Rijndael S-box," *Katholieke Universiteit Leuven, Dept. ESAT. Belgium*, 2000. [Online]. Available: http://dm.ing.unibs.it/~giuzzi/corsi/Support/papers-cryptography/rijndael-sbox.pdf

[4] A. Satoh, S. Morioka, K. Takano, and S. Munetoh, "A Compact Rijndael Hardware Architecture with S-Box Optimization," in *Advances in Cryptology – ASIACRYPT 2001*, ser. Lecture Notes in Computer Science, C. Boyd, Ed. Springer Berlin Heidelberg, 2001, vol. 2248, pp. 239–254. [Online]. Available: http://dx.doi.org/10.1007/3-540-45682-1_15

[5] J. Wolkerstorfer, E. Oswald, and M. Lamberger, "An ASIC Implementation of the AES SBoxes," in *Topics in Cryptology – CT-RSA 2002*, ser. Lecture Notes in Computer Science, B. Preneel, Ed. Springer Berlin Heidelberg, 2002, vol. 2271, pp. 67–78. [Online]. Available: http://dx.doi.org/10.1007/3-540-45760-7_6