Implementation of the Advanced Encryption Standard (AES) on the ATMega644V Microcontroller – Draft

F. De Santis
Sichere Implementierung kryptographischer Verfahren WS 2015-2016
Technische Universität München **October 21, 2015**

Contents

1 The ATMega644V Microcontroller

The ATMega644V is an 8-bit microcontroller based on the AVR architecture (modified Harvard architecture) operating at a maximum frequency of 20 MHz. It features 64 kBytes on-chip non-volatile flash memory where the program code instructions and constant data are stored, 4 kBytes of internal data SRAM and 32×8-bit general purpose working registers to handle variables and arithmetic operations. The ATMega644V additionally features a 2 kBytes EEPROM for data storage, 64 I/O registers and 160 extended I/O registers for interfacing with the outside world. The ATMega644V has a two stage single level pipeline design and executes most of the instruction in one clock cycle. Notable exceptions are the load, store and the branching instructions which take two cycles to execute. Please refer to the datasheet for more details, cf. `http://www.atmel.com/Images/2593s.pdf`.

2 Assignment

The goal of the assignment is to provide an *efficient* implementation of the Advanced Encryption Standard algorithm for the ATMega644V microcontroller and write a report of the design choices thereof. Possible targets for optimization are: minimum `RAM` usage, maximum throughput and minimum code size.

2.1 Installing the Toolchain

This section describes how to install the toolchain required to complete the assignment on Ubuntu Linux distributions. The toolchain provides all the necessary tools to compile and evaluate the AES implementation. In case an Ubuntu Linux distribution is not available, an off-the-shelf VirtualBox machine[1] with a pre-installed version of the toolchain can be downloaded from the website `https://tueisec-sica.sec.ei.tum.de/`. Please note that the virtual machine must be imported into VirtualBox (please refer to `https://www.virtualbox.org/` for more information). Further instructions from this section can be skipped in case the virtual machine is used. Also, please note that a registration to the lecture over the portal `https://campus.tum.de` is mandatory to get access to the website `https://tueisec-sica.sec.ei.tum.de/` and access is granted only from the TUM intranet or over VPN. Please refer to `https://www.lrz.de/services/netz/mobil/vpn/` for more information on how to configure the VPN connection.

In order to install the toolchain, the archive `SIKA_Aufgabe_1.zip` must be downloaded from the website `https://tueisec-sica.sec.ei.tum.de/`. The following Linux distributions are currently supported: Ubuntu Linux 10.04, 13.10 and 14.10.

[1] `login/password: student/student`

Listing 1: Installing the Toolchain

```
$ unzip SIKA_Aufgabe_1.zip
$ cd framework
$ make install
```

Listing 1 provides the commands to install the development toolchain on Ubuntu Linux distributions. The installation will place *all* required files in the folder `$HOME/SIKA_WS_2013_2014/avr/`. Please note that the installation procedure can take up to several minutes and requires at least 2GB of available disk space. In case any problem is encountered during the installation process, please send the `install_toolchain.log` file to `desantis@tum.de` per E-mail.

2.2 Framework

In this section, the framework to implement the AES for the ATMega644V microcontroller is introduced. Please refer to the lecture material and to [1, 3, 2] for a detailed description of AES and implementations thereof.

The framework is located in the folder `code/` of the archive `SIKA_Aufgabe_1.zip`. The folder contains the following files: `Aufgabe_1.cproj`, `main.c`, `student.h`, `student.c` and `AES.py`. The `Aufgabe_1.cproj` is the project file for the Atmel Studio 6.1 development environment, cf. `http://www.atmel.com/tools/atmelstudio.aspx?tab=overview`. Please note that Atmel Studio 6.1 is not required for the completion of the assignment, but may be used to facilitate the development. In any case, the scripts described below should be used for the final evaluation of the implementation.

Listing 2 shows the content of the `main.c` file. The main body defines five variables, namely: **`counter`**, **`key`**, **`buf`**, **`param`** and **`n_enc`**. The variable **`counter`** is just used as internal counter, while the array **`key[16]`** and **`buf[16]`** contain the secret key and the input/output data, respectively. Please note that the array **`buf[16]`** is used for both the input and the output of the AES encryption in order to save resources. This way, the array **`buf[16]`** contains the plaintext before the AES encryption and should contain the ciphertext after the AES encryption.

Listing 2: main.c

```
#include <avr/io.h>
#include <inttypes.h>
#include "student.h"

int main()
{

    uint8_t key[16] =
        { 0xF3, 0x54, 0x1F, 0xA3, 0x4B, 0x33, 0x9C, 0x0D,
          0x80, 0x23, 0x7A, 0xF9, 0x7C, 0x21, 0xD7, 0x3B };
    uint8_t buf[16] =
        { 0x83, 0x85, 0x1F, 0xAB, 0x60, 0x41, 0xCD, 0xF5,
          0x4A, 0x41, 0x6C, 0xDA, 0xF0, 0x12, 0xC2, 0xD4 };

    #define n_enc 30                     // number of encryptions
    volatile uint8_t counter = n_enc;    // counter (must be volatile)
    uint8_t *param;

    param = aes128_init(key);

    while (counter > 0) {
        aes128_encrypt(buf, param);      // actual AES encryption

        counter--;

    }

    counter--;

    // Endless loop
    while (1) {
    }

}

void asmInj()
{
  asm volatile("in r28, 0x3d"::);
}
```

The variable **param** is used to exchange data between the function `aes128_init` and the function `aes128_encrypt` and the variable **n_enc** denotes the number of encryptions which are performed in a row one after the other. Please note that the `aes128_init` function runs only once while the `aes128_encrypt` function is iterated for the amount of times defined by the variable **n_enc**. Therefore, the `aes128_init` function can be used for performing pre-computations, such as computing the AES round keys, while the `aes128_encrypt` function must contain the implementation of the AES encryption algorithm.

The `aes128_init` and `aes128_encrypt` functions are defined in the `student.h` file as shown in the Listing 3 and must be implemented in the `student.c` file shown in the Listing 4.

Listing 3: student.h

```
#ifndef __STUDENT_H__
#define __STUDENT_H__

void aes128_encrypt(void *buffer, void *param);

void *aes128_init(void *key);

#endif
```

The file `student.c` is the only file that should be modified during the implementation, being the only *source file* that must be submitted for completing the assignment. Please note that any modification to the `main.c` or the `student.h` files has no effect during the submission's evaluation and must be avoided to keep the `student.c` file compatible with the evaluation framework.

Listing 4: student.c

```
// OPTIONS: -O2 -std=gnu99
#include "student.h"
#include <inttypes.h>
```

```
void *aes128_init(void *key)
{
        return 0;
}

void aes128_encrypt(void *buffer, void *param)
{

}
```

The first line of the file `student.c` contains the compilation options, which will be used for compiling the source code. Please note that valid compilation options can be specified freely after the keyword `OPTIONS`. In case the keyword `OPTIONS` is not found or the compiling options are not recognized by the compiler, then the standard compiling options as provided in Listing 4 are used for compiling the source code.

Listing 5: AES.py

```
$ cd framework/code/
$ python AES.py 0x3243f6a8885a308d313198a2e0370734 \\
                0x2b7e151628aed2a6abf7158809cf4f3c -i
```

The `AES.py` file contains a Python reference implementation of AES, which can be used during the development to support the implementation of the AES encryption algorithm for the ATMega644V microcontroller. The `AES.py` file takes two strings on inputs prefixed by `0x` corresponding to the secret key and the input plaintext, respectively. The option `-i` can be used to have a verbose dump of all intermediate values resulting from the computation of the AES encryption routine. A sample usage of `AES.py` is provided in Listing 5.

Eventually, the source code can be compiled and verified using the commands provided in Listing 6. The source code is compiled according to the compilation options specified in the first line of the `student.c` file, if any. In case the source code compiles successfully, then the correctness of AES encryptions is also verified automatically. Alternatively, the command `make verify` can be used. The verification process executes the compiled code twice and output the verification's results to the standard output screen.

Listing 6: Compiling and Verifying the Implementation

```
$ make compile
$ make verify
```

In case the verification is successful, then the performances of the AES implementation are automatically evaluated. The evaluation process computes the average number of clock cycles required by bulk encryptions, the `RAM` usage and the code size of the given implementation. The number of clock cycles of bulk encryptions corresponds to the number of clock cycles taken by the `aes128_encrypt` function only, while the `RAM` usage corresponds to the number of bytes which are either read or written by the complete program execution. The code size refers to the size of `text` and `data` segments, that is, the size of the program code and constants in bytes.

2.3 Some Ideas for Optimizations

For the assignment, one optimization goal must be chosen. Possible optimization goals are: minimizing the `RAM` usage, maximizing the throughput or minimizing the code size. In this section, some hints for optimization are *briefly* summarized.

Hints for Minimizing the `RAM` Usage

- Store lookup tables (*e.g.*, S-box) into the flash memory. Beware that the `const` keyword alone does not place the variable into the flash memory, as they will be still placed into the `RAM` by the compiler. In order to store a variable into the flash memory, use the `PROGMEM` macro defined by the `avr/pgmspace.h` file, c.f. `http://www.nongnu.org/avr-libc/user-manual/pgmspace.html`.
- Compute round keys and the S-box on-the-fly, rather than storing them in `RAM`. The computation of round keys and S-box on-the-fly typically takes several clock cycles, but it helps to save memory.
- Do not use the `malloc()` and `free()` functions as they are typically not necessary and problematic on small MCUs. Additionally, they can potentially lead to a larger `RAM` usage, due to the dynamic allocation of memory.
- Use types of appropriate size. For example, the S-box can be defined using the type `uint8_t`, whose size is one byte, in place of the type `int` whose size is two bytes.
- Compute the key schedule back and forth by overwriting the round key every time. This helps saving memory at the cost of some more clock cycles. Please note that the secret key must be computed back to get the original secret key before the next encryption.
- Loop unrolling can be employed to save yet another small amount of `RAM`, as loop counters are not stored in memory.

Hints for Maximizing the Throughput

- In general, do not perform on- the-fly calculations and store lookup tables in memory. Table lookups can possibly be stored in flash memory.
- Take advantage of the `aes128_init` function to pre-compute the round keys. The secret key does not change over the encryptions, so round key can be pre-computed.
- Avoid direct use of multiplication and divisions if not necessary. Use simple shift operations whenever applicable.
- Avoid frequent function calls as they typically require some clock cycles for setting up the stack
- Count backwards in `for` loops. The "branch if zero" instruction can be used to make the resulting code relatively faster and smaller.

Hints for Minimizing the Code Size

- In general, do not use lookup tables and compute data on-the-fly as much as possible. Bruteforce search can be used to implement the S-box lookup.
- Avoid loop unrolling and inline functions as they use the program space.
- Use types of appropriate size for constants.

2.4 Writing the Report

For completing the assignment, a report must be submitted. The report must specify the selected design goal (*e.g.* maximizing the throughput or minimizing `RAM` usage or code size) and describe the design choices which were taken during the development of the AES implementation to pursue the targeted design goal with a sufficient level of detail.

A LaTeX template file for the report named `SIKA_AES_Report.tex` can be found in the folder `report/` of the archive `SIKA_Aufgabe_1.zip`. The following tools and references are recommended for writing the report in LaTeX:

- TeXstudio (Windows/Linux/Mac OS X): `http://texstudio.sourceforge.net/#download`
- TexLive: `http://www.tug.org/texlive/`
- Wikibook LaTeX: `http://en.wikibooks.org/wiki/LaTeX`

2.5 Submitting Results

In order to complete the assignment, the source code of the implementation (`student.c` file) along with a report in `PDF` format must be submitted. The files can either be submitted using the commands provided in Listing 7 or by uploading the files at the address `https://tueisec-sica.sec.ei.tum.de/handin/`. Upon successful upload of the submission, a confirmation E-mail is sent back. Please verify the checksum of submitted files against the confirmation E-mail always.

Listing 7: Submitting results

```
$ make submit
$ md5sum code/student.c report/SIKA_AES_Report.pdf
```

3 FAQ

Q: How do I *completely* remove the installed toolchain from my computer?

A: Issue the following command:

```
$ make remove
```

References

[1] J. Daemen and V. Rijmen. Aes proposal: Rijndael. NIST Web page, March 1999.

[2] J. Daemen and V. Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Information Security and Cryptography. Springer, 2002.

[3] National Institute of Standards and Technology. Federal information processing standards FIPS 197. *Advanced Encryption Standard (AES)*, 2001.