ECSE426 Microprocessor Systems

Fall 2013

Lab 1: Tiny Encryption Algorithm (TEA)

Objective

This exercise introduces the ARM Cortex assembly language, the instruction set and the addressing modes. The ARM calling convention will need to be respected, such that the assembly code can later be used with C programming language. The lab and a tutorial prior to it will also introduce you to the Keil compiler, simulator and associated tools. The code developed here will be used in the project to secure the wireless communication channel between communicating nodes.

Background and Preparation

Calling convention

In assembly, parameters for a subroutine are passed on the memory stack and via internal registers. In ARM processors, the scratch registers are R_0:R_3 for integer variables. Up to four parameters are placed in these registers, and the result is placed in R0 - R1. If any parameter requires more than 32 bits, then multiple registers are used. If there are no free scratch registers, or the parameter requires more registers than remain, then the parameter is pushed onto the stack. In addition to the class notes, please refer to the document "Procedure Call Standard for the ARM Architecture", especially its sections 5.3-5.5. This particular order of passing parameters is eventually a convention applied by specific compilers. Please be aware that the several different procedure calling and ordering conventions exist beyond the one used here, but this procedure call convention is standardized by ARM.

Other documents that will be of importance include the Cortex M4 programming manual, and Quick reference cards for ARM ISA, all available within the course online documentation. More useful references are found in the tutorial slides.

Using the Keil Integrated Development Environment Tool

To prepare for Lab 1, you will need to go through Tutorial 1, where you will learn how to create and define projects, specifically purely assembly code projects. The tutorial shows you how to let the tool insert the proper startup code for the given processor, write and compile the code, as well as provide the basics of the program debugging. Extensive details about debugging in Keil are provided in the document entitled Debugging with Keil also uploaded on my courses.

The Experiment

Part I-A – Pure Assembly Programming

You will develop the working assembly language code for the Tiny Encryption Algorithm that can be used in the final project. You are required to write two subroutines, one that encrypts a message and another that decrypts it.

TEA was designed by Cambridge University computer scientists David Wheeler and Roger Needham in 1994. The aim was to design a cryptographic algorithm which is effective (in terms of security strength), simple (in terms of ease of implementation) and performance. Instead of algorithmic complexity, the authors opted for the use of multiple passes over simpler operations to achieve a high security level. The algorithm is symmetric, that is it uses the same key for encryption and decryption. This fast algorithm makes use of basic operations which are readily available in hardware. This allows for easy assembly language mapping. The algorithm is easily implemented in various high level and assembly languages. It has been used in Microsoft Xbox as well. Due to its small footprint and low memory requirement, it has been used in resource constrained devices, such as mobile and embedded devices. The algorithm was later succeeded by the XTEA (TEA Extensions) and block XTEA (XXTEA) to address some weaknesses which were discovered later (i.e. hacked Xbox console). The algorithm is available in the public domain and is not patented.

The algorithm uses a key of 128 bits length (4 unsigned integers) and works on data in two blocks of 32 bits each (two unsigned integers). It goes through 32 iterations to achieve higher diffusion and therefore higher level of security. Since the body of iterations is symmetric in nature, it is prone to attacks. To guard against this, a value (a magic constant) called *delta* is used where *delta* is equal $2^{31}(5^{½} - 1)$, or 2654435769 ($9E3779B9_{16}$).

Given that D_0 and D_1 collectively represent the 64 bit data input block, that $K_0 – K_3$ collectively represent the 128 bit key, and N is the number of iterations (default 32) then the pseudo-code of the algorithm can be presented as follows:

A Reference implementation	
TEA Encryption	**TEA Decryption**
Sum = 0 δ = 0x9e3779b9 **Repeat N times:** *Increment sum by δ* T_1= *Left shift* D_1 *by 4 and add* K_0 *to it* T_2 = *Right shift* D_1 *by 5 and add* K_1 *to it* $T_3 = D_1 + sum$ $T_4 = T_1 \oplus T_2 \oplus T_3$ $D_0 = D_0 + T_4$ T_1= *Left shift* D_0 *by 4 and add* K_2 *to it* T_2 = *Right shift* D_0 *by 5 and add* K_3 *to it* $T_3 = D_0 + sum$ $T_4 = T_1 \oplus T_2 \oplus T_3$ $D_1 = D_1 + T_4$ **End Repeat**	Sum = 0xC6EF3720 δ = 0x9e3779b9 **Repeat N times:** T_1= *Left shift* D_0 *by 4 and add* K_2 *to it* T_2 = *Right shift* D_0 *by 5 and add* K_3 *to it* $T_3 = D_0 + sum$ $T_4 = T_1 \oplus T_2 \oplus T_3$ $D_1 = D_1 - T_4$ T_1= *Left shift* D_1 *by 4 and add* K_0 *to it* T_2 = *Right shift* D_1 *by 5 and add* K_1 *to it* $T_3 = D_1 + sum$ $T_4 = T_1 \oplus T_2 \oplus T_3$ $D_0 = D_0 - T_4$ *Decrement sum by δ* **End Repeat**
$\oplus$ *denotes an XOR operation*	

You can choose whatever keys you wish during your implementation. However, during demo time, we will give you a specific key such that we will be able to compare your output with ours. In C, the key is stored in a struct or array.

Part I-B – Mixing C and Assembly

You are required to write the same subroutines in the C language. Then you are to call both the C and assembly subroutines from main. You should compare the performance between the two implementations and present analysis on execution time differences. Regardless to say, the output of both implementations should match. Also, the same parameters are to be passed to the both assembly and C routines; that is the function call to both should be similar.

Part II – C Programming

Alphanumerical characters on a computer have a designated code assigned to them conforming to a certain standard such as ASCII (American Standard Code for Information Interchange). For example, the number zero has a value of 48d (0x30); the capital letter Q has the value 81d (0x64).

Long ago, one of your TA's encrypted a message with the Tiny Encryption Algorithm using multiple encryption keys. Due to a series of unfortunate events however, he has lost the original unencrypted message along with all the encryption keys used. Since then, he has managed to recover all encrypted versions of the message, along with the format used to generate encryption keys.

The encryption key for the message can be separated into two parts. The first two 32-bit unsigned integers is generated by encrypting the ASCII representations of the capital letters T and A, using the key {1, 2, 3, 4} by the Tiny Encryption Algorithm. So basically, the key which is used to decode the hidden messages is itself partially encrypted by the key {1, 2, 3, 4}. The last two 32-bit unsigned integers of the encryption key for the message consists of two lower case characters (in ASCII representation) ranging from d to h.

For Part 2, you will be given one of the encrypted messages to decrypt (see .h file and find the encrypted message associated with your group number), producing the original message, along with the encryption key used. The task is difficult, and brute force methods of decryption are expected. Remember, the original message is a line of formatted English with proper spelling and grammar.

Assembly Implementation hints

Your subroutine should follow the ARM compiler parameter passing convention. Recall that up to 4 arguments can be passed by four integer registers. If the datatype is more complex (e.g, struct or a matrix), or you need to pass more values than allowed by the convention, then a **pointer** to it is passed instead. For the function return value, the register R0 and R1 are used for integer result of the subroutine. In your case, there is a need for two integer parameters (a pointer to the encryption/decryption keys and a pointer to the message).

The operation of the algorithm should be correct regardless of the message length. If the message is shorter than a block of 64 bits (i.e. could be the case by the end of the message), you should manually pad it in your program by adding a string of zeros to the end of the message.

You should store the original message in the code segment right after your code. You should define an empty space in the data segment to place your encrypted message. The decrypt routine will also read/write the message in the data segment. You may wish to consult ARM Info center or Keil help on Assembly for assembler directives on how to do that. The Data segment should be stored in a separate file (otherwise this will cause errors). Refer to tutorial slides to solve visibility and referencing issues.

Remember that 32 bit constants cannot be directly added to other variables, you may need to load the constant into a register using LDR instruction before using it in further operations.

To view memory content, in Keil debugging view go to View→Memory windows then choose any memory window 1 – 4 you like. Write down the address you wish to start seeing content at (You can get this during program execution when you load the address into a register i.e. LDR R0, =memoryRegion). You can then right click anywhere in the memory window to choose how the data is to be interpreted and displayed (i.e. ASCII, integers, floats .. etc)

Function Requirements

1. Your code should use minimum number of registers, careful analysis and tracking of register usage is important.
2. Your code should have minimum code density; that is; it should consume minimum memory footprint.
3. You are required to use the stack
4. Your code should run as fast as possible. You should optimize beyond your initial crude implementation.
5. Your code should make use of modular design and function reuse whenever possible
6. Codes will be compared against each other for the above criteria during demo time. Those who achieve best results will ***get the highest demo grades.***
7. The subroutine should be robust and correct for all cases. Grading of the experiment will depend on how efficiently you solve the problem, with all the corner cases (if any) being correct.
8. All registers specified by ARM calling convention are to be preserved, as well as the stack position. It should be unaffected by the subroutine call upon returning.
9. The calling convention will obey that of the compiler.
10. The subroutine should not use any global variables

Linking

1. When creating a new project, it is best to include the startup code, for which the tool will ask you whether to include it. Then, modify that code to branch to the encryption subroutine followed by the decryption subroutine rather than __main for testing assembly code alone, prior to embedding into C main program. Please note that you will need to declare as exported the subroutine name in your assembly code.
2. If the linker complains about some other missing variable to be imported in startup code, you can either declare it as “dummy” in your assembly code, or comment its mention in the startup code.
3. For linking with C code that has main, none of the above two measures are needed.

Test Samples

To test your algorithm, use any one of the provided samples (without quotation marks and references, but with capitalization, punctuation and spaces preserved). Yet, during demo time, the TAs will use any sample codes they wish. We will compare your output to ours and determine if your implementation works perfectly.

Test Sample 1:

> *"Space: the final frontier. These are the voyages of the starship Enterprise. Its continuing mission to explore strange new worlds, to seek out new life and new civilizations, to boldly go where no one has gone before."* **Captain Picard [opening narration], Star Trek: The Next Generation (1987 - 1994)**

Test Sample 2:

> *"It began with the forging of the great rings. Three were given to the Elves: Immortal, wisest and fairest of all beings. Seven to the dwarf lords: Great miners and craftsmen of the mountain halls. And nine, nine rings were gifted to the race of men who, above all else, desire power. For within these rings was bound the strength and the will to govern each race. But they were, all of them, deceived; for another ring was made. In the land of Mordor, in the fires of Mount Doom, the Dark Lord Sauron forged, in secret, a master ring. And into this ring he poured his cruelty, his malice and his will to dominate all life. One ring to rule them all. One by one, the free lands of middle-earth fell to the power of the ring."* **Galadriel [opening narration], The Lord of the Rings: The Fellowship of the Ring (2001)**

Test Sample 3:

> *"When we first met, you told me that a disguise is a self-portrait. How true of you. The combination to your safe: your measurements. But this, this is far more intimate; this is your heart, and you should never let it rule your head. You could've chosen any random number and walked out of here today with everything you've worked for, but you just couldn't resist it, could you? I've always assumed that love is a dangerous disadvantage. Thank you for the final proof."* **Sherlock Holmes to Irene Adler, Sherlock TV series, A scandal in Belgravia (2012)**

Demonstration and Documentation

The demonstration involves showing your source code and demonstrating a working program. You should be able to call your subroutine several times consecutively. You should be able to explain what every line in your code does – there will be questions in the demo dealing with all the aspects of your code and its performance. The questions might regard the skeleton code to initiate and start the assembly code that we gave you in Tutorial 1; ask if you do not know!

It is by far the most efficient that you have the full documentation on your assembly code subroutine completed upon demonstrating the Lab 1, especially that future labs will require lots of documentation and additional code development on its own and following Doxygen documentation standard.

Demos will be held on Friday, October 4th, 2013, This lab has a weight of 6 marks