Assignment Name	Assignment 1: Efficient Implementation
Student ID	1675746, 2107884

University of Birmingham Assessment and Feedback

Section One

Reflecting on the feedback that we have received on previous assessments (including from your previous studies), the following **three** issues/topics have been identified as areas for improvement:

1	Further detail on some ideas
2	Focus on some areas of study over others
3	Efficiently getting to the crux of the question

Section Two

In this assignment, we have attempted to act on previous feedback in the following ways:

1	Dividing up responsibility equally on all areas of the project
2	Time management for trickier areas
3	Being concise with answers and cutting out as much as possible

Section Three

Feedback on the following aspects of this assignment (i.e. content/style/approach) would be particularly helpful to us:

1	Technical Accuracy
2	Approach effectiveness
3	Describing all aspects of what you were looking for

Hardware and Embedded Systems Security
Assignment 1: Implementation

Graeme Brodie 2107884, Dee Dixon 1675746

1.1 Reference Implementation

1. Average number of cycles:

Cycles per Execution:

(165824 + 165568 + 167672 + 169224 + 169224) / 5 = 167502.4

167502 Cycles per Execution

Cycles per bit (Throughput):

167502 / 64 = 2617.225

2617 Cycles per bit

2. Special techniques or Optimizations

This implementation went through several stages of improvements. Initially, we constructed a basic implementation with reference to the course material to ensure that we had a functioning foundation. Once working, we began replacing portions with experimental code and examining speed improvements.

We experimented with the function `cpybit()/setbitbyvalue()` and compared its performance to a second technique that checked extracted bits with `if` in the permutation stage. The function was noticeably slower; the function involves both a clearing and setting of a bit (with multiple shift operations each time) every permutation, whereas with `if` we could select the operation specifically. These function calls would also have consumed additional clock cycles.

We also exploited the fact that permutations cannot be done in-place, and that C arrays can be initialised to zero, to remove the need for a `clearbit()` function. As a newly declared state contains only zero bits, we could ignore all permutation steps which move zero bits. Though the initialisation costs clock cycles, it was a significantly lower cost than calls to `clearbit()`.

We have deliberately not compressed expressions used in this final implementation into single lines; this is to aid readability and commenting for this reference implementation. We instead attempted these kinds of optimisations in the bitslicing implementation.

1.2 Bitslicing Implementation

1. Optimizations and performance affect

The initial bitslicing implementation took an average of 25420 cycles to encrypt a block, based on the test script. This implementation already included a pair of optimisations:

- `getbit()`, which uses multiple bitshift operations to create a bitmask for an integer (expensive on the MSP430), was replaced in `enslice` and `unslice` by destructively iterating over a copy of the input integer. This involved a single shift and mask operation per iteration to repeatedly extract a new least significant bit.
- The state and expanded key arrays were initialised with all elements zeroed, as per the reference implementation optimisations. Only set bits had to be copied when slicing and unslicing, thus eliminating the need for the similarly expensive `clearbit()` and `cpybit()`.

From this initial stage, further optimisations were made:

Optimisation	Benefit (average cycles per block)
Applied the destructive iteration principle to the expansion of round keys.	4825 (19%)
Permutations operate on a pair of buffers, swapped using pointers between input and output states. This eliminates both the declaration of new, emptied arrays for each permutation and the need to copy the updated state.	5588 (22%)
Reuse of the input array `pt` rather than using a temporary buffer.	1086 (4%)
Combined the sbox stage with the permutation layer, eliminating several loops and function calls.	2024 (8%)
Removed a call to `memset` that was being used to zero backstates unnecessarily.	1086 (4%)
Removed an unnecessary zeroing of the expanded key array. Merged the key addition step with the sbox/pbox process, eliminating a loop and function call.	2616 (10%)

We attempted several other improvements that, possibly due to optimisations already being performed by the compiler, were not as effective. Some had a very minimal positive effect on clock rate, some had no effect, and some actually increased the number of cycles. Some of these failed optimisations have been left as comments in the code.

Attempted Optimisation	Effect on clock cycles
Unrolling bit setting loops in `enslice()` and `unslice()`.	None
Filling the state from the most significant bit and shifting down for each iteration in `enslice()`.	Minor increase
Concatenating arithmetic operations in various locations throughout the code (at the expense of readability).	<1% reduction (implemented) <1% increase Most had no effect.
Pre-calculating frequently used values in the combined pbox/sbox function.	None

2. Average number of cycles

Cycles per Execution:

(146320 + 146016 + 146296) / 3 = 146210.7 / 16 = 9138.2

9138 Cycles per Execution

Cycles per bit (Throughput):

9138.2 / 64 = 142.8

148.8 Cycles per bit

1.3 Hardware AES

1. Average number of cycles

Original
Cycles per Execution:
(2240 + 2240) / 2 = 2240
2240 Cycles per Execution

Cycles per bit (Throughput):
2240 / 128 = 17.5
17.5 Cycles per bit

Optimised
Cycles per Execution:
(2120 + 2120) / 2 = 2120
2120 Cycles per Execution

Cycles per bit (Throughput):
2120 / 128 = 16.5625
16.6 Cycles per bit

2. Results Comparison
When comparing cycles per bit, the AES implementation is quicker than the optimised bitslicing algorithm by almost a factor of 10; with regard to the reference implementation, it is faster by almost a factor of 200. This shows the significant speed benefits of using dedicated hardware.

Our implementation removed unnecessary code paths for our specific encryption to further reduce clock cycles. We considered removing arithmetic operations made between some constant values, but compiler optimisations made this unnecessary in most cases. There were also some instances where the default contents of the memory-mapped registers were already correct for the type of encryption we were trying to perform; however, we could not assume that those registers would retain those default values.

3. Low-Level Firmware Steps
In the MSP430, hardware registers are memory mapped so that they can be addressed directly. For the AES Module, there are several registers that are used to set the parameters of the encryption process, some of which are shown below (Taken from datasheet):

Table 6-48. AES Accelerator Registers (Base Address: 09C0h)

REGISTER DESCRIPTION	REGISTER	OFFSET
AES accelerator control 0	AESACTL0	00h
AES accelerator control 1	AESACTL1	02h
AES accelerator status	AESASTAT	04h
AES accelerator key	AESAKEY	06h
AES accelerator data in	AESADIN	008h
AES accelerator data out	AESADOUT	00Ah
AES accelerator XORed data in	AESAXDIN	00Ch
AES accelerator XORed data in (no trigger)	AESAXIN	00Eh

Register read/write operation	Purpose
AES256_setCipherKey()	
Write to first control register: OFS_AESACTL0	Selects key type by & the register with bitmask ~(AESKL_1 + AESKL_2) = 0xFFF3
Load key into the key register OFS_AESAKEY	The register is 16 bits wide, so the function maps 16 bytes into eight 16-bit words. Each 16 bit word of the key is loaded into the register sequentially.
Read the bit 1 of status register OFS_AESASTAT. (Mask 0x0002 = AESKEYWR used)	This bit in the register is checked repeatedly until it changes from 0x00; this indicates that the key has been loaded by the AES module.
AES256_encryptData()	
Write to first control register: OFS_AESACTL0	Sets the AES module to encrypt mode by & the register with bitmask ~AESOP_3 = 0xFFFC
Load the plaintext into the input register: OFS_AESADIN	The register is 16 bits wide, so the function condenses 16 bytes into eight 16-bit words. Each 16 bit word of the plaintext is loaded into the register sequentially.
Sets bit 1 of the status register OFS_AESASTAT. (Mask 0x0002 = AESKEYWR used)	Initialises encryption phase for the AES module.
Read the bit 0 of status register OFS_AESASTAT. (Mask 0x0001 = AESBUSY used)	This bit in the register is checked repeatedly until it changes from 0x1 = AESBUSY; this indicates that the encryption has been completed by the AES module.
Repeatedly read register AESADOUT	The register is 16 bits wide, so the function maps eight 16-bit words into 16 bytes. Each byte of this ciphertext is copied into the output array.

4. Circumstances for Software

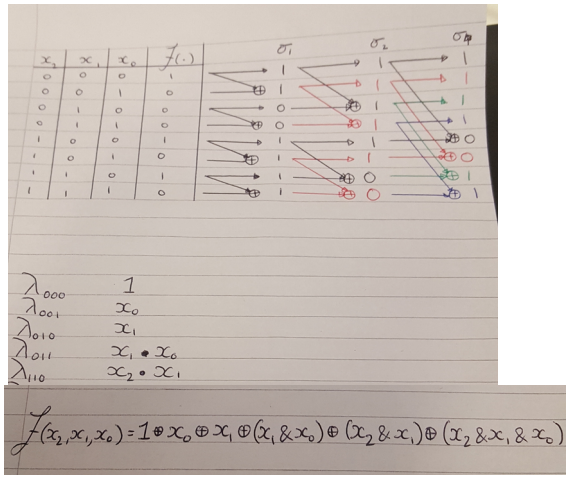
Not all microcontrollers have a hardware cryptography module; in these situations, a software solution is the only available means of implementing cryptography.

The inclusion of a AES hardware modules in a microcontroller increases the cost of the device, which may be undesirable if large numbers of that microcontroller are required and budget is limited.

Software implementations can be updated if vulnerabilities are found. This may be in the specifics of the algorithm they have implemented, or if future cryptographic options become available. Most hardware implementations can’t be altered.

2.1: ANF

x_2	x_1	x_0	f (·)	σ_1	σ_2	σ_4
0	0	0	1	1	1	1
0	0	1	0	1	1	1
0	1	0	0	0	1	1
0	1	1	0	0	1	1
1	0	0	1	1	1	0
1	0	1	0	1	1	0
1	1	0	1	1	0	1
1	1	1	0	1	0	1



Equation Simplification:

$1 \wedge X_0 \wedge X_1 \wedge (X_1 \& X_0) \wedge (X_2 \& X_1) \wedge (X_2 \& X_1 \& X_0)$

$1 \wedge X_0 \wedge (X_1 \& (1 \wedge X_0 \wedge X_2 \wedge (X_2 \& X_0)))$

2.2 Long Number Arithmetic

Base = 10

2.2.1 Addition

a + b for a = $(12345)_{10}$ and b = $(54321)_{10}$ using schoolbook algorithm

a	|	1	2	3	4	5
b	|	5	4	3	2	1
carry	|	0	0	0	0	0
sum	|	6	6	6	6	6

Elementary base-b additions: k = 5

2.2.2 Multiplication

a · b for a = $(1234)_{10}$ and b = $(12345)_{10}$ using schoolbook algorithm

```
1234  *  12345
          20
+        150
+       1000
+       5000
-------------
=       6170
+        160
+       1200
+       8000
+      40000
         1 1
-------------
=      55530
+       1200
+       9000
+      60000
+     300000
         11
-------------
=     425730
+       8000
+      60000
+     400000
+    2000000
          1
-------------
=    2893730
+      40000
+     300000
+    2000000
+   10000000
          1
-------------
=   15233730
```

Elementary base-b multiplications:

m * k = 4 * 5 = 20