by William Harrington

International Data Encryption Algorithm

The International Data Encryption Algorithm (IDEA) is a symmetric-key block cipher that operates on 64-bit plaintext blocks. The encryption key is 128-bits long, and the same algorithm is used for both encryption and decryption. IDEA uses Exclusive-OR ($\oplus$), addition modulo 2^{16} ($\boxplus$), and multiplcation modulo $2^{16}+1$ ($\odot$) operations during its "rounds" for encryption and decryption. All operations are performed on 16-bit sub-blocks with a 16-bit subkey.

Operations

Understanding these operations are critical for implementing an application specific processor to perform the IDEA algorithm. In this section, I explore the IDEA algorithm by coding it in Python at a high level to verify the behavior.

Addition modulo 2^{16}

Addition modulo 2^{16} is defined as $a + b \mod 2^{16}$. The additive inverse of an operand for this operation is the twos compliment of the operand (i.e. ~a+1). As an example, if $Sum = a + b \mod 2^{16}$ then $a = Sum+(\sim b+1) \mod 2^{16}$. The following code implements this operation with a = 1 and runs a couple of random test cases to verify the behavior.

```
from myhdl import *
from IDEA import *

# generate random test cases
numbers = [randint(0,15) for x in range(4)]

# my number
mynum = 1

# iterate through test cases, only looking at first 4 bits
for i in numbers:
   print 'mynum + i: %s, inverse(i): %s, (mynum + i) + inverse(i): %s' % \
                         (bin(addition(mynum, i)[4:]),
                         bin(inv_addition(i)[4:]),
                addition(addition(mynum, i), inv_addition(i))[4:])
```

```
mynum + i: 101, inverse(i): 1100, (mynum + i) + inverse(i): 1
mynum + i: 1000, inverse(i): 1001, (mynum + i) + inverse(i): 1
mynum + i: 1101, inverse(i): 100, (mynum + i) + inverse(i): 1
mynum + i: 100, inverse(i): 1101, (mynum + i) + inverse(i): 1
```

Multiplication Modulo $2^{16}+1$

Multiplication Modulo $2^{16}+1$ is defined as $a \times b \mod 2^{16}+1$. To perform multiplication modulo $2^{16}+1$, two numbers say A and B are multiplied together and that product is then modded with $2^{16}+1$. If A or B is equal to 0, this is interpreted as 2^{16}. The MyHDL library handles this behavior with the modular bit vector type (modbv).

Their documentation[3] explains the behavior as follows:

```
val = (val - min) % (max - min) + min
```

Where val is the value specified in the modbv declaration.

The multiplicative inverse of some number say X, where $X \neq 0$ is $X^{p^2} \mod p$, where p is $2^{16}+1$ [4]. As an example, if $Product = a \times b \mod 2^{16}+1$ then a = $Product \times b^{(2^{16}+1)2} \mod 2^{16}+1$. The following code implements this operation with a = 1 and runs a couple of random test cases to verify the behavior.

```
# generate random test cases
numbers = [randint(0,15) for x in range(4)]

# iterate over test cases, only looking at the first 4-bits of the results
for i in numbers:
   print 'mynum * i: %s, inverse(i): %s, (mynum * i) * inverse(i): %s' % \
                        (bin(multiply(mynum, i)[4:]),
                        bin(inv_mult(i)[4:]),
                    multiply(multiply(mynum, i), inv_mult(i))[4:])

mynum * i: 1101, inverse(i): 100, (mynum * i) * inverse(i): 1
mynum * i: 1011, inverse(i): 110, (mynum * i) * inverse(i): 1
mynum * i: 1100, inverse(i): 110, (mynum * i) * inverse(i): 1
mynum * i: 10, inverse(i): 1, (mynum * i) * inverse(i): 1
```

Key scheduling

The algorithm uses 52 subkeys: six for each of the eight rounds and four more for the output transformation. In order to do this, the original key is broken up into six subkeys then rotated left by 25 bits. This is done until all 52 subkeys are created.

Algorithm for Encryption/Decryption

The encryption and decryption algorithm is composed of 8 "rounds" that take 14 identical steps followed by a final output transformation that is referred to as a "half-round". The operations are performed with a 64-bit plaintext block and a 128-bit key. The plaintext block is separated into four 16-bit sub-blocks: X_1, X_2, X_3, X_4. The key is eventually broken down into 52 sub-keys. However, it is first split up into eight 16-bit sub-keys: $K_1, ..., K_8$. The first six $(K_1, .., K_6)$

are used in the first round and the last two (K_7, K_8) are the first two used in the second round. The key is then rotated 25 bits to the left and split up into eight more 16-bit sub-keys: $K_9, ..., K_{16}$. The first four $(K_9, .., K_{12})$ are used in round two after K_7, K_8; the last four are used in round 3. The key is rotated to the left again 25-bits to obtain the next eight subkeys, and so on until the end of the algorithm.[5]

The steps for each round are listed below. Exclusive-OR = $\oplus$, Addition modulo $2^{16} = \boxplus$, and Multiplcation modulo $2^{16} + 1 = \odot$.

Step	Operation	Step	Operation
1.	$X_1 \odot K_1$	8.	Step6 $\boxplus$ Step7
2.	$X_2 \boxplus K_2$	9.	Step8 $\odot$ K_6
3.	$X_3 \boxplus K_3$	10.	Step7 $\boxplus$ Step9
4.	$X_4 \odot K_4$	11.	Step1 $\oplus$ Step9
5.	Step1 $\oplus$ Step3	12.	Step3 $\oplus$ Step9
6.	Step2 $\oplus$ Step4	13.	Step2 $\oplus$ Step10
7.	Step5 $\odot$ K_5	14.	Step4 $\oplus$ Step10

The output of the round is the four sub-blocks: O_1, O_2, O_3, O_4 that are the results of steps 11, 12, 13, and 14. These are the input to the next round. This happens eight times and on the eight round the inner blocks are swapped then followed by a final output transformation.

The final output transformation consists of the following four steps:

Step	Operation	Step	Operation
1.	$X_1 \odot K_1$	3.	$X_3 \boxplus K_3$
2.	$X_2 \boxplus K_2$	4.	$X_4 \odot K_4$

These four steps are then combined to produce the cipher test. For decryption, the keys are used in reverse order and are either the additive or multiplicative inverses of the encryption subkeys.

The following code performs encryption with the following values: plaintext=0x20822C1109510840, key=0x7802c45144634a43fa10a15c405a4a42. The ciphertext should be equal to 0x627bbcdcbe7bd9ac.

Tools like this one: http://lpb.canb.auug.org.au/adfa/src/IDEAcalc/ can be used for verifying this behavior

```
plaintext = modbv(0x20822C1109510840)[64:]
key = modbv(0x7802c45144634a43fa10a15c405a4a42)[128:]
keys = create_subkeys(key)

ciphertext = encrypt(plaintext, keys)
```

```
print 'Plaintext: %s' % hex(plaintext)
print 'Ciphertext: %s' % hex(ciphertext)

Plaintext: 0x20822c1109510840L
Ciphertext: 0x627bbcdcbe7bd9acL
```