# Hardware Implementation of IDEA (International Data Encryption Algorithm)

Gerald Lai
Oregon State University
laige@ece.orst.edu

**ABSTRACT**

In 1973, the National Bureau of Standards (NBS, now known as the National Institute of Standards and Technology, or NIST) selected the Data Encryption Algorithm (DEA, later known as DES) to serve as a common standard. DES was designed at the IBM TJ Watson Research Center at the request of the NBS for protecting sensitive data.

It is a block cipher operating on a 64-bit plaintext to produce a 64-bit ciphertext. The symmetric key used is a 56-bit encryption key that was strong at the time. In other words, there are 72,057,594,037,927,936 (or 2 to the 56th power) possible combinations of keys. It was claimed that a message encrypted with DES would take an immensely long time to crack.

Yet, despite its sophistication, many future attempts at cracking DES showed significant signs of success. For example, the distributive computing approach of spreading cracking computation power over the Internet earned Rocke Verser and Michael Sanders the prize of the 1997 DES Challenge. DES Challenge II was also cracked the following year. With the invention of the Electronic Frontier Foundation DES Cracker, it was shown that a 56-bit key protection is insufficient against exhaustive search employed with today's technology. Therefore, there was an urgent call for a stronger secret-key encryption algorithm. IDEA was one of the algorithms to answer that call.

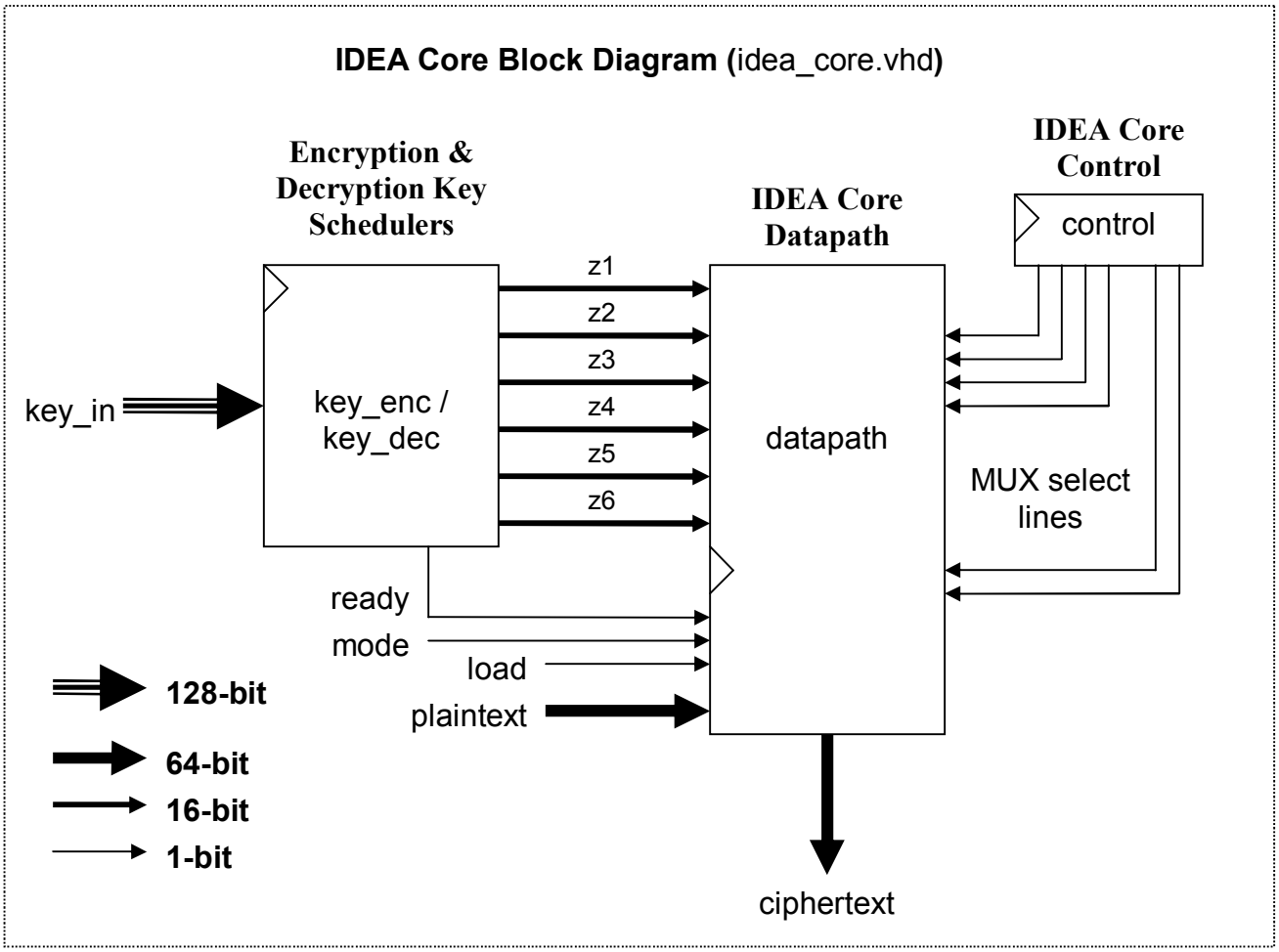
## I. INTRODUCTION

The International Data Encryption Algorithm (IDEA) was developed in Zurich, Switzerland by James Massey and Xuejia Lai and published in 1990. It operates on 64-bit plaintext and ciphertext blocks with a 128-bit key. IDEA is used by the popular program Pretty Good Privacy (PGP) to encrypt files and electronic mail. Unfortunately, wider use of IDEA has been hampered by a series of software patents on the algorithm, which is currently held until 2011 by Ascom-Tech AG in Solothurn, Switzerland. MediaCrypt offers a royalty-free license for non-commercial use.

IDEA is somewhat different from the rest of the symmetric key encryption algorithms in that it uses algebraic operations completely and does without table lookup methods. It employs a

modified 4-word Feistel style round function system. The strength of IDEA lies in its modulo multiplication operations and therefore, it relies heavily on modular inversion.

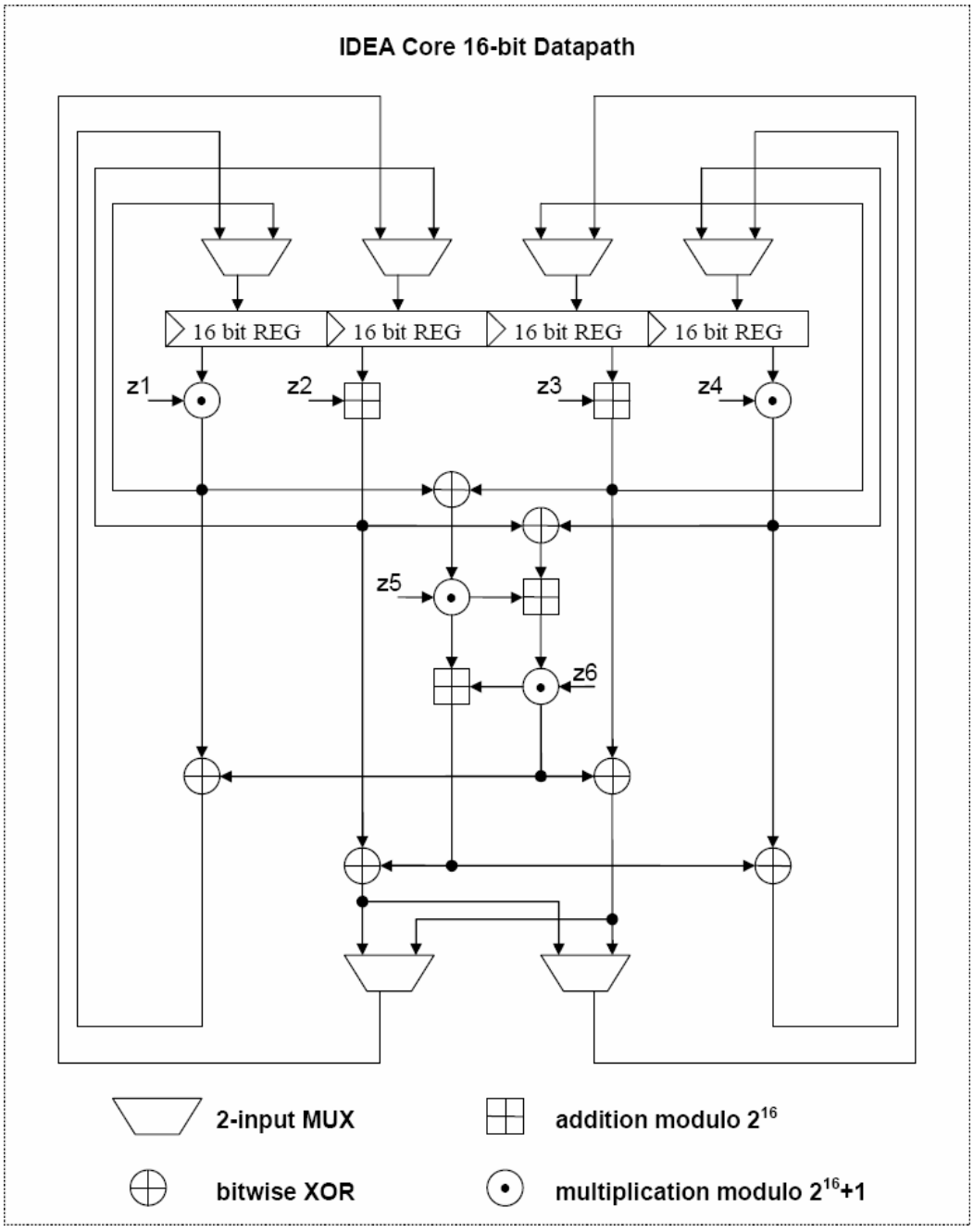
The project is sub-divided into 5 parts, which are the IDEA core, the key schedulers for encryption and decryption, the modular multiplier and the modular inverter. It is divided this way to separate the encryption portion from the decryption portion. Both encryption and decryption functions share the same datapath and control (state machine) but are scheduled differently. The modular multiplier is an integral part of the main core. The modular inverter will be part of the decryption key scheduler.




The entire project is written in VHDL. The source files will be simulated using Mentor Graphics tools. In particular, VCOM will be used to compile the *.VHD files and VSIM to simulate it.

## II. IDEA CORE

The IDEA Core consists of the datapath and control unit that facilitates the encryption and decryption process. Algebraic operations are performed on 16-bit words. The operations are inter-related with one another as shown in the datapath diagram below. The logic path from the output of the four 16-bit data registers back into the input represents one round of IDEA. By definition of IDEA, a total of nine rounds are required, where the last round is known as the Output Transform round and is handled differently.



The standard round of IDEA requires selecting the top MUXes to let the registers latch onto the result that appears at the bottom of the datapath, and selecting the bottom MUXes so that there is a swap of the left and right 16-bit word lines. The Output Transform round requires selecting the top MUXes to let the registers latch onto the result that appears at the middle of the datapath, and selecting the bottom MUXes the previous round (i.e, on round 8) so that there is no swap of the left and right 16-bit word lines.

An encryption operation is performed on 64 bits of plaintext given a 128-bit key. The key is held at the input for 9 clock cycles in order to obtain the correct ciphertext at the output. For decryption, the ciphertext is given as an input. However, the key is held for an unaccounted number of clock cycles in order to get the resulting plaintext. This is due to the hardware computation of the inverse of the sub-keys.

## III. ENCRYPTION KEY SCHEDULER

For encryption key scheduling, IDEA partitions the original 128-bit key into eight 16-bit sub-blocks that are directly used as the first eight key sub-blocks. The 128-bit key is then rotationally shifted left by 25 bits, after which the resulting 128-bit block is again partitioned to produce another eight 16-bit key sub-blocks. This continues until all the sub-keys are generated.

The 16-bit sub-keys are denoted in the datapath and block diagram by **Z**i, where i = 1,…,6 (i.e., six sub-keys are used per standard round). Since only four 16-bit key sub-blocks are required for the Output Transformation round, a total of 52 (= 8·6 + 4) sub-keys have to be generated from the original 128-bit key. This infers that seven rotational shifts were performed on the original key.

All encryption sub-keys required per round can be generated per clock cycle.

| **Encryption key sub-blocks** | |
|---|---|
| Round 1 | $Z_1^{(1)} Z_2^{(1)} Z_3^{(1)} Z_4^{(1)} Z_5^{(1)} Z_6^{(1)}$ |
| Round 2 | $Z_1^{(2)} Z_2^{(2)} Z_3^{(2)} Z_4^{(2)} Z_5^{(2)} Z_6^{(2)}$ |
| Round 3 | $Z_1^{(3)} Z_2^{(3)} Z_3^{(3)} Z_4^{(3)} Z_5^{(3)} Z_6^{(3)}$ |
| Round 4 | $Z_1^{(4)} Z_2^{(4)} Z_3^{(4)} Z_4^{(4)} Z_5^{(4)} Z_6^{(4)}$ |
| Round 5 | $Z_1^{(5)} Z_2^{(5)} Z_3^{(5)} Z_4^{(5)} Z_5^{(5)} Z_6^{(5)}$ |
| Round 6 | $Z_1^{(6)} Z_2^{(6)} Z_3^{(6)} Z_4^{(6)} Z_5^{(6)} Z_6^{(6)}$ |
| Round 7 | $Z_1^{(7)} Z_2^{(7)} Z_3^{(7)} Z_4^{(7)} Z_5^{(7)} Z_6^{(7)}$ |
| Round 8 | $Z_1^{(8)} Z_2^{(8)} Z_3^{(8)} Z_4^{(8)} Z_5^{(8)} Z_6^{(8)}$ |
| Output Transform | $Z_1^{(9)} Z_2^{(9)} Z_3^{(9)} Z_4^{(9)}$ |

## IV. DECRYPTION KEY SCHEDULER

The same 52 key sub-blocks generated for encryption are rearranged and inverted accordingly to produce the decryption key schedule. Addition sub-keys are inverted by negation. Multiplication sub-keys are inverted using the modular inverter.

Decryption sub-keys required per round cannot be generated per clock cycle. This is due to the modular inversion process. Hence, for decryption, the IDEA Core will stall until all the decryption sub-keys are generated for that round.

| **Decryption key sub-blocks** | |
|---|---|
| Round 1 | $Z_1^{(9)-1} Z_2^{(9)-1} Z_3^{(9)-1} Z_4^{(9)-1} Z_5^{(8)} Z_6^{(8)}$ |
| Round 2 | $Z_1^{(8)-1} Z_3^{(8)-1} Z_2^{(8)-1} Z_4^{(8)-1} Z_5^{(7)} Z_6^{(7)}$ |
| Round 3 | $Z_1^{(7)-1} Z_3^{(7)-1} Z_2^{(7)-1} Z_4^{(7)-1} Z_5^{(6)} Z_6^{(6)}$ |
| Round 4 | $Z_1^{(6)-1} Z_3^{(6)-1} Z_2^{(6)-1} Z_4^{(6)-1} Z_5^{(5)} Z_6^{(5)}$ |
| Round 5 | $Z_1^{(5)-1} Z_3^{(5)-1} Z_2^{(5)-1} Z_4^{(5)-1} Z_5^{(4)} Z_6^{(4)}$ |
| Round 6 | $Z_1^{(4)-1} Z_3^{(4)-1} Z_2^{(4)-1} Z_4^{(4)-1} Z_5^{(3)} Z_6^{(3)}$ |
| Round 7 | $Z_1^{(3)-1} Z_3^{(3)-1} Z_2^{(3)-1} Z_4^{(3)-1} Z_5^{(2)} Z_6^{(2)}$ |
| Round 8 | $Z_1^{(2)-1} Z_3^{(2)-1} Z_2^{(2)-1} Z_4^{(2)-1} Z_5^{(1)} Z_6^{(1)}$ |
| Output Transform | $Z_1^{(1)-1} Z_2^{(1)-1} Z_3^{(1)-1} Z_4^{(1)-1}$ |

## V. MODULAR MULTIPLIER

The strength of IDEA lies in this combinational module of multiplication modulo 2^16+1 (or mod 65537). This module is part of the IDEA Core. There are as many as four multipliers in the datapath. It is important to note that a 16-bit sub-block consisting of all 0 bits (i.e., 0x0000) is not interpreted in its total value as zero but is treated as 2^16 = 0x10000.

The multiplier performs the calculation by first multiplying [w = x·y] two 17-bit vectors (possible value for each vector: 1 to 2^16, no zero) together to produce a 34-bit vector that is stored in two registers: 17-bit register for [w / 2^16] and 16-bit register for [w mod 2^16] by truncating the MSB.

In order to account for converting modulo 2^16 to modulo 2^16+1, the product exhibits a very interesting property: Any bits that appear past 2^16 for w will contribute to subtracting [w mod 2^16] to produce the result. Hence, the result would be [w mod 2^16] - [w / 2^16] + b where b is the correction bit that is determined from a negative subtraction result.

The following MATLAB code illustrates that property:

```
['0x00000FFFF mod 2^16+1 = ',dec2hex(mod(hex2dec('00000FFFF'),2^16+1))]
['0x00001FFFF mod 2^16+1 = ',dec2hex(mod(hex2dec('00001FFFF'),2^16+1))]
['0x00002FFFF mod 2^16+1 = ',dec2hex(mod(hex2dec('00002FFFF'),2^16+1))]
['0x0FFFF0000 mod 2^16+1 = ',dec2hex(mod(hex2dec('0FFFF0000'),2^16+1))]
['0x0FFFF0001 mod 2^16+1 = ',dec2hex(mod(hex2dec('0FFFF0001'),2^16+1))]
['0x0FFFFFFFF mod 2^16+1 = ',dec2hex(mod(hex2dec('0FFFFFFFF'),2^16+1))]
% resulted from multiplying 0x10000 with 0x10000 (two "zero" 16 bits)
['0x100000000 mod 2^16+1 = ',dec2hex(mod(hex2dec('100000000'),2^16+1))]
```

## VI. MODULAR INVERTER

This module finds the multiplicative inverse of a sub-key mod 2^16+1. The inverse always exists since 2^16+1 is relatively prime for all numbers 1 to 2^16. This module is part of the decryption key scheduler. Like the modular multiplier, a 16-bit sub-block consisting of all 0 bits (i.e., 0x0000) is not interpreted in its total value as zero but is treated as 2^16 = 0x10000.

This multiplicative inverter is essentially a modular divider based on the extended binary Euclidean GCD plus-minus algorithm presented by Naofumi Takagi in his paper "A VLSI Algorithm for Modular Division Based on the Binary GCD Algorithm".

The algorithm is laid out in this MATLAB code:

```
function f = inverse(a);
m = 2^16+1;
b = m;
u = 1;
v = 0;
d = 0;
while (a ~= 0)
    while (mod(a,2) == 0)
        a = floor(a/2);
        if (mod(u,2) == 0)
            u = floor(u/2);
        else
            u = floor((u+m)/2);
        end
        d = d-1;
    end
    if (d < 0)
        t = a; a = b; b = t;
        t = u; u = v; v = t;
        d = -d;
    end
    if (mod(a+b,4) == 0)
        a = floor((a+b)/2);
        if (mod(u+v,2) == 0)
            u = floor((u+v)/2);
        else
            u = floor((u+v-m)/2);
        end
    else
        a = floor((a-b)/2);
        if (mod(u-v,2) == 0)
            u = floor((u-v)/2);
        else
            u = floor((u-v+m)/2);
        end
    end
end
if (b < 0)
    f = m-v;
else
    f = v;
end
```

## VII. TESTING AND CONCLUSION

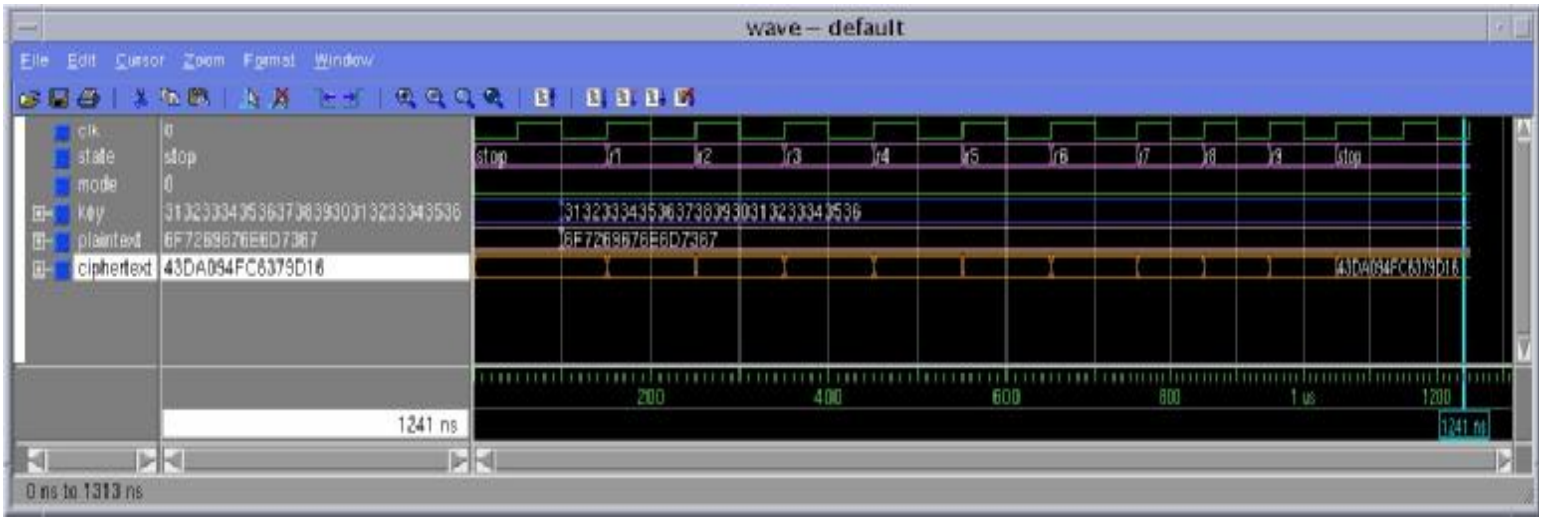
The entire implementation was simulated using Mentor Graphics VSIM tool. The same test vectors used by Irwin Yoon in his C++ implementation of IDEA were tested with this design.

Test vectors: key = 0x31323334353637383930313233343536 (1234567890123456)

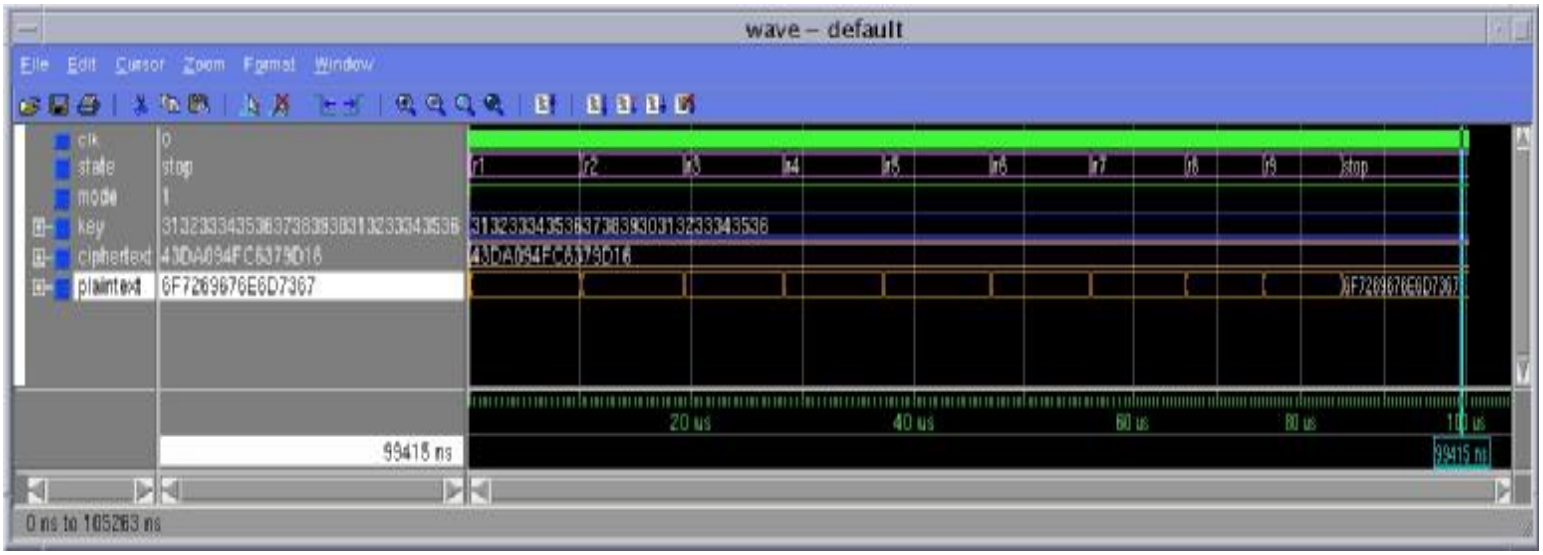
plaintext = 0x6F7269676E6D7367 (orignmsg)

All results were in sync with Irwin's and are shown to be correct. Below are snapshots of the wave simulation window:

**Encryption Results (http://web.engr.oregonstate.edu/~laige/ece575/encryption.jpg)**




**Decryption Results (http://web.engr.oregonstate.edu/~laige/ece575/decryption.jpg)**




There are a couple of optimization issues with this design. For one, the modular multiplier takes up a large amount of area. Optimizations of fast multipliers could be applied in order to reduce the area if necessary. Also, the modular inverter was implemented without carry-

save form. Further delay can be reduced if the inverter was implemented with fast carry-save adders, allowing it to be clocked at a much higher rate.

As observed from the snapshots above, the execution time for the decryption process is approximately a 100 times longer than the encryption process. Given the fact that decryptions are done more often than encryptions, it would be much more feasible to decrypt using the IDEA encryption function. For IDEA, the encryption and decryption functions are interchangeable.

## REFERENCES


[1] MediaCrypt: IDEA Technical Description <http://web.engr.oregonstate.edu/~laige/ece575/IDEA%20Technical%20Description_0503.pdf>

[2] Irwin Yoon's ECE575 Project - IDEA Algorithm <http://islab.oregonstate.edu/koc/ece575/03Project/Yoon/>

[3] Takagi N., *A VLSI Algorithm for Modular Division Based on the Binary GCD Algorithm*, IEICE TRANS. FUNDAMENTALS, VOL. E81-A No. 5, May 1998.