System Specification of the International Data Encryption Algorithm using SystemC

Gregory Bird, John Matthews

10-28-2002

1 Introduction

The goal of this assignment is to write a system specification for the International Data Encryption Algorithm (IDEA) using the SystemC specification language. A Test Bench will also be designed to interface with the IDEA module in order to verify the design's functionality. The implementation of the IDEA algorithm will be as modular as possible to allow for easier hardware/software partitioning later on.

The paper is organized as follows: in section 2 we will discuss the functionality of the IDEA algorithm, in section 3 we will discuss the organization of the specification, data structures, and algorithms, in 4 we will give a description of the Test Bench, in section 5 we will discuss the simulation results, and in section 6 we will give our conclusions.

2 Description of the IDEA Module Functionality

The International Data Encryption Algorithm (IDEA) was first created in 1990 by Xuejia Lai and James Massey. It was originally called PES (Proposed Encryption Standard). The PES algorithm was improved a year later to strengthen the cipher against the newly discovered differential cryptoanalysis attack. This new version of the PES was renamed Improved-PES (IPES) and then renamed again in 1992 to IDEA.[1]

The IDEA algorithm is considered to be a very strong block cipher. It works on 64-bit plaintext blocks and uses a 128-bit secret key. The same algorithm is used to encrypt and decrypt the data, which means that it is a symmetric encryption algorithm.

The algorithm uses *confusion* and *diffusion* to encrypt its data, and combines *operations from three different algebraic families*: the XOR logic function, 2^{16} modulo addition, and $2^{16}+1$ modulo multiplication. These operations can

Figure 1: Secret Key

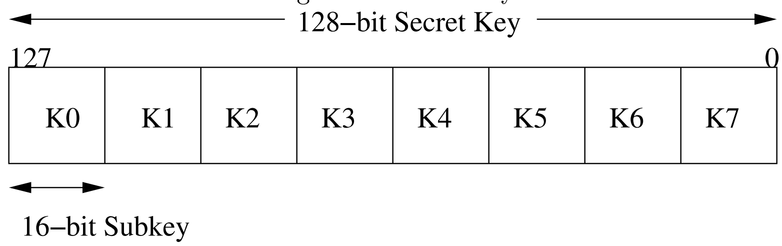


Table 1: Subkey Schedule

Round	Encryption Subkeys	Decryption Subkeys
Round 1	$Z_1^{(1)}, Z_2^{(1)}, Z_3^{(1)}, Z_4^{(1)}, Z_5^{(1)}, Z_6^{(1)}$	$Z_1^{(9)-1}, -Z_2^{(9)}, -Z_3^{(9)}, Z_4^{(9)-1}, Z_5^{(8)}, Z_6^{(8)}$
Round 2	$Z_1^{(2)}, Z_2^{(2)}, Z_3^{(2)}, Z_4^{(2)}, Z_5^{(2)}, Z_6^{(2)}$	$Z_1^{(8)-1}, -Z_3^{(8)}, -Z_2^{(8)}, Z_4^{(8)-1}, Z_5^{(7)}, Z_6^{(7)}$
Round 3	$Z_1^{(3)}, Z_2^{(3)}, Z_3^{(3)}, Z_4^{(3)}, Z_5^{(3)}, Z_6^{(3)}$	$Z_1^{(7)-1}, -Z_3^{(7)}, -Z_2^{(7)}, Z_4^{(7)-1}, Z_5^{(6)}, Z_6^{(6)}$
Round 4	$Z_1^{(4)}, Z_2^{(4)}, Z_3^{(4)}, Z_4^{(4)}, Z_5^{(4)}, Z_6^{(4)}$	$Z_1^{(6)-1}, -Z_3^{(6)}, -Z_2^{(6)}, Z_4^{(6)-1}, Z_5^{(5)}, Z_6^{(5)}$
Round 5	$Z_1^{(5)}, Z_2^{(5)}, Z_3^{(5)}, Z_4^{(5)}, Z_5^{(5)}, Z_6^{(5)}$	$Z_1^{(5)-1}, -Z_3^{(5)}, -Z_2^{(5)}, Z_4^{(5)-1}, Z_5^{(4)}, Z_6^{(4)}$
Round 6	$Z_1^{(6)}, Z_2^{(6)}, Z_3^{(6)}, Z_4^{(6)}, Z_5^{(6)}, Z_6^{(6)}$	$Z_1^{(4)-1}, -Z_3^{(4)}, -Z_2^{(4)}, Z_4^{(4)-1}, Z_5^{(3)}, Z_6^{(3)}$
Round 7	$Z_1^{(7)}, Z_2^{(7)}, Z_3^{(7)}, Z_4^{(7)}, Z_5^{(7)}, Z_6^{(7)}$	$Z_1^{(3)-1}, -Z_3^{(3)}, -Z_2^{(3)}, Z_4^{(3)-1}, Z_5^{(2)}, Z_6^{(2)}$
Round 8	$Z_1^{(8)}, Z_2^{(8)}, Z_3^{(8)}, Z_4^{(8)}, Z_5^{(8)}, Z_6^{(8)}$	$Z_1^{(2)-1}, -Z_3^{(2)}, -Z_2^{(2)}, Z_4^{(2)-1}, Z_5^{(1)}, Z_6^{(1)}$
Output	$Z_1^{(9)}, Z_2^{(9)}, Z_3^{(9)}, Z_4^{(9)}$	$Z_1^{(1)-1}, -Z_2^{(1)}, -Z_3^{(1)}, Z_4^{(1)-1}$

easily be implemented in hardware or software, and the fact that they are 16-bit operations means that they could run efficiently on a small 16-bit embedded platform.[1]

The 128-bit secret key is used to create 52 16-bit encryption, or decryption, subkeys. This is done by splitting the 128-bit secret key into 8 16-bit subkeys, rotating the 128-bit secret key 25 places to the left, and splitting it again. This is repeated until you obtain the 52 subkeys needed for the algorithm. Figure 1 shows the 128-bit secret key being split into the first eight subkeys. The decrypt subkeys are generated from the encrypt subkeys by using the schedule shown in Table 1.

The IDEA algorithm is broken into eight rounds of operations and an output transformation stage. Each round uses 6 subkeys in its operations. The subkey schedule is explained in Table 1. Figure 2 shows the data flow of the plaintext through the round operations. The 64-bit plaintext block is split into 4 16-bit chunks which are passed through the rounds and output transformation and then re-assembled into a 64-bit block at the other side.

Using the same secret key, and algorithm, the IDEA module will be able to encrypt plaintext and decrypt ciphertext. The Test Bench is designed to interface with the IDEA module. It will provide a clock, the secret key, and data to the IDEA module and it will collect the output of the IDEA module.

Figure 2: IDEA Data Flow Graph

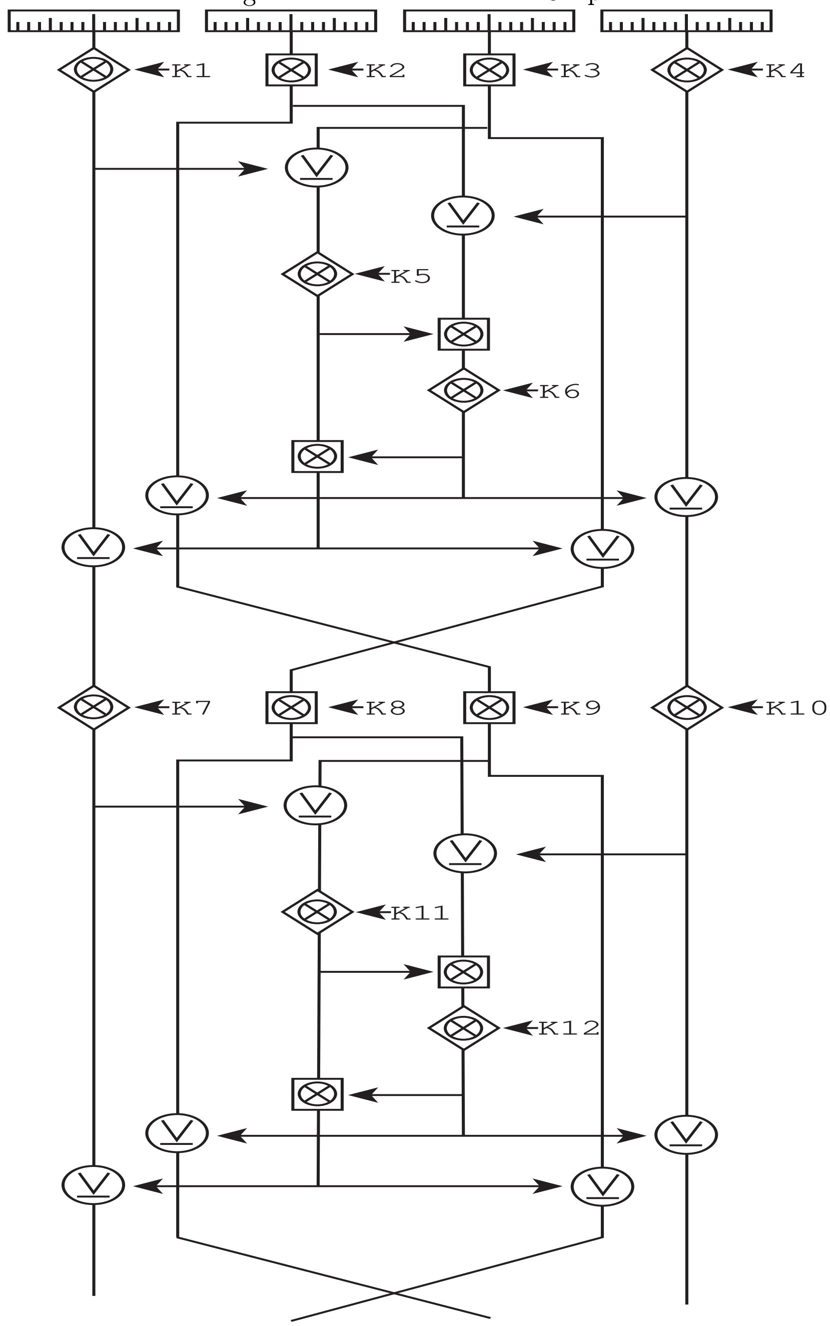
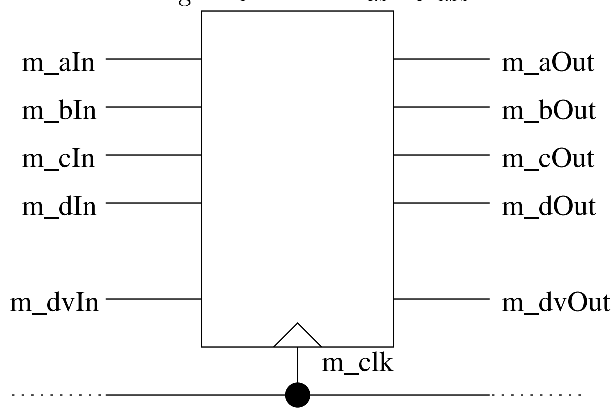


Figure 3: IDEA Base Class



3 Organization of the Specification, Data Structures, and Algorithms

3.1 Data Types for IDEA

Using the *typedef* operator, a few new data types were created to make the code a little clearer. sc_biguint<128> was *typedef* ed to IdeaKey. This was used when ever a secret key or temp variables in producing the subkeys was needed. sc_uint<16> was *typedef* ed to IdeaShort. This was used for all of the 16-bit data values that are passed through each module. sc_logic was *typedef* ed to DV_Type. This was used for the Data Valid line, which is used as flow control for each stage's mathematic operations and for the Test Bench's output.

3.2 IDEA Base Class

The IDEA base class has the structure shown in Figure 3. It provides the basic input/output channels and can be inherited from all of the other modules in the IDEA design. The m_aIn, m _bIn, m_cIn, and m_dIn signals are of type sc_in<IdeaShort> , which is a 16-bit unsigned integer input port and is used to pass data values into the module. The m_aOut, m_bOut, m_cOut and m_dOut signals are of type sc_out<IdeaShort, which is a 16-bit unsigned integer output port and is used to output data values from the module. The module also has input and output ports for the Data Valid line of type sc_in<DV_Type> and sc_out<DV_Type> respectively, and a clock signal, m_clk, of type sc_in_clk.

Each stage and round, and the IDEA module itself, inherit from this base class. This makes for a very clean Object Oriented design and it makes it easier to manage the code when the need for change occurs. Figure 4 shows the class inheritance graph.

Figure 4: Class Inheritance Tree

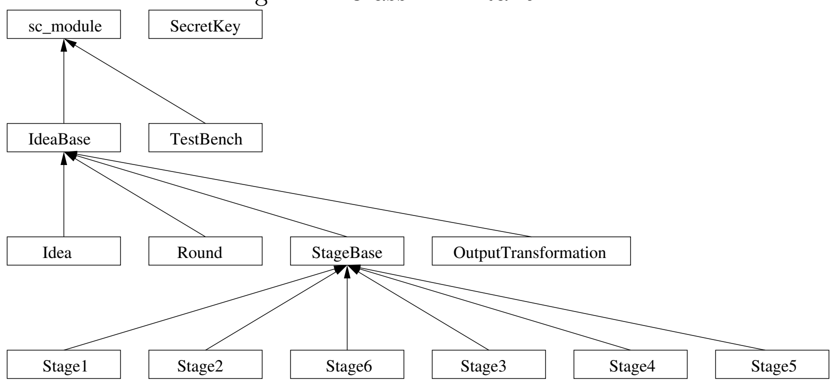
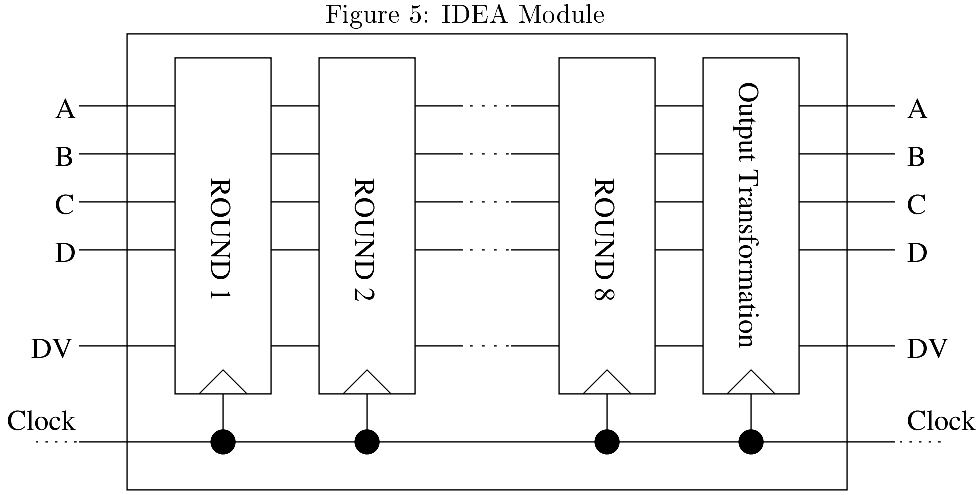


Figure 5: IDEA Module



3.3 IDEA Module

The IDEA module is basically a container for eight Round modules and one OutputTransformation module. It inherits from the IdeaBase class, which inherits from sc_module. Its constructor simply uses sc_signals to connect the Rounds and the OutputTransformation together as shown in Figure 5. The Idea module has one SC_METHOD, called doIdea(), that is sensitive to the positive edge of the clock.

3.4 Round Module

Each Round module is a container for the six different Stage modules. It inherits from the IdeaBase class, which inherits from sc_module. Its constructor simply uses sc_signals to connect the six different Stage modules together as shown in

Figure 6: Round Module

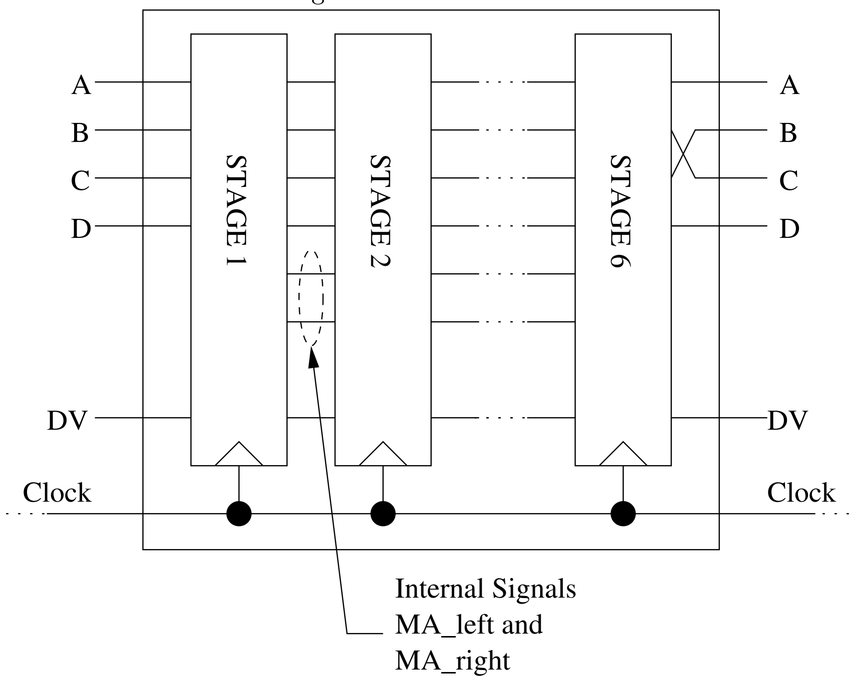


Figure 6. The Round module has one SC_METHOD, called doRound(), that is sensitive to the positive edge of the clock.

3.5 Stages

The IDEA algorithm is broken in to 6 different parts that are implemented as six different Stage modules. The operations that are present in each Stage were chosen so that there were no data dependencies within the Stage. This allows for maximum modularity in the design and it allows us to make the clock cycle period equal to the slowest Stage and make each Stage's computation last one clock cycle. With this design the Round module acts like a pipeline and, once full, will output a 64-bit block every clock cycle.

Each Stage inherits from a StageBase class, see Figure 4, which inherits from IdeaBase and sc_module. StageBase adds two more signals, MA_left and MA_right, that are used in the IDEA algorithm, see Figure 6. Stage 6 swaps the output of channels B and C as called out in the IDEA algorithm. The StageBase class has one SC_METHOD, called stage(), that is sensitive to the positive edge of the clock. It also contains a purely virtual method called doIt() that each Stage implements differently according to the functionality that it is trying to perform.

Four of the subkeys that are created from the 128-bit secret key are used in

Stage 1, one subkey is used in Stage 2, and one subkey in Stage 4.

3.6 Output Transformation

The OutputTransformation module inherits from IdeaBase class, which inherits from sc_module. Its purpose is to perform the final transformation on the data before it leaves the IDEA module. It has one SC_METHOD, called doIt(), that is sensitive to the positive edge of the clock. It uses four subkeys to perform its operation.

3.7 Secret Key

The SecretKey class was implemented as a purely C++ class with no inheritance from SystemC classes. It is basically a library for the math routines needed to create the 52 encrypt and decrypt subkeys from the 128-bit secret key. It uses two arrays to store the arrays, and four methods called generate_encrypt(), generate_decrypt(), shift(), and mulInv() to compute the subkeys.

4 Description of the Test Bench

The Test Bench module inherits only from sc_module and is responsible for parsing the command line arguments, creating the subkeys, and passing data through the IDEA module. Figure 7 shows how the Test Bench interacts with the IDEA module and the SecretKey class.

The TestBench starts by reading in the Secret Key, number of clock cycles to run, and input/output filenames from the command line. It then uses the SecretKey class to generate the subkeys and inserts them into the IDEA module via the insertKeys() method of the IDEA class.

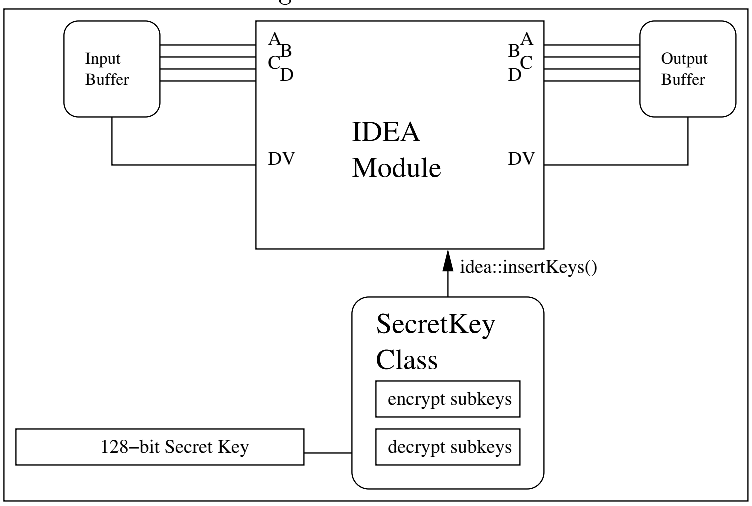
The TestBench then reads the input file to a temporary buffer, and passes 64-bits at a time through the IDEA module. Every time it hands the IDEA module a block it sets the Data Valid line to a logic one. The Data Valid signal travels with the data signaling each module that the current block is valid data and should be operated on.

The Test Bench module monitors the output of the IDEA module, and anytime it sees the Data Valid line set high it write the data block to the output buffer. When the User specified number of clock cycles are up, the Test Bench module writes the output buffer data to the output file.

5 Simulation Results

Running the TestBench module revealed the results that we expected, namely that the Idea module was able to encrypt plaintext into cipher text and decrypt the ciphertext back into the original plaintext. It takes 67 clock cycles before we see output through our pipeline, but then we see output every clock cycle after that. Also, if the input file size is not a multiple of 8-bytes, then we needed

Figure 7: Test Bench



to pad the input data to fill in the block. We chose 0x00 for the fill character. The output ciphertext file and the output plaintext (after decryption) file will be slightly larger than the original file due to these fill characters.

Here is a sample run through the encryption and decryption steps. Below is the input file *FOX*, which contains the text "The quick brown fox jumped over the lazy dog!!!!", the file *FOX.enc*, which contains the cipher text, and the file *FOX.dec*, which contains the decrypted cipher text. Notice that since the input file size is a multiple of 8-bytes the file size doesn't increase.[1]

```
--- FOX : 48 bytes ---------------------------------------------------
The quick brown fox jumped over the lazy dog!!!!
----------------------------------------------------------------------


---FOX.enc :  48 bytes -----------------------------------------------
M-^FM-.nhM-^@M-VnM-7M-3%hM-a^.^Ls%M-KM-!M-i^VEdM-T
=YM-U^W)^D>HeOpfM-^VM-mM-2[^D^GM-GM-hM-AM-^SXq
----------------------------------------------------------------------


--- FOX.dec :  48 bytes -----------------------------------------------
The quick brown fox jumped over the lazy dog!!!!
----------------------------------------------------------------------
```

[1]The command 'cat -vte FOX.enc' was used to output the contents of the binary encoded FOX.enc file. This command allows you to see the binary characters by using escaped character sequences. This makes the output appear to have more than 48-bytes, but it really only contains 48 binary characters.

6 Conclusions

The IDEA algorithm is a nice clean encryption algorithm that was relatively easy to implement in SystemC. Since it is not that large we were able to unroll the only loop in the algorithm (the 8 rounds) and create a streamlined pipeline implementation. Given more time, we would have liked to learn about SystemC's sc_trace features and the gtkwave program. This would have given us the ability to display waveforms similar to a VHDL simulation.

References

[1] The *International Data Encryption Algorithm* - B. Schneier, *Other Block Ciphers*, Applied Cryptography, pp. 319-325, John Wiley, 1996.

[2] *IDEA (International Data Encryption Algorithm)* - http://home.ecn.ab.ca/~jsavard/crypto/co0404.htm

[3] *The Block Cipher IDEA* - Chapter 3 - http://www.zone-h.com