# Implementation of Morse Code Decoder in Verilog with RSA Encryption and Decryption

Jennifer Xiong
*California State University, Fresno*
Fresno, CA
jenerex@mail.fresnostate.edu

Robert Wong
*California State University, Fresno*
Fresno, CA
robertwong@mail.fresnostate.edu

***Abstract*—Morse code is a communication technique that was originally used for the telegraph, but can be applied to a simple input signal. A high value would represent on units and a low value would represent off units to communicate Morse code. RSA encryption and decryption allows for secure communication of a certain degree with limited message lengths. This project combines both concepts into a hardware level implementation based on Verilog for a complete Morse code decoder, RSA key generator, encryption, and decryption.**

## I. Introduction

The objective of this project was to create a Morse code decoder and RSA encryption and decryption system. The project was designed using Verilog and synthesized, then the final result was reported on. The work for the project was divided between two teammates. Jennifer worked on the code for the RSA key generation and the final synthesis of all of the RTL design files. Robert worked on the code for the Morse code to ASCII translator and the top module design that controlled all of the sub modules. Both worked on the block and state diagrams, the implementation for RSA encryption and decryption and on the presentation and final report.

## II. Background

Morse code is based off of high and low signals for early electronic communication, i.e. via the telegraph. A high transmitted value (logical 1) would represent an on part of the signal and a low transmitted value (logical 0) would represent an off part of the signal. A dot can be represented by an on signal for one time unit and a dash can be represented by an on signal for three consecutive time units. Each of these individual dots and dashes are separated by a single off time unit to separate them. A combination of dots and dashes creates an alphanumeric character as demonstrated in Figure 1. Letters can be separated by three consecutive time units of an off signal, and a space between words can be represented as seven consecutive units of an off signal.

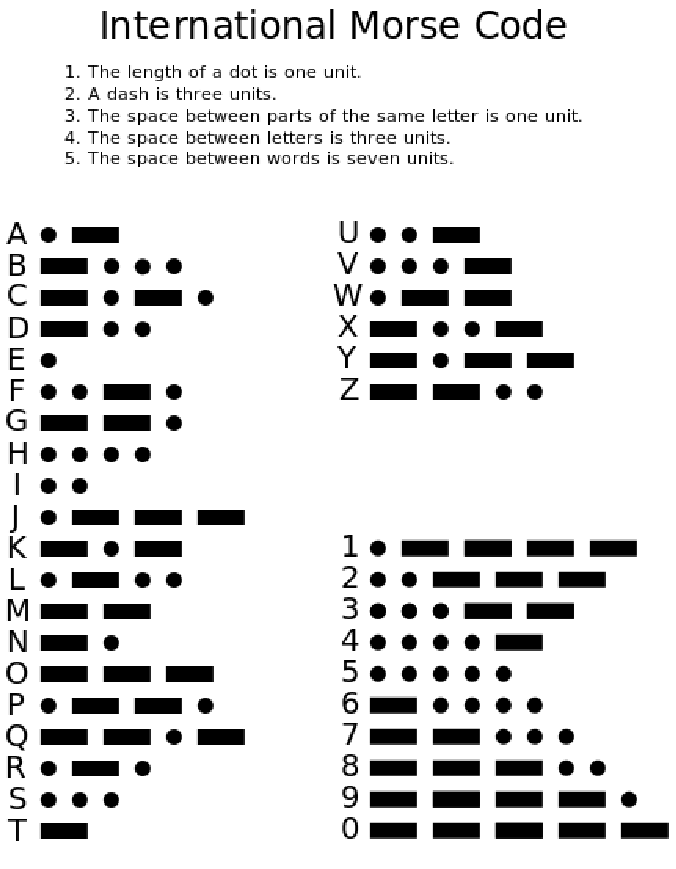
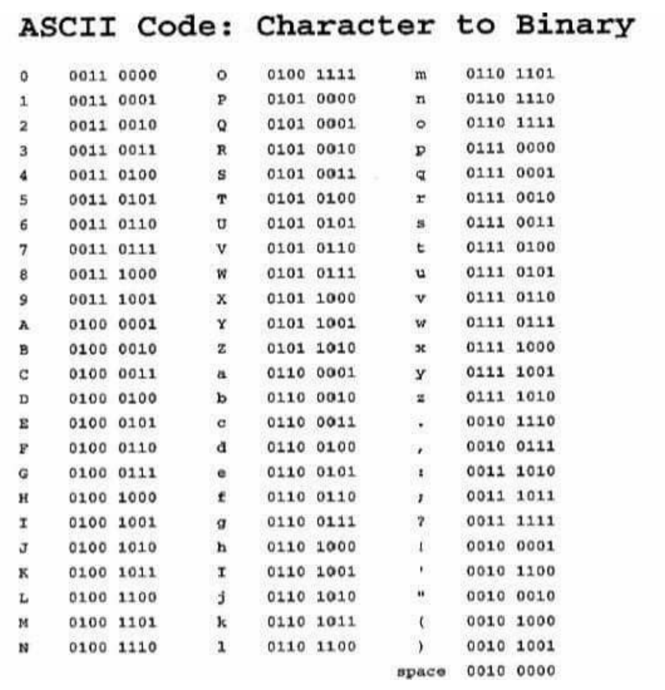

International Morse Code
1. The length of a dot is one unit.
2. A dash is three units.
3. The space between parts of the same letter is one unit.
4. The space between letters is three units.
5. The space between words is seven units.
A B C D E F G H I J K L M N O P Q R S T
U V W X Y Z
1 2 3 4 5 6 7 8 9 0

Fig. 1. Summary of Morse Code translation taken from [1].

Another way to represent alphanumeric characters is using the American Standard Code for Information Exchange (ASCII). ASCII is a standard character encoding pattern for communicating text-based messages using computers [2]. Each character is represented as a 7-bit binary number as illustrated in Figure 2. While ASCII designates values for many different types of standard characters, only the values for alphanumeric characters and a space were used for this implementation.



**ASCII Code: Character to Binary**

| | | | | | |
|---|---|---|---|---|---|
| 0 | 0011 0000 | O | 0100 1111 | m | 0110 1101 |
| 1 | 0011 0001 | P | 0101 0000 | n | 0110 1110 |
| 2 | 0011 0010 | Q | 0101 0001 | o | 0110 1111 |
| 3 | 0011 0011 | R | 0101 0010 | p | 0111 0000 |
| 4 | 0011 0100 | S | 0101 0011 | q | 0111 0001 |
| 5 | 0011 0101 | T | 0101 0100 | r | 0111 0010 |
| 6 | 0011 0110 | U | 0101 0101 | s | 0111 0011 |
| 7 | 0011 0111 | V | 0101 0110 | t | 0111 0100 |
| 8 | 0011 1000 | W | 0101 0111 | u | 0111 0101 |
| 9 | 0011 1001 | X | 0101 1000 | v | 0111 0110 |
| A | 0100 0001 | Y | 0101 1001 | w | 0111 0111 |
| B | 0100 0010 | Z | 0101 1010 | x | 0111 1000 |
| C | 0100 0011 | a | 0110 0001 | y | 0111 1001 |
| D | 0100 0100 | b | 0110 0010 | z | 0111 1010 |
| E | 0100 0101 | c | 0110 0011 | . | 0010 1110 |
| F | 0100 0110 | d | 0110 0100 | , | 0010 0111 |
| G | 0100 0111 | e | 0110 0101 | : | 0011 1010 |
| H | 0100 1000 | f | 0110 0110 | ; | 0011 1011 |
| I | 0100 1001 | g | 0110 0111 | ? | 0011 1111 |
| J | 0100 1010 | h | 0110 1000 | ! | 0010 0001 |
| K | 0100 1011 | I | 0110 1001 | ' | 0010 1100 |
| L | 0100 1100 | j | 0110 1010 | " | 0010 0010 |
| M | 0100 1101 | k | 0110 1011 | ( | 0010 1000 |
| N | 0100 1110 | l | 0110 1100 | ) | 0010 1001 |
| | | | | space | 0010 0000 |

Fig. 2. Summary of ASCII translation taken from [3].

## III. SPECIFICATION

The design requires two 32-bit prime numbers, $p$ and $q$, for key generation. Based on the algorithm discussed later, this creates a 128-bit key (64 bits for n and 64 bits for e/d). This allows for a message that is less than 64 bits to be encrypted and decrypted, and since each ASCII is 7 bits, we opted for 56-bit messages, or an input of 8 characters through Morse code. The device also requires a clock and a reset signal to reset the device for the next message. As a result, the device will output a 64-bit value, which is the 56-bit message originally sent in Morse code.

The number of clock cycles it takes for the entire system to operate and output the result varies because the length of the input morse code varies as discussed previously.

## IV. DESIGN

The design for the Morse code to ASCII decoder was somewhat complex. Originally, the idea was to use a lookup table to translate input of the Morse code directly into ASCII. However, because of the nature of Morse code communication, the values are time dependent, and each individual character can range in the amount of time it takes to be transmitted or communicated. This leads to a difficulty of having a simple translation table, and requires the use of states to check which character is being transmitted a single time unit at a time. As such, the complex state diagram was designed for the Morse code to ASCII decoder as shown in Figure 3. In the figure, the letters and numbers are used to easier illustrate for the design which state is for each number in the registers, and the letters are used to more easily follow the path for which character is being transmitted. At the end of each trail, the code output a 7-bit value which is the ASCII translation of the character. In the diagram, it is also important to note that some of the states only have one next possible state. These are indicated by the value that they should be for a correct translation of Morse code in the diagram, but for the actual implementation a don't care was used so that it didn't matter if the input was wrong and so the program would not get stuck. Finally, for state zero, the start state, a counter was also implemented to check for seven consecutive low time units to represent the output of a space character.

The block diagram in Figure 4 shows the entire design. Since the key generator and modular exponentiation required the most calculation, their specific block diagrams are displayed as well.

For the key generation, two 32-bit primes are already provided, i.e. $p$ and $q$, for generation of the key in order to verify that they are primes that are valid for key generation. If p and q were not provided, an additional component to the system would have been required to randomly generate two 32-bit numbers and validate that they are primes. Also, a value for the public key, $e$, was already selected to make the entire key generation process simpler, and because $e = 65537$ is a standard public

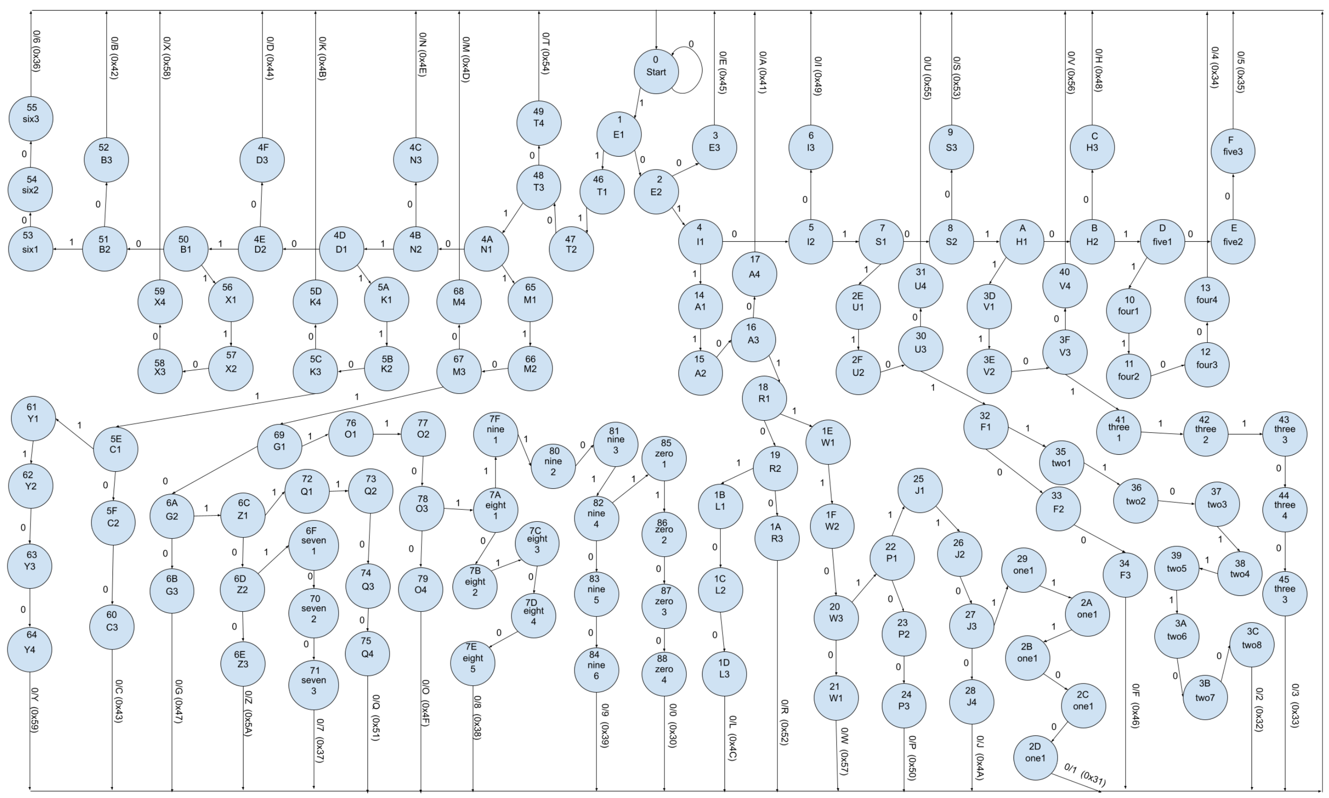


Fig. 3. State diagram for the Morse code to ASCII decoder.

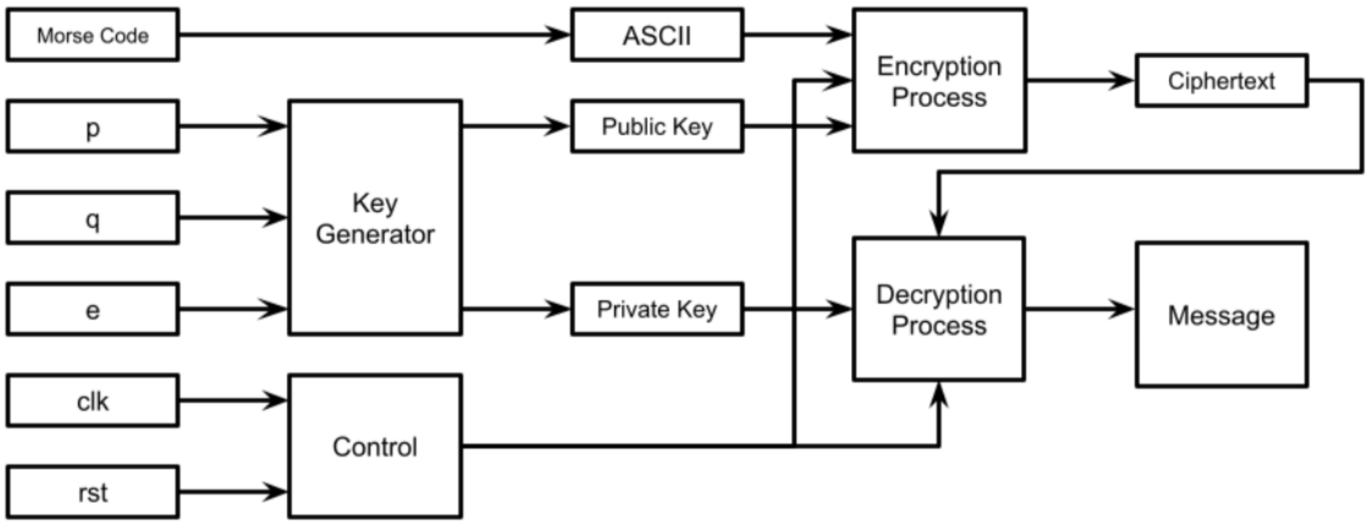


Fig. 4. Block diagram for full design.

key that is used because of the ease to use in modular exponentiation having only two bits as ones in the binary value and it being a valid prime [4]. From these inputs, only $n$ and $d$, $d$ only for the private key, have to be generated. The specifics of how $d$ is calculated is more complex and requires the Euclidean algorithm, which will be discussed further in the next section. A block diagram of how the key generation portion of the system is supposed to work is shown in Figure 5.

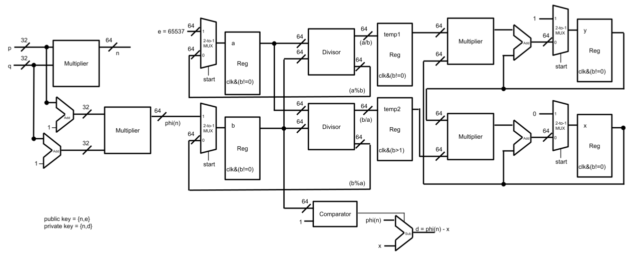


Fig. 5. Block diagram of the key generation part of the system.

Finally, the last step for RSA encryption and decryption is the actual process of encryption and decryption. This requires modular exponentiation because the equation for encryption is $ct = m^e modulo n$ and the equation for decryption is $m = ct^d modulo n$. Because a 56 to 64-bit value raised to another 64 bit value is too large for storage, it is simpler to perform modular exponentiation, a process that performs the modulo after every bit exponentiation to reduce memory usage. The exact algorithm will be discussed further in the next section. A block diagram of how the encryption and decryption portion of the system, using modular exponentiation, is supposed to work is shown in Figure 6.

## V. Implementation

The first step in implementing the design is to understand the encryption process.

For our implementation, the first step is to generate the RSA keys in order to allow whatever message is decoded to be encrypted and decrypted. This process calculated $n$ for both keys by multiplying the two primes provided, and already has a selected value for the other half of the public key, $e = 65537$. The longest part is to calculate the other half of the private key, $d$, which requires the

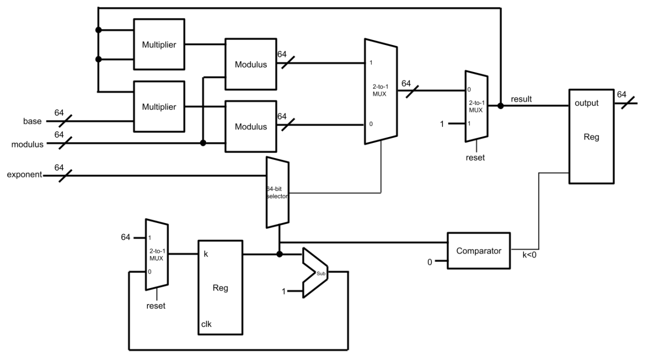


Fig. 6. Block diagram of the modular exponentiation part of the system for encryption and decryption.

Euclidean algorithm as discussed later. This process will take multiple clock cycles to compute before it is complete. At the same time, the Morse code can be decoded into ASCII because it does not require the RSA keys yet. This process goes through the state diagram discussed previously to translate the incoming Morse code waveform into ASCII. Based on the background given for Morse code and the state diagram, each character can range from requiring 4 clock cycles, E, to requiring 22 clock cycles, 0. Since there are a total of 8 characters, to translate into 7-bit ASCII each for a total of 56 bits, the time it takes to translate can range from 32 to 176 clock cycles depending on the message. Then, the resulting ASCII can be encrypted using the equation $ct = m^e \% n$, then after decrypted using the equation $m = ct^d \% n$. This process is a little more complicated than just exponentiation and modulus because each value is 64 bits and would require a very large amount of storage to compute. The algorithm to simplify this process, modular exponentiation, is discussed later in the section. Generally, the process requires a clock cycle for every bit of the exponent, which is the key and 64 bits in this case, so each of these processes will take about 64 clock cycles. Finally, the resulting $m$ from decryption is the result of the system.

In order to calculate $d$, the modular inverse of $e$ modulo $phi(n)$ must be performed, but that is not as simple as inverting $e$ and taking the modulus. To calculate $d$, the Euclidean algorithm must be used as demonstrated by [6]. The algorithm ba-

**Algorithm 1:** Complete Algorithm

```
Input: $p$ and $q$ two 32 bit primes for the key
  generation, $morse$ signal for the Morse code
Output: $m$ final message in ASCII
$\mathbf{main}(p,q,morse)$
$\{privateKey, publicKey\} = keyGen(p,q)$
$\{n,d\} = privateKey$
$\{n,e\} = publicKey$
$ascii = morse$ decoded using the state
  diagram in Figure 3
$encrypted = \mathbf{modexp}(ascii,e,n)$
$decrypted = \mathbf{modexp}(ascii,d,n)$
$m = decrypted$
$return\{m\}$
end
```

**Algorithm 2:** Key Generation Algorithm, adapted from [5]

```
Input: $p$ and $q$ primes of about same length
Output: $private$ $n$ and $d$ for the private key,
  $public$ $n$ and $e$ for the public key
$\mathbf{keyGen}(p,q)$
$e = 65537$
$n = p*q$
$phi(n) = (p-1)*(q-1)$
$d = (e^{-1})\%phi(n)$ using $\mathbf{dgen}(e, phi(n))$
$public = \{n,e\}$
$private = \{n,d\}$
$return\{public, private\}$
end
```

sically reverses the process of taking the modulus multiple times in order to find a value where $d*e = 1\%phi(n)$. A while loop is typically used to implement this algorithm until $b = 1$ or $b = 0$. However, since this is a hardware level design, a for loop can not be used, and the implementation uses branches that are evaluated every clock cycle. Each clock cycle, a step backwards in the process of the Euclidean algorithm is performed, if $b = 1$ or $b = 0$ then this part stops and outputs the correct value for $d$. A reset input is also required in order to load the initial values for the algorithm to work. The complete algorithm is shown in 3, however the actual implementation is included in the design files and not in this report.

**Algorithm 3:** d Generation Algorithm, adapted from [6]

```
Input: $e$ prime, $phi(n)$ key generation value
Output: $d$ private key component
$\mathbf{dgen}(e, phi(n))$
$a = e$
$b = phi(n)$
$x = 0$
$y = 1$
while $b! = 0$ do
    $q = int(b/a)$
    $b = b - (a*q)$
    $x = x + (q*y)$
    if $b == 1$ then
        $d = phi(n) - x$
        $return\{d\}$
    else
        $q = int(a/b)$
        $a = a - (b*q)$
        $y = y + (q*x)$
    end
end
end
```

In order to simplify the modular exponentiation to not take up too much memory for a 64-bit value raised to another 64-bit value, a modular exponentiation algorithm is used.

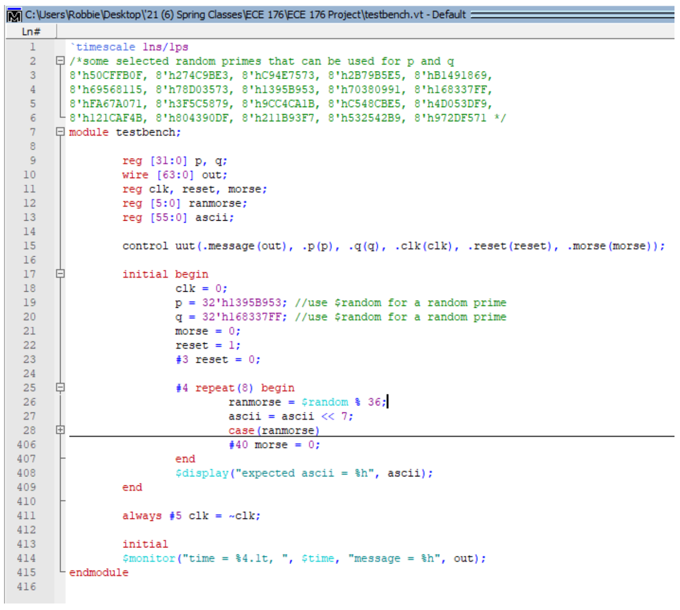
## VI. Testing

The code used in the testbench for testing the HDL code is shown in Figure 7. The testbench has selected values for $p$ and $q$ that are already verified 32-bit primes for key generation. Then the tesbench goes through a loop to randomly select 8 alphanumeric characters, a-z or 0-9. In the case statement, the contents of which are hidden, has a case for each alphanumeric character using timing to simulate the input of the selected Morse code character and saves the ASCII value of that character to the ascii variable for comparison.

Figure 8 and Figure 9 depict two tests generated for the system. In order to use a different seed for the second test, a second random command was implemented to alter the output.

**Algorithm 4:** Modular Exponentiation Algorithm, adapted from [7]

```
Input: $b$ base value, $ex$ exponent, $m$ modulo value
Output: $R$ result of performing modular exponentiation
**modexp**$(b, ex, m)$
$R = 1$
$k = length\_of\_ex\_in\_bits$
**while** $k > 0$ **do**
    **if** $ex[k] == 1$ **then**
        $R = (R * b)\%m$
    **end**
    $R = (R^2)\%m$
    $k = k - 1$
**end**
**if** $ex[k] == 1$ **then**
    $R = (R * b)\%m$
**end**
$return\{R\}$
**end**
```



```
C:\Users\Robbie\Desktop\'21 (6) Spring Classes\ECE 176\ECE 176 Project\testbench.vt - Default
`timescale 1ns/1ps
/*some selected random primes that can be used for p and q
8'h50CFFB0F, 8'h274C9BE3, 8'hC94E7573, 8'h2B79B5E5, 8'hB1491869,
8'h69568115, 8'h78D03573, 8'h1395B953, 8'h70380991, 8'h168337FF,
8'hFA67A071, 8'h3F5C5879, 8'h9CC4CA1B, 8'hC548CBE5, 8'h4D053DF9,
8'h121CAF4B, 8'h804390DF, 8'h211B93F7, 8'h532542B9, 8'h972DF571 */
module testbench;

        reg [31:0] p, q;
        wire [63:0] out;
        reg clk, reset, morse;
        reg [5:0] ranmorse;
        reg [55:0] ascii;

        control uut(.message(out), .p(p), .q(q), .clk(clk), .reset(reset), .morse(morse));

        initial begin
                clk = 0;
                p = 32'h1395B953; //use $random for a random prime
                q = 32'h168337FF; //use $random for a random prime
                morse = 0;
                reset = 1;
                #3 reset = 0;

                #4 repeat(8) begin
                        ranmorse = $random % 36;
                        ascii = ascii << 7;
                        case(ranmorse)
                        #40 morse = 0;
                end
                $display("expected ascii = %h", ascii);
        end

        always #5 clk = ~clk;

        initial
        $monitor("time = %4.1t, ", $time, "message = %h", out);
endmodule
```

Fig. 7. Testbench code for complete Morse code decoder and RSA encryption and decryption system.

Figure 10 and Figure 11 show the specifics for the first test with a closer view of the waveform and the testbench output in ASCII of the input value and the system output value.

Figure 12 and Figure 13 show the specifics for the second test with a closer view of the waveform and the testbench output in ASCII of the input value and the system output value.

## VII. Synthesis

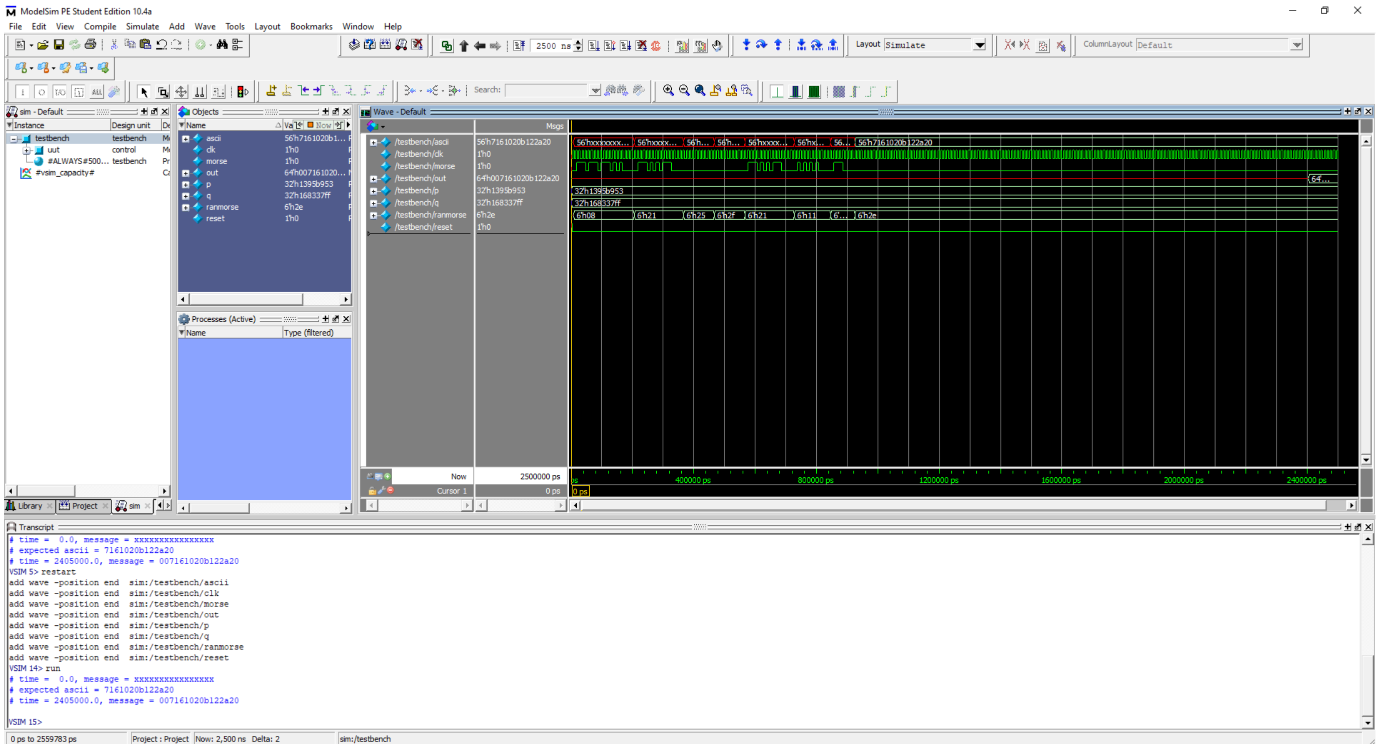
Using Synopsys, an EDA tool used for silicon design and verification, the synthesis took approximately three hours. Figure 14 shows the area report of the combined Verilog files. The area report gives details of the number of cells, ports, nets and other information in regards to area.

The timing report was cropped to show the data required time, data arrival time and the slack. It can be seen the slack has been violated. This information can be seen in Figure 15.

The power report shows an estimated cell internal power and net switching power, among other things. Figure 16 provides more information on the power report.

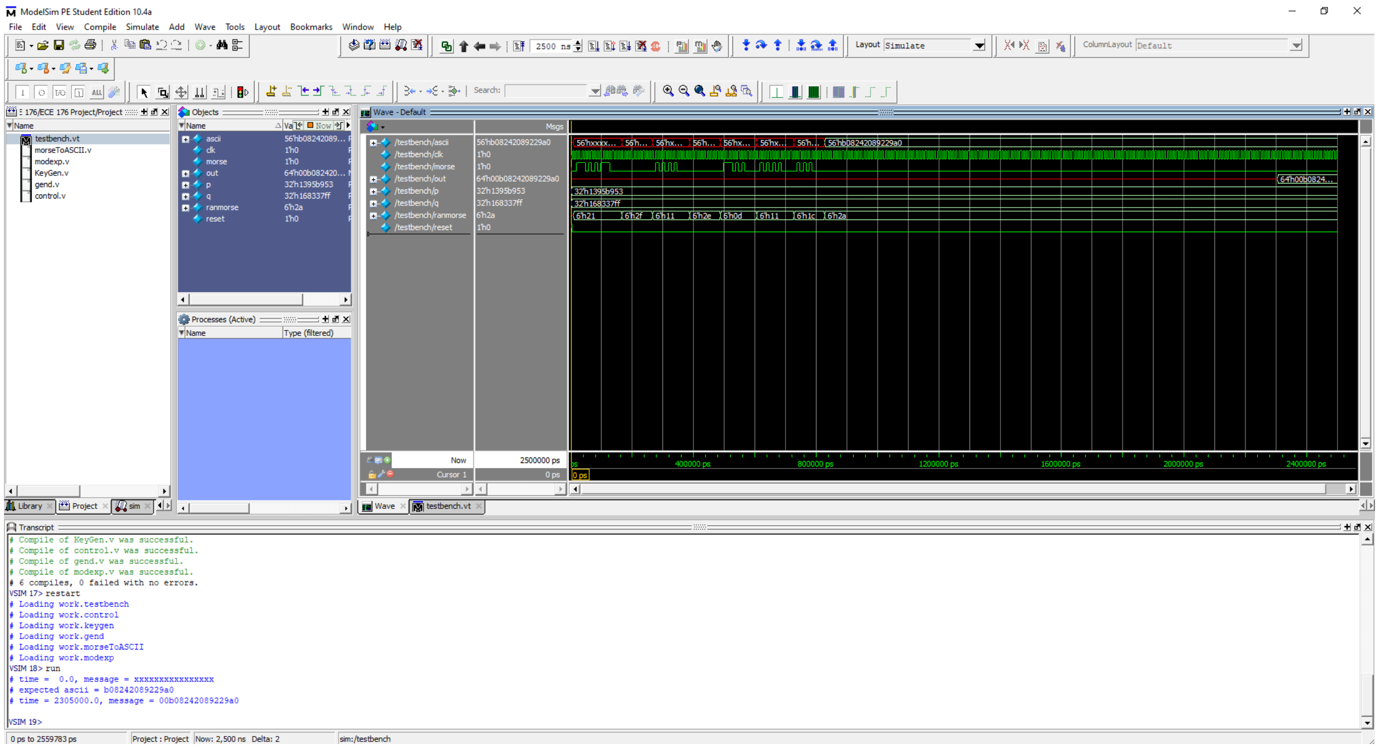
## VIII. Conclusion

In implementing a public key cryptosystem, RSA allows for secure information to be transmitted using the complex method of calculating large prime numbers. Although multiplying two large prime numbers is not considered a secure process, the security relies on the complexity of factoring the product to obtain a result usable to decrypt the public key. Understanding Morse code is a challenge in itself to understand, and the idea behind implementing a Morse code cryptosystem using RSA algorithm was to provide a challenge in which Verilog could be used to demonstrate a working design. The steps throughout the report shows the process needed to duplicate the design, and using the lessons taught throughout the course of the semester, the report follows the steps of design, RTL code, synthesis and simulation. Understanding how to proceed with each step is important to how the final design will be implemented. More importantly, the experience gained from this project serves as an initial step to understanding hardware design.



```
# time =  0.0, message = xxxxxxxxxxxxxxxx
# expected ascii = 7161020b122a20
# time = 2405000.0, message = 007161020b122a20
VSIM 5> restart
add wave -position end  sim:/testbench/ascii
add wave -position end  sim:/testbench/clk
add wave -position end  sim:/testbench/morse
add wave -position end  sim:/testbench/out
add wave -position end  sim:/testbench/p
add wave -position end  sim:/testbench/q
add wave -position end  sim:/testbench/ranmorse
add wave -position end  sim:/testbench/reset
VSIM 14> run
# time =  0.0, message = xxxxxxxxxxxxxxxx
# expected ascii = 7161020b122a20
# time = 2405000.0, message = 007161020b122a20

VSIM 15>
```

Fig. 8.  First test of the designed system.



```
# Compile of KeyGen.v was successful.
# Compile of control.v was successful.
# Compile of gend.v was successful.
# Compile of modexp.v was successful.
# 6 compiles, 0 failed with no errors.
VSIM 17> restart
# Loading work.testbench
# Loading work.control
# Loading work.keygen
# Loading work.gend
# Loading work.morseToASCII
# Loading work.modexp
VSIM 18> run
# time =  0.0, message = xxxxxxxxxxxxxxxx
# expected ascii = b08242089229a0
# time = 2305000.0, message = 00b08242089229a0

VSIM 19>
```

Fig. 9.  Second test of the designed system.

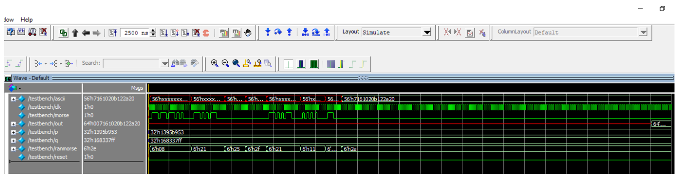
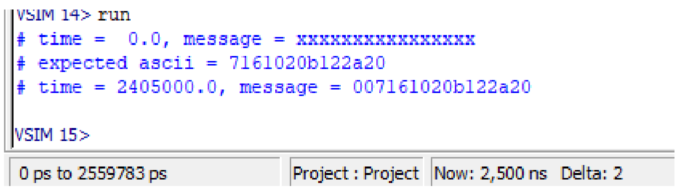


Fig. 10. Waveform of signals used in the first test for the designed system.



```
VSIM 14> run
# time =   0.0, message = xxxxxxxxxxxxxxxx
# expected ascii = 7161020b122a20
# time = 2405000.0, message = 007161020b122a20

VSIM 15>
```

```
0 ps to 2559783 ps    Project : Project    Now: 2,500 ns   Delta: 2
```

Fig. 11. Input and output displayed for first test of the designed system.

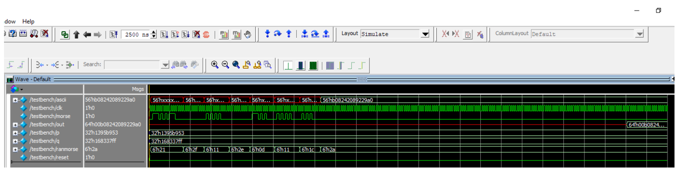
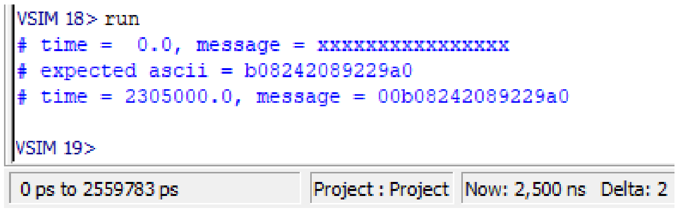


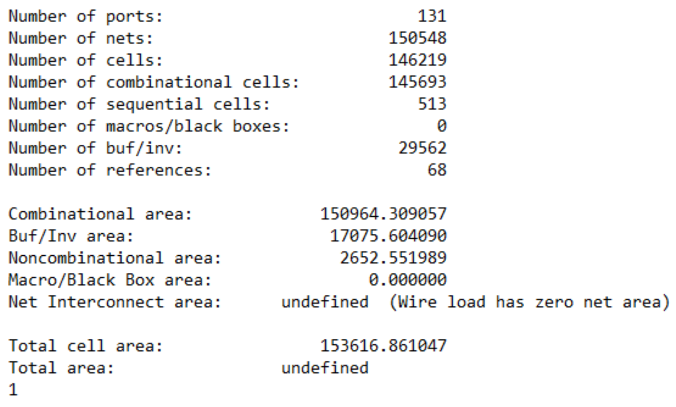
Fig. 12. Waveform of signals used in the second test for the designed system.



```
VSIM 18> run
# time =   0.0, message = xxxxxxxxxxxxxxxx
# expected ascii = b08242089229a0
# time = 2305000.0, message = 00b08242089229a0

VSIM 19>
```

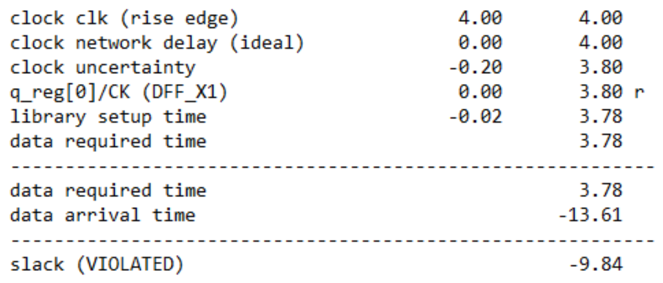
```
0 ps to 2559783 ps    Project : Project    Now: 2,500 ns   Delta: 2
```

Fig. 13. Input and output displayed for second test of the designed system.



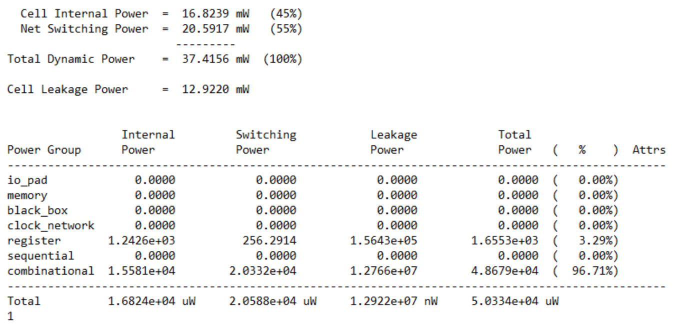
```
Number of ports:                          131
Number of nets:                        150548
Number of cells:                       146219
Number of combinational cells:         145693
Number of sequential cells:               513
Number of macros/black boxes:               0
Number of buf/inv:                      29562
Number of references:                      68

Combinational area:            150964.309057
Buf/Inv area:                   17075.604090
Noncombinational area:           2652.551989
Macro/Black Box area:               0.000000
Net Interconnect area:       undefined  (Wire load has zero net area)

Total cell area:               153616.861047
Total area:                  undefined
1
```

Fig. 14. Area Report



```
clock clk (rise edge)                    4.00       4.00
clock network delay (ideal)              0.00       4.00
clock uncertainty                       -0.20       3.80
q_reg[0]/CK (DFF_X1)                     0.00       3.80 r
library setup time                      -0.02       3.78
data required time                                  3.78
-------------------------------------------------------------
data required time                                  3.78
data arrival time                                 -13.61
-------------------------------------------------------------
slack (VIOLATED)                                   -9.84
```

Fig. 15. Timing Report



```
  Cell Internal Power  =  16.8239 mW   (45%)
  Net Switching Power  =  20.5917 mW   (55%)
                         ---------
Total Dynamic Power    =  37.4156 mW  (100%)

Cell Leakage Power     =  12.9220 mW


                 Internal         Switching           Leakage            Total
Power Group      Power            Power               Power              Power   (   %    )  Attrs
--------------------------------------------------------------------------------------------------
io_pad             0.0000            0.0000            0.0000            0.0000  (   0.00%)
memory             0.0000            0.0000            0.0000            0.0000  (   0.00%)
black_box          0.0000            0.0000            0.0000            0.0000  (   0.00%)
clock_network      0.0000            0.0000            0.0000            0.0000  (   0.00%)
register        1.2426e+03          256.2914       1.5643e+05        1.6553e+03  (   3.29%)
sequential         0.0000            0.0000            0.0000            0.0000  (   0.00%)
combinational   1.5581e+04        2.0332e+04       1.2766e+07        4.8679e+04  (  96.71%)
--------------------------------------------------------------------------------------------------
Total           1.6824e+04 uW     2.0588e+04 uW     1.2922e+07 nW     5.0334e+04 uW
1
```

Fig. 16. Power Report

## References

[1] J. Mangual, "Is morse code without spaces uniquely decipherable?" Sep 1963. [Online]. Available: https://cs.stackexchange.com/questions/34067/is-morse-code-without-spaces-uniquely-decipherable

[2] R. Singh, "Ascii table, ascii code, ascii chart, ascii charset." [Online]. Available: https://www.asciitable.xyz/

[3] Ixnsane and Instructables, "Learn to talk binary (the efficient/hard way)," Oct 2017. [Online]. Available: https://www.instructables.com/Learn-to-talk-Binary-The-efficienthard-way/

[4] J. D. Cook, "Rsa encryption exponents are mostly all the same," Jun 2020. [Online]. Available: https://www.johndcook.com/blog/2018/12/12/rsa-exponent/#: :text=The%20big%20idea%20of%20public,number%2C%20specifically%20e%20%3D%2065537.

[5] R. Shams, F. H. Khan, and M. Umair, "Cryptosystem an implementation of rsa using verilog," *2013*, vol. 1, no. 2013, p. 102–109, Aug 2013.

[6] fgrieu, "Calculating rsa private exponent when given public exponent and the modulus factors using

extended euclid," Nov 2020. [Online]. Available: https://crypto.stackexchange.com/questions/5889/calculating-rsa-private-exponent-when-given-public-exponent-and-the-modulus-fact

[7] C. E., C. H.V.A, N. P, U. T.H, and C. K. M., "Implementation of rsa cryptosystem using verilog," *International Journal of Scientific &amp; Engineering Research*, vol. 2, no. 5, May 2011.