San Francisco State University

Engineering 697 - Engineering Design Project II

Hardware Implementation of 1024 bit RSA Decryption using Modular Exponentiation

Undergraduate Team:
Lukas Pettersson
Kevin Manago
Zachary Bachman
Muhib Noorali

Faculty Advisors:
Dr. Hamid Mahmoodi
Dr. George Anwar

December 16, 2020

GitHub Repository:

https://github.com/LukasPettersson/Engr_697_RSA_Decrytion_module/tree/master

Contents

1 Summary of proposal

Today's process of manufacturing hardware goes through 4 stages:

1. Design starts at Intellectual Property Vendors.
2. From there, it is sent to the Design Integration Facility.
3. The next step, sends the design is sent to a separate facility for Testing and then Application.
4. Lastly, it is sent out to the market.

There exists a lot of threats in the process of creating the hardware component. In the above process, all these locations would have full access to the entire netlist of the hardware. This creates avenues for Black Market Manufacturing, Selling, and illegal reproduction of a company's IP.

To solve this problem, a process known as Look Up Table (LUT) based logic obfuscation is used. In short, this prevents the product from being used without the proper key (configuration bits) loaded into the design. Without access to the correct configuration bits, the design will not work properly. This is where the scope of our project lies, to safely transmit these configuration bits to a given design we need a robust encryption algorithm. In this case the algorithm that is being used is RSA.

The RSA algorithm is an asymmetric algorithm, meaning it uses one key to encrypt the data known as the public key, and one key to decrypt the data known as the private key. Our project goal is to design a 1024-bit RSA decryption module which will be tested on a DE2-115 FPGA board using a custom built edge detector.

2 Original Design Goals

For our project, we were guided towards RSA Decryption. It is widely used to secure data transmission.

Our design goals for this project is to allow for a 1024-bit encrypted message to be decrypted with the use of a 1024-bit cipher text, private key, and "n" (a multiplication of select prime numbers). From there, we wanted to push our project onto an Intel Cyclone IV DE2-115 FPGA. This allows an input (cipher text) to be fed into the module through a custom built UART. We would then send our output, the decrypted cipher text, to an edge detector for the use of visualization and verification.

3 Justification of Goals

The decision to use a 1024-bit key was defined by the other research projects in Dr. Mahmoodi's lab. The size of the key relates to the level of security that would be provided from the system. The current recommended bit size of 2048-bit as of 2010[2]. That being said, having our system configured to work with with 1024-bit keys was a minimum standard that we had to reach for our project. Because we were aware that a longer bit size would result in a higher level of security, we made our project scalable.

Originally, our design was to have our FPGA take input from a UART and have the FPGA output the decrypted message to an edge detector. Our test plan was to compare two images using an edge detector. Firstly, The edge detector would first receive a bit streamed image and display it. We would then use software to encrypt our bit stream and send that to the edge detector to be display. Lastly we would run the encrypted bit stream through our decryption module to display the original image as proof that our decryption was successful. We wanted to include this to be able to visually represent a successful decryption, as well as show what a failed/incomplete decryption would look like.

We decided to facilitate the transfer of data to our decryption block through UART because this would ease the testing process for bigger values. For smaller values, input through a PS2 keyboard would have been sufficient, however this method has a wider margin of error for larger inputs.

4 Discussion of Design Methodologies

4.1 RSA Decryption

RSA, named after it's inventors Rivest, Shamir, and Adleman is a public key cryptography system allowing encryption and decryption of messages between two parties without first sharing a secret key. The security of RSA relies on the practical difficulty of large prime number factorization, also known as the "factoring problem"[3]. There are no published methods to defeat the system in a realistic time-frame if large enough keys are used. The implementation of RSA decryption consists of computing a power calculation with an encrypted message and private key, followed by a modulo calculation with the product of two prime numbers. The overall decryption process is based on Equation 1 below.

$$M = C^d \; mod(n) \tag{1}$$

Where M is the Original Message, C is the CipherText, d is the private key, and n is a product of the two prime numbers p and q.

The Private Key, d, is calculated in the following equation:

$$d = e^{-1} mod(\phi(n)) \tag{2}$$

where $\phi(n)$, the Euler's Totient is defined as

$$\phi(n) = (p-1)(q-1) \tag{3}$$

and e, the public key, is a number $1 < e < \phi(n)$ such that these e and $\phi(n)$ are co-prime.

4.2 Montgomery Modular Multiplication

Due to the large bit size of the private key, implementing the calculation $C^d mod(n)$ directly would be both space and time inefficient. To solve the space and time inefficiency problem we can use the Montgomery Modular Multiplication technique to solve this problem.

The Montgomery modular multiplication algorithm when given two integers a, b, and modulus N, computes the double-width product ab, and then performs a division by subtracting multiples of N to cancel out the unwanted high bits until the remainder is once again less than N.[4] Equation 4 shown below.

$$\bar{a} \equiv \bar{b} \; mod(n) \tag{4}$$

Montgomery reduction adds multiples of N to cancel out the low bits until the result is a multiple of a convenient (i.e. power of two) constant $R > N$. The low bits are then discarded, producing a result less than 2N. One conditional final subtraction reduces this to less than N. This procedure has a better computational complexity than standard division algorithms, since it avoids the quotient digit estimation and correction that they would need. The result is the desired product divided by R.

To multiply a and b, they are first converted to Montgomery form or Montgomery representation and multiplied together.

$$aR \; mod(N) * \; bR \; mod(N) = \; abR2 \; mod(N)$$

This ends up being reduced to the Montgomery form of the desired product:

$$abR \; mod(N)$$

There are three secondary inputs based on n that are used in the Montgomery Modular Multiplication algorithm, the formulas for which are shown below:

4.2.1 Secondary Constant: n_0'

$$n_0' = -n^{-1}\ mod(2^w) \tag{5}$$

To take an inverse modulo we used the Extended Euclidean Algorithm, which is shown in section 4.3. We also needed to divide two 1024 bit numbers and for that be needed to use Non-Restoring division (Section 4.4).

4.2.2 Secondary Constant: r

$$r = 2^{sw}\ mod(n) \tag{6}$$

The first problem we had was to do the calculation 2^{sw} where $s = 32$ and $w = 32$. This resulted in a large number that is not able to be compiled by the IDE. To alleviate this, we performed bit shifting instead. The calculation is shown below:

$$1'b1 << 1024 = 2^{1024}$$

Once we calculated this value, we performed Non-Restoring Division (Section 4.4) with n to calculate r.

4.2.3 Secondary Constant: t

$$t = r^2\ mod(n) \tag{7}$$

To solve for t, we had to look into another algorithm to do the division, since through Non-Restoring Division by itself, we would need three registers of size 2048. This was not possible with the specifications of the FPGA, since it would take up too much space.

Instead, we constructed an algorithm using the Divide-and-Conquer[8] algorithm inspired by the Karatsuba algorithm to solve for t, and save on register space. This algorithm uses the following identity:

$$r^2 = a = a_0 + a_1 * (1'b1 << 1024)$$

where $a_0 = a[1023 : 0]$ and $a_1 = a[2047 : 1024]$.
With these values, we can use the distributive property of modulo to transform the equation into

$$a\ mod(n) =\ a_0\ mod(n) +\ a_1\ mod(n) *\ (1'b1 << 1024)\ mod(n)$$

This can further be simplified into

$$a\ mod(n) =\ a_0\ mod(n) +\ a_1\ mod(n) * r$$

From this step, we perform Non-Restoring Division (Section 4.4) on a_0 and a_1. These values are then added together to the new "a". Lastly, we check if "a" satisfies the following condition:

$$a < 2^{1024}$$

If this is true, we do the following operation using Non-Restoring Division, to get the final answer.

$$t =\ a\ mod(n)$$

4.3 Extended Euclidean Algorithm

Since n_0' uses an inverse modulo, We used the extended Euclidean algorithm to calculate it, shown below.

$$ax + by = gcd(a, b) \tag{8}$$

This is particularly useful when a and b are co-prime.[7] With that assumption, we can do the following:

$$x^{-1} = \ a\ mod(b)$$

$$y^{-1} = \ b\ mod(a)$$

The Euclidean algorithm performs a continual repetition of the division algorithm for integers. The point is to repeatedly divide the divisor by the remainder until the remainder is 0. The GCD is the last non-zero remainder in this algorithm.

4.4 Non-Restoring division

Quartus, the programmable logic device design software we used, only allows division of 2 values that are a maximum of 64 bits, which is nowhere near the size of the values that are used in our project.

The most efficient division algorithm that allows us to divide very large bit values is the Non-restoring division algorithm.[5]. This algorithm uses the digit set $\{-1, 1\}$ for the quotient digits instead of $\{0, 1\}$.[6] The algorithm is more complex, but has the advantage when implemented in hardware that there is only one decision and addition/subtraction per quotient bit; there is no restoring step after the subtraction, which potentially cuts down the numbers of operations by up to half and lets it be executed faster.

5 Summary of Results

To prove the correctness of our results, we constructed the following Python scripts.

5.1 Python Code Verification

```
#inputs
p = 7541092308233152668357869885699361884571624408337246323798
816651566266664302850892156040420078769518129683113313376866
01590605304203042314518789051692626 1
q = 8169592591906960658612239663689510229719413712868542159663
8164516758638873201735316010583156804174876779351024866371 0
3620718659512211981273172051042284669
N = p*q
M = 5
E = 65537
# Euler's Totient
PHI = (p-1)*(q-1)
# Private Key
# d = E^-1 mod (PHI)
d = 1893624271772982251140641370517276046460701972915798293808
39370904360542309829281710112863638419328151462482177679227331
8698755502360424197705661117632204752037467729720966776877529
2571908036367195932874437860665940243367886006965982919316640 9
2079172254398654761284234082672779554717649369031643989 4
318821633

# encryption #
# C = M^e mod n
cipher = pow(M, E, N)
print('cipher: ', hex(cipher))

if d < 0:
    d += PHI

# decryption #

# m = c^d mod n
message = pow(cipher, d, N)

print('message: ', message)
```

Code Segment 1: Encryption and Decryption Python Verification

```
# function for extended Euclidean Algorithm, used for
# n0prime and private key
def gcdExtended(a, b):
    # Base Case
```

```python
    if a == 0:
        return b, 0, 1

    gcd, x1, y1 = gcdExtended(b % a, a)

    # Update x and y using results of recursive
    # call
    x = y1 - (b // a) * x1
    y = x1

    return gcd, x, y
# Driver code

a = 6160765185622812638651916873490405096152453594071816559835
2213318899309279980548229922888921123349429420158332399540913
6653893662421540511542076611726875510495827653905495431176664
1356873082297158915733117570529420038586974357247085793577866
707717398292476790295325083643149950127469869870424080878 3
5284581680
b = 65537

g, x, y = gcdExtended(a, b)
print(x)
print(y)
print("gcd(", a, ",", b, ") = ", g)
```

Code Segment 2: Euclidean Algorithm to calculate n0prime and private key

```python
BIT_LENGTH = 1024
r_test = int("56554009773775337999892181609094371438748626 01
2794326076725654519934057245539866672862699480160837162279 79
72150723115303280110363301085201844321082940989908572325 6236
3539745100754849448885406105593693916993932959564226803209 91
0218266350023001812239130843537389089681483377982663560633 84
594170343407053655199 8")
n_test = int("616076518562281263865191687349040509615245359 4
071816559835221331889930927998054822992288892112334942942015
83323995409136653893662421540511542076611726875667602676655
306628700877736850761803440069296945175455364046369619395662
7633160378154376950749901625348664774830837828463163745954122
94468264477768437926 09")

# Debugging flag
DEBUG = False

if DEBUG:
    BIT_LENGTH = 1024
```

```
    r_test = 800 << 1010
    n_test = 939 << 1010


def expectedResult(r, n):
    return (r * r) % n


def halve(num):
    # We need to remove the `0b` prefix.
    num_binary = bin(num)[2:]
    # We pad the binary with enough zeroes.
    num_binary = ("0" * ((2 * BIT_LENGTH) - len(num_binary)))
        + num_binary
    assert len(num_binary) == (2 * BIT_LENGTH),
        "The bitstring is of the correct length."
    return num_binary[0:BIT_LENGTH], num_binary[BIT_LENGTH:]


def karatsubaLike(r, n):
    shift_mod = (1 << BIT_LENGTH) % n

    result, count = r * r, 0
    # As long as the upper half is not all zero...
    while result > (1 << BIT_LENGTH):
        result_upper_binary, result_lower_binary = halve(result)
        print("*{}* *{}*".format(result_upper_binary,
            result_lower_binary))

        result_upper_mod = int(result_upper_binary, 2) % n
        result_lower_mod = int(result_lower_binary, 2) % n

        result  = result_lower_mod
        result += (result_upper_mod * shift_mod)

        count += 1

    print("Number of iterations: {}".format(count))
    result %= n
    return result


if __name__ == "__main__":
    r_binary   = bin(r_test)[2:]
    n_binary   = bin(n_test)[2:]
    rsq_binary = bin(r_test * r_test)[2:]
```

```python

    print("The bitsize of r is {}, of n is {}, and of r^2 is
        {}.".format(
        len(r_binary),
        len(n_binary),
        len(rsq_binary)))

    expected = expectedResult(r_test, n_test)
    actual   = karatsubaLike(r_test, n_test)
    print("Expected result is {};\nActual   result is {}."
        .format(hex(expected), actual))
    assert expected == actual, "The result is not as expected."
```

Code Segment 3: Divide by Conquer based algorithm to calculate secondary constant t

```python
n = '57BB720287B13E536673FB40D826ED1DC7E6276153B6A6374FF
D0F4FD8319257AA5679E60058B7E0B2CA6F16B6F61CD8703A5407B7
5166C23C12800D46D4783519DBB74FAD3C39D546CE4CE83E1B6B6E
196C0889D1C8EEF49E7D555CF8E571AD6BBF7557DE96786ED1A48D
93EF8EEBE50C11CD2BAFCE0635569644860E8780E1'
m = '3960397594E02C302145DF601BC20F281E3C830235785515FF
2DB9FE31A4194BE95FBE795CDC7392295EC227A1237E06EC65AB4A
9A91A3A52165C63DBFD712134A7736FF2C4E4A6DF52F5C71D6A55
D070F1C6D2E1D8BC7D4C9E76D68E6676168A65842292AFE4167558
ABFE95437D60D49CD89DA43EB5576E410C0748FC30498'
e = '1af752a3d4717666cc26fa23844cf2b524ac698e50dce35a3
d0bdb98ff2abd8094aadfed024e42b69e3dfac75095756851969c1
d830b523a819c6e1fa695c81cae122e622a7b4ba5ee95fa8098ba
69b6b9ae3b75f9060b0a4f64499bdf36a80ea6890298f6fab0a15
ffc14d7da1c7f679d19cc113f24e294f39317ac534ab101'

chnk_len = 8
res_n = []
res_m = []
res_e = []
for idx in range(0, len(n), chnk_len):
    res_n.append(n[idx: idx + chnk_len])
    res_m.append(m[idx: idx + chnk_len])
    res_e.append(e[idx: idx + chnk_len])

p = 31
for x in range(32):
    print('n_input = 32\'h' + res_n[p] + ";")
    print('m_input = 32\'h' + res_m[p] + ";")
    print('e_input = 32\'h' + res_e[p] + ";")
    print('#100')
    p = p - 1
```

Code Segment 4: Splits hex values for Test Bench Stimulus

5.2 Waveform Simulation

Based off of our Python Script, we used our original message as 5, and we were able to calculate a Cipher-text based off of that script. We used that Cipher-text in our decryption module and based off of Figure 1, we can see the following:

1. res_full, our output, is equal to our original message, 5 based on Code Segment 1
2. m, the Cipher-text is correct based off of the Python script
3. e, the private key is correct based off of the Python script

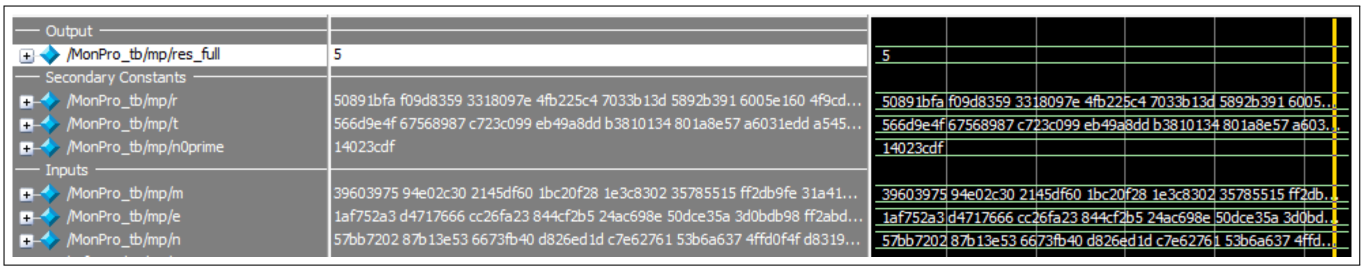

Figure 1: Modelsim Simulation of Decryption, showing input, output, and secondary constants

6 Complete Detailed Design Documents

6.1 Flowcharts

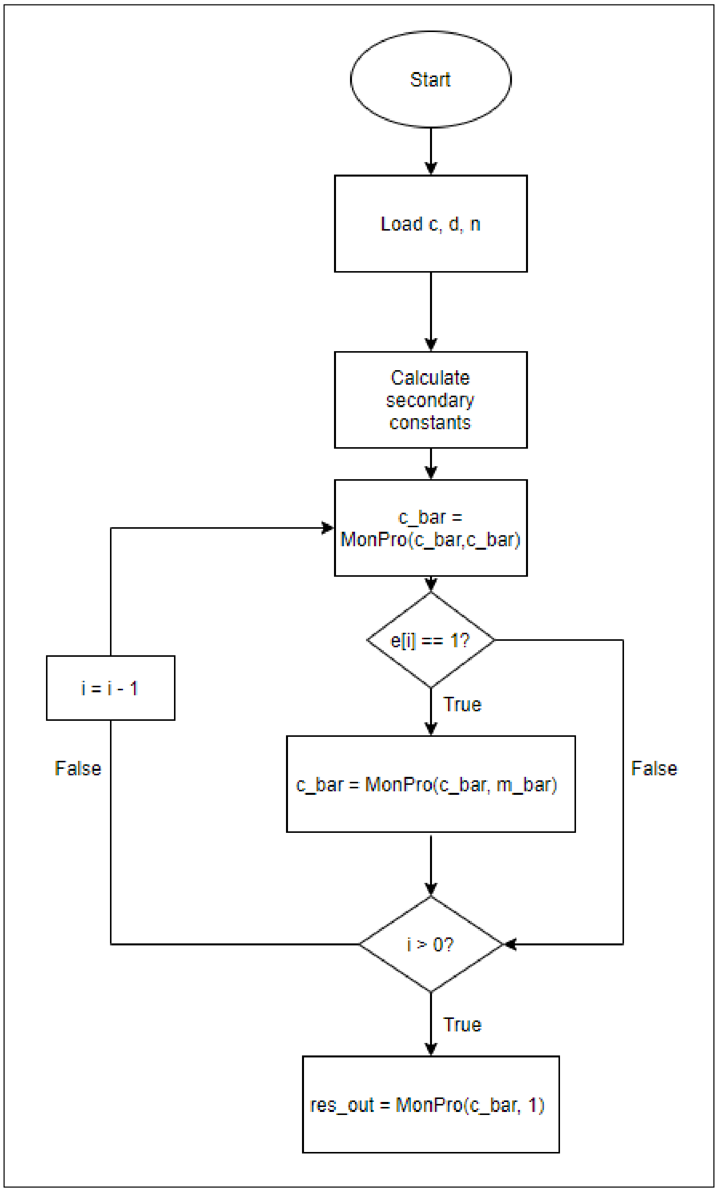

Figure 2: Flowchart for Decryption Process

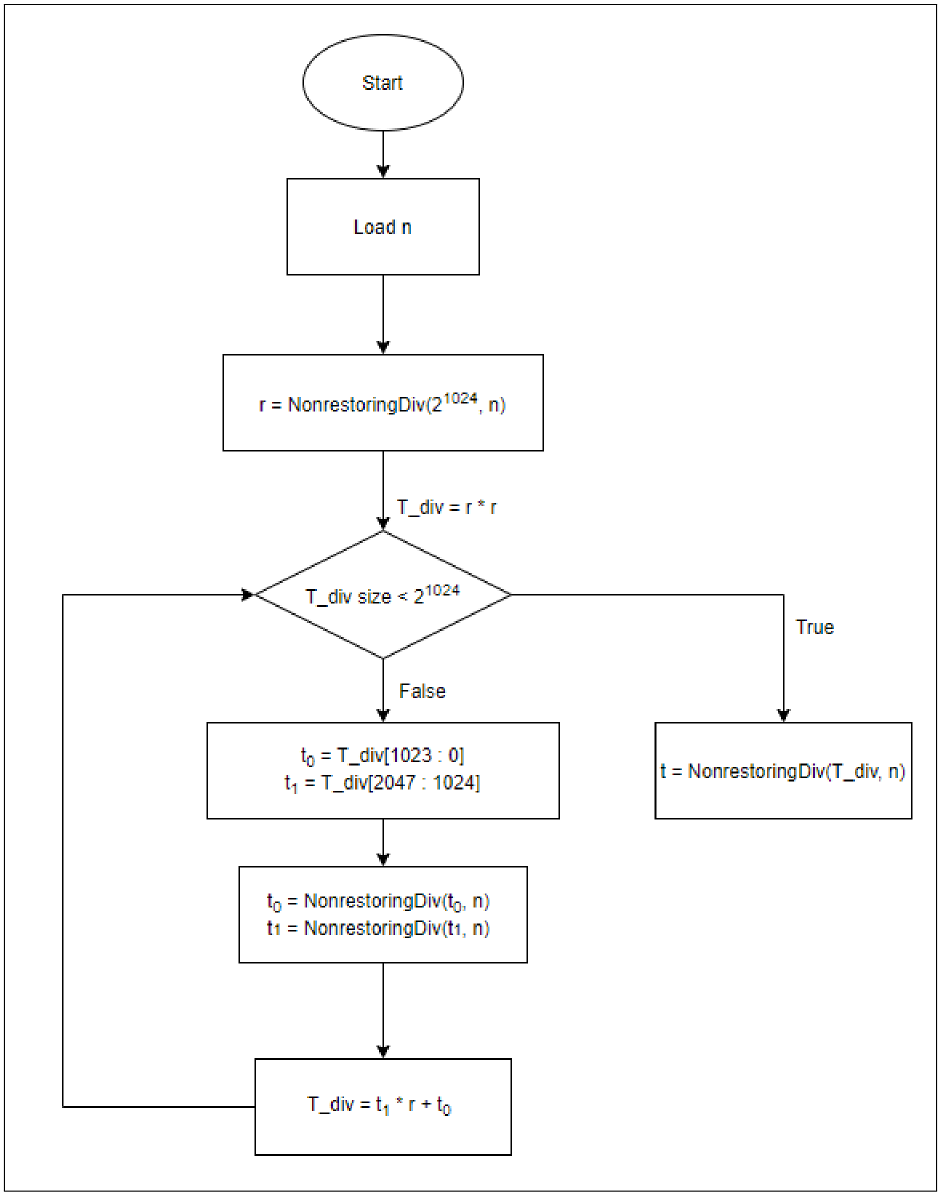

Figure 3: Flowchart for Secondary Constants r and t

6.2 Block Diagrams

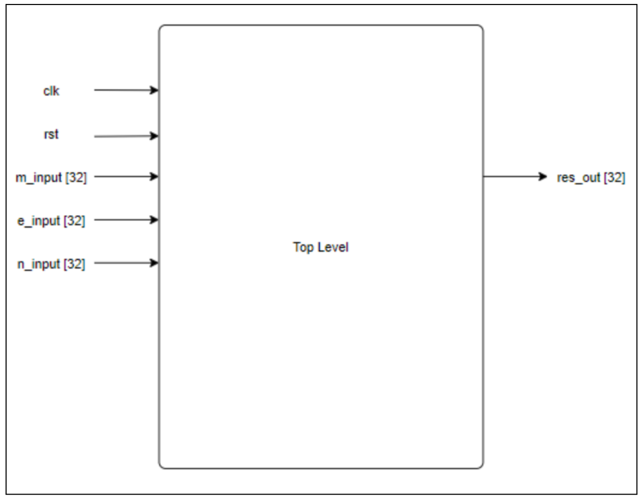

Figure 4: Top Level Block Diagram

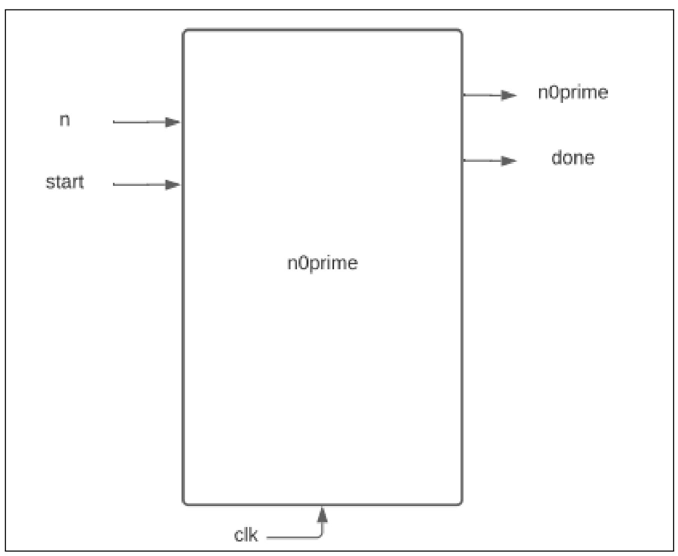

Figure 5: Block Diagram for n_0'

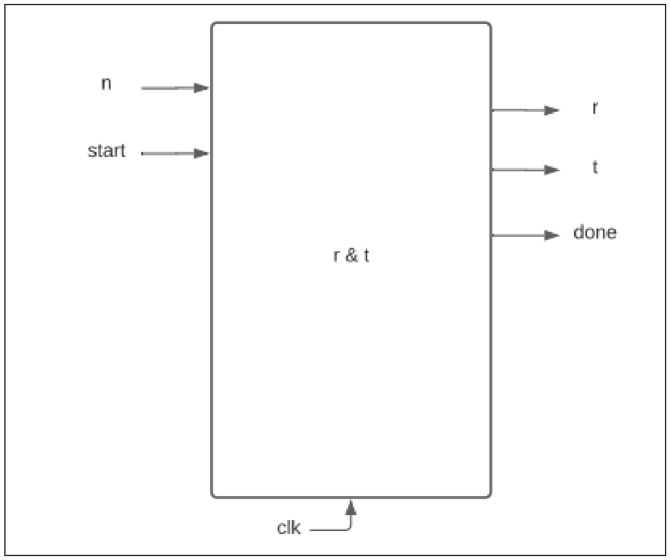

Figure 6: Block Diagram for Secondary Constants r and t

7 References

1. https://github.com/fatestudio/RSA4096/tree/master/ModExp%202.0
2. https://crypto.stackexchange.com/questions/10700/when-nist-disallows-the-use-of-1024-bit-keys-what-effect-will-that-have-on-sha
3. https://en.wikipedia.org/wiki/RSA_(cryptosystem)
4. https://en.wikipedia.org/wiki/Montgomery_modular_multiplication
5. https://www.geeksforgeeks.org/non-restoring-division-unsigned-integer/
6. https://en.wikipedia.org/wiki/Division_algorithm#Non-restoring_division
7. https://en.wikipedia.org/wiki/Extended_Euclidean_algorithm
8. https://en.wikipedia.org/wiki/Divide_and_conquer
9. https://asecuritysite.com/encryption/getprimen
10. https://www.rapidtables.com/convert/number/decimal-to-binary.html