Digital Circuit Lab — Lab 2 Tutorial

RSA Decrypter

Date: 04/12/2017

Team 05 Member: 郭子生 b03901032、曾耕森 b03901154、楊其昇 b03901101

Github: https://github.com/awinder0230/Digital-Circuit-Design-Lab

- **Introduction**

RSA, named after its designers Ron Rivest, Adi Shamir, and Leonard Adleman, is a public-key cryptosystem for secure data transmission. In this project, we are going to implement a circuit system to decrypt RSA messages, given ciphertext and receiver's private key.

- **Tools**: Altera DE2-115 FPGA board, System Verilog, Quartus II, RS-232 Qsys Module

- **RSA Theory**

1. Key Generation

The keys of RSA algorithm[1] are generated in the following manner:

i. Choose two prime numbers p and q. For security purpose, these prime numbers should be large and chosen randomly.
ii. Compute N = pq, which used as modulus for both public key and private key.
iii. Compute r = lcm(p-1,q-1), then choose an integer e such that $1 < e < r$ and gcd(e,r)=1.
iv. Compute $d = e^{-1} \bmod r$.
v. N and e are public keys for anyone who would like to send messages to the receiver. N and d are private keys which should be kept secret by the receiver him/her self.

Encryption:

i. For message m, compute ciphertext $y = m^e \bmod N$ for encryption.

Decryption:

i. For ciphertext y, compute $m = y^d \bmod N$ for message recovery.

2. Compute $y^d \bmod N$

In this project, we are going to implement a circuit system to decrypt RSA message. Therefore, the core of the circuit should be able to compute $y^d \bmod N$. The most intuitive way to solve the problem is simply multiply y by itself d times, which results in time complexity O(d).

```
m = 1
for i in range(d):
  m = (m * y) mod N
return m
```

A tip to reduce time complexity is to compute exponentiation by squaring. The idea is to square the exponential term for each iteration, which results in time complexity O(lgd).

```
function Exponentiation by Squaring (y, d, N):
  y_sqr = y
  m = 1
  for i in range(256):
    if (i-th bit of d == 1):
      m = m * y_sqr mod N
    end if
    y_sqr = y_sqr ^ 2 mod N
  end for
  return m
end function
```

Although with the algorithm above, it still takes great effort to compute modulo of product in circuit (line 6 and line 8 above). To implement a modulo of product in digital circuit system, a * b mod N for example, one may consider **adding** 'b' to 'm' (which is initialize to 0 at the beginning) 'a' times. This idea is actually identical to the algorithm describe above, which **multiplies** 'y' to 'm' (which is initialize to 1 at the beginning in that case due to the task is for multiplication) 'd' time. Thus, with the algorithm down below, we are able to compute a * b mod N in circuit. Note that the algorithm is similar to the one above.

```
1  function Modulo of Product (a, b, N):
2    b_dbl = b
3    m = 0
4    for i in range(num): # num is number of bits
5      if (i-th bit of a == 1):
6        if (m + b_dbl >= N):
7          m = m + b_dbl - N
8        else
9          m = m + b_dbl
10       end if
11       if b_dbl + b_dbl >= N
12         b_dbl = b_dbl + b_dbl - N
13       else
14         b_dbl = b_dbl + b_dbl
15       end if
16     end if
17   end for
18   return m
19 end function
```

For this algorithm, please do notice that there are still plenty spaces for optimization as the algorithm is implemented in hardware. For example, one may execute the if-statements in line 6 and line 11 in parallel. Also, it is easier to compute b_dbl + b_dbl by shifting left by 1 bit, rather than adding them together directly.

Apart from some optimization tips describe above, there exists an approach named Montgomery Algorithm[2] to simplify and reduce the computation cost even much more. Rather than computing a * b mod N, Montgomery Algorithm computes a * b * 2^{-256}mod N which takes the advantage of low computation cost when multiplying 2^{-1} and checking parity in circuits. The basic idea of Montgomery Algorithm is based on this equation:

$$\begin{aligned} ab2^{-256} &= (\sum_{i=0}^{255} a_i 2^i) b 2^{-256} \\ &= \sum_{i=0}^{255} a_i b 2^{i-256} \\ &= ((((a_0 b)2^{-1} + a_1 b)2^{-1} + \cdots)2^{-1} + a_{255} b)2^{-1} \end{aligned}$$

Also, note that to compute u * 2^{-1} ≡ v mod N, it is equivalent to compute u ≡ 2 * v mod N. Thus, if u is even, then v = u / 2; else if u is odd, then u = 2 * v - N (N must be odd since N = pq, where p and q are large prime numbers), which results in v = (u + N) / 2.

To check the parity of a number in circuit, we only have to look at the LSB of the number is whether 1 or 0. Also, as we have mentioned above, multiplying 2^{-1} to a number in circuit, may be implemented by shifting the number right by 1 bit. Last but not least, by Montgomery Algorithm, one may not need to check if the current value of m is larger than N for each iteration, which also reduce a great amount of computation cost.

Based on these ideas, Montgomery Algorithm is shown down below:

```
1  function Montgomery Algorithm (a, b, N):
2    m = 0
3    for i in range(256):
4      if (i-th bit of a == 1):
5        m = m + b
6      end if
7      if (m is odd):
8        m = m + N
9      end if
10     m = m / 2
11   end for
12   if m >= N:
13     m = m - N
14   end if
15   return m
16 end function
```

Now take a look back to the second algorithm we have mentioned above: exponentiation by squaring. By replacing the 2nd line with y_sqr = y * 2^{256} mod N, we are able to compute line 6 and line 8 with Montgomery Algorithm, rather than computing with the function modulo of product. The reason of replacing the 2nd line is because Montgomery Algorithm computes a * b * 2^{-256} mod N, and by letting y_sqr = y * 2^{256} mod N, two exponential terms are going to cancel each other, so that we are able to get the correct result while applying Montgomery Algorithm.

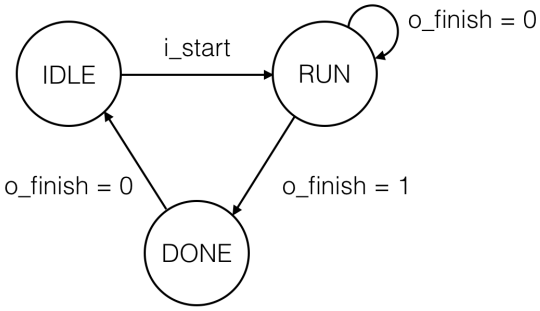
Therefore, the final algorithm which computes y^d mod N in our task would be like:

```
1  function RSA256Core (y, d, N):
2    y_tmp = Modulo_of_Product(y, 2^256, N)
3    m = 1
4    for i in range(256):
5      if (i-th bit of d == 1):
6        m = MontgomeryAlgorithm(m, y_tmp)
7      end if
8      y_tmp = MontgomeryAlgorithm(y_tmp, y_tmp)
9    end for
10   return m
11 end function
```

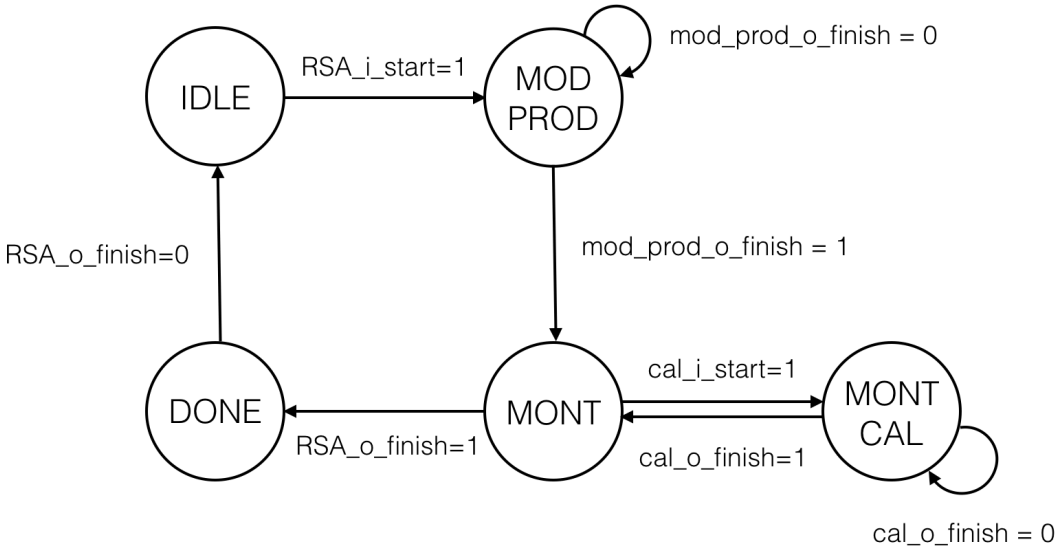
To implement the algorithm above, it is highly recommended to implement two modules: "Modulo_of_Product" and "Montgomery_Algorithm" aside from RSA256Core module. Since the implementation would be much more simple in this way, and one may try to accelerate the algorithm above by computing line 6 and line 8 in parallel.

3. Finite State Machine

In our case, both modules "Modulo_of_Product" and "Montgomery_Algorithm" have three states: IDLE, RUN, DONE. Notice that it is important to pull o_finish down to 0 fast to avoid incorrect output result.



For module “RSA256Core”, we have five states: IDLE, MOD_PROD, MONT, MONT_CAL, and DONE. MOD_PROD is to calculate modulo of product (line 2), MONT is to keep track on for loop (line 4), MONT_CAL is for Montgomery Algorithm (in line 6 and line 8) , and DONE is used to pull signal o_finish up and turn it off quickly.



- **Qsys**

Qsys is a system integration tool built in Quartus. We use Qsys to wrap up our designs and set up connections between modules. The modules used in this project are described below:

- **ALTPLL**

ALTPLL is a clock rate conversion tool provided in Qsys. In this lab, we convert the original clock rate from 50 MHz to 25 MHz and distribute to other modules.

Signal	Type	Description
inclk_interface	input	native 50MHz clock signal
inclk_interface_reset	input	reset signal triggered by KEY_0 (button)
c0	output	tuned 25 MHz clock signal

- **UART**

Universal Asynchronous Receiver/Transmitter, UART, serves as a communicator between serial data and parallel data. It converts data to serial or parallel form to help communication between devices. We use RS232 as serial data transmit interface. UART convert the RS232 serial signal to parallel form and send it to Rsa256Wrapper to perform the decoding calculation.

Signal	Type	Description
clk	input	tuned 25 MHz clock signal from altpll
reset	input	reset signal triggered by KEY_0 (button)
s1	Avalon-MM	associated with r/w address
external_connection	output (conduit)	where signals finally output to (computer)

- **Rsa256Wrapper**

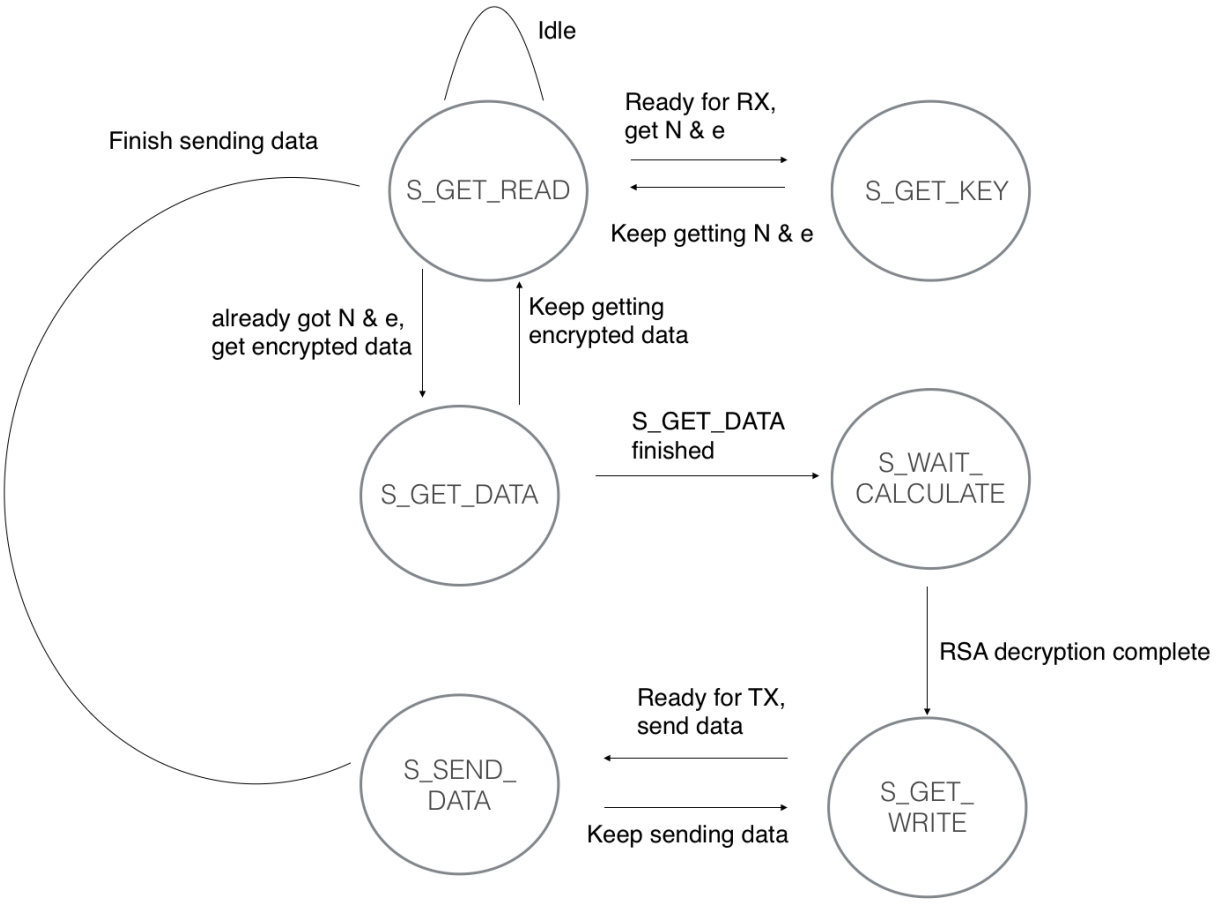
  Rsa256Wrapper is a user defined component created in this project. It wraps up the Rsa256 decoder circuit and provide I/O signals that agree with Avalon-MM interface. Avalon-MM (Avalon Memory Mapped) interface is a bus interface set up by Altera. By observing the specification, we can easily set up communication between our own modules and other predefined modules. Avalon-MM interface can be categorized to two types: Master and Slave. Master components can send I/O request to the bus, while Slave components only performs passive read/write operation.

  When creating new components, you should set the signal interfaces and signal type properly. For further description of Rsa256Wrapper, take a look at the section down below.

signal	type	description
Avalon_master_0	Avalon-MM Master	Act as the Master among the interface of Avalon-MM
reset_sink	input	Tuned 25 MHz clock signal from altpll
clock_sink	Input	Reset signal triggered by KEY_0 (button)

- **RSA Wrapper**

  1. **Finite State Machine**



- **S_GET_READ:**
  This state is also the initial state. When RX is ready (i.e. avm_waitrequest = 0 & avm_read_r = 0 & avm_readdata[RX_OK_BIT] = 1), the circuit start to execute Read operation. If read_ne = 1, the circuit will read N & e and the next state will jump to S_GET_KRY, or read encrypted data and the state will jump to S_GET_DATA.

- **S_GET_KEY:**
  This state will query 1 byte data (i.e. avm_readdata[7:0]) each time, and the next state will jump to S_GET_READ. If the circuit doesn't get full N & e, the state of the circuit will jump back to S_GET_KEY from S_GET_READ, or jump to S_GET_DATA.

- **S_GET_DATA:**
  Similar to S_GET_KEY, the circuit will query 1 byte data each time on this state. If the circuit get the whole data, the next state will be S_WAIT_CALCULATE.

- **S_WAIT_CALCULATE:**
  Wrapper will trigger Rsa256Core module. The next state is S_GET_WRITE if calculation is finished.

- **S_GET_WRITE:**
  Write operation is similar to Read operation. Once TX is ready, the circuit will begin to send the decrypted data, and the next state will be changed to S_SEND_DATA.

- **S_SEND_DATA:**
  The circuit will send 1 byte data each time, the same as Read operation. The next state will jump back to S_GET_WRITE if there are data to be transmitted; otherwise, the next state will jump back to S_GET_READ.

- **WARNING:** Always check avm_waitrequest before Read or Write operation!

2. Signal Description Table

Signal	I/O Type	Description
avm_clk	Input	Clock signal (tuned by altpll module)
avm_rst	Input	Reset signal triggered by KEY0 (button)
avm_address	Output	Where Read & Write operation occurred
avm_read	Output	Trigger Read operation
read_ne	Input	Trigger Read N & e
avm_readdata	Input	Data read from master
avm_write	Output	Trigger Write operation
avm_writedata	Output	Data written from master
avm_waitrequest	Input	Determine whether Read or Write are completed

- **Simulation**
    1. Premise: to run simulation, you should have the permission to access NTU DCLab server.
    2. Connect to DCLab work station.
    3. Type “tool 2” to enable verdi, ncverilog, and nWave.
    4. Change directory to where your code stores.
    5. To test Rsa256Core, type “ncverilog + access + r tb.sv Rsa256Core.sv”. The result of “dec” and “gold” should be the same. If not, you can use nWave to check your waveform and debug.
    6. To test Rsa256Wrapper, type “ncverilog + access + r test_wrapper.sv PipelineCtrl.v PipelineTb.v Rsa256Wrapper.sv Rsa256Core.sv”.

- **Reference**

[1] https://en.wikipedia.org/wiki/RSA_(cryptosystem)
[2] https://en.wikipedia.org/wiki/Montgomery_modular_multiplication