Project Report - RSA Decryption

Late Day Used - 1 day

A59023194 Ya-Chi Liao, A59026047 Sin-Yu Chen

December 13, 2024

Abstract

This project explores FPGA-accelerated RSA decryption using Vitis High-Level Synthesis (HLS) to address the computational demands of modular exponentiation in RSA. Starting with a baseline implementation, optimizations including the modulo of products algorithm, Montgomery algorithm, and Chinese Remainder Theorem (CRT) were applied. These enhancements reduced latency and resource usage while ensuring timing compliance on the PYNQ-Z2 board. The final design, integrating CRT and Montgomery algorithm, achieved a 16.2x speedup with minimal resource overhead. Finally, a demonstration validated the implementation's correctness.

1 Introduction

In our increasingly digital world, the need for robust data security mechanisms has never been more critical. Data breaches, unauthorized access, and cyber threats have underscored the importance of securing sensitive information across various platforms. Public-key cryptography, and in particular the RSA algorithm, has become a cornerstone for securing communications, financial transactions, and confidential data. RSA's asymmetric encryption ensures that private keys remain confidential while allowing secure exchange of public keys. However, as data security demands grow, so do the computational challenges associated with RSA, particularly during the decryption phase.

RSA decryption, an essential part of the algorithm's functionality, involves modular exponentiation. This process, which entails raising a ciphertext to a large exponent and reducing it modulo a significant composite number, is computationally intensive. To maintain security, RSA relies on large key sizes, typically 2048 bits or more. These large key sizes exponentially increase the complexity of decryption, posing significant performance bottlenecks. This challenge highlights the necessity of exploring hardware acceleration techniques to enhance the efficiency and scalability of RSA decryption.

This project presents a hardware implementation of RSA decryption using Vitis HLS, designed for the PYNQ-Z2 FPGA platform. The implementation uses the exponentiation by squaring method for modular exponentiation, the Montgomery algorithm to simplify arithmetic operations, and the Chinese Remainder Theorem (CRT)to split computations into smaller parts for efficiency. These algorithms work together to address the challenges of modular arithmetic in RSA decryption, especially for large key sizes.[SIN11] By combining these techniques with FPGA acceleration, this project achieves improved performance compared to software-only approaches. The report explains the implementation, presents results, and discusses challenges, offering insights into optimizing RSA decryption for hardware.

2 Background

RSA decryption is a fundamental component of public-key cryptography, widely used to secure communications and protect sensitive data. The decryption process involves complex mathematical operations, particularly modular exponentiation, which can be computationally intensive, especially with large key sizes. To address these challenges, various hardware implementations have been explored to enhance the efficiency and speed of RSA decryption.

One notable approach is the use of the Montgomery multiplication algorithm, which facilitates efficient modular multiplication without trial division. This method has been effectively implemented in hardware to accelerate RSA operations. [Bri90]

Another significant technique is the application of the Chinese Remainder Theorem (CRT) [WHW01] RSA decryption. By decomposing the problem into smaller, more manageable computations, CRT enables parallel processing, thereby reducing the overall decryption time.

3 Implementation and Results

Source Code: https://github.com/Karniela/RSA_Implementation_on_PYNQ.git In this GitHub repository, it includes our implementation of baseline designs, optimized designs, their synthesis results and demos.

3.1 Baseline - RSA256 with Exponentiation by Squaring

The most straightforward implementation involves directly calculating the exponentiation of y^d and then applying the modulo operation with the modulus N. However, this method is unsuitable for hardware-based designs due to its high resource requirements and the risk of overflow during computation.

To address these issues, we adopted the **exponentiation by squaring** algorithm as our baseline approach. This algorithm is both efficient and hardware-friendly, making it a more suitable choice. The process of the algorithm is described as follows:

Algorithm 1 RSA256 with Exponentiation by Squaring

function EXPONENTIATION_SQUARING(N, y, d)
  $t \leftarrow y$
  $m \leftarrow 1$
  for $i \leftarrow 0$ **to** 255 **do**
    if i-th bit of d is 1 **then**
      $m \leftarrow (m \cdot t) \mod N$
    end if
    $t \leftarrow (t^2) \mod N$
  end for
  return m
end function

The drawback of this algorithm is evident: it still relies on two computationally expensive operations—multiplication and modulo. However, during our initial trial, the resource usage of FFs and LUTs appeared unexpectedly low.

Upon further investigation, we discovered that this was due to the use of DSPs specialized for multiplication. By leveraging these DSPs, the synthesis tool avoided using FFs and LUTs for the multiplication modules, creating the illusion that the design was hardware-efficient.

To enable a meaningful comparison of both computation time and resource usage with our optimized versions, we needed to prevent the synthesis from utilizing multiplication DSPs. To achieve this, we added the directive `#pragma HLS bind_op variable=result op=mul impl=fabric` to the design.

As a result, the synthesis faced a timing violation under a clock period of 10 ns, and the resource usage increased dramatically. The synthesis report for the baseline method, without the use of DSPs, is presented in Table 1.

Table 1: Synthesis result and throughput of RSA256 baseline

		Clock Cycles		Utilization				
Optimization	Est.Clk(ns)	Latency	Interval	BRAM	DSP	FF	LUT	Throughput(Hz)
baseline (No DSP)	29.749	265483	265484	0	0	8321	140819	126.61
baseline	6.882	267280	267281	0	200	25218	20843	543.64

3.2 Optimization 1 - Modulo of Products

In the baseline method, the primary challenge of implementing the exponentiation by squaring algorithm in hardware lies in the reliance on multiplication and modulo operations. To address this, we incorporated the modulo of products algorithm into the original design. The algorithm is described as follows:

Algorithm 2 Modulo of Products (a · b mod N)

1: **function** ModuloProduct(N, a, b)
2: $t \leftarrow b$
3: $m \leftarrow 0$
4: **for** $i \leftarrow 0$ **to** 255 **do**
5: **if** $i-$th bit of a is 1 **then**
6: **if** $m + t \geq N$ **then**
7: $m \leftarrow m + t - N$
8: **else**
9: $m \leftarrow m + t$
10: **end if**
11: **end if**
12: **if** $t + t > N$ **then**
13: $t \leftarrow t + t - N$
14: **else**
15: $t \leftarrow t + t$
16: **end if**
17: **end for**
18: **return** m
19: **end function**

By inserting this function into line 6 and line 8 of the exponentiation by squaring algorithm, we successfully eliminated both the multiplication and modulo operations. However, this modification introduced an additional loop with 256 iterations, which inevitably added delays to the design. Here's where Vitis_HLS proved valuable, as it allowed us to explore different hardware optimization strategies by simply adding pragmas.

To streamline the loop execution, we tested two pragmas: `#pragma HLS PIPELINE` and `#pragma HLS UNROLL`. We have tested with different settings, and the result can be found in Table 2. Here we give a quick summary for the pragmas. For `#pragma HLS UNROLL`, unrolling ModuloProduct with a factor of 4 yields a 22% speedup with a reasonable resource usage increase. For `#pragma HLS PIPELINE`, setting the iteration latency to 3 is the limit, otherwise it will introduce timing violation.

Table 2: Synthesis result and throughput of RSA256 with Modulo of Products, *MP = mod_product, ME = mod_exp

		Clock Cycles		Utilization				
Optimization	Est.Clk(ns)	Latency	Interval	BRAM	DSP	FF	LUT	Throughput(Hz)
No optimization	6.882	526082	526083	0	0	8061	3783	276.20
UNROLL MP factor=2	6.882	460546	460547	0	0	11671	5276	315.5
UNROLL MP factor=4	6.882	427778	427779	0	0	18891	8267	339.68
UNROLL MP factor=8	6.882	411934	411935	0	0	33331	14271	352.74
MP II=2	8.155	264450	264451	0	0	7548	3813	463.69
UNROLL ME factor=2	6.882	525954	525955	0	0	8567	3829	276.27
UNROLL ME factor=4	6.882	525890	525891	0	0	9085	3885	276.31
UNROLL ME factor=8	6.882	525858	525859	0	0	10121	3985	278.83
ME II=1	6.882	393474	393475	0	0	1000461	403242	369.29

3.3 Optimization 2 - Montgomery Algorithm

By employing the modulo product algorithm, we not only resolved the timing violation issues but also significantly reduced resource usage by replacing multiplications with additions and modulo operations with subtractions. Despite these improvements, the current throughput remains unsatisfactory, with a total latency exceeding **400,000** cycles.

The primary bottleneck lies in the nested loop structure. Even with loop pipelining, we are constrained to an iteration latency of 3 cycles, which hinders further improvement. To enhance performance, we must make the modulo product algorithm more efficient and reduce the minimum latency between iterations.

Upon closer analysis of the modulo product algorithm, we found that in each iteration, the following steps occur:

1. Compare the value of $m + t$ with N.
2. Use this comparison to decide how to update m.
3. Apply a similar process for t.

This analysis highlights two potential optimization targets:

1. Eliminate the comparison.
2. Simplify the updates for m and t to only one possibility.

Achieving either of these objectives offers a valuable opportunity to enhance throughput. This is where the Montgomery algorithm becomes highly relevant, as it effectively addresses the first target. By adopting this algorithm, we gain a structured and efficient pathway to further improve the design's performance.

Algorithm 3 Montgomery algorithm for calculating $ab2^{-256}$ mod N

```
1:  function MONTGOMERY(N, a, b)
2:      m ← 0
3:      for i ← 0 to 255 do
4:          if i−th bit of a is 1 then
5:              m ← m + b
6:          end if
7:          if m is odd then
8:              m ← m + N
9:          end if
10:         m ← m/2
11:     end for
12:     if m ≥ N then
13:         m ← m − N
14:     end if
15:     return m
16: end function
```

The idea of Montgomery algorithm is simple: to prevent overflow by left shifting the maximum bitwidth in advance. But as we introduce the additional 2^{-256} term, we need to multiply the input message by 2^{256} to ensure the computation is correct. So, the full algorithm is shown as follows:

Algorithm 4 RSA256 with Montgomery

```
function RSA256Mont(N, y, d)
    t ← ModuloProduct(N, 2^256, y)
    m ← 1
    for i ← 0 to 255 do
        if i−th bit of d is 1 then
            m ← Montgomery(N, m, t)
        end if
        t ← Montgomery(N, t, t)
    end for
    return m
end function
```

From a theoretical perspective, the Montgomery algorithm appears promising as it eliminates the need for comparison operations on m and avoids the calculation of t. However, empirical results reveal a drawback—it actually increases the loop iteration latency. Specifically, the iteration latency rises to 4 cycles with the Montgomery algorithm.

Our analysis suggests that while the comparisons for m and the calculations for t are removed, the computation of m now involves two dependent if statements, which extend the processing time. The synthesis report can be found in Table 3.

Table 3: Synthesis result and throughput of RSA256 with Montgomery

		Clock Cycles		Utilization				
Optimization	Est.Clk(ns)	Latency	Interval	BRAM	DSP	FF	LUT	Throughput(Hz)
RSA_Mont	6.584	528643	528644	0	0	8592	4152	287.31

3.4 Optimization 3 - Optimized Montgomery Algorithm

To address this issue, we experimented with two implementation strategies, both grounded in a central concept: breaking down the two dependent if statements into a single flattened if-else logic. To achieve this, we analyzed the relationship between the two if statements and the resulting value of m using a truth table, which is presented in Table 4.

Table 4: Truth table for m considering the parity of a, b, and m

a	b	m	m
0	0	0	m
0	0	1	$m+N$
0	1	0	m
0	1	1	$m+N$
1	0	0	$m+b$
1	0	1	$m+b+N$
1	1	0	$m+b+N$
1	1	1	$m+b$

In the first method, we fixed the update of m to be m + d1 + d2, where d1 is chosen from either b or 0, and d2 is chosen from either N or 0. The logic for both terms is listed here:

1. **if** (a & 1) **then** $d_1 = b$ **else** $d_1 = 0$ **endif**

2. **if** (a & 1) & ((b & 1) $\oplus$ (m & 1))) $\|$ $\neg$ (a & 1) & (m & 1) **then** $d_2 = N$ **else** $d_2 = 0$ **endif**

The core idea behind this approach is to decouple the logic of the if-else statement from the arithmetic addition. This separation was expected to enable better pipelining of the computation. Using this implementation, we successfully reduced the iteration latency to 2 cycles, as shown in the synthesis report in Table 5.

Table 5: Synthesis result and throughput of RSA256 with optimized Montgomery 1

		Clock Cycles		Utilization				
Optimization	Est.Clk(ns)	Latency	Interval	BRAM	DSP	FF	LUT	Throughput(Hz)
RSA_Mont1	6.882	266244	266245	0	0	7564	4177	545.76

The second method is to separate the all the possible m with a single if-else statement.
From the table, we can summarize that the result of m has four possibilities.

1. **if** $(a$ & $1)$ & $((b$ & $1) \oplus (m$ & $1)) = 1 \rightarrow$ m = m + b + N
2. **else if** $\neg(a$ & $1)$ & $(m$ & $1) = 1 \rightarrow$ m = m + N
3. **else if** $(a$ & $1) = 1 \rightarrow$ m = m + b
4. **else** m = m

Moreover, by refining the flattened if-else structure further, we achieved an iteration latency of 1 cycle without violating the clock period constraint of 10 ns. The final results of this improvement are detailed in Table 6.

Table 6: Synthesis result and throughput of RSA256 with Optimized Montgomery

		Clock Cycles		Utilization				
Optimization	Est.Clk(ns)	Latency	Interval	BRAM	DSP	FF	LUT	Throughput(KHz)
RSA_Mont2	6.584	135684	135685	0	0	9171	4558	1.12

3.5 Optimization 4 - Chinese Remainder Theorem(CRT)

This section introduces another widely used algorithm for RSA decoding: the Chinese Remainder Theorem (CRT). The primary advantage of CRT lies in its ability to reduce the bitwidth of operations by half, significantly decreasing computation time and resource usage. Additionally, it allows for a performance comparison with our optimized Montgomery algorithm to determine which approach yields superior results. The algorithm is presented as follows:

Algorithm 5 RSA256 with CRT

function RSA256CRT(N, y, d, p, q)
    $d_p \leftarrow$ d mod (p-1)
    $d_q \leftarrow$ d mod (q-1)
    $m_p \leftarrow$ ExponentiationSquaring(p, y, d_p)
    $m_q \leftarrow$ ExponentiationSquaring(q, y, d_q)
    $q_{inv} \leftarrow$ ModuloInverse(q, p)
    if $m_p > m_q$ **then**
        ModuloProduct(p, $m_p - m_q$, q_{inv})
    else
        ModuloProduct(p, $m_q - m_p$, q_{inv})
        $h \leftarrow p - h$
    end if
    $x \leftarrow m_q + h * q$
    return x
end function

Algorithm 6 Modulo Inverse

function ModuloInverse(q, p)
    $m \leftarrow p$
    $x_0 = 0,\ x_1 = 1$
    for $i \leftarrow 0$ **to** 127 **do**
        while $q > 1$ **do**
            $a \leftarrow q\ /\ p$
            $a \leftarrow p,\ p \leftarrow a$ mod p
            $x_0 \leftarrow x_1 - q * x_0,\ x_1 \leftarrow x_0$
        end while
    end for
    if $x_1 < 0$ **then**
        $x_1 += m$
    end if
    return x_1
end function

From the algorithm, we observe that instead of computing the original 256-bit exponentiation by squaring function, CRT divides the process into two 128-bit exponentiation by squaring functions. A notable benefit is that these two computations, along with the modulo inverse function, have no data dependencies and can execute concurrently, leading to significantly reduced computation time. However, this improvement reintroduces multiplication and modulo operations, creating a dilemma of whether to use DSPs or risk timing violations.

To address the challenges posed by multiplication within CRT, we used a trick using the modulo product function. Specifically, the multiplication operation was replaced with ModuloProduct(N, a, b) with N = ∞, effectively converting a high-latency operation into one with computation time proportional to the bitwidth. This adjustment resolves timing violations elegantly.

This approach inevitably introduces additional delays for multiplication; however, its impact on the overall design is minimal. This is because the modulo inverse function accounts for only about 33% of the execution time for exponentiation by squaring. Since both functions execute concurrently, the delay remains negligible unless the modulo inverse becomes the new bottleneck.

To ensure this does not happen, we pipelined the modulo product function with an iteration interval of 1 and separated internal computations into distinct functions to meet timing constraints. Table 7 summarizes the results of this optimization.

Table 7: Synthesis result and throughput of RSA256 with CRT

		Clock Cycles		Utilization				
Optimization	Est.Clk(ns)	Latency	Interval	BRAM	DSP	FF	LUT	Throughput(KHz)
RSA_CRT	7.284	66965	66966	0	0	19823	18779	2.05

3.6 Final Version - CRT with Montgomery

The results at this stage are promising, achieving a speedup of 16.2 times with the number of FF increases by only 1.38 times. Furthermore, the number of LUT even decreases to 13% of the original design. Nevertheless, there is still an opportunity to apply the Montgomery algorithm to CRT and further optimize design latency. The key idea in CRT is splitting the 256-bit exponentiation by squaring function into two 128-bit computations. Through prior optimizations, we refined the exponentiation by squaring function into an optimized Montgomery implementation. By integrating this with CRT, we achieve a fully optimized result. While there are numerous intricate implementation details required to ensure functionality and timing compliance, these can be found in our GitHub repository. Table 8 presents the synthesis report for this implementation. By integrating CRT with the Montgomery algorithm and replacing multiplication with the modulo product function, we have developed a high-performance RSA decoding design that balances computation speed and resource efficiency.

Table 8: Synthesis result and throughput of RSA256 with CRT and Montgomery

		Clock Cycles		Utilization				
Optimization	Est.Clk(ns)	Latency	Interval	BRAM	DSP	FF	LUT	Throughput(KHz)
RSA_CRT_Mont	7.284	35729	35730	0	0	17633	14211	3.842

3.7 DEMO

This DEMO is designed to validate the correctness of an RSA decryption algorithm implemented on FPGA hardware. It uses Jupyternotebook to interface with the the board-PYNQ-Z2. The testbench begins by generating the necessary RSA parameters: two large primes p and q, a public key exponent e, and the corresponding private key d. These parameters are used to compute the public modulus $n=p\times q$. Using these, the script generates random values and encrypts them using the public key. The encrypted input is then decrypted with the board.

In this project, MMIO (Memory-Mapped Input/Output) is chosen because of several reasons. First, it enables direct and precise communication with specific hardware registers, ensuring efficient control over the input and output data required for decryption. Additionally, the simplicity of the MMIO interface makes it straightforward to implement, debug, and maintain, which is particularly beneficial for cryptographic accelerators. Finally, its deterministic and reliable communication ensures accurate data placement and integrity, which are critical for the correctness and security of cryptographic computations.

To accommodate the memory bandwidth limitations of the PYNQ-Z2 board, which operates on a 32-bit interface, the 256-bit input data is divided into smaller 32-bit chunks prior to transmission. This segmentation ensures compatibility with the board's memory architecture and enables efficient data transfer. When retrieving the data from the board, the individual 32-bit chunks are carefully recombined to reconstruct the 256-bit computed result.

For the DEMO, randomly generated test cases were utilized to validate the functionality of the implementation. The computed results were compared against the golden outputs (the original data prior to encryption). The system successfully passed all test cases, confirming its correctness.

```
Result from FPGA: 167207299911554103370433520453501655914170375072753711787423386423862955674 86
Golden x: 167207299911554103370433520453501655914170375072753711787423386423862955674 86
Test 1: PASS - FPGA output matches golden output
Result from FPGA: 227327422588219480081939527350484009356872979332834070799153169575489092731 55
Golden x: 227327422588219480081939527350484009356872979332834070799153169575489092731 55
Test 2: PASS - FPGA output matches golden output
Result from FPGA: 978286130479641355239057196768482037085497001350078422408791266619001967894 6
Golden x: 978286130479641355239057196768482037085497001350078422408791266619001967894 6
Test 3: PASS - FPGA output matches golden output
Result from FPGA: 793218748970311516937631766641842599759449424015600235419190174950167998515 4
Golden x: 793218748970311516937631766641842599759449424015600235419190174950167998515 4
Test 4: PASS - FPGA output matches golden output
Result from FPGA: 268574709716993898952910687991773302324806931152338570083222547510113800604 89
Golden x: 268574709716993898952910687991773302324806931152338570083222547510113800604 89
Test 5: PASS - FPGA output matches golden output
```

Figure 1: Demo Results

4 Challenges and Future Work

In this project, we faced various implementation challenges, including handling long bitwidth integer representations and calculations, managing pipeline stages synthesized as black boxes, and carefully using pragmas to constrain the design according to our requirements. Among these challenges, managing the black box property of high-level synthesis was the most undesirable aspect for us as designers. Ensuring accurate timing is a critical concern in hardware design, but unfortunately, HLS does not provide detailed information about the critical path, and the schedule viewer offers limited help in this regard.

If we were to extend this project to a real-world application, more challenges would likely arise from high-level synthesis. Additionally, in this project, we disabled the use of DSPs to focus on showcasing the strength of the algorithms we applied. However, in practical scenarios, DSPs are powerful design resources. A future direction for this project could explore how to implement the RSA256 decoder using all available resources effectively, and how to manage them efficiently to unlock their full potential.

5 Conclusion

In summary, by utilizing the Montgomery algorithm and the Chinese Remainder Theorem, we significantly reduce the computation time for RSA decoders, especially beneficial for higher bitwidth. In scenarios where high latency in message decoding can be a critical issue, these techniques become increasingly valuable.

Additionally, this project highlights the advantages of high-level synthesis: enabling rapid prototyping and ensuring that ideas are on the right track. It offers valuable insights into how promising a hardware description language approach can be in developing efficient and optimized hardware solutions.

References

[Bri90] Ernest F. Brickell. A survey of hardware implementations of rsa. In Gilles Brassard, editor, *Advances in Cryptology — CRYPTO' 89 Proceedings*, pages 368–370, New York, NY, 1990. Springer New York.

[SIN11] Bo Song, Yasuaki Ito, and Koji Nakano. Crt-based dsp decryption using montgomery modular multiplication on the fpga. In *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, pages 532–541, 2011.

[WHW01] Chung-Hsien Wu, Jin-Hua Hong, and Cheng-Wen Wu. Rsa cryptosystem design based on the chinese remainder theorem. In *Proceedings of the ASP-DAC 2001. Asia and South Pacific Design Automation Conference 2001 (Cat. No.01EX455)*, pages 391–395, 2001.