Design of Digital Platforms - Final report - Group 31

Thomas Debelle
ESAT
KU Leuven
Leuven, Belgium
thomas.debelle@student.kuleuven.be

Sjouke Spijkerman
ESAT
KU Leuven
Leuven, Belgium
sjouke.spijkerman@student.kuleuven.be

***Abstract*—This report presents the multiple design steps for a specific design of the RSA algorithm using a Co-Design implementation of the Pynq Z-2 FPGA in Verilog and C code combined with the ARM assembly. The report will elaborate on design choices and provide further analyses and metrics on the implementation. Finally, a method for testing the implementation and benchmarking is proposed.**

I. Design & Motivation

The desired implementation of the RSA algorithm is fast and flexible. A fast algorithm is desirable to maximize the hardware implementation. Furthermore, a flexible algorithm can adapt to the various needs of the user and is considered more resilient to errors. For instance, the adder is equipped with registers to save user inputs throughout the entire addition, even if the user changes the inputs of the adder after the start of operation. Flexibility also translates into resilient code that can easily be replaced and maintained. To guarantee this, robust software practices are utilized to achieve efficient, readable, and adjustable code.

II. Architecture

A. Adder

The implementation of the RSA algorithm uses a 514-bit wide 3 clock cycle adder with a sufficient Worst Negative Slack (WNS) and fast operation. Moreover, the Verilog implementation of the adder is highly flexible, which allows quick adjustments between 64 bits to 514 bits width adder to find the best compromise between speed and timing constraints (see Fig. 4, please refer to the appendices IX for bigger figures). For the full addition, a multi-precision adder is used to split the addition into two parts, the LSB and the MSB. The critical path starts at `regB` and passes through the inverter, multiplexer (MUX), and 514-bit wide adder until it reaches `regresult_D`.

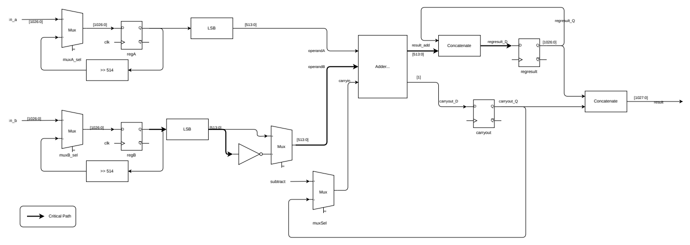


Fig. 1: Adder Hardware Implementation

The largest impact on the WNS is caused by the chain 4-bit carry adder. On another note, the net delay has to be considered when multiple adders are generated on the board, making it as important as the logic delay. Moreover, the concatenation of the carry with the result adds an extra constraint to the implementation. More information on this is provided in Section IV.

B. Montgomery Multiplier

To minimize the area used by the Montgomery multiplier, only a single adder is implemented. However, this approach significantly complicates the signal wiring. The design evolved through several iterations, starting with a large, slow multiplier and gradually refining it into a more compact and faster version, as shown in Fig. 5. The critical path in the Montgomery implementation begins at the adder's output, passes through the 2-bit shifter and the 2 MUXes, and concludes at the `operand_A` wire.

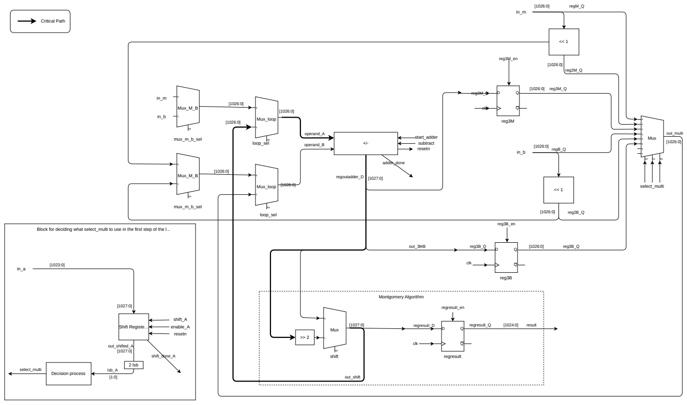


Fig. 2: Montgomery Multiplier Hardware Implementation

C. RSA

The RSA is split between two parts. The hardware part performs the Montgomery multiplications while the software part copes with the interfacing. In addition, the software handles the data and runs the power ladder algorithm. Since the Direct Access Memory (DMA) will output the value in memory for only one clock cycle, four large registers store the various inputs needed for the Montgomery multiplication.

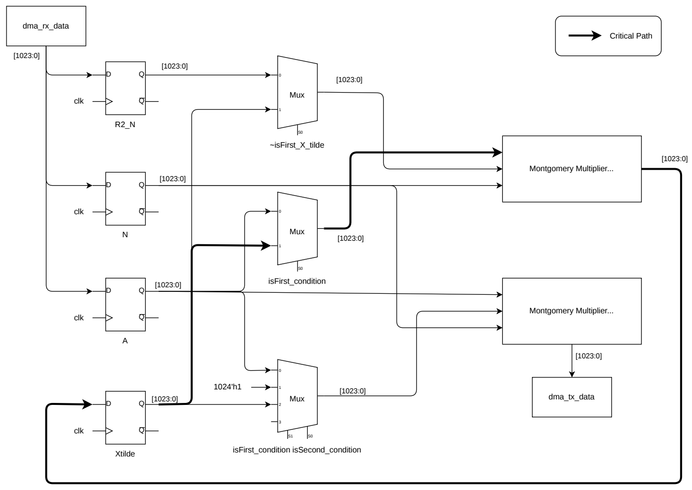


Fig. 3: RSA Hardware Implementation

To reduce this amount of MUXes and the complexity of the hardware, N, $R2\_N$, and A are transferred to a register and X_tilde is maintained in a register without leaving the FPGA. By keeping X_tilde in hardware, errors are less prone to occur and the complexity of the hardware is decreased.
One drawback of this implementation is the added overhead with A. When running the power ladder algorithm, the transmission and reception of the 1024-bit A values add a measurable overhead as can be observed in Fig. 7. This value could have remained in memory but would have, on the other hand, caused a higher complexity and more LUTs usage due to added MUXes.

III. Area & Number of Cycles

A. Adder

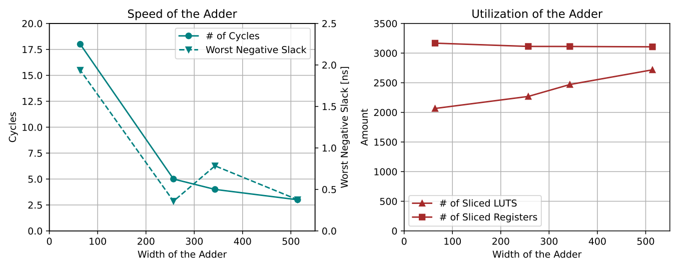


Fig. 4: Performance of the Adder

As anticipated, the speed decreases as the adder width increases, while the use of LUTs goes up. On the Pynq Z-2, the LUTs usage is critical, since each logic slice is composed of 4 6-input LUTs and 8 flip-flops [1]. Reducing LUT usage was, therefore, one of the main targets of this project. However, using larger inputs and thus bigger MUXes impacted the amount of sliced LUTs.
Relevant to note is the linear decrease of the WNS and consequently its sudden jump. This, however, is a result of Vivado using another implementation, since a bigger adder width will introduce more constraints on the implementation run and its WNS. Further details are provided in Section IV.

B. Montgomery Multiplier

Initially, a 28.230 cycles Montgomery multiplication in combination with a 64-bit, 18-cycle adder was realized. This design was characterized by a WNS of 0.601, 7261 sliced LUTs, and 8243 sliced registers. Currently, the Montgomery multiplication has a duration of 3097 cycles with a slight increase in sliced LUTs and registers, translating into an acceleration of approximately $911, 5\%$ without compromising in the size and respecting the timing constraint. In addition, the WNS leaves some margin to overclock the FPGA resulting in even faster operation.

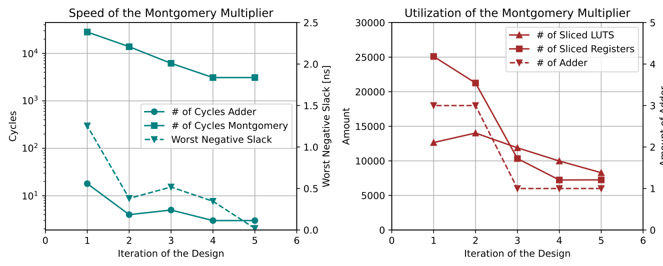


Fig. 5: Performance of the Montgomery Multiplier

C. RSA - Montgomery Exponentiation

The initial design approach involved directly implementing the full power ladder in hardware to minimize the overhead between software and hardware. However, this approach quickly revealed significant design complexities and challenges, such as debugging the hardware via software. This led to a reevaluation of the strategy. The second design choice was to use the hardware to compute the Montgomery multiplication only. In this manner, the hardware would handle the data transfer and the computation.
The integration of modules proved to be a difficult design challenge for the WNS. This effectively led to the return to the fourth iteration of the Montgomery multiplier.

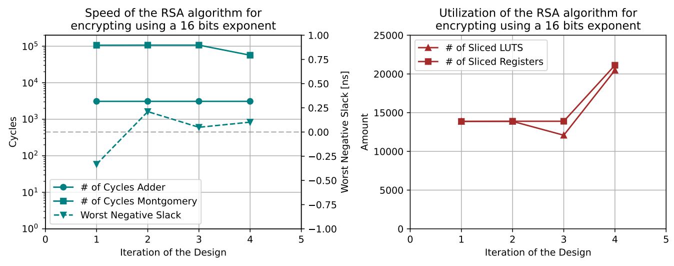


Fig. 6: Performance of the RSA - Montgomery Exponentiation

After successfully implementing the algorithm using one Montgomery multiplier, a design with two parallel multipliers was created. This design reached an increase in speed of 88.7% with only an increase of LUT usage of 69.4% and register usage by 52.1%. Using two Montgomery multipliers simplified certain multiplexers, which explains why the observed increase in area was not as equal to the increase in speed.

IV. Implementation and Strategies for the FPGA

Transitioning from one to two Montgomery multipliers required more than simply initializing another module in Verilog. Although the most efficient Montgomery multiplier implementation used less than 10% of the FPGA's resources, strict timing and physical constraints posed significant challenges. Synthesizing the same two Montgomery multipliers proved infeasible because Vivado required the carry adder and associated components to be placed so closely that locality became insufficient for two multipliers. A previous version of the multiplier, offering better WNS performance but consuming more space, was selected instead.
As mentioned earlier, locality and timing emerged as the primary constraints, and the default synthesis and implementation strategies proved inadequate. Drawing on test results conducted by Alberto L. [2], it was identified that the most effective strategies for the design were `Flow_AlternateRoutability` for synthesis and `Performance_ExplorePostRoutePhysOpt` for implementation. This strategy achieved a WNS of 0.102 ns, providing some margin for increasing the FPGA clock frequency. An alternative approach involved registering the carry and result of the adder to reduce FPGA constraints and allow Vivado more flexibility in component placement. However, adding a register to the adder would have resulted in a speed reduction of approximately 33.33%, making it an unsuitable option. Instead, the clock frequency of the FPGA could be slightly increased to around 101 MHz.

V. Codesign

A. Boundaries

For clarity, the algorithm is divided into 3 parts: An all-software part, an all-hardware part, and a mixed interfacing part that connects software and hardware with the API. Values are assigned in the C code, which can include test vectors or actual text strings, as will be explained in subsection V-B. Then N and $R2\_N$ are loaded into the FPGA using Algorithm 1 before executing the actual encryption. The encryption algorithm, detailed in Algorithm 2, is then executed, involving the transfer of the value A to and from the DMA at each step.

B. Interfaces

First, the correct addresses are set on the DMA before sending a command, as outlined in Table I.
During the power ladder phase (Table II), the software and hardware interface continuously, with the results of the Montgomery multiplication being loaded and received at each step. A notable feature of the implementation is the added interface layer. The design extends beyond a proof of concept that works with test vectors and is fully functional and capable of encrypting and decrypting user-defined messages. A set of functions was developed that allows users to input any string of text, with the program encrypting each block of text. This approach enhances the flexibility of the implementation and brings it closer to real-world applications, enabling interaction with users in everyday situations.

1) Command & Data transfer:

loading_command	RXADDR	TXADDR
$0b1001$	N	///
$0b1011$	R2_N	///

TABLE I: Status of the Registers for the loading Phase

Command	RXADDR	TXADDR
$0b001$	M	///
$0b0011$	A	A
$0b0101$	A	A
$0b0111$	A	A

TABLE II: Status of the Registers for the Power Ladder

Algorithm 1 Loading Variables

```
1:  procedure LOAD_DATA(N,R2_N)
2:      RX_ADDR ← address of N
3:      loading_command ← $8 + 1$
4:      while busy
5:      loading_command ← 0
6:
7:      RX_ADDR ← address of R2_N
8:      loading_command ← $8 + 3$
9:      while busy
10:     loading_command ← 0
```

Algorithm 2 Power Ladder

```
1:  procedure POWER_LADDER(A,R_N,M,X_tilde)
2:      RX_ADDR ← address of M
3:      command ← 1
4:      while busy
5:      command ← 0
6:
7:      for $(i = 0; i < 32; i++)$ do
8:          A[i] = R_N[i]
9:
10:     for $(i = 0; i <$ exponent_length$; i++)$ do
11:         RX_ADDR ← address of A
12:         TX_ADDR ← address of A
13:         if bit(exponent, exponent_length $- i - 1$) then
14:             command ← 3
15:             while busy
16:             command ← 0
17:         else
18:             command ← 5
19:             while busy
20:             command ← 0
21:
22:     RX_ADDR ← address of A
23:     TX_ADDR ← address of A
24:     command ← 7
25:     while busy
26:     command ← 0
```

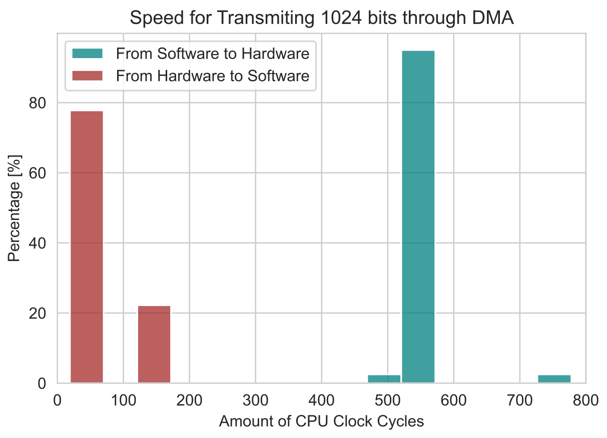


Fig. 7: Interface Overhead Measurements

2) Interface Overhead: It takes around 50 CPU clock cycles to receive data from the FPGA through the DMA after the hardware finishes computing the result. And for sending data around 556 CPU clock cycles are required to finish the operation. It has to be noted that the FPGA runs at 100 MHz and the ARM CPU at 650 MHz at base clock [1]. That translates into 556 clock cycles in the CPU being equivalent to around 86 clock cycles for the FPGA.
Completing a whole operation of sending, computing the Montgomery multiplication result, and receiving its result requires on average $86 + 3097 + 8 = 3191$ FPGA clock cycles or 20741 CPU clock cycles. This result correlates with the result for encrypting a message using a 16-bit key. We need around $20741 \cdot (1+16+1) \approx 373 \cdot 10^3$ and we measured $\sim 381 \cdot 10^3$ on the board. This difference of $7 \cdot 10^3$ cycles can be explained due to extra work from the CPU due to the loops or the assignments of the variables.

VI. Test strategy

The majority of components were broken down into smaller modules to test them individually and ensure proper operation. For the adder, over 20 tests were conducted, covering various addition and subtraction cases, as well as edge cases such as negative subtraction and overflow. This interconnected and systematic approach kept the design and testing process clear and manageable. A similar procedure was followed for the Montgomery multiplication. Various test inputs, generated by `testvectors.py`, were provided to the Montgomery multiplier, and the algorithm was also recreated in Python to monitor each step. The Python script assisted us in debugging by identifying the locations of issues. Multiple tests were run to ensure all conditions were explored, maximizing coverage testing.
For the RSA implementation, the software part was first designed using five different inputs generated by the command `python3 testvectors.py rsa 2024.X`, where X ranges from 1 to 5. This C code was then translated into Verilog to replicate its behavior. The four provided tasks in `tb_rsa_warmup.v` were heavily utilized to run tests and ensure close alignment with the C code. Initial debugging was performed using the testbenches, followed by debugging in the C code with the use of monitoring registers, $rout1$ to $rout7$. A small library of functions was developed to compare the actual results with the expected ones. Finally, the software hardware implementation was tested using actual examples of messages and exponents. To facilitate the comparison of string and array values, several functions were implemented to print decoded strings and compare arrays.

VII. Performance, limitation and improvements

A. The Power of the Co-Design

Fig. 8 illustrates the true power of a co-design approach for cryptographic applications. Even when using ARM assembly to get as close as possible to machine code, the speed of an FPGA implementation of the Montgomery multiplication cannot be matched. Additionally, the overhead introduced by this co-design algorithm is minimal in comparison to the speed gain achieved.

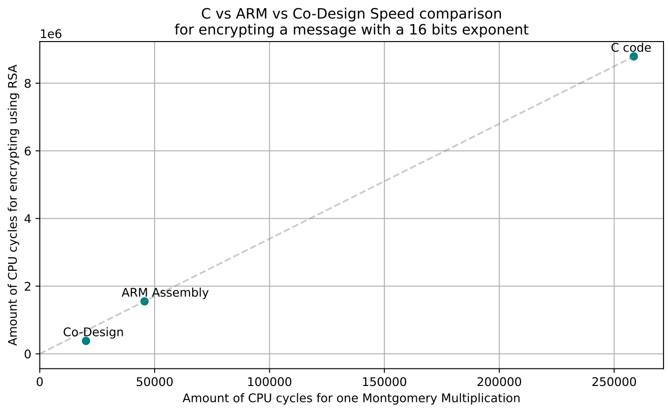


Fig. 8: Speed comparison using different methods

B. The Chinese Remainder Theorem

A major drawback of the RSA algorithm is the high cost of decrypting a message, which makes the receiver vulnerable to denial-of-service (DOS) attacks. To address this, modern RSA implementations use the Chinese Remainder Theorem (RSA-CRT) to accelerate the decryption process [3] [4] [5]. To implement the RSA-CRT algorithm, three new inputs—d_P, d_Q, and q_{inv}—are required.

Algorithm 3 RSA-CRT Decryption

```
1: function DECRYPT_DATA(Ct,d,p,q) return m
2:     $d_P \leftarrow d \quad (mod \quad p-1)$
3:     $d_Q \leftarrow d \quad (mod \quad q-1)$
4:     $q_{inv} = q^{-1} \quad (mod \quad p)$
5:
6:     $m_1 = Ct^{d_P} \quad (mod \quad p)$
7:     $m_2 = Ct^{d_Q} \quad (mod \quad q)$
8:     $h = q_{inv}(m_1 - m_2) \quad (mod \quad p)$
9:     $m = m_2 + h \cdot q$
```

Due to time and logistical constraints, the development and testing of this version of the algorithm on the FPGA could not be completed. While it remains in the source code, further fine-tuning is needed before it is fully operational. However, experiments were conducted, and the algorithm was implemented in Python within `testvectors.py` (also included in the source code). The performance was benchmarked both with and without the use of CRT. Improvements in decryption speed were observed, though the added complexity and the initial computations required for the faster RSA version must be considered.

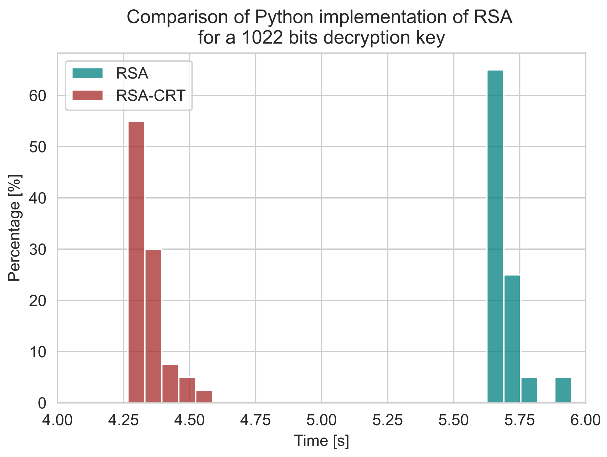


Fig. 9: Interface Overhead Measurements

VIII. Conclusion

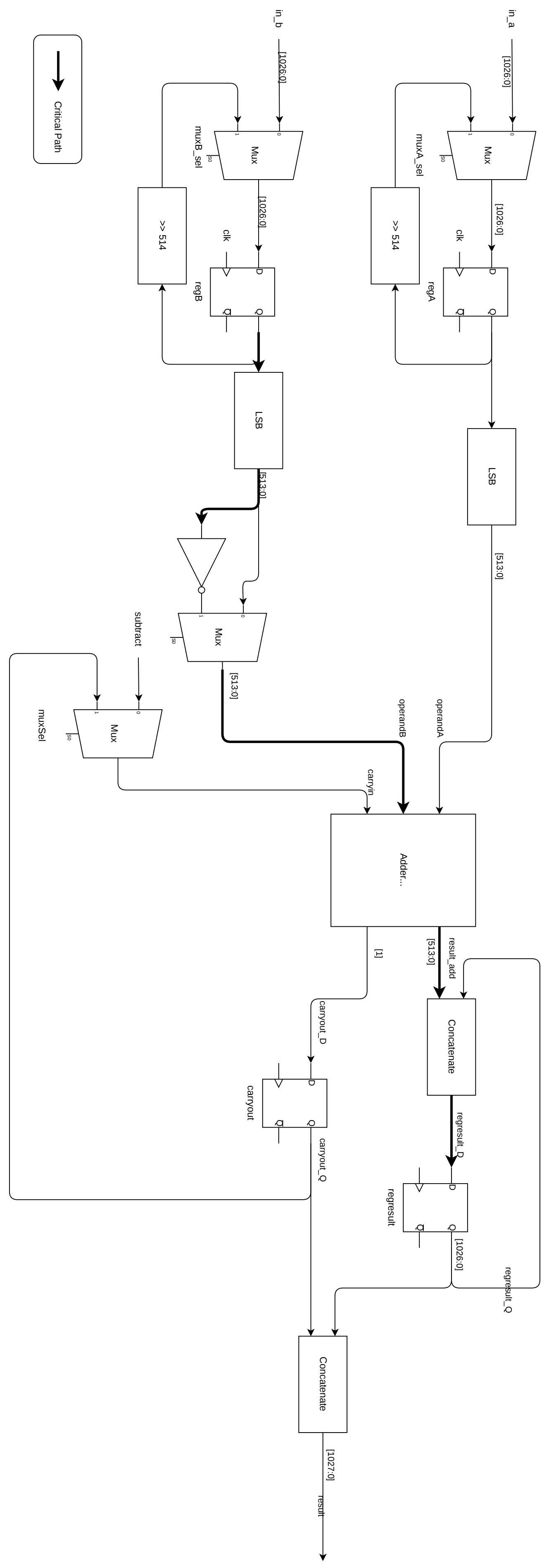
In summary, a functional implementation of the RSA algorithm using the power ladder and Montgomery representation approach using a co-design strategy is realized. This work is the result of numerous iterations and careful consideration to design the most efficient implementation while maximizing speed and space.
This project highlighted the importance of space and its impact on speed, demonstrating how clever mathematics can enable the creation of smarter, faster hardware. The goals were not only achieved but also exceeded, driven by a constant pursuit of incremental improvements.
While there is no ”free lunch,” strategic design decisions and effective optimization techniques have been key to achieving these outcomes.

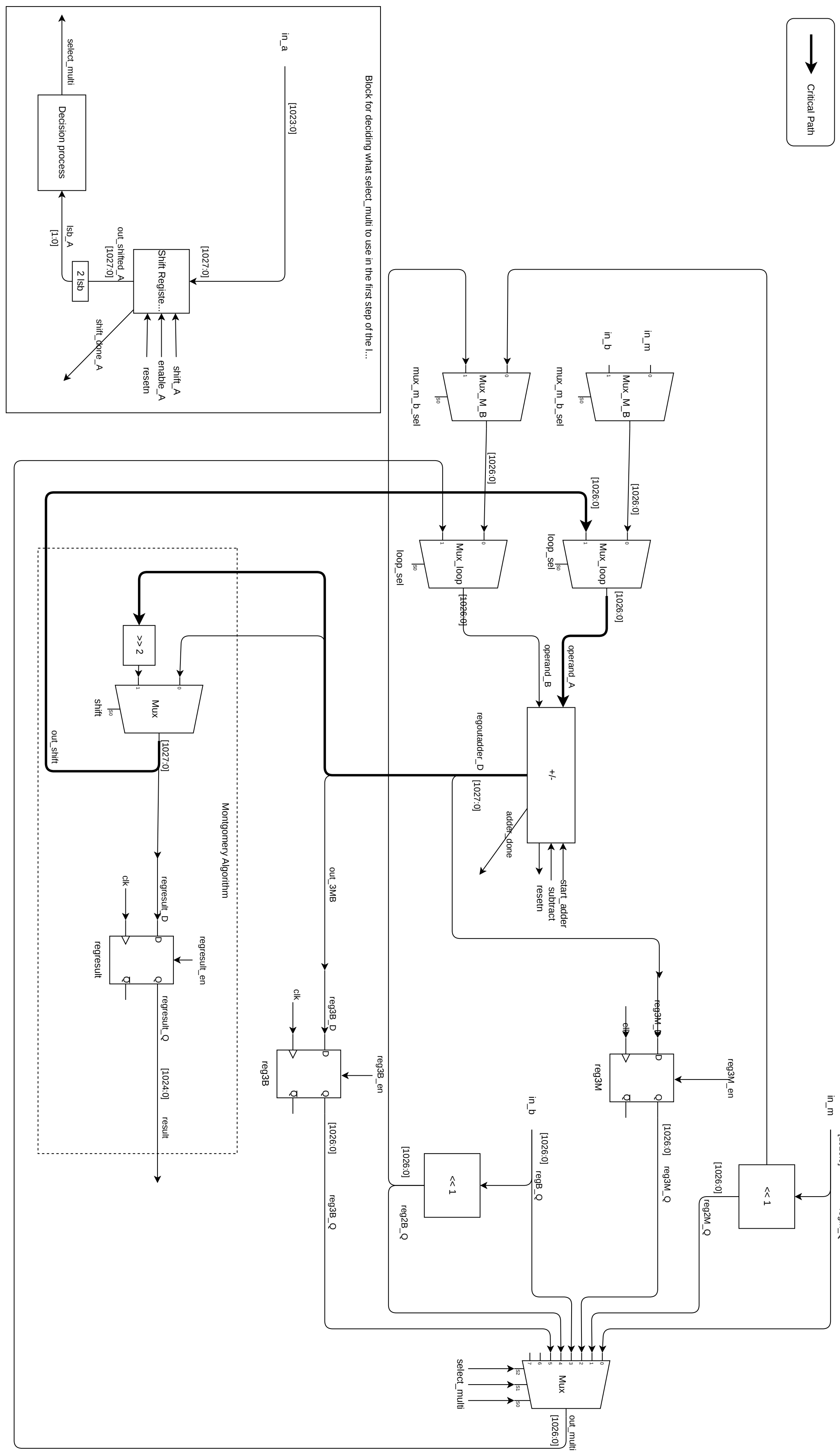
References

[1] “https://www.mouser.com/datasheet/2/744/pynqz2_user_manual_v1_0-1525725.pdf.”
[2] A. L, “What are the Best Vivado Synthesis and Implementation Strategies???,” Nov. 2017.
[3] “Chinese remainder theorem,” Dec. 2024. Page Version ID: 1260643623.
[4] M. Radcliffe, “Math 127: Chinese Remainder Theorem,”
[5] G. N. Shinde and H. S. Fadewar, “Faster RSA Algorithm for Decryption Using Chinese Remainder Theorem,”
[6] “RSA (cryptosystem),” Nov. 2024. Page Version ID: 1260272229.
[7] M. Joye and S.-M. Yen, “The Montgomery Powering Ladder,” in *Cryptographic Hardware and Embedded Systems - CHES 2002* (G. Goos, J. Hartmanis, J. Van Leeuwen, B. S. Kaliski, K. Koç, and C. Paar, eds.), vol. 2523, pp. 291–302, Berlin, Heidelberg: Springer Berlin Heidelberg, 2003. Series Title: Lecture Notes in Computer Science.
[8] “Minimizing FPGA Resource Utilization.”
[9] W. Dehaene, I. Verbauwhede, M. Verhelst, and J. Verplancke, “Design of Digital Platforms,” 2024.

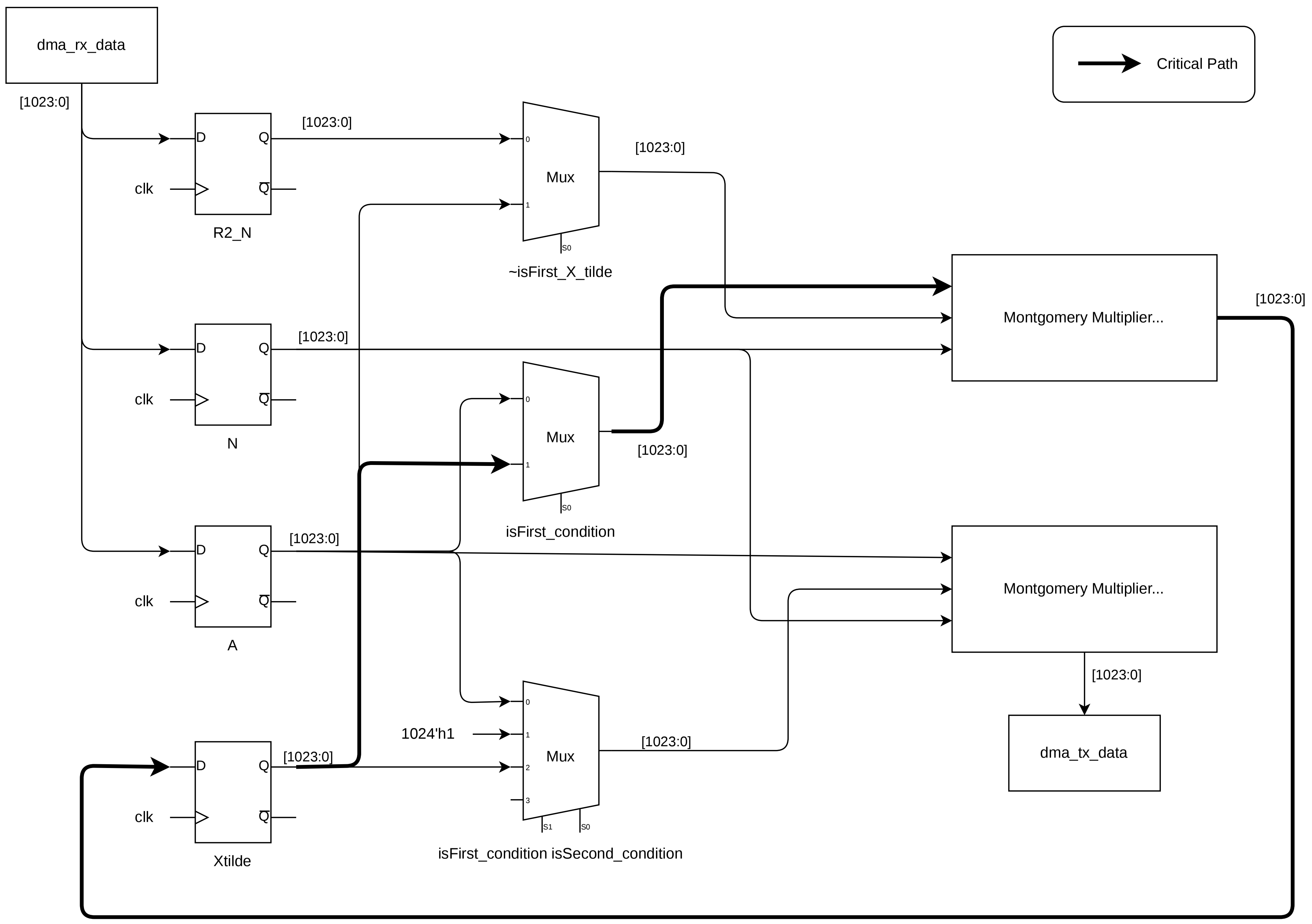
IX. Appendices

In the next few pages, larger illustrations of Fig. 1, 2 and 3 can be found.
The source code of this work can be found at this Github repository under a CC BY-NC-ND 4.0 license.


Critical Path
in_a
in_b
[1026:0]
Mux
muxA_sel
muxB_sel
clk
regA
regB
D
Q
>> 514
LSB
[513:0]
operandA
operandB
subtract
muxSel
carryin
Adder...
[1]
result_add
Concatenate
carryout_D
carryout
carryout_Q
regresult_D
regresult
regresult_Q
[1027:0]
result



Critical Path
Block for deciding what select_multi to use in the first step of the l...
in_a
[1023:0]
[1027:0]
Shift Registe...
shift_A
enable_A
resetn
shift_done_A
out_shifted_A
[1027:0]
2 lsb
lsb_A
[1:0]
Decision process
select_multi
in_m
in_b
Mux_M_B
mux_m_b_sel
[1026:0]
Mux_loop
loop_sel
[1026:0]
operand_A
operand_B
+/-
regoutadder_D
[1027:0]
adder_done
start_adder
subtract
resetn
>> 2
Mux
shift
out_shift
[1027:0]
Montgomery Algorithm
regresult_D
clk
regresult_en
regresult
regresult_Q
[1024:0]
result
out_3MB
reg3B_D
clk
reg3B_en
reg3B
[1026:0]
reg3B_Q
reg3M_D
clk
reg3M_en
reg3M
[1026:0]
reg3M_Q
in_b
[1026:0]
regB_Q
<< 1
[1026:0]
reg2B_Q
in_m
[1026:0]
regM_Q
<< 1
[1026:0]
reg2M_Q
Mux
select_multi
out_multi
[1026:0]


dma_rx_data
[1023:0]
D
Q
clk
Q̄
R2_N
N
A
Xtilde
[1023:0]
Mux
0
1
S0
~isFirst_X_tilde
isFirst_condition
1024'h1
2
3
S1
isFirst_condition isSecond_condition
Montgomery Multiplier...
Montgomery Multiplier...
dma_tx_data
Critical Path