Project 3 Theory of Operation

Introduction

For project 3, we designed a functional hardware encryption engine, implemented using the Spartan-6-driven Nexys3 Development board. Throughout the design process, maximum clock speed was consistently the primary goal of all code written and all optimizations performed.

Overview

- Functionally verified 128-bit encryption/decryption
- Maximum operational frequency of 175MHZ
- Encrypts/Decrypts 7 custom generated 128-bit "Messages"
- Implements Montgomery Math algorithms for exponentiation and multiplication.
- Uses pre-calculated Nr = 2^{2n} mod M values for Montgomery Math
- Python script for verifying algorithm and calculating Nr values for any bit width
- Encryption in approximately 285us for 128 bits (~50,000 clock cycles at 175MHZ)

Development

Stage 1

We began stage 1 by compiling and synthesizing the distributed code without any changes. The longest path was quickly identified through the modulus operation in the RSA module [ENCRIPTION.v]. This module implements the following RSA encryption function:

$$Y = X^E \bmod M$$

By observing that

$$AB \text{ Mod } M= [(A \bmod M) * (B \text{ Mod } M)] \text{ Mod } M$$

the encryption output can be obtained after executing the mod and multiplication operation in about E times.

The original design implements these operations using [*] and [%] logic at 100MHz. At 16bit data width, it was not working properly due to timing violation. To overcome this, a Spartan-6 DCM was directly instantiated in the top-level module [s6fpga_rsa.v], and the maximum division was performed (/16), yielding a 6.25MHZ clock signal. The 6.26 MHZ clock signal was used to drive all parts of the project except for the debouncer and 7-segment driver, which retained the original, 100MHZ clock, as illustrated below.

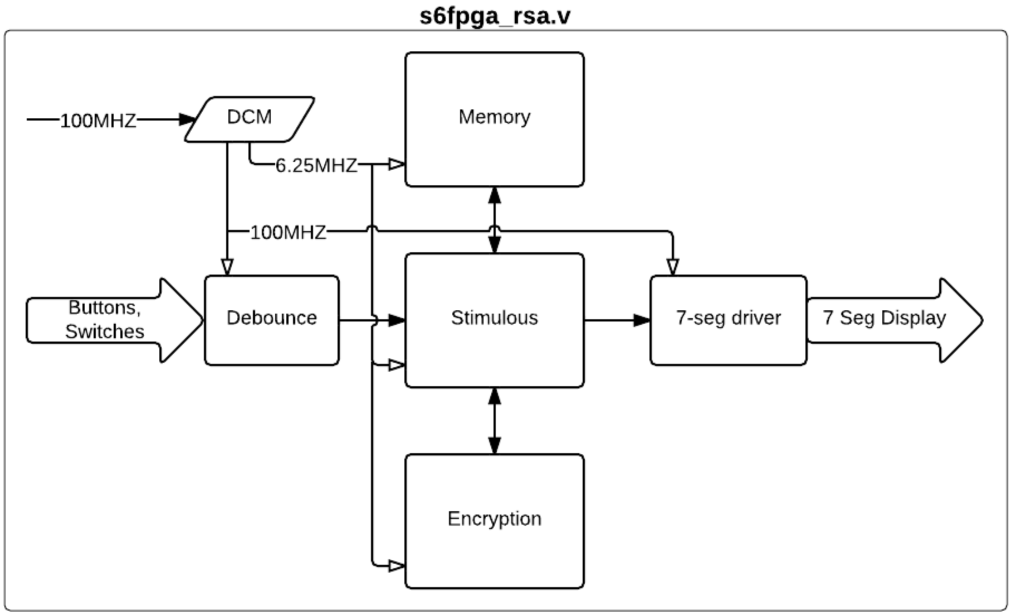

Figure 1: Stage 1 Design

Stage 1 was tested on the board and performed encryption and decryption at 16 bits with no perceivable delay during operation.

Stage 2

Having identified the modulus operation [%] as being responsible for the longest path in the project, we used Xilinx CoreGen to produce the Divider core v4.0 LogiCORETM IP, and instantiated it into the project before increasing to 32-bit operation. Additional control signals were required to properly use the divider IP, namely `tvalid` signals to signify when the divisor and dividend are valid, and a `dout` signal to signify when the division operation is complete and the output is valid.

The RSA module [ENCRIPTION.v] was rewritten as a very explicit, synchronous FSM. The overall functionality of the RSA module did not change drastically, but the FSM was created so that adding and removing additional control signals would be simplified if needed. The main functionality change made to the project during stage 2 is that when a modulus operation needs to be performed, the FSM enters a state “MOD” wherein `tvalid` is asserted for both inputs to the divider IP. The FSM will stay in this state until receiving the `dout` signal back from the divider.

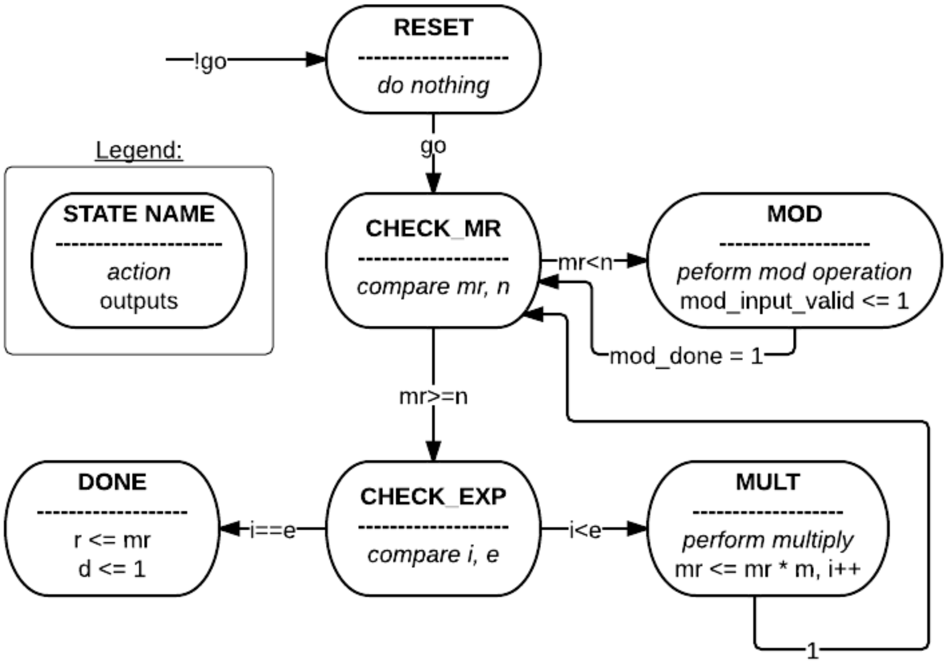

Figure 2: RSA FSM State Transition Diagram

After the required changes were made to the rest of the project to facilitate 32-bit operation (updating the exponent and modulus values contained in [STIMULOUS.v]), the project was verified on the hardware. The results produced were correct but as anticipated, the receive algorithm took a long time (3-4) minutes to complete, even after increasing the clock speed for the memory, encryption, and stimulus modules to 50MHZ. This is because increasing the exponent essentially increases the iterations through the above state machine. Combined with the fact that the Divider Generator requires 60+ clock cycles for each division, there were many billions of operations clock cycles required to produce a result, so even running at 50MHZ, operation was slow.

Stage 3

[The decision was made to pursue 128-bit encryption rather than complete the optional stage 3.]

Stage 4

To remedy the long decryption time, we implemented algorithms to perform Montgomery Exponentiation (ME) and Montgomery Multiply[1] (MM), as processing time using these algorithms increases linearly with bit width rather than exponentially. Again we used a discrete FSM to control advancement through these algorithms. With these techniques, clock speeds of 100MHZ were achieved for 32-bit RSA and verified functional in hardware, thus the need to divide the clock (see figure 1) was removed. The state transition diagrams for the modified RSA module and Montgomery Multiply module are below.

[1] Fry, J., and Langhammer, M. RSA & Public Key Cryptography in FPGAs. Tech. rep., Altera Corporation, 2005., Sections 3.1-3.2

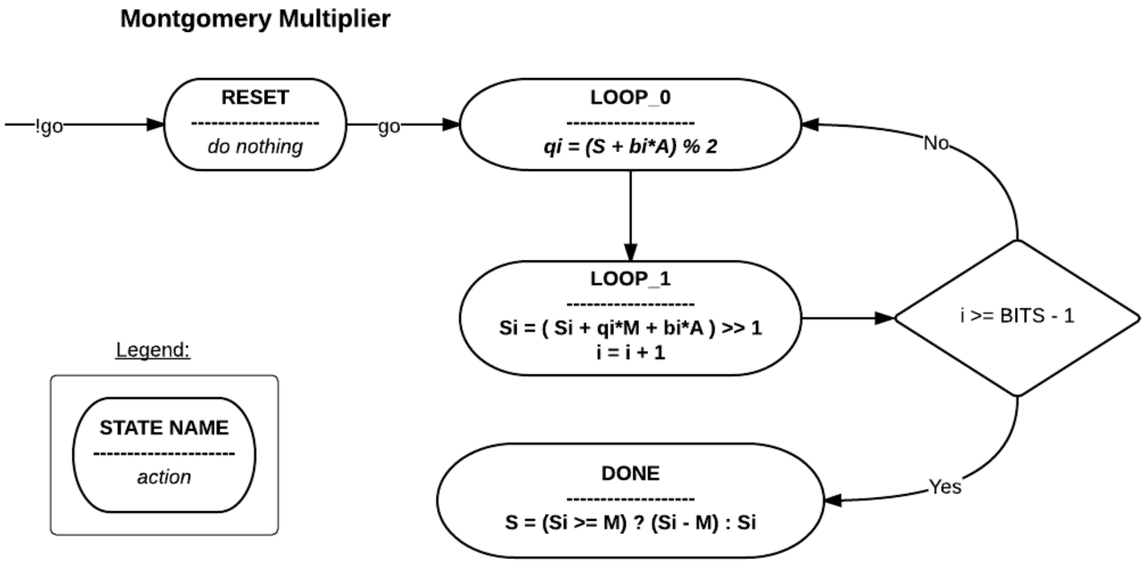

Figure 3: Montgomery Multiplier FSM Transition Diagram

This algorithm replaces simple operation

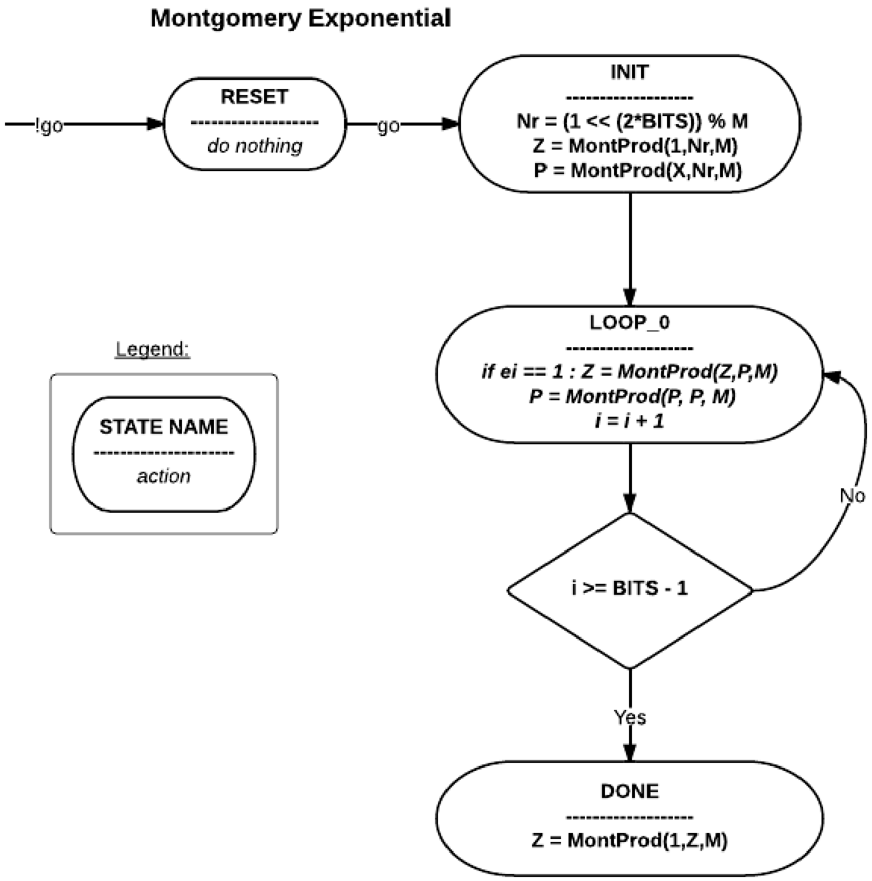

Figure 4: Montgomery Exponentiation FSM State Transition Diagram

The advantages of this algorithm are

- Execution runs in fixed n x n computational cycles, where n is bit width (vs E cycles in previous approach, E's max range is up to 2^n-1)
- Replace the costly mod and multiplication by rather cheap shift, add/subtract and comparison operation.

To further reduce computational cycles, we utilized two MM units in ME module since each INIT and LOOP step requires two Montgomery productions (fig.6).

One issue we encountered during translating the algorithm (suggested in Fry 's paper[1]) was the missing step in MM (final subtraction). After discover this from the other paper[2,] we quickly verified its correct behavior and apply changes to our verilog code.

Once the algorithm has been implemented, completing stage 4 was trivial. All Montgomery math code written during this stage was fully parameterized, so to get the project running at 64 bits, the only required change was to change the global constants (for bit width), modify the stimulus file (to use 64-bit exponents and modulus), and resynthesize. The project was functionally verified on the hardware at a clock frequency of 100MHZ.

Notice from the above transition diagram, the longest path is at DONE step of MM with n bit comparator and subtractor.

Stage 5

128-bit initial results

The transition to 128-bit was also simplified by the design changes in stage 4, and again only required we change the global constants, modify the stimulus, and resynthesize. Since encrypted/decrypted messages were not provided for 128 bits, a python script was written to implement Montgomery algorithm for calculating (X^E)%M (see appendix). This way, we could verify the results of encryption/decryption produced by the board, rather than relying on the hard-coded values in the stimulus file.

The main obstacle in this attempt was to make the decision on how to find Nr = 2^{2n} mod M value during the initialization state of ME. We originally utilized a fast MOD module (shift and subtraction). At 128bit encryption, a 256 bit MOD module is required (i.e. 256-bit subtractor is needed). With this approach, we encountered Place and Route error with unroutable signals. Since the *Nr* constant only needs to be calculated once[2] for a given Modulo M, and this can be done pre-runtime, we decided to follow the advice in the above paper to calculate and use a constant *Nr* and thus avoid the need to perform subtraction. Pre-calculated Nr can be stored in ROM/RAM or fed from additional input port. In this implementation, we embedded single Nr value for the currently used 128-bit key.

Again, the project was functionally verified on the hardware at 100MHZ.

256-bit attempt

The transition to 256-bit functionality did not go as smooth as previous stages. Though the chip utilization was low (<50%), we were ultimately unable to route the design and/or achieve timing closure.

On our first attempt, we simply changed the global constants, modified the stimulus file, and resynthesized. Place and Route failed repeatedly because of an unrouteable signal. We reduced the

[2] Iput Heri, K.; Asep Bagja, N.; Purba, R.S.; Adiono, T.; , "Very fast pipelined RSA architecture based on Montgomery's algorithm," *Electrical Engineering and Informatics, 2009. ICEEI '09. International Conference on* , vol.02, no., pp.491-495, 5-7 Aug. 2009
doi: 10.1109/ICEEI.2009.5254686
URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5254686&isnumber=5254684

clock speed to 6.25MHZ, but the design was still unrouteable, due to a single signal in a 256-bit subtraction performed during the modulus operation.

Only using the most aggressive congestion reduction strategies in SmartExplorer yielded routeable designs, however these results suffered from very high timing scores (50,000+) that we were certain could not be simply optimized away using design strategy alterations. We suspect many of the problems encountered in this stage stemmed from limited resources on the chip (other than LUTs and FFs).

Final Design

Process

We retained the pre-calculated *Nr* methodology adopted in our 128-bit attempt, and wrote 7 new 128-bit messages. We used the Python script to generate *Nr* at 128-bits and hard coded it into the design, eliminating the need for subtraction. We increased the clock speed until we began to have timing problems, then used an iterative process of altering clock speed, design constraints, and synthesis options, (combined with SmartExplorer to test multiple PAR attempts at once) to ultimately settle at 175MHZ as the maximum design frequency. Our design is functionally verified on the Nexys3 Board at 128 bits. Ultimately, we used the DCM to multiply the system clock rather than divide it as we did in stage 1.

Functional Description

The top-level view of the design follows in figure 5. It remained relatively unchanged throughout the project, with the exception of using the DCM as a multiplier in the final implementation, rather than a divider, as in stage 1. The functionality of the Montgomery Product and Montgomery Exponentiation/Encryption algorithms remained unchanged from stage 2.

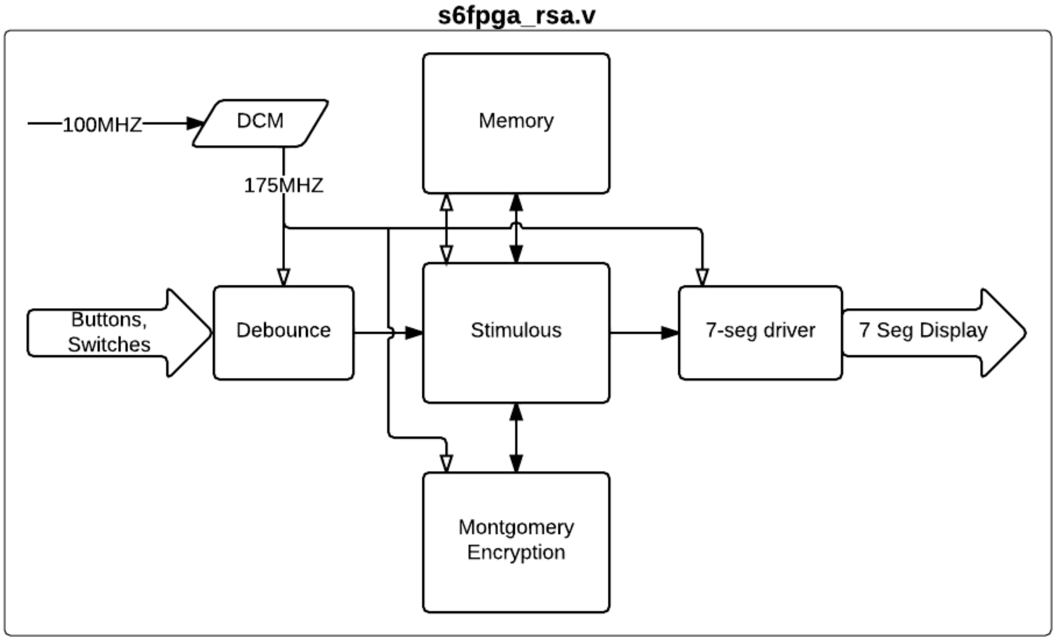

Figure 5: Final Design Top-Level Structure

Below is the architecture of ME encryption module

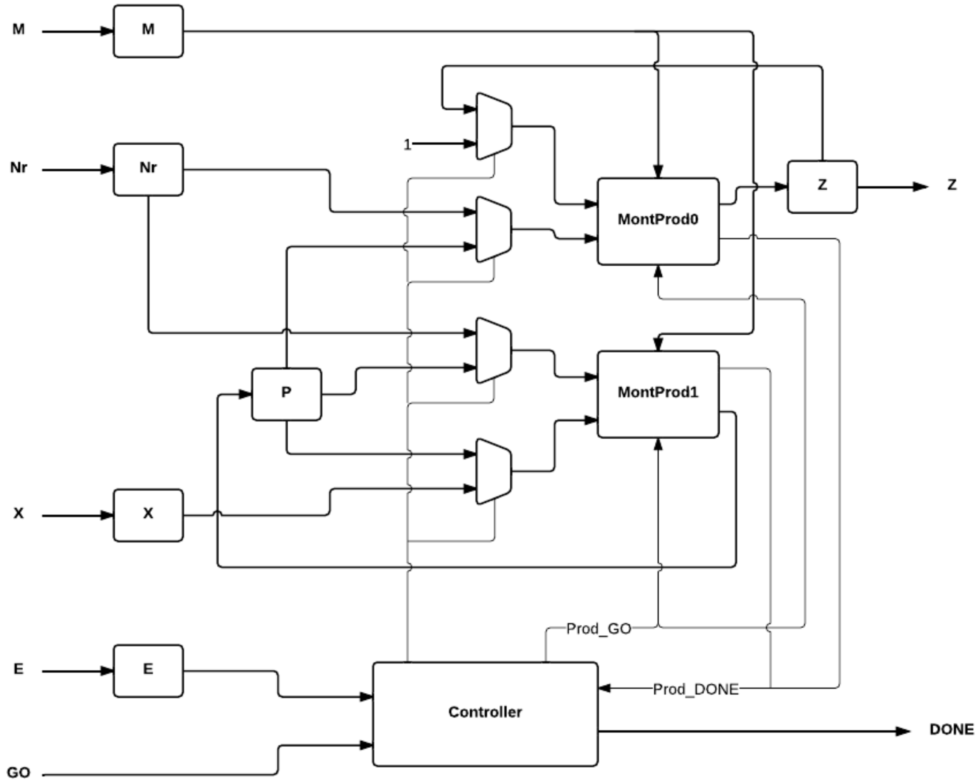

Figure 6- Montgomery Encryption Architecture

Performance

As noted above, the design is functional at 175MHZ. The clock period at this frequency is 5.714ns. Using simulation, we estimate the required clock cycles for the complete encryption or decryption at 128-bits to be roughly 50,000, yielding a completion time of approximately 285us. Our final design uses 11% of FPGA registers, and 30% LUTS, and the slowest path in the design is the path through the comparison function in the Montgomery Product module.

Discussion

The above result shows that our design meets the project requirements. However 128 bit is actually small key size in comparing with most current encryption system. We have made several attempts for getting higher bit size without success, as mentioned in stage 5. For further improvements, we envision the following tweaks on the current design:

1. Optimize our code for better area utilization (e.g., increased code parallelization for higher utilization of FPGA resources). From the synthesis and PAR report, we suspect one of our roadblocks is the long carry chain in the adder/subtrator. The design currently only occupies ~30% of FPGA resources, so it is likely possible to utilize a higher percentage of the resources and get better performance.

2. To avoid timing issue, we implemented fully pipelined states in our FSMs (MM and ME module). This however increases input to output latency. Further optimization can be made to remove redundant states for better performance.
3. As mentioned, the longest delay path in this implementation is the final comparison/subtraction of MM function. In [2] the authors proposed an optimize MM algorithm for eliminating this final step.