VLSI 2000 Design Contest: Student Entry

Finding Large Prime Numbers

(A Hardware-Software Codesign System)

Vaibhav Vaish **Ashish Kapoor**

(Senior Students)

Department of Computer Science & Engineering
Indian Institute of Technology, New Delhi

September, 1999

Contact Author: Vaibhav Vaish
Email: csu96173@cse.iitd.ernet.in
Address:
Department of Computer Science & Engineering,
Indian Institute of Technology,
Hauz Khas,
New Delhi 110 016

Phone: (11) 6861977 - 6861986
FAX: (11)6868765

Abstract

We describe a fast method to find primes of the form $2^p - 1$, by implementing special circuits to perform modulo $2^p - 1$ arithmetic, where p itself is an input to the circuit. We have followed a codesign approach. The primality verification is implemnented in hardware. The software component supplies inputs to the randomized verification algorithm.

Currently, we have a working implementation that uses the Xilinx[tm] 4010E FPGA and the 8051 microcontroller. In addition, we have simulated the FPGA portion extensively. We are working on running the FPGA at higher frequencies, and developing variants for specific applications.

Introduction

Large primes are required in almost all aspects of secure communication. Modern cryptography relies extensively on the computational hardness of problems like prime factorization to provide a secure communication channel. Public key cryptosystems, such as RSA require large primes to generate the necessary keys. For fast and secure communication, it becomes necessary to perform arithmetic operations modulo a prime number at high speed.

An interesting case arises when the (prime) number is of the form $2^n - 1$. From the results we describe in the next section, it will become apparent that this case lends itself to an elegant hardware implementation, that eliminates the requirement for doing division and computing remainders. In section 2 we describe the design of modules to perform addition, multiplication and exponentiation modulo $2^n - 1$. Here we describe how these modules can be utilised for verifying if $2^n - 1$ is a prime. We would like to point out, however, that with these circuits, augmented by appropriate software components, we can perform more sophisticated cryptographic systems. An example is the El Gamal digital signature implementation [4], which requires an additional module for computing g.c.d.

In passing, we would like to mention that primes of the form $2^n - 1$ are well known to mathematicians, who have named these Mersenne primes. It is known that there are infinitely many of them. The largest known primes today are Mersenne primes.

Mathematical Background

We begin by introducing some notation. Two integers, a and b are said to be *congruent* modulo n if n divides $a - b$, this is written as $a \equiv b(mod\ n)$. Let $N = 2^n - 1$. Note

that N is the largest n-bit number. The following lemma shows us how to do addition mod N.

Lemma 1 *Let a, b be two n-bit numbers. Then $(a+b) mod\ N$ can be computed by adding a and b with an n-bit full adder, deleting the carry bit of the result, and adding it to the remaining bits of the result.*

Proof. Let $a + b = 2^n Q + R$, where $R < 2^n$. Then, Q is the carry bit and R the number formed by the rest of the bits. Now, $a+b = 2^nQ+R = (2^n-1)Q+Q+R = NQ+Q+R \equiv (Q+R) mod\ N$. ∎

Theorem 1 *If p is a prime, and $0 < x < p$, then $x^{p-1} \equiv 1 (mod\ p)$.*

This is Fermat's little theorem, for a proof, refer to [3]. This gives a necessary condition for primality. Using the multiplication and exponentiation circuits, we can compute $x^{N-1} (mod\ N)$ in $O(n)$ time, doing n-bit arithmetic. This is a popular primality test, if a number fails this, we know it is composite, but if not, there is a high probability that it is prime. For a justification of using this probablistic algorithm, refer to [5].

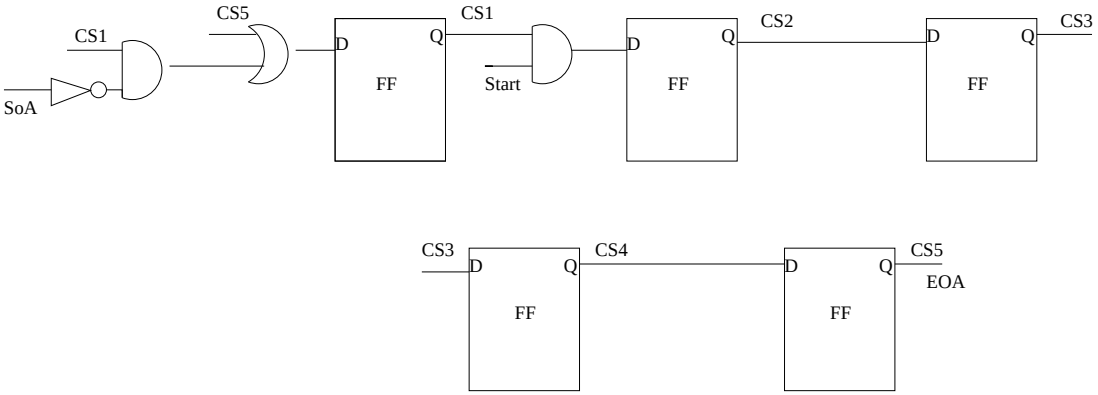
Hardware modules

We give an AHPL [1] specification of the hardware modules, along with a circuit diagram of the datapath. Throughout, we are doing n-bit arithmetic in a 64-bit circuit, with the n rightmost bits holding the data values.

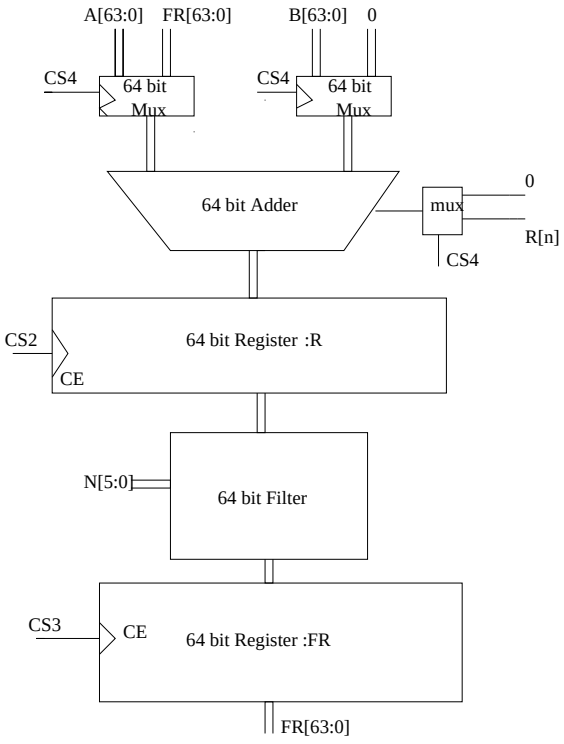
Adder

The adder works on the principle described in Lemma 1. A filter circuit is used to extract the rightmost n bits of the sum of the two inputs, the number obtained this way is added to the carry bit to get the final result. This approach requires two additions. Mask[64] is a global 64-bit bus, whose rightmost n bits are one, the rest zero. It is generated by subtracting one from the output of a 6 to 64 decoder, with input n.

Mod $2^n - 1$ Adder Module
Inputs: A[64], B[64], n[6], Mask[64], Soa
Memory: R[64],FR[64]
Outputs: Eoa
Clunits: Add[64],Filter[64][6]
1. → (¬Soa,Soa)/(1,2)
2. R← Add(A,B)
3. FR← Filter(R,n)
4. R← Add(FR, 64 T 0);Carryin=Add(A,B)[n];
5. Eoa=1; →(1)



Control Unit



Data Path

Figure 1.1: Mod $2^n - 1$ Adder

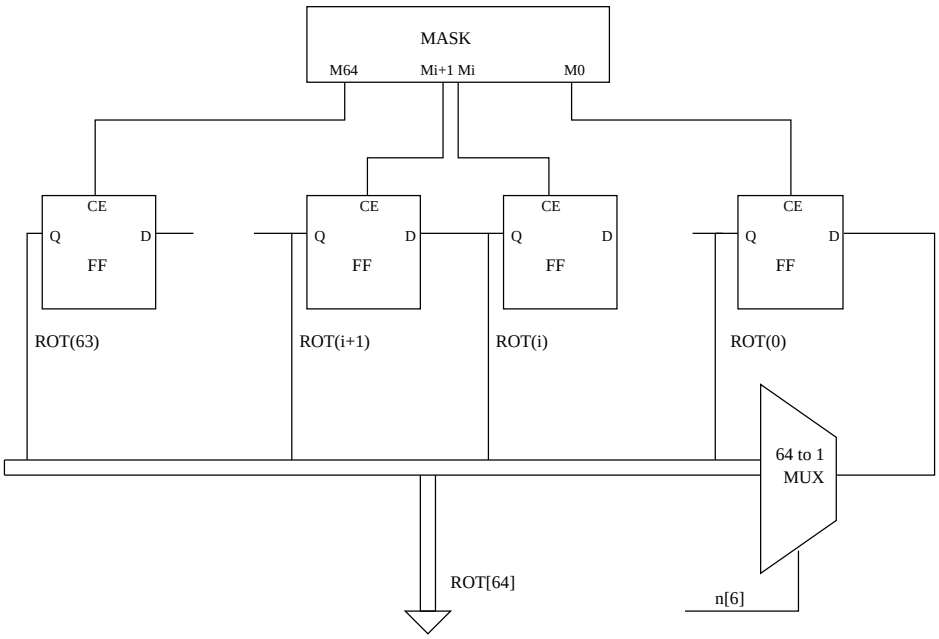


Figure 1.2: Rotation Register

Multiplier

The multiplier design is similar to that of the conventional shift and add multiplier, with a few differences. The output of multiplying two n-bit numbers, ($n < 64$) modulo N is another n-bit number, so the output goes to a 64-bit register instead of 128 bits, as it would in a usual multiplier. The adder module defined in the previous section is used in place of the conventional full adder.

The most interesting variation, however, is the left shift. From Lemma 1, the leftmost bit, which goes to the $(n+1)^{th}$ position - that of a carry bit - should be added to the number formed by the rest of the bits. However, the left shift ensures the rightmost bit is zero. Hence, in place of a shift, all we have to do is perform a left rotation of the n bits. Recall that n is just an input to the circuit, and we must rotate the n rightmost bits in a 64 bit register. This is facilitated by using a decoder to enable only the rightmost n bits of the rotation register, and using a multiplexor to obtain the rotated bit.

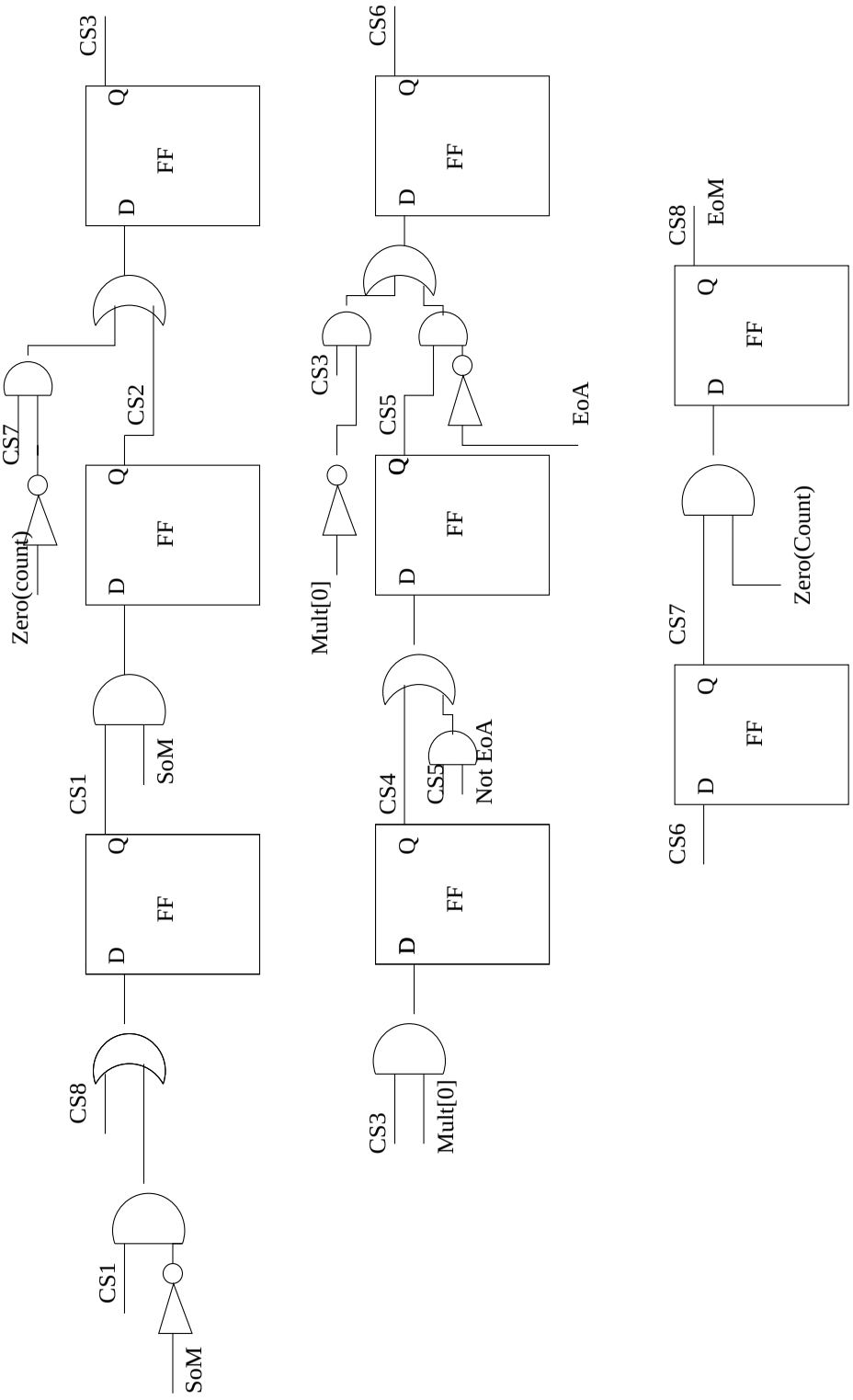
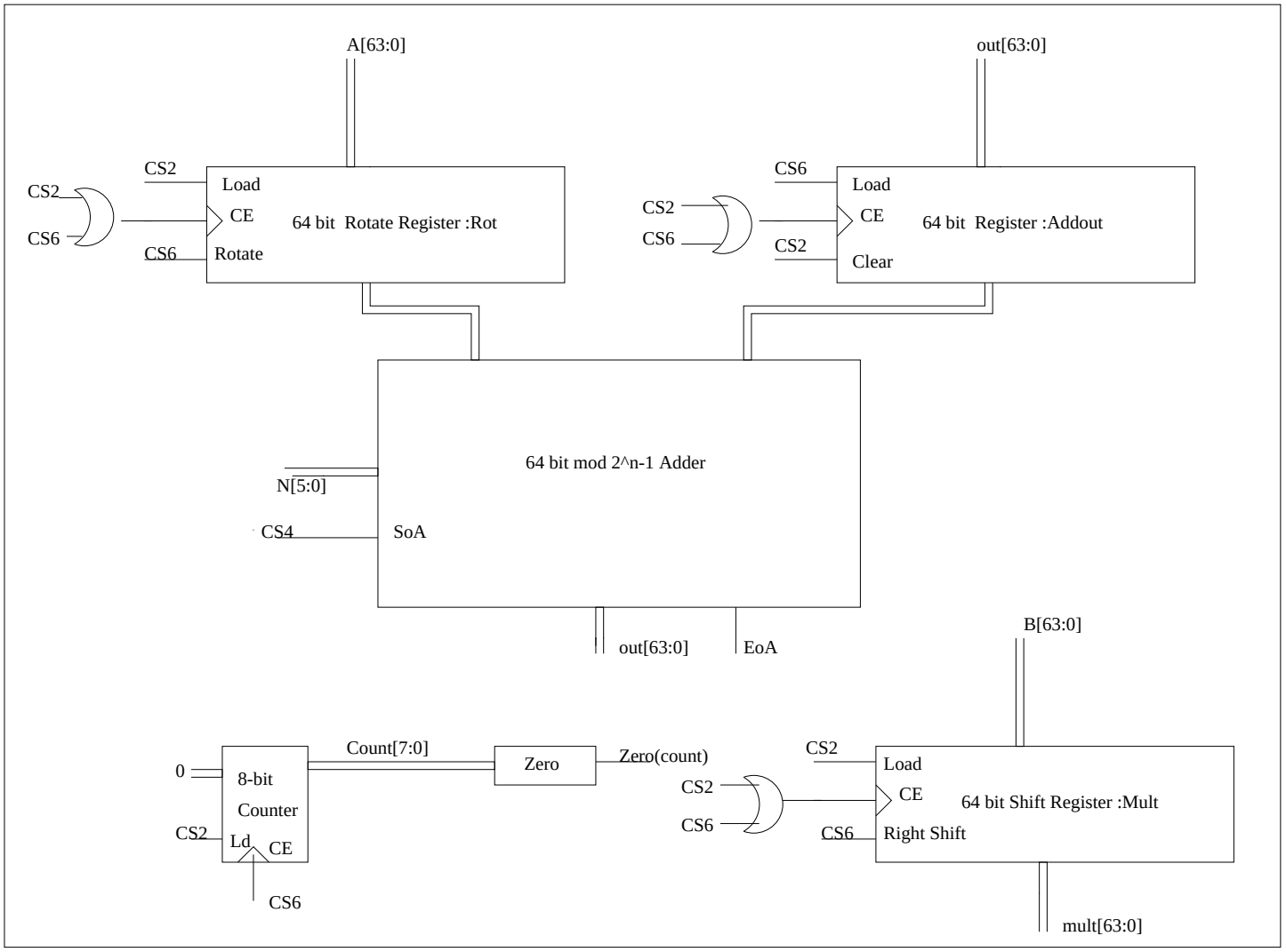


Figure 1.3: Mod $2^n - 1$ Multiplier



Data Path

Figure 1.4: Mod $2^n - 1$ Multiplier

Mod $2^n - 1$ Multiplier Module

Inputs: A[64], B[64], n[6], Mask[64], Som
Memory: Rot[64], Mult[64], Addout[64], Count[8]
Outputs: Eom
 	
1. →(¬Som,Som)/(1,2)
2. Rot← A;
3. Mult← B;
Count← 0;
Addout← 0
4. →(Mult[0],¬Mult[0])/(4,6).
5. Add[mod $2^n - 1$](Rot,Addout); # Adder output goes to Addout
Soa=1.
6. →(Eoa/¬Eoa)/(6,5).
7. Rotate(Rot);
Rshift(Mult);
Inc(Count).
8. →(∨/count,¬ ∨/count)/(3,8).
9. Eom=1; →(1).

The 64-bit register Addout holds the result of the multiplication after the ocmputation is over. This register is accessible from the exponentiator also.

Exponentiator

To compute x^{N-1}, we need $O(log_2(N-1)$ multiplications. The algorithm is as follows:

Exponentiation Algorithm
$acc \leftarrow 1; buf \leftarrow x; count \leftarrow 0;$ while $(count < n)$ $buf \leftarrow buf \times buf;$ $acc \leftarrow acc \times buf;$ $count \leftarrow count + 1;$ endwhile.

The multiplier module is used twice in each iteration step.

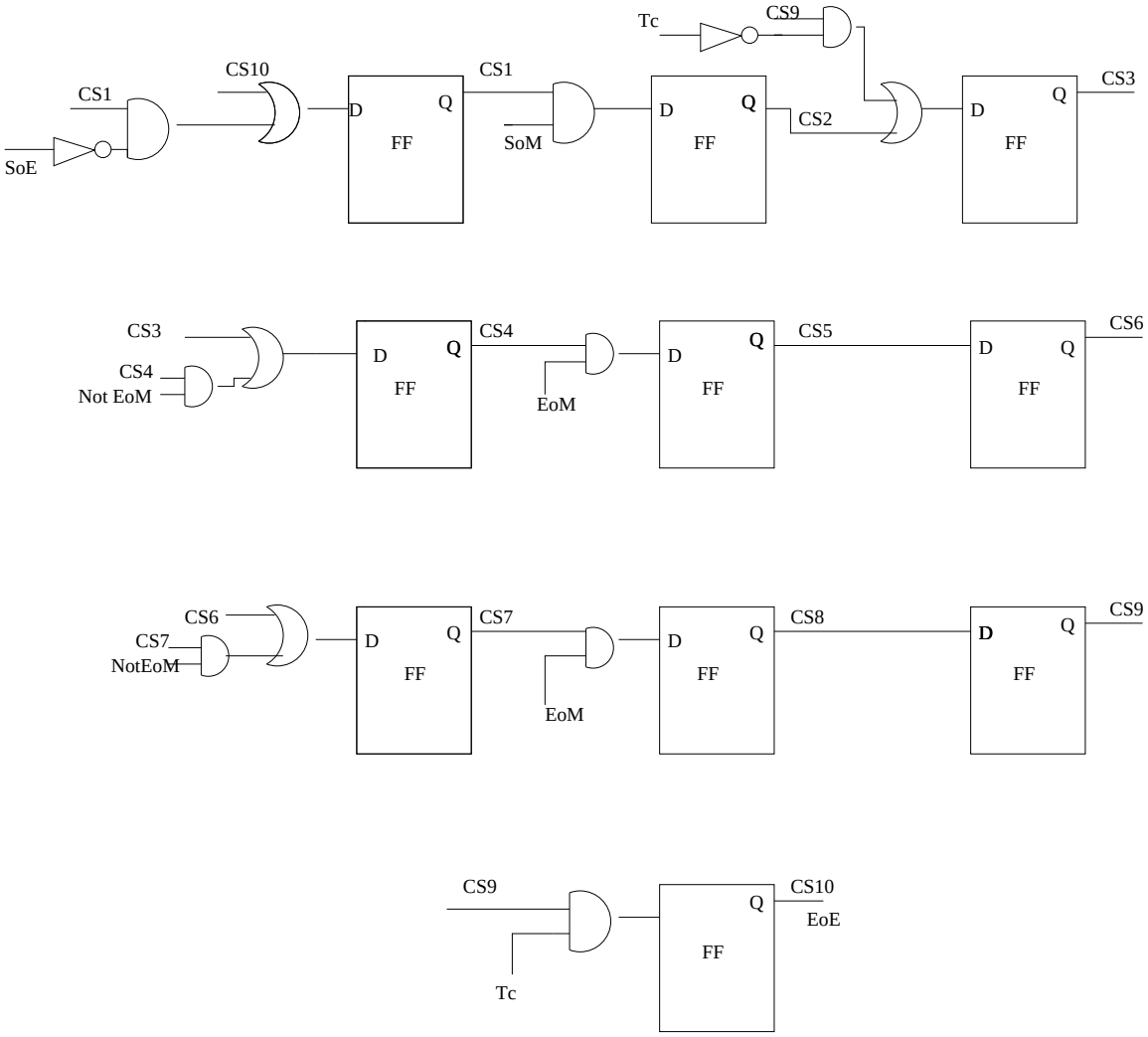
Mod $2^n - 1$ Exponentiation Module
Inputs: x[64],n[6],Soe
Memory: Accum[64],buff[64],Addout[64],Count[6]
Clbuses: Tc[1]
Outputs: Eoe
1. →(¬Soe,Soe)(1,2)
2. Buff← x;
Accum← 1;
Count← 0;
Addout← 0
3. Mult[mod $2^n - 1$](Buff,Buff);
Som=1
4. →(Eom/¬Eom)/(5,4)
5. Buff← Addout
6. Mult[mod $2^n - 1$](Buff,Accum);
Som=1
7. →(Eom/¬Eom)/(8,7)
8. Accum← Addout;
Inc(count);
9. →(Tc/¬Tc)/(10,3)
10. Eoe=1; →(1)

Implementation Details

We implemented the system on a demo board used at the FPGA laboratory in the department [2]. This consists of a Xilinx$^{[tm]}$ 4010E FPGA, and an 8051 microcontroller, which can communicate using common ports.

The software component runs on the 8051. This is responsible for taking the value of n as input and returning the result. Given the value of n, it invokes the verification procedure with a fixed number of values of x. It polls the FPGA port, until the result is known. If the test is passed for all x, it returns that $N = 2^n - 1$ is a prime. If even a single test is failed, it immediately returns failure.

While the modules above were described for 64 bit implementation, our initial implementation used a 32 bit version ($n < 32$). This was because the 64-bit version required 453 CLBs, as opposed to the 400 available on the FPGA. We are implementing a 64-bit version, saving CLBs by implementing some of the multiplexors using tri-state buffers. The major part of memory is used by the 9 64-bit registers. Since the FPGA has only 800 flip-flops, we cannot scale the deisgn to a 128-bit version.



Control Unit

Figure 1.5: Exponentiator for computing x^{N-1} mod N

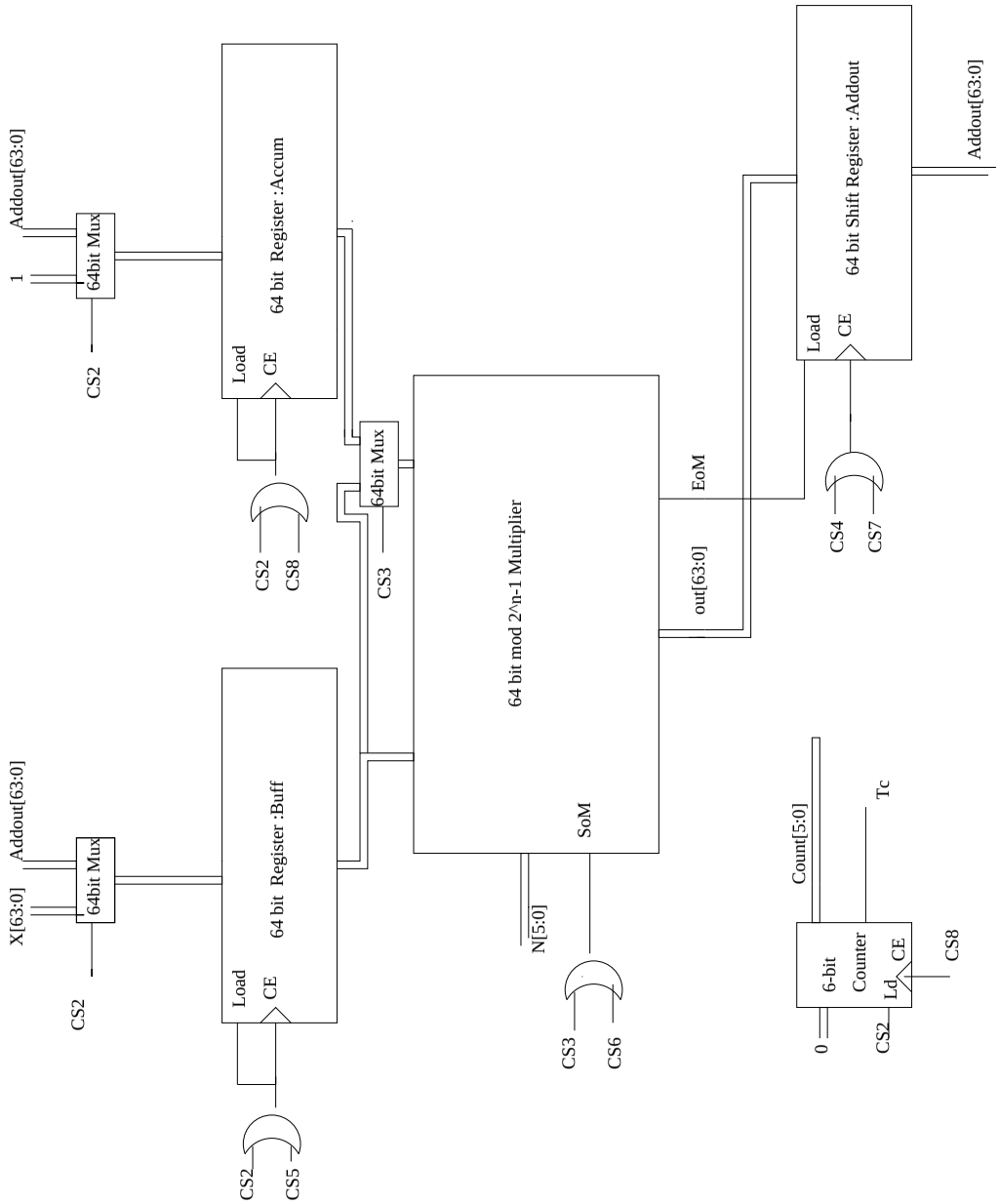


Figure 1.6: Exponentiator for computing x^{N-1} mod N

Conclusion and Future Work

We have identified an interesting special case of an important problem, that of performing arithmetic modulo N, when N is of the form $2^n - 1$. When N is a prime number, it can be used as a part of public / private key for most cryptosystems. Our work illustrates a fast method of generating such keys. We will try and extend this approach to handle more involved cryptographic operations, such as digital signatures.

Acknowledgements

We are grateful to Prof. M. Balakrishnan and Dr. V.B. Taneja for giving us an oppurtunity to study codesign systems in our microprocessors course. We are thankful to them for their guidance and encouragement in this project.

Bibliography

[1] Digital Systems: Hardware - Software Organisation and Design
Hill and Peterson *(Wiley)*

[2] Development of a User Friendly Environment for Hardware - Software Codesign
Kapil Verma *(Mini-project undertaken at IIT, New Delhi)*

[3] An Introduction to the Theory of Numbers
Ivan Niven and Herbert Zuckermann *(Wiley Eastern)*

[4] Handbook of Applied Cryptography
A. Menezes, P. van Oorschot and S. Vanstone *(CRC Press)*

[5] Structure and Interpretation of Computer Programs
H. Abelson and G. Sussman *(MIT Press)*