International Journal of Trend in Scientific Research and Development (IJTSRD)
International Open Access Journal
ISSN No: 2456 - 6470 | www.ijtsrd.com | Volume - 2 | Issue – 5

Fast Modular Multiplication using Parallel Prefix Adder

Sanduri Akshitha[1], Mrs. P. Navitha[2], Mrs. D. Mamatha[2]
[1]PG Scholar, [2]Assistant Professor
Dept of ECE (VLSI),CMR Institute of Technology,
Kandlakoya(V), Medchal-Road, Hyderabad, Telangana, India

ABSTRACT:
Public key cryptography applications involve use of large integer arithmetic operations which are compute intensive in term of power, delay and area. Modular multiplication, which is frequently, used most resource hungry block. Generally, last stage of modular multiplication is implemented by using carry propagate adder whose long carry chain takes more time. In this paper, modulo multiplication architectures using Carry Save and Kogge-Stone parallel prefix adder are presented to reduce this problem. Proposed implementations are faster as compared to conventional carry save adder and carry propagate adder implementations.

1. INTRODUCTION

Modular arithmetic is a system of arithmetic for integers, which considers the remainder. In modular arithmetic, numbers "wrap around" upon reaching a given fixed quantity (this given quantity is known as the modulus) to leave a remainder. Modular arithmetic is often tied to prime numbers, for instance, in Wilson's theorem, Lucas's theorem, and Hensel's lemma, and generally appears in fields like cryptography, computer science, and computer algebra.

An intuitive usage of modular arithmetic is with a 12-hour clock. If it is 10:00 now, then in 5 hours the clock will show 3:00 instead of 15:00. 3 is the remainder of 15 with a modulus of 12.

A number x mod N is the equivalent of asking for the remainder of when divided by . Two integers and are said to be congruent (or in the same equivalence class) modulo if they have the same remainder upon division by N. In such a case, we say that a=b (mod N).

2. MONTGOMERY MODULAR MULTIPLICATION

In modular arithmetic computation, Montgomery modular multiplication, more commonly referred to as Montgomery multiplication, is a method for performing fast modular multiplication. It was introduced in 1985 by the American mathematician Peter L. Montgomery.

Given two integers a and b and modulus N, the classical modular multiplication algorithm computes the double-width product ab mod N, and then performs a division, subtracting multiples of N to cancel out the unwanted high bits until the remainder is once again less than N. Montgomery reduction instead *adds* multiples of N to cancel out the *low* bits until the result is a multiple of a convenient (i.e. power of two) constant $R > N$. Then the low bits are discarded, producing a result less than $2N$. One final conditional subtract reduces this to less than N. This procedure avoids the complexity of quotient digit estimation and correction found in standard division algorithms.

The result is the desired product divided by R, which is less inconvenient than it might appear. To multiply a and b, they are first converted to Montgomery form or Montgomery representation aR mod N and bR mod N. When multiplied, these produce abR^2 mod N, and the following Montgomery reduction produces abR mod N, the Montgomery form of the desired product.(A final second Montgomery reduction converts out of Montgomery form.)Converting to and

from Montgomery form makes this slower than the conventional or Barrett reduction algorithms for a single multiply. However, when performing many multiplications in a row, as in modular exponentiation, intermediate results can be left in Montgomery form, and the initial and final conversions become a negligible fraction of the overall computation. Many important cryptosystems such as RSA and Diffie–Hellman key exchange are based on arithmetic operations modulo a large number, and for these cryptosystems, the computation by Montgomery multiplication is faster than the available alternatives.

An example

Let x = 43, y = 56, p = 97, R = 100. You want to compute x * y (mod p). First you convert x and y to the Montgomery domain. For x, compute x' = x * R (mod p) = 43 * 100 (mod 97) = 32, and for y, compute y' = y * R (mod p) = 56 * 100 (mod 97) = 71.

Compute a:= x' * y' = 32 * 71 = 2272.
In order to zero the first digit, compute
a:= a + (4p) = 2272 + 388 = 2660.
In order to zero the second digit, compute
a:= a + (20p) = 2660 + 1940 = 4600.
Compute a:= a / R = 4600 / 100 = 46.

We have that 46 is the Montgomery representation of x * y (mod p), that is, x * y * R (mod p). In order to convert it back, compute a * (1/R) (mod p) = 46 * 65 (mod 97) = 80. You can check that 43 * 56 (mod 97) is indeed 80.

3. CARRY SAVE ADDER

A **Carry-Save Adder** is just a set of one-bit full adders, without any carry-chaining. Therefore, an n-bit CSA receives three n-bit operands, namely A(n-1)..A(0), B(n-1)..B(0), and CIN(n-1)..CIN (0), and generates two n-bit result values, SUM(n-1)..SUM (0) and COUT(n-1)..COUT (0).

The most important application of a carry-save adder is to calculate the partial products in integer multiplication. This allows for architectures, where a tree of carry-save adders (a so called *Wallace tree*) is used to calculate the partial products very fast. One 'normal' adder is then used to add the last set of carry bits to the last partial products to give the final multiplication result. Usually, a very fast carry-look ahead or carry-select adder is used for this last stage, in order to obtain the optimal performance.

Using carry save addition, the delay can be reduced further still. The idea is to take 3 numbers that we want to add together, x + y + z, and convert it into 2 numbers c+ S such that x + y + z = c + s, and do this in O (1) time. The reason why addition cannot be performed in O(1)time is because the carry information must be propagated. In carry save addition, we refrain from directly passing on the carry information until the very last step. We will first illustrate the general concept with a base 10 example.

To add three numbers by hand, we typically align the three operands, and then proceed column by column in the same fashion that we perform addition with two numbers. The three digits in a row are added, and any overflow goes into the next column. Observe that when there is some non-zero carry, we are really adding four digits (the digits of x, y and z, plus the carry).

```
carry:    1 1 2 1
x:      1 2 3 4 5
y:      3 8 1 7 2
z:    +2 0 5 8 7
-----------------
sum:  7 1 1 0 4
```

The carry save approach breaks this process down into two steps. The first is to compute the sum ignoring any carries:

```
x:    1 2 3 4 5
y:    3 8 1 7 2
z:  + 2 0 5 8 7
-----------------
s:    6 0 9 9 4
```

Each si is equal to the sum of xi +yi +ziModulo 10. Now, separately, we can compute the carry on a column bycolumn basis:

```
x:    1 2 3 4 5
y:    3 8 1 7 2
z:  + 2 0 5 8 7
-----------------
c:    1 0 1 1
```

In this case, each ci is the sum of the bits from the previous column divided by 10 (ignoring any remainder). Another way to look at it is that any carry over from one column gets put into the next column.

Now, we can add together c and s, and we'll verify that it indeed is equal to x + y + z.

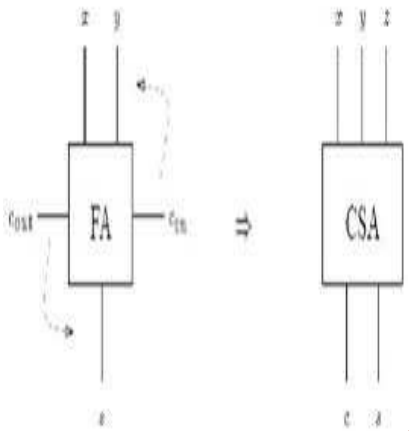

Figure 1: The carry save adder block is the same circuit as the full adder

Input			Output	
A	B	Cin	Sum	Carry
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Figure 2:Truth table

4. RIPPLE CARRY ADDER

At the point when numerous full adders are utilized with the carry ins and carry outs anchored together then this is known as a Ripple carry adder inspight of the fact that the right estimation of the carry bit swells starting with one piece then onto the next (allude to figure 2.16). It is conceivable to make a coherent circuit utilizing a few full adders to include numerous piece numbers. Each full adder inputs a Cin, which is the Cout of the past input. This sort of carry is a ripple carry adder, since each carry bit "ripples" to the following full adder. Note that the first (and just the principal) full adder might be supplanted by a half adder.

The format of a ripple carry adder is basic, which takes into consideration quick outline time; be that as it may, the ripple carry adder is generally moderate, since each full adder must sit tight for the carry bit to be figured from the past full adder. The door postponement can undoubtedly be figured by review of the full adder circuit. Following the way from Cin to Cout indicates 2 doors that must be gone through.

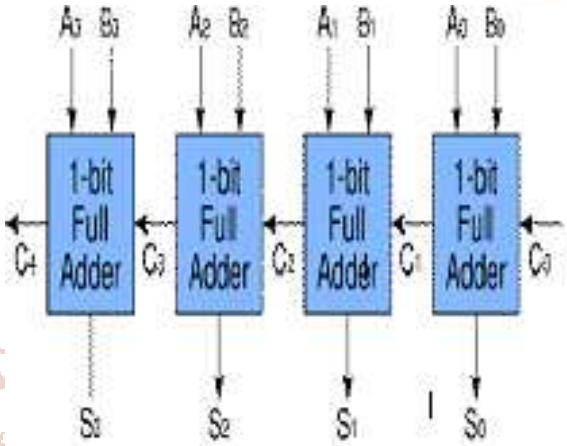

Figure 3: 4-bit ripple carry adder circuit diagram

5. PIPELINING

As the frequency of operation is increased, the cycle time measured in gate delays continues to shrink. Pipelining has emerged as the design technique of choice that helps to achieve high throughput digital systems. This technique breaks down a single complex computational block into discrete blocks separated by clock storage elements (CSE) -like flip-flops, latches. Pipelining improves throughput at the expense of latency, however once the pipe is filled we can expect one data item per unit of time. The gain in speed is achieved by clocking sub-circuits faster and also achieves path delay equalization by inserting registers. As result, it achieve performance gains also the propagation delay and delay variation decreasing. The project used the applications of pipeline to achieve the objective.

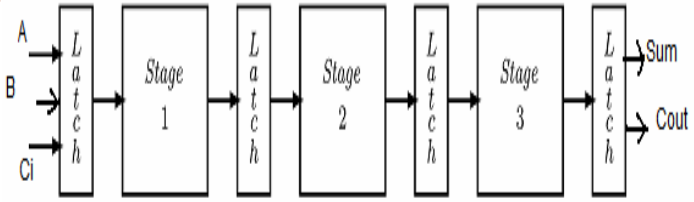

Figure 4: Pipeline applications in 16-bit CSA

Figure 2.13 shows how pipeline applications in CSA circuit based on CSA per stage. The design is in 3 stages of 6 operands 16-bit CSA. Latches between stages 1 and 2 store intermediate results of step 1 "Used by stage 2 to execute step 2 of algorithm". Stage 1 starts executing step 1 on next set of

operands X, Y. Pipeline was just another transformation which is adding the delay and retiming it based on clock using D-flip- flop.

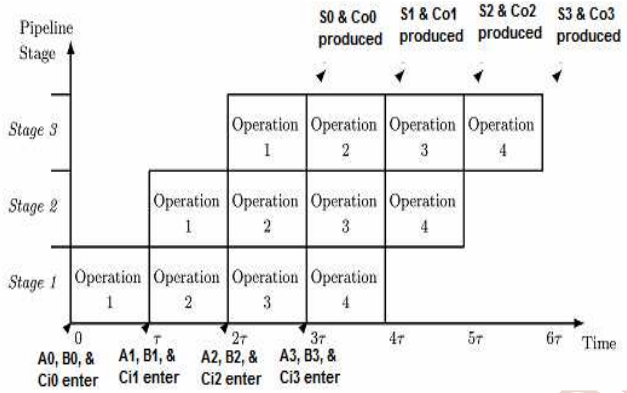

Figure 5: Pipelining Timing Diagram

Pipeline shows how it reduces delay by multiple are overlap in execution. Based on the figure 2.14, when the inputs were given, the operation 1 execute at the 0 time in ladder. The process transforms continuously at the end of 3 times at the stage 3 of pipeline. Without pipeline, the operation 2 would execute at the 3 times. But in this diagram, the operation 2 execute next to the operation 1 has begun. Thus, the delay can be reduced. The process continuously executes per stage as explained.

6. EXTENSION METHOD

The CSA tree proposed in the paper will be enhanced by adding faster adder like parallel prefix adder which would further reduce the delay and increase the speed of operation. The parallel prefix operation is done in 3 stages. i.e. pre processing stage, calculation of carries, post processing stage.

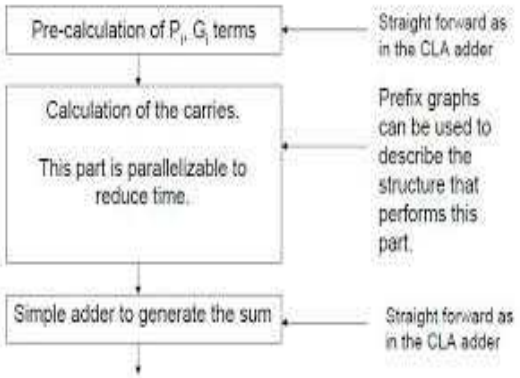

Fig 6: Parallel prefix adder operation

In the pre calculation stage propagate and generate terms are calculated.

i.e. Pi= ai xor bi

gi= ai and bi

The calculated values are passed to next stage i.e. calculation of carries. In this the components are seen in the prefix graph.

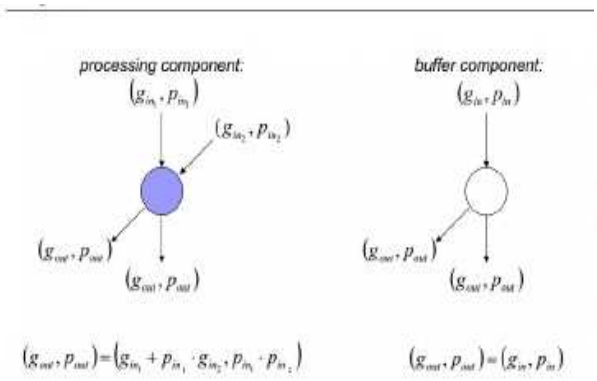

Fig 7 Carry calculation of parallel prefix adder

The execution is done in parallel by decomposing into smaller pieces. The combining operator consists of two AND gates and the OR gate. Each vertical stage produces respective propagate and generate values.

G2 = G1 OR (G0 AND P1)

P2 = P1 AND P0

The calculated carry values are forwarded to the post processing stage. In this stage the final sum values are calculated.

Sn =Pn XOR Cin

Here in this we are using kogge stone adder. One of the parallel prefix adder is kogge stone adder. Kogge stone adder is used for high speed applications but it consumes more area.

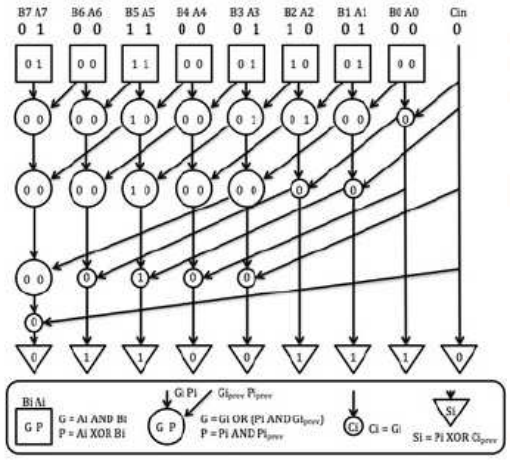

Fig 8 Kogge Stone adder

REFERENCES

1. R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Commun. ACM*, vol. 21, no. 2, pp. 120–126, Feb. 1978.

2. V. S. Miller, "Use of elliptic curves in cryptography," in *Advances in Cryptology*. Berlin, Germany: Springer-Verlag, 1986, pp. 417–426.

3. N. Koblitz, "Elliptic curve cryptosystems," *Math. Comput.*, vol. 48, no. 177, pp. 203–209, 1987.

4. P. L. Montgomery, "Modular multiplication without trial division," *Math. Comput.*, vol. 44, no. 170, pp. 519–521, Apr. 1985.

5. Y. S. Kim, W. S. Kang, and J. R. Choi, "Asynchronous implementation of 1024-bit modular processor for RSA cryptosystem," in *Proc. 2nd IEEE Asia-Pacific Conf. ASIC*, Aug. 2000, pp. 187–190.

6. V. Bunimov, M. Schimmler, and B. Tolg, "A complexity-effective version of Montgomery's algorihm," in *Proc. Workshop Complex. Effective Designs*, May 2002.

7. H. Zhengbing, R. M. Al Shboul, and V. P. Shirochin, "An efficient architecture of 1024-bits cryptoprocessor for RSA cryptosystem based on modified Montgomery's algorithm," in *Proc. 4th IEEE Int. Workshop Intell. Data Acquisition Adv. Comput. Syst.*, Sep. 2007, pp. 643–646.

8. Y.-Y. Zhang, Z. Li, L. Yang, and S.-W. Zhang, "An efficient CSA architecture for Montgomery modular multiplication," *Microprocessors Microsyst.*, vol. 31, no. 7, pp. 456–459, Nov. 2007.

9. C. McIvor, M. McLoone, and J. V. McCanny, "Modified Montgomery modular multiplication and RSA exponentiation techniques," *IEE Proc.-Comput. Digit. Techn.*, vol. 151, no. 6, pp. 402–408, Nov. 2004.

Author profile:

Sanduri Akshitha, she received bachelors of degree in 2015 from Electronics and Communication of engineering from Sudheer Reddy college of engineering and ctechnoogy for women. She is pursuing M.Tech in VLSI System Design from CMR Institute of Technology.

Mrs. P. Navitha
She is working as Assistant professor in CMR Institute of Technology and has 6 years experience in teaching field.

Mrs. D. Mamatha
She is working as Assistant professor in CMR institute of Technology nd has 4 years experience in teaching field