Efficient FPGA Implementation of a Programmable Architecture for GF(p) Elliptic Curve Crypto Computations

Lo'ai Tawalbeh, Abidalrahman Mohammad, Adnan Gutub

Lo'ai Tawalbeh and Abidalrahman Mohammad are with
Computer Engineering Department, Jordan University of Science and Technology, Irbid, Jordan
Email: tawalbeh@just.edu.jo, abdmoh@gmail.com

Adnan Gutub is with
Computer Engineering Department
King Fahd University of Petroleum & Minerals, Dhahran 31261, Saudi Arabia
Email: gutub@kfupm.edu.sa

Abstract:

This paper presents a processor architecture for elliptic curve cryptography computations over GF(p). The speed to compute the Elliptic-curve point multiplication over the prime fields GF(p) is increased by using the maximum degree of parallelism, and by carefully selecting the most appropriate coordinates system. The proposed Elliptic Curve processor is implemented using FPGAs. The time, area and throughput results are obtained, analyzed, and compared with previously proposed designs showing interesting performance and features.

1 Introduction

Elliptic curves cryptography (ECC) was proposed in the mid 1980s by Miler [1] and Koblitz [2]. It provides a public-key cryptosystem based on the elliptic curve discrete logarithm problem. It is considered the most secure public-key scheme until these days [21].

In this paper, a high speed/throughput crypto-processor architecture for computing point multiplication for the elliptic curves defined over the prime fields $GF(p)$ is proposed, and implemented in hardware, using Field Programmable Gate Arrays (FPGAs). The proposed architecture utilizes the parallelism of multiplication operations to speed up ECC point doubling and addition. Also, the new architecture takes advantage of an efficient projective coordinates system to convert $GF(p)$ inversion needed in elliptic point operations into several multiplication steps. In more details, the affine coordinates (X, Y) input is projected into $(X/Z, Y/Z, Z)$ which is more efficient than the conventional choice of projecting it to $(X/Z^2, Y/Z^3, Z)$. This choice was made because the proposed architecture depends on performing several multiplications in parallel which will not be that efficient when using the conventional projective coordinates system, due to data dependency between operands. In addition to that, to increase the processor speed, we used very efficient algorithms to compute modular multiplication, addition, and subtraction, based on the

argument that the speed of the Elliptic Curve Crypto-processor depends on how fast these arithmetic operations can be performed.

After presenting the proposed architecture for the ECC-processor, the synthesis results for area and delay are compared with other related architectures in this research area.

In the next section, three ECC coordinates systems are discussed based on their relation to this work. Section 3 presents the modular multiplication method considered based on Montgomery's approach. The modular addition operation is detailed in Section 4. The proposed elliptic curve crypto hardware is described in Section 5. Section 6 provides the simulation and synthesis strategy and results which are compared with other designs showing the benefit of this work. The conclusions are presented in Section 7 summarizing the paper and its contributions.

2 ECC Coordinates Systems

One of the crucial decisions when implementing an efficient elliptic curve cryptosystem over *GF(p)* is deciding which point coordinates system to use. This section investigates the efficiencies of three different coordinates systems. The first one is the Affine coordinates where a point is represented as (X_A, Y_A). The other two are and two forms of the Projective coordinates: *(X, Y)* where $X_A = X/Z$ and $Y_A = Y/Z$, and *(X, Y)* where $X_A = X/Z^2$ and $Y_A = Y/Z^3$ [3, 4, 5, 22].

Other types of coordinates systems were not considered, either because they are not efficient in general or because they are not compatible with the parallel design of the ECC architecture we build.

2.1 Affine Coordinates System

Affine coordinates system is the simplest coordinates system. However, the modular inversions needed when adding and doubling points which are represented using Affine coordinates system make it an inefficient choice. Other coordinates systems require at least one extra value to represent a point and do not use modular inversions in point addition and doubling, but extra multiplications and squarings are required instead [4, 6, 7, 21].

The addition of two different points on the EC in Affine coordinates is computed as:

$(x_1, y_1) + (x_2, y_2) = (x_3, y_3)$, where $x_1 \neq x_2$

$\lambda = (y_2 - y_1)/(x_2 - x_1)$

$x_3 = \lambda^2 - x_1 - x_2$

$y_3 = \lambda (x_1 - x_3) - y_1$

The addition of a point to itself (doubling a point) is computed as:

$(x_1, y_1) + (x_1, y_1) = (x_3, y_3)$, where $x_1 \neq 0$

$\lambda = (3(x_1)^2 + a)/(2y_1)$

$x_3 = \lambda^2 - 2x_1$

$y_3 = \lambda (x_1 - x_3) - y_1$

We assume in here that the squaring is equivalent to a multiplication, and addition and subtraction are equivalent. Thus, to add two different points in *GF(p)* we need: *six additions, one inversion,* and *three multiplication* operations. To double a point we need: *four additions, one inversion,* and *four multiplications.*

2.2 Projective Coordinates System $(x, y) \rightarrow (X/Z^2, Y/Z^3)$

Projective coordinates are used to eliminate the need for performing inversion [8, 9, 22]. For an elliptic curve defined over *GF(p)*, the Affine elliptic point *(x, y)* is projected to *(X, Y, Z)*, where $x = X/Z^2$, and $y = Y/Z^3$ [8]. This transformation between Affine and projective coordinates is performed only twice: at the beginning and at the end of computations.

Point addition of $P + Q$ in projective coordinates $(x, y) \Rightarrow (X/Z^2, Y/Z^3)$ is computed as:

$P = (X_1, Y_1, Z_1), Q = (X_2, Y_2, Z_2), P + Q = (X_3, Y_3, Z_3)$, where $P \neq \pm Q$

$(x, y) = (X/Z^2, Y/Z^3) \rightarrow (X, Y, Z)$

$\lambda_1 = X_1 Z_2^2$

$\lambda_2 = X_2 Z_1^2$

$\lambda_3 = \lambda_1 - \lambda_2$

$\lambda_4 = Y_1 Z_2^3$

$\lambda_5 = Y_2 Z_1^3$

$\lambda_6 = \lambda_4 - \lambda_5$

$\lambda_7 = \lambda_1 + \lambda_2$

$\lambda_8 = \lambda_4 + \lambda_5$

$Z_3 = Z_1 Z_2 \lambda_3$

$X_3 = \lambda_6^2 \lambda_3^2$

$\lambda_9 = \lambda_7 \lambda_3^2 - 2X^3$

$Y_3 = (\lambda_9 \lambda_6 - \lambda_8 \lambda_3^3)/2$

The doubling of a point *(P+P)* in projective coordinates is computed as:

$P = (X_1, Y_1, Z_1); P + P = (X_3, Y_3, Z_3)$

$(x, y) = (X/Z^2, Y/Z^3) \rightarrow (X, Y, Z)$

$\lambda_1 = 3X_1^2 + aZ_1^4$

$Z_3 = 2Y_1Z_1$

$\lambda_2 = 4X_1Y_1^2$

$X_3 = \lambda_1^2 - 2\lambda_2$

$\lambda_3 = 8Y_1^4$

$\lambda_4 = \lambda_2 - X_3$

$Y_3 = \lambda_1\lambda_4 - X_3$

The needed multiplications to add two points are *sixteen*, while there are *ten* multiplications to double a point.

2.3 Projective Coordinates System (x, y) → (X/Z, Y/Z)

In this architecture, we will use another projection system, the normal elliptic point *(x, y)* will be projected to *(X, Y, Z)*, where $x = X/Z$, and $y = Y/Z$, this way is not usually used because the previous way is simpler and has less number of multiplication [7]. A comparison between the two projection ways is done in the next section, it will show that projecting to *(X/Z, Y/Z)* is faster for parallel design.

The Point addition of *P+Q* in projective coordinates *(x, y)* $\Rightarrow$ *(X/Z,Y/Z)*is computed as follows:

$P = (X_1, Y_1, Z_1)$, $Q = (X_2, Y_2, Z_2)$, $P + Q = (X_3, Y_3, Z_3)$, where $P \neq \pm Q$

$(x, y) = (X/Z, Y/Z) \rightarrow (X, Y, Z)$

$\lambda_1 = X_1Z_2^2$

$\lambda_2 = X_2Z_1$

$\lambda_3 = \lambda_2 - \lambda_1$

$\lambda_4 = Y_1Z_2$

$\lambda_5 = Y_2Z_1$

$\lambda_6 = \lambda_5 - \lambda_4$

$\lambda_7 = \lambda_1 + \lambda_2$

$\lambda_8 = \lambda_6^2 Z_1Z_2 - \lambda_3^2 \lambda_7$

$Z_3 = Z_1Z_2 \lambda_3^3$

$X_3 = \lambda_8 \lambda_3$

$\lambda_9 = \lambda_3^2 X_1Z_2 - \lambda_8$

$Y_3 = \lambda_9 \lambda_6 - \lambda_3^3 \lambda_4$

The doubling of a point *(P+P)* in this projective coordinates is computed as:

$P = (X_1, Y_1, Z_1); P + P = (X_3, Y_3, Z_3)$

$(x, y) = (X/Z, Y/Z) \rightarrow (X, Y, Z)$

$\lambda_1 = 3X_1^2 + aZ_1^4$

$\lambda_2 = Y_1Z_1 \, \lambda_2$

$\lambda_3 = X_1Y_1 \, \lambda_4 = \lambda_1^2 - 8 \, \lambda_3$

$X_3 = 2 \, \lambda_4 \, \lambda_2$

$Y_3 = \lambda_1(4 \, \lambda_3 - \lambda_1) - 8(Y_1 \, \lambda_2)^2$

$Z_3 = 8 \, \lambda_3^3$

In this projective coordinates system the number of multiplications for adding two points is *fifteen*, which is one multiplication less than previous projection scheme. But the number of multiplications for doubling a point is found to be *twelve* which is two multiplications more than the previous scheme.

Table 1 compares the point addition and doubling for the three coordinates systems in terms of the number of additions (A), multiplications (M), and inversion (I) operations.

Table 1: Comparison between the three coordinates systems

Coordinates System	Doubling	Addition
Affine	$4A + 4M + I$	$6A + 3M + I$
Projective $(x, y) \rightarrow (X/Z^2, Y/Z^3)$	$4A + 10M$	$6A + 16M$
Projective $(x, y) \rightarrow (X/Z, Y/Z)$	$4A + 12M$	$6A + 15M$

2.4 ECC Coordinates System Selection

The Affine coordinates system uses inversion operation in both point addition and point doubling, which is a costly and an inefficient operation, so it will be excluded from our research. We will concentrate on the two projective systems discussed above because of their efficiency for parallel designs.

Field multiplication is the basic EC operation used in computing the point *kP* from *P*. Algorithm 1 below is the known Scalar Binary Multiplication Algorithm to be used for this field multiplication [21], which is used in this work too. The number of bits in the binary representation of an integer *k* is *n* indicating the exact number of point doublings, but not point additions. Assume the bits of *k* are half ones and half zeros (an average estimation for

comparison reasons similar to [7]), then the EC arithmetic operations required are n point doublings and approximately $n/2$ point additions.

Algorithm 1: Scalar Binary Multiplication Algorithm

Define n: number of bits in k.

    k_i: the i-th bit of K

Input: P (a point on the elliptic curve).

Output: $Q = kP$ (another point on the elliptic curve).

1. if $k_{n-1} = 1$, then $Q = P$ else $Q = 0$.

2. for $i = n\text{-}2$ down to 0

3. $Q = Q + Q$.

4.   if $k_i = 1$ then $Q = Q + P$.

5. return Q.

The total number of Montgomery multiplications in EC scalar point multiplication kp in the $(x,\ y) \rightarrow (X/Z,\ Y/Z)$ projective coordinates can be calculated as:

$$15\ (point\ additions) + 12\ (point\ doublings) = 15(n/2) + 12n = 19.5n$$

And when projecting $(x,\ y)$ to $(X/Z^2,\ Y/Z^3)$ it is:

$$16\ (point\ additions) + 10\ (point\ doublings) = 16(n/2) + 10n = 18n$$

It is obvious here that when projecting $(x,\ y)$ to $(X/Z^2,\ Y/Z^3)$ the total number of multiplications needed is less than that when projecting $(x,\ y)$ to $(X/Z,\ Y/Z)$ by $1.5n$ multiplications. This made projecting $(x,\ y)$ into $(X/Z^2,\ Y/Z^3)$ always been the candidate for implementing ECC since it has the minimum number of multiplication operations among all projection systems [9, 11, 22].

One way to speedup computing kp is to parallelize multiplication operations. In fact, the most important factor is found to be the number of multiplications in the critical path, especially if the same amount of hardware is used. Deep analysis of the critical paths of both projective coordinates indicates that for parallel implementation the maximum number of multiplication operations that fit to be parallelized is *four* in point addition and point doubling for both projective coordinates [7].

Figures 1 and 2 show the exact analysis of most achievable parallelism for $(x,\ y) \rightarrow (X/Z^2,\ Y/Z^3)$ projective coordinates for point doubling and addition respectively. The corresponding critical path of each dataflow diagram is

effectively of *4 GF(p)* multiplications for point doubling and of *5 GF(p)* multiplications for point addition, and it can be achieved efficiently by using four multipliers.

The time of *GF(p)* addition and subtraction is ignored since it is very small compared to the multiplication time. Therefore, the average time to perform one EC point multiplication operation when calculating of *kp* operation can be found from Figures 1 and 2, which is *(4 + 5/2) = 6.5 GF(p)* multiplications.

Figures 3 and 4 show the analysis of most achievable parallelism for *(x, y)* → *(X/Z, Y/Z)* projective coordinates. Similarly, *four* multipliers are found sufficient to exploit the full parallelism inherent in this projective coordinates. The corresponding critical path of each dataflow diagram is effectively of *3 GF(p)* multiplications for point doubling and of *4 GF(p)* multiplications for point addition. Therefore, the average time to perform one ECC point multiplication operation when calculating of *kp* is *(3 + 4/2) = 5 GF(p)* multiplications.

It can be concluded that using projecting *(x, y)* to *(X/Z, Y/Z)* is more efficient and appropriate for parallel designs. So, it is adopted for the proposed ECC architecture. Furthermore, the utilization of the *four* multipliers is very high as can be seen in Figures 3 and 4 since all *four* multipliers are utilized in most steps.

3 Montgomery Modular Multiplication

The modular multiplication problem is defined as the computation of $P = X \times Y \bmod M$ given the integers X, Y, M with $0 \leq X, Y < M$. For practical applications only the case of odd M is interesting, where $M < 2n \leq 2M$, i.e. the most significant bit $n - 1$ of M is 1, too. There have been several approaches for computing P in hardware. Residue number systems, for example, have been recently receiving more attention [12]. Reported implementations, however, have ignored the need for conversion between binary and residues numbers. This has discouraged researchers from using such an approach [13]. Another approach is based on look-up tables, i.e. ROM-based methods. This method is quite expensive since the required memory space grows exponentially with the word size [13].

The classical method to compute modulo multiplication is by performing the multiplication and then subtracting the modulus several times until the result is less than the modulus. This approach is inefficient and suffers from low speed [12,13,14]. The idea of Montgomery is to reduce the lengths of the intermediate results to a fixed quantity of *n+1* bits. This is achieved by interleaving the computations and additions of new partial products with divisions by *2*, each of them reducing the bit-length of the intermediate result by one.

Most existing modular multiplication solutions are based on Montgomery's algorithm [9, 15, 16]. The basic idea of Montgomery is the following: adding a multiple of *M* to the intermediate results does not change the value of the final result, because the result is computed modulo *M*; where *M* is an odd number. So, after each addition in the inner loop

the least significant bit of the intermediate result is inspected. If it is *1*, i.e. the intermediate result is odd, we add *M* to make it even. This even number can be divided by *2* without remainder. This division by *2* reduces the intermediate result to *n+1* bits again.

Algorithm 2: Conventional Montgomery Multiplication

Require: *X, Y, M, 0 ≤ X, Y < M.*

Ensure: $P = (X \times Y \times (2^n)^{-1})$ *mod M.*

n: number of bits in *X.*

x_i : *i*-th bit of *X.*

s_0: LSB of S.

1. *P = 0.*

2. for *i = 0* to *n − 1* do

3. $P = P + x_i \times Y$.

4. $P = P + s_0 \times M$.

5.   *P = P / 2;*

6. if $P \geq M$ then $P = P − M$.

After *n* steps these divisions add up to one division by *2n*. Montgomery Multiplication requires two passes through the same multiplication process, therefore doubling the computation time. The first pass computes $P=(X \times Y \times (2^n)^{-1})$ *mod M* and the second pass computes $(P \times 2^{2n} \times (2^n)^{-1})$ *mod M = (X × Y) mod M* which is the desired result.

On the other hand, it is very easy to implement since it operates on the least significant bit first and does not require any comparisons. Therefore, it can be implemented using carry save adders and a redundant representation of the intermediate results. Carry save adders have a latency of *O(1)* in comparison to standard ripple carry adders with a latency of *O(n)* and carry lookahead adders with a latency of *O(log n)*. Their disadvantage is that they add three operands to two results, thus producing a result in redundant form which does not allow comparisons to other values in constant time. A modification of Montgomerys Algorithm with carry save adders is shown in Algorithm 3 [9,12,13,15,16].

In this algorithm the delay of one pass through the loop is reduced from *O(log n)* to *O(1)*. We take the notion of an assignment of the sum of three operands to two values *S,C* as in order to indicate the use of a carry save adder.

Algorithm 3: Montgomery Multiplication using Redundant CSA

Require: $X, Y, M, 0 \leq X, Y < M$.

Ensure: $P = (X \times Y \times (2^n)^{-1}) \bmod M$.

n: number of bits in X.

x_i : i-th bit of X.

s_0: LSB of S.

1. $S = 0; C = 0$.

2. for $i = 0$ to $n - 1$ do

3. $S,C = S + C + x_i \times Y$.

4. $S,C = S + C + s_0 \times M$.

5. $S = S / 2; C = C / 2$.

6. $P = S + C$.

Of course, the addition in step *6* is conventional addition. But since they are performed only once while the additions in the loop are performed *n* times this is subdominant with respect to time complexity.

Figure 5 shows the design of the Montgomery multiplier mentioned in Algorithm 3, the design consists of 2 carry save adders. The three inputs of the first adder are the sum and carry of the previous iteration shifted by one bit denoted by " << " to the right (zeros the first iteration), and $Y.x_i$ (*Y* if the i − th bit of vector *X* is one and zero otherwise). The inputs of the second carry save adder are the sum and carry outputs of the first adder and $s_0.M$ (*M* if bit zero of the sum is one and zero otherwise).

The critical path delay of this multiplier could be calculated as follows:

critical path delay   = 2 (CSA delay) + 2 (AND gate delay).

= 4 (XOR gate delay) + 2 (AND gate delay).

The operating frequency of the multiplier is expected to be high, since it is the reciprocal of the critical path delay which is relatively small.

4 Modular Addition Subtraction

The modular addition operation adds two inputs, *A* and *B* (both < *M*), and subtracts the modulus *M* from the sum if *(A+B)≥M*, if not the answer remains *(A+B)*. Implementing the two cases in parallel by calculating *(A+B)* and *((A+B)−M)* and choosing one of them depending on the sign of *(A+B−M)* omits the cost of reduction. To perform modular subtraction, *(A−B)* and *(A−B+M)* are calculated in parallel. If the result of *(A−B)* is positive then it is the

correct answer, and if it is negative then *(A−B+M)* is the correct answer.

By this method, both possible results are computed in slightly more time than a single *n*-bit addition, and the correct result is selected depending on the sign of the results. This eliminates the necessity to wait a full *n*-bit magnitude comparison before deciding whether to add *M*, subtract *M* or do nothing. Algorithm 4 summarizes the idea:

<u>Algorithm 4: Modular Addition Subtraction</u>

Require: *M, 0 ≤ A < M, −M ≤ B < M*

Ensure: *C = A ± B mod M*

1. *C' = A ± B*

2. if *B > 0* and *C' > M* then

3.   *C'' = A + B − M*

4. elsif *B < 0* and *C' < 0* then

5.   *C'' = A − B + M*

6. else

7.   *C'' = C'*

To eliminate the cost of reduction (subtracting or adding *M* when the result is grater than *M* or less than zero) a especial design is used. Both the reduced and non-reduced values are calculated and the correct answer is then selected depending on the result.

Figure 6 shows the design of the Modular Adder Subtractor Shifter that implements Algorithm 3. The design consists of 2 Carry Propagate Adders (CPA), a Carry Save Adder (CSA) and four Multiplexors (MUX). The inputs of the design are selected using multiplexors depending on the operation. For example if addition is to be performed, the first MUX selects −*p*, the second and the third select *a* and *b* to calculate both *(a + b − p)* on CPA1 and *(a + b)* on CPA2. One of the results is then selected depending on the sign of *(a + b − p)*, if it is positive, then it is the correct answer, if not then *(a + b)* is the correct answer.

Similarly, in case of subtraction, multiplexors select *p, a,* and −*b* to calculate both *(a − b + p)* on CPA1 and *(a − b)* on CPA2. One of the results is then selected depending on the sign of *(a − b)*, if it is positive then it is the correct answer, otherwise then *(a − b + p)* is the correct answer. In case of shifting, the inputs are −*p, 2a,* and zero, the result of CPA1 is selected if it is positive *(2a−p ≥ 0)* if not, the result of CPA2 *(2a)* is selected.

Carry propagation delay is a problem in this part of the design. It was avoided in Montgomery multiplier by using CSAs, but here CSAs cannot be used because their output is in redundant form, and the inputs to the Montgomery

multipliers in the next iteration must be in conventional form. The carry propagation delay could be minimized by using advanced adders such as carry look ahead adders but this will require large area that might not be available for this design. Instead of using such adders, we use a especial carry propagate adder that has a small precision and it calculates the result in many cycles, which minimizes the area and increases the operating frequency of the adder.

Figure 7 shows the input register width (n bits) is divided to many words each is L-bits width (padding with zeros is used if less than L bits remain in the last word). At each clock cycle L-bit word from input registers is added, the carry out of each part is used as a carry in to the next iteration and the output is written to the output register. The delay of the addition mainly depends on the width of the adder L. The critical path delay calculation will be different here, because special built-in adders of the target device (will be investigated in next chapter) have much less propagation time than manually built adders. (For example: the delay of a *32*-bit built in adder is *16.67ns* but *28.5ns* for manually designed adders on *VirtexE XCV600E* device).

The value of L could be determined from the view of the overall design, a trade-off between the number of cycles and the cycle delay occurs. Number of cycles increased by n/L, but the benefit of decreasing cycle time is much larger, because the Montgomery multiplier has relatively small cycle time and large number of cycles. The cycle time of the whole design will depend on the greatest cycle time of either the modular adder or Montgomery multiplier (it will be explained in the next section).

The number of cycles in the Montgomery multiplication the EC point multiplication operation is *5n* and the number of cycles taken in addition is *8(n/L)*, this mean that if this adder is used *8(n L −1)* extra cycles are needed. But *5n+8(n/L)* cycles will benefit from decreasing the cycle time, which results in a good increase in performance if L is chosen correctly (selecting L for this design is done based on the synthesis results and its typical value is found to be 32).

5 EC Crypto-processor Datapath

The hardware architecture of the datapath of the EC crypto-processor is shown in Figure 8. It consists of four Montgomery Modular multipliers (MM), four Modular-Adder Subtractors (MA-S), a control unit, General Purpose (G.P) Register file and *16* input/output registers.

At each state, signals from the control unit start either the MM units or the MA-S units as needed. The inputs to MM units could be taken from the user or from the output of MA-S units. Inputs of MA-S units are taken from the output of MA-S units themselves or from MM units. Also control signals enable or disable all input/output registers and general purpose registers as needed.

It is worth mentioning a simple protocol for providing the start signal as the inputs are ready, and finish signal when the outputs are ready is used for both MM and MA-S units. This allows easy modifications on the datapath. For example radix-*2* MM could be replaced with radix-*4* or word-based MMs.

6 Simulations Synthesis and Comparison Results

The previously described EC crypto-processor was implemented in VHDL, and simulated in Mentor-Graphic simulation tool (ModelSim) to validate its functional correctness. It was synthesized using Xilinx synthesis tool, with the target FPGA chip family chosen to be *virtex5*. It is a new unique one that provides large area (330000 logic cells) with $65 - nm$ copper CMOS process technology, and operates on high frequency (up to 550 MHz). The used chip within this family was *XC5VLX30*.

Firstly, synthesis was carried independently for the MM and MA-S units. Figure 9 shows the operating frequency of the MM units operate on the range from *230* to *340* MHz depending on datapath width. The area of MM units for different datapath widths is shown in Figure 10 in terms of the number of slices and LUTs (Look Up Tables).

For MA-S units Figure 11 shows that the operating frequency range is from *190* to *300* MHz depending on datapath width. The area of the results are in terms of the number of slices and LUTs as shown on Figure 12. Synthesis results show that the EC crypto-processor operate at frequency range from *200* to *228* MHz as can be seen from Figure 13. Compared with other designs, it is a high frequency, that is due to two reasons, the first is the efficient datapath design that use such as using CSAs in Montgomery Multiplication and built-in CPA with small width and the efficient reduction algorithm, the second reason is using a new device with advanced features, high operational speed and large area.

From the previous figures, anyone can note that the frequency of the whole processor could be more than the frequency of some of its components (such as adders), which seems to be not logical. Register retiming or register balancing is a technique in which part of the hardware before or after the register is transferred to other side within the datapath, in order to balance the delay before and after it, so the frequency of the design is increased. If this property is disabled, the results would be logical but the frequency will decrease. Figure 14 and Table 2 show the area of the EC crypto-processor. The area increase is expected. The area of the datapath it is almost four times the area of a MM and a MA-S units, which sound logical.

Table 2: Datapath Area

Datapath Width (bits)	Number of Slices	Number of LUTs

50	4598	8284
100	8568	16018
150	11926	22652
200	15892	29868
250	19250	37082
300	23216	44029

The amount of increase in the throughput compared to a similar serial design could be estimated by dividing the number of multiplications in serial design by the number of multiplications in parallel design:

$$Throughput\ increase = 19.5/5\ \% = 390\%$$

The time needed to compute one Montgomery Multiplication can be computed by:

$$T_{mult} = (cycles/bit)\ \times n \times clockperiod.$$

At operand precision of $n=256$ bits, the time required to compute one Montgomery multiplication:

$$T_{mult} = 1 \times 256 \times 5 \times 10^{-9} = 1.28\ \mu sec$$

From section 3.2 EC point addition needs 4 multiplications so:

$$T_{add} = 4\ T_{mult} = 4\ (1.28) = 5.12\ \mu sec$$

And EC point doubling needs 3 multiplications so:

$$T_{double} = 3\ T_{mult} = 3\ (1.28) = 3.84\ \mu sec$$

To compute the scalar point multiplication (kp), an inversion is required to transform the results from projective to affine coordinates, the time needed to compute modular inversion is estimated by: $T_{inv} = 3\ T_{mult} = 3.84\ \mu sec$.

From the above equation we can compute the time needed for an EC point multiplication:

$$T_{kp} = k\ T_{double} + 0.5\ k\ T_{add} + T_{inv} = 3\ k\ T_{mult} + 4\ (0.5)\ (k)\ T_{mult} + 3\ T_{mult} = 5\ (k + 3)\ T_{mult}$$

Assuming that the length of binary vector k is equal to n then:

$$T_{kp} = 5\ (n + 3)\ (n)\ clockperiod = 5\ (n^2 + 3n)\ clockperiod$$

Comparisons with other designs could be inconsistent because of the use of different designs, Galois Fields and devices technologies. But comparing the number of EC point multiplication operations per second could be the most compatible way for comparison.

The number of EC point multiplications within a second is the inverse of T_{kp}:

$$EC\ mult/sec = 1/T_{kp} = frequency/5\ (n^2 + 3n).$$

For comparison purpose it can be approximated by:

$$= frequency/5 \; (n^2).$$

Eberle et al. [17] processor operate on *66* MHz . The number of ECC point multiplications per second $= 66 \times 10^6/(19.5 \times n^2)$, which equals: $3.4 \times 10^6/n^2$. For our processor it is: $200 \times 10^6/(5 \times n^2)$ which equals $40 \times 10^6/n^2$. So our processors throughput is *40/3.4 = 11.7* times greater.

Satoh and Takano [18] processor can perform *700* point multiplications per second on *192*-bit prime field. For our processor, $206 \times 10^9/(5 \times 192^2) = 1120$ point multiplications per second for the same field can be done, compared with it our design is *1120/700 = 1.6* times faster, although there was implemented on ASIC device and is dedicated for one prime field which makes it simpler and faster.

Also, our design is *3.5* times faster than Orlando and Paar [19] processor which can perform *330* multiplications per second on a dedicated *192* bit prime field, and *14.5* and *6.4* times faster than Ors et al. [20] and Crowe et al. [11] processors that can operate *70* and *174* point multiplications per second on a dedicated *160* and *256* bit prime fields, respectively.

7 Conclusions

In this paper, a new crypto-processor architecture to compute the point multiplication for the elliptic curves defined over the prime fields GF(p) is implemented. Unlike known architectures, this EC crypto-processor depends on parallelizing the multiplication operations to speed up EC point doubling and addition rather than sequential multiplication which result in great increases in performance.

We also showed that the maximum parallelism occurs when projecting the input *(x,y)* into *(X/Z,Y/Z)* rather than conventional choice of projecting it to $(X/Z^2, \; Y/Z^3)$, the affine coordinates system was excluded because it requires the costly inversion operation in point addition and doubling.

Very efficient algorithms to compute modular multiplication, addition, and subtraction where used, such as Montgomery Multiplication with CSA adders, pre-computing of reduced and non-reduced results in Modular Addition, and the usage of small width adders for many cycles to add larger precisions operands.

The Usage of parallelism, efficient coordinates system, and fast algorithms in field operations lead to a very large increase in the throughput and performance of the proposed Elliptic Curve Crypto-processor compared to other well known ones.

8 Acknowledgments

9 References

[1] Miller, V., "Elliptic curves in cryptography", *Lecture Notes in Computer Science No. 218 on Advances in Cryptology Crypto '85*, pp. 417–246, , Springer-Verlag Berlin, Germnay, 1986.

[2] Koblitz, N., "Elliptic curve cryptosystems", *Mathematics of computation*, Vol. 48, No. 177, pp. 203–209, January 1987.

[3] Cohen, H., Miyaji A. and Ono T., "Efficient elliptic curve exponentiation using mixed coordinates", *Lecture Notes in Computer Science on Advances in Cryptology - ASIACRYPT 98*, Vol. 1514, pp. 51–65, January 1998.

[4] Certicom, http://www.secg.org/collateral/proposal-for-sec1v2.pdf , Certicom Proposal to Revise SEC 1: Elliptic Curve Cryptography, Version 1.0, Prepared by Daniel R. L. Brown, January 14, 2005, accessed 29 April 2008.

[5] Certicom, http://www.certicom.com/index.php?action=ecc_tutorial,home , Online elliptic curve cryptography tutorial, accessed 29 April 2008.

[6] Gutub, A., "Merging GF(p) elliptic curve point adding and doubling on pipelined VLSI cryptographic ASIC architecture", *International Journal of Computer Science and Network Security - IJCSNS*, Vol. 3A, No. 6, pp. 44–52, March 2006.

[7] Gutub, A., "VLSI core architecture for GF(p) elliptic curve crypto processor", *IEEE 10th International Conference on Electronics, Circuits and Systems - ICECS*, pp. 84–87, University of Sharjah, United Arab Emirates, December 2003.

[8] Miyaji, A., "Elliptic curves over Fp suitable for cryptosystems," *Lecture Notes In Computer Science; Vol. 718 on Advances in cryptology – AUSCRUPT 92*, pp. 479-491, Australia, December 1992.

[9] Gutub, A. and Ibrahim, M., "High radix parallel architecture for GF(p) elliptic curve processor", *IEEE Conference on Acoustics, Speech, and Signal Processing, ICASSP 2003*, pp. 625- 628, Hong Kong, April 6-10, 2003.

[10] Montgomery, P., "Modular multiplication without trial division", *Mathematics of Computation*, Vol. 44, No. 170, pp. 519–521, April 1985.

[11] Blum, T., and Paar, C., "Montgomery modular exponentiation on reconfigurable hardware", *14th IEEE Symposium on Computer Arithmetic - ARITH-14*, pp. 70-77, April 1999

[12] Brickell, E., "A fast modular multiplication algorithm with application to two key cryptography," *Advances in Cryptology – CRYPTO 82*, pp. 51–60, Santa Barbara, California, USA, Edited by Chaum, D., Rivest, R., and Sherman, A., Plenum Press, Ney York,1983.

[13] http://www.nsa.gov last acceded in 29 April 2008.

[14] Bernal, A., and Guyot, A., "Design of a modular multiplier based on Montgomery's algorithm", *13th Conference on Design of Circuits and Integrated Systems – DCIS'98*, pp. 680–685, November 1998.

[15]   Wu, C., and Chou, Y., "General modular multiplication by block multiplication and table lookup," *IEEE International Symposium on Circuits and Systems – ISCAS'94*, Vol. 4, pp. 295–298, London, UK, 1994.

[16]   Eldridge, S., and Walter, C., "Hardware implementation of Montgomery's modular multiplication algorithm", *IEEE Transaction on Computers*, Vol. 42, No. 6, pp. 693–699, June 1993.

[17]   Eberle, H., Gura, N., Shantz, S., Gupta, V., Rarick, L. and Sundaram, S., "A public-key cryptographic processor for RSA and ECC", *Proceedings of the 15th IEEE International Conference on Application-Specific Systems, Architectures and Processors*, pp. 98-110, September 2004.

[18]   Satoh, A., and Takano, K., "A scalable dual-field elliptic curve cryptographic processor", *IEEE Transactions on Computers*, Vol. 52, No. 4, pp. 449-460, April 2003.

[19]   Orlando, G., and Paar, C., "A scalable GF(p) elliptic curve processor architecture for programmable hardware," *Lecture Notes in Computer Science on Cryptographic Hardware and Embedded Systems - CHES 2001*, pp. 348-363, 2001.

[20]   Ors, S., Batina, L., Preneel, B., and Vandewalle, J., "Hardware implementation of an elliptic curve processor over GF(p)", *Proceedings of IEEE International Conference on Application-Specific Systems, Architectures, and Processors – ASAP'03*, pp. 433-443, June 2003.

[21] Gutub, A. "Efficient Utilization of Scalable Multipliers in Parallel to Compute GF(p) Elliptic Curve Cryptographic Operations", *Kuwait Journal of Science & Engineering (KJSE)*, Vol . 34, No. 2, Pages: 165-182, December 2007.

[22] Francis, C., Daly, and Marnane, W., "A scalable dual mode arithmetic unit for public key cryptosystems", *International Conference on Information Technology: Coding and Computing - ITCC'05*, Vol. I, pp. 568-573, 2005.

Figure Captions:

Figure 1. ECC point doubling in $(x, y) \rightarrow (X/Z^2, Y/Z^3)$ projection

Figure 2. ECC point addition in $(x, y) \rightarrow (X/Z^2, Y/Z^3)$ projection

Figure 3. ECC point doubling in $(x, y) \rightarrow (X/Z, Y/Z)$ projection

Figure 4. ECC point addition in $(x, y) \rightarrow (X/Z, Y/Z)$ projection

Figure 5. Montgomery modular multiplier design

Figure 6. Modular adder subtractor shifter

Figure 7. Adder design

Figure 8. ECC datapath

Figure 9. Montgomery modular multiplier frequency

Figure 10. Montgomery modular multiplier area

Figure 11. Modular adder-subtractor frequency

Figure 12. Modular adder-subtractor area

Figure 13. Datapath frequency

Figure 14. Datapath area

Figure 1
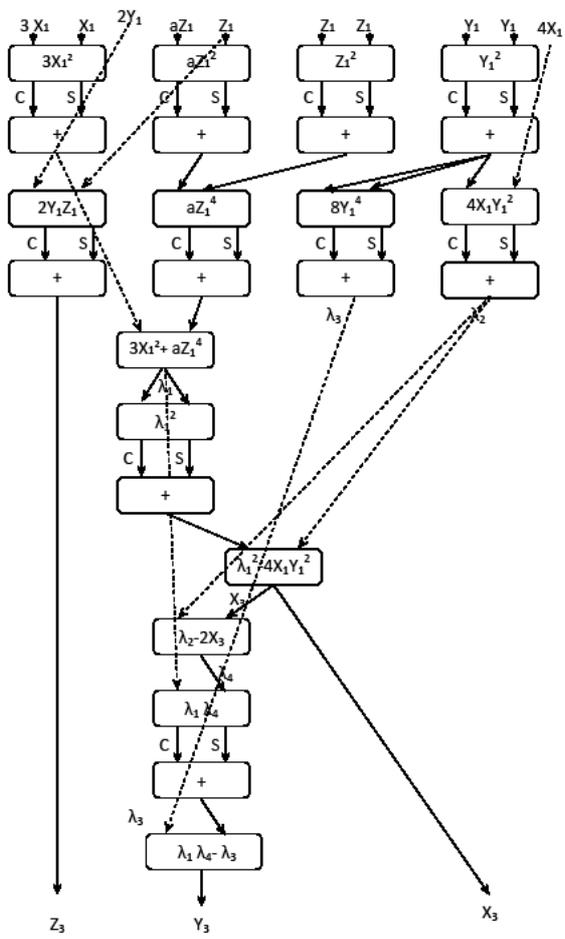
Figure 2

Figure 3

Figure 4

Y_1 Z_2 Y_2 Z_1 X_2 Z_1 X_1 Z_2

| $Y_1 Z_2$ | $Y_2 Z_1$ | $X_2 Z_1$ | $X_1 Z_2$ |

C   S   C   S   C   S   C   S

| + | + | + | + |

λ_5 λ_4 λ_2 λ_1

| $\lambda_5 - \lambda_4$ | $\lambda_1 + \lambda_2$ | $\lambda_1 - \lambda_2$ |

λ_3 λ_3 λ_3

| $Z_2 \lambda_3$ | λ_6 | $Z_1 Z_2$ | $\lambda_3{}^2$ |

C   S   C   S   C   S   C   S

| + | + | + | + |

λ_1 λ_7

| $\lambda_1 \lambda_3{}^2$ | $Z_2 \lambda_3{}^3$ | $Z_1 Z_2 \lambda_6{}^2$ | $\lambda_7 \lambda_3{}^2$ |

C   S   C   S   C   S   C   S

| + | + | + | + |

| $Z_1 Z_2 \lambda_6{}^2 - \lambda_7 \lambda_3{}^2$ |

λ_8

| $\lambda_1 \lambda_3{}^2 - \lambda_8$ |

λ_6 λ_9 λ_8

Y_1 Z_1

| $Y_1 Z_2 \lambda_3{}^3$ | $Z_1 Z_2 \lambda_3{}^3$ | $\lambda_9 \lambda_6$ | $\lambda_3 \lambda_8$ |

C   S   C   S   C   S   C   S

| + | + | + | + |

| $\lambda_9 \lambda_6 - Y_1 Z_2 \lambda_3{}^3$ |

Z_3 Y_3 X_3
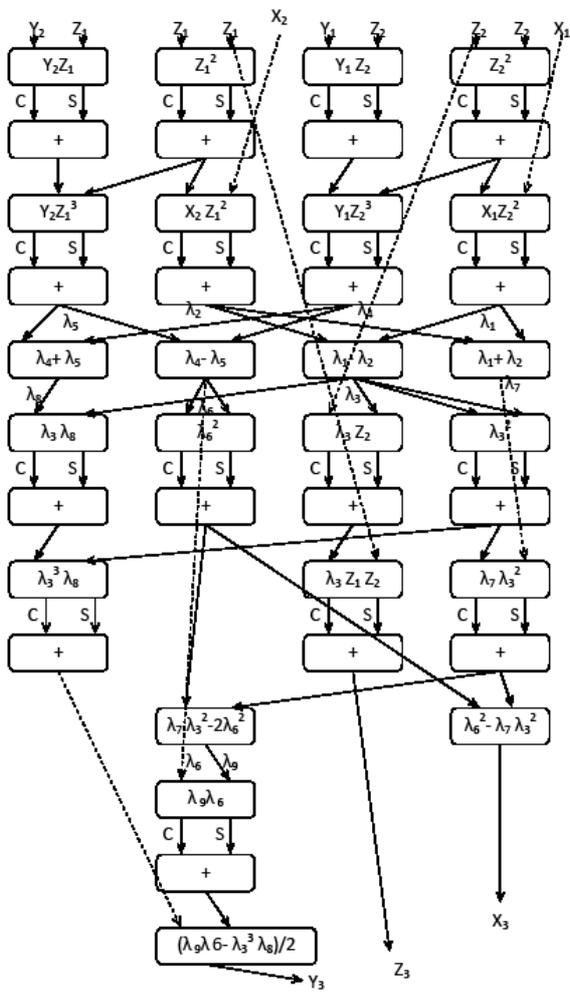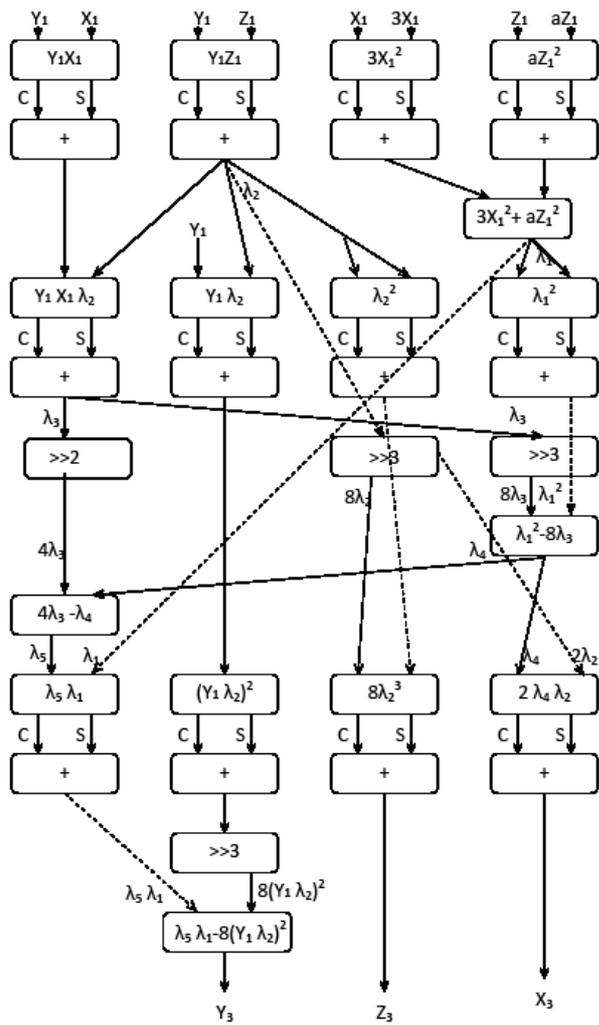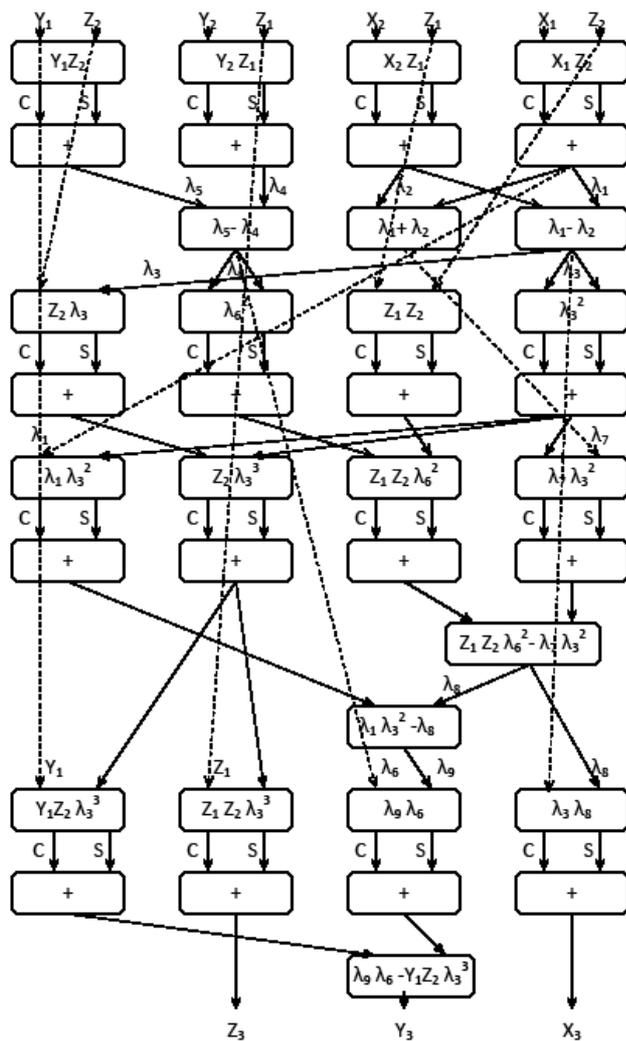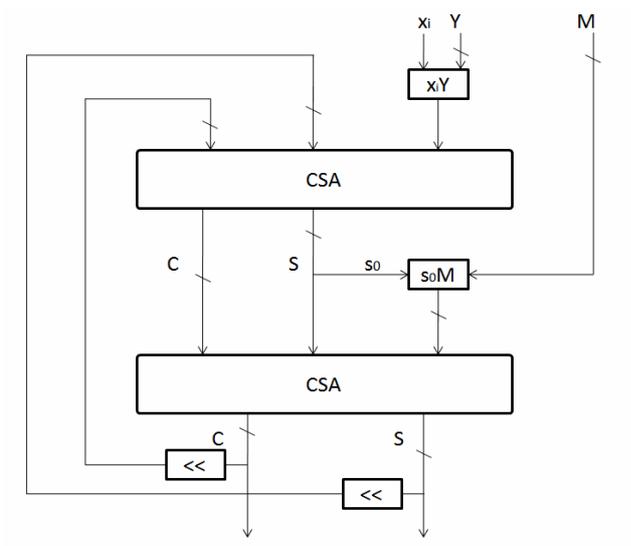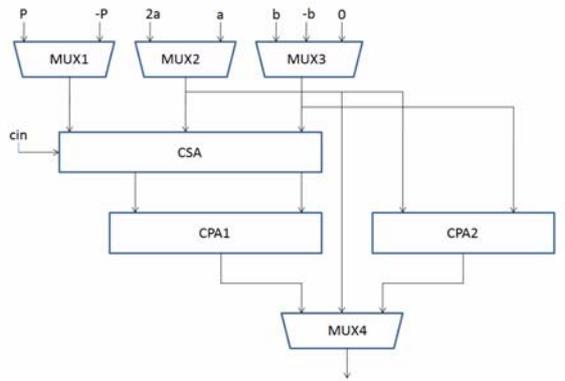
Figure 5

Figure 6

Figure 7
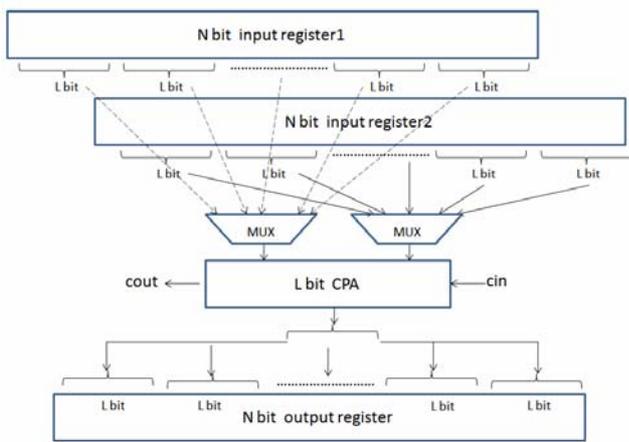
Figure 8
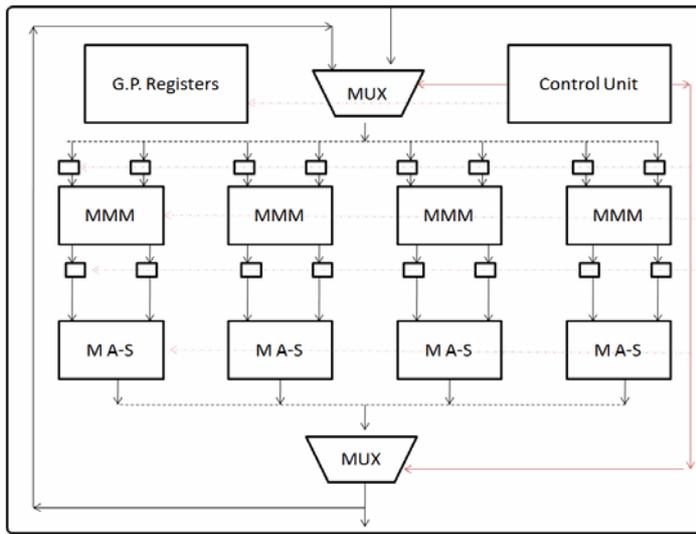
Figure 9
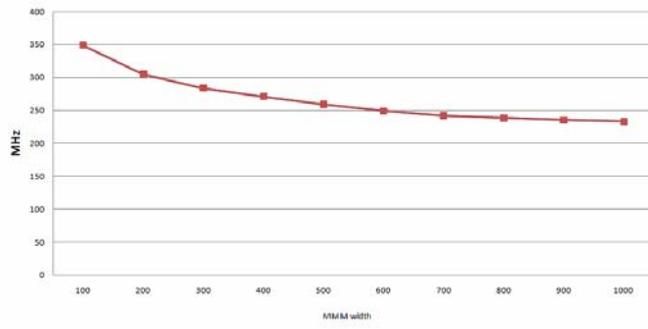
Figure 10
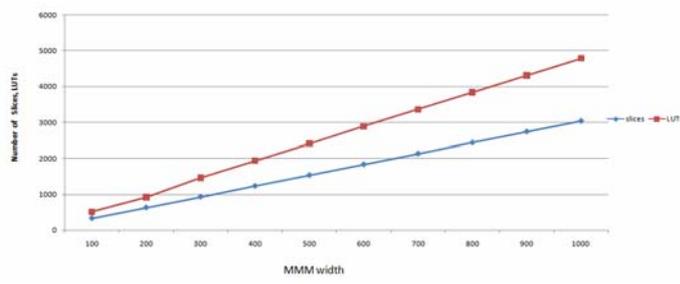
Figure 11
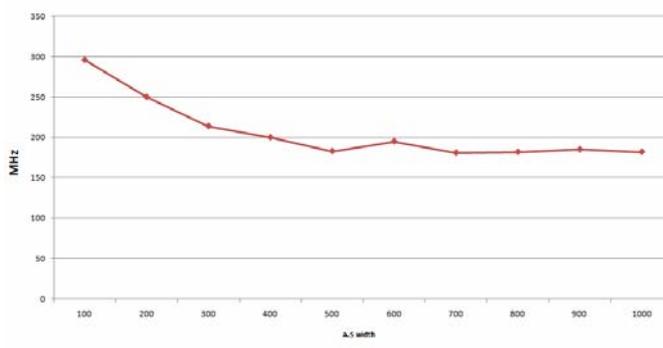
Figure 12

Figure 13

Figure 14