Pushing the Limits of High-Speed $GF(2^m)$ Elliptic Curve Scalar Multiplication on FPGAs

Chester Rebeiro, Sujoy Sinha Roy, and Debdeep Mukhopadhyay

Department of Computer Science and Engineering
Indian Institute of Technology Kharagpur, India
{*chester,sujoyetc,debdeep*}*@cse.iitkgp.ernet.in*

Abstract. In this paper we present an FPGA implementation of a high-speed elliptic curve scalar multiplier for binary finite fields. High speeds are achieved by boosting the operating clock frequency while at the same time reducing the number of clock cycles required to do a scalar multiplication. To increase clock frequency, the design uses optimized implementations of the underlying field primitives and a mathematically analyzed pipeline design. To reduce clock cycles, a new scheduling scheme is presented that allows overlapped processing of scalar bits. The resulting scalar multiplier is the fastest reported implementation for generic curves over binary finite fields. Additionally, the optimized primitives leads to area requirements that is significantly lesser compared to other high-speed implementations. Detailed implementation results are furnished in order to support the claims.

Keywords. Elliptic curve scalar multiplication, FPGA, high-speed implementation, Montgomery ladder

1 Introduction

Elliptic curve cryptography (ECC) is an asymmetric key cipher adopted by the IEEE [21] and NIST [22] as it offers more security per key bit compared to other contemporary ciphers. Security in ECC based cryptosystems is achieved through elliptic curve scalar multiplication. The complex finite field operations involved in ECC often mandates dedicated accelerators for cryptographic and cryptanalytic applications. Field programmable gate arrays (FPGAs) are a popular platform for accelerating curve scalar multiplication due to features such as in-house programmability, shorter time to market, reconfigurability, low non-recurring costs, and simpler design cycles [23]. However, the constrained resources, large granularity, and high costs of routing makes the development of high-speed hardware on FPGAs difficult. The challenges involved in development with FPGAs have led to several published articles on high-speed designs of elliptic curve scalar multiplication for FPGA platforms [1–3, 5, 6, 9, 10, 17]. For binary finite fields over generic curves, most notable works are by Chelton and Benaissa in [5] and more recently Azarderakhsh and Reyhani-Masoleh in [2]. Chelton and Benaissa

are capable of doing a scalar multiplication in $19.5\mu sec$, while Azarderakhsh and Reyhani-Masoleh's implementation requires $17.2\mu sec$. In this paper, we propose an elliptic curve multiplier (ECM) capable of doing scalar multiplications in $10.7\mu sec$ in the same finite field and FPGA family as [5] and [2].

The speed of a hardware design is dictated by 2 parameters: the frequency of the clock and the number of clock cycles required to perform the computation. One method to boost the maximum operable clock frequency is by reducing area. Smaller area generally implies lesser routing delay, which in turn implies higher operable clock frequencies. Another method to increase clock frequency is by pipelining. Chelton and Benaissa [5] extensively rely on this in order to achieve high-speeds. However extensive pipelining in the design is likely to increase the clock cycles required for the computation. Clock cycles can be reduced by parallelization, efficient scheduling, and advanced pipeline techniques such as data-forwarding. Parallelization by replication of computing units was used in [2] to achieve high speeds. The drawback of parallelization however is the large area requirements. Our ECM achieves high-speeds by (1) reducing area, (2) appropriate usage of FPGA hardware resources, (3) optimal pipelining enhanced with data-forwarding, (4) and efficient scheduling mechanisms.

The area requirements of the ECM is primarily due to the finite field arithmetic primitives, in particular multiplication and inversion. In [17], it was shown that an ECM developed with highly optimized field primitives is capable of achieving high computation speeds in-spite of using a naïve scalar multiplication algorithm, no pipelining, or parallelization. Our choice of finite field primitives is based on [17], and has an area requirement which is 50% lesser than [5] and 37% lesser than [2]. The reduced area results in better routing thus leading to increased operating frequencies. Besides the finite field primitives, the registers used in the ECM contribute significantly to the area. Each register in the ECM stores a field element, which can be large. Besides, there are several such registers present. We argue that the area as well as delay can be reduced by placing the registers efficiently in the FPGA.

Ideally, an L stage pipeline can boost the clock frequency up to L times. In order to achieve the maximum effectiveness of the pipelines, the design should be partitioned into L equal stages. That is, each stage of the pipeline should have the same delay. However to date the only means of achieving this is by trial-and-error. In this paper, we show that a theoretical model for FPGA designs, when applied for the ECM, can be used to first estimate the delay in the critical path and there by find the ideal pipelining. As L increases, there is likely to be more data dependencies in the computations, thus resulting in more stalls (*bubbles*) in the pipeline. The paper investigates scheduling strategies for the Montgomery scalar multiplication algorithm, which is an efficient method for pipelining the ECM [13] Compared to [5], which also uses the Montgomery ladder, our scheduling techniques require $3m$ clock cycles lesser for scalar multiplication in the field $GF(2^m)$.

The structure of the paper is as follows: Section 2 has the brief mathematical background required to understand this paper. The organization of the ECM

is discussed in Section 3. Section 4 formally analyzes pipelining the ECM while Section 5 discusses the scheduling of instructions in the pipeline. Section 6 determines the right pipeline for the ECM and Section 7 presents the architecture of the ECM with the *right pipeline*. Implementation results are presented and compared the state-of-the-art in Section 8, while the final section has the conclusion for the paper.

2 Background

An elliptic curve is either represented by 2 point *affine coordinates* or 3 point *projective coordinates*. The smaller number of finite field inversions required by projective coordinates makes it the preferred coordinate system. For the field $GF(2^m)$, the equation for an elliptic curve in projective coordinates is $Y^2 + XYZ = X^3Z + aX^2Z^2 + bZ^4$, where the curve constants a and $b \in GF(2^m)$ and $b \neq 0$. The points on the elliptic curve together with the point at infinity ($\mathcal{O}$) form an Abelian group under addition with group operations *point addition* and *point doubling*. For a given point P on the curve (called the *base point*)

Algorithm 1: *Montgomery Point Multiplication*

```
   Input: Base point P and scalar s = {s_{t-1} s_{t-2} ... s_0}_2 with s_{t-1} = 1
   Output: Point on the curve Q = sP
 1 begin
 2     P_1(X_1, Z_1) ← P(X, Z); P_2(X_2, Z_2) ← 2P(X, Z)
 3     for k = t − 2 to 0 do
 4         if s_k = 1 then
 5             P_1 ← P_1 + P_2
 6             P_2 ← 2P_2
 7         end
 8         else
 9             P_2 ← P_1 + P_2
10             P_1 ← 2P_1
11         end
12     end
13     return Q ← Projective2Affine(P_1, P_2)
14 end
```

and a scalar s, *scalar multiplication* is the computation of the scalar product sP. Algorithm 1 depicts the Montgomery algorithm [11, 14] for computing sP. For each bit in s, a point addition followed by a point doubling is done (lines 5,6 and 9,10). In these operations (listed in Equation 1) only the X and Z coordinates of the points are used.

$$\begin{aligned} X_i \leftarrow X_i \cdot Z_j \ ; \ Z_i \leftarrow X_j \cdot Z_i \ ; \ T \leftarrow X_j \ ; \ X_j \leftarrow X_j^4 + b \cdot Z_j^4 \\ Z_j \leftarrow (T \cdot Z_j)^2 \ ; \ T \leftarrow X_i \cdot Z_i \ ; \ Z_i \leftarrow (X_i + Z_i)^2 \ ; \ X_i \leftarrow x \cdot Z_i + T \end{aligned} \quad (1)$$

Depending on the value of the bit s_k, operand and destination registers for the point operations vary. When $s_k = 1$ then $i = 1$ and $j = 2$, and when $s_k = 0$ then $i = 2$ and $j = 1$. The final step in the algorithm, $Projective2Affine(\cdot)$, converts the 3 coordinate scalar product in to the acceptable 2 coordinate affine form. This step involves a finite field inversion along with 9 other multiplications [13].

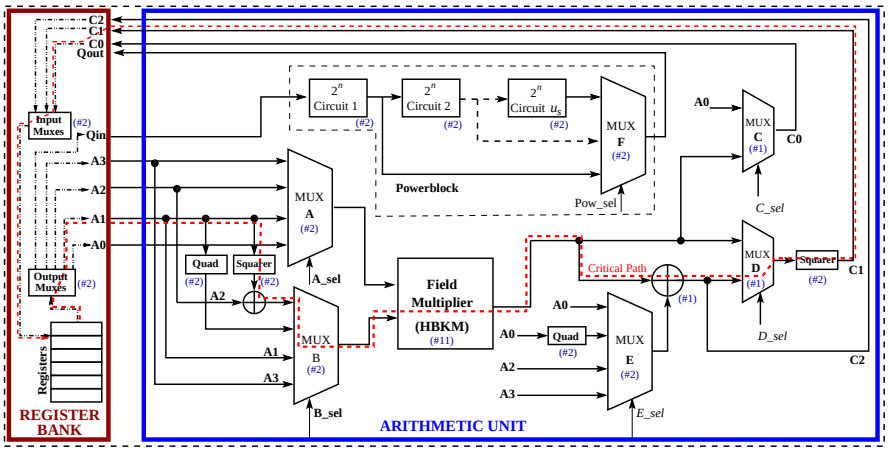


Fig. 1. Block Diagram of the Processor Organization

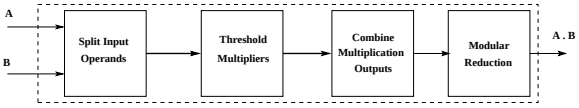


Fig. 2. Different Stages in the HBKM

3 The Processor Organization

The functionality of the ECM is to execute Algorithm 1. It comprises of 2 units: the register bank and the arithmetic unit as seen in Figure 1. In each clock cycle, control signals are generated according to the value of the bit s_k, which reads operands from the register bank, performs the computation in the arithmetic unit, and finally write back the results. In this section we present the architecture for the register bank and arithmetic units.

3.1 Arithmetic Unit

The *field multiplier* is the central part of the arithmetic unit. We choose to use a *hybrid bit-parallel Karatsuba field multiplier* (HBKM), which was first introduced in [18] and then used in [17]. The advantage of the HBKM is the sub-quadratic complexity of the Karatsuba algorithm coupled with efficient utilization of the FPGA's LUT resources. Further, the bit-parallel scheme requires lesser clock cycles compared to digit level multipliers used in [2]. The HBKM recursively splits the input operands until a threshold (τ) is reached, then threshold (school-book) multipliers are applied. The outputs of the threshold multipliers are combined and then reduced (Figure 2).

Field inversion is performed by a generalization of the Itoh-Tsujii inversion algorithm for FPGA platforms [19]. The generalization requires a cascade of

2^n exponentiation circuits (implemented as the *Powerblock* in Figure 1), where $1 \leq n \leq m-1$. The ideal choice for n depends on the field and the FPGA platform. For example, in $GF(2^{163})$ and FPGAs having 4 input LUTs (such as Xilinx Virtex 4), the optimal choice for n is 2. More details on choosing n can be found in [8]. The number of cascades, u_s, depends on the critical delay of the ECM and will be discussed in Section 4. Further, an addition chain for $\lfloor \frac{m-1}{n} \rfloor$ is required. Therefore, for $GF(2^{163})$ and $n = 2$, an addition chain for 81 is needed. The number of clock cycles required for inversion, assuming a Brauer chain, is given by Equation 2, where the addition chain has the form $(u_1, u_2, \cdots, u_l)$, and L is the number of pipeline stages in the ECM [4].

$$cc_{ita} = L(l+1) + \sum_{i=2}^{l} \left\lceil \frac{u_i - u_{i-1}}{u_s} \right\rceil \tag{2}$$

3.2 Register Bank

There are six registers in the register bank, each capable of storing a field element. Five of the registers are used for the computations in Equation 1, while one is used for field inversion. There are 3 ways in which the registers can be implemented in FPGAs. The first approach, using *block RAM*, is slow due to constraints in routing. The two other alternatives are *distributed RAM* and flip-flops. Distributed RAM allows the FPGA's LUTs to be configured as RAM. Each bit of the 6 registers will share the same LUT. However each register is used for a different purpose therefore the centralization effect of distributed RAM will cause long routes, leading to lowering of clock frequencies. Additionally, there is an impact on the area requirements. Flip-flops on the other hand allow de-centralization of the registers, there by allowing registers to be placed in locations close to their usage, thus routing is easier. Further, each slice in the FPGA has equal number of LUTs and flip-flops. The ECM is an LUT intensive design, due to which several of the flip-flops in the slice remain unutilized. By configuring the registers to make use of these flip-flops, no additional area (in terms of the number of slices) is required.

4 Pipelining the ECM

All combinational data paths in the ECM start from the register bank output and end at the register bank input. The maximum operable frequency of the ECM is dictated by the longest combinational path, known as the *critical path*. There can be several critical paths, one such example is highlighted through the (red) dashed line in Figure 1.

Estimating Delays in the ECM : Let t^*_{cp} be the delay of the critical paths and $f^*_1 = \frac{1}{t^*_{cp}}$ the maximum operable frequency of the ECM prior to pipelining. Consider the case of pipelining the ECM into L stages, then the maximum operable frequency can be increased to at-most $f^*_L = L \times f^*_1$. This *ideal frequency* can be achieved if and only if the following two conditions are satisfied.

Table 1. LUT delays of Various Combinational Circuit Components

Component	$\mathrm{k} - LUT$ **Delay for** $\mathrm{k} \geq 4$	$m = 163, \mathrm{k} = 4$
m bit field adder	1	1
m bit $n:1$ Mux ($D_{n:1}(m)$)	$\lceil log_{\mathrm{k}}(n + log_2 n) \rceil$	2 (for $n = 4$) 1 (for $n = 2$)
Exponentiation Circuit ($D_{2^n}(m)$)	$max(LUTDelay(d_i))$, where d_i is the i^{th} output bit of the exponentiation circuit	2 (for $n = 1$) 2 (for $n = 2$)
Powerblock ($D_{powerblk}(m)$)	$u_s \times D_{2^n}(m) + D_{u_s:1}(m)$	4 (for $u_s = 2$)
Modular Reduction (D_{mod})	1 for irreducible trinomials 2 for pentanomials	2 (for pentanomials)
HBKM ($D_{\mathsf{HBKM}}(m)$)	As seen in Figure 2, this can be written as $D_{split} + D_{threshold} + D_{combine} + D_{mod}$ $= \lceil log_k(\frac{m}{\tau}) \rceil + \lceil log_k(2\tau) \rceil$ $+ \lceil log_2(\frac{m}{\tau}) \rceil + D_{mod}$	11 (for $\tau = 11$)

1. Every critical path in the design should be split into L stages with each stage having a delay of exactly $\frac{t^*_{cp}}{L}$.
2. All other paths in the design should be split so that any stage in these paths should have a delay which is less than or equal to $\frac{t^*_{cp}}{L}$.

While it is not always possible to exactly obtain f^*_L, we can achieve close to the ideal clock frequency by making a theoretical estimation of t^*_{cp} and then identifying the locations in the architecture where the pipeline stages have to be inserted. We denote this theoretical estimate of delay by $t^{\#}_{cp}$. The theoretical analysis is based on the following prepositions. These propositions were first stated in [19] and used to design high-speed inversion circuits. Their correctness have been extensively validated in [19] for 4 and 6 input LUT based FPGAs.

Proposition 1. *[19] For circuits which are implemented using LUTs, the delay of a path in the circuit is proportional to the number of LUTs in the path.*

Proposition 2. *[19] The number of LUTs in the critical path of an n variable Boolean function having the form $y = g_n(x_1, x_2, \cdots, x_n)$ is given by $\lceil log_{\mathrm{k}}(n) \rceil$, where* k *is the number of inputs to the LUTs ($\mathrm{k} - LUT$).*

Using these two propositions it is possible to analyze the delay of various combinational circuit components in terms of LUTs. The *LUT delays* of relevant combinational components are summarized in Table 1. The reader is referred to [19] for detailed analysis of the LUT delays. The LUT delays of all components in Figure 1 are shown in parenthesis for $\mathrm{k} = 4$. Note that the analysis also considers optimizations by the synthesis tool (such as the merging of the squarer and adder before Mux B (Figure 1), which reduces the delay from 3 to 2).

Pipelining Paths in the ECM : Table 1 can be used to determine the LUT delays of any path in the ECM. For the example critical path, (the red dashed

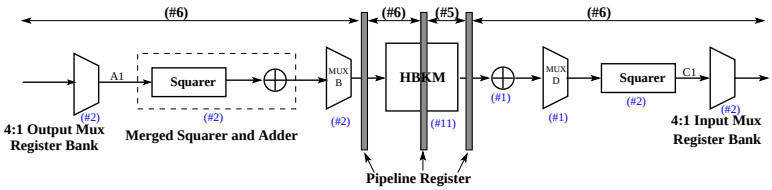


Fig. 3. Example Critical Path (with $L = 4$ and k = 4)

line in Figure 1), the estimate for t^*_{cp} is the sum of the LUT delays of each component in the path. This evaluates to $t^{\#}_{cp} = 23$. Figure 3 gives a detailed view of this path. Pipelining the paths in the ECM require pipeline registers to be introduced in between the LUTs. The following proposition determines how the pipeline registers have to be inserted in a path in order to achieve the maximum operable frequency ($f^{\#}_L$) as close to the ideal (f^*_L) as possible (Note that $f^{\#}_L \leq f^*_L$).

Proposition 3. *If $t^{\#}_{cp}$ is the LUT delay of the critical paths, and L is the desired number of stages in the pipeline, then the best clock frequency ($f^{\#}_L$) is achieved only if no path has delay more than $\lceil \frac{t^{\#}_{cp}}{L} \rceil$.*

For example for $L = 4$, no path should have a LUT delay more than $\lceil \frac{23}{4} \rceil$. This identifies the exact locations in the paths where pipeline registers have to be inserted. Figure 3 shows the positions of the pipeline register for $L = 4$ for the critical path.

On the Pipelining of the Powerblock : The powerblock is used only once during the computation; at the end of the scalar multiplication. There are two choices with regard to implementing the powerblock, either pipeline the powerblock as per Proposition 3 or reduce the number of 2^n circuits in the cascade so that the following LUT delay condition is satisfied (refer Table 1),

$$D_{powerblock}(m) \leq \lceil \frac{t^{\#}_{cp}}{L} \rceil - 1 \tag{3}$$

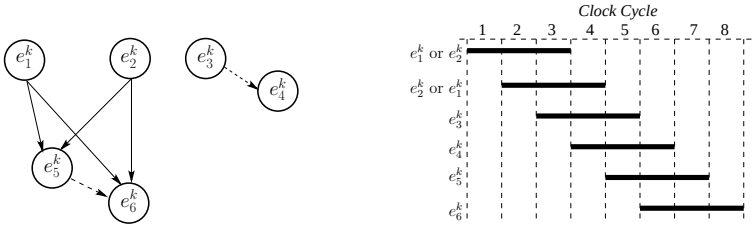
, where -1 is due to the output mux in the register bank. However the sequential nature of the Itoh-Tsujii algorithm [7] ensures that the result of one step is used in the next. Due to the data dependencies which arise the algorithm is not suited for pipelining and hence the latter strategy is favored. For $k = 4$ and $m = 163$, the optimal exponentiation circuit is $n = 2$ having an LUT delay of 2 [19]. Thus a cascade of two 2^2 circuits would best satisfy the inequality in (3).

5 Scheduling for the ECM

In this section we discuss the scheduling of the addition-doubling loop in Algorithm 1. For each bit in the scalar (s_k), the eight operations in Equation 1 are

Table 2. Scheduling Instructions for the ECM

$e_1^k: \ X_i \leftarrow X_i \cdot Z_j$	$e_4^k: \ Z_j \leftarrow (T \cdot Z_j)^2$
$e_2^k: \ Z_i \leftarrow X_j \cdot Z_i$	$e_5^k: \ T \leftarrow X_i \cdot Z_i; \ Z_i \leftarrow (X_i + Z_i)^2$
$e_3^k: \ T \leftarrow X_j; \ X_j \leftarrow X_j^4 + b \cdot Z_j^4$	$e_6^k: \ X_i \leftarrow x \cdot Z_i + T$



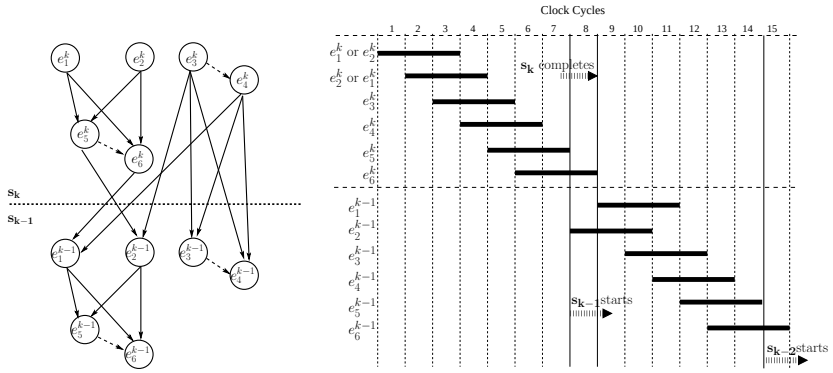
(a) Data Dependencies (b) Timing Diagram

Fig. 4. Scheduling the Scalar Bit s_k

computed. Unlike [2], where 2 finite field multipliers are used in the architecture, we, like [5], use a single field multiplier. This restriction makes the field multiplier the most critical resource in the ECM as Equation 1 involves six field multiplications, which have to be done sequentially. The remaining operations comprise of additions, squarings, and data transfers can be done in parallel with the multiplications. Equation 1 can be rewritten as in Table 5 using 6 instructions, with each instruction capable of executing simultaneously in the ECM.

Proper scheduling of the 6 instructions is required to minimize the impact of data dependencies, thus reducing pipeline stalls. The dependencies between the instructions e_1^k to e_6^k are shown in Figure 4(a). In the figure a *solid arrow* implies that the subsequent instruction cannot be started unless the previous instruction has completed, while a *dashed arrow* implies that the subsequent instruction cannot be started unless the previous instruction has started. For example e_6^k uses Z_i, which is updated in e_5^k. Since the update does not require a multiplication (an addition followed by a squaring here), it is completed in one clock cycle. Thus e_5^k to e_6^k has a dashed arrow, and e_6^k can start one clock cycle after e_5^k. On the other hand, dependencies depicted with the solid arrow involve the multiplier output in the former instruction. This will take L clock cycles, therefore a longer wait.

The dependency diagram shows that in the longest dependency chain, e_5^k and e_6^k has dependency on e_1^k and e_2^k. Thus e_1^k and e_2^k are scheduled before e_3^k and e_4^k. Since the addition in e_6^k has a dependency on e_5^k, operation e_5^k is triggered just after completion of e_1^k and e_2^k; and operation e_6^k is triggered in the next clock cycle. When $L \geq 3$, the interval between starting and completion of e_1^k and e_2^k can be utilized by scheduling e_3^k and e_4^k. Thus, the possible scheduling schemes



(a) Data Dependencies (b) Timing Diagram

Fig. 5. Schedule for Two Scalar Bits when $s_{k-1} = s_k$

for the 6 instructions is

$$(\{e_1^k, e_2^k\}, e_3^k, e_4^k, e_5^k, e_6^k) \tag{4}$$

Where { } implies that there is no strict order in the scheduling (either e_1 or e_2 can be scheduled first). An example of a scheduling for $L = 3$ is shown in Figure 4(b). For $L \geq 3$[1], the number of clock cycles required for each bit in the scalar is $2L + 2$. In the next part of this section we show that the clock cycles can be reduced to $2L + 1$ (and in some cases $2L$) if two consecutive bits of the scalar are considered.

5.1 Scheduling for Two Consecutive Bits of the Scalar

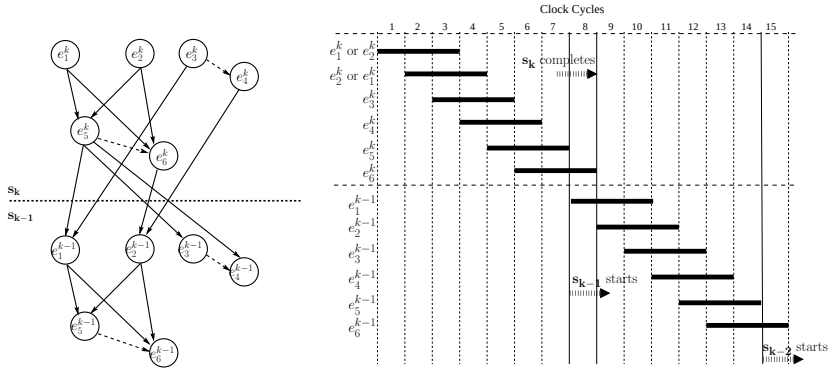
Consider the scheduling of operations for two bits of the scalar, s_k and s_{k-1} (Algorithm 1). We assume that the computation of bit s_k is completed and the next bit s_{k-1} is to be scheduled. Two cases arise: $s_{k-1} = s_k$ and $s_{k-1} \neq s_k$. We consider each case separately.

When the Consecutive Key Bits are Equal : Figure 5(a) shows the data dependencies when the two bits are equal. The last two instructions to complete for the s_k bit are e_5^k and e_6^k. For the subsequent bit (s_{k-1}), either e_1^{k-1} or e_2^{k-1} has to be scheduled first according to the sequence in (4). We see from Figure 5(a) that e_1^{k-1} depends on e_6^k, while e_2^{k-1} depends on e_5^k. Further, since e_5^k completes earlier than e_6^k, we schedule e_2^{k-1} before e_1^{k-1}. Thus the scheduling for 2 consecutive equal bits is

$$(\{e_1^k\ , e_2^k\}\ , e_3^k\ , e_4^k\ , e_5^k\ , e_6^k\ , e_2^{k-1}, e_1^{k-1}, e_3^{k-1}, e_4^{k-1}, e_5^{k-1}, e_6^{k-1}\ )$$

An example is shown in Figure 5(b).

[1]The special case of $L <= 2$ can trivially be analyzed. The clock cycles required in this case is six.



(a) Data Dependencies (b) Timing Diagram

Fig. 6. Schedule for Two Scalar Bits when $s_{k-1} \neq s_k$

When the Consecutive Key Bits are Not Equal : Figure 6(a) shows the data dependency for two consecutive scalar bits that are not equal. Here it can be seen that e_1^{k-1} and e_2^{k-1} depend on e_5^k and e_6^k respectively. Since, e_5^k completes before e_6^k, we schedule e_1^{k-1} before e_2^{k-1}. The scheduling for two consecutive bits is as follows

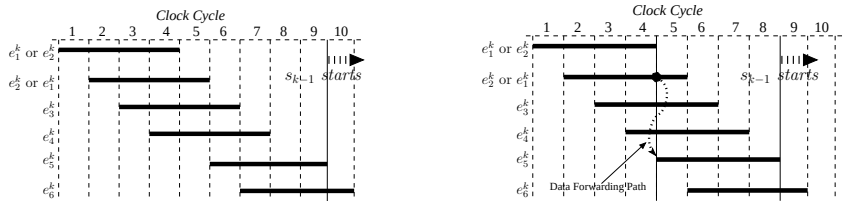
$$(\{e_1^k\ , e_2^k\}\ , e_3^k\ , e_4^k\ , e_5^k\ , e_6^k\ , e_1^{k-1}, e_2^{k-1}, e_3^{k-1}, e_4^{k-1}, e_5^{k-1}, e_6^{k-1}\ )$$

An example is shown in Figure 6(b).

Effective Clock Cycle Requirement : Starting from e_1^k (or e_2^k), completion of e_2^k (or e_1^k) takes $L+1$ clock cycles, for an L stage pipelined ECM. After completion of e_1^k and e_2^k, e_5^k starts. This is followed by e_6^k in the next clock cycle. So in all $2L+2$ clock cycles are required. The last clock cycle however is also used for the next bit of the scalar. So effectively the clock cycles required per bit is $2L+1$. Compared to the work in [5], our scheduling strategy saves two clock cycles for each bit of the scalar. For an m bit scalar, the saving in clock cycles compared to [5] is $2m$. Certain values of L allow data forwarding to take place. In such cases the clock cycles per bit reduces to $2L$, thus saving $3m$ clock cycles compared to [5].

5.2 Data Forwarding to Reduce Clock Cycles

For a given value of L, Proposition 3 specifies where the pipeline registers have to be placed in the ECM. If the value of L is such that a pipeline register is placed at the output of the field multiplier, then data forwarding can be applied to save one clock cycle per scalar bit. For example, consider $L = 4$. This has pipeline registers placed immediately after the multiplier as shown in Figure 3. This register can



(a) Without Data Forwarding (b) With Data Forwarding

Fig. 7. Effect of Data Forwarding in the ECM for $L = 4$

be used to start the instruction e_5^k one clock cycle earlier. Figure 5.1 compares the execution of a single bit with and without data forwarding. Though e_2^k (or e_1^k) finishes in the fifth clock cycle, the result of the multiplication is latched into the pipeline register after the fourth clock cycle. With data forwarding from this register, we start e_5^k from the fifth clock cycle, thus reducing clock cycle requirement by one to $2L$.

6 Finding the Right Pipeline

The time taken for a scalar multiplication in the ECM is the product of the number of clock cycles required and the time period of the clock. For an L stage pipeline, Section 4 determines the best time period for the clock. In this section we would first estimate the number of clock cycles required and then analyze the effect of L on the computation time.

6.1 Number of Clock Cycles

There are two parts in Algorithm 1. First the scalar multiplication in projective coordinates and then the conversion to affine coordinates. The conversion comprises of finding an inverse and 9 multiplications. The clock cycles required is given by $cc_{2scm} = cc_{3scm} + cc_{ita} + cc_{conv}$.

cc_{3scm} is the clock cycles required for the scalar multiplication in projective coordinates. From the analysis in Section 5 this can be written as $2mL$ if data forwarding is possible and $m(2L + 1)$ otherwise. For the conversion to affine coordinates, finding the inverse requires cc_{ita} clock cycles (from Equation 2), while the 9 multiplications following the inverse requires cc_{conv} clock cycles. The value of cc_{conv} for the ECM was found to be $7 + 9L$. Thus,

$$cc_{2scm} = \left\lceil cc_{3scm} \right\rceil + \left\lceil L(l+1) + \sum_{i=2}^{l} \left\lceil \frac{u_i - u_{i-1}}{u_s} \right\rceil \right\rceil + \left\lceil 7 + 9L \right\rceil \qquad (5)$$

Table 3. Computation Time Estimates for Various Values of L for an ECM over $GF(2^{163}$ and FPGA with 4 input LUTs

L	u_s	DataForwarding Feasible	cc_{3scm}	cc_{ita}	cc_{conv}	cc_{2scm}	ct
1	9	No	978	25	16	1019	$1019t_{cp}^{\#}$
2	4	No	978	44	25	1047	$524t_{cp}^{\#}$
3	3	No	1141	61	34	1236	$412t_{cp}^{\#}$
4	2	Yes	1304	82	43	1429	$357t_{cp}^{\#}$
5	1	No	1793	130	52	1975	$395t_{cp}^{\#}$
6	1	Yes	1956	140	61	2157	$360t_{cp}^{\#}$
7	1	Yes	2282	150	70	2502	$358t_{cp}^{\#}$

6.2 Analyzing Computation Time

The procedure involved in analyzing the computation time for an L stage pipeline is as follows.

1. Determine $t_{cp}^{\#}$ (the LUT delay of the critical path of the combinational circuit) using Table 1.
2. Compute the maximum operable frequency ($\lceil \frac{t_{cp}^{\#}}{L} \rceil$) and determine the locations of the pipeline registers. Therefore determine if data forwarding is possible.
3. Determine u_s, the number of cascades in the power block, using Equation 3 and the delay of a single 2^n block (Table 1).
4. Compute cc_{2scm}, using Equation 5.
5. The computation time ct is given by $cc_{2scm} \times \lceil \frac{t_{cp}^{\#}}{L} \rceil$.

For an ECM over $GF(2^{163})$, the threshold for the HBKM set as 11, an addition chain of $(1, 2, 4, 5, 10, 20, 40, 80, 81)$, and 2^2 exponentiation circuits in the power block, the $t_{cp}^{\#}$ is 23. The estimated computation time for various values of L are given in Table 3. The cases $L = 1$ and $L = 2$ are special as for these $cc_{3scm} = 6m$. The table clearly shows that the least computation time is obtained when $L = 4$.

7 Detailed Architecture of the ECM

Figure 8 shows the detailed architecture for $L = 4$. The input to the architecture is the scalar, reset signal, and the clock. At reset, the curve constants and base point are loaded from ROM. At every clock cycle, the control unit generates signals for the register bank and the arithmetic unit. Registers are selected through multiplexers in the register bank and fed to the arithmetic unit through the buses $A0$, $A1$, $A2$, $A3$, and Qin. Multiplexers again channel the data into the multiplier. The results are written back into the registers through buses $C0$, $C1$, $C2$, $Qout$. Note the placement of the pipeline registers dividing the circuit in 4 stages and ensuring that each stage has an LUT delay which is less than or

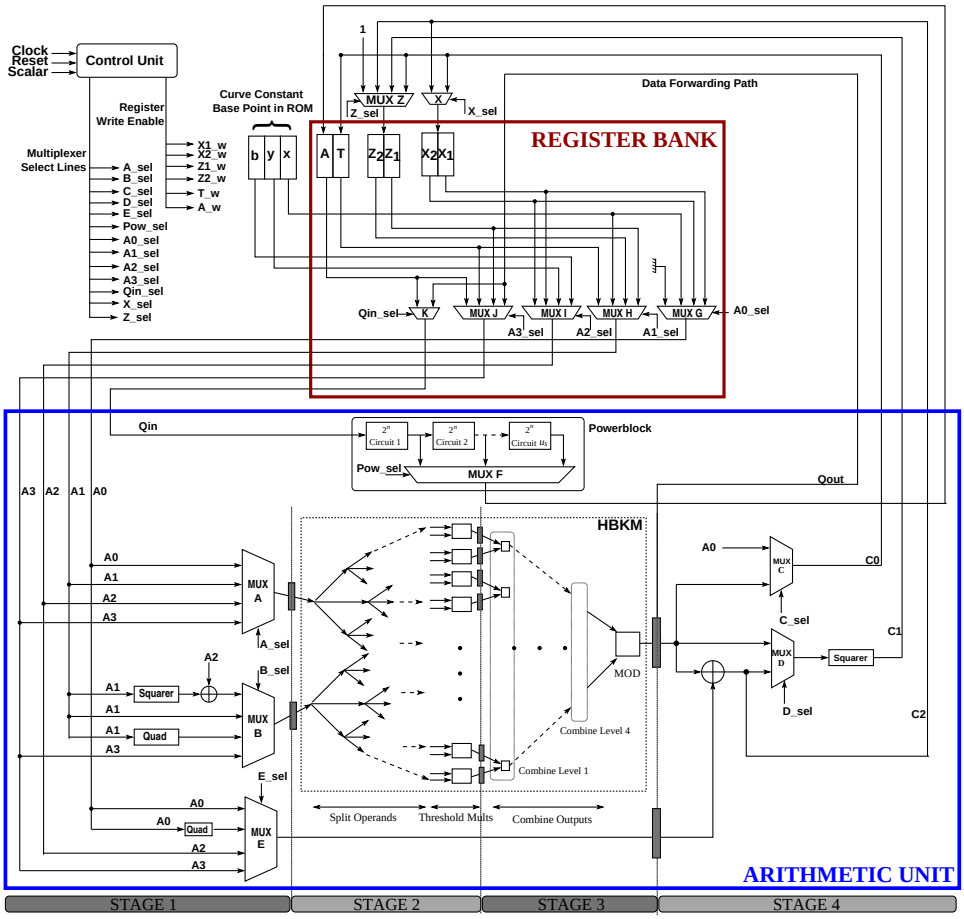


Fig. 8. Detailed Architecture for a 4 Stage Pipelined ECM

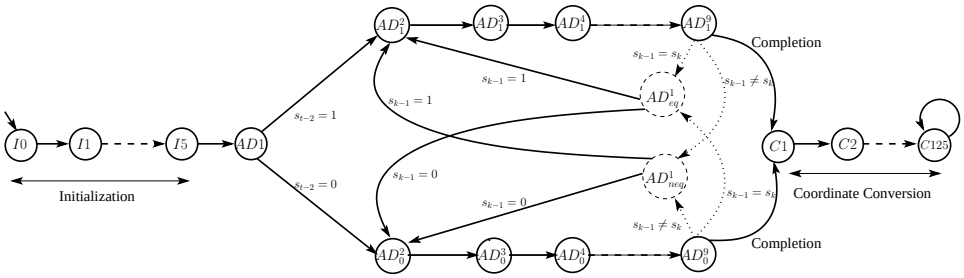


Fig. 9. Finite State Machine for 4-Stage ECM

equal to $\lceil \frac{23}{4} \rceil = 6$. Note also the pipeline register present immediately after the field multiplier (HBKM) used for data forwarding.

Table 4. Comparison of the Proposed ECM with FPGA based Published Results

Work	Platform	Field (m)	Slices	LUTs	Freq (MHz)	Comp. Time (μs)
Orlando [15]	XCV400E	163	-	3002	76.7	210
Bednara [3]	XCV1000	191	-	48300	36	270
Gura [6]	XCV2000E	163	-	19508	66.5	140
Lutz [12]	XCV2000E	163	-	10017	66	233
Saqib [20]	XCV3200	191	18314	-	10	56
Pu [16]	XC2V1000	193	-	3601	115	167
Ansari [1]	XC2V2000	163	-	8300	100	42
Rebeiro [17]	XC4V140	233	19674	37073	60	31
Järvinen[1] [9]	Stratix II	163	(11800ALMs)	-	-	48.9
Kim [2] [10]	XC4VLX80	163	24363	-	143	10.1
Chelton [5]	XCV2600E	163	15368	26390	91	33
	XC4V200	163	16209	26364	153.9	19.5
Azarderakhsh[3] [2]	XC4CLX100	163	12834	22815	196	17.2
	XC5VLX110	163	6536	17305	262	12.9
Our Result (Virtex 4 FPGA)	XC4VLX80	163	8070	14265	147	9.7
	XC4V200	163	8095	14507	132	10.7
	XC4VLX100	233	13620	23147	154	12.5
Our Result (Virtex 5 FPGA)	XC5VLX85t	163	3446	10176	167	8.6
	XC5VSX240	163	3513	10195	148	9.5
	XC5VLX85t	233	5644	18097	156	12.3

1. uses 4 field multipliers; 2. uses 3 field multipliers; 3. uses 2 field multipliers

Figure 9 shows the finite state machine for $L = 4$. The states $I0$ to $I5$ are used for initialization (line 2 in Algorithm 1). State $AD1$ represents the first clock cycle for the scalar bit s_{t-2}. States AD_1^2 to AD_1^9 represent the computations when $s_k = 1$, while AD_0^2 to AD_0^9 are for $s_k = 0$. Each state corresponds to a clock cycle in Figure 7(b). Processing for the next scalar bit (s_{k-1}) begins in the same clock cycle as AD_0^9 and AD_1^9 in states AD_{eq}^1 and AD_{neq}^1. The states AD_{eq}^1 or AD_{neq}^1 are entered depending on the equality of s_k and s_{k-1}. If $s_k = s_{k-1}$ then AD_{eq}^1 is entered, else AD_{neq}^1 is entered. After processing of all scalar bits is complete, the conversion to affine coordinates ($cc_{ita} + cc_{conv}$) takes place in states C_1 to C_{125}.

8 Implementation Results and Comparisons

We evaluated the ECM using Xilinx Virtex 4 and Virtex 5 platforms. Table 4 shows the place and route results using the Xilinx ISE tool. There have been several implementations of elliptic curve processors on different fields, curves, platforms, and for different applications. Due to the vast variety of implementations available, we restrict comparisons with FPGA implementations for generic elliptic curves over binary finite fields (Table 4). In this section we analyze recent high-speed implementations.

The implementation in [17] is over the field $GF(2^{233})$ and does a scalar multiplication in $31\mu s$. The implementation relied heavily on optimized finite-field primitives and was not pipelined or parallelized. Our implementation on the same field uses enhanced primitives from [17], and therefore has smaller area requirements. Additionally higher speeds are achieved due to efficient pipelining and scheduling of instructions.

The implementation in [5] uses a 7 stage pipeline, thus achieves high operating clock frequency. However, the un-optimized pipeline and large clock cycle requirement limits performance. In comparison, the ECM uses better scheduling

there by saving around $1,600$ clock cycles and a better pipeline, there by obtaining frequencies close to [5], in-spite of having only 4 pipeline stages. Further, efficient field primitives and the sub-quadratic Karatsuba multiplier instead of the quadratic Mastrovito multiplier result in 50% reduction in area on Virtex 4.

In [2], two highly optimized digit field multipliers were used. This enabled parallelization of the instructions and higher clock frequency. However, the use of digit field multipliers resulted in large clock cycle requirement for scalar multiplication (estimated at 3,380). We use a single fully parallel field multiplier requiring only $1,429$ clock cycles and an area which is 37% lesser in Virtex 4.

In [10], a computation time of $10.1\mu s$ was achieved while on the same platform our ECM achieves a computation time of $9.7\mu s$. Although the speed gained is minimal, it should be noted that [10] uses 3 digit-level finite field multipliers compared to one in ours, thus has an area requirement which is about 3 times ours. The compact area is useful especially for cryptanalytic applications where our ECM can test thrice as many keys compared to [10].

9 Conclusion

The papers presents techniques to reduce the computation time for scalar multiplications on elliptic curves. The techniques involve the use of highly optimized finite field primitives and efficient utilization of FPGA resources in order to reduce the area requirements, which in turn leads to better routing, hence higher clock frequencies. Additionally, a theoretical analysis of the data paths, help pipeline the multiplier. Further, efficient scheduling of elliptic curve operations, supported with data-forwarding mechanisms, reduce the number of clock cycles required to execute a scalar multiplication. These mechanisms result in a scalar multiplier that is faster than any other reported implementations, in-spite of having just a single finite field multiplier. The presence of a single optimized field multiplier additionally leads to area requirements, which is considerably lesser than contemporary implementations. Results are presented for generic curves over the field $GF(2^{163})$, however these mechanisms can be applied for other curves and fields as well.

References

1. B. Ansari and M. Hasan. High-performance architecture of elliptic curve scalar multiplication. *Computers, IEEE Transactions on*, 57(11):1443 –1453, Nov. 2008.
2. R. Azarderakhsh and A. Reyhani-Masoleh. Efficient FPGA Implementations of Point Multiplication on Binary Edwards and Generalized Hessian Curves Using Gaussian Normal Basis. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, PP(99):1, 2011.
3. M. Bednara, M. Daldrup, J. von zur Gathen, J. Shokrollahi, and J. Teich. Reconfigurable Implementation of Elliptic Curve Crypto Algorithms. In *Parallel and Distributed Processing Symposium., Proceedings International, IPDPS 2002, Abstracts and CD-ROM*, pages 157–164, 2002.

4. C. Rebeiro, S. S. Roy, D. S. Reddy and D. Mukhopadhyay. Revisiting the Itoh Tsujii Inversion Algorithm for FPGA Platforms. *IEEE Transactions on VLSI Systems.*, 19(8):1508–1512, 2011.
5. W. N. Chelton and M. Benaissa. Fast Elliptic Curve Cryptography on FPGA. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 16(2):198–205, Feb. 2008.
6. N. Gura, S. C. Shantz, H. Eberle, S. Gupta, V. Gupta, D. Finchelstein, E. Goupy, and D. Stebila. An End-to-End Systems Approach to Elliptic Curve Cryptography. In *CHES '02: Revised Papers from the 4th International Workshop on Cryptographic Hardware and Embedded Systems*, pages 349–365, London, UK, 2003. Springer-Verlag.
7. T. Itoh and S. Tsujii. A Fast Algorithm For Computing Multiplicative Inverses in $GF(2^m)$ Using Normal Bases. *Inf. Comput.*, 78(3):171–177, 1988.
8. K. Järvinen. On repeated squarings in binary fields. In M. Jacobson, V. Rijmen, and R. Safavi-Naini, editors, *Selected Areas in Cryptography*, volume 5867 of *Lecture Notes in Computer Science*, pages 331–349. Springer Berlin / Heidelberg, 2009.
9. K. Järvinen and J. Skytta. On parallelization of high-speed processors for elliptic curve cryptography. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 16(9):1162 –1175, sept. 2008.
10. C. H. Kim, S. Kwon, and C. P. Hong. FPGA Implementation of High Performance Elliptic Curve Cryptographic processor over $GF(2^{163})$. *Journal of Systems Architecture - Embedded Systems Design*, 54(10):893–900, 2008.
11. J. López and R. Dahab. Fast multiplication on elliptic curves over gf(2m) without precomputation. In *Proceedings of the First International Workshop on Cryptographic Hardware and Embedded Systems*, CHES '99, pages 316–327, London, UK, UK, 1999. Springer-Verlag.
12. J. Lutz and A. Hasan. High Performance FPGA based Elliptic Curve Cryptographic Co-Processor. In *ITCC '04: Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC'04) Volume 2*, page 486, Washington, DC, USA, 2004. IEEE Computer Society.
13. A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 2001.
14. P. L. Montgomery. Speeding the pollard and elliptic curve methods of factorization. In *Mathematics of Computation*, volume 48, pages 243–264, January 1987.
15. G. Orlando and C. Paar. A High Performance Reconfigurable Elliptic Curve Processor for $GF(2^m)$. In *CHES '00: Proceedings of the Second International Workshop on Cryptographic Hardware and Embedded Systems*, pages 41–56, London, UK, 2000. Springer-Verlag.
16. Q. Pu and J. Huang. A Microcoded Elliptic Curve Processor for $GF(2^m)$ Using FPGA Technology. In *Communications, Circuits and Systems Proceedings, 2006 International Conference on*, volume 4, pages 2771–2775, June 2006.
17. C. Rebeiro and D. Mukhopadhyay. High Speed Compact Elliptic Curve Cryptoprocessor for FPGA Platforms. In D. R. Chowdhury, V. Rijmen, and A. Das, editors, *INDOCRYPT*, volume 5365 of *Lecture Notes in Computer Science*, pages 376–388. Springer, 2008.
18. C. Rebeiro and D. Mukhopadhyay. Power Attack Resistant Efficient FPGA Architecture for Karatsuba Multiplier. In *VLSID '08: Proceedings of the 21st International Conference on VLSI Design*, pages 706–711, Washington, DC, USA, 2008. IEEE Computer Society.

19. S. S. Roy, C. Rebeiro, and D. Mukhopadhyay. Theoretical Modeling of the Itoh-Tsujii Inversion Algorithm for Enhanced Performance on k-LUT based FPGAs. In *Design, Automation, and Test in Europe DATE-2011*, 2011.
20. N. A. Saqib, F. Rodríguez-Henríquez, and A. Diaz-Perez. A Parallel Architecture for Fast Computation of Elliptic Curve Scalar Multiplication Over $GF(2^m)$. In *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings*, Apr. 2004.
21. I. C. Society. IEEE Standard Specifications for Public-key Cryptography, 2000.
22. U.S. Department of Commerce,National Institute of Standards and Technology. Digital signature standard (DSS), 2000.
23. T. Wollinger, J. Guajardo, and C. Paar. Security on FPGAs: State-of-the-art Implementations and Attacks. *Trans. on Embedded Computing Sys.*, 3(3):534–574, 2004.

Appendix A

In this appendix we summarize the ideal design parameters for k = 4 (Xilinx Virtex 4 FPGA) and k = 6 (Xilinx Virtex 5) for the field $GF(2^{163})$.

Table 5. Summary of Design Parameters for $GF(2^{163})$ for k = 4 and k = 6

Parameter	k = 4	k = 6
Threshold used in HBKM (τ)	11	11
Exponentiation Circuit in Powerblock	2^2 circuit (Quad)	2^4 circuit
Addition Chain	(1, 2, 4, 5, 10, 20, 40, 80, 81)	(1, 2, 4, 5, 10, 20, 40)
Number of Cascades in Powerblock (u_S)	2	1
LUT Delay ($t_{cp}^{\#}$)	23	17
Ideal Number of Pipeline Stages (L)	4	4

Table 6. LUT requirement for different Primitives in $GF(2^{163})$

Primitives	No. of Instances	LUTs in Virtex 4		LUTs in Virtex 5	
		per unit	total	per unit	total
Adder	1	163	163	163	163
Squarer	1	163	163	163	163
Adder merged with Squarer[1]	1	163	163	163	163
Quad Circuit[2]	4	315	1260	249	996
Mux 2:1[3]	5, 4	163	815	163	652
Mux 4:1	8	326	2608	163	1304
Multiplier	1	9092	9092	6313	6313
Total	-	-	14264	-	9754

1. This is present before Mux B
2. Two of these are present in the Powerblock
3. On Virtex 4, Mux F is 2 : 1. On Virtex 5, this is not required as there is single 2^4 circuit

Appendix B

In this appendix we present more details about the implementation. In order to understand how the FPGA's LUTs have been utilized, we have synthesized each

module individually. Table 6 gives the details according to Figure 8. It may be noted that these results may not exactly match the results in Table 4 because (1) they have been synthesized individually (2) and it does not have the top module which contains the control unit.

For the Virtex 5 FPGA, the Powerblock should ideally have a single 2^4 circuit as seen in Table 5. This we have implemented using a cascade of two quad circuits.

The critical path for the design (both in Virtex 4 and Virtex 5) obtained from the Xilinx tool, was through Mux H (in the register bank), the quad circuit, and then the Mux B (refer Figure 8). This path is present in the first stage of the pipeline and corresponds to the maximum operating clock frequency specified in Table 4.