Efficient FPGA-based ECDSA Verification Engine for Permissioned Blockchains

Rashmi Agrawal
Boston University, Boston, USA
rashmi23@bu.edu

Ji Yang
AMD, San Jose, USA
ji.yang@amd.com

Haris Javaid
AMD, Singapore
haris.javaid@amd.com

***Abstract*—Permissioned blockchain platforms heavily depend on cryptography to provide a layer of trust within the blockchain network, thus verification of cryptographic signatures often becomes the bottleneck. ECDSA is the most commonly used cryptographic scheme in permissioned blockchains. In this work, we propose an efficient implementation of ECDSA signature verification on FPGA, in order to improve performance of permissioned blockchains that aim to use FPGA-based hardware accelerators. We propose several optimizations for modular arithmetic (e.g., custom multipliers and fast modular reduction) and point arithmetic (e.g., reduced number of point double and addition operations, and optimal width NAF representation). Based on these optimized modular and point arithmetic modules, we propose an ECDSA verification engine that can be used by any application for fast verification of signatures. We further optimize our ECDSA verification engine for Hyperledger Fabric (one of the most widely used blockchain platforms) by moving carefully selected operations to a precomputation block, thus simplifying the critical path of ECDSA signature verification. Our ECDSA engine running at 250MHz on Xilinx Alveo U250 accelerator card can perform a verification in $760\mu s$ with a throughput of $1,315$ verifications/sec, which is ~ 2.5 times faster than state-of-the-art FPGA-based implementations. Our Hyperledger Fabric-specific ECDSA engine can perform a verification in $368\mu s$, and $2,717$ verifications/sec. With 10 engines, Hyperledger Fabric can achieve a throughput of $7,520$ transactions/sec.**

***Index Terms*—ECDSA signature verification, FPGA, Hyperledger Fabric, Blockchain**

I. Introduction

Beyond the hype, blockchain technology is emerging as one of the most disruptive technologies, with real-world use cases in many domains from digital identity management to financial services, supply chains, and product provenance. A blockchain combines consensus mechanisms with cryptography to provide a layer of trust for executing and recording transactions in an immutable ledger within a network of mutually untrusting nodes. In public blockchains, such as Bitcoin and Ethereum, any node can participate in the network without a specific identity and proof-of-work based consensus is used, which becomes the bottleneck (due to computation of massive amounts of hashes). In permissioned blockchains, on the other hand, only nodes with known identities are part of and interact with the network, while the consensus is delegated to only a few nodes. Consequently, cryptographic operations (to authenticate nodes and validate transactions) become the bottleneck rather than the consensus mechanism [1].

Hyperledger (HL) Fabric [2] is an open-source, enterprise-grade implementation of a permissioned blockchain, thus is one of the most widely used blockchain with many real-world applications from finance and supply chain domains [3]. In HL Fabric network, one of the nodes is a validator peer, which validates a block and its transactions before committing that block to the ledger. Many recent works [1], [4]–[6] have shown that verification of cryptographic signatures (Fabric uses 256-bit ECDSA scheme) inside a validator peer is the major bottleneck and critically affects the peak throughput. Recently, hardware acceleration was proposed for validation of blocks in HL Fabric. The work in [7] proposed a CPU-FPGA based system where a multi-core server with a network-attached FPGA card (connected to the CPU via PCIe) is used to accelerate validator peer. Although they demonstrated an order of magnitude speedup compared to CPU-only implementation, ECDSA signature verification still turned out to be the critical path in FPGA accelerator (latency of a single verification is much larger than other operations, thus critically affecting throughput in signature verifications/sec or blockchain transactions/sec). Therefore, in this paper, we focus on an efficient FPGA-based implementation of ECDSA signature verification, in order to improve performance of permissioned blockchains that aim to use FPGA-based accelerators.

More specifically, we focus on accelerating ECDSA signature verification over NIST P-256 elliptic curve, which requires performing 256-bit modular and point arithmetic operations. This is challenging for FPGAs because 256-bit wide multipliers, adders/subtractors, and dividers are not readily available. A naive implementation will result in resource intensive design, making it challenging to fit the entire accelerator on an FPGA and meet the required timing constraints. Our contributions are:

- **FPGA-specific optimizations:** We propose a custom 256-bit multiplier for use in modular multiplication and a 258-bit multiplier for Barrett reduction by efficiently leveraging DSP blocks. We also propose an efficient algorithm to perform fast modular reduction over P-256 without using expensive 256-bit comparators.
- **Algorithmic optimizations:** We present optimizations for simultaneous-point and fixed-point multiplication algorithms (used in ECDSA verification) to reduce the overall number of point double and addition operations (e.g., use of projective Chudnovsky coordinates with optimal non-adjacent form (NAF) width).
- **HL Fabric-specific ECDSA verification:** We present a fast ECDSA verification engine by leveraging the fact that the generator point $\mathcal{G}$ is fixed and the public key $\mathcal{K}$ can be extracted in advance. This allows us to move a major chunk

of point arithmetic operations to a precompute block. Consequently, the operations during actual ECDSA computation reduce to just point addition operations, resulting in a much faster signature verification.

We implemented ECDSA verification engines on Xilinx Alveo U250 card [8] with 250MHz target frequency, achieving $368\mu s$ per verification and $2,717$ verifications/sec.

II. Background and Preliminaries

A. Blockchain Machine

Figure 1(a) depicts a simplified overview of hardware accelerator proposed in [7] for HL Fabric. The blocks are received in FPGA card through the integrated network interface. The first module, ProtocolProcessor, processes the incoming packets and extracts relevant data, such as block id, transaction ids, ECDSA signatures, public keys, etc. The second module, BlockProcessor, uses this data to validate the block and its transactions, and then commits valid transactions. The Fabric software running on CPU accesses validation results from hardware and commits the block to disk-based ledger.

Internally, the BlockProcessor uses a configurable number of ECDSA verification engines distributed across multiple stages to process signature verifications as fast as possible. Each ECDSA engine accepts a verification request in the form of {signature, public_key, data_hash}. Typically, the hash is computed during ECDSA verification, but in Blockchain Machine, it is precomputed by the ProtocolProcessor for better performance. In this paper, we design an ECDSA verification engine that can be used inside the Blockchain Machine, or similar hardware accelerators for permissioned blockchains.

B. ECC

An elliptic curve $\mathcal{E}$ over a prime field $\mathbb{F}_p$ is defined by a pair of tuple (x, y) satisfying the Weierstrass equation $y^2 = x^3+ax+b$ where a and b belong to a Galois field $GF(p)$ with $p > 3$ and $\mathbb{F}_p = GF(p)$. Point arithmetic allows us to compute any point $\mathcal{P}_i = (x_i, y_i)$ on the elliptic curve. Addition of two points $\mathcal{P}$ and $\mathcal{Q}$, where $\mathcal{P} \neq \mathcal{Q}$, is defined by $\mathcal{R} = \mathcal{P} + \mathcal{Q}$. However, when $\mathcal{P} = \mathcal{Q}$, point addition is performed as a point double operation, resulting in $2\mathcal{P}$. Although point addition and double operations can be performed in affine or projective coordinates, we use projective Chudnovsky coordinates (using equations 5 and 6 from [9]), where each point $\mathcal{P}$ is represented as a quintuple (X, Y, Z, Z^2, Z^3) corresponding to the affine point $(x = X/Z^2, y = Y/Z^3)$. These coordinates offer a speed benefit over affine coordinates when the cost for

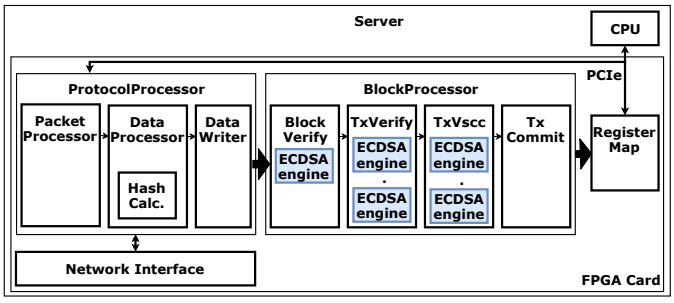

Fig. 1: Blockchain Machine: FPGA-based hardware accelerator.

Algorithm 1: ECDSA Verification

Input : Digest z, sig (r, s), and key $\mathcal{K} = (x_\mathcal{K}, y_\mathcal{K})$
Output: Valid or invalid

1 **if** (r, s) *not in range* $[1, n-1]$ **then return** Invalid;
2 *Compute* $w = s^{-1} \mod n$;
3 *Compute* $k_1 = z * w \mod n$;
4 *Compute* $k_2 = r * w \mod n$;
5 *Compute* $(x_2, y_2) = k_1 G + k_2 \mathcal{K}$;
6 **if** $r = x_2 \mod n$ **then return** Valid;

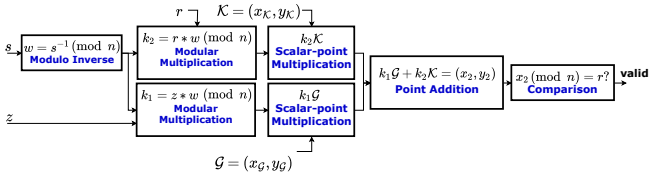

Fig. 2: Operations in ECDSA signature verification.

modulo inversion is significantly higher than the modular multiplication, which is the case with FPGAs because DSP blocks can perform fast multiplications.

C. ECDSA Verification Algorithm

A sender sends the message digest/hash $z = H(m)$, the signature (r, s), and the public key $\mathcal{K} = (x_\mathcal{K}, y_\mathcal{K})$, which are verified by the receiver. It is assumed that the receiver knows the ECC domain parameters (a, $\mathcal{G} = (x_\mathcal{G}, y_\mathcal{G})$, n, and p given by NIST [10] for P-256 elliptic curve). Typically, message digest is computed during verification, however, we assume that it is precomputed (to be aligned with Blockchain Machine accelerator) and hence is provided as the input. Algorithm 1 describes the verification process, where the output indicates whether the input signature is valid or not. The algorithm performs modular reductions with respect to two different primes: prime p for point arithmetic (line 5), and order of the curve n for the rest of the operations (lines 2-4 and 6). Figure 2 illustrates a pictorial representation of the verification algorithm, where it is evident that we need to implement several modular arithmetic operations such as modulo inverse, multiplication and reduction followed by various point arithmetic operations such as addition, double and scalar-point multiplication.

D. Related Work

Many previous works have proposed acceleration of modular multiplication, point arithmetic for ECC, and ECDSA signature verification in hardware (for ASICs, FPGAs and microcontrollers), however, we report only the FPGA works [11]–[17] here. The work in [17] focused on elliptic curves over binary fields $GF(2^m)$ for simpler hardware implementation, while other works [14], [18], [19] targeted non-256-bits prime field. All these works are not of interest to blockchain platforms because they typically use 256-bit prime field.

The works in [11], [12], [16] focused on accelerating only point arithmetic for ECC over NIST P-256 elliptic curve, and did not implement the entire ECDSA verification. Tachibana et al. [15] accelerated ECDSA verification over Secp256k1 elliptic curve for Bitcoin on an Intel Cyclone IV FPGA, achieving

145.52 ms per verification. Glas et al. [13] implemented an ECDSA verification core for 256-bit prime field on Xilinx Virtex-5 FPGA board. We achieve better performance than these works, and will present a comparison in Section V.

III. FPGA-based ECDSA Verification Engine

In this section, we present the architecture of our efficient ECDSA verification engine. Since ECDSA signature verification algorithm deals with the information that is publicly known, there is no secret information to leak through side-channels while performing verification. This widens our choice of algorithms for implementing various modules within the ECDSA verification engine. Therefore, in our hardware implementation, we select algorithms that utilize minimal hardware resources while resulting in low latency.

A. Modular Arithmetic

To efficiently implement 256-bit wide modular arithmetic on an FPGA, we implement all modular arithmetic modules using multi-word integer arithmetic [20]. In multi-word arithmetic, a 256-bit field element a can be represented as

$$a = 2^{(t-1)W}A[t-1] + \cdots + 2^{2W}A[2] + 2^{W}A[1] + A[0] \quad (1)$$

where W and $t = 256/W$ define the word length and the number of words to operate on respectively. We set the value of W diligently for every operation to efficiently utilize hardwired DSP blocks of the FPGA, and hence achieve a higher maximum operating frequency.

Modular Subtraction: Subtraction in $\mathbb{F}_p$ can be performed using Algorithm-2.8 from [20] with the multi-word integer approach. The adder/subtractor in DSP blocks have 48-bit wide inputs. However, for the multi-word integer approach, the maximum bit width we can use is 32 bits (32 being the largest integer dividing 256 symmetrically). Therefore, to use DSP for subtraction, we set the parameter W as 32 and thus, we have $t = 8$ words to operate on. Along with each subtraction operation, we subtract the previous carry bit and store the next carry bit. As we operate on a single word at a time, we utilize only one DSP block to implement the modular subtraction operation. We perform modular addition using the same module with $2'$s complement input for the second operand as point arithmetic (including point double and add) in projective Chudnovsky coordinate system requires only one modular addition operation. Implementing a separate adder leads to inefficient resource utilization as the adder will remain idle for most of the time.

Integer Multiplication Module: We adopt a hybrid approach to integer multiplication through a combined schoolbook [21] and multi-word Karatsuba [20] approach. Our target FPGA board has DSP blocks with 27x18 bit wide multipliers. However, both multi-word arithmetic and schoolbook algorithm require operands to be split symmetrically (i.e., both operands must have the same base). Therefore, we can use at most a 16x16 bit wide multiplier to multiply two 256-bit operands (16 being the largest number that can split 256 symmetrically). If we set W as 16, we will have $t = 16$ words to operate on, thus implementing just the schoolbook multiplication algorithm [21] will lead to a high latency (at least 256 clock cycles), which is not acceptable as many multiplications are performed in point arithmetic. Therefore, we use multi-word approach with schoolbook multiplication to lower multiplication latency, as shown in Algorithm 2.

Algorithm 2: Hybrid Integer Multiplication

Input : Integers $a, b \in [0, p-1]$, $t = 8$, $l = 16$
Output: $c = (a.b)$

1 *Set (A[t-1],…,A[0])* $\leftarrow$ *a, (B[t-1],…,B[0])* $\leftarrow$ *b*;
2 *Set C[i]* $\leftarrow 0$ *for* $0 \le i \le 2t-1$;
3 **for** *i from* 0 *to* $t-1$ **do**
4 *U* $\leftarrow$ *0*;
5 **for** *j from* 0 *to* $t-1$ **do**
6 $(a_1, a_0) \leftarrow$ *A[i]*, $(b_1, b_0) \leftarrow$ *B[i]*;
7 $ab = a_1b_1 2^{2l} + [(a_0 + a_1)(b_0 + b_1) - a_1b_1 - a_0b_0]2^l + a_0b_0$;
8 *(U,V)* $\leftarrow$ *C[i+j] + ab + U*;
9 *C[i+j]* $\leftarrow$ *V*;
10 **end**
11 *C[i+t]* $\leftarrow$ *U*;
12 **end**

We first take the schoolbook multiplication algorithm and set the parameter W as 32 (twice the input width of a multiplier in DSP block) to split 256-bit operands into $t = 8$ 32-bit words. Next, we target 32-bit operands and split them into 16-bit operands and multiply these 16-bit operands using Karatsuba equation (line 7) where l is 16. Moreover, as multiplication operations in line 7 are independent, we unroll the loops in the algorithm to perform multiplications and accumulations in parallel. As a result, we can perform a 256-bit integer multiplication in just 39 clock cycles using Algorithm 2 while efficiently utilizing multipliers in DSP blocks. Note that various existing works [22], [23] (implementing different cryptographic schemes) exploit Karatsuba algorithm to perform modular multiplication on an FPGA, however, novelty of our work lies in the way we combine schoolbook multiplication and Karatsuba algorithm to implement a low-latency, parallel multiplier.

Fast P-256 Modular Reduction Module: The output of an integer multiplication yields a 512 bits result that needs to be reduced to 256 bits by performing modular reduction. In point arithmetic, modular reduction is performed using the prime $p = P\text{-}256$ while other modular reductions are performed using order of the curve n (which is explained later).

NIST recommends a fast modulo reduction P-256 algorithm [20] as P-256 is a general Mersenne prime. This algorithm replaces large division operations with simple additions and subtractions by exploiting the structure of the Mersenne prime. Therefore, by using this algorithm, we can perform a 256-bit modular reduction using two left shifts (multiplication by 2), four additions, and four subtractions. However, the result generated from this algorithm can be in the range $-4p$ to $5p$ instead of 0 to p. So, we need to perform a correction by either adding to or subtracting from the result a suitable value of p within this range. This correction step,

Algorithm 3: Fast Reduction Modulo P-256

Input : 512-bit $c = (c_{15}, \ldots, c_2, c_1, c_0)$ in base 2^{32}
Output: $r = c \mod P\text{-}256$

1 $s[0] = (c_7, c_6, c_5, c_4, c_3, c_2, c_1, c_0)$;
2 $s[1] = (c_{15}, c_{14}, c_{13}, c_{12}, c_{11}, 0, 0, 0) << 1$;
3 $s[2] = (0, c_{15}, c_{14}, c_{13}, c_{12}, 0, 0, 0) << 1$;
4 $s[3] = (c_{15}, c_{14}, 0, 0, 0, c_{10}, c_9, c_8)$;
5 $s[4] = (c_8, c_{13}, c_{15}, c_{14}, c_{13}, c_{11}, c_{10}, c_9)$;
6 $s[5] = (c_{10}, c_8, 0, 0, 0, c_{13}, c_{12}, c_{11})$;
7 $s[6] = (c_{11}, c_9, 0, 0, c_{15}, c_{14}, c_{13}, c_{12})$;
8 $s[7] = (c_{12}, 0, c_{10}, c_9, c_8, c_{15}, c_{14}, c_{13})$;
9 $s[8] = (c_{13}, 0, c_{11}, c_{10}, c_9, 0, c_{15}, c_{14})$;
10 **for** *i from* 0 *to* 4 **do**
11 $\quad r = r + s[i]$, $r = (r \geq p)\ ?\ r - p : r$
12 **end**
13 **for** *i from* 5 *to* 8 **do**
14 $\quad r = r - s[i]$, $r = (r < 0)\ ?\ r + p : r$
15 **end**

Algorithm 4: Efficient Comparison with P-256

Input : 257-bit r with $0 \leq r < 2p$, C_0, C_1, C_2, C_3
Output: $r \geq p$

1 $r_0 = \&r[95:0]$, $r_1 = r[191:96]$, $r_2 = r[223:192]$, $r_3 = \&r[255:224]$, $r_4 = r[256]$;
2 **if** *($r_4 = 1$ or ($r_3 = C_3$ and $r_2 \geq C_2$ and $r_1 > C_1$))* **then return** greater;
3 **else if** *($r_4 = 0$ and $r_3 = C_3$ and $r_2 = C_2$ and $r_1 = C_1$ and $r_0 = C_0$)* **then return** equal;

however, requires performing many 256-bit comparisons to figure out the exact range in which the result lies. On FPGAs, a 256-bit comparator leads to long carry chains adversely impacting the timing constraints of the design. Therefore, to avoid performing many of such wide comparisons at once, we check and correct the result immediately after each step of the computation. The steps of our proposed fast modulo reduction P-256 algorithm are shown in Algorithm 3. As we perform addition operations in lines 10-12, we perform an immediate correction by comparing the result with p. Similarly, after subtraction operations in lines 13-15, we compare the result with 0 to see if it is negative and correct it accordingly.

From Algorithm 3, it is evident that we avoid using many 256-bit wide comparators in parallel, but we still need a 256-bit comparator to check if the intermediate result is $\geq p$ or not. We further exploit the structure of the Mersenne prime and propose Algorithm 4 to perform this comparison efficiently without actual 256-bit comparators. Our algorithm is based on the observation that P-256 can be split into four parts as follows:

$P_0 = P[95:0] = fffffffffffffffffffffffff$
$P_1 = P[191:96] = 0$, $P_2 = P[223:192] = 1$
$P_3 = P[255:224] = ffffffff$

Using these four parts, we can generate the following four conditions: $C_0 : \&P_0 = 1$, $C_1 : P_1 = 0$, $C_2 : P_2 = 1$, $C_3 : \&P_3 = 1$, where $\&$ means an AND reduction. The algorithm

Algorithm 5: Hardware-friendly Barrett Reduction

Input : n, $b = 4$, $k = \lfloor \log_b n \rfloor + 1$, z, $\mu = \lfloor \frac{b^{2k}}{n} \rfloor$
Output: $r = z \mod n$

1 $\hat{q} \leftarrow (z >> 2(k-1)).(\mu >> 2(k+1))$;
2 $r \leftarrow z\ \&\ (b^{k+1} - 1) - \hat{q}.n\ \&\ (b^{k+1} - 1)$;
3 **if** *($r < 0$)* **then** $r \leftarrow r + b^{k+1}$;
4 **while** *($r \geq n$)* **do** $r \leftarrow r - n$;

starts by splitting the input integer r in a similar fashion as P-256 and an additional r_4 for the $257th$ bit (as the input integer is of 257-bits). First, we check if r_4 is 1, then r is definitely $> p$. However, if r_4 is 0, then we need to evaluate other conditions. If $r_2 \geq 1$ or $r_1 > 0$ (line 2), then $r > p$. Otherwise, if all four conditions on line 3 are satisfied, then $r = p$. This algorithm converts 256-bit wide comparisons to four 1-bit comparisons and can be efficiently implemented in hardware using bit-slicing and unary $\&$ operator.

Modular Reduction over n: We propose to use the standard Barrett reduction algorithm [20] for modulo reduction over the prime n. Barrett reduction does not exploit the structure of n but computes $r = z \mod n$, by computing a value $\hat{q}$, which when multiplied with n and subtracted from z will give the desired modular reduction value r. The algorithm requires selecting a base b, which when chosen as a power of two gives an efficient implementation in hardware. We select b as 4 and we precompute the parameters of Barrett reduction (k and μ) as they are fixed and do not change at any point in computation. Our modified hardware-friendly version of the Barrett reduction is shown in Algorithm 5. We perform divisions using right shift operation (line 1) and modular reductions using AND operation (line 2). This optimization is possible because b is a power-of-2 and for powers-of-2, modular reduction can be efficiently done by masking the lower-order bits using AND operation. Thus, we avoid all wide division operations. However, we need to perform two 258-bit multiplications as the parameter μ is a 258-bit integer.

For 258-bit multiplication, we again use the hybrid approach proposed in Algorithm 2. However, we set $W = 6$ and split the input operands into 43-bit words ($258/6 = 43$). Further, instead of a single Karatsuba split as in Algorithm 2, we perform a two-level Karatsuba split to efficiently leverage the DSP blocks. At level-1, we split 43-bit operands into 32- and 11-bit integers. Then at level-2, we split 32-bit integers into 16-bit integers (see Figure 3). Thus, we perform only 16×16, 16×11 and 11×11-bit multiplications instead of a 43-bit multiplication.

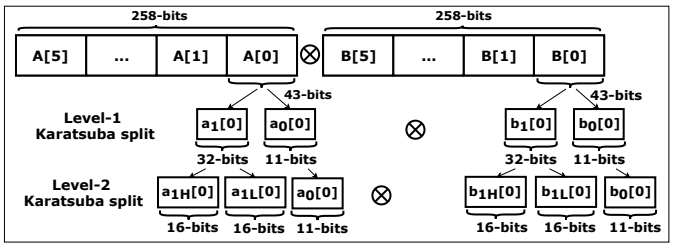

Fig. 3: Proposed 258-bit multiplication approach.

Modulo Inverse Module: NIST recommends using Extended Euclidean algorithm [10] to compute modulo inverse in ECDSA verification algorithm. However, extended Euclidean algorithm is expensive to implement in hardware requiring 256-bit division and multiplication to compute quotient and remainder respectively. We choose to implement an optimized, faster modulo inverse algorithm proposed by Chen and Qin [24]. This algorithm is suitable for hardware implementation as it has very low resource footprint and also incurs a low latency. The algorithm computes a modulo inverse using only right shift and addition operations and at any given time only two 256-bit adders are operating in parallel. We observe that the latency of this algorithm ranges from 35-600 clock cycles depending on the input, however for real test cases, the latency averages close to 550 clock cycles across multiple evaluations. It is worth noting that we modified the actual algorithm to take modulus as an input because we leverage the same algorithm to perform modulo inverse with respect to n (line 2 of Algorithm 1) as well as to convert projective Chudnovsky coordinates back to affine coordinates wherein we need to compute modulo inverse with respect to p.

B. Point Arithmetic

During ECDSA signature verification, we need to perform two scalar-point multiplications and one point addition operation (line 5 in Algorithm 1). We first present our generic approach to point arithmetic that can be leveraged in any application requiring fast ECDSA signature verification. We leverage the simultaneous-point multiplication (SPM) Algorithm 6, also known as "Shamir's trick", to operate on both the generator point ($\mathcal{P}$) and public key coordinates ($\mathcal{Q}$) at the same time. In addition, this algorithm eliminates the need to perform the point addition separately. We conducted an analysis on how the number of operations varies when different point representations such as binary, NAF, joint-sparse form (JSF), and width-w NAF are used in Algorithm 6. We observe that point double operations largely remain the same while point addition operations can be reduced to as low as 112 when width-w NAF is used. Note that width-w NAF conversion can be done in hardware using Algorithm-3.35 from [20], and is trivial in comparison to point arithmetic. We use $w = 4$ to keep the storage requirements minimal.

Algorithm 6: Width-w NAF method for SPM

Input : Width w, k_1 and k_2, points $\mathcal{P}$ and $\mathcal{Q}$
Output: $\mathcal{A} = k_1\mathcal{P} + k_2\mathcal{Q}$
1 *Compute:* $i\mathcal{P}$ *and* $i\mathcal{Q}$ *for* $i \in \{1, 3, \ldots, 2^w - 1\}$;
2 *Compute* $NAF_w(k_1)$ *and* $NAF_w(k_2)$;
3 $l = max\{\ell_1, \ell_2\}$ *where* ℓ_1 *and* ℓ_2 *are lengths of* $NAF_w(k_1)$ *and* $NAF_w(k_2)$, $\mathcal{A} = \infty$;
4 **for** i *from* $l - 1$ *down to* 0 **do**
5 $\quad \mathcal{A} = 2\mathcal{A}$;
6 $\quad$ **if** $k_1[i] > 0$ **then** $\mathcal{A} = \mathcal{A} + k_1[i]\mathcal{P}$;
7 $\quad$ **else** $\mathcal{A} = \mathcal{A} - k_1[i]\mathcal{P}$;
8 $\quad$ **if** $k_2[i] > 0$ **then** $\mathcal{A} = \mathcal{A} + k_2[i]\mathcal{Q}$;
9 $\quad$ **else** $\mathcal{A} = \mathcal{A} - k_2[i]\mathcal{Q}$;
10 **end**

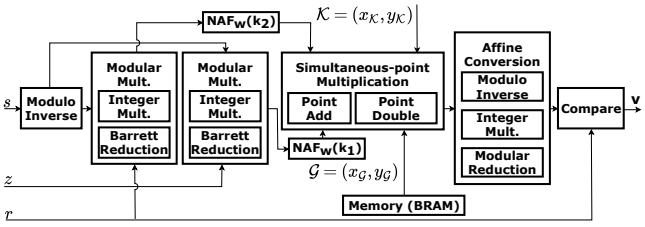

Fig. 4: Data flow in generic ECDSA verification engine.

In Algorithm 6, point double operation (line 5) and point additions (lines 6-9) cannot be done in parallel as they depend on each other. However, we reduce the number of these operations by computing various values of $\mathcal{G}$ $(3\mathcal{G}, 5\mathcal{G}, 7\mathcal{G}, 9\mathcal{G}, 11\mathcal{G}, 13\mathcal{G}, 15\mathcal{G})$ offline and storing them in BRAM because $\mathcal{G}$ is known in advance. This requires 1120 bytes of storage space but reduces the computation on line 1 to only $\mathcal{K}$ $(3\mathcal{K}, 5\mathcal{K}, 7\mathcal{K}, 9\mathcal{K}, 11\mathcal{K}, 13\mathcal{K}, 15\mathcal{K})$ in hardware. For these values, $3\mathcal{K}$ is computed by performing a point double on $\mathcal{K}$ followed by a point addition. We store the $2\mathcal{K}$ value temporarily and reuse it; for example, $5\mathcal{K}$ is computed by performing a point addition between $3\mathcal{K}$ and $2\mathcal{K}$. Therefore, we only need one point double and seven point addition operations in hardware to compute all the required values. Overall, we significantly reduce the number of point double and addition operations with the use of width-w NAF, and offline and optimized computation for $\mathcal{G}$ and $\mathcal{K}$.

C. ECDSA Verification Engine

Putting it all together, Figure 4 depicts the data flow in our ECDSA verification engine using SPM algorithm. Although the figure shows multiple modular arithmetic modules, we instantiate only a single instance of these modules and schedule different operations in parallel to efficiently utilize the hardware resources. For example, we perform the $NAF_w(k_2)$ conversion in parallel with the second modular multiplication. Similarly, $NAF_w(k_1)$ conversion happens in parallel to the computation on line 1 in Algorithm 6. Thus, no additional clock cycles are spent in NAF conversions. From amongst all the modules, integer multiplication and modular reduction are the heavily utilized modules. It is worthwhile to reiterate that an additional point add operation is not required here at it is absorbed in the SPM operation. However, after the multiple-point multiplication operation, we need an additional operation to convert x from projective Chudnovsky coordinate back to affine coordinate for final comparison.

IV. ECDSA Verification Engine for HL Fabric

In this section, we propose optimizations in the context of permissioned blockchains specifically HL Fabric. We exploit the fact that some parameters are fixed apriori while other parameters are available in advance, and hence both of these can be preprocessed to speedup ECDSA verification operation. More specifically, we take advantage of the fact that the generator point $\mathcal{G}$ is fixed and the public key coordinates

Algorithm 7: Fixed-base NAF windowing for FPM

Input : Window width w, $d = 256/w$, k, point $\mathcal{P}$
Output: $\mathcal{A} = k\mathcal{P}$

1 *Precompute:* $\mathcal{P}_i = 2^{wi}\mathcal{P}$, $0 \leq i \leq d$;
2 *NAF(k),*$I = (2^{w+1} - 2)/3$, $\mathcal{A} = \infty$, $\mathcal{B} = \infty$;
3 **for** j *from* I *down to* 1 **do**
4 | *For each* i *for which* $k_i = \pm j$ *do* $\mathcal{B} = \mathcal{B} \pm \mathcal{P}_i$;
5 | $\mathcal{A} = \mathcal{A} + \mathcal{B}$;
6 **end**

$\mathcal{K} = (x_\mathcal{K}, y_\mathcal{K})$ are known well in advance before the ECDSA verification starts. The ProtocolProcessor in Blockchain Machine (see Figure 1) processes the incoming data and extracts the public key and ECDSA signature information. Therefore, as soon as the public key coordinates are available, we can start processing them. With this goal in mind, we leverage the fixed-point multiplication (FPM) algorithm (see Algorithm 7) to perform point arithmetic instead of simultaneous-point multiplication algorithm (see Algorithm 6).

Algorithm 7 starts by precomputing various powers-of-2 point multiplications for a point $\mathcal{P}$ which is known apriori (for example, when $w = 4$, then precomputations will be $\mathcal{P}, 16\mathcal{P}, 256\mathcal{P}$, and so on). As the generator point is fixed, we precompute these values offline and store the values in BRAM on FPGA. For the public key coordinates, we design a separate precompute block, outside of the ECDSA verification engine, which runs binary scalar-point multiplication algorithm (Algorithm-3.27 in [20]) with point double operations only to precompute the values mentioned earlier (i.e., $\mathcal{P}$,$16\mathcal{P}$, etc). Point addition operations are not required as we are computing power-of-2 point multiplications only, which can be computed using successive point double operations. We further optimize the precompute block by reducing the number of point double operations. For example, if we want to compute $256\mathcal{P}$, we need not start all the way from $\mathcal{P}$. Instead, we can use the value of $16\mathcal{P}$ that was computed in the previous step, thus reducing the number of point double operations from wi to w in each step i where $0 \leq i \leq d$. This optimization helps reduce the number of point double operations from over 8000 to only 252 when $w = 4$ and $d = 64$.

The computation within the actual ECDSA verification operation reduces to lines 3-6 of Algorithm 7. Consequently, ECDSA verification comprises of just point addition operations with the point double operations moved to precomputation. We use the fixed-base (with base $w = 4$) NAF windowing method to reduce the number of point addition operations. Note that Algorithm 7 is executed twice; once for the generator point $\mathcal{G}$ accessing offline computed point values in line 1, and the second time for public key $\mathcal{K}$ accessing precomputed point values in line 1.

Figures 5(a) shows the architecture of our HL Fabric-specific ECDSA verification engine. The precompute block is placed inside the ProtocolProcessor to store all the required point values in BRAM. With projective Chudnovsky coordinates, we need about $20KB$ of memory to store all the precomputed point $\mathcal{G}$ and $\mathcal{K}$ values. In a permissioned blockchain like HL Fabric, the number of nodes are limited and hence the number of unique identities (public keys) is limited and those identities are known apriori. Moreover, there may only be tens of unique identities, thus storage of precomputed points will not incur a high memory overhead. The precompute block has its own modular arithmetic modules and FSM controlling movement of data between these modules. The ECDSA verification engine instantiates its own modular arithmetic modules, again only one module per operation to keep the resource utilization low. This is especially beneficial for Blockchain Machine where many ECDSA verification engines are desirable inside the BlockProcessor module.

Figure 5(b) depicts the data flow in our ECDSA verification engine with the precompute block. The precomputed points are read from BRAM by the ECDSA verification engine. The read from BRAM is not a bottleneck as read will happen once in a while and then hundreds of clock cycles are spent on processing the data. Since the number of unique public keys is limited, precompute block is executed only when a new public key is encountered. Hence, the precompute block does not become a bottleneck. Both in precompute and ECDSA verification engine, with single instantiation of the modular arithmetic modules, some operations are performed serially while different operations are efficiently scheduled in parallel. For example, the NAF conversion of k_1 and k_2 happens in parallel with the first FPM and the second modular multiplication involving k_1 respectively. After FPM, a point addition operation is required (line 5 of Algorithm 1) and conversion back to affine coordinates for comparison.

V. Evaluation

We designed our ECDSA verification engines in Verilog 2001 and synthesized them using Xilinx Vivado 2019.2. For

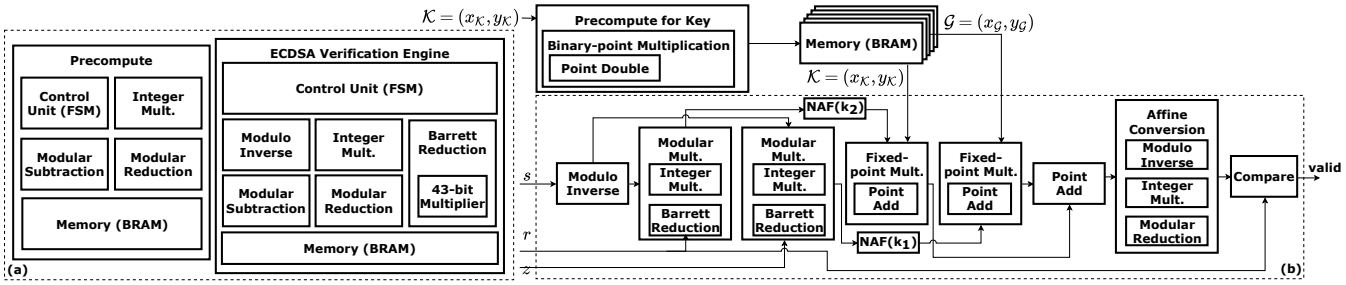

Fig. 5: ECDSA verification engine with Precompute block: (a) Architecture (b) Data flow.

functional verification, we generated test cases using open-source code from OpenSSL library [25] as well as from actual data (public key, signature, and hash from Hyperledger Fabric). We also successfully verified the test vectors [26] from NIST for P-256 ECDSA signature verification. We targeted Xilinx Alveo U250 card because blockchains are typically deployed on a cloud server with an FPGA accelerator card. Since our goal is to integrate the ECDSA verification engine into the blockchain hardware accelerator from [7], which is quite complex and operates at 250 MHz, we limit the operating frequency of our designs to 250 MHz as well even when it is possible to obtain higher frequencies with DSP blocks like in [11]. This restriction enables better scalability within the blockchain accelerator by instantiating multiple ECDSA engines for distributed computation.

Throughout this section, we report frequency (freq.) in MHz, latency in clock cycles, and throughput (TP) in operations per second. When comparing our work with existing state-of-the-art implementations, we cautiously compare the clock cycles of different designs instead of absolute runtimes to overcome the inherent improvements from upgraded technologies and operating frequencies, and hence provide a fair comparison. In an ideal situation, the existing works should have been implemented on the Alveo U250 FPGA board as well. However, those designs are not open-source and implementation of each requires significant effort. Furthermore, we do not present direct comparison results with ASIC/CPU/GPU implementations because our goal is not to compete with ASIC/CPU/GPU implementations but to provide the best FPGA implementation that can be used in accelerators for permissioned blockchains (since they are naturally suitable for FPGA based acceleration [7]).

A. Modular Arithmetic

We first discuss the area footprint and latency of the individual modular arithmetic modules as listed in Table I. With all the proposed optimizations, our modular arithmetic modules incur low resource utilization. The integer multiplication module consumes the most LUTs, which can be reduced using BRAM-based optimizations. We, however, leave this optimization for future work. Most modules perform fast computations except for Barrett reduction which has the highest latency because of two serial 258-bit multiplications. We made this design choice to keep the resource utilization low. Moreover, modular reduction using Barrett reduction is performed only in lines 3, 4, and 6 of Algorithm 1, which is not the critical path (line 5 is the critical path in ECDSA verification). Note that our modular subtraction module is $1.8\times$ faster and our modular multiplication (integer multiplication + P-256 modular reduction) is $1.2\times$ faster than the implementations in state-of-the-art work [11].

B. Point Arithmetic

We evaluate the performance of our point addition (PA) and point double (PD) operations. For a fair comparison, we also report the latencies from state-of-the-art FPGA-based work [11], which uses projective Chudnovsky coordinates like our implementation. We observe that clock cycles of our point add and double operations are $\sim 1.6\times$ lower than their corresponding operations (see first four rows of Table II). Note that their design involves a dual clock which is much more complicated to implement than our design that uses only a single clock. Moreover, the authors in [11] focused on optimizing only the point arithmetic, and did not implement the entire ECDSA verification algorithm. This makes it much easier for their stand-alone point arithmetic modules to run at a higher frequency.

TABLE I: HARDWARE RESULTS OF MODULAR ARITHMETIC MODULES.

Operation	LUT	FFs	DSP	Latency
Modular subtraction	616	781	1	10
Integer multiplication	5471	7980	128	39
P-256 Modular Reduction	2225	789	0	19
Barrett Reduction	2130	3597	9	1,552
Modulo Inverse	3503	1313	0	550

TABLE II: PERFORMANCE COMPARISON OF POINT ARITHMETIC.

Operation	Platform	Freq.	Latency	TP
PA [Our work]	Alveo U250	250	622	402K
PA [11]	Virtex-4	375	980	382K
PD [Our work]	Alveo U250	250	435	574K
PD [11]	Virtex-4	375	700	535K
PM [Our work]	Alveo U250	250	190,976	1,309
PM [11]	Virtex-4	375	303,450	1,236
PM [12]	Virtex-2 Pro	108.2	451,733	240
PM [16]	Virtex-7	124.2	462,520	268
PM [27]	Virtex-2 Pro	67	567,500	118
PM [28]	Virtex-2	39.5	960,000	41
PM [29]	Virtex-E	39.7	987,500	40
SPM [Our work]	Alveo U250	250	231,406	1,080
SPM [11]	Virtex-4	375	366,905	1,022
SPM-NAF [Our work]	Alveo U250	250	181,024	1,381

Now we compare the latencies of our scalar point multiplication (PM) with state-of-the-art works that implement scalar point multiplication (see Table II). We observe that our PM is $\sim 1.6\times$ to $\sim 5\times$ faster than these existing works in terms of clock cycles. As most of these prior implementations were done using binary double and add algorithm, our PM is also implemented using the same approach. Note that implementation done by Kudithi et al. [16] works with affine coordinates while the rest of the works use projective coordinates. Next, we compare the latencies of our simultaneous-point multiplication (SPM) operation. As mentioned in Section III-B, we perform SPM operation using width-4 NAF approach. However, for a fair comparison with [11], which uses a binary representation, we estimated the latencies using the number of operations performed in Algorithm 6 with binary representation. More importantly, our SPM with width-4 NAF incurs about half the latency owing to our faster modular arithmetic modules.

C. ECDSA Verification Engine

Table III presents the hardware resource utilization and latency of the ECDSA verification engine. Our ECDSA verification engine takes $\sim 190,000$ clock cycles for a single verification leading to a throughput of 1,315 verifications/sec. Although many prior works have accelerated point arithmetic on FPGA, most works did not implement the entire ECDSA

TABLE III: HARDWARE RESULTS OF ECDSA ENGINE.

Design	LUT	FFs	DSP	BRAM	Latency
ECDSA verf.	24394	10961	137	5	190,000

TABLE IV: PERFORMANCE COMPARISON OF ECDSA ENGINE.

Work	Platform	Freq.	Latency	TP
Our work	Alveo U250	250	190,000	1,315
[13]	Virtex-5	50	454,140	110

verification algorithm. We found only one relevant comparable work that accelerated ECDSA signature verification for NIST P-256 on FPGA, which is reported in Table IV. Glas et al. [13] reported implementation results of the complete signature generation and verification unit on a Xilinx XC5VLX110T Virtex-5 FPGA. Their signature verification unit includes a hash generator ip which incurs a latency of 68 clock cycles that we have adjusted accordingly for a fair comparison. Their design achieves a throughput of 110 verifications/sec that is about $12\times$ lower than the throughput of our ECDSA verification engine. We do not compare the hardware resource utilization as this design is implemented on a different FPGA. Moreover, we cannot estimate the hardware cost of this design for Alveo U250 FPGA as it is not open-sourced.

D. ECDSA Verification Engine for HL Fabric

Table V presents the hardware resource utilization and latency of our ECDSA verification engine with precompute block. We observe that the precompute block incurs a latency of $\sim$120,000 clock cycles. Then, the actual ECDSA verification engine requires only $\sim$92,000 clock cycles to perform single signature verification. With this approach, we achieve a throughput of 2,717 signature verifications/sec.

We also evaluate the HL Fabric-specific ECDSA verification engine in the context of Blockchain Machine [7]. Table VI presents the throughput (transactions per second) of Blockchain Machine with both the generic and Hyperledger Fabric-specific ECDSA verification engines. We observe a $2\times$ improvement in throughput with our precompute optimization. We also change the number of ECDSA verification engines from 4-10, and observe that both types of engines scale the throughput in a similar trend ($\sim$1.57$\times$ improvement).

TABLE V: HARDWARE RESULTS OF ECDSA ENGINE (250 MHZ).

Design	LUT	FFs	DSP	BRAM	Latency
Precompute	14088	1417	129	15	120,000
ECDSA verf.	21759	5625	137	15	92,000

TABLE VI: TP OF BLOCKCHAIN MACHINE WITH ECDSA ENGINES.

No. of Engines	Generic ECDSA	HL Fabric ECDSA
4	1,290 tps	2,650 tps
7	2,525 tps	5,200 tps
10	3,650 tps	7,520 tps

VI. CONCLUSION

In this work, we focused on an FPGA-based efficient implementation of ECDSA signature verification, in order to improve performance of permissioned blockchains that aim to use FPGA-based accelerators. We proposed several FPGA-specific algorithmic optimizations for modular and point arithmetic, and combined those to create two different ECDSA verification engines. Our results demonstrate $368\mu s$ per verification and 2,717 verifications/sec at 250 MHz.

REFERENCES

[1] P. Thakkar and S. Nathan, "Scaling hyperledger fabric using pipelined execution and sparse peers," in *CoRR, arXiv:2003.05113*, 2021.

[2] Hyperledger, "Hyperledger Fabric," Online: https://www.hyperledger.org/projects/fabric, 2019.

[3] M. del Castillo, "Forbes Blockchain 50 2021," https://www.forbes.com/sites/michaeldelcastillo/2021/02/02/blockchain-50, 2021.

[4] Gorenflo *et al.*, "FastFabric: Scaling Hyperledger Fabric to 20,000 Transactions per Second," in *IEEE ICBC*, 2019.

[5] Javaid *et al.*, "Optimizing validation phase of hyperledger fabric," in *IEEE 27th International Symposium on MASCOTS*, 2019.

[6] Chung *et al.*, "Performance Tuning and Scaling Enterprise Blockchain Applications," in *CoRR, arXiv:1912.11456*, 2019. [Online]. Available: http://arxiv.org/abs/1912.11456

[7] Javaid *et al.*, "Blockchain Machine: A network-attached hardware accelerator for hyperledger fabric," in *ICDCS*, 2022.

[8] Xilinx, "Xilinx Alveo," Online: https://www.xilinx.com/products/boards-and-kits/alveo.html, 2020.

[9] H. Cohen, A. Miyaji, and T. Ono, "Efficient elliptic curve exponentiation using mixed coordinates," in *Asiacrypt 1998*. Springer, pp. 51–65.

[10] NIST, "FIPS DSS," Online: https://nvlpubs.nist.gov/nistpubs/FIPS, 2013.

[11] T. Güneysu and C. Paar, "Ultra high performance ecc over nist primes on commercial fpgas," in *CHES*. Springer, 2008, pp. 62–78.

[12] Vliegen *et al.*, "A compact fpga-based architecture for elliptic curve cryptography over prime fields," in *IEEE ASAP 2010*, pp. 313–316.

[13] Glas *et al.*, "Prime field ecdsa signature processing for reconfigurable embedded systems," *IJRC*, vol. 2011.

[14] Sghaier *et al.*, "Design and implementation of low area/power elliptic curve digital signature hardware core," *Electronics*, vol. 6, p. 46, 2017.

[15] Tachibana *et al.*, "Fpga implementation of ecdsa for blockchain," in *2019 IEEE ICCE-TW*. IEEE, 2019, pp. 1–2.

[16] T. Kudithi and R. Sakthivel, "High-performance ecc processor architecture design for iot security applications," *The Journal of Supercomputing*, 2019.

[17] Sau *et al.*, "Binary field point multiplication implementation in fpga hardware," in *Intelligent and Cloud Computing*. Springer, 2021.

[18] Z. Liu, H. Seo, J. Großschädl, and H. Kim, "Efficient implementation of nist-compliant elliptic curve cryptography for 8-bit avr-based sensor nodes," *IEEE Transactions on Information Forensics and Security*, vol. 11, no. 7, pp. 1385–1397, 2015.

[19] N. Gura, A. Patel, A. Wander, H. Eberle, and S. C. Shantz, "Comparing elliptic curve cryptography and rsa on 8-bit cpus," in *International workshop on CHES*. Springer, 2004, pp. 119–132.

[20] D. Hankerson, A. J. Menezes, and S. Vanstone, *Guide to elliptic curve cryptography*. Springer Science & Business Media, 2006.

[21] C. Rafferty, M. O'Neill, and N. Hanley, "Evaluation of large integer multiplication methods on hardware," *IEEE Transactions on Computers*, vol. 66, no. 8, pp. 1369–1382, 2017.

[22] Zutter *et al.*, "Acceleration of rsa cryptographic operations using fpga technology," in *20th IEEE International Workshop on DEXA*, 2009.

[23] Xie *et al.*, "Efficient fpga implementation of low-complexity systolic karatsuba multiplier over $gf(2^m)$ based on nist polynomials," *IEEE Transactions on Circuits and Systems I*, vol. 64, pp. 1815–1825, 2017.

[24] C. Chen and Z. Qin, "Fast algorithm and hardware architecture for modular inversion in gf (p)," in *2009 Second International Conference on Intelligent Networks and Intelligent Systems*. IEEE, 2009, pp. 43–45.

[25] T. O. project, "OpenSSL—cryptography and SSL/TLS toolkit," Online: https://www.openssl.org/, 2021.

[26] NIST, "FIPS DSA TEST VECTORS," Online: https://csrc.nist.gov/projects/cryptographic-algorithm-validation-program/digital-signatures.

[27] N. Mentens, "Secure and efficient coprocessor design for cryptographic applications on fpgas," https://lirias.kuleuven.be/retrieve/67565, 2007.

[28] McIvor *et al.*, "An fpga elliptic curve cryptographic accelerator over gf(p)," *IET Conference Proceedings*, pp. 589–594(5), January 2004.

[29] Schinianakis *et al.*, "An rns implementation of an f_p elliptic curve point multiplier," *IEEE Transactions on Circuits and Systems I*, 2008.