A Scalable ECC Processor Implementation for High-Speed and Lightweight

Jens-Peter Kaps Ahmad Salman Ahmed Ferozpuri
Ekawat Homsirikamol Panasayya Yalla Kris Gaj

Cryptographic Engineering Research Group (CERG)
`http://cryptography.gmu.edu`
Department of ECE, Volgenau School of Engineering,
George Mason University, Fairfax, VA, USA

14th CryptArchi Workshop, 2016

Outline

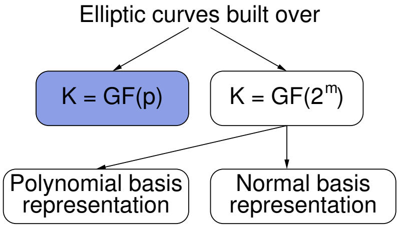
Introduction

- Why ECC:
  - Short key lengths, ciphertexts and signatures $\rightarrow$ smaller storage
  - Fast key generation
  - Fast digital signatures

AES	RSA	ECC
128	3,072	256–383
192	7,680	384–511
256	15,360	512+

- **High-Speed**: High throughput and throughput/area ratio
- **Lightweight**: Moderate throughput/area with minimal resource usage.

Elliptic Curves


Elliptic curves built over
K = GF(p)
K = GF(2^m)
Polynomial basis representation
Normal basis representation

Elliptic Curve over GF(p)

$\boldsymbol{y^2 = x^3 + ax + b}$

where $x, y, a, b \in GF(p)$

$$4a^3 + 27b^2 \not\equiv 0 (\bmod p)$$

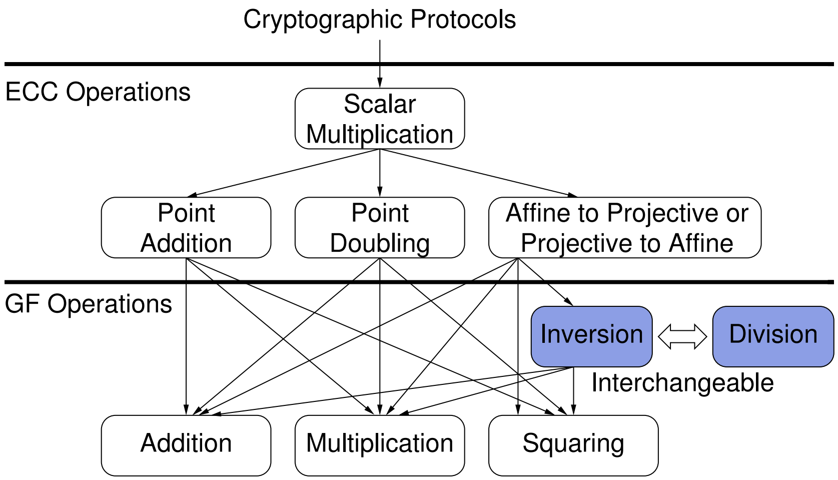
$+$ a special point called "*the point at infinity* **O**"

- K=GF(p): Arithmetic operations present in many libraries
- K=GF(2^m):
    - Fast in hardware
    - Compact in hardware

NIST Curves

- $GF(p)$ denotes a *prime field* with p elements where p is prime.
- $GF(2^m)$ denotes a *binary field* with 2^m elements for some m (called degree of the field)
- Recent improvements in attacking discrete logarithms over small-characteristic fields raised security concerns about binary curves (applies only to pairings for the time being).
- NIST *special curves* are those whose coefficients and underlying field have been selected to optimize the efficiency of the elliptic curve operations.
- NIST *special primes* are of a special type (called generalized Mersenne numbers) for which modular multiplication can be carried out more efficiently than in general.

ECC Operations



Projective coordinates

Coordinates	Point Addition			Point Doubling		
	#Muls	#Adds	#Invs	#Muls	#Adds	#Invs
A + A = A	3	8	1	4	5	1
P + A = A	13	7	0	N/A		
P + P = P	16	7	0	12	4	0
MJ+MJ = MJ	14	7	0	8	14	0

A $\rightarrow$ Affine; P $\rightarrow$ Projective; MJ $\rightarrow$ Modified Jacobian

- **Affine:** Requires time consuming inverse operation
- **Projective:** Only one inversion at the end of a full scalar multiplication
- **Modified Jacobian:** Proposed by Cohen et al.
  - Quadruple representation of a point (X, Y, Z, aZ^4)
  - Fast point doubling
- Easy conversion between Affine and Modified Jacobian

$$\begin{array}{lclclcl} P_A & = & (x, y) & \rightarrow & P_{MJ} & = & (x, y, 1, a) \\ P_{MJ} & = & (X, Y, Z, aZ^4) & \rightarrow & P_A & = & (X/Z^2, Y/Z^3) \end{array}$$

Montgomery Multiplication

X, Y, M are n-bit numbers, $R = 2^n$, $Z = X \cdot Y \bmod M$

Ordinary domain	$\Leftrightarrow$	**Montgomery domain**
X	$\longleftrightarrow$	$X' = X \cdot R \bmod M$
Y	$\longleftrightarrow$	$Y' = Y \cdot R \bmod M$
Z	$\longleftrightarrow$	$Z' = Z \cdot R \bmod M$

$Z' \leftarrow X' \cdot Y'$

$$\begin{aligned} Z' &= \mathrm{Mont}(X', Y', M) \\ &= X' \cdot Y' \cdot R^{-1} \bmod M \\ &= (X \cdot R) \cdot (Y \cdot R) R^{-1} \bmod M \\ &= X \cdot Y \cdot R \bmod M \end{aligned}$$

$X' \leftarrow X$

$$\begin{aligned} X' &= \mathrm{Mont}(X, R^2 \bmod M, M) \\ &= X \cdot R^2 \cdot R^{-1} \bmod M \\ &= X \cdot R \bmod M \end{aligned}$$

$Z \leftarrow Z'$

$$\begin{aligned} Z &= \mathrm{Mont}(Z', 1, M) \\ &= (Z \cdot R) \cdot 1 \cdot R^{-1} \bmod M \\ &= Z \bmod M \\ &= Z \end{aligned}$$

Montgomery Multiplication Architectures

- Tenca and Koc introduced a word-based algorithm for Montgomery multiplication, called Multiple-Word Radix-2 Montgomery Multiplication (MWR2MM), as well as a scalable hardware architecture capable of performing the multiplication operation using a variable number of Processing elements (PEs). (1999)
- The systolic high-radix design by McIvor et al. is capable of very high speed operation with the penalty of using large area requirements for fast multiplier units. (2004)
- Kaihara et al. proposed a concept which enables parallel execution of the Montgomery and Interleaved multiplication. (2005)
- Öksüzoğlu et al. reported DSP-based architecture for low-cost devices. (2008)

Montgomery Multiplication Implementations

- Harris *et al.* implemented the MWR2MM algorithm by left shifting one of the operands (Y) and the modulus (M) instead of right shifting the intermediate result (S). Their approach led to an improvement in terms of latency and latency $\times$ area by factor of two. (2001)
- Suzuki combined MWR2MM with the quotient pipelining technique and proposed an architecture which can be mapped efficiently onto modern high-performance DSP-oriented FPGA structure. (2007)
- Huang *et al.* proposed two architectures to optimize the original MWR2MM algorithm to process n-bit precision multiplication in approximately n clock cycles by precomputing intermediate S values. (2011)

ECC Scalar Multiplier Architectures

- Örs *et al.* introduced a module-based design for ECC processors over GF(P). The architecture is suitable for any prime field and any prime. The design uses Montgomery in a systolic array architecture to perform modular multiplication. (2003)
- Güneysu and Paar designed an ECC processor over GF(P) that is optimized for NIST P-224 and P-256 curves. They combine their multiplier and adder into a single unit and make use of DSP units found in FPGAs to perform fast multiplication and reduction operations. (2008)
- MuthuKumar and Jeevananthan proposed a high-speed ECC scalar multiplier over GF(P) and GF(2^m) for key-size of 256-bits. They use Jacobian coordinates and Montgomery multipliers built of 16 x 16 multiplication units. (2010)

ECC Scalar Multiplier Implementations

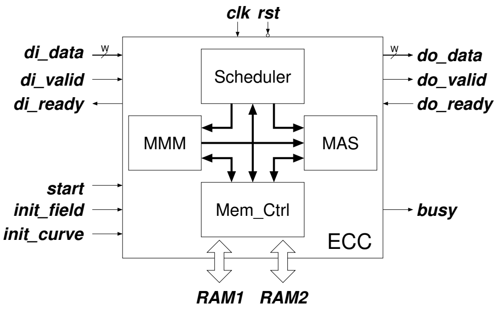
- Q. Xu *et al.* designed a low area ECC multiplier that supports NIST P-160, P-192, and P-256 curves. They proposed a tiny hardware module targeting ASICs. The design has counter measures to side-channel attacks (SPA), while having average performance. (2008)
- Alrimeih *et al.* implemented a hardware/software co-design for ECC processor to perform the scalar multiplication over GF(P). Supports all five prime fields recommended by NIST but also limited to and optimized for their corresponding primes. (2014)
- Sasdrich and Güneysu implemented a hardware accelerator for ECC point multiplication. The design is limited to Curve 25519 using pseudo Mersenne primes. Their work was expanded later to include techniques for counter measures against SPA and DPA attacks. (2015)

Design Decisions

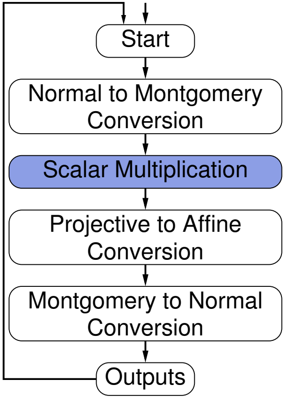
- Generalized for all $GF(P)$ curves for a specified field size
- External Memory usage
  - Support for ASIC implementations
  - Unified high-speed and lightweight storage requirements
- Support for all 5 NIST field sizes for a wide range of applications
- Not limited to special primes
  - Optimizations for special primes might be patent restricted
  - Generic design for FPGA and ASIC, not targeted for special FPGA features: e.g. DSP.
- High-speed design uses different word sizes (16, 32, and 64) to achieve high throughput
- Lightweight design uses a variable number of PE units (2, 4, or 8) to increase flexibility while maintaining low area

Top Level Architecture

- FIFO interface
- Independent initialization of field and curve parameters.
- Interface with external memory for ASIC implementations
- Modular Montgomery Multiplication (MMM)
- Modular Addition and Subtraction (MAS)



Scheduler


Start
Normal to Montgomery Conversion
Scalar Multiplication
Projective to Affine Conversion
Montgomery to Normal Conversion
Outputs

Scalar Multiplication $k \cdot P$

```
Require: Prime $p$, $P = (x, y) \in GF(p)$
    $k \in Z, 0 < k < p$,
    $k = (k_{l-1}, k_{l-2}, \dots, k_0)_2,\ k_{l-1} = 1$
Ensure: $Q = (x', y')$
    $Q = P$
    for $i = l - 2$ downto $0$ do
        $Q = 2Q$
        if $k_i = 1$ then
            $Q = Q + P$
    return $Q$
```

- Each state has its own controller making the design modular.
- **start** signal triggers a state to begin operation and hands control back to the scheduler by returning a **done** signal.

Scheduler: EC Point Addition

Algorithm 3(a)[1]

```
Require: $P_1 = (x, y, 1, a)$,
    $P_2 = (X_2, Y_2, Z_2, aZ_2^4)$
Ensure: $P_1 + P_2 = P_3 = (X_3, Y_3, Z_3, aZ_3^4)$
1: $T_1 \leftarrow Z_2^2$
2: $T_2 \leftarrow xT_1$
3: $T_1 \leftarrow T_1 Z_2$        $T_3 \leftarrow X_2 - T_2$
4: $T_1 \leftarrow yT_1$
5: $T_4 \leftarrow T_3^2$          $T_5 \leftarrow Y_2 - T_1$
6: $T_2 \leftarrow T_2 T_4$
7: $T_4 \leftarrow T_4 T_3$        $T_6 \leftarrow 2T_2$
8: $Z_3 \leftarrow Z_2 T_3$        $T_6 \leftarrow T_4 + T_6$
9: $T_3 \leftarrow T_5^2$
10: $T_1 \leftarrow T_1 T_4$       $X_3 \leftarrow T_3 - T_6$
11: $aZ_3^4 \leftarrow Z_3^2$      $T_2 \leftarrow T_2 - X_3$
12: $T_3 \leftarrow T_5 T_2$
13: $aZ_3^4 \leftarrow (aZ_3^4)^2$ $Y_3 \leftarrow T_3 - T_1$
14: $aZ_3^4 \leftarrow a(aZ_3^4)$
```

Control ROM: $Q = P + Q$

$P =$ (xR,yR,1,a), $Q =$ (X_q,Y_q,Z_q,aZ_q^4)

Multiplier			Adder			Ops
Res	**OP1**	**OP2**	**Res**	**OP1**	**OP2**	
T_1	Z_q	Z_q				mul
T_2	xR	T_1				mul
T_1	T_1	Z_q	T_3	X_q	T_2	mulsub
T_1	yR	T_1				mul
T_4	T_3	T_3	T_5	Y_q	T_1	mulsub
T_2	T_2	T_4				mul
T_4	T_4	T_3	T_6	T_2	T_2	muladd
Z_q	Z_q	T_3	T_6	T_4	T_6	muladd
T_3	T_5	T_5				mul
T_1	T_1	T_4	X_q	T_3	T_6	mulsub
aZ_q^4	Z_q	Z_q	T_2	T_2	X_q	mulsub
T3	T_5	T_2				mul
aZ_q^4	aZ_q^4	aZ_q^4	Y_q	T_3	T_1	mulsub
aZ_q^4	aR	aZ_q^4				mul

[1] S.B. Örs, L. Batina, B. Preneel, and J. Vandewalle, "Hardware Implementation of an Elliptic Curve Processor over *GF*(*p*)," in *ASAP 2003*, IEEE, Jun 2003.

Scheduler: EC Point Doubling

Algorithm 3(b)[1]

```
Require: $P_1 = (X_1, Y_1, Z_1, aZ_1^4)$,
Ensure: $2P_1 = P_3 = (X_3, Y_3, Z_3, aZ_4^4)$
 1: $T_1 \leftarrow Y_1^2$          $T_2 \leftarrow 2X_1$
 2: $T_3 \leftarrow T_1^2$          $T_2 \leftarrow 2T_2$
 3: $T_1 \leftarrow T_2 T_1$        $T_3 \leftarrow 2T_3$
 4: $T_2 \leftarrow X_1^2$          $T_3 \leftarrow 2T_3$
 5: $T_4 \leftarrow Y_1 Z_1$        $T_3 \leftarrow 2T_3$
 6: $T_5 \leftarrow T_3(aZ_1^4)$    $T_6 \leftarrow 2T_2$
 7: $T_2 \leftarrow T_6 + T_2$
 8: $T_2 \leftarrow T_2 + (aZ_1^4)$
 9: $T_6 \leftarrow T_2^2$          $Z_3 \leftarrow 2T_4$
10: $T_4 \leftarrow 2T_1$
11: $X_3 \leftarrow T_6 - T_4$
12: $T_1 \leftarrow T_1 - X_3$
13: $T_2 \leftarrow T_2 T_1$        $aZ_3^4 \leftarrow 2T_5$
14: $Y_3 \leftarrow T_2 - T_3$
```

Control ROM: $Q = 2Q$
$Q =$ (X_q,Y_q,Z_q,aZ_q^4)

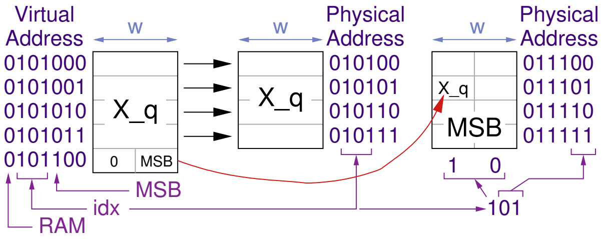
Multiplier			Adder			Ops
Res	**OP1**	**OP2**	**Res**	**OP1**	**OP2**	
T_1	Y_q	Y_q	T_2	X_q	X_q	muladd
T_3	T_1	T_1	T_2	T_2	T_2	muladd
T_1	T_2	T_1	T_3	T_3	T_3	muladd
T_2	X_q	X_q	T_3	T_3	T_3	muladd
T_4	Y_q	Z_q	T_3	T_3	T_3	muladd
T_5	T_3	aZ_q^4	T_6	T_2	T_2	muladd
			T_2	T_6	T_2	add
			T_2	T_2	aZ_q^4	add
T_6	T_2	T_2	Z_q	T_4	T_4	muladd
			T_4	T_1	T_1	add
			X_q	T_6	T_4	sub
			T_1	T_1	X_q	sub
T_2	T_2	T_1	aZ_q^4	T_5	T_5	muladd
			Y_q	T_2	T_3	sub

[1] S.B. Örs, L. Batina, B. Preneel, and J. Vandewalle, "Hardware Implementation of an Elliptic Curve Processor over $GF(p)$," in *ASAP 2003*, IEEE, Jun 2003.

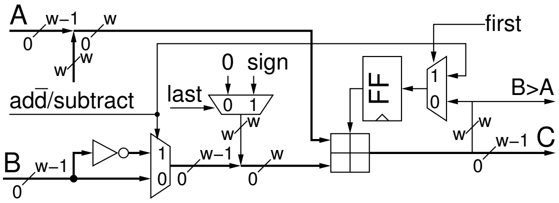
Memory

Nr	RAM	idx	Name
0	0	0	R^2 mod M
1	0	1	a
2	0	2	x
3	0	3	y
4	0	4	R
5	0	5	X_q
6	0	6	Y_q
7	0	7	Z_q
8	0	8	aZ_q^4
9	0	9	T_1
10	0	10	T_2
11	0	11	T_3
12	0	12	T_4
13	0	13	T_5
14	0	14	T_6
15	0	15	MSB521(0-14)
16	1	0	M
17	1	1	M-2
18	1	2	K
19	1	3	MSB521(16-18)

- We need to store 18 operands, incl. temporary values, each of size 521 bits.
- Memory 1: 16 x 512 bits, Memory 2: 4 x 512 bits.
- 9 MSB bits of 521-bit operands are stored in MSB521 locations and packed if $w = 64$. Example:



Modular Adder Subtracter (MAS)



- All supported field sizes except 521 are divisible by 16 and 32.
- Storing them in word-size memory does not leave space for sign.

Subtraction mod 241

```
01001  01001      153
11011  00110  -    73
1  111←11111    carry
---------------------
00101  10000
 0101   0000       80
=====================
```

Positive result ⇒ Done.

Subtraction mod 241

```
00100  01001       73
10110  00110  -   153
  1    1←11111  carry
---------------------
11011  10000
 1011   0000      -80
```

Negative result ⇒ addition of modulus.

Addition mod 241

```
01111  00001      241
11011  00000  +   -80
1111  ←         carry
---------------------
01010  00001
 1010   0001      161
=====================
```

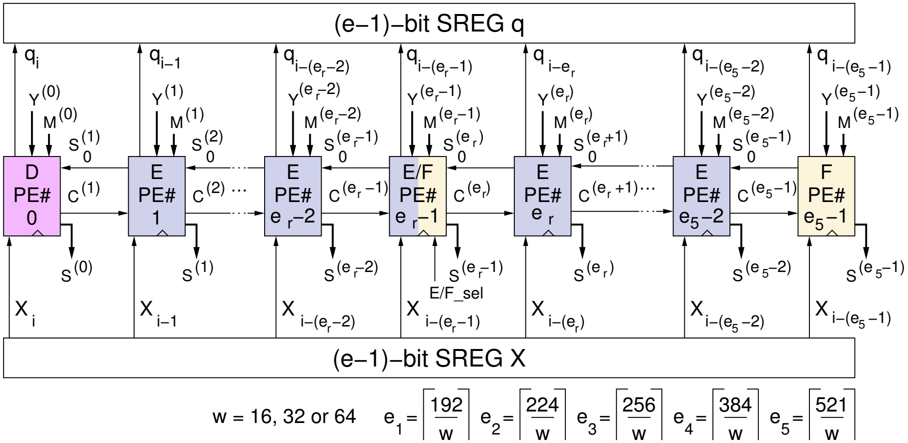
Modular Montgomery Multiplier (MMM)

Optimized MWR2MM [3]

```
1:  if $j = 0$ then
2:      $q_i = (x_i \cdot Y_0^{(0)}) \oplus S_1^{(0)}$
3:      $C^{(0)} = 0$
4:  if $j < e - 1$ then
5:      $(CO^{(j+1)}, SO_{w-1}^{(j)}, S_{w-2\ldots0}^{(j)}) = (1, S_{w-1\ldots1}^{(j)}) + C^{(j)} + x_i \cdot Y^{(j)} + q_i \cdot M^{(j)}$
6:      $(CE^{(j+1)}, SE_{w-1}^{(j)}, S_{w-2\ldots0}^{(j)}) = (0, S_{w-1\ldots1}^{(j)}) + C^{(j)} + x_i \cdot Y^{(j)} + q_i \cdot M^{(j)}$      Task D
7:      if $S_0^{(j+1)} = 1$ then
8:          $C^{(j+1)} = CO^{(j+1)}$      Task E
9:          $S_{w-1\ldots1}^{(j)} = (SO_{w-1}^{(j)}, S_{w-2\ldots1}^{(j)})$
10:     else
11:         $C^{(j+1)} = CE^{(j+1)}$
12:         $S_{w-1\ldots1}^{(j)} = (SE_{w-1}^{(j)}, S_{w-2\ldots1}^{(j)})$
13: else
14:     $(C^{(e)}, S^{(e-1)}) = (C^{(e)}, S_{w-1\ldots1}^{(e-1)}) + C^{(e-1)} + x_i \cdot Y^{(e-1)} + q_i \cdot M^{(e-1)}$      Task F
```

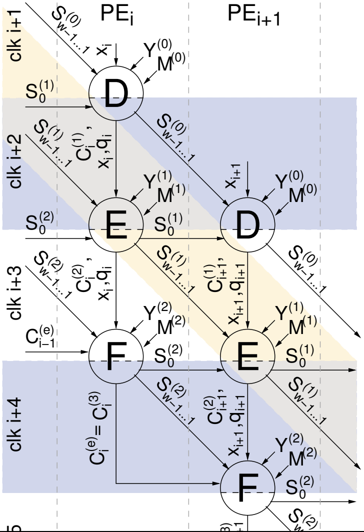
[3] M. Huang, K. Gaj, and T. El-Ghazawi. "New hardware architectures for Montgomery modular multiplication algorithm," in *IEEE ToCo*, 60(7), pp 923–936, Jul, 2011.

High-Speed Design

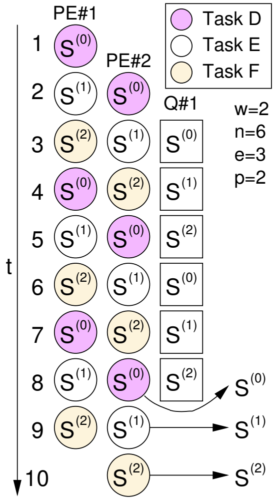

(e–1)–bit SREG q
q_i
q_{i-1}
$q_{i-(e_r-2)}$
$q_{i-(e_r-1)}$
q_{i-e_r}
$q_{i-(e_5-2)}$
$q_{i-(e_5-1)}$
$Y^{(0)}$ $M^{(0)}$ $S_0^{(1)}$
$Y^{(1)}$ $M^{(1)}$ $S_0^{(2)}$
$Y^{(e_r-2)}$ $M^{(e_r-2)}$ $S_0^{(e_r-1)}$
$Y^{(e_r-1)}$ $M^{(e_r-1)}$ $S_0^{(e_r)}$
$Y^{(e_r)}$ $M^{(e_r)}$ $S_0^{(e_r+1)}$
$Y^{(e_5-2)}$ $M^{(e_5-2)}$ $S_0^{(e_5-1)}$
$Y^{(e_5-1)}$ $M^{(e_5-1)}$
D PE# 0
E PE# 1
E PE# e_r-2
E/F PE# e_r-1
E PE# e_r
E PE# e_5-2
F PE# e_5-1
$C^{(1)}$
$C^{(2)}$ ...
$C^{(e_r-1)}$
$C^{(e_r)}$
$C^{(e_r+1)}$...
$C^{(e_5-1)}$
$S^{(0)}$
$S^{(1)}$
$S^{(e_r-2)}$
$S^{(e_r-1)}$
E/F_sel
$S^{(e_r)}$
$S^{(e_5-2)}$
$S^{(e_5-1)}$
X_i
X_{i-1}
$X_{i-(e_r-2)}$
$X_{i-(e_r-1)}$
$X_{i-(e_r)}$
$X_{i-(e_5-2)}$
$X_{i-(e_5-1)}$
(e–1)–bit SREG X
w = 16, 32 or 64
$e_1 = \left\lceil \frac{192}{w} \right\rceil$ $e_2 = \left\lceil \frac{224}{w} \right\rceil$ $e_3 = \left\lceil \frac{256}{w} \right\rceil$ $e_4 = \left\lceil \frac{384}{w} \right\rceil$ $e_5 = \left\lceil \frac{521}{w} \right\rceil$

- Based on Architecture 2 of [3].

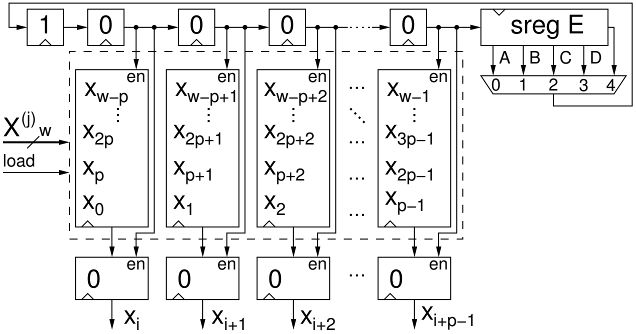
Lightweight Design



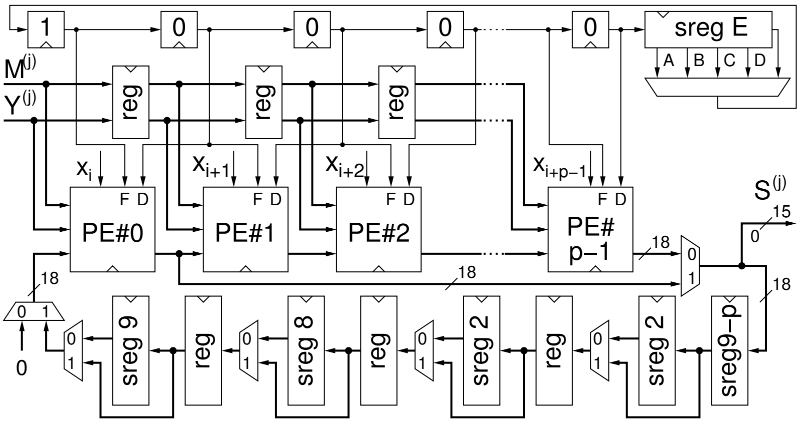
- Based on Architecture 1 [3].
- Each PE is capable of performing E, D, and F operations.
- If #PEs $< e$, we must store inbetween values in a queue of size $e - p$.



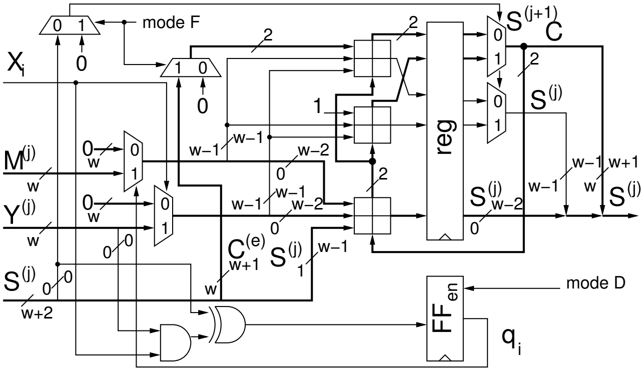
Reading and Reformatting x_i



Main Computational Unit



Inside the PE Unit



Test Setup

- Embedded memories are used only for “external RAM”.
- All implementations are coded in VHDL and do not use any other embedded resources.
- Implemented using Xilinx ISE 14.7 and Quartus Prime 16.0
- Optimized using ATHENa.
- All results are post place-and-route.

Xilinx		**Altera**	
Device	**Technology**	**Device**	**Technology**
		Cyclone-IV	60 nm
Spartan6	45 nm		
Virtex6	40 nm	Stratix-IV	40 nm
Artix7	28 nm	Cyclone-V	28 nm
Virtex7	28 nm	Stratix-V	28 nm
Zynq	28 nm		

Latency and Throughput for a given field and width

Field size	Latency in clock cycles			TP in Op/sec at f=100 MHz		
	W=64	W=32	W=16	W=64	W=32	W=16
192-bit	738,000	808,690	950,872	135.5	123.7	105.2
224-bit	1,000,500	1,083,582	1,267,434	100.0	92.3	78.9
256-bit	1,254,505	1,357,423	1,580,910	79.7	73.7	63.3
384-bit	2,791,687	3,021,972	3,482,490	35.8	33.1	28.7
521-bit	5,208,245	5,663,007	6,572,623	19.2	17.7	15.2
Average	**2,198,587**	**2,386,935**	**2,770,866**	**45.5**	**41.9**	**36.1**

TP$\rightarrow$ Throughput; Op$\rightarrow$ Opearations; $F \rightarrow$ Frequency

- Average TPs are based on average latencies.

Implementations results of high-speed design on Xilinx FPGAs

Width	Slices	LUTs	FFs	BRAMs	Clock [*Cycles*]	f [MHz]	TP $[\frac{Op}{sec}]$	TP/Area $[\frac{Op}{Slices \cdot sec}]$
Virtex7-xc7vx485tffg1761-3								
64	1,057	3,184	4,668	8	2,198,857	184	83.533	0.079
32	1,038	3,063	4,562	4	2,386,935	215	90.057	0.084
16	1,313	2,691	4,596	2	2,770,886	229	82.510	0.063
Virtex6-xc6vlx240tff1156-3								
64	1,125	3,216	4,668	8	2,198,857	163	73.933	0.066
32	1,029	3,028	4,562	4	2,386,935	200	83.823	0.081
16	1,208	2,763	4,596	2	2,770,886	227	81.818	0.068
Zynq-xc7z020clg484-3								
64	993	3,265	4,668	8	2,198,857	121	52.219	0.056
32	1,141	2,906	4,562	4	2,386,935	159	66.436	0.058
16	1,085	2,890	4,596	2	2,770,886	170	61.252	0.056

TP is calculated using the average latency at maximum frequency

Implementations results of high-speed design on Altera FPGAs

Width	ALMs	FFs	MBits	Clock [*Cycles*]	*f* [MHz]	TP $[\frac{Op}{sec}]$	TP/Area $[\frac{Op}{Slices \cdot sec}]$
Stratix V-5SGXEA7K2F40C3							
64	3,145	5,336	20,480	2,198,857	311	141.605	0.045
32	2,719	5,125	20,480	2,386,935	355	148.772	0.055
16	2,735	5,082	20,480	2,770,886	420	151.447	0.055
Stratix IV-EP4SE530H35C4							
64	3,818	4,667	20,480	2,198,857	277	103.280	0.027
32	3,587	4,575	20,480	2,386,935	268	112.286	0.031
16	3,585	4,607	20,480	2,770,886	291	105.126	0.029

TP is calculated using the average latency at maximum frequency

Power measurements using Xpower Analyzer

Dev.	Avail. LUTs	Width	P_{static} [mW]	$P_{dynamic}$ [mW]	P_{total} [mW]
VX7	303,600	64	241	52	293
		32	241	31	272
		16	241	30	271
VX6	150,720	64	3,424	86	3,510
		32	3,423	45	3,468
		16	3,423	54	3,477
ZQ	53,200	64	113	51	164
		32	133	31	164
		16	113	33	146
AX7	63,400	64	82	47	129
		32	82	32	114
		16	82	32	114
SN6	9,112	64	20	28	48
		32	20	4	24
		16	20	18	38

VX→Virtex; SN→Spartan; AX→Artix; ZQ→Zynq

Results are generated under the following conditions:

- Clock at 100 MHz.
- 10 randomly generated values of k for each of the five fields.
- Size of k is equal to curve field size.
- Static power of VX6 as reported by tool does not seem correct.

Comparison of high-speed results

Work	Device	Curve		Slices	LUTs	DSPs	BRAMs	f	TP	TP/Area
		Size	Type					[MHz]	$[\frac{Op}{sec}]$	$[\frac{Op}{Slices \cdot sec}]$
TW[W=32]	VX-6	192	GF(P)	1,029	3,028	0	4	200	247	0.240
		224	GF(P)						185	0.179
		256	GF(P)						147	0.143
		384	GF(P)						66	0.064
		521	GF(P)						35	0.034
Alrimeih *et al.*	VX-6	192	P-192	11,200	32,900	289	128	100	3,334	0.298
		224	P-224						2,858	0.255
		256	P-256						2,500	0.223
		384	P-384						848	0.076
		521	P-521						625	0.056
Roy *et al.*	VX-5	256	P-256	81	212	8	22	172	91	1.123
Baldwin *et al.*	VX-5	192	GF(P)		6,100			97	488	0.320
		256	GF(P)		7,800			82	248	0.127

TW→This Work; VX→ Virtex

Other results

Work	Device	Curve		Slices	LUTs	DSPs	BRAMs	f	TP	TP/Area
		Size	Type	(ALM*)				[MHz]	$[\frac{Op}{sec}]$	$[\frac{Op}{Slices \cdot sec}]$
Ghosh *et al.*	VX-4	192	GF(P)	14,900				53	286	0.019
		224	GF(P)	17,300				47	186	0.011
		256	GF(P)	20,100				43	130	0.006
Ananyi *et al.*	VX-4	192	P-192	20,800		32		60	239	0.011
		224	P-224					61	197	0.009
		256	P-256					62	164	0.008
		384	P-384					63	58	0.003
		521	P-521					64	26	0.001
Güneysu *et al.*	VX-4	224	P-224	24,452	32,688	468	198	372	30,438	1.245
		256	P-256		34,896	512	176	375	19,760	0.804
Güneysu *et al.*	VX-4	256	P-256		1,715	32		490	2,020	2.356
McIvor *et al.*	VX-2	256	GF(P)	15,755		256		40	260	0.017
Guillermin	SX-II	256	GF(P)	9,177*		96		157	1,471	0.160
Schiniakis *et al.*	SX-II	192	GF(P)	6,200*		92		161	2,273	0.367
		256	GF(P)	9,200*		96		157	1,471	0.160
		384	GF(P)	13,000*		177		151	741	0.057
		521	GF(P)	17,000*		244		145	449	0.026

VX$\rightarrow$ Virtex; SX$\rightarrow$Stratix;

Conclusions

- We designed two implementations of a scalable ECC processor, one for high-speed and one lightweight.
- Unlike many published results, our processor is not limited to NIST primes.
- Our TP/Area results are slightly lower than the high-speed design by Alrimeih, however, we use only a fraction of the BRAMs and no DSP units, neither contribute to TP/Area.
- The final version of our presentation will also contain results of our lightweight implementation.

Thanks for your attention.