A Scalable ECC Processor for High-Speed and Light-Weight Implementations with Side-Channel Countermeasures

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

Cryptographic Engineering Research Group (CERG)
http://cryptography.gmu.edu
Department of ECE, Volgenau School of Engineering,
George Mason University, Fairfax, VA, USA
Department of Integrated Science and Technology,
James Madison University, Harrisonburg, VA, USA

ReConFig-2017

Outline

1. Introduction
2. Previous Work
3. Implementation
4. Results and Conclusions

Introduction

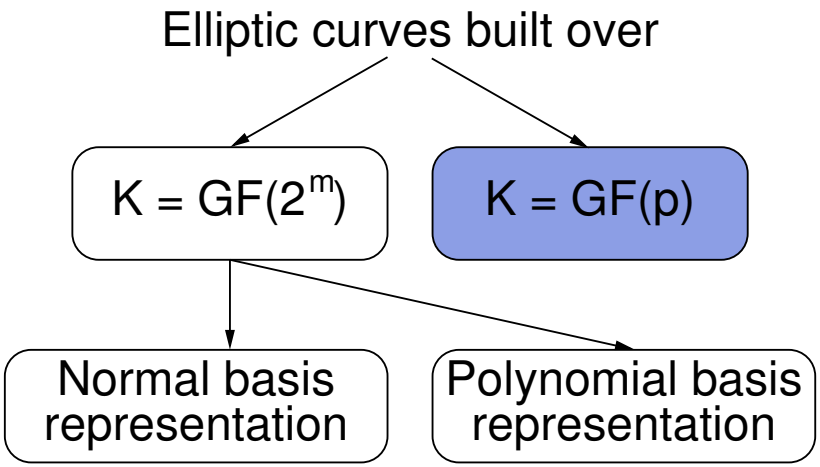
- Advantages of ECC:
  - Short key lengths, ciphertexts and signatures → smaller storage
  - Fast key generation
  - Fast digital signatures

- **High-Speed**: low latency and latency×delay
- **Lightweight**: Moderate latency×delay with minimal resource usage (area as well as power).

Min. of Strength	**Symm. Alg**	**RSA**	**ECC** len(p)
80	2TDEA	1,024	160
112	3TDEA	2,048	224
128	AES-128	3,072	256
192	AES-192	7,680	384
256	AES-256	15,360	521

[1] NIST SP 800-57, Pt.1, Rev.4, *Recommendation for Key Management: General*, Jan 2016.
[2] NIST FIPS PUB 186-4, *Digital Signature Standard (DSS)*, Jul 2013.

Elliptic Curves



Elliptic Curve over $GF(p)$

is the set of all pairs $(x, y) \in GF(p)$ which fulfill

$$y^2 \equiv x^3 + a \cdot x + b \bmod p$$

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

and $4 \cdot a^3 + 27 \cdot b^2 \not\equiv 0 \bmod p$

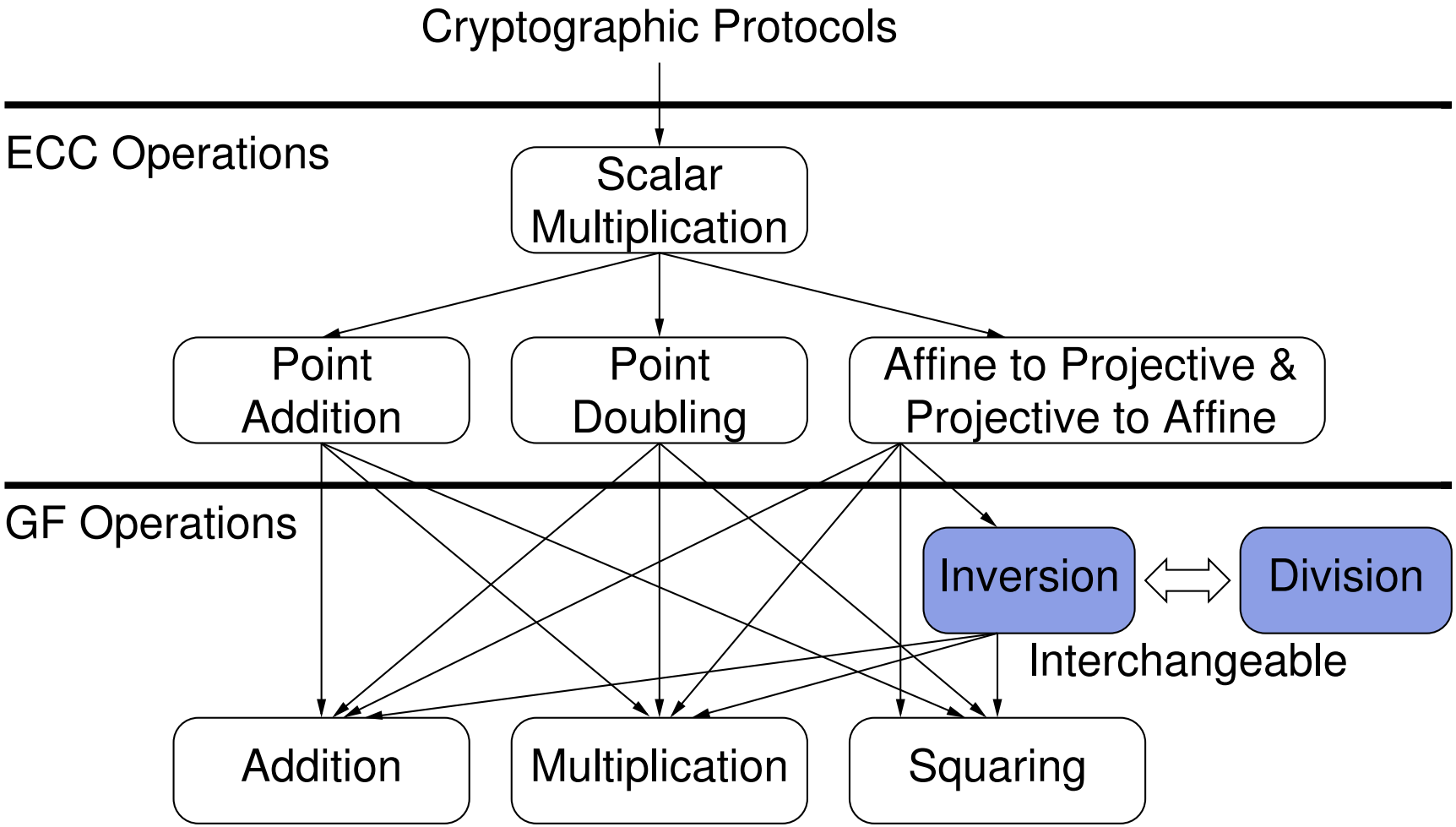
+ a special point called "*the point at infinity* $\mathcal{O}$"

- 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



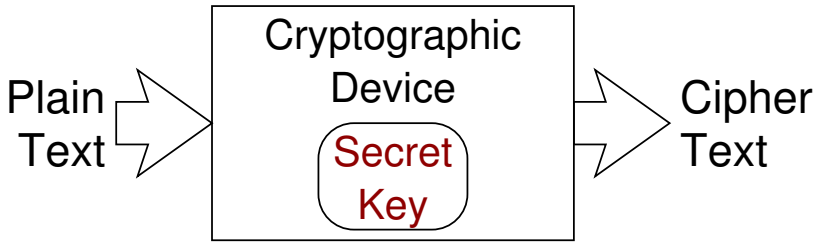
Scalar multiplication

- Fast and efficient in calculating the scalar multiplication.
- Does not require a lot of memory.
- Subject to side-channel attacks.
  - Simple Power Analysis (SPA)
  - Differential Power Analysis (DPA)

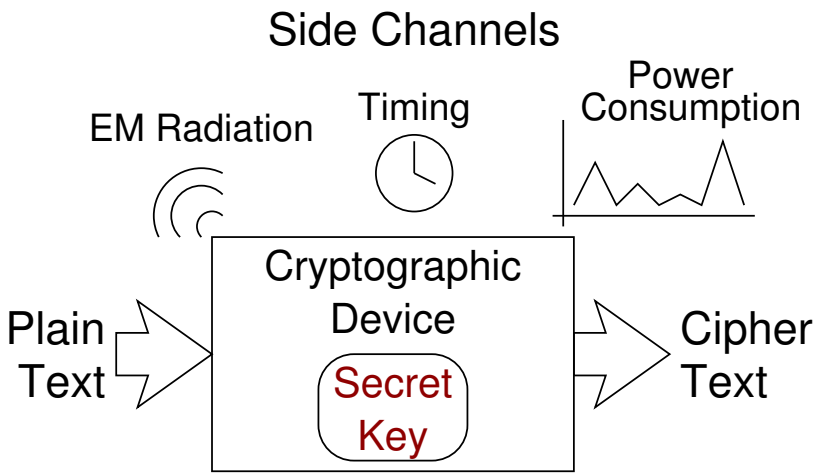
Double-and-Add $k \cdot P$

Require: Prime $p \in E(\mathbb{F}_q)$, $P = (x, y)$, where $x, y \in GF(p)$, $k \in \mathbb{Z}$,$0 < k < \#E$, $k = (k_{l-1}, k_{l-2}, \ldots, k_0)_2$, and $k_{l-1} = 1$
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

Side Channel Analysis (SCA)

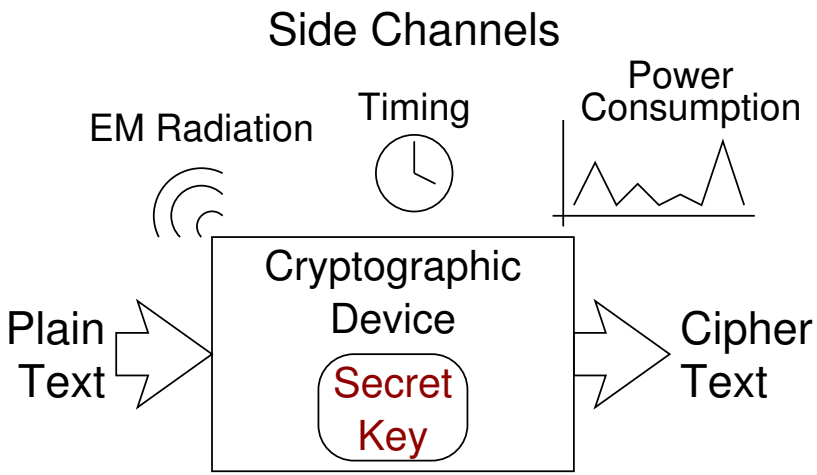


Side Channel Analysis (SCA)



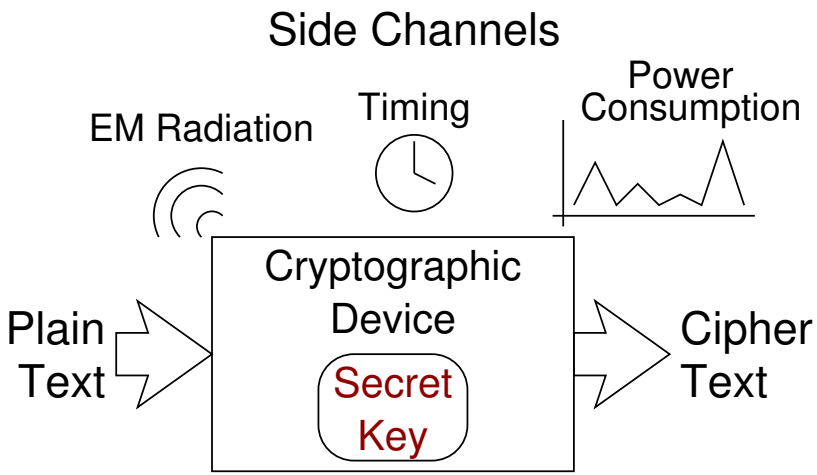
Side Channel Analysis (SCA)

- Danger: Implementations are susceptible to Side Channel Analysis (SCA).
- Key space 256-bit, $2^{256} = 1.2 \cdot 10^{77}$ keys
- Atoms in Universe (wikipedia) $9.4 \cdot 10^{79}$
- SCA allows to attack 8-bit at a time
- SCA complexity $\frac{256}{8} \cdot 2^8 = 8192$



Side Channel Analysis (SCA)

- Danger: Implementations are susceptible to Side Channel Analysis (SCA).
- Key space 256-bit, $2^{256} = 1.2 \cdot 10^{77}$ keys
- Atoms in Universe (wikipedia) $9.4 \cdot 10^{79}$
- SCA allows to attack 8-bit at a time
- SCA complexity $\frac{256}{8} \cdot 2^8 = 8192$

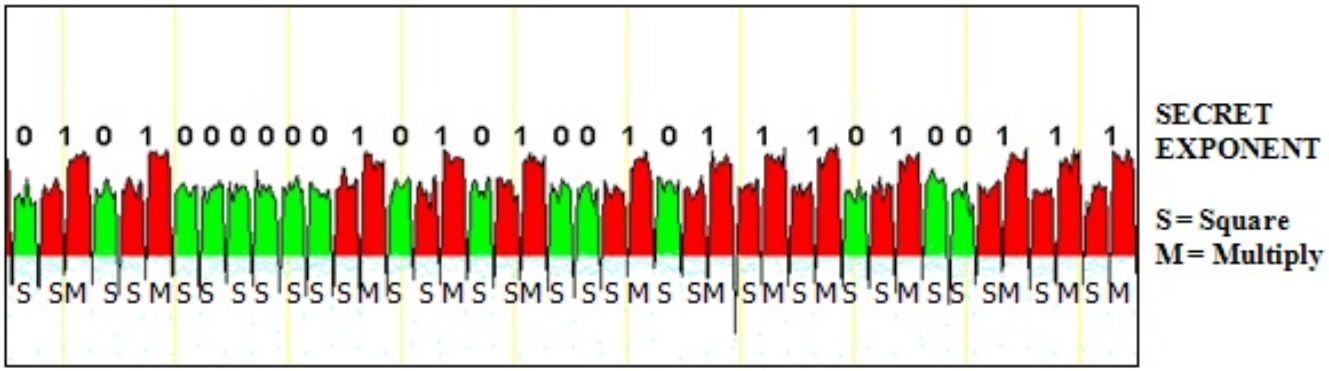


Danger

- These are passive, non invasive attacks.
- They are difficult to detect.
- The measurement setup is not very expensive ($200 – $20,000).
- Applies to Software as well as Hardware implementations.

Simple Power Analysis (SPA)

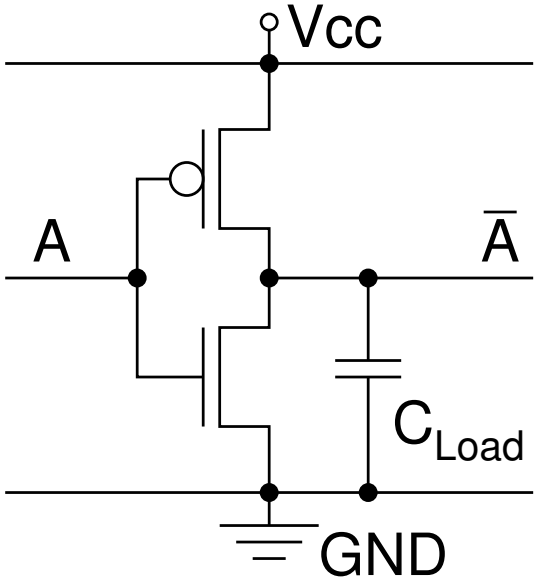
- Involves direct interpretation of power traces.(One or More Traces)
- Hamming weight of certain byte of the key, Number of rounds in the Algorithm(resp to Key Length)
- Memory accesses–have higher power consumption, repetitive patterns



Cons: Attacker needs to have detailed information about the algorithm.

Differential Power Analysis (DPA)

- Kocher et al. proposed power analysis attack in 1999.
- DPA exploits the data dependency between power consumption of cryptographic device and secret.
- Detailed knowledge about the attacked device is not required.

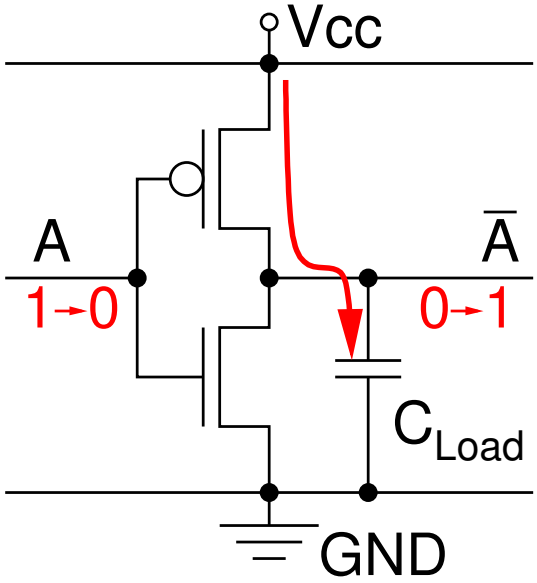


DPA in a Nutshell

1. Choose intermediate result, should be function of data & key.
2. Measure the power consumption.
3. Calculate the hypothetical values and build a power model.
4. Compare hypothetical power model with power consumption.

Differential Power Analysis (DPA)

- Kocher et al. proposed power analysis attack in 1999.
- DPA exploits the data dependency between power consumption of cryptographic device and secret.
- Detailed knowledge about the attacked device is not required.

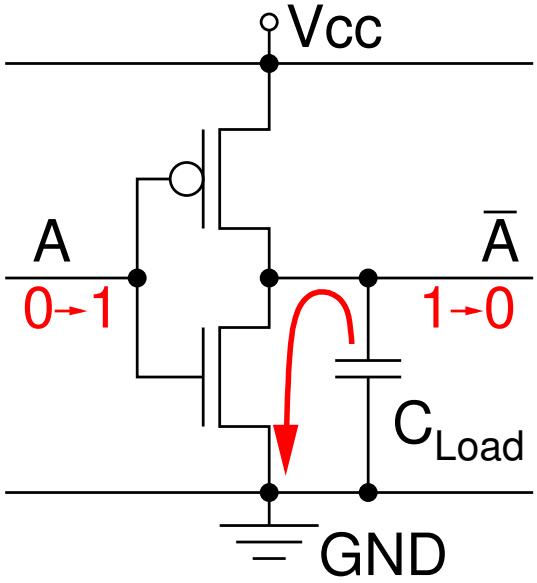


DPA in a Nutshell

1. Choose intermediate result, should be function of data & key.
2. Measure the power consumption.
3. Calculate the hypothetical values and build a power model.
4. Compare hypothetical power model with power consumption.

Differential Power Analysis (DPA)

- Kocher et al. proposed power analysis attack in 1999.
- DPA exploits the data dependency between power consumption of cryptographic device and secret.
- Detailed knowledge about the attacked device is not required.



DPA in a Nutshell

1. Choose intermediate result, should be function of data & key.
2. Measure the power consumption.
3. Calculate the hypothetical values and build a power model.
4. Compare hypothetical power model with power consumption.

Safe Scalar multiplication

- Montgomery Ladder protects against SPA and safe-error attacks.
- Requires more computations to calculate the scalar multiplication.
- Coron's "Randomization of the Private Exponent" method protects against DPA. Calculate $Q = k'P$ instead of $Q = kP$ where $k' = k + d \cdot \#E$.

Montgomery Ladder $k \cdot P$

Require: Prime $p \in E(\mathbb{F}_q)$, $P = (x, y)$,
where $x, y \in GF(p)$
$k \in \mathbb{Z}, 0 < k < \#E$,
$k = (k_{l-1}, k_{l-2}, \ldots, k_0)_2$, $k_{l-1} = 1$
Ensure: $Q = (x', y')$
$Q = P$
$P = 2Q$
for $i = l - 2$ **downto** 0 **do**
if $k_i = 1$ **then**
$Q = Q + P$
$P = 2P$
else
$P = P + Q$
$Q = 2Q$
return Q

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 → Affine; P → Projective; MJ → 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

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

Co-Z Scalar multiplication

- Point addition can be accelerated if both points P and Q share the same Z coordinate.
- Montgomery Ladder can be efficiently calculated using three algorithms
  - Add with update (ZADDU)
  - Add with conjugate (ZADDC)
  - Double with update (DBLU)

Montgomery Ladder $k \cdot P$

Require: Prime $p \in E(\mathbb{F}_q)$, $P = (X, Y, Z)$, where $X, Y, Z \in GF(p)$, $k \in \mathbb{Z}, 0 < k < \#E$, $k = (k_{l-1}, k_{l-2}, \ldots, k_0)_2$, $k_{l-1} = 1$
Ensure: $Q = (X', Y', Z)$
  $P, Q = DBLU(P)$
  for $i = l - 2$ **downto** 0 **do**
    if $k_i = 1$ **then**
      $Q, P = ZADDC(P, Q)$
      $P, Q = ZADDU(Q, P)$
    else
      $P, Q = ZADDC(Q, P)$
      $Q, P = ZADDU(P, Q)$
  return Q

- Almost as fast as Double-and-ADD algorithm.

Montgomery Multiplication

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

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

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

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

$X' \leftarrow X$

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

$Z \leftarrow Z'$

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

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)
- Amiet *et al.* implemented a generalized ECC processor which supports all five of NIST prime fields for any prime p. Their implementation uses two Montgomery multiplier units which work in parallel with a Modular adder/subtractor to perform the prescheduled scalar multiplication operations. (2016)
- 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

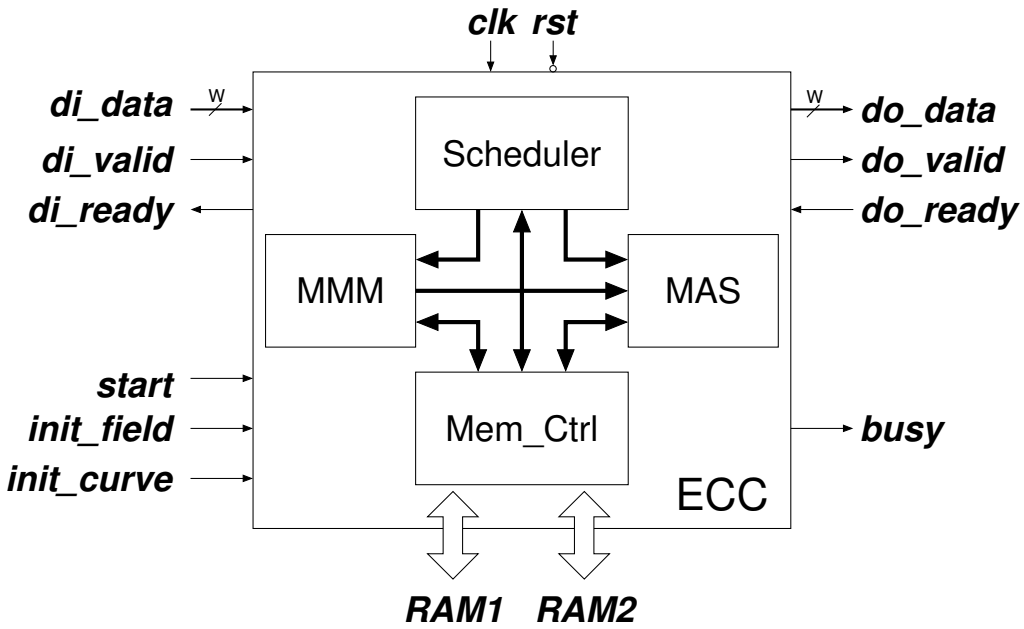
- Q. Xu *et al.* designed a low area ECC multiplier that supports NIST p-160, p-192, and p-256 curves. They proposed a tiny hardware module targeting ASICs. The design has counter measures to side-channel attacks (SPA), while having average performance. (2008)
- Alrimeih *et al.* implemented a hardware/software 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

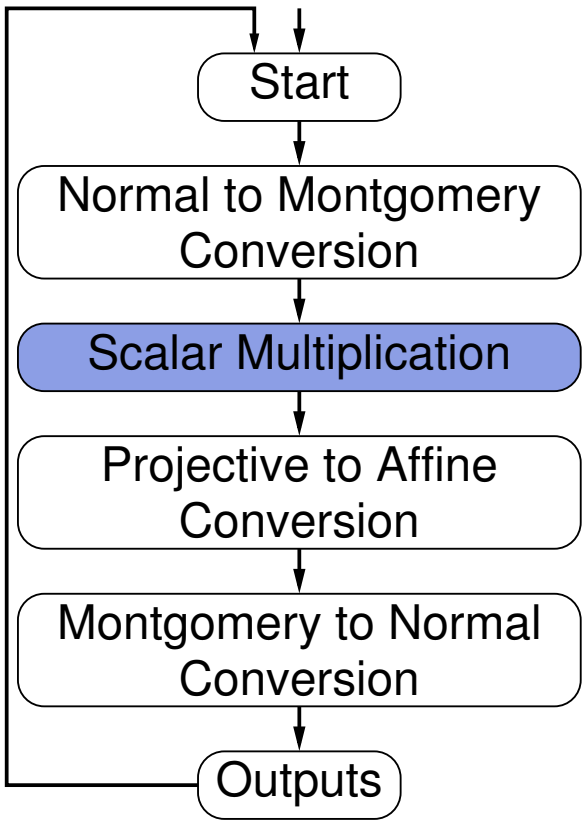
- Generalized for all $GF(p)$ curves for a specified field size.
- External Memory usage
  - Support for ASIC implementations,
  - mapps to embedded memories on FPGAs.
  - Unified high-speed and lightweight storage requirements.
- Support for all 5 NIST field sizes for a wide range of applications: 192, 224, 256, 384, and 521.
- 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) and redundant representation 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



- 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 (unprotected)

Algorithm 3(a)[3]

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: $T_1 \leftarrow T_1 Z_2$ $\quad T_3 \leftarrow X_2 - T_2$
4: $T_1 \leftarrow yT_1$
5: $T_4 \leftarrow T_3^2$ $\quad T_5 \leftarrow Y_2 - T_1$
6: $T_2 \leftarrow T_2 T_4$
7: $T_4 \leftarrow T_4 T_3$ $\quad T_6 \leftarrow 2T_2$
8: $Z_3 \leftarrow Z_2 T_3$ $\quad T_6 \leftarrow T_4 + T_6$
9: $T_3 \leftarrow T_5^2$
10: $T_1 \leftarrow T_1 T_4$ $\quad X_3 \leftarrow T_3 - T_6$
11: $aZ_3^4 \leftarrow Z_3^2$ $\quad T_2 \leftarrow T_2 - X_3$
12: $T_3 \leftarrow T_5 T_2$
13: $aZ_3^4 \leftarrow (aZ_3^4)^2$ $\quad Y_3 \leftarrow T_3 - T_1$
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

[3] 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 (unprotected)

Algorithm 3(b)[3]

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

[3] 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 ZADDC (protected)

Algorithm 13[4]

Require: $P_1 = (X_1, Y_1, Z)$, $P_2 = (X_2, Y_2, Z)$

Ensure: $P_1 + P_2 = P_3 = (X_3, Y_3, Z_3) \sim P_1$

1 :		$T_1 \leftarrow X_1 - X_2$
2 :	$T_2 \leftarrow T_1^2$	$T_3 \leftarrow Y_1 - Y_2$
3 :	$Z_3 \leftarrow ZT_1$	$T_4 \leftarrow Y_1 + Y_2$
4 :	$T_1 \leftarrow X_1 T_2$	
5 :	$X_2 \leftarrow X_2 T_2$	
6 :	$Y_2 \leftarrow T_4^2$	$T_5 \leftarrow T_1 - X_2$
7 :	$T_2 \leftarrow T_3^2$	$Y_2 \leftarrow Y_2 - T_1$
8 :	$T_5 \leftarrow Y_1 T_5$	$T_2 \leftarrow T_2 - T_1$
9 :		$X_3 \leftarrow Y_2 - X_2$
10:		$X_2 \leftarrow T_2 - X_2$
11:		$Y_2 \leftarrow T_1 - X_2$
12:	$T_2 \leftarrow T_3 Y_2$	$T_1 \leftarrow T_1 - X_3$
13:	$T_3 \leftarrow T_4 T_1$	$Y_2 \leftarrow T_2 - T_5$
14:		$Y_3 \leftarrow T_3 - T_5$

Control ROM: $Q, P = P + Q$

P = (X_p,Y_p,Z), Q = (X_q,Y_q,Z)

Multiplier			Adder			Ops
Res	OP1	OP2	Res	OP1	OP2	
			T_1	X_p	X_q	sub
T_2	T_1	T_1	T_3	Y_p	Y_q	mulsub
Z	Z	T_1	T_4	Y_p	Y_q	muladd
T_1	X_p	T_2				mul
X_q	X_q	T_2				mul
Y_q	T_4	T_4	T_5	T_1	X_q	mulsub
T_2	T_3	T_3	Y_q	Y_q	T_1	mulsub
T_5	Y_p	T_5	T_2	T_2	T_1	mulsub
			X_p	Y_q	X_q	sub
			X_q	T_2	X_q	sub
			Y_q	T_1	X_q	sub
T_2	T_3	Y_q	T_1	T_1	X_p	mulsub
T_3	T_4	T_1	Y_q	T_2	T_5	mulsub
			Y_p	T_3	T_5	sub

[4] R. R. Goundar, M. Joye, and A. Miyaji, "Co-z addition formula and binary ladders on elliptic curves" , CHES10. 2010, pp. 6579.

Scheduler: EC Point ZADDU (protected)

Algorithm 12[4]

Require: $P_1 = (X_1, Y_1, Z)$,
$P_2 = (X_2, Y_2, Z)$
Ensure: $P_2 + P_1 = P_3 = (X_3, Y_3, Z_3) \sim P_2$

1 :		$T_1 \leftarrow X_1 - X_2$
2 :	$T_2 \leftarrow T_1^2$	$T_3 \leftarrow Y_1 - Y_2$
3 :	$Z_3 \leftarrow ZT_1$	
4 :	$T_1 \leftarrow X_2 T_2$	
5 :	$X_1 \leftarrow X_2 T_2$	
6 :	$T_4 \leftarrow T_3^2$	$T_5 \leftarrow X_1 - T_1$
7 :	$Y_1 \leftarrow Y_1 T_5$	$T_4 \leftarrow T_4 - X_1$
8 :		$X_3 \leftarrow T_4 - T_1$
9 :		$Y_2 \leftarrow X_1 - X_3$
10:	$Y_2 \leftarrow T_3 Y_2$	
11:		$Y_3 \leftarrow Y_2 - Y_1$

Control ROM: $P, Q = Q + P$

P = (X_p,Y_p,Z), Q = (X_q,Y_q,Z)

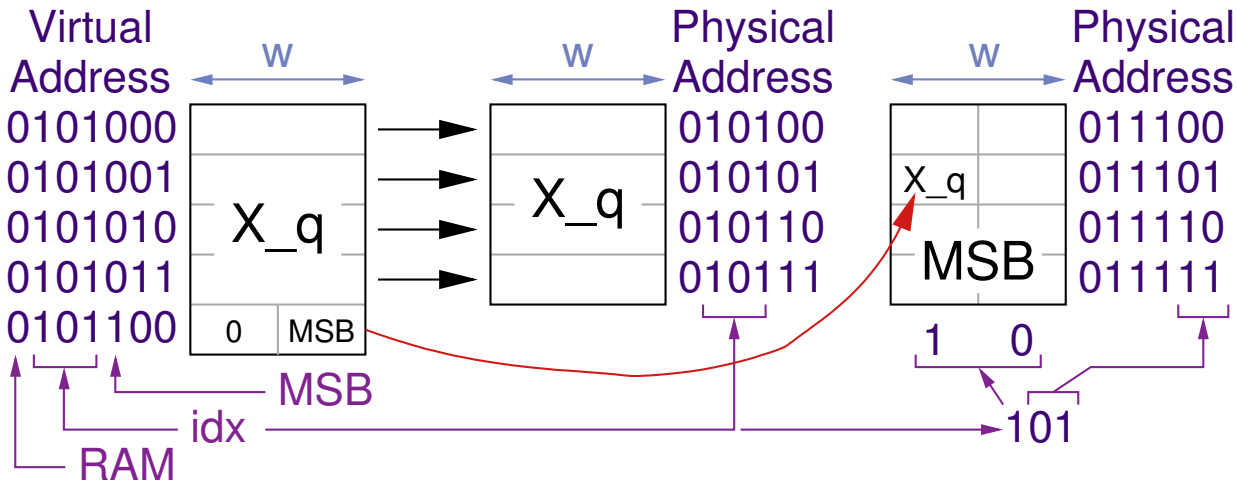
Multiplier			Adder			Ops
Res	OP1	OP2	Res	OP1	OP2	
			T_1	X_p	X_q	sub
T_2	T_1	T_1	T_3	Y_p	Y_q	mulsub
Z	Z	T_1				mul
T_1	X_q	T_2				mul
X_p	X_p	T_2				mul
T_4	T_3	T_3	T_5	X_p	T_1	mulsub
Y_p	Y_p	T_5	T_4	T_4	X_p	mulsub
			X_q	T_4	T_1	sub
			Y_q	X_p	X_q	sub
Y_q	T_3	Y_q				mul
			Y_q	Y_q	Y_p	sub

[4] R. R. Goundar, M. Joye, and A. Miyaji, "Co-z addition formula and binary ladders on elliptic curves" , CHES10. 2010, pp. 6579.

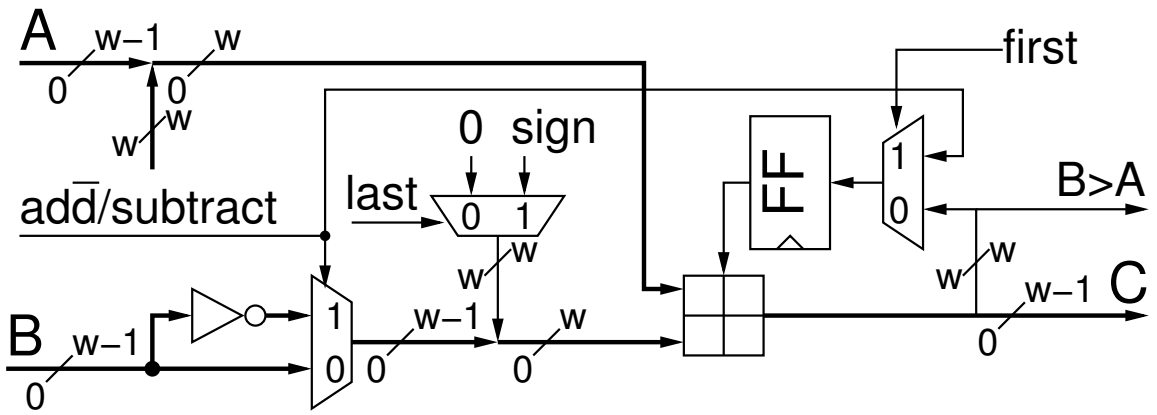
Memory

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 maximum 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)

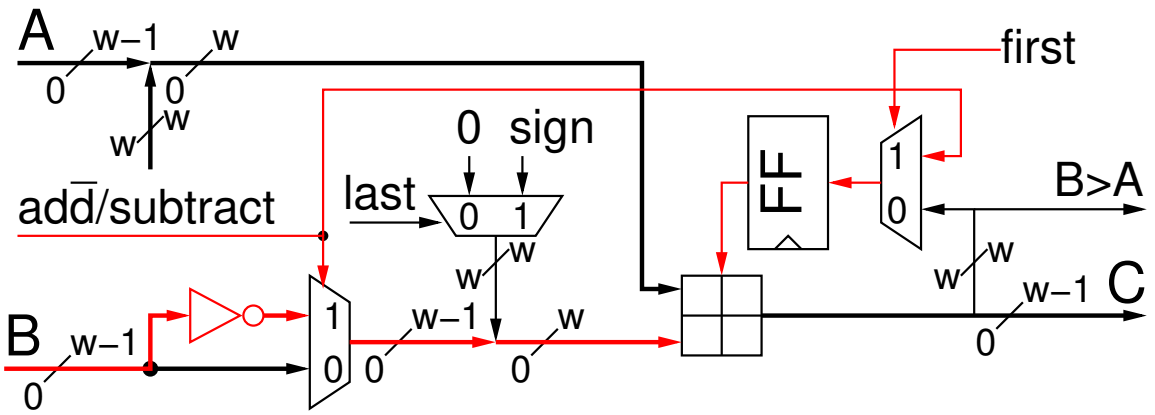


- All supported field sizes except 521 are divisible by 16 and 32.
- Storing them in word-size memory does not leave space for sign.

Subtraction mod 241

1100	1100	204
1001	1001	153

Modular Adder Subtracter (MAS)

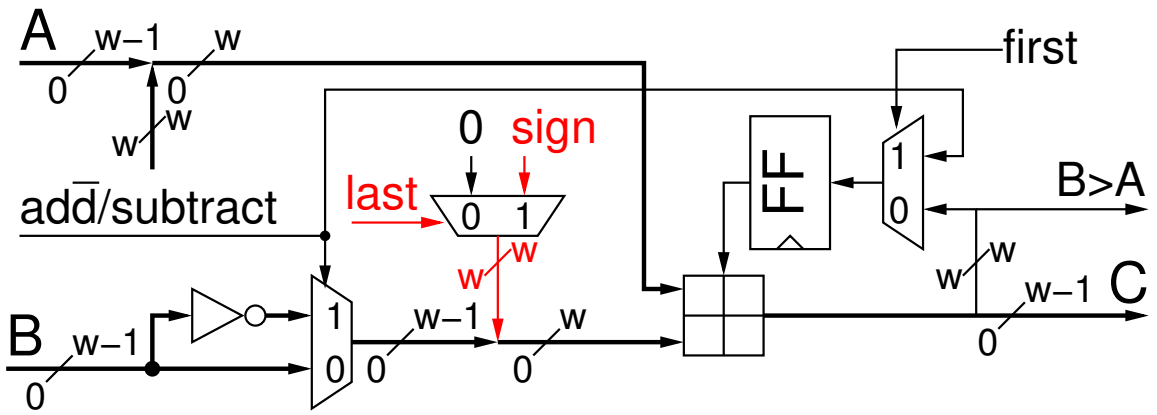


- All supported field sizes except 521 are divisible by 16 and 32.
- Storing them in word-size memory does not leave space for sign.

Subtraction mod 241

1100	1100	204
0110	0110	- 153
	1	

Modular Adder Subtracter (MAS)

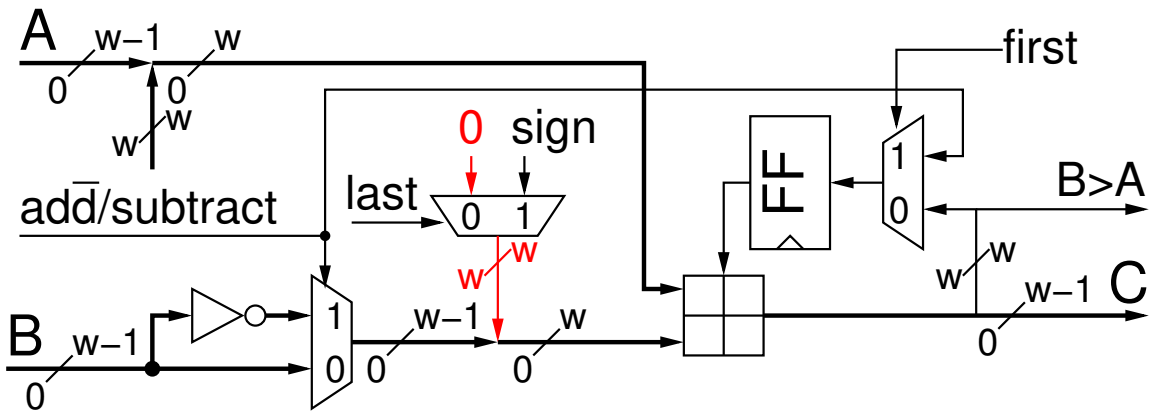


- All supported field sizes except 521 are divisible by 16 and 32.
- Storing them in word-size memory does not leave space for sign.

Subtraction mod 241

1100	1100	204
10110	0110	- 153
	1	

Modular Adder Subtracter (MAS)

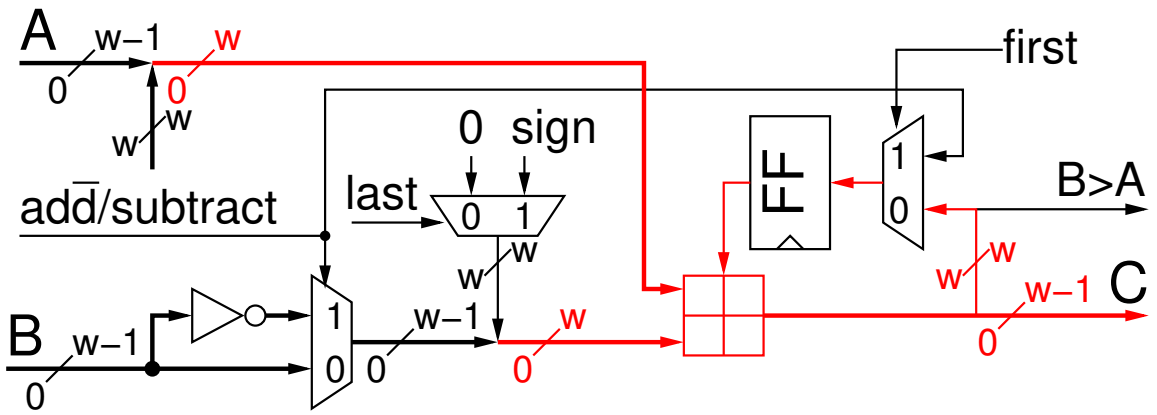


- All supported field sizes except 521 are divisible by 16 and 32.
- Storing them in word-size memory does not leave space for sign.

Subtraction mod 241

```
01100  01100     204
10110  00110   - 153
              1
-------------------
```

Modular Adder Subtracter (MAS)

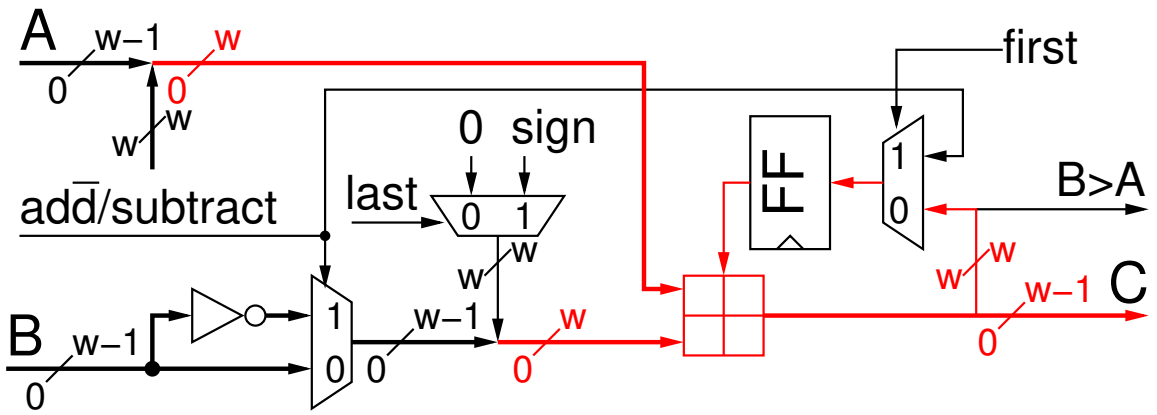


- All supported field sizes except 521 are divisible by 16 and 32.
- Storing them in word-size memory does not leave space for sign.

Subtraction mod 241

```
01100  01100     204
10110  00110   - 153
11   1←11   1    carry
-----------------------
00011  10011
```

Modular Adder Subtracter (MAS)

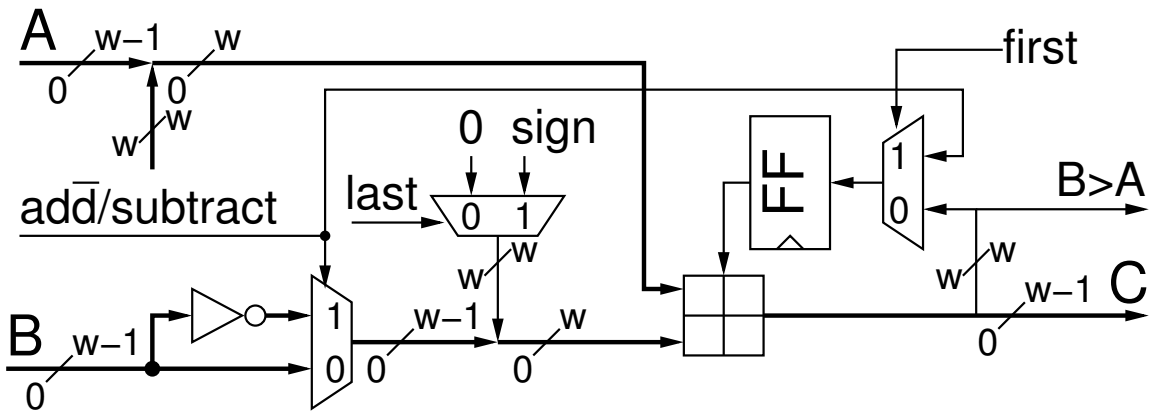


- All supported field sizes except 521 are divisible by 16 and 32.
- Storing them in word-size memory does not leave space for sign.

Subtraction mod 241

01100 01100	204
10110 00110	- 153
11 1←11 1	carry
00011 10011	
0011 0011	51

Modular Adder Subtracter (MAS)



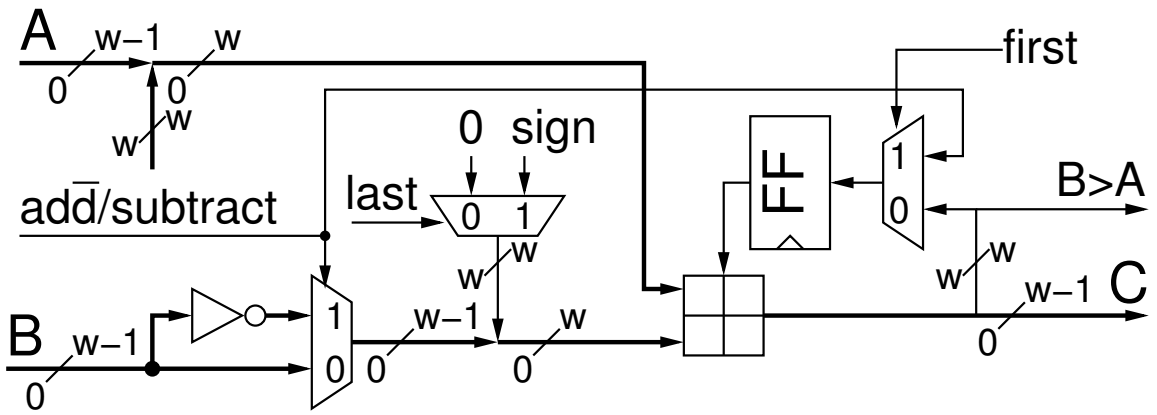
- All supported field sizes except 521 are divisible by 16 and 32.
- Storing them in word-size memory does not leave space for sign.

Subtraction mod 241

01100	01100	204
10110	00110	- 153
11	1←11 1	carry
00011	10011	
0011	0011	51

Positive result ⇒ Done.

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

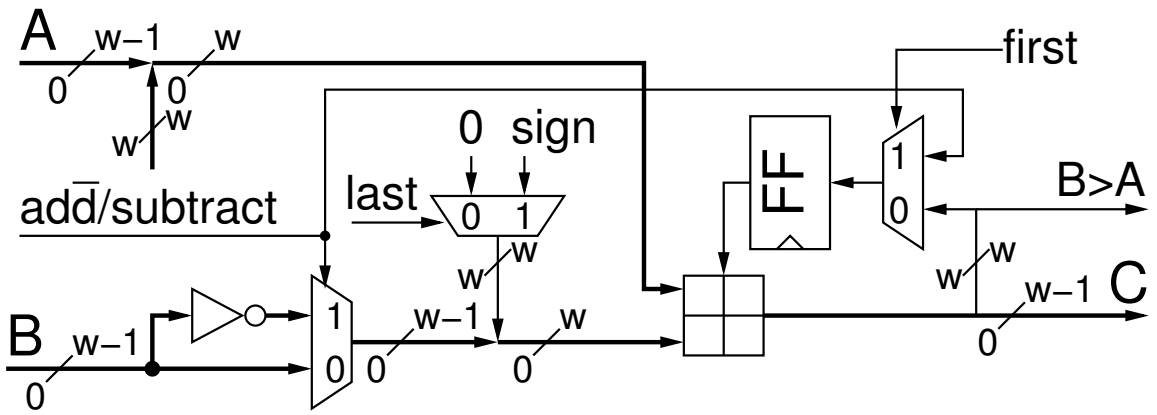
01100	01100	204
10110	00110	- 153
11	1←11	1 carry
00011	10011	
0011	0011	51

Positive result ⇒ Done.

Subtraction mod 241

0100	1001	73
0110	0110	- 153
	1	

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

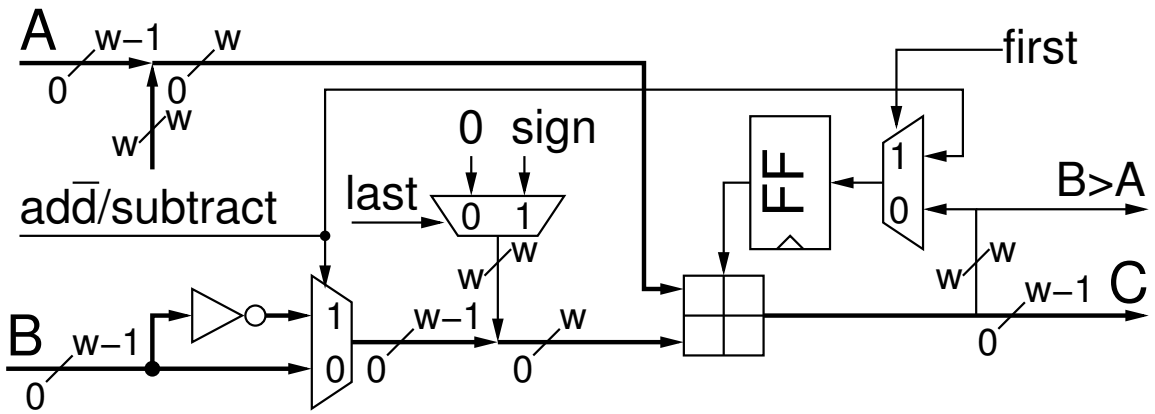
```
01100  01100      204
10110  00110    - 153
11   1←11   1     carry
-----------------------
00011  10011
 0011   0011      51
=======================
```

Positive result ⇒ Done.

Subtraction mod 241

```
00100  01001      73
10110  00110    - 153
              1
-----------------------
```

Modular Adder Subtracter (MAS)



Subtraction mod 241

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

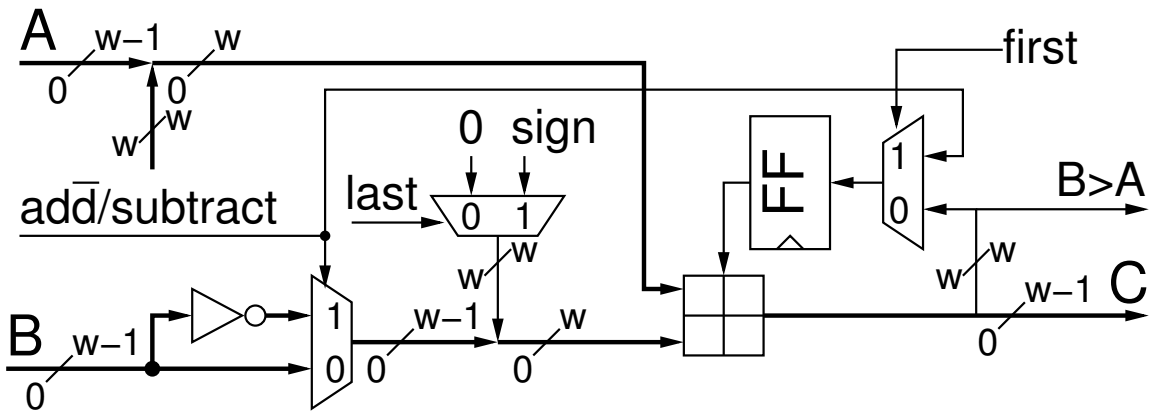
- All supported field sizes except 521 are divisible by 16 and 32.
- Storing them in word-size memory does not leave space for sign.

Subtraction mod 241

```
01100  01100      204
10110  00110    - 153
11    1←11    1   carry
-----------------------
00011  10011
 0011   0011      51
=======================
```

Positive result ⇒ Done.

Modular Adder Subtracter (MAS)



Subtraction mod 241

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

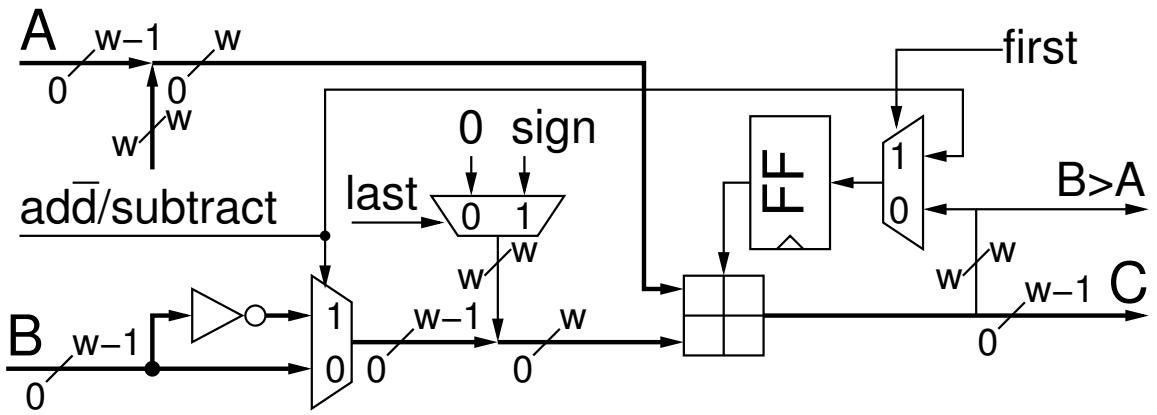
- All supported field sizes except 521 are divisible by 16 and 32.
- Storing them in word-size memory does not leave space for sign.

Subtraction mod 241

01100	01100	204
10110	00110	- 153
11	1←11 1	carry
00011	10011	
0011	0011	51

Positive result ⇒ Done.

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

```
01100  01100     204
10110  00110   - 153
11   1←11   1   carry
-----------------------
00011  10011
 0011   0011      51
```

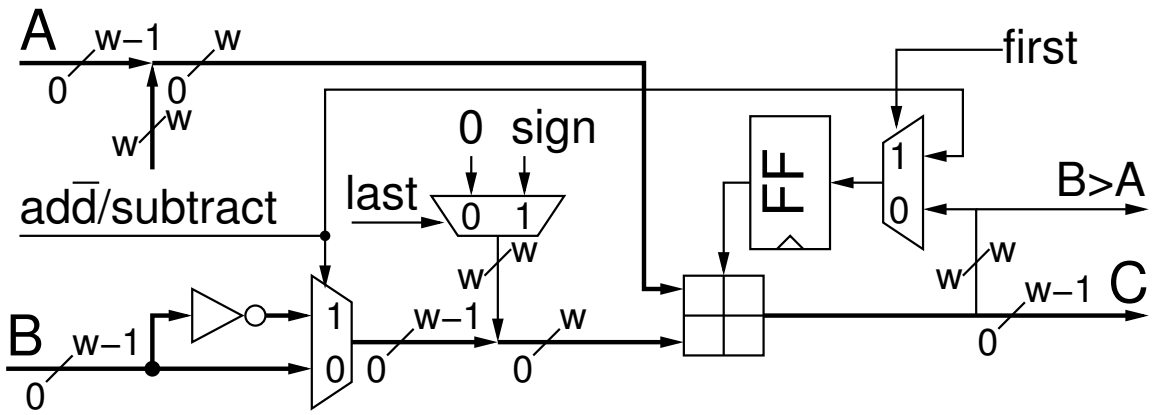
Positive result ⇒ Done.

Subtraction mod 241

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

Negative result ⇒ addition of modulus.

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

01100	01100	204
10110	00110	- 153
11	1←11	1 carry
00011	10011	
0011	0011	51

Positive result ⇒ Done.

Subtraction mod 241

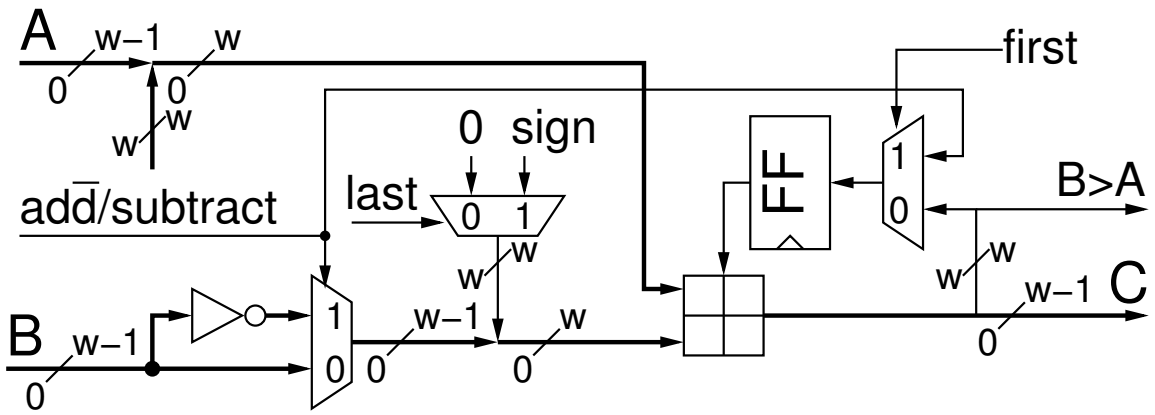
00100	01001	73
10110	00110	- 153
1	1←11111	carry
11011	10000	
1011	0000	-80

Negative result ⇒ addition of modulus.

Addition mod 241

1111	0001	241
1011	0000	+ -80

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

01100	01100	204
10110	00110	- 153
11	1←11	1 carry
00011	10011	
0011	0011	51

Positive result ⇒ Done.

Subtraction mod 241

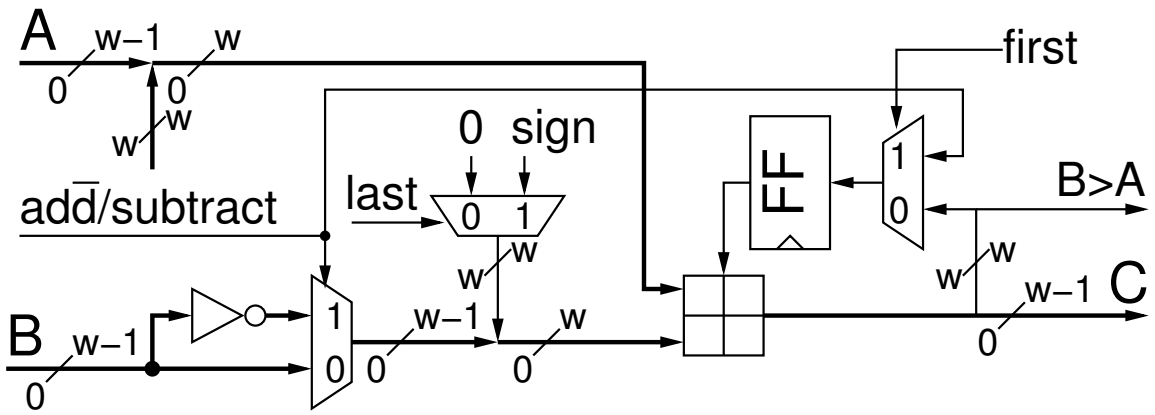
00100	01001	73
10110	00110	- 153
1	1←11111	carry
11011	10000	
1011	0000	-80

Negative result ⇒ addition of modulus.

Addition mod 241

1111	0001	241
11011	0000	+ -80
	0	

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

01100	01100	204
10110	00110	- 153
11	1←11	1 carry
00011	10011	
0011	0011	51

Positive result ⇒ Done.

Subtraction mod 241

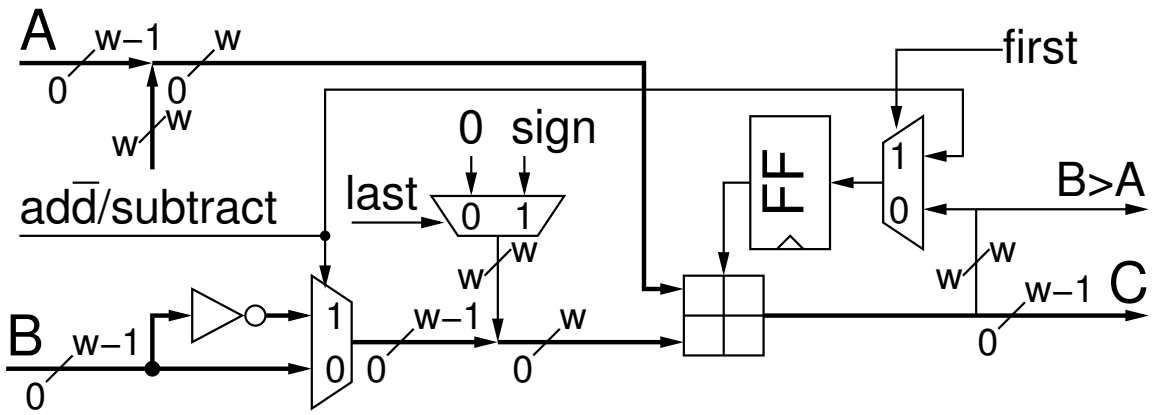
00100	01001	73
10110	00110	- 153
1	1←11111	carry
11011	10000	
1011	0000	-80

Negative result ⇒ addition of modulus.

Addition mod 241

01111	00001	241
11011	00000	+ -80
	0	

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

01100	01100	204
10110	00110	- 153
11	1←11	1 carry
00011	10011	
0011	0011	51

Positive result ⇒ Done.

Subtraction mod 241

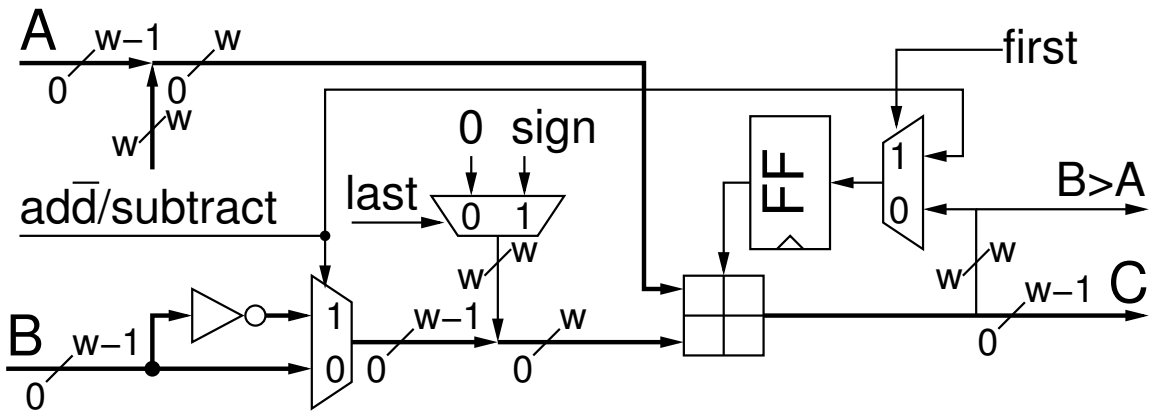
00100	01001	73
10110	00110	- 153
1	1←11111	carry
11011	10000	
1011	0000	-80

Negative result ⇒ addition of modulus.

Addition mod 241

01111	00001	241
11011	00000	+ -80
1111	← 0	carry
01010	00001	

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

01100	01100	204
10110	00110	- 153
11	1←11	1 carry
00011	10011	
0011	0011	51

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	← 0	carry
01010	00001	
1010	0001	161

Modular Montgomery Multiplier (MMM)

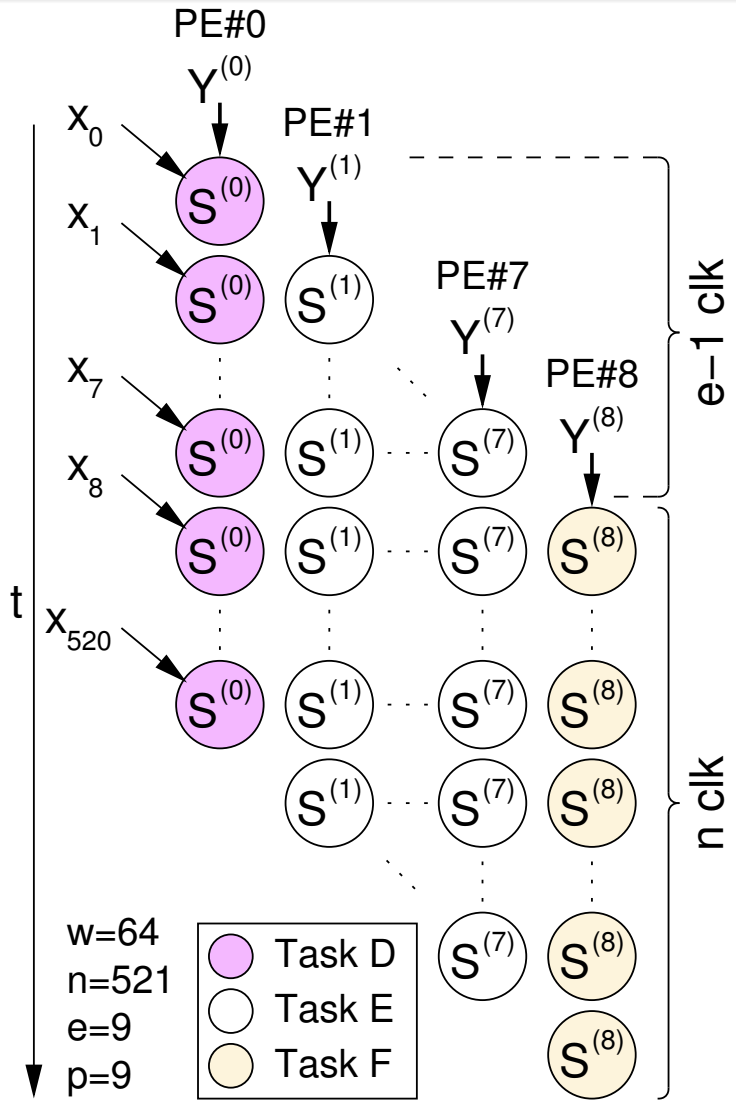
Optimized MWR2MM [4]

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

Task E (lines 4–12)
Task F (line 14)
Task D (lines 1–14)

[4] 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 (based on architecture 2 [4])



$$\text{number of words } e = \left\lceil \frac{n}{w} \right\rceil$$

$$\text{number of processing elements } p = e$$

$$\text{number of clock cycles } T = n + (e - 1)$$

Example: $w = 64, n = 521$

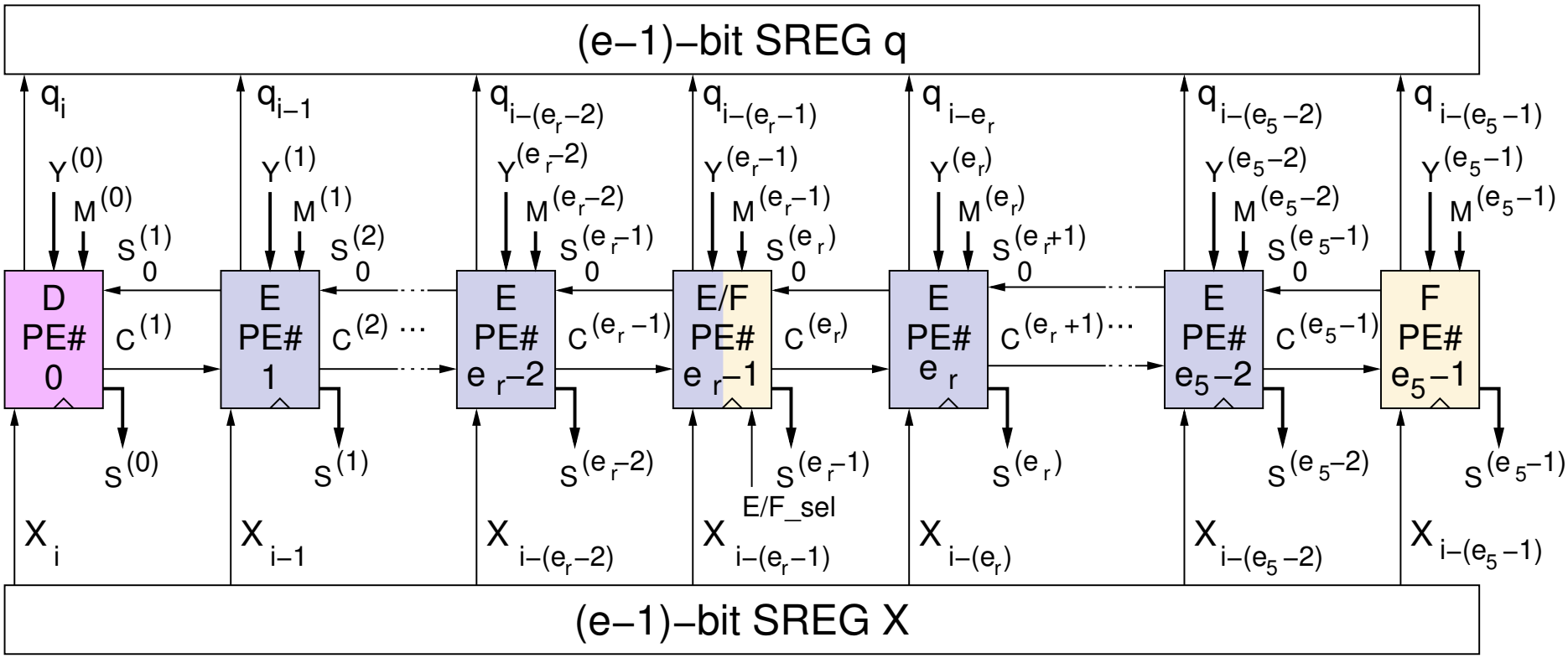
$e = \left\lceil \frac{521}{64} \right\rceil = 9 \quad T = 521 + (9 - 1) = 529$

Example: $w = 32, n = 521$

$e = \left\lceil \frac{521}{32} \right\rceil = 17 \quad T = 521 + (17 - 1) = 537$

$w/2 \Rightarrow T$ minimal bigger
$\Rightarrow 2e \Rightarrow 2p$
$\Rightarrow$ but each PE processes bits/2
$\Rightarrow$ area similar

High-Speed Design: supporting different field sizes



$w = 16, 32 \text{ or } 64 \qquad e_1 = \left\lceil \frac{192}{w} \right\rceil \quad e_2 = \left\lceil \frac{224}{w} \right\rceil \quad e_3 = \left\lceil \frac{256}{w} \right\rceil \quad e_4 = \left\lceil \frac{384}{w} \right\rceil \quad e_5 = \left\lceil \frac{521}{w} \right\rceil$

Lightweight Design (based on architecture 1 [4])

- Each PE can perform E, D, and F
- $p < e \Rightarrow$ we must store inbetween values in a queue.

$$\text{size of queue } Q = e - p$$

$$T = n + \left\lceil \frac{n}{p} \right\rceil \cdot (e - p) + p + 1$$

Example: $p = 8, w = 16, n = 521$

$e = 33, \quad T = 2180, \quad Q = 25$

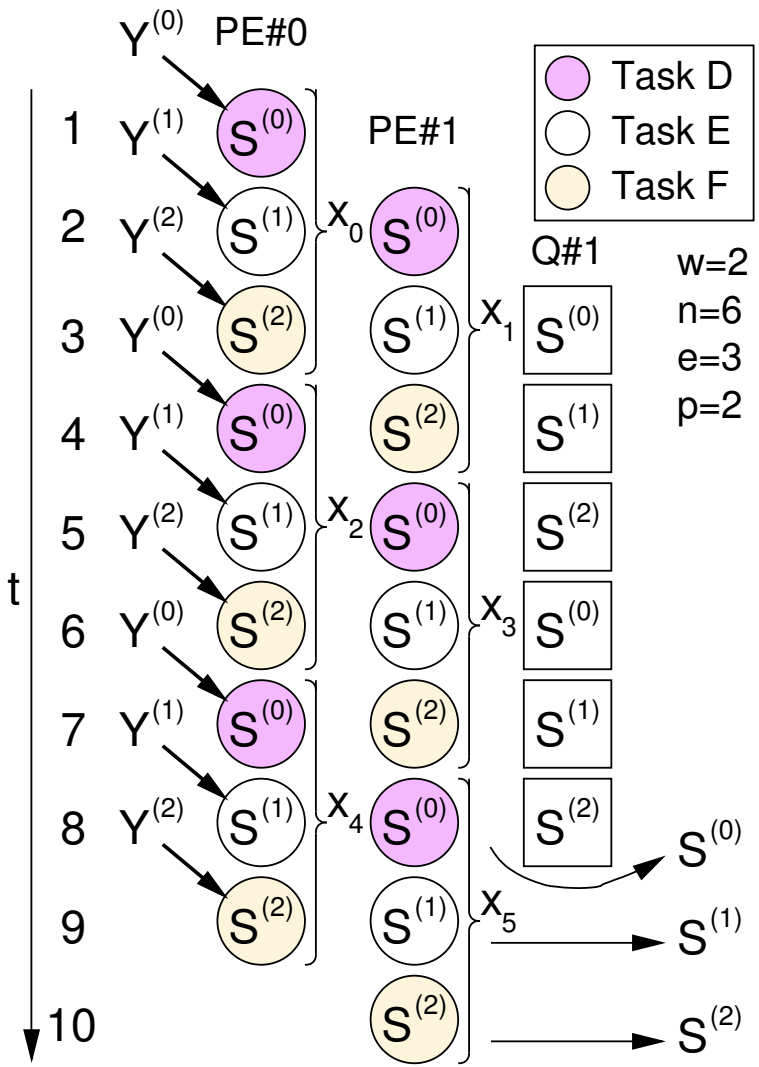
Example: $p = 4, w = 16, n = 521$

$e = 33, \quad T = 4325, \quad Q = 29$

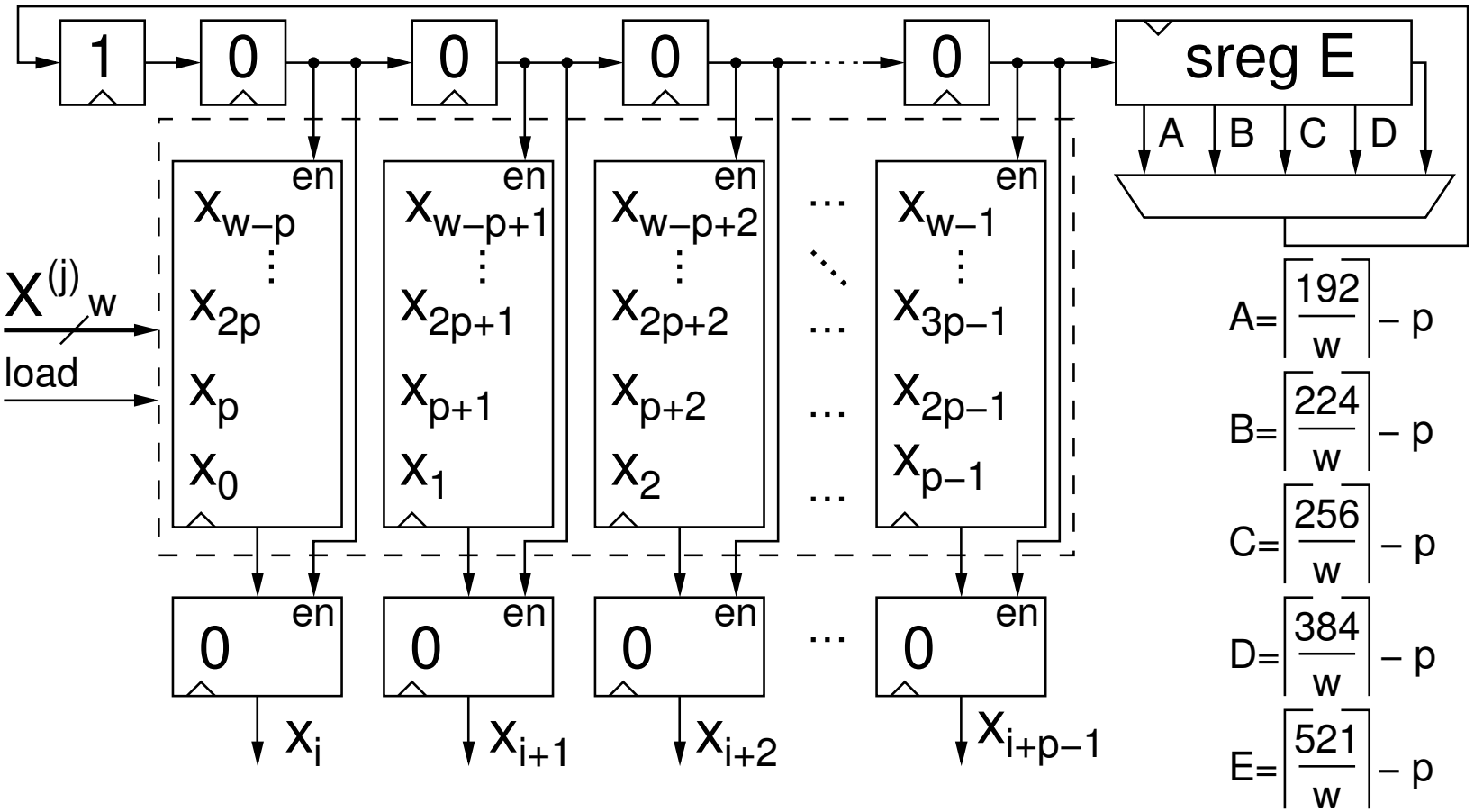
$p/2 \Rightarrow > 2T$

$\Rightarrow Q$ increases a bit

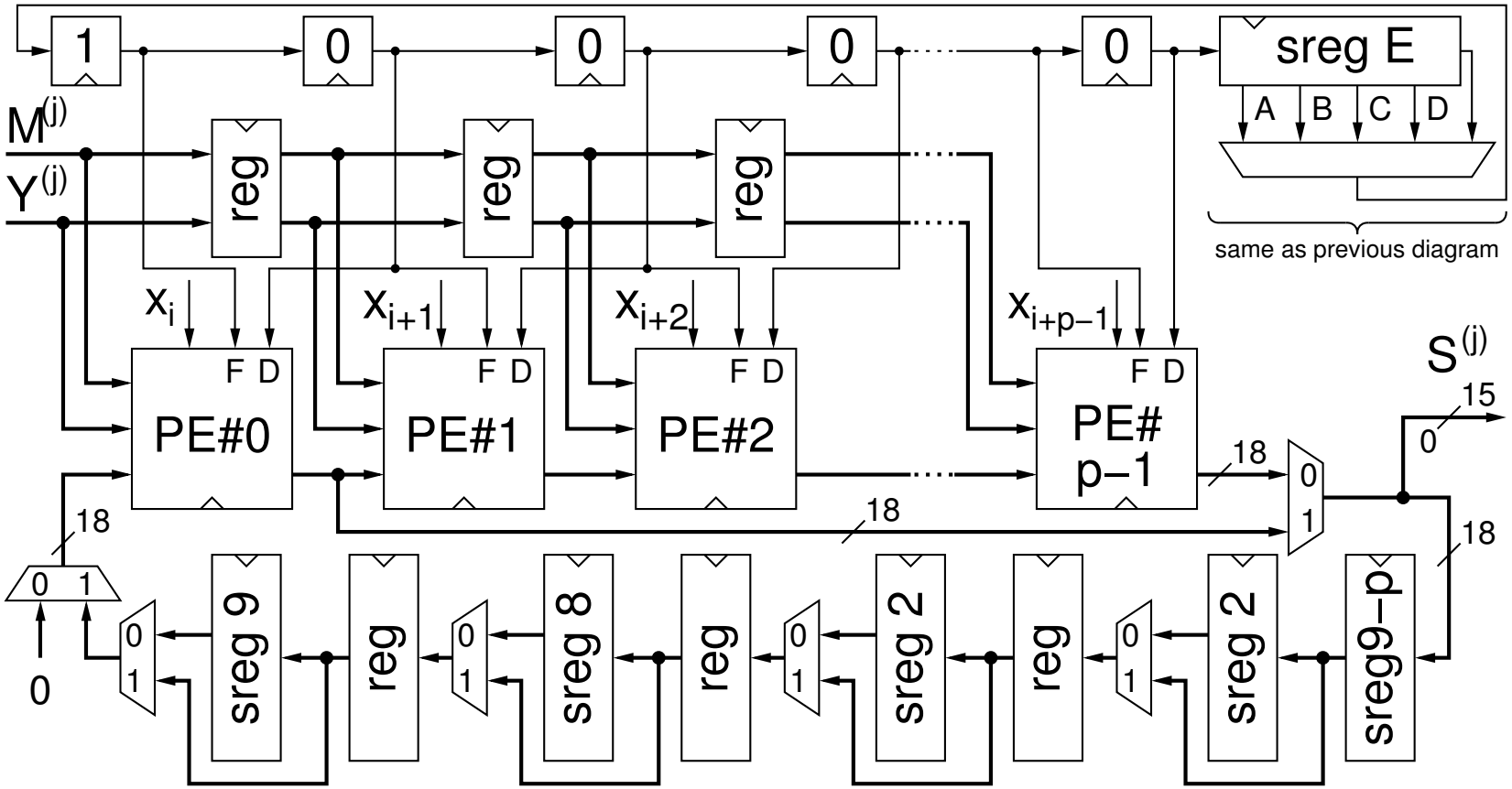
$\Rightarrow$ area/2 for PE



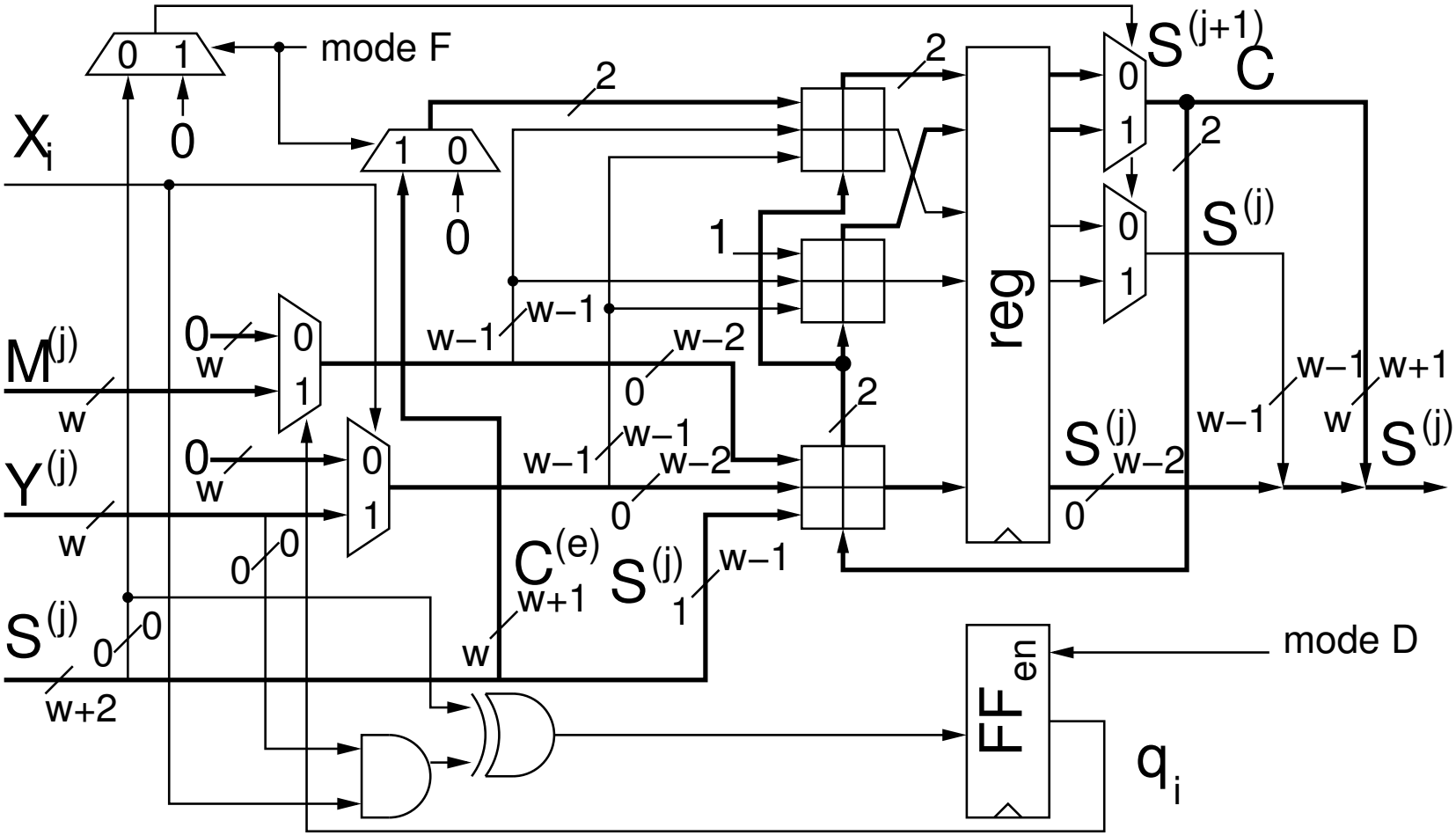
Reading and Reformatting x_i



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, Quartus Prime 16.0 , and Libero SoC 11.7 and Optimized using ATHENa.
- All results are Post-Place & Route.

Xilinx		Altera		Microsemi	
Family	**Tech**	**Family**	**Tech**	**Family**	**Tech**
Spartan-6	45 nm				
Virtex-6	40 nm	Stratix IV	40 nm	IGLOO2	65 nm
Artix-7	28 nm	Cyclone V	28 nm		
Virtex-7	28 nm	Stratix V	28 nm		
Zynq-7000	28 nm				

Latency & Throughput for a given field and width for HS

	Field size	Latency in clock cycles Size of word (W)			TP in Op/sec at f=100 MHz		
		W=64	W=32	W=16	W=64	W=32	W=16
Unprotected	192-bit	766,476	868,229	1,071,735	130	115	93
	224-bit	1,045,978	1,164,717	1,431,804	96	86	70
	256-bit	1,296,709	1,462,177	1,793,113	77	68	56
	384-bit	2,910,581	3,263,447	3,969,182	34	31	25
	521-bit	5,398,490	6,017,514	7,255,754	19	17	14
	Average	**2,283,646**	**2,555,216**	**3,104,317**	**71**	**63**	**52**
Protected	192-bit	824,212	939,969	1,171,483	121	106	85
	224-bit	1,125,214	1,260,229	1,564,664	89	79	64
	256-bit	1,395,369	1,584,853	1,963,821	72	63	51
	384-bit	3,123,729	3,528,699	4,338,639	32	28	23
	521-bit	5,791,134	6,502,510	7,925,454	17	15	13
	Average	**2,451,932**	**2,763,252**	**3,392,812**	**66**	**59**	**47**

TP$\rightarrow$ Throughput; Op$\rightarrow$ Operations; f $\rightarrow$ Frequency

- Average TPs are based on average latencies.

Implementation results of HS designs on Xilinx FPGAs

Width	Slices	LUTs	FFs	BRAMs	Avg. Latency [*Clock cycles*]	f [MHz]	TP $[\frac{Op}{sec}]$	TP/Area $[\frac{Op}{Slices \cdot sec}]$
Virtex7:xc7vx485tffg1761-3								
64	1,269	3,036	4,678	8	2,451,931	184	75	0.059
32	1,227	2,681	4,572	4	2,763,252	214	77	0.063
16	996	3,144	4,606	2	3,392,812	243	72	0.072
Virtex6:xc6vlx240tff1156-3								
64	1,316	3,041	4,678	8	2,451,931	175	71	0.054
32	1,110	2,886	4,572	4	2,763,252	212	77	0.069
16	1,174	2,840	4,606	2	3,392,812	239	70	0.060
Zynq:xc7z020clg484-3								
64	1,135	3,072	4,678	8	2,451,931	130	53	0.047
32	1,172	2,720	4,572	4	2,763,252	168	61	0.052
16	1,224	2,765	4,606	2	3,392,812	178	52	0.043

TP is calculated using the average latency at maximum frequency

Implementation results of HS designs on Altera FPGAs

Width	ALMs	ALUTs	FFs	MBits	Avg. Latency [*Clock cycles*]	f [MHz]	TP $[\frac{Op}{sec}]$	TP/Area $[\frac{Op}{ALMs \cdot sec}]$
Stratix V:5SGXEA7K2F40C3								
64	2,998	5,660	5,383	20,480	2,451,931	173	70	0.023
32	2,766	5,076	5,113	20,480	2,763,252	212	76	0.028
16	2,629	4,841	5,084	20,480	3,392,812	237	70	0.026
Stratix IV:EP4SE530H35C4								
64	4,312	3,936	4,687	20,480	2,451,931	134	55	0.013
32	3,731	4,076	4,582	20,480	2,763,252	176	64	0.017
16	3,681	4,031	4,617	20,480	3,392,812	191	56	0.015

TP is calculated using the average latency at maximum frequency

Implementation results of HS designs on Microsemi FPGAs

Width	LEs	4LUTs	FFs	RAM	Avg. Latency [*Clock cycles*]	f [MHz]	TP $[\frac{Op}{sec}]$	TP/Area $[\frac{Op}{ALMs \cdot sec}]$
IGLOO2:M2GL010TS-1FG484								
64	7,846	6,737	5,226	16	2,451,931	75	30	0.004
32	7,017	6,007	4,962	12	2,763,252	84	30	0.004
16	6,839	5,927	4,926	10	3,392,812	93	27	0.004

TP is calculated using the average latency at maximum frequency

Implementation results of HS designs on ASICs

Width	Area GEs	Avg. Latency [*Clock cycles*]	f [MHz]	TP $[\frac{Op}{sec}]$	TP/Area $[\frac{Op}{GEs \cdot sec}]$
64	57,669	2,198,857	246	112.029	0.0019
32	52,468	2,386,935	284	119.019	0.0023
16	50,602	2,770,886	307	110.705	0.0022

TP is calculated using the average latency at maximum frequency

- Implemented on 90nm technology
- Area is counted in terms of NAND2x1 gates
- Design area does not include size of memories
- Results are after synthesis

Power Estimates for HS using Xpower and Libero

Family	Avail. LUTs	P_{static} [mW] W=64	W=32	W=16	$P_{dynamic}$ [mW] W=64	W=32	W=16	P_{total} [mW] W=64	W=32	W=16
VX7	303,600	241	241	241	52	36	30	293	278	271
VX6	150,720	3,424	3,423	3,423	86	45	54	3,510	3,468	3,477
ZQ	53,200	100	100	100	51	40	31	164	140	144
AX7	63,400	82	82	82	47	40	39	129	122	121
SN6	9,112	20	20	20	36	24	26	56	44	46
IG2	12,084	13	12	11	1	1	0.3	14	13	11

FY→Family; VX→Virtex; SN→Spartan; AX→Artix; ZQ→Zynq; IG→IGLOO

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 HS results

Work	Device	Curve		Slices	LUTs	DSPs	BRAMs	f [MHz]	TP $[\frac{Op}{sec}]$	TP/Area $[\frac{Op}{LUTs \cdot sec}]$
		Size	Type							
TW[W=32] (Protected)	VX-7	192	$GF(p)$	1,227	2,681	0	4	214	247	0.090
		224	$GF(p)$						187	0.069
		256	$GF(p)$						146	0.054
		384	$GF(p)$						67	0.024
		521	$GF(p)$						37	0.013
TW[W=64] (Protected)	VX-7	192	$GF(p)$	1,269	3,036	0	8	184	239	0.078
		224	$GF(p)$						177	0.058
		256	$GF(p)$						142	0.046
		384	$GF(p)$						63	0.020
		521	$GF(p)$						35	0.011
Amiet *et. al*[W=32] (Unprotected)	VX-7	192	$GF(p)$	N/A	6,816	20	N/A	N/A	3,260	0.478
		256	$GF(p)$						1,510	0.221
		384	$GF(p)$						551	0.080
		521	$GF(p)$						231	0.033
Amiet *et al.*[W=64] (Unprotected)	VX-7	384	$GF(p)$	N/A	8,273	64	N/A	N/A	759	0.091
		521	$GF(p)$						320	0.038
Alrimeih *et al.* (N/A)	VX-6	192	p-192	11,200	32,900	289	128	100	3,334	0.101
		224	p-224						2,858	0.086
		256	p-256						2,500	0.075
		384	p-384						848	0.025
		521	p-521						625	0.018
Roy *et al.*	VX-5	256	p-256	81	212	8	22	172	91	0.429
Baldwin *et al.*	VX-5	192	$GF(p)$	N/A	6,100	N/A	N/A	97	488	0.080
		256	$GF(p)$	N/A	7,800	N/A	N/A	82	248	0.031

TW→This Work; VX→ Virtex; $GF(p)$ → any prime for a given size; p-xxx → NIST prime only

Other results

Work	Device	Curve		Slices	LUTs	DSPs	BRAMs	f [MHz]	TP $[\frac{Op}{sec}]$	TP/Area $[\frac{Op}{Slices \cdot sec}]$
		Size	Type							
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[1]		96		157	1,471	0.160[2]
Schiniakis *et al.*	SX-II	192	$GF(p)$	6,200[1]		92		161	2,273	0.367[2]
		256	$GF(p)$	9,200[1]		96		157	1,471	0.160[2]
		384	$GF(p)$	13,000[1]		177		151	741	0.057[2]
		521	$GF(p)$	17,000[1]		244		145	449	0.026[2]

VX→ Virtex; SX→Stratix; [1] → ALMs; [2] → $[\frac{Op}{ALMs \cdot sec}]$

Latency & Throughput for a given field and no. of PE for LW

	Field size	Latency in clock cycles Number of PE units (#PE)			TP in Op/sec at f=100 MHz		
		#PE=8	#PE=4	#PE=2	#PE=8	#PE=4	#PE=2
Unprotected	192-bit	1,477,451	2,400,451	4,265,951	68	42	23
	224-bit	2,212,011	3,684,471	6,652,161	45	27	15
	256-bit	3,073,655	5,213,351	9,518,015	33	19	11
	384-bit	9,453,251	16,857,851	31,705,751	11	6	3
	521-bit	22,890,391	41,810,938	79,687,348	4	2	1
	Average	**7,821,351**	**13,993,412**	**26,365,845**	**32**	**19**	**11**
Protected	192-bit	1,576,957	2,557,883	4,540,489	63	39	22
	224-bit	2,358,523	3,922,551	7,074,793	42	25	14
	256-bit	3,279,973	5,555,813	10,134,373	30	18	10
	384-bit	10,057,513	17,916,721	33,676,213	10	6	3
	521-bit	24,313,723	44,374,347	84,533,037	4	2	1
	Average	**8,317,338**	**14,865,465**	**27,991,781**	**30**	**18**	**10**

TP$\rightarrow$ Throughput; Op$\rightarrow$ Operations; f $\rightarrow$ Frequency

Implementation results of LW designs on Xilinx FPGAs

# of PEs	Slices	LUTs	FFs	BRAMs	Avg. Latency [*Clock cycles*]	f [MHz]	TP $[\frac{Op}{sec}]$	TP/Area $[\frac{Op}{Slices \cdot sec}]$
Zynq:xc7z020clg484-3								
8	469	1,697	1,169	2	8,317,338	179	21	0.046
4	407	1,353	1,015	2	14,865,465	182	12	0.030
2	318	1,118	939	2	27,991,781	164	6	0.020
Artix:xc7a100tcsg324-3								
8	527	1,675	1,169	2	8,317,338	186	22	0.042
4	466	1,310	1,015	2	14,865,465	187	13	0.027
2	312	1,135	939	2	27,991,781	187	7	0.021
Spartan6:xc7vx485tffg1761-3								
8	466	1,758	1,178	2	8,317,338	152	18	0.039
4	371	1,360	1,024	2	14,865,465	143	10	0.026
2	325	1,166	948	2	27,991,781	155	6	0.017

TP is calculated using the average latency at maximum frequency

Implementation results of LW designs on Altera FPGAs

# of PEs	ALMs	ALUTs	FFs	MBits	Avg. Latency [*Clock cycles*]	f [MHz]	TP $[\frac{Op}{sec}]$	TP/Area $[\frac{Op}{ALMs \cdot sec}]$
Stratix V:5SGXEA7K2F40C3								
8	883	1,501	1,222	20,480	8,317,338	224	27	0.030
4	676	1,162	746	20,810	14,865,465	216	14	0.021
2	588	961	647	20,480	27,991,781	224	8	0.014
Cyclone V:5CEBA4F23C7								
8	858	1,517	937	20,786	8,317,338	121	14	0.017
4	680	1,162	692	20,875	14,865,465	122	8	0.012
2	588	961	571	20,912	27,991,781	119	4	0.007
Stratix IV:EP4SE530H35C4								
8	1,007	1,554	974	20,480	8,317,338	182	22	0.022
4	835	1,216	785	20,480	14,865,465	181	12	0.015
2	899	933	952	20,480	27,991,781	185	7	0.009

TP is calculated using the average latency at maximum frequency

Implementation results of LW designs on Microsemi FPGAs

# of PEs	LEs	4LUTs	FFs	RAM	Avg. Latency [*Clock cycles*]	f [MHz]	TP $[\frac{Op}{sec}]$	TP/Area $[\frac{Op}{ALMs \cdot sec}]$
IGLOO2-M2GL005S:1FG484								
8	3,205	2,835	1,491	10	2,451,931	94	11	0.004
4	2,753	2,385	1,341	10	2,763,252	81	5	0.002
2	2,522	2,079	1,269	10	3,392,812	97	3	0.001

TP is calculated using the average latency at maximum frequency

Implementation results of LW designs on ASICs

Width	**Area** *GEs*	**Avg. Latency** [*Clock cycles*]	f [MHz]	**TP** $[\frac{Op}{sec}]$	**TP/Area** $[\frac{Op}{GEs \cdot sec}]$
8	12,696	13,273,732	301	23	0.0018
4	6,402	23,832,507	307	13	0.0020
2	6,402	44,998,584	306	7	0.0011

TP is calculated using the average latency at maximum frequency

- Implemented on 90*nm* technology
- Area is counted in terms of NAND2x1 gates
- Design area does not include size of memories
- Results are after synthesis

Power Estimates for LW using Xpower and Libero

Family	Avail. LUTs	P_{static} [mW]			$P_{dynamic}$ [mW]			P_{total} [mW]		
		PE=8	PE=4	PE=2	PE=8	PE=4	PE=2	PE=8	PE=4	PE=2
ZQ	53,200	100	100	100	31	18	12	131	118	112
AX7	63,400	82	82	82	39	22	17	121	104	99
SN6	9,112	20	20	20	19	8	6	29	28	26
IG2	6,060	11	12	11	1	0.4	1	12	12	12

FY→Family; SN→Spartan; AX→Artix; ZQ→Zynq; IG→IGLOO

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.

Comparison of LW results

Work	Device	Curve		Slices	LUTs	DSPs	BRAMs	f [MHz]	TP [$\frac{Op}{sec}$]	TP/Area [$\frac{Op}{Slices \cdot sec}$]
		Size	Type							
TW[#PEs=8]	SN-6	192	$GF(p)$	481	1,513	0	2	165	156	0.325
		224	$GF(p)$						100	0.207
		256	$GF(p)$						67	0.139
		384	$GF(p)$						20	0.041
		521	$GF(p)$						7	0.015
Driessen *et al.*	SN-6	256	$p-256$	221	630	1	3	N/A	N/A	N/A
Roy*et al.*	SN-6	256	$p-256$	72	193	8	24	156	82	1.139
	VX-5	256	$p-256$	81	212	8	22	172	91	1.123
Varchola *et al.*	VX-2 Pro	224	$p-224$	773	N/A	1^1	3	210	122	0.158
	VX-2 Pro	256	$p-256$	773	N/A	4^1	3	210	100	0.129
Vliegen *et al.*	VX-2 Pro	256	$GF(p)$	1694	N/A	2^1	9	108	34	0.020

TW→This Work; VX→ Virtex; SN→ Spartan; ^{1} → Multipliers;

$GF(p)$ → any prime for a given size; p-xxx → NIST prime only

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 Amiet, however, we use only one MMM, a fraction of the BRAMs, no DSP units neither contribute to TP/Area and our design is protected.
- In case of lighweight, our design is about 6$\times$ larger as compared to Roy's design. But we do not use any DSP units and only two BRAMs.

Thanks for your attention.