Hardware Implementation of Elliptic Curve Cryptography

Debapriya Basu Roy

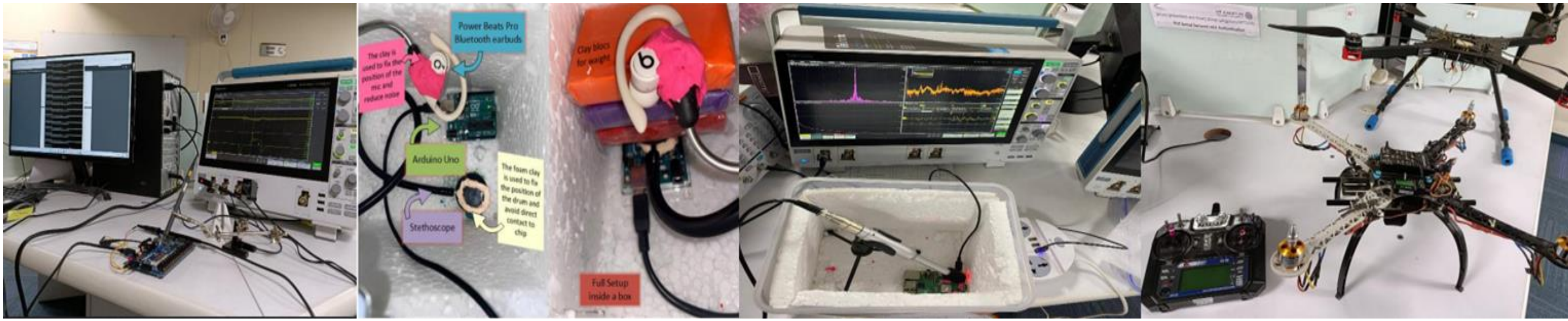
Department of Computer Science & Engineering

Indian Institute of Technology Kanpur

dbroy@cse.iitk.ac.in

dbroy24@gmail.com

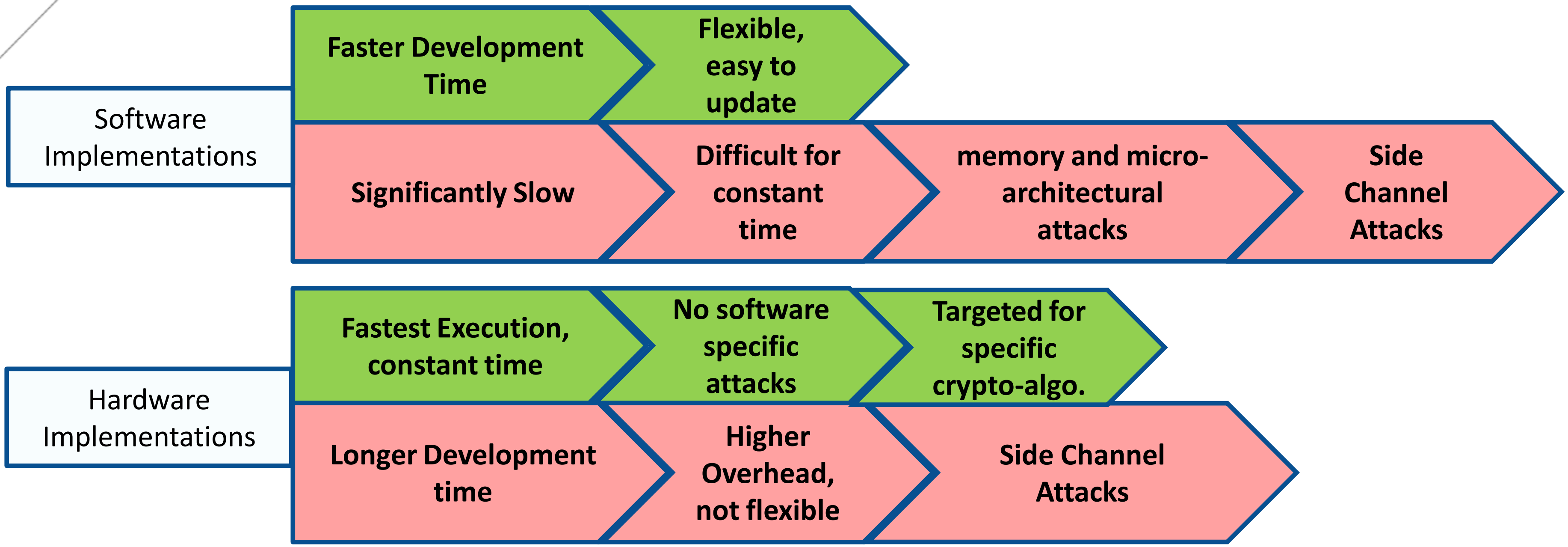
Secure Embedded and Smart Things Laboratory (SETTLOR)
First Floor, C3i Center Building, IIT Kanpur-208016
Laboratory Website: https://cse.iitk.ac.in/users/urbic/research/



Publications:
Conferences: DATE, DAC, IEEE HOST, ESWEEK, VLSID, SPACE, AsianHOST, GLSVLSI.
Journals: IEEE ESL, ACM TECS, Springer JCEN, IEEE TCAS-1.

Research Areas:
Approximate Computing, Acoustic Side Channel Attacks, Physically Unclonable Functions, Timing Attacks on Network-on-Chip, Post-Quntum Cryptography

Hardware Implementation: Why it is Important?



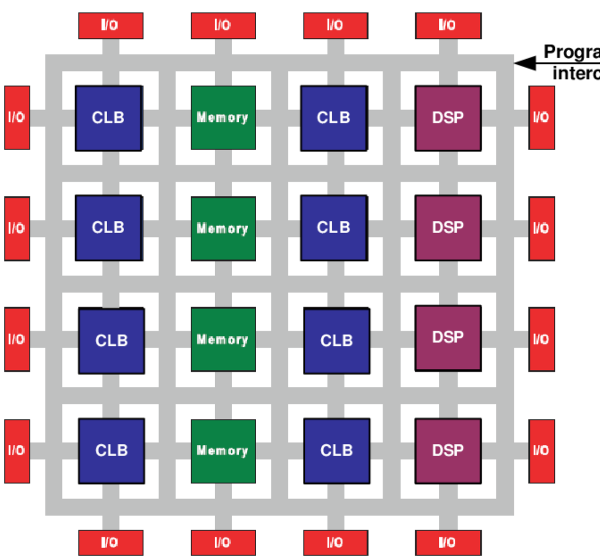
Cryptographic Algorithms: Complex and computationally intensive mathematical functions
Software Implementations: Slow, may create speed bottleneck during the executions of crypto-algorithms
Software specific attacks: Buffer overflow attack, Spectre, Meltdown!!
Hardware: Dedicated architecture for cryptographic algorithms, fast, efficient, but not flexible.

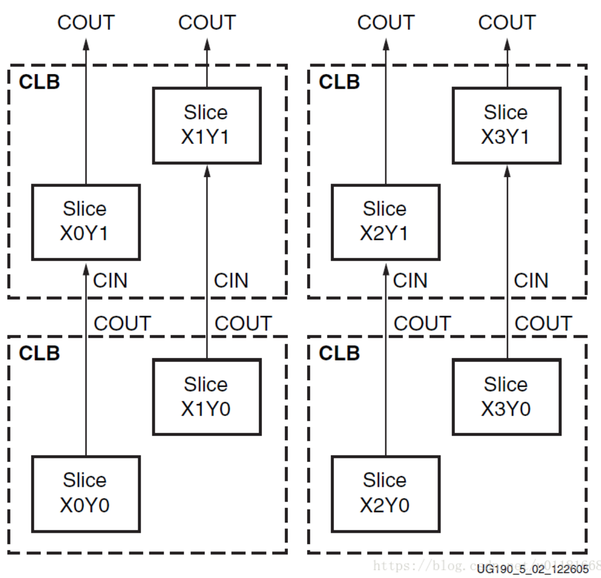
- **Hardware-Software Codesign: Accelerating crypto-algorithms by offloading a portion of the computation to hardware.**
- **Combines flexibility of software+effciency of hardware, generally done by instruction set extension.**
- **Example: AES-NI, PCLMULQDQ Instructions on Intel for accelerating AES and Elliptic Curve Operations.**

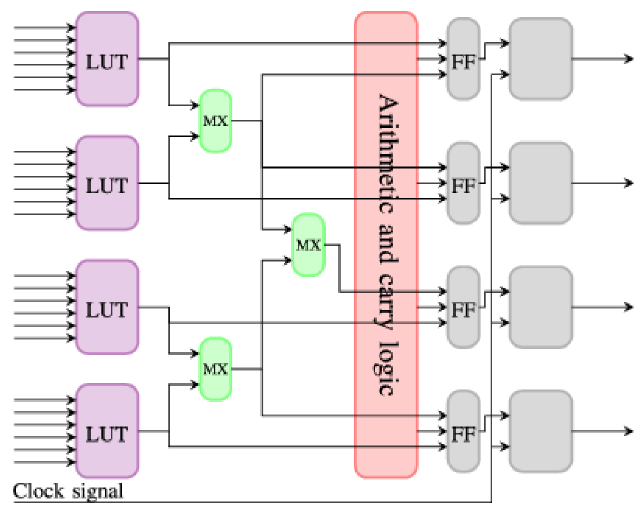
Field Programmable Gate Array (FPGA)

- ASIC Design: Fast and dedicated architecture for the target application
- Expensive and time consuming (typically one need to wait around 5-6 months to get the final chip after layout is finalized)
- Any error in the design will require reiteration of this long procedure

- FPGA: Islands of Programmable logic block in the sea of programmable reconnect



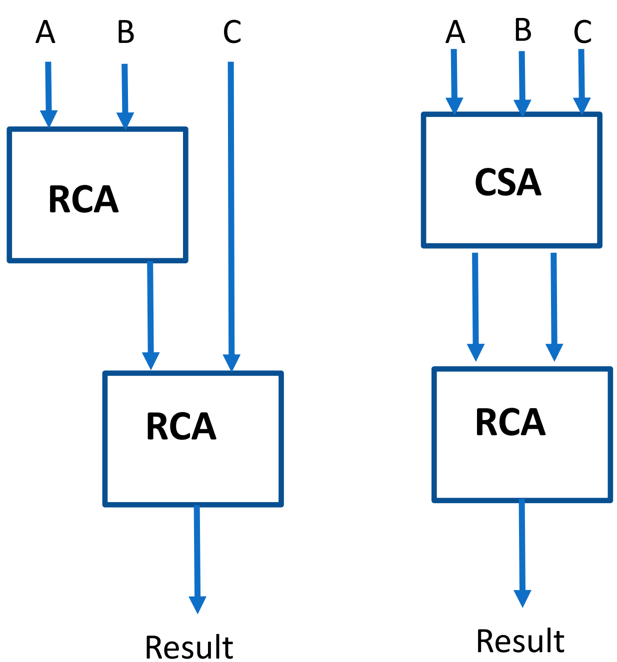




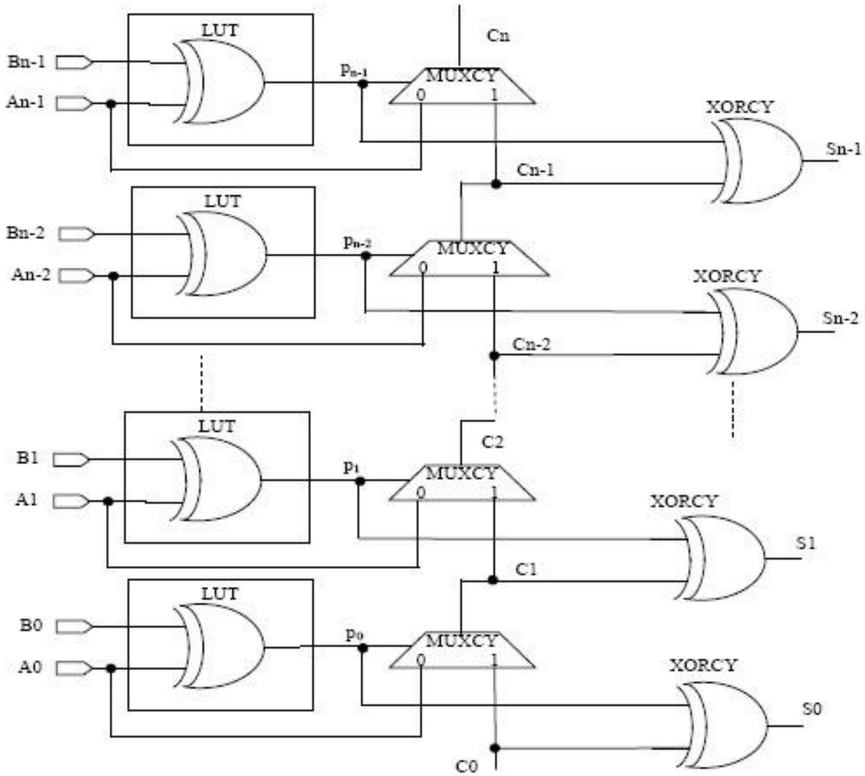
FPGA Advantages

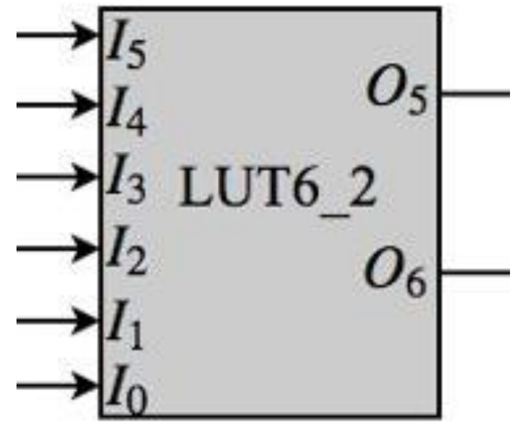
- Modern FPGAs: Equipped with hardware-IPs (hard-IPs: DSP blocks and) to reduce the performance gap between FPGA and ASIC Designs
- Faster development time than ASIC, in house security for crypto algorithms
- Modern processor cores like ARM now being integrated with FPGA to take advantage of the speed gain of FPGA architectures (Xilinx Zedboard)
- FPGA inside CPU: Possibility as Intel has recently bought FPGA company Altera

Developing FPGA architecture is not just writing HDL codes



FPGA Hard-IP: Carry Chain and LUTs





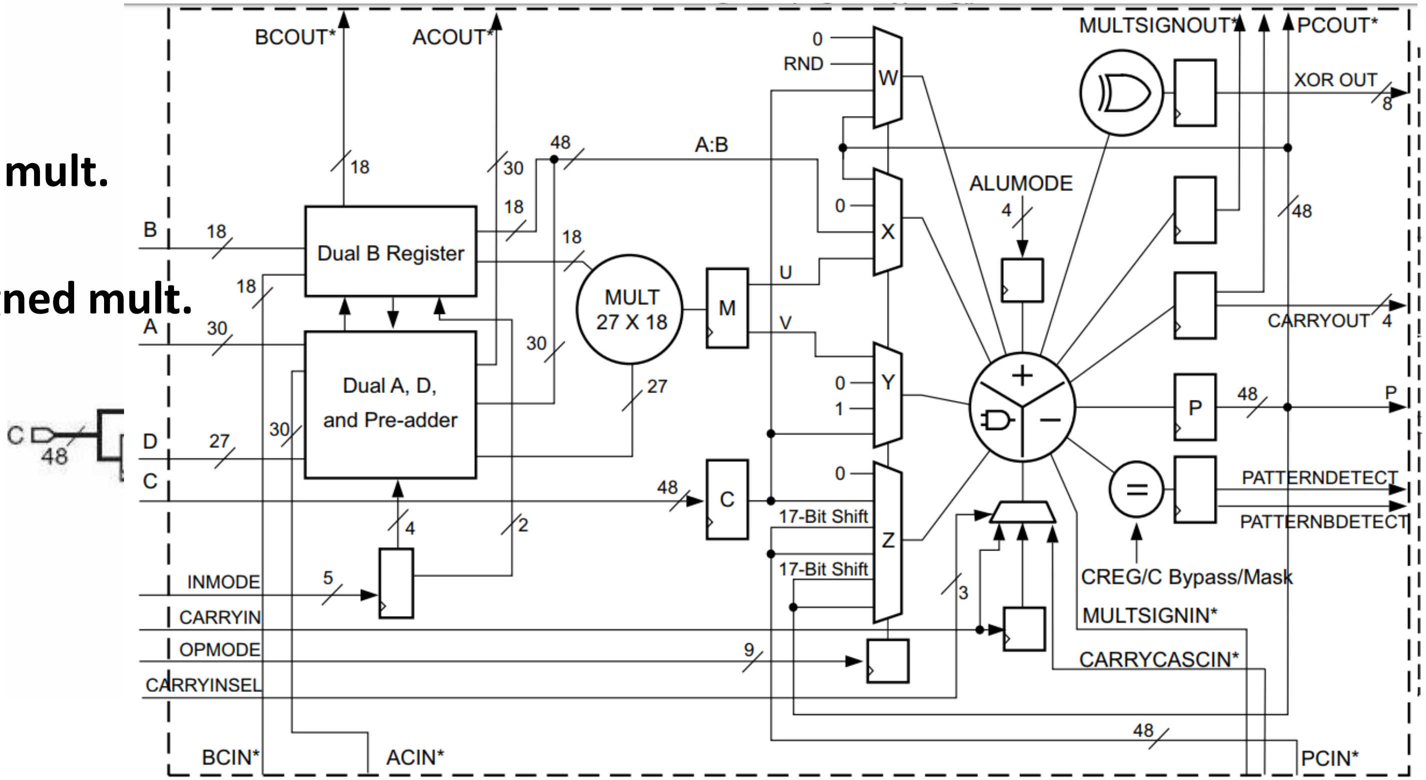
Carry4 : Dedicated fast routing for carry propagation --> Reason behind RCA faster than CSA
LUT6_2: Implement any 6 input, 1 output function or any 5 unput 2 output function

DSP Blocks: Evolution

Virtex-4:18 x18 signed mult.

Virtex-5,6,7: 25 x18 signed mult.

Virtex 7 Ultrascale: 27 x18 signed mult.



Our objective will be to device optimal architecture of field multipliers using asymmetric integer multipliers of these DSP blocks

Assignment 1

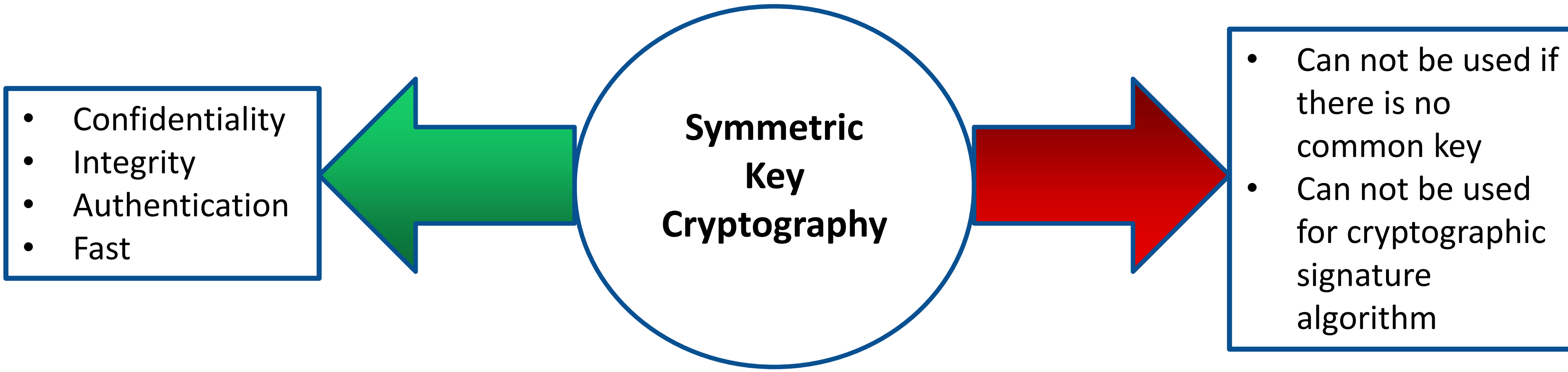
- Desgin a Finite Field Adder in FPGA
- Desgin a Finite Field Multiplier in FPGA

Next: Public Key Cryptography

- Shortcoming of Symmetric Key Cryptography
- Introduction to Public Key Cryptography
- Elliptic Curve Cryptography

Symmetric Key Cryptography

- Till now in the lecture we have learnt about block cipher and stream ciphers
- They are part of broad category of cryptographic algorithms known as symmetric key cryptography

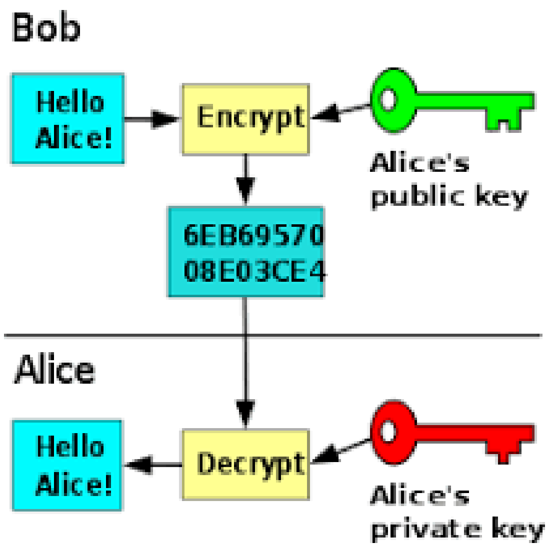


- Block ciphers are generally built using SPN (substitution permutation network) architecture (like PRESENT) or Fiestel architecture (like block cipher CLEFIA)
- Stream ciphers are based on mostly LFSRs and NFSRs
- **Now the challenge is how we can make sure that the two communicating party have a common key??**

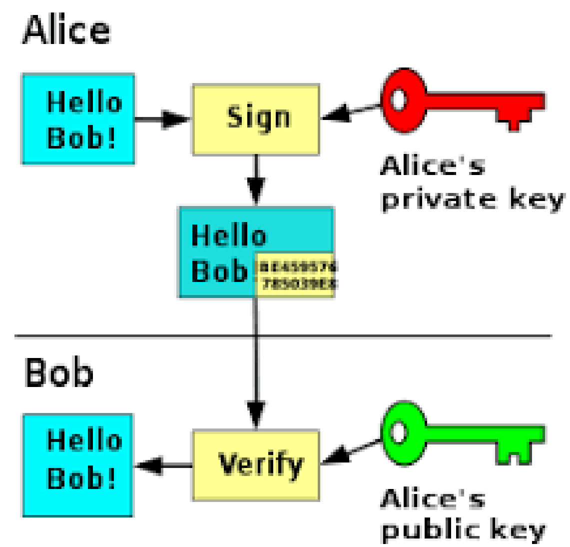
Asymmetric Key Cryptography/ Public Key Cryptography

- Asymmetric key cryptography is used to share the key between two party
- In this case, each communicating party has a pair of key (private key, public key)
- private key is secret, and public key is known to everyone

Encryption



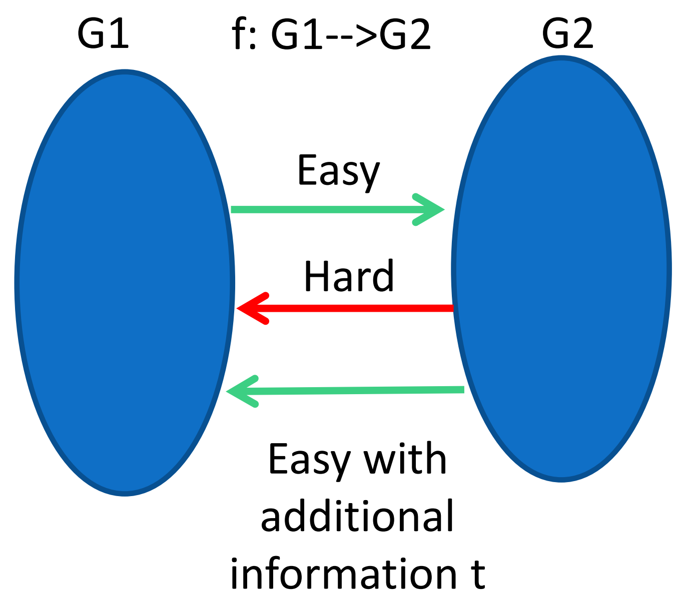
Digital Signature



Picture Source: Wikipedia

Asymmetric Key Cryptography/ Public Key Cryptography

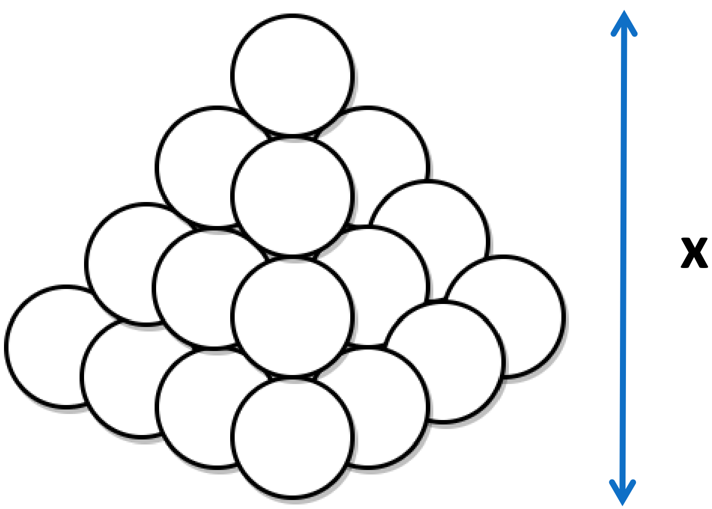
- Public key cryptographic algorithms are generally based on some computationally hard mathematical problem known an trapdoor oneway function.
- Public key : transform values from G1 to G2
- Without t (private key) we can not do the inverse operation.



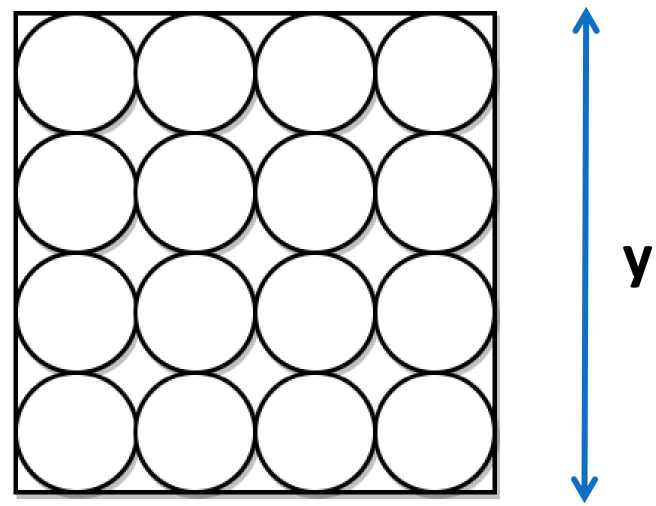
- Example of oneway function with trapdoor:
  - Exponentiation of a number mod N
    - N =p x q , p and Q are prime
  - Elliptic Curve Scalar Multiplication

Cannonball Problem

- We want to place cannonballs in a square pyramid. Square pyramid is a structure where the i^{th} layer contains i^2 cannonballs.



Square Pyramid



Square Array

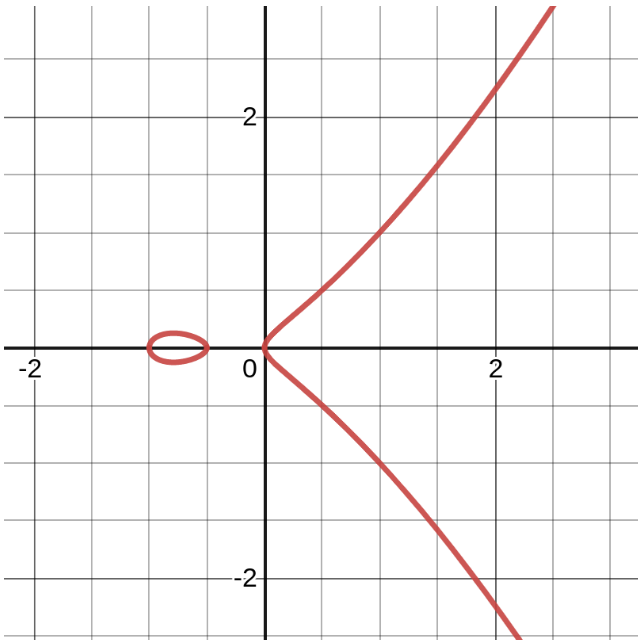
Question: Find out the number of cannonballs for which the square pyramid and square array will have same number of cannonballs (apart from 0 and 1)

Solution

$$y^2 = 1^2 + 2^2 + \cdots + x^2$$

$$y^2 = \frac{x(x+1)(2x+1)}{6}$$

- By plotting this equation what we get is an elliptic curve (nothing to do with an ellipse),
- Feature: symmetric with two distict lobe
- If (x,y) is on the curve, so is (x,-y)
- (0,0) and (1,1) are two points on the curve



How to find other points on the curve?

- (0,0) and (1,1) are on the curve, line y=x intersect the curve on point (0,0) and (1,1).
- However, as the elliptic curve equation is cubic in term of x, it should intersect the curve on another point
- Substituting y=x in the elliptic curve equation, we get $x^3 - \frac{3}{2}x^2 + \frac{1}{2}x = 0$
- From theory of equations, $x_1+x_2+x_3=3/2$, where x_1, x_2, x_3 are the root of the equation
- $x_1=0$ and $x_2=1$, then $x_3=1/2$
- The corresponding y coordinate is 1/2 => (1/2,1/2) is another point on the curve

Elliptic Curve Cont´d:

- (0,0), (1,1), (1/2,1/2) are the points on the curve
- (1,-1), (1/2,-1/2) are two other points on the curve (by symmtry operation)
- We can keep on continuing the same approach to find the other points
- Construct the equation of the line with points (1,1) and (1/2,-1/2) : $y=3x-2$
- Find out the corresponding x and y coordinate using the discussed approach
- In the context of cannonball problem, x is the height of pyramid and y is the dimension of the square array

Elliptic Curve for Cryptography

The usage of Elliptic curve, defined over some finite field, for Diffie-Hellman key exchange was first proposed by Victor Miller and Neal Koblitz in 1985

It is based on the hardness of computing discrete logarithm problem in the Elliptic curve domain

General form of Elliptic Curve

- Weierstrass Equation: An elliptic curve defined over some finite field F can be presented as:

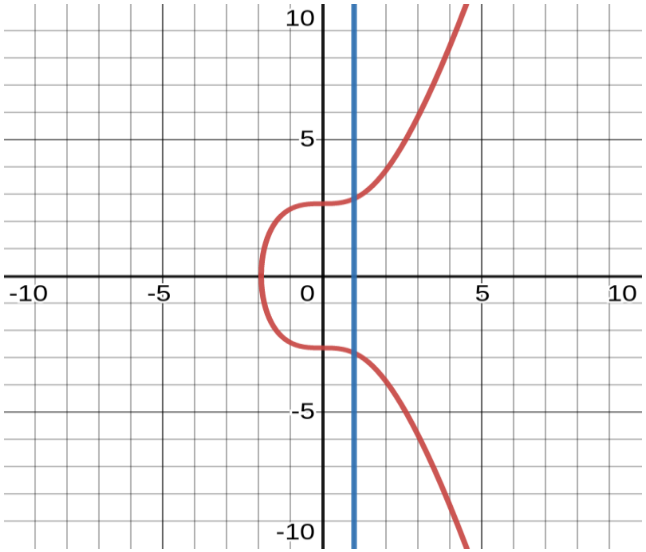
$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6$$

$$a_1, a_2, a_3, a_4, a_6 \in F$$

- Depending upon the underlying finite field, we can simplify the above equation:
  - GF(p): General form --> $y^2 = x^3 + ax + b; a \in GF(p)$ (prime curve)
  - GF(2^m): General form --> $y^2 + xy = x^3 + ax^2 + b; a, b \in GF(2^m)$ (binary curve)
- In this lecture, our concentration would be more on elliptic curve defined over prime fields.
- Apart from these curves, there also exist some special curves:
  - **Montgomery curves**
  - Edware Curves
  - BN Curves
  - Koblitz Curve

Points on Elliptic Curve E defined over field F

- It is a set of points with coordinates (x,y) where each x and y value belongs to the field F $E = \{(x,y), x \in F, y \in F\} \bigcup \{\infty\}$
- Point of Infinity
- Consider the graph: $y^2=x^3+7$ and line x=1
- The graph and and the line should intersect at
  - (1,2.828)
  - (1,-2.828)
  - another point (being a cubic equation)

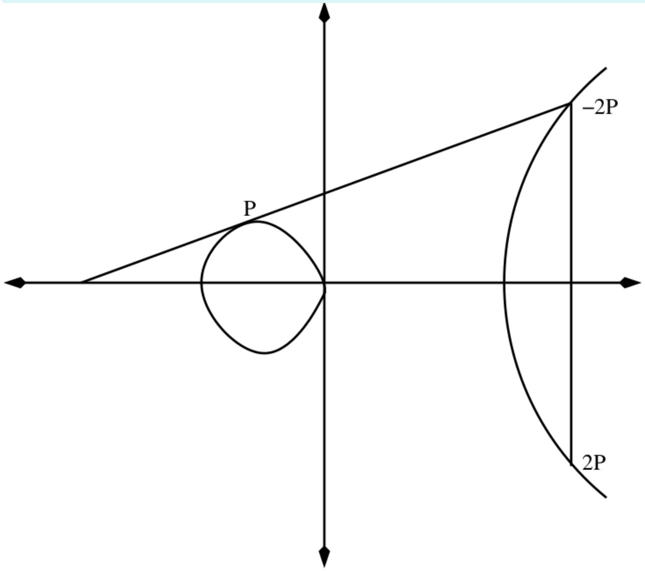


- This another point which does not seem to be on the graph but should be on the curve is the point of infinity (denoted as O)
  - for any point P on the curve: P+O=P (acting as an identity element)
  - for any point P on the curve: P+(-P)=O (acting as an identity element)
- Negative of a point P (x,y) is -P(x,-y)

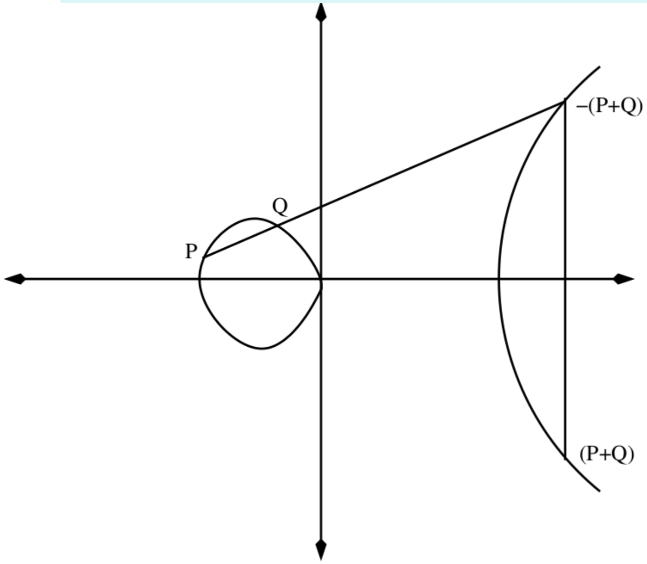
Operation over Elliptic Curve: Point Addition

- Consider the Elliptic curve E defined over some field F. We want to add two point P and Q . This is known as point addition
- For a given point P, we can compute 2P. This is called point doubling

Point Doubling

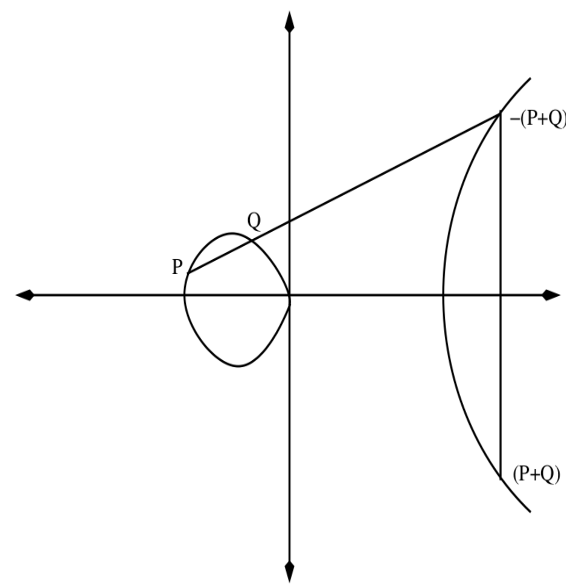


Point Addition



Point Addition Computation

- Two point P (x_1,y_1) and $Q(x_2,y_2)$ defined over curve $y^2=x^3+Ax+B$. Their addition is P+Q (x_3,y_3)
- Slope of the line going through $P(x_1,y_1)$ and $Q(x_2,y_2)$ is $\lambda=(\frac{y_2-y_1}{x_2-x_1})$
- Equation of the line going through these points: $y-y_1=\lambda(x-x_1)$ and this line intersects the curve $y^2=x^3+Ax+B$. We can replace y with $y_1+\lambda(x-x_1)$
- $(y_1+\lambda(x-x_1))^2 = x^3+Ax+B$
- $x^3-\lambda^2x^2+(a+2\lambda^2x_1-2\lambda y_1)x+b-(\lambda x_1-y_1)^2 = 0$
- This equation will have three roots:
  - x_1 --> corresponds to point P
  - x_2 --> corresponds to point Q
  - x_3 --> corresponds to point -(P+Q)
- From the theory of equation
  - $x_1+x_2+x_3=\lambda^2 \Longrightarrow x_3= \lambda^2 -x_1 -x_2$



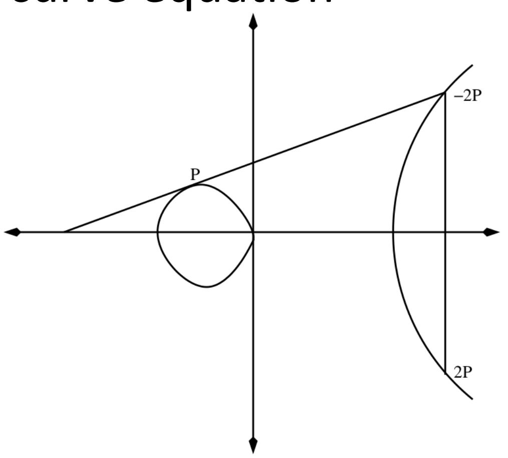
- Simplifying, we get $x_3=(\frac{y_2-y_1}{x_2-x_1})^2 -x_1-x_2$, cost is= 1 squaring+3 subtractions+1 addition +1 inversion+1 multiplication. (all are finite field operations)

Point Addition Continued:

- We can compute the y coordinate value using the equation of the straight line
- $-y_3=y_1+\lambda(x_3-x_1)$ ==> $y_3=\lambda(x_1-x_3)-y_1$ ==> $y_3=(\frac{y_2-y_1}{x_2-x_1})(x_1-x_3)-y_1$
- Cost=4 subtractions+ 2 multiplications+1 inversion (Condition: $x_2 \neq x_1$)
- Therefore total cost=5 subtractions+ 1 Additions+2 Multiplications+**1 Inversion**+1 Squaring

Point Doubling:

- The slope in this case is computed by differentiating the curve equation
- $2y\frac{dy}{dx} = 3x^2 + A$ ==> $\frac{dy}{dx}\Big|_{(x1,\, y1)} = \frac{3x_1^{\,2}+A}{2y_1} = \lambda$
- As this line is a tangent to curve, we can consider that x_1 is actually two solution for the curve equation.



- Therefore $2x_1+x_3=\lambda^2$ ==> $x_3=(\frac{3x_1^{\,2}+A}{2y_1})^2 - 2x_1$
- $y_3=(\frac{3x_1^{\,2}+A}{2y_1})(x_1-x_3)-y_1$
- Again, we require a few additions, subtractions, multiplication and **1 inversion**

Point Addition and Doubling on Binary Curve

- Binary Elliptic Curve: $y^2 + xy = x^3 + ax^2 + b; a, b \in GF(2^m)$

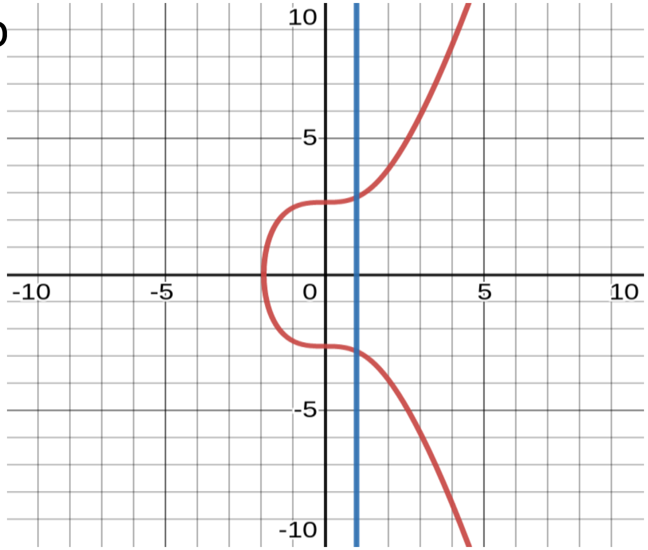
- Point Doubling: Point P (x_1,y_1), Target 2P (x_3,y_3)
  - $x_3=\lambda^2 + \lambda+A$,
  - $y_3=\lambda(x_1+x_3)+x_3y_1$, $\lambda=x_1+y_1/x_1$

- Point Addition: Point P (x_1,y_1) and Q (x_2,y_2), Target P+Q (x_3,y_3)
  - $x_3= \lambda^2+ \lambda + x_1 + x_2 + A$
  - $y_3 = \lambda(x_1+x_3)+x_3y_1$, $\lambda=\frac{y_1+y_2}{x_1+x_2}$

- Negation of Point P (x_1,y_1) --> -P (x_1,x_1+y_1)

Addition between Point of Infinity O and P

- To add with point of infinity, we draw a vertical line going through the point as we have assumed that point of infinity is so
- It will intersect the curve at point -P
- Its projection will be point P itself
- P+O=P



Elliptic Curve as Abelian Group

- P+Q=Q+P (Commutative)
- (P+Q)+R=P+(Q+R) (Associative)
- P+O=(O+P)=P (Existance of additive Identity)
- P+(-P)=O (Existance of additive inverse)

Elliptic Curve Scalar Multiplication

- Consider Elliptic curve E defined over finite field GF(p). The order of the field is n
- Consider a point P on this Elliptic curve E and a random integer k<n (known as scalar)
- Scalar Multiplication: [k]P=P+P+P+... (add point P k times)
- **Elliptic curve discrete logarithm problem:**

For secure Elliptic curve, given points P and [k]P, find out the value of k

- This problem is assumed to be computationally hard. The best possible algorithm which solves this problem is pollard-rho algorithm (on normal comuters).
- Complexity: $O((p)^{1/2})$. For 256 bit prime, the attack complexity will be 2^{128}

Scalar Multiplication Computation

Algorithm 1: Double-and-Add Algorithm

Data: Point P and scalar $k = k_{m-1}, k_{m-2}, k_{m-3} \ldots k_2, k_1, k_0$, where $k_{m-1} = 1$

Result: $Q = kP$

$Q = P$

for $i = m - 2$ *to* 0 **do**

  $Q = 2Q$ (Point Doubling)

  if k_i=*1* **then**

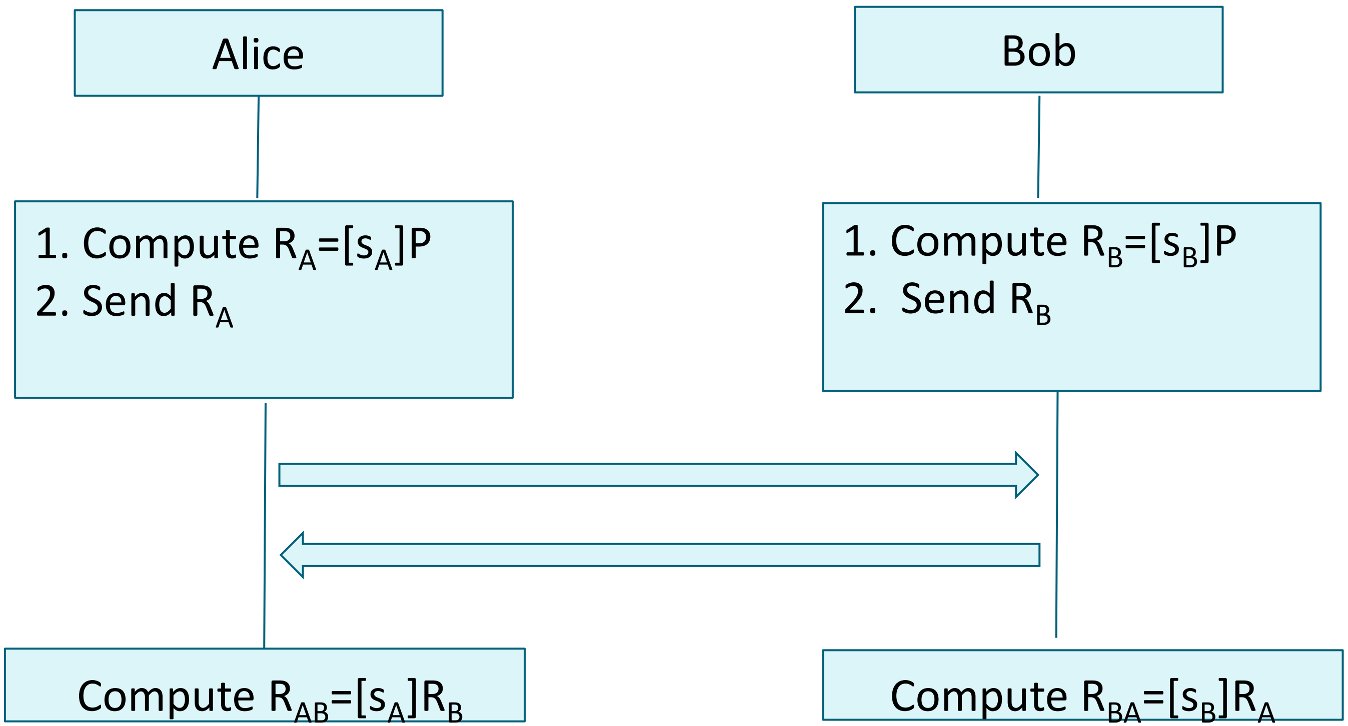
    $Q = Q + P$ (Point Addition)

Example: Compute 10P

- $k=10=(1010)_2$,m=4, Q=P
- iteration i=2, k_i=0, Q=2Q=2P
- iteration i=1, k_i=1, Q=2Q+P=4P+P=5P
- iteration i=0, k_i=0, Q=2Q=10P

This is the most simplistic and most efficient method for computing ECC scalar multiplication, but this algoithm has some serious vulnerability, and therefore is not used in practical implementations

Elliptic Curve Diffie-Hellman Key Exchange



$R_{AB}=[s_A][s_B]P=[s_B][s_A]P=R_{BA}$

Hard Problem: Computing s_A from the knowledge of R_A and P

Elliptic Curve Digital Signature Algorithm

- **Setup:**
  - E defined over GF(p), $y^2=x^3+Ax+B$
  - Order of the curve n and the generator point G ($[n]G=O$)
  - Choose random integer $1<d<n$
  - Compute $H=[d]G$, $k_{pub}=(p,n,G,H)$, $k_{priv}= d$, message=m
- **Signature:**
  - Choose random ephemeral key k_r, $1 < k_r < q$.
  - Compute $R = k_rG$ and r is the x coordinate of R
  - Compute $s = (SHA(m) + d \cdot r)(k_r)^{-1} \bmod n$, (r,s) is the signature
- **Verification:**
  - Compute value $w = s^{-1} \bmod n$.
  - Compute value $u_1 = w \cdot SHA(m) \bmod n$.
  - Compute value $u_2 = w \cdot r \bmod n$.
  - Compute $P = u_1G + u_2H$.
  - If x coordinate of $P = r \bmod n$, the signature is valid. Otherwise, it is invalid.

ECDSA: Proof

$$
\begin{aligned}
&u_1H + u_2G \\
&= (w.SHA(m).G + w.r.H)\ mod\ n \\
&= (w.SHA(m).G + w.r.[d]G)\ mod\ n \\
&= w(SHA(m) + r.d)G\ mod\ n \\
&= (SHA(m) + r.d)^{-1}k_r(SHA(m) + r.d)G\ mod\ n \\
&= [k_r]G = R
\end{aligned}
$$

Elliptic Curve Example:

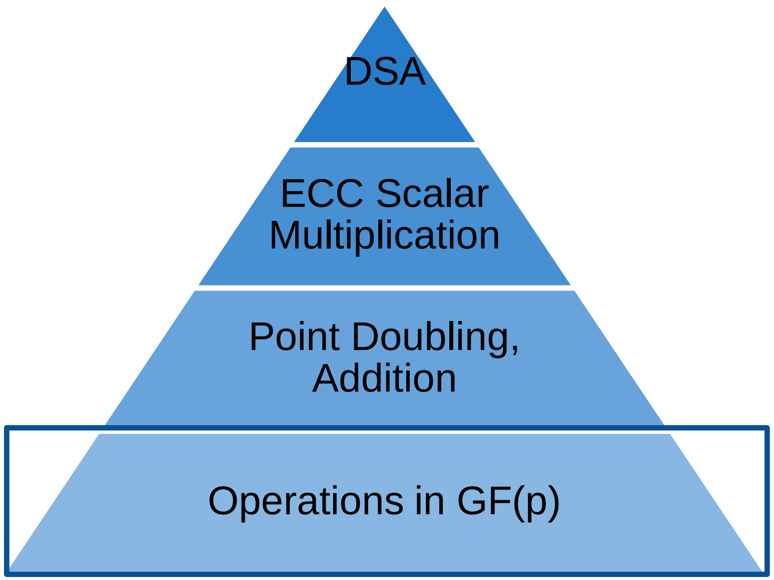
- NIST Curves on GF(p) (Based on Solinas Prime)
  - NIST P-224: $p = 2^{224} - 2^{96} + 1$
  - NIST P-256: $p = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$
  - NIST P-384: $p = 2^{384} - 2^{128} - 2^{96} + 2^{32} - 1$
- Curve 25519: $p=2^{255} -19$ (Based on Pseudo-Mersenne Prime)
- Ed448-Goldilocks: $p = 2^{448} - 2^{224} - 1$ (Based on Solinas Prime)
- NIST Binary Curves defined over $GF(2^m)$ (Based on trinomial and pentanomial)
- The Elliptic curves that have been chosen for cryptographic applications are non-singular curves
- Condition for non-singularity for elliptic curve defined over prime field is **$4a^3+27b^2 \neq 0$**

ECC and RSA Key Strength Comparison

Security	RSA Key Size	ECC Key Size
80	1024	160
112	2048	224
128	3072	256
192	7680	384
256	15360	521

- ECC provides more security per key bit compared to RSA, hence ECC is more lightweight and compact than RSA
- Unfortunately, in the presence of quantum computers, both traditional ECC and RSA will not remain secure and we would require new class of public key cryptography (post quantum secure public key algorithms)

Elliptic Curve Implementation Pyramid:



- Critical operations are finite field computation in GF(p)
- The field elements are of large dimension (from 192 to 521 bits), which makes implementing field operation even more challenging

- ECC can be operated on Mersenne and Solinas reduction friendly prime, but the implementation become curve specific
- For generic implementation of ECC, we require generic finite field architecture
- The standard coordinate system (x,y) is known as affine coordinates
- In affine coordinate, for each point doubling and addition, we require one field inversion

Field Multiplication Algorithms

- Let a and b be two elements in GF(p)/GF(2^n)
- We want to compute **(a x b) mod p**
  - p is a prime for GF(p)
  - p is a primitive polynomial for GF(2^n)
- Strategy-1
  - Compute the standard multiplication
  - After the multiplication, perform a modular reduction
  - This strategy is efficient when
    - p is a prime of form pseudo Mersenne ($2^n \pm c$) or Solinas ($2^n \pm 2^m \pm k$)
    - modular reduction for this kind of prime is easy to execute
    - For GF(2^n), the modular reduction is easy to execute when the primitive polynomial is either a trinomial or pentanomial.
  - This strategy is inefficient when
    - p does not have any specific structure and there is no easy method for performing modular reduction for those values of p
    - We in those cases require new field muliplication algorithms

Standard Multiplication

- School book method (Complexity: O(n^2))
- Karatsuba Algorithm (Complexity: O($n^{\log_2 3}$))
- Number theoretic Multiplication (Complexity: O(nlogn))
  - Schönhagen-Strassen (Complexity: O(n. logn. loglogn))
  - This algorithm is only advantageous when the value of n is very large
- This presentation focuses on Karatsuba multiplication in GF(2^n)
  - So basically, we will learn how to implement polynomial multiplication using Karatsuba multiplication, where coefficient of the polynomial is either 0 or 1
- But before we start, let‘s recap schoolbook method for polynomial multiplication

$$A(x) = \sum_{i=0}^{d} a_i x^i, B(x) = \sum_{i=0}^{d} b_i x^i \quad \text{both are d degree polynomials, with d+1 coefficients}$$

$$C(x) = A(x) \cdot B(x) = \sum_{i=0}^{d} \sum_{j=0}^{d} a_i \cdot b_j x^{i+j}$$

- Complexity: O(n^2)

Karatsuba Multiplication in GF(2^n)

- Let us consider degree 1 polynomials

$$A(x) = a_1 x + a_0, B(x) = b_1 x + b_0, (a_i, b_i \in GF(2^n))$$

$$T_0 = a_0 \cdot b_0, T_1 = a_1 \cdot b_1, T_2 = (a_0 \oplus a_1) \cdot (b_0 \oplus b_1)$$

$$A(x) \cdot B(x)$$

$$= T_1 x^2 + (T_2 \oplus T_1 \oplus T_0)x + T_0$$

Karatsuba Method

$$= \boxed{(a_1 \cdot b_1)x^2 + ((a_0 \oplus a_1) \cdot (b_0 \oplus b_1) \oplus (a_0 \cdot b_0) \oplus (a_1 \cdot b_1))x + (a_0 \cdot b_0)}$$

$$= (a_1 \cdot b_1)x^2 + ((a_0 \cdot b_0) \oplus (a_0 \cdot b_1) \oplus (a_1 \cdot b_0) \oplus (a_1 \cdot b_1) \oplus (a_0 \cdot b_0) \oplus (a_1 \cdot b_1))x + (a_0 \cdot b_0)$$

$$= \boxed{(a_1 \cdot b_1)x^2 + ((a_0 \cdot b_1) \oplus (a_1 \cdot b_0))x + (a_0 \cdot b_0)}$$

School Book Method

- Cost of Karatsuba: 3 Multiplications, 4 Additions
- Cost of Schoolbook Method: 4 multiplications and 1 addition
- Tradeoff: In Karatsuba, We reduce one multiplication, but increase number of additions by 3

When to apply Karatsuba

- Cost of 1 multiplication is greater than 3 additions

$$\mathbf{MUL_{cost} > 3 \times Addition_{cost}}$$

- Let us reconsider the degree 1 polynomial
- The multiplication for degree 1 polynomial is basically AND operation
- The addition for degee 1 polynomial is simple bitwise XOR operation
- Therefore, for Karatsuba algorithm to be efficient, cost of implementing a single AND gate opertion should be more thant cost of implementing three XOR gate, which is not the case.
- Therefore, for degree 1 polynomial, schoolbook algorithm is more efficient than Karatsuba algorithm.

Threshold for Karatsuba: Consider multipliations of polynomials of degree n.

If n > Threshold , $MUL_{cost} > 3 \times Addition_{cost}$ => Karatsuba algorithm is more efficient
If n <= Threshold , $MUL_{cost} < 3 \times Addition_{cost}$ => Schoolbook method is more efficient

Application of Karatsuba

- Consider two polynomials of degree 2^n-1, the number of coefficients in each polynomial is $m=2^n$.

$$A(x) = \sum_{i=0}^{m-1} a_i x^i, \quad A(x) = A_u(x) x^{m/2} + A_l(x)$$

$$A_u(x) = \sum_{i=0}^{m/2-1} a_{i+m/2} \cdot x^i, \quad A_l(x) = \sum_{i=0}^{m/2-1} a_i \cdot x^i$$

$$B(x) = \sum_{i=0}^{m-1} b_i x^i, \quad B(x) = B_u(x) x^{m/2} + B_l(x)$$

$$B_u(x) = \sum_{i=0}^{m/2-1} b_{i+m/2} \cdot x^i, \quad B_l(x) = \sum_{i=0}^{m/2-1} b_i \cdot x^i$$

- A_u and B_u contains the coefficients from index m/2 to m-1
- A_l and B_l contains the coefficients from index 0 to m/2-1
- If we substitute $x^{m/2}$ with y, we get

$$A(x) = A_u(x) \cdot y + A_l(x), \quad B(x) = B_u(x) \cdot y + B_l(x)$$

- So, now we can easily apply Karatsuba method here

Simple Recursive Karatsuba Algorithm

Algorithm 1. KA: Simple Recursive Karatsuba Algorithm

Input: A(x), B(x), threshold

Output: $C(x) = A(x) \cdot B(x)$

N=max(degree(A(x)), degree(B(x)))+1 ;

if $N{=}{=}threshold$ **then**

  return $A(x) \cdot B(x)$ (using schoolbook method);

end

$A(x) = A_u(x)x^{N/2} + A_l(x)$;

$B(x) = B_u(x)x^{N/2} + B_l(x)$;

$T_0 = KA(A_l, B_l)$;

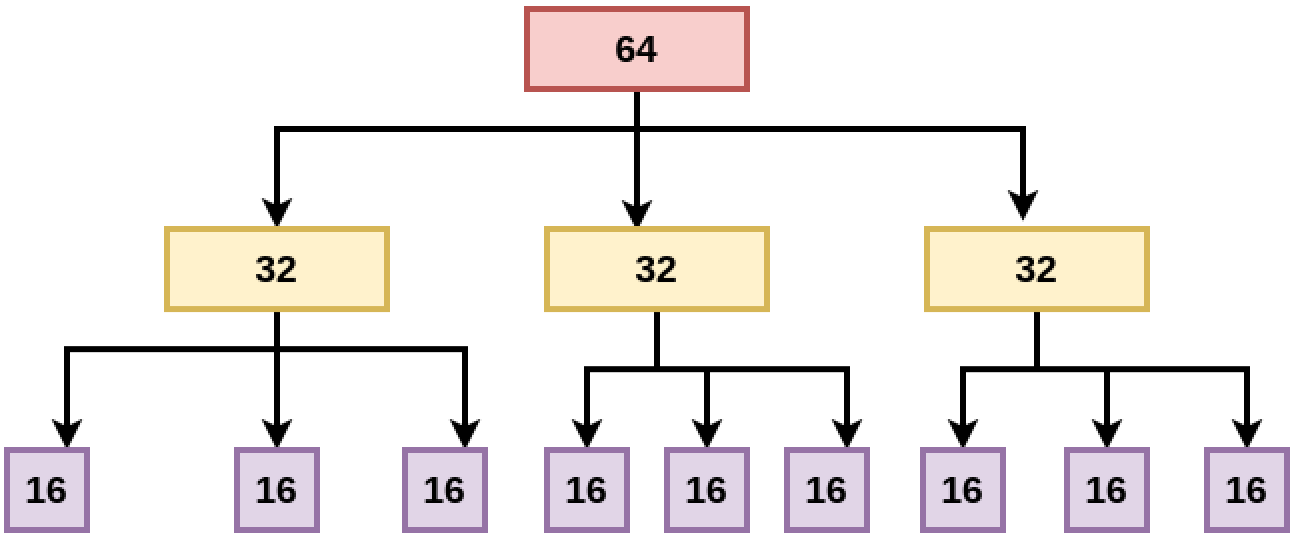
$T_1 = KA(A_u, B_u)$;

$T_2 = KA((A_l \oplus A_u), (B_l \oplus B_u))$;

return $T_1 x^N + (T_2 \oplus T_1 \oplus T_0)x^{N/2} + T_0$;

Example:

- Let us consider two polynomials of degree 63 and number of coefficients 64



Threshold value varied from platform to platform. For FPGA platform, typically 16 is chosen as threshold.

Complexity Analysis

- Master Theorem of Recurrence:

$$T(n) = aT(n/b) + O(n^c)$$
$$T(n) = O(n^{\log_b a}) \quad \text{if } c < \log_b a$$

- Complexity of Karatsuba Algorithm

$$T(n) = 3T(n/2) + O(n)$$

comparing with master theorem, we see that c=1

b=2, a=3, and $c < \log_b a$ holds true

Thus $T(n) = O(n^{\log_2 3}) < O(n^2)$

Applying Recursive KA for Arbitrary polynomial

- Remember the first computation step of Karatsuba algorithm:
  - N=max(degree(A(x)), degree(B(x)))+1
- In this case, the value of N will not be even always
- Trick: Splid the operand polynomial into a lower part of $\lceil N/2 \rceil$ coefficients and upper part of $\lfloor N/2 \rfloor$ coefficients

Example

- In this particular example, we consider threshold as 1.

- We want to multiply $A(x)$ and $B(x)$ where $A(x) = a_5x^5 + a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0$ $B(x) = b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1x +$ b_0 Each polynomial has six coefficients.
- **Step 1:** $A(x) = A_1(x).x^3 + A_0(x)$, $B(x) = B_1(x).x^3 + B_0(x)$ where $A_1(x) = a_5x^2 + a_4x + a_3$, $B_1(x) = b_5x^2 + b_4x + b_3$, $A_0(x) = a_2x^2 + a_1x + a_0$, and $B_0(x) = b_2x^2 + b_1x + b_0$.
- **Step 2:** $A_1(x) = A_{11}(x).x^2 + A_{10}(x)$
  $A_0(x) = A_{01}(x).x^2 + A_{00}(x)$
  $A_{11}(x) = a_5$, $A_{10}(x) = a_4x + a_3$, $A_{01}(x) = a_2$, $A_{00}(x) = a_1x + a_0$
  $B_1(x) = B_{11}(x).x^2 + B_{10}(x)$
  $B_0(x) = B_{01}(x).x^2 + B_{00}(x)$
  $B_{11}(x) = b_5$, $B_{10}(x) = b_4x + b_3$, $B_{01}(x) = b_2$, $B_{00}(x) = b_1x + b_0$
- **Step 3:** Now we have reached the threshold value, so we stop the recursion and apply school book algorithm.

An alternative approach to Recursive Karatsuba

- Let us again consider two polynomial A(x) and B(x)

$$A(x) = a_2x^2 + a_1x + a_0, B(x) = b_2x^2 + b_1x + b_0$$
$$T_0 = a_0 \cdot b_0,\ T_1 = a_1 \cdot b_1,\ T_2 = a_2 \cdot b_2$$
$$T_3 = (a_0 \oplus a_1) \cdot (b_0 \oplus b_1), T_4 = (a_0 \oplus a_2) \cdot (b_0 \oplus b_2)$$
$$T_5 = (a_1 \oplus a_2) \cdot (b_1 \oplus b_2)$$
$$C(x) = T_2x^4 + (T_5 \oplus T_1 \oplus T_2)x^3 + (T_4 \oplus T_2 \oplus T_1 \oplus T_0)x^2 + (T_3 \oplus T_1 \oplus T_0)x + T_0$$

- This kind of methodology can be applied to any generic arbitrary polnomial. But, for most of the cases, recursive KA is as efficient as this method.
- **Integer Multiplication using Karatsuba**:

$$A = \sum_{i=0}^{n-1} a_i 2^i, B = \sum_{i=0}^{n-1} b_i 2^i$$
$$A = A_1 \cdot 2^{n/2} + A_0, B = B_1 \cdot 2^{n/2} + B_0$$
$$A \cdot B = A_1 \cdot B_1 \cdot 2^n + ((A_1 + A_0) \cdot (B_1 + B_0) - A_1 \cdot B_1 - A_0 \cdot B_0)$$

Fast Modular Reduction

- As we have already stated, modular multiplication with some specific prime or primitive polynomial is easy to execute
- Here, we will focus on two such example
  - Modular reduction using trinomial primitive polynomial
  - Modulat reduction using Solinas prime

Polynomial Reduction using Trinomial

- Let us consider a trinomial:

$$x^m + x^n + 1 = 0$$

$$=> x^m = 1 + x^n$$

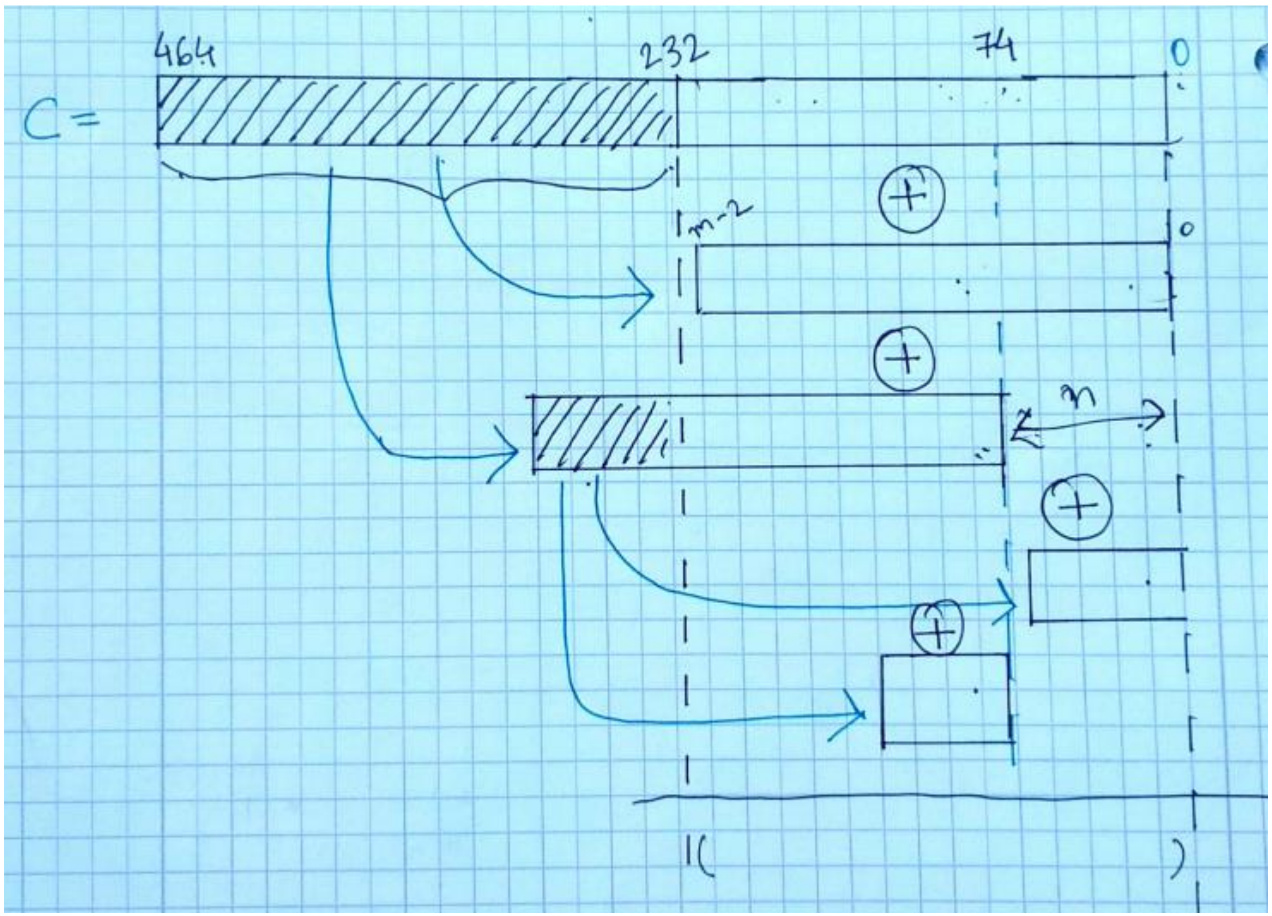
$$=> x^{2m-2} = x^{m-2} + x^{m+n-2}$$

$$A = \sum_{i=0}^{m-1} a_i x^i, \; B = \sum_{i=0}^{m-1} b_i x^i$$

$$A \times B = C = \sum_{i=0}^{m-1} \sum_{i=0}^{m-1} a_i \cdot b_j x^{i+j} = \sum_{i=0}^{2m-2} c_i x^i$$

Example:

- Trinomial: $x^{233} + x^{74} + 1$, therefore in our example, m=233 and n=74.



Similar Strategy can also be applied to pentanomial

Modular Reduction using Solinas Prime

- Let us consider a prime P= 2^{192}-2^{64}-1 => 2^{192}=(2^{64}+1) mod P
- A x B=C, A and B are 192 bit intger and belongs to GF(P), the product C is 384 bit wide. Each C_i

$$C = C_5 2^{320} + C_4 2^{256} + C_3 2^{192} + C_2 2^{128} + C_1 2^{64} + C_0$$

$$C = 2^{192}(C_5 2^{128} + C_4 2^{64} + C_3) + (C_2 2^{128} + C_1^{64} + C_0)$$

This term should be $C_1 2^{64}$

$$= (2^{64} + 1)(C_5 2^{128} + C_4 2^{64} + C_3) + (C_2 2^{128} + C_1^{64} + C_0)$$

$$= C_5 2^{192} + C_4 2^{128} + C_3 2^{64} + C_5 2^{128} + C_4 2^{64} + C_3 + C_2 2^{128} + C_1^{64} + C_0$$

$$= C_5 (2^{64} + 1) + C_4 2^{128} + C_3 2^{64} + C_5 2^{128} + C_4 2^{64} + C_3 + C_2 2^{128} + C_1^{64} + C_0$$

$$= \underbrace{C_5 (2^{128} + 2^{64} + 1)}_{T} + \underbrace{C_4 (2^{128} + 2^{64})}_{S_1} + \underbrace{C_3 (2^{64} + 1)}_{S_2} + \underbrace{C_2 2^{128} + C_1^{64} + C_0}_{S_3}$$

This term should be $C_1 2^{64}$

Final Solution

- $T = C_5 \,||\, C_5 \,||\, C_5 \Rightarrow C_5 2^{128} + C_5 2^{64} + C_5$
  $S_1 = C_4 \,||\, C_4 \,||\, 0 \Rightarrow C_4 2^{128} + C_4 2^{64}$
  $S_2 = 0 \,||\, C_3 \,||\, C_3 \Rightarrow C_3 2^{64} + C_3$
  $S_3 = C_2 \,||\, C_1 \,||\, C_0 \Rightarrow C_2 2^{128} + C_1 2^{64} + C_0$

The final result is obtained by adding T, S_1, S_2 and S_3 . Thus the entire modular reduction can be implemented by usig additions only. To reduce the addition result into GF(P), a few subtractions would be necessary.

I request you to think about how to perform modular reduction for prime $P=2^{255}-19$ using only addition and subtraction

Finite Field Inversion: Fermat´s little theorem

- Let us consider a finite field GF(p) and a is a random element in this field. Also, p does not divide a. Proof the following relation:

$$\mathbf{a^{(p-1)} = 1 \bmod p}$$

- Proof: Let us consider the following elements: a, 2a, 3a, …, (p-1).a. Now suppose there exist two elements r and s < p, such that

$$ra=sa \bmod p.$$

Then we can conclude that r =s mod p (as $a \neq 0$). Moreover, as both r and s are less than p, then r=s. Therefore we can conclude a,2a,3a,…,(p-1).a are all unique elements: total (p-1) unique elements.

Hence, we can write the following expression:

$$a \times 2a \times 3a \ldots \times (p-1).a = 1 \times 2 \times 3 \times \ldots (p-1) \bmod(p) \text{ (in some order)}$$

$$\Rightarrow a^{p-1} \times (p-1)! = (p-1)! \bmod p \Rightarrow a$$

This theorem is known as Fermat's little theorem

From here we can see that inverse of an element a is a^{p-2} mod p

Algorithm to Compute Exponentiation

Input: a, p-2 in its binary from= $\{p_{n-1},\ldots,p_2, p_1, p_0\}$. p is a n bit prime

Output: a^{p-2}

Naive Method

1. $r=1$;
2. For ($i=n-1$ to 0)
3. $\quad r= r \times r$
4. $\quad$ If($p_i==1$)
5. $\qquad r=r \times a$
6. end For
7. Return r

Non constant time, number of multiplication is equal to HW(p-2)

Ladder Method

1. $r_0=1, r_1=a$;
2. For ($i=n-1$ to 0)
3. $\quad$ If($p_i==0$)
4. $\qquad r_1=r_1 \times r_0$
5. $\qquad r_0=r_0 \times r_0$
6. $\quad$ else
7. $\qquad r_0=r_1 \times r_0$
8. $\qquad r_1=r_1 \times r_1$
9. end For
10. Return r_0

Constant time, number of multiplications is n and number of squarings is n

Extended Euclidean Inversion Algorithm

Input: a, p
Output: gcd(a,p), d=a^{-1} mod p

```
u=a,v=p, A=1,B=0, C=0, D=1
while(u!=0)
    while even(u)
        u=u/2
        if even(A) and even(B) then
            A=A/2, B=B/2
        else
            A=(A+p)/2, B=(B-a)/2
    while even(v)
        v=v/2
        if even(C) and even(D) then
            C=C/2, D=D/2
        else
            C=(C+p)/2, D=(D-a)/2
    if u >=v then
        u=u-v, A=A-C, B=B-D
    else
        v=v-u, C=C-A, D=D-B
Return v,d=C mod p
```

- This algorithm is more efficient than the Fermat´s little theorem
- Does not require multiplication and squaring, can be implemented using shifter, additions and subtractions
- Difficult to make constant time
- Implementation of inversion using either of these two algorithms much more expensive then field multiplications.
- For each doubling and addition in affine coordinates require one inversion, for n bit scalar, in worst case we require 2n inversions
- Can we reduce the number of inversions while doing scalar multiplication

Montgomery Multiplication

- **Objective:** To compute a * b mod P for any generic arbitrary prime

Algorithm 2. Mont_Reduc(R,P,T): Montgomery Reduction

Input: R, P, T. P is the prime, $R = 2^k > P$, $0 \leq T < PR$, $gcd(P, R) = 1$

Output: TR^{-1} mod P

```
Compute P' such that RR^{-1} - PP' = 1 (can be computed using
  extended Euclidean algorithm);
m = T × P' mod R ;
t = (T + mP)/R;
if t ≥ P then
  | t = t − P ;
end
return t;
```

R is chosen as 2^k so that “mod R“ and division by R are easy to implement.

RR^{-1}-PP'=1 also means that PP'=-1 mod R

Proof of correctness

- Claim 1: $t=(T+mP)/R$ is an exact division i.e. $(T+mP) \bmod R = 0$
- Proof:
  - $T+mP \bmod R$

  $= (T+P(T.P' \bmod R)) \bmod R$

  $=(T+T(P.P' \bmod R)) \bmod R$

  $=(T-T) \bmod R$ (because $P.P' \bmod R=-1$)

  $=0$

  Therefore, $(T+mP)/R$ is an exact division

- Claim 2: $t=TR^{-1} \bmod P$
- Proof:
  - $0 <= m < R => mP < PR => T+mP < 2PR$
  - Therefore, $t=(T+mP)/R < 2P$, and after the conditonal subtraction $t < P$.
  - Thus $t=(T+mP)/R$ is equivalent to $t=(T+mP)/R \bmod P$
  - Now, $t=(T+mP)/R \bmod P= (TR^{-1} +mPR^{-1}) \bmod P= TR^{-1} \bmod P$

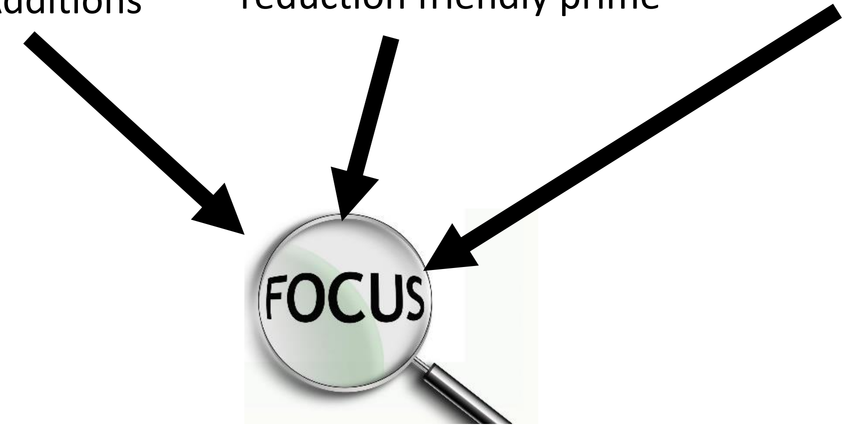
Assignment 2

- Write the python code of mongomary multiplication

Efficient Finite Field Architecture

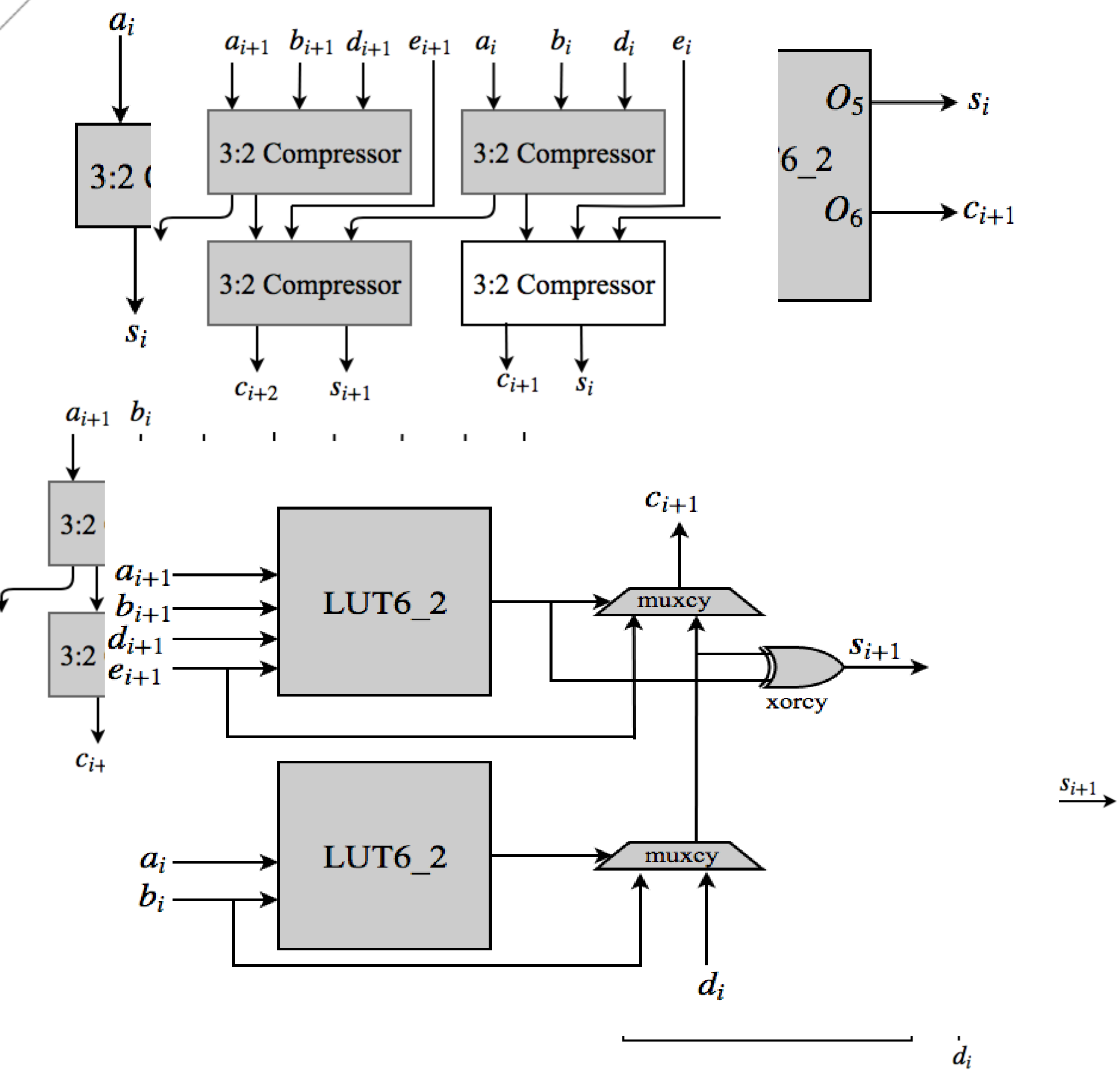
Objective: Finite Field Architecture

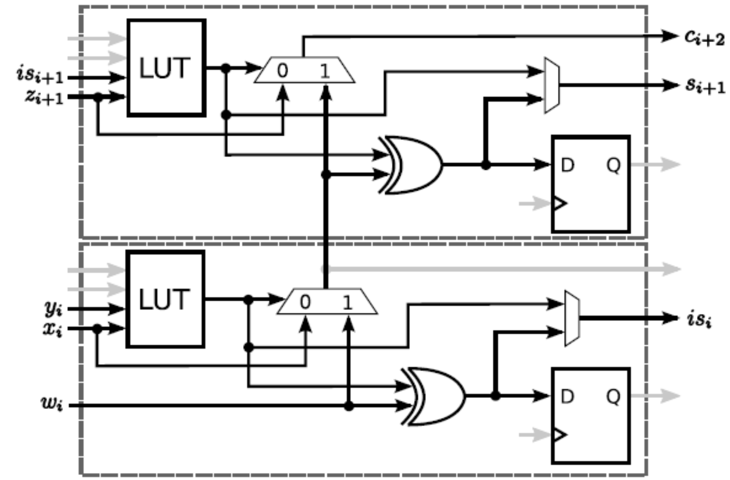
±	++	× ‖ %	× %	$\left\{\frac{1}{x}\right\}$
Modular Addition and Subtraction	Multi-operand Additions	Integer Multiplication for reduction friendly prime	Modular Multiplication for generic primes	Field Inversion



- Challenges: To reach a optimal area-time tradeoff of these designs; Focus will be more on speed for this presentation
- Develop implementations which exhibits better performance compared to state of the art architectures of ECC and SIKE algorithms.

Multi-Operand Addition: 3:2 and 4:2 Compressor

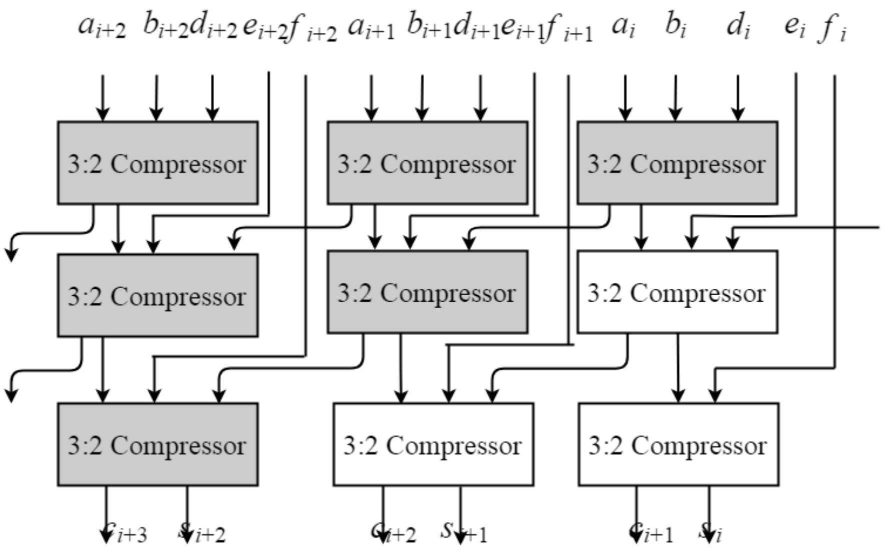


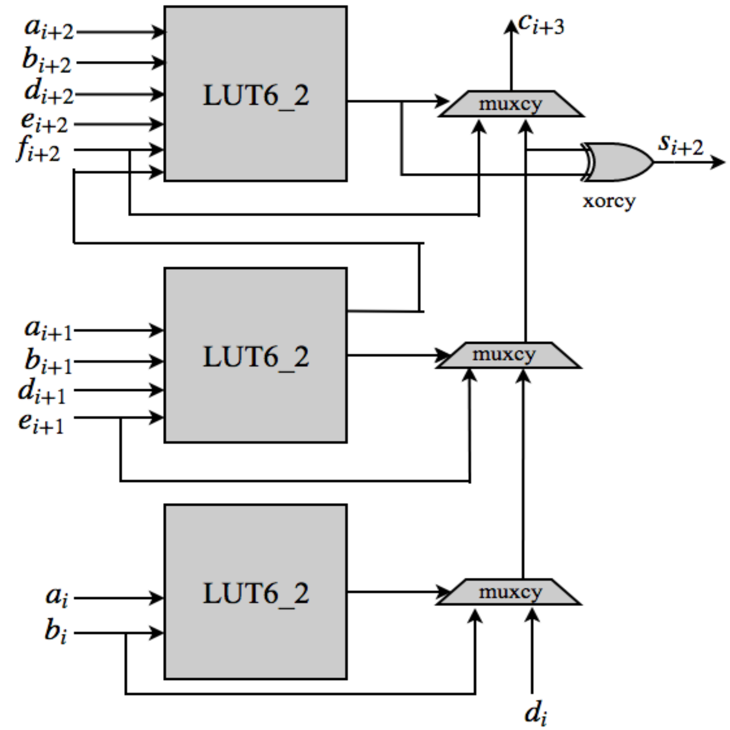


$2LUT_D+muxcy_D$

$Max(LUT_D+xorcy_D+muxcy_D, LUT_D+2muxcy_D)$

5:2 Compressor





$\mathrm{Max}(\mathrm{LUT}_D+2\mathrm{muxcy}_D+\mathrm{xorcy}_D, \mathrm{LUT}_D+3\mathrm{muxcy}_D)$

- Multi-operand addition is useful for implementing modular reduction using pseudo-Mersenne or Solinas prime.
- It will be very useful for implementing Montgomery modular multiplication.

Integer Multiplication: Optimum Usage of DSP Blocks

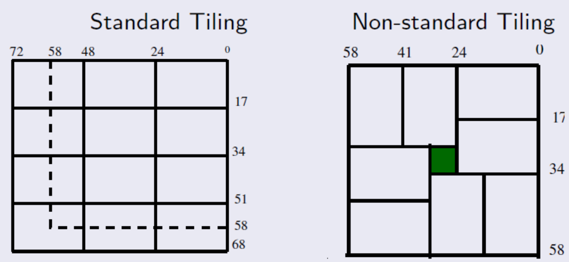
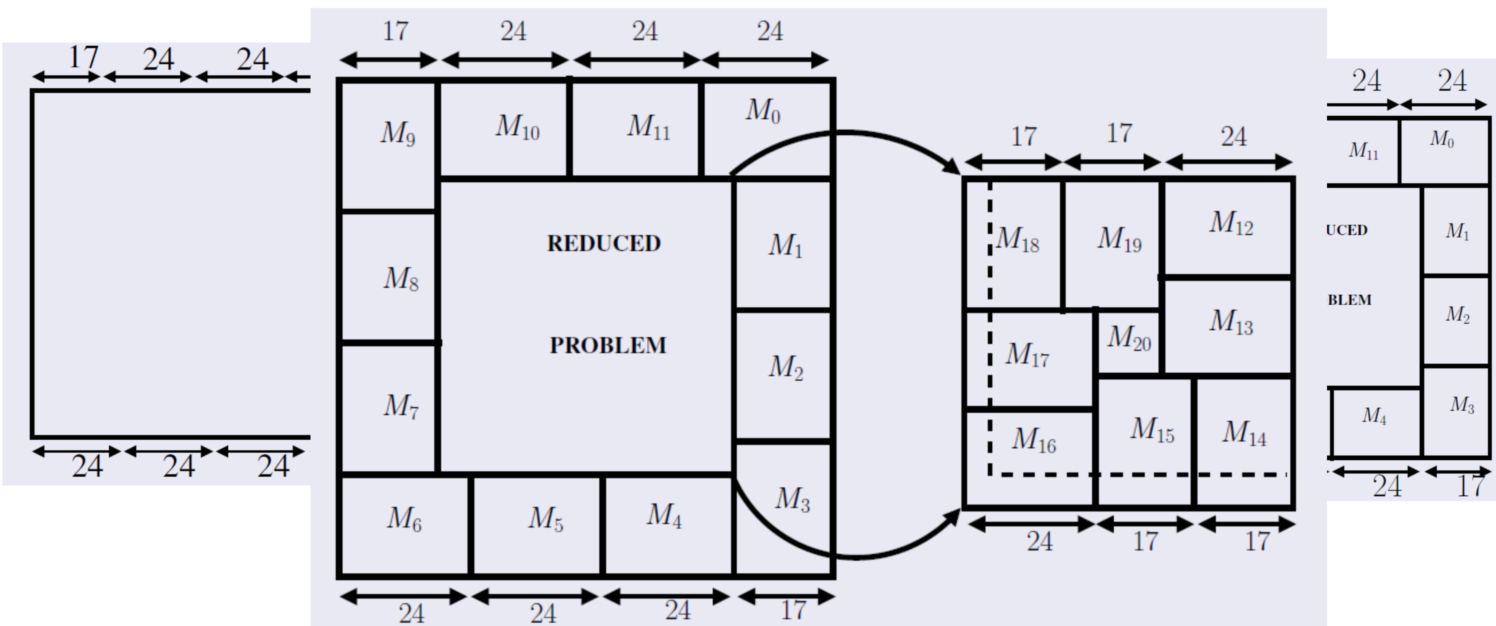


Figure: Multiplying Operands of Width 58 using Asymmetric Multipliers [1]

[1] F. de Dinechin and B. Pasca, Large multipliers with fewer DSP blocks, in Field Programmable Logic and Applications, pp. 250-255, 2009.

DSP Block contains 24 x17 unsigned multiplier

58 x 58 Multiplication
School Book Method: 12 DSP
Non-standard Tiling: 8 DSP, 7x7 small mulltiplier

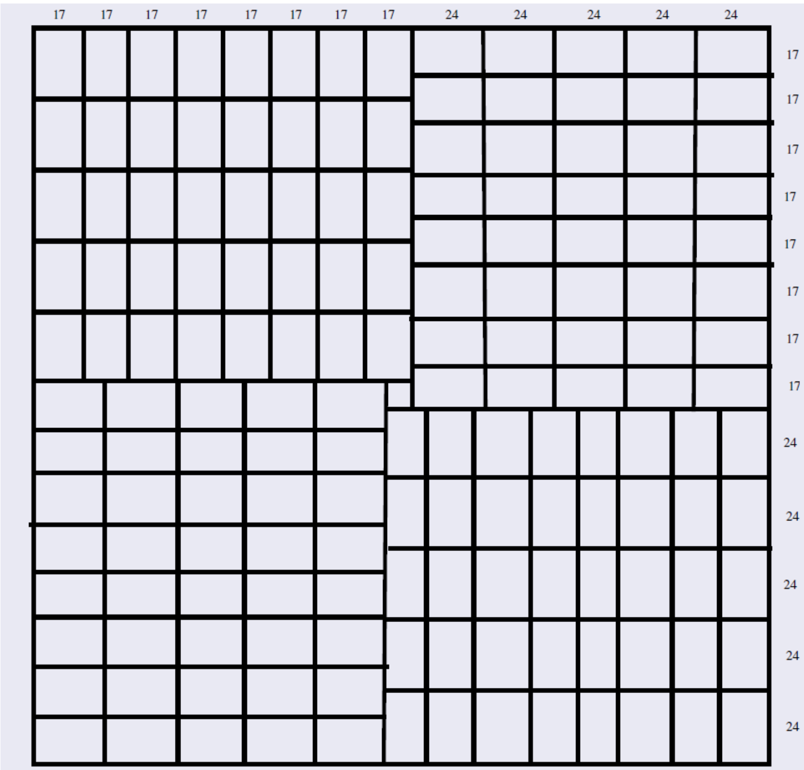


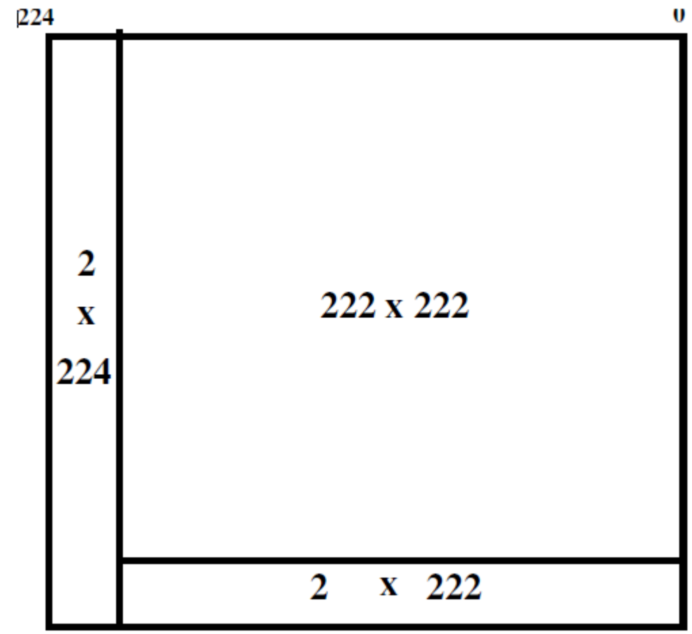
89=24x3+1x17

Standard Requirement =24
Non-Standard Tiling Requirement=20

P-256 and P-224

256=24x5+17x8; DSP Block Requirement=160, additional 16x16 multiplier is required





To multiply two operands of length 224, we map the problem to 222. We need to do three multiplications:

- Both Operands having length 222
- One having length 2 and another one having length 222.
- One having length 223 and another one having length 2.

Results

Operand width(b)	Mapped Operand width(a)	Decomposition	Reduction Step	Multipliers Required by Standard Tiling	Multipliers Required by Non-standard Tiling
192	191	$24 * 3 + 17 * 7$	$191 \rightarrow 47$	96	**90**
224	222	$24 * 5 + 17 * 6$	$222 \rightarrow 18$	140	**120**
256	256	$24 * 5 + 17 * 8$	$256 \rightarrow 16$	176	**160**
384	382	$24 * 6 + 17 * 14$	$382 \rightarrow 94$	368	**360**
521	519	$24 * 11 + 17 * 15$	$519 \rightarrow 9$	682	**660**

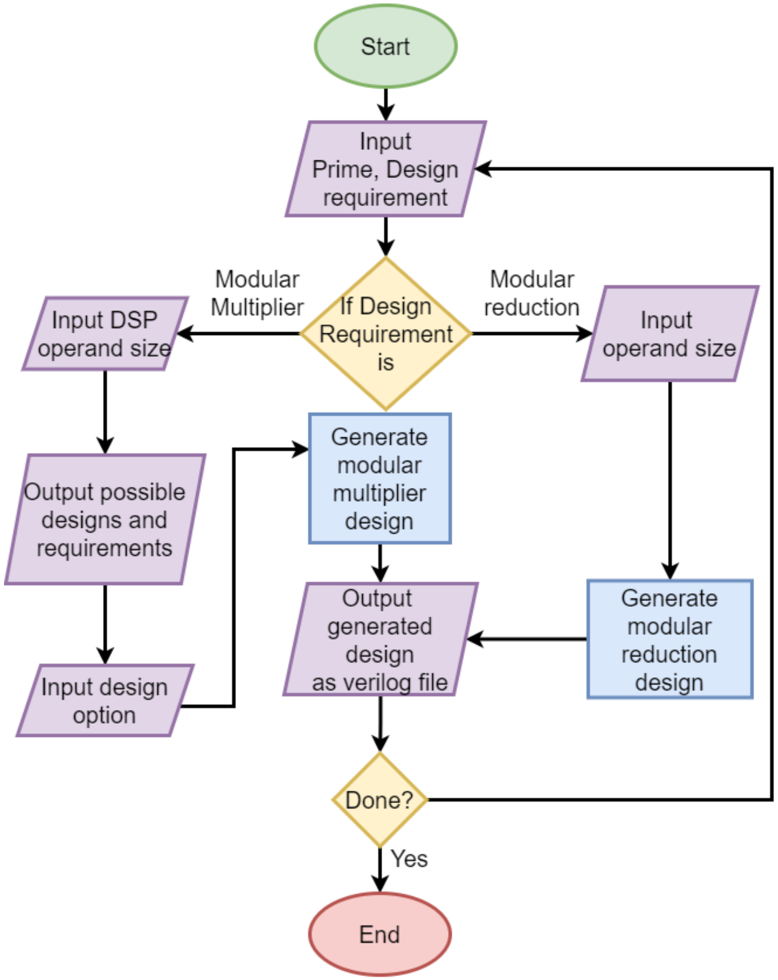
Non-standard tiling requires fewer multiplier i.e. fewer DSP blocks
This multiplier can be useful for implementation with reduction friendly prime, or in any application which requires large integer multiplication.
This multiplier has been used by other researchers also for their own cryptographic implementation, one of them has used it for implementation of SIKE[1,2].

- **Tile Before Multiplication: An Efficient Strategy to Optimize DSP Multiplier for Accelerating Prime Field ECC for NIST Curves, Debapriya Basu Roy, Debdeep Mukhopadhyay, Masami Izumi, Junko Takahashi, DAC 2014: 177:1-177:6**

1. Koppermann, Philipp, et al. "Fast FPGA implementations of Diffie-Hellman on the Kummer surface of a genus-2 curve." *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2018): 1-17.

2. Massolino, P. M., et al. "A compact and scalable hardware/software co-design of SIKE."*IACR Transactions on Cryptographic Hardware and Embedded Systems* (2020).

Automatic Generation of Modular Multipliers Upon Pseudo-Mersenne Primes Using DSP Blocks on FPGAs[1]



Features:

1) Automated modular multiplier generation upon Pseudo-Mersenne Primes
2) Supporting Modular addition and subtraction code generation in RNS bases
3) Conversion to and fro between integer and RNS bases

Outputs:

DSP :- Number of DSPs required excluding the square multiplier.
cycles :- Number of clock cycles required.
A :- Size of one of the operands that is supported by the multiplier.
B :- Size of the other operand that is supported by the multiplier. Inner :- Size of the square multiplier .
RecDes :- Flag that says whether the square multiplier has to be recursively designed.

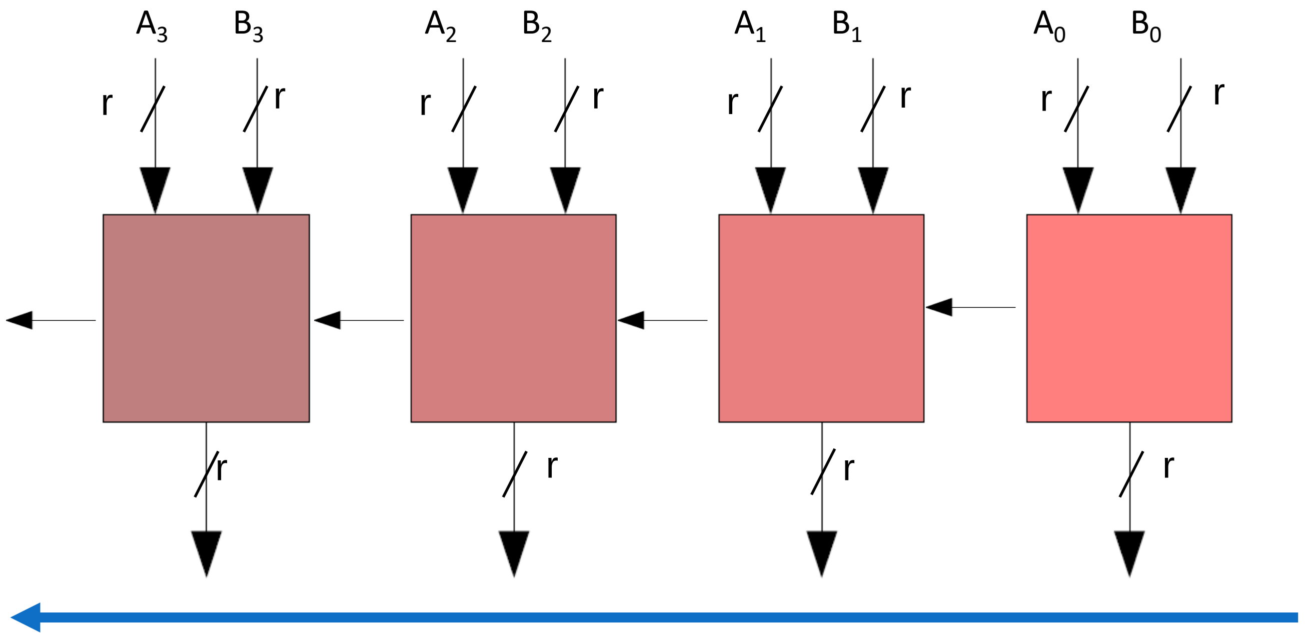
From the requirements provided, the user can select a design that will be generated as a Verilog file.

1. Roy, Debapriya Basu. "Automatic Generation of Modular Multipliers Upon Pseudo-Mersenne Primes Using DSP Blocks on FPGAs." 2024 27th Euromicro Conference on Digital System Design (DSD). IEEE, 2024.

Non-Redundant Number System

A d digit non-redundant number X can be represented as $(X_{d-1},\ldots,X_1,X_0)$ where each X_i is a r bit number.

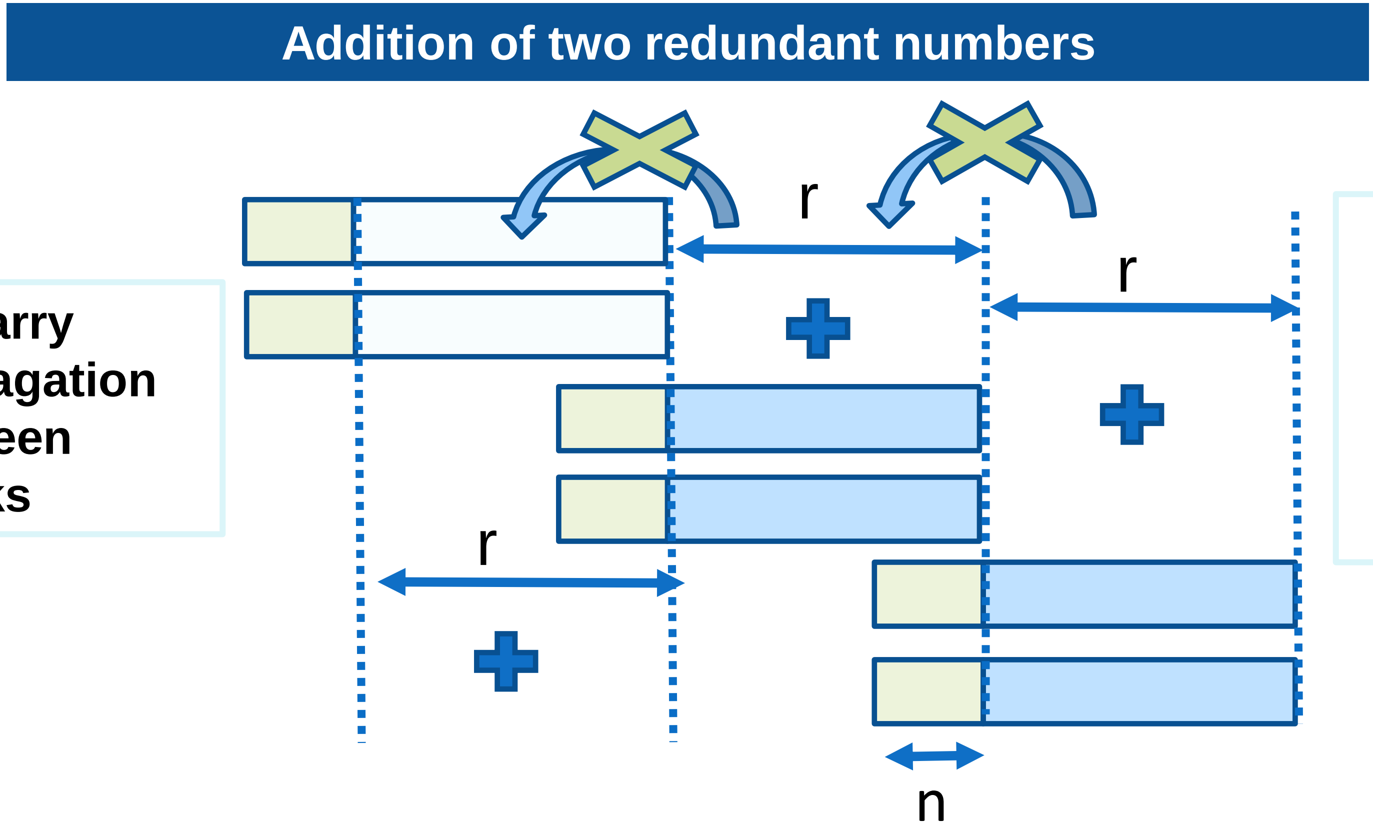
Addition of two non-redundant numbers A and B:



Carry Propagation

Redundant Number System

A d digit redundant number Y can be represented as (Y_{d-1}, …, Y_1, Y_0) where each Y_i is a r+n bit number. The group of r bits are principal bits and the group of n bits are redundant bits.



No carry propagation between blocks

When r=1 and n=1, we obtain the basic carry save form.

We have combined redundant number system with the asymmetric multipliers of DSP blocks to construct a novel Montgomery multiplier architecture that can be applied to both ECC and SIKE

Operations in Redundant Number System

$X' = (X'_{d-1}, \ldots X'_1, X'_0)$ be a d digit redundant number where the length of each X'_i is (r_2+2) bits. Similarly Y' is a single digit redundant number having length (r_1+2), where $2r_2 < r_1+r_2+4 < 3r_2$. We can compute the partial products $P_i = X'_i \,.Y'$ using the asymmetric multiplier of dimension $(r_1+2)\mathrm{x}(r_2+2)$. The length of each P_i would be (r_1+r_2+4).

$$
\begin{aligned}
K_0 &= P_0[r_2 - 1 : 0] \\
K_1 &= P_0[2r_2 - 1 : r_2] + P_1[r_2 - 1 : 0] \\
K_i &= P_{i-2}[r_1 + r_2 + 3 : 2r_2] + P_{i-1}[2r_2 - 1 : r_2] \\
&\quad + P_i[r_2 - 1 : 0] \qquad (2 \le i \le d - 1) \\
K_d &= P_{d-2}[r_1 + r_2 + 3 : 2r_2] + P_{d-1}[2r_2 - 1 : r_2] \\
K_{d+1} &= P_{d-1}[r_1 + r_2 + 3 : 2r_2]
\end{aligned}
$$

3:2 Compressor

- This operation requires DSP blocks to generate the partial products and 3:2 compressor to combine them.
- For our implementation we consider, r_1 as 15 and r_2 as 22 so that generation of each Pi fits into a single DSP block with 24x17 unsigned multiplier. The resulting Montgomery multiplier is called radix-22 redundant Montgomery multiplier

Operation in Redundant Number System

- We now want to compute X'.Y'+C where C=(C_d, …, C_1,C_0)

$$
\begin{aligned}
T_0 &= P_0[r_2-1:0] + C_0[r_2-1:0] \\
T_1 &= P_0[2r_2-1:r] + P_1[r_2-1:0] + C_1[r_2-1:0] \\
&\quad + C_0[r_2+1:r_2] \qquad (2 \le i \le d-1) \\
T_i &= P_{i-2}[r_1+r_2+3:2r_2] + P_{i-1}[2r_2-1:r_2] \\
&\quad + P_i[r_2-1:0] + C_i[r_2-1:0] + C_{i-1}[r_2+1:r_2] \\
T_d &= P_{d-2}[r_1+r_2+3:2r_2] + P_{d-1}[2r_2-1:r_2] \\
&\quad + C_d[r_2-1:0] + C_{d-1}[r_2+1:r_2] \\
T_{d+1} &= P_{d-1}[r_1+r_2+3:2r_2] + C_d[r_2+1:r_2]
\end{aligned}
$$

5:2
Compressor

Algorithm Constant Time Montgomery Multiplication

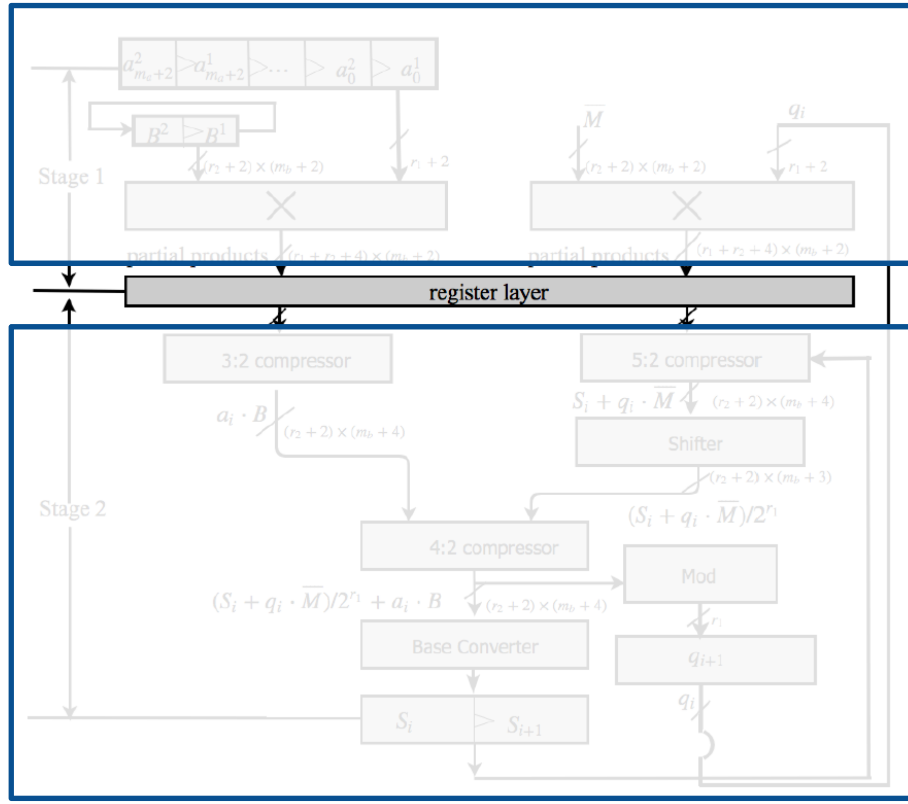
Input: M, $A = \sum_{i=0}^{m_a+2} a_i \cdot 2^{r_1 i}$ with $a_{m_a+2} = 0$, $B = \sum_{i=0}^{m_b+1} b_i \cdot 2^{r_2 i}$, $M' = -M^{-1}$ mod R, $\overline{M} = (M'$mod $2^{r_1}) \cdot M = \sum_{i=0}^{m_b+1} \overline{m}_i \cdot 2^{r_2 i}$, $A, B < 2\overline{M}$, $R = 2^{r_1(m+2)}$

Output: $A \times B \times R^{-1}$mod$M$

```
S_0 = 0, q_0 = 0;
for i ← 0 to m_a + 2 do
    T_1 = a_i.B // Computed using DSP Blocks and 3:2 compressor ;
    T_2 = S_i + q_i · M̄ //Computed using DSP Blocks and 5:2 compressor ;
    T_3 = (T_2)/2^{r_1} // Involves right shift and r_1 bit addition;
    S'_{i+1} = T_1 + T_3 // Computed using 4:2 compressor ;
    q_{i+1} = S'_{i+1} mod 2^{r_1} // Computed using r_1 bits addition ;
    S_{i+1} = Base_Converter(S'_{i+1}) ;
return S_{m_a+3} = A × B × R^{-1} mod M
```

Proposed Architecture:

Input a is a redundant number with radix r_2 and b is a redundant number with radix r_1 (r_1=15, r_2=22)



Key Observation: When multiplayer layers are active, adder layers are inactive and vice versa

Table: 256 bit Montgomery multiplier on Xilinx Zedboard

#Mults.	Slices	LUTS	FFs (Flip Flops)	DSPs	Clock Cycles	Freq. (MHz)	Area overhead wrt. Single multiplier	Speed gain wrt. Single multiplier
1	889	2149	464	40	15	100.2	NA	NA
2	1146	2164	1154	40	31	174.8	1.28 x	1.68 x
4	1354	2779	3985	40	63	305.1	1.52 x	2.89 x

Finite Field Architecture: Summary

Multi-operand Addition

- Efficient usage of carry chain and six input LUTs
- Proposed efficient circuit for 3:2, 4:2 and 5:2 compressors

Non-standard Tiling

- Optimum usage of DSP blocks
- Proposed non-standard tiling decomposition methodology
- Results in reduced usage of DSP blocks

Montgomery Multiplier

- Based on redundant number system
- Fast because of carry less arithmetic
- Can perform multiple modular multiplications simultaneously

Revisiting FPGA Implementation of Montgomery Multiplier in Redundant Number System for Efficient ECC Application in GF(p): Debapriya Basu Roy, Debdeep Mukhopadhyay, FPL2018: 323-326

Next Objective: Building ECC architecture using the developed finite field architecure

Efficient ECC Architecure

Projective Coordinates

- Consider a point P (x,y) which is in affine coordinate
- We can represent this point P into projective coordinates (X,Y,Z) where
  - $x=X/Z^c$, $y=Y/Z^d$
  - For a single point P in affine coordinate, we can have multiple projective coordinate representation: (X_1,Y_1,Z_1), (X_2,Y_2,Z_2) .... (X_n,Y_n,Z_n) as long as

$x=X_1/Z_1^c=X_2/Z_2^c=...=X_n/Z_n^c$, $y=Y_1/Z_1^d=Y_2/Z_2^d=...=Y_n/Z_n^d$

- Some popular projective coordinate
  - Standard Projective Form (c=1, d=1)
  - Jacobian (c=2,d=3)
  - Lopez-Dahab (c=1, d=2) (used mainly in binary curve)
- Let´s consider $y^2=x^3+Ax+B$ with Jacobinan coordinate
- $(Y/Z^3)^2=(X/Z^2)^3+AX/Z^2+B \implies Y^2=X^3+AXZ^4+BZ^6$

Point Addition in Jacobian Coordinate

- We want to add two points P (X_1,Y_1,Z_1) (in projective coordinates) and Q (x_2,y_2)
- Coordinate of P+Q in projective coordinate (X_3,Y_3,Z_3)

$$T_0 = Z_1^2,\ T_1 = Z_1.T_0,\ T_2 = x_2.T_0,$$

$$T_3 = y_2.T_1,\ T_4 = T_2 - X_1,\ T_5 = T_3 - Y_1$$

$$Z_3 = Z_1.T_4,\ T_6 = T_4^2,\ T_7 = T_4^3,$$

$$T_8 = X_1.T_6,\ X_3 = T_5^2 - (T_7 + 2T_8),\ Y_3 = T_5(T_8 - X_3) - Y_1.T_7$$

Point Doubling in Jacobian Coordinate:

- We want to double point P (X_1,Y_1,Z_1) to 2P (X_3,Y_3,Z_3)

$$T_0 = 3(X_1 - Z_1^2)(X_1 + Z_1^2),\ T_1 = 2Y_1,\ Z_3 = T_1.Z_1$$

$$T_2 = T_1^2,\ T_3 = T_2.X_1,\ X_3 = T_0^2 - 2T_3$$

$$Y_3 = (T_3 - X_3).T_0 - T_2^2/2$$

Overall cost

- Point Doubling Cost: 8 Multiplications (considering squaring equivalent to multiplication and ignoring the cost of modular addition and subtraction as thier cost are negligible compared to cost of modular multiplications)
- Point Addition Cost: 11 Multiplications
- Neither of them require any field inversion
- Converting affine point P (x,y) in to Jacobian coordinate P (X,Y,Z)
  - $x=X$, $y=Y$, $Z=1$
- Converting Jacobian point P (X,Y,Z) in to affine coordinate P (x,y)
  - $x=X/Z^2$, $y=Y/Z^3$, cost: 4 Mult.+1 Inversion
- We can actually assign value to the point of infinity in projective coordinate: O=(0,1,0)

Differential Addition:

- In differential addition, to perform addition of two points P and Q,we require the knowledge of P−Q.
- Differential addition formula was first proposed for Montgomery curve.
- One of the advantages of using differential addition in Montgomery curve is that the entire scalar multiplication can be performed using only the x coordinate of the points.
- Scalar multiplication algorithm using differential addition on Montgomery curve is known as the Montgomery ladder.
- A Montgomery curve E defined over GF(p) can be represented as below:
  - $By^2=x^3+Ax^2+x$, A, B are field elements of GF(p)
  - To reduce the inversion, points in affine coordinate will be transformed in to standard projective format (c=1, d=1)

Differential addition with and Doubling with x coordinate

- Consider the point P with coordinate (X_1,Y_1,Z_1) in standard projective domain
- Then we define the x coordinate map as
  - x(P)=(X_1,Z_1) if P!=O
  - x(P)=(1,0) if P=O=(0,1,0)
- xADD(x(P),x(Q),x(P-Q))=x(P+Q) ==> **Differential Addition**
- xDBL(x(P))=x([2]P) ==> **Doubling**

Steps	$\mathbf{xADD(X_P, Z_P, X_Q, Z_Q, X_{P-Q}, Z_{P-Q}) = X_{P+Q}, Z_{P+Q}}$
1.	$T_0 = X_P + Z_P,\ T_1 = X_Q - Z_Q,\ T_1 = T_1.T_0$
2.	$T_0 = X_P - Z_P,\ T_2 = X_Q + Z_Q,\ T_2 = T_2.T_0$
3.	$T_3 = T_1 + T_2,\ T_3 = T_3^2,\ T_4 = T_1 - T_2$
4.	$T_4 = T_4^2,\ X_{P+Q} = Z_{P-Q}.T_3,\ Z_{P+Q} = X_{P-Q}.T_4$

Steps	$\mathbf{xDBL(X_P, Z_P) = X_{[2]P}, Z_{[2]P}}$
1.	$T_1 = X_P + Z_P,\ T_1 = T_1^2,\ T_2 = X_P - Z_P$
2.	$T_2 = T_2^2,\ X_{[2]P} = T_1.T_2,\ T_1 = T_1 - T_2$
3.	$T_3 = ((A+2)/4).T_1,\ T_3 = T_3 + T_2,\ Z_{[2]P} = T_1.T_3$

Montgomery Ladder for Scalar Multiplication

Algorithm 4: Scalar Multiplication using Montgomery Ladder

Data: Point P and scalar $k = k_{m-1}, k_{m-2}, k_{m-3} ... k_2, k_1, k_0$, where $k_{m-1} = 1$
Result: $Q = [k]P$

$R_0 = P$, $R_1 = [2]P$
for $i = m - 2$ *to* 0 **do**
  if $k_i == 0$ **then**
    $(R_0, R_1) = ([2]R_0, R_0 \oplus R_1)$
  else
    $(R_0, R_1) = (R_0 \oplus R_1, [2]R_0)$

Please note that at every iteration difference between R_0 and R_1 is always P

Example:Compute 10P

- $k=10=(1010)_2$,m=4, R_0=P, R_1=2P
- iteration i=2, k_i=0, R_0=2P, R_1=3P
- iteration i=1, k_i=1, R_0=5P, R_1=4P
- iteration i=0, k_i=0, R_0=10P, R_1=9P
- R_0 has the final result

Weakness:
We need to check if k_{m-1}=1 or not.
If we know, which branch is taken, we can get the secret scalar

Montgomery Curve: Curve25519

Algorithm ˆ Curve 25519 Montgomery Ladder

Input: $k = (k_{254}, \ldots, k_0)$, x_p s.t $P = (x_p, y_p)$
Output:x_q s.t $Q = [k]P = (x_q, y_q)$

$X_1 = x_p; X_2 = 1 : Z_2 = 0; X_3 = x_p; Z_3 = 1; k_{255} = 0$
for $i \leftarrow 254$ **to** 0 **do**
  $c \leftarrow k_{i+1} \oplus k_i$
  $(X_2, X_3) \leftarrow cswap(X_2, X_3, c)$, $(Z_2, Z_3) \leftarrow cswap(Z_2, Z_3, c)$
  $t_1 \leftarrow X_2 + Z_2$, $t_2 \leftarrow X_2 - Z_2$
  $t_3 \leftarrow X_3 + Z_3$, $t_4 \leftarrow X_3 - Z_3$
  $t_6 \leftarrow t_1^2$, $t_7 \leftarrow t_2^2$
  $t_5 \leftarrow t_6 - t_7$, $t_8 \leftarrow t_4.t_1$
  $t_9 \leftarrow t_3.t_2$, $t_{10} \leftarrow t_8 + t_9$
  $t_{11} \leftarrow t_8 - t_9$, $X_3 \leftarrow t_{10}^2$
  $t_{12} \leftarrow t_{11}^2$, $t_{13} \leftarrow 121666 t_5$
  $X_2 \leftarrow t_6.t_7$, $t_{14} \leftarrow t_7 + t_{13}$
  $Z_3 \leftarrow X_1.t_{12}$, $Z_2 \leftarrow t_5.t_{14}$
$(X_2, X_3) \leftarrow cswap(X_2, X_3, k_0)$, $(Z_2, Z_3) \leftarrow cswap(Z_2, Z_3, k_0)$
$Z_2 \leftarrow Z_2^{-1}$, $x_q \leftarrow X_2.Z_2$
return x_q

The curve equation for Montgomery curve is *E* :
$y^2 = x^3 + Ax^2 + x$
For Curve-25519, *A* is 486662 and the modulus is $2^{255}-19$

Faster Differential Addition formula compared to generic short Weierstrass curve
X coordinate only formula: Does not require y coordinate values during ladder step
Constant time and simple power attack secure

Scheduling of Montgomery Ladder Steps

Step No:	Modmul-1	Modmul-2	Addition	Subtraction
1	–	–	$t_1 = X_2 + Z_2$	$t_2 = X_2 - Z_2$
2	$t_6 = t_1^2$	$t_7 = t_2^2$	$t_3 = X_3 + Z_3$	$t_4 = X_3 - Z_3$
3	$t_8 = t_4 \cdot t_1$	$t_9 = t_3 \cdot t_2$	–	$t_5 = t_6 - t_7$
4	$X_2 = t_6 \cdot t_7$	$t_{13} = \frac{A-2}{4} \cdot t_5$	$t_{10} = t_8 + t_9$	$t_{11} = t_8 - t_9$
5	$X_3 = t_{10}^2$	$t_{12} = t_{11}^2$	$t_{14} = t_7 + t_{13}$	–
6	$Z_3 = X_1 \cdot t_{12}$	$Z_2 = t_5 \cdot t_{14}$	–	–

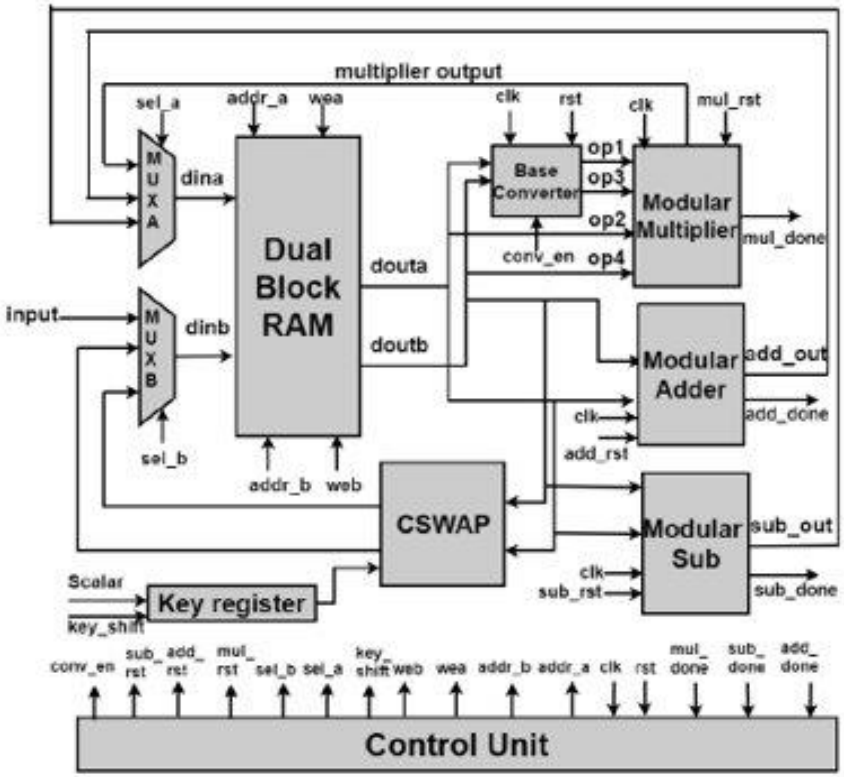
- **Scheduling with two multiplier has only one idle multiplicative step**
- **Scheduling with four multipliers has 3 idle multiplicative steps**
- **Step 6 of scheduling with two multipliers take 31 cycles**
- **Step 6 of scheduling with four multipliers take 63 cycles**

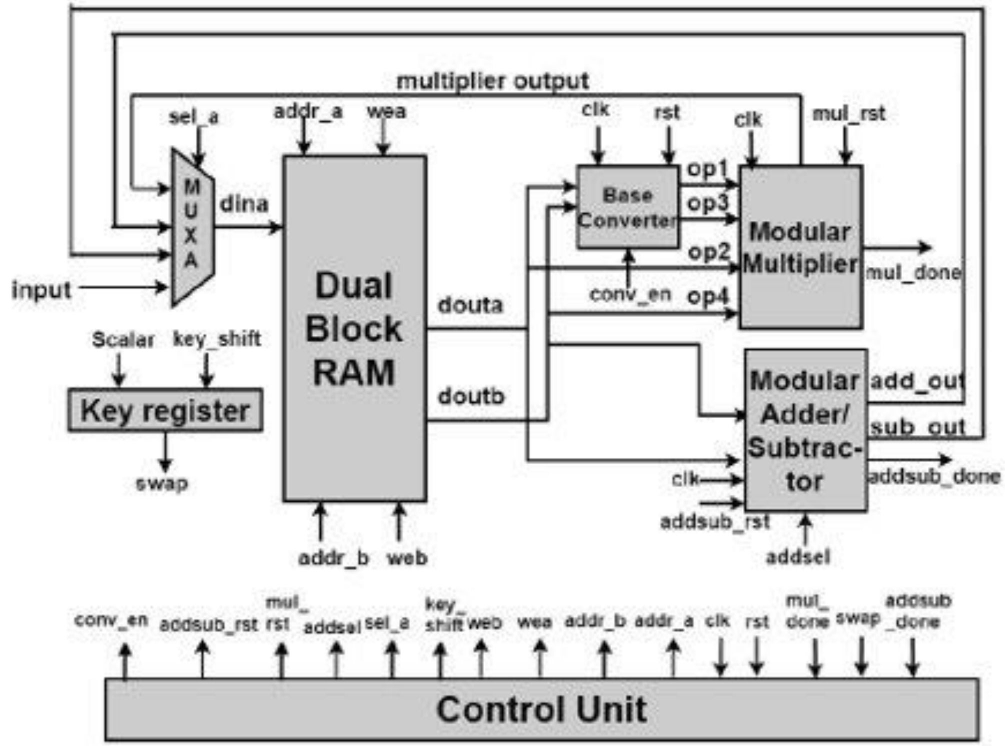
The improvement in the critical path is nullified by the increment in the clock cycles

Step No:	Mul-1	Mul-2	Mul-3	Mul-4	Add-1	Sub-1	Add-2	Sub-2
1	–	–	–	–	$t_1 = X_2 + Z_2$	$t_2 = X_2 - Z_2$	$t_3 = X_3 + Z_3$	$t_4 = X_3 - Z_3$
2	$t_6 = t_1^2$	$t_7 = t_2^2$	$t_8 = t_4 \cdot t_1$	$t_9 = t_3 \cdot t_2$	–	–	–	–
3	–	–	–	–	$t_{10} = t_8 + t_9$	$t_{11} = t_8 - t_9$	–	$t_5 = t_6 - t_7$
4	$X_3 = t_{10}^2$	$t_{12} = t_{11}^2$	$t_{13} = \frac{A-2}{4} \cdot t_5$	$X_2 = t_6 \cdot t_7$	–	–	–	–
5	–	–	–	–	$t_{14} = t_7 + t_{13}$	–	–	–
6	$Z_3 = X_1 \cdot t_{12}$	$Z_2 = t_5 \cdot t_{14}$	–	–	–	–	–	–

Proposed ECC Architecture

Initial Attemmpt	Final Low Area Design





	Component	Used	Available	Utilization
Low Area	Registers	4632	106400	4.35%
	LUTs	4567	53200	8.58%
	Slices	1928	13300	14.50%
	DSP48E1	40	220	18.19%
	Block Rams	9	140	6.42%

ECC Clock Cycle Requirement

Low Area Design (Single Scalar Multiplication)	**Modular Addition**	**10@181 MHz**	**55.25 ns**
	2 Modular Multiplication	31@181 MHz	171.3 ns
	Single Iteration of Montgomery ladder	205@181 MHz	1132.6 ns
	Scalar Multiplication Loop	52275@181 MHz	288819.4 ns
	Field Inversion	9435@181 MHz	52128 ns
	Range Correction	264@181 MHz	1458.6 ns
	Complete Scalar Multiplication	62084@181 MHz	343104
Two Parallel Scalar Multiplier	**Modular Addition**	**10@268.1 MHz**	**37.3 ns**
	4 Modular Multiplication	63@268.1 MHz	234.9ns
	Single Iteration of two parallel steps ofMontgomery ladder	408@268.1 MHz	1521.9 ns
	Scalar Multiplication Loop for 2 scalar multiplicatiom	104040@268.1 MHz	388069 ns
	Field Inversion	18359@268.1 MHz	68479 ns
	Range Correction	714@268.1 MHz	2663.22 ns
	Two Scalar Multiplication	123187@268.1MHz	460000
	One Scalar Multiplication	61593@268.1 MHz	230000

Final Result And Comparison

Architecture	Slices	LUTS	FFs	DSPs	BRAMs	Platform	Freq. (MHz)	Latency (micro-s.)
Low Area	1928	4567	4632	40	9	Zynq-7020	181	343.1
Two parallel Scalar Multiplier	2020	4797	7521	40	9	Zynq-7020	268.1	460 (two scalar mult.)
[1] Single Core	1029	2783	3592	20	2	Zynq-7020	200	397
[2]	8639	21107	26483	260	0	Zynq-7030	115	118
[3]	6161	17939	21077	175	0	Zynq-7030	114	92

[1] P. Sasdrich and T. Güneysu, Efficient Elliptic-Curve Cryptography Using Curve25519 Reconfigurable Devices. Cham: Springer, 2014, pp. 25–36. doi: 10.1007/978-3-319-05960-0_3
[2] P. Koppermann, F. De Santis, J. Heyszl, and G. Sigl, "X25519 hardware implementation for low-latency applications," in Proc. Euromicro Conf. Digit. Syst. Design (DSD), 2016, pp. 99–106.
[3] P. Koppermann, F. De Santis, J. Heyszl, and G. Sigl, "Low-latency x25519 hardware implementation: Breaking the 100 microseconds barrier," Microprocessors Microsyst., vol. 52, pp. 491–497, Jul. 2017

High Speed Implementation of ECC Scalar Multiplication in GF(p) for Generic Montgomery Curves: Debapriya Basu Roy, Debdeep Mukhopadhyay, published in IEEE-TVLSI, 2019

Extension to short Weierstrass Curve

- The overhead of the architecture which supports scalar multiplication in both Montgomery and short Weierstrass curves is 5079 LUTs, 7510 flip-flops, 2223 slices, 40 DSPs and 9 BRAMs.
- The critical path become 4.8 ns (208.3 MHz)
- The total clock cycle requirement to perform two parallel scalar multiplications is 191070 and the corresponding latency is 918 μs.

What We Achieved?

ECC architecture for generic Montgomery Curve

- Have used previously discussed Montgomery multiplier in redundant number system performing two parallel multiplications
- Proposed efficient scheduling for performing two parallel scalar multiplication
- Result shows comparable performance with existing designs of Curve-25519 with the added advantage of flexibility in curve choice
- Can be extended to short Weierstrass curves also