Integer and Prime Field Arithmetic

October 9, 2023
Ahmet Can Mert
ahmet.mert@iaik.tugraz.at

Integer and Prime Field Arithmetic

- All cryptographic operations are based on the arithmetic of number and polynomial groups, rings and fields.
  - RSA and ECC: *Large integer arithmetic.*
  - AES: *Finite field ($\mathbb{GF}(2^8)$) arithmetic.*
  - PQC, HE, ZKP: *Prime field ($\mathbb{GF}(p)$) and polynomial arithmetic.*

Integer and Prime Field Arithmetic

- All cryptographic operations are based on the arithmetic of number and polynomial groups, rings and fields.
  - RSA and ECC: *Large integer arithmetic.*
  - AES: *Finite field ($\mathbb{GF}(2^8)$) arithmetic.*
  - PQC, HE, ZKP: *Prime field ($\mathbb{GF}(p)$) and polynomial arithmetic.*

- For designing efficient software and hardware:
  - Mathematical properties of elements.
  - Efficient representation methods of elements .
  - Algorithms of for arithmetic operations.

Integer and Prime Field Arithmetic

- Cryptographic protocols targets minimizing arithmetic operations for efficiency (without scarifying the security of the protocol).

Integer and Prime Field Arithmetic

- Cryptographic protocols targets minimizing arithmetic operations for efficiency (without scarifying the security of the protocol).

- Efficient implementation of finite field or ring arithmetic leads to efficient cryptographic implementation.

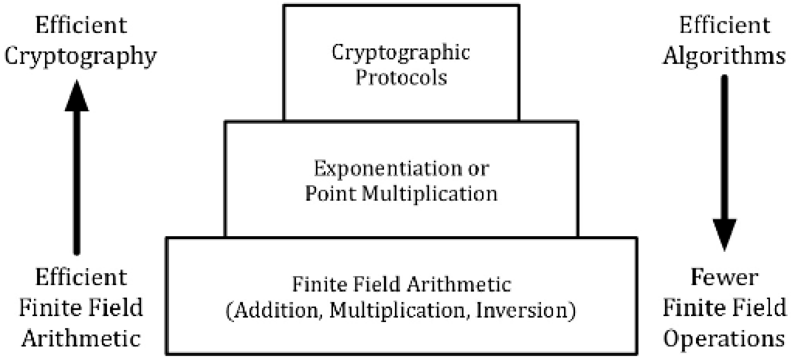


Image is retrieved from https://koclab.net/ffa.html

Integer and Prime Field Arithmetic

- Example problems:

Problem: Design a **multiplier circuit** that takes two 256-bit integers as input and generates 512-bit integer as output.

- Target/Specifications: High performance or low area?
- Algorithm, Resources (DSP, LUT or Hybrid), ...

Integer and Prime Field Arithmetic

- Example problems:

Problem: Design a **multiplier circuit** that takes two 256-bit integers as input and generates 512-bit integer as output.

- Target/Specifications: High performance or low area?
- Algorithm, Resources (DSP, LUT or Hybrid), ...

Problem: Design a **modular reduction circuit** for 256-bit prime 115792089237316195423570984634543488696558837605497246864089130975994398638081. The circuit takes one 500-bit integer as input and performs *(mod p)* operation.

Integer and Prime Field Arithmetic

- Most cryptographic algorithms are built upon mathematics of finite sets of integers.
  - Set of positive integers *modulo* q, $Z_q = \{0, 1, \ldots, q\text{-}1\}$
  - Fields $\mathrm{GF}(q^m)$

Integer and Prime Field Arithmetic

- Most cryptographic algorithms are built upon mathematics of finite sets of integers.
  - Set of positive integers *modulo* q, Z_q = {0, 1, …, q-1}
  - Fields $\mathbb{GF}(q^m)$

- When q is prime and m is 1, we have prime finite field.
  - $m>1$ gives us extension fields (e.g., AES)

- Finite field properties:
  - Closed
  - Associative / Commutative: $(a \,.\, b) \,.\, c = a \,.\, (b \,.\, c)$ / $a \,.\, b = b \,.\, a$
  - Identity: $a \,.\, 1 = a$
  - Inverse: $a \,.\, a^{-1} = 1$

Integer and Prime Field Arithmetic

- The arithmetic of such structures are often called **modular arithmetic.**
  - In cryptography, addition/subtraction, multiplication and inversion *mod q* are operations of interest.

Integer and Prime Field Arithmetic

- The arithmetic of such structures are often called **modular arithmetic.**
  - In cryptography, addition/subtraction, multiplication and inversion *mod q* are operations of interest.

- Example: GF(5) : {0, 1, 2, 3, 4}

  - +: 3 + 3 (mod 5) = 1
  - - : 1 - 3 (mod 5) = 3
  - *: 2 * 4 (mod 5) = 3
  - / (inverse): 3 * 2 (mod 5) = 1 $\longrightarrow$ 3^{-1} (mod 5) = 2

Modular Addition

- Computation of $A + B \ (mod\ q)$
  - Add and reduce:

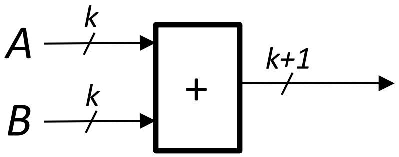
Input: $A, B < q, q$
Output: $C = A + B$ (mod q)
1: $t = A + B$
2: $s = t - q$
3: if ($s \geq 0$) **then** $C = s$ **else** $C = t$
4: return C

- Sign detection: $s \geq 0$?

Modular Addition

- Computation of $A + B \ (mod\ q)$
  - Add and reduce:

Input: $A, B < q, q$
Output: $C = A + B$ (mod q)
1: $t = A + B$
2: $s = t - q$
3: if ($s \geq 0$) **then** $C = s$ **else** $C = t$
4: return C

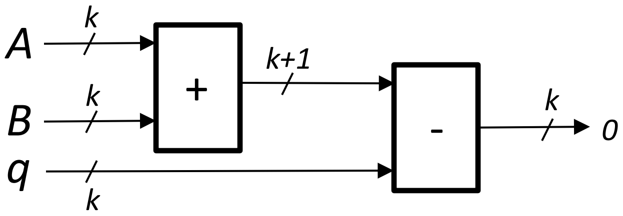


- Sign detection: $s \geq 0$?

Modular Addition

- Computation of $A + B \ (mod\ q)$
  - Add and reduce:

Input: $A,\ B < q,\ q$
Output: $C = A + B$ (mod q)
1: $t = A + B$
2: $s = t - q$
3: if ($s \geq 0$) **then** $C = s$ **else** $C = t$
4: return C

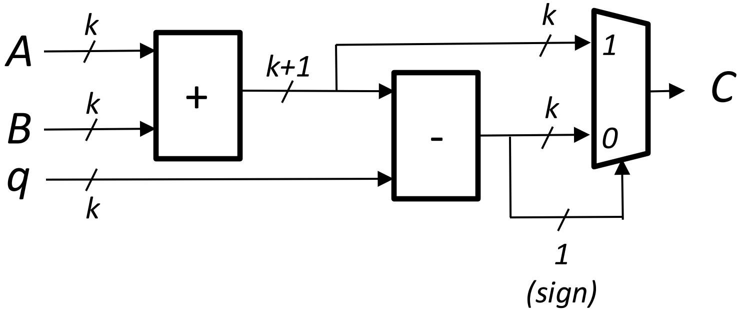


- Sign detection: $s \geq 0$?

Modular Addition

- Computation of *A + B (mod q)*
  - Add and reduce:

Input: $A, B < q, q$
Output: $C = A + B$ (mod q)
1: $t = A + B$
2: $s = t - q$
3: if ($s \geq 0$) **then** $C = s$ **else** $C = t$
4: return C



- Sign detection: $s \geq 0$?

Modular Subtraction

- Computation of $A - B \ (mod\ q)$
  - Subtract and reduce:

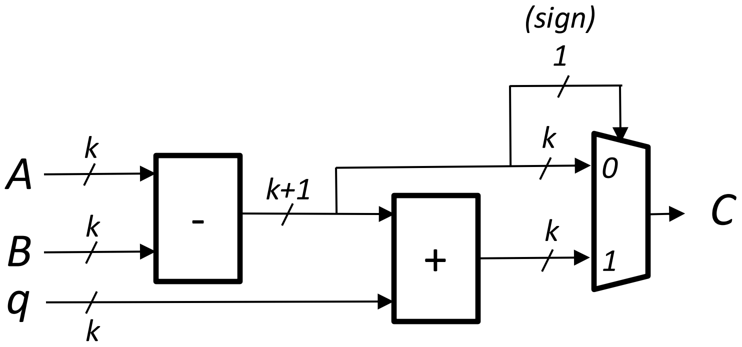
Input: $A, B < q, q$
Output: $C = A - B$ (mod q)
1: $t = A - B$
2: $s = t + q$
3: if ($t \geq 0$) **then** $C = t$ **else** $C = s$
4: return C

- Sign detection: $t \geq 0$?

Modular Subtraction

- Computation of *A - B (mod q)*
  - Subtract and reduce:

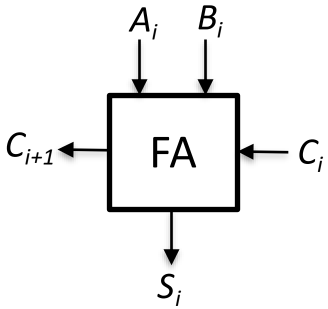
Input: $A, B < q, q$
Output: $C = A - B$ (mod q)
1: $t = A - B$
2: $s = t + q$
3: if ($t \geq 0$) **then** $C = t$ **else** $C = s$
4: return C



- Sign detection: $t \geq 0$?

Integer Addition

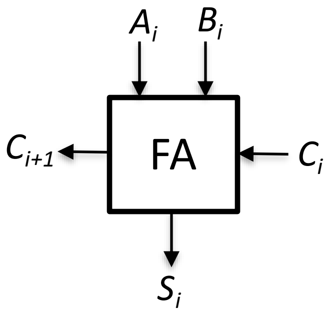
- Carry propagate adder (CPA) and Carry save adder (CSA)
  - Full Adder box:



$$S_i = A_i \oplus B_i \oplus C_i$$
$$C_{i+1} = A_i \cdot B_i + A_i \cdot C_i + B_i \cdot C_i$$

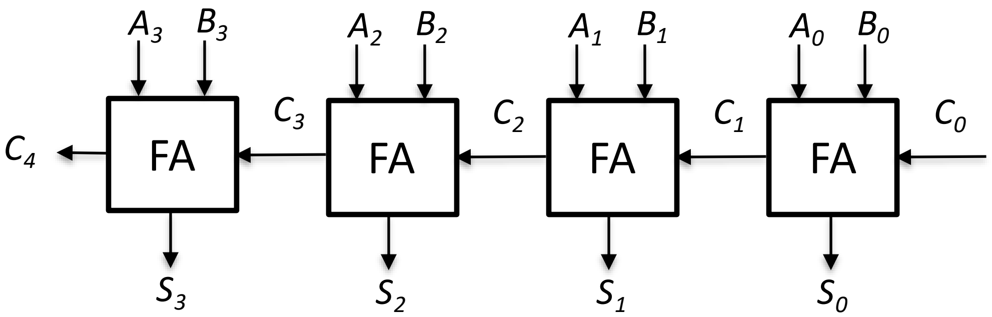
Integer Addition

- Carry propagate adder (CPA) and Carry save adder (CSA)
  - Full Adder box:



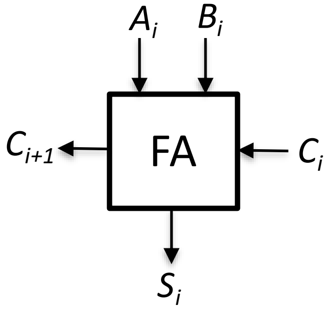
$$S_i = A_i \oplus B_i \oplus C_i$$
$$C_{i+1} = A_i \cdot B_i + A_i \cdot C_i + B_i \cdot C_i$$

  - CPA Topology:



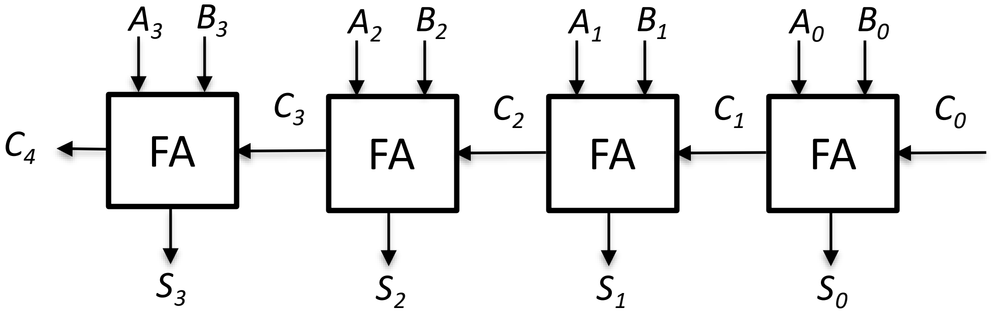
Integer Addition

- Carry propagate adder (CPA) and Carry save adder (CSA)
  - Full Adder box:



$$S_i = A_i \oplus B_i \oplus C_i$$
$$C_{i+1} = A_i \cdot B_i + A_i \cdot C_i + B_i \cdot C_i$$

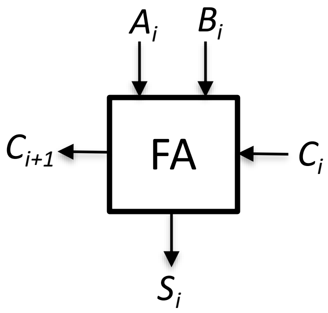
  - CPA Topology:



Total area: $k \cdot$ FA
Total delay: $k \cdot$ FA

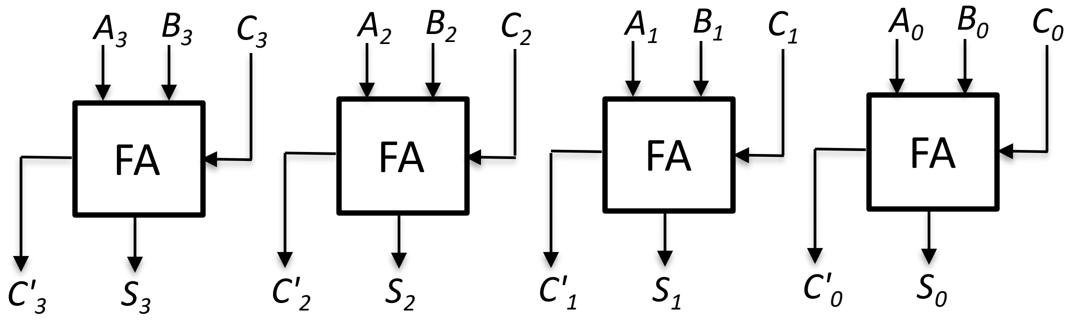
Integer Addition

- Carry propagate adder (CPA) and Carry save adder (CSA)
  - Full Adder box:



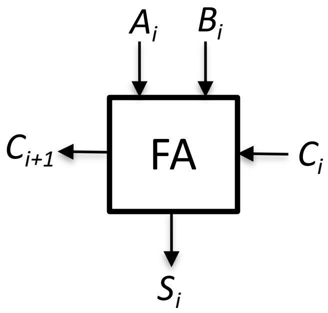
$$S_i = A_i \oplus B_i \oplus C_i$$
$$C_{i+1} = A_i \cdot B_i + A_i \cdot C_i + B_i \cdot C_i$$

  - CSA Topology:



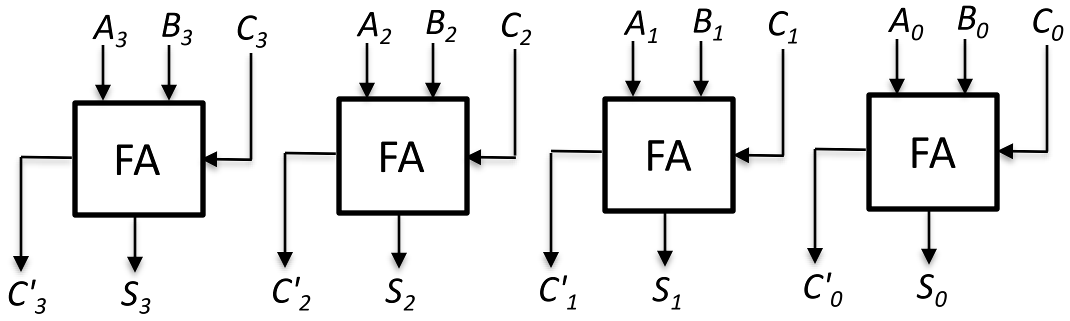
Integer Addition

- Carry propagate adder (CPA) and Carry save adder (CSA)
  - Full Adder box:



$$S_i = A_i \oplus B_i \oplus C_i$$
$$C_{i+1} = A_i \cdot B_i + A_i \cdot C_i + B_i \cdot C_i$$

  - CSA Topology:

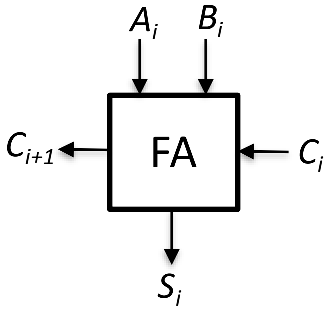


Example:

A	$=$	40	101000
B	$=$	25	011001
C	$=$	20	010100

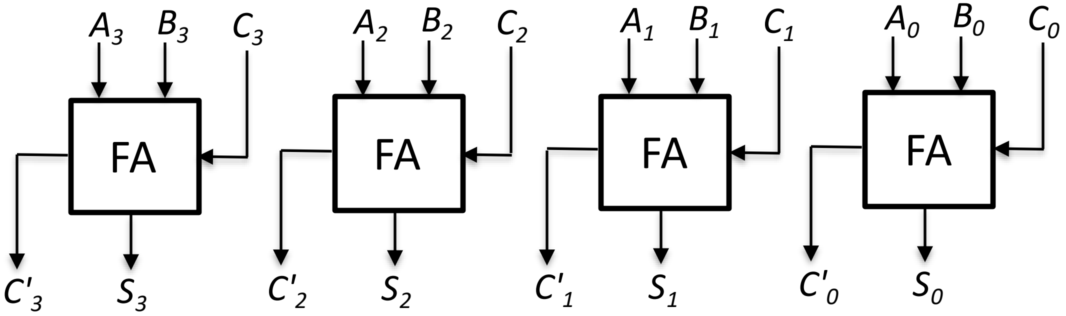
Integer Addition

- Carry propagate adder (CPA) and Carry save adder (CSA)
  - Full Adder box:



$$S_i = A_i \oplus B_i \oplus C_i$$
$$C_{i+1} = A_i \cdot B_i + A_i \cdot C_i + B_i \cdot C_i$$

  - CSA Topology:

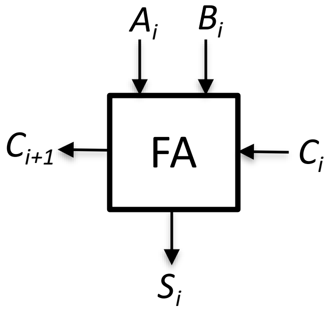


Example:

A	$=$	40	101000
B	$=$	25	011001
C	$=$	20	010100
S	$=$	37	100101
C'	$=$	48	011000

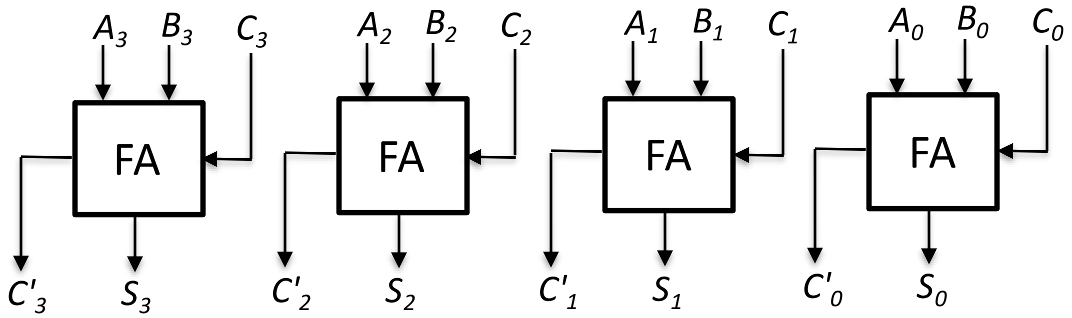
Integer Addition

- Carry propagate adder (CPA) and Carry save adder (CSA)
  - Full Adder box:



$$S_i = A_i \oplus B_i \oplus C_i$$
$$C_{i+1} = A_i \cdot B_i + A_i \cdot C_i + B_i \cdot C_i$$

  - CSA Topology:



Total area: $k \cdot$ FA
Total delay: $1 \cdot$ FA

Modular Multiplication

- Computation of $A \cdot B \ (mod\ q)$
  - Multiply and reduce:

- Integer Multiplication: $D = A \cdot B$

- Modular Reduction: $C = D \ (mod\ q)$

Integer Multiplication

- Most of PKC algorithms require (large) integer multiplication.
    - Multipliers with large bit length have a major impact on the performance.

Integer Multiplication

- Most of PKC algorithms require (large) integer multiplication.
  - Multipliers with large bit length have a major impact on the performance.

- Schoolbook (Standard) Approach:

$$
\begin{array}{rcccc}
 & 2 & 0 & 5 & 3 \\
\times & 1 & 1 & 7 & 6 \\
\hline
\end{array}
$$

Integer Multiplication

- Most of PKC algorithms require (large) integer multiplication.
  - Multipliers with large bit length have a major impact on the performance.

- Schoolbook (Standard) Approach:

			2	0	5	3
		x	1	1	7	6
			2.6	0.6	5.6	3.6
		2.7	0.7	5.7	3.7	
	2.1	0.1	5.1	3.1		
2.1	0.1	5.1	3.1			

Integer Multiplication

- Most of PKC algorithms require (large) integer multiplication.
  - Multipliers with large bit length have a major impact on the performance.

- Schoolbook (Standard) Approach:

			2	0	5	3
		x	1	1	7	6
			12	0	30	18
		14	0	35	21	
	2	0	5	3		
2	0	5	3			

Integer Multiplication

- Most of PKC algorithms require (large) integer multiplication.
  - Multipliers with large bit length have a major impact on the performance.

- Schoolbook (Standard) Approach:

				2	0	5	3
			x	1	1	7	6
		1		2	3	1	8
	1	4		3	7	1	
	2	0		5	3		
2	0	5		3			

Integer Multiplication

- Most of PKC algorithms require (large) integer multiplication.
  - Multipliers with large bit length have a major impact on the performance.

- Schoolbook (Standard) Approach:

				2	0	5	3
x				1	1	7	6
			1	2	3	1	8
		1	4	3	7	1	
		2	0	5	3		
+	2	0	5	3			
	2	4	1	4	3	2	8

Integer Multiplication: Divide-and-conquer Approach

- Divide a large multiplication into smaller chunks.
  - Multiply two n-bit (or digit) integers using ($n/2$)-bit multiplications

Integer Multiplication: Divide-and-conquer Approach

- Divide a large multiplication into smaller chunks.
  - Multiply two *n*-bit (or digit) integers using (*n/2*)-bit multiplications

  - Example: $a, b < r^n$ where r is the radix

Integer Multiplication: Divide-and-conquer Approach

- Divide a large multiplication into smaller chunks.
  - Multiply two *n*-bit (or digit) integers using (*n/2*)-bit multiplications

  - Example: $a, b < r^n$ where r is the radix

$$a = a_H \,.\, r^{n/2} + a_L$$
$$b = b_H \,.\, r^{n/2} + b_L$$

Integer Multiplication: Divide-and-conquer Approach

- Divide a large multiplication into smaller chunks.
  - Multiply two *n*-bit (or digit) integers using (*n/2*)-bit multiplications
  - Example: $a, b < r^n$ where r is the radix

$$a = a_H \, . \, r^{n/2} + a_L$$
$$b = b_H \, . \, r^{n/2} + b_L$$

$$a \, . \, b = (a_H \, . \, r^{n/2} + a_L) \, . \, (b_H \, . \, r^{n/2} + b_L)$$

Integer Multiplication: Divide-and-conquer Approach

- Divide a large multiplication into smaller chunks.
  - Multiply two *n*-bit (or digit) integers using (*n/2*)-bit multiplications

  - Example: $a, b < r^n$ where r is the radix

$$a = a_H \,.\, r^{n/2} + a_L$$
$$b = b_H \,.\, r^{n/2} + b_L$$

$$\begin{aligned} a \,.\, b &= (a_H \,.\, r^{n/2} + a_L) \,.\, (b_H \,.\, r^{n/2} + b_L) \\ &= a_H.b_H.r^n + a_H.b_L.r^{n/2} + a_L.b_H.r^{n/2} + a_L.b_L \\ &= a_H.b_H.r^n + (a_H.b_L. + a_L.b_H).r^{n/2} + a_L.b_L \end{aligned}$$

Integer Multiplication: Divide-and-conquer Approach

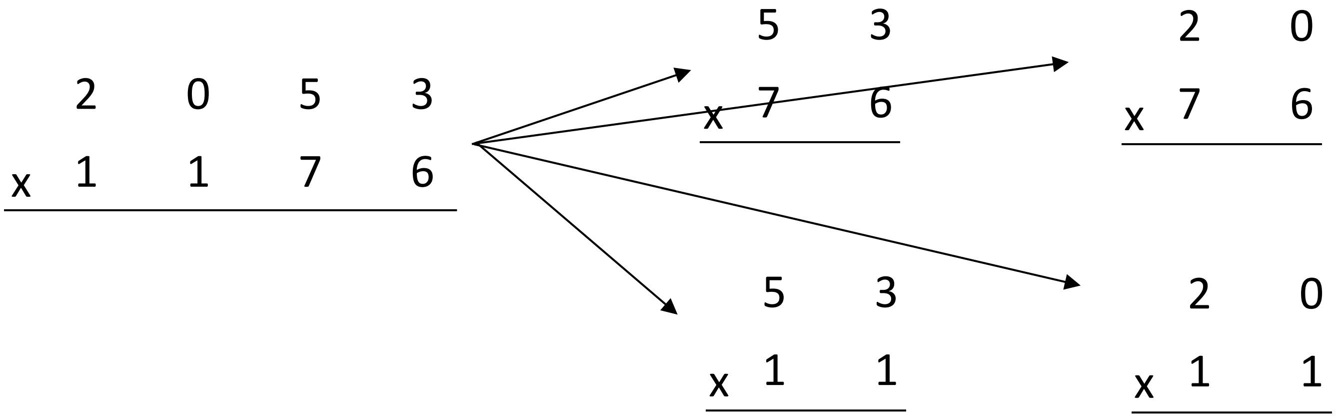
- Divide a large multiplication into smaller chunks.
  - Multiply two *n*-bit (or digit) integers using (*n/2*)-bit multiplications

  - Example: $a, b < 10^4$

$$
\begin{array}{rcccc}
 & 2 & 0 & 5 & 3 \\
\times & 1 & 1 & 7 & 6 \\
\hline
\end{array}
$$

Integer Multiplication: Divide-and-conquer Approach

- Divide a large multiplication into smaller chunks.
  - Multiply two *n*-bit (or digit) integers using (*n/2*)-bit multiplications
  - Example: $a, b < 10^4$



Integer Multiplication: Divide-and-conquer Approach

- Divide a large multiplication into smaller chunks.
  - Multiply two *n*-bit (or digit) integers using (*n/2*)-bit multiplications
  - Example: $a, b < 10^4$

$$\begin{array}{rcccc} & 2 & 0 & 5 & 3 \\ \times & 1 & 1 & 7 & 6 \\ \hline \end{array}$$

$$\begin{array}{rcc} & 5 & 3 \\ \times & 7 & 6 \\ \hline \end{array} \qquad \begin{array}{rcc} & 2 & 0 \\ \times & 7 & 6 \\ \hline \end{array}$$

$$\boxed{4\;0\;2\;8} \qquad \boxed{1\;5\;2\;0}$$

$$\begin{array}{rcc} & 5 & 3 \\ \times & 1 & 1 \\ \hline \end{array} \qquad \begin{array}{rcc} & 2 & 0 \\ \times & 1 & 1 \\ \hline \end{array}$$

$$\boxed{\;\;5\;8\;3} \qquad \boxed{\;\;2\;2\;0}$$

Integer Multiplication: Divide-and-conquer Approach

- Divide a large multiplication into smaller chunks.
  - Multiply two n-bit (or digit) integers using (n/2)-bit multiplications

  - Example: a, b < 10^4

$$\begin{array}{rcccc} & 2 & 0 & 5 & 3 \\ \times & 1 & 1 & 7 & 6 \\ \hline & \boxed{4} & \boxed{0} & \boxed{2} & \boxed{8} \end{array}$$

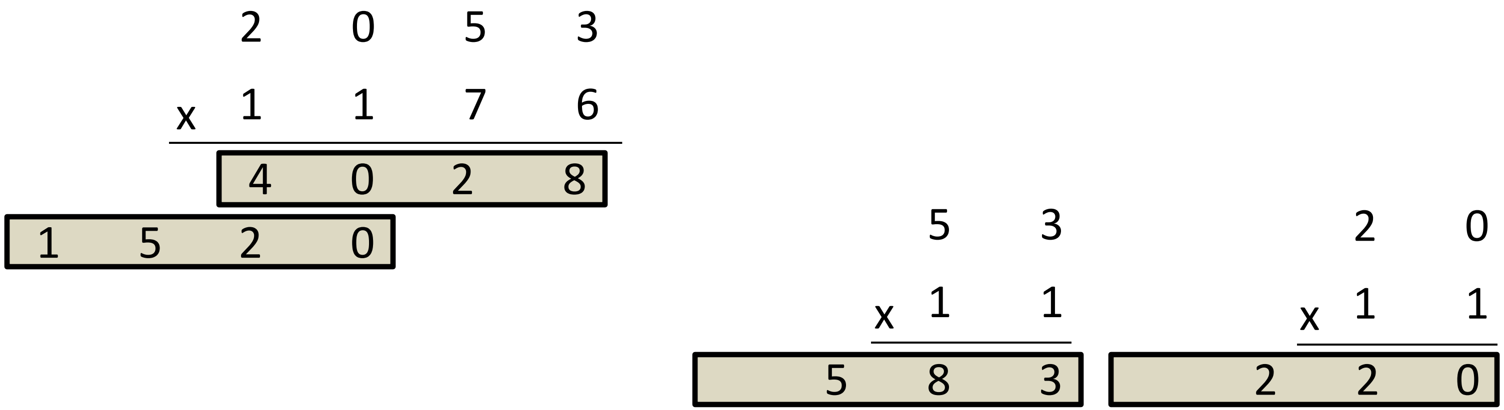
$$\begin{array}{rcccc} & & & 2 & 0 \\ & & \times & 7 & 6 \\ \hline & 1 & 5 & 2 & 0 \end{array}$$

$$\begin{array}{rcccc} & & & 5 & 3 \\ & & \times & 1 & 1 \\ \hline & & 5 & 8 & 3 \end{array} \qquad \begin{array}{rcccc} & & & 2 & 0 \\ & & \times & 1 & 1 \\ \hline & & 2 & 2 & 0 \end{array}$$

Integer Multiplication: Divide-and-conquer Approach

- Divide a large multiplication into smaller chunks.
  - Multiply two n-bit (or digit) integers using (n/2)-bit multiplications

  - Example: a, b < 10^4



Integer Multiplication: Divide-and-conquer Approach

- Divide a large multiplication into smaller chunks.
  - Multiply two n-bit (or digit) integers using (n/2)-bit multiplications

  - Example: a, b < 10^4

			2	0	5	3
		x	1	1	7	6
			4	0	2	8
1	5	2	0			
	5	8	3			

	2	0
x	1	1
2	2	0

Integer Multiplication: Divide-and-conquer Approach

- Divide a large multiplication into smaller chunks.
  - Multiply two n-bit (or digit) integers using (n/2)-bit multiplications

  - Example: a, b < 10^4

$$
\begin{array}{rccccccc}
 & & & & 2 & 0 & 5 & 3 \\
\times & & & & 1 & 1 & 7 & 6 \\
\hline
 & & & & 4 & 0 & 2 & 8 \\
 & & 1 & 5 & 2 & 0 & & \\
 & & & 5 & 8 & 3 & & \\
+ & 2 & 2 & 0 & & & & \\
\hline
 & 2 & 4 & 1 & 4 & 3 & 2 & 8
\end{array}
$$

Integer Multiplication: Divide-and-conquer Approach

- How to multiply two integers using Xilinx DSPs? How many DSPs are required?
  - One Xilinx DSP has 25-bit x 18-bit signed multiplier.

Integer Multiplication: Divide-and-conquer Approach

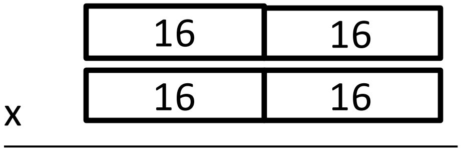
- How to multiply two integers using Xilinx DSPs? How many DSPs are required?
  - One Xilinx DSP has 25-bit x 18-bit signed multiplier.

32-bit integers

Integer Multiplication: Divide-and-conquer Approach

- How to multiply two integers using Xilinx DSPs? How many DSPs are required?
  - One Xilinx DSP has 25-bit x 18-bit signed multiplier.

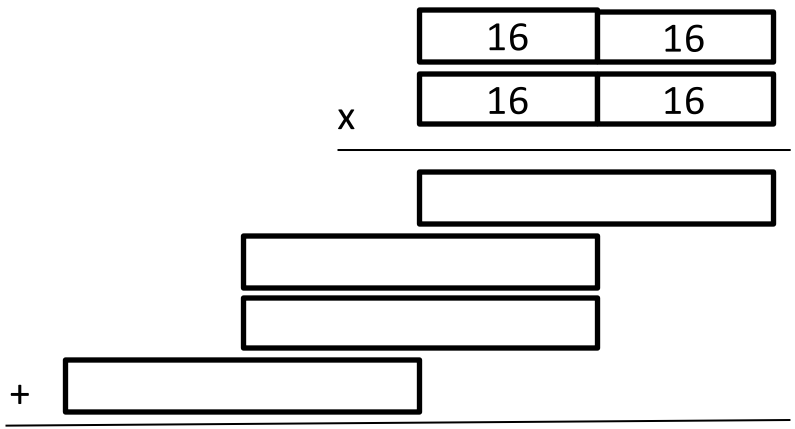
32-bit integers



Integer Multiplication: Divide-and-conquer Approach

- How to multiply two integers using Xilinx DSPs? How many DSPs are required?
  - One Xilinx DSP has 25-bit x 18-bit signed multiplier.

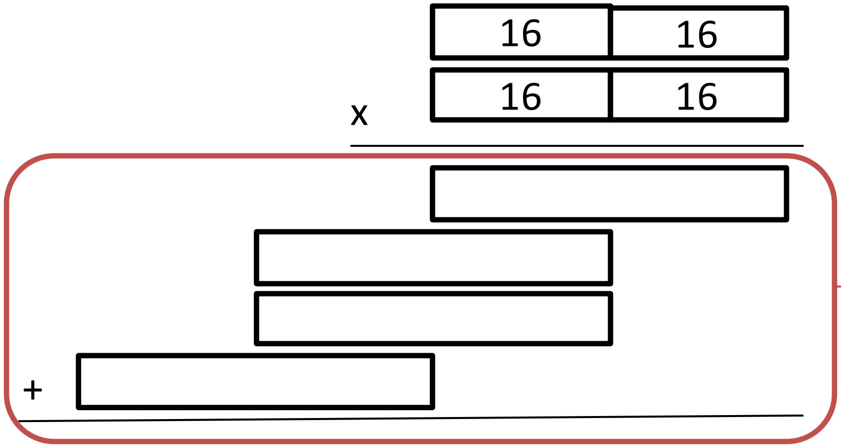
32-bit integers



Integer Multiplication: Divide-and-conquer Approach

- How to multiply two integers using Xilinx DSPs? How many DSPs are required?
  - One Xilinx DSP has 25-bit x 18-bit signed multiplier.

32-bit integers

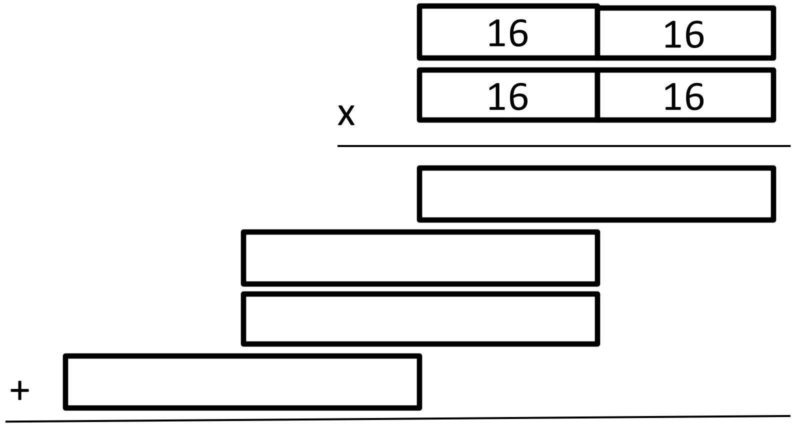


How to implement addition operation?

Integer Multiplication: Divide-and-conquer Approach

- How to multiply two integers using Xilinx DSPs? How many DSPs are required?
  - One Xilinx DSP has 25-bit x 18-bit signed multiplier.

32-bit integers



What about squaring?

Integer Multiplication: Divide-and-conquer Approach

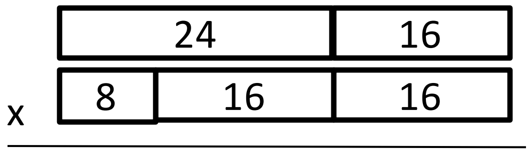
- How to multiply two integers using Xilinx DSPs? How many DSPs are required?
  - One Xilinx DSP has 25-bit x 18-bit signed multiplier.

40-bit integers

Integer Multiplication: Divide-and-conquer Approach

- How to multiply two integers using Xilinx DSPs? How many DSPs are required?
  - One Xilinx DSP has 25-bit x 18-bit signed multiplier.

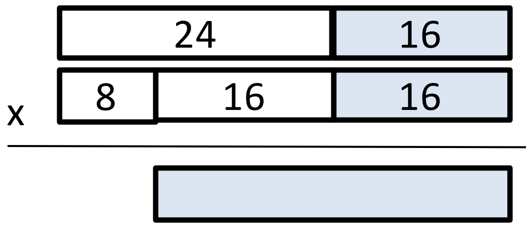
40-bit integers



Integer Multiplication: Divide-and-conquer Approach

- How to multiply two integers using Xilinx DSPs? How many DSPs are required?
    - One Xilinx DSP has 25-bit x 18-bit signed multiplier.

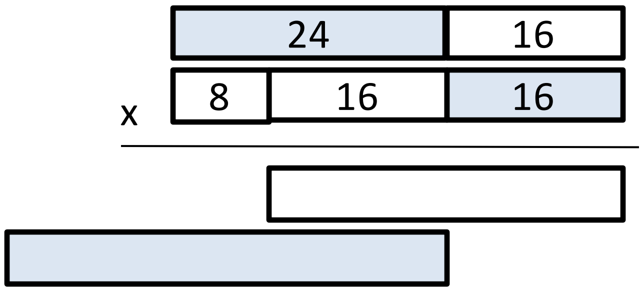
40-bit integers



Integer Multiplication: Divide-and-conquer Approach

- How to multiply two integers using Xilinx DSPs? How many DSPs are required?
  - One Xilinx DSP has 25-bit x 18-bit signed multiplier.

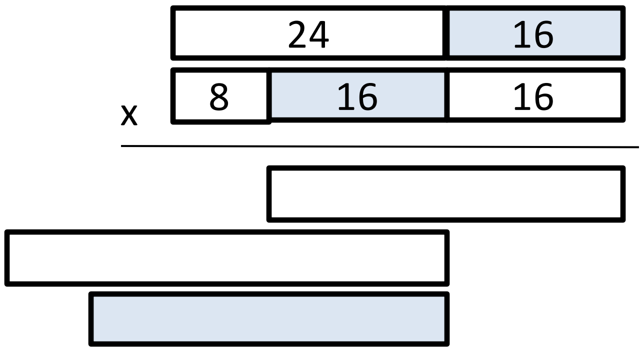
40-bit integers



Integer Multiplication: Divide-and-conquer Approach

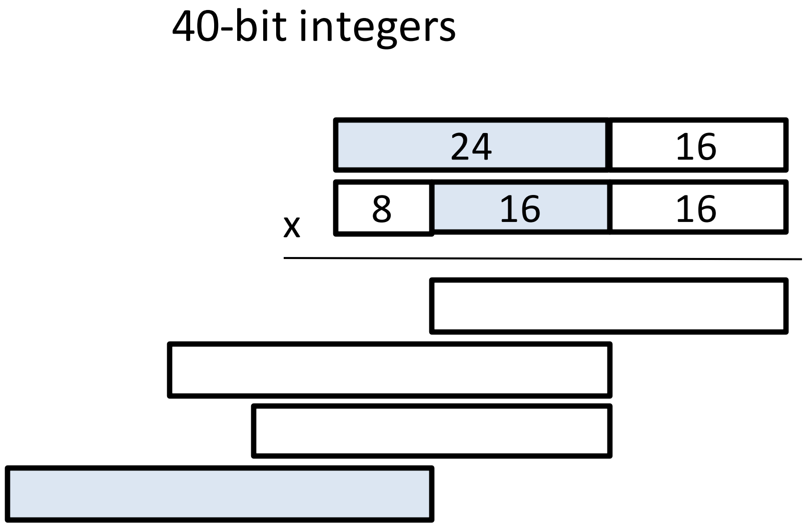
- How to multiply two integers using Xilinx DSPs? How many DSPs are required?
  - One Xilinx DSP has 25-bit x 18-bit signed multiplier.

40-bit integers



Integer Multiplication: Divide-and-conquer Approach

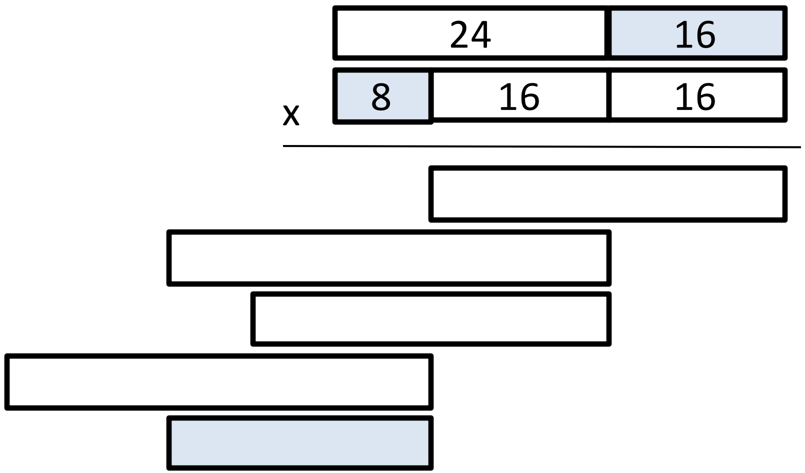
- How to multiply two integers using Xilinx DSPs? How many DSPs are required?
  - One Xilinx DSP has 25-bit x 18-bit signed multiplier.



Integer Multiplication: Divide-and-conquer Approach

- How to multiply two integers using Xilinx DSPs? How many DSPs are required?
  - One Xilinx DSP has 25-bit x 18-bit signed multiplier.

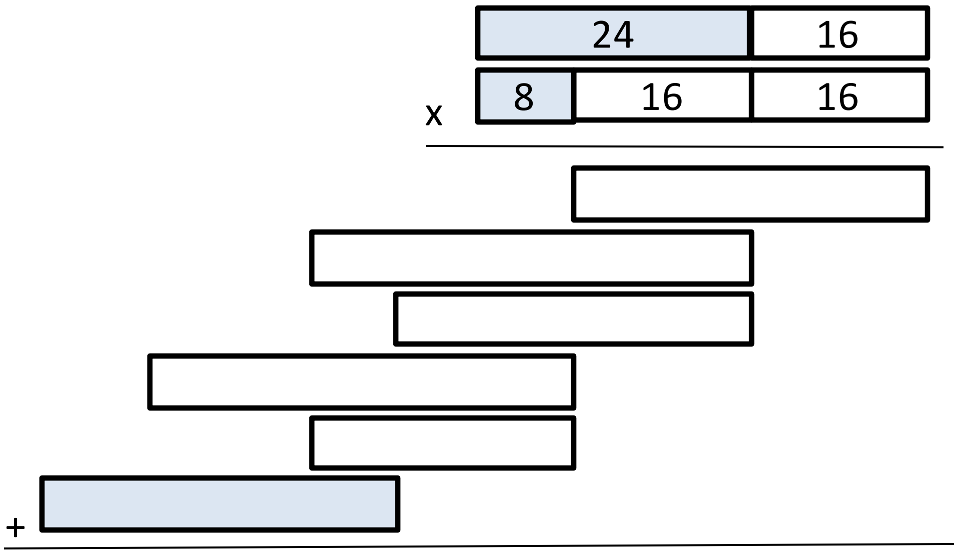
40-bit integers



Integer Multiplication: Divide-and-conquer Approach

- How to multiply two integers using Xilinx DSPs? How many DSPs are required?
  - One Xilinx DSP has 25-bit x 18-bit signed multiplier.

40-bit integers



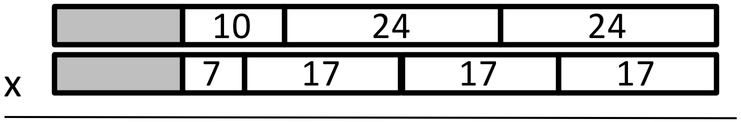
Integer Multiplication: Divide-and-conquer Approach

- Tiling problem (cover a given region using a given set of tiles without overlap).
  - Xilinx DSPs have asymmetric multipliers.
  - How to decompose inputs efficiently?

Integer Multiplication: Divide-and-conquer Approach

- Tiling problem (cover a given region using a given set of tiles without overlap)
  - Xilinx DSPs have asymmetric multipliers
  - How to decompose inputs efficiently?

- Example: 58-bit multiplication [1]

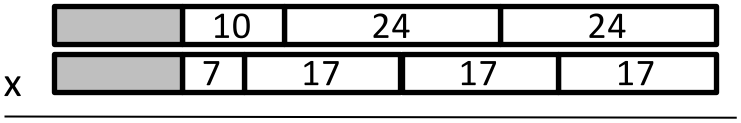


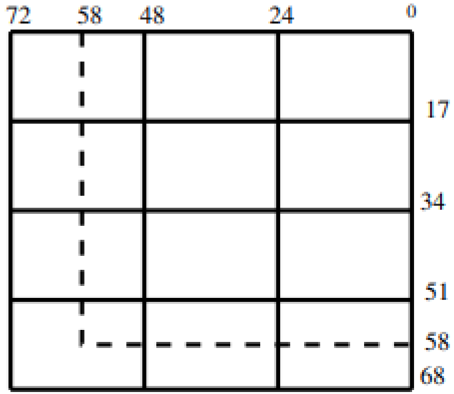
[1] Roy et al., *Tile Before Multiplication: An Efficient Strategy to Optimize DSP Multiplier for Accelerating Prime Field ECC for NIST Curves*. DAC, 2014.

Integer Multiplication: Divide-and-conquer Approach

- Tiling problem (cover a given region using a given set of tiles without overlap)
  - Xilinx DSPs have asymmetric multipliers
  - How to decompose inputs efficiently?

- Example: 58-bit multiplication [1]



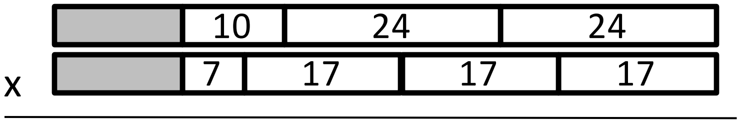


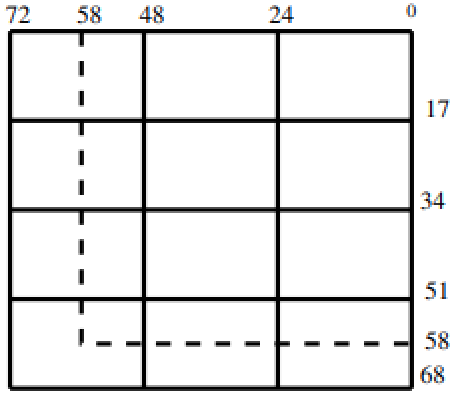
[1] Roy et al., *Tile Before Multiplication: An Efficient Strategy to Optimize DSP Multiplier for Accelerating Prime Field ECC for NIST Curves*. DAC, 2014.

Integer Multiplication: Divide-and-conquer Approach

- Tiling problem (cover a given region using a given set of tiles without overlap)
  - Xilinx DSPs have asymmetric multipliers
  - How to decompose inputs efficiently?

- Example: 58-bit multiplication [1]





Key observations:

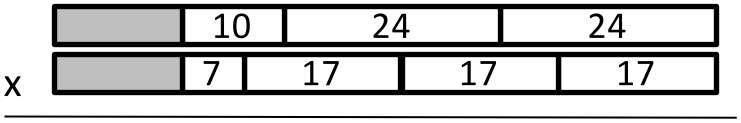
1. $\text{mul}_{best} = [(b \,.\, b) / (w1 \,.\, w2)]$

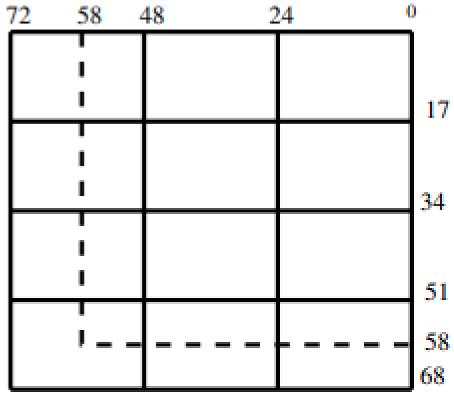
[1] Roy et al., *Tile Before Multiplication: An Efficient Strategy to Optimize DSP Multiplier for Accelerating Prime Field ECC for NIST Curves*. DAC, 2014.

Integer Multiplication: Divide-and-conquer Approach

- Tiling problem (cover a given region using a given set of tiles without overlap)
  - Xilinx DSPs have asymmetric multipliers
  - How to decompose inputs efficiently?

- Example: 58-bit multiplication [1]





Key observations:

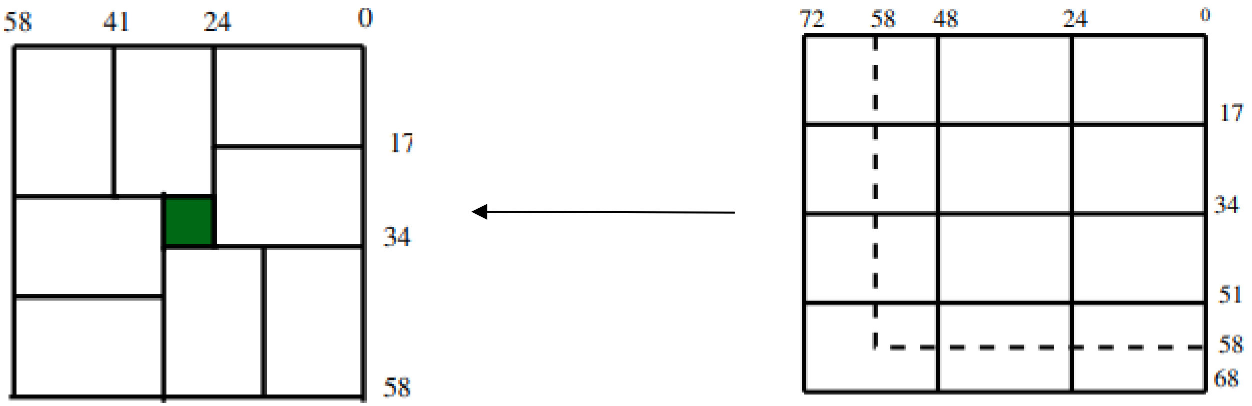
1. $mul_{best} = [(b \cdot b) / (w1 \cdot w2)]$
2. $b = m \cdot w1 + n \cdot w2$

[1] Roy et al., *Tile Before Multiplication: An Efficient Strategy to Optimize DSP Multiplier for Accelerating Prime Field ECC for NIST Curves*. DAC, 2014.

Integer Multiplication: Divide-and-conquer Approach

- Tiling problem (cover a given region using a given set of tiles without overlap)
  - Xilinx DSPs have asymmetric multipliers
  - How to decompose inputs efficiently?

- Example: 58-bit multiplication [1]

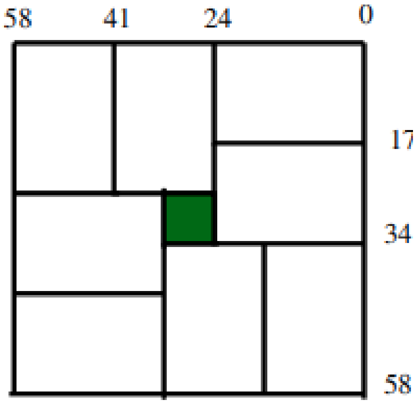


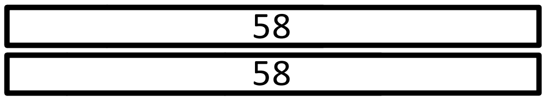
[1] Roy et al., *Tile Before Multiplication: An Efficient Strategy to Optimize DSP Multiplier for Accelerating Prime Field ECC for NIST Curves*. DAC, 2014.

Integer Multiplication: Divide-and-conquer Approach

- Tiling problem (cover a given region using a given set of tiles without overlap)
  - Xilinx DSPs have asymmetric multipliers
  - How to decompose inputs efficiently?

- Example: 58-bit multiplication [1]



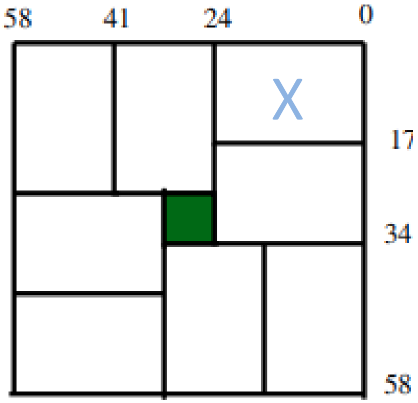


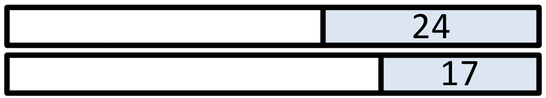
[1] Roy et al., *Tile Before Multiplication: An Efficient Strategy to Optimize DSP Multiplier for Accelerating Prime Field ECC for NIST Curves*. DAC, 2014.

Integer Multiplication: Divide-and-conquer Approach

- Tiling problem (cover a given region using a given set of tiles without overlap)
  - Xilinx DSPs have asymmetric multipliers
  - How to decompose inputs efficiently?

- Example: 58-bit multiplication [1]



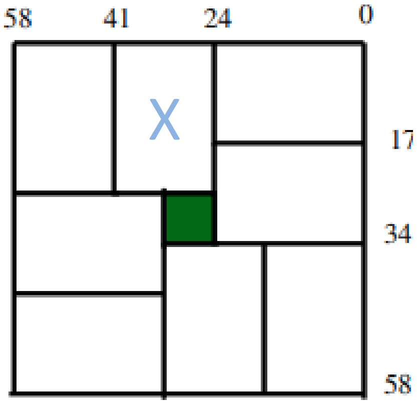


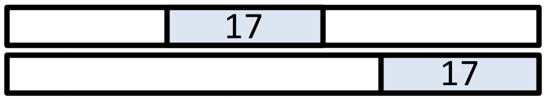
[1] Roy et al., *Tile Before Multiplication: An Efficient Strategy to Optimize DSP Multiplier for Accelerating Prime Field ECC for NIST Curves*. DAC, 2014.

Integer Multiplication: Divide-and-conquer Approach

- Tiling problem (cover a given region using a given set of tiles without overlap)
  - Xilinx DSPs have asymmetric multipliers
  - How to decompose inputs efficiently?

- Example: 58-bit multiplication [1]



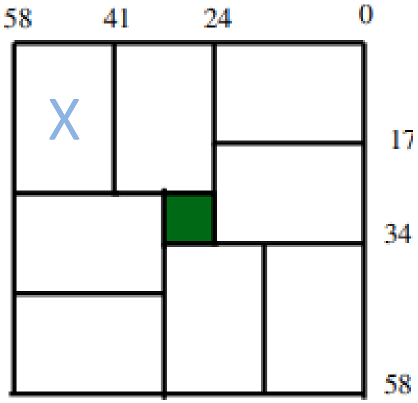


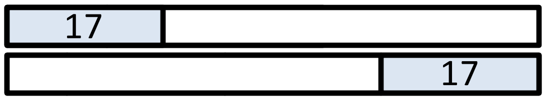
[1] Roy et al., *Tile Before Multiplication: An Efficient Strategy to Optimize DSP Multiplier for Accelerating Prime Field ECC for NIST Curves*. DAC, 2014.

Integer Multiplication: Divide-and-conquer Approach

- Tiling problem (cover a given region using a given set of tiles without overlap)
  - Xilinx DSPs have asymmetric multipliers
  - How to decompose inputs efficiently?

- Example: 58-bit multiplication [1]



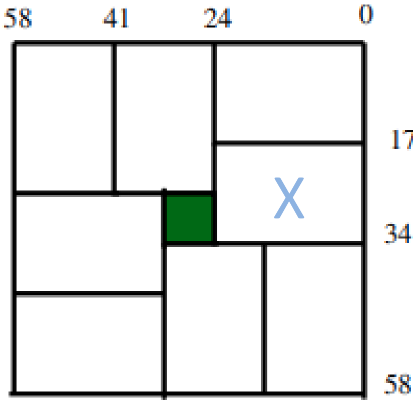


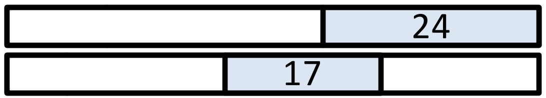
[1] Roy et al., *Tile Before Multiplication: An Efficient Strategy to Optimize DSP Multiplier for Accelerating Prime Field ECC for NIST Curves*. DAC, 2014.

Integer Multiplication: Divide-and-conquer Approach

- Tiling problem (cover a given region using a given set of tiles without overlap)
  - Xilinx DSPs have asymmetric multipliers
  - How to decompose inputs efficiently?

- Example: 58-bit multiplication [1]



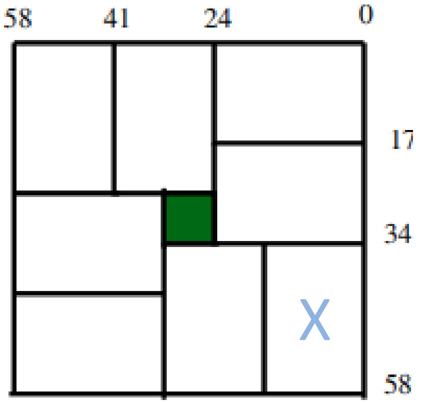


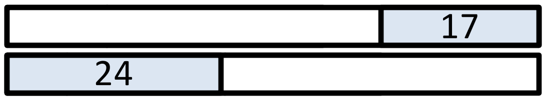
[1] Roy et al., *Tile Before Multiplication: An Efficient Strategy to Optimize DSP Multiplier for Accelerating Prime Field ECC for NIST Curves*. DAC, 2014.

Integer Multiplication: Divide-and-conquer Approach

- Tiling problem (cover a given region using a given set of tiles without overlap)
  - Xilinx DSPs have asymmetric multipliers
  - How to decompose inputs efficiently?

- Example: 58-bit multiplication [1]



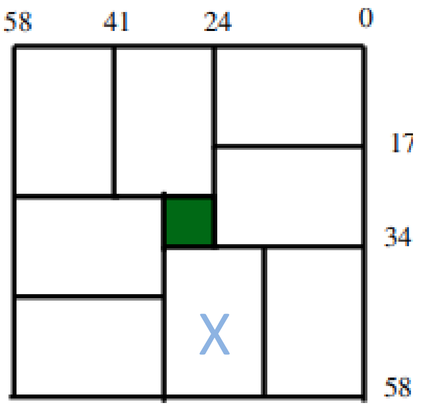


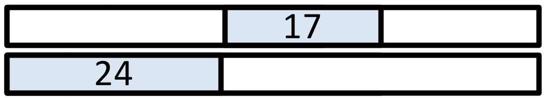
[1] Roy et al., *Tile Before Multiplication: An Efficient Strategy to Optimize DSP Multiplier for Accelerating Prime Field ECC for NIST Curves*. DAC, 2014.

Integer Multiplication: Divide-and-conquer Approach

- Tiling problem (cover a given region using a given set of tiles without overlap)
  - Xilinx DSPs have asymmetric multipliers
  - How to decompose inputs efficiently?

- Example: 58-bit multiplication [1]



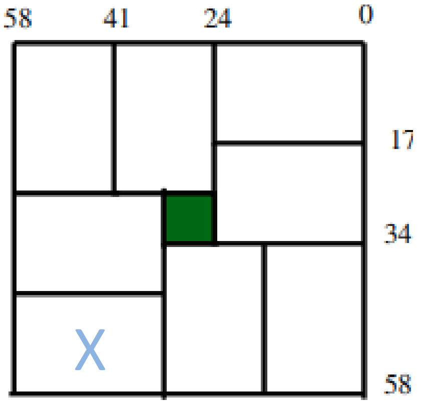


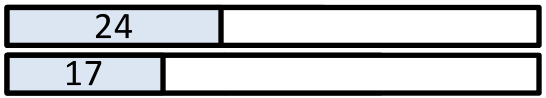
[1] Roy et al., *Tile Before Multiplication: An Efficient Strategy to Optimize DSP Multiplier for Accelerating Prime Field ECC for NIST Curves*. DAC, 2014.

Integer Multiplication: Divide-and-conquer Approach

- Tiling problem (cover a given region using a given set of tiles without overlap)
    - Xilinx DSPs have asymmetric multipliers
    - How to decompose inputs efficiently?

- Example: 58-bit multiplication [1]



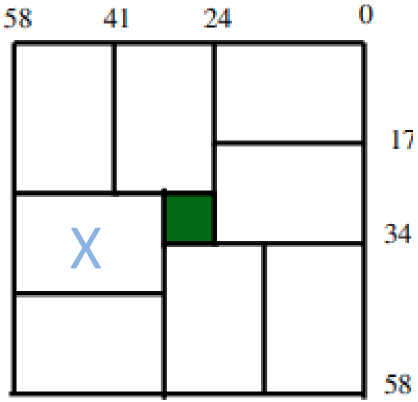


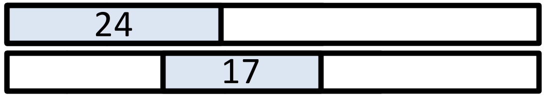
[1] Roy et al., *Tile Before Multiplication: An Efficient Strategy to Optimize DSP Multiplier for Accelerating Prime Field ECC for NIST Curves*. DAC, 2014.

Integer Multiplication: Divide-and-conquer Approach

- Tiling problem (cover a given region using a given set of tiles without overlap)
  - Xilinx DSPs have asymmetric multipliers
  - How to decompose inputs efficiently?

- Example: 58-bit multiplication [1]



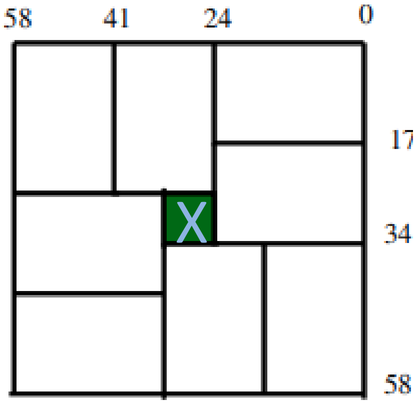


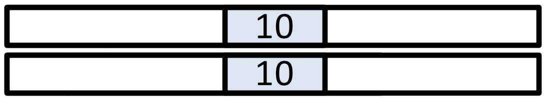
[1] Roy et al., *Tile Before Multiplication: An Efficient Strategy to Optimize DSP Multiplier for Accelerating Prime Field ECC for NIST Curves*. DAC, 2014.

Integer Multiplication: Divide-and-conquer Approach

- Tiling problem (cover a given region using a given set of tiles without overlap)
    - Xilinx DSPs have asymmetric multipliers
    - How to decompose inputs efficiently?

- Example: 58-bit multiplication [1]





[1] Roy et al., *Tile Before Multiplication: An Efficient Strategy to Optimize DSP Multiplier for Accelerating Prime Field ECC for NIST Curves*. DAC, 2014.

Integer Multiplication: Parallel and Sequential Architectures

- Sequential and Parallel Architectures
  - Single or multiple DSPs
  - Low-cost or High-throughput oriented design

- Example: 32-bit multiplier

Integer Multiplication: Karatsuba Algorithm

- Schoolbook method has $O(n^2)$ complexity.
- Karatsuba Algorithm uses a divide-and-conquer method and reduces complexity to $O(n^{1.58})$.

Integer Multiplication: Karatsuba Algorithm

- Schoolbook method has $O(n^2)$ complexity.
- Karatsuba Algorithm uses a divide-and-conquer method and reduces complexity to $O(n^{1.58})$.

$a, b < r^n$ where r is the radix

$a = a_H \,.\, r^{n/2} + a_L$
$b = b_H \,.\, r^{n/2} + b_L$
$a \,.\, b = a_H.b_H \,.\, r^n + (a_H.b_L + a_L.b_H).r^{n/2} + a_L.b_L$

Integer Multiplication: Karatsuba Algorithm

- Schoolbook method has $O(n^2)$ complexity.
- Karatsuba Algorithm uses a divide-and-conquer method and reduces complexity to $O(n^{1.58})$.

$a, b < r^n$ where r is the radix

$a = a_H \, . \, r^{n/2} + a_L$

$b = b_H \, . \, r^{n/2} + b_L$

$a \, . \, b = a_H . b_H \, . \, r^n + (a_H . b_L + a_L . b_H) . r^{n/2} + a_L . b_L = z_0 \, . \, r^n + (z_1 + z_2) . r^{n/2} + z_3$

Integer Multiplication: Karatsuba Algorithm

- Schoolbook method has $O(n^2)$ complexity.
- Karatsuba Algorithm uses a divide-and-conquer method and reduces complexity to $O(n^{1.58})$.

$a, b < r^n$ where r is the radix

$$a = a_H \,.\, r^{n/2} + a_L$$
$$b = b_H \,.\, r^{n/2} + b_L$$
$$a \,.\, b = a_H . b_H \,.\, r^n + (a_H . b_L + a_L . b_H) . r^{n/2} + a_L . b_L = z_0 \,.\, r^n + (z_1 + z_2) . r^{n/2} + z_3$$

Standard divide-and-conquer uses 4 multiplication.

Integer Multiplication: Karatsuba Algorithm

- Schoolbook method has $O(n^2)$ complexity.
- Karatsuba Algorithm uses a divide-and-conquer method and reduces complexity to $O(n^{1.58})$.

$a, b < r^n$ where r is the radix

$$a = a_H \, . \, r^{n/2} + a_L$$
$$b = b_H \, . \, r^{n/2} + b_L$$
$$a \, . \, b = a_H . b_H \, . \, r^n + (a_H . b_L + a_L . b_H) . r^{n/2} + a_L . b_L = z_0 \, . \, r^n + (z_1 + z_2) . r^{n/2} + z_3$$

Standard divide-and-conquer uses 4 multiplication.

$$1. \; z_0 = a_H . b_H$$

Integer Multiplication: Karatsuba Algorithm

- Schoolbook method has $O(n^2)$ complexity.
- Karatsuba Algorithm uses a divide-and-conquer method and reduces complexity to $O(n^{1.58})$.

$a, b < r^n$ where r is the radix

$$
\left.
\begin{aligned}
a &= a_H \,.\, r^{n/2} + a_L \\
b &= b_H \,.\, r^{n/2} + b_L \\
a \,.\, b &= a_H.b_H \,.\, r^n + (a_H.b_L + a_L.b_H).r^{n/2} + a_L.b_L = z_0 \,.\, r^n + (z_1 + z_2).r^{n/2} + z_3
\end{aligned}
\right\}
$$

Standard divide-and-conquer uses 4 multiplication.

1. $z_0 = a_H.b_H$
2. $z_3 = a_L.b_L$

Integer Multiplication: Karatsuba Algorithm

- Schoolbook method has $O(n^2)$ complexity.
- Karatsuba Algorithm uses a divide-and-conquer method and reduces complexity to $O(n^{1.58})$.

$a, b < r^n$ where r is the radix

$$\left.\begin{aligned}
&a = a_H \,.\, r^{n/2} + a_L \\
&b = b_H \,.\, r^{n/2} + b_L \\
&a \,.\, b = a_H.b_H \,.\, r^n + (a_H.b_L + a_L.b_H).r^{n/2} + a_L.b_L = z_0 \,.\, r^n + (z_1 + z_2).r^{n/2} + z_3
\end{aligned}\right\}$$

Standard divide-and-conquer uses 4 multiplication.

$$\left.\begin{aligned}
&1.\ z_0 = a_H.b_H \\
&2.\ z_3 = a_L.b_L \\
&3.\ z_1 + z_2 = (a_H + a_L)\,.\,(b_H + b_L) - z_0 - z_3
\end{aligned}\right\}$$

Karatsuba Algorithm uses 3 multiplication.

Integer Multiplication: Karatsuba Algorithm

- Karatsuba algorithm can be applied recursively.
  - How many DSPs are required for 58-bit multiplication?

Integer Multiplication: Literature

- Many works following Karatsuba's invention
  - Toom-Cook
  - Schonhage-Strassen
    - Uses FFT
  - Harvey's Method

Date	Authors	Time complexity
<3000 BC	Unknown [37]	$O(n^2)$
1962	Karatsuba [30, 31]	$O(n^{\log 3/\log 2})$
1963	Toom [51, 50]	$O(n\, 2^{5\sqrt{\log n/\log 2}})$
1966	Schönhage [45]	$O(n\, 2^{\sqrt{2\log n/\log 2}}\, (\log n)^{3/2})$
1969	Knuth [32]	$O(n\, 2^{\sqrt{2\log n/\log 2}} \log n)$
1971	Schönhage–Strassen [47]	$O(n \log n \log\log n)$
2007	Fürer [18]	$O\big(n \log n\, 2^{O(\log^* n)}\big)$
2014	This paper	$O(n \log n\, 8^{\log^* n})$

Table 1.1. Historical overview of known complexity bounds for n-bit integer multiplication.

* Harvey et al., *Even faster integer multiplication,* arXiv/1407.3360, 2014

- State-of-the-art (2019)

Integer multiplication in time $O(n \log n)$

David Harvey and Joris van der Hoeven

Abstract. We present an algorithm that computes the product of two n-bit integers in $O(n \log n)$ bit operations, thus confirming a conjecture of Schönhage and Strassen from 1971. Our complexity analysis takes place in the multitape Turing machine model, with integers encoded in the usual binary representation. Central to the new algorithm is a novel "Gaussian resampling" technique that enables us to reduce the integer multiplication problem to a collection of multidimensional discrete Fourier transforms over the complex numbers, whose dimensions are all powers of two. These transforms may then be evaluated rapidly by means of Nussbaumer's fast polynomial transforms.

Integer Multiplication: Constant Multiplication

- Sometimes, one of the operands is a fixed integer.
  - Using a generic integer multiplier will not be optimal.

Integer Multiplication: Constant Multiplication

- Sometimes, one of the operands is a fixed integer.
  - Using a generic integer multiplier will not be optimal.

  - Example: 24-bit multiplication: $(A \cdot 8519937)$

Integer Multiplication: Constant Multiplication

- Sometimes, one of the operands is a fixed integer.
  - Using a generic integer multiplier will not be optimal.

  - Example: 24-bit multiplication: $(A \cdot 8519937)$

    - DSP-based approach will require 2 DSPs.

Integer Multiplication: Constant Multiplication

- Sometimes, one of the operands is a fixed integer.
  - Using a generic integer multiplier will not be optimal.

  - Example: 24-bit multiplication: $(A \cdot 8519937)$

    - DSP-based approach will require 2 DSPs

    - $8519937 = 2^{23} + 2^{17} + 2^{8} + 1$
      $A \cdot 8519937 = A \cdot (2^{23} + 2^{17} + 2^{8} + 1)$
      $A \cdot 8519937 = A \cdot 2^{23} + A \cdot 2^{17} + A \cdot 2^{8} + A$

Integer Multiplication: Constant Multiplication

- Shift-Add based approach
  - Example: $C \cdot X$

$$C = \sum_{i=0}^{n-1} c_i 2^i$$

where c_i is {0, 1}

$$CX = \sum_{i=0}^{n-1} c_i 2^i X$$

$$C \cdot X = X \cdot c_0 \cdot 2^0 + X \cdot c_1 \cdot 2^1 + X \cdot c_2 \cdot 2^2 + \ldots$$

- Complexity depends on the number of 1s in the binary representation of *C.*

Integer Multiplication: Constant Multiplication

- Use different number representation/encoding.
    - Canonical Signed-Digit (CSD) (also called non-adjacent form) uses the digits {−1, 0, 1} to represent a number in such a way that no two adjacent digits are non-zero.

Integer Multiplication: Constant Multiplication

- Use different number representation/encoding.
  - Canonical Signed-Digit (CSD) (also called non-adjacent form) uses the digits $\{-1, 0, 1\}$ to represent a number in such a way that no two adjacent digits are non-zero.

- Example: implementation of $477 \cdot X$

$$
\begin{aligned}
477 \cdot X &= (111011101)_2 \cdot X \\
&= (X \ll 8) + (X \ll 7) + (X \ll 6) + (X \ll 4) + (X \ll 3) + (X \ll 2) + X
\end{aligned}
$$

Integer Multiplication: Constant Multiplication

- Use different number representation/encoding.
  - Canonical Signed-Digit (CSD) (also called non-adjacent form) uses the digits {−1, 0, 1} to represent a number in such a way that no two adjacent digits are non-zero.

- Example: implementation of $477 \cdot X$

$$
\begin{aligned}
477 \cdot X &= (111011101)_2 \cdot X \\
&= (X \ll 8) + (X \ll 7) + (X \ll 6) + (X \ll 4) + (X \ll 3) + (X \ll 2) + X
\end{aligned}
$$

$$
\begin{aligned}
477 \cdot X &= (1000\bar{1}00\bar{1}01)_2 \cdot X \\
&= (X \ll 9) - (X \ll 5) - (X \ll 2) + X
\end{aligned}
$$