Hardware design of cryptographic algorithms

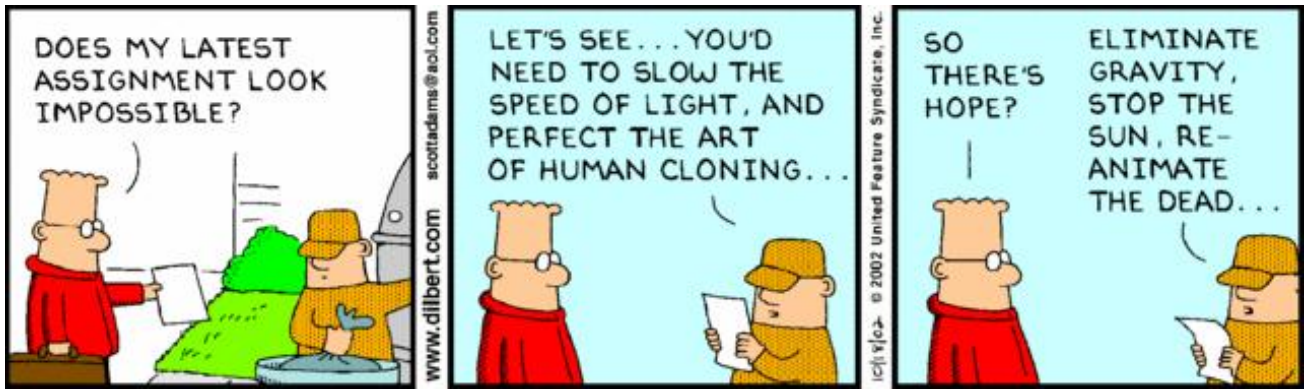
Francisco Rodríguez-Henríquez
CINVESTAV-IPN, México
francisco@cs.cinvestav.mx

Tutorial Talk at Indocrypt 2012 - Sunday December 9th, 2012

Outline of the talk

Cinvestav
Centro de Investigación y de Estudios Avanzados del Instituto Politécnico Nacional
50 AÑOS

But.... why should one bother implementing crypto-schemes in Hardware?



But.... Why should one bother implement pairings in Hardware?

- computation not very well suited for general purpose processor

But.... Why should one bother implement pairings in Hardware?

- computation not very well suited for general purpose processor
- There exist specific targets, one of the most prominent ones being smart cards

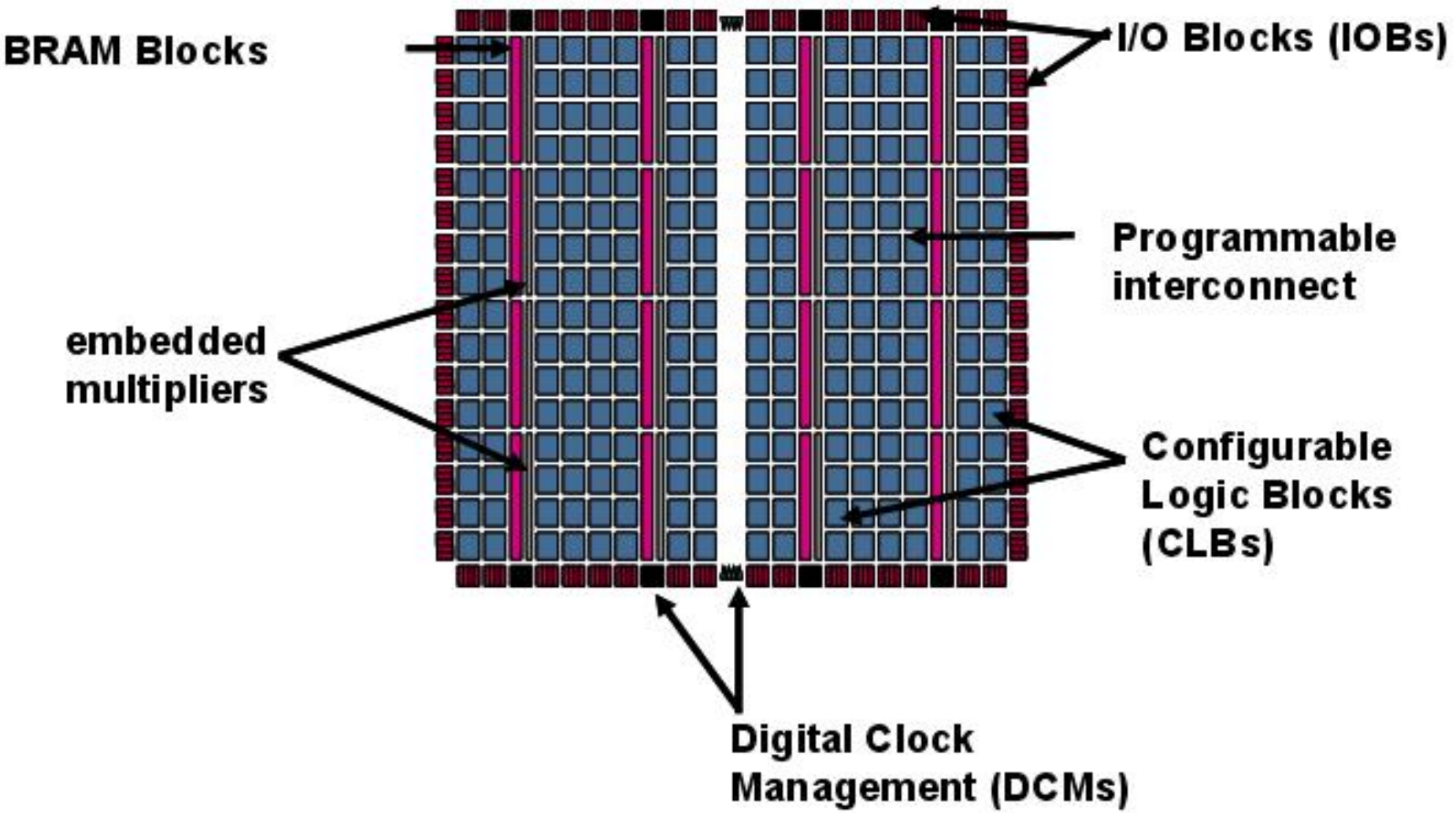
But.... Why should one bother implement pairings in Hardware?

- computation not very well suited for general purpose processor
- There exist specific targets, one of the most prominent ones being smart cards
- Hardware may be the fastest/most efficient way to implement cryptographic algorithms.

But.... Why should one bother implement pairings in Hardware?

- computation not very well suited for general purpose processor
- There exist specific targets, one of the most prominent ones being smart cards
- Hardware may be the fastest/most efficient way to implement cryptographic algorithms.
- However if a cryptographic hardware accelerator is going to be attractive at all, a significant performance improvement should be observed with respect to software implementations.

FPGA General architecture



Main Xilinx FPGA families

- Kintex 7 economical models for high performance applications
- Virtex 5 and 6 some of the most sophisticated models. the Look-up Tables can be configurated with up to six inputs
- Artix 7 some of the models include a dual-core ARM Cortex-A9
- Spartan 6 economical and simple devices
- Virtex 7 the newest family with many embedded components working at high speed. It is still costly

Main characteristics of Xilinx FPGA Families [as of 2006]

	Virtex-5	Virtex-4	Virtex II Pro	Spartan 3 & 3E
Logic Cells	up to 330K	12K-200K	3K-99K	1.7K-74K
BRAM (18Kbits each)	576	36-512	12-444	4-104
Multipliers	$32 - 192$[1]	32-512	12-444	4-104
DCM	up to 18	4-20	4-12	2-18
IOBs	up to 1200	240-960	204-1164	63-633
DSP Slices	32-192	32-192	—	–
PowerPC Blocks	N/A	0-2	0-2	–
Max. freq.	550MHz	500MHz	547 MHz	up to 300MHz
Price	$\approx \$400 USD$	From \$300	From \$139	From \$2 up to \$85

[1] 25×18 embedded multipliers

FPGA Manufacturers and Their Devices

Manufacturer	FPGA Family	Feature
Xilinx	Virtex-4-Virtex-7, VirtexII, Spartan III	FPGA market leader 28ηm technology
Altera	Stratix, Stratix II, Cyclone	28ηm technology
Lattice	LatticeXP	ultra low power for mobile applications
Actel	Fusion, M7Fusion	first mixed-signal FPGA
Quick Logic	Eclipse II	programmable-only-once FPGA
Atmel	AT40KAL	fine-grain reconfigurable
Achronix	Achronix-ULTRA	1.6GHz - 2.2GHz speed

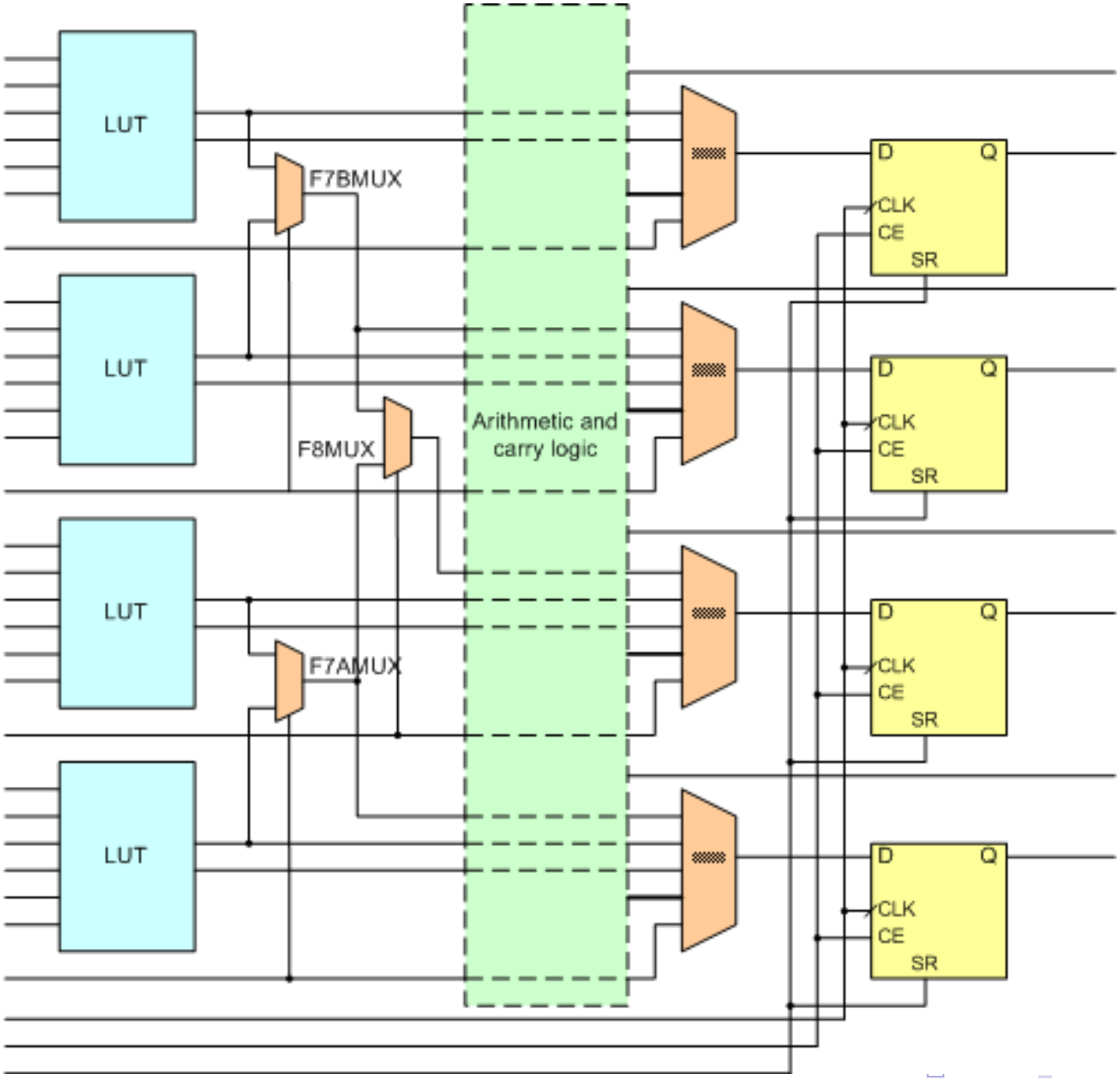
General Xilinx Virtex 5 Slice architecture

- Each Virtex 5 slice has 4 Look-Up Tables (LUTs), eight registers and several multiplexers

General Xilinx Virtex 5 Slice architecture

- Each Virtex 5 slice has 4 Look-Up Tables (LUTs), eight registers and several multiplexers
- A LUT can be configured to perform any Boolean operation of 6 inputs/1 output or 5 inputs/ 2 outputs or as a memory elements of 64 inputs of one-bit size

General Xilinx Virtex 5 Slice architecture



Block RAMs

Virtex devices include built-in 32K-bit RAM memory, called BRAM, which are intended for storing big amounts of data. Some of its features are,

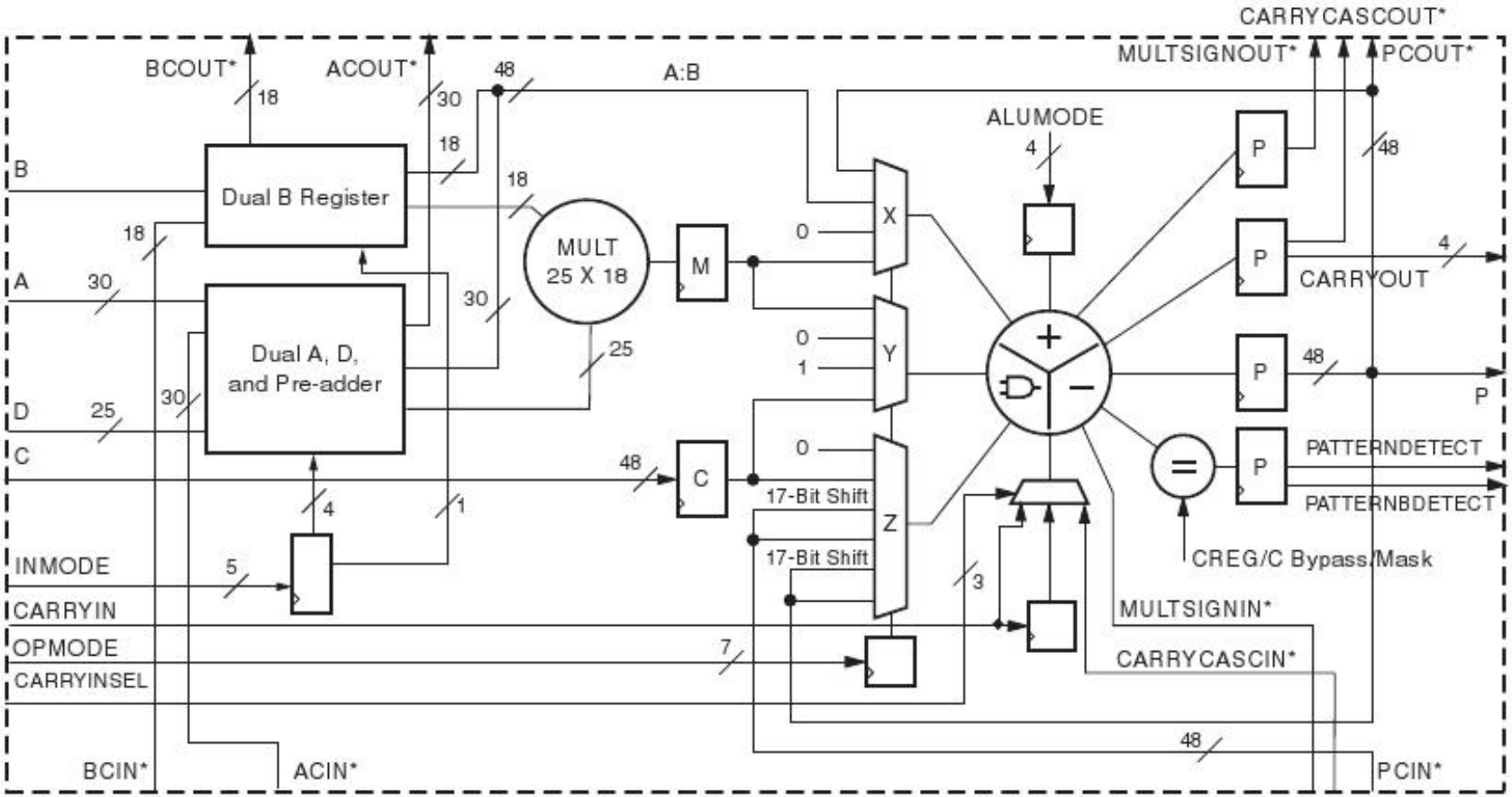
- Polymorphic [bus size programmable]
- Dual port [they can perform two data reads and one write in the same clock cycle]
- can be configured for a size of up to 4K bytes

DSP Slices

DSP slices are embedded devices equipped with the following components

- 25×18 two's-complement multiplier:
- 48-bit accumulator
- pre-adders
- Single-instruction-multiple-data (SIMD) arithmetic unit
- Can generate any one of ten different logic functions of the two operands
- execute all the operations at a extremely high frequency

DSP Slices


CARRYCASCOUT*
MULTSIGNOUT*
PCOUT*
BCOUT*
ACOUT*
A:B
ALUMODE
B
Dual B Register
A
Dual A, D,
and Pre-adder
D
C
MULT
25 X 18
M
C
X
Y
Z
0
1
17-Bit Shift
P
CARRYOUT
PATTERNDETECT
PATTERNBDETECT
CREG/C Bypass/Mask
INMODE
CARRYIN
OPMODE
CARRYINSEL
MULTSIGNIN*
CARRYCASCIN*
BCIN*
ACIN*
PCIN*

FPGAs Vs. ASIC

- Advantages
  - They have been utilized for fast prototyping of hardware designs
  - They are reconfigurable devices
  - They allow for a shorter design cycle
  - They permit hardware-software co-design

FPGAs Vs. ASIC

- Advantages
  - They have been utilized for fast prototyping of hardware designs
  - They are reconfigurable devices
  - They allow for a shorter design cycle
  - They permit hardware-software co-design
- Disadvantages
  - They tend to consume much more power and energy than ASIC designs
  - Their reconfigurability implies redundancy
  - Their speed is minor than the one achievable with ASICs

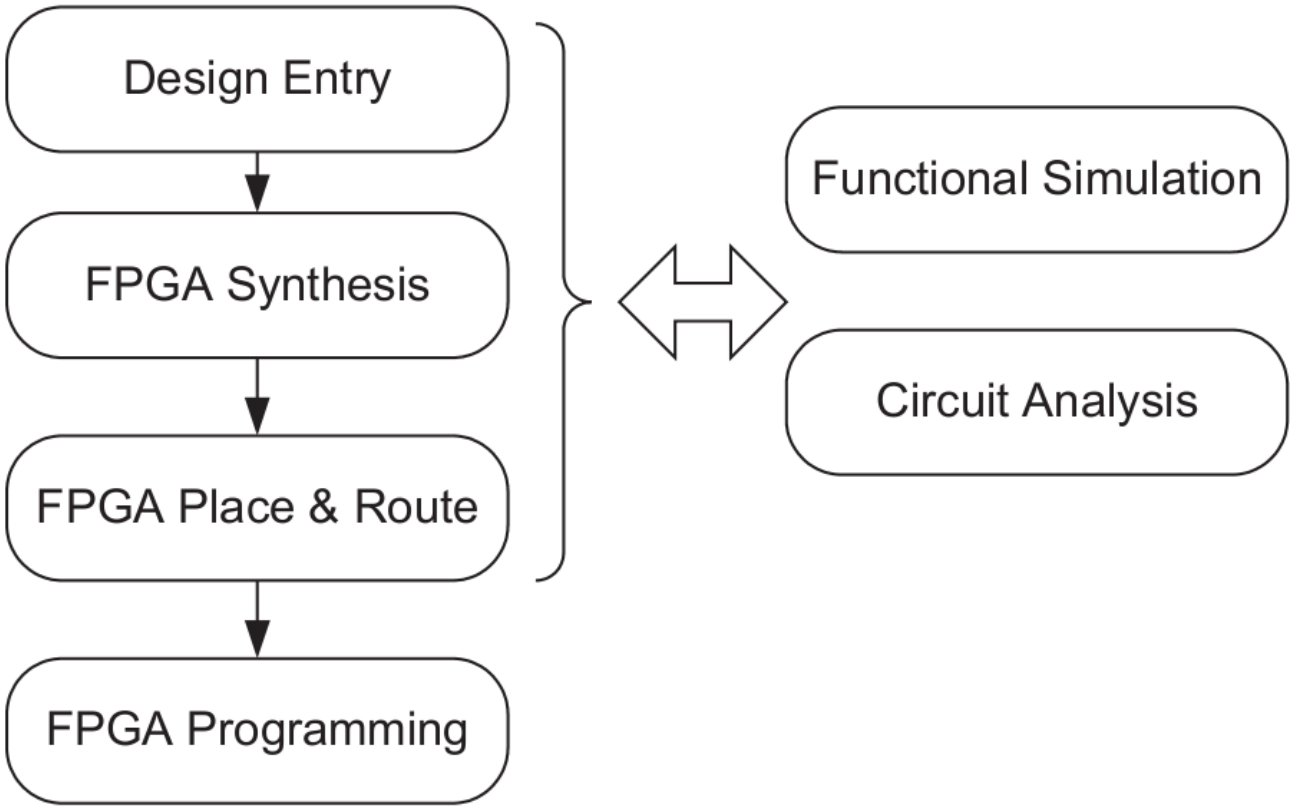
FPGAs Vs. General purpose processors

- Advantages
  - More often than not, they are faster than software applications
  - some operations are almost free of cost [such as shifts, rotations, etc.]
  - They allow for a versatile data-path
  - They inherently enjoy fine-grain parallelism

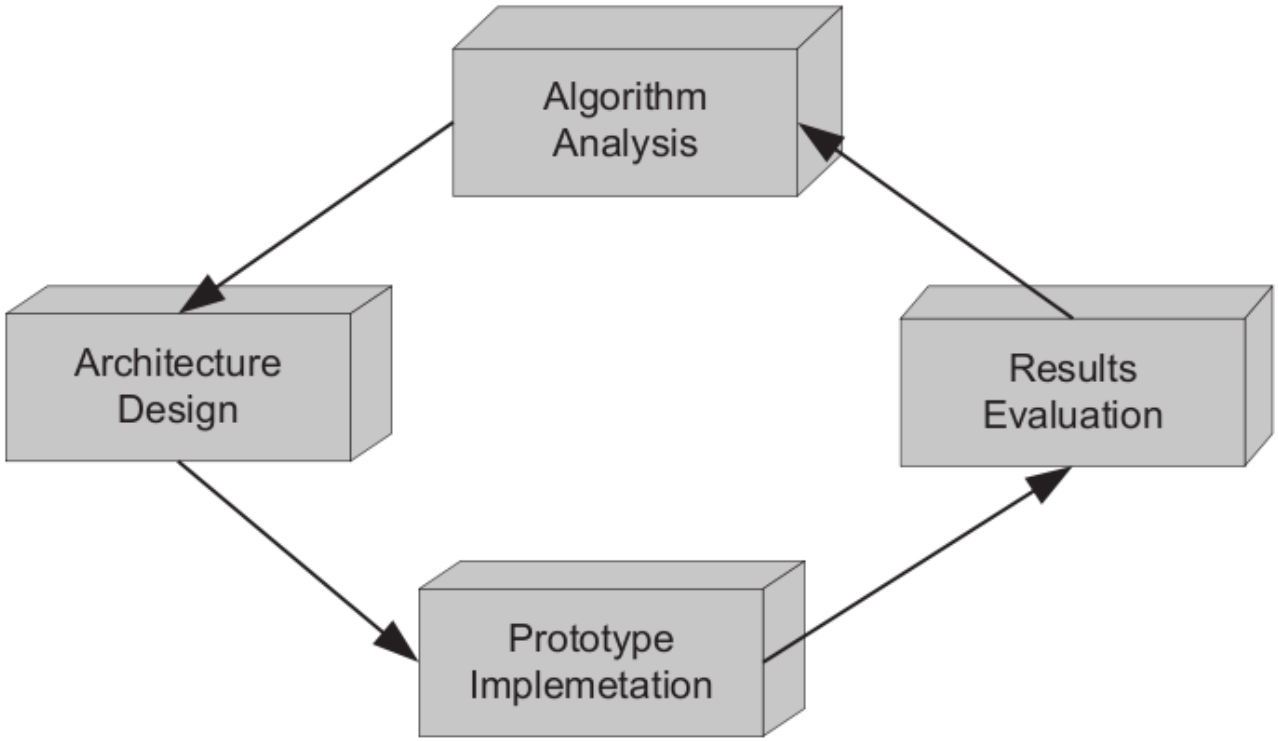
FPGAs Vs. General purpose processors

- Advantages
  - More often than not, they are faster than software applications
  - some operations are almost free of cost [such as shifts, rotations, etc.]
  - They allow for a versatile data-path
  - They inherently enjoy fine-grain parallelism
- Disadvantages
  - It is a bit more difficult to code and test designs
  - Their maximum clock frequency is ten times slower
  - prime field arithmetic tends to be more difficult to handle

Design Methodology for FPGA designs

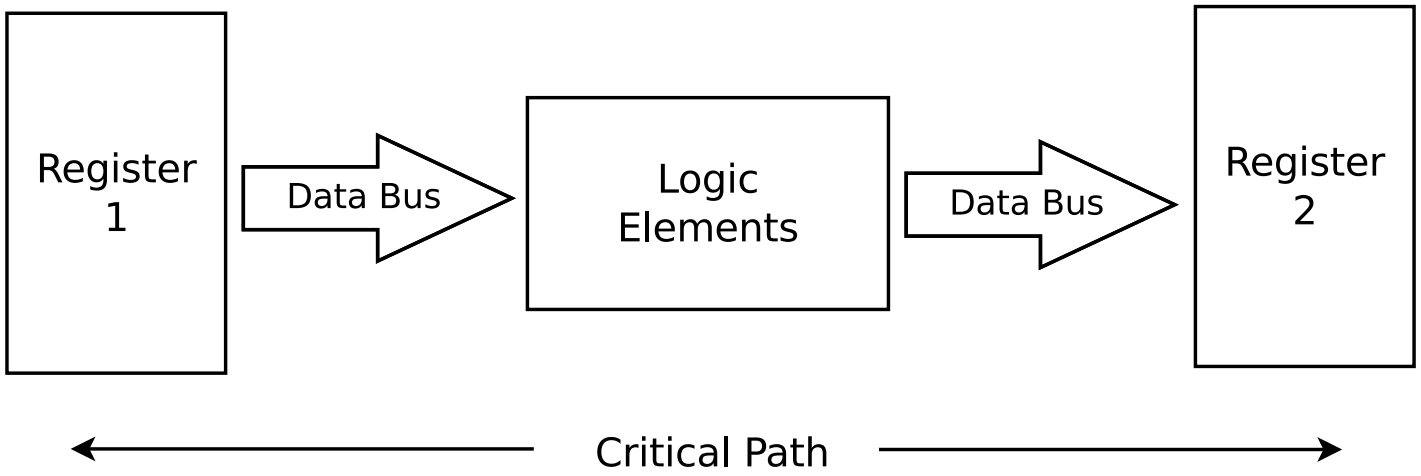


Design Methodology for FPGA designs



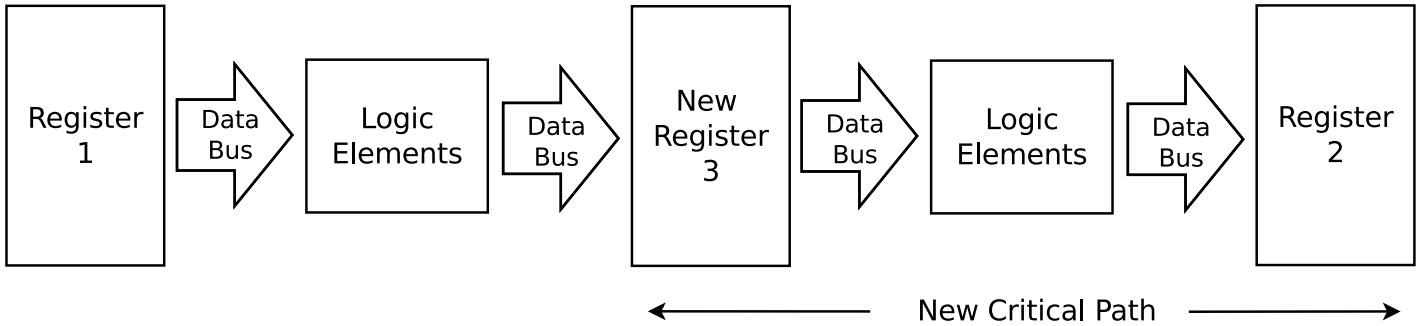
FPGA Design issues: circuit's critical path

Circuit's critical path: The maximum allowed clock frequency is determined by the longest/slowest combinatorial path present in the circuit



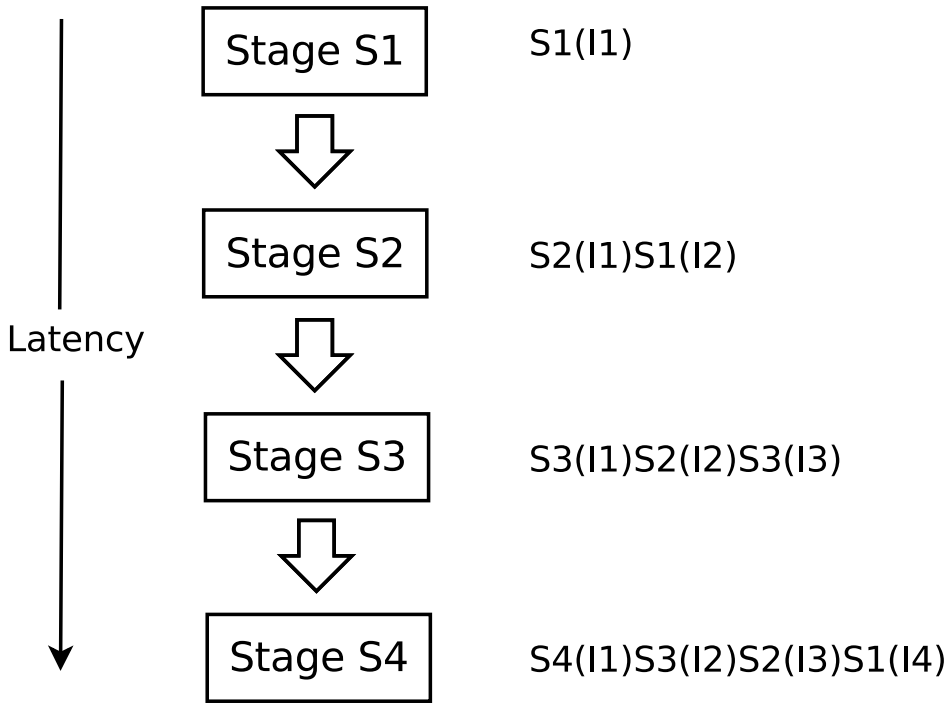
FPGA Design issues: circuit's critical path

Circuit's critical path: The maximum allowed clock frequency is determined by the longest/slowest combinatorial path present in the circuit



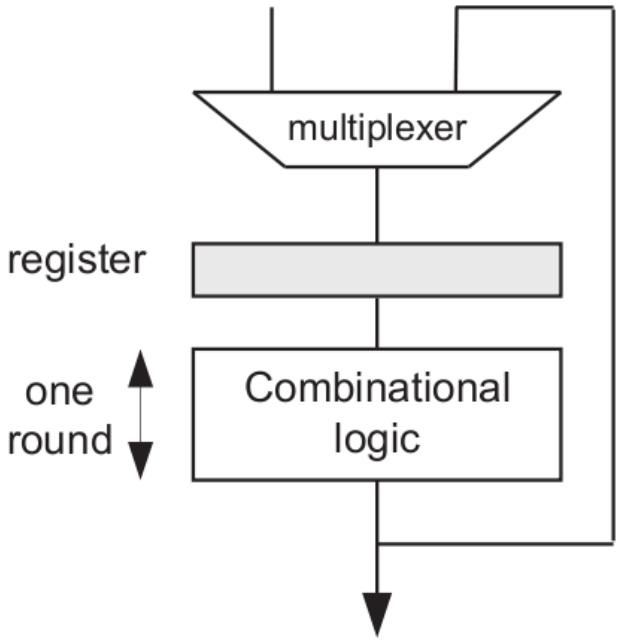
FPGA Design issues: Latency

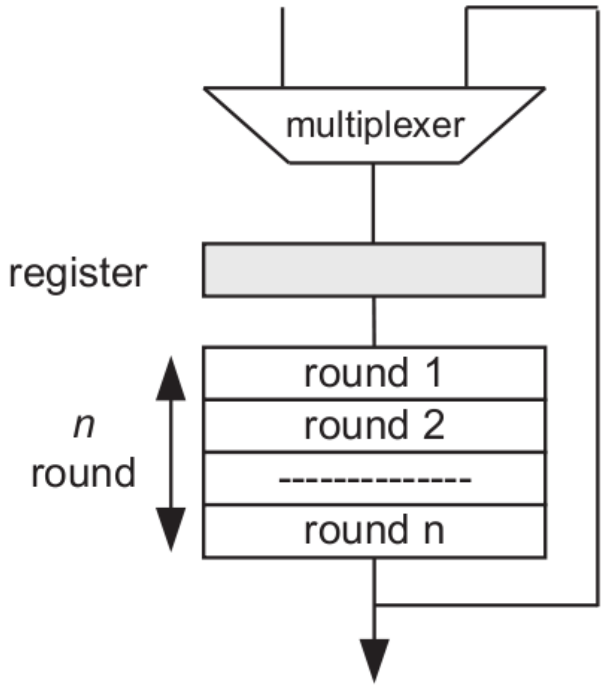
Circuit's latency: the amount of time [often given in number of clock cycles] required for producing the first output



Parallel techniques in hardware: pipelining

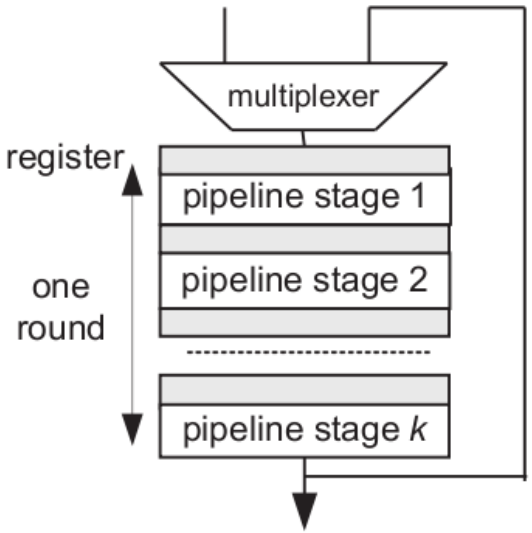
pipelining is a natural technique for producing more parallelism, however, the designer must balance the pipe stages, carefully trying to avoid dependencies and pipeline stalls

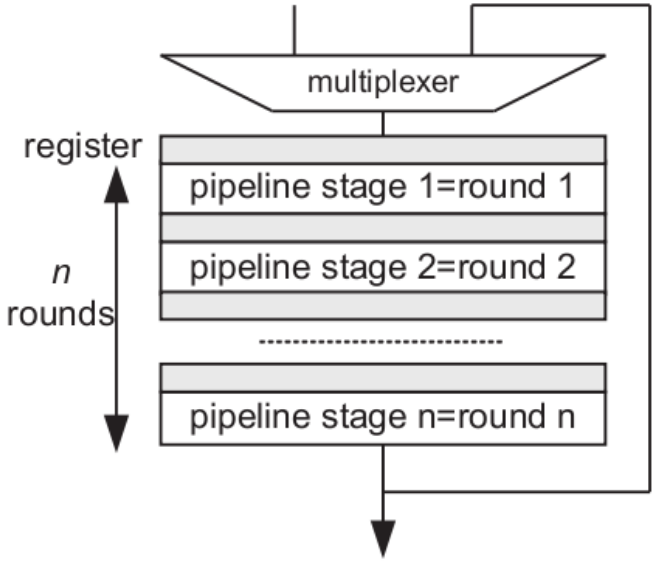




Parallel techniques in hardware: pipelining

pipelining is a natural technique for producing more parallelism, however, the designer must balance the pipe stages, carefully trying to avoid dependencies and pipeline stalls





Measures of performance in reconfigurable Hardware devices

Measures of performance in reconfigurable Hardware devices

- Computational time defined as:

$$\frac{\#\text{ of clock cycles}}{\text{clock cycle frequency}}$$

Measures of performance in reconfigurable Hardware devices

- Computational time defined as:

$$\frac{\#\text{ of clock cycles}}{\text{clock cycle frequency}}$$

- Throughput defined as:

$$\frac{\#\text{ of bits processed } \cdot \text{clock cycle frequency}}{\#\text{ of clock cycles}}$$

Measures of performance in reconfigurable Hardware devices

- Computational time defined as:

$$\frac{\#\text{ of clock cycles}}{\text{clock cycle frequency}}$$

- Throughput defined as:

$$\frac{\#\text{ of bits processed } \cdot \text{clock cycle frequency}}{\#\text{ of clock cycles}}$$

- Latency:# of clock cycles required for producing the first computation

Measures of performance in reconfigurable Hardware devices

- Computational time defined as:

$$\frac{\#\text{ of clock cycles}}{\text{clock cycle frequency}}$$

- Throughput defined as:

$$\frac{\#\text{ of bits processed } \cdot \text{clock cycle frequency}}{\#\text{ of clock cycles}}$$

- Latency:# of clock cycles required for producing the first computation
- Amount of hardware resources utilized by the design. Including slices, dedicated memories, DSP slices, etc.

Measures of performance in reconfigurable Hardware devices

- Computational time defined as:

$$\frac{\#\text{ of clock cycles}}{\text{clock cycle frequency}}$$

- Throughput defined as:

$$\frac{\#\text{ of bits processed } \cdot \text{clock cycle frequency}}{\#\text{ of clock cycles}}$$

- Latency:# of clock cycles required for producing the first computation
- Amount of hardware resources utilized by the design. Including slices, dedicated memories, DSP slices, etc.
- Time-Area product

Measures of performance in reconfigurable Hardware devices

- Computational time defined as:

$$\frac{\#\text{ of clock cycles}}{\text{clock cycle frequency}}$$

- Throughput defined as:

$$\frac{\#\text{ of bits processed } \cdot \text{clock cycle frequency}}{\#\text{ of clock cycles}}$$

- Latency:# of clock cycles required for producing the first computation
- Amount of hardware resources utilized by the design. Including slices, dedicated memories, DSP slices, etc.
- Time-Area product
- Power consumption, energy consumption, ...

Measures of performance in reconfigurable Hardware devices

- Computational time defined as:

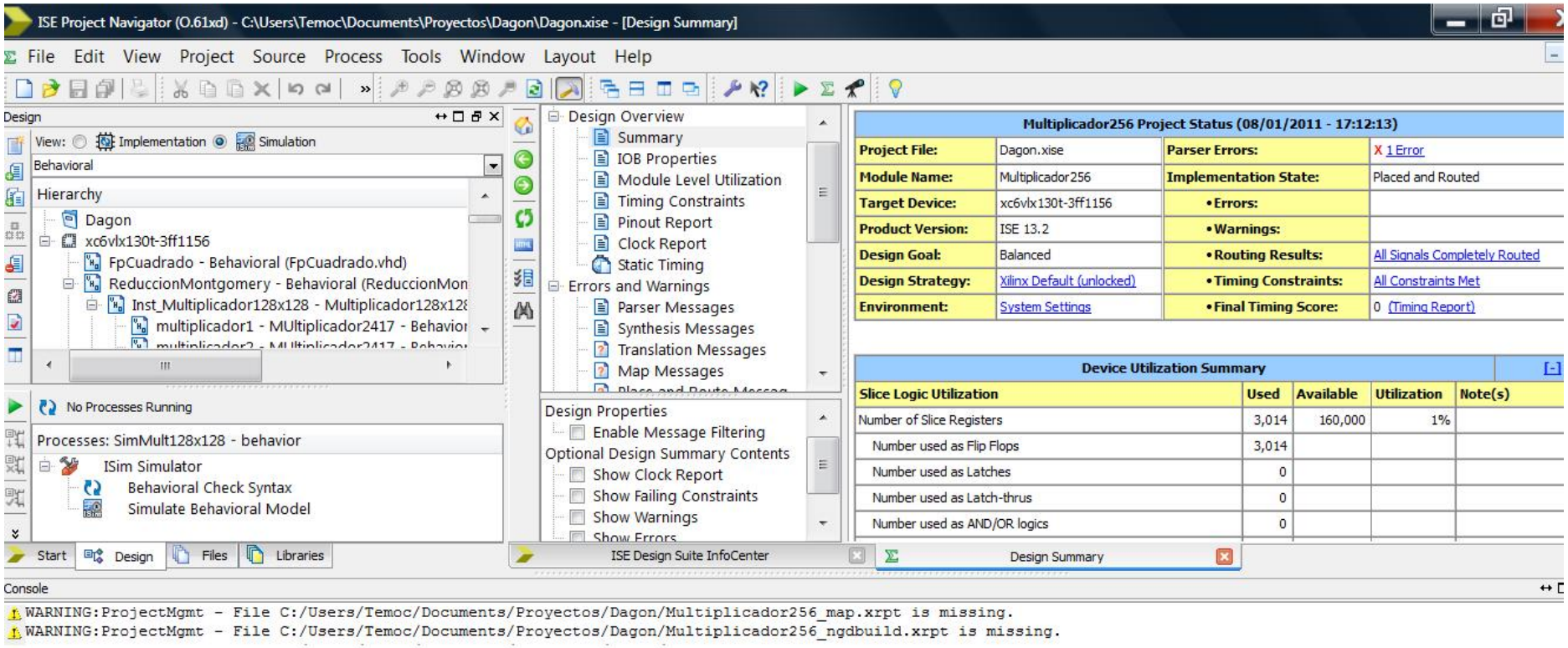
$$\frac{\#\text{ of clock cycles}}{\text{clock cycle frequency}}$$

- Throughput defined as:

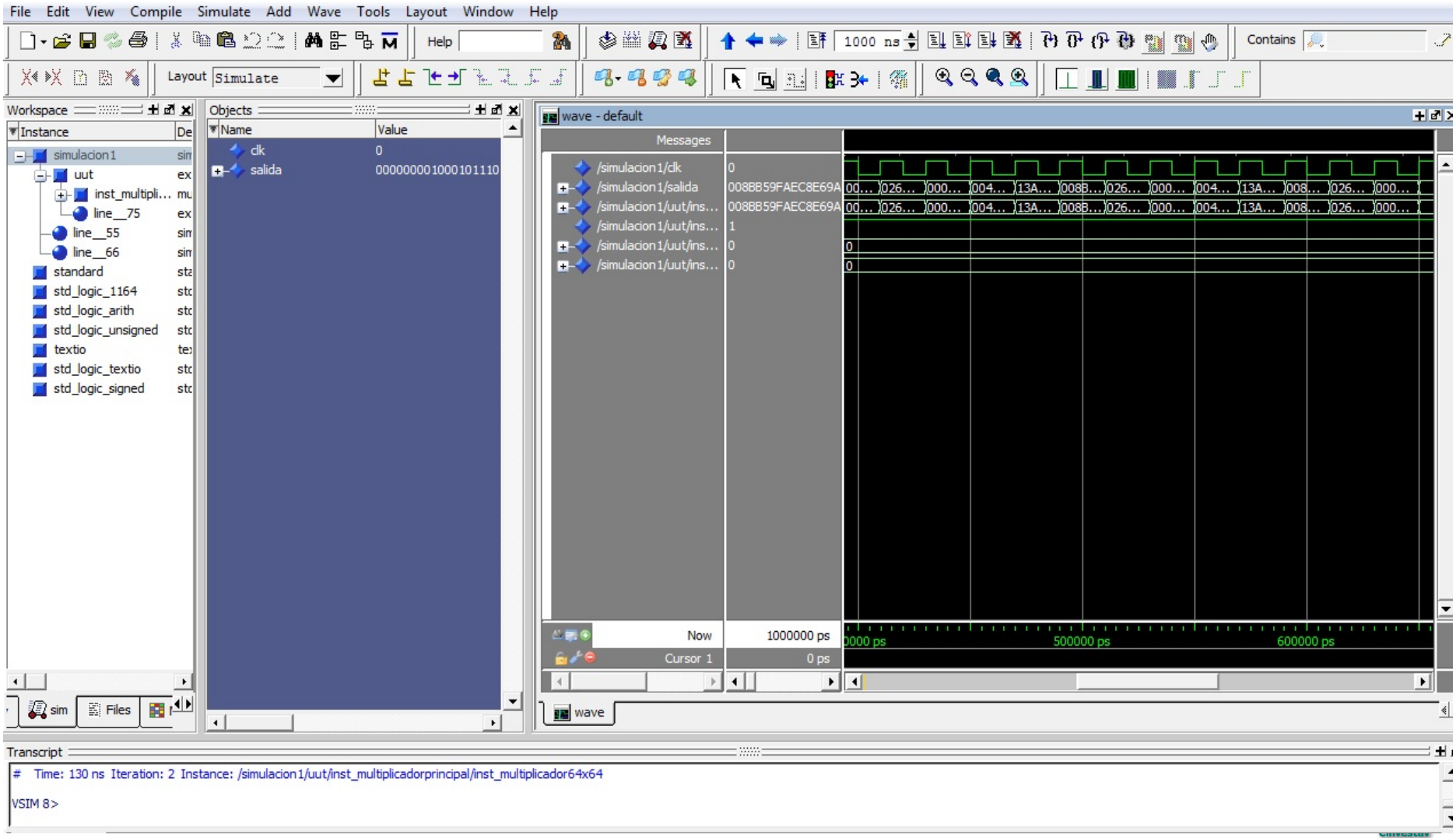
$$\frac{\#\text{ of bits processed } \cdot \text{clock cycle frequency}}{\#\text{ of clock cycles}}$$

- Latency:# of clock cycles required for producing the first computation
- Amount of hardware resources utilized by the design. Including slices, dedicated memories, DSP slices, etc.
- Time-Area product
- Power consumption, energy consumption, ...
- In the case of cryptographic designs: Side-channel resistance

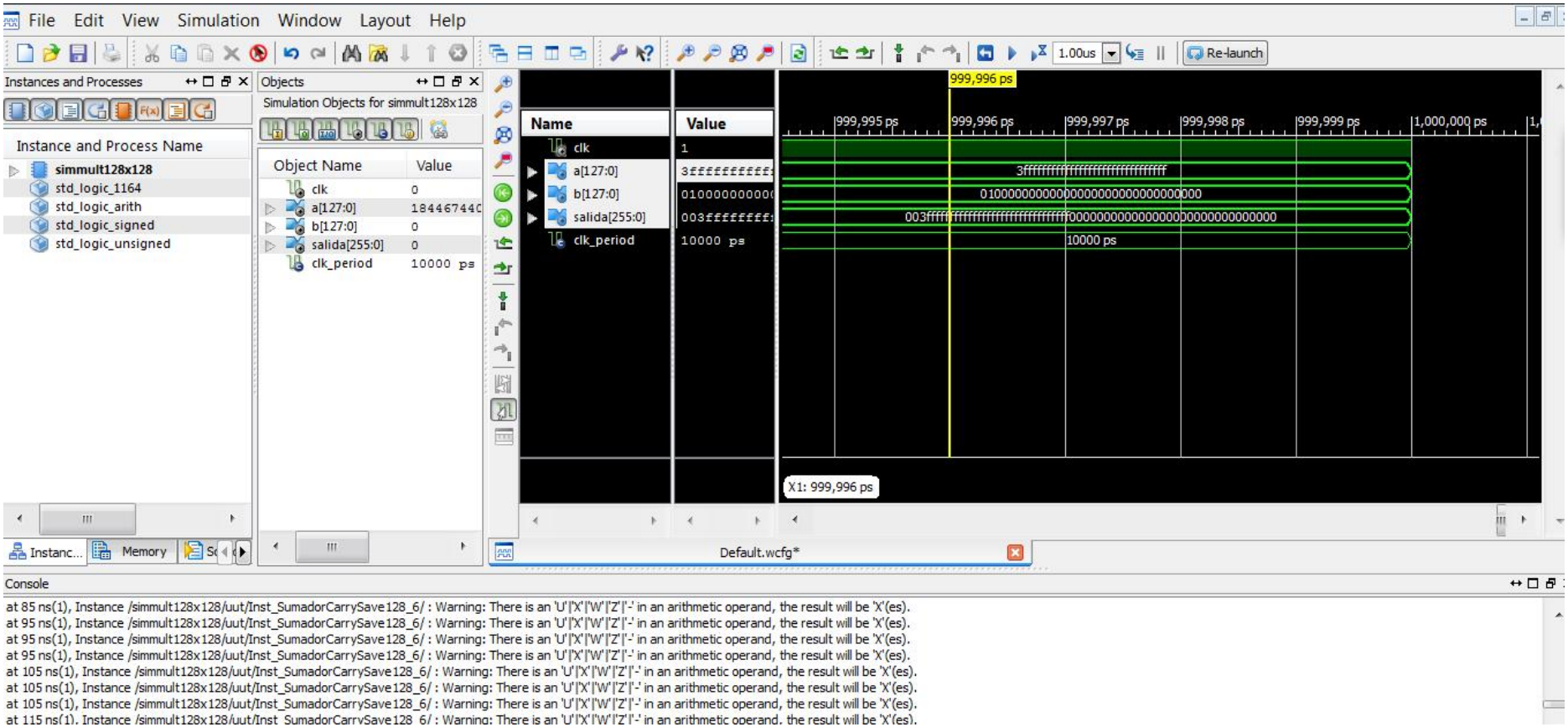
Design tools



Design tools



Design tools



Finite fields

Every prime number p defines a finite field of order p, denoted as, $\mathbb{F}_p$.

The smallest finite field is $\langle \mathbb{F}_2, \oplus, \odot \rangle$, that contains only two elements $\{0, 1\}$ and its binary operations act as the Boolean operators `XOR` and `AND`, respectively.

Field Extensions

Given a positive integer $m > 1$,the field $\mathbb{F}_{p^m}$ is a field extension of $\mathbb{F}_p$.

It can be shown that $\mathbb{F}_{p^m}$ is isomorphic to $\mathbb{F}_p[x]/(f(x))$, where $f(x)$ is a monic polynomial of degree $m > 1$, irreducible over $\mathbb{F}_p$.

We denote by $\mathbb{F}_p[x]/(f(x))$ the set of equivalence classes of the polynomials $\mathbb{F}_p[x] \pmod{f(x)}$.

Agenda

Cinvestav
Centro de Investigación y de Estudios Avanzados del Instituto Politécnico Nacional
50 AÑOS

Arithmetic over $\mathbb{F}_{3^m}$

- $f \in \mathbb{F}_2[x]$: degree-m irreducible polynomial over $\mathbb{F}_2$

$$f = x^m + f_{m-1}x^{m-1} + \cdots + f_1 x + f_0$$

Arithmetic over $\mathbb{F}_{3^m}$

- $f \in \mathbb{F}_2[x]$: degree-m irreducible polynomial over $\mathbb{F}_2$

$$f = x^m + f_{m-1}x^{m-1} + \cdots + f_1 x + f_0$$

- $\mathbb{F}_{2^m} \cong \mathbb{F}_2[x]/(f)$
- $a \in \mathbb{F}_{2^m}$:

$$a = a_{m-1}x^{m-1} + \cdots + a_1 x + a_0$$

- Each element of $\mathbb{F}_2$ stored using one bit, ergo,a field element $\mathbb{F}_{2^m}$ can be represented as a vector of m bits.
- Usually the irreducible polynomial f is selected as a trinomial or a pentanomial

Addition over $\mathbb{F}_{2^m}$

- $r = a + b = (a_{m-1} + b_{m-1})x^{m-1} + \cdots + (a_1 + b_1)x + (a_0 + b_0)$

Addition over $\mathbb{F}_{2^m}$

- $r = a + b = (a_{m-1} + b_{m-1})x^{m-1} + \cdots + (a_1 + b_1)x + (a_0 + b_0)$
  - coefficient-wise additions over $\mathbb{F}_2$: $r_i = (a_i + b_i) \bmod 2$

Addition over $\mathbb{F}_{2^m}$

- $r = a + b = (a_{m-1} + b_{m-1})x^{m-1} + \cdots + (a_1 + b_1)x + (a_0 + b_0)$
  - coefficient-wise additions over $\mathbb{F}_2$: $r_i = (a_i + b_i) \bmod 2$
  - addition over $\mathbb{F}_2$: XOR gates
  - This operation directly benefits from the parallel processing of the `XOR` operation

Field Squaring

Due to the action of the Frobenius map, polynomial squaring of an element $a \in \mathbb{F}_{2^m}$ is a linear operation over binary fields,

$$\begin{aligned} a(x)^2 &= \left[\sum_{i=0}^{m-1} a_i x^i\right]^2 \\ &= \sum_{i=0}^{m-1} a_i x^{2i} \end{aligned}$$

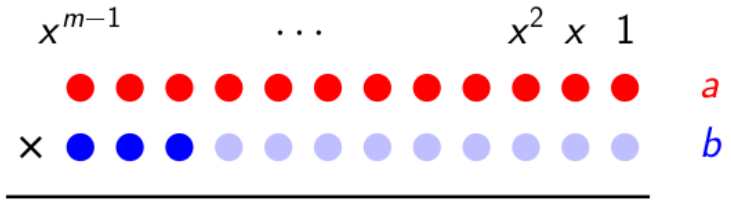
This can be implemented by interleaving zeroes among the polynomial coefficients,

$$\begin{aligned} \vec{a} &\rightarrow (\vec{a})^2 \\ (a_{m-1}, a_{m-2}, \ldots, a_1, a_0) &\rightarrow (a_{m-1}, 0, \ldots, a_2, 0, a_1, 0, a_0) \end{aligned}$$

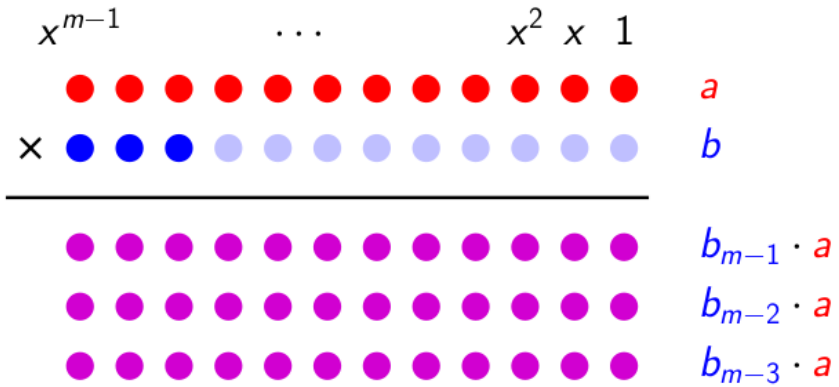
Multiplication over $\mathbb{F}_{2^m}$

- Parallel-serial multiplication
  - multiplicand loaded in a parallel register
  - multiplier loaded in a shift register
- Most significant coefficients first (Horner scheme)
- D coefficients processed at each clock cycle: $\left\lceil \frac{m}{D} \right\rceil$ cycles per multiplication

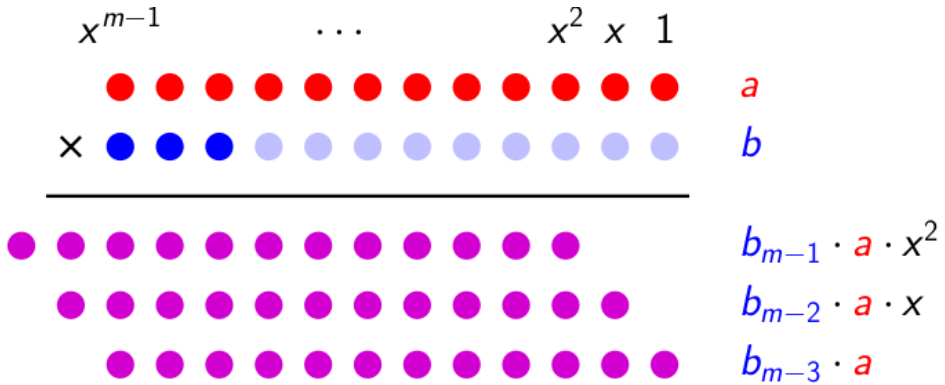
Field multiplication: interleaving products and reduction



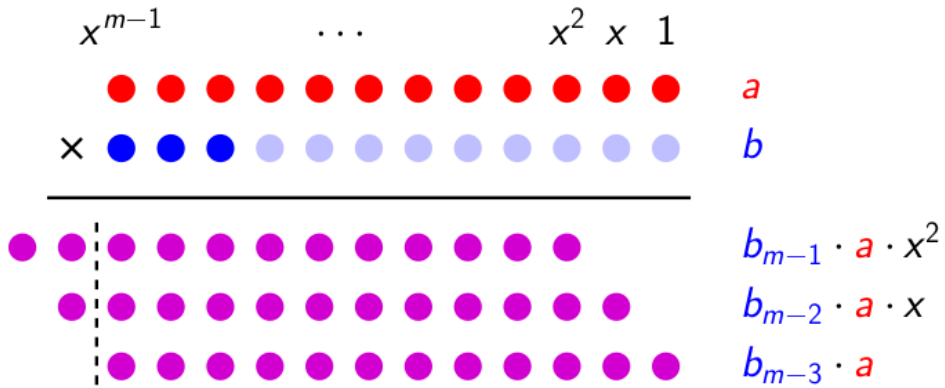
Field multiplication: interleaving products and reduction



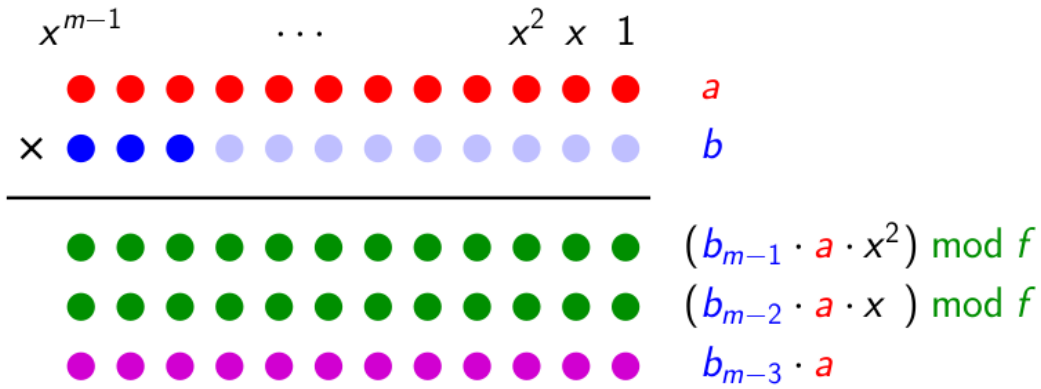
Field multiplication: interleaving products and reduction



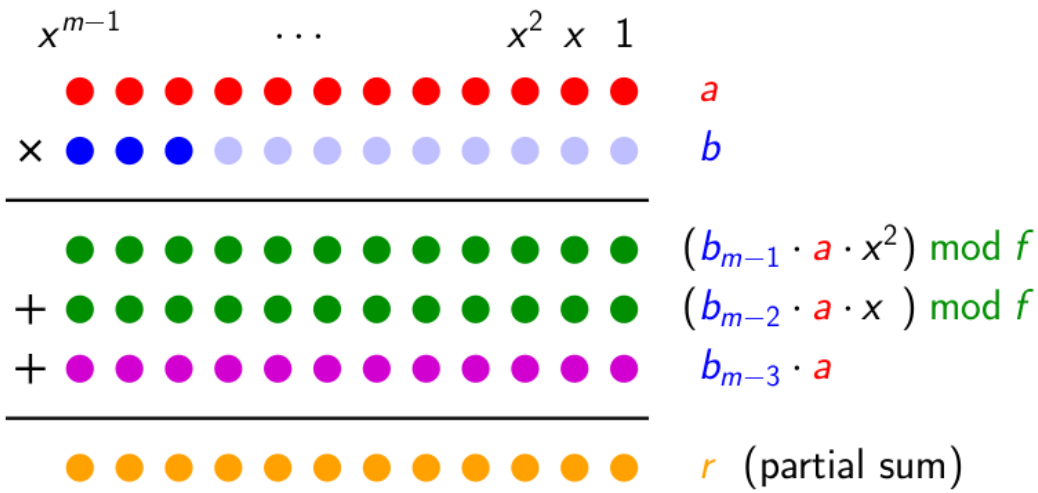
Field multiplication: interleaving products and reduction



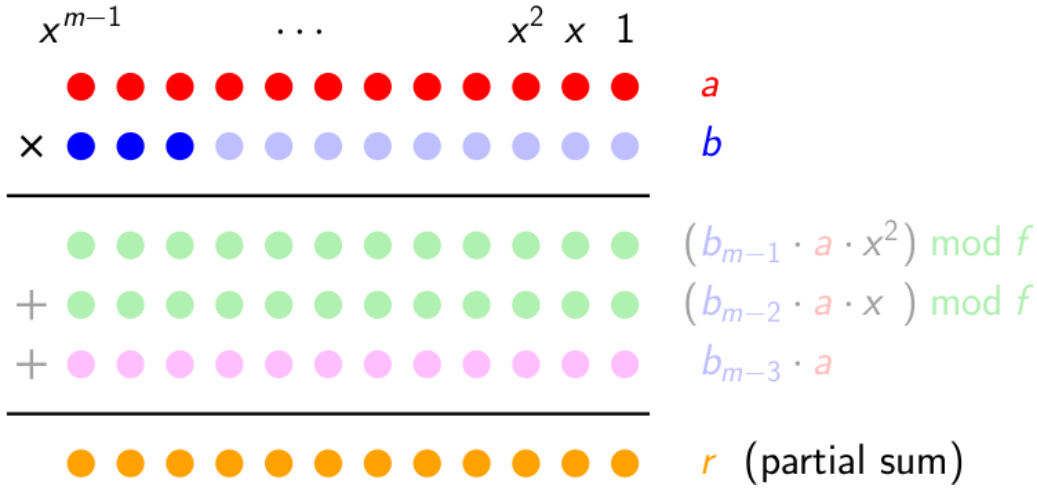
Field multiplication: interleaving products and reduction



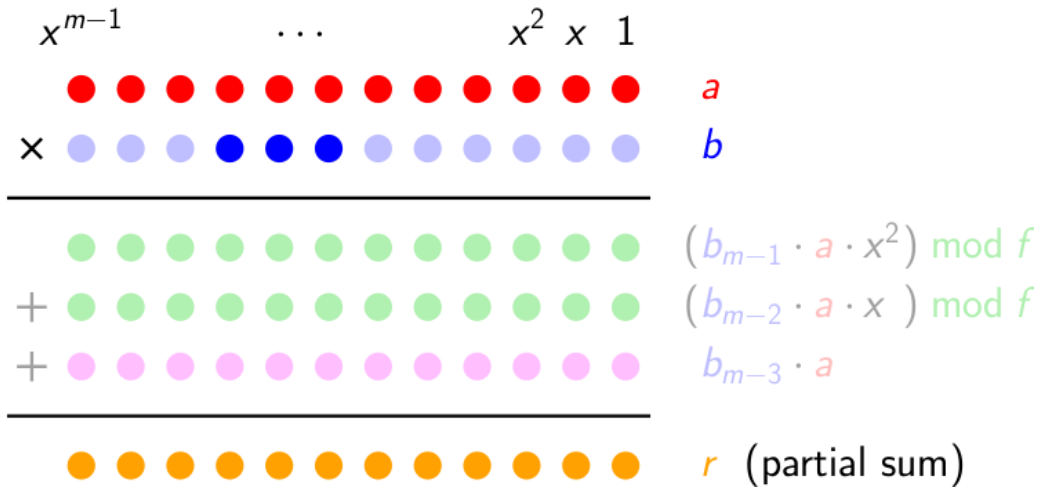
Field multiplication: interleaving products and reduction



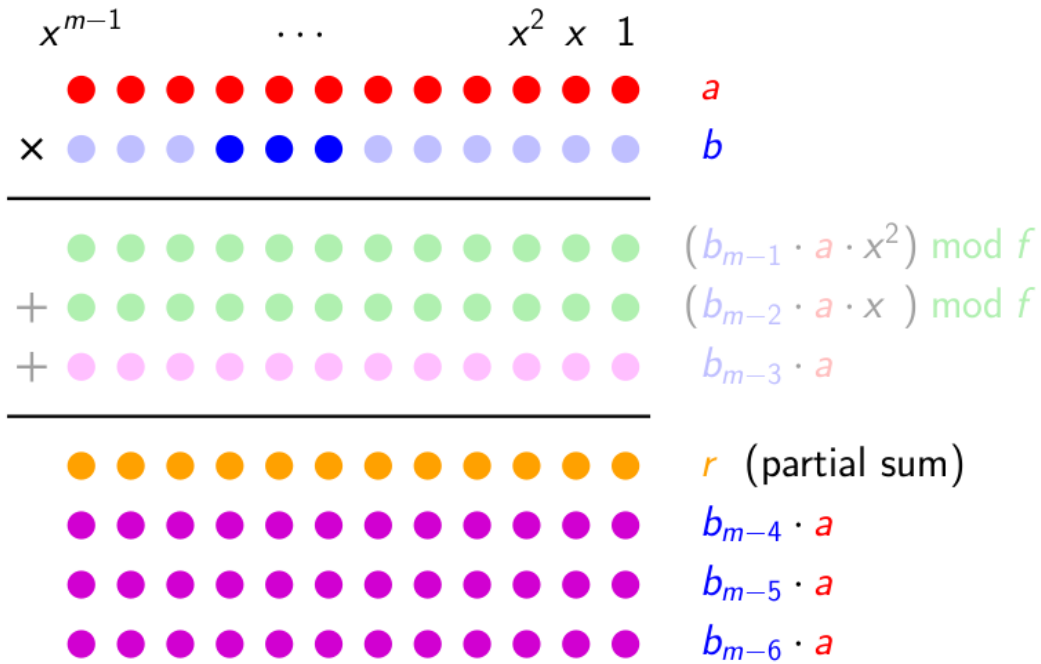
Field multiplication: interleaving products and reduction



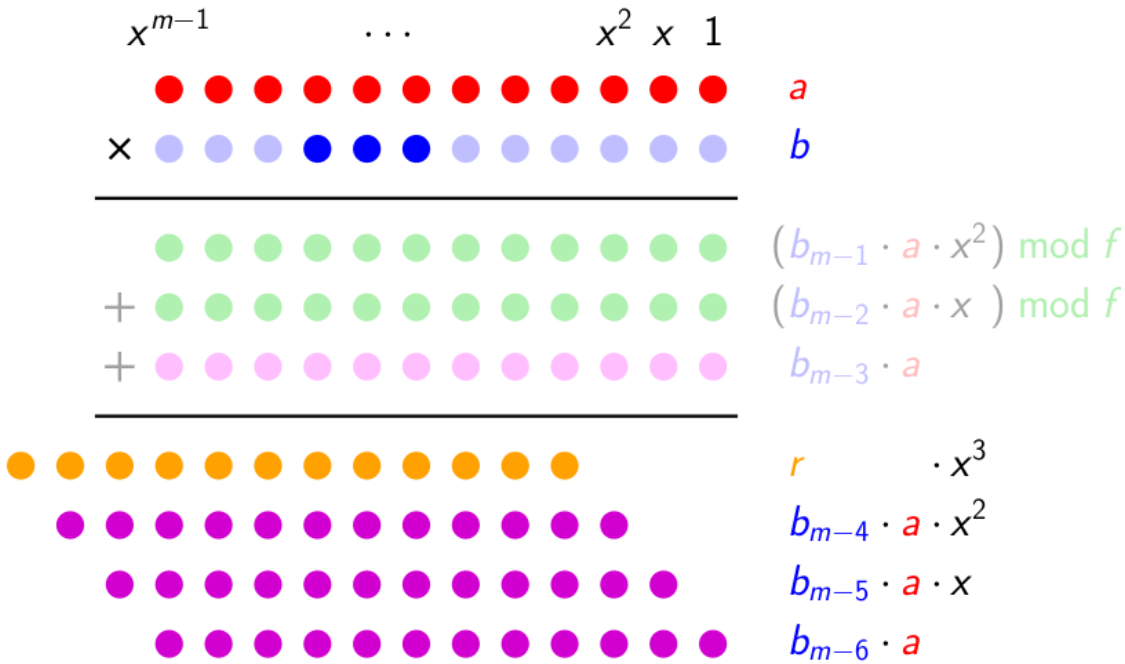
Field multiplication: interleaving products and reduction



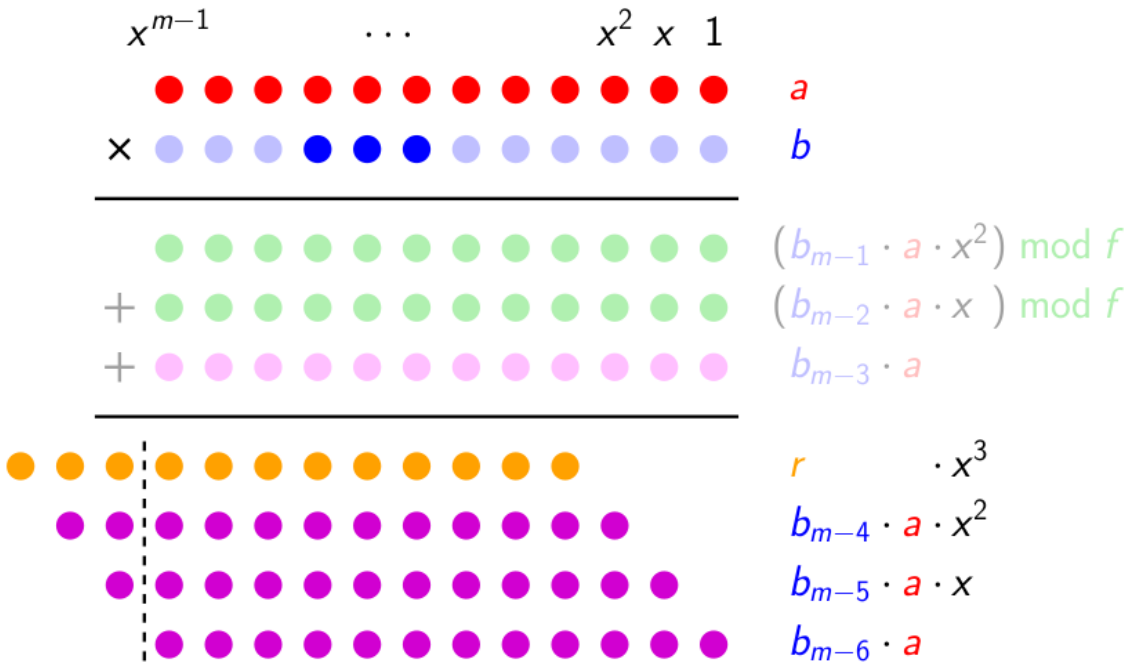
Field multiplication: interleaving products and reduction



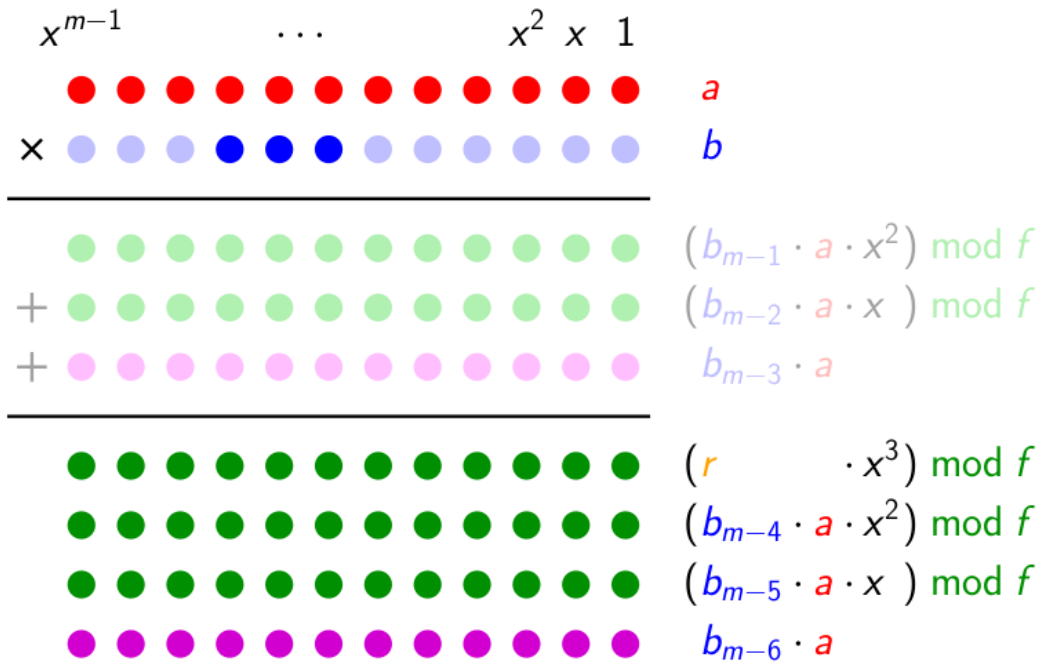
Field multiplication: interleaving products and reduction



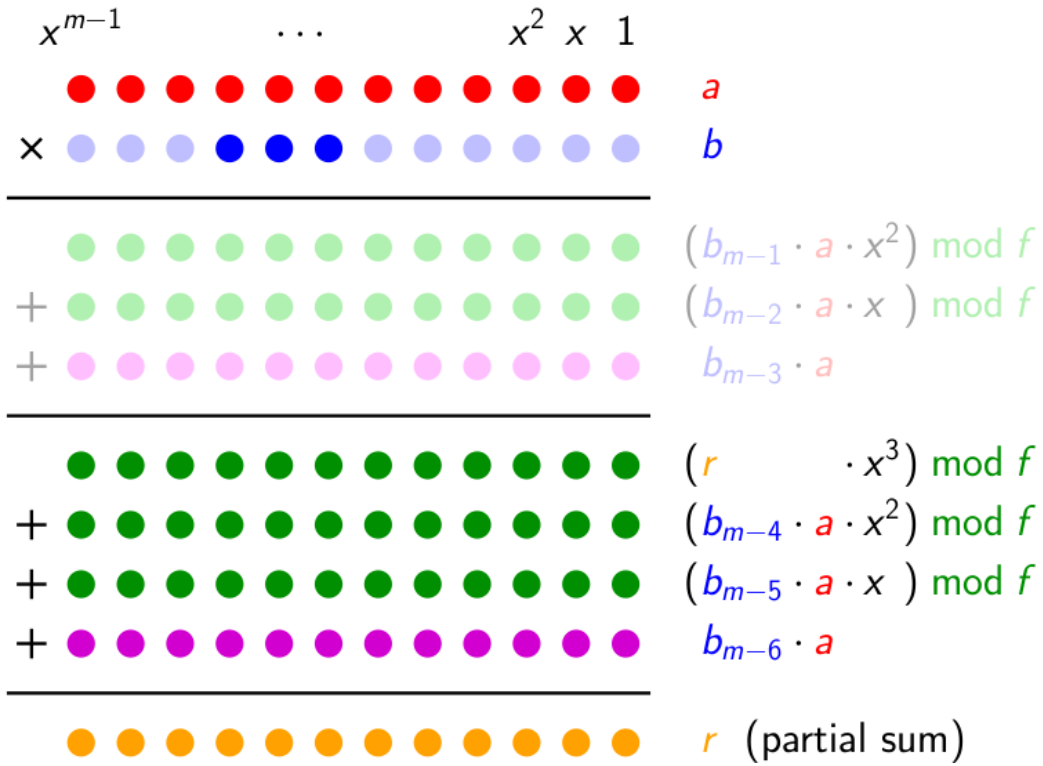
Field multiplication: interleaving products and reduction



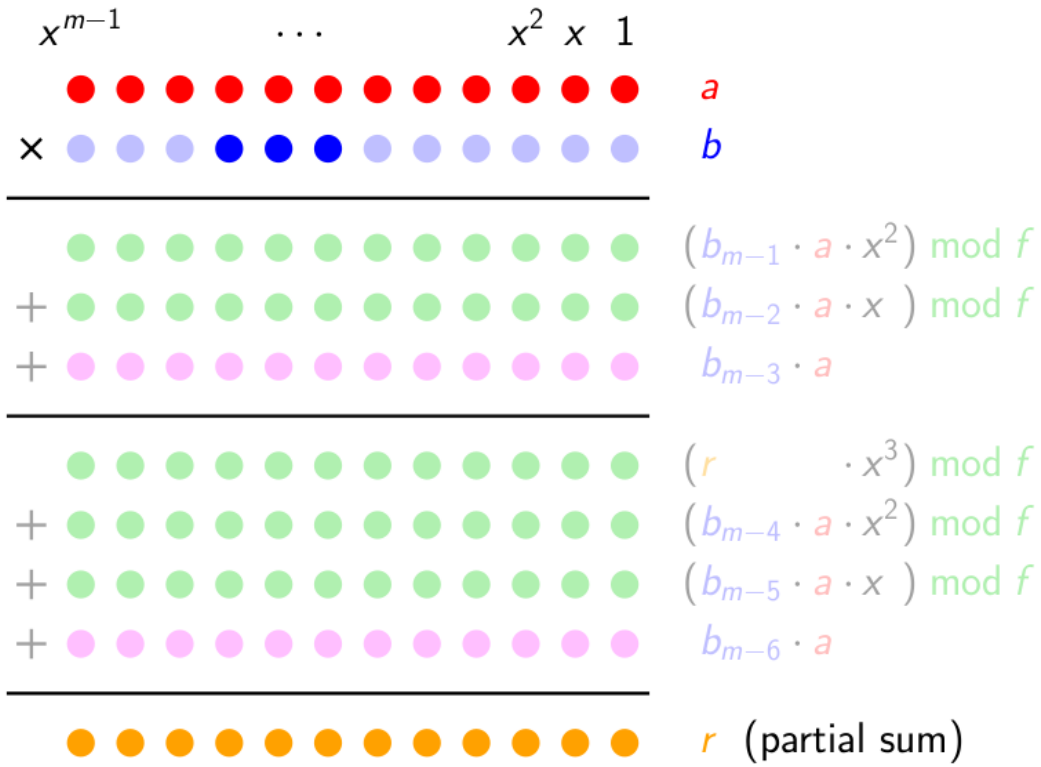
Field multiplication: interleaving products and reduction



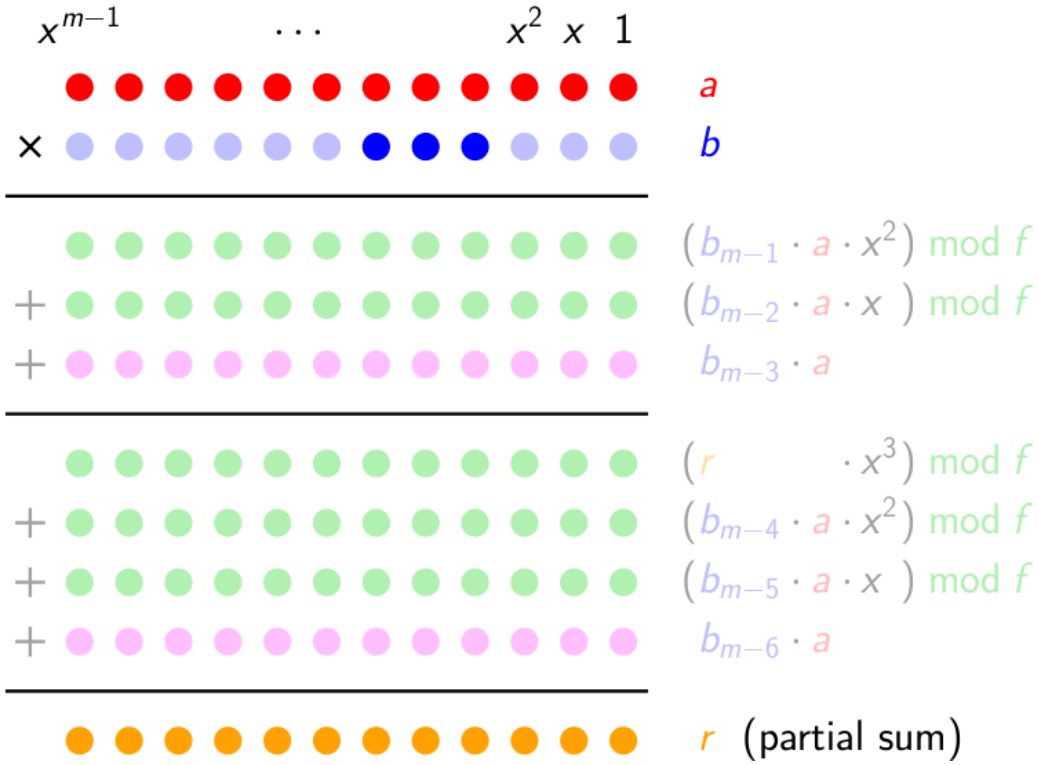
Field multiplication: interleaving products and reduction



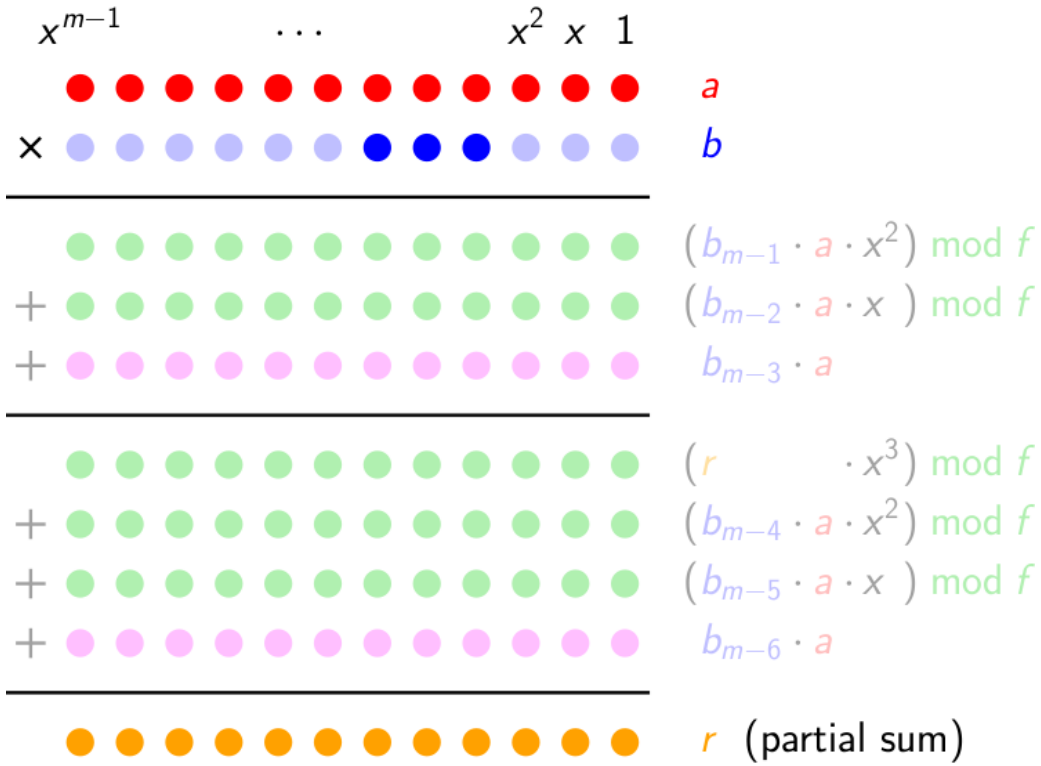
Field multiplication: interleaving products and reduction



Field multiplication: interleaving products and reduction



Field multiplication: interleaving products and reduction



Bit parallel multiplication

- For this case, field multiplication is usually performed in two steps: polynomial multiplication followed by polynomial reduction

Bit parallel multiplication

- For this case, field multiplication is usually performed in two steps: polynomial multiplication followed by polynomial reduction
- The first phase consists on multiplying two polynomials of degree $m-1$ to obtain a polynomial of degree $2m-2$, where the arithmetic operations are performed over $\mathbb{F}_2$

Bit parallel multiplication

- For this case, field multiplication is usually performed in two steps: polynomial multiplication followed by polynomial reduction
- The first phase consists on multiplying two polynomials of degree $m-1$ to obtain a polynomial of degree $2m-2$, where the arithmetic operations are performed over $\mathbb{F}_2$
- The second phase performs modular reduction using $f(x)$, the irreducible polynomial that generated the field.

Polynomial multiplication

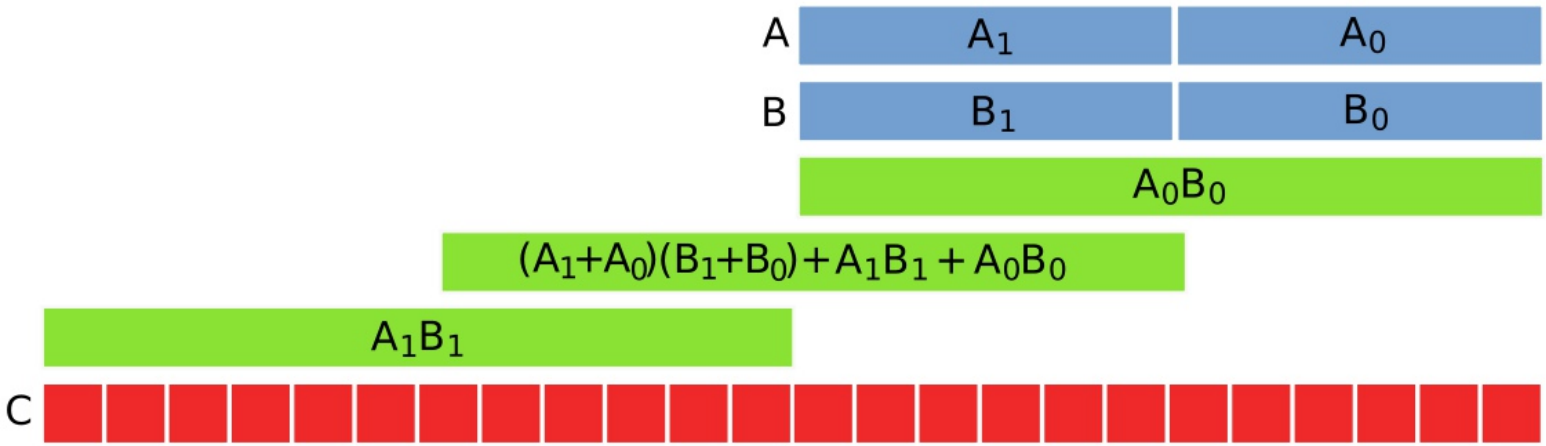
Given the polynomials A y B of degree $m-1$, the product $C = A \cdot B$ of degree $2m-2$ can be computed as,

$$\begin{aligned} C &= A \cdot B \\ &= (a_0 + a_1 x^{\frac{m-1}{2}})(b_0 + b_1 x^{\frac{m-1}{2}}) \\ &= a_0 b_0 + [(a_0 + a_1)(b_0 + b_1) + a_0 b_0 + a_1 b_1] x^{\frac{m-1}{2}} \\ &\quad + a_1 b_1 x^{m-1} \end{aligned}$$

Polynomial multiplication

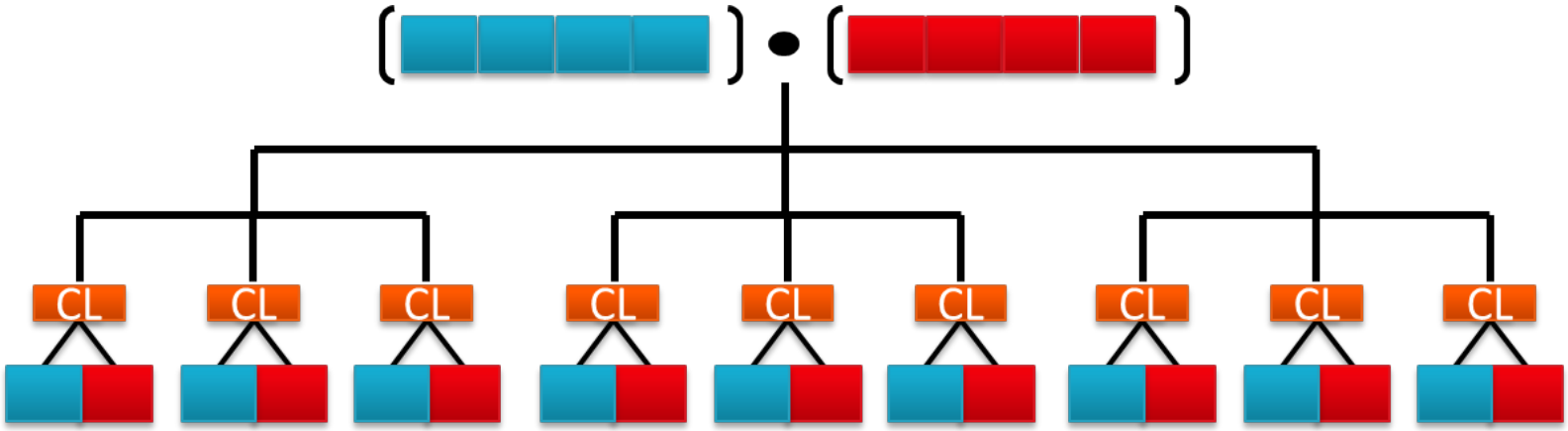
Given the polynomials A y B of degree $m-1$, the product $C = A \cdot B$ of degree $2m-2$ can be computed as,

$$\begin{aligned} C &= A \cdot B \\ &= (a_0 + a_1 x^{\frac{m-1}{2}})(b_0 + b_1 x^{\frac{m-1}{2}}) \\ &= a_0 b_0 + [(a_0 + a_1)(b_0 + b_1) + a_0 b_0 + a_1 b_1] x^{\frac{m-1}{2}} \\ &\quad + a_1 b_1 x^{m-1} \end{aligned}$$



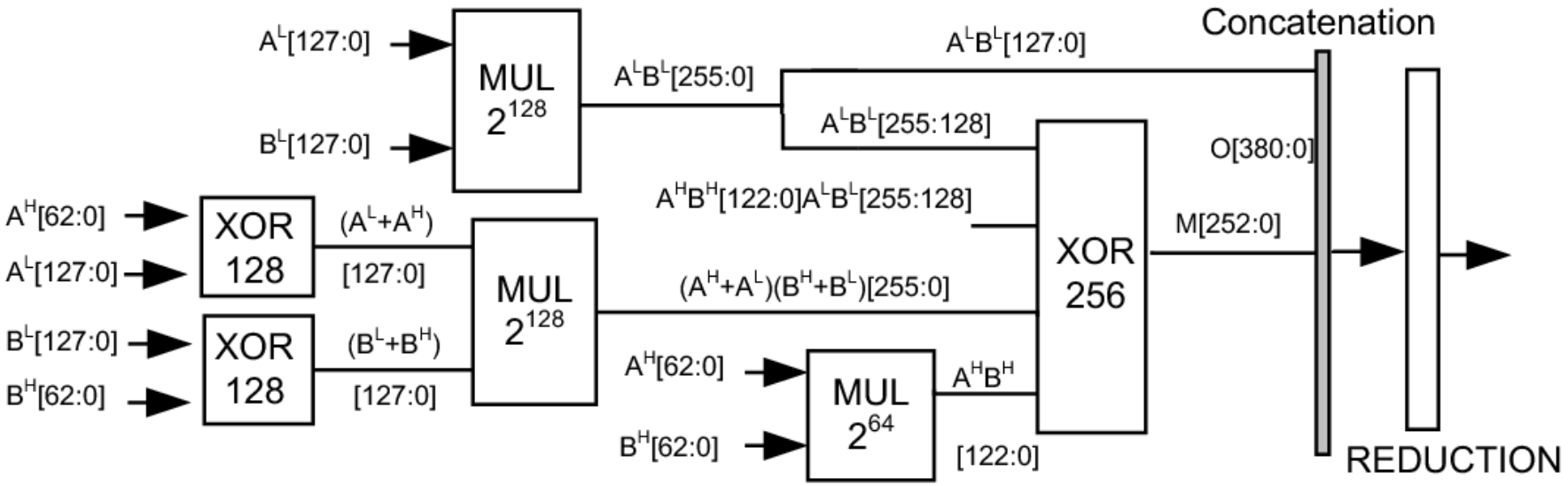
Polynomial multiplication

This operation can be recursively repeated until the bit level

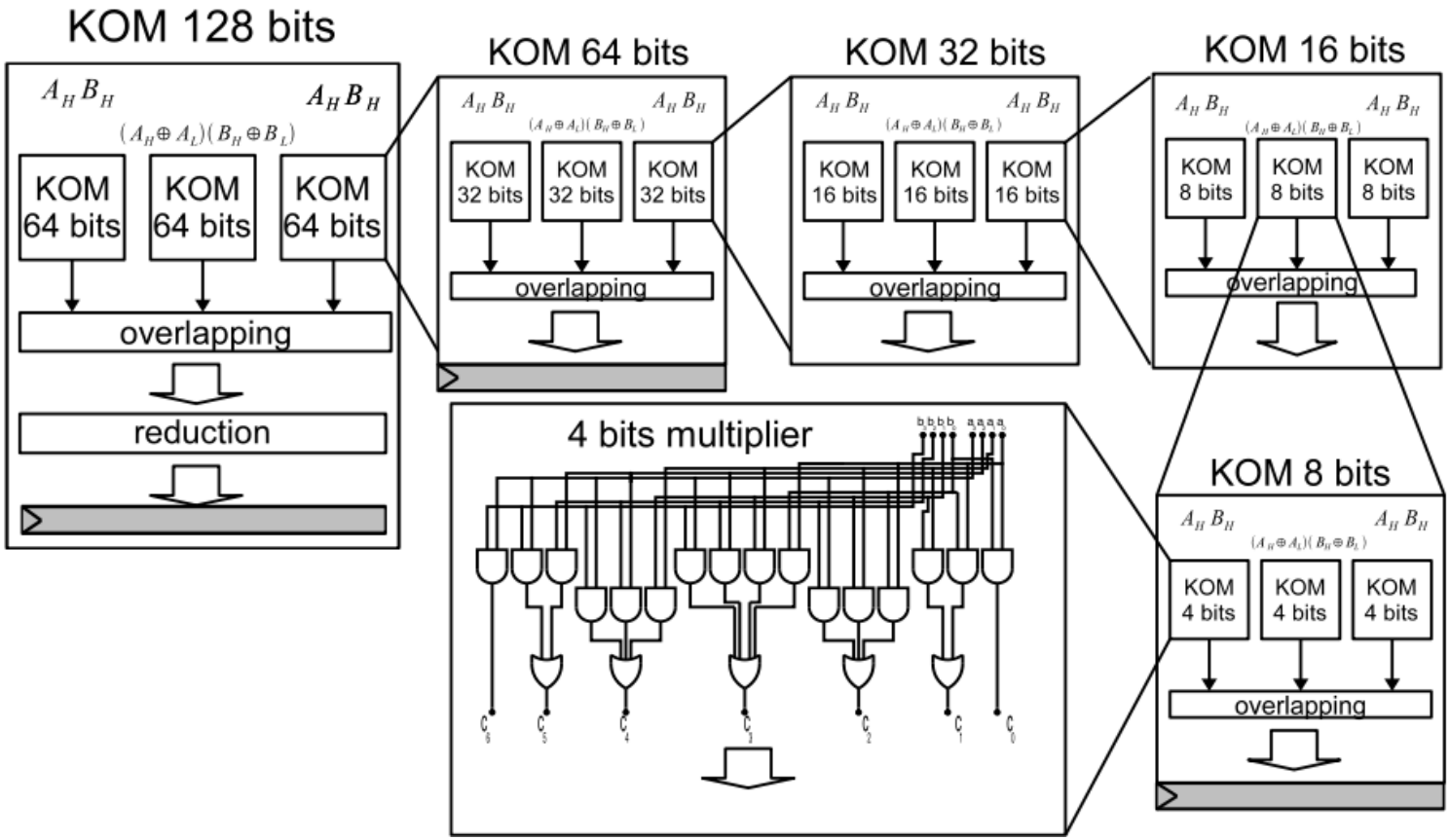


Fully parallel Karatusuba Multiplier

A Fully parallel Karatusuba Multiplier can compute one product every clock cycle but at the price of a large critical path.

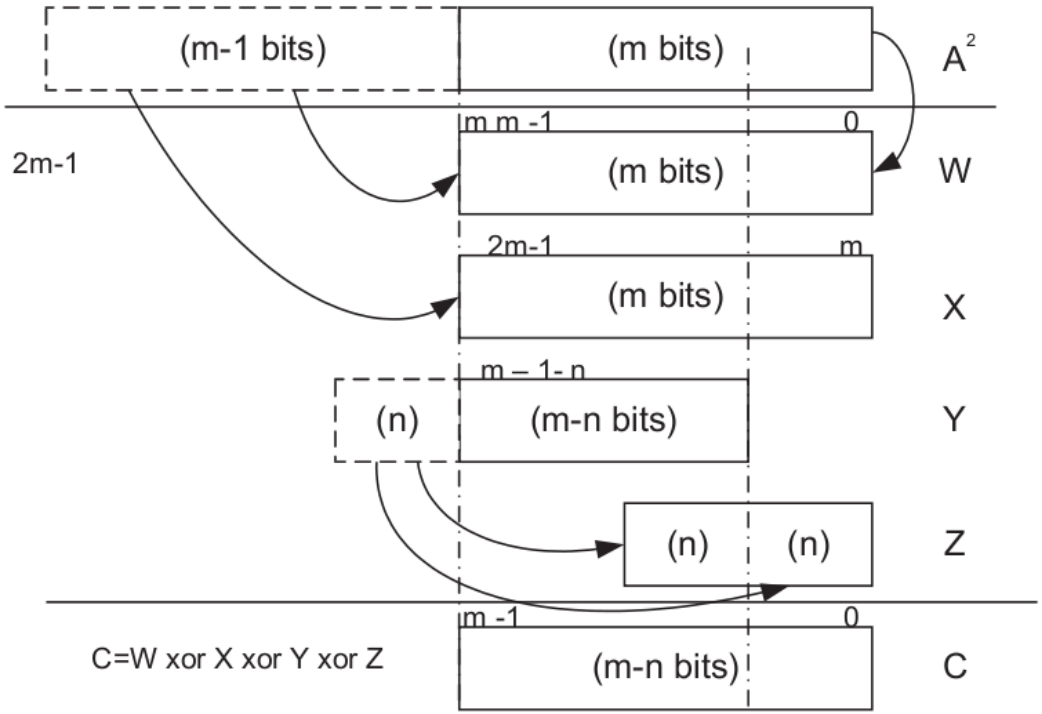


3 Stages Pipelined Karatusuba Multiplier



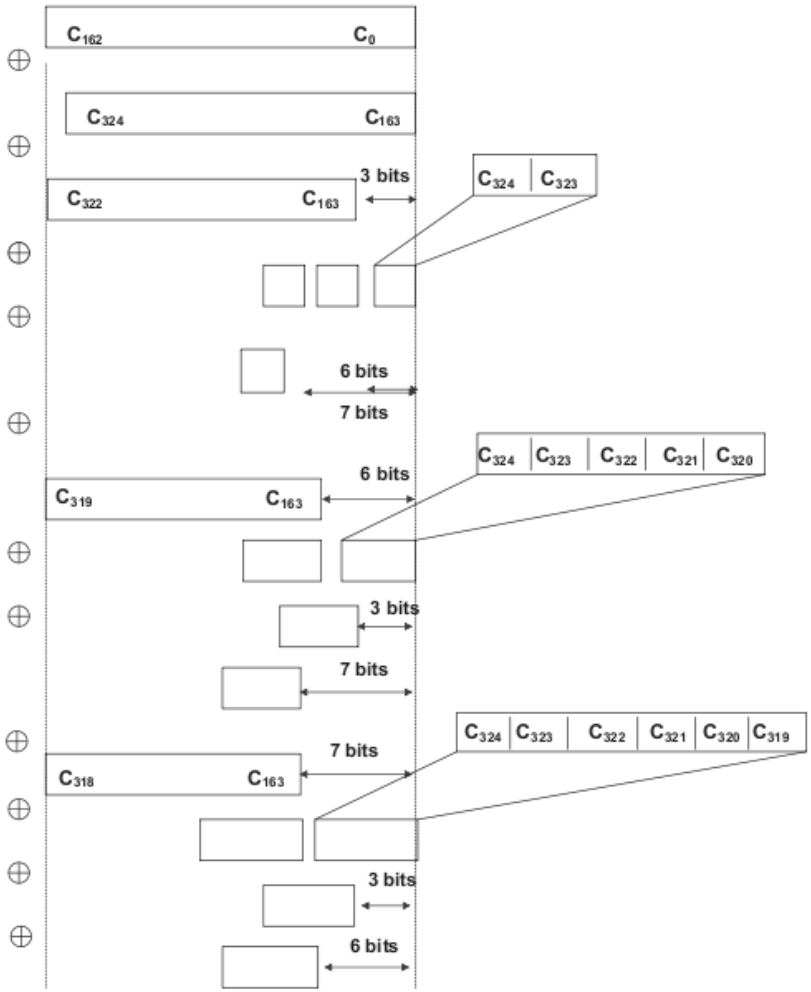
Reduction

Usually the irreducible polynomial f is selected as a trinomial or a pentanomial



Reduction

Usually the irreducible polynomial f is selected as a trinomial or a pentanomial



Agenda

Cinvestav
Centro de Investigación y de Estudios Avanzados del Instituto Politécnico Nacional
50 AÑOS

Block-Ciphers

- Let n be the block length then the block cipher can be seen as a function

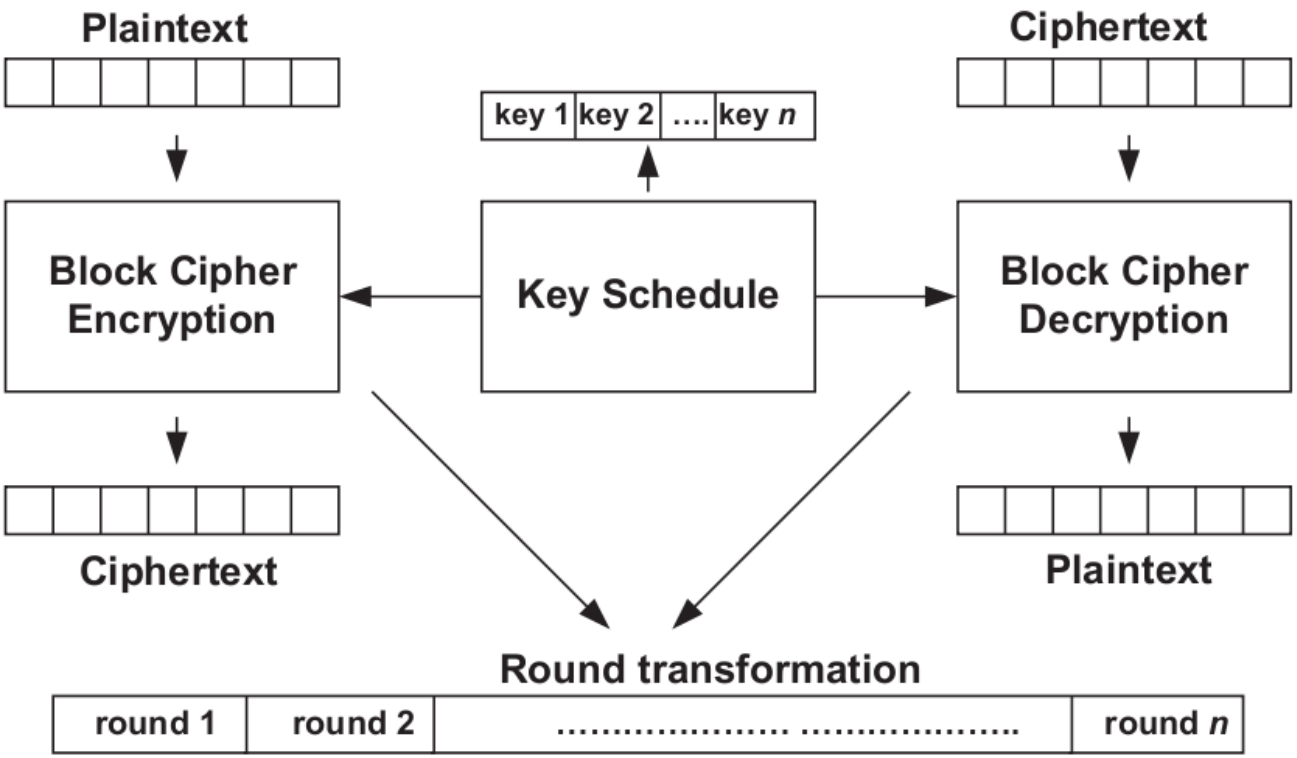
$$E : \{0,1\}^n \times \boldsymbol{K} \rightarrow \{0,1\}^n$$

- Denoted by $E(K, M) = E_K(M)$.
- For each K, E_K must be a permutation. So, each $E_K()$ has an inverse such that

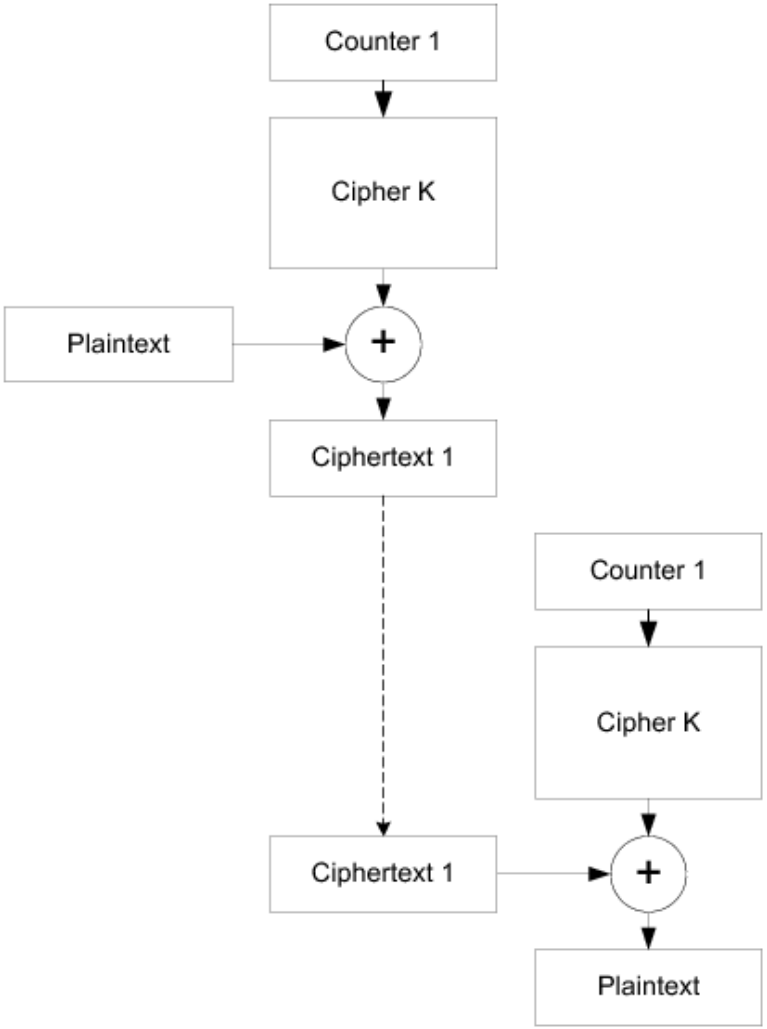
$$D_K(E_K(M)) = M$$

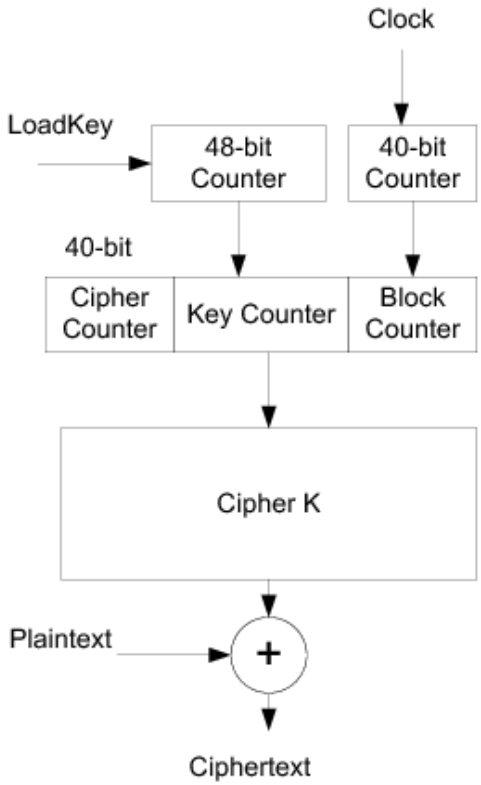
- A secure block cipher is considered to be a **Strong Pseudo Random Permutation (SPRP)**.

Block-Ciphers



Block cipher in counter mode



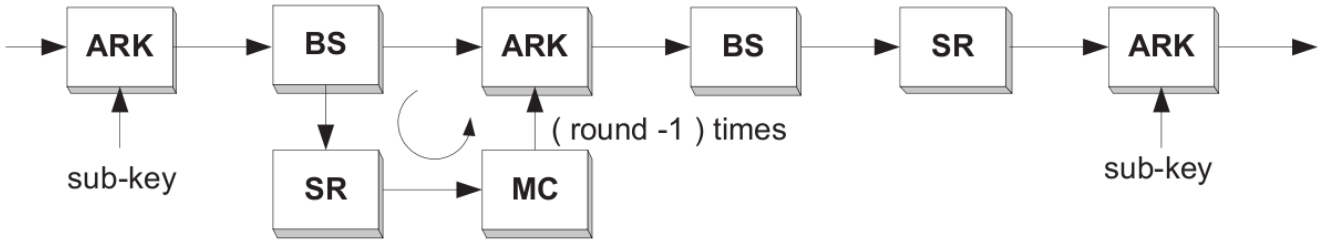


A block-Cipher instantiation: AES

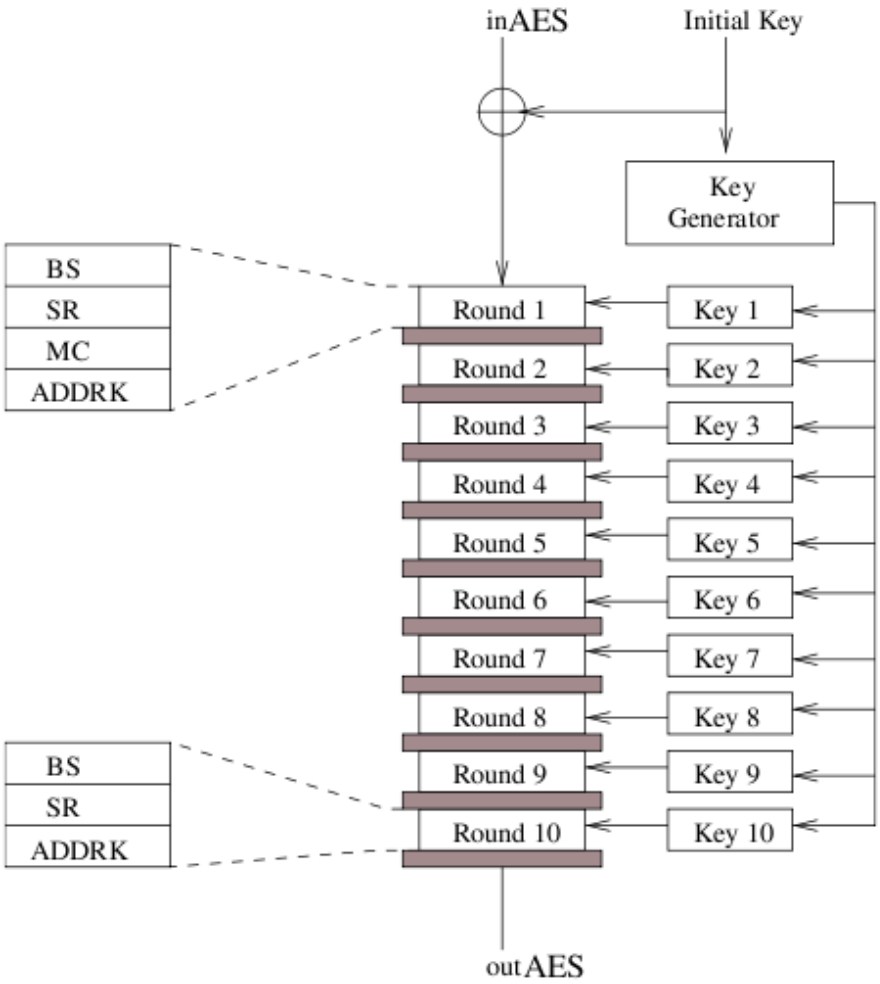
- Rijndael, Designed by Daemen and Rijman, became the industry standard in 2000 when NIST decided to choose it as the winner of the AES contest.
- AES is usually deployed to offer about 128 bits of security, which means that the plaintext, ciphertext and key-length are all equal to 128 bits. In this mode AES performs 10 round transformation to encrypt a single block of data

A block-Cipher instantiation: AES

- Rijndael, Designed by Daemen and Rijman, became the industry standard in 2000 when NIST decided to choose it as the winner of the AES contest.
- AES is usually deployed to offer about 128 bits of security, which means that the plaintext, ciphertext and key-length are all equal to 128 bits. In this mode AES performs 10 round transformation to encrypt a single block of data



Parallel techniques in hardware: AES example



Agenda

Cinvestav
Centro de Investigación y de Estudios Avanzados del Instituto Politécnico Nacional
50 AÑOS

Polynomial Hash

- Informally a hash function maps a big string into a small one. Among those function, there exists a specific type of hash called the polynomial hash

$$H : \{0,1\}^n \times \{0,1\}^{nm} \rightarrow \{0,1\}^n$$

defined as

$$H_h(P_1||...||P_m) = P_1 h^m \oplus P_2 h^{m-1} \oplus ... \oplus P_m h$$

All operations are in $GF(2^n)$,$h, P_i \in \{0,1\}^n$

Polynomial Hash

- Informally a hash function maps a big string into a small one. Among those function, there exists a specific type of hash called the polynomial hash

$$H : \{0,1\}^n \times \{0,1\}^{nm} \rightarrow \{0,1\}^n$$

defined as

$$H_h(P_1||...||P_m) = P_1 h^m \oplus P_2 h^{m-1} \oplus ... \oplus P_m h$$

All operations are in $GF(2^n)$,$h, P_i \in \{0,1\}^n$

- This type of functions are AXU (almost xor universal hash), because for any $G \in \{0,1\}^n$, and $P \neq P'$.

$$\Pr[h \xleftarrow{\$} \{0,1\}^n : H_h(P) \oplus H_h(P^{'}) = G] \leq \frac{maxdegree(P, P^{'})}{2^n}$$

Polynomial hash

- Polynomial hashes are an important part of many cryptographic protocols like message authentication codes, authenticated encryption, tweakable enciphering schemes (TES), etc.

Polynomial hash

- These schemes generally involve the computation of an univariate polynomial of degree $m-1$ defined over a finite field $\mathbb{F}_q$ as,

$$\text{Poly}_h(X) = x_1 h^{m-1} + x_2 h^{m-2} + \cdots + x_{m-1} h + x_m,$$

where $X = (x_1, \ldots, x_m) \in \mathbb{F}_q^m$ and $h \in \mathbb{F}_q$.

Polynomial hash

- These schemes generally involve the computation of an univariate polynomial of degree $m-1$ defined over a finite field $\mathbb{F}_q$ as,

$$\text{Poly}_h(X) = x_1 h^{m-1} + x_2 h^{m-2} + \cdots + x_{m-1} h + x_m,$$

  where $X = (x_1, \ldots, x_m) \in \mathbb{F}_q^m$ and $h \in \mathbb{F}_q$.
- Traditionally, the evaluation of $\text{Poly}_h(X)$ has been done using Horner's rule, which requires $(m-1)$ multiplications and $m-1$ additions in $\mathbb{F}_q$.

BRW Polynomials

- BRW polynomials were introduced by Bernstein in 2007, although the the origin of these polynomials can be traced back to Rabin and Winograd in 1972

BRW Polynomials

- Unlike the normal polynomial they can be evaluated using only $\lfloor \frac{m}{2} \rfloor$ multiplications in $\mathbb{F}_q$ and $\lceil \log_2 m \rceil$ squarings. Thus, these polynomials potentially offer a computational advantage over the normal ones.

BRW Polynomials

- Unlike the normal polynomial they can be evaluated using only $\lfloor \frac{m}{2} \rfloor$ multiplications in $\mathbb{F}_q$ and $\lceil \log_2 m \rceil$ squarings. Thus, these polynomials potentially offer a computational advantage over the normal ones.
- the recursive definition of a BRW polynomial gives it a certain structure which is amenable to parallelization.

BRW Polynomials

- Unlike the normal polynomial they can be evaluated using only $\lfloor \frac{m}{2} \rfloor$ multiplications in $\mathbb{F}_q$ and $\lceil \log_2 m \rceil$ squarings. Thus, these polynomials potentially offer a computational advantage over the normal ones.
- the recursive definition of a BRW polynomial gives it a certain structure which is amenable to parallelization.
- It turns out that to take advantage of this parallel structure one needs to carefully schedule the order of multiplications involved in the polynomial evaluation.

BRW definition

The BRW polynomial BRW is defined recursively as follows:

$$
\begin{aligned}
\mathrm{BRW}_h() &= 0 \\
\mathrm{BRW}_h(x_1) &= x_1 \\
\mathrm{BRW}_h(x_1, x_2) &= x_1 + x_2 h \\
\mathrm{BRW}_h(x_1, x_2, x_3) &= (h + x_1)(h^2 + x_2) + x_3 \\
\mathrm{BRW}_h(x_1, x_2, \ldots, x_m) &= \mathrm{BRW}_h(x_1, x_2, \ldots, x_{t-1})(h^t + x_t) + \\
&\quad \mathrm{BRW}_h(x_{t+1}, \ldots, x_m)
\end{aligned}
$$

where $t \in \{4, 8, 16, \ldots\}$ and $t \leq m < 2t$

The number of multiplications is given by $\lfloor \frac{m}{2} \rfloor$.

Additions: $m + \lfloor \frac{m-3}{2} \rfloor$.

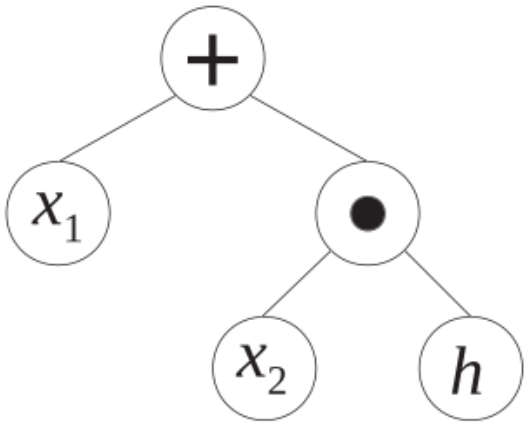
Squarings: $\lfloor lgm \rfloor$.

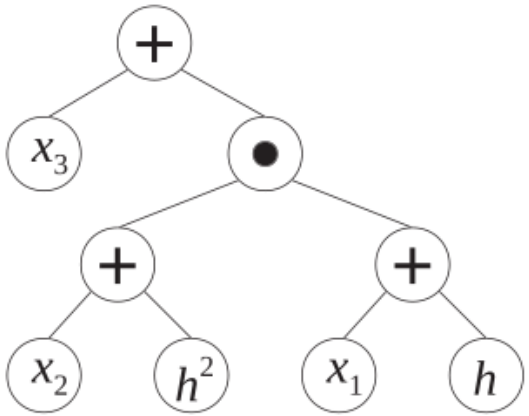
BRW-Polynomials

- A BRW polynomial $H_h(X_1, \ldots, X_m)$ can be represented as a tree T_m which contains three types of nodes, namely, multiplication nodes, addition nodes and leaf nodes
- The tree T_m will be called a BRW tree and can be recursively constructed as we will discuss next

BRW-Polynomials

- A BRW polynomial $H_h(X_1, \ldots, X_m)$ can be represented as a tree T_m which contains three types of nodes, namely, multiplication nodes, addition nodes and leaf nodes
- The tree T_m will be called a BRW tree and can be recursively constructed as we will discuss next





BRW-Polynomials

- We propose a framework to construct an efficient circuit to compute BRW polynomials using a pipelined multiplier.
- To achieve a good performance in the implementations of BRW polynomial, there are two important aspects:
  - Scheduling of the blocks of information, trying to have the pipeline always full.
  - The number of accumulators or registers required.

BRW-Polynomials Representation

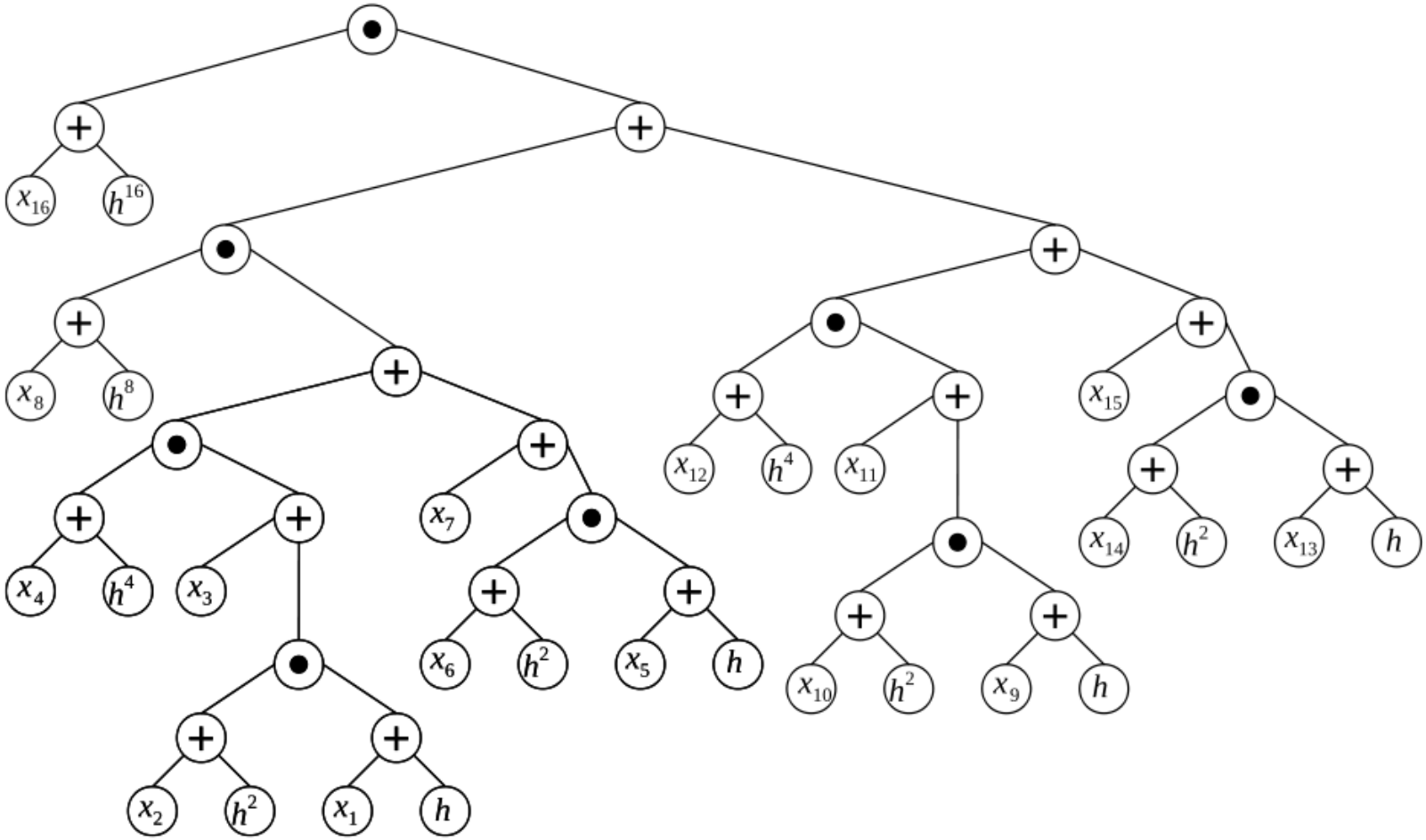
Let's see the BRW-Polynomial with 16 coefficients

$$\begin{aligned} \mathrm{BRW}_h(x_1, ..., x_{16}) &= ((((h + x_1)(h^2 + x_2) + x_3)(h^4 + x_4) \\ &+ (h + x_5)(h^2 + x_6) + x_7)(h^8 + x_8) \\ &+ ((h + x_9)(h^2 + x_{10}) + x_{11})(h^4 + x_{12}) \\ &+ (h + x_{13})(h^2 + x_{14}) + x_{15})(h^{16} + x_{16}) \end{aligned}$$

The total number of operations are 8 multiplications, 4 squarings and 19 additions.

BRW-Polynomials Representation

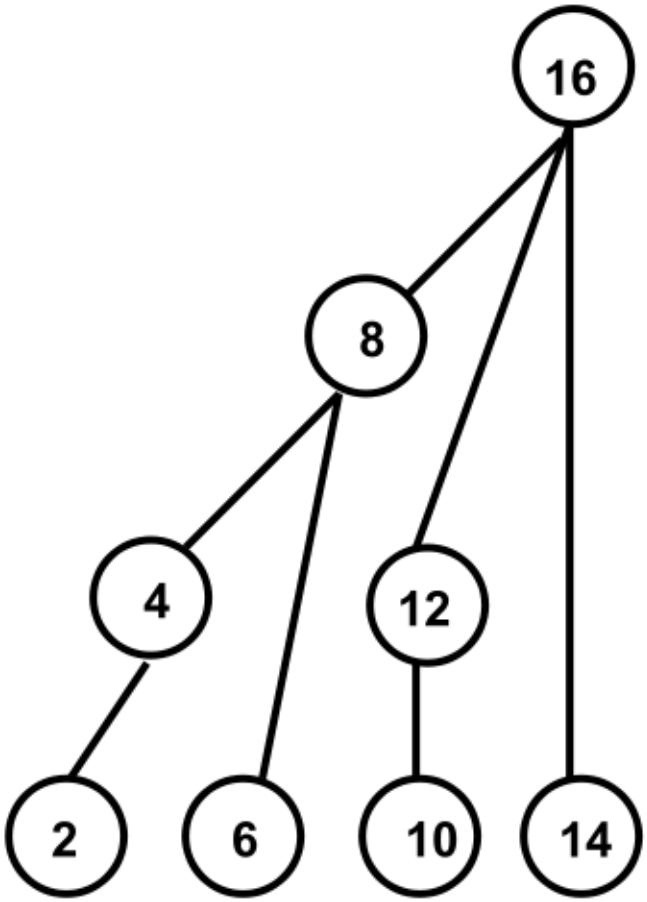
It can be represented as a tree T_m.



$$\begin{aligned}\mathrm{BRW}_h(x_1,\ldots,x_{16}) = {} & ((((h+x_1)(h^2+x_2)+x_3)(h^4+x_4)+(h+x_5)(h^2+x_6)+x_7)(h^8+x_8) \\ & +((h+x_9)(h^2+x_{10})+x_{11})(h^4+x_{12})+(h+x_{13})(h^2+x_{14})+x_{15})(h^{16}+x_{16})\end{aligned}$$

BRW-Polynomials Representation

It can be represented as a tree T_m.

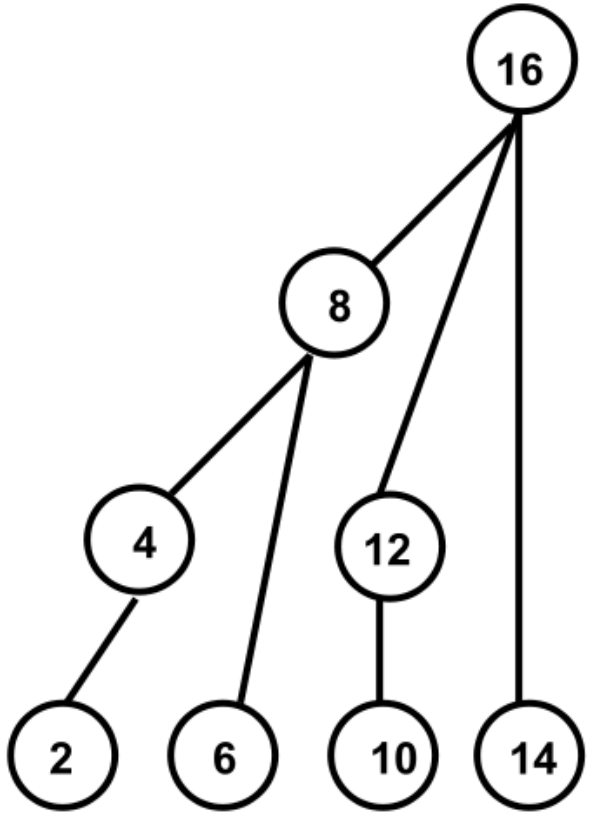


$$\begin{aligned} \mathrm{BRW}_h(x_1, ..., x_{16}) \quad = \quad & ((((h + x_1)(h^2 + x_2) + x_3)(h^4 + x_4) + (h + x_5)(h^2 + x_6) + x_7)(h^8 + x_8) \\ & +((h + x_9)(h^2 + x_{10}) + x_{11})(h^4 + x_{12}) + (h + x_{13})(h^2 + x_{14}) + x_{15})(h^{16} + x_{16}) \end{aligned}$$

Some Properties of the Tree

Let $p = \lfloor \frac{m}{2} \rfloor$ and k is a label of a node.

- Number of nodes in T_m is p.
- The number of connected components is given by hamming weight of p.
- If the bit i of p is 1, T_m contains a tree of size 2^i.
- If $k \equiv 2 \, mod \, 4$, then k is an independent node.
- If $k \equiv 0 \, mod \, 8$, k has at least $k-2$ and $k-4$ as its children.
- If $k \equiv 4 \, mod \, 8$, $k-2$ is the only child of k.



Scheduling Algorithm

Algorithm 1 Scheduling algorithm

Input: number of blocks m, number of pipe stages NS
Output: sequence of multiplications to be done

1. Construct the collapsed forest F_m;
2. **for** each node x in F_m **do**
3. $\quad x$.NC $\leftarrow$ number of children of x;
4. $\quad x$.ST $\leftarrow$ undefined;
5. $\quad$ **if** $level_{F_m(x)=0}$ **then**
6. $\quad\quad$ Insert(x, L_1);
7. $L_2 \leftarrow$ Empty;
8. clock $\leftarrow 1$;
9. **while** (L_1 and L_2 are both not empty) **do**
10. $\quad x \leftarrow$ Pop(L_2);
11. $\quad$ **if** ($x \neq$ NULL and clock $- x$.ST $>$ NS) **then**
12. $\quad\quad$ Process$(x, L_2,$ clock$)$;
13. $\quad$ **else**
14. $\quad\quad x \leftarrow$ Pop(L_1);
15. $\quad\quad$ **if** ($x \neq$ NULL) **then**
16. $\quad\quad\quad$ Process$(x, L_1,$ clock$)$;
17. $\quad$ clock $\leftarrow$ clock $+ 1$;
18. **Function** Process$(x, L,$ *clock*$)$
19. Delete(L);
20. $y \leftarrow$ Parent(x);
21. Output x;
22. **if** $y \neq$ NULL **then**
23. $\quad y$.NC $\leftarrow y$.NC $- 1$
24. $\quad$ **if** y.NC $= 0$ **then**
25. $\quad\quad y$.ST $=$ *clock*
26. $\quad\quad$ Insert(y, L_2);
27. **return**

Scheduling algorithm: an example

As an example let us consider the case when $m = 16$ and $\mathrm{NS} = 2$.

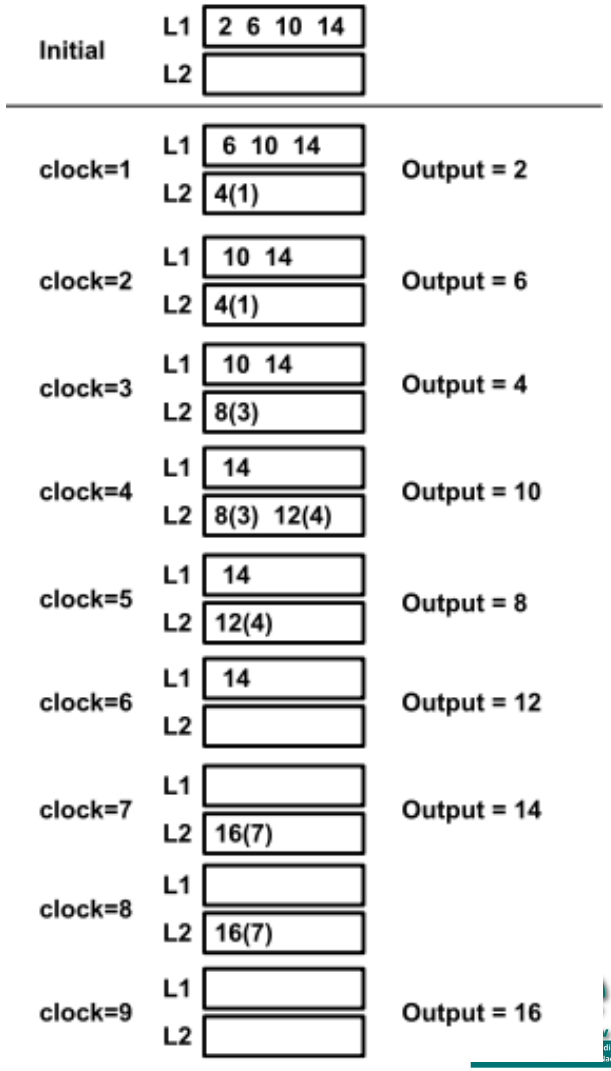
$$
\begin{aligned}
M_1\text{:} &\quad R_1 = (X_2 + h^2)(X_1 + h);\\
M_2\text{:} &\quad R_2 = (X_6 + h^2)(X_5 + h);\\
M_3\text{:} &\quad R_3 = (X_4 + h^4)(X_3 + R_1);\\
M_4\text{:} &\quad R_4 = (X_{10} + h^2)(X_9 + h);\\
M_5\text{:} &\quad R_5 = (X_8 + h^8)(R_3 + R_2 + X_7);\\
M_6\text{:} &\quad R_6 = (X_{12} + h^4)(X_{11} + R_4);\\
M_7\text{:} &\quad R_7 = (X_{14} + h^2)(X_{13} + h);\\
M_8\text{:} &\quad R_8 = (X_{16} + h^{16})(R_5 + R_6 + R_7 + X_{15}).
\end{aligned}
$$

Scheduling algorithm: an example

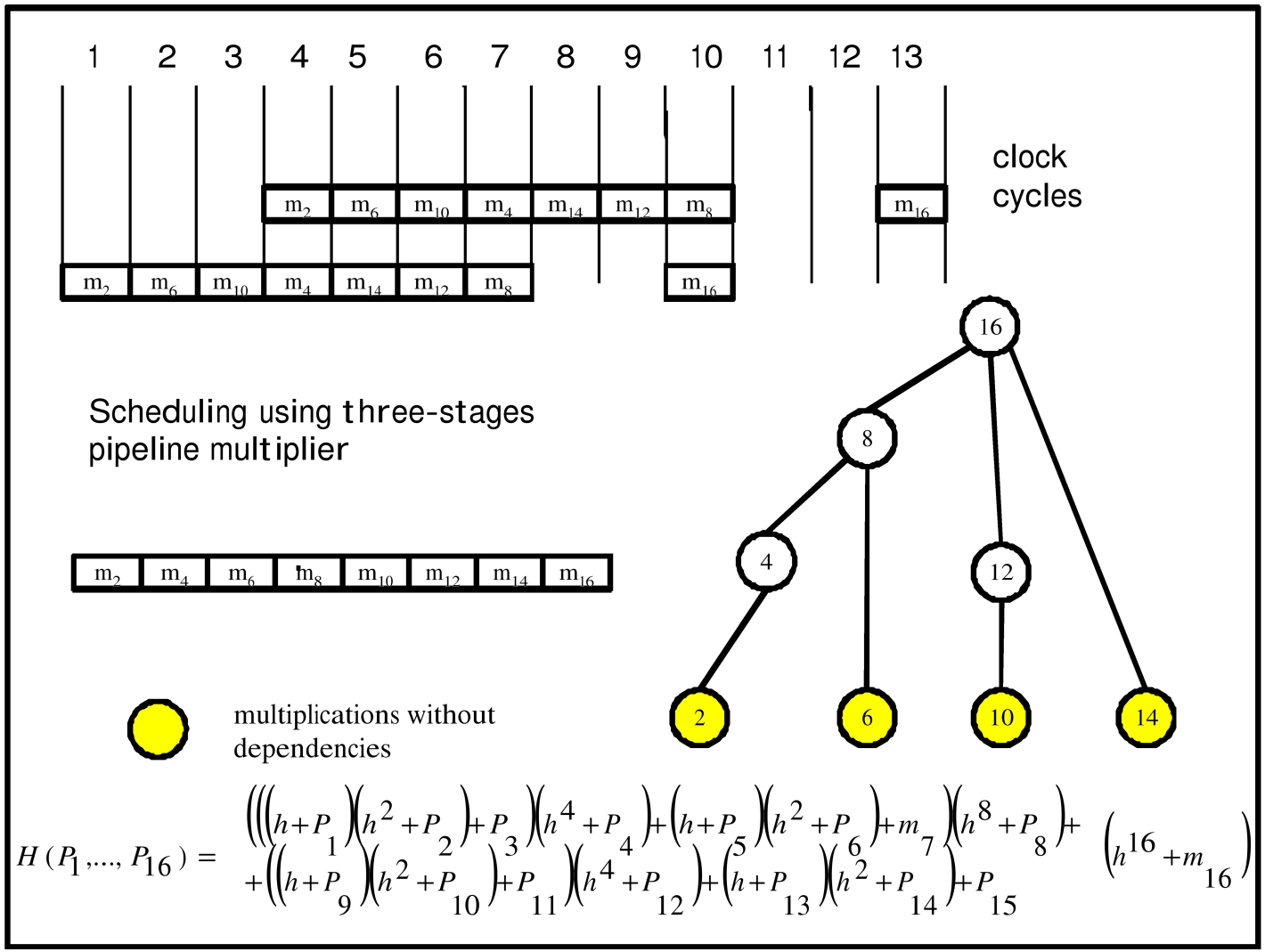
As an example let us consider the case when $m = 16$ and NS $= 2$.

$$
\begin{aligned}
M_1: &\quad R_1 = (X_2 + h^2)(X_1 + h);\\
M_2: &\quad R_2 = (X_6 + h^2)(X_5 + h);\\
M_3: &\quad R_3 = (X_4 + h^4)(X_3 + R_1);\\
M_4: &\quad R_4 = (X_{10} + h^2)(X_9 + h);\\
M_5: &\quad R_5 = (X_8 + h^8)(R_3 + R_2 + X_7);\\
M_6: &\quad R_6 = (X_{12} + h^4)(X_{11} + R_4);\\
M_7: &\quad R_7 = (X_{14} + h^2)(X_{13} + h);\\
M_8: &\quad R_8 = (X_{16} + h^{16})(R_5 + R_6 + R_7 + X_{15}).
\end{aligned}
$$

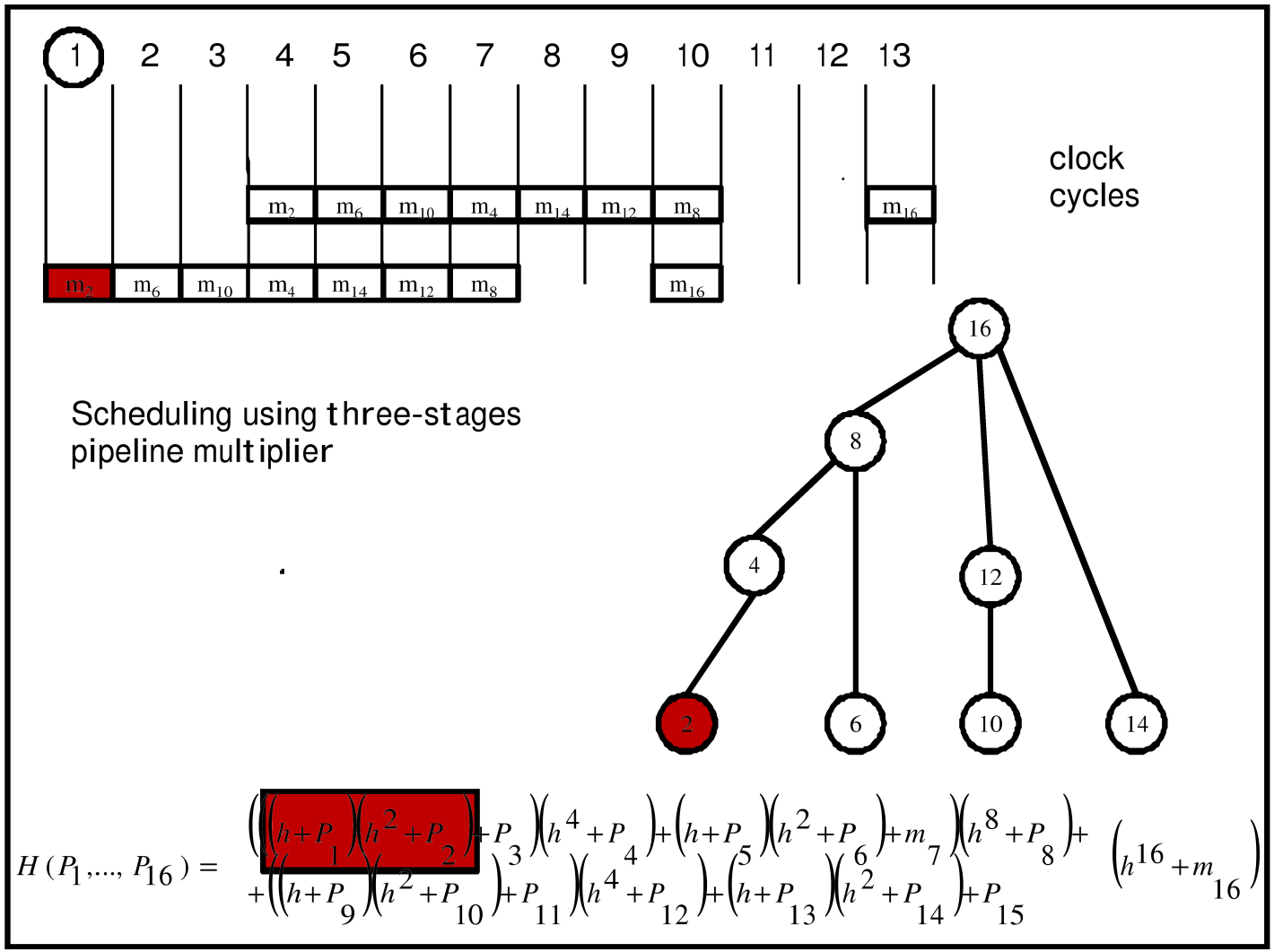
This example requires 9 clock cycles to be executed (there is one delay)



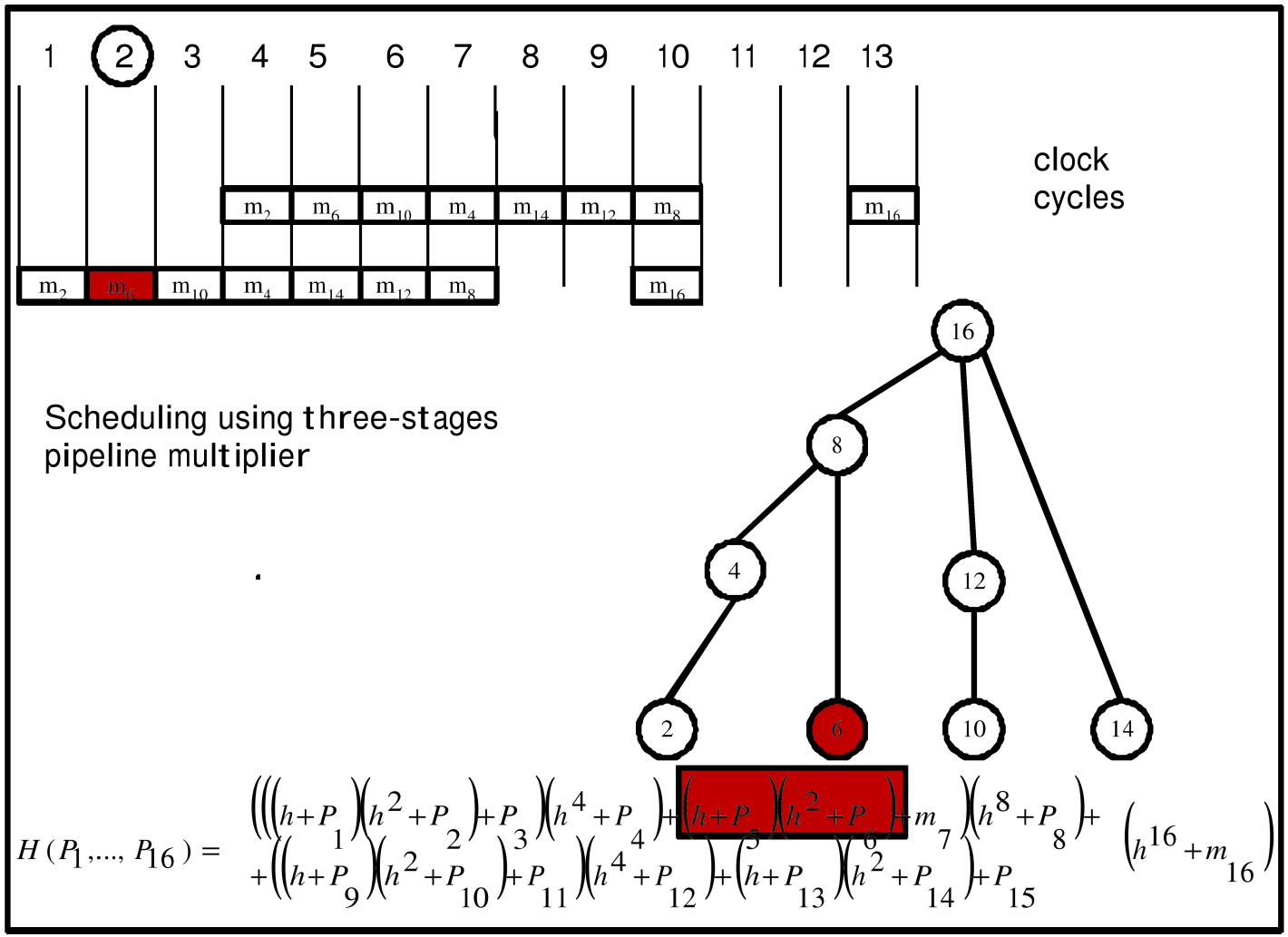
Scheduling of the blocks



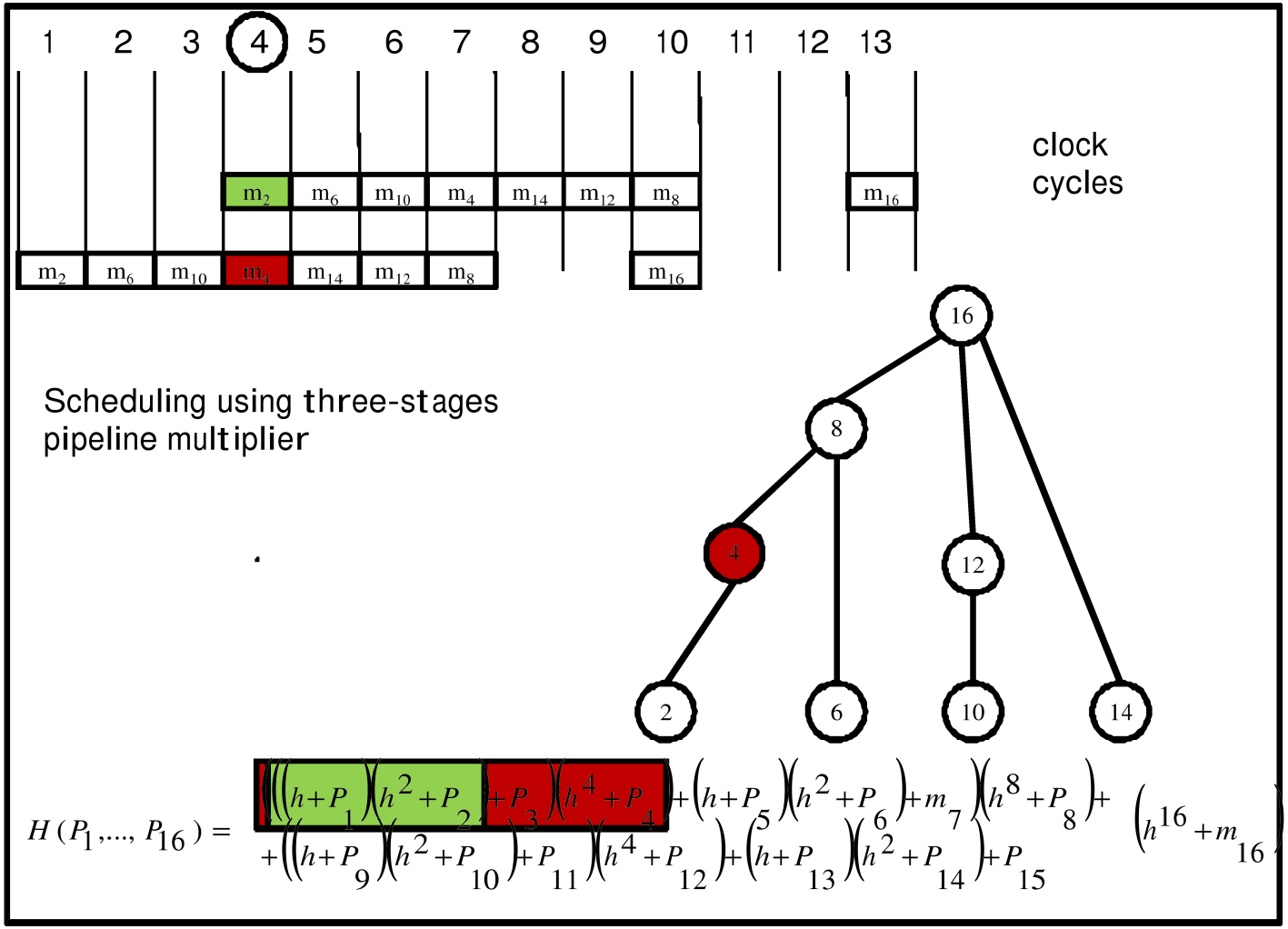
Scheduling of the blocks



Scheduling of the blocks



Scheduling of the blocks



Scheduling of the blocks

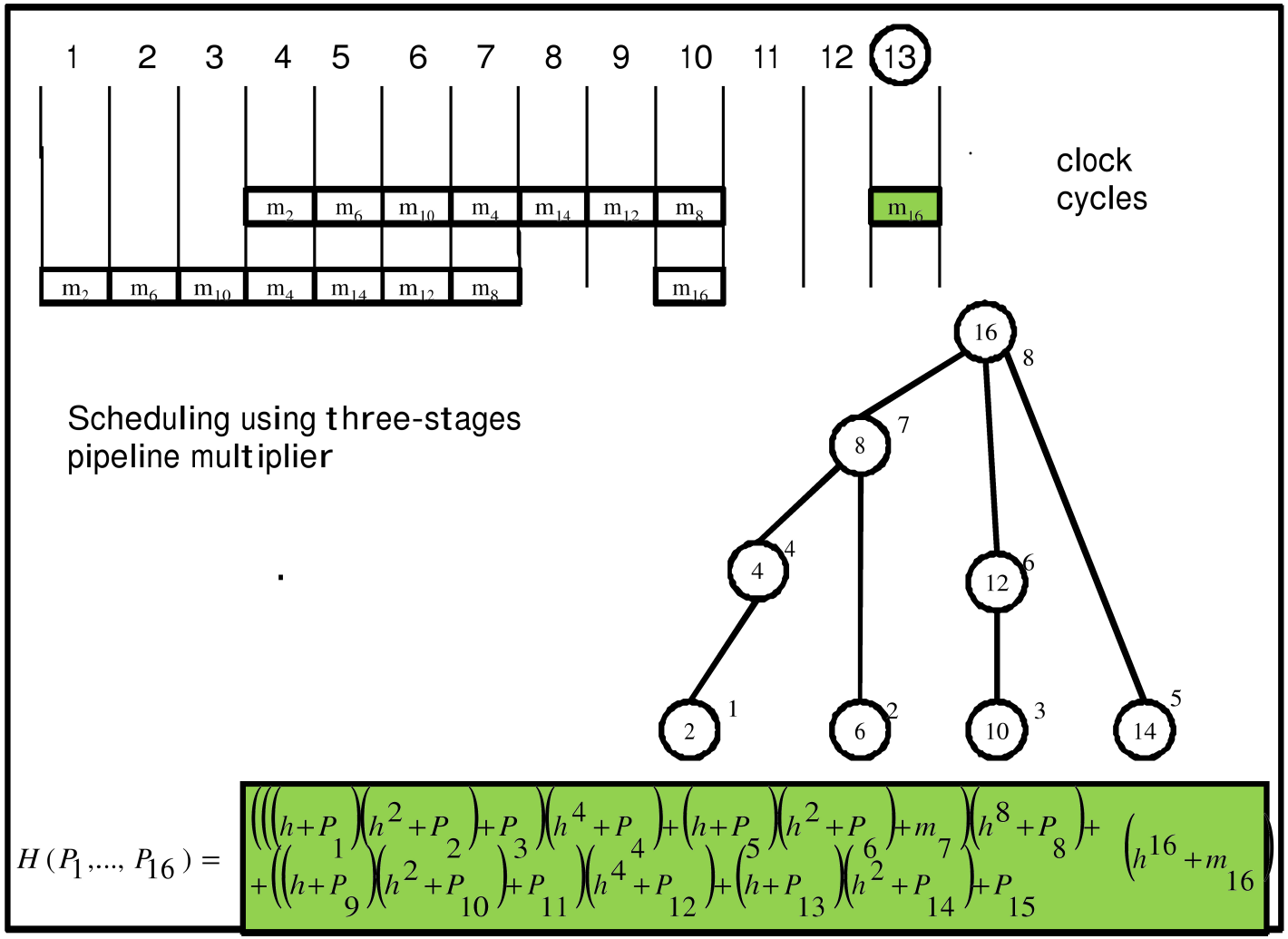


Table: The output of Schedule for NS = 2 for small number of blocks

Number of pipeline stages NS=2

Blocks (m)	Clock 1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Total clocks
2	*2*															1
4	*2*	–	*4*													3
6	*2*	*6*	*4*													3
8	*2*	*6*	*4*	–	*8*											5
10	*2*	*6*	*4*	*10*	*8*											5
12	*2*	*6*	*4*	*10*	*8*	*12*										6
14	*2*	*6*	*4*	*10*	*8*	*12*	*14*									7
16	*2*	*6*	*4*	*10*	*8*	*12*	*14*	–	*16*							9
18	*2*	*6*	*4*	*10*	*8*	*12*	*14*	*18*	*16*							9
20	*2*	*6*	*4*	*10*	*8*	*12*	*14*	*18*	*16*	*20*						10
22	*2*	*6*	*4*	*10*	*8*	*12*	*14*	*18*	*16*	*20*	*22*					11
24	*2*	*6*	*4*	*10*	*8*	*12*	*14*	*18*	*16*	*20*	*22*	–	*24*			13
26	*2*	*6*	*4*	*10*	*8*	*12*	*14*	*18*	*16*	*20*	*22*	*26*	*24*			13
28	*2*	*6*	*4*	*10*	*8*	*12*	*14*	*18*	*16*	*20*	*22*	*26*	*24*	*28*		14
30	*2*	*6*	*4*	*10*	*8*	*12*	*14*	*18*	*16*	*20*	*22*	*26*	*24*	*28*	*30*	15

Table: The output of Schedule for NS = 3 for small number of blocks

Number of pipeline stages NS=3

Blocks (m)	Clock 1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Total clocks
2	*2*															1
4	*2*	–	–	*4*												4
6	*2*	*6*	–	*4*												4
8	*2*	*6*	–	*4*	–	–	*8*									7
10	*2*	*6*	*10*	*4*	–	–	*8*									7
12	*2*	*6*	*10*	*4*	–	*12*	*8*									7
14	*2*	*6*	*10*	*4*	*14*	*12*	*8*									7
16	*2*	*6*	*10*	*4*	*14*	*12*	*8*	–	–	*16*						10
18	*2*	*6*	*10*	*4*	*14*	*12*	*8*	*18*	–	*16*						10
20	*2*	*6*	*10*	*4*	*14*	*12*	*8*	*18*	–	*16*	*20*					11
22	*2*	*6*	*10*	*4*	*14*	*12*	*8*	*18*	*22*	*16*	*20*					11
24	*2*	*6*	*10*	*4*	*14*	*12*	*8*	*18*	*22*	*16*	*20*	–	–	*24*		14
26	*2*	*6*	*10*	*4*	*14*	*12*	*8*	*18*	*22*	*16*	*20*	*26*	–	*24*		14
28	*2*	*6*	*10*	*4*	*14*	*12*	*8*	*18*	*22*	*16*	*20*	*26*	–	*24*	*28*	15
30	*2*	*6*	*10*	*4*	*14*	*12*	*8*	*18*	*22*	*16*	*20*	*26*	*30*	*24*	*28*	15

Optimal Scheduling

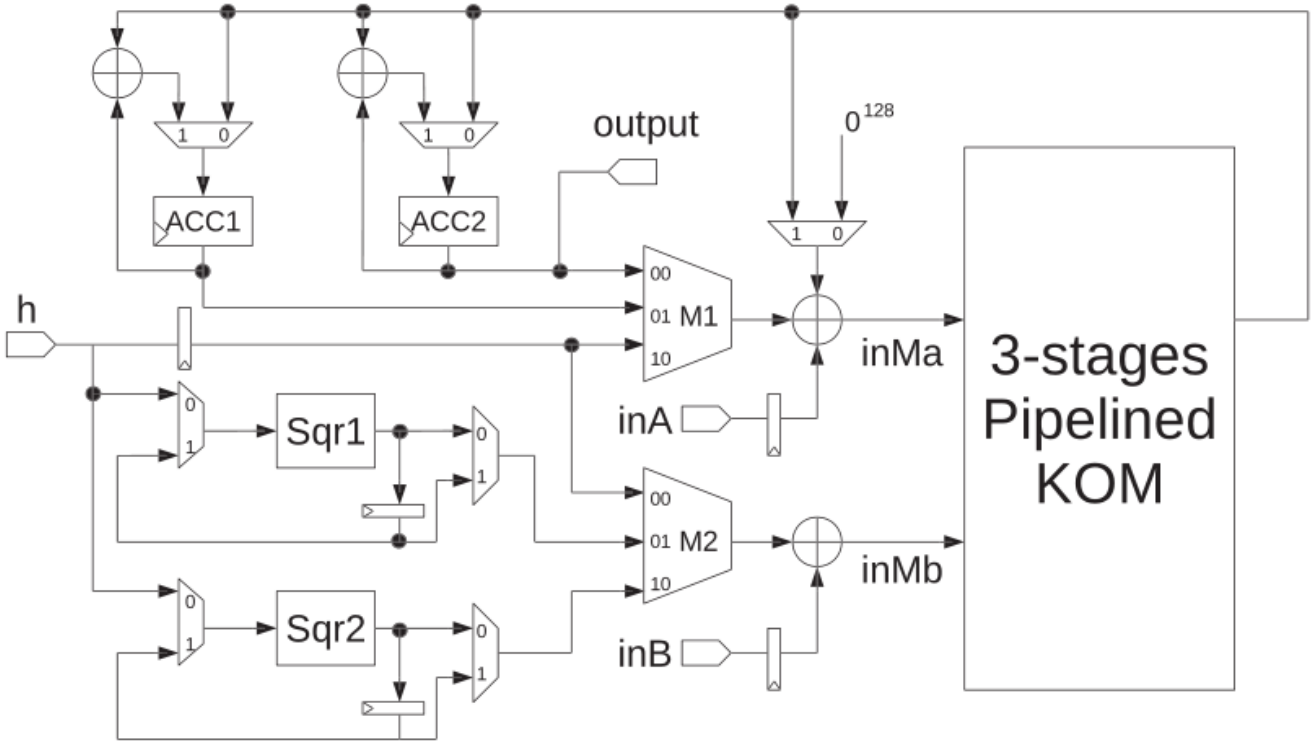
Theorem

Let $H_h(X_1, X_2, \ldots, X_m)$ *be a BRW polynomial and let* $p = \lfloor m/2 \rfloor$ *be the number of nodes in the corresponding collapsed tree. Let clks be the number of clock cycles taken by* Schedule *to schedule all nodes, then,*

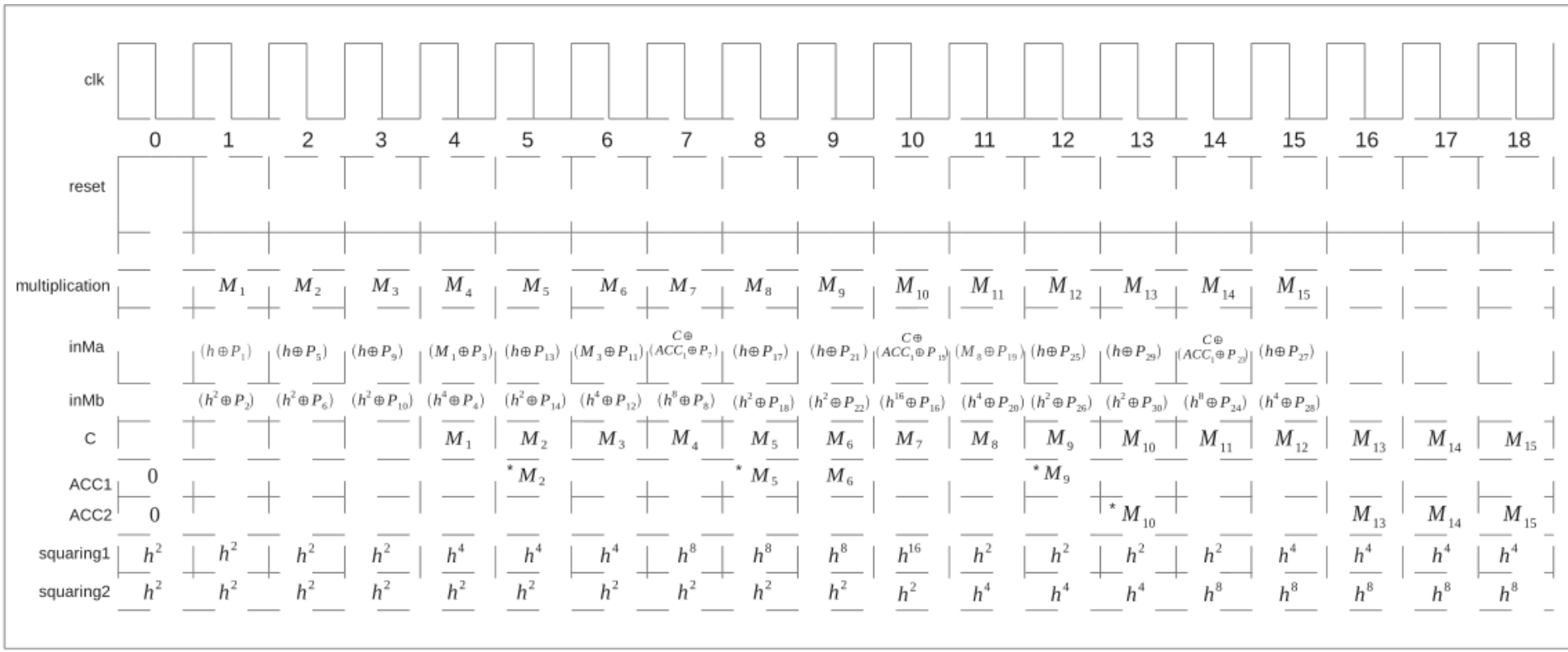
1. *If* NS $= 2$, *and* $p \geq 3$, $clks = p + 1$ *if* $p \equiv 0 \bmod 4$; *and* $clks = p$ *otherwise.*
2. *If* NS $= 3$ *and* $p \geq 7$, *then*

$$clks = \begin{cases} p+2 & \text{if } p \equiv 0 \bmod 4 \\ p+1 & \text{if } p \equiv 1 \bmod 4 \\ p+1 & \text{if } p \equiv 2 \bmod 4 \\ p & \text{if } p \equiv 3 \bmod 4 \end{cases}$$

Scheduling of the blocks



Scheduling of the blocks



Tweakable Enciphering Schemes

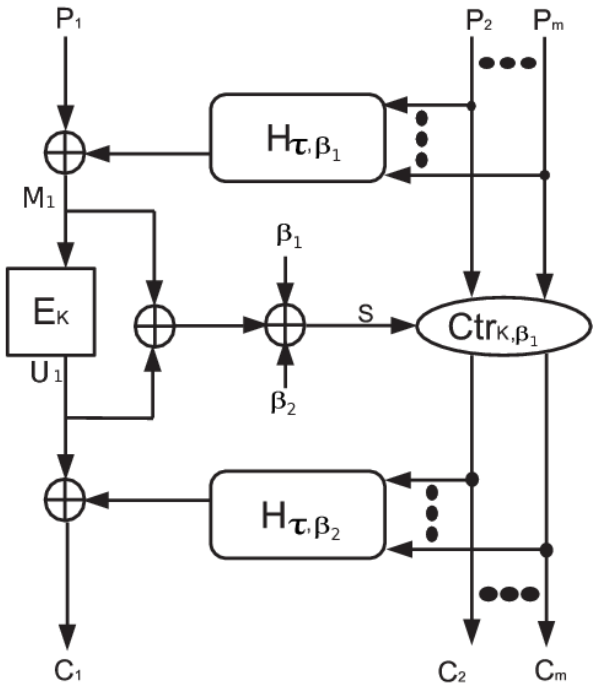
- They are length preserving encryption schemes.
- These schemes takes in an extra public quantity called the tweak.
- They can provide partial authentication.
- A potential application area of such schemes is in-place disk encryption.
- Security of such schemes are that of a strong pseudorandom permutation.

HMCH: A Hash-Counter Hash TES

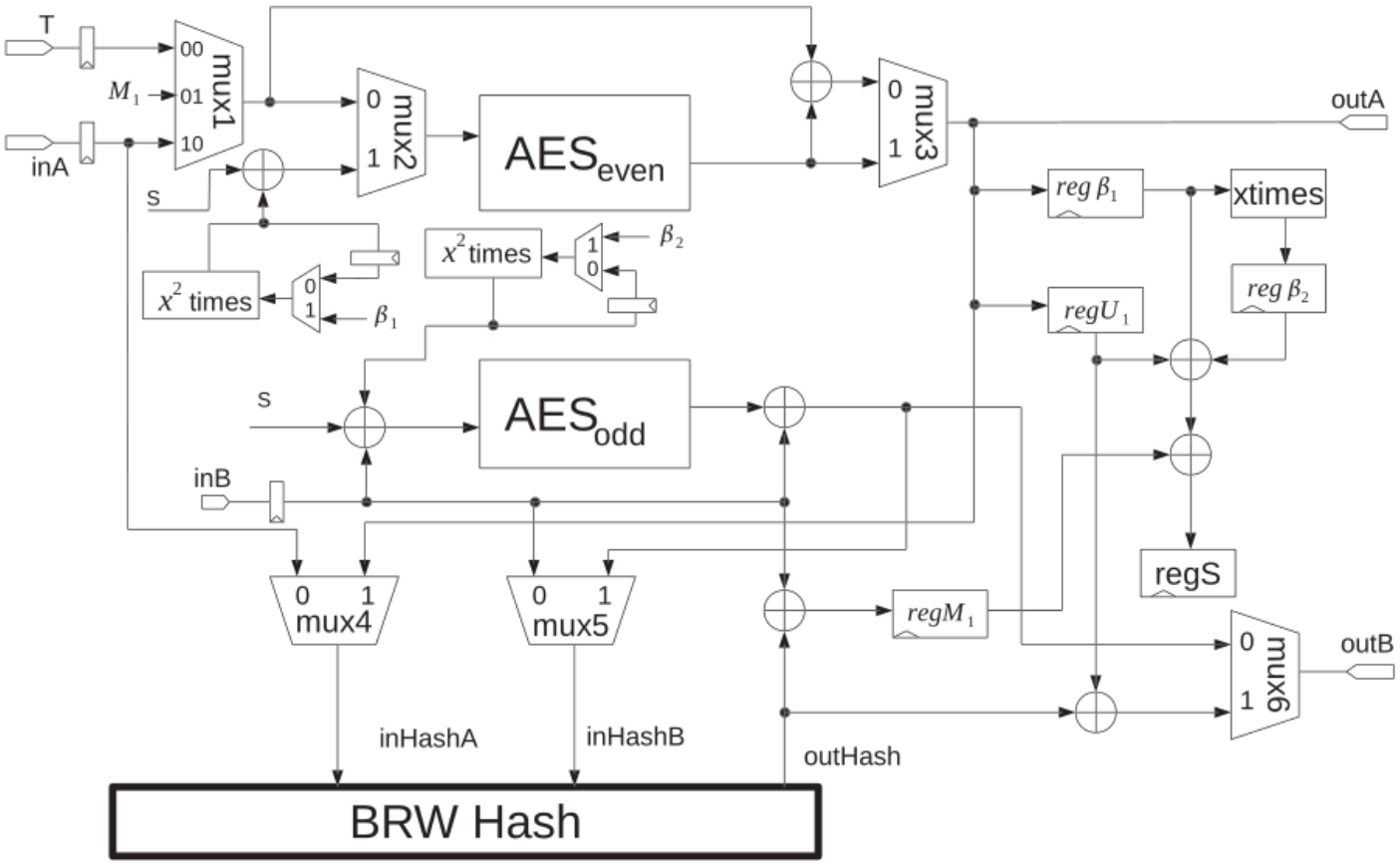
Algorithm 2 HMCH Encryption algorithm

Input: keys h, k, tweak T and message blocks to encrypt $P_1, \ldots, P_m$
Output: ciphertext $(C_1, \ldots, C_m)$

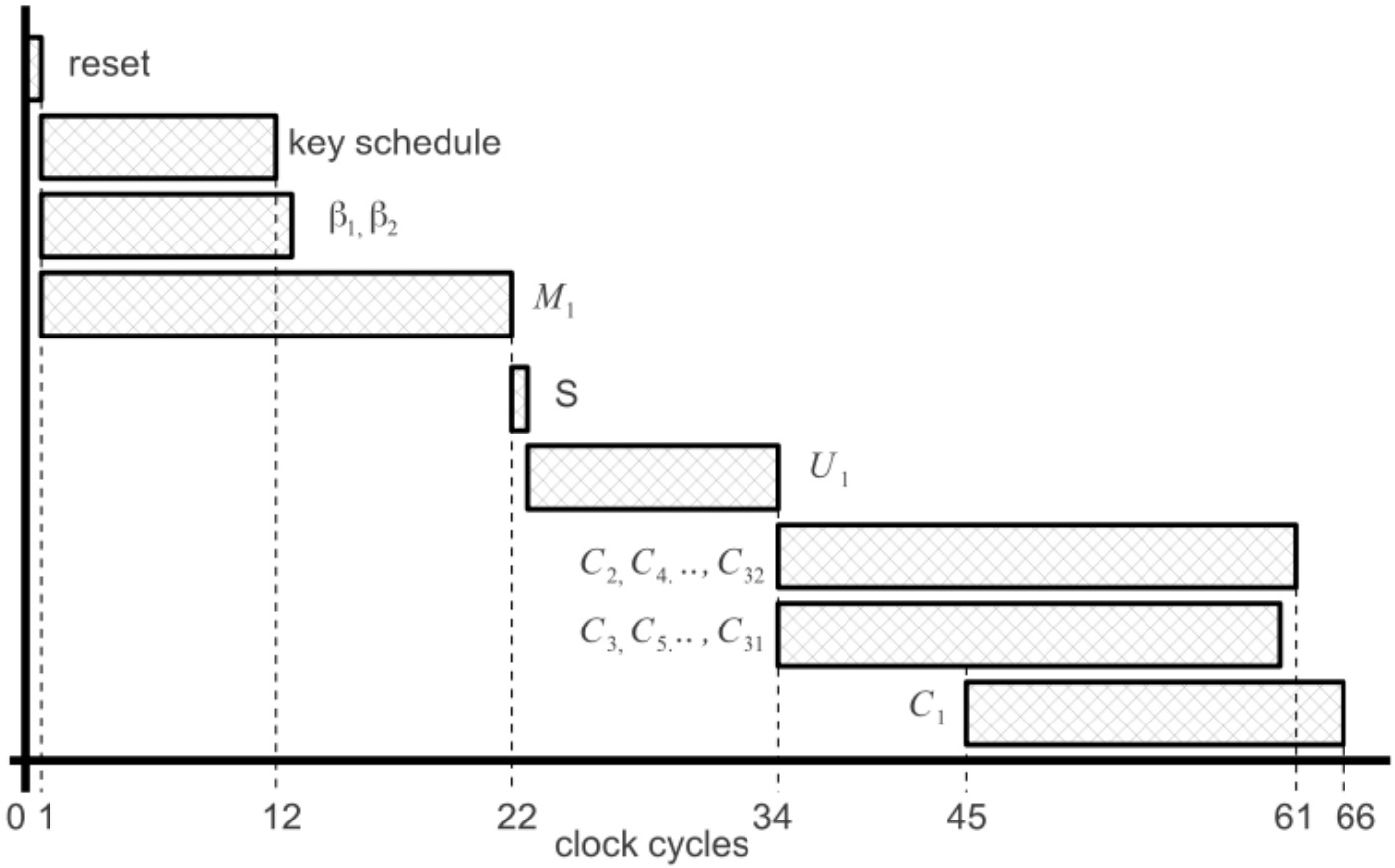
1. $\beta_1 \leftarrow E_K(T)$; $\beta_2 \leftarrow x\beta_1$;
2. $M_1 \leftarrow P_1 \oplus \psi_h(P_2, \ldots, P_m)$;
3. $U_1 \leftarrow E_K(M_1)$;
4. $S \leftarrow M_1 \oplus U_1 \oplus \beta_1 \oplus \beta_2$;
5. **for** $i = 2$ to m **do**
6. $\quad C_i \leftarrow P_i \oplus E_K(x^{i-2}\beta_1 \oplus S)$;
7. $C_1 \leftarrow U_1 \oplus \psi_h(C_2, \ldots, C_m)$;
8. **return** $(C_1, \ldots, C_m)$;



HCMH circuit architecture and time diagram



HCMH circuit architecture and time diagram



Experimental Results

Table: Modes of operation on Virtex-5 device. AES-PEC: AES pipelined encryption core, AES-PDC: AES pipelined decryption core, AES-SDC: AES sequential decryption core, SOF : squares computed on the fly, SPC: squares pre-computed

Mode	Implementation Details	Slices	Frequency (MHz)	Clock Cycles	Time (nS)	Throughput (Gbits/Sec)	$\frac{1}{(\text{Slice} * \text{Time})}$
HMCH[BRW]-1	2 AES-PEC, 1 AES-SDC, SOF	8040	211.785	66	311.637	13.143	399.112
HMCH[BRW]-2	2 AES-PEC, 1 AES-SDC, SPC	8140	212.589	66	310.458	13.193	395.706
HMCH[BRW]-3	1 AES-PEC, 1 AES-SDC, SOF	6112	223.364	80	358.160	11.436	**456.814**
HEH[BRW]-1	2 AES-PEC, 2 AES-PDC, SOF	11850	202.856	55	271.128	15.170	311.248
HEH[BRW]-2	2 AES-PEC, 2 AES-PDC, SPC	12002	203.894	55	269.748	**15.184**	308.879
HEH[BRW]-3	1 AES-PEC, 1 AES-PDC, SOF	8012	218.384	69	315.957	12.964	395.020
HMCH[Poly]	1 AES-PEC, 1 AES-SDC	**5345**	225.485	94	416.879	9.825	448.789
HEH[Poly]	1 AES-PEC, 1 AES-PDC	6962	218.198	83	380.388	10.768	377.606

Agenda

Cinvestav
Centro de Investigación y de Estudios Avanzados del Instituto Politécnico Nacional
50 AÑOS

$\mathbb{F}_p$ Arithmetic

$\mathbb{F}_p$ field arithmetic has crucial importance for the performance of any cryptosystem. The field elements $a, b \in \mathbb{F}_p$ are integers in the interval $[0, p-1]$

Addition	$a + b \bmod p$
Multiplication	$a \cdot b \bmod p$
Multiplicative inversion	$a^{-1} \bmod p$

Field Addition

We would like to compute the sum of two k-bit integers A and B. Let A_i and B_i for $i = 1, 2, \ldots, k-1$ represent the bits of the integers A and B, respectively, then the sum bits S_i for $i = 1, 2, \ldots, k-1$ and the final carry-out C_k are defined as,

$$\begin{array}{cccccc} & A_{k-1} & A_{k-2} & \cdots & A_1 & A_0 \\ + & B_{k-1} & B_{k-2} & \cdots & B_1 & B_0 \\ \hline C_k & S_{k-1} & S_{k-2} & \cdots & S_1 & S_0 \end{array}$$

Full adders

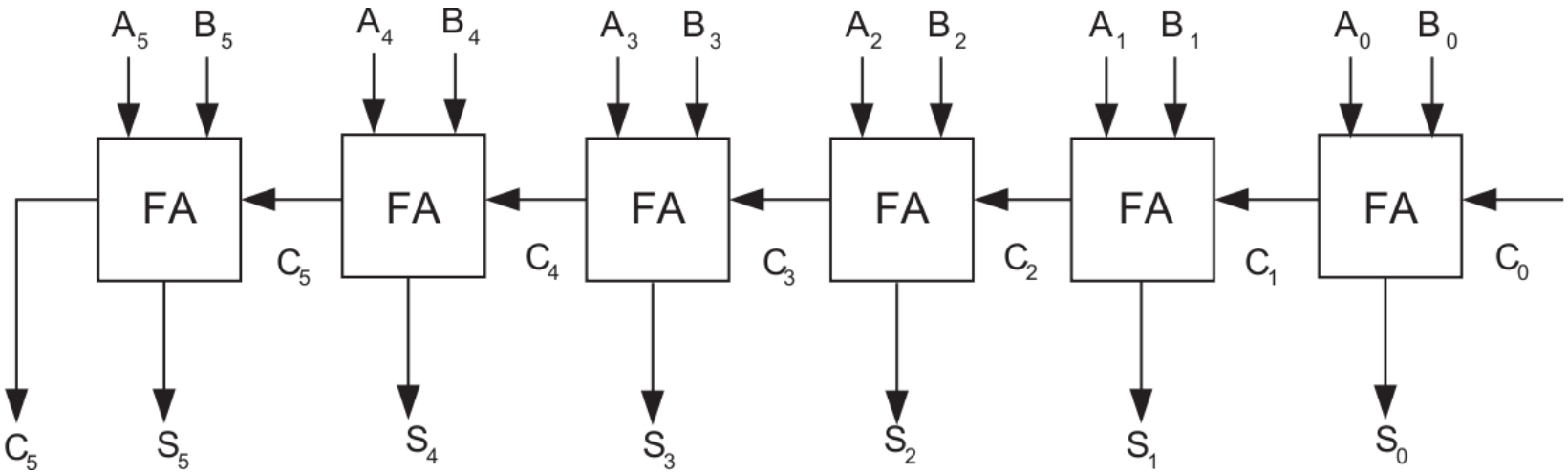
The truth table of a full adders cell is as follows:

A_i	B_i	C_i	C_{i+1}	S_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

The boolean functions of the output values are as

$$\begin{aligned} C_{i+1} &= A_i B_i + A_i C_i + B_i C_i \\ S_i &= A_i \oplus B_i \oplus C_i \end{aligned}$$

Carry propagate adder



Carry Save adder

Its main function is to add three k-bit integers A, B, and C to produce two integers C' and S such that

$$C' + S = A + B + C$$

Carry Save adder

As an example, let $A = 40$, $B = 25$, and $C = 20$, we compute S and C' as shown below:

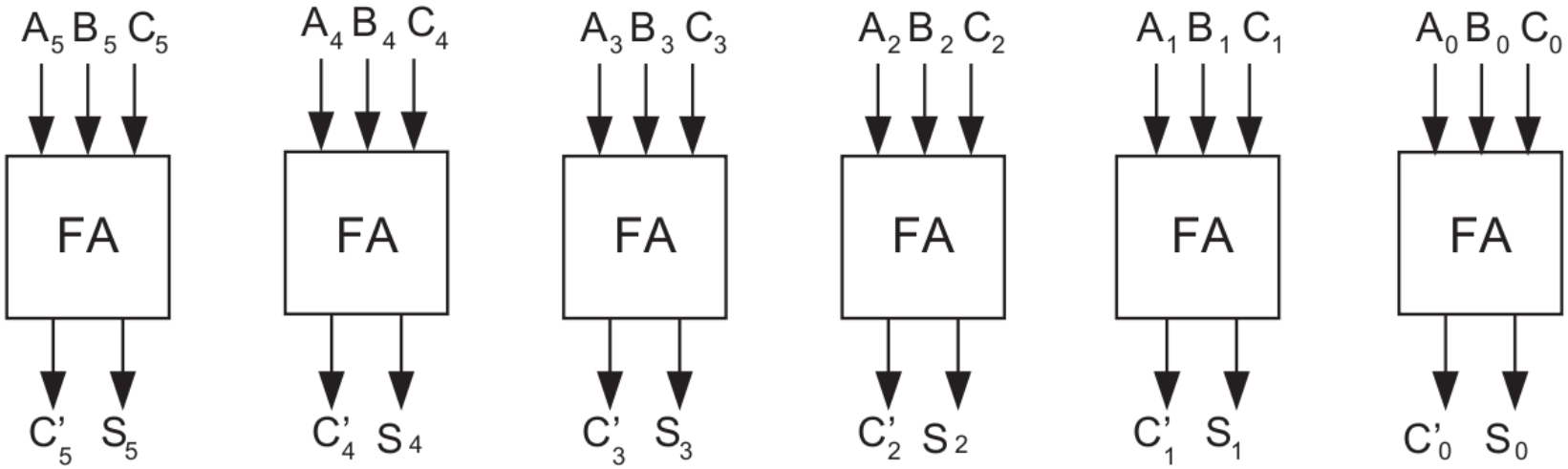
$A = 40$	$=$		1	0	1	0	0	0
$B = 25$	$=$		0	1	1	0	0	1
$C = 20$	$=$		0	1	0	1	0	0
$S = 37$	$=$		1	0	0	1	0	1
$C' = 48$	$=$	0	1	1	0	0	0	

Carry Save adder

The ith bit of the sum S_i and the $(i+1)$st bit of the carry C'_{i+1} is calculated using the equations

$$\begin{aligned} S_i &= A_i \oplus B_i \oplus C_i \\ C'_{i+1} &= A_i B_i + A_i C_i + B_i C_i \end{aligned}$$

Carry Save adder



Field multiplication

We would like to compute $C = A \cdot B \bmod p$, with $A, B, C \in \mathbb{F}_p$ There are basically four approaches for computing the product,

- Multiply and then divide [by possibly using fast reduction]
- The steps of the multiplication and reduction are interleaved.
- Brickell's method.
- Montgomery's method.

Field multiplication

We would like to compute $C = A \cdot B \bmod p$, with $A, B, C \in \mathbb{F}_p$ The multiply-and-divide method first multiplies A and B to obtain the $2k$-bit number

$$M' := AB$$

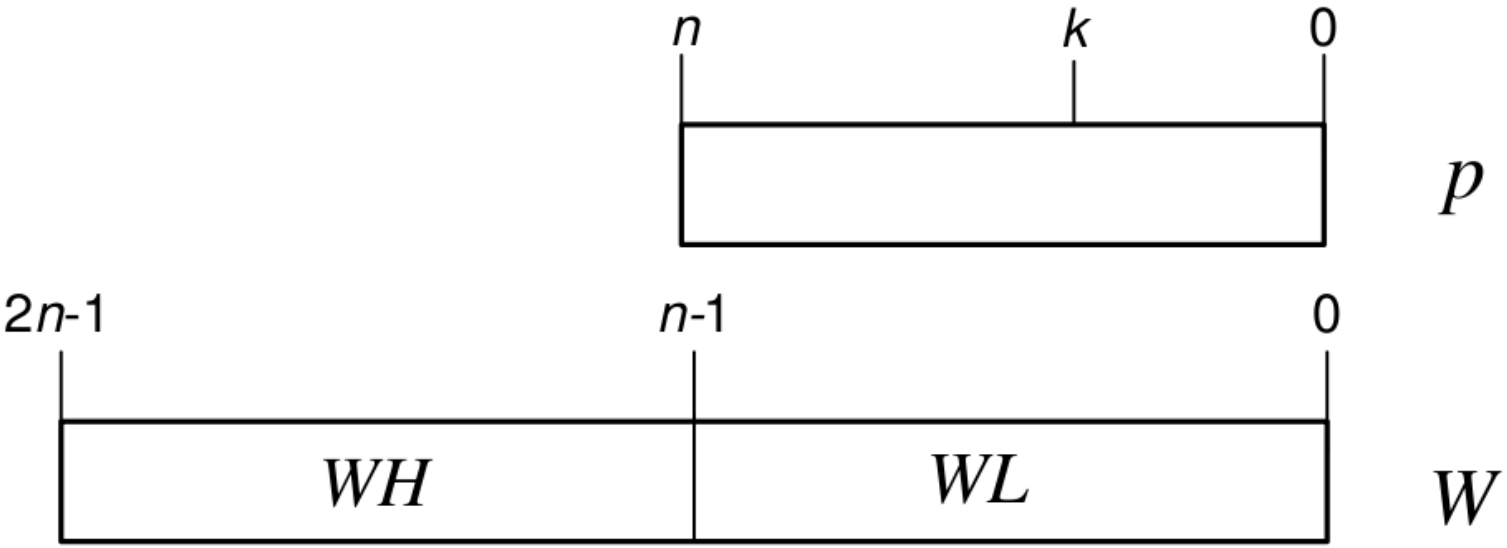
Then, the result C' is divided (reduced) by p to obtain the k-bit number

$$C := C' \mod p$$

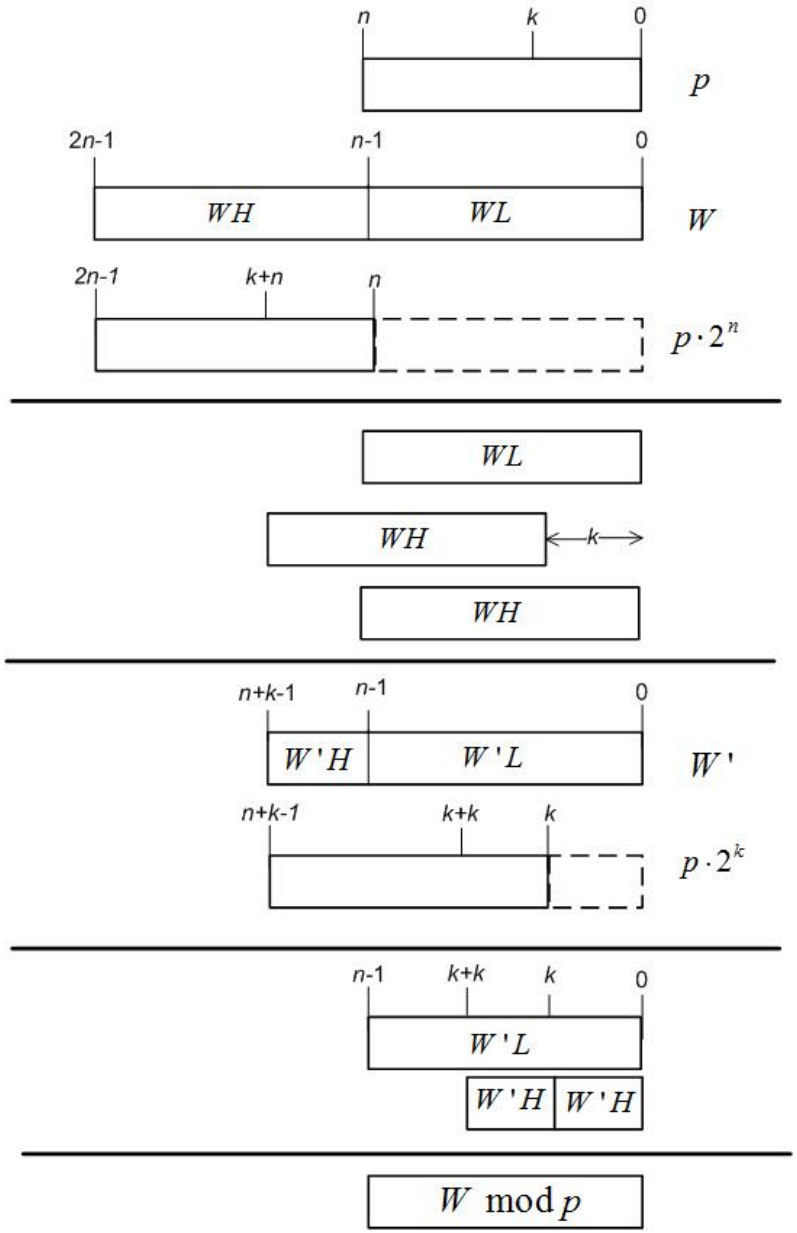
The result C is a k-bit or s-word number.

Field multiplication: fast reduction

$$p = 2^n \pm 2^k \pm 1$$



Field multiplication: fast reduction



Montgomery Multiplier

The problem of performing a division by p is traded with divisions by r, where $r = 2^k$ with $k - 1 < |p| < k$.

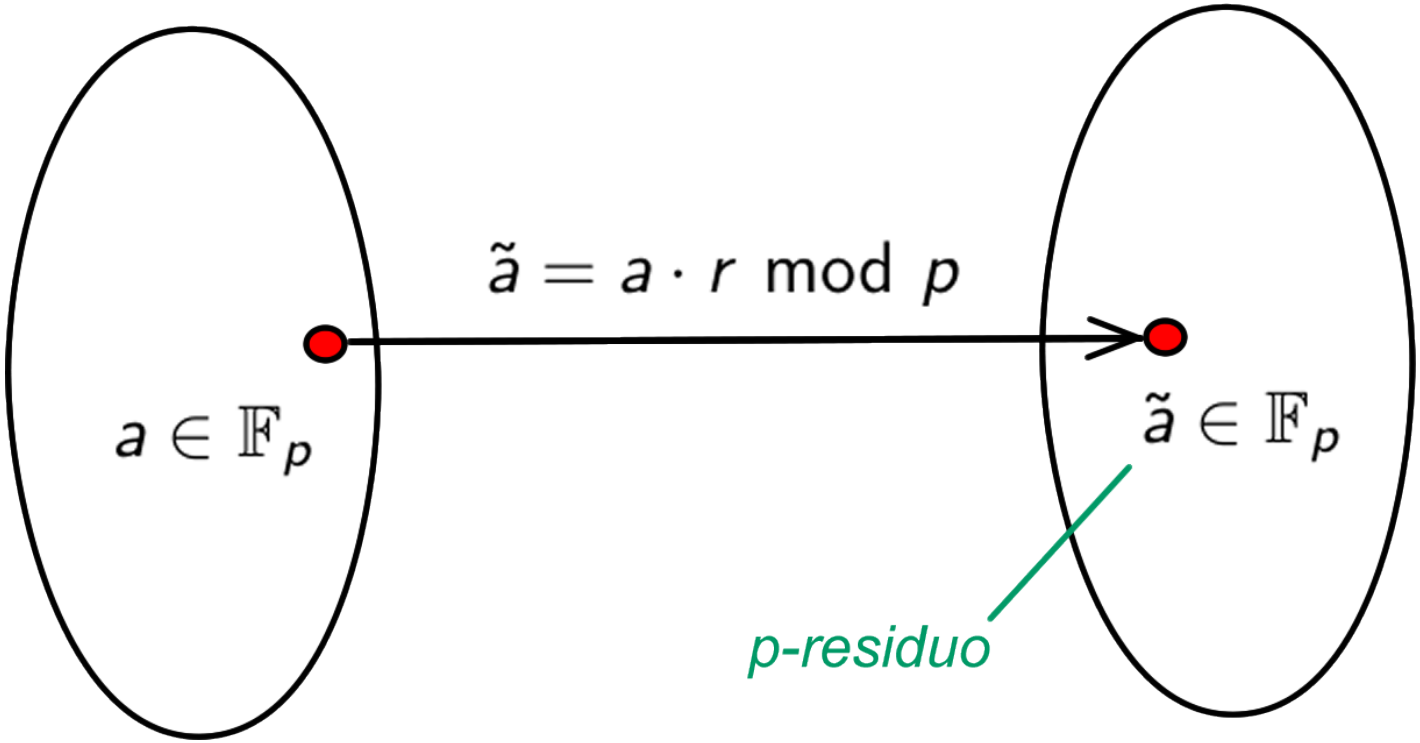


Figure: Montgomery p-Residues

Montgomery Multiplier

The montgomery product is defined as,

$$\text{MontPr}(\tilde{a}, \tilde{b}) = \tilde{a} \cdot \tilde{b} \cdot r^{-1} \bmod p$$

Given its p-residue $\tilde{a}$, one can compute a by performing,

$$\text{MontPr}(\tilde{a}, 1) = \tilde{a} \cdot 1 \cdot r^{-1} \bmod p = a \bmod p$$

Where p' can be obtained from Bezout's identity as,

$$r \cdot r^{-1} - p \cdot p' = 1,$$

provided that $\gcd(r, p) = 1$.

Montgomery Multiplier

Input: Prime p, p', $r = 2^k$ y $\tilde{a}$, $\tilde{b} \in \mathbb{F}_p$
Output: MontPr$(\tilde{a}, \tilde{b})$

1. $t \leftarrow \tilde{a} \cdot \tilde{b}$
2. $m \leftarrow t \cdot p' \bmod r$
3. $u \leftarrow (t + m \cdot p)/r$
4. **if** $u > p$ **then**
5. **return** $u - p$
6. **else**
7. **return** u
8. **return** u

Montgomery Multiplier

Input: Prime p, p', $r = 2^k$ y $\tilde{a}$, $\tilde{b} \in \mathbb{F}_p$
Output: MontPr$(\tilde{a}, \tilde{b})$

1. $t \leftarrow \tilde{a} \cdot \tilde{b}$
2. $m \leftarrow t \cdot p' \bmod r$ $\quad m \equiv -t \cdot p^{-1} \bmod r$
3. $u \leftarrow (t + m \cdot p)/r$ $\quad (t + m \cdot p) \equiv 0 \bmod r$
4. **if** $u > p$ **then**
5. $\quad$ **return** $u - p$
6. **else**
7. $\quad$ **return** u
8. **return** u

Montgomery multiplier

Input: Prime p, p', $r = 2^k$ y $\tilde{a}$, $\tilde{b} \in \mathbb{F}_p$
Output: MontPr$(\tilde{a}, \tilde{b})$

1. $t \leftarrow \tilde{a} \cdot \tilde{b}$
2. $m \leftarrow t \cdot p' \bmod r$
3. $u \leftarrow (t + m \cdot p)/r$ $\qquad t + m \cdot p \equiv \tilde{a} \cdot \tilde{b} \bmod p$
4. **if** $u > p$ **then**
5. **return** $u - p$
6. **else**
7. **return** u
8. **return** u

Montgomery multiplier variants: the SOS Separated Operand Scanning method

Computes first the product $t = a \cdot b$ and then u.

Input: $a = (a_0, a_1, ..., a_{n-1})$ and
$b = (b_0, b_1, ..., b_{n-1})$
Output: $t = a \cdot b$ with $t = (t_0, t_1, ..., t_{2n-1})$

1. $t \leftarrow 0$
2. **for** $i = 0 \rightarrow n - 1$ **do**
3. $\quad C \leftarrow 0$
4. $\quad$ **for** $j = 0 \rightarrow n - 1$ **do**
5. $\quad\quad (C, S) \leftarrow t_{i+j} + a_j \cdot b_i + C$
6. $\quad\quad t_{i+j} = S$
7. $\quad t_{i+n} = C$
8. $\quad$ **return** t

				a_3	a_2	a_1	a_0
				b_3	b_2	b_1	b_0
				t_{03}	t_{02}	t_{01}	t_{00}
			t_{13}	t_{12}	t_{11}	t_{10}	
		t_{23}	t_{22}	t_{21}	t_{20}		
	t_{33}	t_{32}	t_{31}	t_{30}			
t_7	t_6	t_5	t_4	t_3	t_2	t_1	t_0

The complexity of this algorithm is $\mathcal{O}(n^2)$

Montgomery multiplier variants: the SOS Separated Operand Scanning method

Input: $t = (t_0, t_1, ..., t_{2n-1})$, $p = (p_0, p_1, ..., p_n)$ and p'_0, where $|p'_0| = \omega$
Output: $u \leftarrow (t + (t \cdot p' \bmod r) \cdot p)/r$

1. **for** $i = 0 \rightarrow n-1$ **do**
2. $\quad C \leftarrow 0$
3. $\quad m \leftarrow t_i \cdot p'_0 \bmod 2^{\omega}$
4. $\quad$ **for** $j = 0 \rightarrow n-1$ **do**
5. $\quad\quad (C, S) \leftarrow t_{i+j} + m \cdot p_j + C$
6. $\quad\quad t_{i+j} = S$
7. $\quad \mathrm{ADD}(t_{i+n}, C)$
8. **for** $i = 0 \rightarrow n-1$ **do**
9. $\quad u_i = t_{i+n}$
10. **return** u

Montgomery multiplier variants: the SOS Separated Operand Scanning method

Input: $t = (t_0, t_1, ..., t_{2n-1})$, $p = (p_0, p_1, ..., p_n)$ and p'_0, where $|p'_0| = \omega$
Output: $u \leftarrow (t + (t \cdot p' \bmod r) \cdot p)/r$

1. **for** $i = 0 \rightarrow n-1$ **do**
2. $\quad C \leftarrow 0$
3. $\quad m \leftarrow t_i \cdot p'_0 \bmod 2^\omega$
4. $\quad$ **for** $j = 0 \rightarrow n-1$ **do**
5. $\quad\quad (C, S) \leftarrow t_{i+j} + m \cdot p_j + C$
6. $\quad\quad t_{i+j} = S$
7. $\quad \mathrm{ADD}(t_{i+n}, C)$
8. **for** $i = 0 \rightarrow n-1$ **do**
9. $\quad u_i = t_{i+n}$
10. **return** u

The number of products of this method is $2n^2 + n$.

Agenda

1. Context and motivation
2. Hardware design issues
3. basic $\mathbb{F}_{2^m}$ field arithmetic building blocks
   - $\mathbb{F}_{2^m}$ Field Arithmetic
4. basic cryptographic building blocks
   - block ciphers
   - polynomial hash
5. BRW polynomials
6. A 256-bit multiplier over $\mathbb{F}_p$
   - $\mathbb{F}_p$ Field Arithmetic
   - pairings and elliptic curves
   - Karatsuba-like multiplier

Cinvestav
Centro de Investigación y de Estudios Avanzados del Instituto Politécnico Nacional
50 AÑOS

Elliptic curves

- E defined by a Weierstraß equation of the form

$$y^2 = x^3 + Ax + B$$

Elliptic curves

- E defined by a Weierstraß equation of the form

$$y^2 = x^3 + Ax + B$$

- $E(K)$ set of rational points over a field K

Elliptic curves

- E defined by a Weierstraß equation of the form

$$y^2 = x^3 + Ax + B$$

- $E(K)$ set of rational points over a field K
- Additive group law over $E(K)$

Elliptic curves

- E defined by a Weierstraß equation of the form

$$y^2 = x^3 + Ax + B$$

- $E(K)$ set of rational points over a field K
- Additive group law over $E(K)$
- Many applications in cryptography since 1985
  - EC-based Diffie-Hellman key exchange
  - EC-based Digital Signature Algorithm
  - ...

Elliptic curves

- E defined by a Weierstraß equation of the form

$$y^2 = x^3 + Ax + B$$

- $E(K)$ set of rational points over a field K
- Additive group law over $E(K)$
- Many applications in cryptography since 1985
  - EC-based Diffie-Hellman key exchange
  - EC-based Digital Signature Algorithm
  - ...
- Interest: smaller keys than usual cryptosystems (RSA, ElGamal, ...)

Elliptic curves

- E defined by a Weierstraß equation of the form

$$y^2 = x^3 + Ax + B$$

- $E(K)$ set of rational points over a field K
- Additive group law over $E(K)$
- Many applications in cryptography since 1985
  - EC-based Diffie-Hellman key exchange
  - EC-based Digital Signature Algorithm
  - ...
- Interest: smaller keys than usual cryptosystems (RSA, ElGamal, ...)
- Moreover, elliptic curves can be used to construct secure bilinear pairings

Group cryptography

- Let $(\mathbb{G}_1, +)$ be an additively-written cyclic groups of prime order $\#\mathbb{G}_1 = \ell$

Group cryptography

- Let $(\mathbb{G}_1, +)$ be an additively-written cyclic groups of prime order $\#\mathbb{G}_1 = \ell$
- P, a generator of the group: $\mathbb{G}_1 = \langle P \rangle$

Group cryptography

- Let $(\mathbb{G}_1, +)$ be an additively-written cyclic groups of prime order $\#\mathbb{G}_1 = \ell$
- P, a generator of the group: $\mathbb{G}_1 = \langle P \rangle$
- Scalar multiplication: for any integer k, we have $kP = \underbrace{P + P + \cdots + P}_{k \text{ times}}$

Group cryptography

- Let $(\mathbb{G}_1, +)$ be an additively-written cyclic groups of prime order $\#\mathbb{G}_1 = \ell$
- P, a generator of the group: $\mathbb{G}_1 = \langle P \rangle$
- Scalar multiplication: for any integer k, we have $kP = \underbrace{P + P + \cdots + P}_{k \text{ times}}$
- Discrete logarithm: given $Q \in \mathbb{G}_1$, compute k such that $Q = kP$

Group cryptography

- Let $(\mathbb{G}_1, +)$ be an additively-written cyclic groups of prime order $\#\mathbb{G}_1 = \ell$
- P, a generator of the group: $\mathbb{G}_1 = \langle P \rangle$
- Scalar multiplication: for any integer k, we have $kP = \underbrace{P + P + \cdots + P}_{k \text{ times}}$
- Discrete logarithm: given $Q \in \mathbb{G}_1$, compute k such that $Q = kP$
- We assume that the discrete logarithm problem (DLP) in $\mathbb{G}_1$ is hard

Bilinear pairings

- Let $(\mathbb{G}_1, +)$, $(\mathbb{G}_2, +)$ be two additively-written cyclic groups of prime order $\#\mathbb{G}_1 = \#\mathbb{G}_2 = \ell$
- $(\mathbb{G}_\tau, \times)$, a multiplicatively-written cyclic group of order $\#\mathbb{G}_\tau = \ell$

Bilinear pairings

- Let $(\mathbb{G}_1, +)$, $(\mathbb{G}_2, +)$ be two additively-written cyclic groups of prime order $\#\mathbb{G}_1 = \#\mathbb{G}_2 = \ell$
- $(\mathbb{G}_\tau, \times)$, a multiplicatively-written cyclic group of order $\#\mathbb{G}_\tau = \ell$
- A non-degenerate bilinear pairing is a map

$$\hat{e} : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_\tau$$

that satisfies the following conditions:

Bilinear pairings

- Let $(\mathbb{G}_1, +)$, $(\mathbb{G}_2, +)$ be two additively-written cyclic groups of prime order $\#\mathbb{G}_1 = \#\mathbb{G}_2 = \ell$
- $(\mathbb{G}_\tau, \times)$, a multiplicatively-written cyclic group of order $\#\mathbb{G}_\tau = \ell$
- A non-degenerate bilinear pairing is a map

$$\hat{e} : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_\tau$$

that satisfies the following conditions:

  - non-degeneracy: $\hat{e}(P, P) \neq 1_{\mathbb{G}_\tau}$ (equivalently $\hat{e}(P, P)$ generates $\mathbb{G}_\tau$)

Bilinear pairings

- Let $(\mathbb{G}_1, +)$, $(\mathbb{G}_2, +)$ be two additively-written cyclic groups of prime order $\#\mathbb{G}_1 = \#\mathbb{G}_2 = \ell$
- $(\mathbb{G}_\tau, \times)$, a multiplicatively-written cyclic group of order $\#\mathbb{G}_\tau = \ell$
- A non-degenerate bilinear pairing is a map

$$\hat{e} : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_\tau$$

that satisfies the following conditions:

  - non-degeneracy: $\hat{e}(P, P) \neq 1_{\mathbb{G}_\tau}$ (equivalently $\hat{e}(P, P)$ generates $\mathbb{G}_\tau$)
  - bilinearity:
    $\hat{e}(Q_1+Q_2, R) = \hat{e}(Q_1, R)\cdot\hat{e}(Q_2, R) \quad \hat{e}(Q, R_1+R_2) = \hat{e}(Q, R_1)\cdot\hat{e}(Q, R_2)$

Bilinear pairings

- Let $(\mathbb{G}_1, +)$, $(\mathbb{G}_2, +)$ be two additively-written cyclic groups of prime order $\#\mathbb{G}_1 = \#\mathbb{G}_2 = \ell$
- $(\mathbb{G}_\tau, \times)$, a multiplicatively-written cyclic group of order $\#\mathbb{G}_\tau = \ell$
- A non-degenerate bilinear pairing is a map

$$\hat{e} : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_\tau$$

that satisfies the following conditions:

  - non-degeneracy: $\hat{e}(P, P) \neq 1_{\mathbb{G}_\tau}$ (equivalently $\hat{e}(P, P)$ generates $\mathbb{G}_\tau$)
  - bilinearity:
    $\hat{e}(Q_1+Q_2, R) = \hat{e}(Q_1, R)\cdot\hat{e}(Q_2, R) \quad \hat{e}(Q, R_1+R_2) = \hat{e}(Q, R_1)\cdot\hat{e}(Q, R_2)$
  - computability: $\hat{e}$ can be efficiently computed

Bilinear pairings

- Let $(\mathbb{G}_1, +)$, $(\mathbb{G}_2, +)$ be two additively-written cyclic groups of prime order $\#\mathbb{G}_1 = \#\mathbb{G}_2 = \ell$
- $(\mathbb{G}_\tau, \times)$, a multiplicatively-written cyclic group of order $\#\mathbb{G}_\tau = \ell$
- A non-degenerate bilinear pairing is a map

$$\hat{e} : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_\tau$$

that satisfies the following conditions:

  - non-degeneracy: $\hat{e}(P, P) \neq 1_{\mathbb{G}_\tau}$ (equivalently $\hat{e}(P, P)$ generates $\mathbb{G}_\tau$)
  - bilinearity:
    $\hat{e}(Q_1+Q_2, R) = \hat{e}(Q_1, R)\cdot\hat{e}(Q_2, R) \quad \hat{e}(Q, R_1+R_2) = \hat{e}(Q, R_1)\cdot\hat{e}(Q, R_2)$
  - computability: $\hat{e}$ can be efficiently computed

- Immediate property: for any two integers k_1 and k_2

$$\hat{e}(k_1 Q, k_2 R) = \hat{e}(Q, R)^{k_1 k_2}$$

Pairings in cryptography

- At first, used to attack supersingular elliptic curves
  - Menezes-Okamoto-Vanstone and Frey-Rück attacks, 1993 and 1994

$$\begin{array}{ccc} \mathrm{DLP}_{\mathbb{G}_1} & <_{\mathrm{P}} & \mathrm{DLP}_{\mathbb{G}_\tau} \\ kP & \longrightarrow & \hat{e}(kP, P) = \hat{e}(P, P)^k \end{array}$$

  - for cryptographic applications, we will also require the DLP in $\mathbb{G}_\tau$ to be hard

Pairings in cryptography

- At first, used to attack supersingular elliptic curves
  - Menezes-Okamoto-Vanstone and Frey-Rück attacks, 1993 and 1994

$$\text{DLP}_{\mathbb{G}_1} \quad <_{\text{P}} \quad \text{DLP}_{\mathbb{G}_\tau}$$

$$kP \quad \longrightarrow \quad \hat{e}(kP, P) = \hat{e}(P, P)^k$$

  - for cryptographic applications, we will also require the DLP in $\mathbb{G}_\tau$ to be hard
- One-round three-party key agreement (Joux, 2000)

Pairings in cryptography

- At first, used to attack supersingular elliptic curves
  - Menezes-Okamoto-Vanstone and Frey-Rück attacks, 1993 and 1994

$$\begin{array}{ccc} \mathrm{DLP}_{\mathbb{G}_1} & <_{\mathrm{P}} & \mathrm{DLP}_{\mathbb{G}_\tau} \\ kP & \longrightarrow & \hat{e}(kP, P) = \hat{e}(P, P)^k \end{array}$$

  - for cryptographic applications, we will also require the DLP in $\mathbb{G}_\tau$ to be hard
- One-round three-party key agreement (Joux, 2000)
- Identity-based encryption
  - Boneh–Franklin, 2001
  - Sakai–Kasahara, 2001

Pairings in cryptography

- At first, used to attack supersingular elliptic curves
  - Menezes-Okamoto-Vanstone and Frey-Rück attacks, 1993 and 1994

$$\begin{array}{ccc} \mathrm{DLP}_{\mathbb{G}_1} & <_{\mathrm{P}} & \mathrm{DLP}_{\mathbb{G}_\tau} \\ kP & \longrightarrow & \hat{e}(kP, P) = \hat{e}(P, P)^k \end{array}$$

  - for cryptographic applications, we will also require the DLP in $\mathbb{G}_\tau$ to be hard
- One-round three-party key agreement (Joux, 2000)
- Identity-based encryption
  - Boneh–Franklin, 2001
  - Sakai–Kasahara, 2001
- Short digital signatures
  - Boneh–Lynn–Shacham, 2001
  - Zang–Safavi-Naini–Susilo, 2004

Pairings in cryptography

- At first, used to attack supersingular elliptic curves
  - Menezes-Okamoto-Vanstone and Frey-Rück attacks, 1993 and 1994

$$\begin{array}{ccc} \mathrm{DLP}_{\mathbb{G}_1} & <_{\mathrm{P}} & \mathrm{DLP}_{\mathbb{G}_\tau} \\ kP & \longrightarrow & \hat{e}(kP, P) = \hat{e}(P, P)^k \end{array}$$

  - for cryptographic applications, we will also require the DLP in $\mathbb{G}_\tau$ to be hard
- One-round three-party key agreement (Joux, 2000)
- Identity-based encryption
  - Boneh–Franklin, 2001
  - Sakai–Kasahara, 2001
- Short digital signatures
  - Boneh–Lynn–Shacham, 2001
  - Zang–Safavi-Naini–Susilo, 2004
- Aggregate signatures, etc.

Barreto–Naehrig elliptic curves

- Defined by the equation $E : y^2 = x^3 + b$, where $b \neq 0$. Their embedding degree k is equal to 12.

Barreto–Naehrig elliptic curves

- Defined by the equation $E : y^2 = x^3 + b$, where $b \neq 0$. Their embedding degree k is equal to 12.
- The characteristic p of the prime field, the group order r, and the trace of Frobenius t_r of the curve are parametrized as,

$$\begin{aligned} p(t) &= 36t^4 + 36t^3 + 24t^2 + 6t + 1, \\ r(t) &= 36t^4 + 36t^3 + 18t^2 + 6t + 1, \\ t_r(t) &= 6t^2 + 1, \end{aligned}$$

where $t \in \mathbb{Z}$ is an arbitrary integer such that $p = p(t)$ and $r = r(t)$ are both prime numbers.

Barreto–Naehrig elliptic curves

- Defined by the equation $E : y^2 = x^3 + b$, where $b \neq 0$. Their embedding degree k is equal to 12.
- The characteristic p of the prime field, the group order r, and the trace of Frobenius t_r of the curve are parametrized as,

$$\begin{aligned} p(t) &= 36t^4 + 36t^3 + 24t^2 + 6t + 1, \\ r(t) &= 36t^4 + 36t^3 + 18t^2 + 6t + 1, \\ t_r(t) &= 6t^2 + 1, \end{aligned}$$

  where $t \in \mathbb{Z}$ is an arbitrary integer such that $p = p(t)$ and $r = r(t)$ are both prime numbers.
- For efficiency purposes, t should have a low Hamming weight. If $\log_2 t \approx 64$ then $\log_2 t \approx 256$ and the pairing achieves the 128-bit security level

Barreto–Naehrig Curves

Let $E[r]$ denote the r-torsion subgroup of E. We define,

- $\mathbb{G}_1 = E(\mathbb{F}_p)[r]$,
- $\mathbb{G}_2 \subseteq E(\mathbb{F}_{p^{12}})[r]$,
- $\mathbb{G}_\tau = \mu_r \subset \mathbb{F}^*_{p^{12}}$ (i.e. the group of r-th roots of unity).
- The optimal ate pairing on the BN curve E is given as,

$$a_{\text{opt}} : \mathbb{G}_2 \times \mathbb{G}_1 \longrightarrow \mathbb{G}_3$$

- In practice, pairing computations can be restricted to points P and Q' that belong to $E(\mathbb{F}_p)$ and $E'(\mathbb{F}_{p^2})$, respectively, where, $E'/\mathbb{F}_{p^2} : y^2 = x^3 + b/\xi$.

Field Multiplier, the crucial building block (1/3)

- More than $10,000$ and $5,000$ multiplications over $\mathbb{F}_p$ and $\mathbb{F}_{p^2}$, respectively are required for computing a pairing defined over BN curves

Field Multiplier, the crucial building block (1/3)

- More than $10,000$ and $5,000$ multiplications over $\mathbb{F}_p$ and $\mathbb{F}_{p^2}$, respectively are required for computing a pairing defined over BN curves
- BN curves enjoy several useful features for computing the Montgomery reduction, namely,
  - $gcd(t, p) = 1$
  - $p \equiv 1 \bmod t$, which implies, $p^{-1} \bmod t = 1$
  - the coefficients of the polynomial $p(t)$ $(36, 36, 24, 6, 1)$ are relatively small.

Field Multiplier, the crucial building block (2/3)

- Above features can be exploited better if we represent the operands $a, b \in \mathbb{F}_p$ as polynomials of the variable t, i.e.,

$$a(t) = \sum_{i=0}^{4} a_i t^i, b(t) = \sum_{i=0}^{4} b_i t^i, |a_i| = |b_i| = 64 \text{ bits}$$

such that the product c is now seen as,

$$c(t) = a(t)b(t) = \sum_{i=0}^{8} c_i t^i \bmod p(t)$$

Using the so-called polynomial version of the Montgomery product

Field Multiplier, the crucial building block (3/3)

- notice that we have traded a 256-bit integer multiplication by a 4-degree polynomial product, with 64-bit integer coefficients. Hence, for this approach, it is now necessary to build a 64-bit integer multiplier as a basic building block.

Field Multiplier, the crucial building block (3/3)

- notice that we have traded a 256-bit integer multiplication by a 4-degree polynomial product, with 64-bit integer coefficients. Hence, for this approach, it is now necessary to build a 64-bit integer multiplier as a basic building block.
- For performing the 4-degree polynomial product we propose to use a Karatsuba-like multiplier [Montgomery, TC'05]], using fourteen 64-bit products and forty 128-bit additions.

Field Multiplier, the crucial building block (3/3)

- notice that we have traded a 256-bit integer multiplication by a 4-degree polynomial product, with 64-bit integer coefficients. Hence, for this approach, it is now necessary to build a 64-bit integer multiplier as a basic building block.
- For performing the 4-degree polynomial product we propose to use a Karatsuba-like multiplier [Montgomery, TC'05]], using fourteen 64-bit products and forty 128-bit additions.
- The new challenge is to come out with a scheduling that allows that the 64-bit integer multiplier block stays always busy

Field Multiplier Product [using the polynomial version of the Montgomery mult]

Input: $a(t) \bmod p = a(t) = \sum_{i=0}^{4} a_i t^i$,
$b(t) \bmod p = b(t) = \sum_{i=0}^{4} b_i t^i$,
$p(t) = 36t^4 + 36t^3 + 24t^2 + 6t + 1,\ t = 2^n + s$
Output: $c(t) = a(t)b(t) \cdot t^{-1} \bmod p$
1. $c(t) = \text{5-term\_KaratsubaProduct}(a(t), b(t))$ (Polynomial Product)

Field Multiplier Product [using the polynomial version of the Montgomery mult]

Input: $a(t) \bmod p = a(t) = \sum_{i=0}^{4} a_i t^i$,
$b(t) \bmod p = b(t) = \sum_{i=0}^{4} b_i t^i$,
$p(t) = 36t^4 + 36t^3 + 24t^2 + 6t + 1,\ t = 2^n + s$
Output: $c(t) = a(t)b(t) \cdot t^{-1} \bmod p$

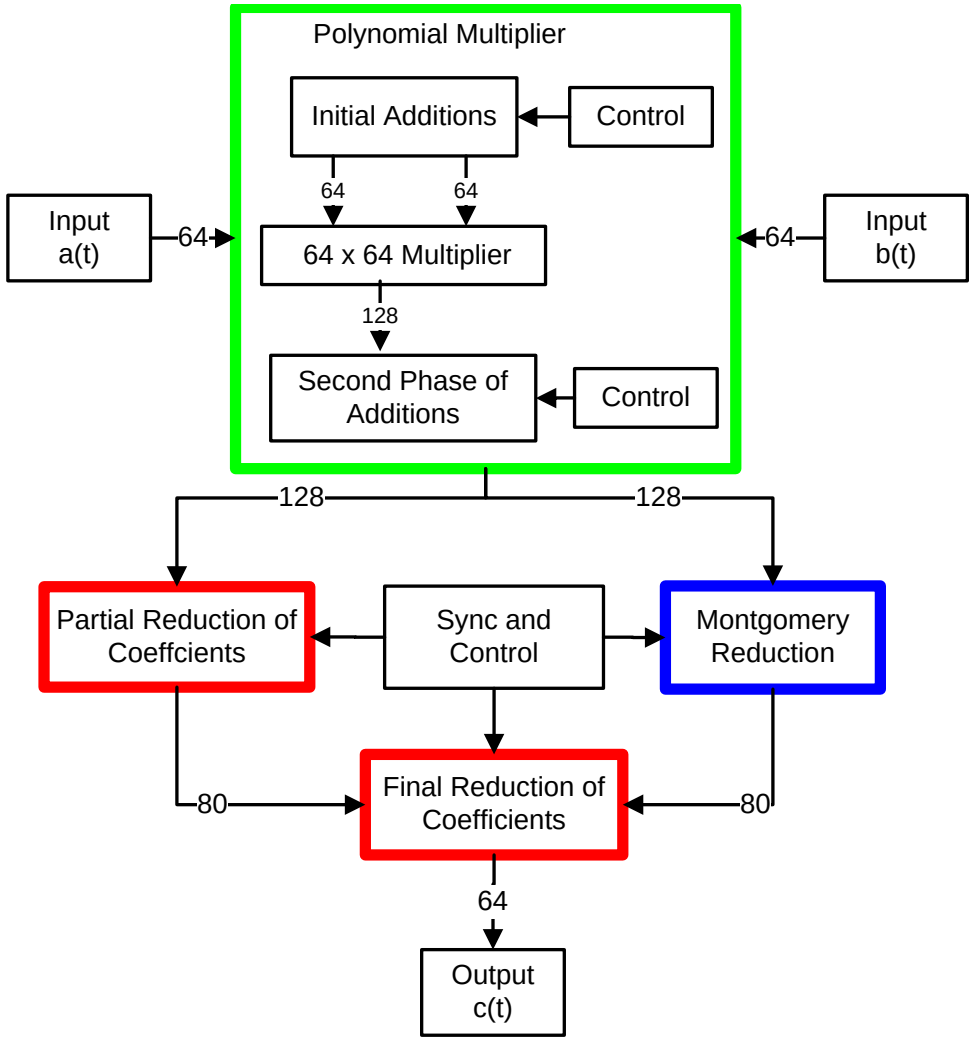
1. $c(t) = 5\text{-term\_KaratsubaProduct}(a(t), b(t))$ (Polynomial Product)
2. **for** $i = 0$ to 4 **do**
3. $\quad \mu \leftarrow c_0 \text{ div } 2^n;\ \gamma \leftarrow c_0 \bmod 2^n - \mu s$
4. $\quad g(t) \leftarrow p(t)(-\gamma)$ (Montgomery Reduction Phase)
5. $\quad c(t) \leftarrow (c(t) + g(t))/t + \mu$

Field Multiplier Product [using the polynomial version of the Montgomery mult]

Input: $a(t) \bmod p = a(t) = \sum_{i=0}^{4} a_i t^i$,
$b(t) \bmod p = b(t) = \sum_{i=0}^{4} b_i t^i$,
$p(t) = 36t^4 + 36t^3 + 24t^2 + 6t + 1,\ t = 2^n + s$
Output: $c(t) = a(t)b(t) \cdot t^{-1} \bmod p$

1. $c(t) = 5\text{-term\_KaratsubaProduct}(a(t), b(t))$ (Polynomial Product)
2. **for** $i = 0$ to 4 **do**
3. $\quad \mu \leftarrow c_0 \text{ div } 2^n;\ \gamma \leftarrow c_0 \bmod 2^n - \mu s$
4. $\quad g(t) \leftarrow p(t)(-\gamma)$ (Montgomery Reduction Phase)
5. $\quad c(t) \leftarrow (c(t) + g(t))/t + \mu$
6. **for** $k = 0$ to 1 **do**
7. $\quad$ **for** $i = 0$ to 3 **do**
8. $\quad\quad \mu \leftarrow c_i \text{ div } 2^n;\ \gamma \leftarrow c_i \bmod 2^n - \mu s$ (Coefficient Reduction Phase)
9. $\quad\quad c_{i+1} \leftarrow c_{i+1} + \mu;\ c_i \leftarrow \gamma$
10. **return** $c(t)$

proposed architecture for the field multiplication

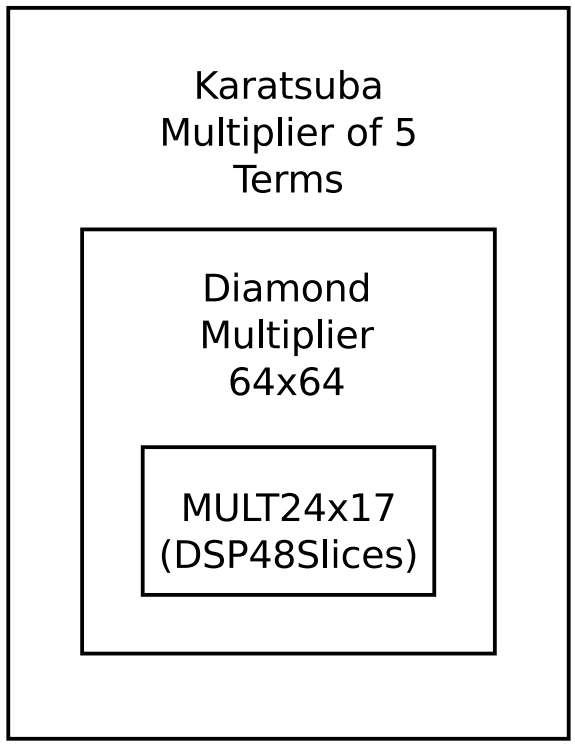


Agenda

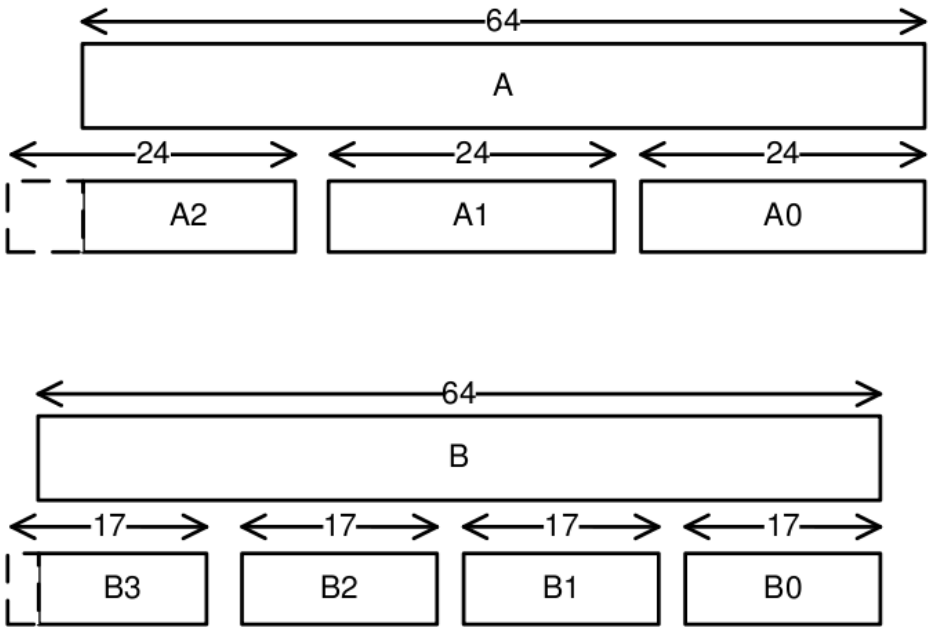
Cinvestav
Centro de Investigación y de Estudios Avanzados del Instituto Politécnico Nacional
50 AÑOS

Hierarchy of a 64-bit integer multiplier

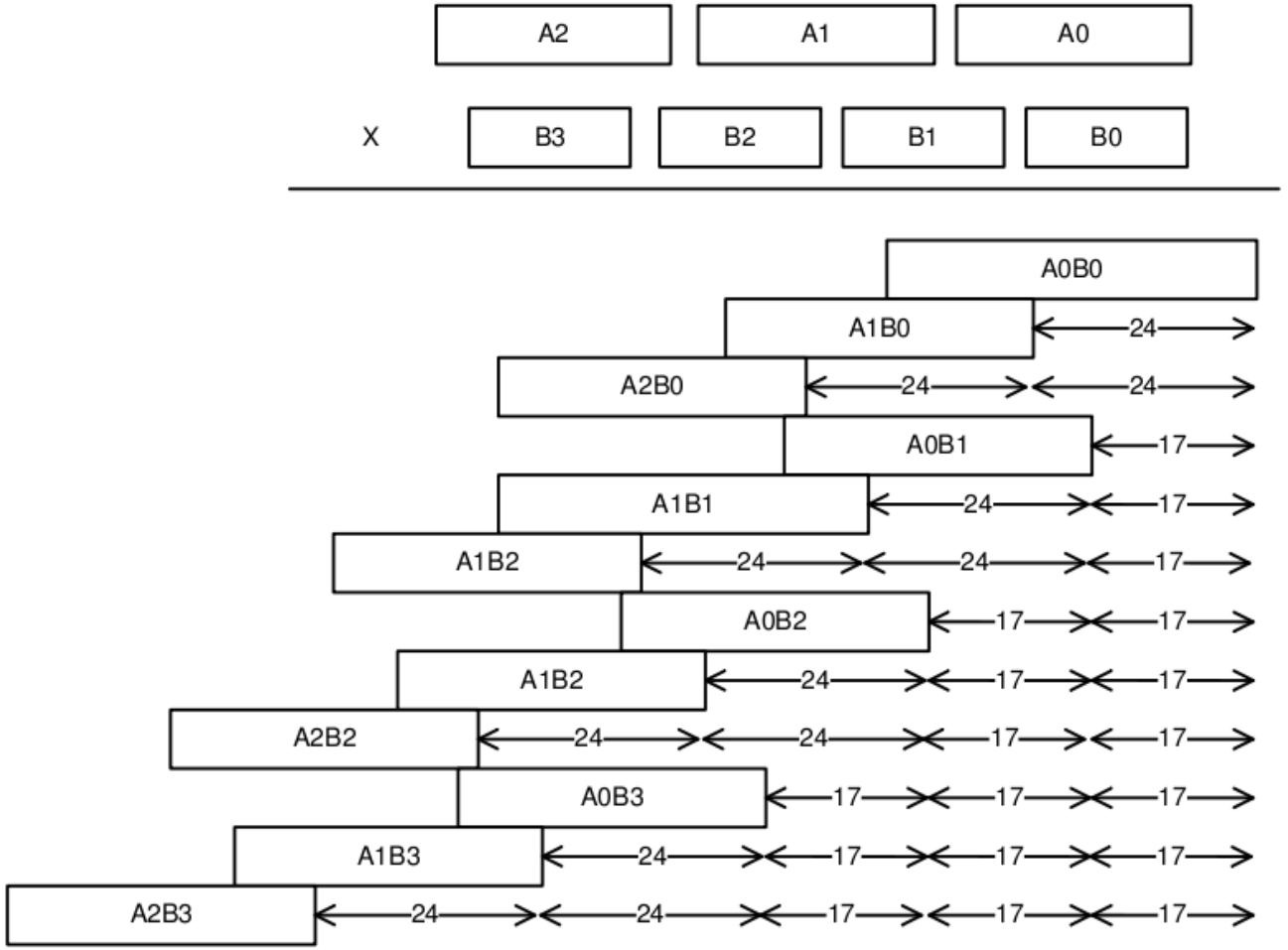
We took the design decision of using the embedded 24×17 multiplier available in the Xilinx DSP48Slices.



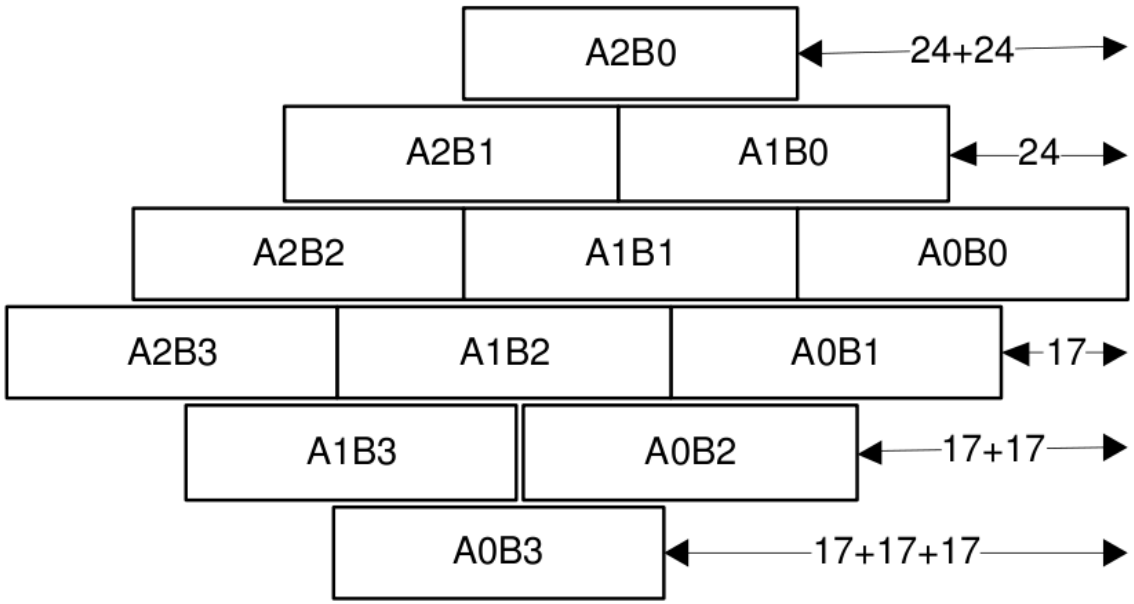
Operand representation using the Xilinx DSP48Slices



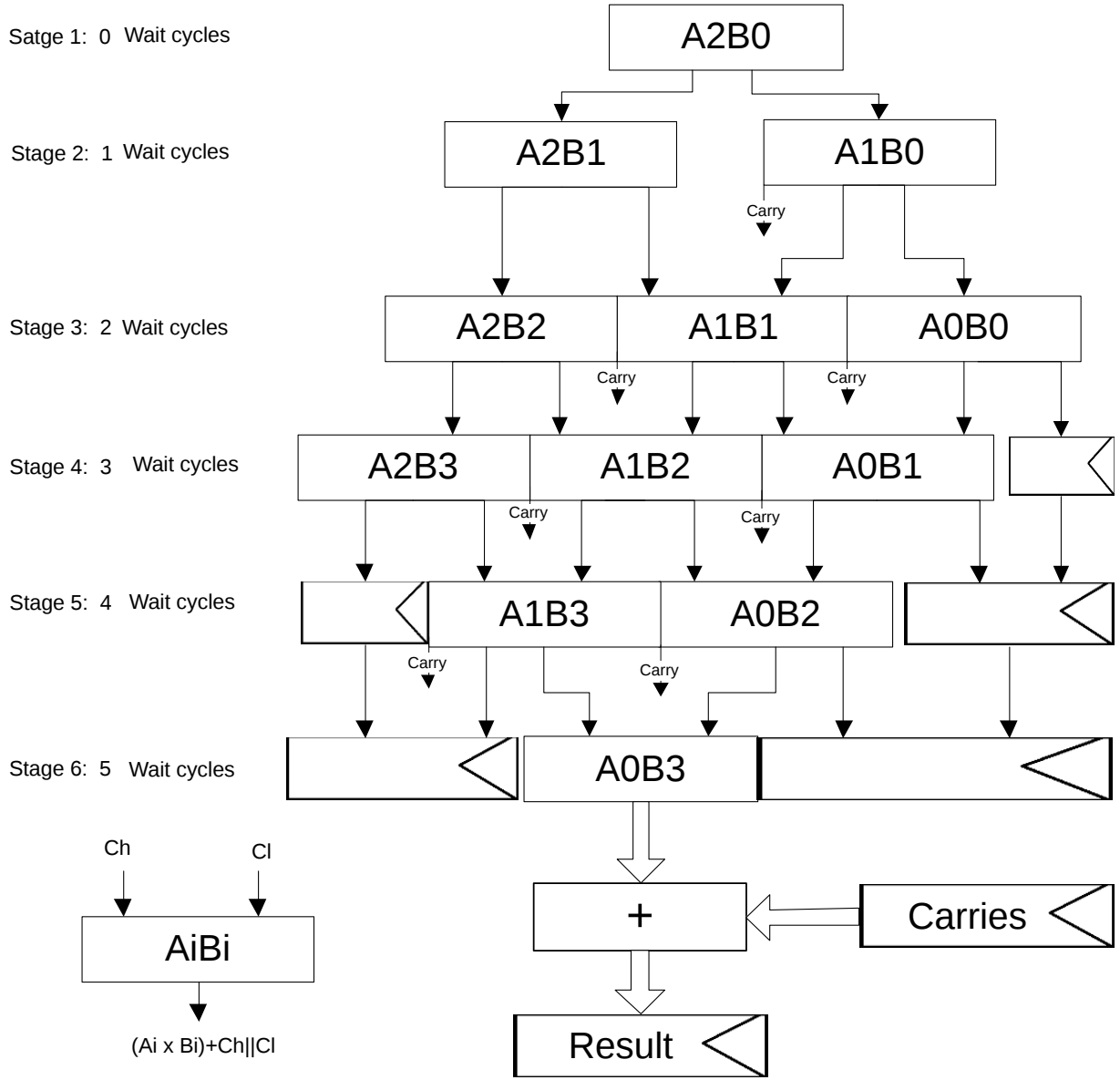
School book multiplication method



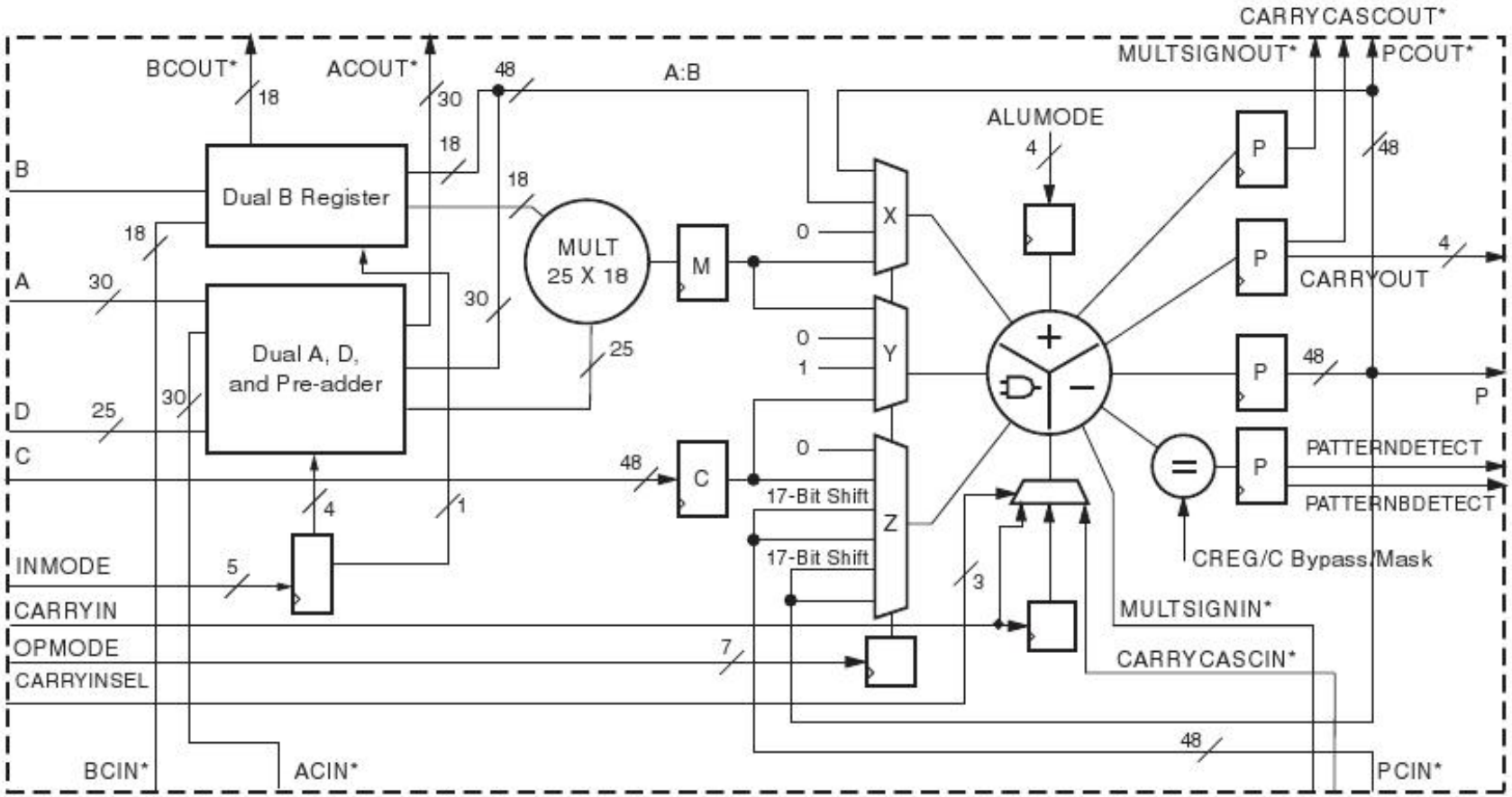
diamond-shape multiplication method



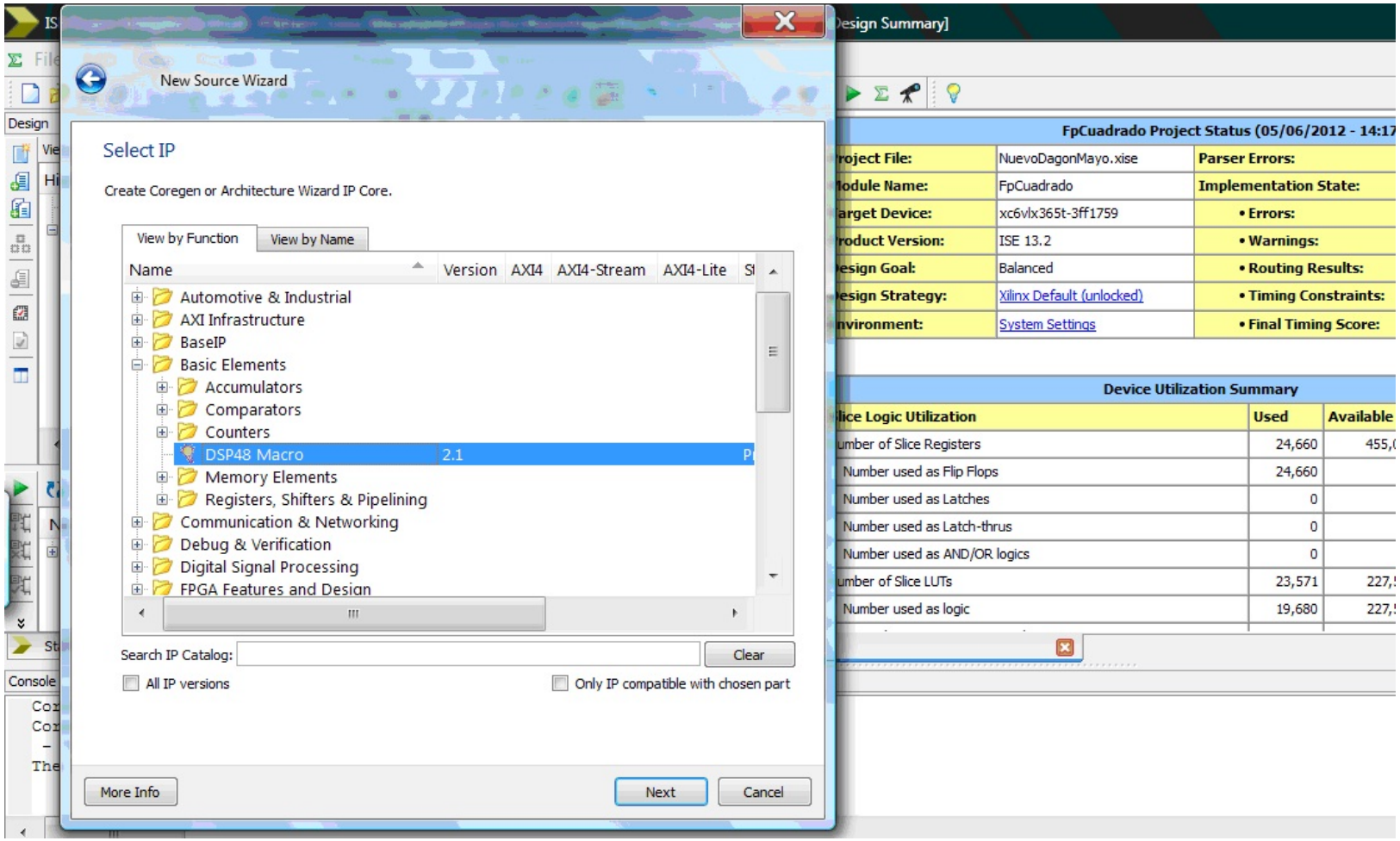
64-bit integer multiplication architecture



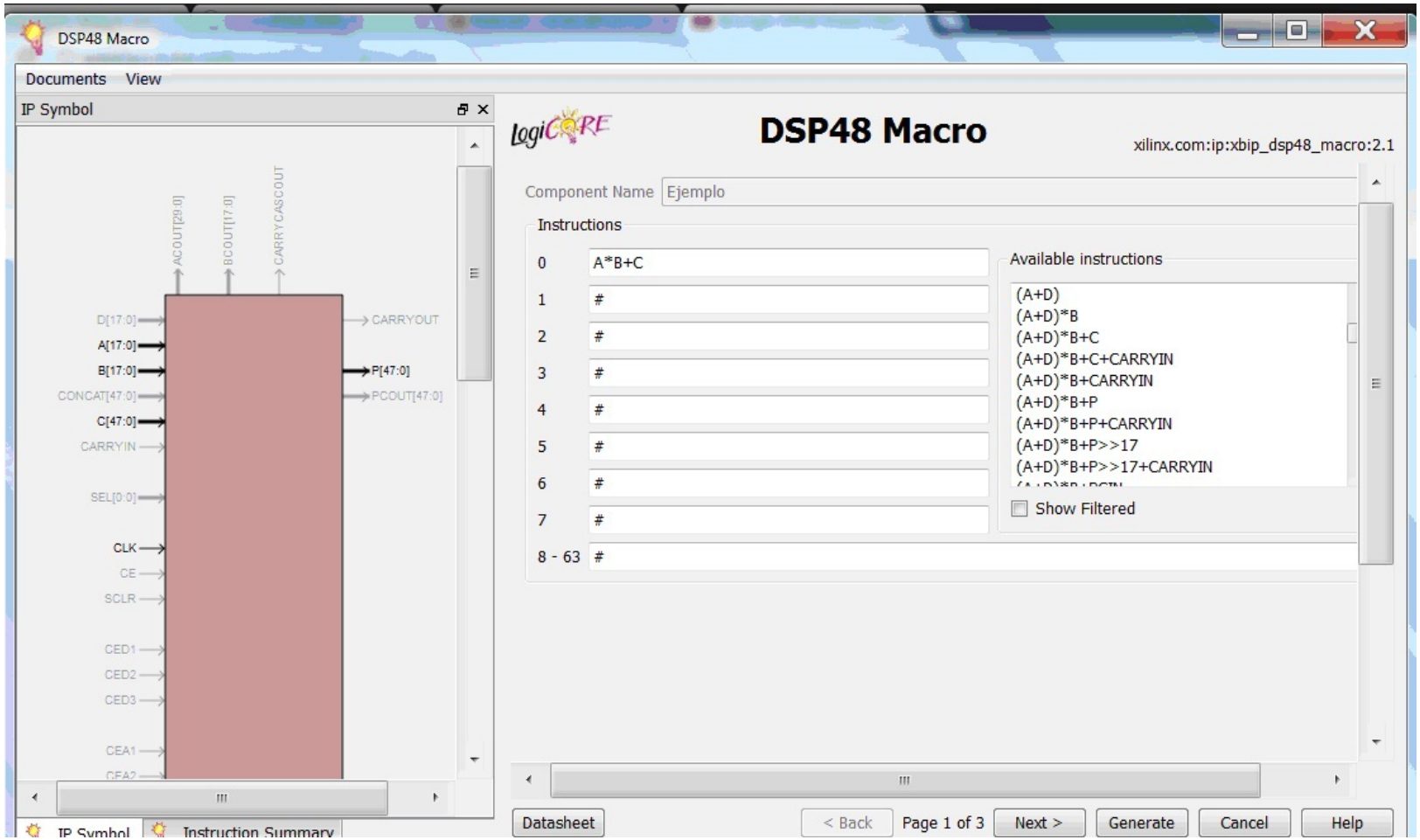
Programming a Xilinx DSP48Slice



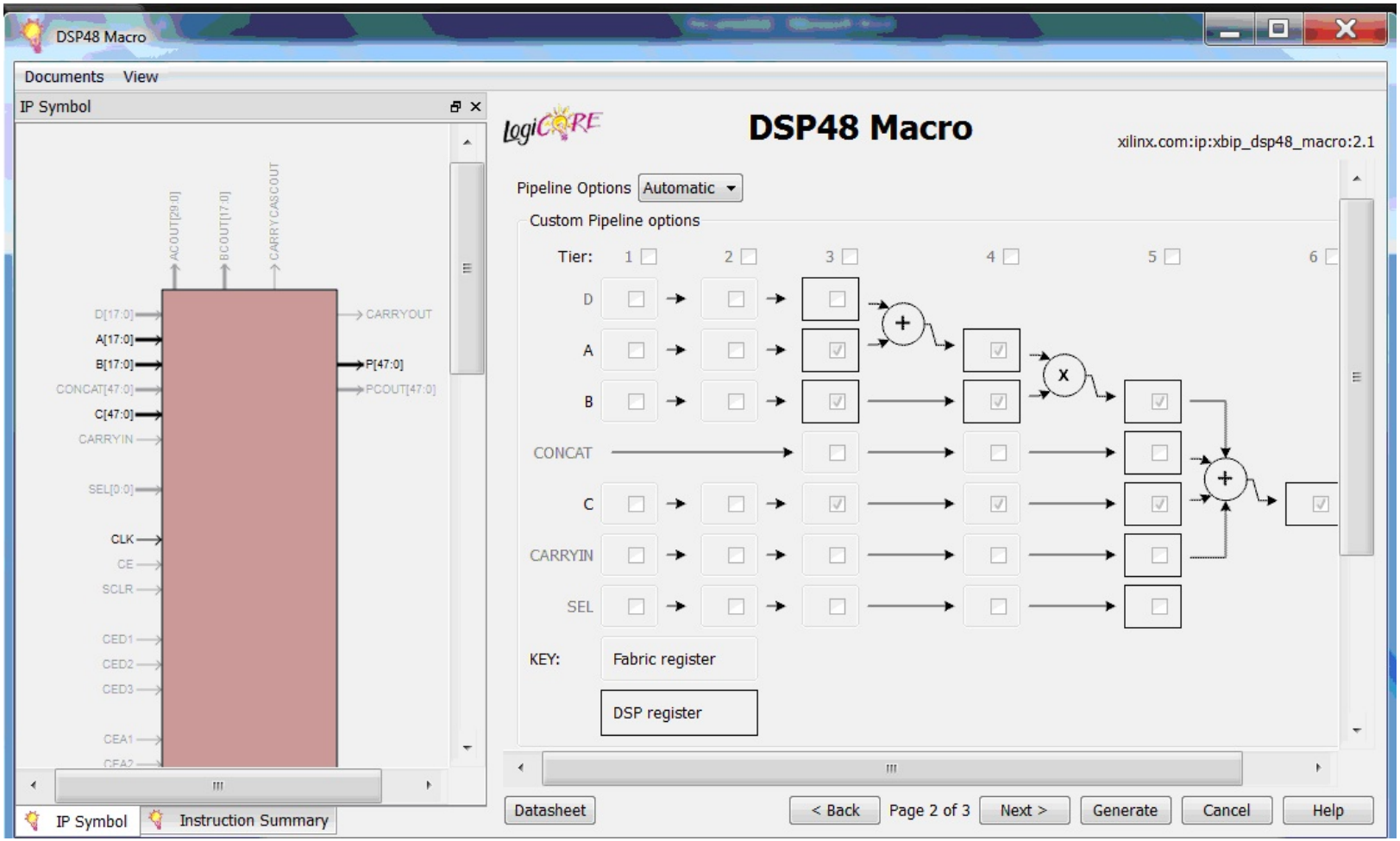
Programming a Xilinx DSP48Slice



Programming a Xilinx DSP48Slice



Programming a Xilinx DSP48Slice



5-term Karatsuba-like multiplication algorithm

Input: $a(t) = \sum_{i=0}^{4} a_i t^i$, $b(t) = \sum_{i=0}^{4} b_i t^i$
Output: $c(t) = a(t)b(t)$

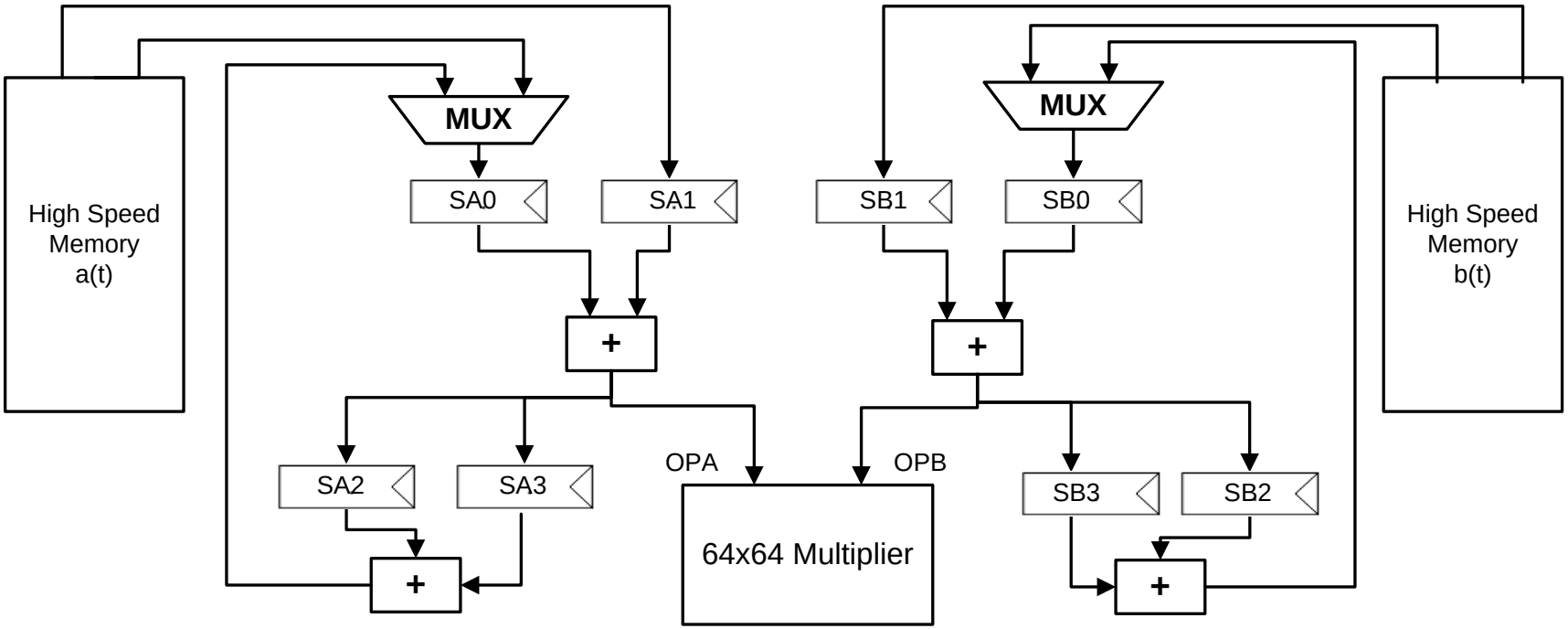
1. $c(t)(= \sum_{i=0}^{8} c_i t^i) \leftarrow 0$;
2. $p_0 = a_0 b_0$;
3. $p_1 = a_1 b_1$;
4. $p_2 = (a_0 + a_1)(b_0 + b_1)$;
5. $p_3 = a_2 b_2$;
6. $p_4 = (a_0 + a_2)(b_0 + b_2)$;
7. $p_5 = a_3 b_3$;
8. $p_6 = (a_2 + a_3)(b_2 + b_3)$;
9. $p_7 = (a_1 + a_3)(b_1 + b_3)$; (Initial Addition and product phase)
10. $p_8 = (a_0 + a_1 + a_2 + a_3)(b_0 + b_1 + b_2 + b_3)$;
11. $p_9 = a_4 b_4$; (each of these are 64 integer multiplications)
12. $p_{10} = (a_0 + a_4)(b_0 + b_4)$;
13. $p_{11} = (a_0 + a_1 + a_4)(b_0 + b_1 + b_4)$;
14. $p_{12} = (a_2 + a_4)(b_2 + b_4)$;
15. $p_{13} = (a_2 + a_3 + a_4)(b_2 + b_3 + b_4)$
16. $c_0 = p_0$
17. $c_1 = p_2 - p_1 - p_0$
18. $c_2 = p_4 + p_1 - p_0 - p_3$
19. $S1 = p_6 - p_5 - p_3$
20. $c_3 = p_8 - p_7 - p4 - c_1 - S1$ (Final Addition phase)
21. $c_4 = p_{10} - p_9 - p_0 + p_3 + p_5 - p_1 + p_7$
22. $c_5 = p_{11} - p_1 - p_{10} - c_1 + S1$
23. $c_6 = p_{12} - p_9 + p_5 - p_3$ (each of these are 64 integer additions)
24. $c_7 = p_{13} - p_5 - p_{12} - S1$
25. $c_8 = p_9$
26. **return** $c(t)$

5-term Karatsuba-like multiplication algorithm

Input: $a(t) = \sum_{i=0}^{4} a_i t^i$, $b(t) = \sum_{i=0}^{4} b_i t^i$
Output: $c(t) = a(t)b(t)$

1. $c(t)(= \sum_{i=0}^{8} c_i t^i) \leftarrow 0$;
2. $p_0 = a_0 b_0$;
3. $p_1 = a_1 b_1$;
4. $p_2 = (a_0 + a_1)(b_0 + b_1)$;
5. $p_3 = a_2 b_2$;
6. $p_4 = (a_0 + a_2)(b_0 + b_2)$;
7. $p_5 = a_3 b_3$;
8. $p_6 = (a_2 + a_3)(b_2 + b_3)$;
9. $p_7 = (a_1 + a_3)(b_1 + b_3)$; (Initial Addition and product phase)
10. $p_8 = (a_0 + a_1 + a_2 + a_3)(b_0 + b_1 + b_2 + b_3)$;
11. $p_9 = a_4 b_4$; (each of these are 64 integer multiplications)
12. $p_{10} = (a_0 + a_4)(b_0 + b_4)$;
13. $p_{11} = (a_0 + a_1 + a_4)(b_0 + b_1 + b_4)$;
14. $p_{12} = (a_2 + a_4)(b_2 + b_4)$;
15. $p_{13} = (a_2 + a_3 + a_4)(b_2 + b_3 + b_4)$
16. $c_0 = p_0$
17. $c_1 = p_2 - p_1 - p_0$
18. $c_2 = p_4 + p_1 - p_0 - p_3$
19. $S1 = p_6 - p_5 - p_3$
20. $c_3 = p_8 - p_7 - p4 - c_1 - S1$ (Final Addition phase)
21. $c_4 = p_{10} - p_9 - p_0 + p_3 + p_5 - p_1 + p_7$
22. $c_5 = p_{11} - p_1 - p_{10} - c_1 + S1$
23. $c_6 = p_{12} - p_9 + p_5 - p_3$ (each of these are 64 integer additions)
24. $c_7 = p_{13} - p_5 - p_{12} - S1$
25. $c_8 = p_9$
26. **return** $c(t)$

Initial addition phase



5-term Karatsuba-like multiplication algorithm

Input: $a(t) = \sum_{i=0}^{4} a_i t^i$, $b(t) = \sum_{i=0}^{4} b_i t^i$
Output: $c(t) = a(t)b(t)$

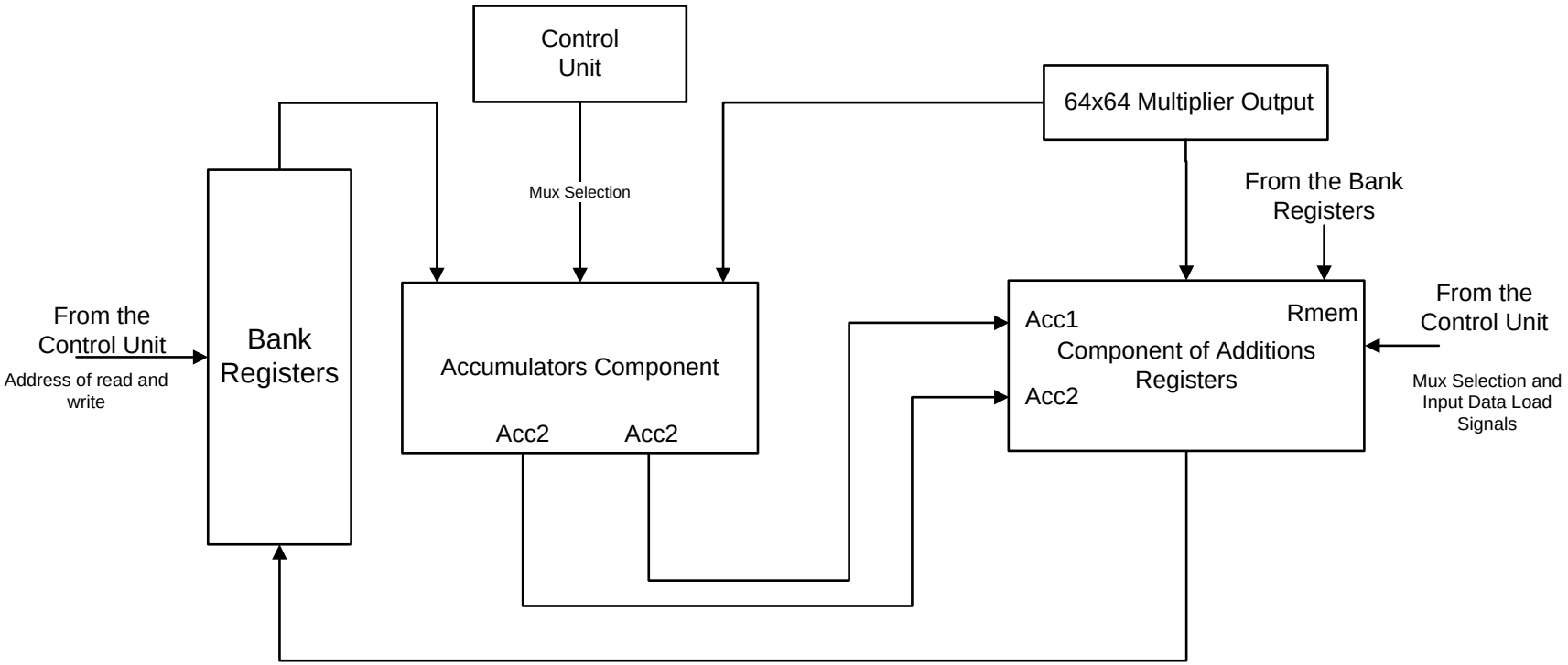
1. $c(t)(= \sum_{i=0}^{8} c_i t^i) \leftarrow 0$;
2. $p_0 = a_0 b_0$;
3. $p_1 = a_1 b_1$;
4. $p_2 = (a_0 + a_1)(b_0 + b_1)$;
5. $p_3 = a_2 b_2$;
6. $p_4 = (a_0 + a_2)(b_0 + b_2)$;
7. $p_5 = a_3 b_3$;
8. $p_6 = (a_2 + a_3)(b_2 + b_3)$;
9. $p_7 = (a_1 + a_3)(b_1 + b_3)$; (Initial Addition and product phase)
10. $p_8 = (a_0 + a_1 + a_2 + a_3)(b_0 + b_1 + b_2 + b_3)$;
11. $p_9 = a_4 b_4$; (each of these are 64 integer multiplications)
12. $p_{10} = (a_0 + a_4)(b_0 + b_4)$;
13. $p_{11} = (a_0 + a_1 + a_4)(b_0 + b_1 + b_4)$;
14. $p_{12} = (a_2 + a_4)(b_2 + b_4)$;
15. $p_{13} = (a_2 + a_3 + a_4)(b_2 + b_3 + b_4)$
16. $c_0 = p_0$
17. $c_1 = p_2 - p_1 - p_0$
18. $c_2 = p_4 + p_1 - p_0 - p_3$
19. $S1 = p_6 - p_5 - p_3$
20. $c_3 = p_8 - p_7 - p4 - c_1 - S1$ (Final Addition phase)
21. $c_4 = p_{10} - p_9 - p_0 + p_3 + p_5 - p_1 + p_7$
22. $c_5 = p_{11} - p_1 - p_{10} - c_1 + S1$
23. $c_6 = p_{12} - p_9 + p_5 - p_3$ (each of these are 64 integer additions)
24. $c_7 = p_{13} - p_5 - p_{12} - S1$
25. $c_8 = p_9$
26. **return** $c(t)$

5-term Karatsuba-like multiplication algorithm

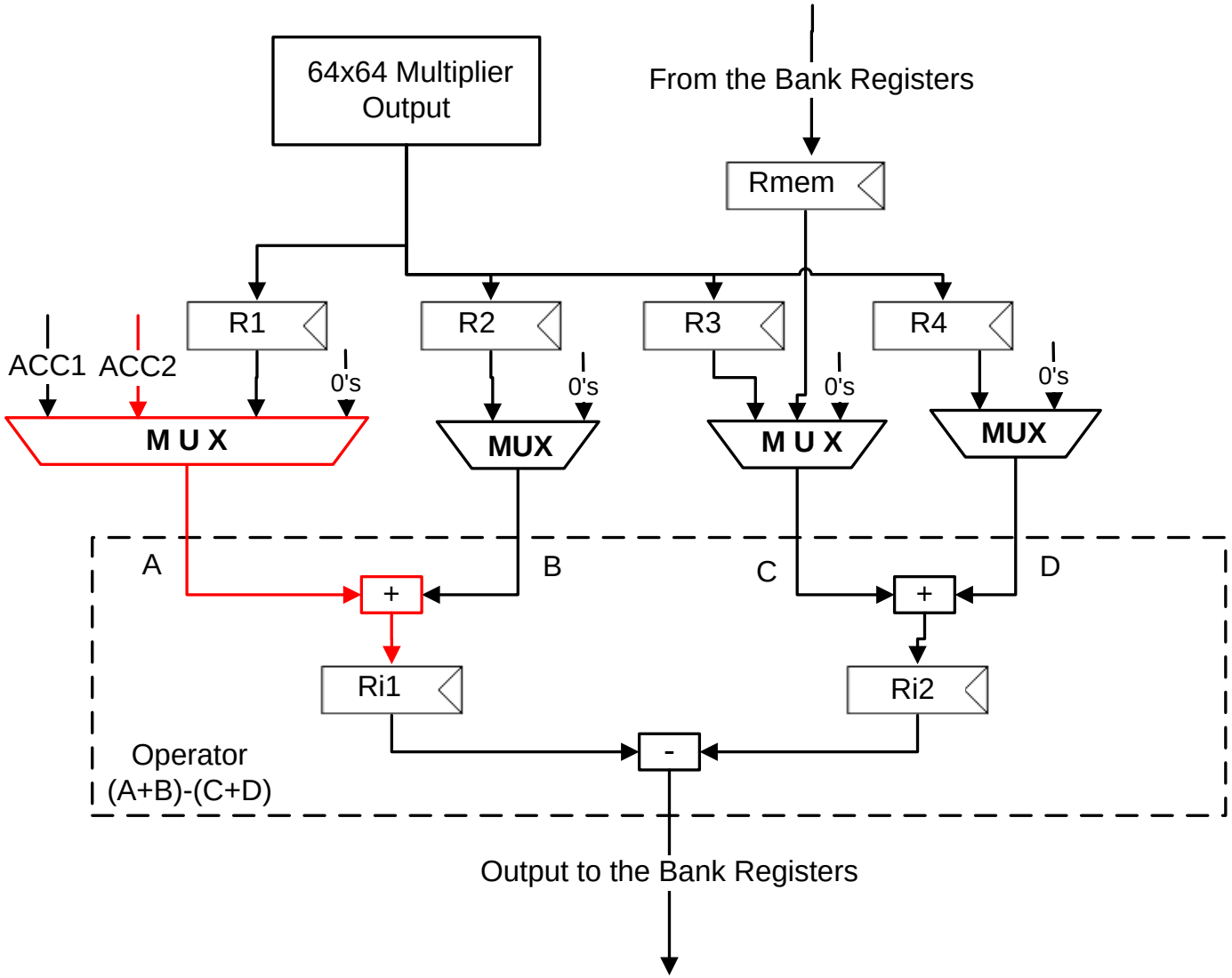
Input: $a(t) = \sum_{i=0}^{4} a_i t^i$, $b(t) = \sum_{i=0}^{4} b_i t^i$
Output: $c(t) = a(t)b(t)$

1. $c(t)(= \sum_{i=0}^{8} c_i t^i) \leftarrow 0$;
2. $p_0 = a_0 b_0$;
3. $p_1 = a_1 b_1$;
4. $p_2 = (a_0 + a_1)(b_0 + b_1)$;
5. $p_3 = a_2 b_2$;
6. $p_4 = (a_0 + a_2)(b_0 + b_2)$;
7. $p_5 = a_3 b_3$;
8. $p_6 = (a_2 + a_3)(b_2 + b_3)$;
9. $p_7 = (a_1 + a_3)(b_1 + b_3)$; (Initial Addition and product phase)
10. $p_8 = (a_0 + a_1 + a_2 + a_3)(b_0 + b_1 + b_2 + b_3)$;
11. $p_9 = a_4 b_4$; (each of these are 64 integer multiplications)
12. $p_{10} = (a_0 + a_4)(b_0 + b_4)$;
13. $p_{11} = (a_0 + a_1 + a_4)(b_0 + b_1 + b_4)$;
14. $p_{12} = (a_2 + a_4)(b_2 + b_4)$;
15. $p_{13} = (a_2 + a_3 + a_4)(b_2 + b_3 + b_4)$
16. $c_0 = p_0$
17. $c_1 = p_2 - p_1 - p_0$
18. $c_2 = p_4 + p_1 - p_0 - p_3$
19. $S1 = p_6 - p_5 - p_3$
20. $c_3 = p_8 - p_7 - p4 - c_1 - S1$ (Final Addition phase)
21. $c_4 = p_{10} - p_9 - p_0 + p_3 + p_5 - p_1 + p_7$
22. $c_5 = p_{11} - p_1 - p_{10} - c_1 + S1$
23. $c_6 = p_{12} - p_9 + p_5 - p_3$ (each of these are 64 integer additions)
24. $c_7 = p_{13} - p_5 - p_{12} - S1$
25. $c_8 = p_9$
26. **return** $c(t)$

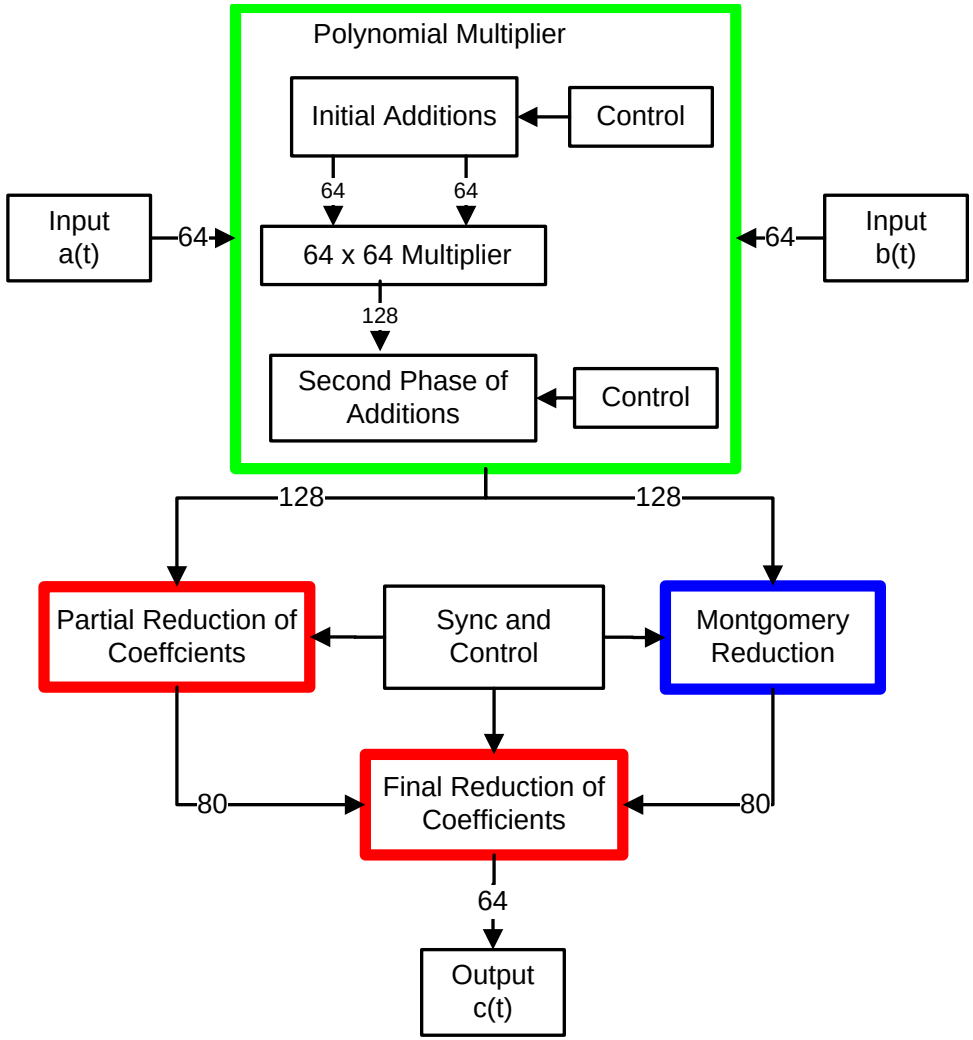
Final addition phase



Final addition phase

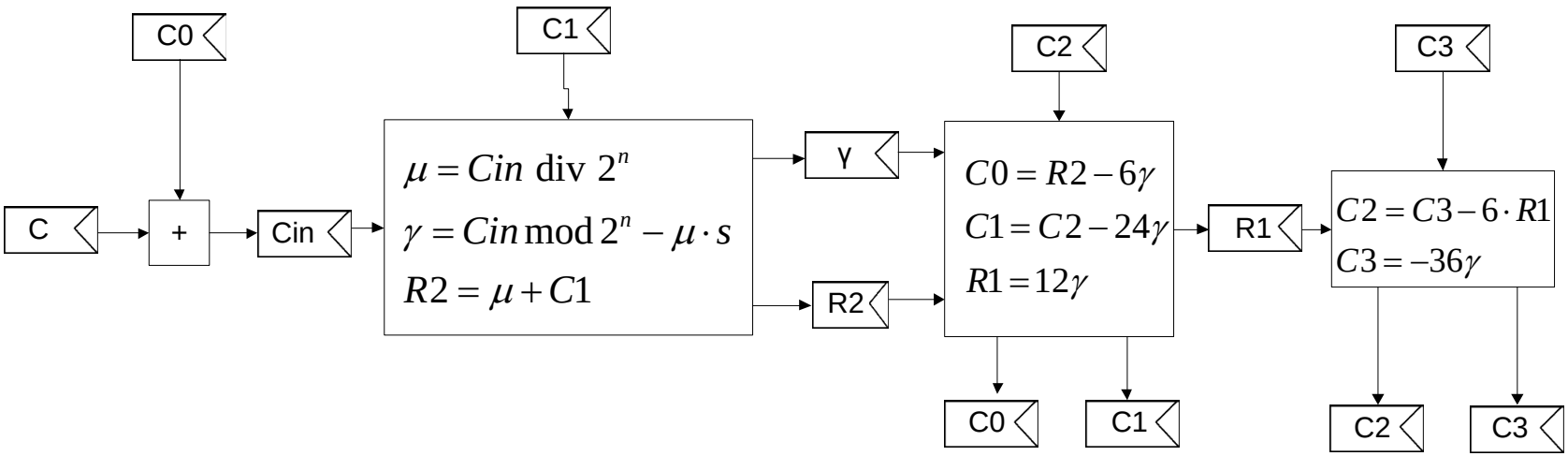


proposed architecture for the field multiplication



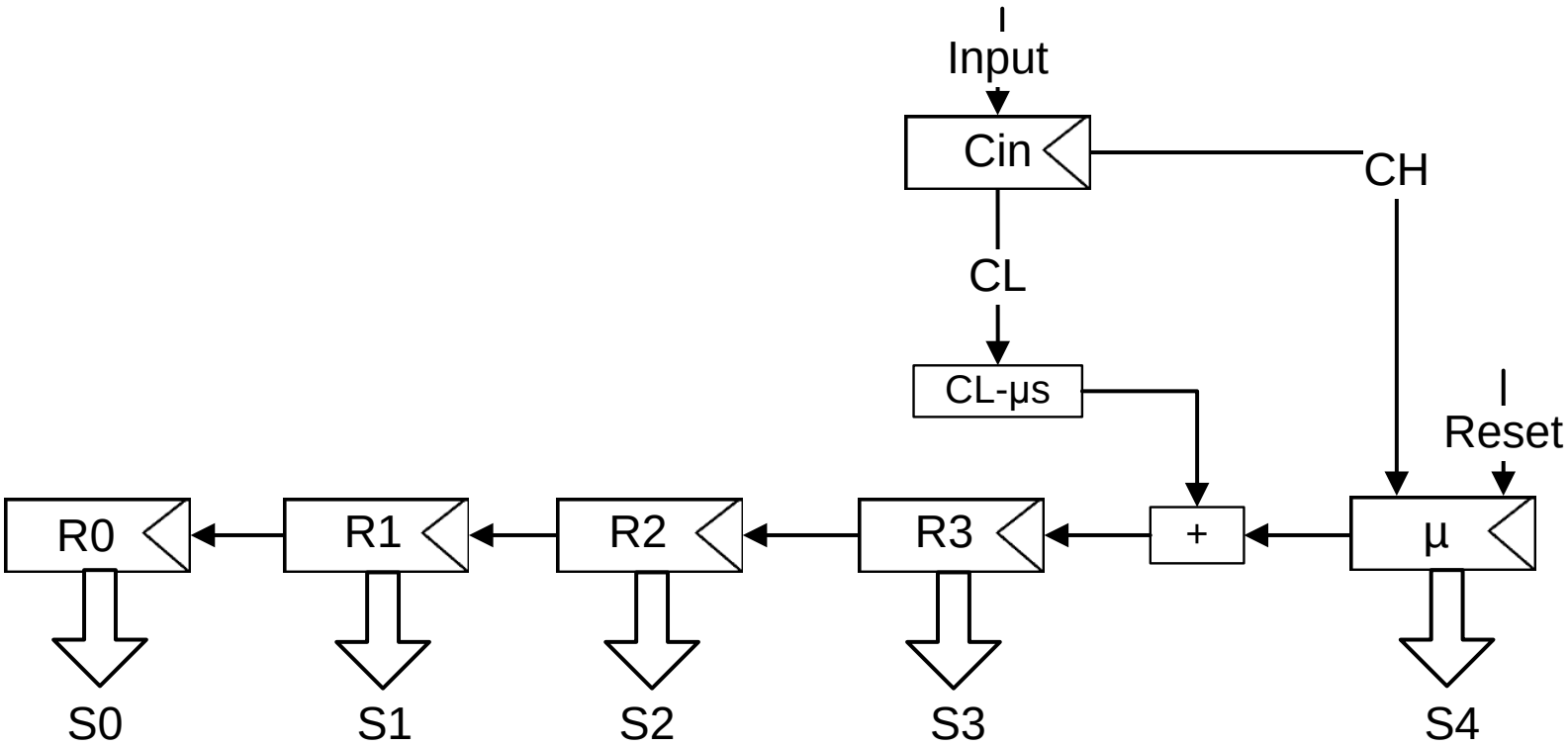
Montgomery reduction

Main Objective: To reduce the nine 128-bit coefficients produced by the Karatsuba multiplier to five coefficients of whom the least significant one is always zero



Coefficient reduction

Main Objective: To guarantee that the parameter t is greater than each coefficient C_i for $i = 5, \ldots 8$. This procedure receives four input coefficients but produces an output of a fifth coefficient always less than 36.



Speed Motivations

Experimental results (1/2)

Table: Performance per stage

Stage	**Latency**	**Frecuency (MHz)**
Multiplier input	3	270
64×64 Multiplier	4	313
Additions	18	223.7
Montgomery Reduction	18*	230
Coefficient Reduction	15*	223.7
Total	40	223.7

(*) Montgomery and coefficient reductions are performed in parallel

Experimental results (2/2)

Table: Comparison table

Design	platform	area Slices(DSPs)	cycles per product	Latency	Frecuency (MHz)
[Fan et. al. CHES'09]	ASIC	183KGates	23	0	204MHz
[Fan et. al. TC'11]	Virtex 6	4014(46)	5	25	210MHz
[Yao et. al. eprint'11]	Virtex 6	-(36)	15	8	250MHz
This design	Virtex 6	1983(12)	15	40	224MHz

Multiplication over $\mathbb{F}_{p^2}$

Input: $X, Y \in \mathbb{F}_{p^2}$, where $X = x_0 + x_1 u$ y $Y = y_0 + y_1 u$, y $u^2 = -5$
Output: $W = X \cdot Y \in \mathbb{F}_{p^2}$

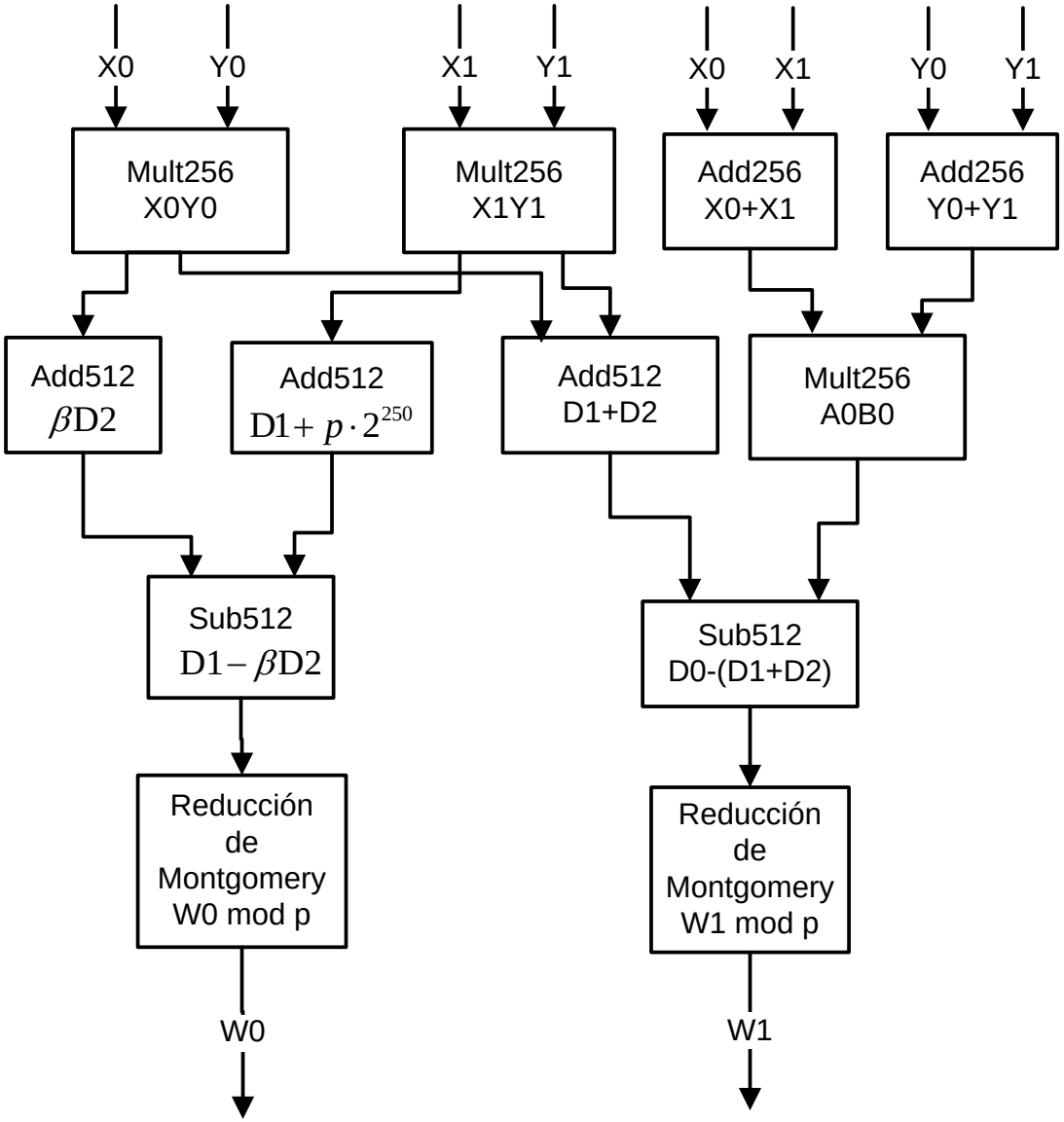
1. $s \leftarrow$ add256(x_0, x_1)
2. $t \leftarrow$ add256(y_0, y_0)
3. $d_0 \leftarrow$ mult256(s, t)
4. $d_1 \leftarrow$ mult256(x_0, y_0)
5. $d_2 \leftarrow$ mult256(x_1, y_1)
6. $d_0 \leftarrow$ sub512(d_0, d_1)
7. $d_0 \leftarrow$ sub512(d_0, d_2)
8. $w_1 \leftarrow$ mod512(d_0)
9. $d_2 \leftarrow 5d_2$
10. $d_1 \leftarrow$ sub512(d_1, d_2)
11. $w_0 \leftarrow$ mod512(d_1)
12. **return** $W = w_0 + w_1 u$

Multiplication over $\mathbb{F}_{p^2}$

Hence, the following components are required,

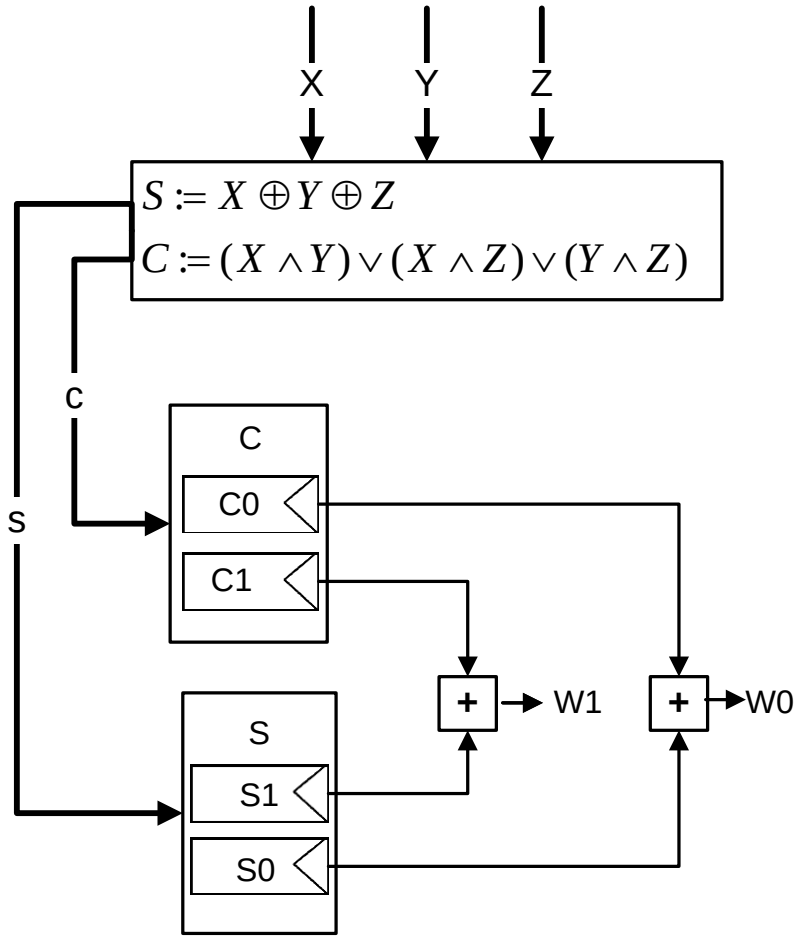
- 256-bit Adders/Subtracters
- 256 multipliers
- reduction logic

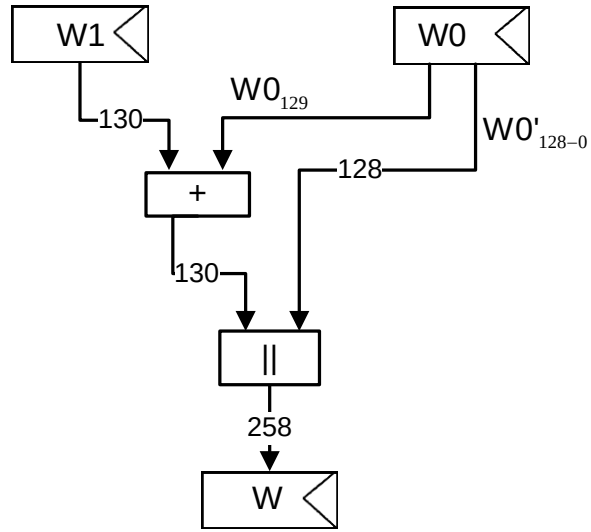
Multiplication over $\mathbb{F}_{p^2}$: proposed architecture



Multiplication over $\mathbb{F}_{p^2}$: carry-save adders

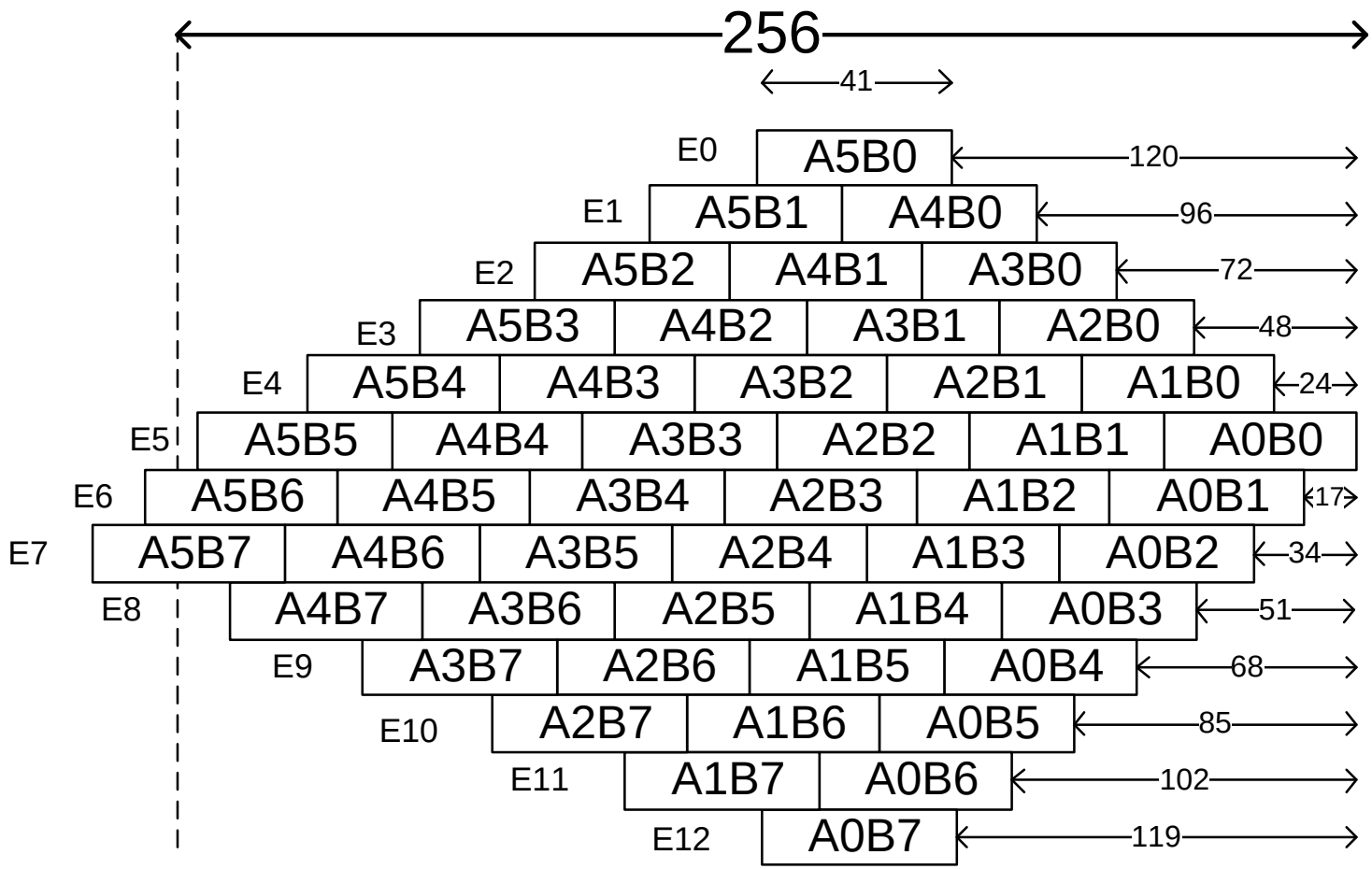
Taking advantage of the carry-save adders we can add three operands using that representation





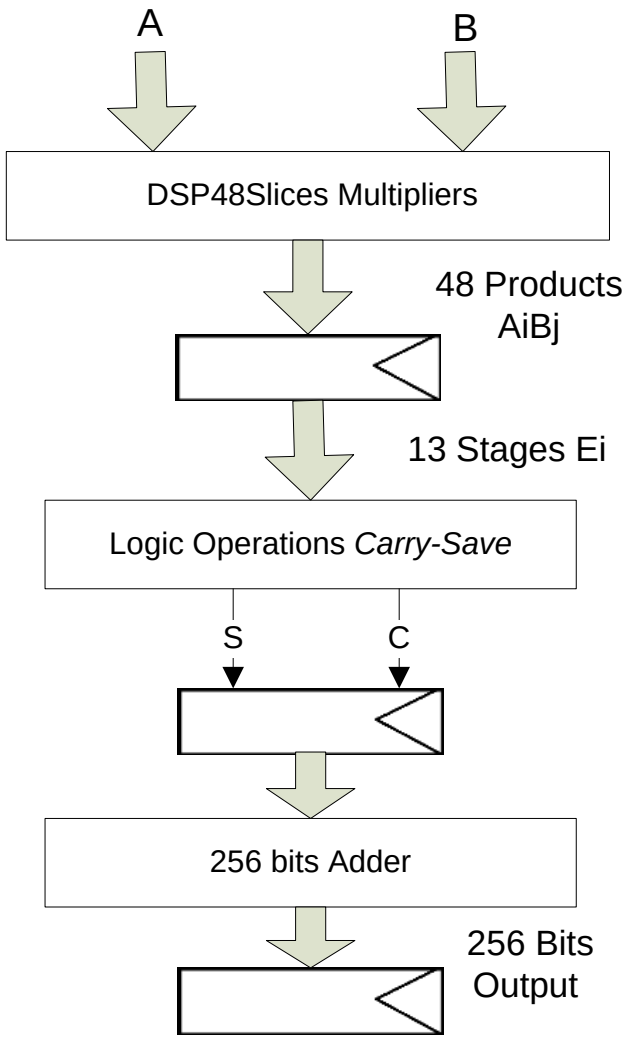
Multiplication over $\mathbb{F}_{p^2}$: 128-bit diamond multiplier

We once again use the diamond technique for defining a 128 bit multiplier



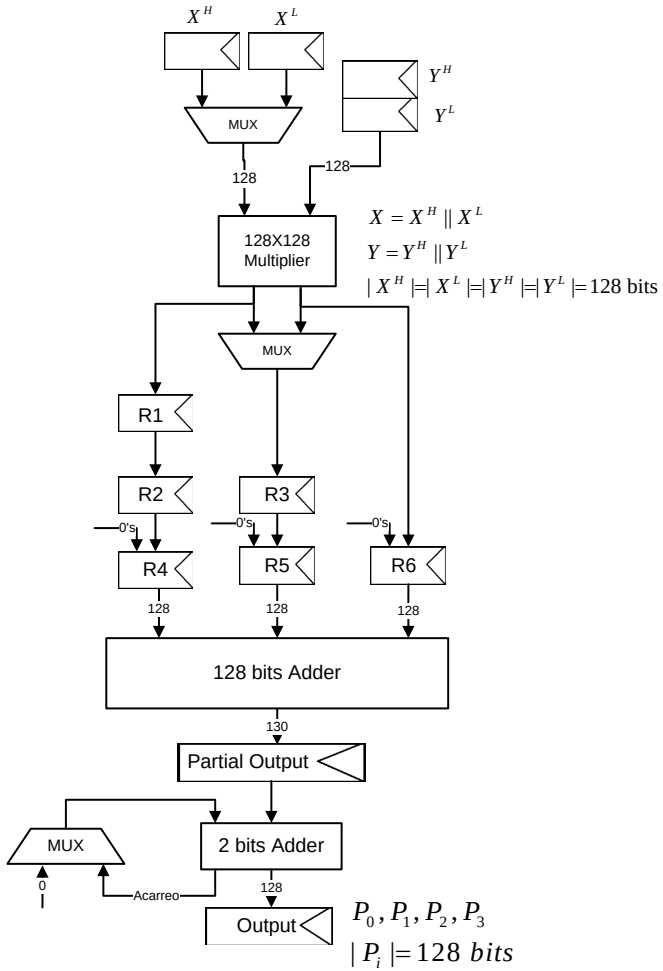
Multiplication over $\mathbb{F}_{p^2}$: 128-bit diamond multiplier

We once again use the diamond technique for defining a 128 bit multiplier



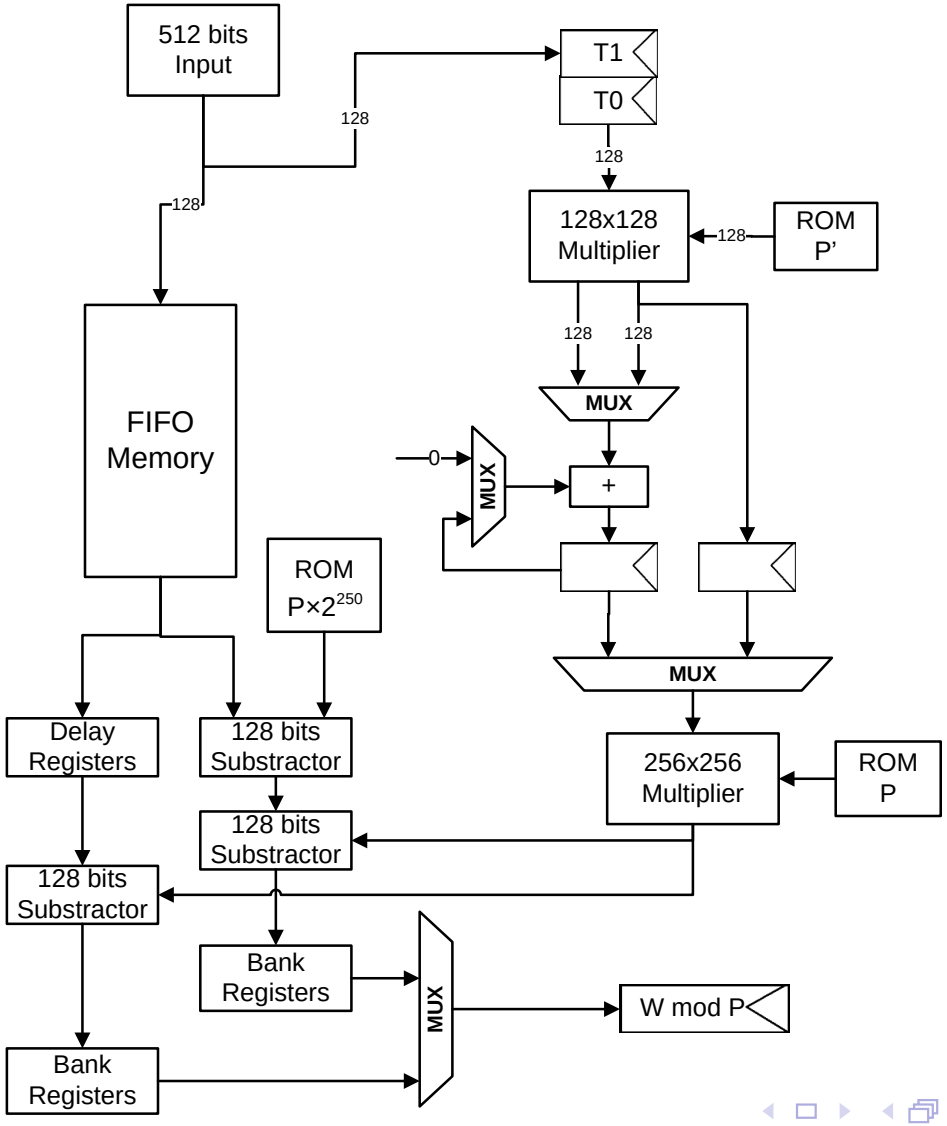
Multiplication over $\mathbb{F}_{p^2}$: 256-bit multiplier

The 256-bit multiplier is computed using the 128-bit multiplier as its main building block. Notice that the output of this circuit is of four 128-bit words.



Multiplication over $\mathbb{F}_{p^2}$: Montgomery reduction

The Montgomery reduction is performed keeping an eye on avoiding pipeline stalls



Performance revisited

Experimental results revisited

Table: Comparison table

Design	platform	area Slices(DSPs)	cycles per product	Latency	Frecuency (MHz)
[Fan et. al. CHES'09]	ASIC	183KGates	23	0	204MHz
[Fan et. al. TC'11]	Virtex 6	4014(46)	5	25	210MHz
[Yao et. al. eprint'11]	Virtex 6	-(36)	15	8	250MHz
$\mathbb{F}_p$ **design**	Virtex 6	1983(12)	15	40	224MHz
$\mathbb{F}_{p^2}$ **design**	Virtex 6	8754(336)	4	45	235MHz
$\mathbb{F}_p$ **mult using the** $\mathbb{F}_{p^2}$ **design**	Virtex 6	–(144)	4	35	235MHz

Credits

The material presented here is an adaptation of joint work with:

Jean-Luc Beuchat	LCIS, University of Tsukuba, Japan
Debrup Chakraborty	CINVESTAV-IPN, Computer Science, México
Cuauhtémoc Chávez-Corona	CINVESTAV-IPN, Computer Science, México
Cuauhtemoc Mancillas-López	CINVESTAV-IPN, Computer Science, México
Palash Sarkar	ISI, India

Thank you for your attention

Questions?