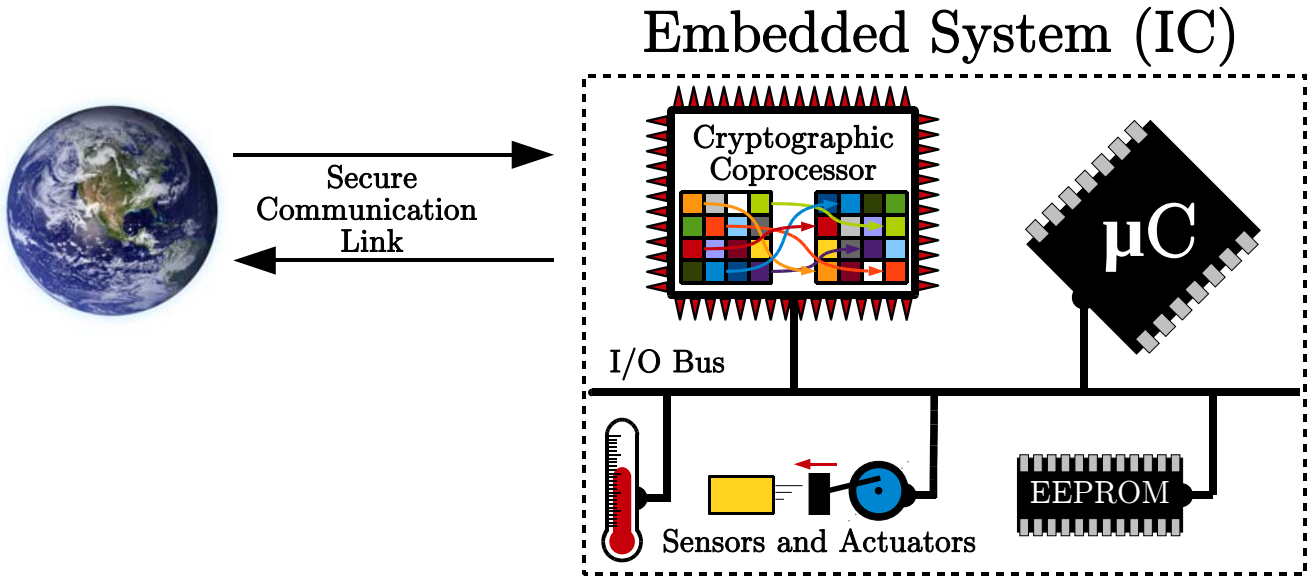
Sharing is Caring

On the Protection of Arithmetic Logic Units against Passive Physical Attacks

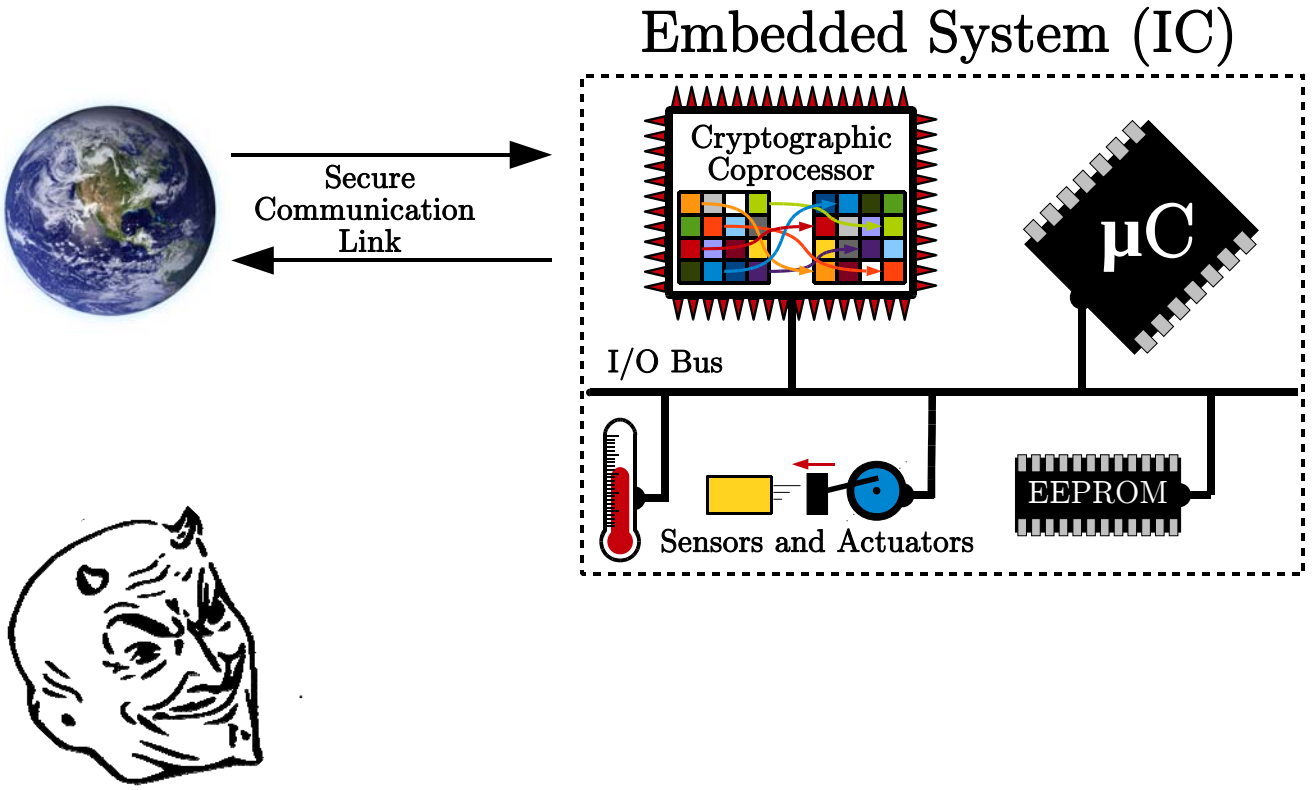
Hannes Groß

Institute for Applied Information Processing and Communications
Graz University of Technology
hannes.gross@iaik.tugraz.at

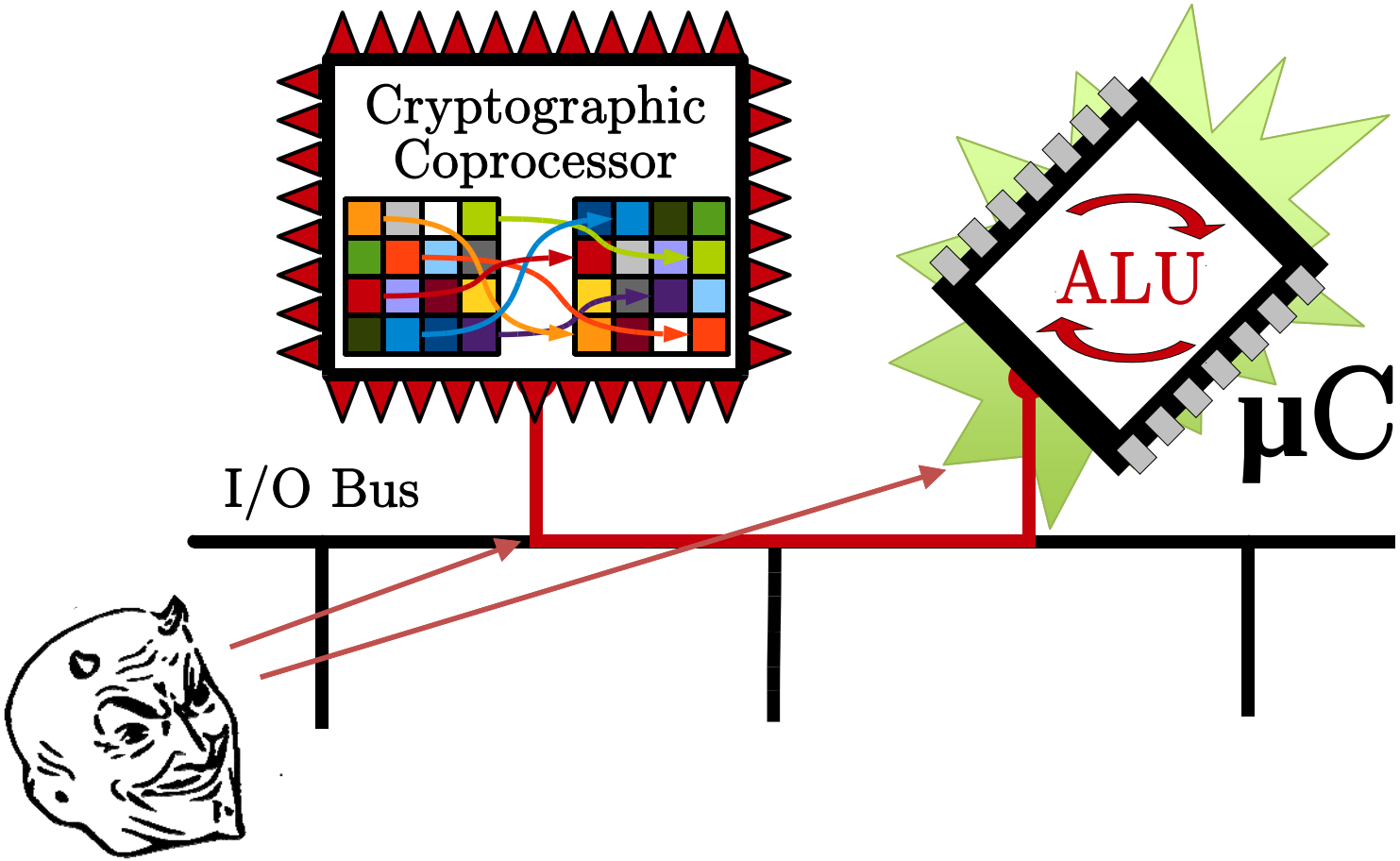
Motivation


Embedded System (IC)
Secure
Communication
Link
Cryptographic
Coprocessor
µC
I/O Bus
EEPROM
Sensors and Actuators

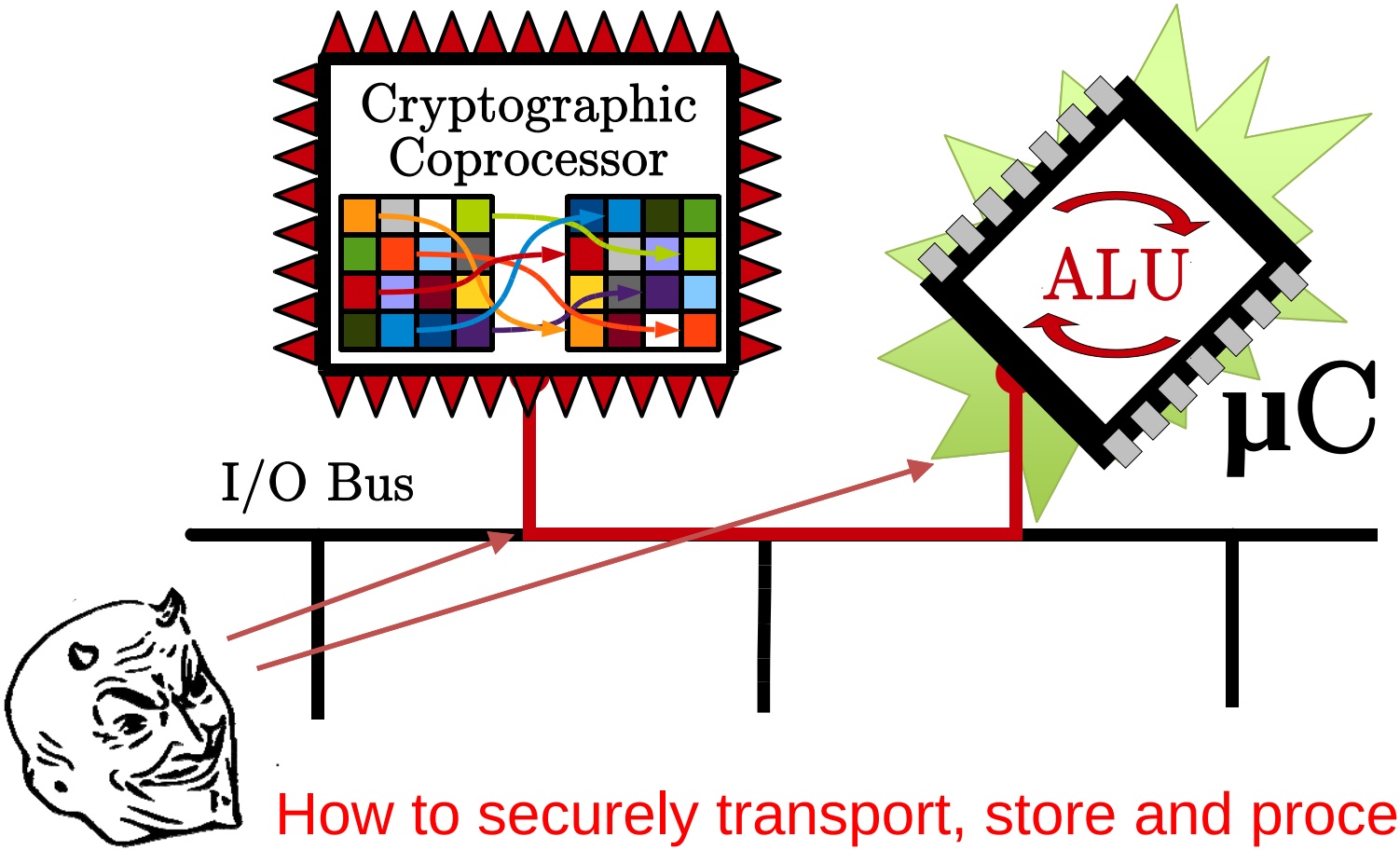
Motivation


Embedded System (IC)
Secure
Communication
Link
Cryptographic
Coprocessor
µC
I/O Bus
EEPROM
Sensors and Actuators

Motivation



Motivation

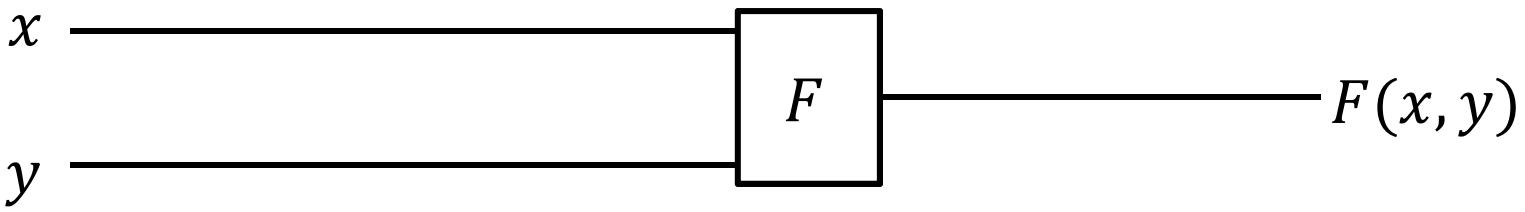


How to securely transport, store and process critical plaintext data inside the system?

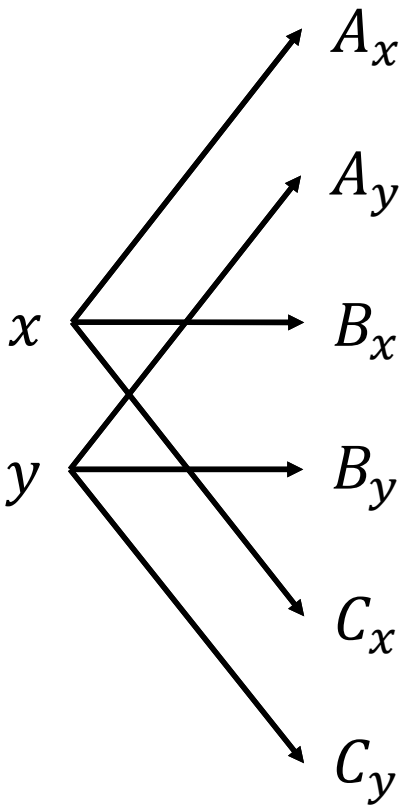
Adversary

- Strong but not almighty
- Limited to first-order passive attacks
  - Probe one chip wire
  - Perform a first-order DPA
  - …

Threshold Implementations

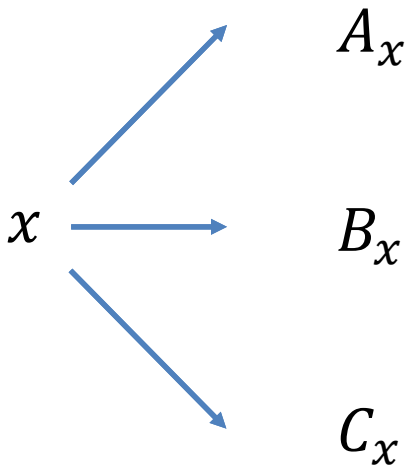


Threshold Implementations

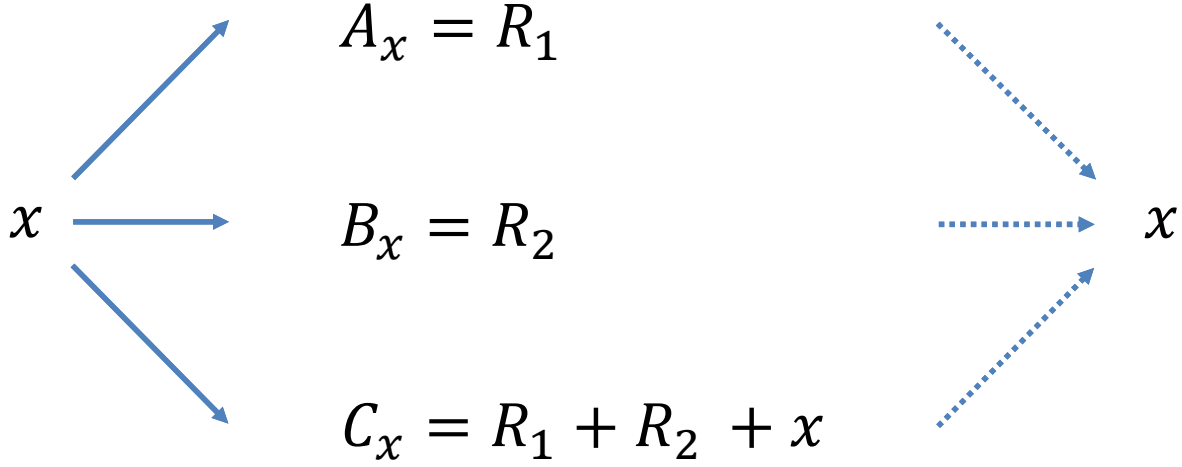


input shares

Core Idea: “Secret Sharing”



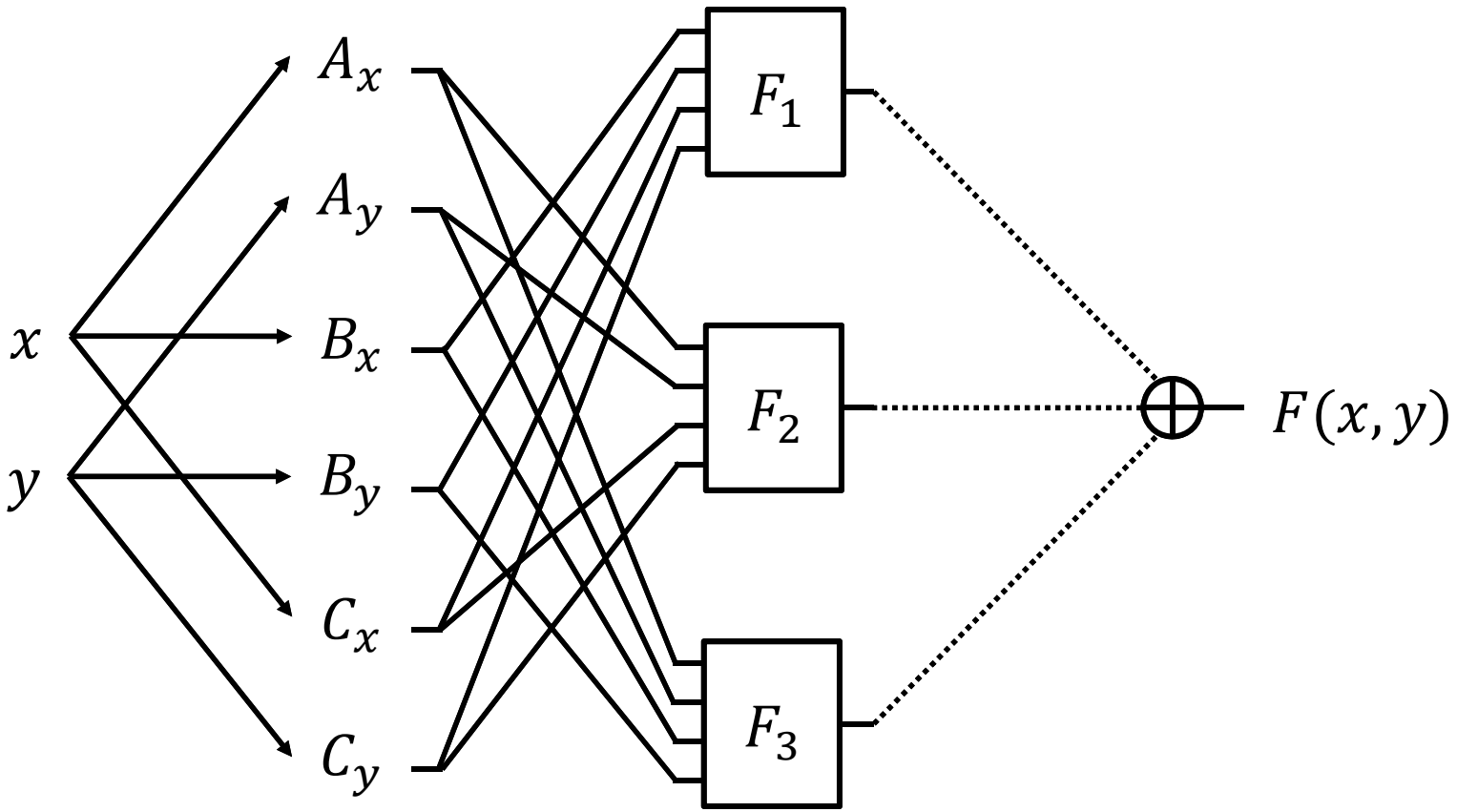
Core Idea: “Secret Sharing”



$$A_x, B_x, C_x, R_1, R_2, x \in GF(2^n)$$

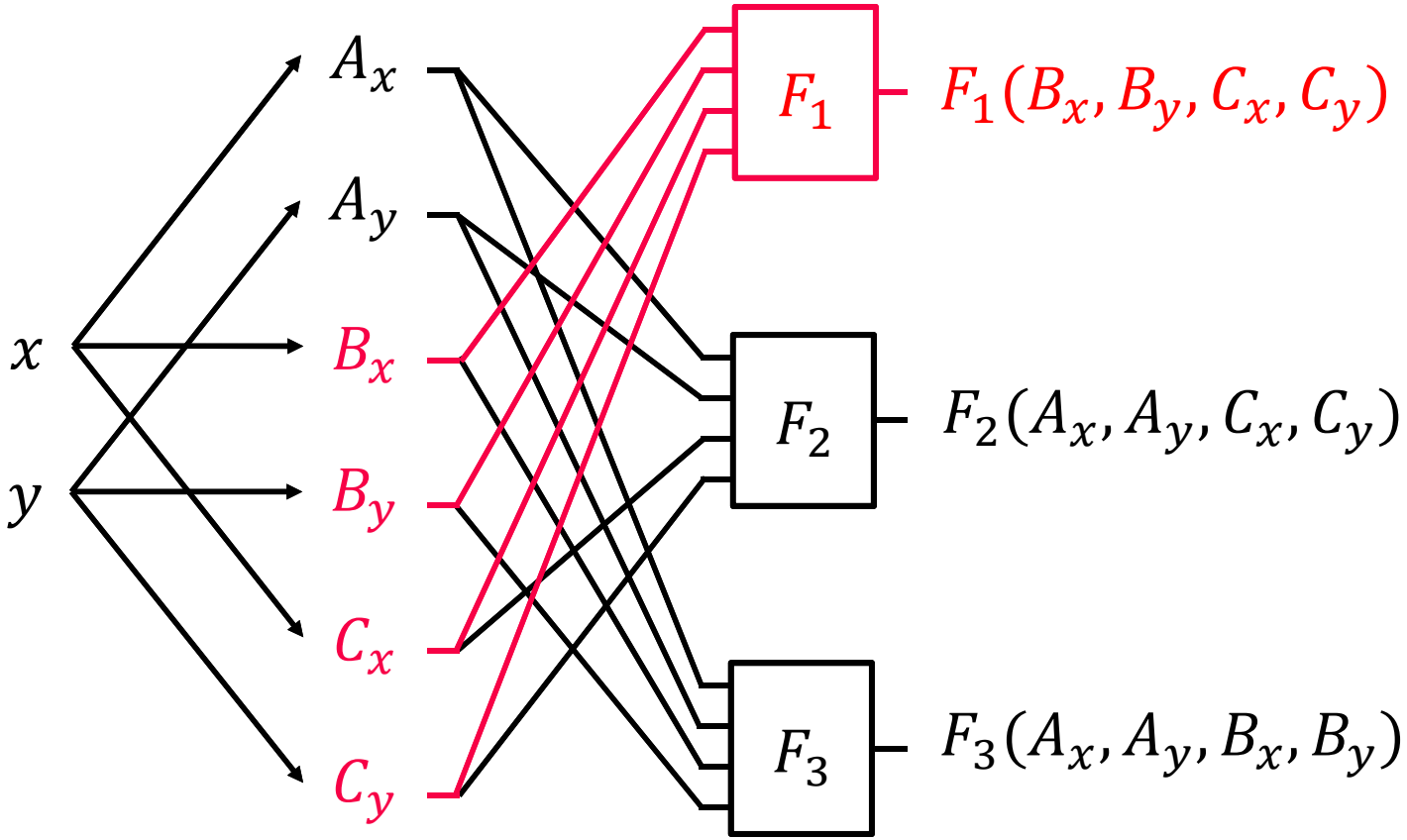
Secure transportation and storage ✓

TI Requirements: “1. Correctness”

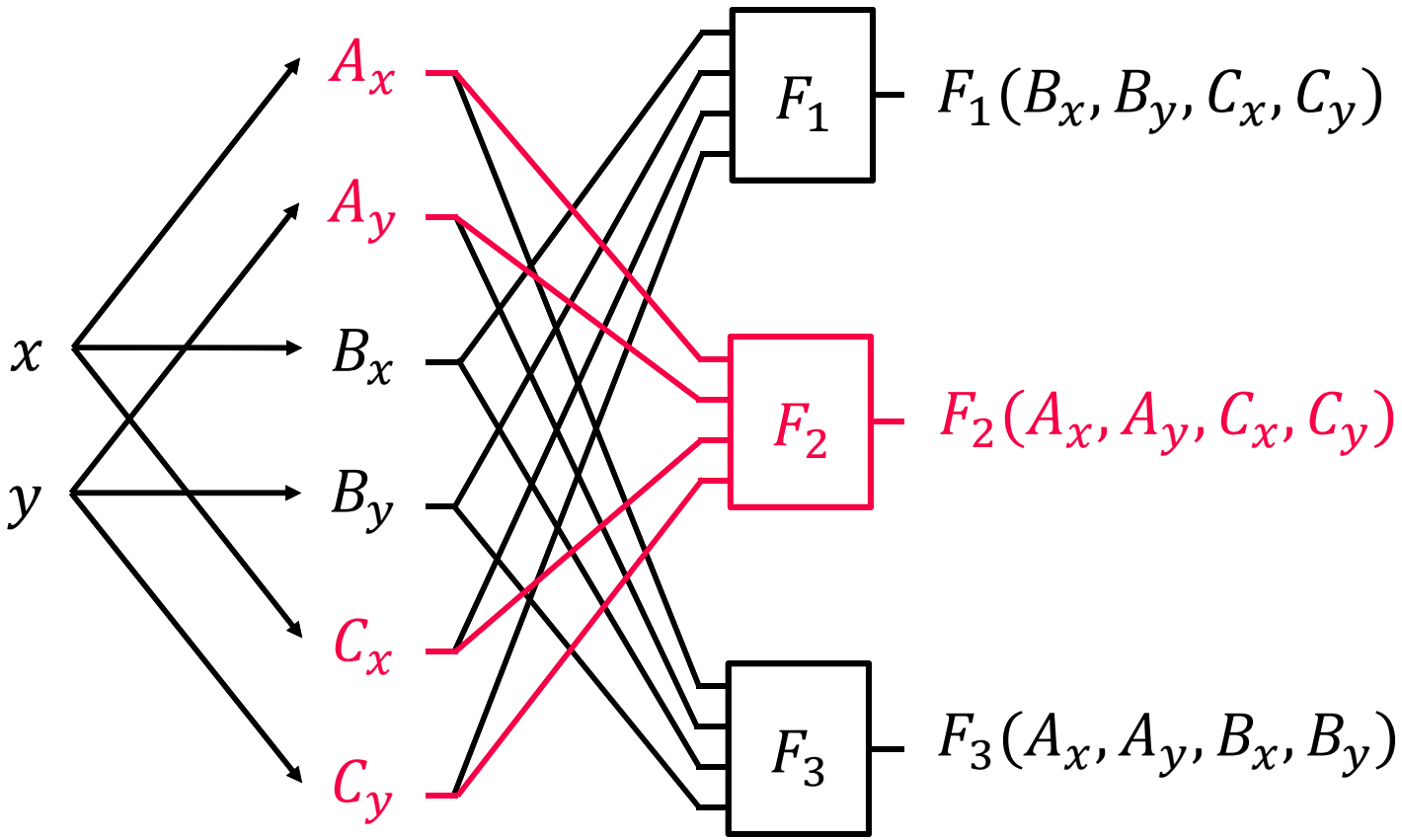


component functions

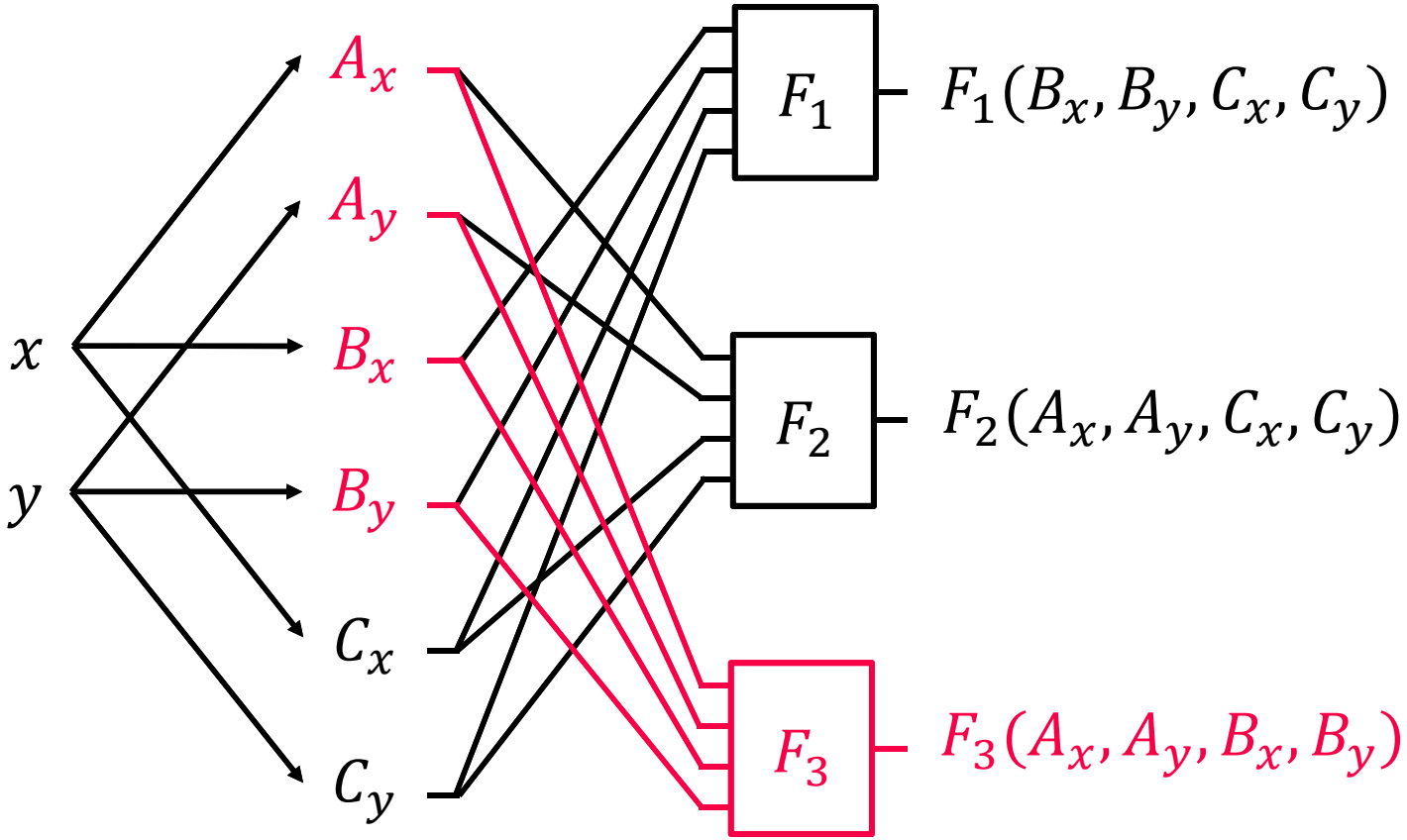
TI Requirements: “2. Non-completeness”



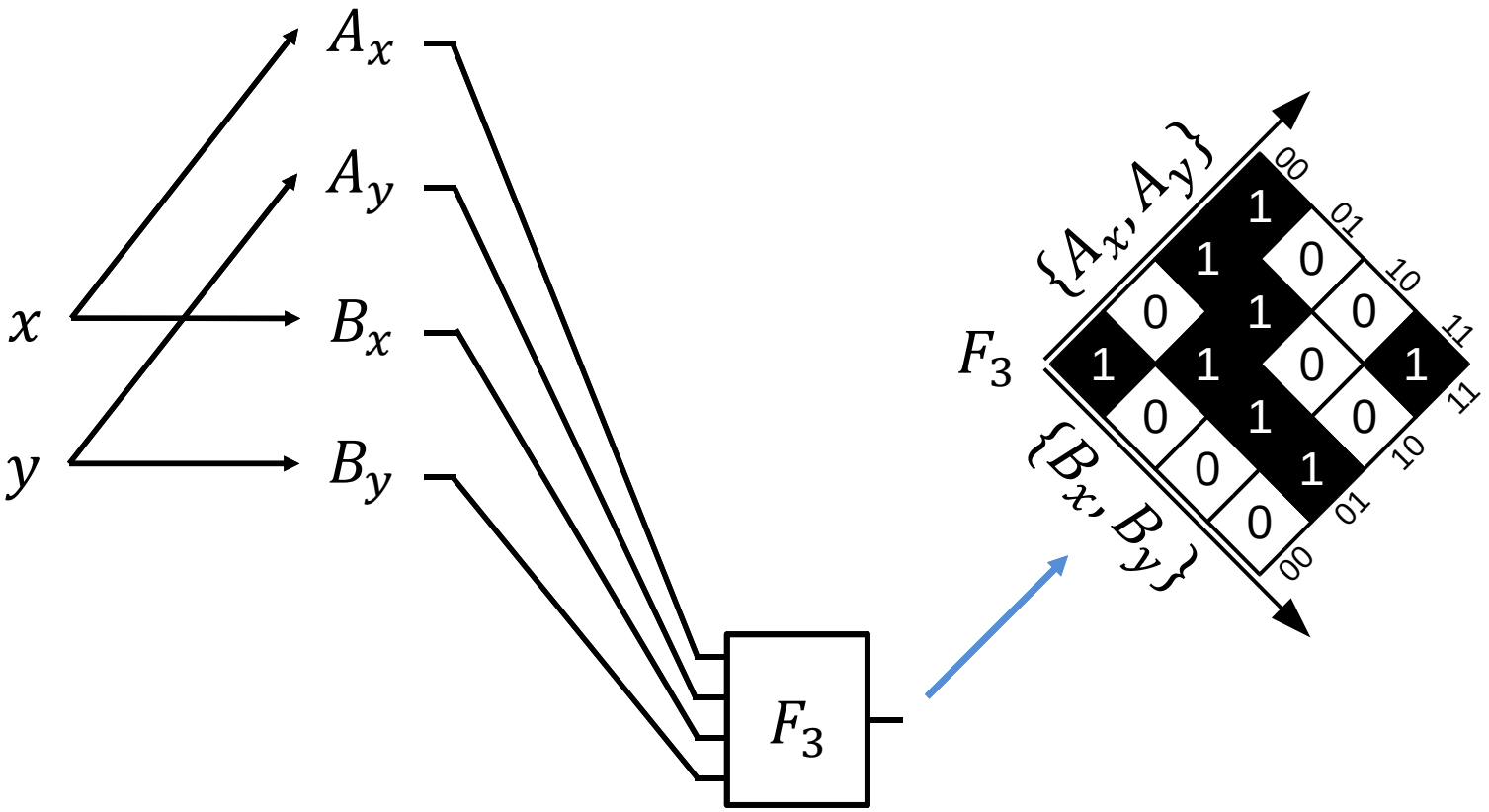
TI Requirements: “2. Non-completeness”



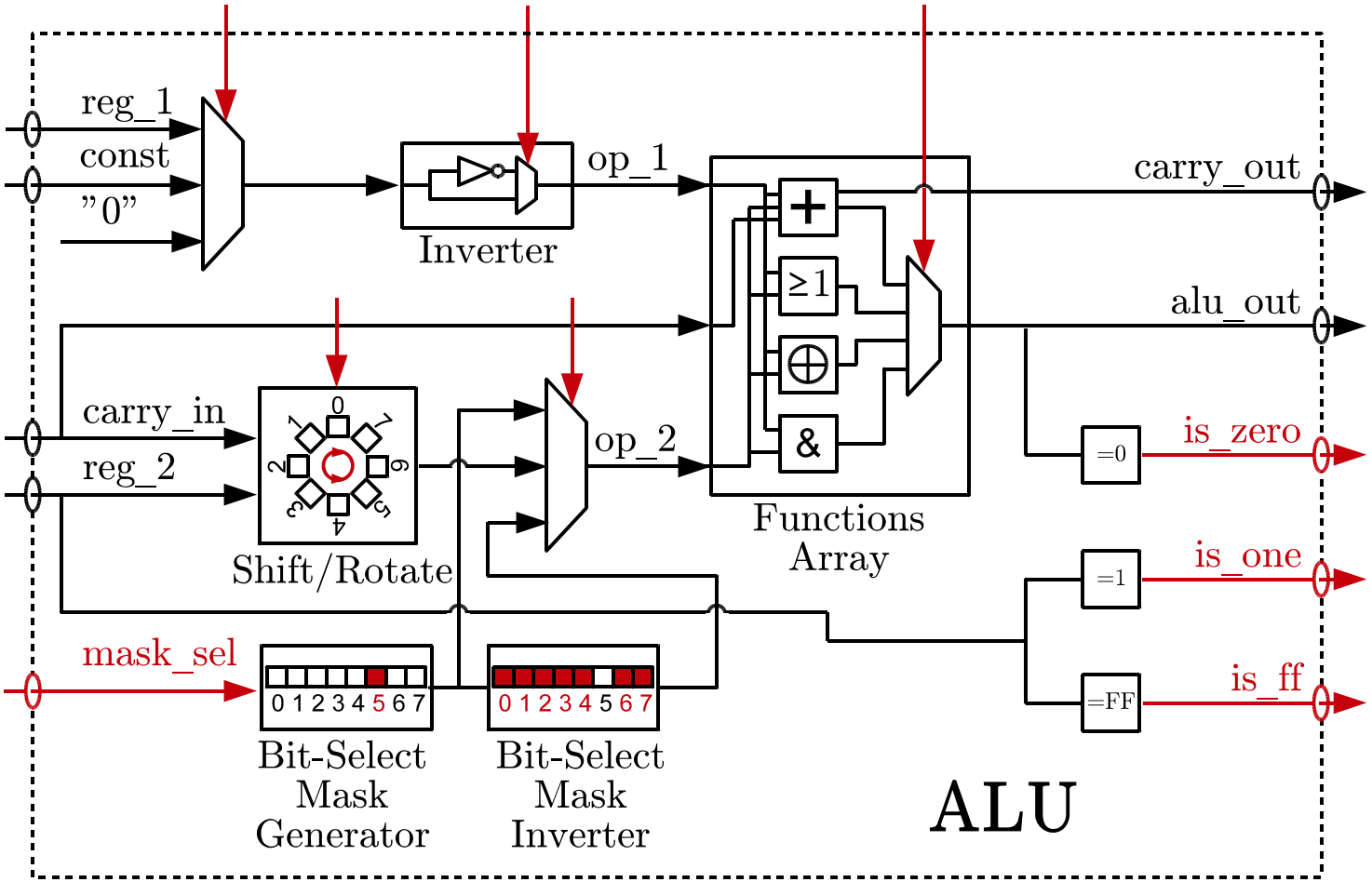
TI Requirements: “2. Non-completeness”



TI Requirements: “3. Uniformity”



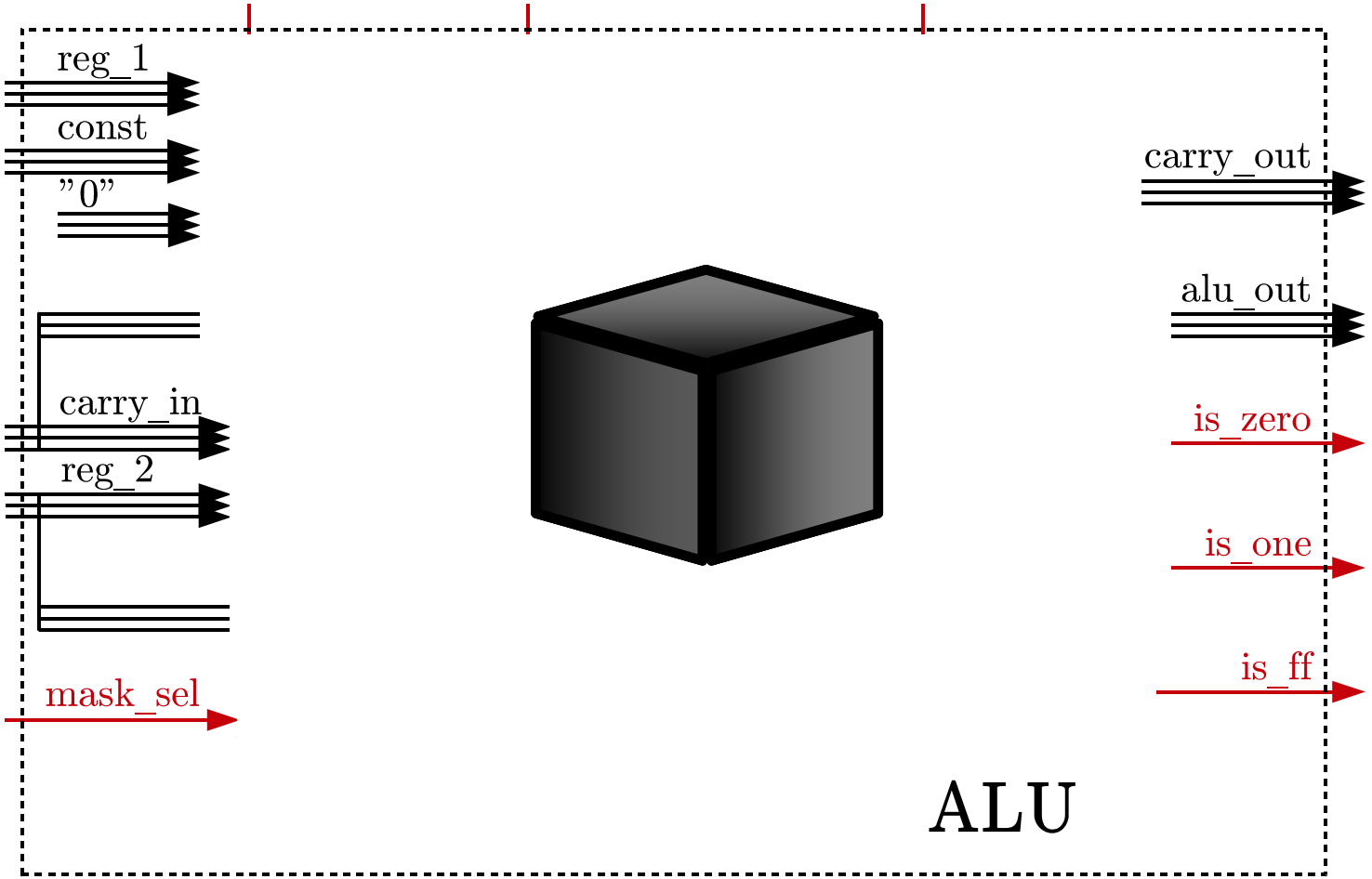
Arithmetic Logic Unit



Assumption and Requirements

- ALU data inputs need to be uniformly shared
- Data independent control signals are not shared
- ALU output is again a valid and uniform sharing
- Goal: Minimize implementation overhead

Shared Arithmetic Logic Unit

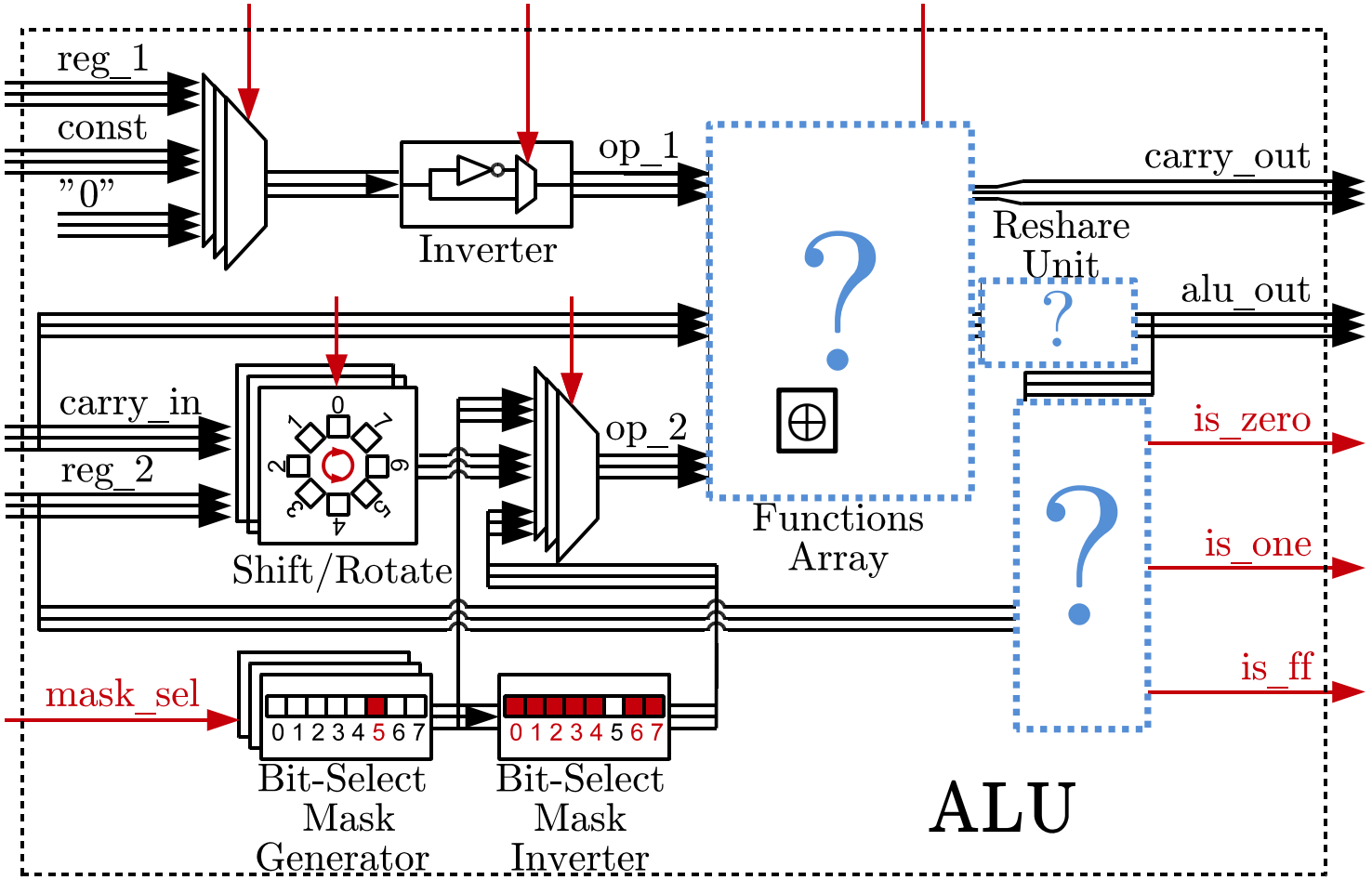


Functions that need to be shared

- Functions linear over $GF(2^n)$
  - Shift/Rotate, Negation, XOR
  - → simple, operate on shares separately

- Non-linear functions

Functions that need to be shared



Functions that need to be shared

- Functions linear over $GF(2^n)$
  - Shift/Rotate, Negotiation, XOR
  - → simple, operate on shares separately

- Non-linear functions
  - Boolean operations: AND, OR
  - Arithmetic operations: Addition

Example: Direct Sharing of OR

$$x \vee y =$$

Example: Direct Sharing of OR

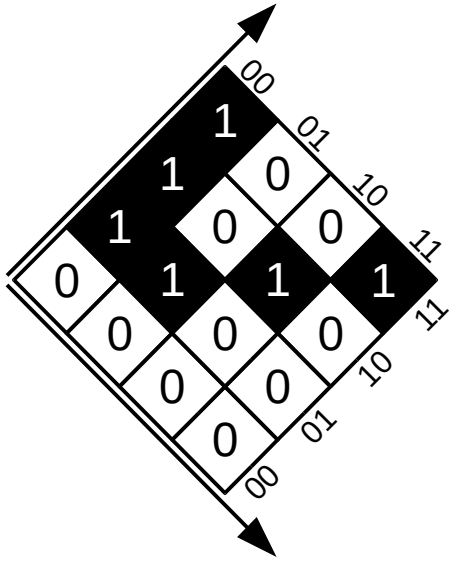
$$x \vee y = (A_x + B_x + C_x) \vee (A_y + B_y + C_y)$$
$$= \ldots$$

Example: Direct Sharing of OR

$$
\begin{aligned}
x \,V\, y &= (A_x + B_x + C_x)\, V\, (A_y + B_y + C_y) \\
&= \ldots \\
&= B_x B_y + C_x B_y + B_y + B_x C_y + B_x \qquad F_1 \\
&+ A_x A_y + B_x A_y + A_y + A_x B_y + A_x \qquad F_2 \\
&+ C_x C_y + A_x C_y + C_y + C_x A_y + C_x \qquad F_3
\end{aligned}
$$

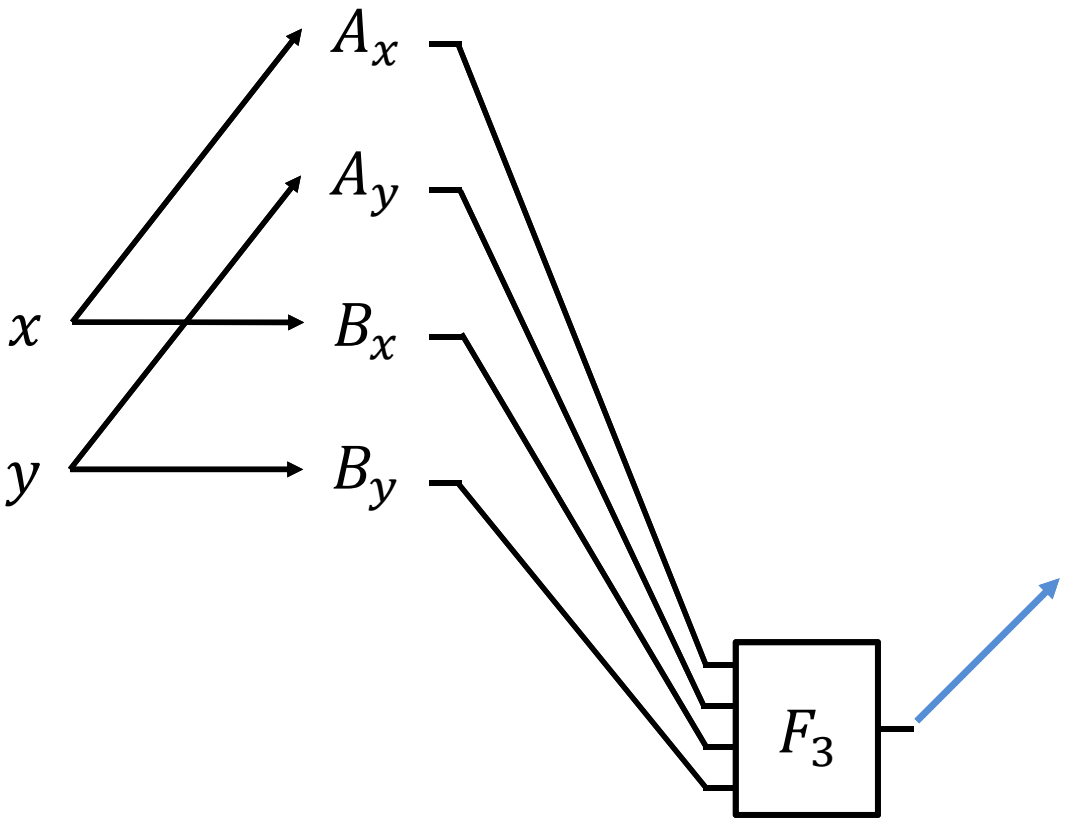
Example: Direct Sharing of OR

$$F_1 = B_x B_y + C_x B_y + B_y + B_x C_y + B_x$$
$$F_2 = A_x A_y + B_x A_y + A_y + A_x B_y + A_x$$
$$F_3 = C_x C_y + A_x C_y + C_y + C_x A_y + C_x$$



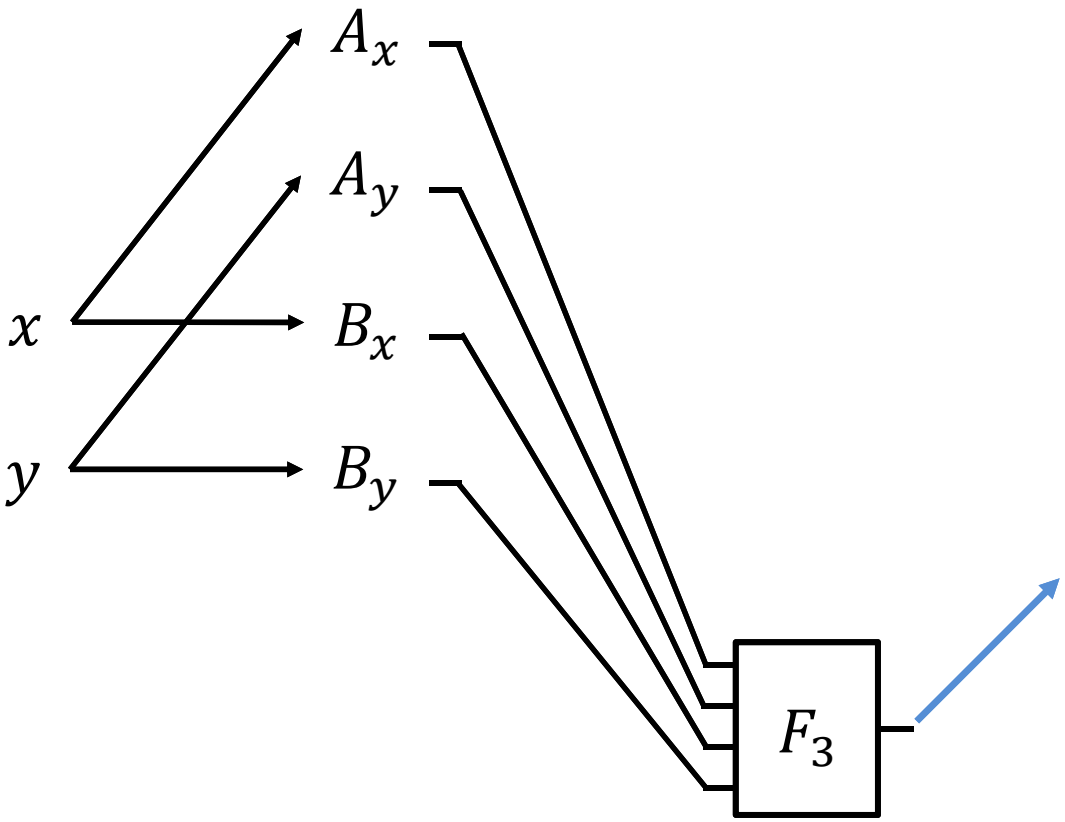
Bilgin et al. → no uniform sharing for a nonlinear function with three shares

Exhaustive Search Approach



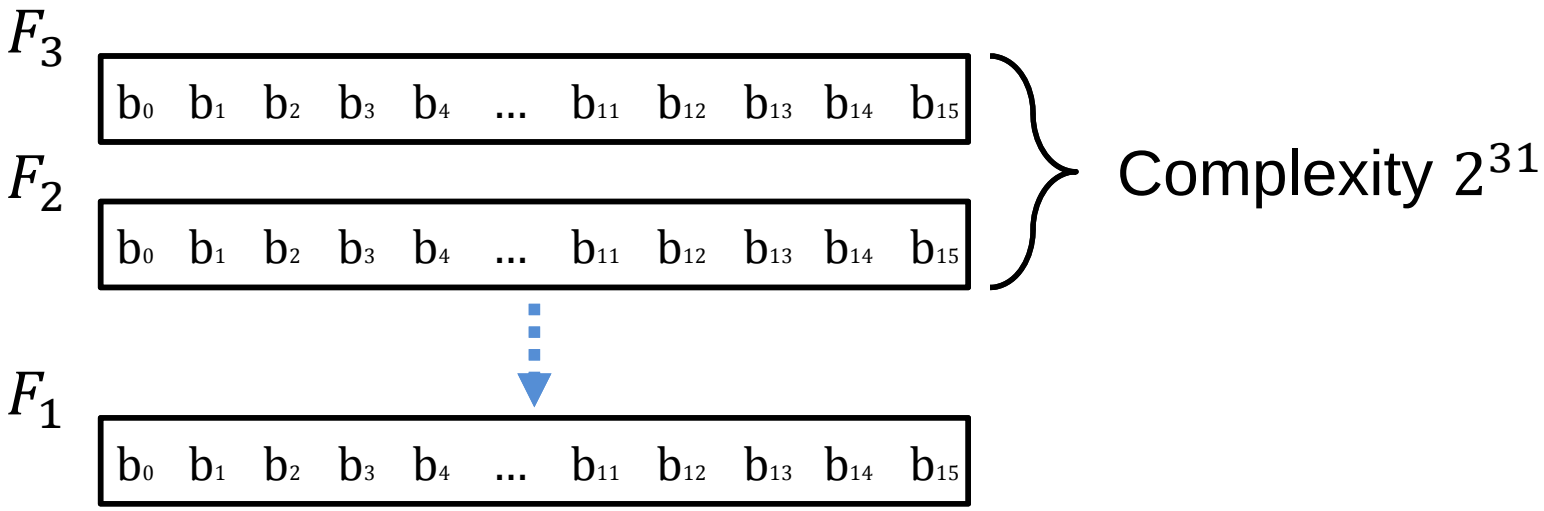
A_x	A_y	B_x	B_y	F_3
0	0	0	0	b_0
0	0	0	1	b_1
0	0	1	0	b_2
0	0	1	1	b_3
0	1	0	0	b_4
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.
1	0	1	1	b_{11}
1	1	0	0	b_{12}
1	1	0	1	b_{13}
1	1	1	0	b_{14}
1	1	1	1	b_{15}

Exhaustive Search Approach



A_x	A_y	B_x	B_y	F_3
0	0	0	0	b_0
0	0	0	1	b_1
0	0	1	0	b_2
0	0	1	1	b_3
0	1	0	0	b_4
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.
1	0	1	1	b_{11}
1	1	0	0	b_{12}
1	1	0	1	b_{13}
1	1	1	0	b_{14}
1	1	1	1	b_{15}

Exhaustive Search Approach



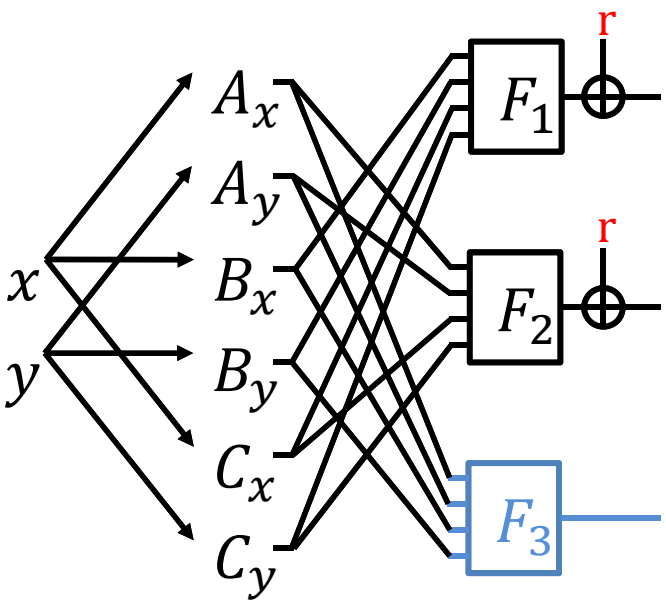
Exhaustive Search Approach

	$F_1(B_x, B_y, C_x, C_y)$	$F_2(A_x, A_y, C_x, C_y)$	$F_3(A_x, A_y, B_x, B_y)$
#1	0000001101010110	1001101011000000	**0111010000101110**
#2	0000001101010110	**1101111010000100**	0011000001101010
#3	0000001101011001	1101111010001011	0011000001101010
#4	0000001110101001	1101111001111011	0011000001101010
#5	0000110010101001	1101000101111011	0011000001101010
#6	**0001110110111000**	1101000101111011	0011000001100101
#7	0011111110011010	1101000101111011	0011000010010101
#8	0111101111011110	1101000101111011	0011111110010101

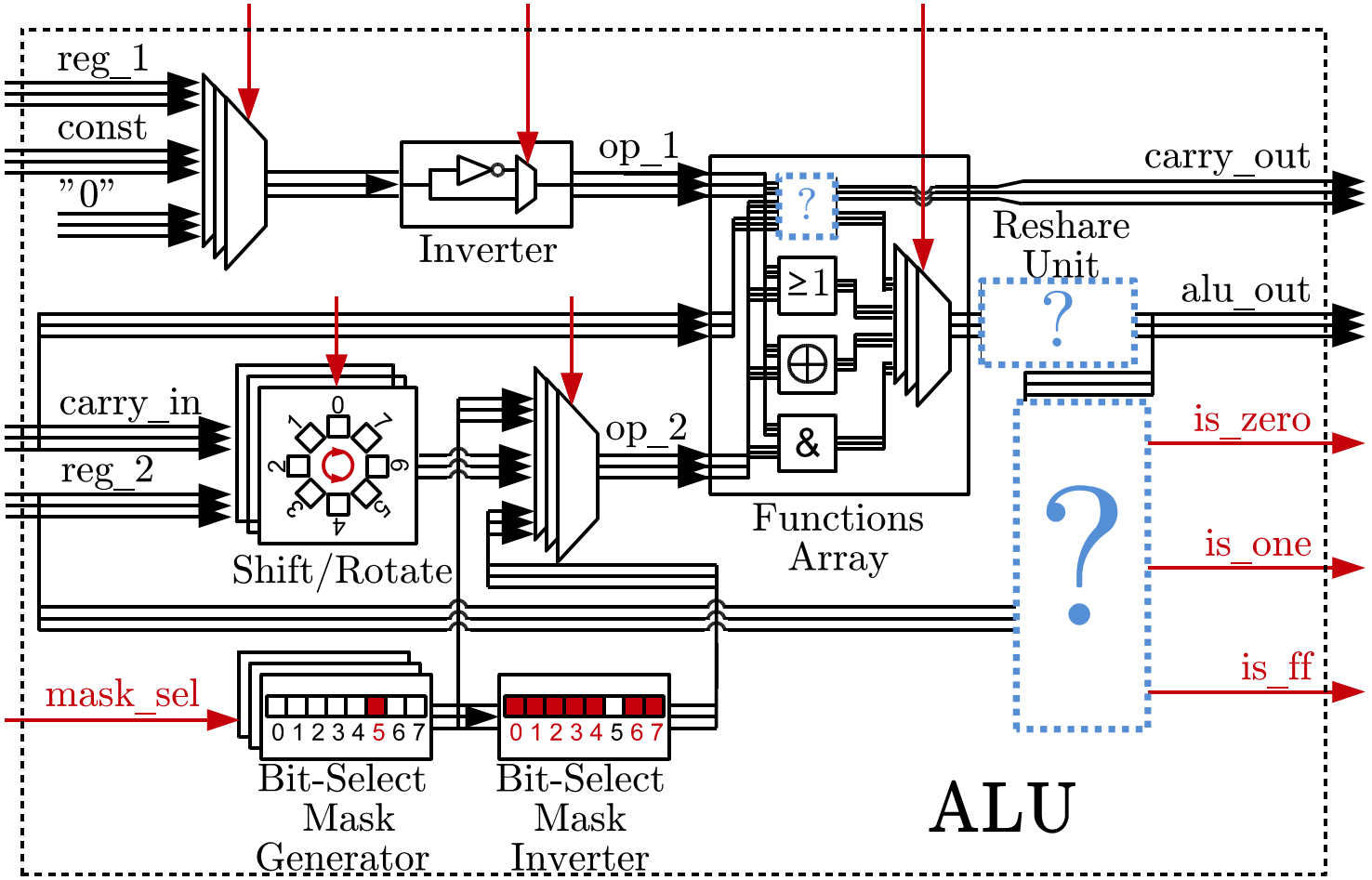
Exhaustive Search Approach

	$F_1(B_x, B_y, C_x, C_y)$	$F_2(A_x, A_y, C_x, C_y)$	$F_3(A_x, A_y, B_x, B_y)$
#1	0000001101010110	1001101011000000	0111010000101110
#2	0000001101010110	1101111010000100	0011000001101010
#3	0000001101011001	1101111010001011	0011000001101010
#4	0000001110101001	1101111001111011	0011000001101010
#5	0000110010101001	1101000101111011	0011000001101010
#6	0001110110111000	1101000101111011	0011000001100101
#7	0011111110011010	1101000101111011	0011000010010101
#8	0111101111011110	1101000101111011	0011111110010101

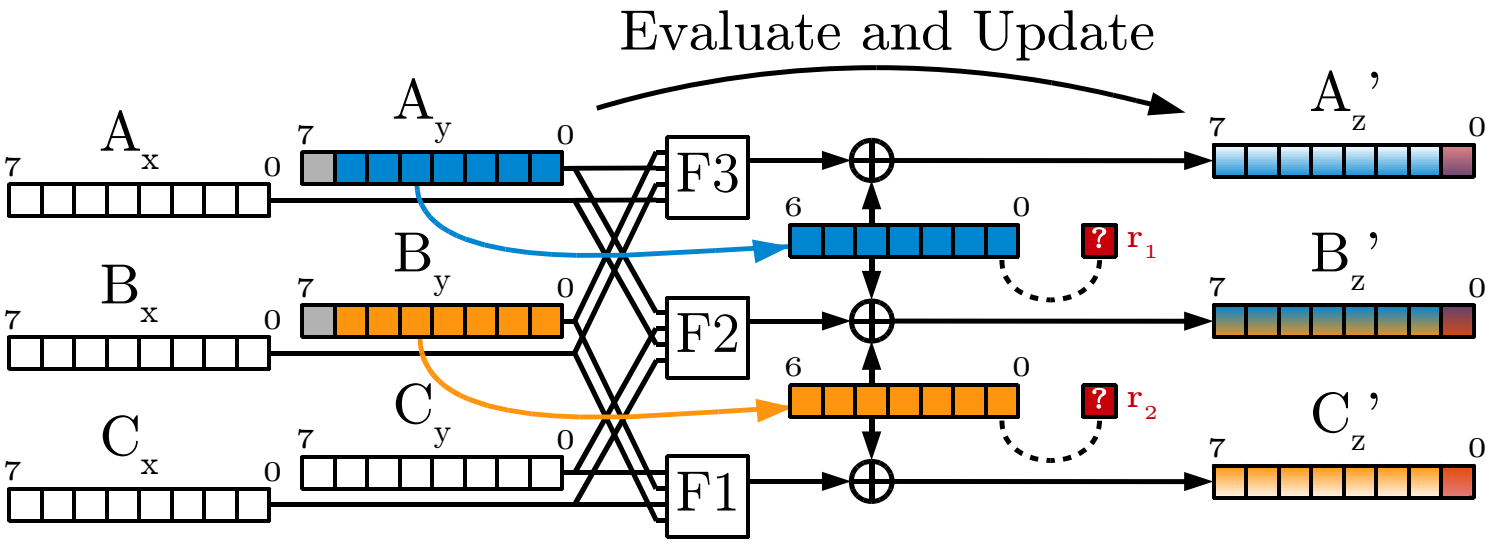
Exhaustive Search Approach



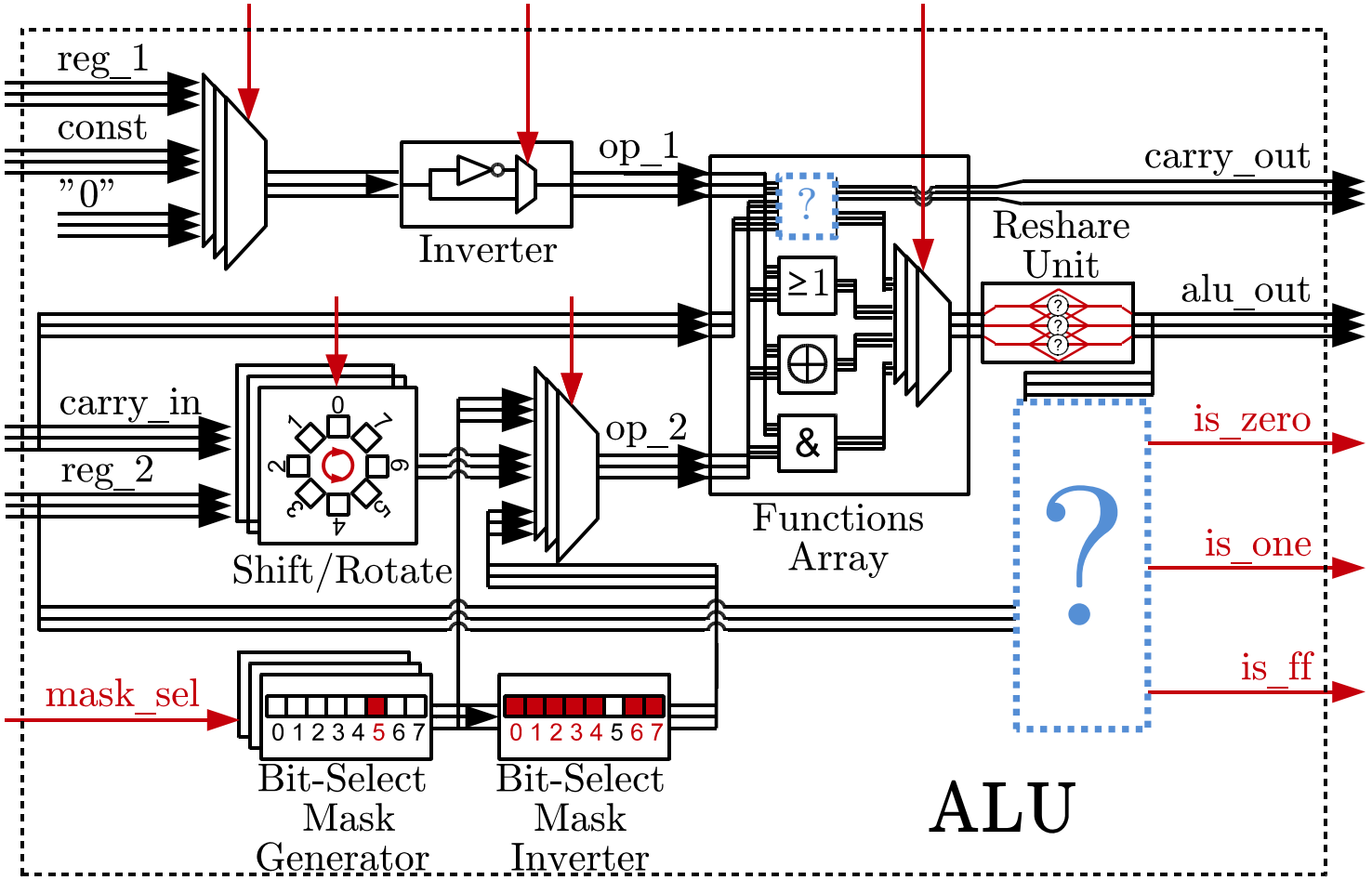
Boolean Operations



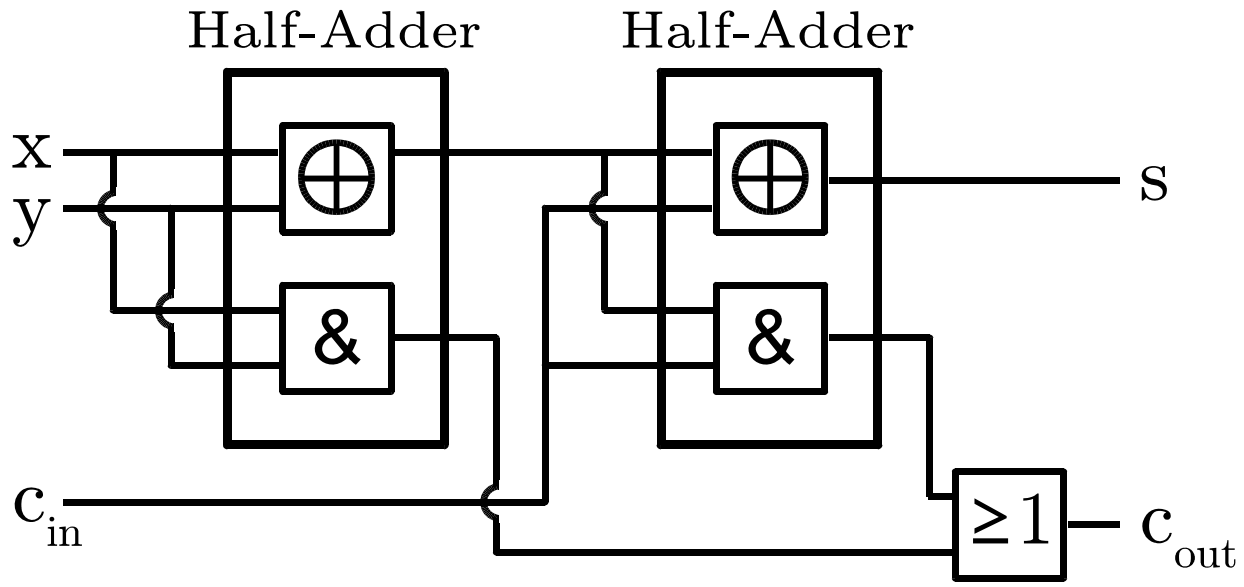
How to use the random bits more efficiently



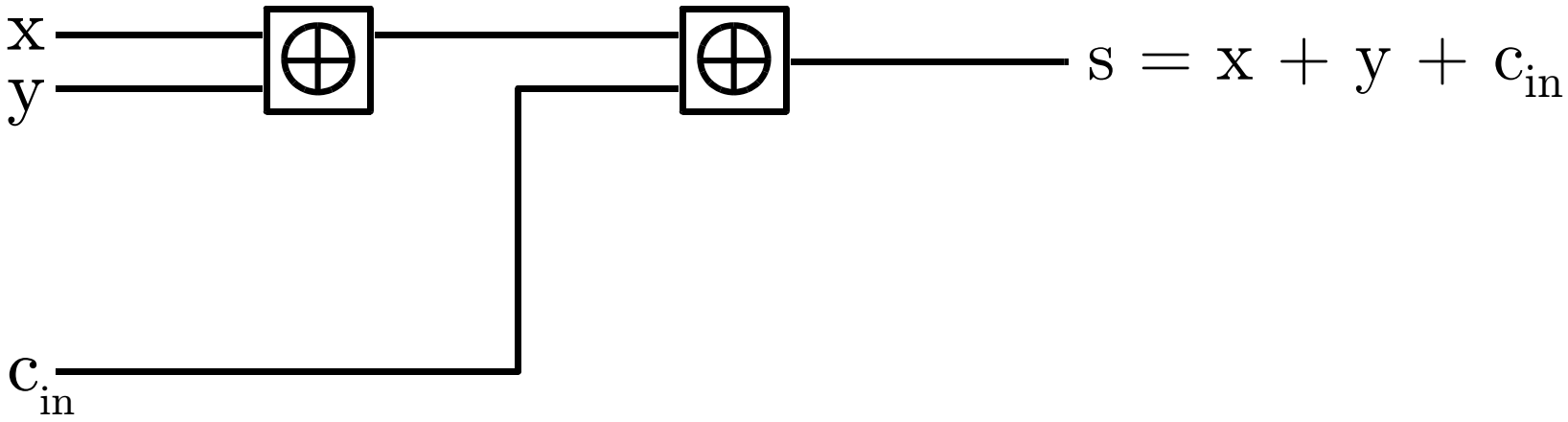
Reshare Unit



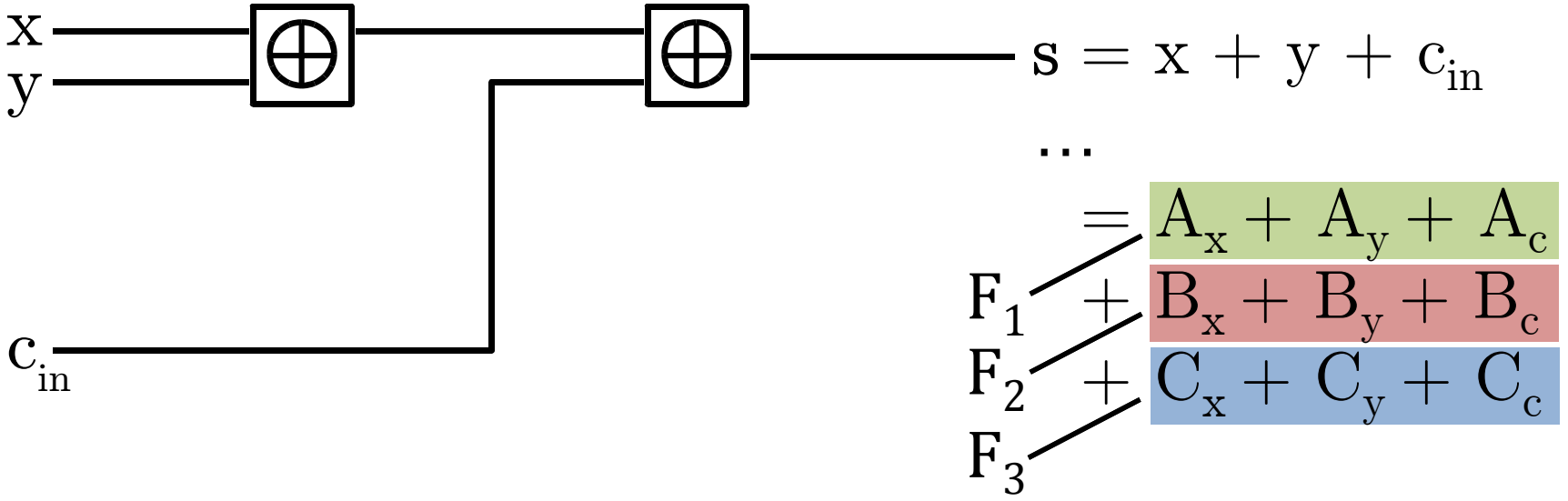
Sharing of the Adder



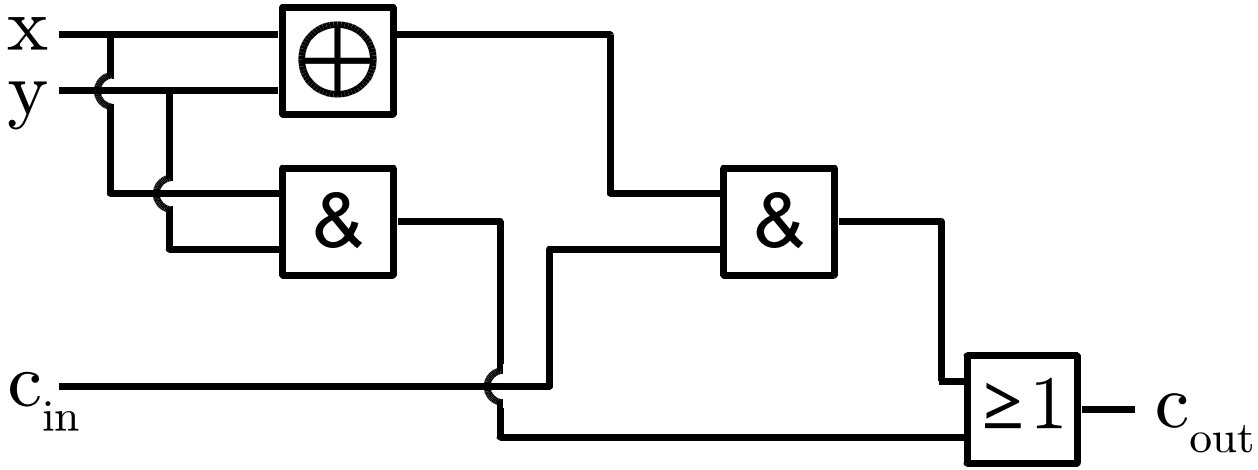
Sharing of the Adder – Sum Bit



Sharing of the Adder – Sum Bit



Sharing of the Adder – Carry Bit



Sharing of the Adder – Carry Bit

$$c_{out} = (xy) \mid (x + y + c)$$

...

Sharing of the Adder – Carry Bit

$$c_{out} = (xy) \mid (x + y + c)$$

$$\ldots$$

$$= B_xB_y + B_xC_y + B_yC_x + B_xB_c + B_xC_c + B_cC_x + B_cB_y + B_yC_c + B_cC_y \quad F_1$$

$$+ C_xC_y + A_xC_y + A_yC_x + C_xC_c + A_xC_c + A_cC_x + C_cC_y + A_yC_c + A_cC_y \quad F_2$$

$$+ A_xA_y + A_xB_y + A_yB_x + A_xA_c + A_xB_c + A_cB_x + A_cA_y + A_yB_c + A_cB_y \quad F_3$$

Sharing of the Adder – Carry Bit

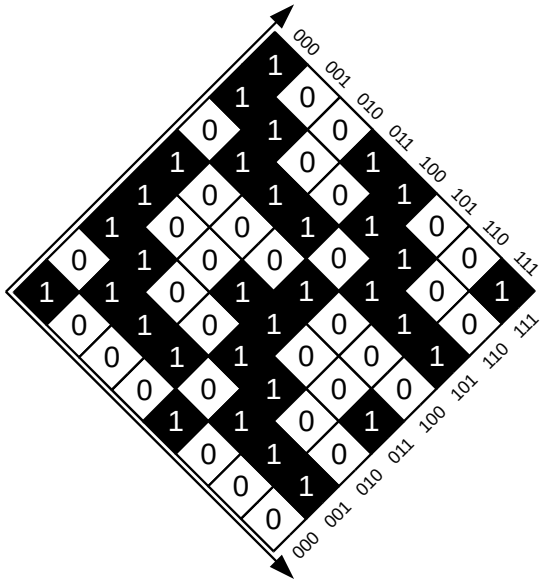
$$c_{out} = (xy) \mid (x + y + c)$$

$$\ldots$$

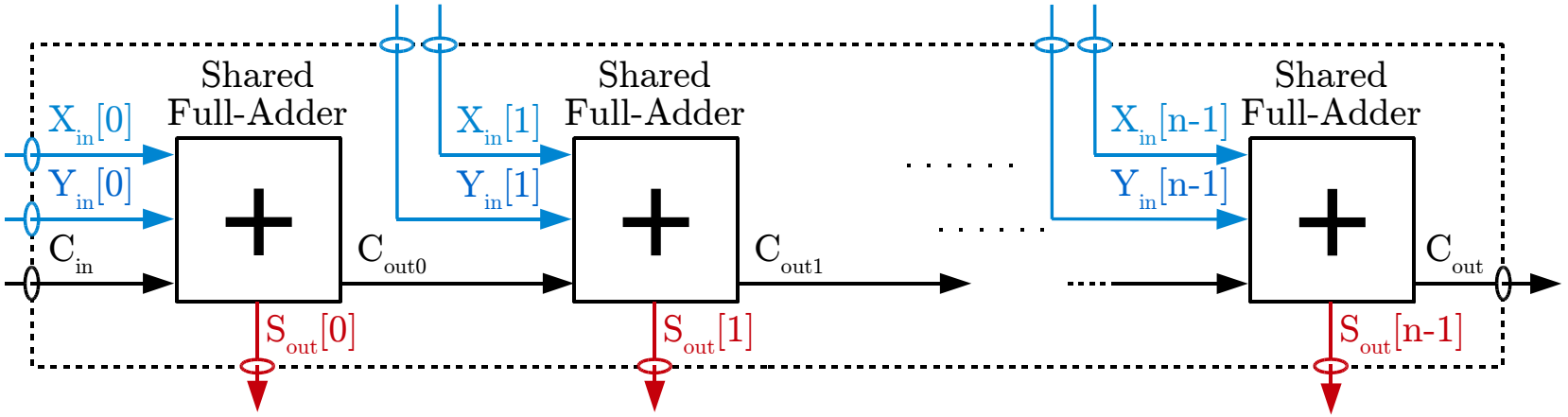
$$= B_xB_y + B_xC_y + B_yC_x + B_xB_c + B_xC_c + B_cC_x + B_cB_y + B_yC_c + B_cC_y \quad F_1$$

$$+ C_xC_y + A_xC_y + A_yC_x + C_xC_c + A_xC_c + A_cC_x + C_cC_y + A_yC_c + A_cC_y \quad F_2$$

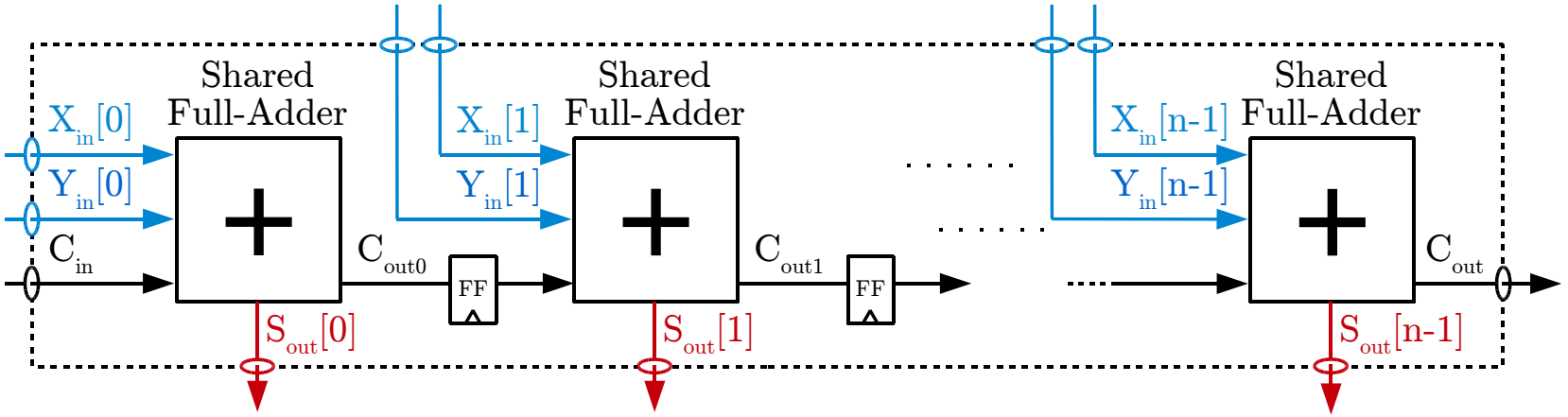
$$+ A_xA_y + A_xB_y + A_yB_x + A_xA_c + A_xB_c + A_cB_x + A_cA_y + A_yB_c + A_cB_y \quad F_3$$



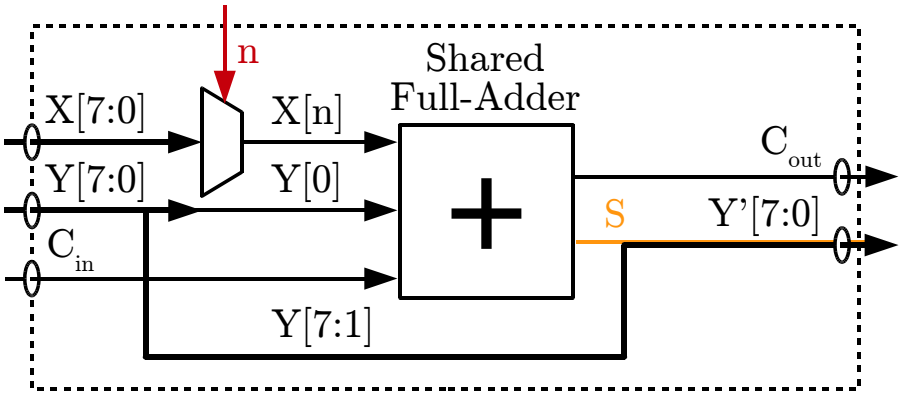
From 1-Bit Adder to n-bit Adder



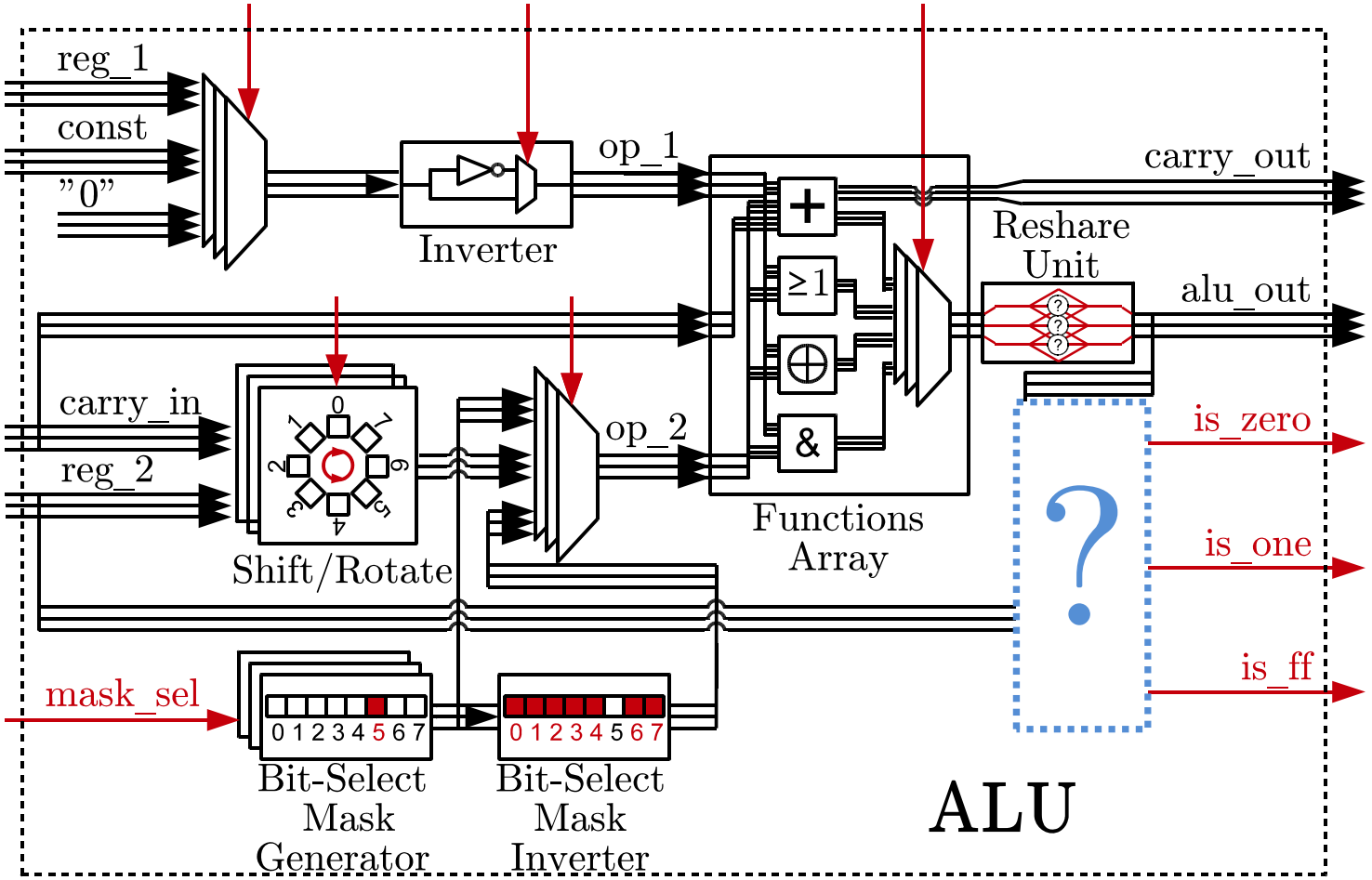
From 1-Bit Adder to n-bit Adder



From 1-Bit Adder to n-bit Adder

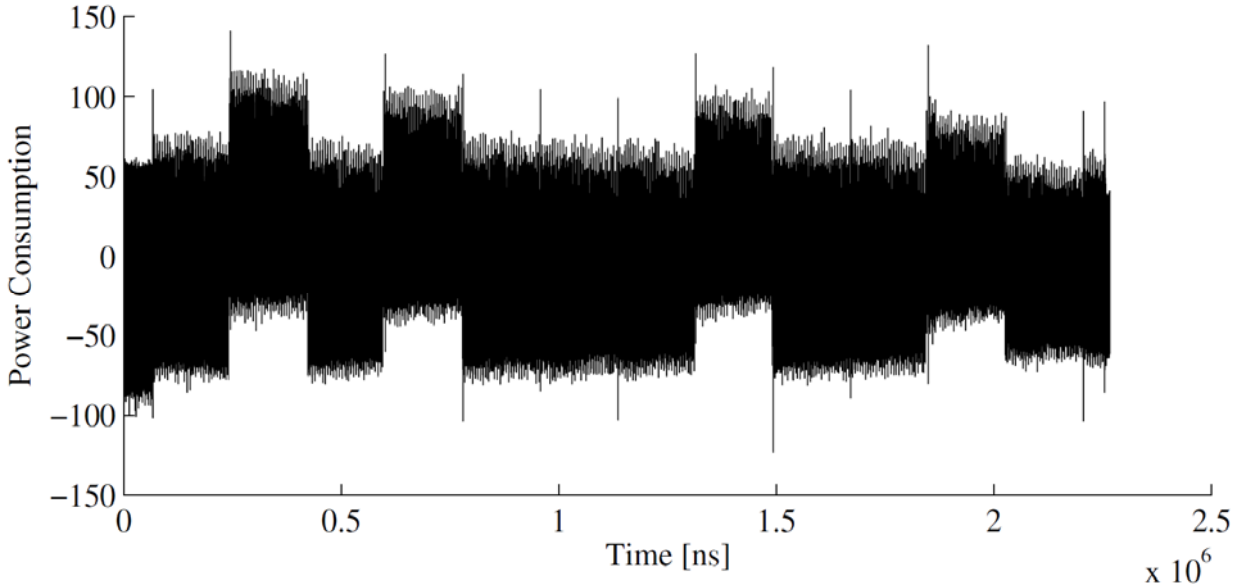


Functions Array

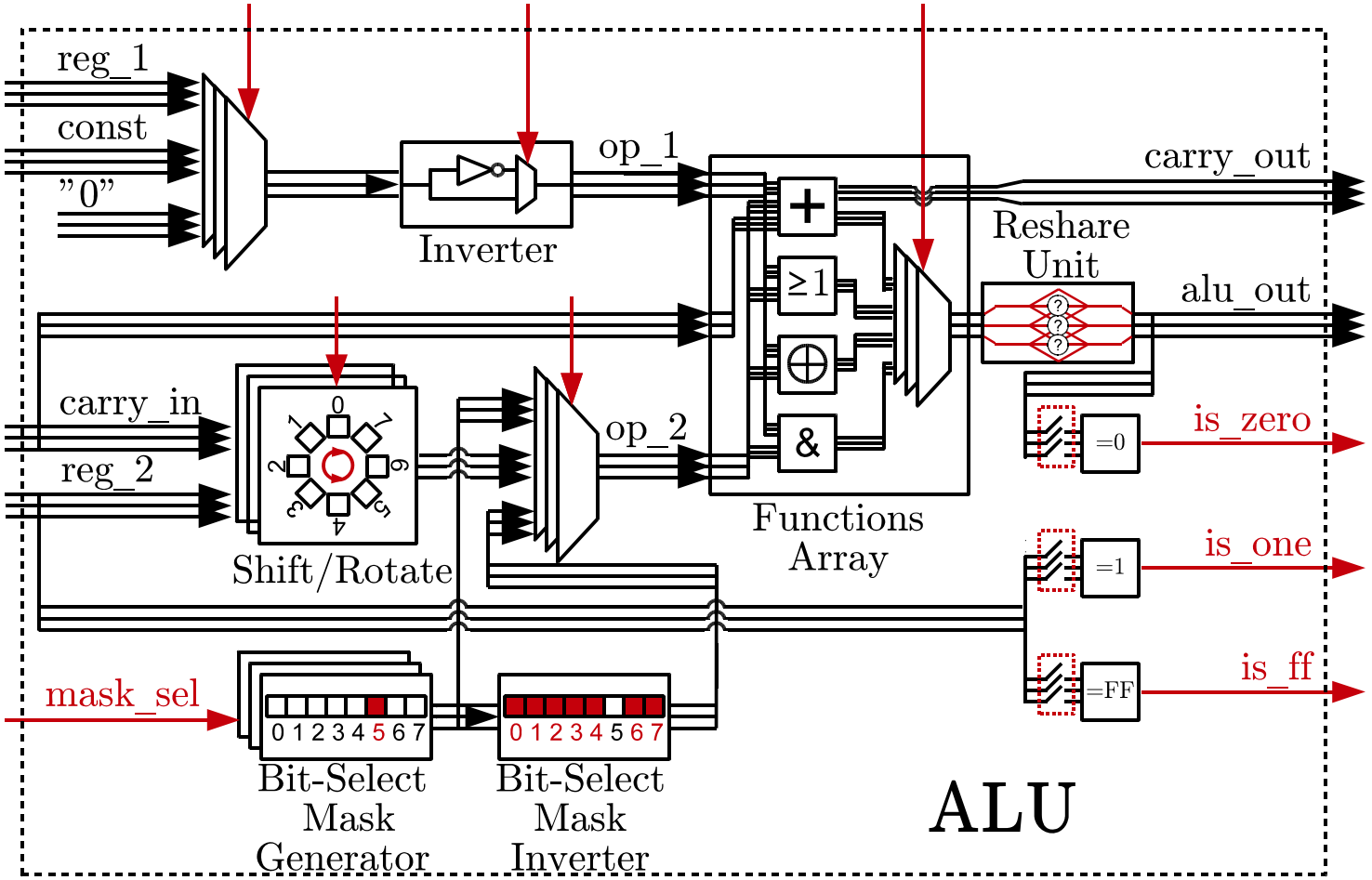


Sharing of Control Signals

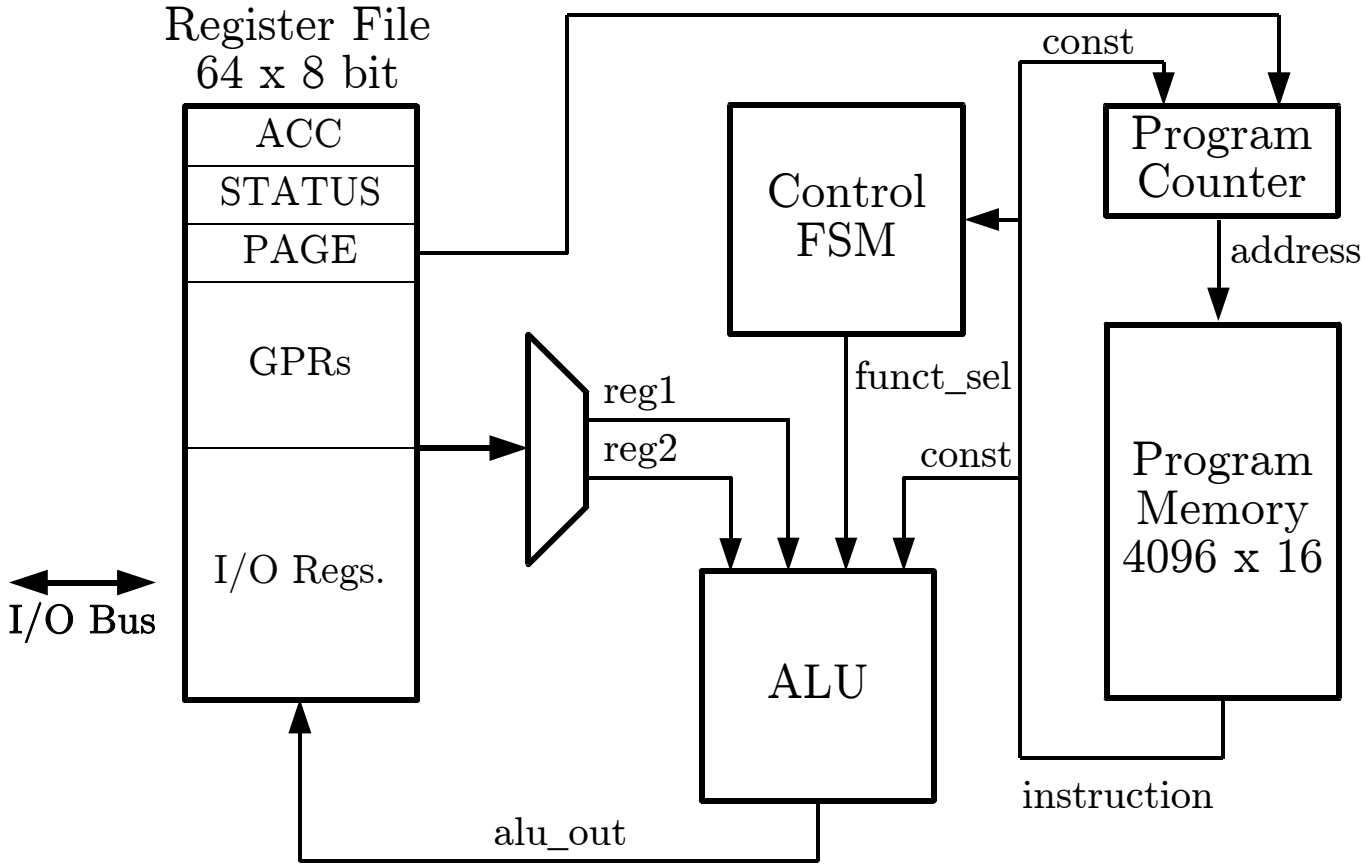
- Used by the control FSM for conditional branches
- Branches are visible in the power traces (SPA)
- Sharing does not help here



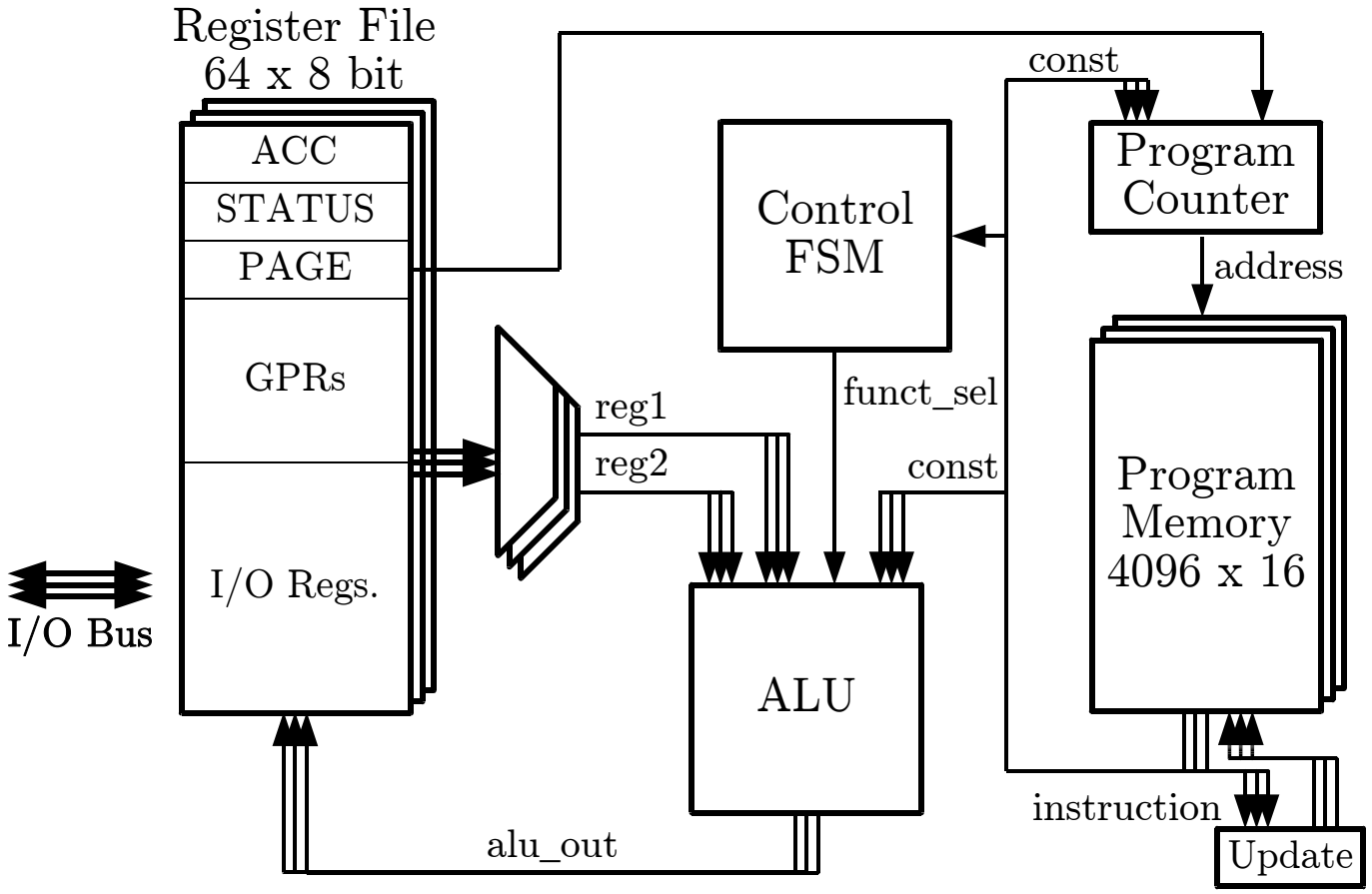
The Arithmetic Logic Unit



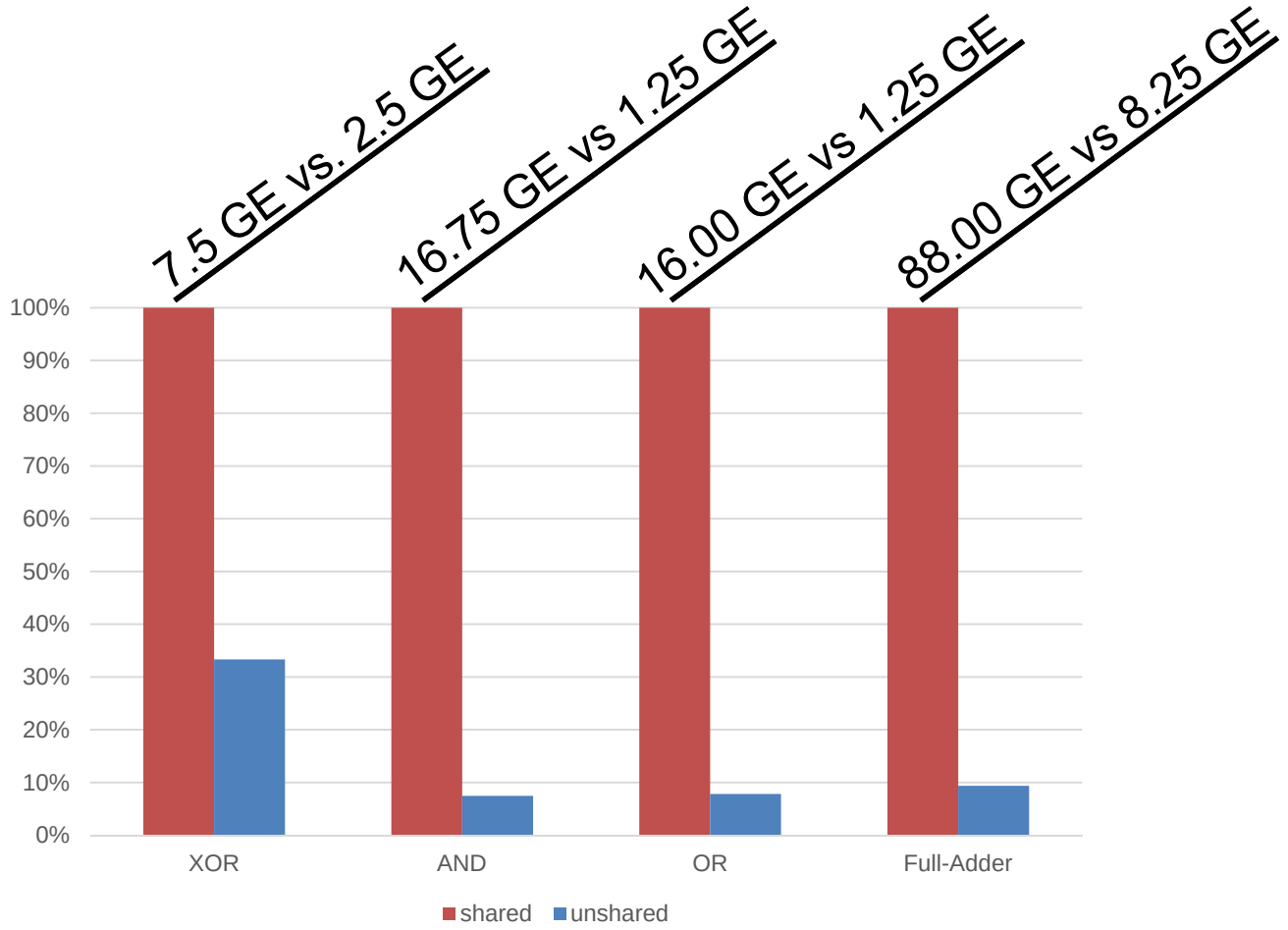
µC Case Study

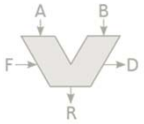


µC Case Study

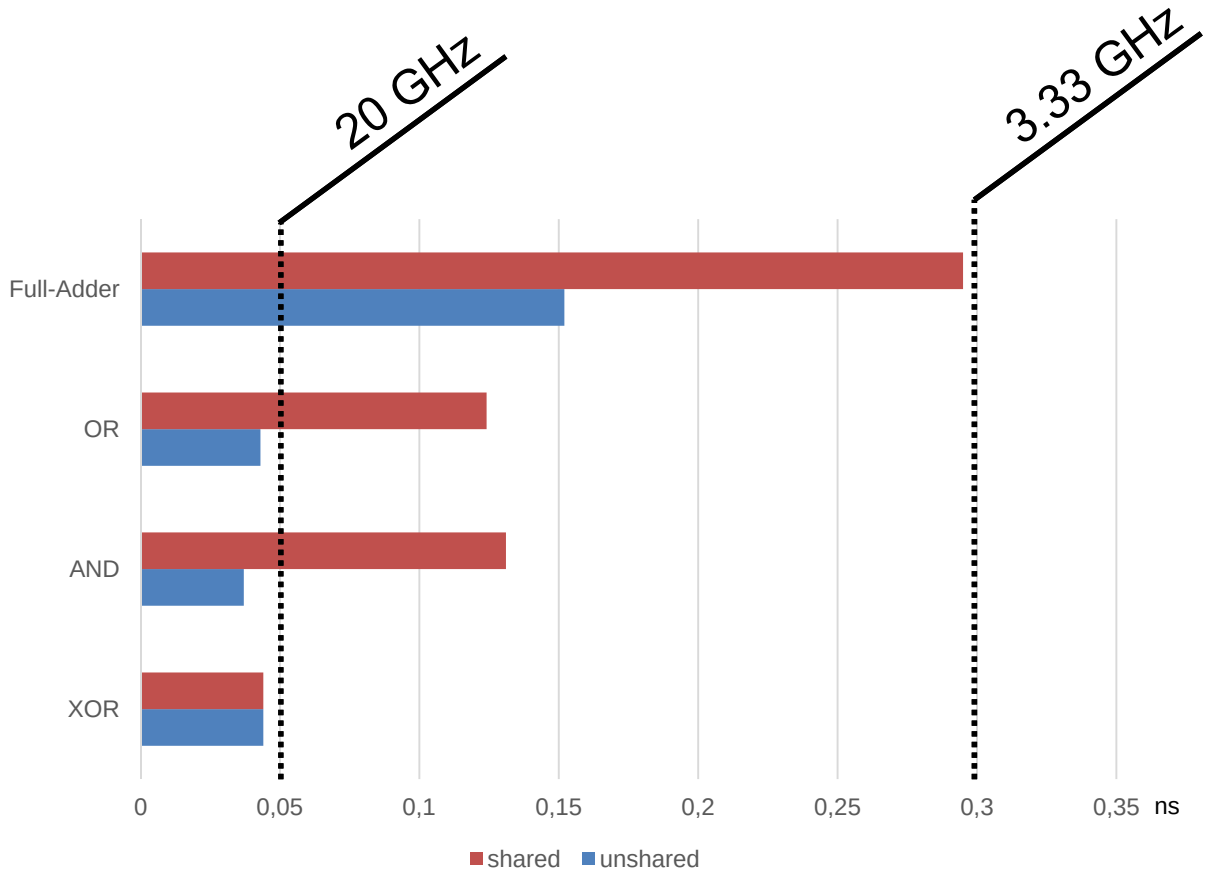


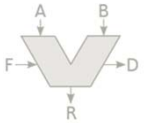
Results – Area Comparison on Gate Level



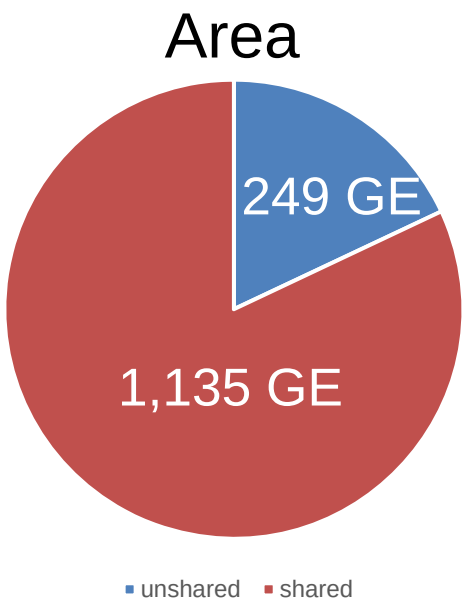


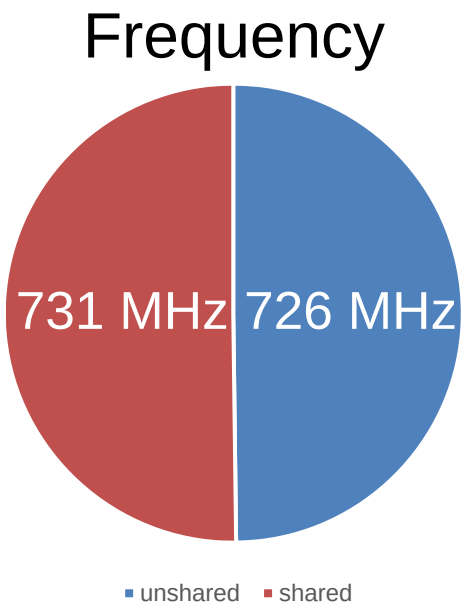
Results – Delay Comparison on Gate Level

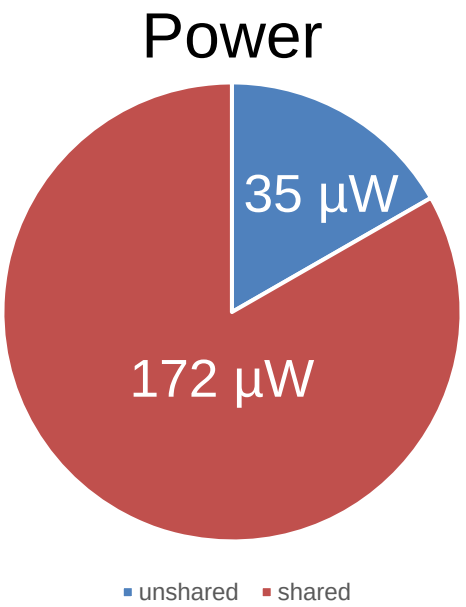


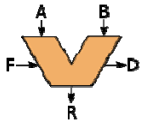


Results – ALU Level Comparison

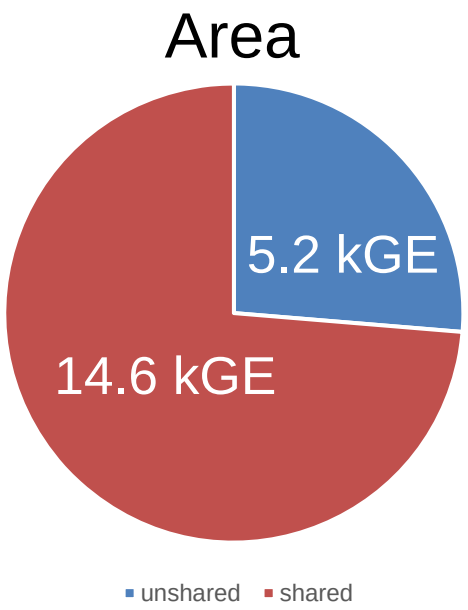


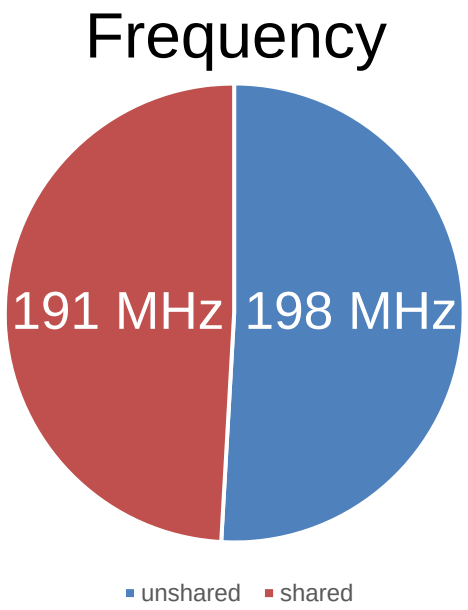


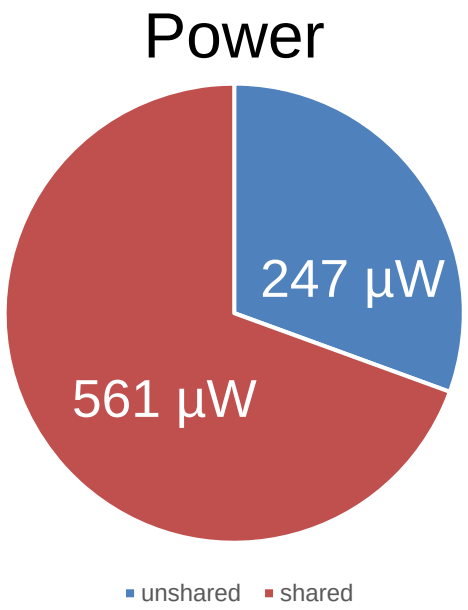


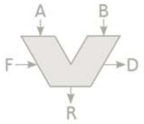


Results – Microcontroller Level Comparison









Conclusions

- Protect ALU against first-order passive attacks
  - Strong attacker
  - First step towards secure µC

Conclusions

- Protect ALU against first-order passive attacks
  - Strong attacker
  - First step towards secure µC
- ALU requires only one random bit per cycle
- Area overhead ~ 2.8 for µC (AES Bilgin et al. ~ 4.6)

Conclusions

- Protect ALU against first-order passive attacks
  - Strong attacker
  - First step towards secure µC
- ALU requires only one random bit per cycle
- Area overhead ~ 2.8 for µC (AES Bilgin et al. ~ 4.6)
- Some things cannot be protected: branches, table lookups, etc. → software designer need to be aware

Sharing is Caring

On the Protection of Arithmetic Logic Units against Passive Physical Attacks

Hannes Groß

Institute for Applied Information Processing and Communications
Graz University of Technology
hannes.gross@iaik.tugraz.at

Joint Distribution of Sum and Carry Bits