# AI ASIC: Design and Practice (ADaP)

# Fall 2024

# Digital Arithmetic & Circuits

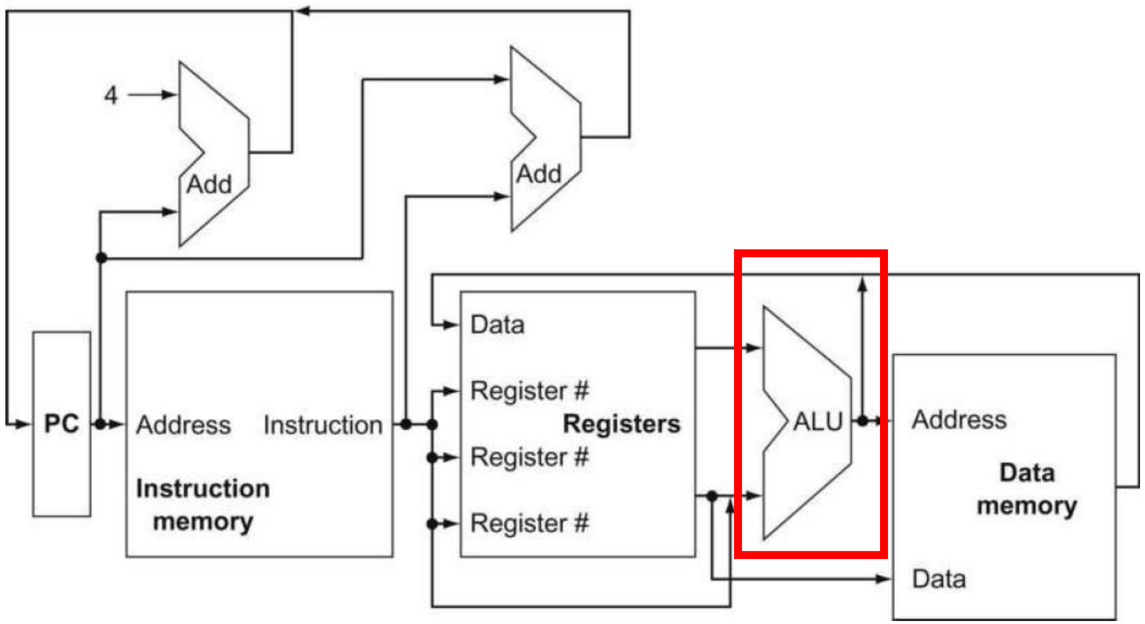
燕博南

北京大学 PEKING UNIVERSITY

# Outline

- Number Systems
  - Integer
  - Fixed-Point
  - Floating-Point
- Arithmetic
- Circuits & Implementation

# Why We Need to Introduce Arithmetic




- Arithmetic Logic Unit (ALU): heart of von Neumann architecture
- Deal with various precisions: decimals, fractions, integers, …

# Part 1

Unsigned Integers

# Number System of Digital Computers

北京大学 PEKING UNIVERSITY

Digital Numbers

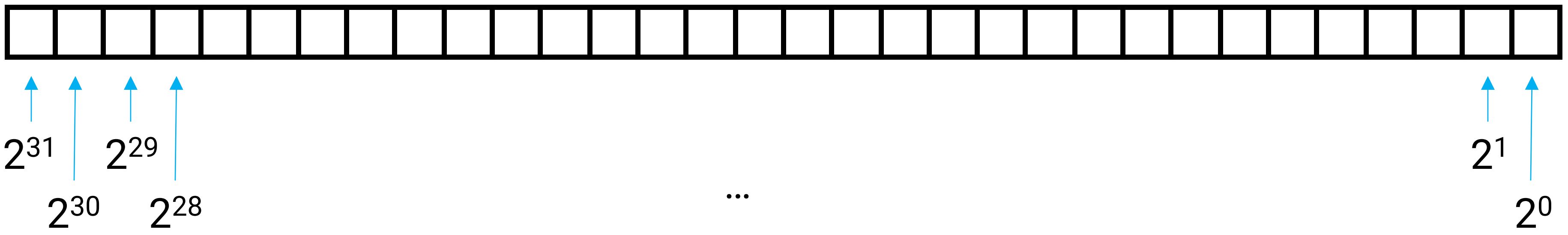
- Integer
  - Unsigned
  - Signed
- Fixed-Point
  - Unsigned
  - Signed
- Floating-Point
  - FP16
  - FP32
  - FP64
  - BF16
  - TF32
  - POSIT
- Still evolving

# Unsigned Integer



- Unsigned INT32

Significance:




- INT16, INT8, ...
- Example:

  32’d7 (=32’h0000_0007)
  8’b1100_1101 (=8’hCD)

Recommend tool:
programmer’s calculator

# Unsigned Integer Arithmetic – Add & Subtract



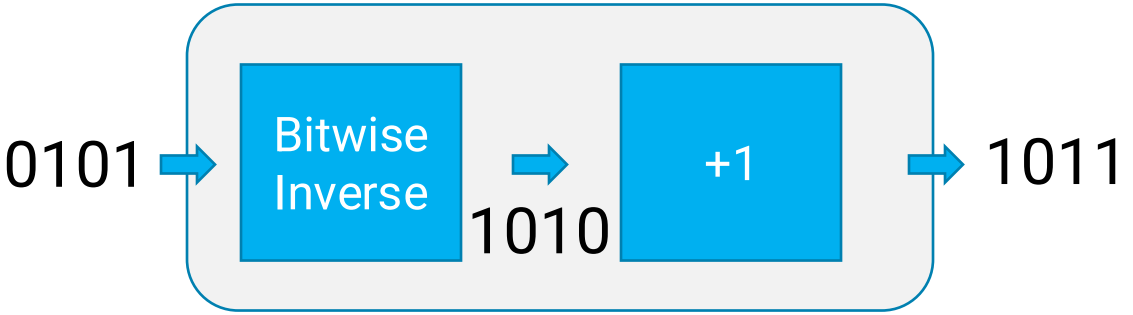
- INT4 as example:

Add: 4’d7 + 4’d5 = 4’d12

```
     0111
+    0101
---------
     1100
```

Subtract: 4’d12 - 4’d5 = 4’d7

Two’s Complement




```
     0111            1100
-    0101     →  +   1011
---------        ---------
                     0111
```

# Unsigned Integer Arithmetic – Multiply & Divide



- INT4 as example:

Multiply: 4’d3 * 4’d5 = 4’d15

```
         0011
   *     0101
   -----------
         0011
        0000
       0011
  +   0000
   -----------
         1111
```

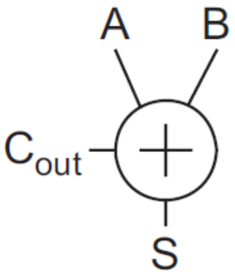
Divide: 4’d15 / 4’d3 = 4’d5

```
        101
     ------
  11 | 1111
     - 11
     ------
        01
        00
     ------
         11
         11
     ------
          0
```

# Unsigned Integer Circuits – Adder



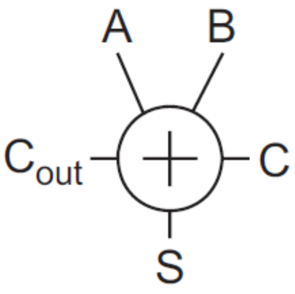
## Half-Adder




| $A$ | $B$ | $C_{out}$ | $S$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 |

$$S = A \oplus B$$
$$C_{out} = A \cdot B$$

## Full-Adder



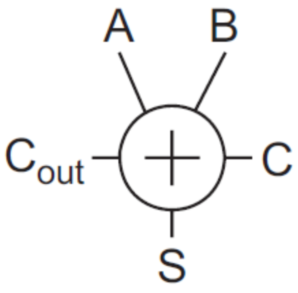

| $A$ | $B$ | $C$ | $G$ | $P$ | $K$ | $C_{out}$ | $S$ |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| | | 1 | | | | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 |
| | | 1 | | | | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 |
| | | 1 | | | | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 | 0 | 1 | 0 |
| | | 1 | | | | 1 | 1 |

$$S = A\bar{B}\bar{C} + \bar{A}B\bar{C} + \bar{A}\bar{B}C + ABC$$
$$= (A \oplus B) \oplus C = P \oplus C$$
$$C_{out} = AB + AC + BC$$
$$= AB + C(A+B)$$
$$= \overline{\bar{A}\bar{B} + \bar{C}(\bar{A}+\bar{B})}$$
$$= \mathrm{MAJ}(A,\ B,\ C)$$

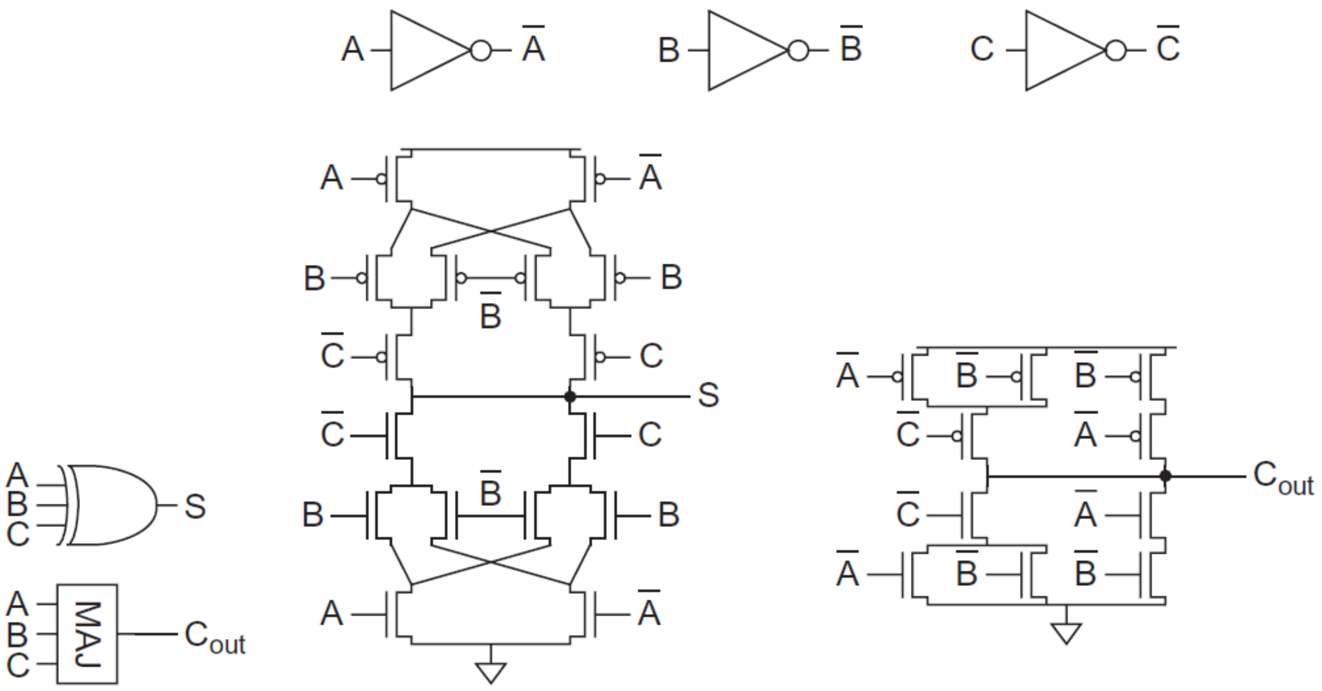
# Unsigned Integer Circuits – Adder (Cont.)



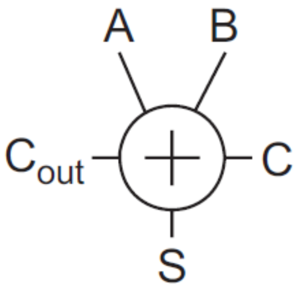





$$S = A\bar{B}\bar{C} + \bar{A}B\bar{C} + \bar{A}\bar{B}C + ABC$$
$$= (A \oplus B) \oplus C = P \oplus C$$
$$C_{out} = AB + AC + BC$$
$$= AB + C(A + B)$$
$$= \overline{\bar{A}\bar{B} + \bar{C}(\bar{A} + \bar{B})}$$
$$= \mathrm{MAJ}(A,\ B,\ C)$$

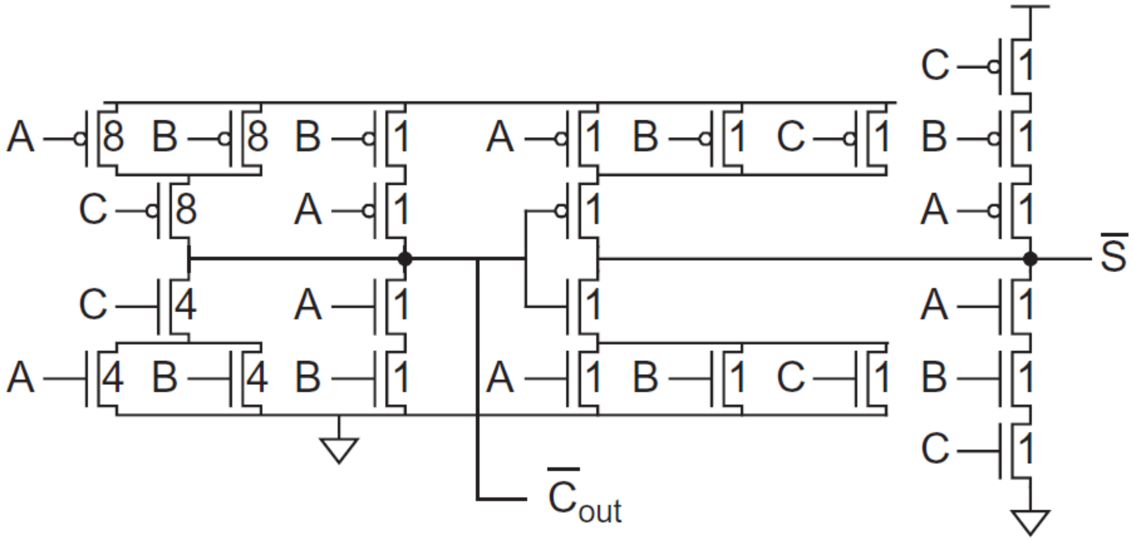


# Unsigned Integer Circuits – Adder (Cont.)




$$S = A\bar{B}\bar{C} + \bar{A}B\bar{C} + \bar{A}\bar{B}C + ABC$$
$$= (A \oplus B) \oplus C = P \oplus C$$
$$C_{out} = AB + AC + BC$$
$$= AB + C(A + B)$$
$$= \overline{\bar{A}\bar{B} + \bar{C}(\bar{A} + \bar{B})}$$
$$= \mathrm{MAJ}(A, B, C)$$

Improved:

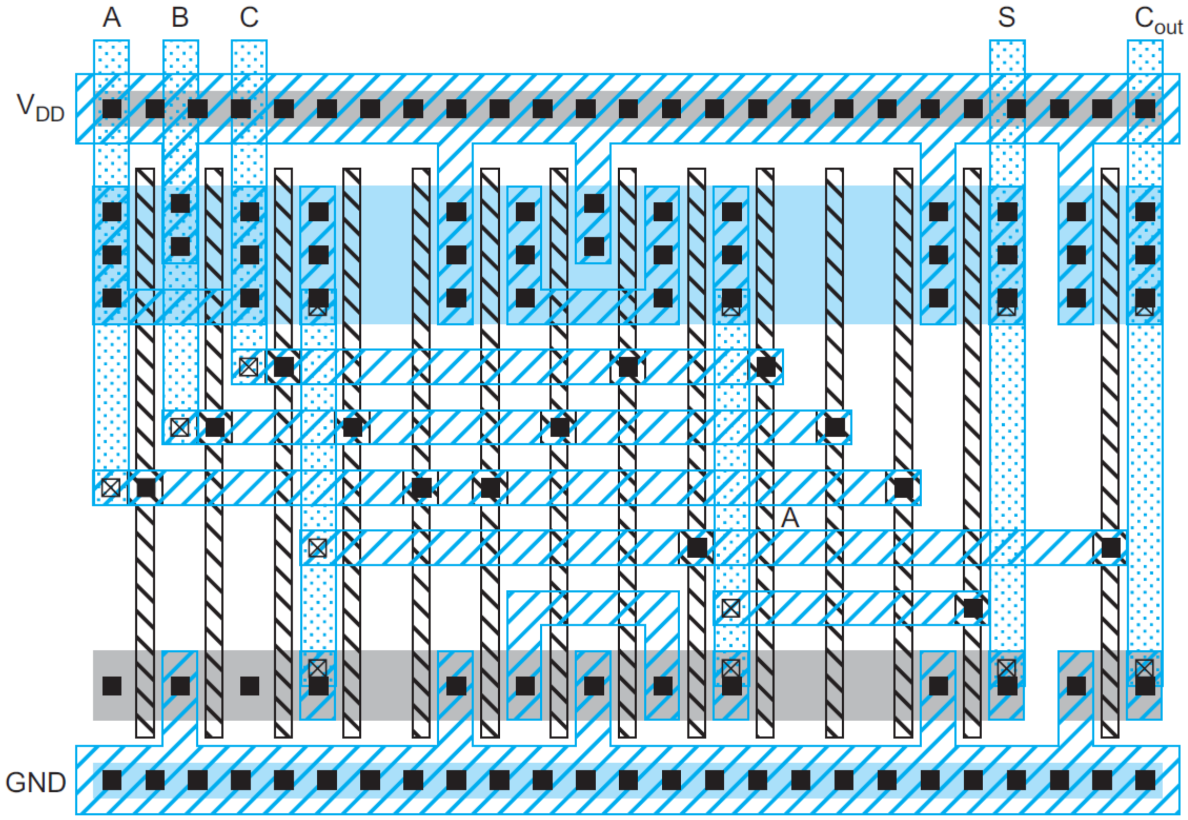



$$S = ABC + (A + B + C)\bar{C}_{out}$$

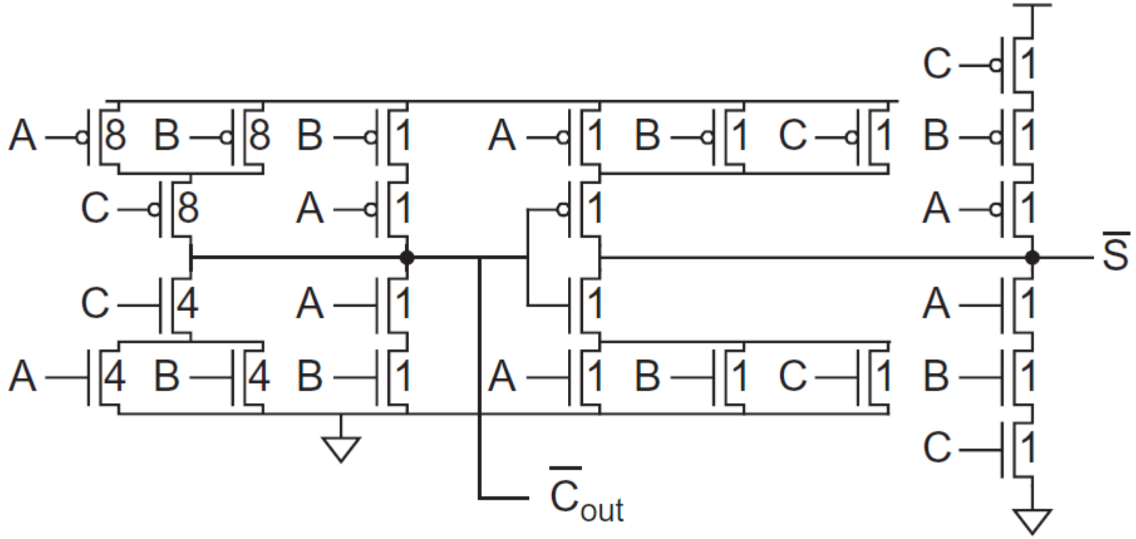
Idea behind:
Reuse Cout compute circuits to obtain both S

# Unsigned Integer Circuits – Adder (Cont.)






Improved:



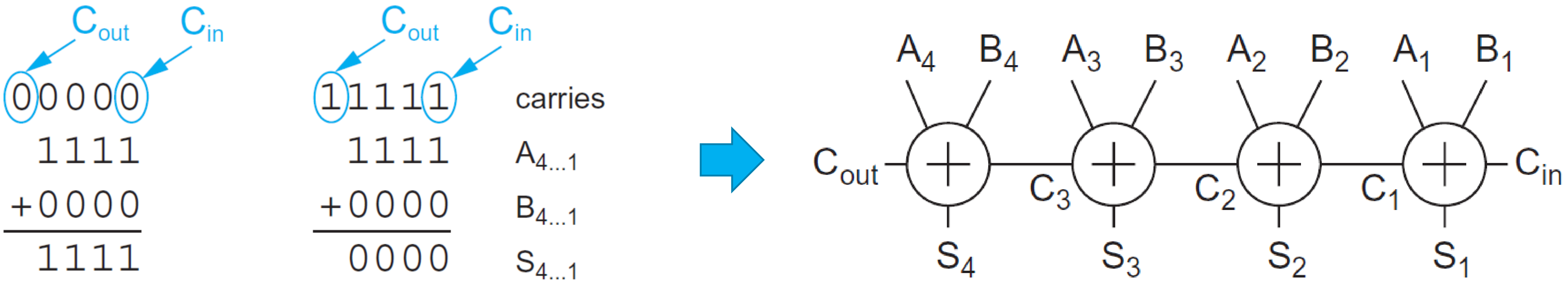

$$S = ABC + (A + B + C)\overline{C}_{out}$$

Idea behind:

Reuse Cout compute circuits to obtain both S

# Adder Family – Carry-Ripple Adder



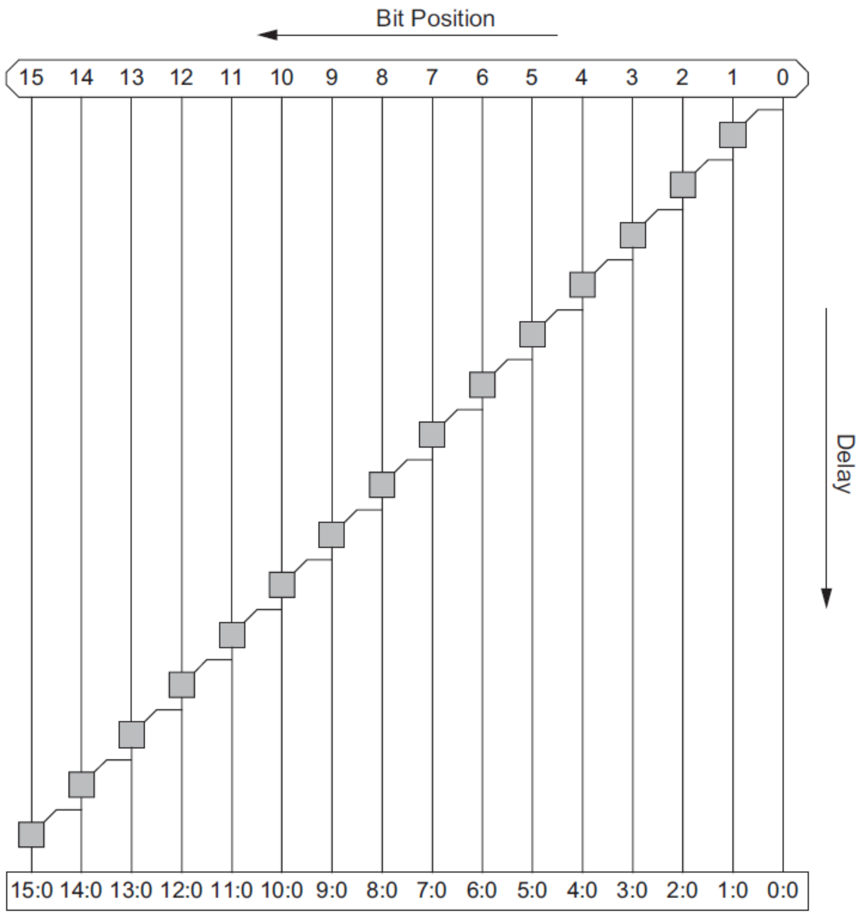


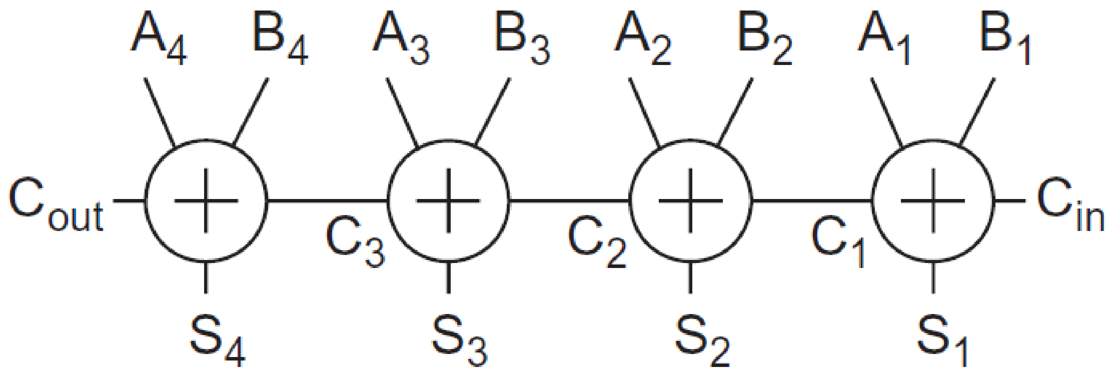

Natural & Intuitive

However, carry propagation path too long

# Adder Family – Carry-Ripple Adder





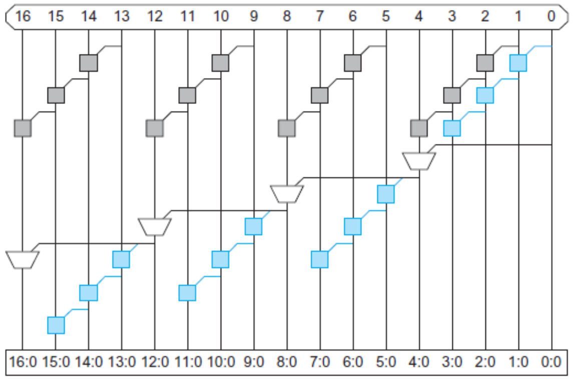



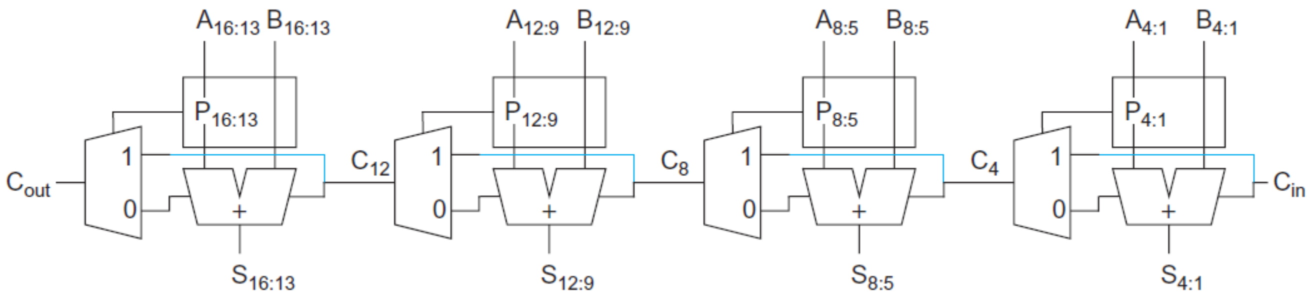

Weste, Neil HE, and David Harris. *CMOS VLSI design: a circuits and systems perspective*. Pearson Education India, 2015.

# Adder Family – Carry-Skip Adder

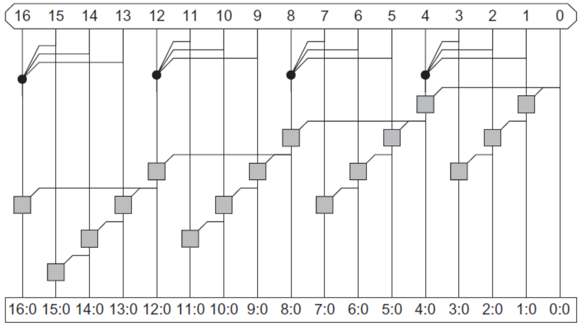
Numbers → Arithmetic → Circuits

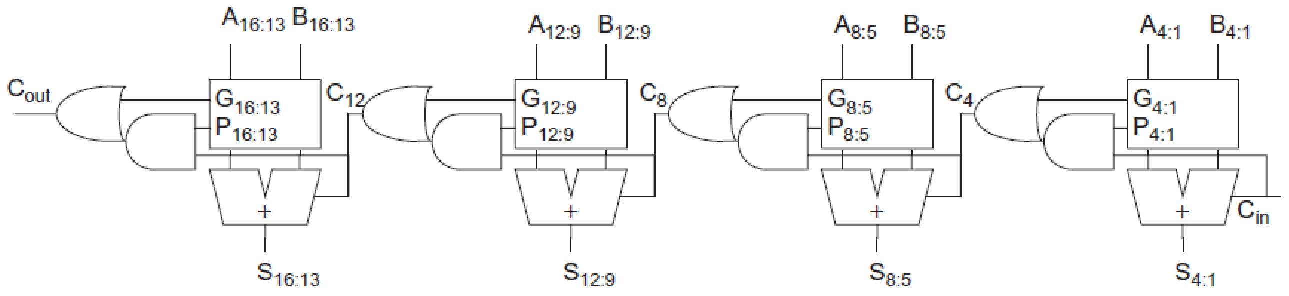
Carry-Skip Adder




Carry-Lookahead Adder



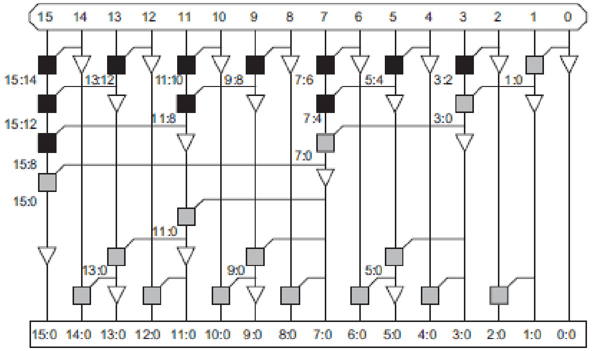




Idea Behind: Group and Divide!

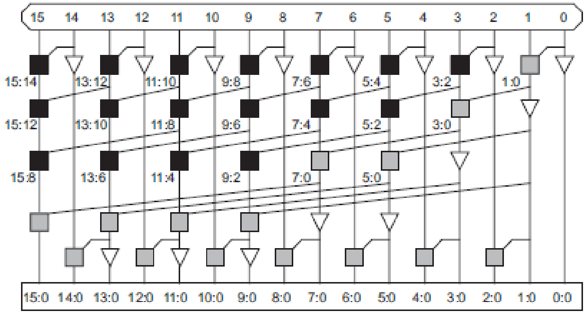
# Adder Family – Big Family!



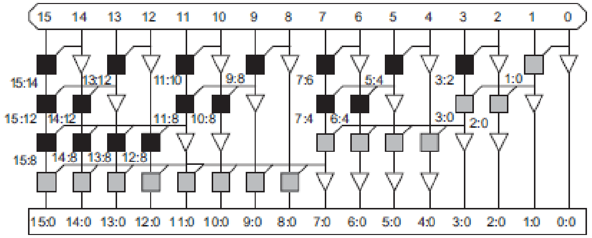
## Tree Adder Family



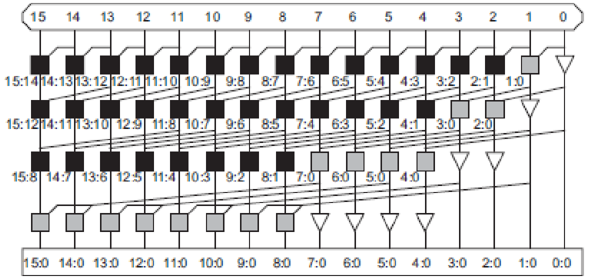
(a) Brent-Kung



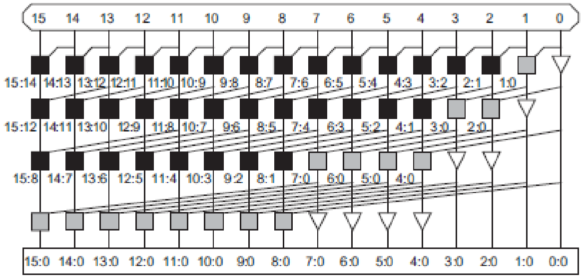
(d) Han-Carlson



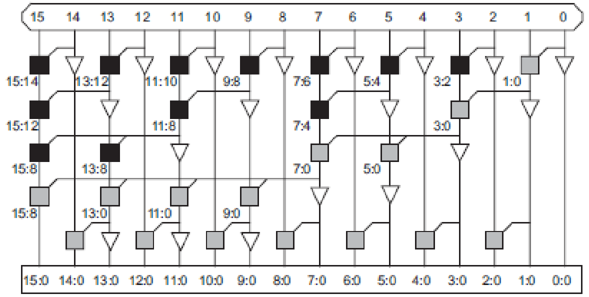
(b) Sklansky



(e) Knowles [2,1,1,1]

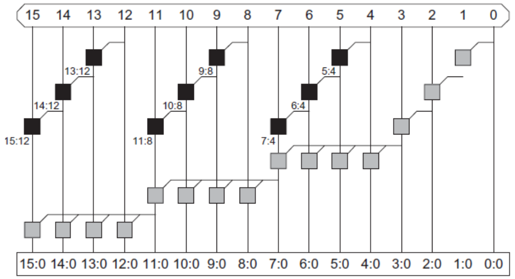


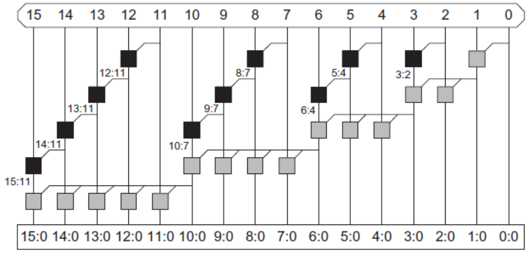
(c) Kogge-Stone



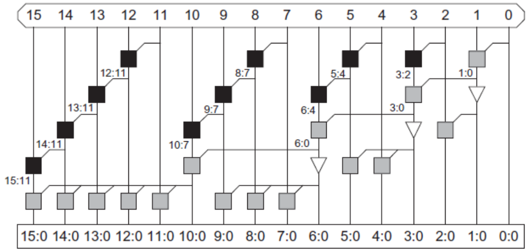
(f) Ladner-Fischer

## Carry-Incremental Adder





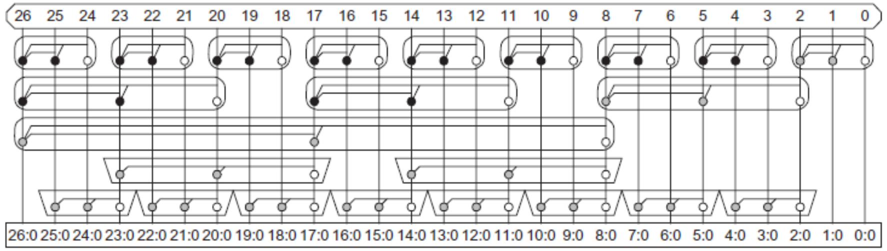
(a)

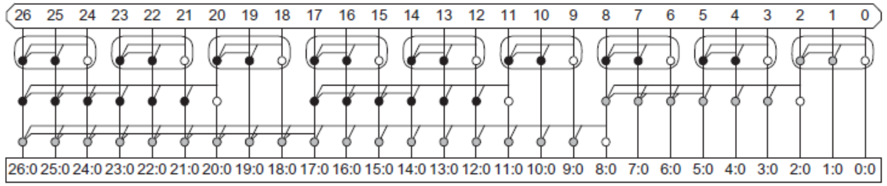


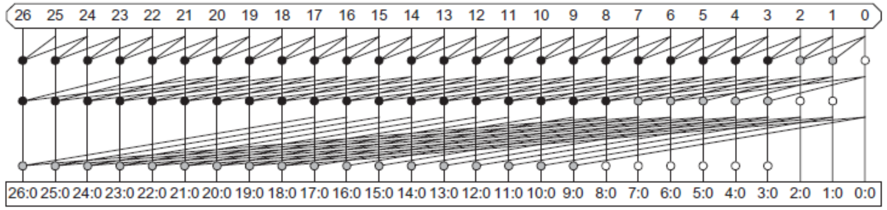
(b)

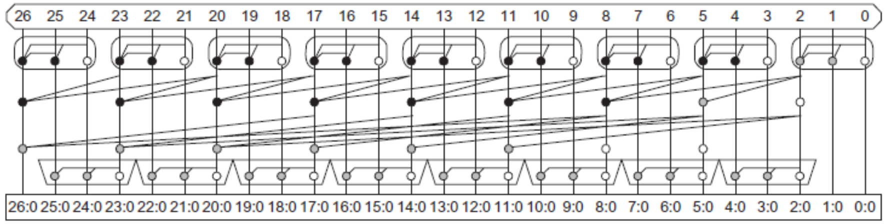
# Adder Family – Big Family! (Cont.)

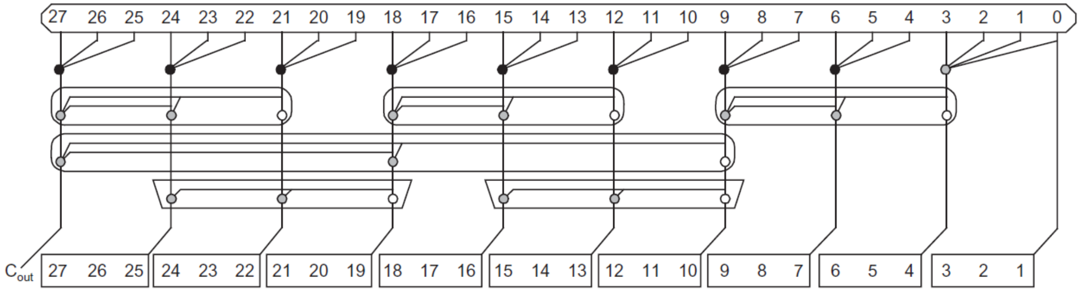
## Sparse Tree Adders

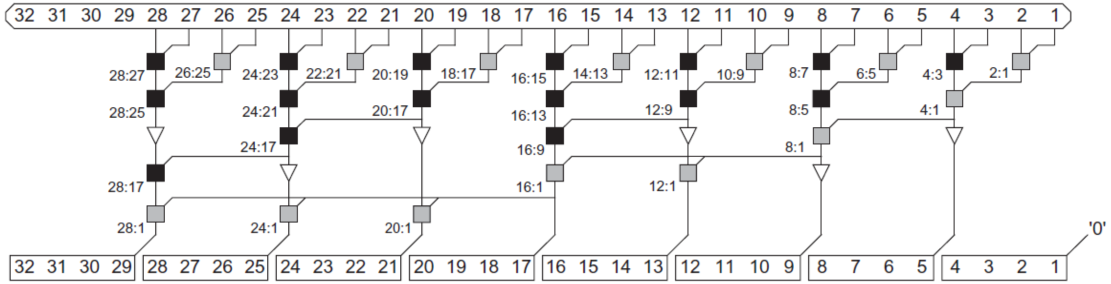

(a) Brent-Kung

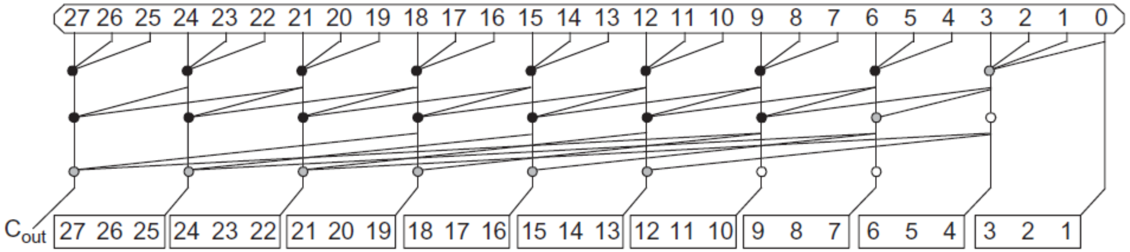

(b) Sklansky


(c) Kogge-Stone


(d) Han-Carlson

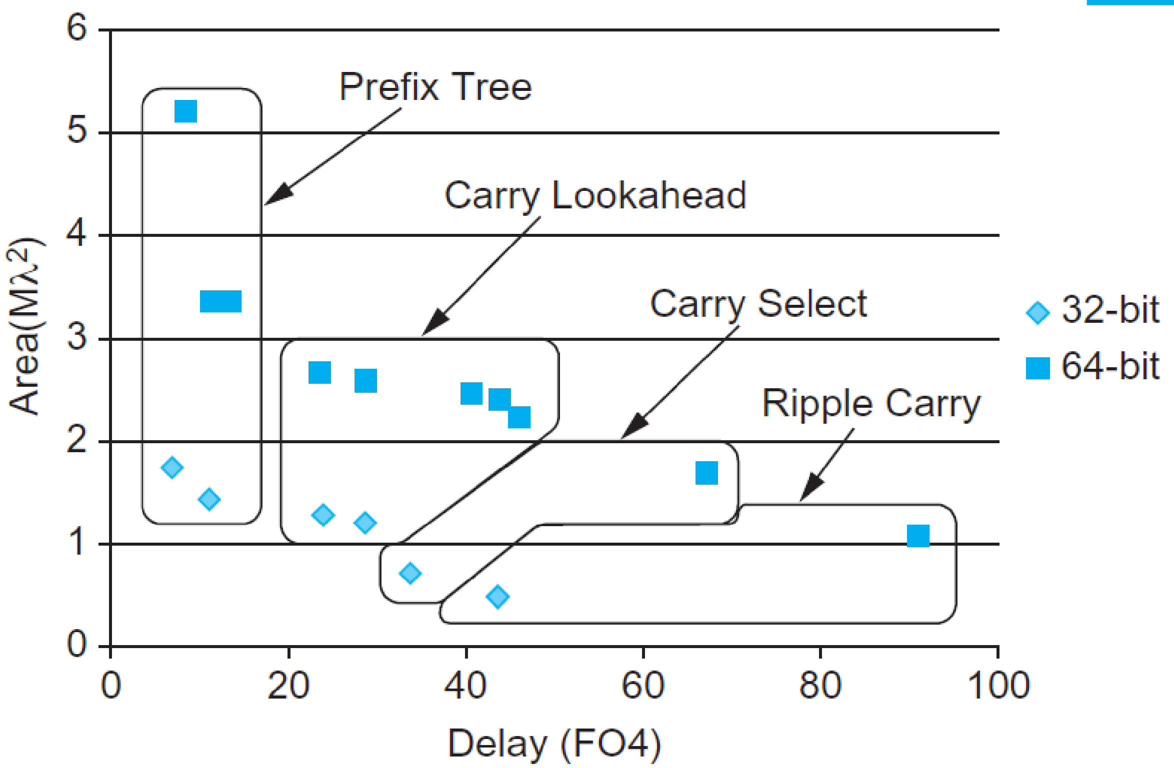






# Adder Family – Choose Wisely



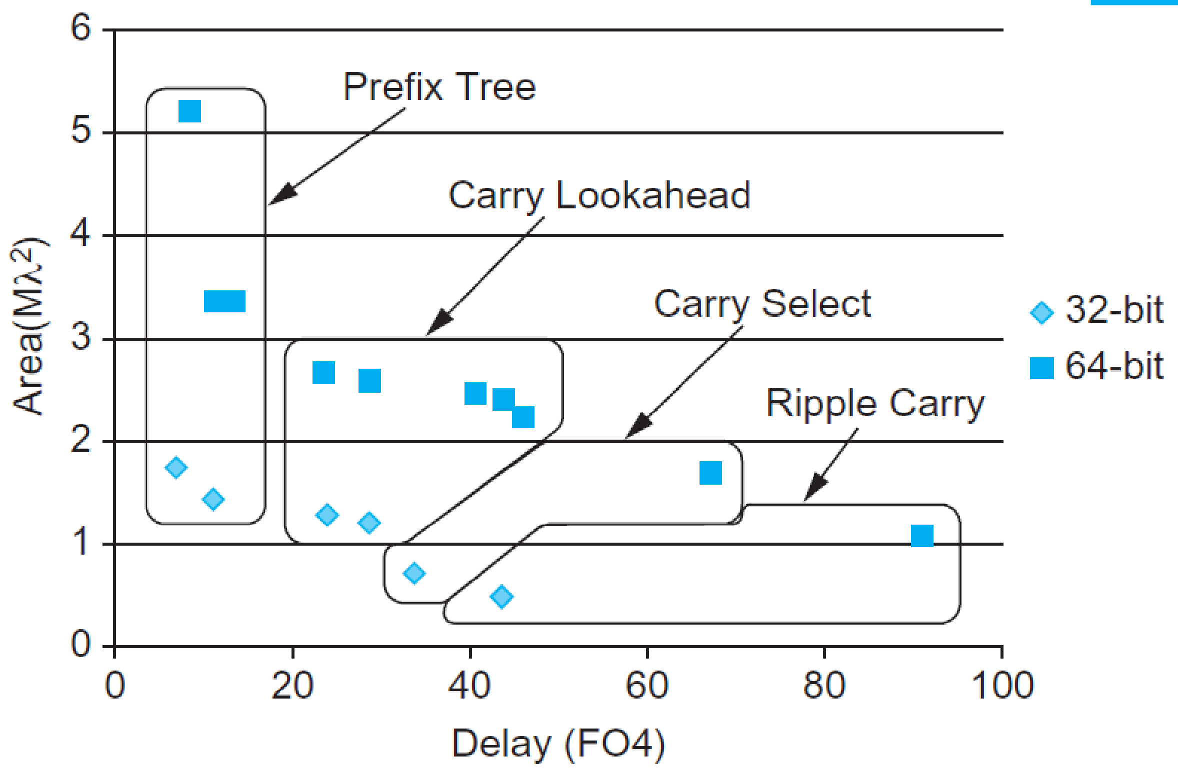



FO: Fan-Out

Everything has trade-off!

# Adder Family – Choose Wisely

Numbers > Arithmetic > Circuits



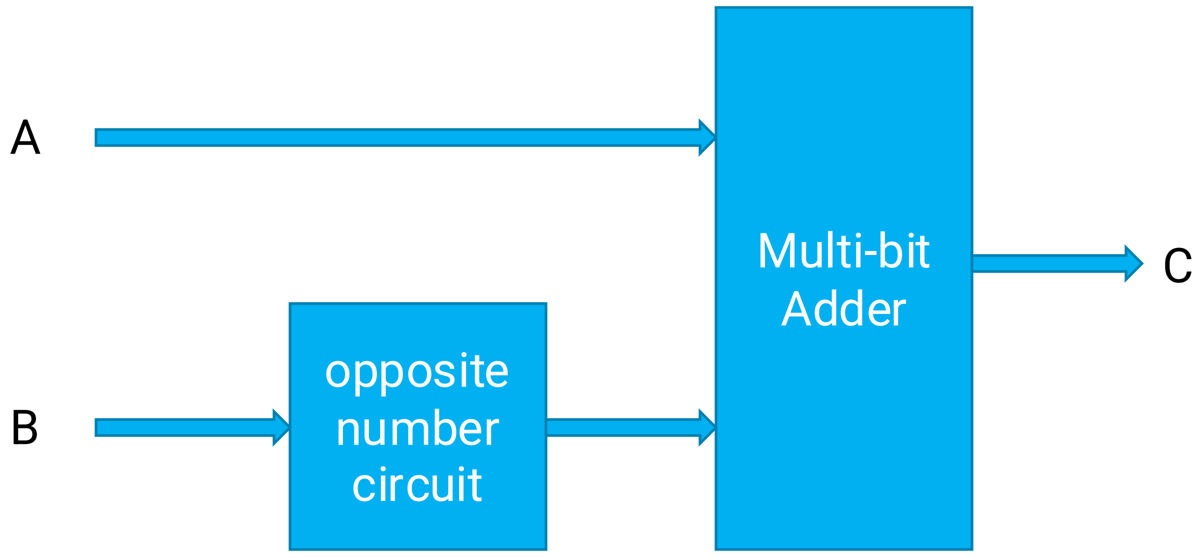

FO: Fan-Out

Everything has trade-off!

# Unsigned Integer Circuits – Subtractor



C = A-B = A+ (-B)




What is "opposite number circuit" though?
[Save for a moment later]

# Unsigned Integer Circuits – Multiplier

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | $y_5$ | $y_4$ | $y_3$ | $y_2$ | $y_1$ | $y_0$ | Multiplicand |
| | | | | | | $x_5$ | $x_4$ | $x_3$ | $x_2$ | $x_1$ | $x_0$ | Multiplier |
| | | | | | | $x_0y_5$ | $x_0y_4$ | $x_0y_3$ | $x_0y_2$ | $x_0y_1$ | $x_0y_0$ | Partial Products |
| | | | | | $x_1y_5$ | $x_1y_4$ | $x_1y_3$ | $x_1y_2$ | $x_1y_1$ | $x_1y_0$ | | |
| | | | | $x_2y_5$ | $x_2y_4$ | $x_2y_3$ | $x_2y_2$ | $x_2y_1$ | $x_2y_0$ | | | |
| | | | $x_3y_5$ | $x_3y_4$ | $x_3y_3$ | $x_3y_2$ | $x_3y_1$ | $x_3y_0$ | | | | |
| | | $x_4y_5$ | $x_4y_4$ | $x_4y_3$ | $x_4y_2$ | $x_4y_1$ | $x_4y_0$ | | | | | |
| | $x_5y_5$ | $x_5y_4$ | $x_5y_3$ | $x_5y_2$ | $x_5y_1$ | $x_5y_0$ | | | | | | |
| $p_{11}$ | $p_{10}$ | $p_9$ | $p_8$ | $p_7$ | $p_6$ | $p_5$ | $p_4$ | $p_3$ | $p_2$ | $p_1$ | $p_0$ | Product |

Multiple-input addition

How do you implement it?

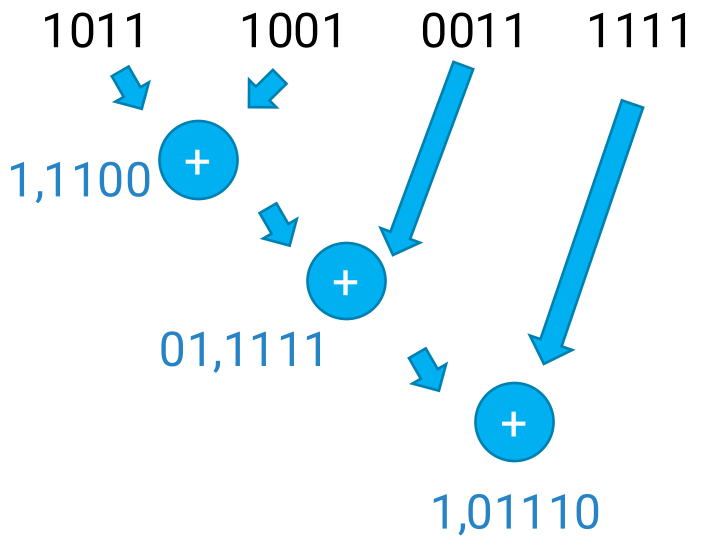
**Yes! Adders!**

# Unsigned Integer Circuits – Multiplier

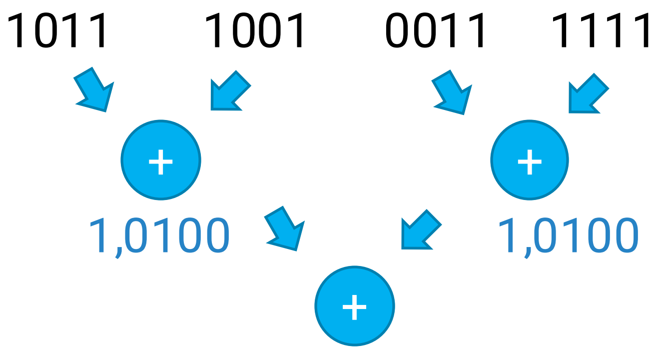


Carry-Save Adder (CSA) & "*carry-save redundant format*"

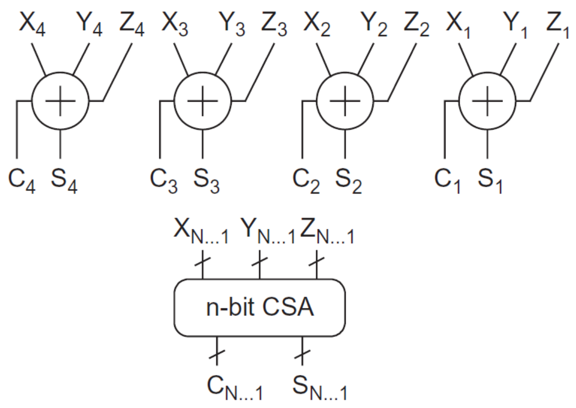
Example: Sum of 1011 1001 0011 1111 ?




( Carry, Sum )



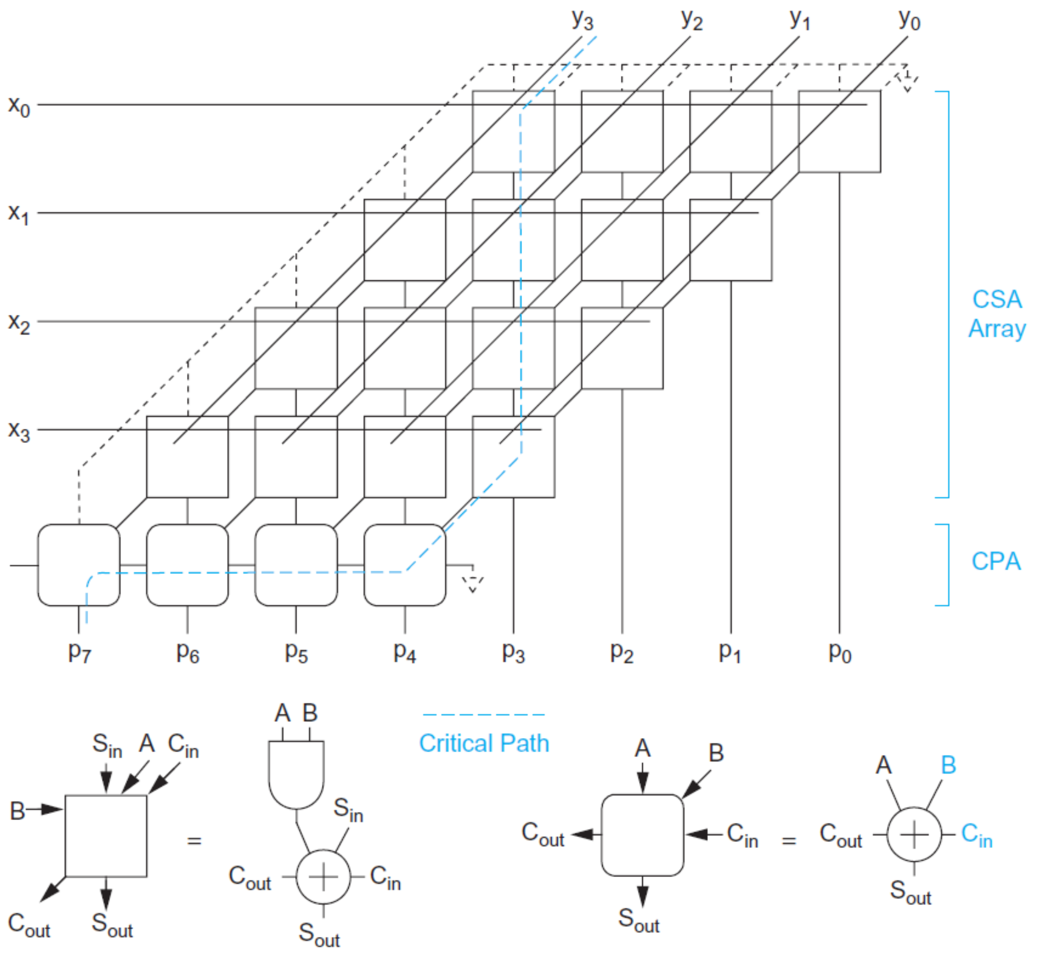

Carry add together,
Sum add together
> Final sum



# Unsigned Integer Circuits – Multiplier



| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | $y_5$ | $y_4$ | $y_3$ | $y_2$ | $y_1$ | $y_0$ | Multiplicand |
| | | | | | | $x_5$ | $x_4$ | $x_3$ | $x_2$ | $x_1$ | $x_0$ | Multiplier |
| | | | | | | $x_0y_5$ | $x_0y_4$ | $x_0y_3$ | $x_0y_2$ | $x_0y_1$ | $x_0y_0$ | Partial Products |
| | | | | | $x_1y_5$ | $x_1y_4$ | $x_1y_3$ | $x_1y_2$ | $x_1y_1$ | $x_1y_0$ | | |
| | | | | $x_2y_5$ | $x_2y_4$ | $x_2y_3$ | $x_2y_2$ | $x_2y_1$ | $x_2y_0$ | | | |
| | | | $x_3y_5$ | $x_3y_4$ | $x_3y_3$ | $x_3y_2$ | $x_3y_1$ | $x_3y_0$ | | | | |
| | | $x_4y_5$ | $x_4y_4$ | $x_4y_3$ | $x_4y_2$ | $x_4y_1$ | $x_4y_0$ | | | | | |
| | $x_5y_5$ | $x_5y_4$ | $x_5y_3$ | $x_5y_2$ | $x_5y_1$ | $x_5y_0$ | | | | | | |
| $p_{11}$ | $p_{10}$ | $p_9$ | $p_8$ | $p_7$ | $p_6$ | $p_5$ | $p_4$ | $p_3$ | $p_2$ | $p_1$ | $p_0$ | Product |

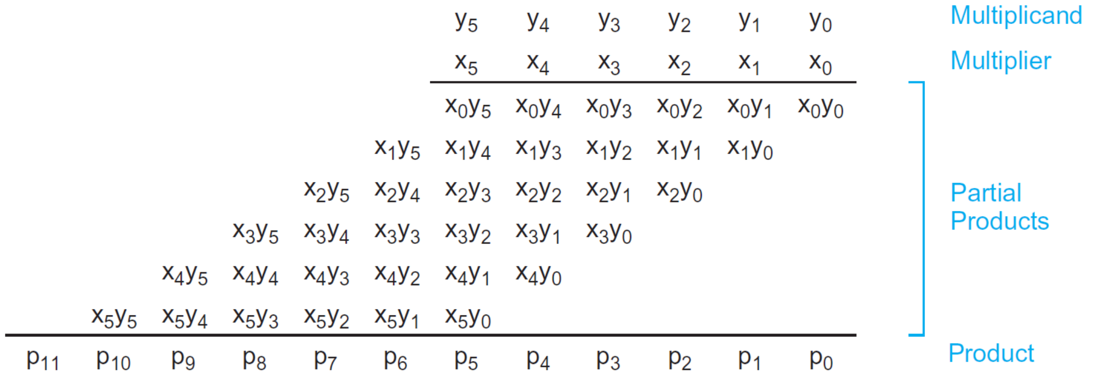



Partial Product:
Single-bit multiplication is equivalent to "AND"

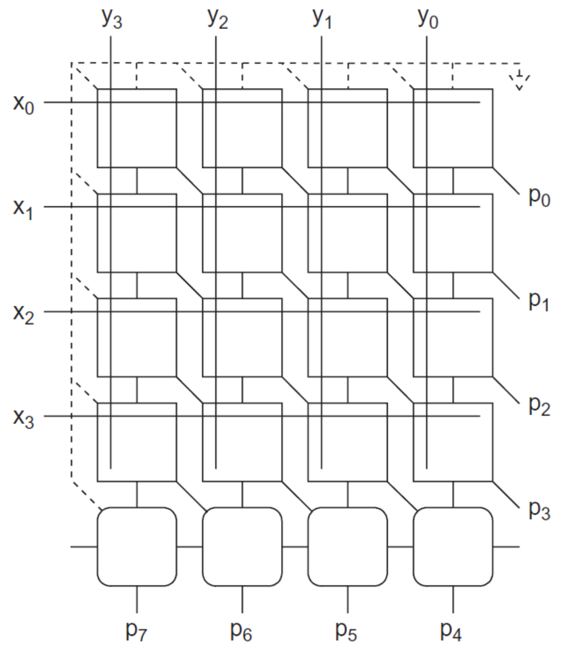
| x | y | Partial product |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 0 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

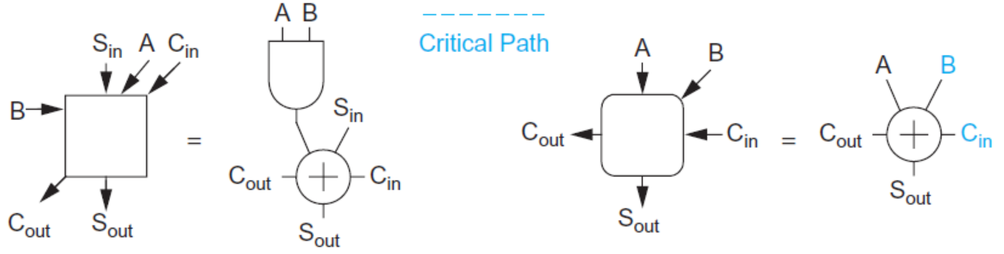
# Unsigned Integer Circuits – Multiplier






Reshape to Rectangular:



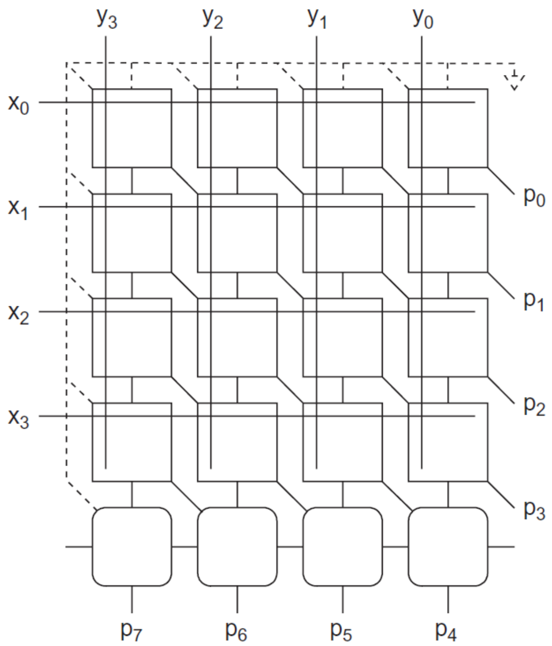

Partial Product:
Single-bit multiplication is equivalent to "AND"

| x | y | Partial product |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 0 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

# Unsigned Integer Circuits – Multiplier



| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | $y_5$ | $y_4$ | $y_3$ | $y_2$ | $y_1$ | $y_0$ | Multiplicand |
| | | | | | | | $x_5$ | $x_4$ | $x_3$ | $x_2$ | $x_1$ | $x_0$ | Multiplier |
| | | | | | | | $x_0y_5$ | $x_0y_4$ | $x_0y_3$ | $x_0y_2$ | $x_0y_1$ | $x_0y_0$ | |
| | | | | | | $x_1y_5$ | $x_1y_4$ | $x_1y_3$ | $x_1y_2$ | $x_1y_1$ | $x_1y_0$ | | |
| | | | | | $x_2y_5$ | $x_2y_4$ | $x_2y_3$ | $x_2y_2$ | $x_2y_1$ | $x_2y_0$ | | | Partial Products |
| | | | | $x_3y_5$ | $x_3y_4$ | $x_3y_3$ | $x_3y_2$ | $x_3y_1$ | $x_3y_0$ | | | | |
| | | | $x_4y_5$ | $x_4y_4$ | $x_4y_3$ | $x_4y_2$ | $x_4y_1$ | $x_4y_0$ | | | | | |
| | | $x_5y_5$ | $x_5y_4$ | $x_5y_3$ | $x_5y_2$ | $x_5y_1$ | $x_5y_0$ | | | | | | |
| $p_{11}$ | $p_{10}$ | $p_9$ | $p_8$ | $p_7$ | $p_6$ | $p_5$ | $p_4$ | $p_3$ | $p_2$ | $p_1$ | $p_0$ | Product |



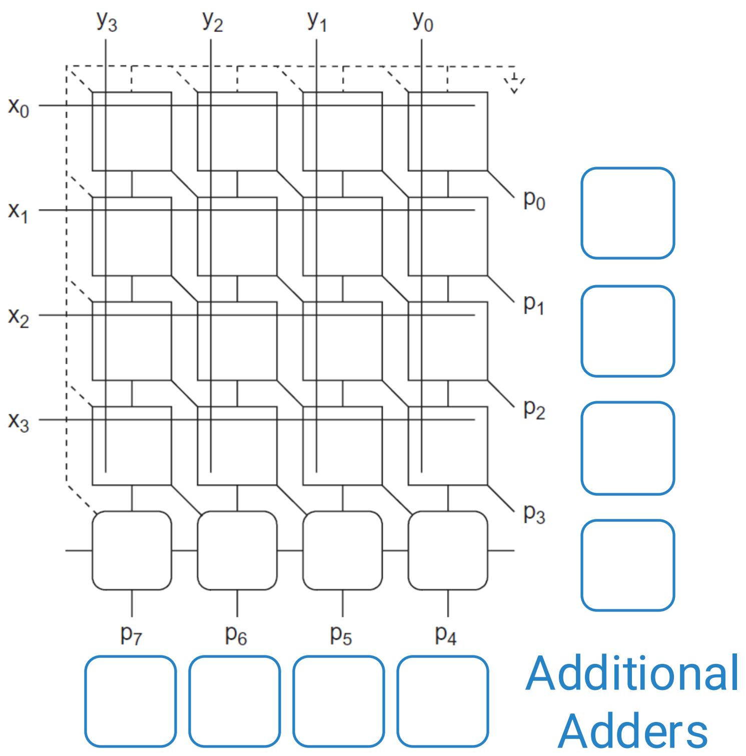

**Anyway to Optimize This?**

**Booth Encoding**

# Unsigned Integer Circuits – Multiply-Add



| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | $y_5$ | $y_4$ | $y_3$ | $y_2$ | $y_1$ | $y_0$ | Multiplicand |
| | | | | | | | $x_5$ | $x_4$ | $x_3$ | $x_2$ | $x_1$ | $x_0$ | Multiplier |
| | | | | | | | $x_0y_5$ | $x_0y_4$ | $x_0y_3$ | $x_0y_2$ | $x_0y_1$ | $x_0y_0$ | Partial Products |
| | | | | | | $x_1y_5$ | $x_1y_4$ | $x_1y_3$ | $x_1y_2$ | $x_1y_1$ | $x_1y_0$ | | |
| | | | | | $x_2y_5$ | $x_2y_4$ | $x_2y_3$ | $x_2y_2$ | $x_2y_1$ | $x_2y_0$ | | | |
| | | | | $x_3y_5$ | $x_3y_4$ | $x_3y_3$ | $x_3y_2$ | $x_3y_1$ | $x_3y_0$ | | | | |
| | | | $x_4y_5$ | $x_4y_4$ | $x_4y_3$ | $x_4y_2$ | $x_4y_1$ | $x_4y_0$ | | | | | |
| | | $x_5y_5$ | $x_5y_4$ | $x_5y_3$ | $x_5y_2$ | $x_5y_1$ | $x_5y_0$ | | | | | | |
| | $p_{11}$ | $p_{10}$ | $p_9$ | $p_8$ | $p_7$ | $p_6$ | $p_5$ | $p_4$ | $p_3$ | $p_2$ | $p_1$ | $p_0$ | Product |



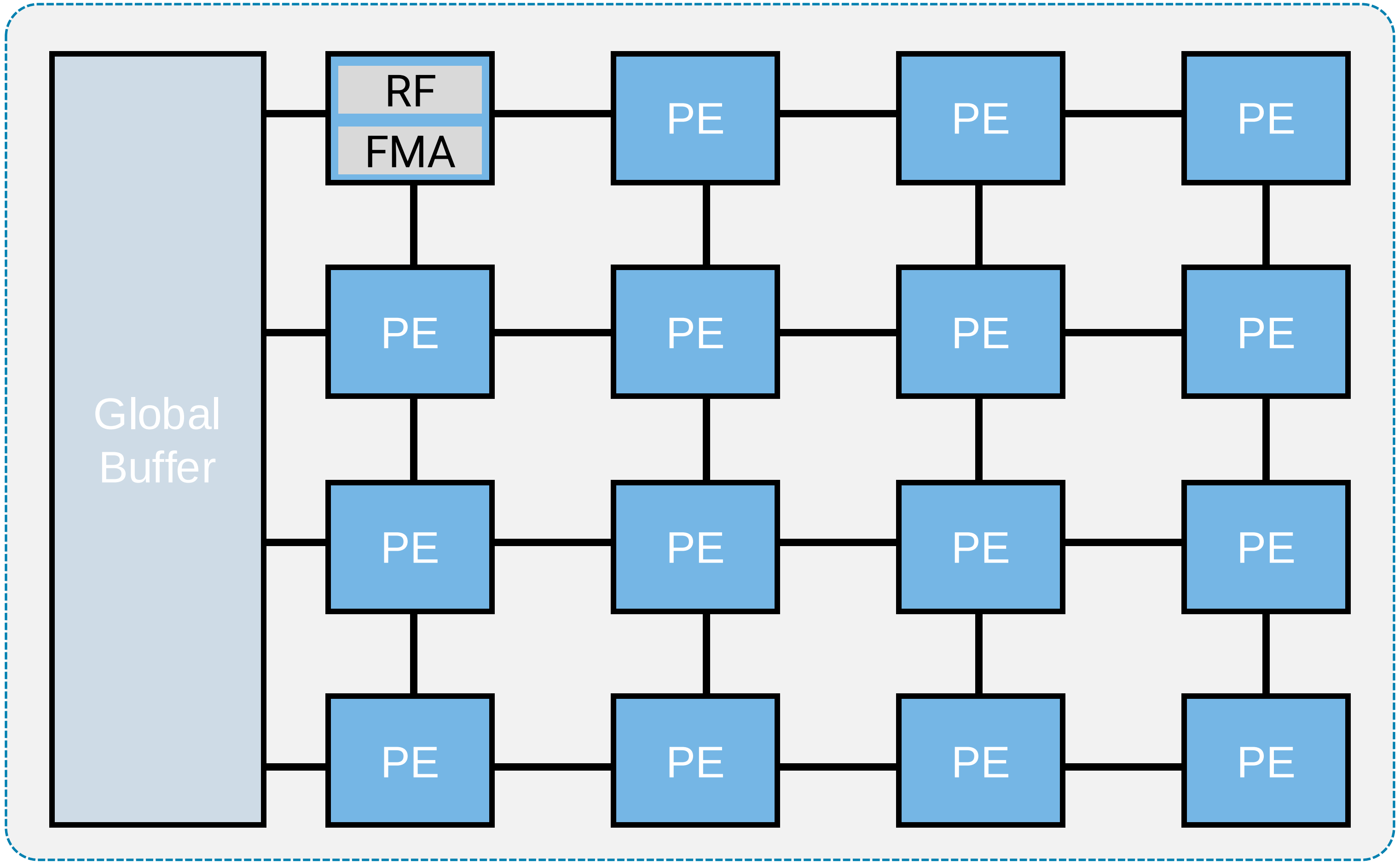

**Application-Specific Optimization**

Fused-Multiply-Add (FMA)

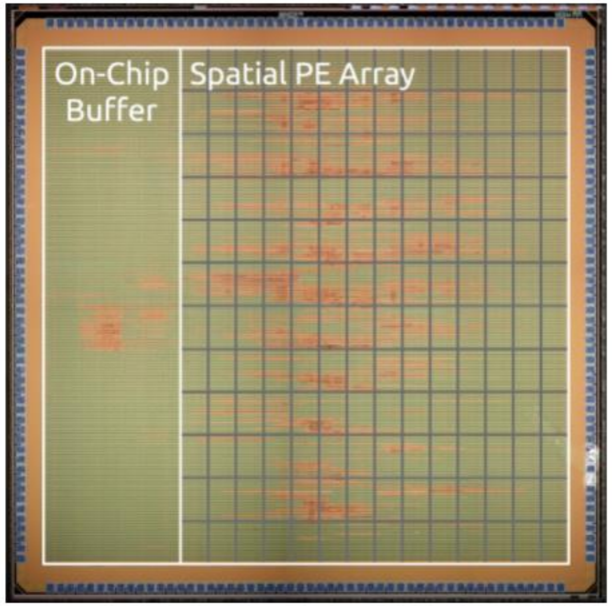
# Multiply-Add Application Example



- Eyeriss: CNN Accelerator




- FMA as processing elements
- local register file (RF)




Chen, Yu-Hsin, et al. "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks." *IEEE journal of solid-state circuits* 52.1 (2016): 127-138.

# Unsigned Integer Circuits – Divider



quotient

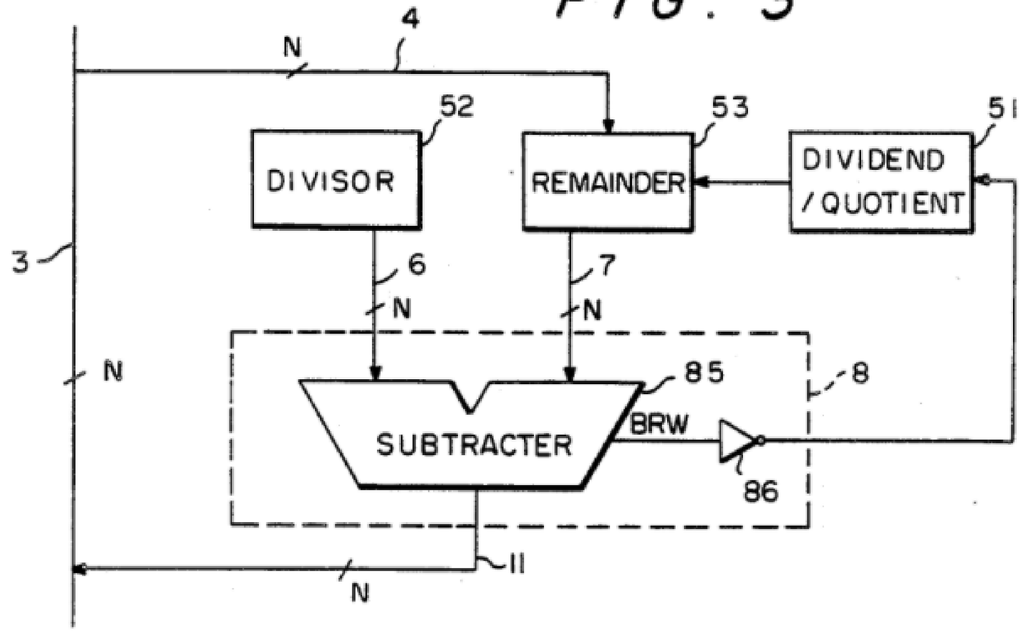
```
       101
    ┌─────
11  │ 1111
    - 11
    ─────
       01
       00
    ─────
       11
       11
    ─────
        0
```

divisor: 11

dividend: 1111

remainder: 0

FIG. 3




Yamahata, Hitoshi. "Integer division circuit provided with a overflow detector circuit." U.S. Patent No. 4,992,969. 12 Feb. 1991.

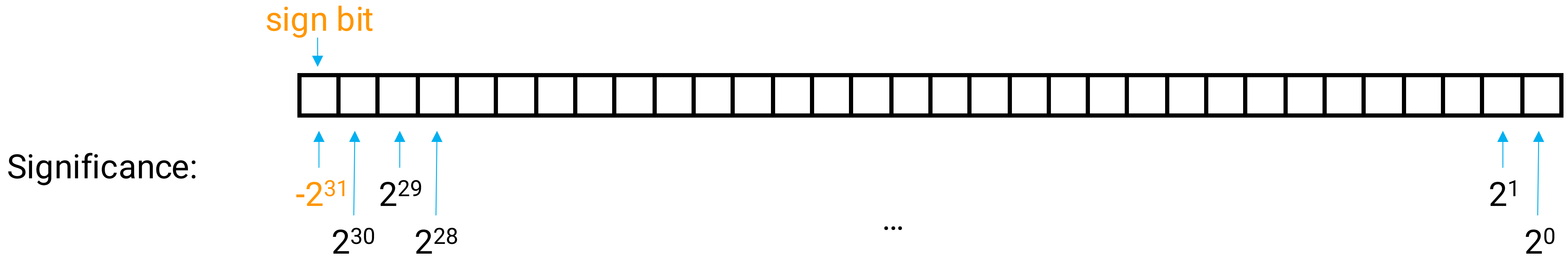
# Part 2

Signed Integers

# Signed Integer



- Signed INT32




- INT16, INT8, …
- Example:

  -32’d7 (=32’hff_ff_ff_f9)
  8’b0100_1101 (=8’hCD)

| Signed Bit | Meaning |
| --- | --- |
| 0 | Positive |
| 1 | Negative |

A Useful Tool: Cryptii

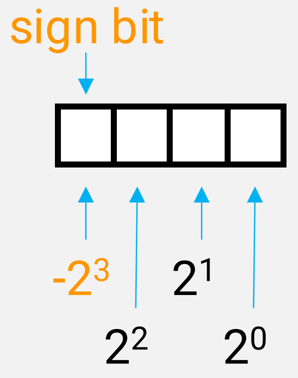
# Signed Integer



- Problem: What is the range of signed vs. unsigned integers?

For UINTn: $0 \sim 2^n-1$
For INTn: $-2^{n-1} \sim 2^{n-1}-1$

Example of Signed INT4




**Here is the answer of "opposite number circuits"!**

| Binary Codeword | Unsigned INT4 | Signed INT4 |
|---|---|---|
| 0000 | 0 | 0 |
| 0001 | 1 | 1 |
| 0010 | 2 | 2 |
| 0011 | 3 | 3 |
| 0100 | 4 | 4 |
| 0101 | 5 | 5 |
| 0110 | 6 | 6 |
| 0111 | 7 | 7 |

| Binary Codeword | Unsigned INT4 | Signed INT4 |
|---|---|---|
| 1000 | 8 | **-8** |
| 1001 | 9 | -7 |
| 1010 | 10 | -6 |
| 1011 | 11 | -5 |
| 1100 | 12 | -4 |
| 1101 | 13 | -3 |
| 1110 | 14 | -2 |
| 1111 | 15 | -1 |

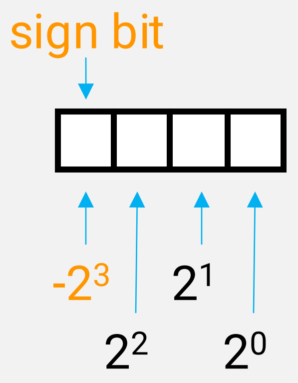
# Signed Integer – Two Types of Shift



- Verilog HDL supports 2 types of shift:

- Logic Shift Operators (<<, >>)
- Arithmetic Shift Operators (<<<, >>>)

Example of Signed INT4




- Logic shift >> <<: filling with zeros
- 3’b100 >> 1’d1 gives 3’b010
- 3’b101 >> 1’d1 gives 3’b010
- 3’b101 << 1’d2 gives 3’b100

Not really stable rule: <<1 : multiply 2 ; >>2: divided by 2

- Arithmetic shift:
  - <<<: Shift left specified number of bits, filling with zero.
  - >>>: Shift right specified number of bits, fill with value of sign bit if expression is signed, othewise fill with zero.

# Signed Integer – Two Types of Shift

PEKING UNIVERSITY 北京大學

Numbers → Arithmetic → Circuits

- Verilog HDL supports 2 types of shift:

- Logic Shift Operators (<<, >>)
- Arithmetic Shift Operators (<<<, >>>)

| Binary Codeword | Unsigned INT4 | Signed INT4 |
|---|---|---|
| 1000 | 8 | **-8** |
| 1001 | 9 | -7 |
| 1010 | 10 | -6 |
| 1011 | 11 | -5 |
| 1100 | 12 | -4 |
| 1101 | 13 | -3 |
| 1110 | 14 | -2 |
| 1111 | 15 | -1 |

4’b1110 >>> 1’d1 gives 4’b1111
4’b0110 >>> 1’d1 gives 4’b0011

- Logic shift >> <<: filling with zeros
- 3’b100 >> 1’d1 gives 3’b010
- 3’b101 >> 1’d1 gives 3’b010
- 3’b101 << 1’d2 gives 3’b100

Not really stable rule: <<1 : multiply 2 ; >>1: divided by 2

- Arithmetic shift:
  - <<<: Shift left specified number of bits, filling with zero.
  - >>>: Shift right specified number of bits, fill with value of sign bit if expression is signed, othewise fill with zero.

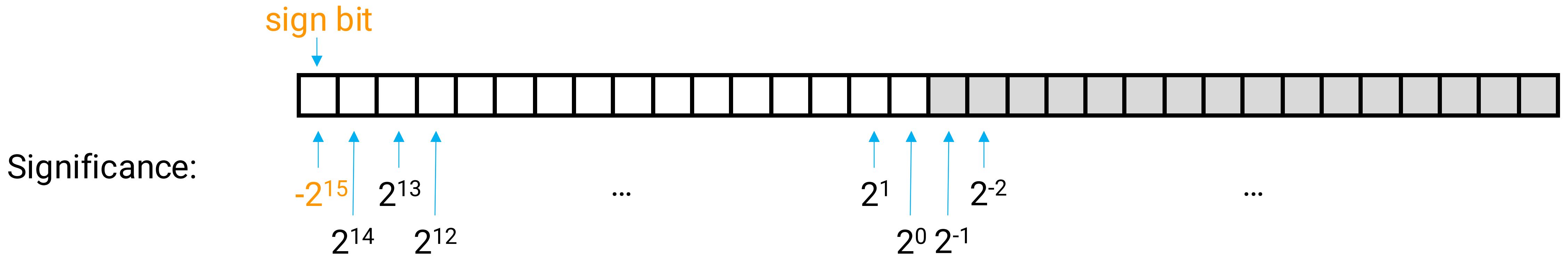
# Part 3 About Fraction

Fixed-Point

# Fixed-Point Number

Numbers → Arithmetic → Circuits

- Fixed32




- Example:

$$(1.10)_2 = (1*2^0+1*2^{-1}+0*2^{-2})_{10}=(1.50)_{10}$$

| Signed Bit | Meaning |
|---|---|
| 0 | Positive |
| 1 | Negative |

# Fixed-Point Number





- Arithmetic Just Works the Same Way!

Integer Arithmetic ⟺ Fixed-Point Arithmetic

- Verilog HDL does not support fixed-point natively

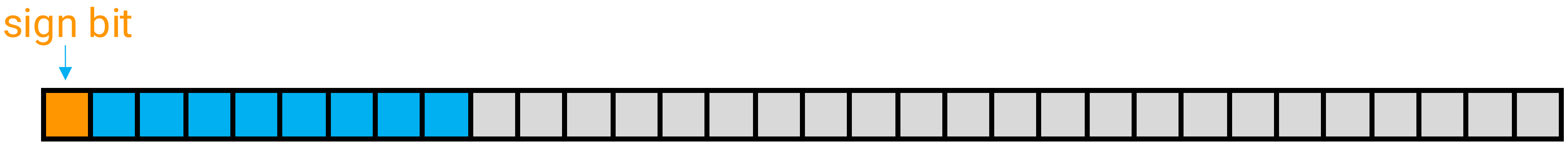
# Part 4

Floating-Point

# Floating-Point Number



- FP32




1b Sign **S** 8b Exponent **E** 23b Mantissa/Fraction **F**

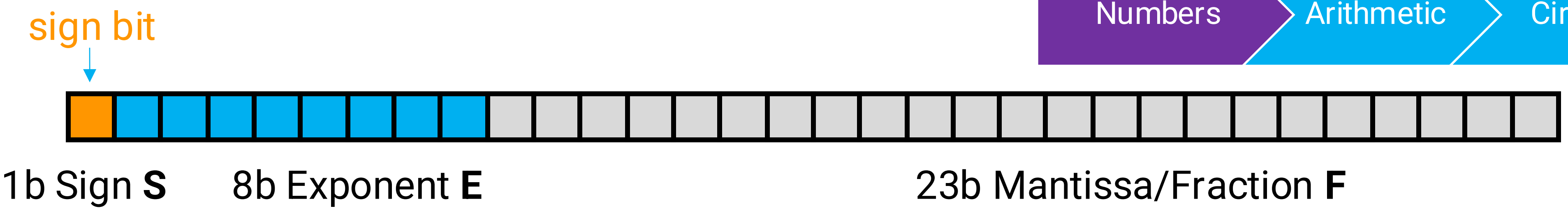
- Meaning:

| Exponent | Fraction | Object | Value |
|---|---|---|---|
| 0 | 0 | 0 | |
| 0 | Nonzero | Denormalized number | $(-1)^{S}\times(0.\mathbf{F})\times 2^{E-B}$ |
| Nonzero | Anything | Floating-point number | $(-1)^{S}\times(1.\mathbf{F})\times 2^{E-B}$ |
| All “1” | 0 | infinity | |
| All “1” | Nonzero | NaN (not a number) | |

# Floating-Point Number



- FP32




| Exponent | Fraction | Object | Value |
| --- | --- | --- | --- |
| 0 | 0 | 0 | -- |
| 0 | Nonzero | Denormalized number | $(-1)^{S}\times(0.\mathbf{F})\times 2^{E-B}$ |
| Nonzero | Anything | Floating-point number | $(-1)^{S}\times(1.\mathbf{F})\times 2^{E-B}$ |
| All “1” | 0 | infinity | -- |
| All “1” | Nonzero | NaN (not a number) | -- |

S=1, E=0, F=0, what is the value? -0=0

# Floating-Point Number





- FP32

| Exponent | Fraction | Object | Value |
|---|---|---|---|
| 0 | 0 | 0 | -- |
| 0 | Nonzero | Denormalized number | $(-1)^S \times (0.\mathbf{F}) \times 2^{1-B}$ |
| Nonzero | Anything | Floating-point number | $(-1)^S \times (1.\mathbf{F}) \times 2^{E-B}$ |
| All "1" | 0 | infinity | -- |
| All "1" | Nonzero | NaN (not a number) | -- |

| Type | Sign | Exponent | Exponent bias | significand | total |
|---|---|---|---|---|---|
| Half (IEEE 754-2008) | 1 | 5 | 15 | 10 | 16 |
| Single | 1 | 8 | B 127 | 23 | 32 |
| Double | 1 | 11 | 1023 | 52 | 64 |
| Quad | 1 | 15 | 16383 | 112 | 128 |

S=0, E=0,
F=23'b10000_00000_00000_00000_000
what is its decimal value?

$$(0.1)_2 \times 2^{-127} = 0.5 \times 2^{-127}$$

# Floating-Point Number



- FP32

| Exponent | Fraction | Object | Value |
| --- | --- | --- | --- |
| 0 | 0 | 0 | -- |
| 0 | Nonzero | Denormalized number | $(-1)^S \times (0.F) \times 2^{1-B}$ |
| Nonzero | Anything | Floating-point number | $(-1)^S \times (1.F) \times 2^{E-B}$ |
| All “1” | 0 | infinity | -- |
| All “1” | Nonzero | NaN (not a number) | -- |

| Type | Sign | Exponent | Exponent bias | significand | total |
| --- | --- | --- | --- | --- | --- |
| Half (IEEE 754-2008) | 1 | 5 | 15 | 10 | 16 |
| Single | 1 | 8 | B 127 | 23 | 32 |
| Double | 1 | 11 | 1023 | 52 | 64 |
| Quad | 1 | 15 | 16383 | 112 | 128 |

S=0, E=8’b127,
F=23’b10000_00000_00000_00000_000
what is its decimal value?

$$(1.1)_2 \times 2^{127-127} = 1.5$$

# Floating-Point Number

- Range

Numbers → Arithmetic → Circuits

| Exponent | Fraction | Object | Value |
|---|---|---|---|
| 0 | 0 | 0 | -- |
| 0 | Nonzero | Denormalized number | $(-1)^{S}\times(0.F)\times 2^{1-B}$ |
| Nonzero | Anything | Floating-point number | $(-1)^{S}\times(1.F)\times 2^{E-B}$ |
| All “1” | 0 | infinity | -- |
| All “1” | Nonzero | NaN (not a number) | -- |

Absolute Max: S=0, E=8’b1111_1110, F=23’h7FFFFF: $(1.11111111111111111111111111111111111)_2 * 2^{8'b1111\_1110-8'b127}$

=3.40282346639e+38

Absolute Min: S=0, E=8’b0000_0001, F=23’h0000_0001: $(1.000000000000000000001)_2 * 2^{1-127}$

# Floating-Point Number



- Range

| Exponent | Fraction | Object | Value |
|---|---|---|---|
| 0 | 0 | 0 | -- |
| 0 | Nonzero | Denormalized number | $(-1)^S \times (0.\mathbf{F}) \times 2^{1-B}$ |
| Nonzero | Anything | Floating-point number | $(-1)^S \times (1.\mathbf{F}) \times 2^{E-B}$ |
| All “1” | 0 | infinity | -- |
| All “1” | Nonzero | NaN (not a number) | -- |

Absolute Max: S=0, E=8’b1111_1110, F=23’h7FFFFF: $(1.11111111111111111111111)_2 * 2^{8'b1111\_1110 - 8'b127}$
=3.40282346639e+38

Absolute Min: S=0, E=8’b0000_0001, F=23’h0000_0000: $(1.00000000000000000000000)_2 * 2^{1-127}$
=1.17549435082e-38

Denormalized:

Absolute Min: S=0, E=8’b0000_0000, F=23’h0000_0001: $(0.00000000000000000000001)_2 * 2^{1-127}$
=1.40129846432e-45

# Floating-Point Number

PEKING UNIVERSITY

Numbers → Arithmetic → Circuits

- Add

```
123456.7 = 1.234567 * 10^5
101.7654 = 1.017654 * 10^2 = 0.001017654 * 10^5

Hence:
123456.7 + 101.7654 = (1.234567 * 10^5) + (1.017654 * 10^2)
                    = (1.234567 * 10^5) + (0.001017654 * 10^5)
                    = (1.234567 + 0.001017654) * 10^5
                    =  1.235584654 * 10^5
```

```
  E=5;  F=1.234567     (123456.7)
+ E=2;  F=1.017654     (101.7654)

  E=5;  F=1.234567
+ E=5;  F=0.001017654  (after shifting)
--------------------
  E=5;  F=1.235584654  (true sum: 123558.4654)
```

Round-off error

Actually, result is: e=5; s=1.235585 (final sum: 123558.5)

Try by yourself:
(E=5, F=1.234567) + (E=-3, F=9.876543) = ??

```
  E=5;   F=1.234567
+ E=-3;  F=9.876543

  E=5;   F=1.234567
+ E=5;   F=0.00000009876543 (after shifting)
---------------------
  E=5;   F=1.23456709876543 (true sum)
  E=5;   F=1.234567         (after rounding/normalization)
```

# Floating-Point Number

Numbers → Arithmetic → Circuits

- Subtract

Try by yourself:
(E=5, F=1.234571) - (E=5, 1.234567) = ??

```
  E=5;  F=1.234571
- E=5;  F=1.234567
----------------
  E=5;  F=0.000004
  E=-1; F=4.000000 (after rounding/normalization)
```

Change to normalized form of FP numbers

# Floating-Point Number



- Multiply:

```
  E=3;  F=4.734612
× E=5;  F=5.417242
-----------------------
  E=8;  F=25.648538980104 (true product)
  E=8;  F=25.64854        (after rounding)
  E=9;  F=2.564854        (after normalization)
```

Exponent: Sum Operation

Mantissa: Multiply Operation

Don't forget normalization

- Divide:

Exponent: Subtract Operation

Mantissa: Divide Operation

Don't forget normalization

Q: What if normalized number multiplies denormalized number?

# Floating-Point Number



## Incompleteness of Floating-Point Arithmetic

- May not associative:

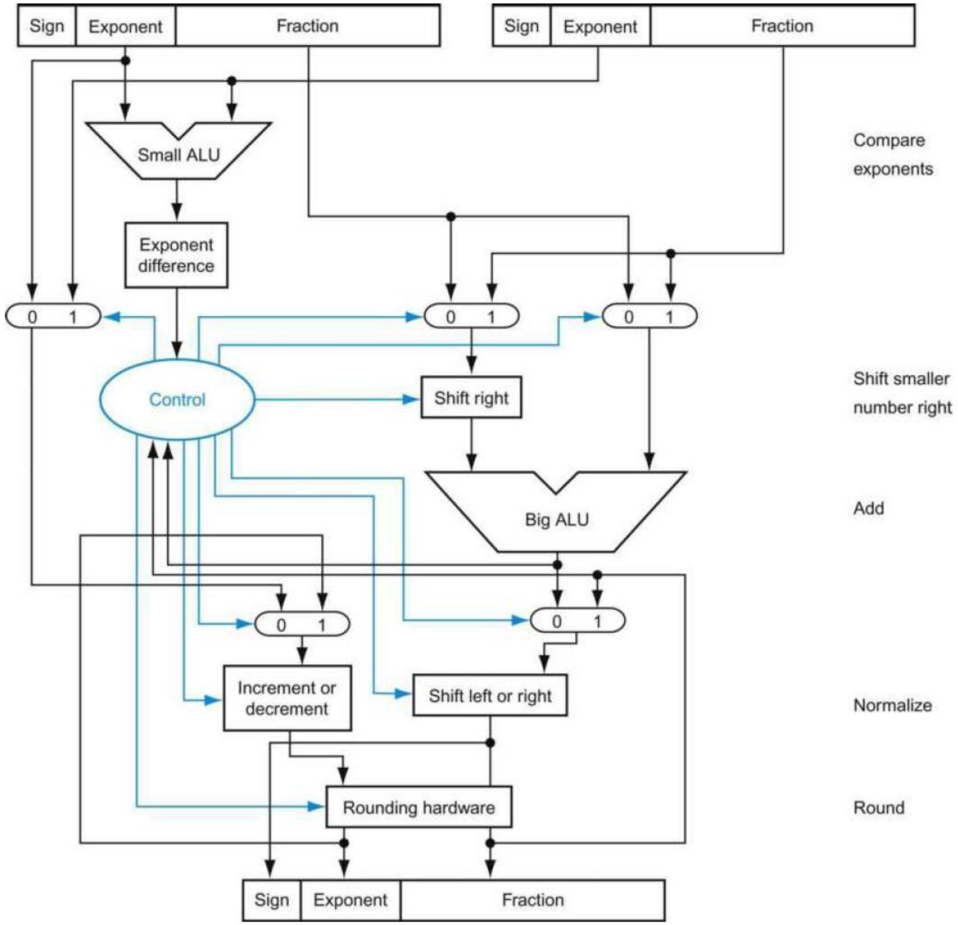
```
1234.567 + 45.67844 = 1280.245
                      1280.245 + 0.0004 = 1280.245
but
45.67840 + 0.00004 = 45.67844
                     45.67844 + 1234.567 = 1280.246
```

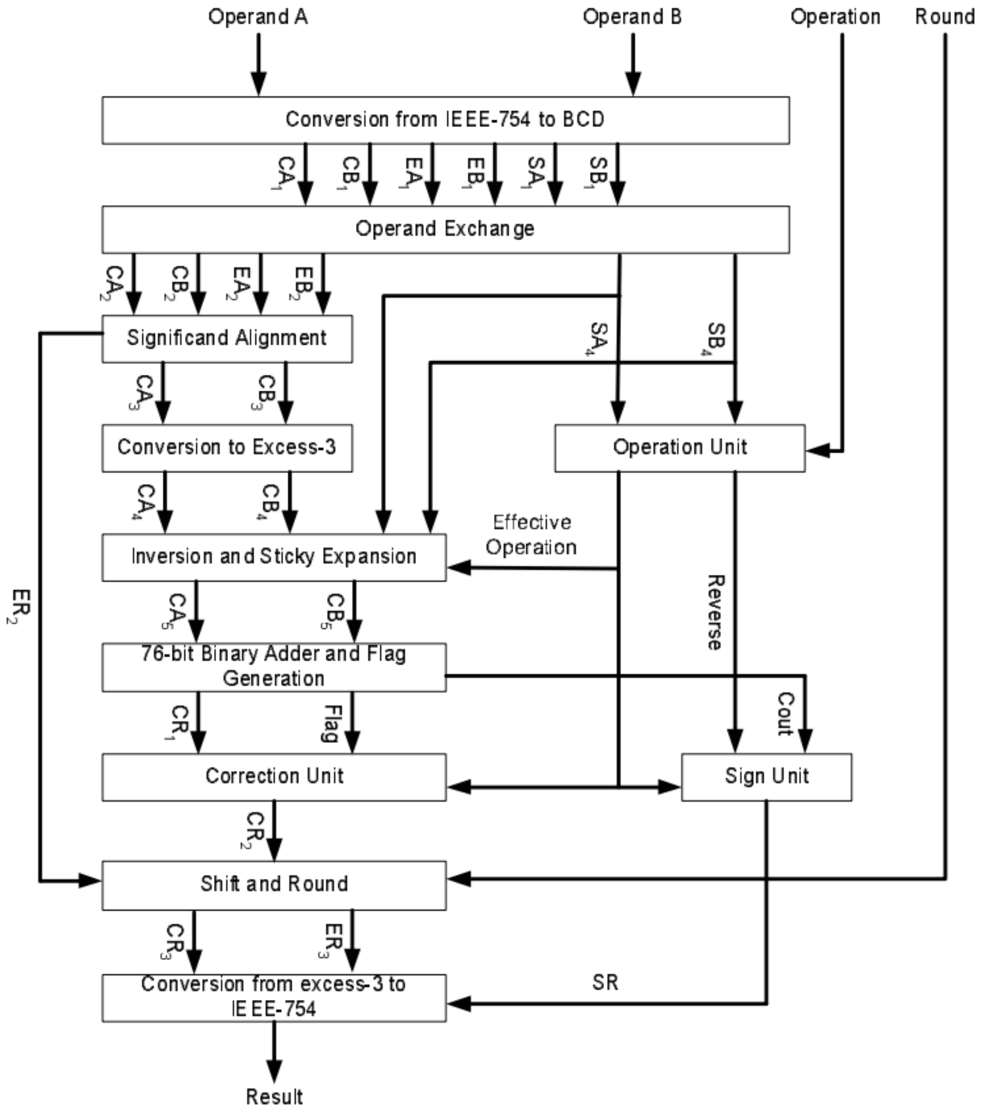
- May not distributive:

```
1234.567 × 3.333333 = 4115.223
1.234567 × 3.333333 = 4.115223
                      4115.223 + 4.115223 = 4119.338
but
1234.567 + 1.234567 = 1235.802
                      1235.802 × 3.333333 = 4119.340
```

# Floating-Point Number

**Adder:**



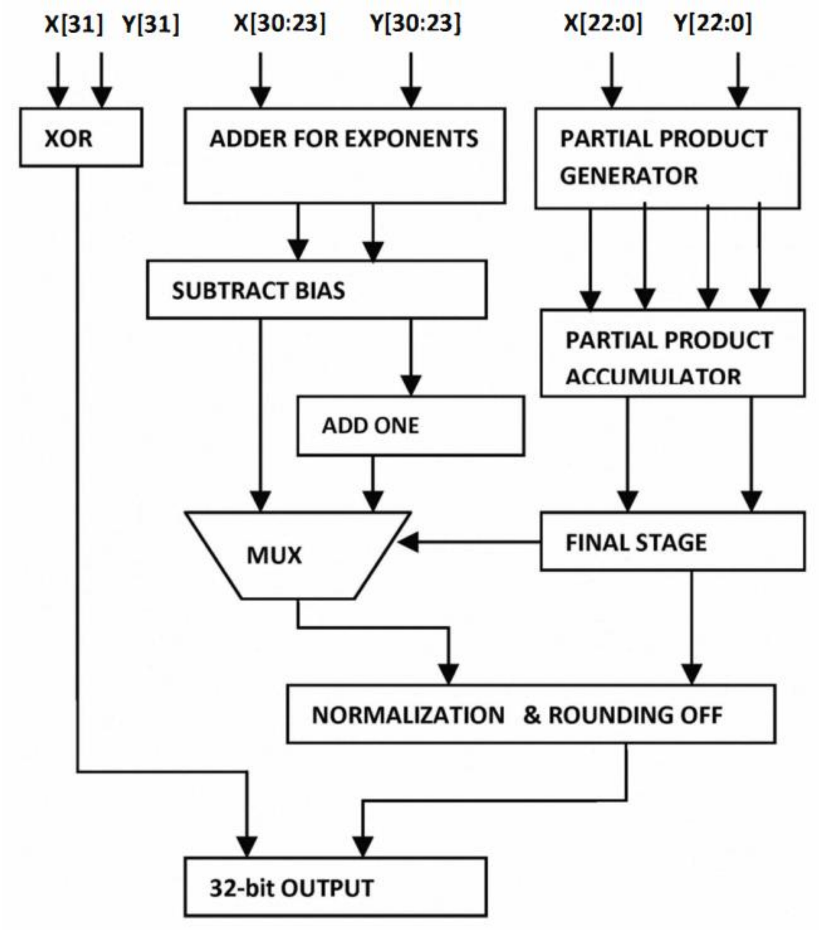




Thompson, John, Nandini Karra, and Michael J. Schulte. "A 64-bit decimal floating-point adder." *IEEE Computer Society Annual Symposium on VLSI*. IEEE, 2004.

# Floating-Point Number



**Multiply:**




Sign: XOR

Exponent: Add

Mantissa: Multiply

Jain, Anna, et al. "FPGA design of a fast 32-bit floating point multiplier unit." *2012 International Conference on Devices, Circuits and Systems (ICDCS)*. IEEE, 2012.

# Floating-Point Number



Floating Point ALU supports +-*/

**RISC-V floating-point assembly language**

| Category | Instruction | Example | Meaning | Comments |
|---|---|---|---|---|
| Arithmetic | FP add single | fadd.s f0, f1, f2 | f0 = f1 + f2 | FP add (single precision) |
| | FP subtract single | fsub.s f0, f1, f2 | f0 = f1 - f2 | FP subtract (single precision) |
| | FP multiply single | fmul.s f0, f1, f2 | f0 = f1 * f2 | FP multiply (single precision) |
| | FP divide single | fdiv.s f0, f1, f2 | f0 = f1 / f2 | FP divide (single precision) |
| | FP square root single | fsqrt.s f0, f1 | f0 = √f1 | FP square root (single precision) |
| | FP add double | fadd.d f0, f1, f2 | f0 = f1 + f2 | FP add (double precision) |
| | FP subtract double | fsub.d f0, f1, f2 | f0 = f1 - f2 | FP subtract (double precision) |
| | FP multiply double | fmul.d f0, f1, f2 | f0 = f1 * f2 | FP multiply (double precision) |
| | FP divide double | fdiv.d f0, f1, f2 | f0 = f1 / f2 | FP divide (double precision) |
| | FP square root double | fsqrt.d f0, f1 | f0 = √f1 | FP square root (double precision) |
| Comparison | FP equality single | feq.s x5, f0, f1 | x5 = 1 if f0 == f1, else 0 | FP comparison (single precision) |
| | FP less than single | flt.s x5, f0, f1 | x5 = 1 if f0 < f1, else 0 | FP comparison (single precision) |
| | FP less than or equals single | fle.s x5, f0, f1 | x5 = 1 if f0 <= f1, else 0 | FP comparison (single precision) |
| | FP equality double | feq.d x5, f0, f1 | x5 = 1 if f0 == f1, else 0 | FP comparison (double precision) |
| | FP less than double | flt.d x5, f0, f1 | x5 = 1 if f0 < f1, else 0 | FP comparison (double precision) |
| | FP less than or equals double | fle.d x5, f0, f1 | x5 = 1 if f0 <= f1, else 0 | FP comparison (double precision) |
| Data transfer | FP load word | flw f0, 4(x5) | f0 = Memory[x5 + 4] | Load single-precision from memory |
| | FP load doubleword | fld f0, 8(x5) | f0 = Memory[x5 + 8] | Load double-precision from memory |
| | FP store word | fsw f0, 4(x5) | Memory[x5 + 4] = f0 | Store single-precision from memory |
| | FP store doubleword | fsd f0, 8(x5) | Memory[x5 + 8] = f0 | Store double-precision from memory |