TC2009B: Digital design
Addition & Subtraction

Isaac Pérez Andrade

ITESM Guadalajara
School of Engineering & Science
Department of Computer Science
August – December 2021

References

The following material has been adopted and adapted from

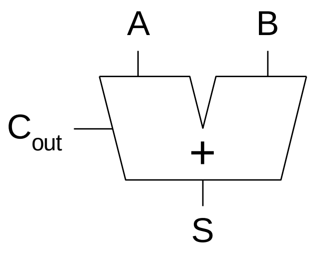
Patterson, D. A., Hennessy, J. L., *Computer Organization and design: The hardware/software interface – ARM edition*, Morgan Kaufmann, 2017.

S. L. Harris and D. M. Harris, *Digital design and computer architecture - ARM edition*, Morgan Kaufmann, 2016.

Adder

1-bit adders

Half Adder

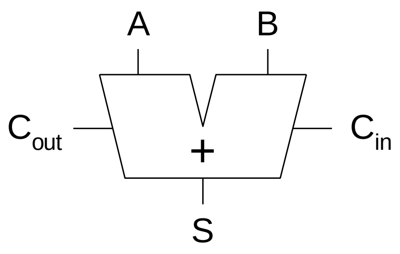


A	B	C_{out}	S
0	0		
0	1		
1	0		
1	1		

$S =$

$C_{out} =$

Full Adder



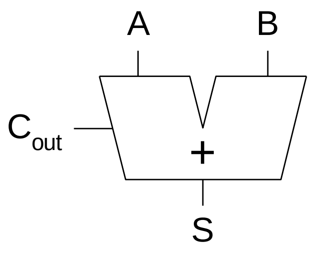
C_{in}	A	B	C_{out}	S
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

$S =$

$C_{out} =$

1-bit adders

Half Adder

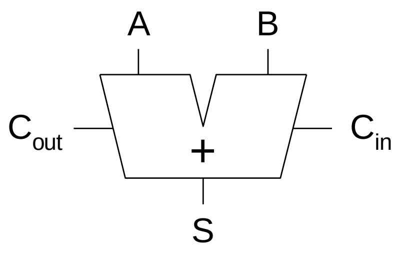


A	B	C_{out}	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

S =

C_{out} =

Full Adder



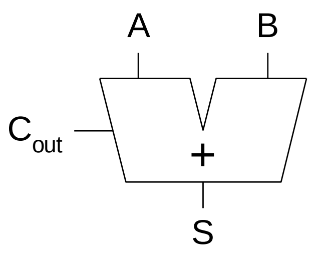
C_{in}	A	B	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

S =

C_{out} =

1-bit adders

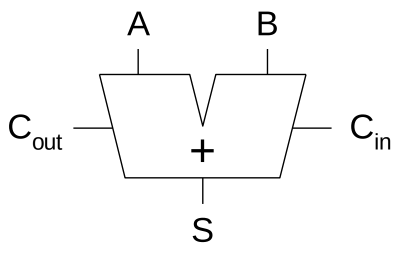
Half Adder



A	B	C_{out}	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

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

Full Adder

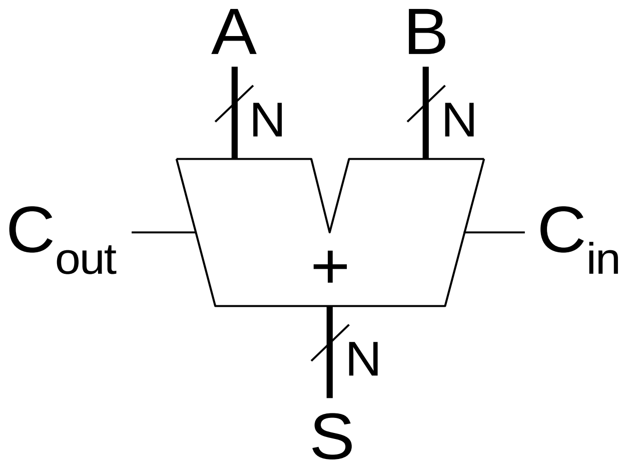


C_{in}	A	B	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

$$S = A \oplus B \oplus C_{in}$$
$$C_{out} = AB + AC_{in} + BC_{in}$$

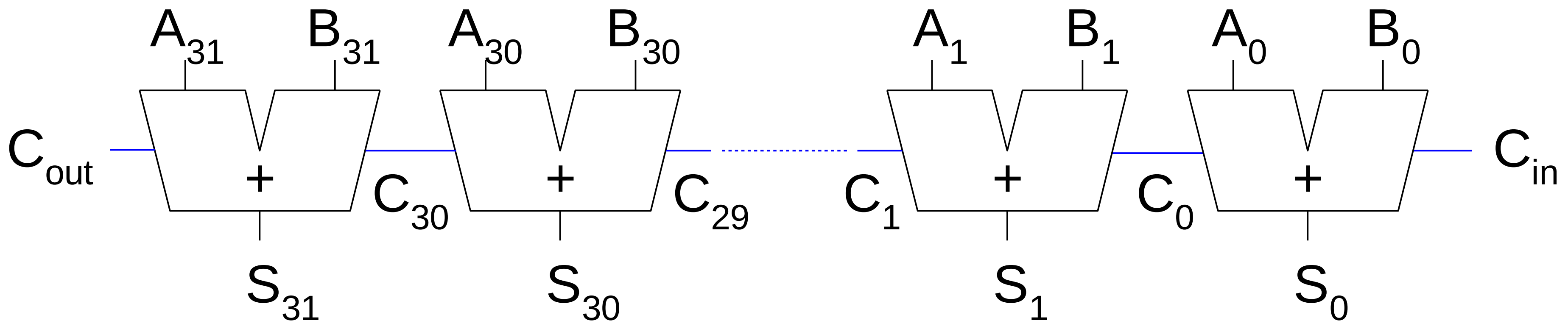
Multibit adders

- Types of carry propagate adders (CPAs):
  - Ripple-carry (slow)
  - Carry-lookahead (fast)
  - Prefix (faster)
- Carry-lookahead and prefix adders faster for large adders but require more hardware



Ripple-Carry adder

- Chain 1-bit adders together
- Carry ripples through entire chain
- Disadvantage: **slow**



- $t_{ripple} = Nt_{FA}$

  t_{FA} is the delay of a 1-bit full adder

Carry-lookahead adder

Compute C_{out} for *k*-bit blocks using *generate* and *propagate* signals

- **Some definitions:**
  - Column *i* produces a carry out by either ***generating*** a carry out or ***propagating*** a carry in to the carry out
  - Generate (G_i) and propagate (P_i) signals for each column:
    - **Generate:** Column *i* will generate a carry out if A_i **and** B_i are both 1.

$$G_i = A_i B_i$$

    - **Propagate:** Column *i* will propagate a carry in into the carry out if A_i **or** B_i is 1.

$$P_i = A_i + B_i$$

    - **Carry out:** The carry out of column *i* (C_i) is:

$$C_i = A_i B_i + (A_i + B_i) C_{i-1} = G_i + P_i C_{i-1}$$

Carry-lookahead adder

- Now use column **propagate** and **generate** signals to compute **block propagate** and **block generate** signals for k-bit blocks, *i.e.*:
    - Compute if a k-bit group will propagate a carry in (to the block) to the carry out (of the block)
    - Compute if a k-bit group will generate a carry out (of the block)

Carry-lookahead adder

- **Example**: Block propagate and generate signals for 4-bit blocks ($P_{3:0}$ and $G_{3:0}$):

$$P_{3:0} = P_3 P_2 P_1 P_0$$

$$G_{3:0} = G_3 + P_3 (G_2 + P_2 (G_1 + P_1 G_0))$$

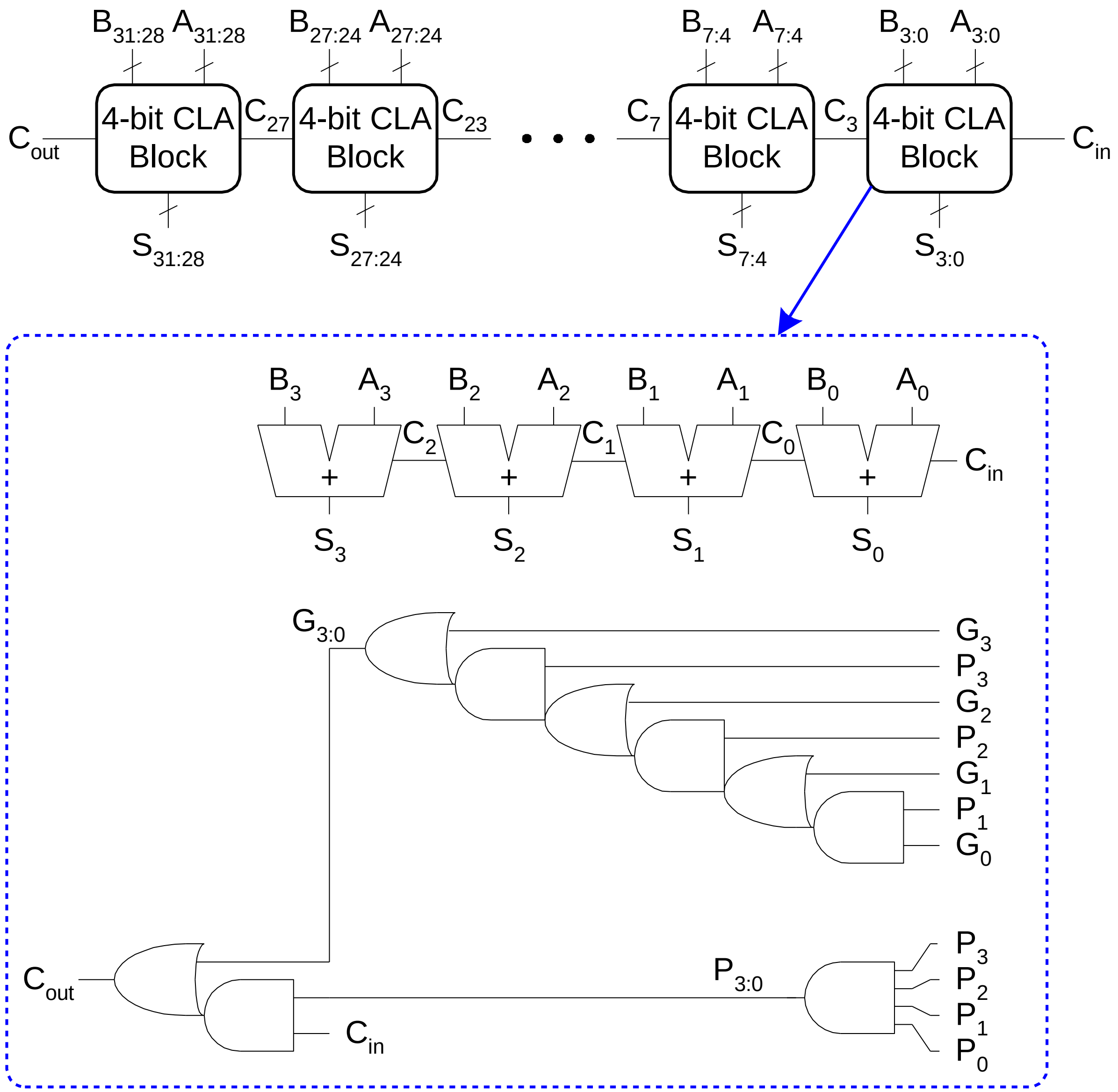
- **Generally**,

$$P_{i:j} = P_i P_{i-1} P_{i-2} \cdots P_j$$

$$G_{i:j} = G_i + P_i (G_{i-1} + P_{i-1} (G_{i-2} + P_{i-2} (G_{i-3} + P_{i-3} (\ldots G_{j+1} + P_{j+1} G_j))$$

$$C_i = G_{i:j} + P_{i:j} C_{i-1}$$

32-bit Carry-lookahead with 4 blocks



Carry-lookahead addition

- **Step 1:** Compute G_i and P_i for all columns
- **Step 2:** Compute G and P for k-bit blocks
- **Step 3:** C_{in} propagates through each k-bit propagate/generate logic (meanwhile computing sums)
- **Step 4:** Compute sum for most significant k-bit block

Carry-lookahead adder delay

- For N-bit CLA with k-bit blocks:

$$t_{CLA} = t_{pg} + t_{pg\_block} + (N/k - 1)t_{AND\_OR} + kt_{FA}$$

where:

- t_{pg}: delay to generate all P_i and G_i
- t_{pg_block} : delay to generate all $P_{i:j}$ and $G_{i:j}$
- t_{AND_OR} : delay from C_{in} to C_{out} of final AND/OR gate in k-bit CLA block

- An N-bit carry-lookahead adder is generally much faster than a ripple-carry adder for $N > 16$

Prefix adder

- Computes carry in ($C_{i\text{-}1}$) for each column, then computes sum:

$$S_i = (A_i \oplus B_i) \oplus C_i$$

- It computes $C_{i\text{-}1}$ by:
  - Computing G and P for 1-, 2-, 4-, 8-bit blocks, etc. until all G_i (carry in) known
- $\log_2 N$ stages

Prefix adder

- Carry is either *generated* in a column or *propagated* from a previous column.
- Column -1 holds C_{in}, so

  $G_{-1} = C_{in}$

  $P_{-1} = 0$
- Carry in into column i = carry out of column i-1:

  $C_{i-1} = G_{i-1:-1}$
- $G_{i-1:-1}$: generate signal spanning columns i-1 to -1
- Sum equation:

  $S_i = (A_i \oplus B_i) \oplus G_{i-1:-1}$
- **Goal:** Quickly compute $G_{0:-1}, G_{1:-1}, G_{2:-1}, G_{3:-1}, G_{4:-1}, G_{5:-1}, \ldots$ (called ***prefixes***) $(C_0, C_1, C_2, C_3, C_4, C_5,)$

Prefix adder

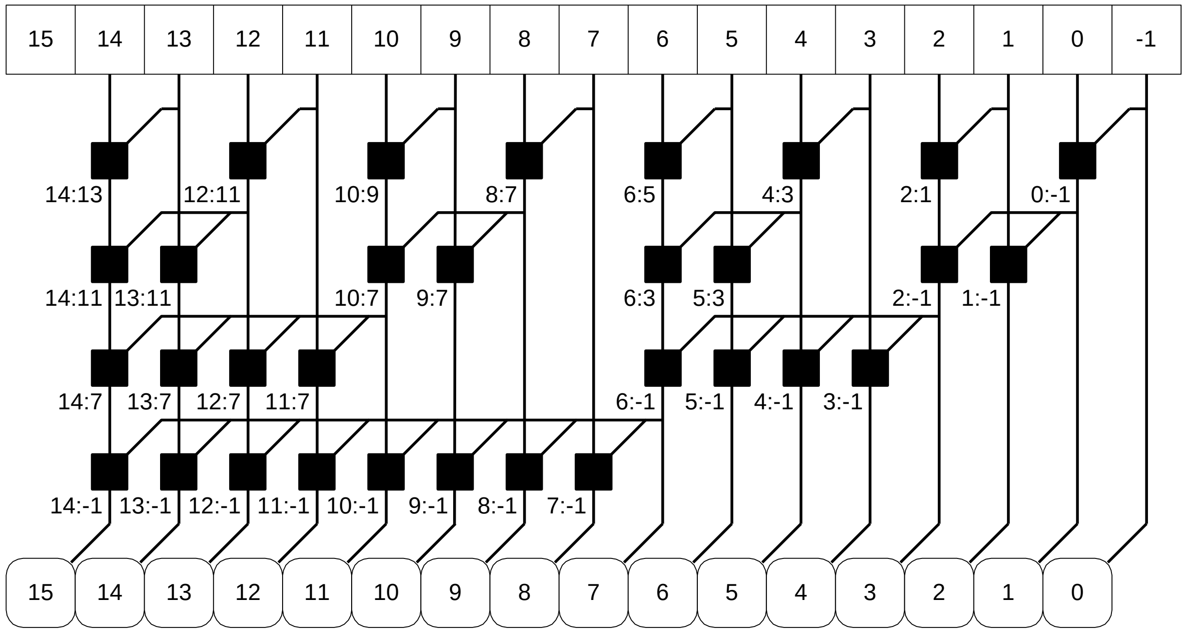
- Generate and propagate signals for a block spanning bits *i:j*

$$G_{i:j} = G_{i:k} + P_{i:k}\, G_{k-1:j}$$

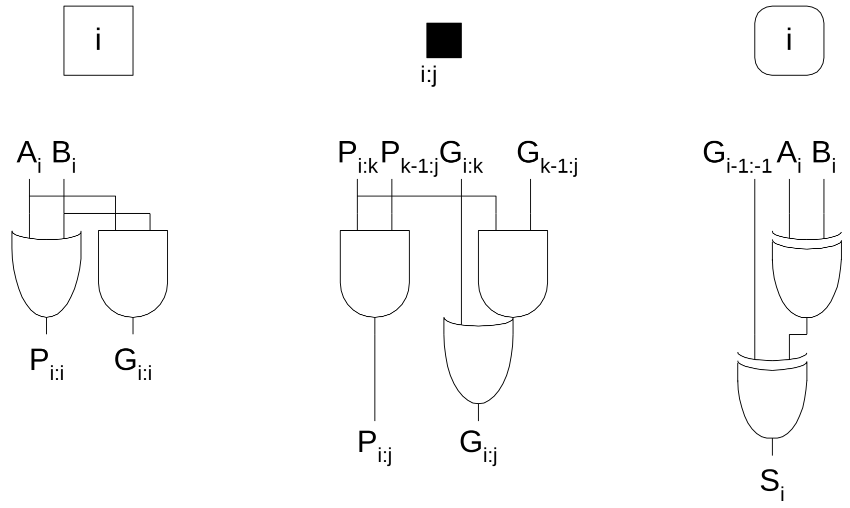
$$P_{i:j} = P_{i:k} P_{k-1:j}$$

- In words:
  - **Generate:** block *i:j* will generate a carry if:
    - upper part (*i*:*k*) generates a carry or
    - upper part (*i*:*k*) propagates a carry generated in lower part (*k*-1:*j*)
  - **Propagate:** block *i:j* will propagate a carry if *both* the upper and lower parts propagate the carry

16-bit prefix adder



Legend



Prefix adder delay

$$t_{PA} = t_{pg} + \log_2 N(t_{pg\_prefix}) + t_{XOR}$$

- t_{pg}: delay to produce P_i, G_i (AND or OR gate)
- t_{pg_prefix}: delay of black prefix cell (AND-OR gate)

Adder delay comparison

Compare delay of 32-bit adder structures: ripple-carry, carry-lookahead, and prefix adders

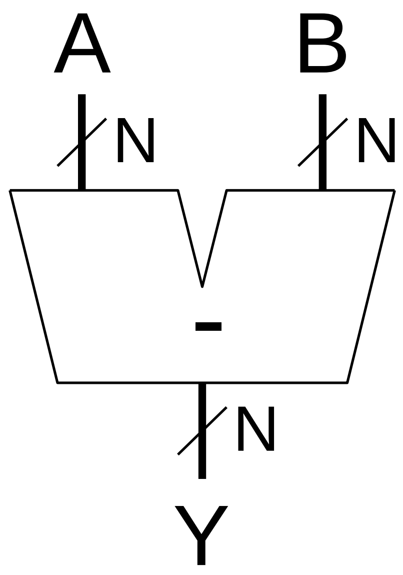
- Carry-lookahead has 4-bit blocks
- 2-input gate delay = 100 ps;
- full adder delay = 300 ps

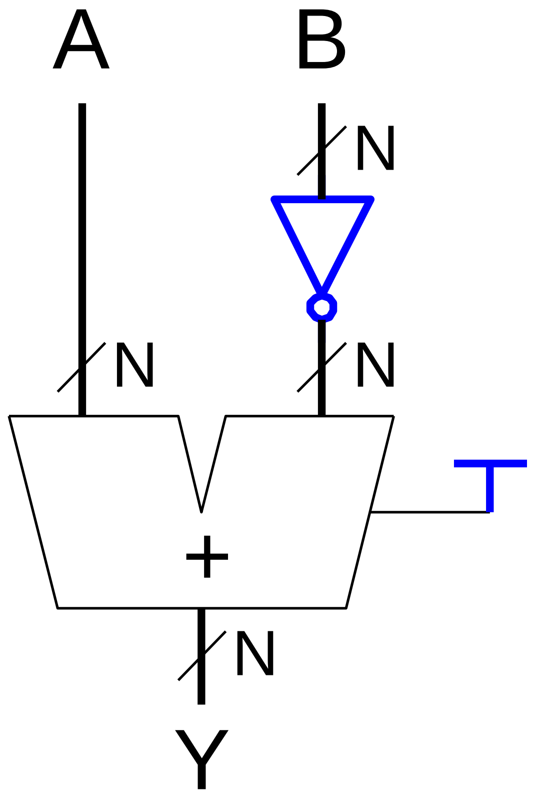
$$
\begin{aligned}
\boldsymbol{t}_{\textbf{ripple}} &= Nt_{FA} = 32(300\text{ ps}) \\
&= \mathbf{9.6\ ns} \\
\boldsymbol{t}_{\boldsymbol{CLA}} &= t_{pg} + t_{pg\_\text{block}} + (N/k - 1)t_{\text{AND\_OR}} + kt_{FA} \\
&= [100 + 600 + (7)200 + 4(300)]\text{ ps} \\
&= \mathbf{3.3\ ns} \\
\boldsymbol{t}_{\boldsymbol{PA}} &= t_{pg} + \log_2 N(t_{pg\_\text{prefix}}) + t_{\text{XOR}} \\
&= [100 + \log_2 32(200) + 100]\text{ ps} \\
&= \mathbf{1.2\ ns}
\end{aligned}
$$

Misc arithmetic circuits

Subtractor

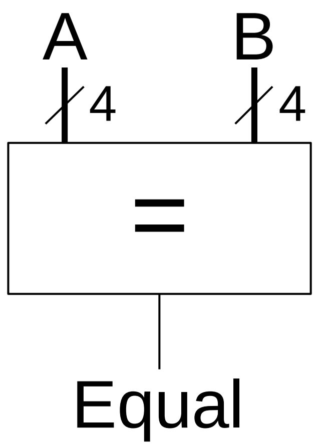
Symbol **Implementation**



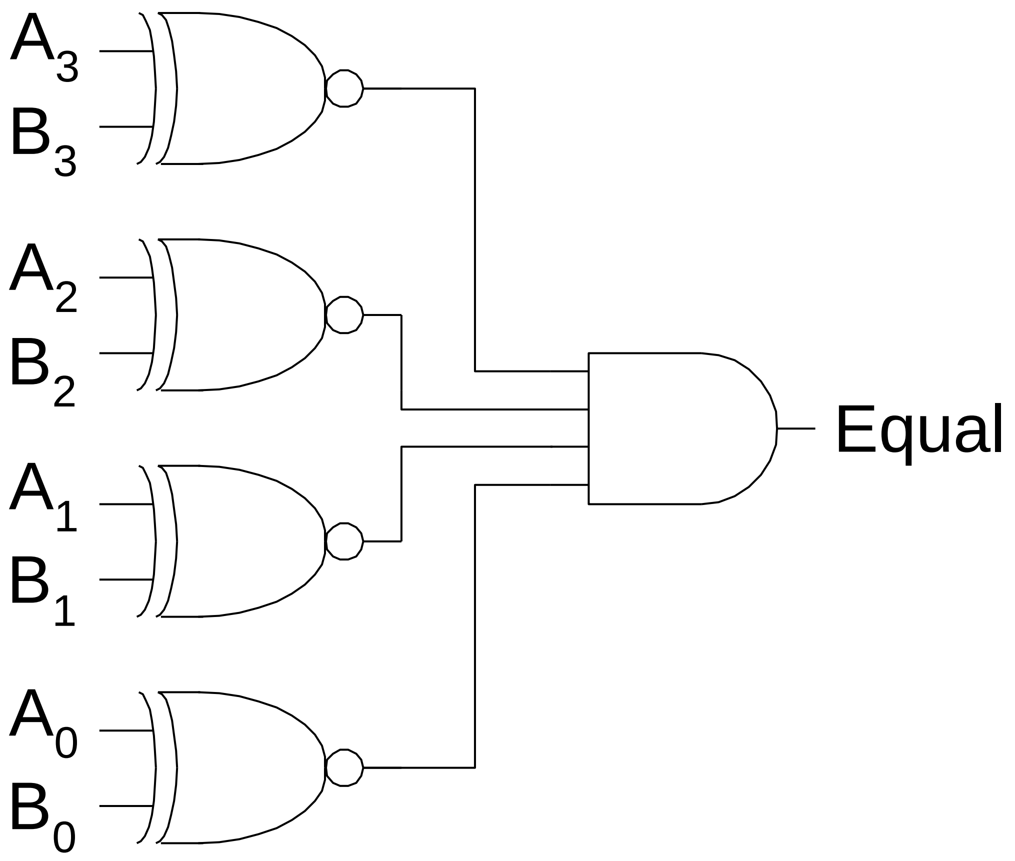


Comparator

Symbol



Implementation



Adder/subtractor circuit

- How can we implement an adder/subtractor circuit reutilizing adder's HW?