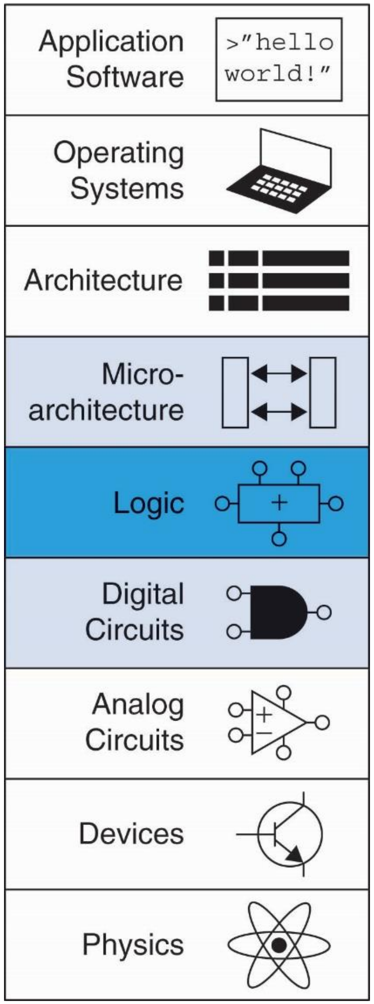
Lesson 14

Digital Logic

Junying Chen

Chapter 5 :: Topics

- Introduction
- Arithmetic Circuits
- Number Systems
- Sequential Building Blocks
- Memory Arrays
- Logic Arrays

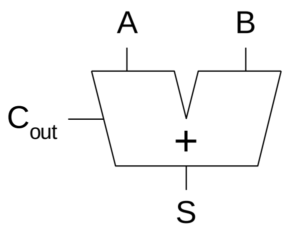


Introduction

- **Digital building blocks:**
  - Gates, multiplexers, decoders, registers, arithmetic circuits, counters, memory arrays, logic arrays
- **Building blocks demonstrate hierarchy, modularity, and regularity:**
  - Hierarchy of simpler components
  - Well-defined interfaces and functions
  - Regular structure easily extends to different sizes
- **These building blocks can be used to build microprocessor.**

1-Bit Adders

Half Adder

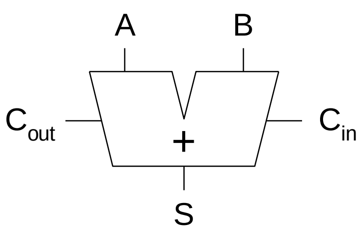


A	B	C_{out}	S
0	0		
0	1		
1	0		
1	1		

S =

C_{out} =

Full Adder



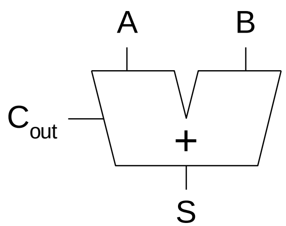
C_{in}	A	B	C_{out}	S
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

S =

C_{out} =

1-Bit Adders

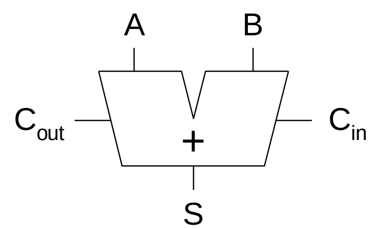
Half Adder



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

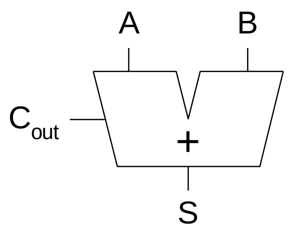


C_{in}	A	B	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

S =
C_{out} =

1-Bit Adders

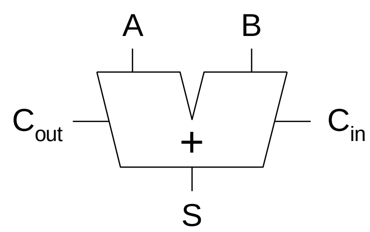
Half Adder



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



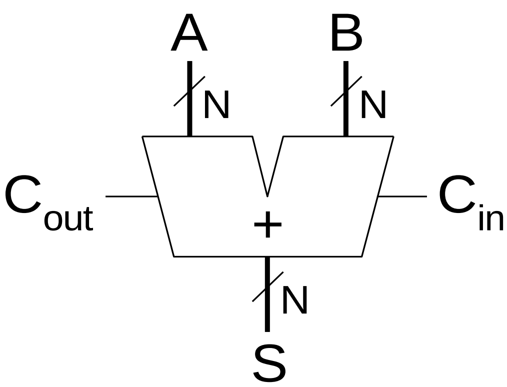
C_{in}	A	B	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

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

Multibit Adders (CPAs)

- 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

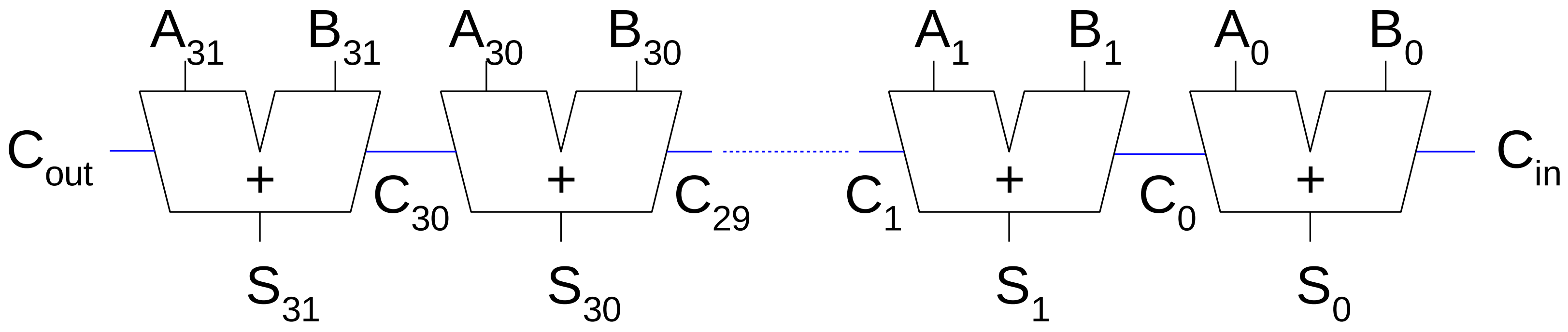
Symbol



ELSEVIER

Ripple-Carry Adder

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



Ripple-Carry Adder Delay

$$t_{\mathrm{ripple}} = Nt_{FA}$$

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

Carry-Lookahead Adder

- Compute carry out (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:
    - Column i will generate a carry out if A_i AND B_i are both 1.

$$G_i = A_i B_i$$

    - Column i will propagate a carry in to the carry out if A_i OR B_i is 1.

$$P_i = A_i + B_i$$

    - 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 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 block

Carry-Lookahead Adder

- **Example:** 4-bit blocks ($G_{3:0}$ and $P_{3:0}$) :

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

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

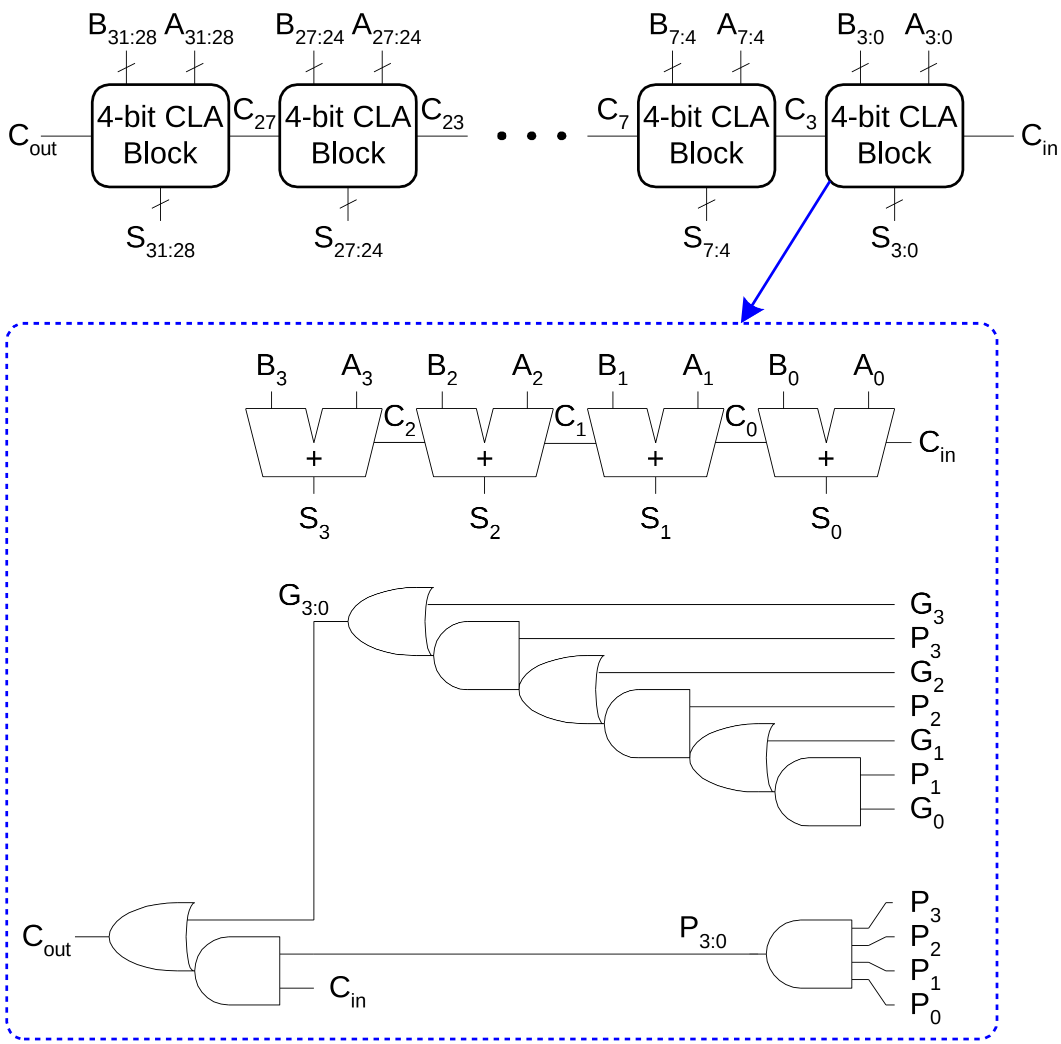
- **Generally,**

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

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

$$C_i = G_{i:j} + P_{i:j} C_j$$

32-bit CLA with 4-bit Blocks



ELSEVIER

Carry-Lookahead Adder Delay

For N-bit CLA with k-bit blocks:

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

- t_{pg} : delay to generate all P_i, G_i
- t_{pg_block} : delay to generate all $P_{i:j}$, $G_{i:j}$
- 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:

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

- Computes 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 in 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 to 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 $G_{0:-1}$, $G_{1:-1}$, $G_{2:-1}$, $G_{3:-1}$, $G_{4:-1}$, $G_{5:-1}$, … (called ***prefixes***)

Prefix Adder*

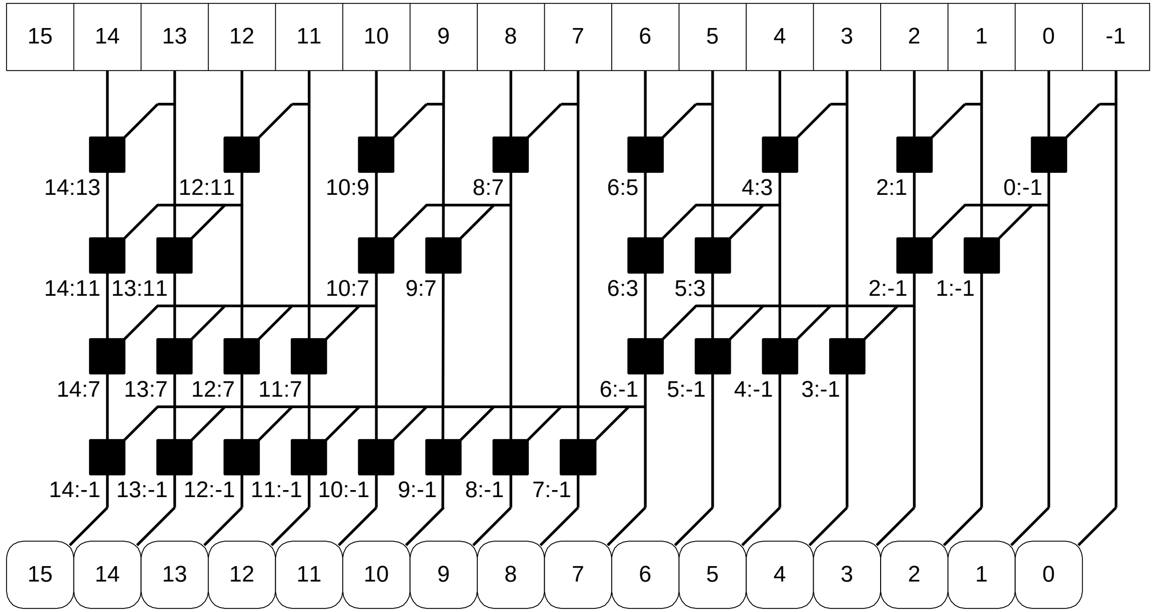
- Generate and propagate signals for a block spanning bits *i*:*j*:

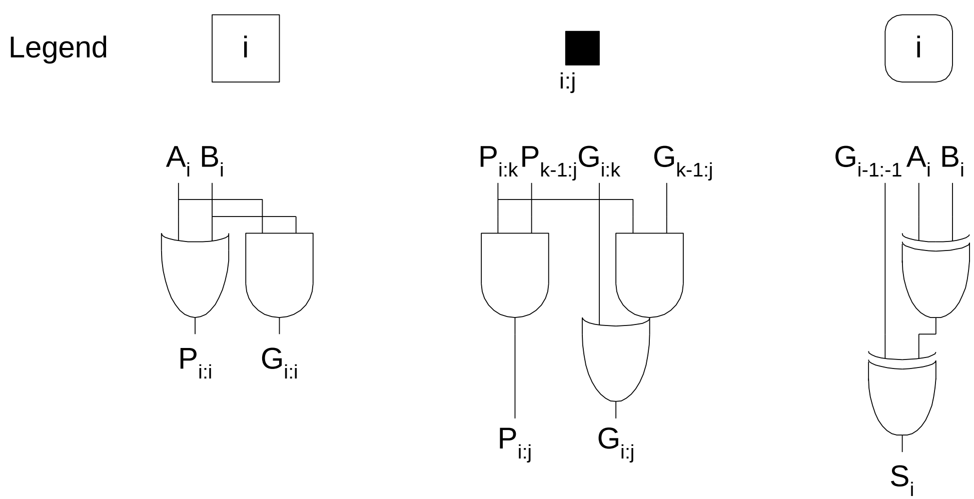
$$G_{i:j} = G_{i:k} + P_{i:k}\, G_{k\text{-}1:j}$$

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

- In words:
  - **Generate:** block *i*:*j* will generate a carry if:
    - upper part (*i*:*k*) generates a carry or
    - upper part 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

Prefix Adder Schematic*





Prefix Adder Delay

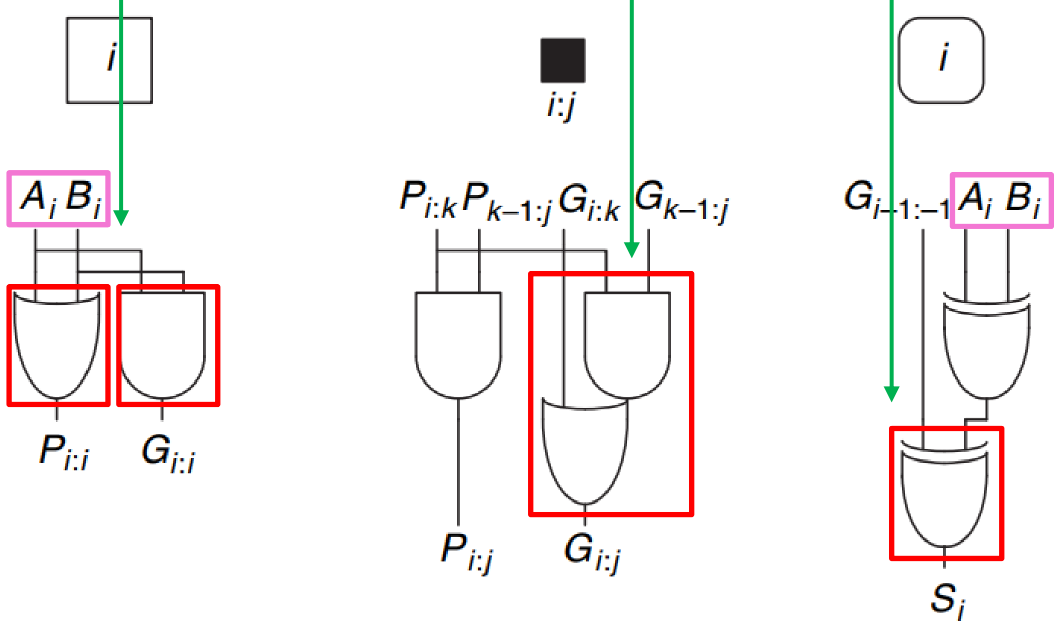
$$t_{PA} = t_{pg} + \log_2 N(t_{pg\_prefix}) + t_{XOR}$$

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

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 Comparisons

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

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

Adder Delay Comparisons

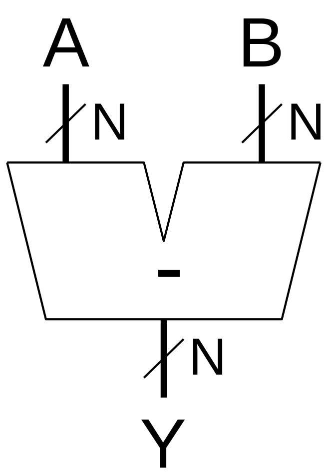
Compare delay of: 32-bit ripple-carry, carry-lookahead, and prefix adders

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

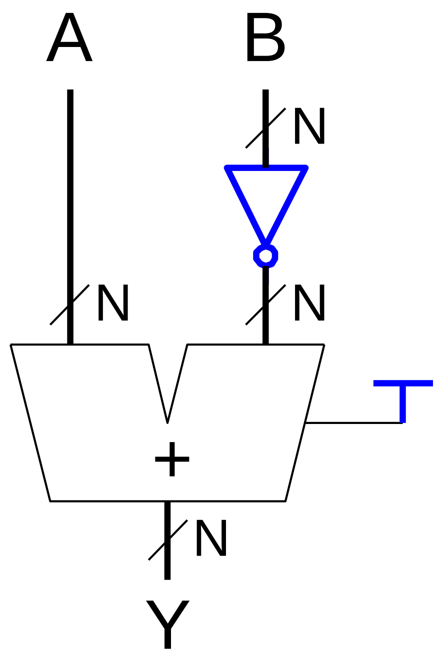
$$
\begin{aligned}
t_{\text{ripple}} &= Nt_{FA} = 32(300 \text{ ps}) \\
&= \mathbf{9.6\ ns} \\
t_{CLA} &= t_{pg} + t_{pg\_\text{block}} + (N/k - 1)t_{\text{AND\_OR}} + kt_{FA} \\
&= [100 + 600 + (7)200 + 4(300)] \text{ ps} \\
&= \mathbf{3.3\ ns} \\
t_{PA} &= t_{pg} + \log_2 N(t_{pg\_\text{prefix}}) +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

Subtractor

Symbol

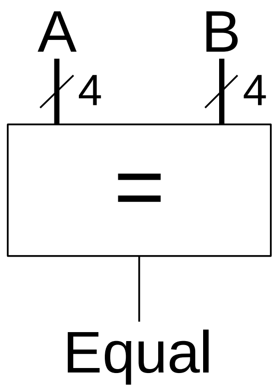


Implementation

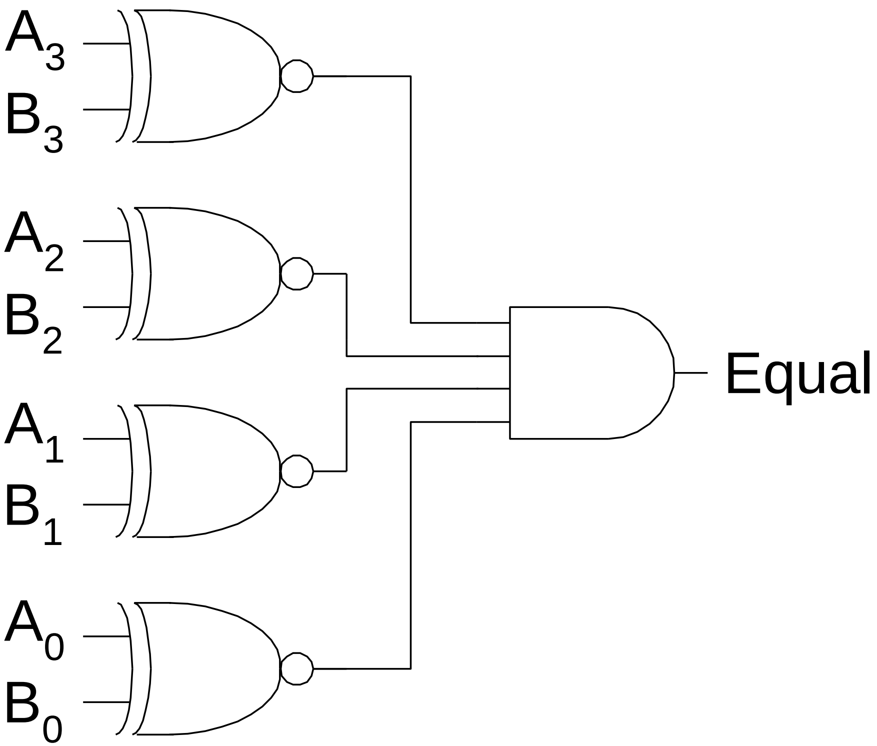


Comparator: Equality

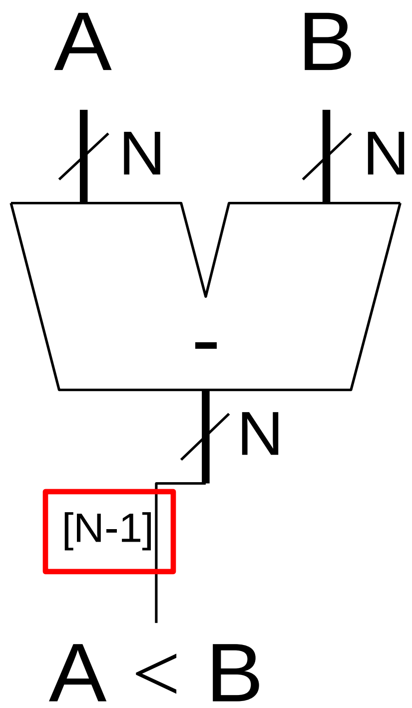
Symbol



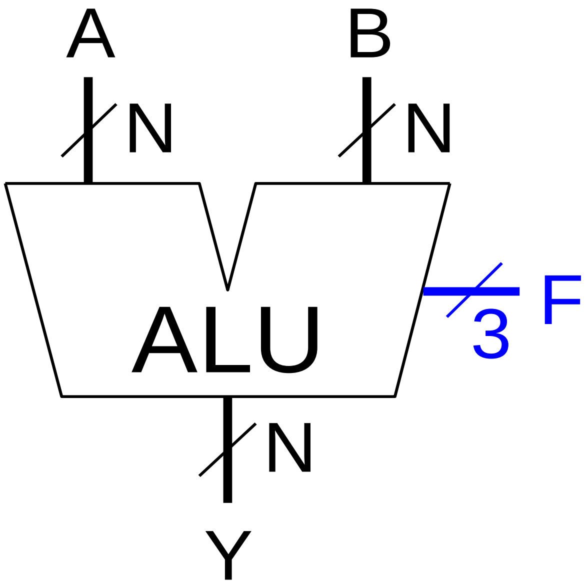
Implementation



Comparator: Less Than

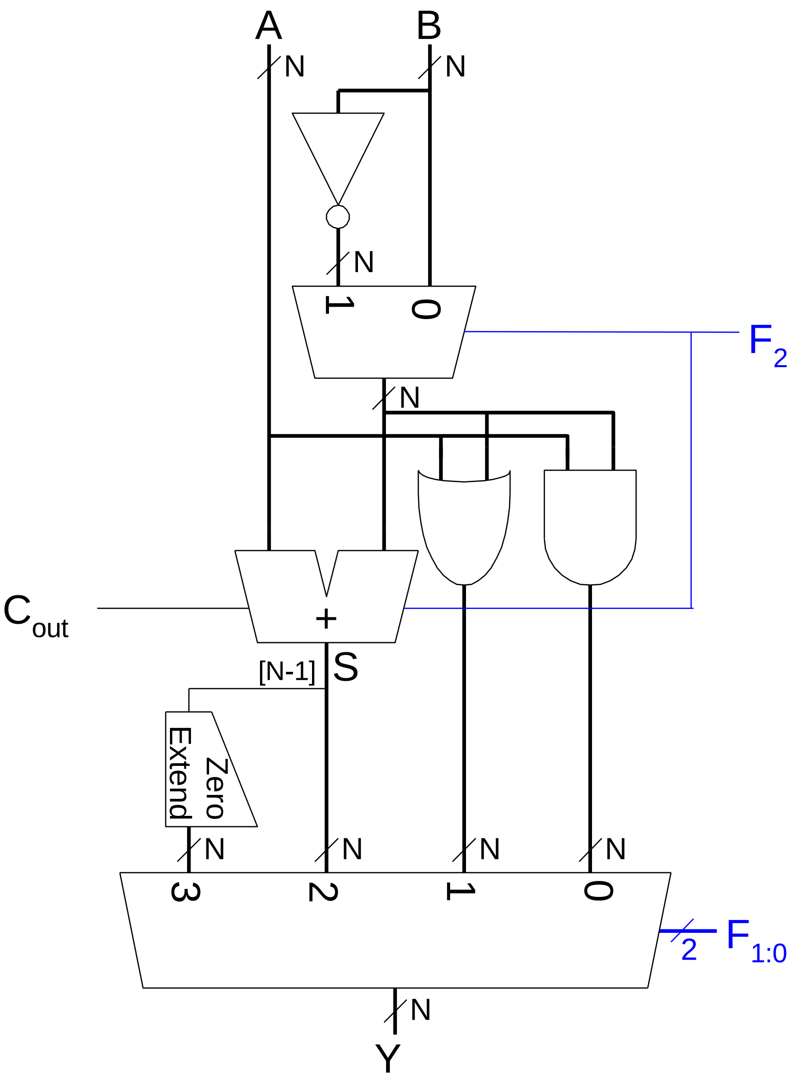


Arithmetic Logic Unit (ALU)



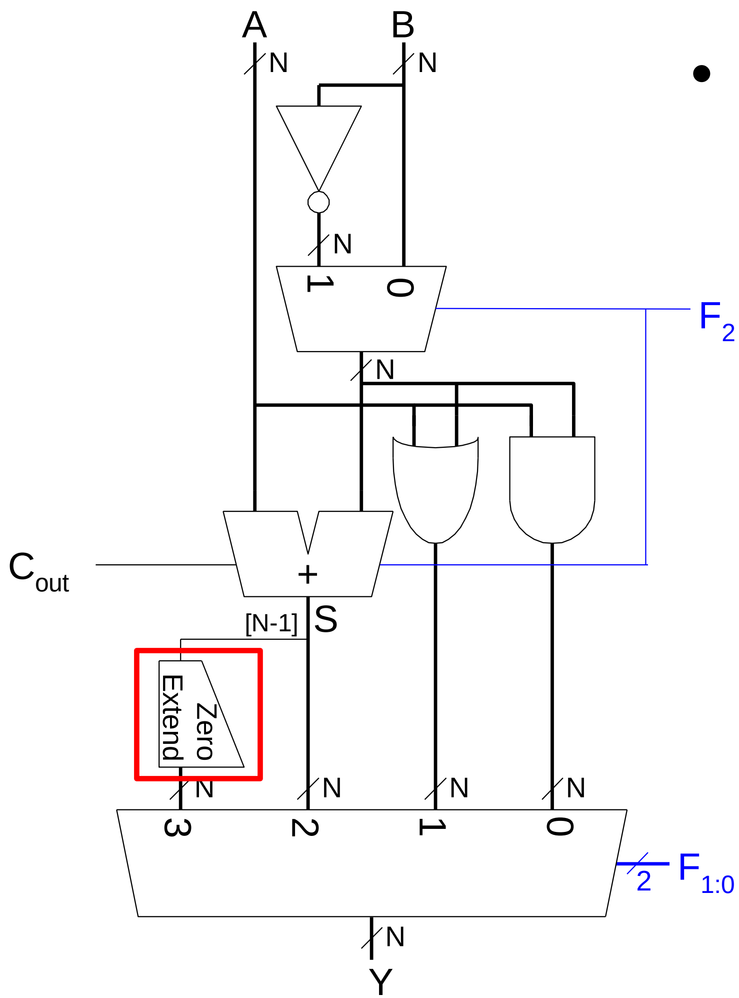
$F_{2:0}$	Function
000	A & B
001	A | B
010	A + B
011	not used
100	A & ~B
101	A | ~B
110	A - B
111	SLT

ALU Design



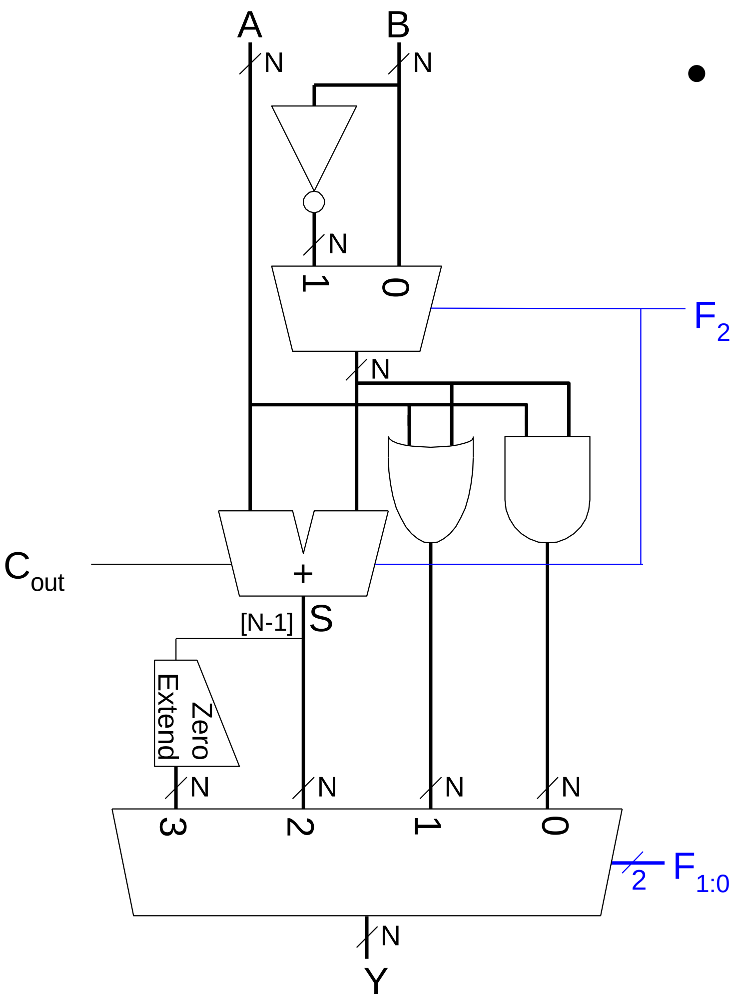
$F_{2:0}$	Function
000	A & B
001	A | B
010	A + B
011	not used
100	A & ~B
101	A | ~B
110	A - B
111	SLT

Set Less Than (SLT) Example



- Configure 32-bit ALU for SLT operation: $A = 25$ and $B = 32$

Set Less Than (SLT) Example



- Configure 32-bit ALU for SLT operation: A = 25 and B = 32
  - ***A* < *B***, so Y should be 32-bit representation of 1 (0x00000001)
  - $\boldsymbol{F_{2:0}}$ **= 111**
    - $\boldsymbol{F_2}$ **= 1** (adder acts as subtractor), so 25 - 32 = -7
    - -7 has 1 in the most significant bit (S_{31} = 1)
    - $\boldsymbol{F_{1:0}}$ **= 11** multiplexer selects $Y = S_{31}$ (zero extended) = 0x00000001.

Shifters

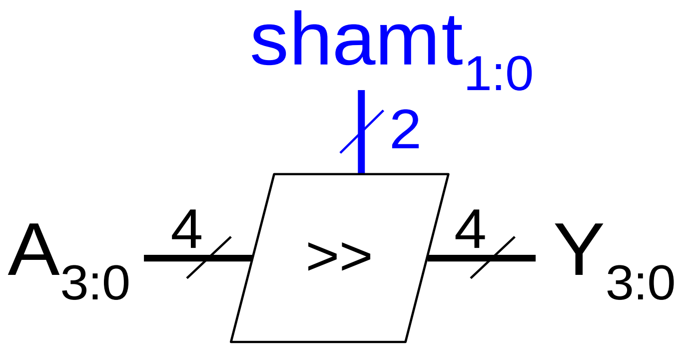
- **Logical shifter:** shifts value to left or right and fills empty spaces with 0's
  - Ex: 11001 >> 2 =
  - Ex: 11001 << 2 =
- **Arithmetic shifter:** same as logical shifter, but on right shift, fills empty spaces with the old most significant bit (msb).
  - Ex: 11001 >>> 2 =
  - Ex: 11001 <<< 2 =
- **Rotator:** rotates bits in a circle, such that bits shifted off one end are shifted into the other end
  - Ex: 11001 ROR 2 =
  - Ex: 11001 ROL 2 =

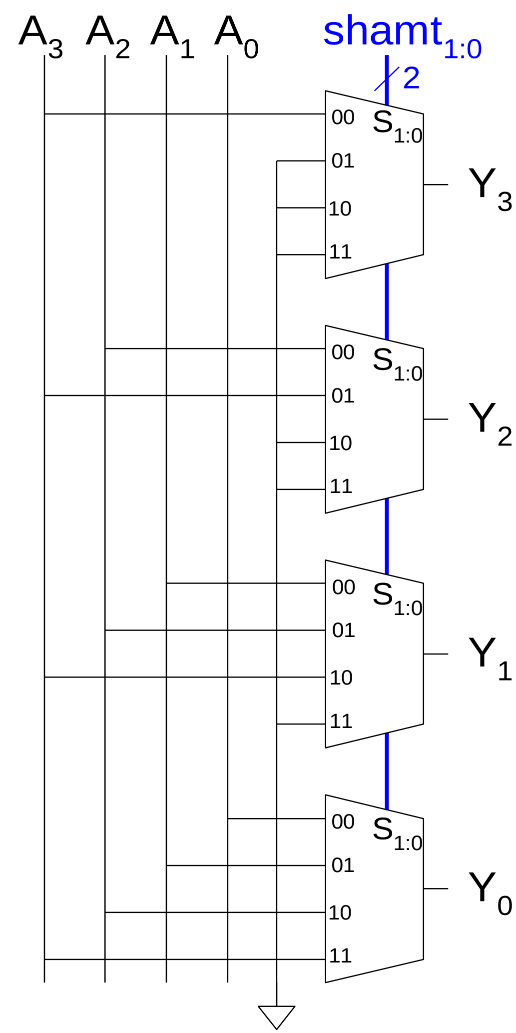
DIGITAL BUILDING BLOCKS

Shifters

- **Logical shifter:**
  - Ex: 11001 >> 2 = 00110
  - Ex: 11001 << 2 = 00100
- **Arithmetic shifter:**
  - Ex: 11001 >>> 2 = 11110
  - Ex: 11001 <<< 2 = 00100
- **Rotator:**
  - Ex: 11001 ROR 2 = 01110
  - Ex: 11001 ROL 2 = 00111

Shifter Design





Shifters as Multipliers, Dividers

- $A << N = A \times 2^N$
  - **Example:** 00001 << 2 = 00100 ($1 \times 2^2 = 4$)
  - **Example:** 11101 << 2 = 10100 ($-3 \times 2^2 = -12$)
- $A >>> N = A \div 2^N$
  - **Example:** 01000 >>> 2 = 00010 ($8 \div 2^2 = 2$)
  - **Example:** 10000 >>> 2 = 11100 ($-16 \div 2^2 = -4$)