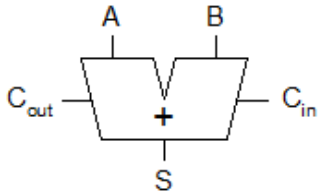
CSCE 611
Advanced Digital Design

Digital Building Blocks

Integer Addition

Full Adder

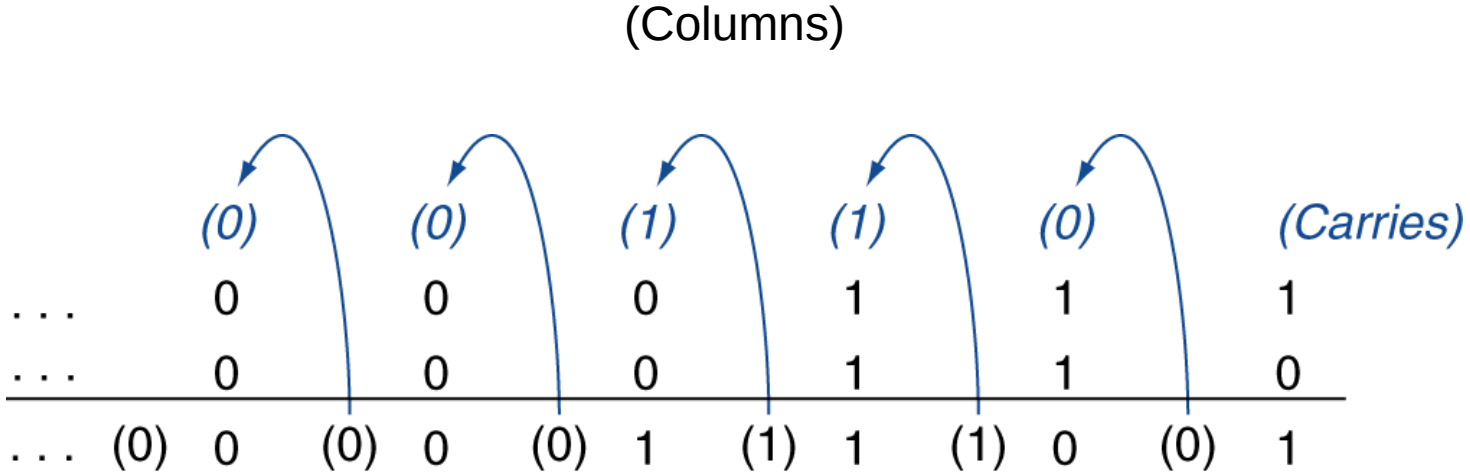


C_{in}	A	B	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

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

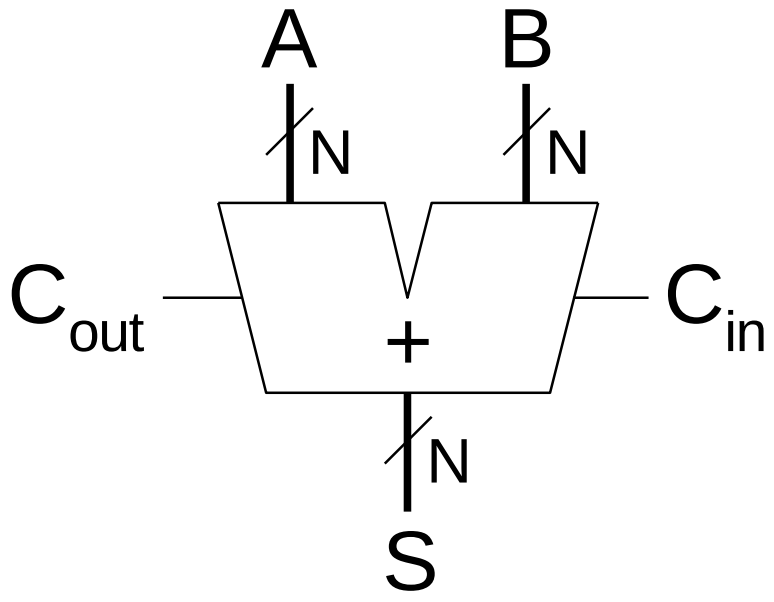
Integer Addition

- Example: 7 + 6



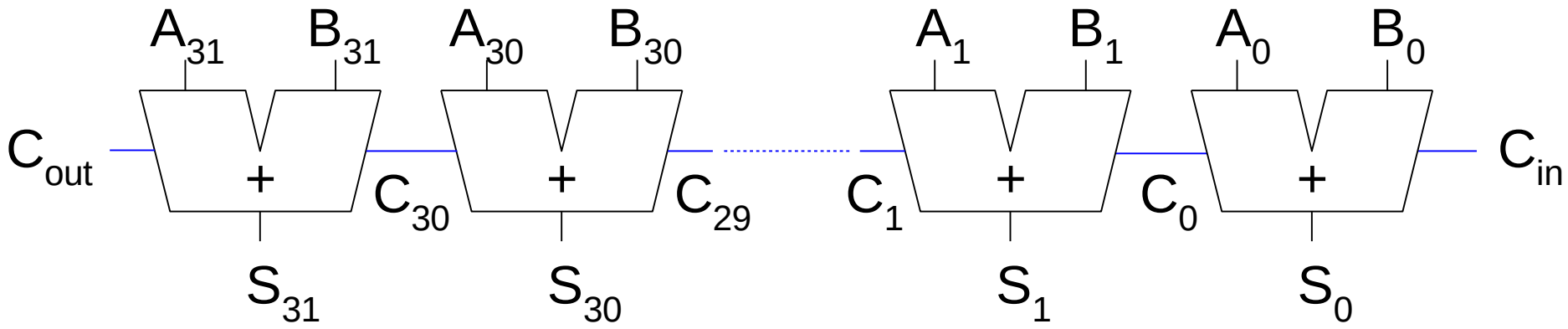
Multibit Adders (CPAs)

- Types of carry propagate adders (CPAs):
  1. Ripple-carry (slow)
  2. Carry-lookahead (fast)
  3. Prefix (faster)



Ripple-Carry Adder

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



Ripple-Carry Adder Delay

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

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:

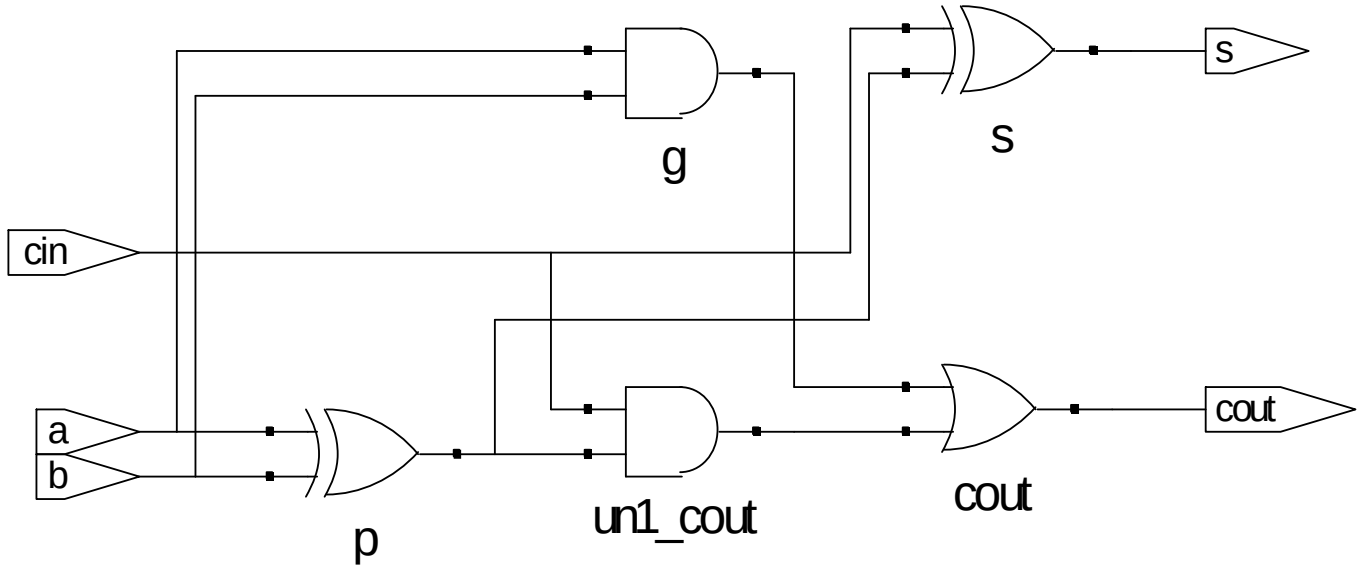
$$C_i = A_i B_i + (A_i + B_i) C_{i-1} = G_i + P_i C_{i-1}$$

Propagate and Generate Logic

```
module fulladder(input logic a, b, cin,
                 output logic s, cout);
  logic p, g; // internal nodes

  assign p = a ^ b;
  assign g = a & b;

  assign s = p ^ cin;
  assign cout = g | (p & cin);
endmodule
```



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

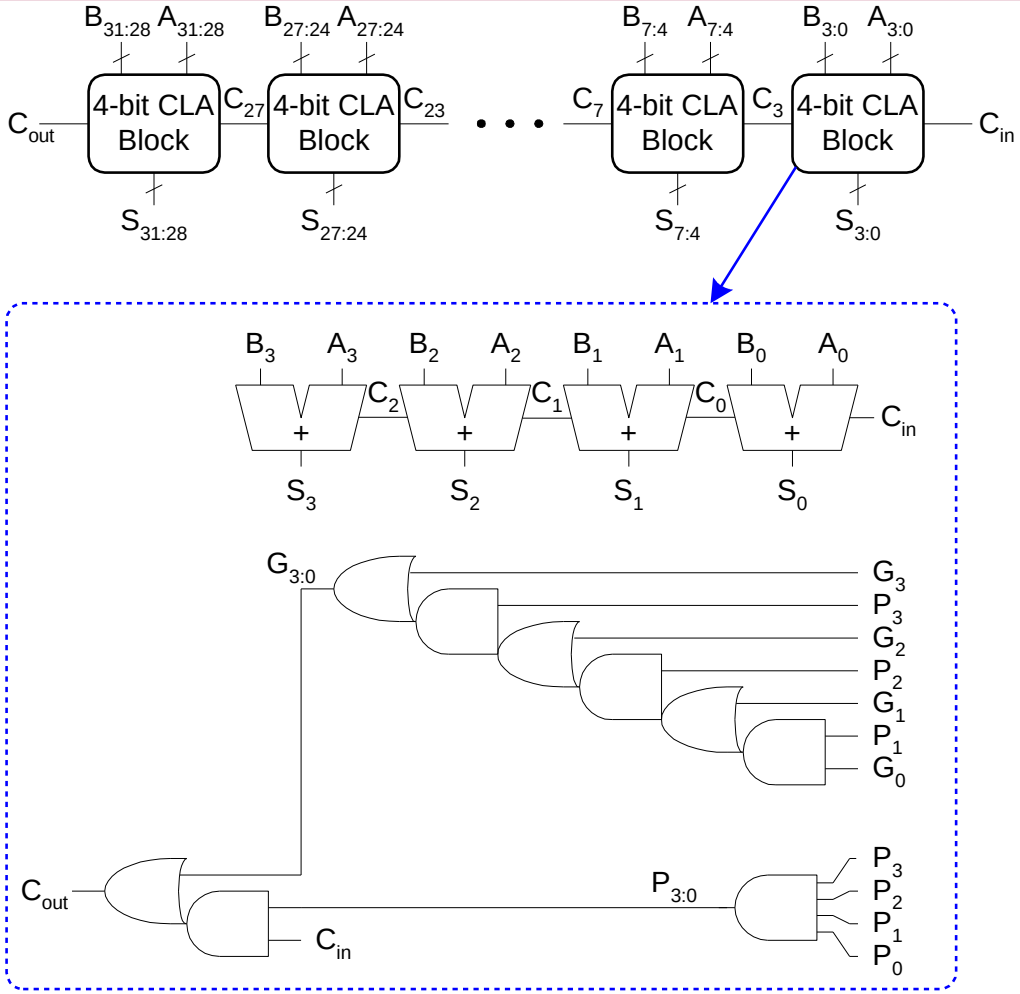
- **Example:** 4-bit blocks ($G_{3:0}$ and $P_{3:0}$):

$$G_{3:0} = G_3 + P_3 (G_2 + P_2 (G_1 + P_1 G_0))$$
$$P_{3:0} = P_3 P_2 P_1 P_0$$

- **Generally,**

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

32-bit CLA with 4-bit Blocks



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-1}) for each column, then computes sum:

$$\boldsymbol{S_i = (A_i \oplus B_i) \oplus C_i}$$

- Carry in to column i = carry out of column i-1:

$$\boldsymbol{C_i = G_{i-1:-1}}$$

$G_{i-1:-1}$: generate signal spanning columns i-1 to -1

- Sum equation:

$$\boldsymbol{S_i = (A_i \oplus B_i) \oplus G_{i-1:-1}}$$

- **Goal:** Quickly compute $G_{0:-1}$, $G_{1:-1}$, $G_{2:-1}$, $G_{3:-1}$, $G_{4:-1}$, $G_{5:-1}$, … (called ***prefixes***)

- Column -1 holds C_{in}, so $\boldsymbol{G_{-1} = C_{in}}$, $\boldsymbol{P_{-1} = 0}$

Prefix Adder

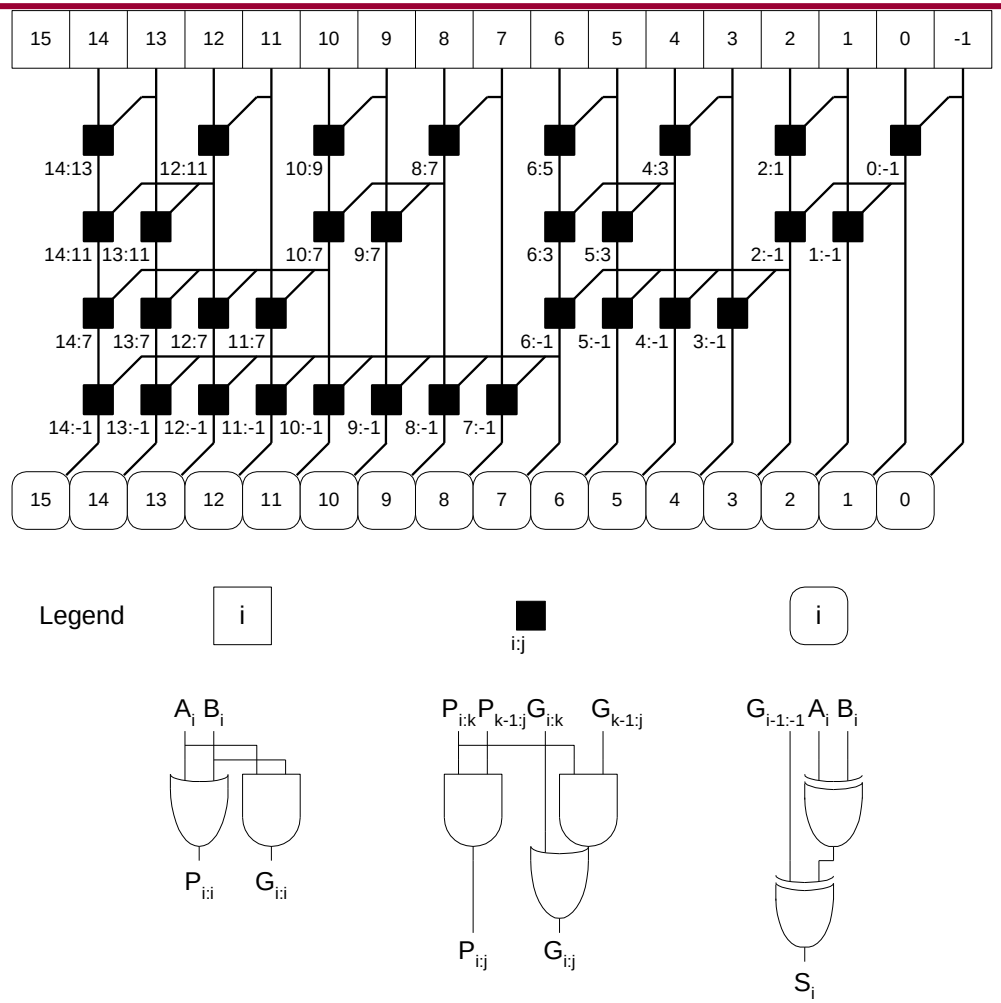
- Generate and propagate signals for a block spanning bits i:j:

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

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

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



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

- 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

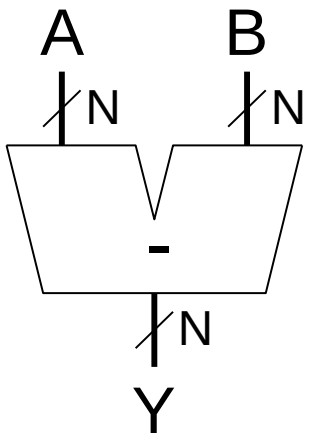
Compare delay of: 32-bit ripple-carry, carry-lookahead, and prefix adders

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

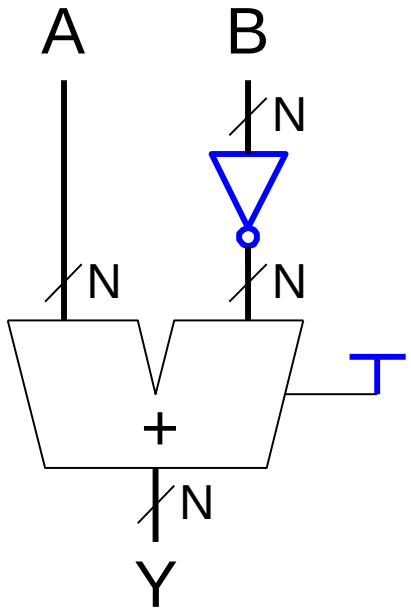
$$
\begin{aligned}
t_{\text{ripple}} &= Nt_{FA} = 32(300\text{ ps}) \\
&= \mathbf{9.6\text{ ns}} \\
t_{CLA} &= t_{pg} + t_{pg_\text{block}} + (N/k - 1)t_{\text{AND_OR}} + kt_{FA} \\
&= [100 + 700 + (7)200 + 4(300)]\text{ ps} \\
&= \mathbf{3.4\text{ ns}} \\
t_{PA} &= t_{pg} + \log_2 N(t_{pg_\text{prefix}}) + t_{\text{XOR}} \\
&= [100 + \log_2 32(200) + 100]\text{ ps} \\
&= \mathbf{1.2\text{ ns}}
\end{aligned}
$$

Subtracter

Symbol

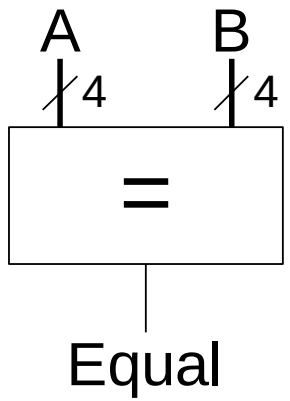


Implementation

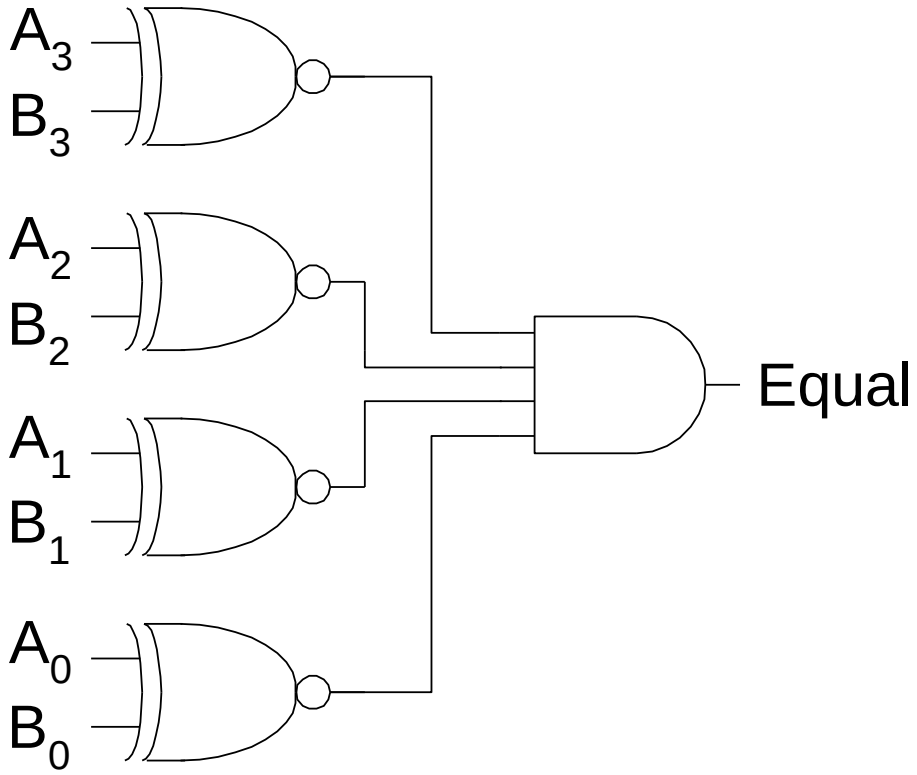


Comparator: Equality

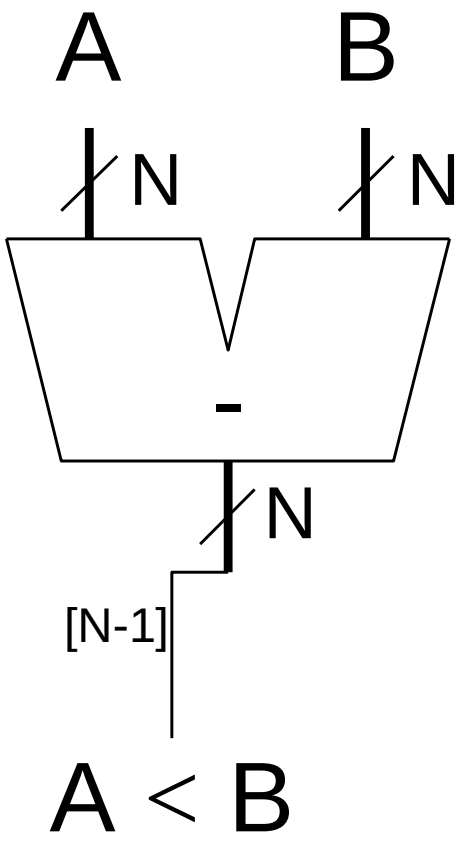
Symbol



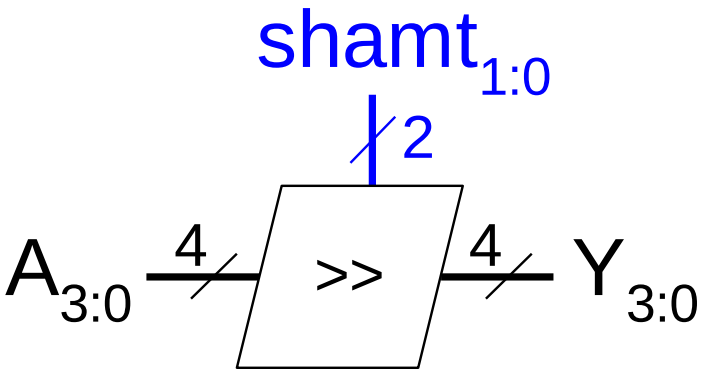
Implementation

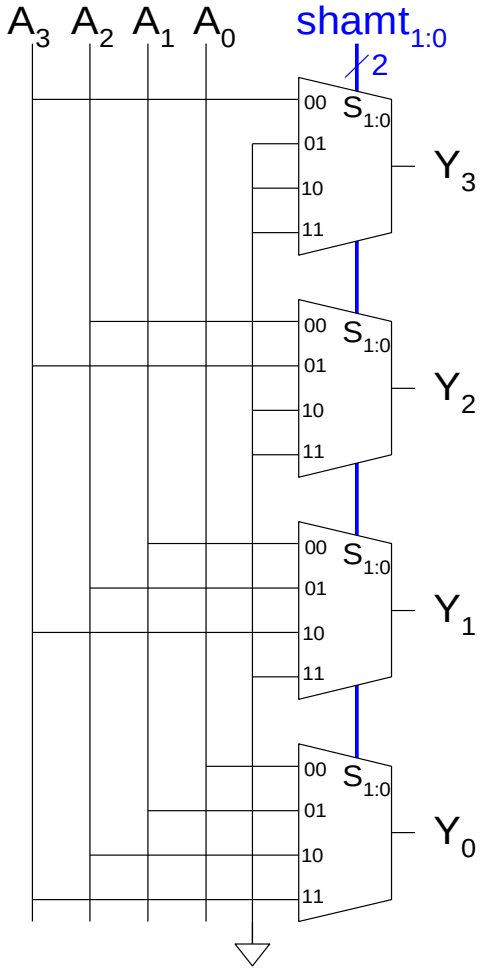


Comparator: Less Than



Shifter Design

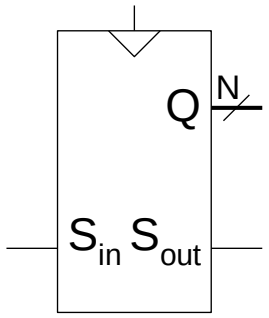




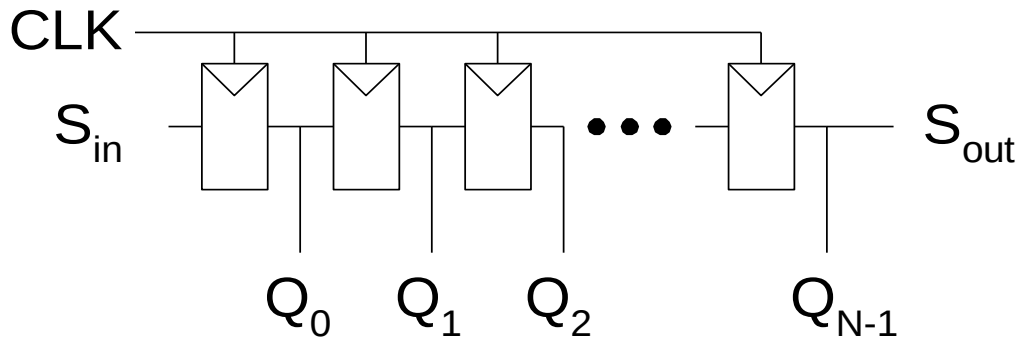
Shift Registers

- Shift a new bit in on each clock edge
- Shift a bit out on each clock edge
- *Serial-to-parallel converter*: converts serial input (S_{in}) to parallel output ($Q_{0:N-1}$)

Symbol:

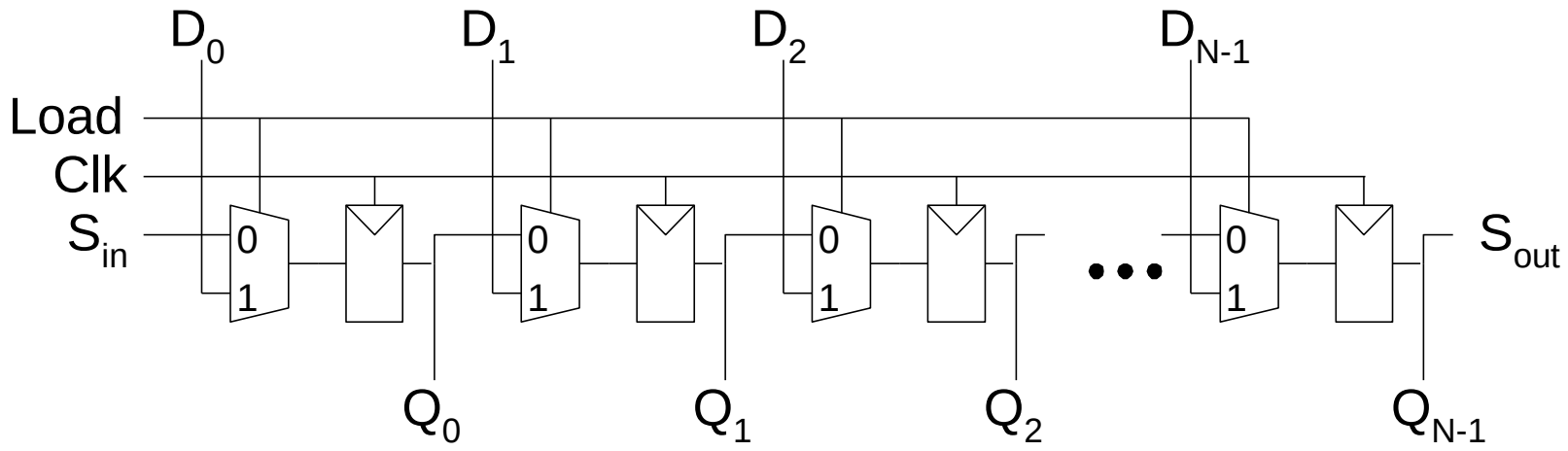


Implementation:



Shift Register with Parallel Load

- When *Load* = 1, acts as a normal *N*-bit register
- When *Load* = 0, acts as a shift register
- Now can act as a *serial-to-parallel converter* (S_{in} to $Q_{0:N-1}$) or a *parallel-to-serial converter* ($D_{0:N-1}$ to S_{out})



Shifters as Multipliers, Dividers

- ***A << N = A* × 2^N**
  - **Example:** 00001 << 2 = 00100 ($1 \times 2^2 = 4$)
  - **Example:** 11101 << 2 = 10100 ($-3 \times 2^2 = -12$)

- ***A >>> N = A* ÷ 2^N**
  - **Example:** 01000 >>> 2 = 00010 ($8 \div 2^2 = 2$)
  - **Example:** 10000 >>> 2 = 11100 ($-16 \div 2^2 = -4$)

Multipliers

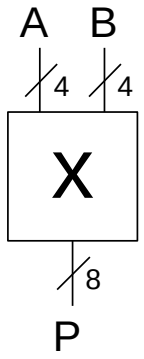
- **Partial products** formed by multiplying a single digit of the multiplier with multiplicand
- **Shifted** partial products **summed** to form result

Decimal		Binary
230	multiplicand	0101
x 42	multiplier	x 0111
460	partial products	0101
+ 920		0101
		0101
		+ 0000
9660	result	0100011

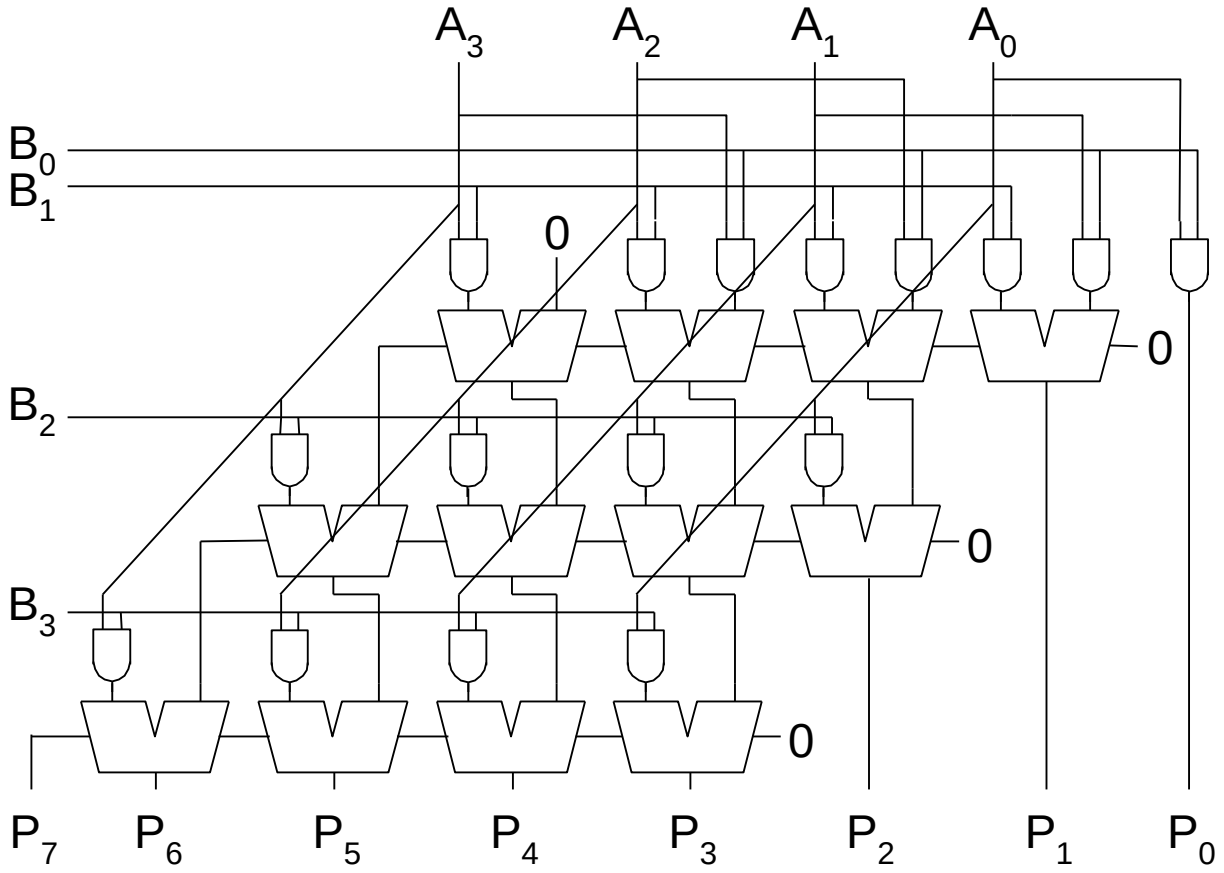
230 x 42 = 9660

5 x 7 = 35

4 x 4 Multiplier

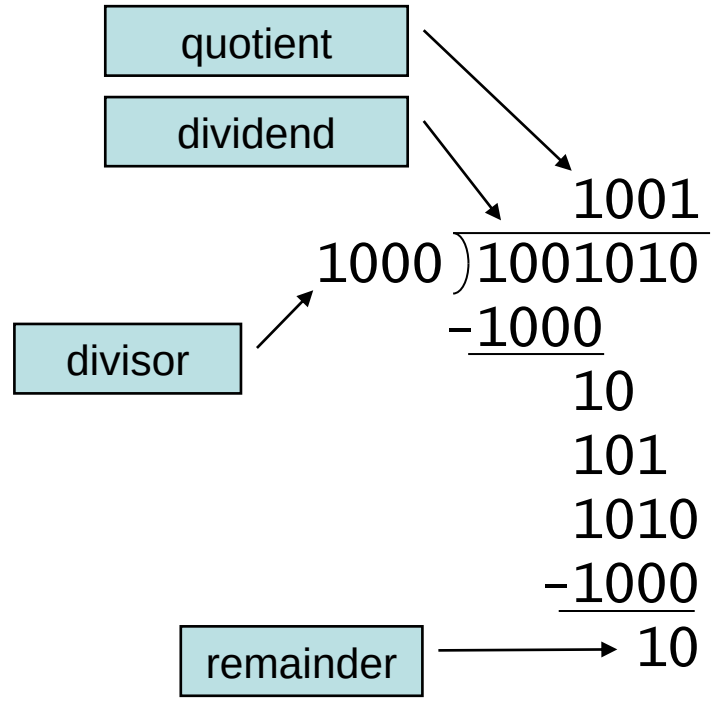


$$
\begin{array}{rcccccccc}
 & & & A_3 & A_2 & A_1 & A_0 & & \\
\times & & & B_3 & B_2 & B_1 & B_0 & & \\
\hline
 & & & A_3B_0 & A_2B_0 & A_1B_0 & A_0B_0 & & \\
 & & A_3B_1 & A_2B_1 & A_1B_1 & A_0B_1 & & & \\
 & A_3B_2 & A_2B_2 & A_1B_2 & A_0B_2 & & & & \\
+ & A_3B_3 & A_2B_3 & A_1B_3 & A_0B_3 & & & & \\
\hline
P_7 & P_6 & P_5 & P_4 & P_3 & P_2 & P_1 & P_0 &
\end{array}
$$



Division

- Check for 0 divisor
- Long division approach
  - If divisor ≤ dividend bits
    - 1 bit in quotient, subtract
  - Otherwise
    - 0 bit in quotient, bring down next dividend bit
- Restoring division
  - Do the subtract, and if remainder goes < 0, add divisor back
- Signed division
  - Divide using absolute values
  - Adjust sign of quotient and remainder as required



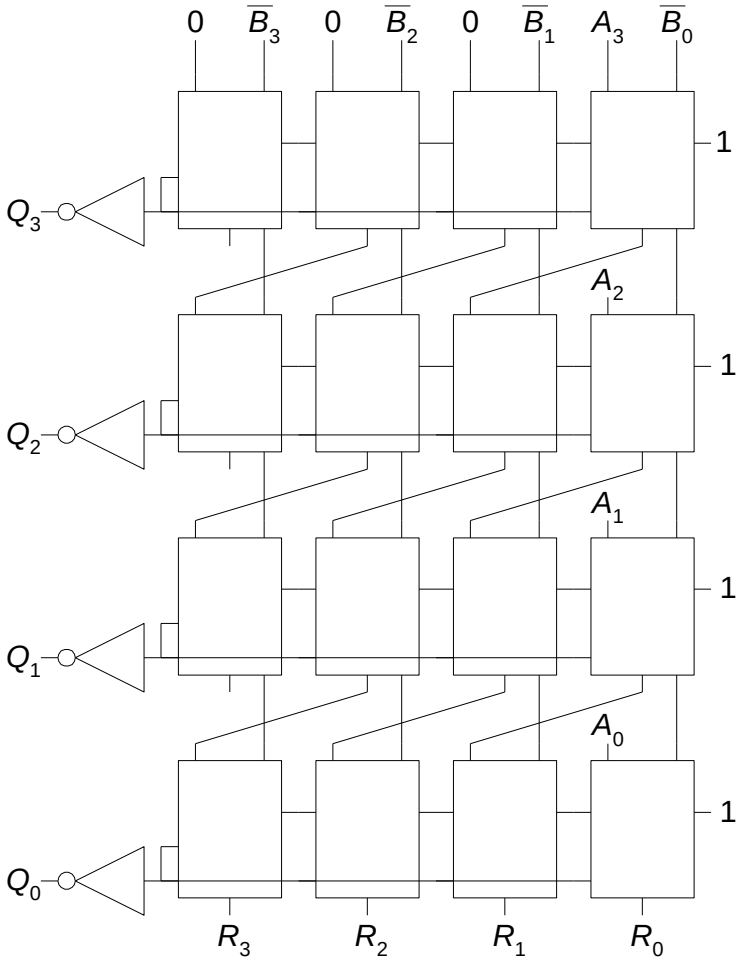
n-bit operands yield *n*-bit quotient and remainder

Restoring Division

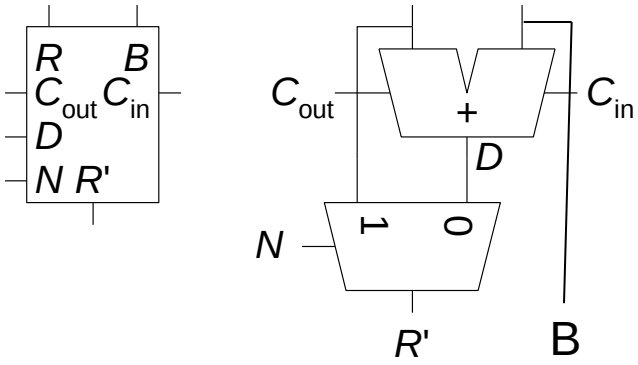
$$\begin{array}{r} 1 \\ 1000\overline{)1001010} \\ -1000 \end{array} \qquad \begin{array}{r} 1 \\ 1000\overline{)1001010} \\ -1000 \end{array} \qquad \begin{array}{r} 1 \\ 1000\overline{)1001010} \\ -1000 \end{array} \qquad \begin{array}{r} 1 \\ 1000\overline{)1001010} \\ -1000 \end{array}$$

$$\begin{array}{r} 11 \\ 1000\overline{)0001010} \\ -1000 \end{array} \qquad \begin{array}{r} 101 \\ 1000\overline{)0001010} \\ -1000 \end{array} \qquad \begin{array}{r} 1001 \\ 1000\overline{)0001010} \\ -1000 \end{array} \qquad \begin{array}{r} 1001 \\ 1000\overline{)0000010} \end{array}$$

4 x 4 Divider

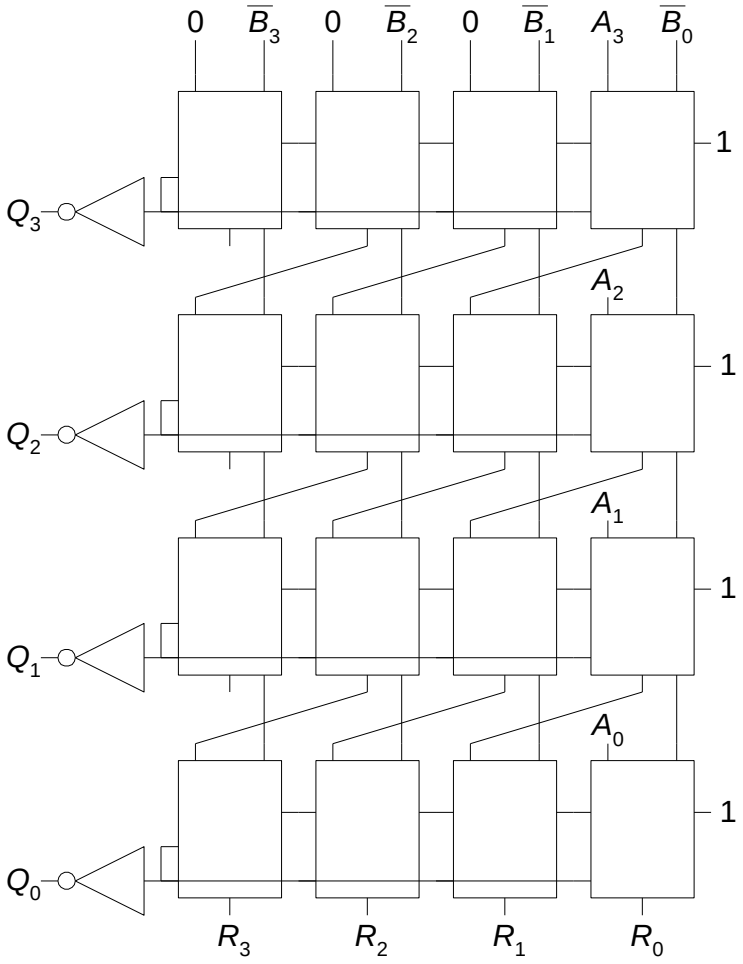


Legend

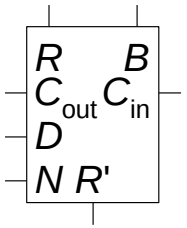


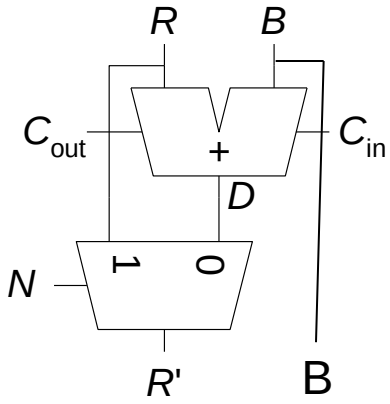
$$A/B = Q + R/B$$

4 x 4 Divider



Legend

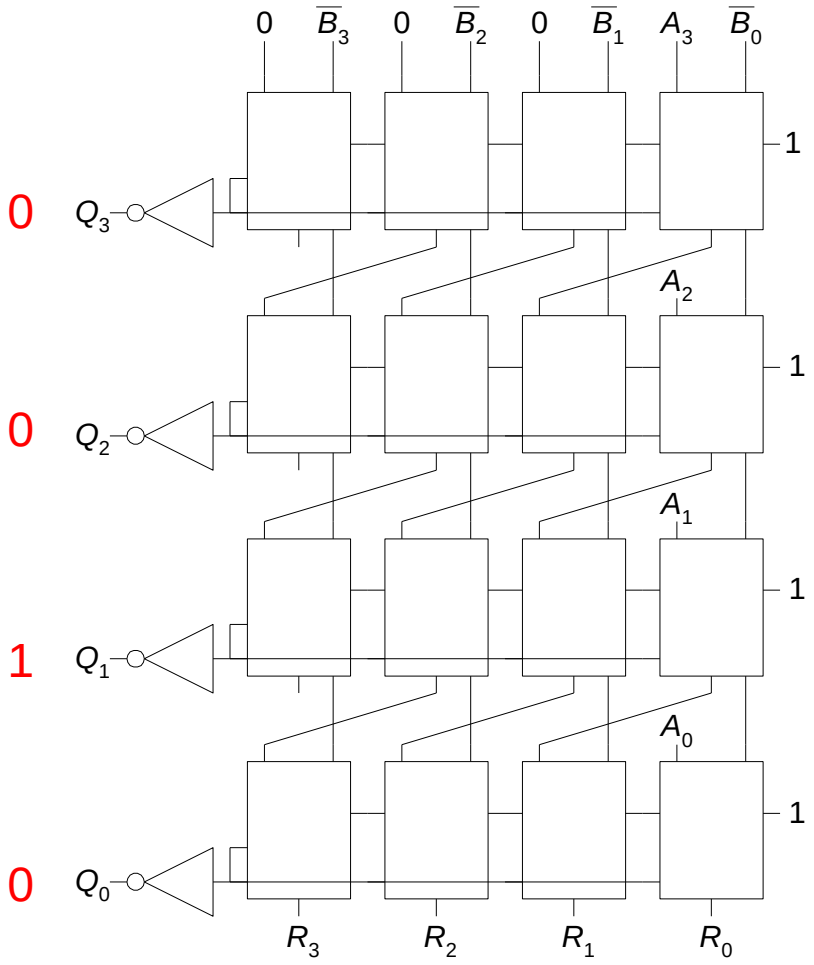




A/B = Q + R/B

11/5 = 2 + 1/5

4 x 4 Divider



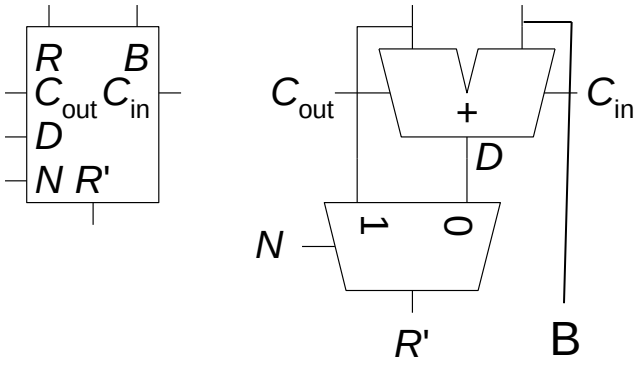
0001 – 0101
= -4
= (r) 0001

0010 – 0101
= -3
= (r) 0010

0101 – 0101
= 0

0001 – 0101
= -4
= (r) 0001

Legend

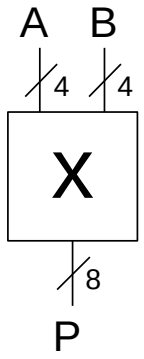


A/B = Q + R/B

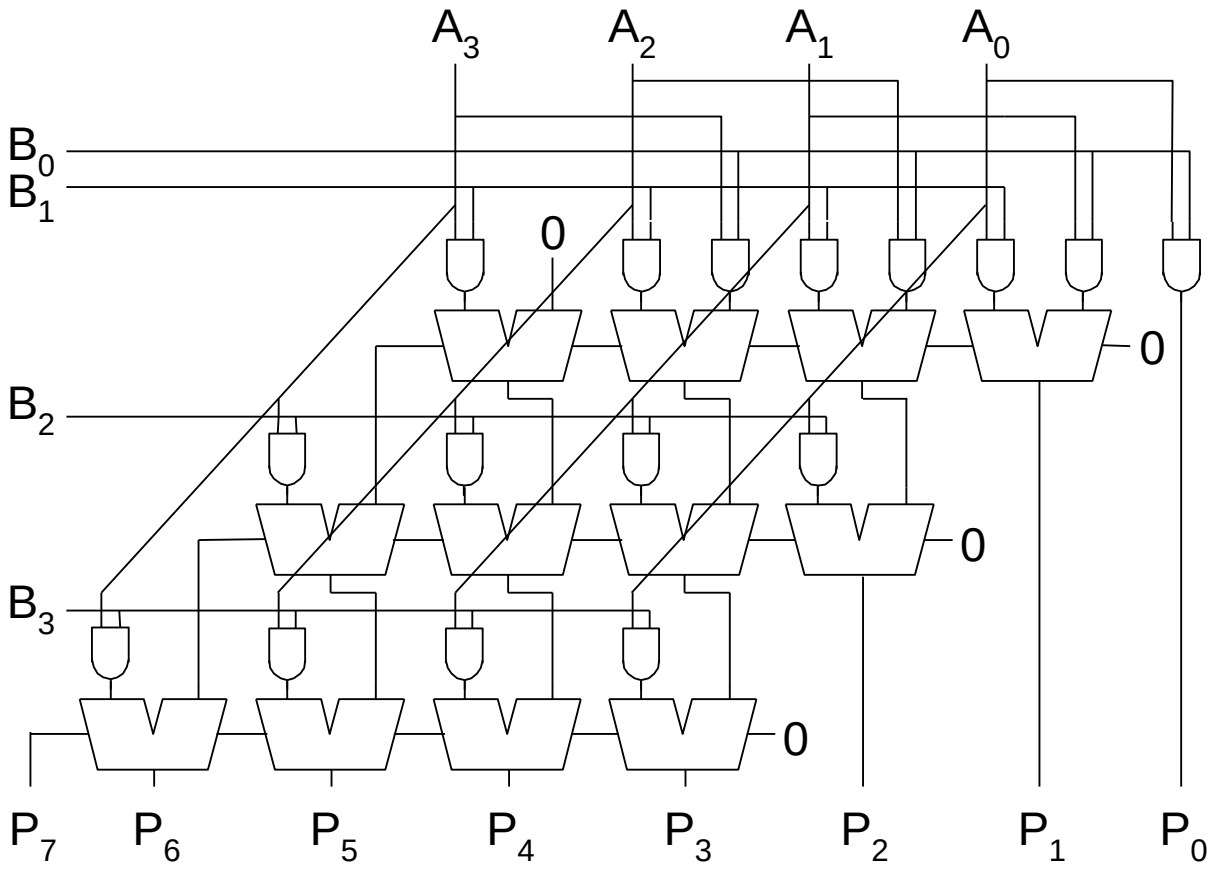
$11_{10}/5_{10} = 2_{10} + 1_{10}/5_{10}$

$A = 1011_2$
$B = 0101_2$

4 x 4 Multiplier



$$
\begin{array}{rcccccccc}
 & & & A_3 & A_2 & A_1 & A_0 & & \\
\times & & & B_3 & B_2 & B_1 & B_0 & & \\
\hline
 & & & A_3B_0 & A_2B_0 & A_1B_0 & A_0B_0 & & \\
 & & A_3B_1 & A_2B_1 & A_1B_1 & A_0B_1 & & & \\
 & A_3B_2 & A_2B_2 & A_1B_2 & A_0B_2 & & & & \\
+ & A_3B_3 & A_2B_3 & A_1B_3 & A_0B_3 & & & & \\
\hline
P_7 & P_6 & P_5 & P_4 & P_3 & P_2 & P_1 & P_0
\end{array}
$$



Faster Multiplication

