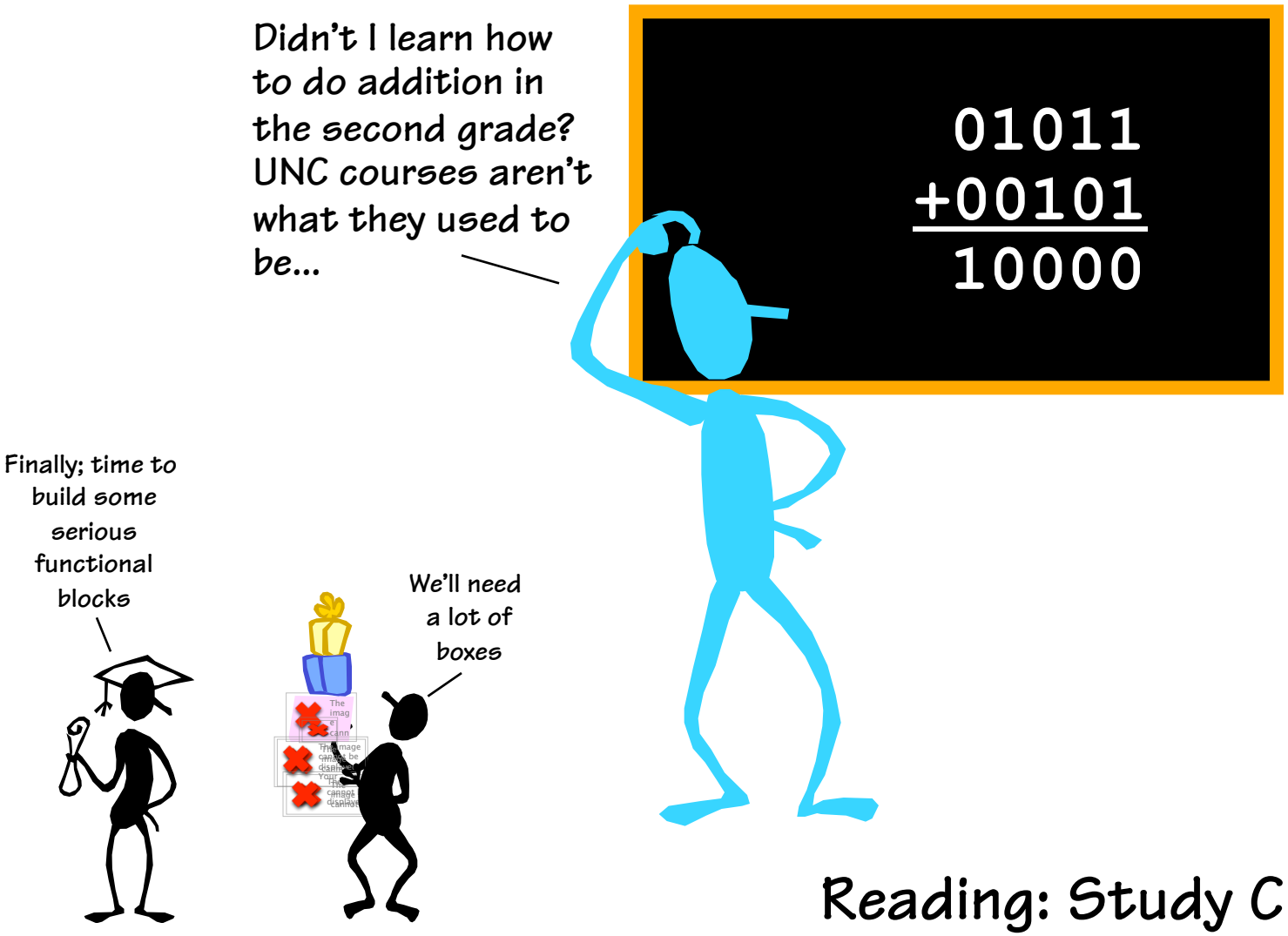
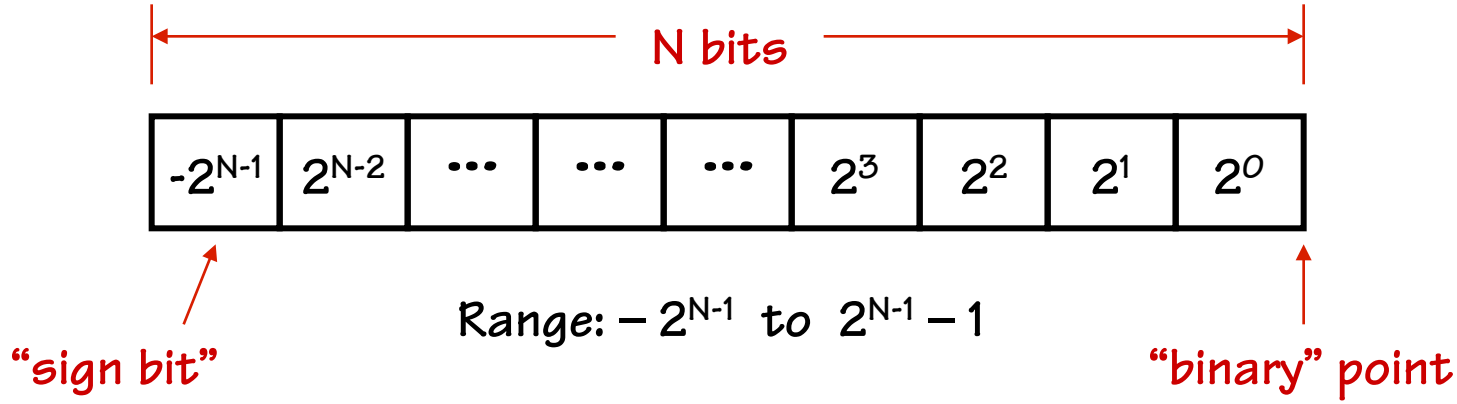
Arithmetic Circuits



Reading: Study Chapter 3.

Review: 2's Complement



8-bit 2's complement example:

$11010110 = -2^7 + 2^6 + 2^4 + 2^2 + 2^1 = -128 + 64 + 16 + 4 + 2 = -42$

If we use a two's-complement representation for signed integers, the same binary addition procedure will work for adding both signed and unsigned numbers.

By moving the implicit "binary" point, we can represent fractions too:

$1101.0110 = -2^3 + 2^2 + 2^0 + 2^{-2} + 2^{-3} = -8 + 4 + 1 + 0.25 + 0.125 = -2.625$

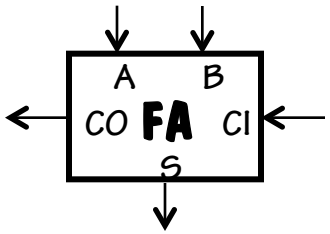
Binary Addition

Here's an example of binary addition as one might do it by "hand":

```
          1 1 0 1
A:      1101
B:    + 0101
      -------
     10010
```

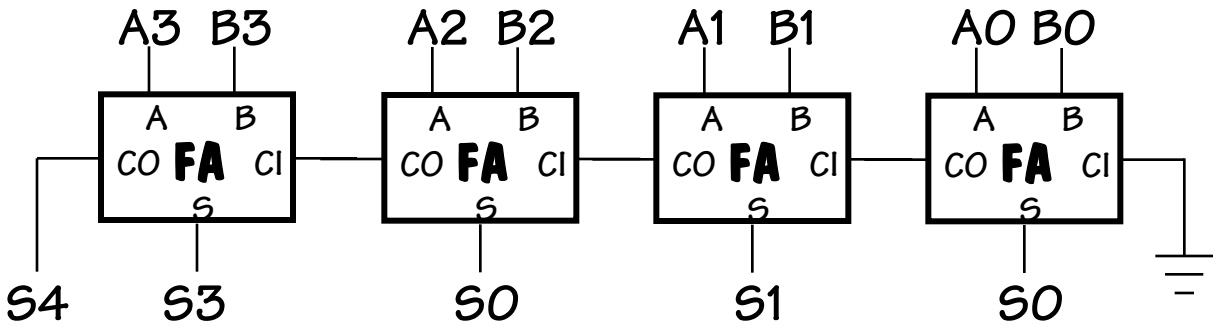
Carries from previous column

Adding two N-bit numbers produces an (N+1)-bit result



Let's start by building a block that adds one column:

Then we can cascade them to add two numbers of any size…



Designing a Full Adder: From Last Time

1) Start with a truth table:

C_i	A	B	C_o	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

2)Write down eqns for the "1" outputs

$$C_o = \overline{C}_i AB + C_i \overline{A} B + C_i A \overline{B} + C_i AB$$
$$S = \overline{C}_i \overline{A} B + \overline{C}_i A \overline{B} + C_i \overline{A} \overline{B} + C_i AB$$

3)Simplifing a bit

$$C_o = C_i(A + B) + AB$$
$$S = C_i \oplus A \oplus B$$

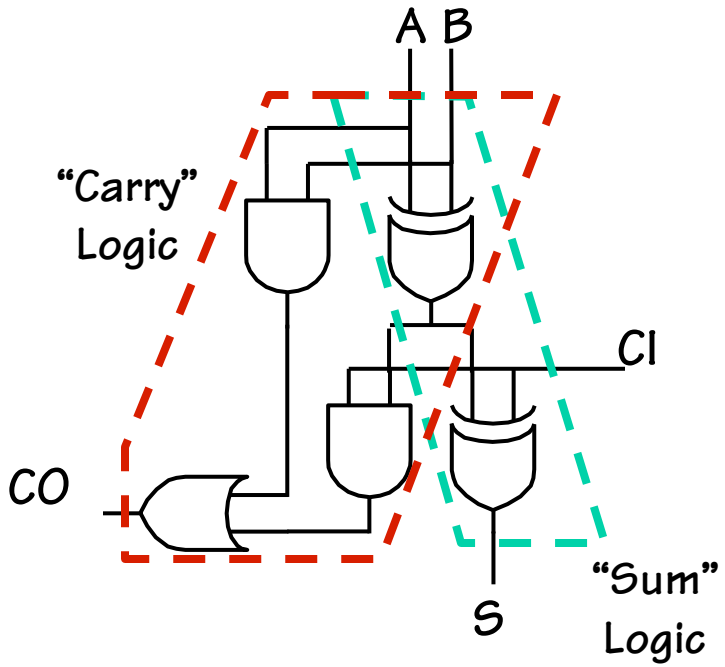
$$C_o = C_i(A \oplus B) + AB$$
$$S = C_i \oplus (A \oplus B)$$

For Those Who Prefer Logic Diagrams …

$C_o = C_i(A \oplus B) + AB$

$S = C_i \oplus (A \oplus B)$

- A little tricky, but only 5 gates/bit

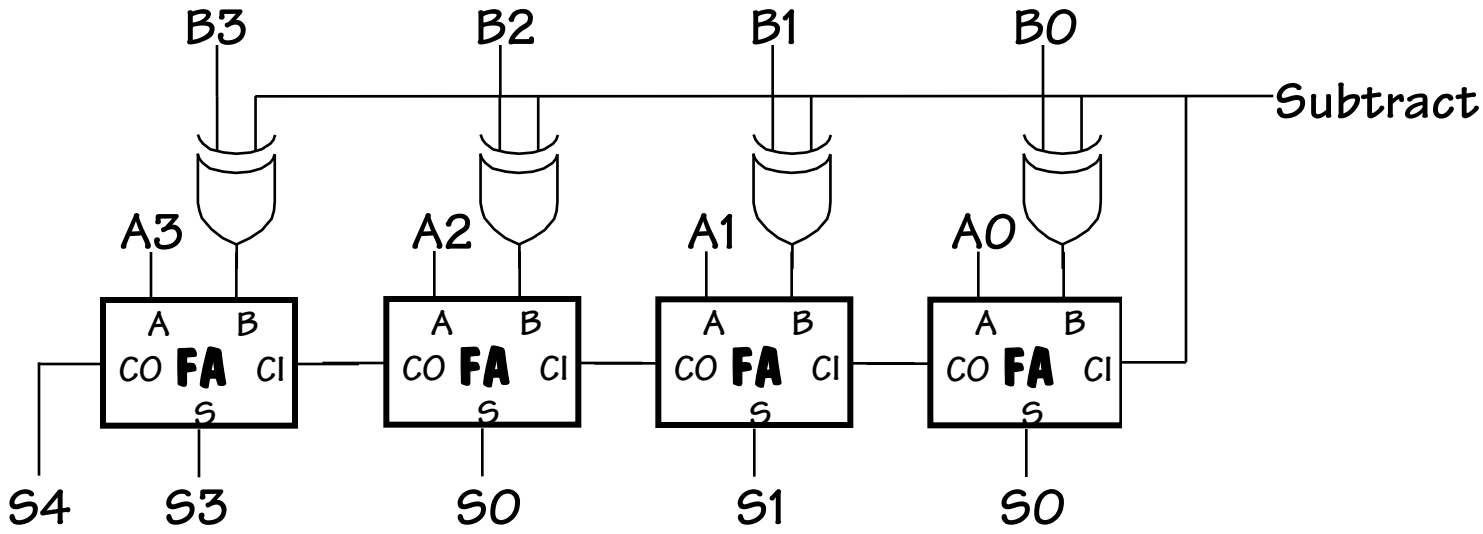


Subtraction: A-B = A + (-B)

Using 2's complement representation: –B = ~B + 1

~ = bit-wise complement

So let's build an arithmetic unit that does both addition and subtraction.
Operation selected by *control input*:



Condition Codes

Besides the sum, one often wants four other bits of information from an arithmetic unit:

Z (zero): result is = 0 *big NOR gate*

N (negative): result is < 0 S_{N-1}

C (carry): indicates that add in the most significant position produced a carry, e.g., "1 + (-1)" *from last FA*

V (overflow): indicates that the answer has too many bits to be represented correctly by the result width, e.g., "$(2^{i-1} - 1) + (2^{i-1} - 1)$"

$$V = A_{i-1} B_{i-1} \overline{N} + \overline{A}_{i-1} \overline{B}_{i-1} N$$

-or-

$$V = CO_{i-1} \oplus CI_{i-1}$$

To compare A and B, perform A–B and use condition codes:

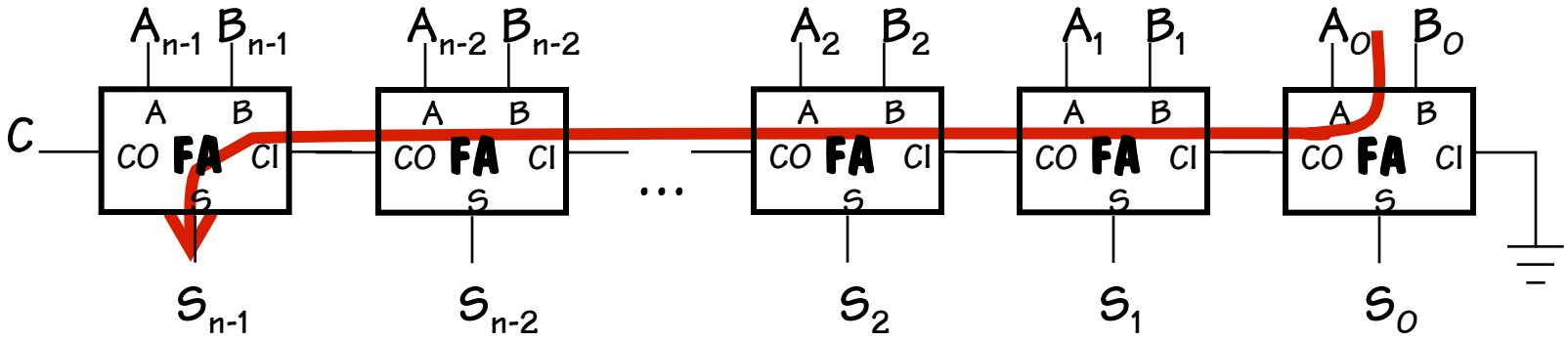
Signed comparison:

LT	N⊕V
LE	Z+ (N⊕V)
EQ	Z
NE	~Z
GE	~ (N⊕V)
GT	~ (Z+ (N⊕V))

Unsigned comparison:

LTU	C
LEU	C+Z
GEU	~C
GTU	~ (C+Z)

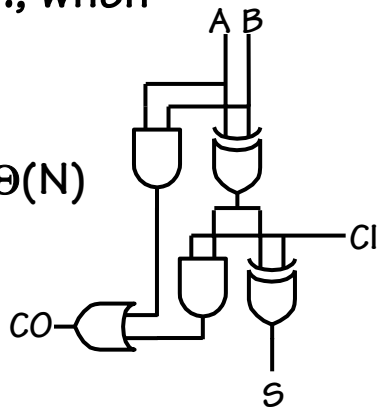
T_{PD} of Ripple-Carry Adder



Worse-case path: carry propagation from LSB to MSB, e.g., when adding 11…111 to 00…001.

$$t_{PD} = (t_{PD,XOR} + t_{PD,AND} + t_{PD,OR}) + (N-2)*(t_{PD,OR} + t_{PD,AND}) + t_{PD,XOR} \approx \Theta(N)$$

A_0,B_0 to CO_0 | CI to CO | CI_{N-1} to S_{N-1}



$\Theta(N)$ is read "order N" and tells us that the latency of our adder grows in proportion to the number of bits in the operands.

Faster Carry Logic

Let's see if we can improve the speed by first "rewriting" and then "reinterpreting" the equations for C_{OUT}:

$$C_{OUT} = AB + AC_{IN} + BC_{IN}$$

$$= AB + (A + B)C_{IN}$$

$$= G + P\,C_{IN} \qquad \text{where } G = AB \text{ and } P = A + B$$

generate — propagate

To generate the Carry of the N^{th} bit:

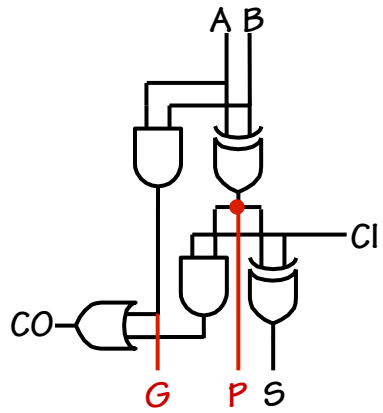
$$C_N = G_{N-1} + P_{N-1}C_{N-1}$$

$$= G_{N-1} + P_{N-1}G_{N-2} + P_{N-1}P_{N-2}C_{N-2}$$

$$= G_{N-1} + P_{N-1}G_{N-2} + P_{N-1}P_{N-2}G_{N-3} + \ldots + P_{N-1}\ldots P_0 C_{IN}$$

C_N in only 3 levels of logic!
1 for P/G generation, 1 for ANDs, 1 for final OR

Actually, P was can be either A + B or A ⊕ B, because the G = AB term of C_{OUT} handles the only case where they differ.



N-Bit Addition in Constant Time?

So if we had (N+1)-input gates and didn't mind a lot of loading on the P signals, the propagation delay of adder built using P/G equation to compute C_{IN} of each bit would be:

4 gate delays ≈ O(1) (independent of N)

Recall large *fan-in* gates (many inputs) are implemented using trees (see last lecture). So for large N we expect more like $O(\log_2 N)$ gate delays. This concept does lead to some interesting adder designs:

- faster ripple-carry implementations
- hierarchical carry-lookahead adders

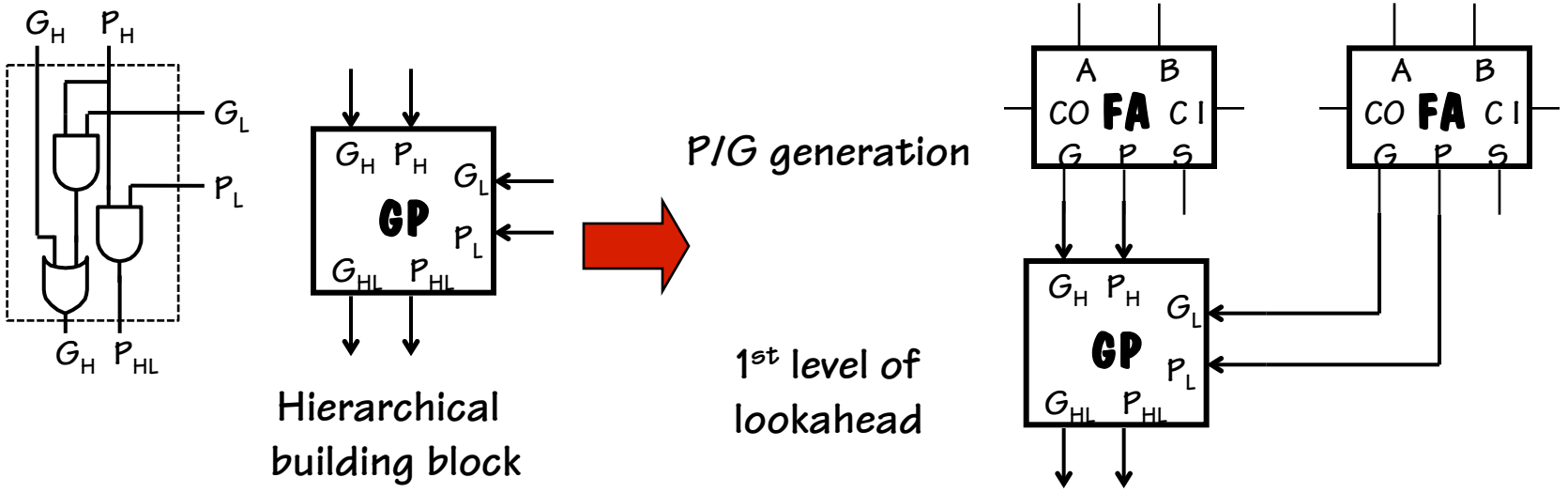
Carry-Lookahead Adders (CLA)

We can build a hierarchical carry-lookahead chain by generalizing our definition of the Carry Generate/Propagate (GP) Logic. We start by dividing our addend into two parts, a higher part, H, and a lower part, L. The GP function can be expressed as follows:

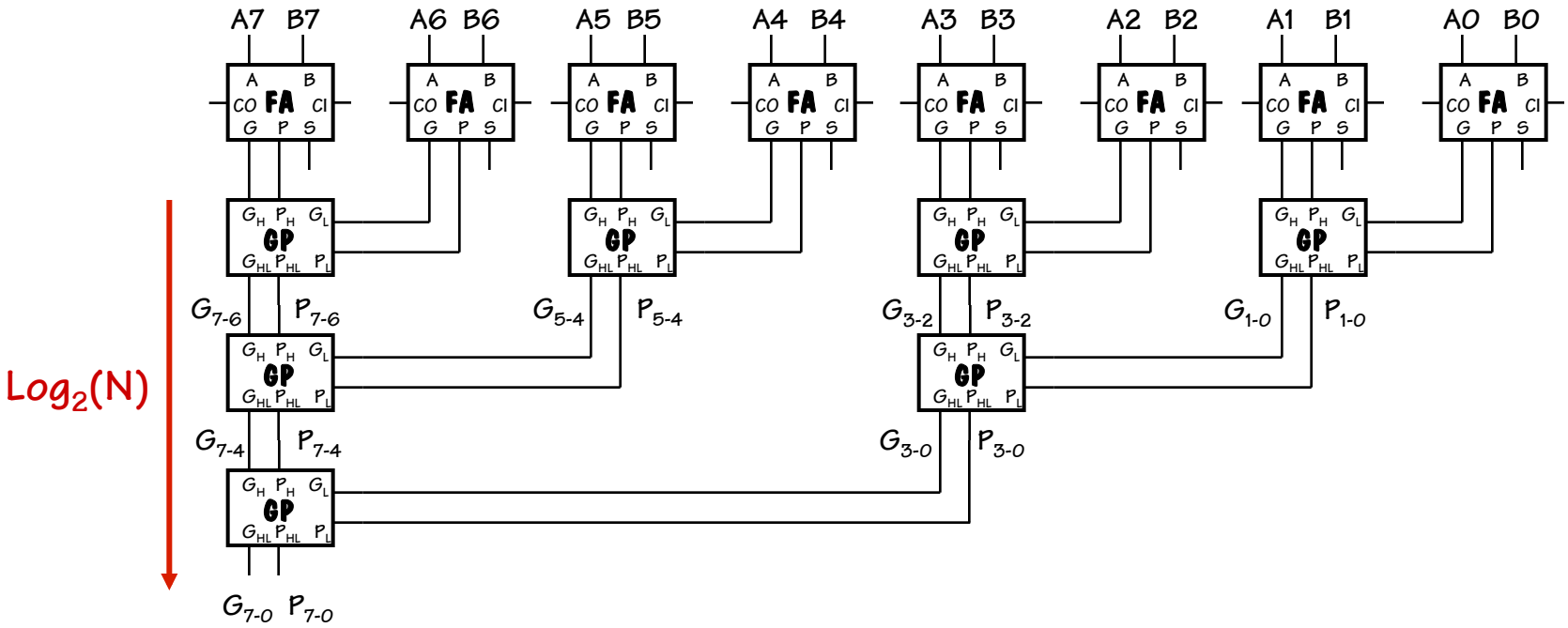
$$G_{HL} = G_H + P_H G_L$$

$$P_{HL} = P_H P_L$$

Generate a carry out if the high part generates one, or if the low part generates one and the high part propagates it. Propagate a carry if both the high and low parts propagate theirs.



8-bit CLA (GP Generation)

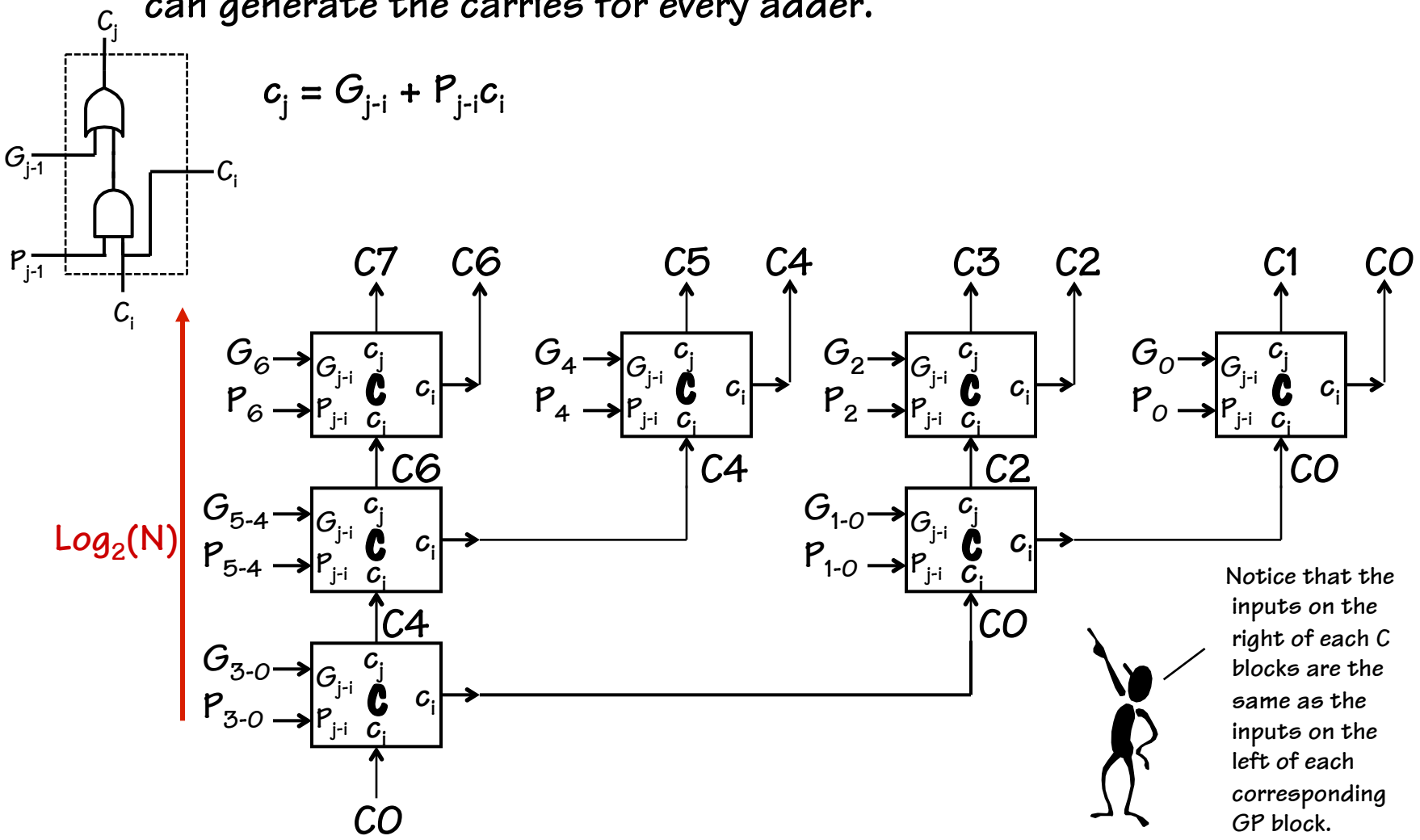


We can build a tree of GP units to compute the generate and propagate logic for any sized adder. For a 2^N-bit adder, we need 2^N-1 GP units.

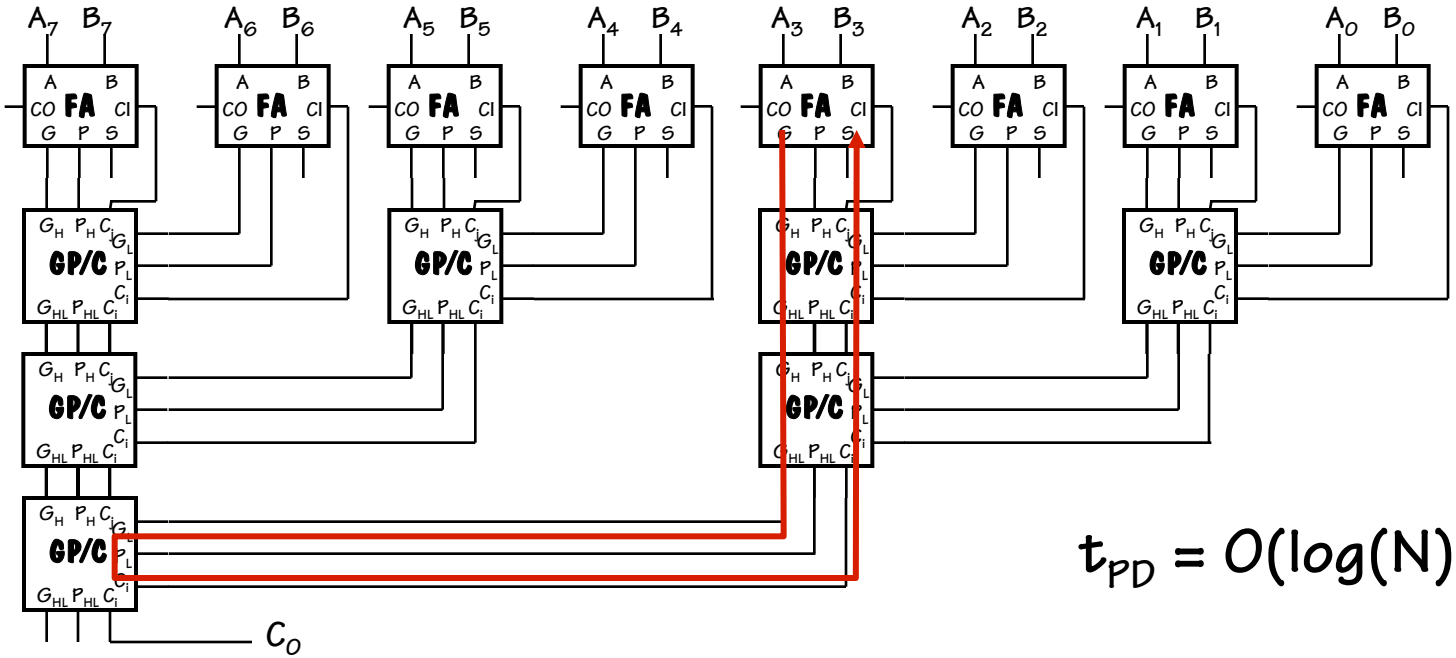
$$C = \underbrace{G_7 + P_7 G_6 + P_7 P_6 G_5 + P_7 P_6 P_5 G_4 + \ldots}_{G_{7-0}} + \underbrace{P_7 \ldots P_0}_{P_{7-0}} C_{IN}$$

8-bit CLA (Carry Generation)

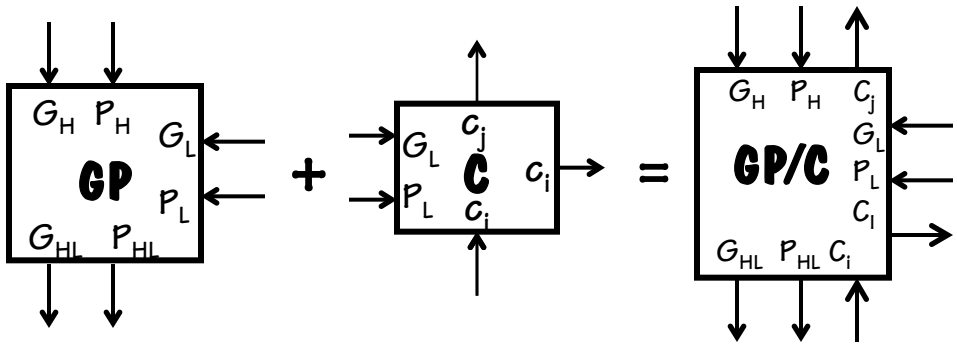
Now, given a the value of the carry-in of the least-significant bit, we can generate the carries for every adder.



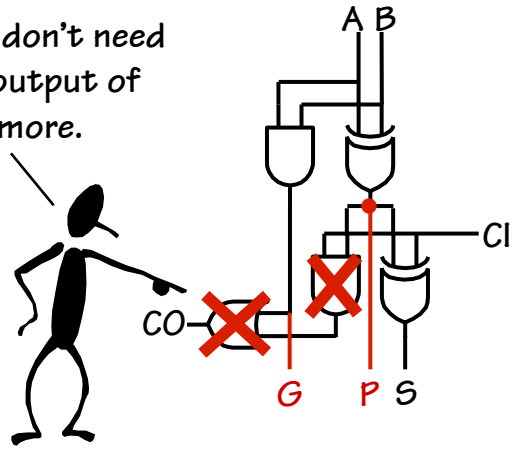
8-Bit CLA (Complete)



$t_{PD} = O(\log(N))$

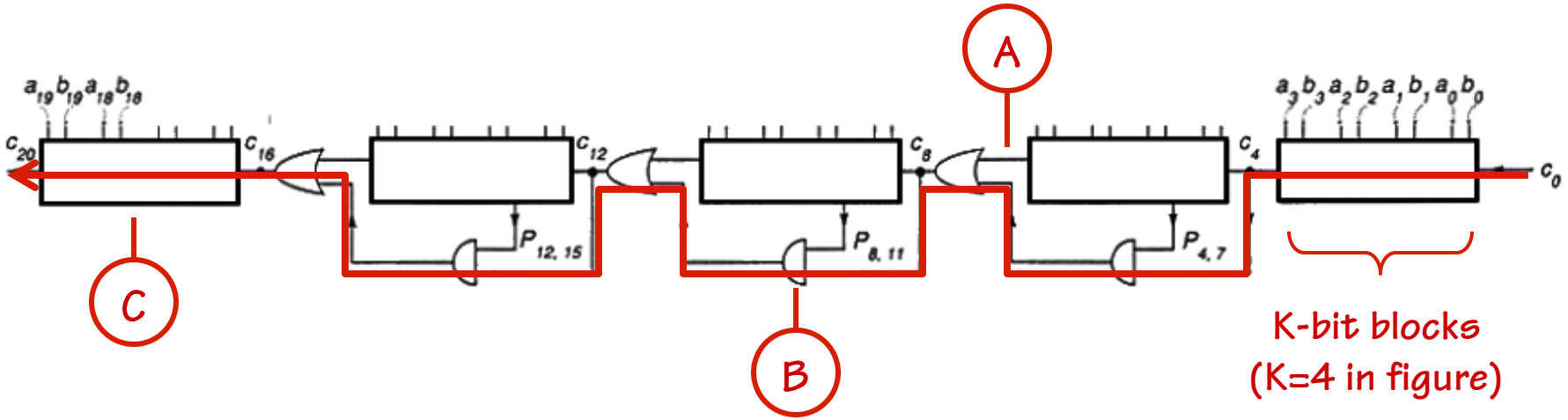


Notice that we don't need the carry-out output of the adder any more.



Carry-Skip Adders

Idea: full P/G equations are complicated, but P by itself is simple. So just use P to "skip" carry across a block of ripple-carry adders:



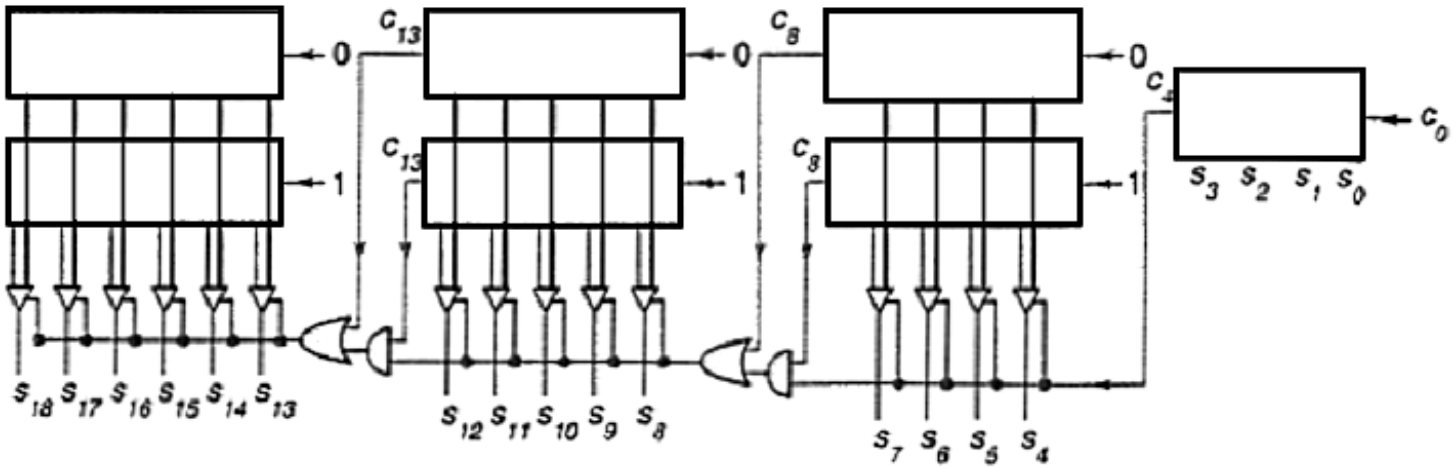
(A) Carries ripple *simultaneously* through each block; if block generates a carry, it appears on carry-out of block (similar to G). If carry-in is 0 at start of operation, no spurious carry-outs will be generated.

(B) If carry-in and P_{BLOCK} are both true, carry *skips* to next block

(C) Carry ripples though final block. $t_{PD} = 2*[K+ (N/K - 2) + K]$
With variable size blocks $t_{PD} \rightarrow O(\text{sqrt}(N))$

Carry-Select Adders

Idea: do two additions, one assuming carry-in is 0, the other assuming carry-in is 1. Use MUX to select correct answer when correct carry-in is known.



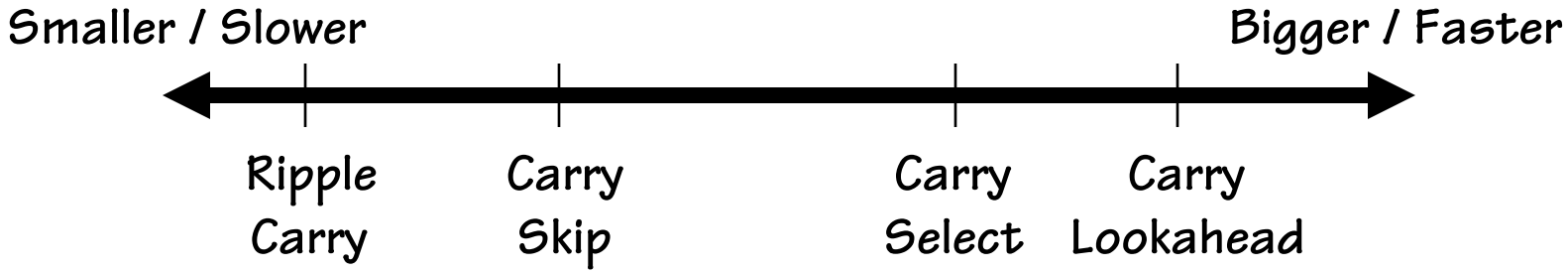
Blocks on the left can be bigger (more bits) –
allowing more ripple time time while waiting for select

With one stage: 50% more gates, but twice as fast as ripple-carry

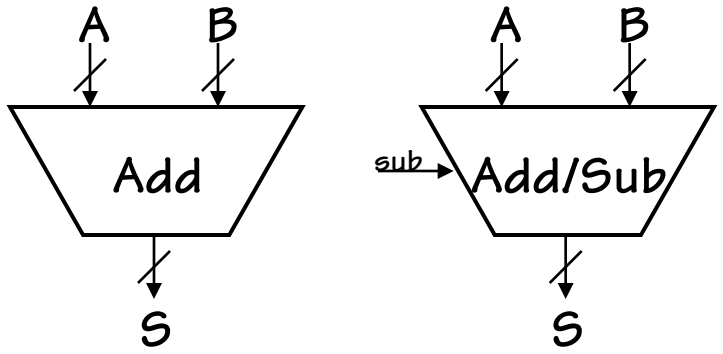
With multiple (variable-size) blocks: $t_{PD} \rightarrow O(\sqrt{N})$

Adder Summary

Adding is not only a common, but it is also tends to be one of the most time-critical of operations. As a result, a wide range of adder architectures have been developed that allow a designer to tradeoff complexity (in terms of the number of gates) for performance.



A this point we'll define a high-level functional unit for an adder, and specify the details of the implementation as necessary.



Shifting Logic

Shifting is a common operation that is applied to groups of bits. Shifting can be used for alignment, as well as for arithmetic operations.

X << 1 is approx the same as 2*X

X >> 1 can be the same as X/2

For example:

$$X = 20_{10} = 00010100_2$$

Left Shift:

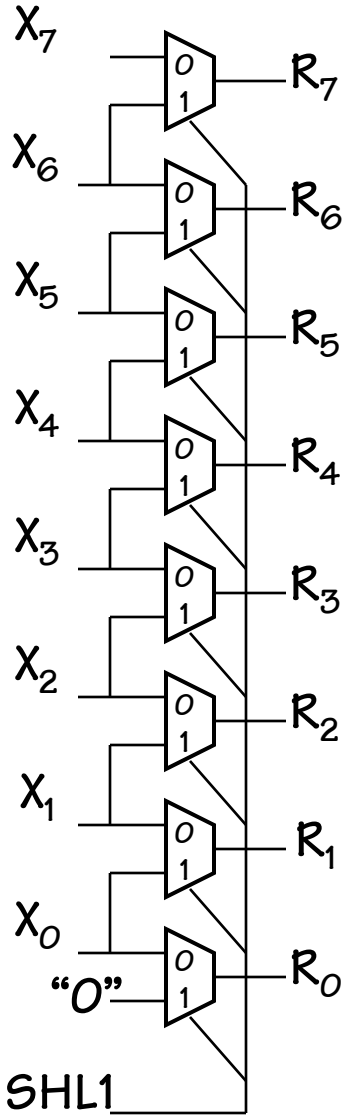
$(X << 1) = 00101000_2 = 40_{10}$

Right Shift:

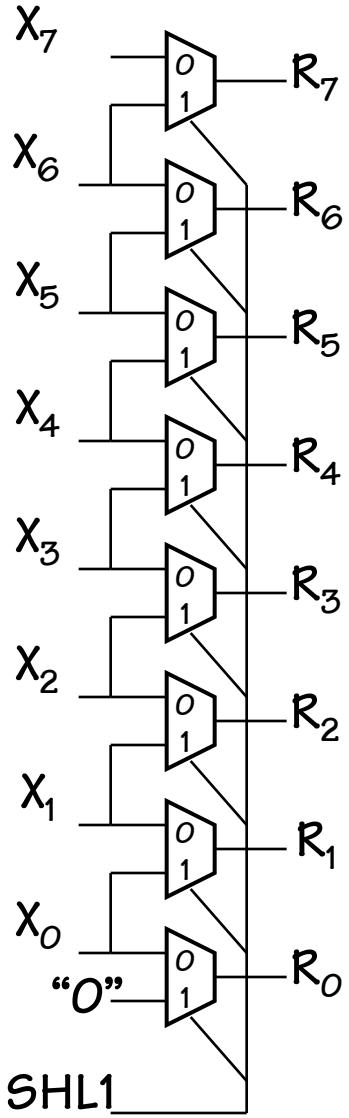
$(X >> 1) = 00001010_2 = 10_{10}$

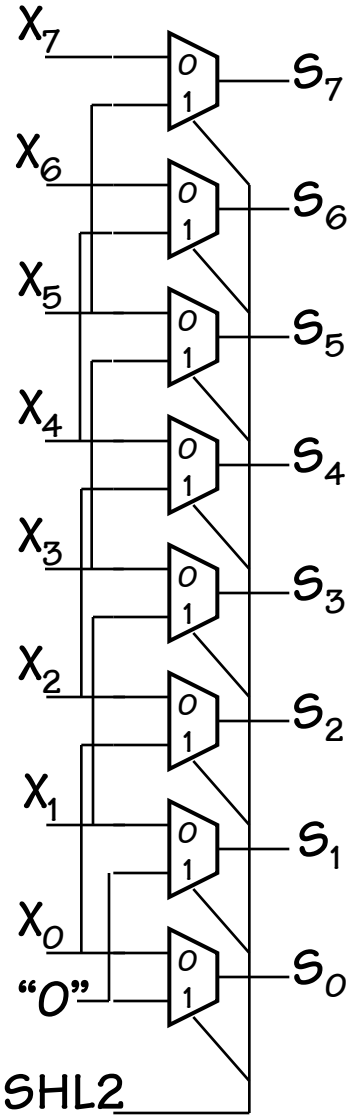
Signed or "Arithmetic" Right Shift:

$(-X >> 1) = (11101100_2 >> 1) = 11110110_2 = -10_{10}$



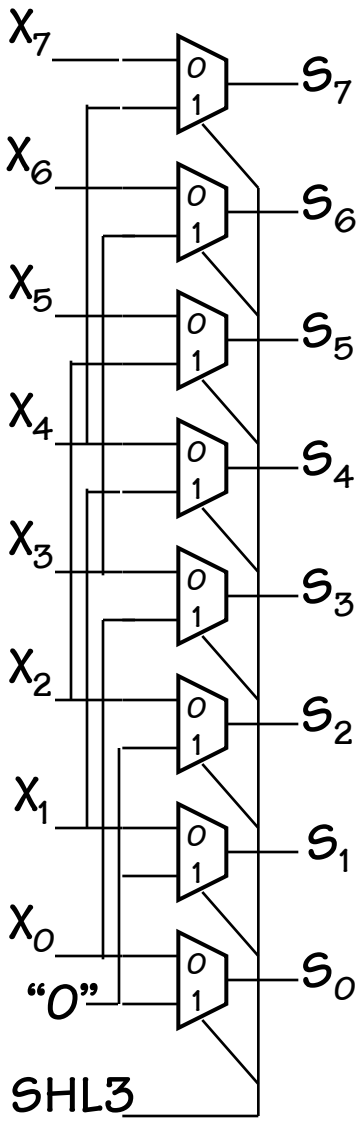
More Shifting



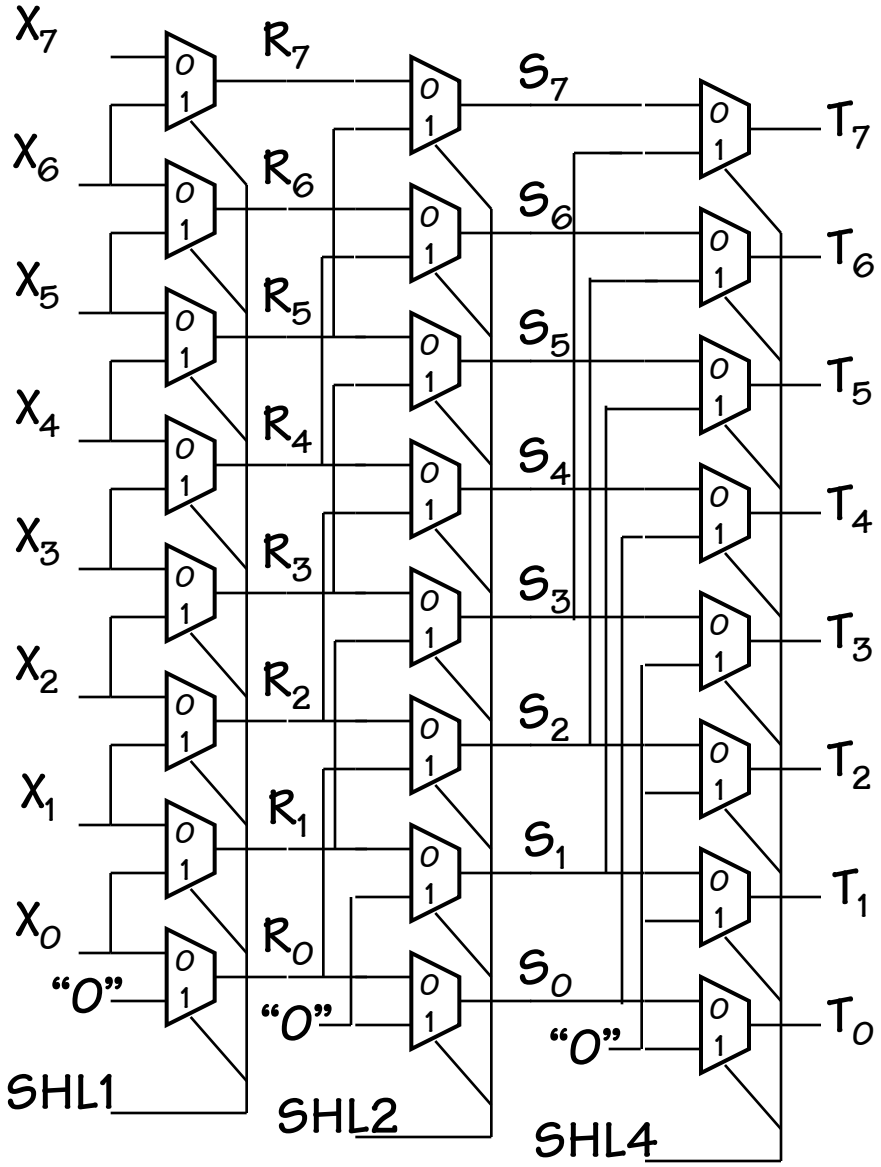


Using the same basic idea we can build left shifters of arbitrary sizes using muxes.

Each shift amount requires its own set of muxes.



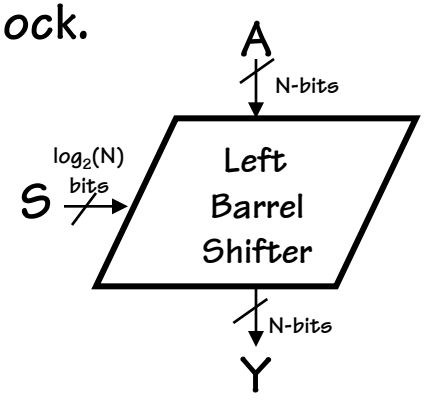
Barrel Shifting



If we connect our "shift-left-two" shifter to the output of our "shift-left-one" we can shift by 0, 1, 2, or 3 bits.

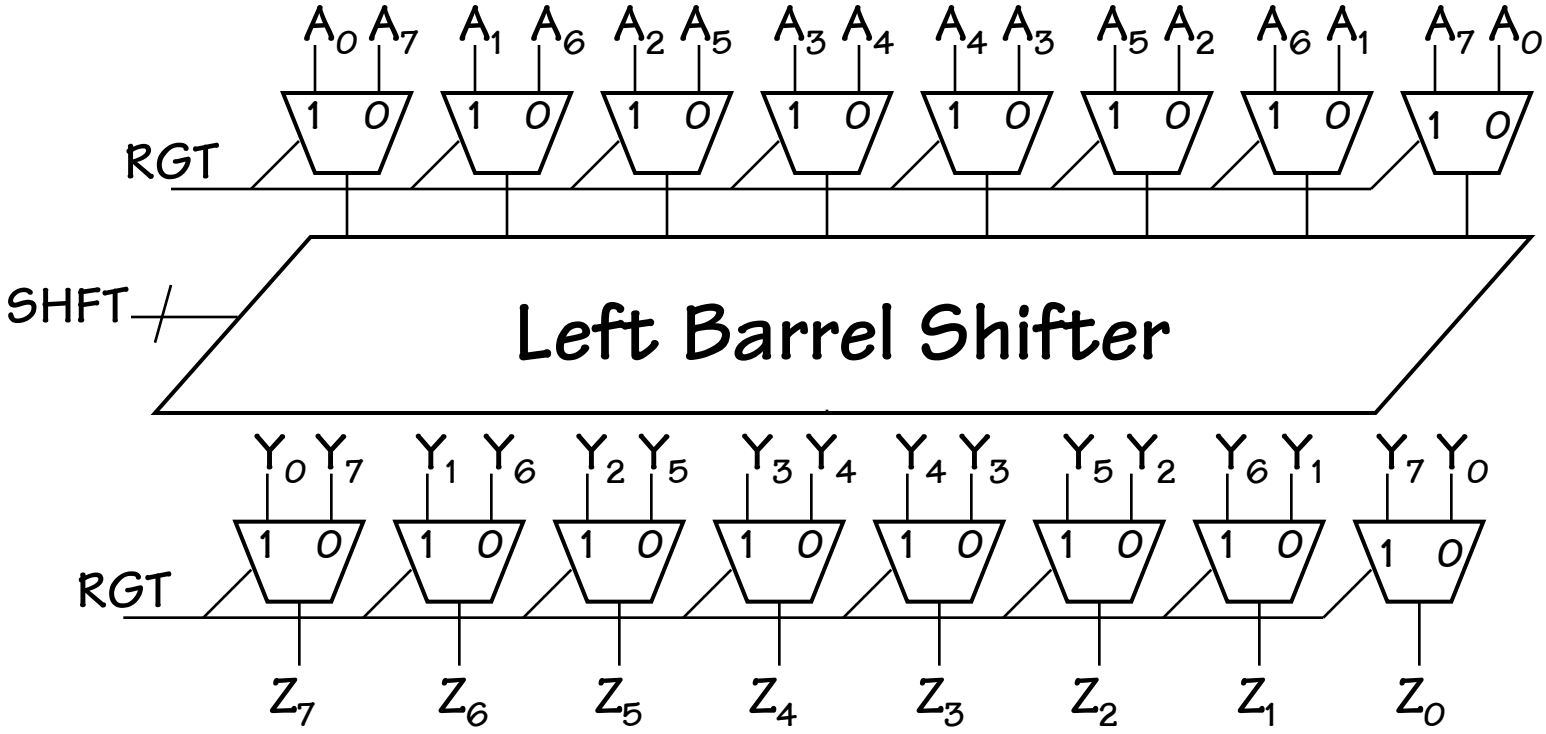
And, if we add one more "shift-left-4" shifter we can do any shift up to 7 bits!

So, let's put a box around it and call it a new functional block.



Barrel Shifting with a Twist

At this point it would be straightforward to construct a "Right barrel shifter" unit. However, a simple trick that enables a left shifter to do both.



Boolean Operations

We also need to perform logical operations on groups of bits.
Which ones?

ANDing is useful for "masking" off groups of bits.
ex. 10101110 & 00001111 = 00001110 (mask selects last 4 bits)

ANDing is also useful for "clearing" groups of bits.
ex. 10101110 & 00001111 = 00001110 (0's clear first 4 bits)

ORing is useful for "setting" groups of bits.
ex. 10101110 | 00001111 = 10101111 (1's set last 4 bits)

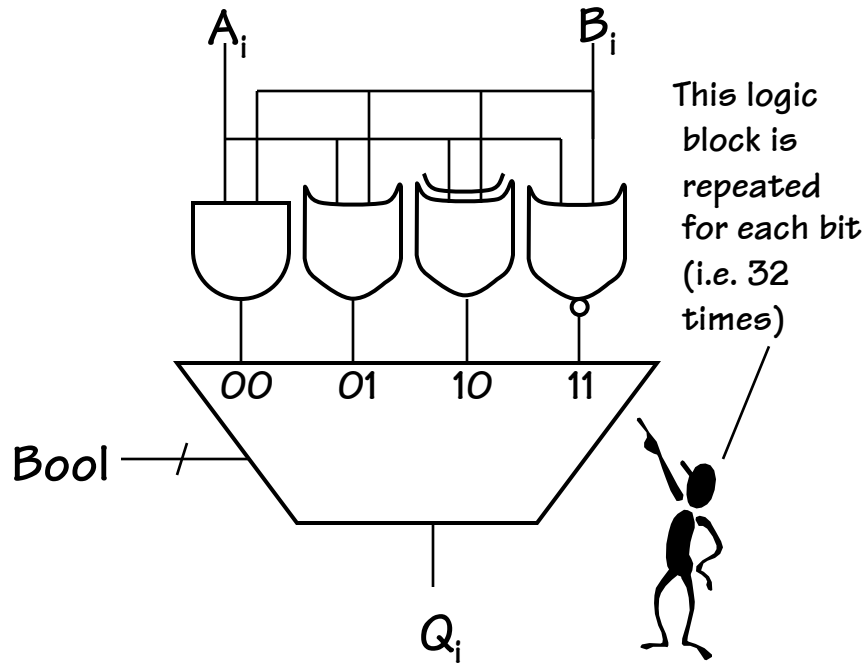
XORing is useful for "complementing" groups of bits.
ex. 10101110 ^ 00001111 = 10100001 (1's complement last 4 bits)

NORing is useful.. Uhm, because John Hennessy says it is!
ex. ~(10101110 | 00001111) = 01010000 (0's complement, 1's clear)

Boolean Unit (The book's way)

It is simple to build up a Boolean unit using primitive gates and a mux to select the function.

Since there is no interconnection between bits, this unit can be simply replicated at each position. The cost is about 7 gates per bit. One for each primitive function, and approx 3 for the 4-input mux.



This is a straightforward, but not too elegant of a design.

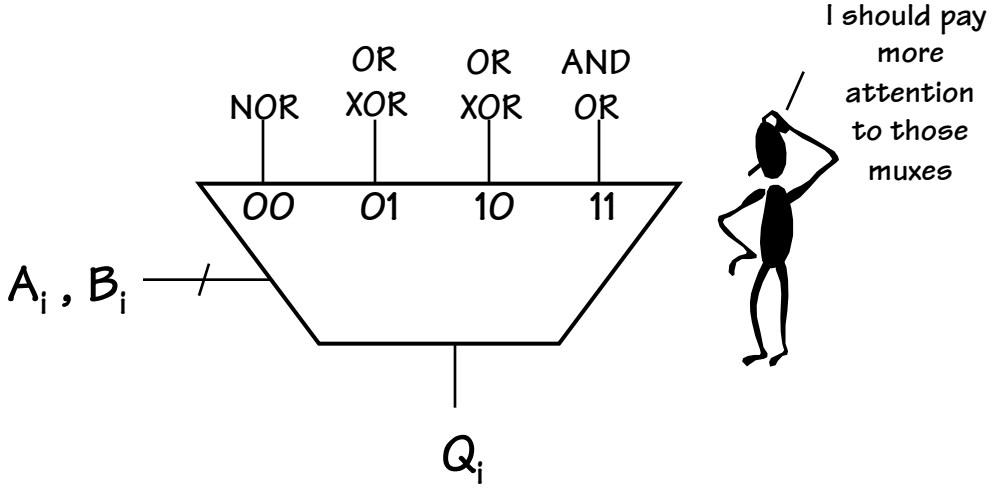
Cooler Bools

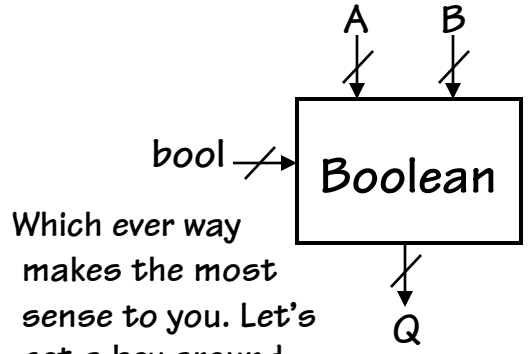
We can better leverage a mux's capabilities in our Boolean unit design, by connecting the bits to the select lines.

Why is this better?

1) While it might take a little logic to decode the truth table inputs, you only have to do it once, independent of the number of bits.

2) It is trivial to extend this module to support any 2-bit logical function.
(How about NAND, John?
Actually A & /B might be more useful)





An ALU, at Last

We give the "Math Center" of a computer a special name--the Arithmetic Logic Unit. For us, it just a big box!