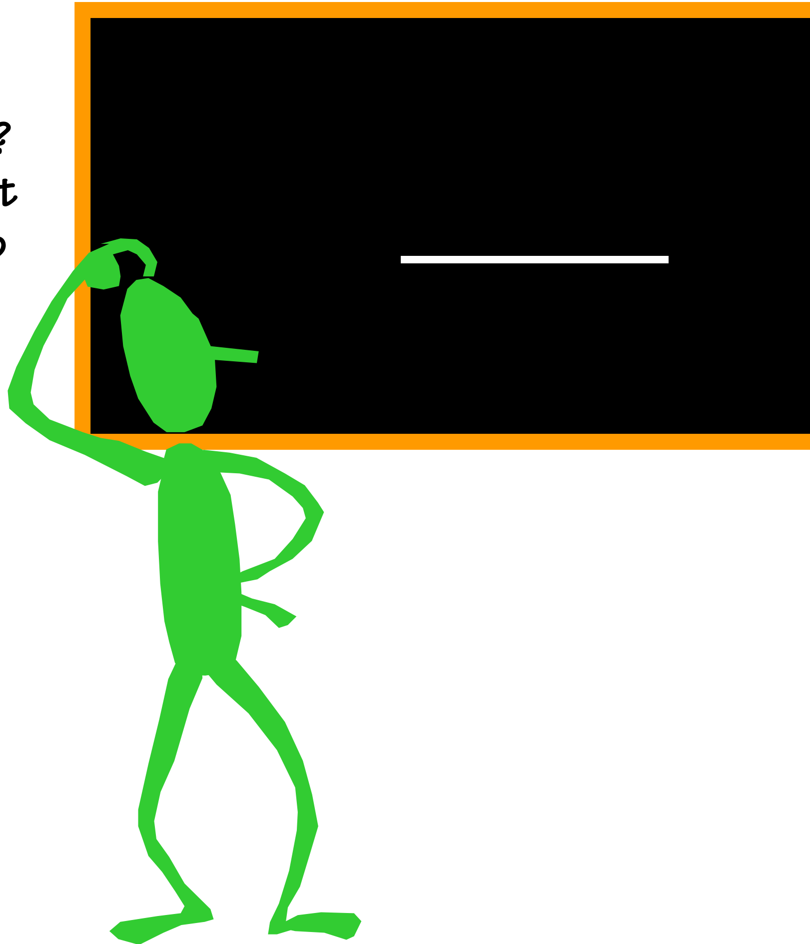
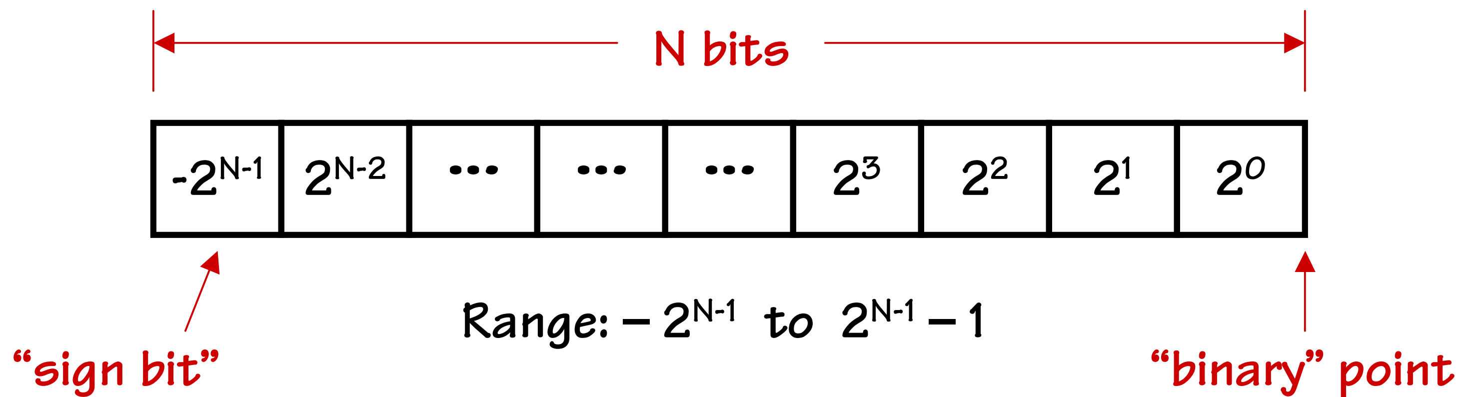
Case Study: Arithmetic Circuits

Didn't I learn how to do addition in the second grade? MIT courses aren't what they used to be...



Handouts: Lecture Slides

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:

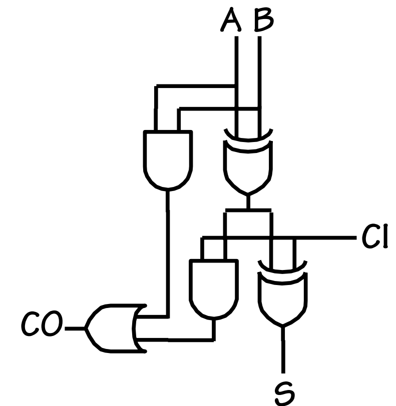
$1101.0110 = -2^3 + 2^2 + 2^0 + 2^{-2} + 2^{-3} = -8 + 4 + 1 + 0.25 + 0.125 = -2.625$

Binary Addition

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

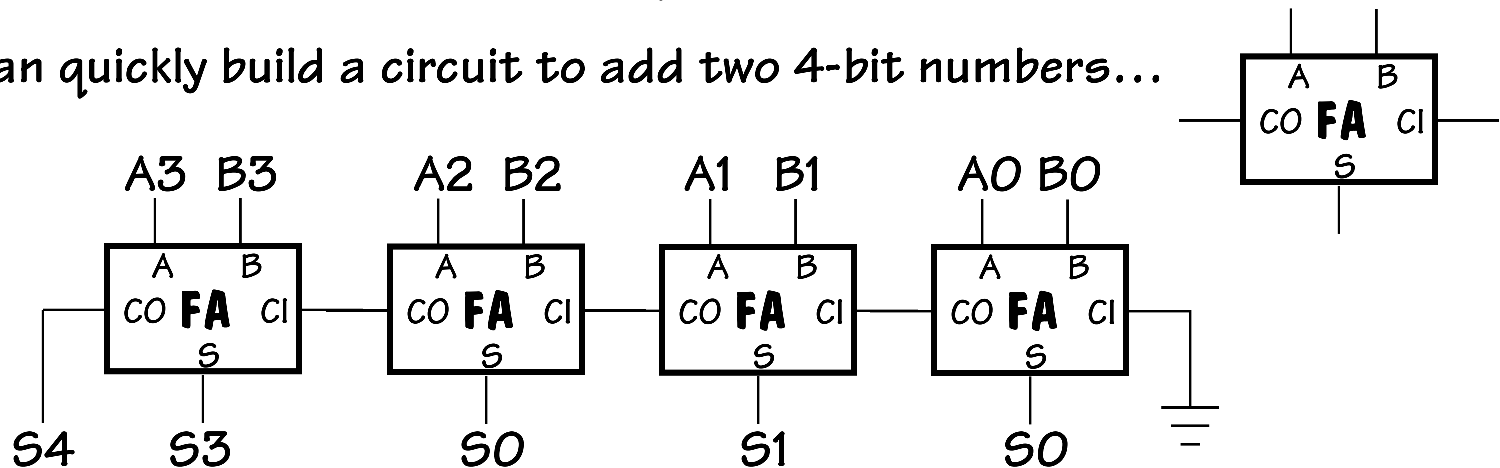
1 1 0 1 ← Carries from previous column

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



We've already built the circuit that implements one column:

So we can quickly build a circuit to add two 4-bit numbers…

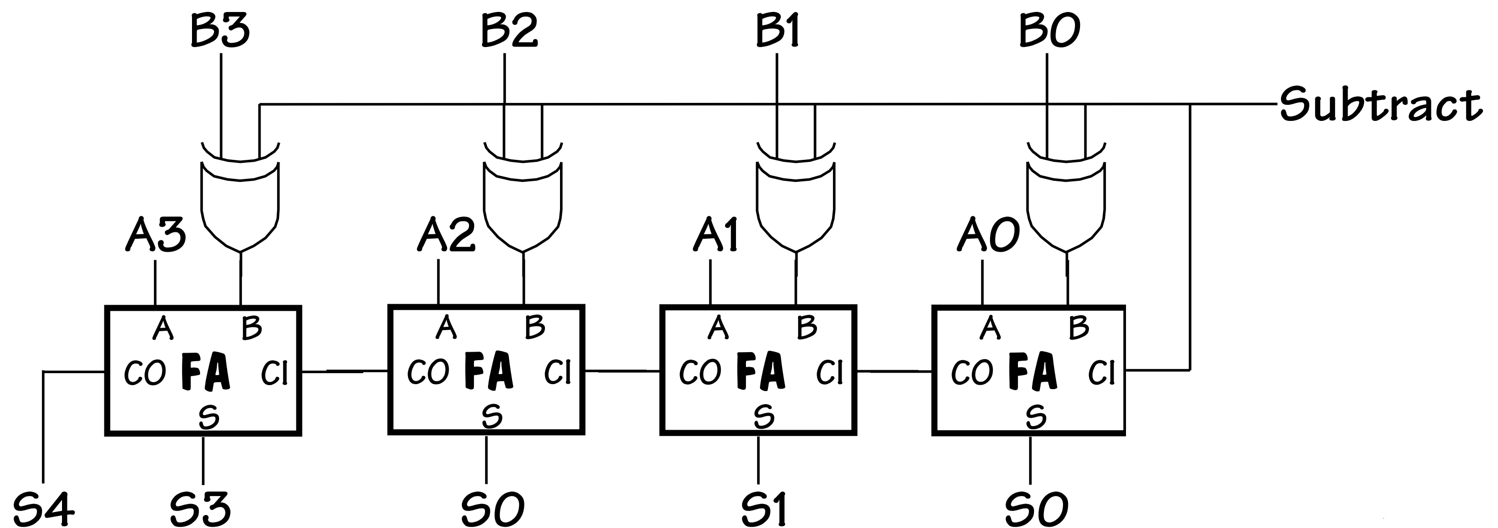


Subtraction: A-B = A + (-B)

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*

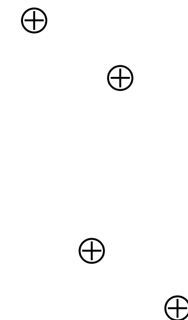
V (overflow): indicates that the answer has too many bits to be represented correctly by the result width, e.g., "$(2^{i-1} - 1) + (2^{i-1} - 1)$"

$$V = A_{i-1} B_{i-1} \overline{N} + \overline{A_{i-1}} \, \overline{B_{i-1}} N$$

$$V = COUT_{i-1} \oplus CIN_{i-1}$$

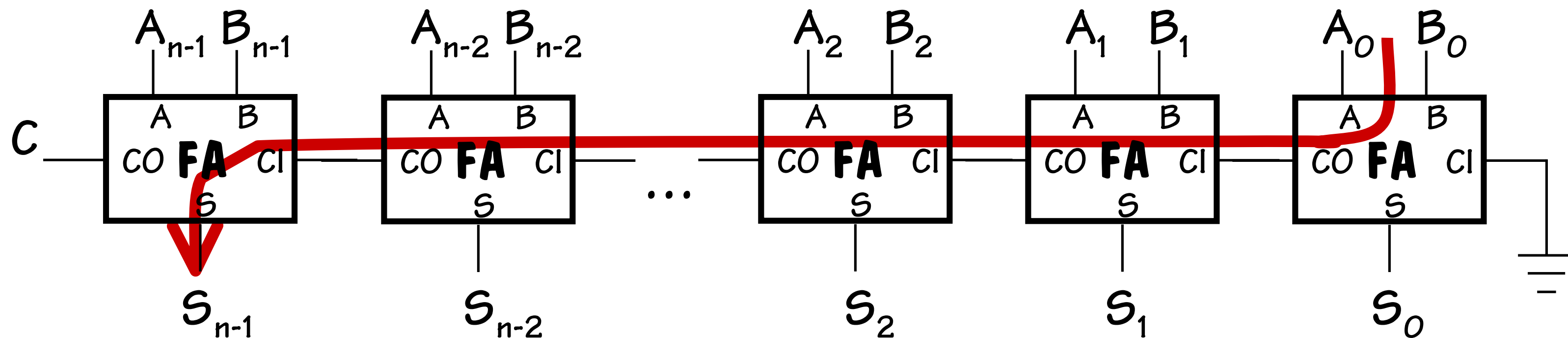
To compare A and B, perform A–B and use condition codes:

Signed comparison:



Unsigned comparison:

t_{PD} of Ripple-carry Adder

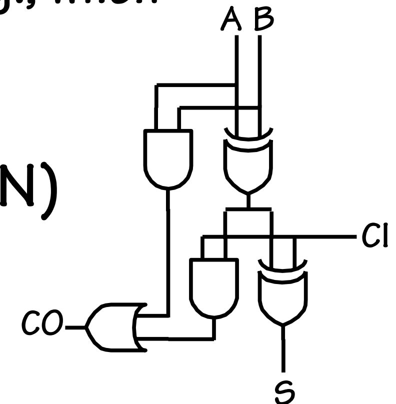


Worse-case path: carry propagation from LSB to MSB, e.g., when adding 11…111 to 00…001.

$$t_{PD} = (N-1)*(t_{PD,OR} + t_{PD,AND}) + t_{PD,XOR} \approx \Theta(N)$$

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 rewriting 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

Actually, P is usually defined as $P = A \oplus B$ which won't change C_{OUT} but will allow us to express S as a simple function of P and C_{IN}: $S = P \oplus C_{IN}$

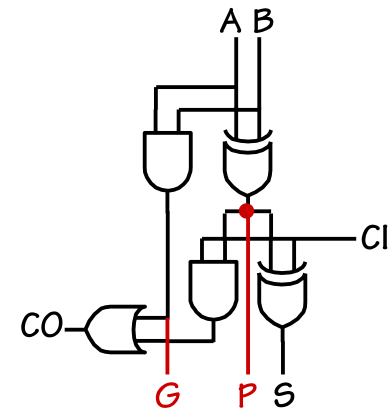
For adding two N-bit numbers:

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

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

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

C_N in only 3 (!) gate delays:
1 for P/G generation, 1 for ANDs, 1 for final OR



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:

$$4 \text{ gate delays} \approx \Theta(1)$$

Of course, this is impractical when N is "large" (i.e. > 4) but it does lead to some interesting ideas:

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

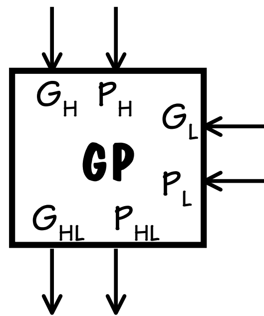
Carry-Lookahead Adders (CLA)

We can build a hierarchical carry 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:

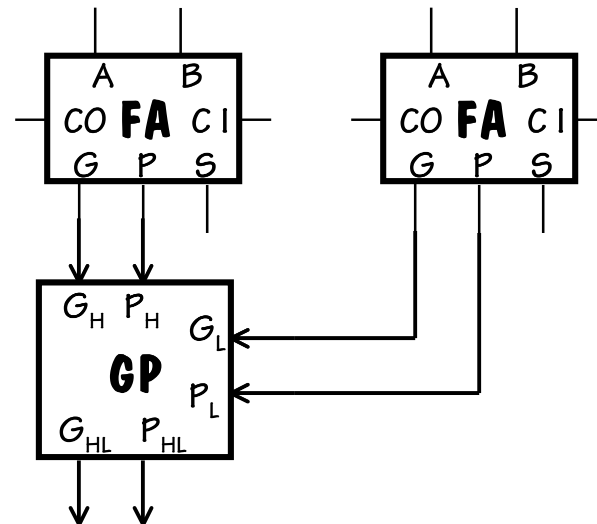
$$G_{HL} = G_H + P_H G_L$$

$$P_{HL} = P_H P_L$$

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.



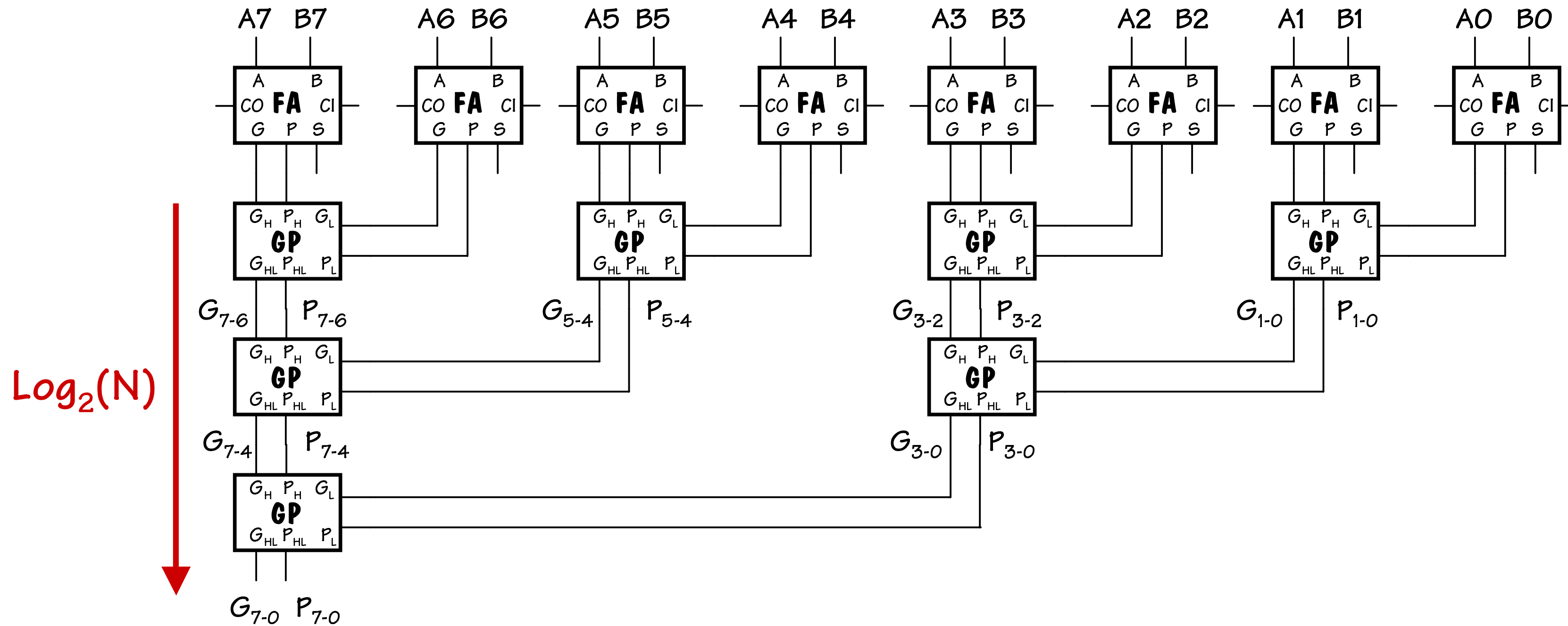
Hierarchical building block



P/G generation

1st level of lookahead

8-bit CLA (GP generation)



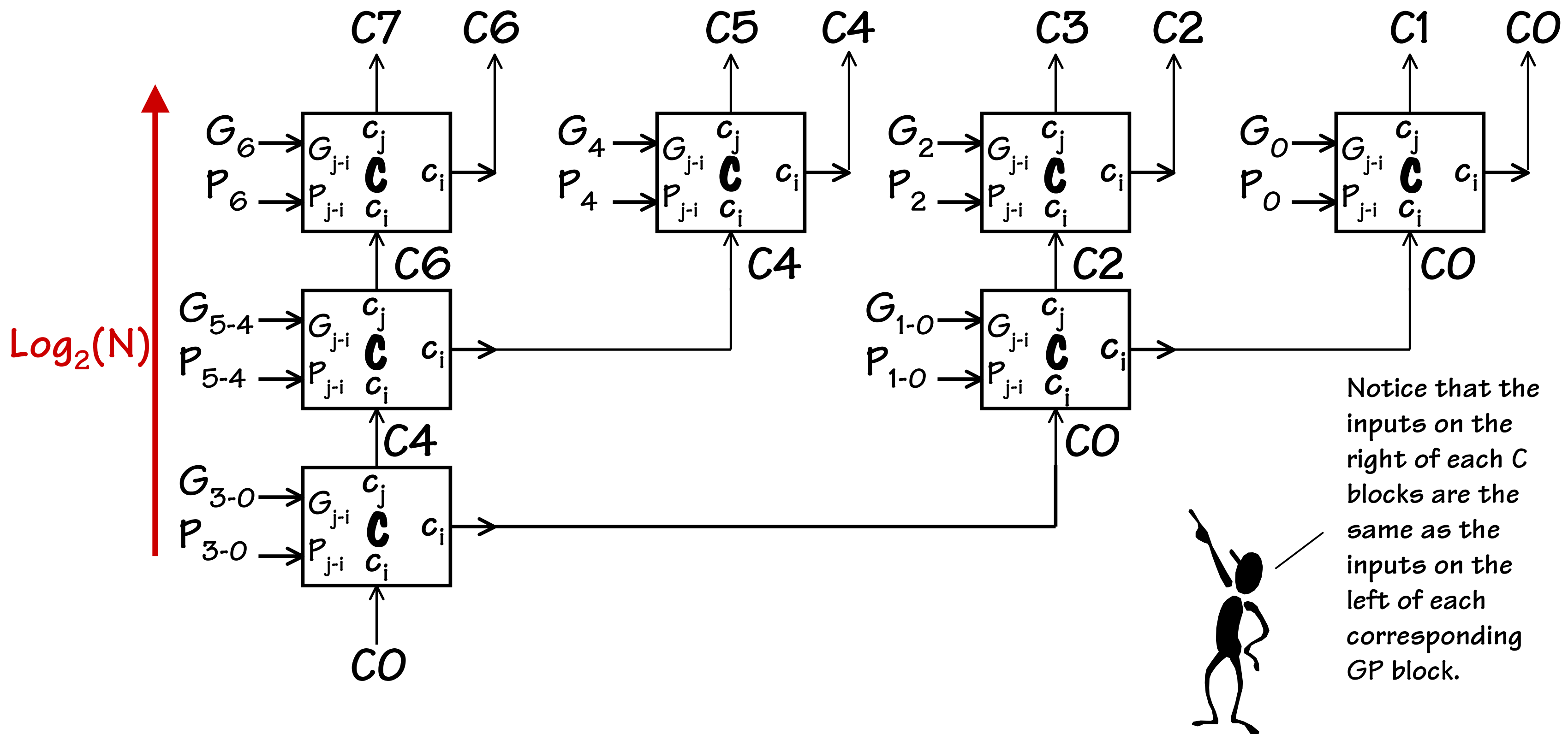
We can build a tree of GP units to compute the generate and propagate logic for any sized adder. For a 2^N-bit adder, we need 2^{N-1} GP units.

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

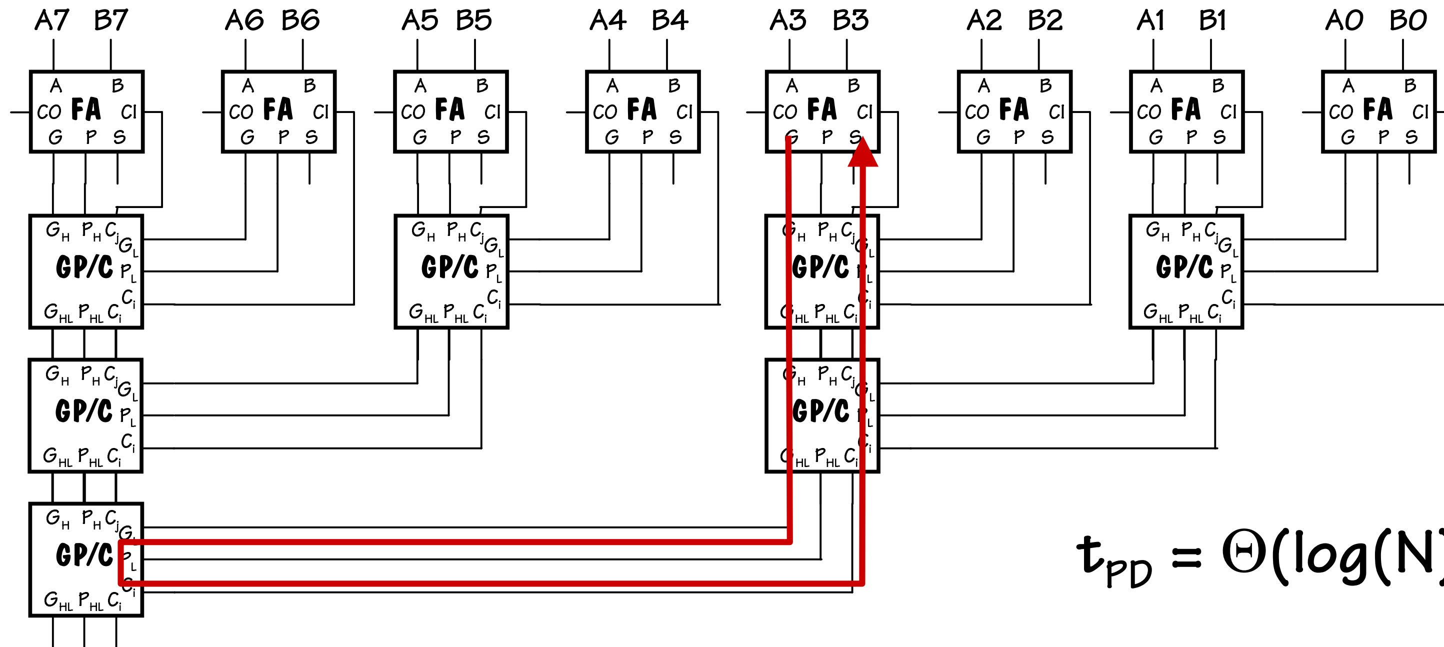
8-bit CLA (carry generation)

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

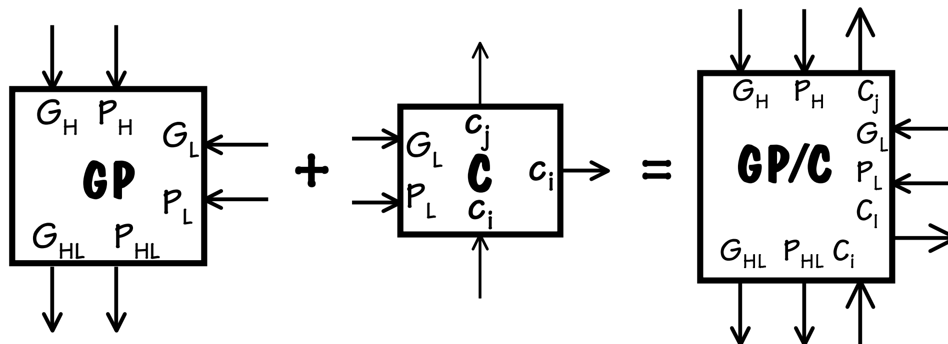
$$c_j = G_{j-i} + P_{j-i}c_i$$



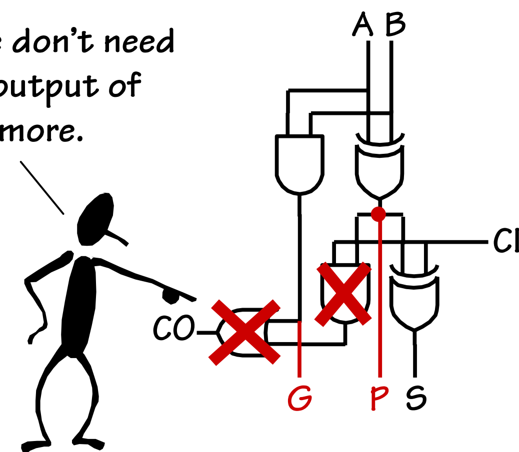
8-bit CLA (complete)



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

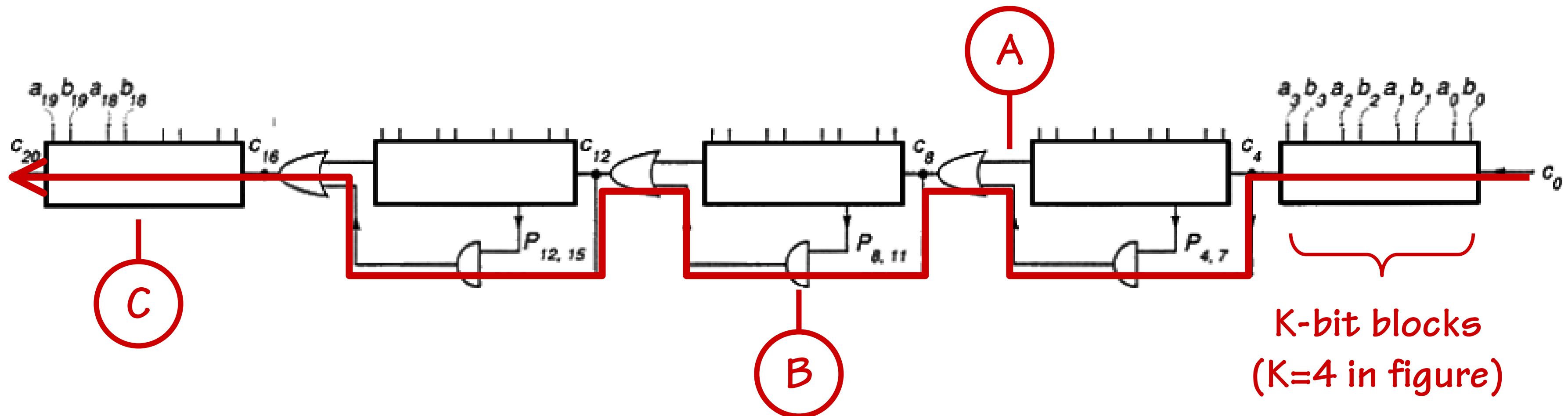


Notice that we don't need the carry-out output of the adder any more.



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.

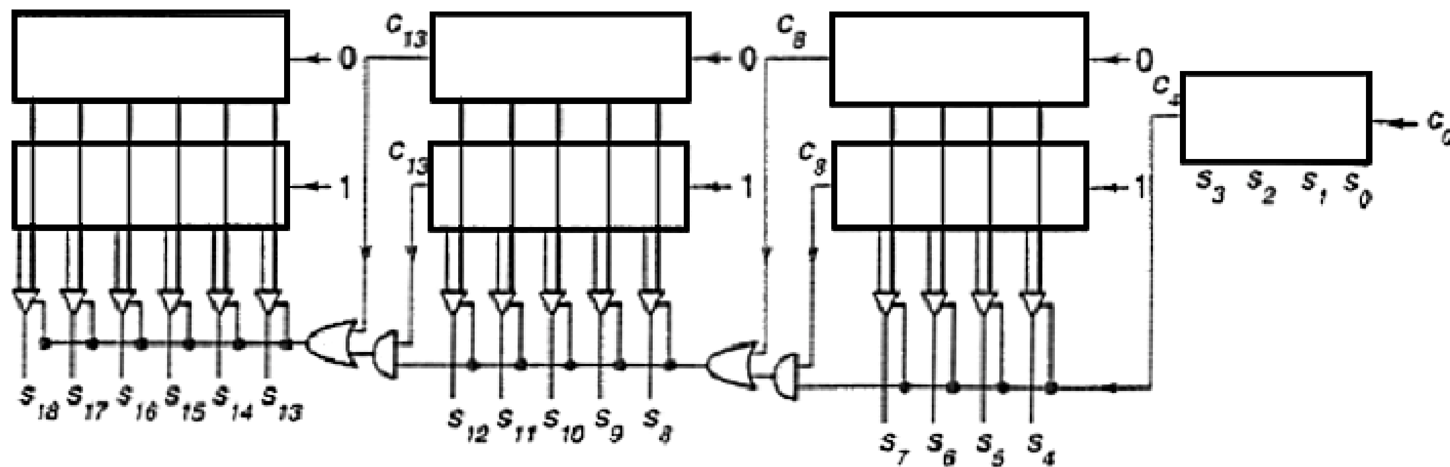
(B) If carry-in and P_{BLOCK} are both true, carry *skips* to next block

(C) Carry ripples though final block. $t_{PD} = 2*[K + (N/K - 2) + K]$

With variable size blocks $t_{PD} \rightarrow \Theta(\text{sqrt}(N))$

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.

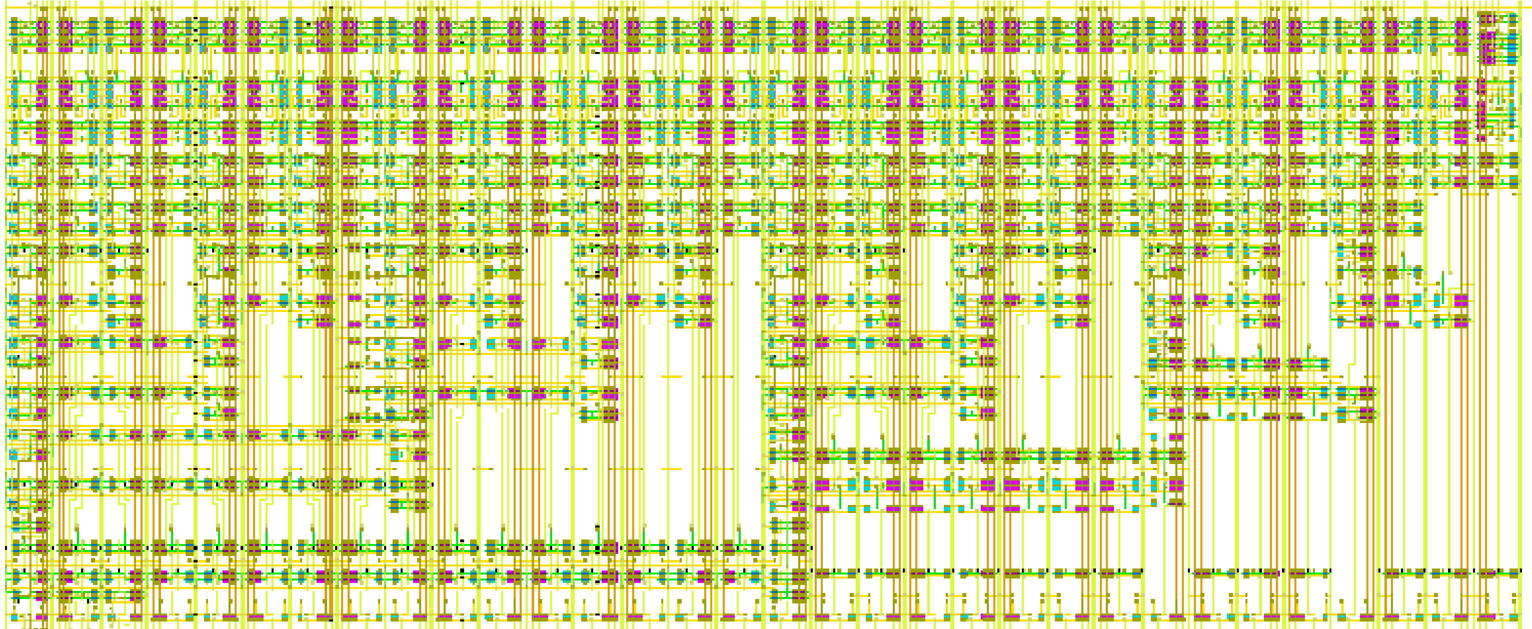


Left blocks can be bigger – more ripple time time while waiting for select

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

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

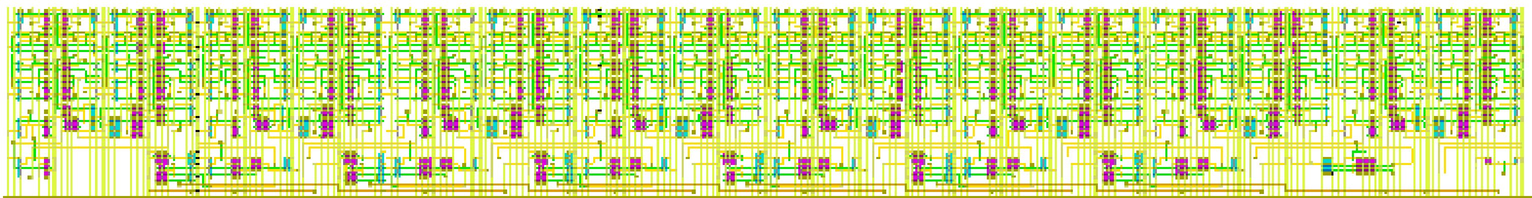
Adder layouts



⇧ 32-bit carry-select adder

MSB LSB

⇩ 32-bit carry-lookahead adder



Multiplication

				A_3	A_2	A_1	A_0
			x	B_3	B_2	B_1	B_0
				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			

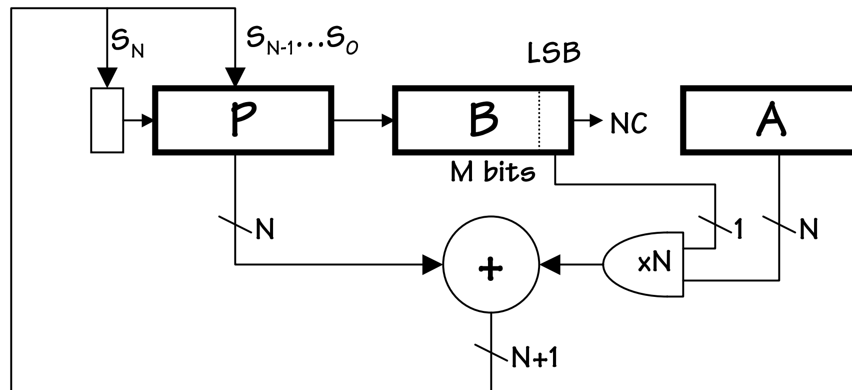
AB_i called a "partial product" → (first row of partial products)

Multiplying N-bit number by M-bit number gives (N+M)-bit result

Easy part: forming partial products (just an AND gate since B_I is either 0 or 1)
Hard part: adding M N-bit partial products

Sequential Multiplier

Assume the multiplicand (A) has N bits and the multiplier (B) has M bits. If we only want to invest in a single N-bit adder, we can build a sequential circuit that processes a single partial product at a time and then cycle the circuit M times:



Init: P←0, load A&B

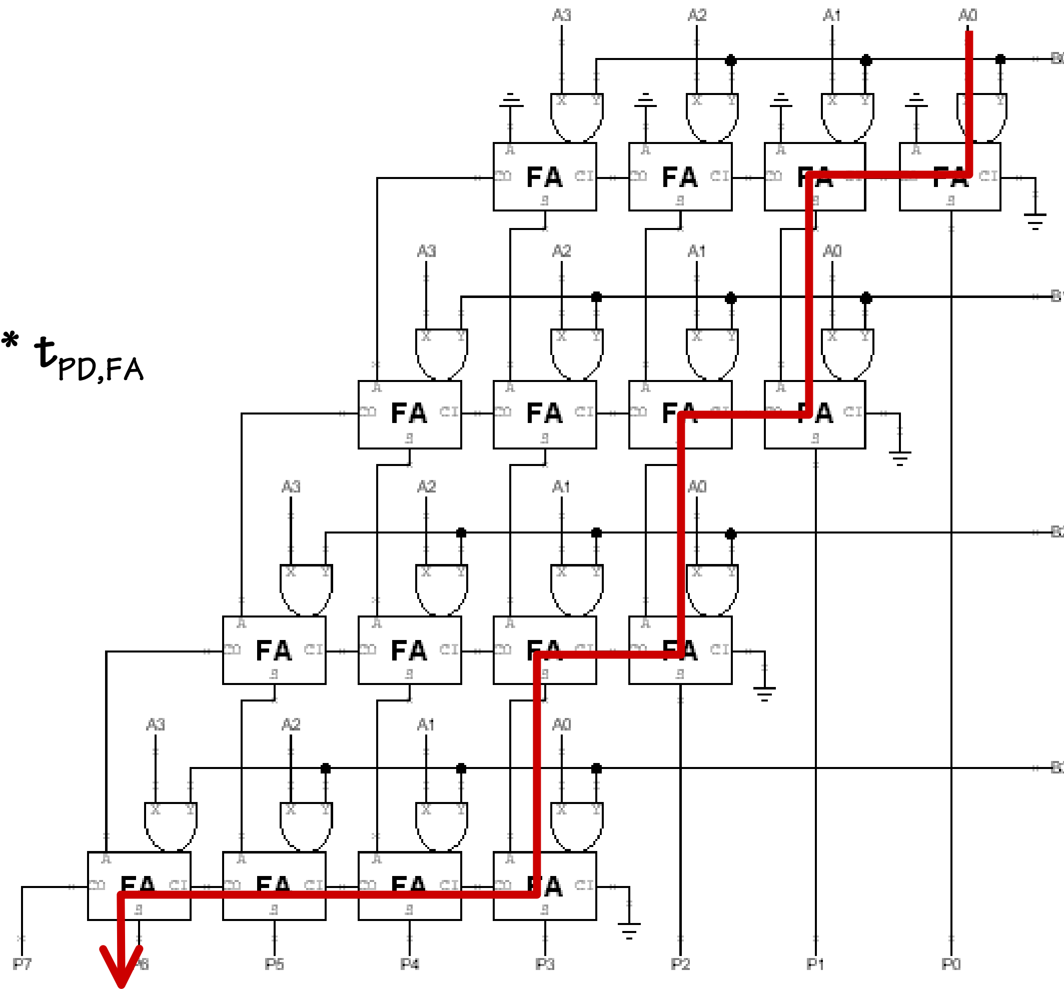
Repeat M times {
  P ←P + (B_{LSB}==1 ? A : 0)
  shift P/B right one bit
}

Done: (N+M)-bit result in P/B

Combinational Multiplier

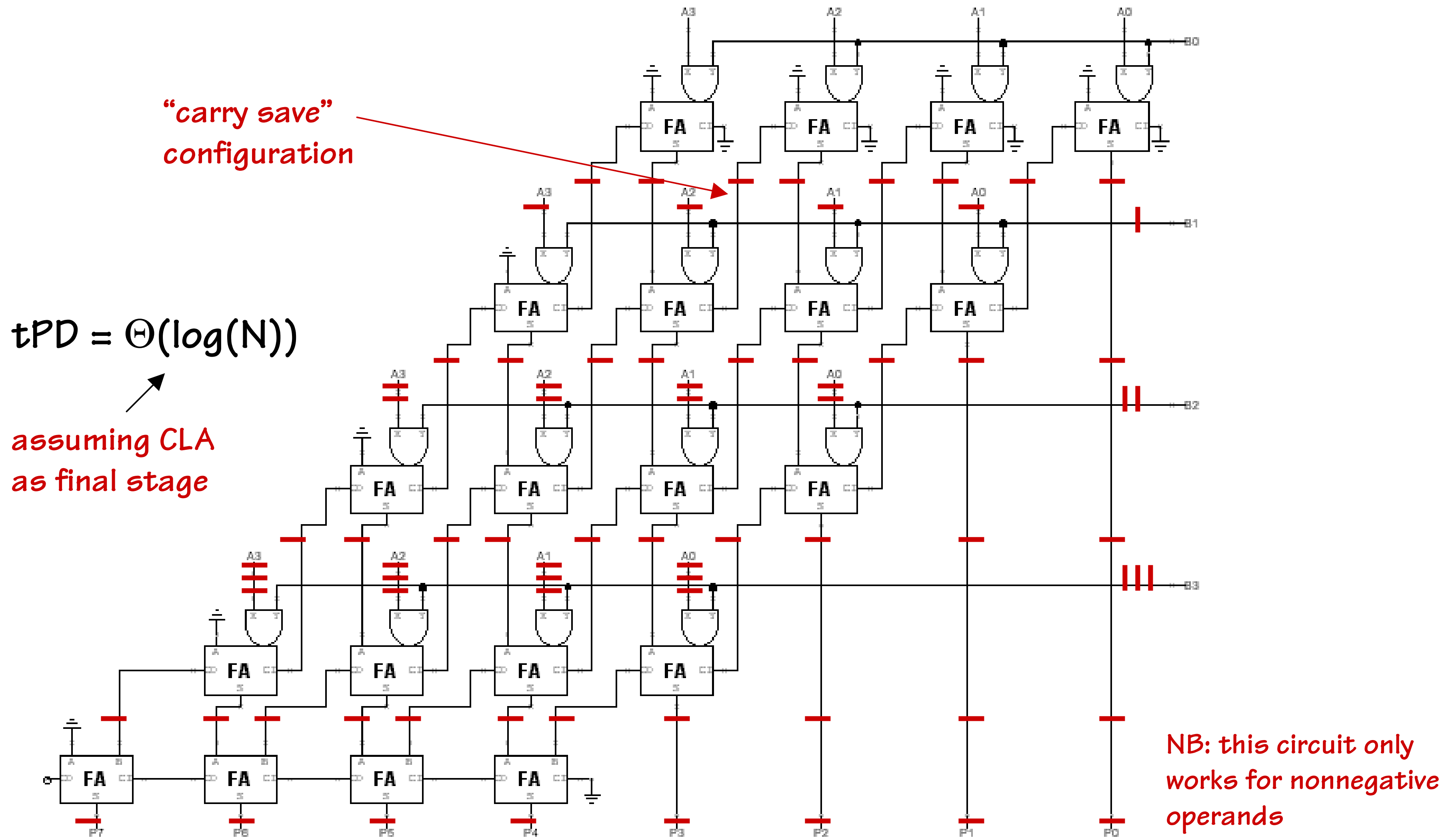
$t_{PD} = 10 * t_{PD,FA}$

not 16



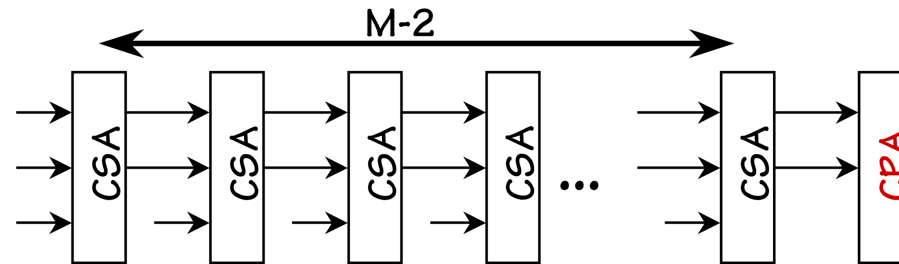
NB: this circuit only works for nonnegative operands

Pipelined Multiplier

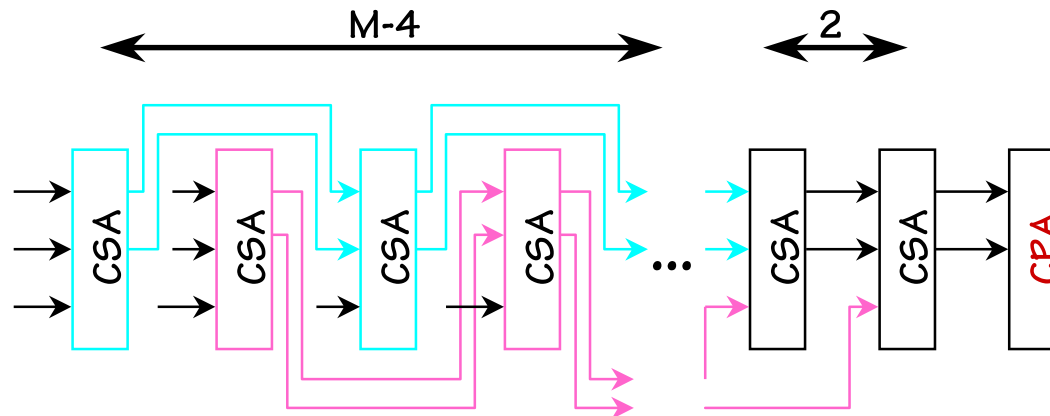


Latency Improvements

Abstract carry-save picture :

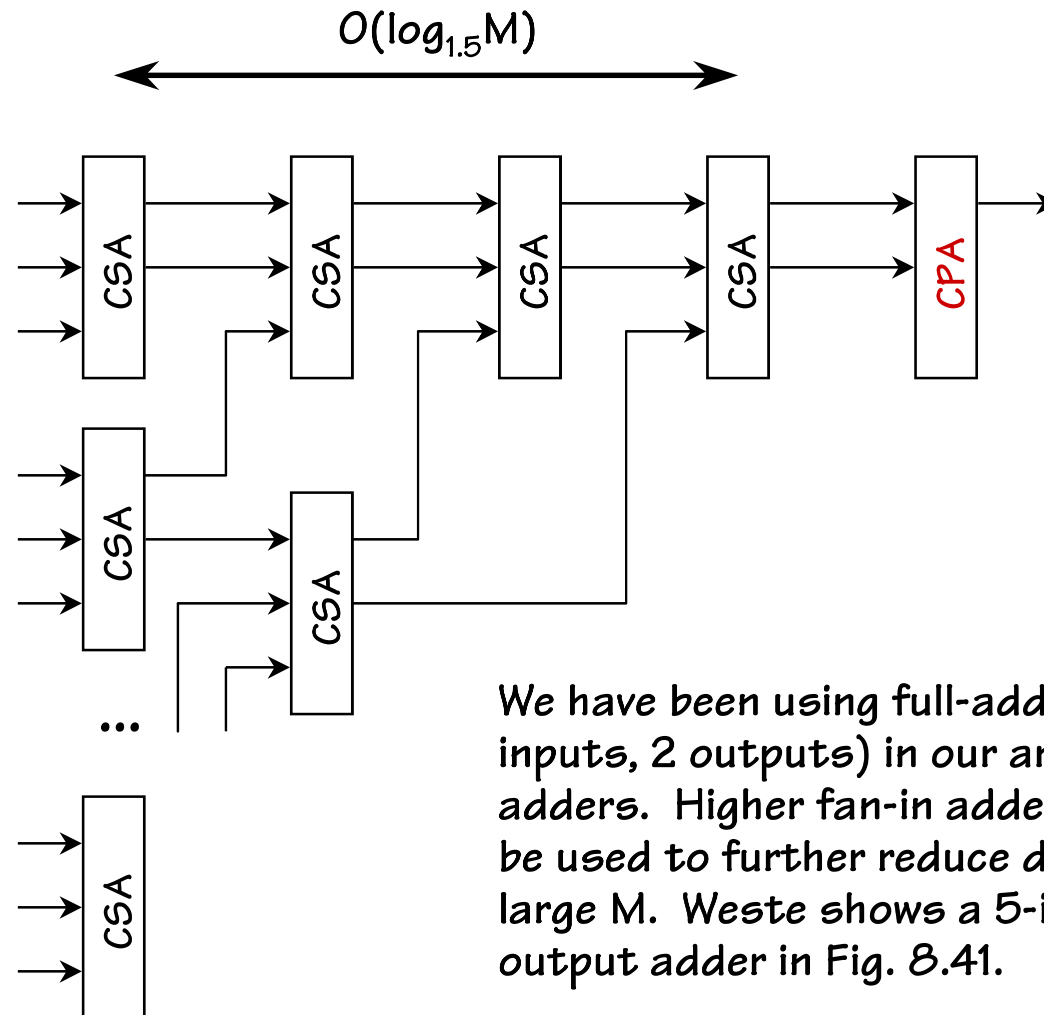


Rewire so that first two adders work in parallel. Feed results into third and fourth adders which also work in parallel, etc.



Even and odd streams pass through half the adders so even/odd design runs at almost twice the speed of simple CSA implementation.

More Latency Improvements

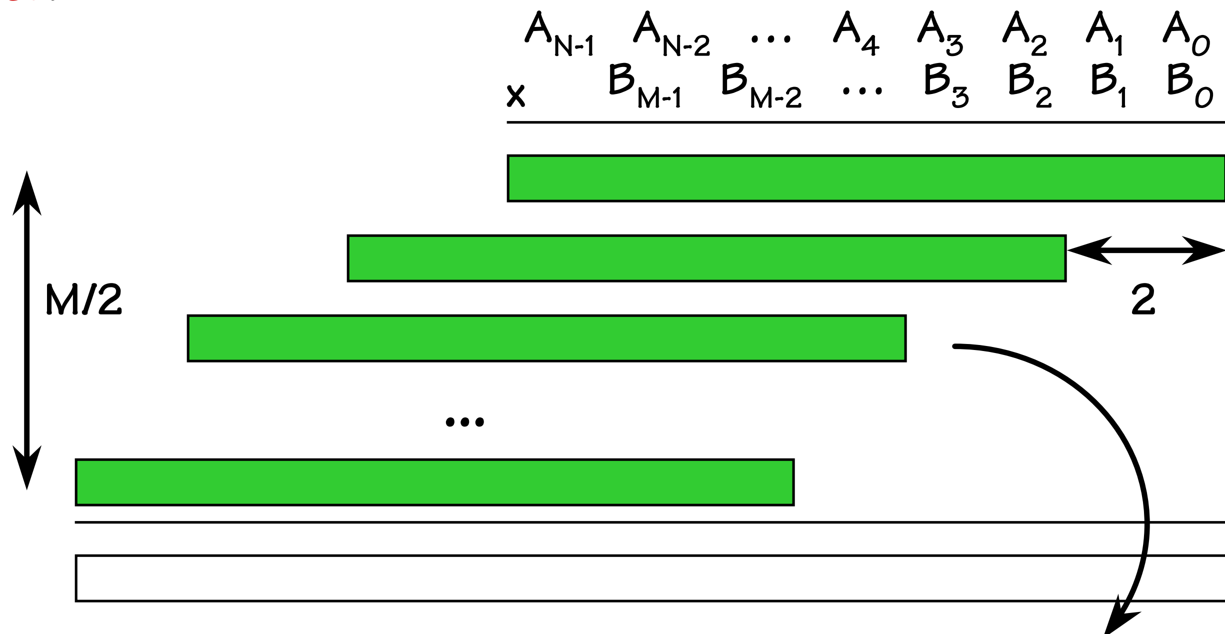


Wallace Tree

We have been using full-adders (3 inputs, 2 outputs) in our array adders. Higher fan-in adders can be used to further reduce delays for large M. Weste shows a 5-input, 3-output adder in Fig. 8.41.

Higher-radix multiplication

Idea: If we could use, say, 2 bits of the multiplier in generating each partial product we would **halve the number of columns and halve the latency of the multiplier**!



$B_{K+1,K}$*A = 0*A ⇨ 0
= 1*A ⇨ A
= 2*A ⇨ 4A – 2A
= 3*A ⇨ 4A – A

Booth's insight: rewrite 2*A and 3*A cases, leave 4A for next partial product to do!

Booth recoding

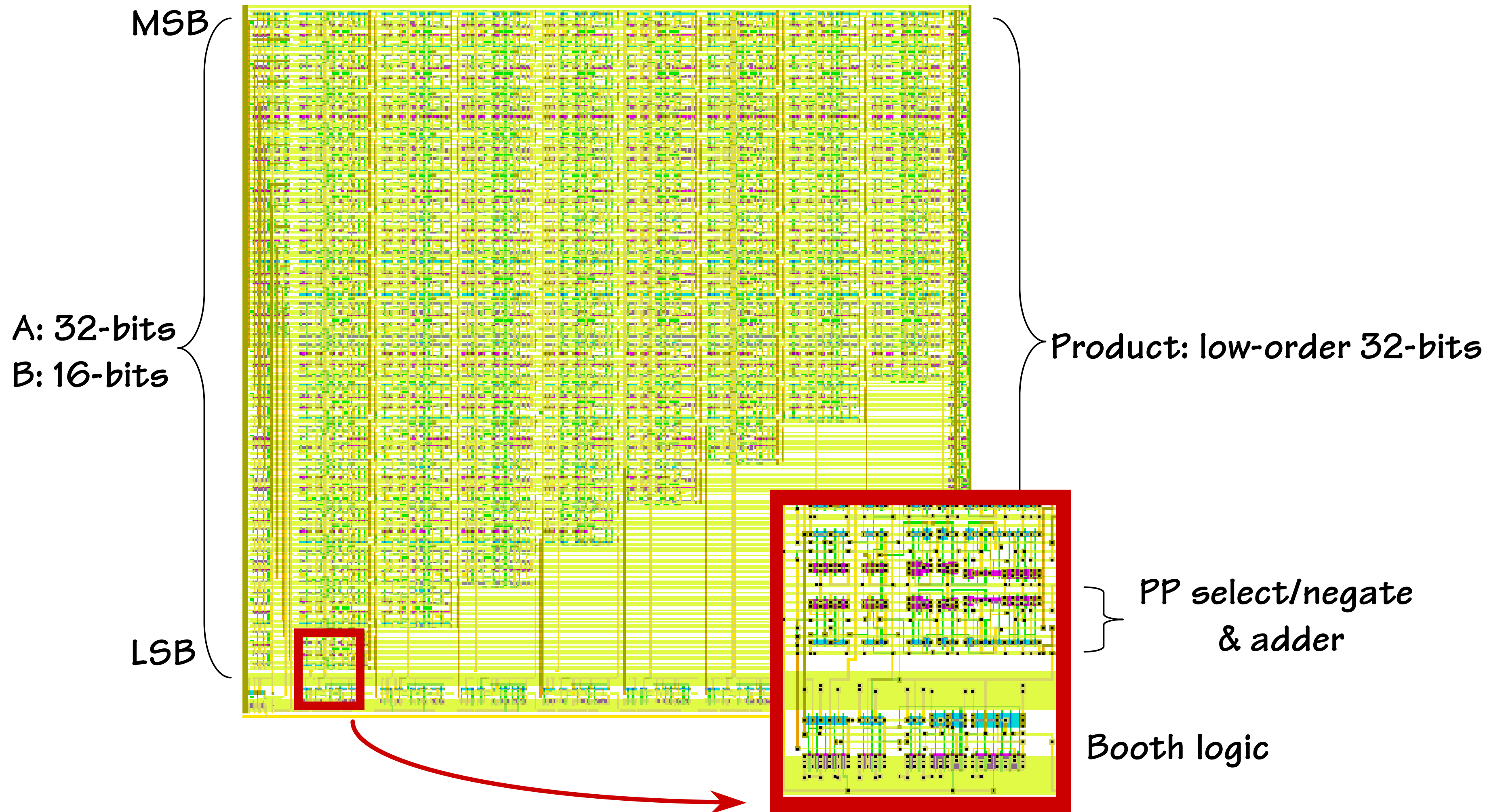
current bit pair → B_{K+1}, B_K

from previous bit pair → B_{K-1}

B_{K+1}	B_K	B_{K-1}	action	
0	0	0	add 0	
0	0	1	add A	
0	1	0	add A	
0	1	1	add 2*A	
1	0	0	sub 2*A	
1	0	1	sub A	← -2*A+A
1	1	0	sub A	
1	1	1	add 0	← -A+A

A "1" in this bit means the previous stage needed to add 4*A. Since this stage is shifted by 2 bits with respect to the previous stage, adding 4*A in the previous stage is like adding A in this stage!

Multiplier Layout



IEEE-754 Floating point

S *sign*	E *exponent*	F *significand*

$$\text{Number} = (-1)^S \times 2^{E\text{-bias}} \times (1.F)$$

Special values: ±0, ±∞, NaN, *denormalized numbers*

Exponent notes:

exponents of all 0's and all 1's are used for special values

biased notation means nonnegative numbers are ordered in the same way as integers

Floating point operations

Multiplication:

$$(s_1 \times 2^{e1}) \bullet (s_2 \times 2^{e2}) = (s_1 \bullet s_2) \times 2^{e1+e2}$$

Addition/subtraction:

(1) shift significand of smaller operand right, incrementing exponent, repeat until exponents are equal
(2) add (or subtract) significands
(3) normalize result by shifting left and decrementing exponent, repeat until there is a "1" to the left of the decimal point
(4) Round result

Next time:
we start to build a computer…

Good luck on tonight's Quiz!

What's *LUCK* got to do with it???

I'll let you know tomorrow.

2. If you arrive at the quiz in a metastable state what is the probability you will resolve into a valid state before the quiz concludes?

A. All of the above
B. None of the below
C. All of the above
D. One of the above
E. None of the above
F. None of the above

I knew there would be logic on this quiz, but this is not what I expected.