

Arithmetic Circuits

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

$$\begin{array}{r} 01011 \\ +00101 \\ \hline 10000 \end{array}$$

Finally; time to build some serious functional blocks



We'll need a lot of boxes



Reading: Study Chapter 3.
(Chapter 4 in old book)

Comp 120 Spring 2005

2/10/05

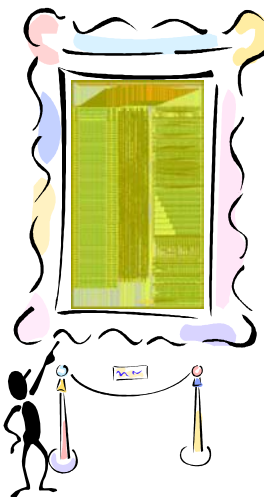
LO9 - Arithmetic Circuits 1

Today's BIG Picture

Thus far, we've been designing "in the small", concentrating on technique, rather than the design of useful systems. Today, we make a HUGE leap to designing in the large.

Our focus will be on designing combinational logic with so many inputs and outputs that they would be impractical to design using the logic optimization techniques we've discussed previously. These processing blocks are also impractical to implement using regular logic structures such as ROMs or PLAs.

Many of these blocks hinge on clever tricks that have withstood the test of time.

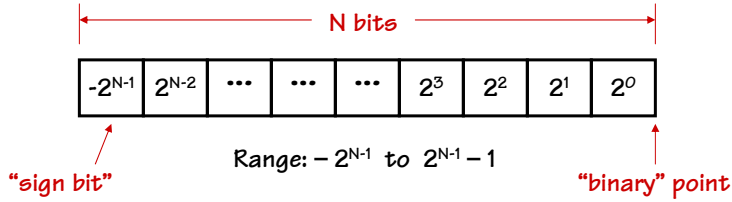


Comp 120 Spring 2005

2/10/05

LO9 - Arithmetic Circuits 2

Review: 2's Complement



8-bit 2's complement example:

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

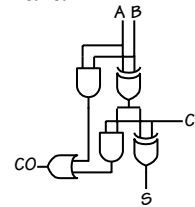
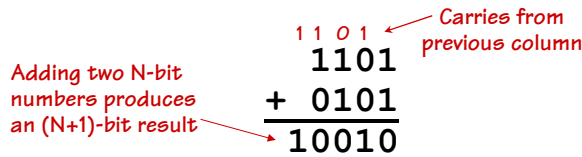
If we use a two's-complement representation for signed integers, the same binary addition procedure will work for adding both signed and unsigned numbers.

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

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

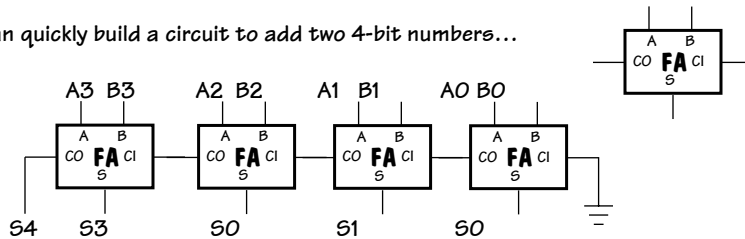
Binary Addition

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



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

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



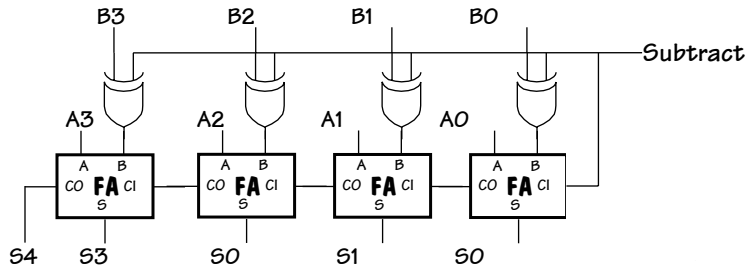
Subtraction: $A - B = A + (-B)$

Using 2's complement representation: $-B = \sim B + 1$

\sim = 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., " $(2^{i-1} - 1) + (2^{i-1} - 1)$ "

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

-or-

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

To compare A and B, perform $A - B$ and use condition codes:

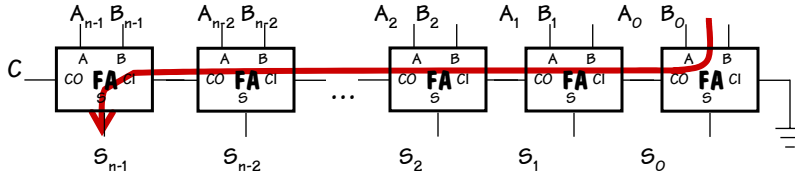
Signed comparison:

- LT $N \oplus V$
- LE $Z + (N \oplus V)$
- EQ Z
- NE $\sim Z$
- GE $\sim (N \oplus V)$
- GT $\sim (Z + (N \oplus V))$

Unsigned comparison:

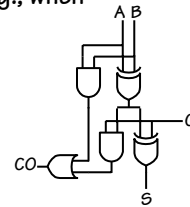
- LTU C
- LEU $C + Z$
- GEU $\sim C$
- GTU $\sim (C + Z)$

T_{PD} of Ripple-Carry Adder



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

$$t_{PD} = \underbrace{(t_{PD,XOR} + t_{PD,OR} + t_{PD,AND})}_{\text{XOR(A,B) to CO}} + \underbrace{(N-2) * (t_{PD,OR} + t_{PD,AND})}_{\text{CI to CO}} + \underbrace{t_{PD,XOR}}_{\text{CI}_{N-1} \text{ to } S_{N-1}} \approx \Theta(N)$$



$\Theta(N)$ 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} :

$$\begin{aligned} C_{OUT} &= AB + AC_{IN} + BC_{IN} \\ &= AB + (A + B)C_{IN} \\ &= G + P C_{IN} \quad \text{where } G = AB \text{ and } P = A + B \end{aligned}$$

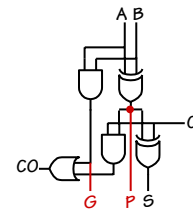
↑ generate ↑ propagate

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:

$$\begin{aligned} C_N &= G_{N-1} + P_{N-1}C_{N-1} \\ &= G_{N-1} + P_{N-1}G_{N-2} + P_{N-1}P_{N-2}C_{N-2} \\ &= G_{N-1} + P_{N-1}G_{N-2} + P_{N-1}P_{N-2}G_{N-3} + \dots + P_{N-1} \dots P_0 C_{IN} \end{aligned}$$

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

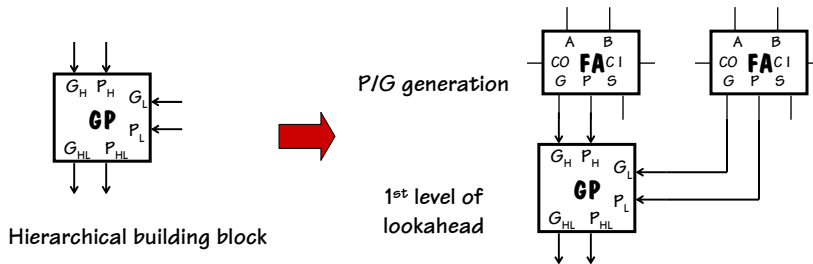
Carry-Lookahead Adders (CLA)

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:

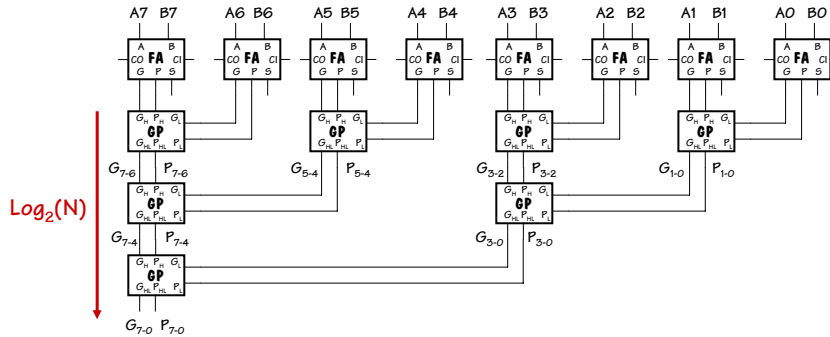
$$G_{HL} = G_H + P_H G_L$$

$$P_{HL} = P_H P_L$$

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.



8-bit CLA (GP Generation)



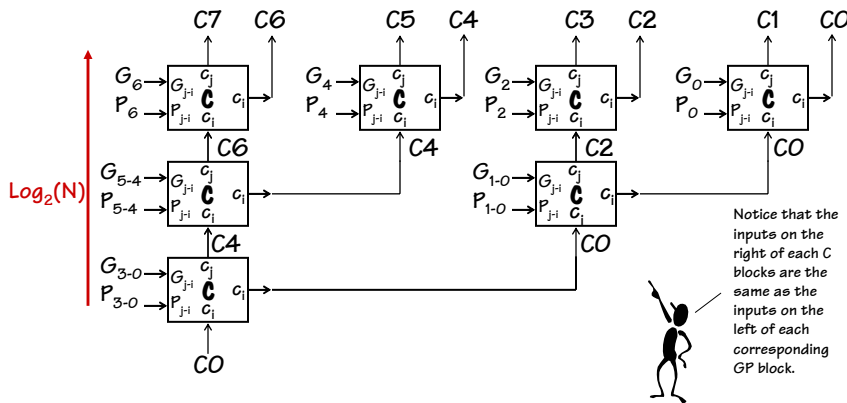
We can build a tree of GP units to compute the generate and propagate logic for any sized adder. For a 2^N -bit adder, we need $2^N - 1$ GP units.

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

8-bit CLA (Carry Generation)

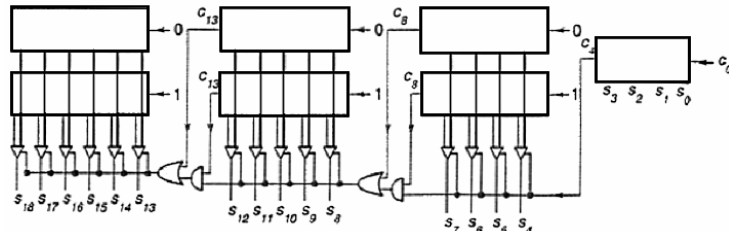
Now, given the value of the carry-in of the least-significant bit, we can generate the carries for every adder.

$$c_j = G_{j-1} + P_{j-1} c_i$$



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.

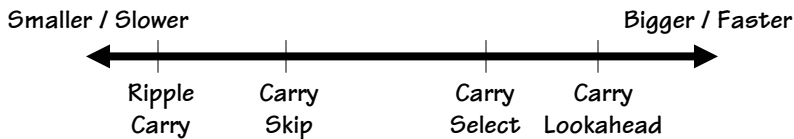


Left blocks can be bigger – more ripple time while waiting for select

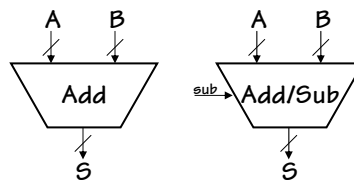
With one stage: 50% more cost, but twice as fast as ripple-carry
 With multiple (variable-size) blocks: $t_{PD} \rightarrow \Theta(\sqrt{N})$

Adder Summary

Adding is not only a common, but it 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.

$X \ll 1$ is approx the same as $2 * X$
 $X \gg 1$ can be the same as $X / 2$

For example:

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

Left Shift:

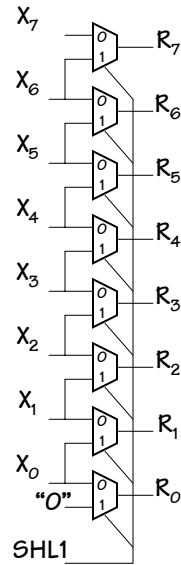
$$(X \ll 1) = 00101000_2 = 40_{10}$$

Right Shift:

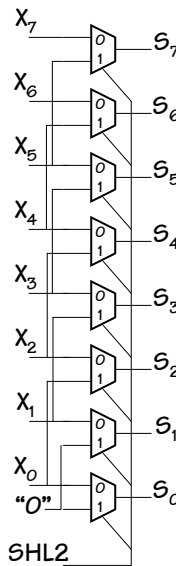
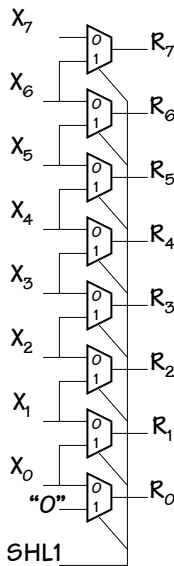
$$(X \gg 1) = 00001010_2 = 10_{10}$$

Signed or "Arithmetic" Right Shift:

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



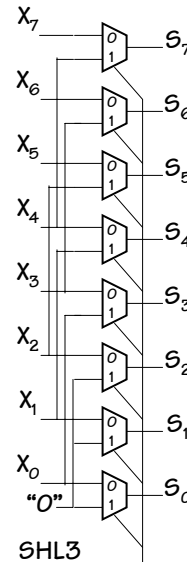
More Shifting



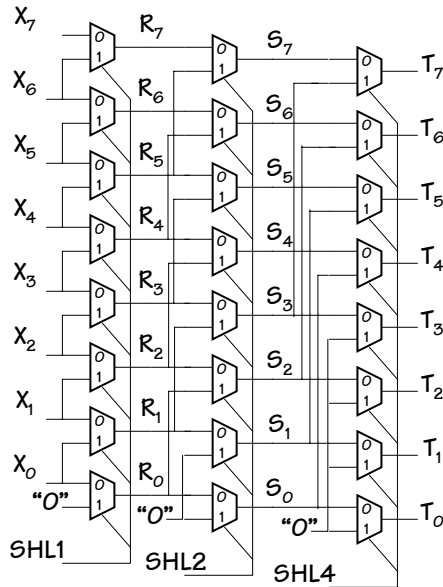
Using the same basic idea we can build left shifters of arbitrary sizes using muxes.

Each shift amount requires its own set of muxes.

Hum, maybe we could do something more clever.



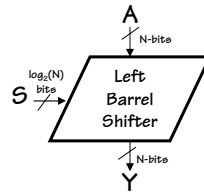
Barrel Shifting



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.

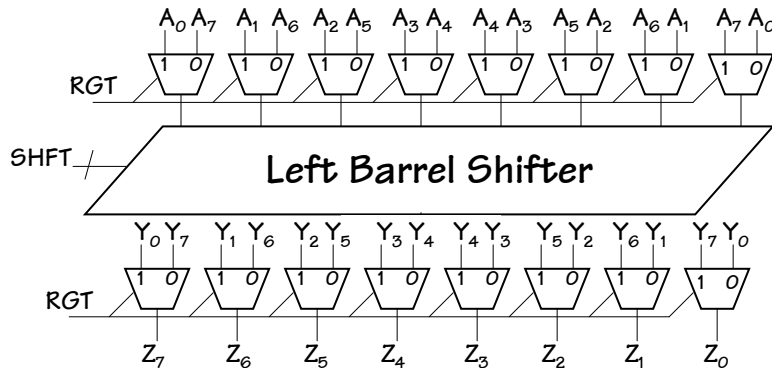
And, if we add one more "shift-left-4" shifter we can do any shift up to 7 bits!

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

It will also be useful 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. $10101110 \mid 00001111 = 10101111$ (1's set last 4 bits)

XORing is useful for "complementing" groups of bits.

ex. $10101110 \wedge 00001111 = 10100001$ (1's complement last 4 bits)

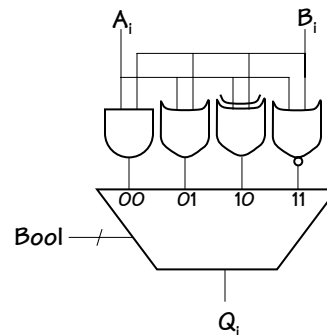
NORing is useful.. Uhm, because John Hennessy says it is!

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.



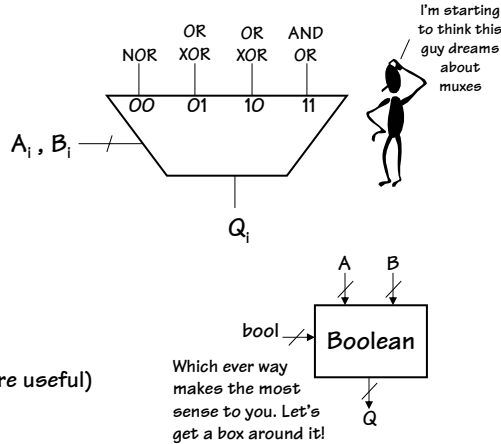
This is a straightforward, but not too elegant of a design.

Cooler Bools

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

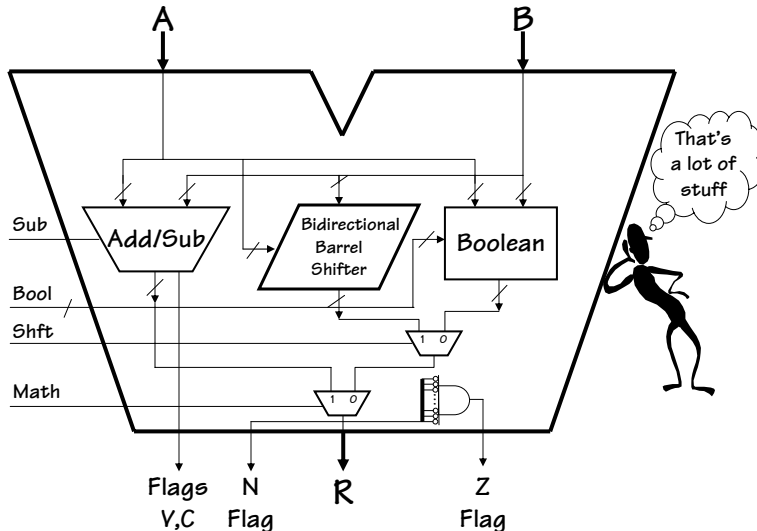
Why is this better?

- 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

Now we're ready for a big one! An Arithmetic Logic Unit.



Did We Forget Something?

Even rabbits know how to multiply.



But, it is a huge step in terms of logic...

Including a multiplier unit in an ALU can potentially double the number of gates used.

A good (compact and high performance) multiplier can also be tricky to design. Here we will give an overview of some of the tricks used.

Multiplication

The "Binary" Multiplication Table

X	0	1
0	0	0
1	0	1

Hey, that looks like an AND gate



Binary multiplication is implemented using the same basic longhand algorithm that you learned in grade school.

$$\begin{array}{r} A_3 \quad A_2 \quad A_1 \quad A_0 \\ x \quad B_3 \quad B_2 \quad B_1 \quad B_0 \end{array}$$

AB_i called a "partial product" →

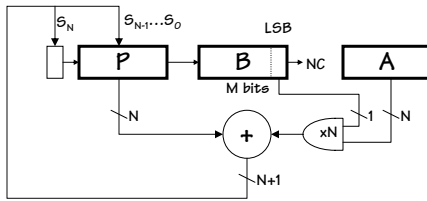
$$\begin{array}{r} A_3B_0 \quad A_2B_0 \quad A_1B_0 \quad A_0B_0 \\ A_3B_1 \quad A_2B_1 \quad A_1B_1 \quad A_0B_1 \\ A_3B_2 \quad A_2B_2 \quad A_1B_2 \quad A_0B_2 \\ + \quad A_3B_3 \quad A_2B_3 \quad A_1B_3 \quad A_0B_3 \end{array}$$

Multiplying N-digit number by M-digit number gives (N+M)-digit 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: $P \leftarrow 0$, load A & B

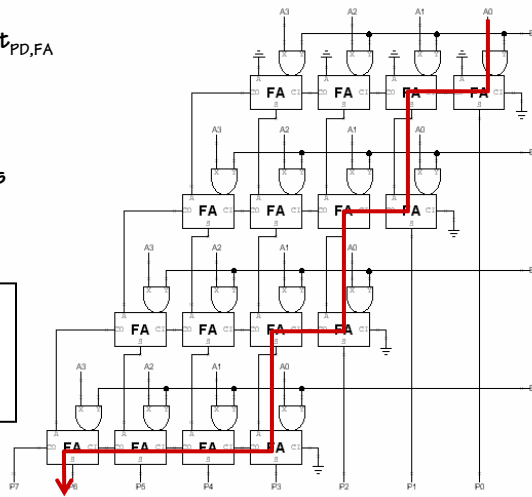
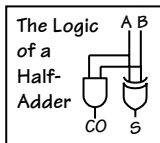
Repeat M times {
 $P \leftarrow P + (B_{LSB} == 1 ? A : 0)$
 shift P/B right one bit
}

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

Simple Combinational Multiplier

$t_{PD} = 10 * t_{PD,FA}$
 not 16

Components
 $N * HA$
 $N(N-1) * FA$



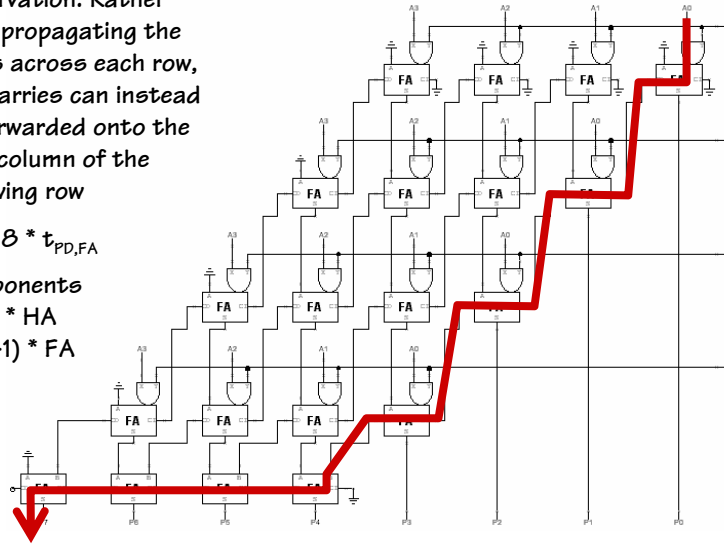
NB: this circuit only works for nonnegative operands

Carry-Save Combinational Multiplier

Observation: Rather than propagating the sums across each row, the carries can instead be forwarded onto the next column of the following row

$$t_{PD} = \delta * t_{PD,FA}$$

Components
 $2N * HA$
 $N(N-1) * FA$



This small improvement in performance hardly seems worth the effort, however, this design is easier to pipeline.

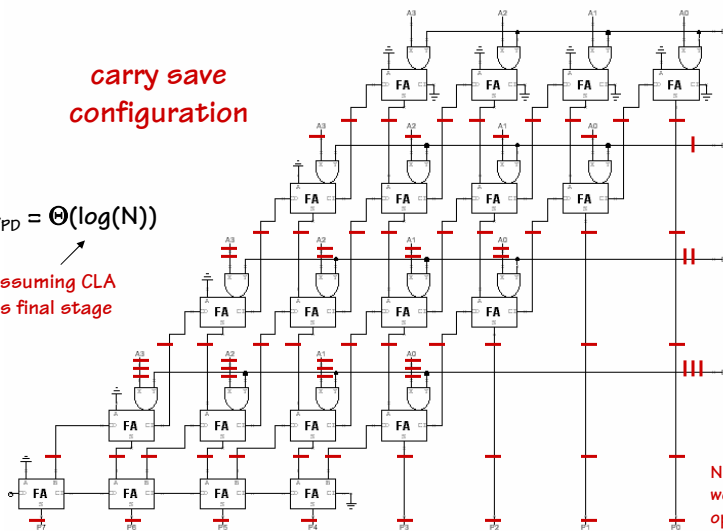


Pipelined Multiplier

carry save configuration

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

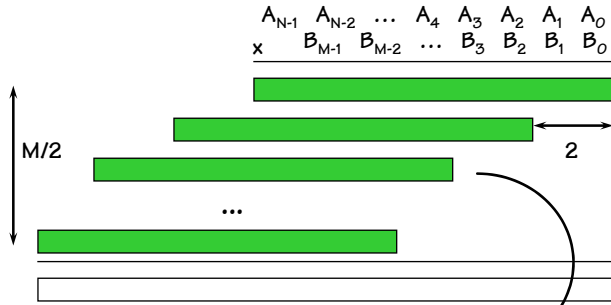
assuming CLA as final stage



NB: this circuit only works for nonnegative operands

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!**



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

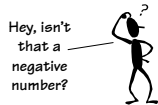
$$\begin{aligned}
 B_{K+1,K}^*A &= 0^*A \Rightarrow 0 \\
 &= 1^*A \Rightarrow A \\
 &= 2^*A \Rightarrow 2A \text{ or } 4A - 2A \\
 &= 3^*A \Rightarrow 4A - A
 \end{aligned}$$

Booth Recoding

	current bit pair		from previous bit pair	action	
	B_{2K+1}	B_{2K}	B_{2K-1}		
$-89 = 10100111.0$	0	0	0	add 0	
$= -1 * 2^0 \quad (-1)$	0	0	1	add A	
$+ 2 * 2^2 \quad (8)$	0	1	0	add A	
$+ (-2) * 2^4 \quad (-32)$	0	1	1	add 2^*A	
$+ (-1) * 2^6 \quad (-64)$	1	0	0	sub 2^*A	
	1	0	1	sub A	$\leftarrow -2^*A + A$
	1	1	0	sub A	
	1	1	1	add 0	$\leftarrow -A + A$

Each bit can be considered to have the following weights:

$$\begin{aligned}
 W(B_{2K+1}) &= 2 \\
 W(B_{2K}) &= -1 \\
 W(B_{2K-1}) &= -1
 \end{aligned}$$

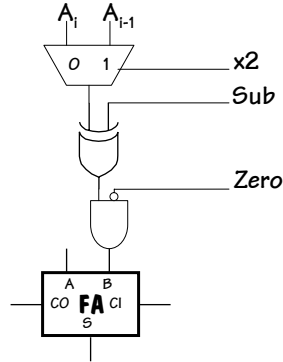


A "1" in this bit means the previous stage needed to add 4^*A . Since this stage is shifted by 2 bits with respect to the previous stage, adding 4^*A in the previous stage is like adding A in this stage!

Booth Recoding

Logic surrounding each basic adder:

- Control lines (x2, Sub, Zero) are shared across each row
- Must handle the "+1" when Sub is 1 (extra half adders in a carry save array)



NOTE:

- Booth recoding can be used to implement signed multiplications

B_{2k+1}	B_{2k}	B_{2k-1}	x2	Sub	Zero
0	0	0	X	X	1
0	0	1	0	0	0
0	1	0	0	0	0
0	1	1	1	0	0
1	0	0	1	1	0
1	0	1	0	1	0
1	1	0	1	1	0
1	1	1	X	X	1

Finally, it's Starting to Look Like a Computer...

Good luck on next week's quizz!

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

