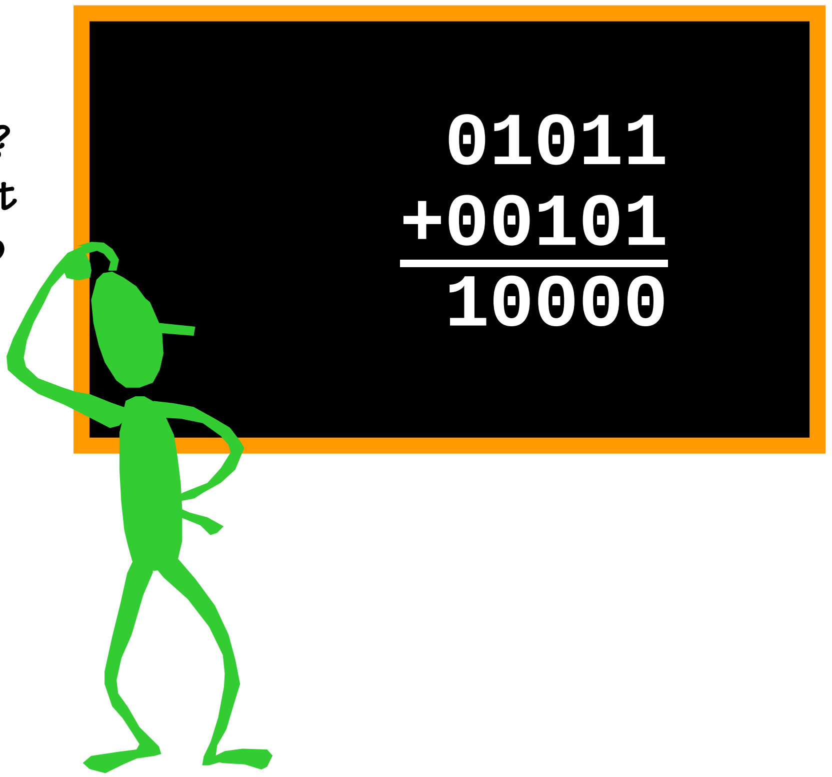
Arithmetic Circuits

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



Acknowledgements:

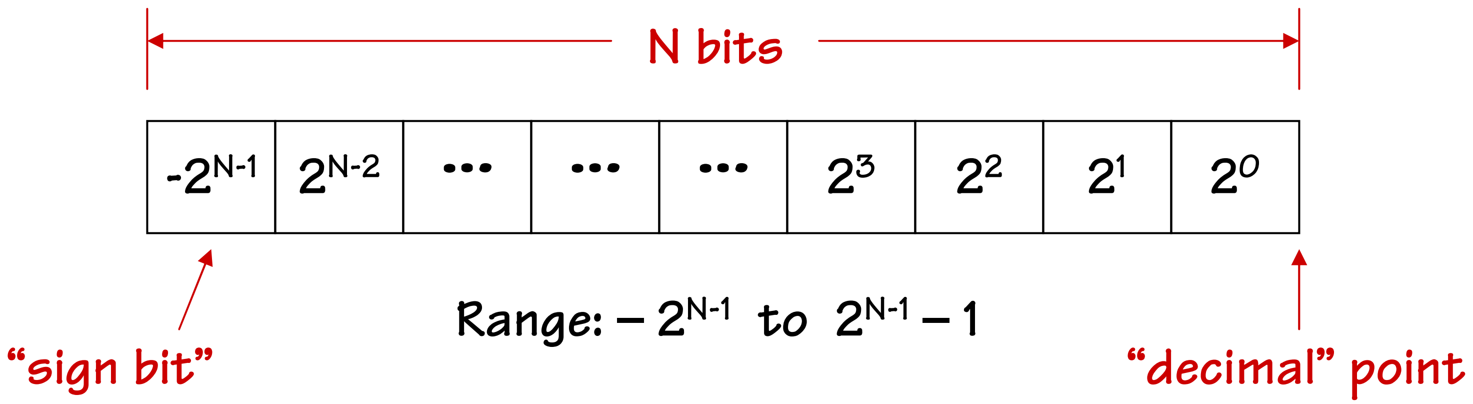
• R. Katz, "*Contemporary Logic Design*", Addison Wesley Publishing Company, Reading, MA, 1993. (Chapter 5)
• J. Rabaey, A. Chandrakasan, B. Nikolic, "Digital Integrated Circuits: A Design Perspective" Prentice Hall, 2003.
• Kevin Atkinson, Alice Wang, Rex Min

Number Systems Basics

How to represent negative numbers?

- Three common schemes: sign-magnitude, ones complement, twos complement
- Sign-magnitude: MSB = 0 for positive, 1 for negative
  - Range: $-(2^{N-1} - 1)$ to $+(2^{N-1} - 1)$
  - Two representations for zero: 0000… & 1000…
  - Simple multiplication but complicated addition/subtraction
- Ones complement: if N is positive then its negative is $\overline{N}$
  - Example: 0111 = 7, 1000 = -7
  - Range: $-(2^{N-1} - 1)$ to $+(2^{N-1} - 1)$
  - Two representations for zero: 0000… & 1111…
  - Subtraction implemented as addition followed by ones complement

2's Complement



Range: -2^{N-1} to $2^{N-1}-1$

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.

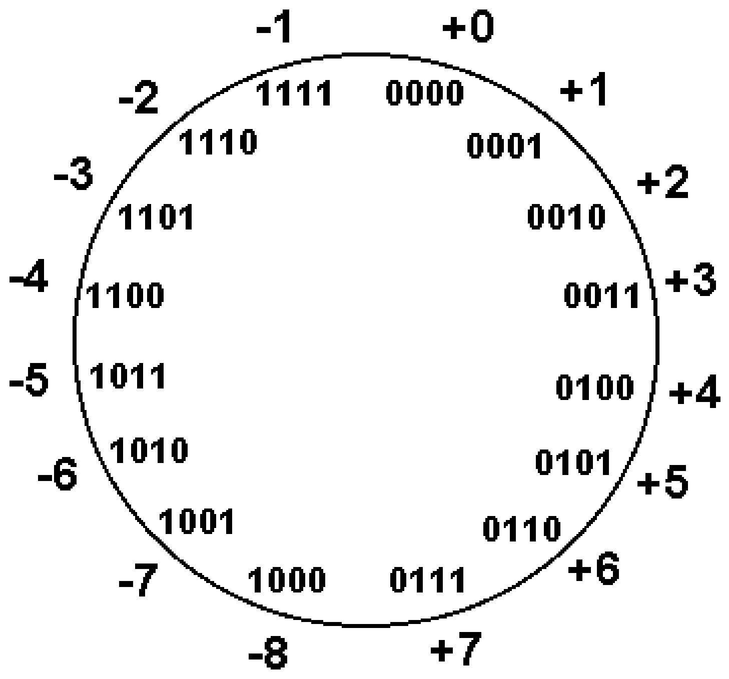
By moving the implicit location of "decimal" point, we can represent fractions too:

$$1101.0110 = -2^3 + 2^2 + 2^0 + 2^{-2} + 2^{-3} = -8 + 4 + 1 + 0.25 + 0.125 = -2.25$$

Twos Complement Representation

Twos complement = bitwise complement + 1

0111 → 1000 + 1 = 1001 = -7
1001 → 0110 + 1 = 0111 = 7



- **Asymmetric range: -2^{N-1} to $+2^{N-1}-1$**
- **Only one representation for zero**
- **Simple addition and subtraction**
- **Most common representation**

```
   4   0100        -4   1100        4   0100        -4   1100
+  3   0011   + (-3)    1101      - 3   1101      +  3   0011
   7   0111        -7  11001        1  10001        -1   1111
```

[Katz93, chapter 5]

Binary Addition

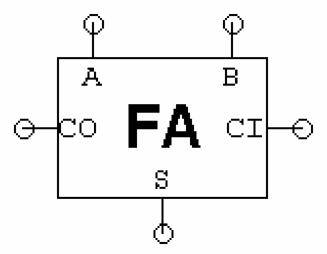
Here's an example of binary addition as one might do it by "hand":

Carries from previous column

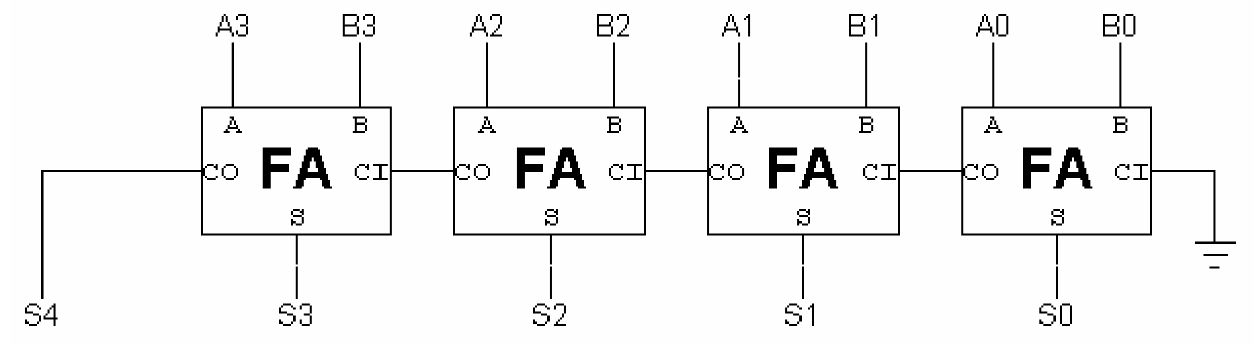
```
 1 1  0 1
    1101
+   0101
   -----
   10010
```

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

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



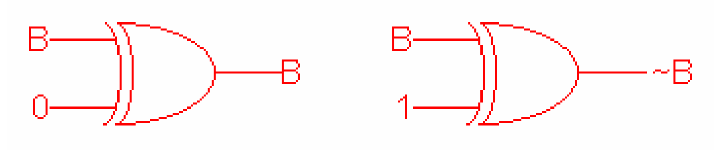
So we can quickly build a circuit two add two 4-bit numbers…



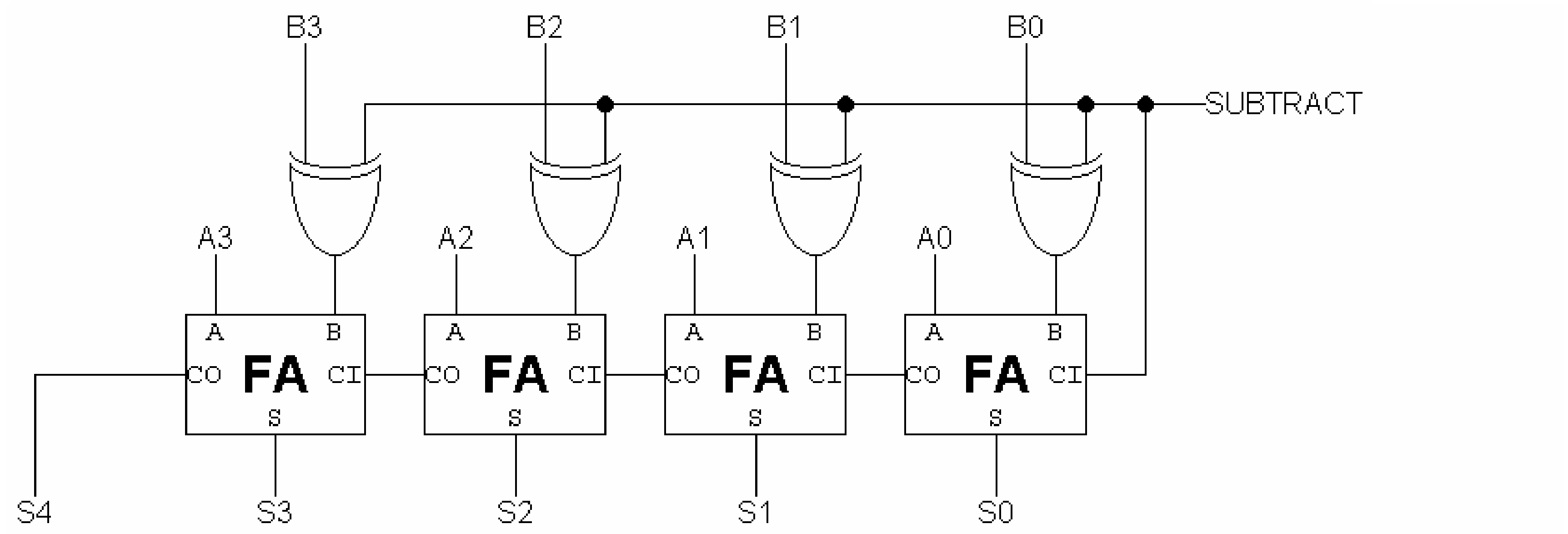
Subtraction: A-B = A + (-B)

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., "$(2^{N-1} - 1) + (2^{N-1} - 1)$"

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

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

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

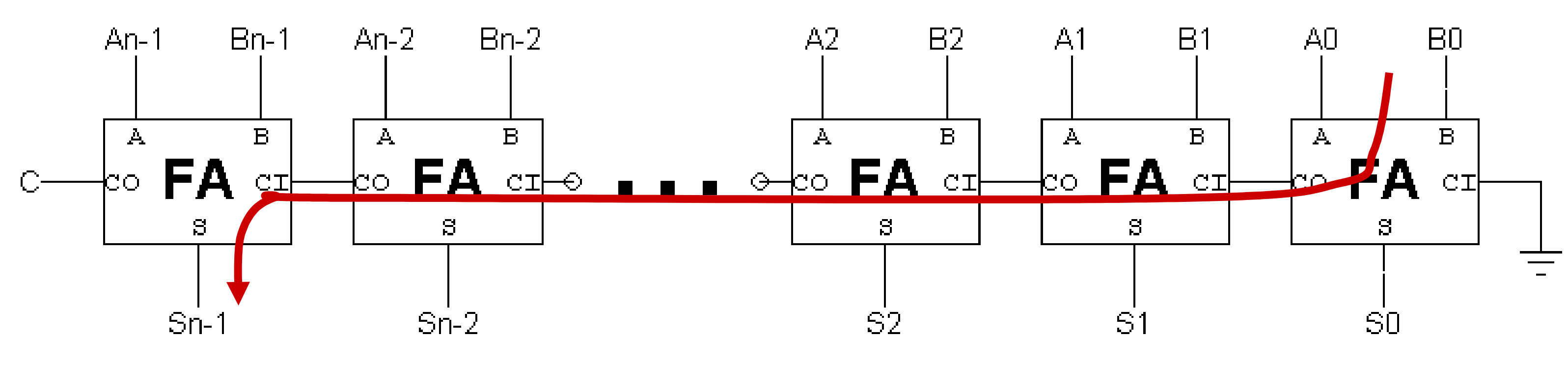
Signed comparison:

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.

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

Θ(N) is read “order N” and tells us that the latency of our adder grows proportional 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}:

$$\begin{aligned} C_{OUT} &= AB + AC_{IN} + BC_{IN} \\ &= AB + (A + B)C_{IN} \\ &= \underset{\text{generate}}{G} + \underset{\text{propagate}}{P}\, C_{IN} \end{aligned}$$

where $G = AB$ and $P = A + B$

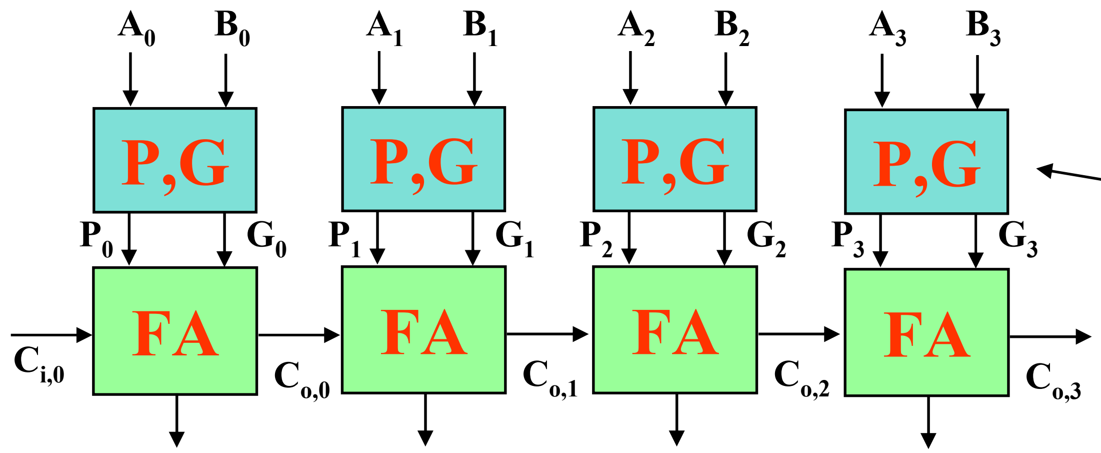
For adding two N-bit numbers:

$$\begin{aligned} C_N &= G_{N-1} + P_{N-1}C_{N-1} \\ &= G_{N-1} + P_{N-1}G_{N-2} + P_{N-1}P_{N-2}C_{N-2} \\ &= \underbrace{G_{N-1} + P_{N-1}G_{N-2} + P_{N-1}P_{N-2}G_{N-3} + \ldots + P_{N-1}\ldots P_0C_{IN}} \end{aligned}$$

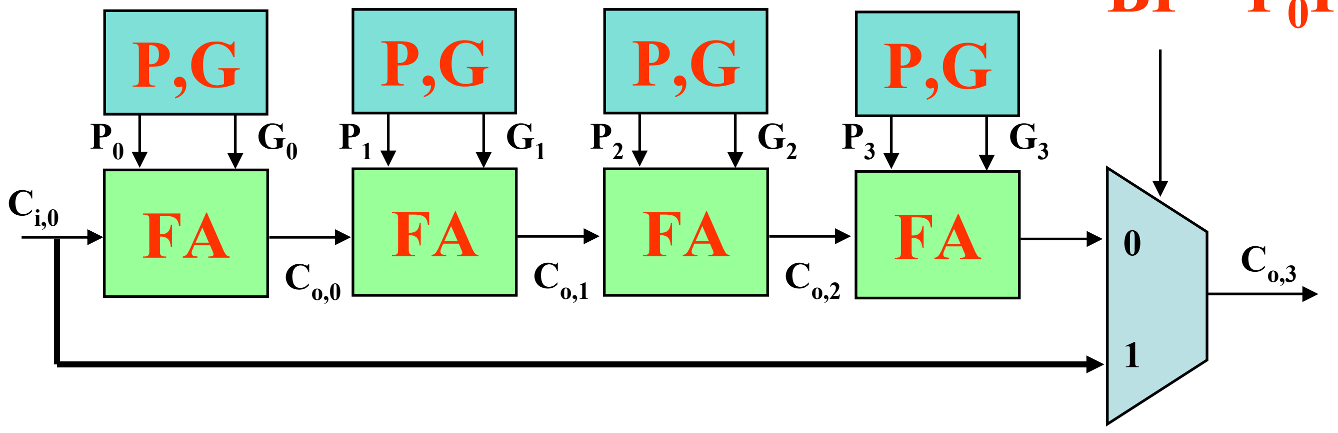
C_N in only 3 (!) gate delays:
1 for P/G generation, 1 for ANDs, 1 for final OR

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}: $S = P \oplus C_{IN}$

Carry Bypass Adder



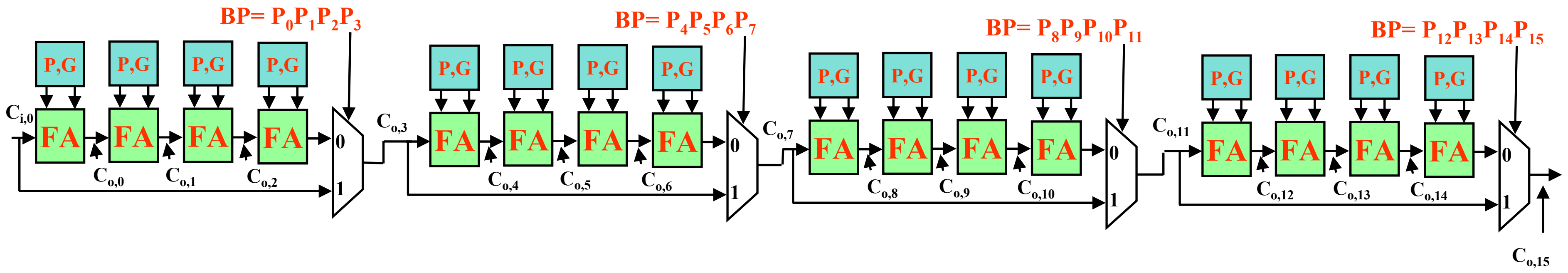
Can compute P, G in parallel for all bits



$BP = P_0 P_1 P_2 P_3$

Key Idea: if $(P_0\ P_1\ P_2\ P_3)$ then $C_{o,3} = C_{i,0}$

16-bit Carry Bypass Adder



Assume the following for delay each gate:

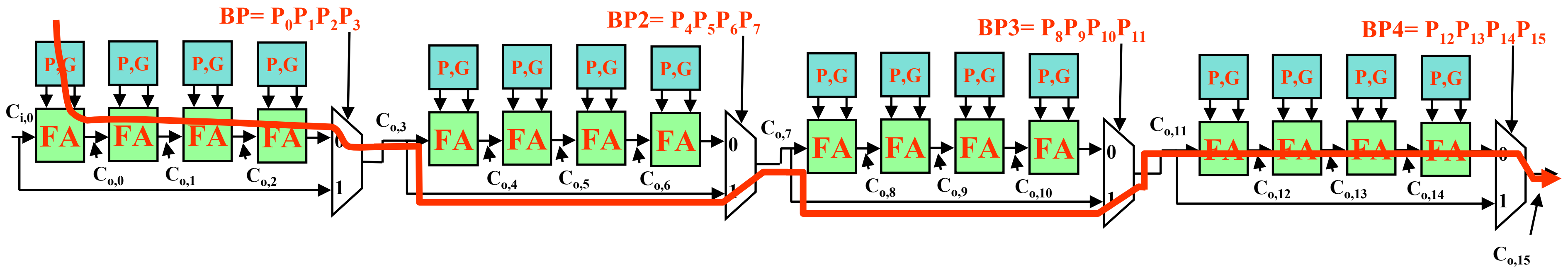
P, G from A, B: 1 delay unit

P, G, C_i to C_o or Sum for a FA: 1 delay unit

2:1 mux delay: 1 delay unit

What is the worst case propagation delay for the 16-bit adder?

Critical Path Analysis



For the second stage, is the critical path:

BP2 = 0 or BP2 = 1?

Message: Timing Analysis is Very Tricky – Must Carefully Consider Data Dependencies For *False Paths*

Carry-lookahead Adders (CLA)

We can choose the maximum fan-in we want for our logic gates and then build a hierarchical carry chain using these equations:

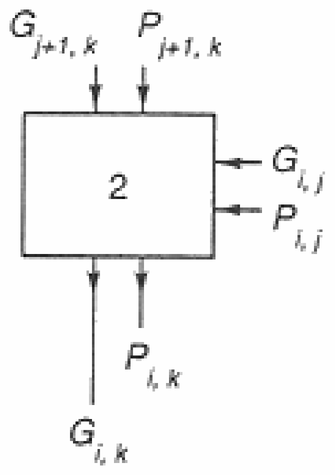
$$C_{J+1} = G_{IJ} + P_{IJ}C_I$$

$$G_{IK} = G_{J+1,K} + P_{J+1,K}\, G_{IJ}$$

$$P_{IK} = P_{IJ}\, P_{J+1,K}$$

where $I < J$ and $J+1 < K$

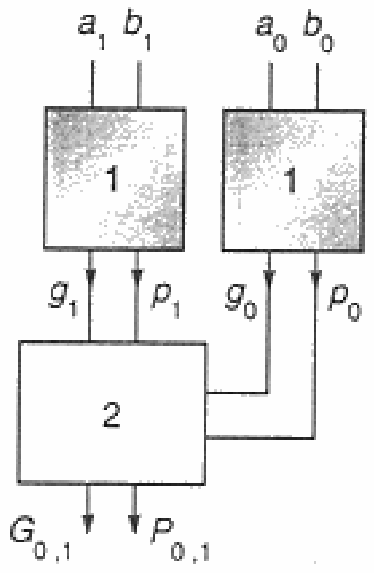
"generate a carry from bits I thru K if it is generated in the high-order (J+1,K) part of the block or if it is generated in the low-order (I,J) part of the block and then propagated thru the high part"



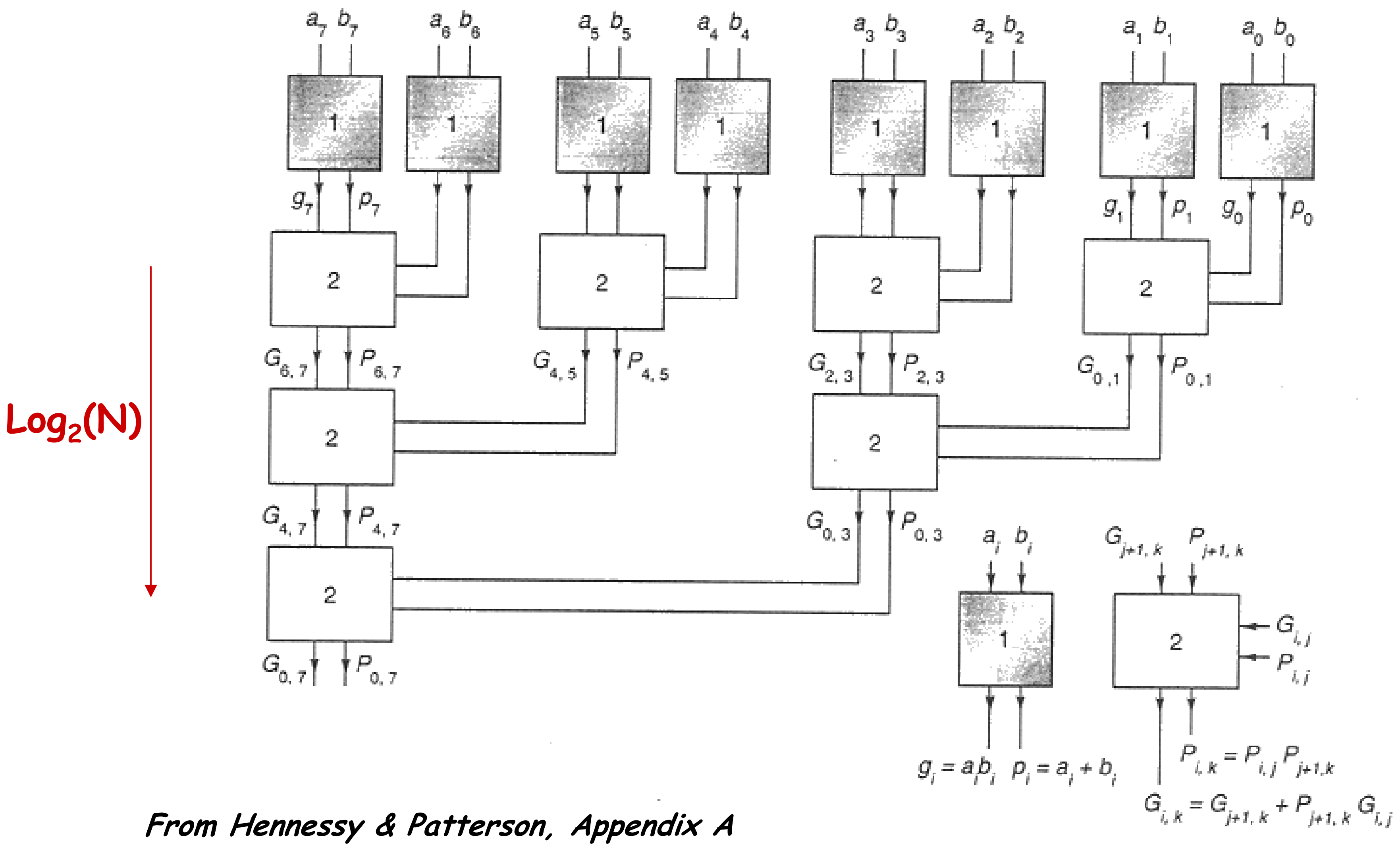
Hierarchical building block

P/G generation

1st level of lookahead

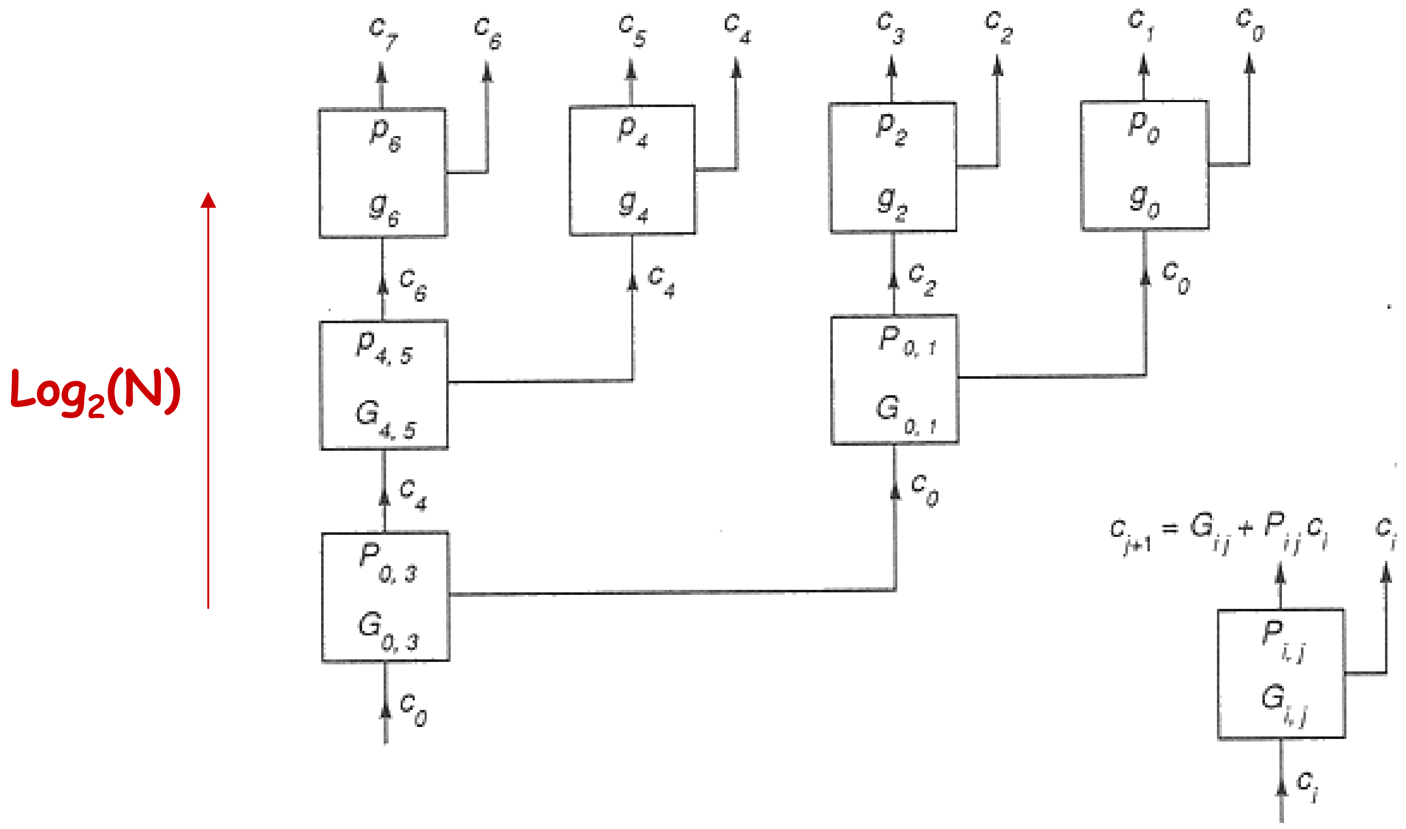


8-bit CLA (P/G generation)

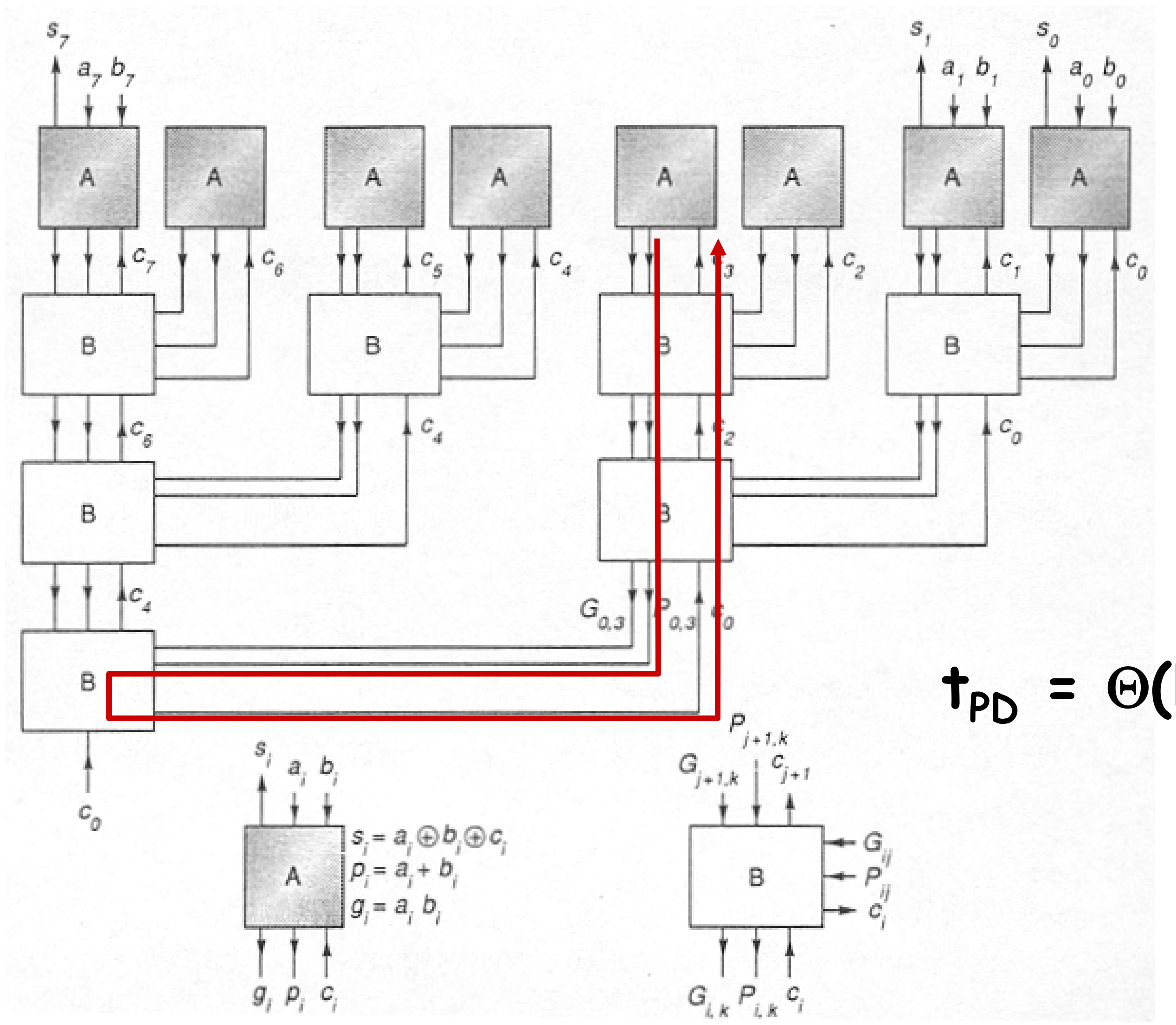


From Hennessy & Patterson, Appendix A

8-bit CLA (carry generation)



8-bit CLA (complete)



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

Unsigned Multiplication

$$\begin{array}{rcccccccc}
 & & & & & A_3 & A_2 & A_1 & A_0\\
 & & & & \times & B_3 & B_2 & B_1 & B_0\\
\hline
\text{AB}_i \text{ called a "partial product"} \longrightarrow & & & & & A_3B_0 & A_2B_0 & A_1B_0 & A_0B_0\\
 & & & & A_3B_1 & A_2B_1 & A_1B_1 & A_0B_1 & \\
 & & & A_3B_2 & A_2B_2 & A_1B_2 & A_0B_2 & & \\
 & + & A_3B_3 & A_2B_3 & A_1B_3 & A_0B_3 & & & \\
\hline
\end{array}$$

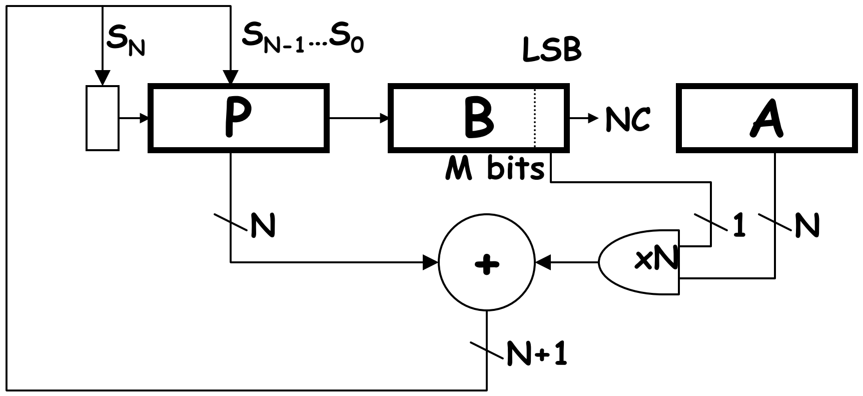
$$\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}_{}$$

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:

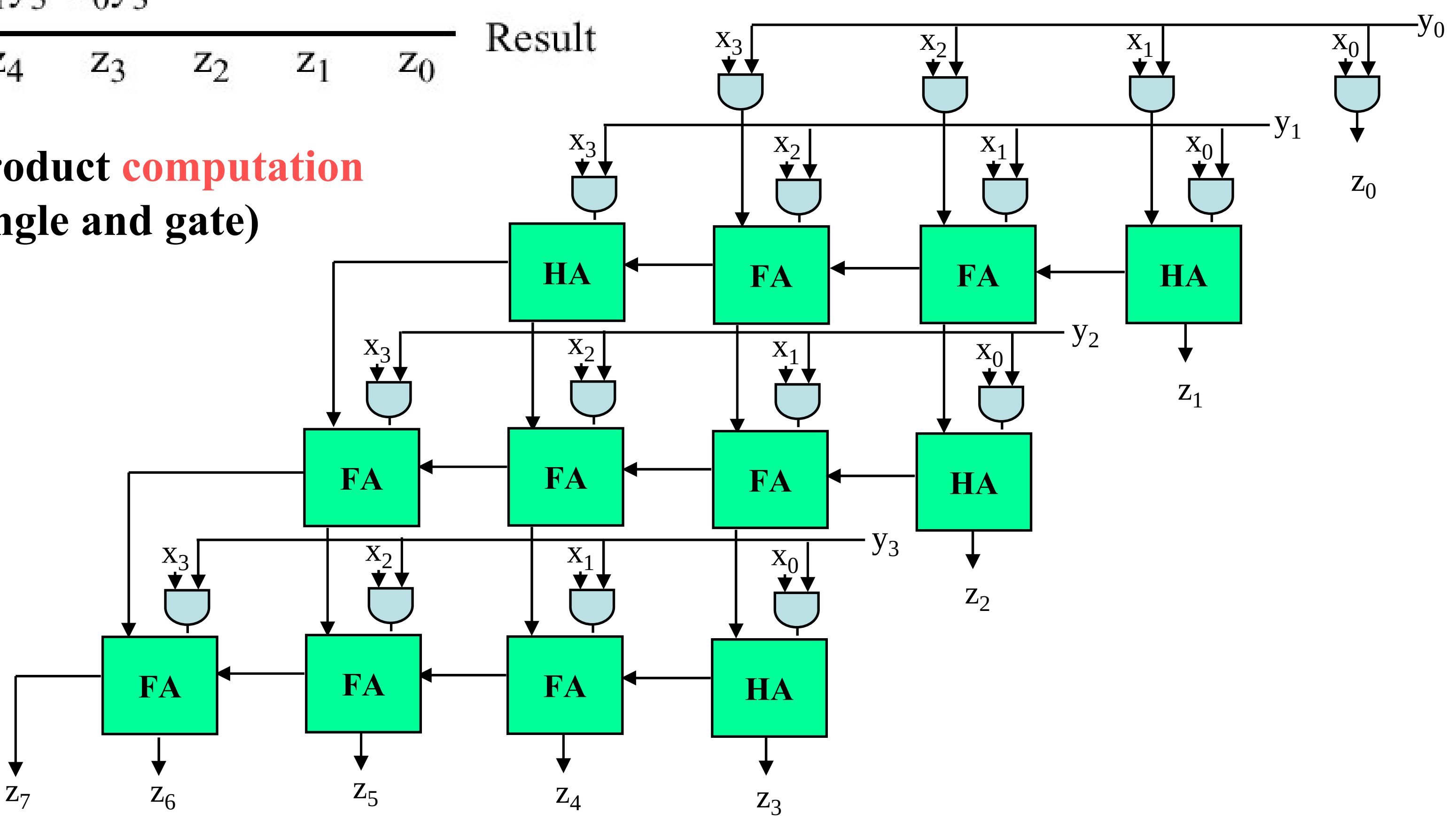


```
Init: P←0, load A and B

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

$$
\begin{array}{cccccccccl}
 & & & & & x_3 & x_2 & x_1 & x_0 & \text{Multiplicand} \\
\times & & & & & y_3 & y_2 & y_1 & y_0 & \text{Multiplier} \\
\cline{4-9}
 & & & & & x_3y_0 & x_2y_0 & x_1y_0 & x_0y_0 & \\
 & & & & x_3y_1 & x_2y_1 & x_1y_1 & x_0y_1 & & \text{Partial Product} \\
 & & & x_3y_2 & x_2y_2 & x_1y_2 & x_0y_2 & & & \\
+ & & x_3y_3 & x_2y_3 & x_1y_3 & x_0y_3 & & & & \\
\hline
 & z_7 & z_6 & z_5 & z_4 & z_3 & z_2 & z_1 & z_0 & \text{Result}
\end{array}
$$

➢ **Partial product computation is simple (single and gate)**



x_3 x_2 x_1 x_0 y_0
y_1 y_2 y_3
HA
FA
z_0 z_1 z_2 z_3 z_4 z_5 z_6 z_7

2's Complement Multiplication

Step 1: two's complement operands so high order bit is -2^{N-1}. Must sign extend partial products and subtract the last one

$$\begin{array}{ccccccccc}
 & & & & & X3 & X2 & X1 & X0 \\
 & & & & * & Y3 & Y2 & Y1 & Y0 \\
\hline
 & X3Y0 & X3Y0 & X3Y0 & X3Y0 & X3Y0 & X2Y0 & X1Y0 & X0Y0 \\
+ & X3Y1 & X3Y1 & X3Y1 & X3Y1 & X2Y1 & X1Y1 & X0Y1 & \\
+ & X3Y2 & X3Y2 & X3Y2 & X2Y2 & X1Y2 & X0Y2 & & \\
- & X3Y3 & X3Y3 & X2Y3 & X1Y3 & X0Y3 & & & \\
\hline
 & Z7 & Z6 & Z5 & Z4 & Z3 & Z2 & Z1 & Z0
\end{array}$$

Step 2: don't want all those extra additions, so add a carefully chosen constant, remembering to subtract it at the end. Convert subtraction in add of (complement + 1).

$$\begin{array}{ccccccccc}
 & X3Y0 & X3Y0 & X3Y0 & X3Y0 & X3Y0 & X2Y0 & X1Y0 & X0Y0 \\
+ & & & & & 1 & & & \\
+ & X3Y1 & X3Y1 & X3Y1 & X3Y1 & X2Y1 & X1Y1 & X0Y1 & \\
+ & & & & 1 & & & & \\
+ & X3Y2 & X3Y2 & X3Y2 & X2Y2 & X1Y2 & X0Y2 & & \\
+ & & & 1 & & & & & \\
+ & \overline{X3Y3} & \overline{X3Y3} & \overline{X2Y3} & \overline{X1Y3} & \overline{X0Y3} & & & \\
+ & & & & & 1 & & & \\
+ & & 1 & & & & & & \\
- & & 1 & 1 & 1 & 1 & & &
\end{array}$$

$\left.\begin{array}{c} \overline{X3Y3}\ \overline{X3Y3}\ \overline{X2Y3}\ \overline{X1Y3}\ \overline{X0Y3} \\ 1 \end{array}\right\} -B = \sim B + 1$

Step 3: add the ones to the partial products and propagate the carries. All the sign extension bits go away!

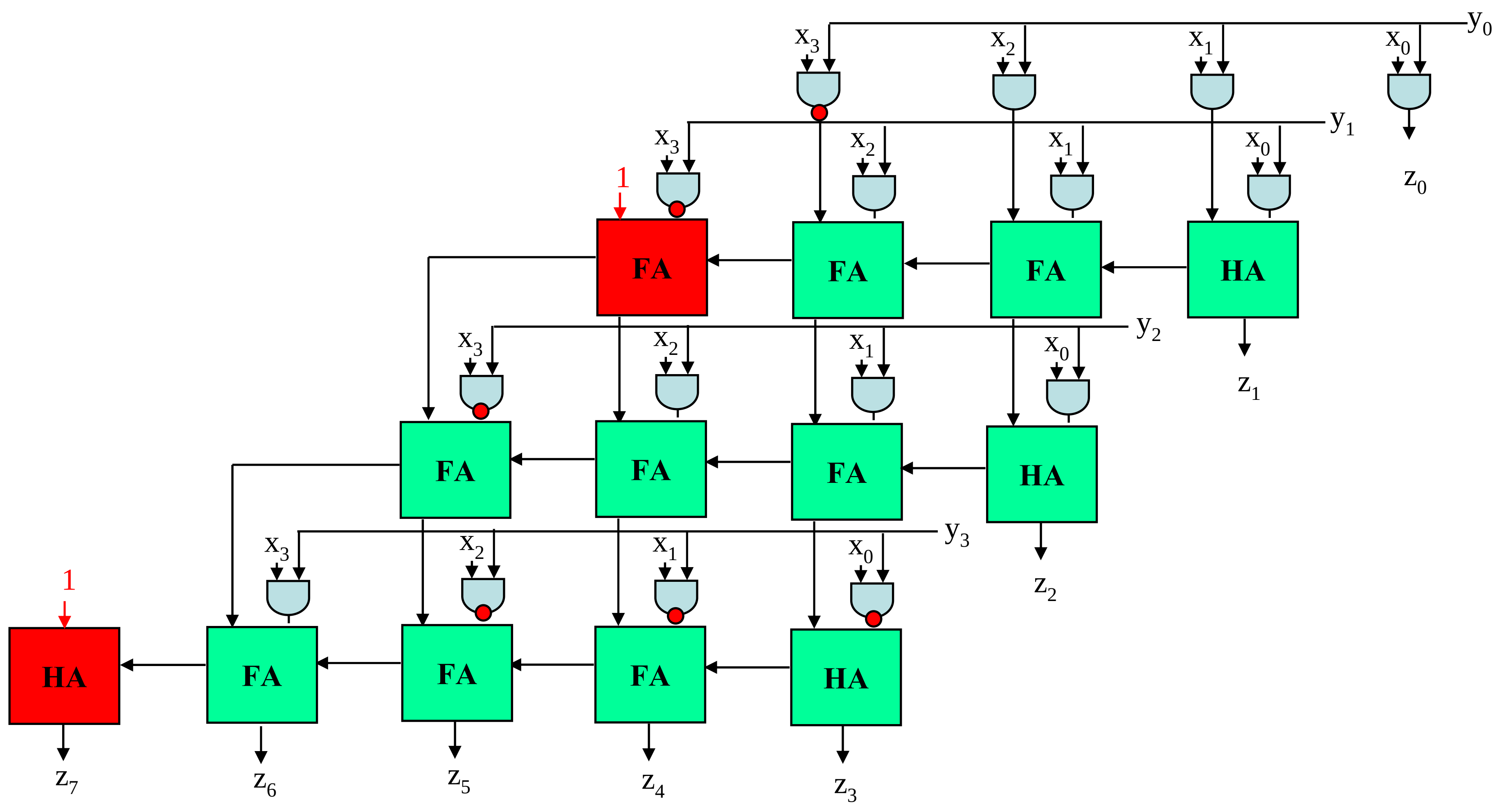
$$\begin{array}{cccccccc}
 & & & & \overline{X3Y0} & X2Y0 & X1Y0 & X0Y0 \\
+ & & & \overline{X3Y1} & X2Y1 & X1Y1 & X0Y1 & \\
+ & & \overline{X2Y2} & X1Y2 & X0Y2 & & & \\
+ & X3Y3 & \overline{X2Y3} & \overline{X1Y3} & \overline{X0Y3} & & & \\
+ & & & & 1 & & & \\
- & 1 & 1 & 1 & 1 & & &
\end{array}$$

Step 4: finish computing the constants…

$$\begin{array}{ccccccccc}
 & & & & & \overline{X3Y0} & X2Y0 & X1Y0 & X0Y0 \\
+ & & & & \overline{X3Y1} & X2Y1 & X1Y1 & X0Y1 & \\
+ & & & \overline{X2Y2} & X1Y2 & X0Y2 & & & \\
+ & & X3Y3 & \overline{X2Y3} & \overline{X1Y3} & \overline{X0Y3} & & & \\
+ & 1 & & & 1 & & & &
\end{array}$$

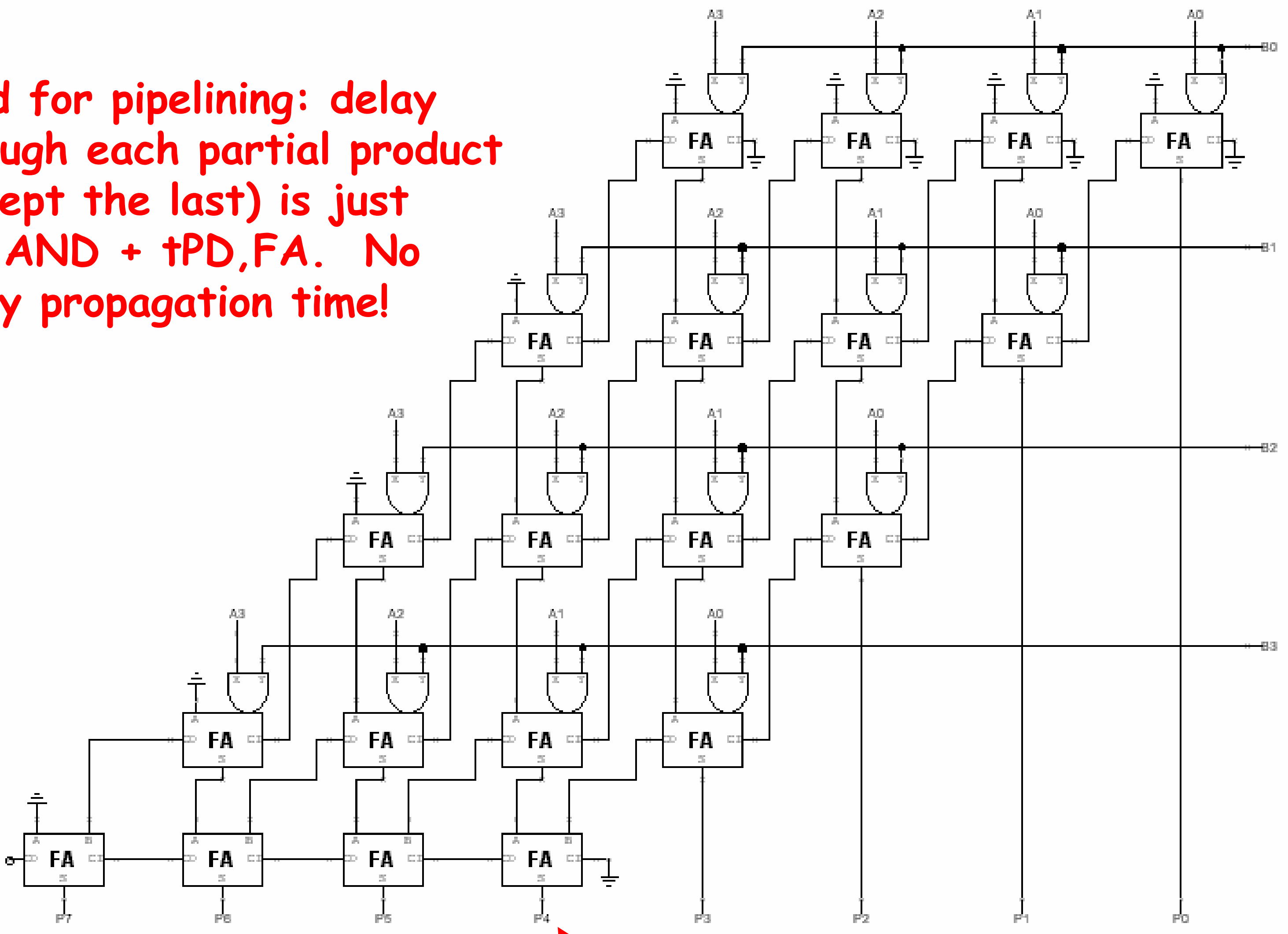
Result: multiplying 2's complement operands takes just about same amount of hardware as multiplying unsigned operands!

2's Complement Multiplication



Carry-Save Adder (CSA)

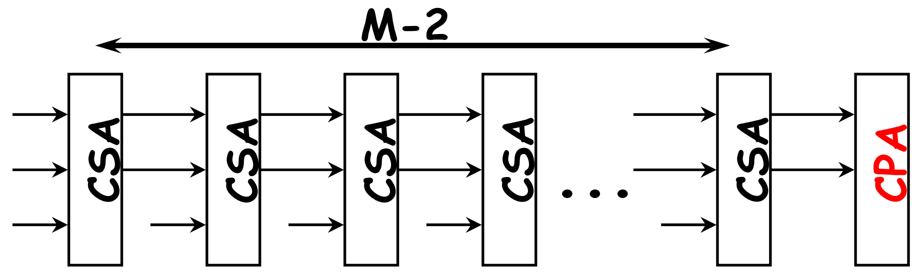
Good for pipelining: delay through each partial product (except the last) is just tPD,AND + tPD,FA. No carry propagation time!



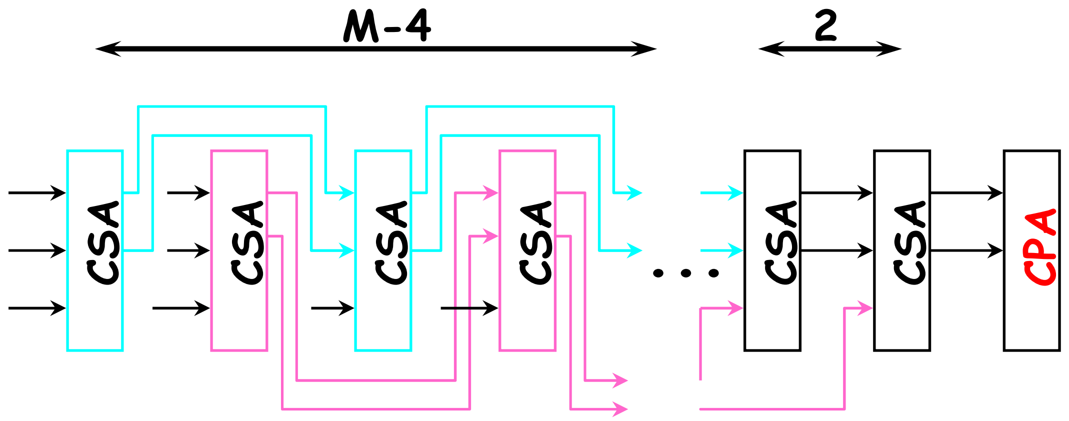
Last stage is still a carry-propagate adder (CPA)

Latency Improvements

Abstract partial product picture :

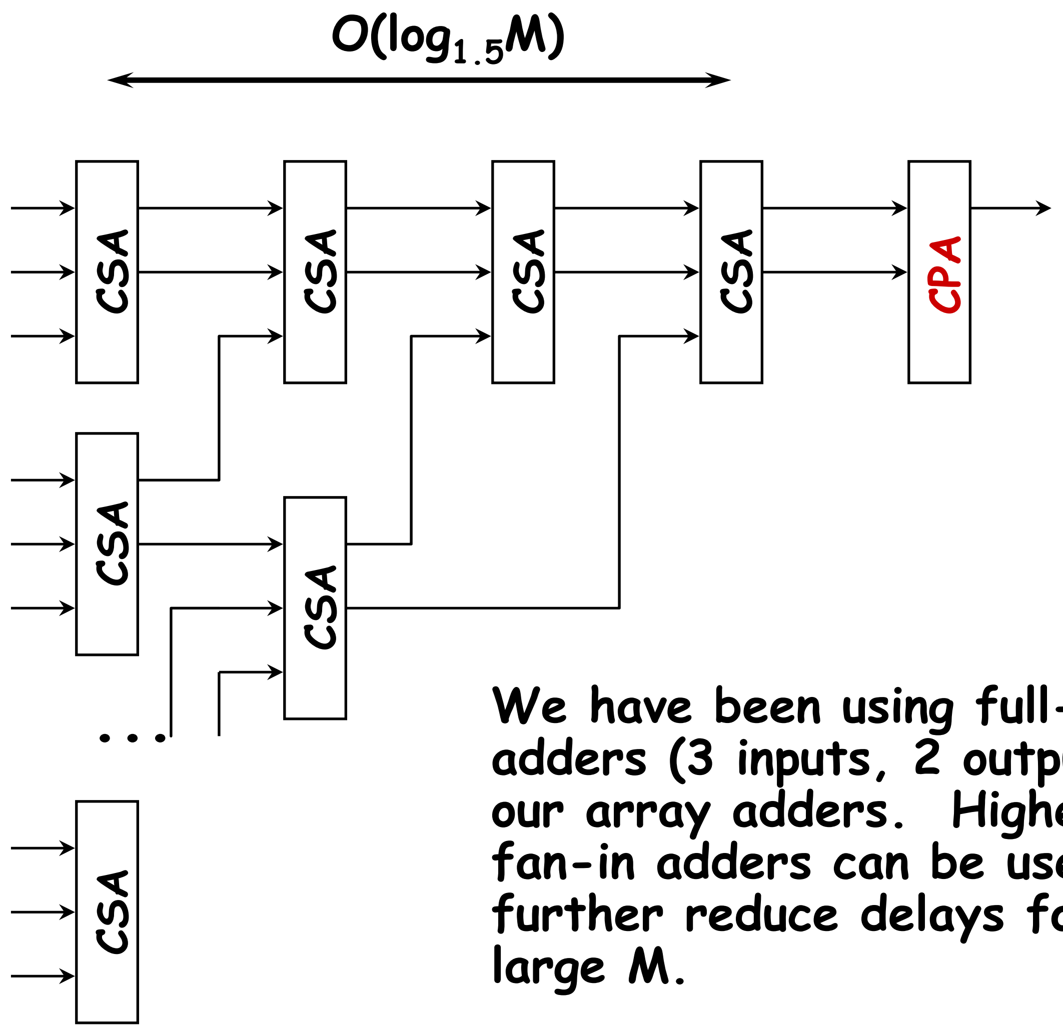


Rewire so that first two adders work in parallel. Feed results into third and fourth adders which also work in parallel, etc.



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

More Latency Improvements

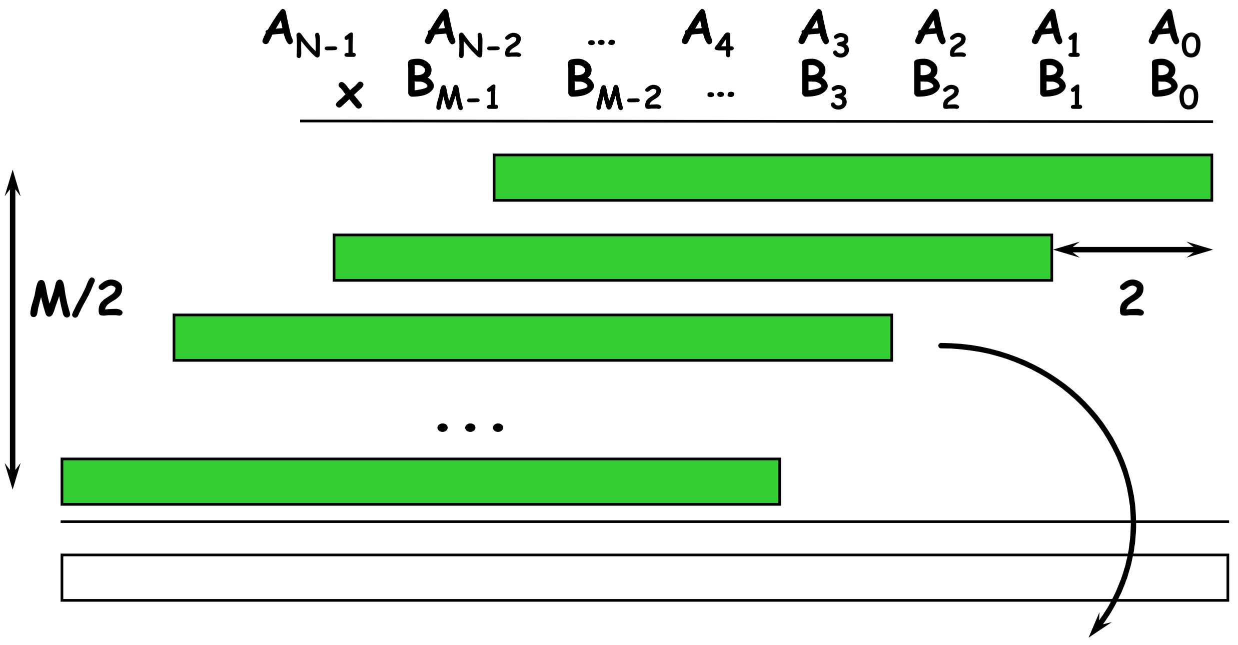


Wallace Tree

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.

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!



A_{N-1} A_{N-2} … A_4 A_3 A_2 A_1 A_0
× B_{M-1} B_{M-2} … B_3 B_2 B_1 B_0
M/2
2
. . .

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

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

Booth recoding

current bit pair

from previous bit pair

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

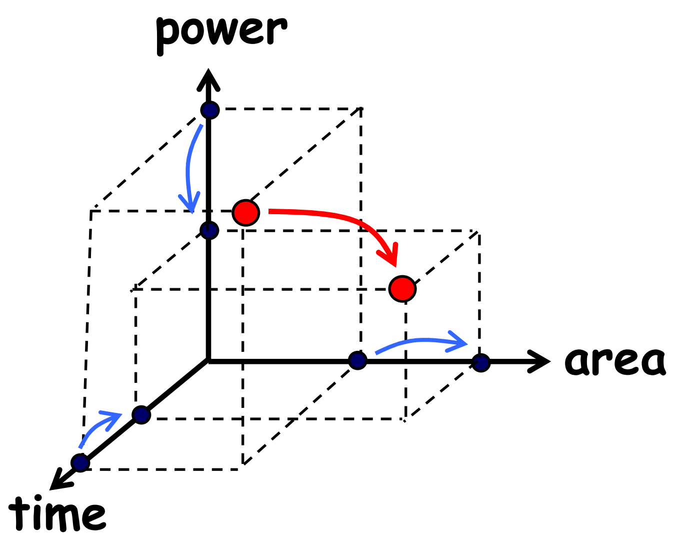
A "1" in this bit means the previous stage needed to add 4*A. Since this stage is shifted by 2 bits with respect to the previous stage, adding 4*A in the previous stage is like adding A in this stage!

Behavioral Transformations

- There are a large number of implementations of the same functionality
- These implementations present a different point in the area-time-power design space
- Behavioral transformations allow exploring the design space a high-level

Optimization metrics:

1. Area of the design
2. Throughput or sample time T_S
3. Latency: clock cycles between the input and associated output change
4. Power consumption
5. Energy of executing a task
6. …



Fixed-Coefficient Multiplication

Conventional Multiplication

$Z = X \cdot Y$

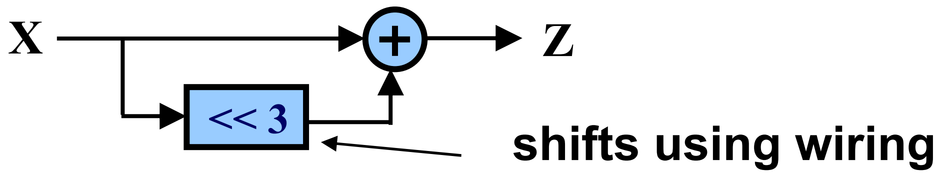
				X_3	X_2	X_1	X_0
				Y_3	Y_2	Y_1	Y_0
				$X_3 \cdot Y_0$	$X_2 \cdot Y_0$	$X_1 \cdot Y_0$	$X_0 \cdot Y_0$
			$X_3 \cdot Y_1$	$X_2 \cdot Y_1$	$X_1 \cdot Y_1$	$X_0 \cdot Y_1$	
		$X_3 \cdot Y_2$	$X_2 \cdot Y_2$	$X_1 \cdot Y_2$	$X_0 \cdot Y_2$		
	$X_3 \cdot Y_3$	$X_2 \cdot Y_3$	$X_1 \cdot Y_3$	$X_0 \cdot Y_3$			
Z_7	Z_6	Z_5	Z_4	Z_3	Z_2	Z_1	Z_0

Constant multiplication (become hardwired shifts and adds)

$Z = X \cdot (1001)_2$

				X_3	X_2	X_1	X_0
				1	0	0	1
				X_3	X_2	X_1	X_0
	X_3	X_2	X_1	X_0			
Z_7	Z_6	Z_5	Z_4	Z_3	Z_2	Z_1	Z_0

$Y = (1001)_2 = 2^3 + 2^0$



Transform: Canonical Signed Digits (CSD)

Canonical signed digit representation is used to increase the number of zeros. It uses digits {-1, 0, 1} instead of only {0, 1}.

Iterative encoding: replace string of consecutive 1's

0	1	1	…	1	1

$2^{N-2} + \ldots + 2^1 + 2^0$

➡

1	0	0	…	0	-1

$2^{N-1} - 2^0$

Worst case CSD has 50% non zero bits

01101111

=

$100\bar{1}000\bar{1}$

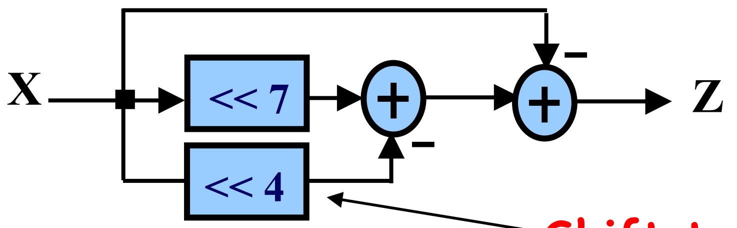
0	1	1	0	1	1	1	1

➡

0	1	1	1	0	0	0	-1

⬇

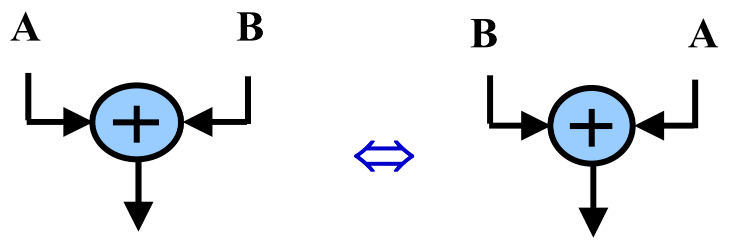
1	0	0	-1	0	0	0	-1



Shift translates to re-wiring

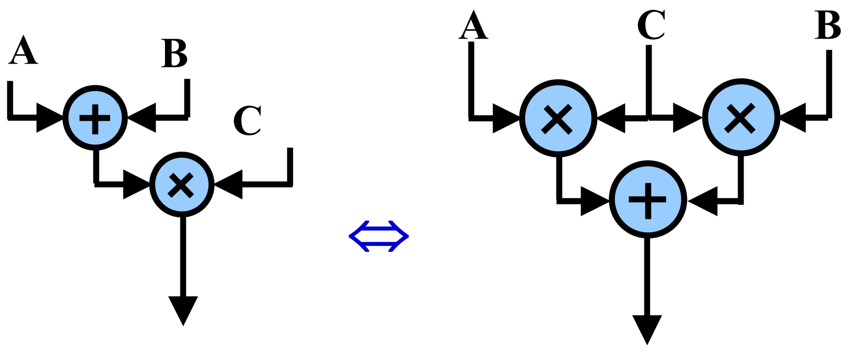
Algebraic Transformations

Commutativity



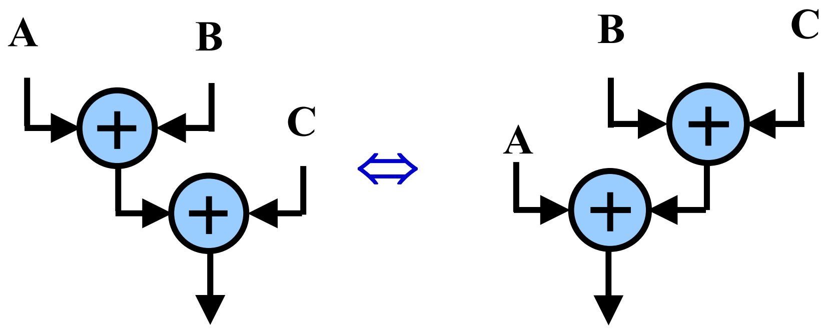
A + B = B + A

Distributivity



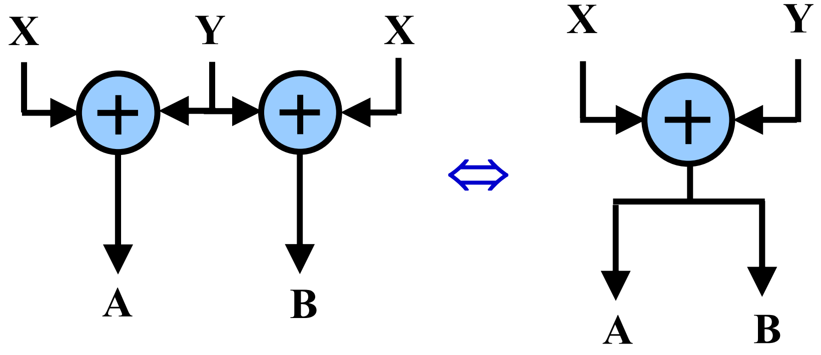
(A + B) C = AB + BC

Associativity

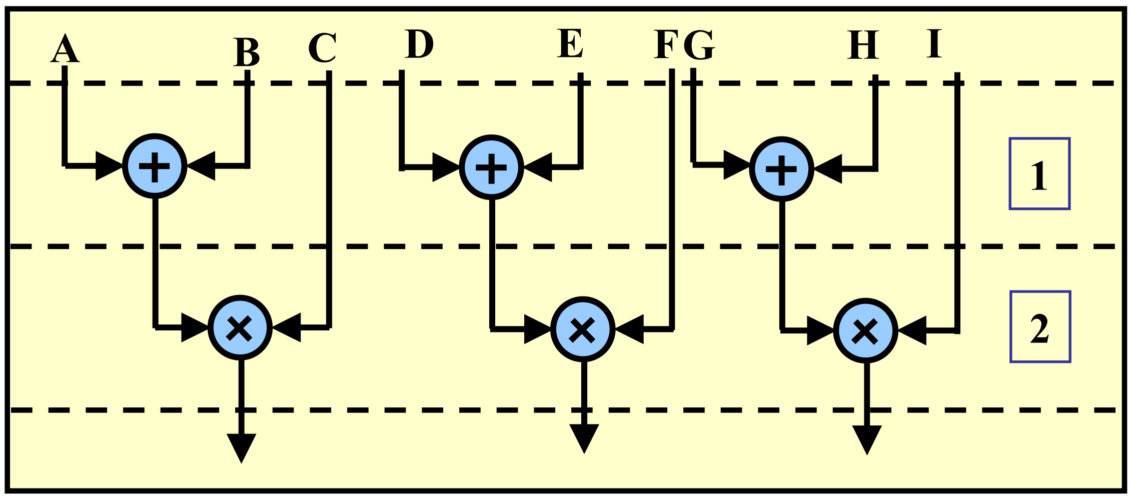


(A + B) + C = A + (B+C)

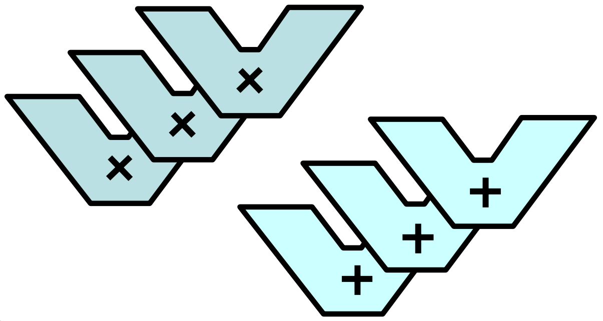
Common sub-expressions



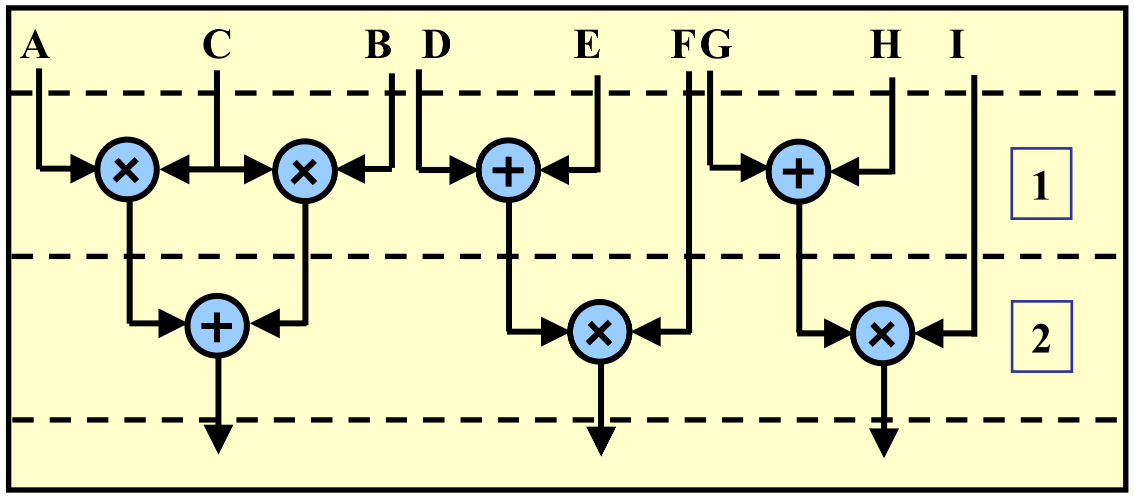
Transforms for Efficient Resource Utilization



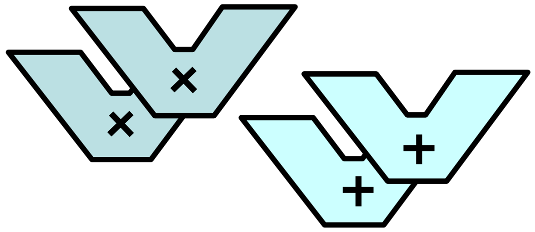
Time multiplexing: mapped to 3 multipliers and 3 adders



distributivity



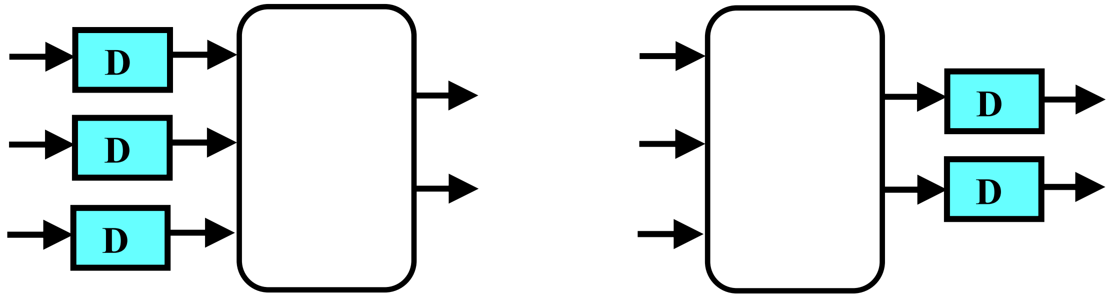
Reduce number of operators to 2 multipliers and 2 adders



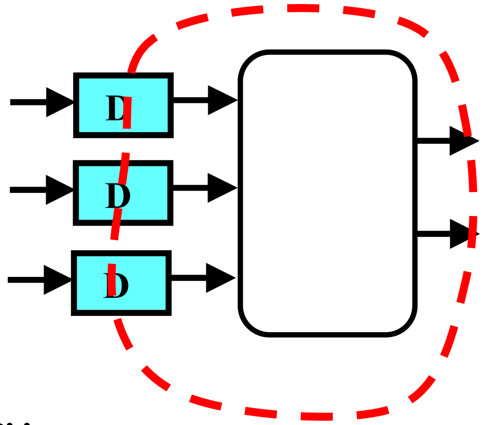
Retiming: A very useful transform

Retiming is the action of moving delay around in the systems

- Delays have to be moved from ALL inputs to ALL outputs or vice versa



Cutset retiming: A cutset intersects the edges, such that this would result in two disjoint partitions of these edges being cut. To retime, delays are moved from the ingoing to the outgoing edges or vice versa.

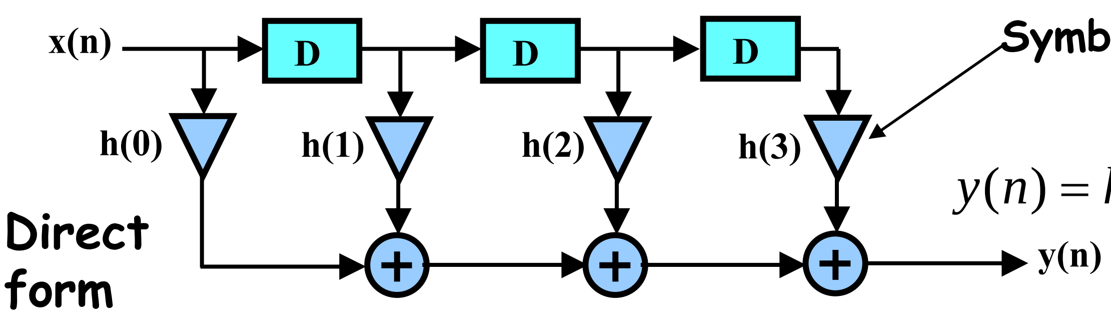


Retiming Synchronous Circuitry
Charles E. Leiserson and James B. Saxe
August 20, 1986.

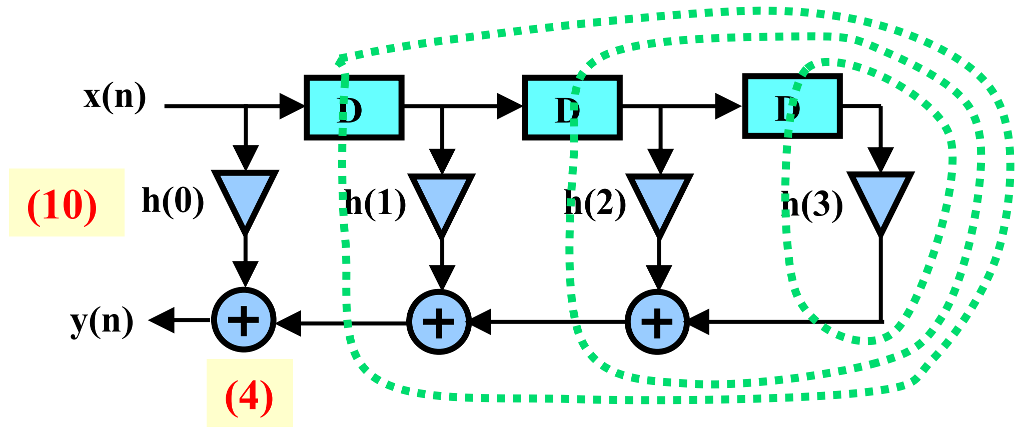
Benefits of retiming:

- Modify critical path delay
- Reduce total number of registers

Retiming Example: FIR Filter



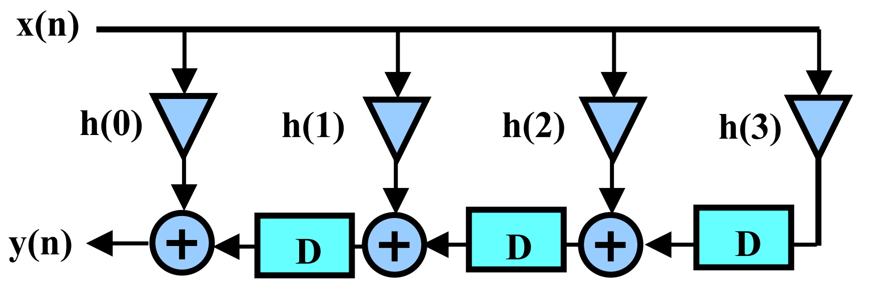
$$y(n) = h(n) \otimes x(n) = \sum_{i=0}^{K} x(n-i) \cdot h(i)$$



associativity of addition

T_{clk} = 22 ns

retime

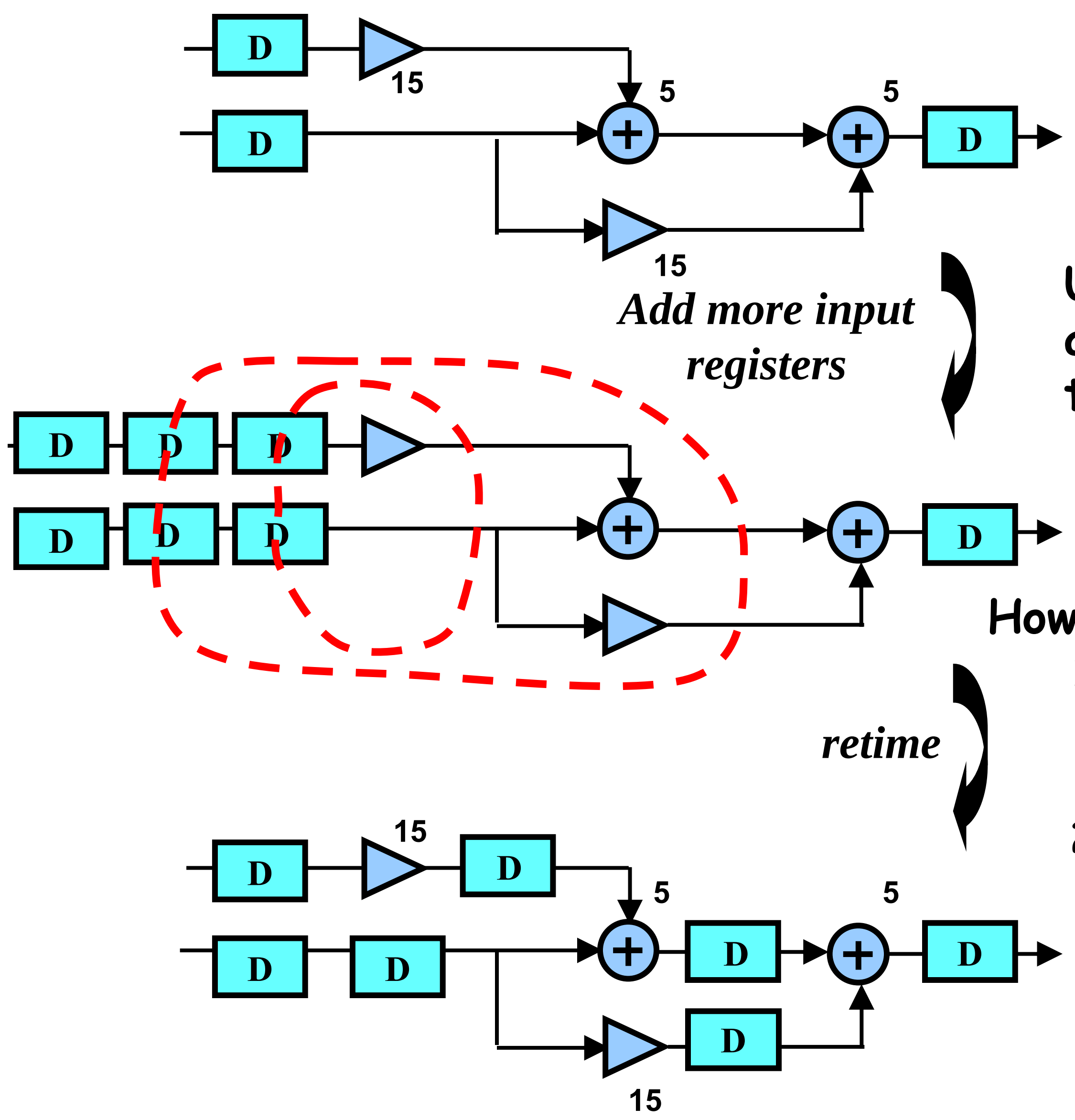


Transposed form

T_{clk} = 14 ns

Note: here we use a first cut analysis that assumes the delay of a chain of operators is the sum of their individual delays. This is not accurate.

Pipelining = Adding Registers + Retiming



T_{CLK} = 25 (w/ ideal regs)
Latency = 1 clock cycle
Throughput = 1/clock cycle

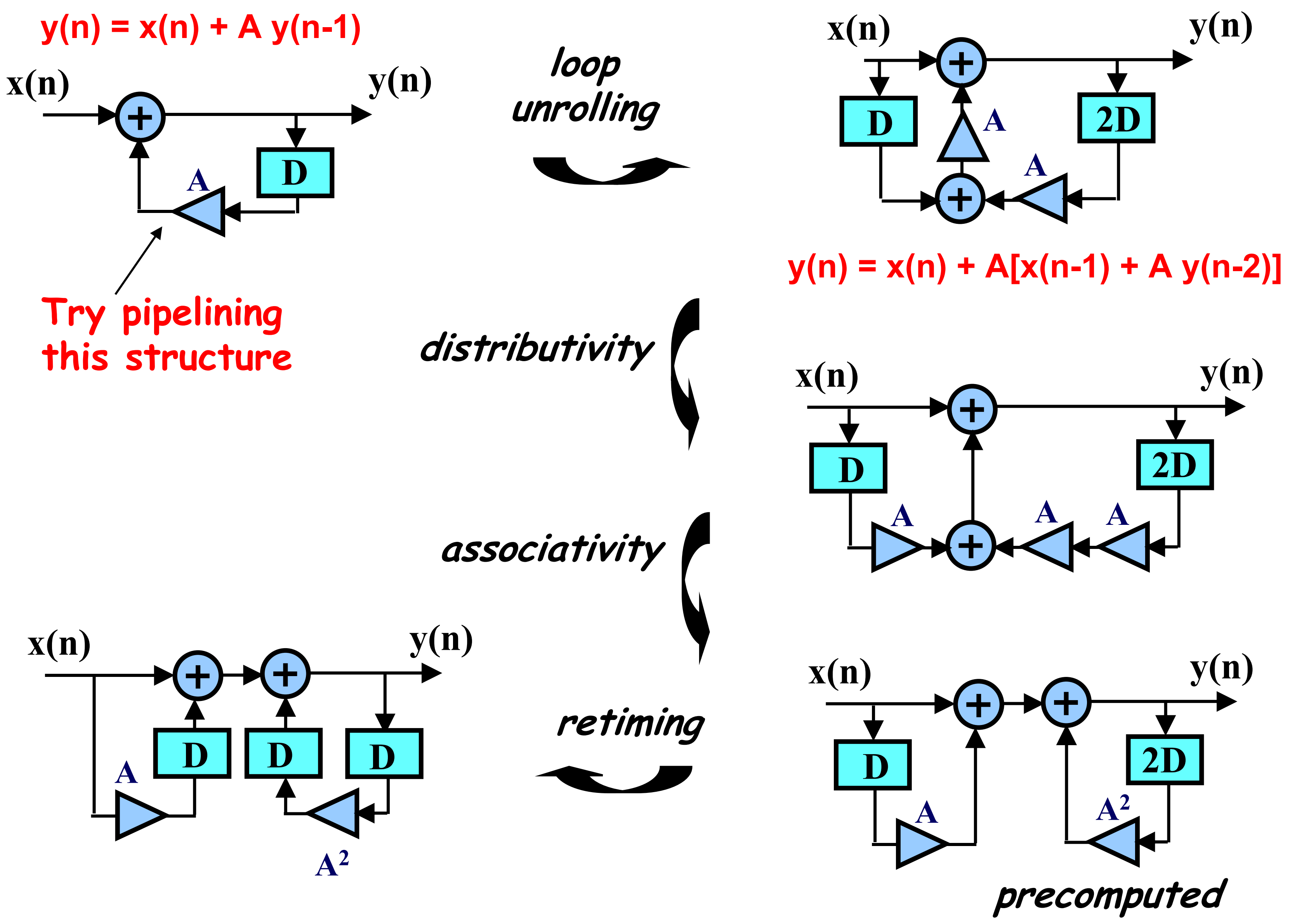
Unlike retiming, pipelining adds extra registers to the system

How to pipeline:

1. Add extra registers at *all* inputs (or, equivalently, *all* outputs)
2. Retime

T_{CLK} = 15 (w/ ideal regs)
Latency = 3 clock cycles
Throughput = 1/clock cycle

The Power of Transforms: Lookahead



y(n) = x(n) + A y(n-1)
x(n)
y(n)
D
A
Try pipelining this structure
loop unrolling
x(n)
y(n)
D
A
2D
A
y(n) = x(n) + A[x(n-1) + A y(n-2)]
distributivity
x(n)
y(n)
D
2D
A
A
A
associativity
x(n)
y(n)
D
2D
A
A²
precomputed
retiming
x(n)
y(n)
D
D
D
A
A²