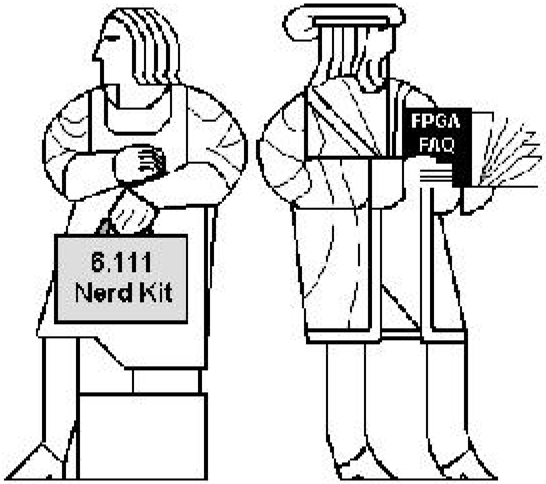
6.111 Lecture 13

Today: Arithmetic: Multiplication

1. Simple multiplication
2. Twos complement mult.
3. Speed: CSA & Pipelining
4. Booth recoding
5. Behavioral transformations:
   Fixed-coef. mult., Canonical Signed Digits, Retiming



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

1. Simple Multiplication

Unsigned Multiplication

$$
\begin{array}{rccccccc}
 & & & & A_3 & A_2 & A_1 & A_0 \\
 & & & \times & B_3 & B_2 & B_1 & B_0 \\
\hline
 & & & & A_3B_0 & A_2B_0 & A_1B_0 & A_0B_0 \\
 & & & A_3B_1 & A_2B_1 & A_1B_1 & A_0B_1 & \\
 & & A_3B_2 & A_2B_2 & A_1B_2 & A_0B_2 & & \\
+ & A_3B_3 & A_2B_3 & A_1B_3 & A_0B_3 & & & \\
\hline
\end{array}
$$

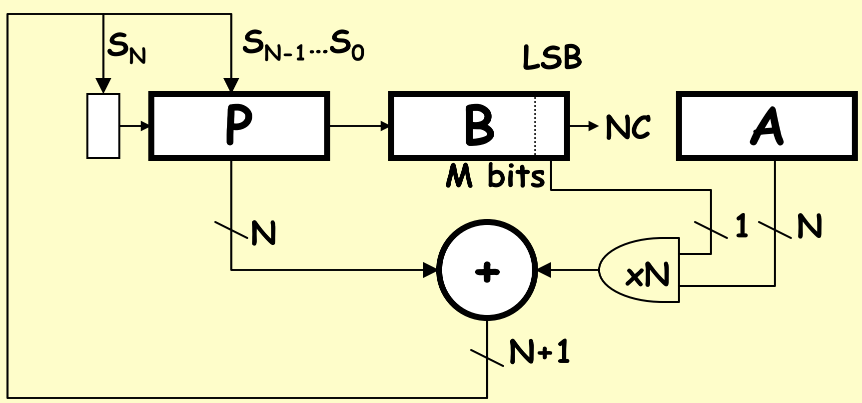
AB_i called a "partial product" →

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:

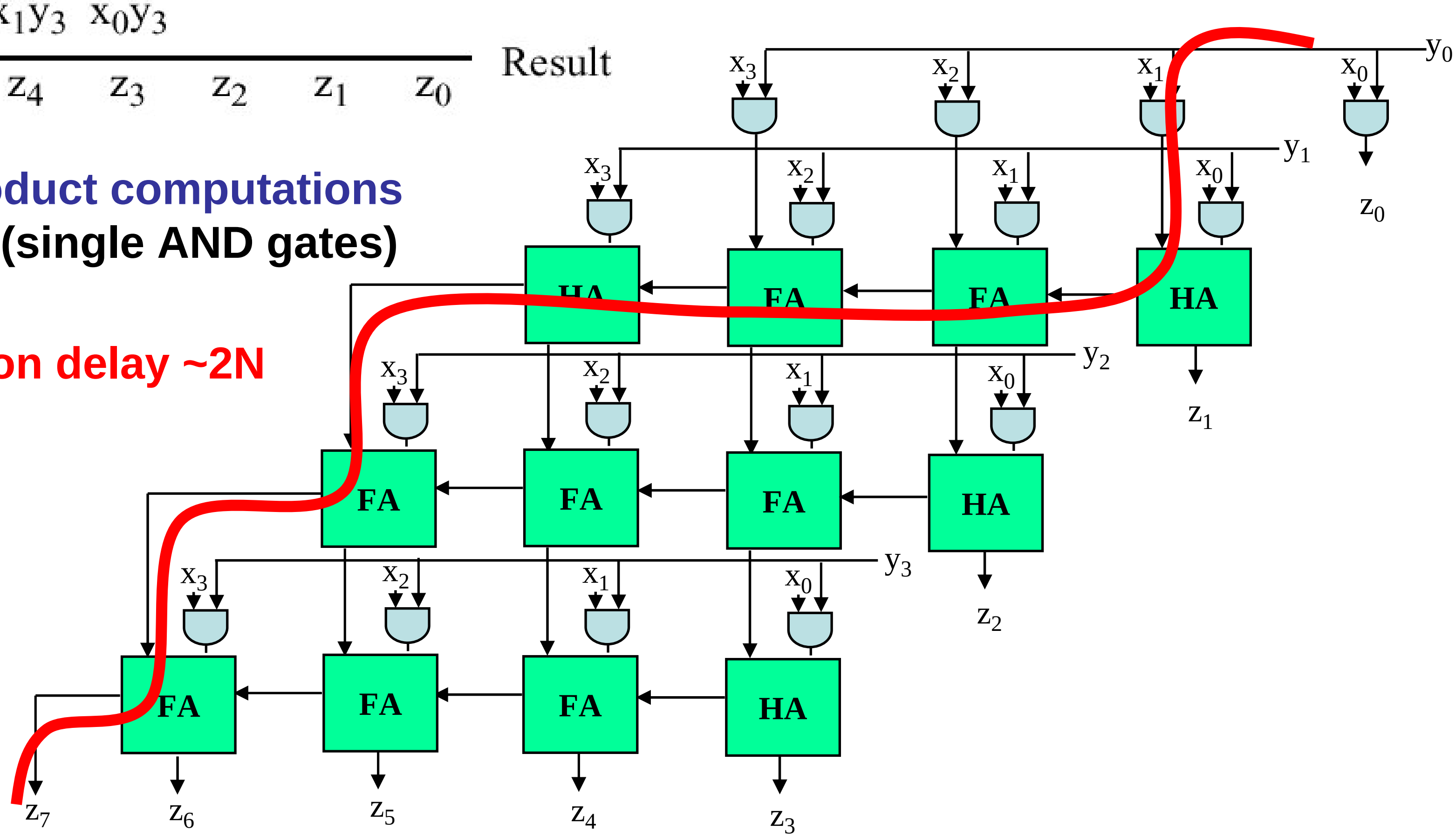



```
Init: P←0, load A and B

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

				x_3	x_2	x_1	x_0	Multiplicand
$\times$				y_3	y_2	y_1	y_0	Multiplier
				x_3y_0	x_2y_0	x_1y_0	x_0y_0	
			x_3y_1	x_2y_1	x_1y_1	x_0y_1		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				
z_7	z_6	z_5	z_4	z_3	z_2	z_1	z_0	Result

- **Partial product computations** are simple (single AND gates)
- **Propagation delay ~2N**



2's Complement Multiplication
(Baugh-Wooley)

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

```
                            X3   X2   X1   X0
                         *  Y3   Y2   Y1   Y0
                         --------------------
     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
   -----------------------------------------
       Z7   Z6   Z5   Z4   Z3   Z2   Z1   Z0
```

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

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

–B = ~B + 1

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

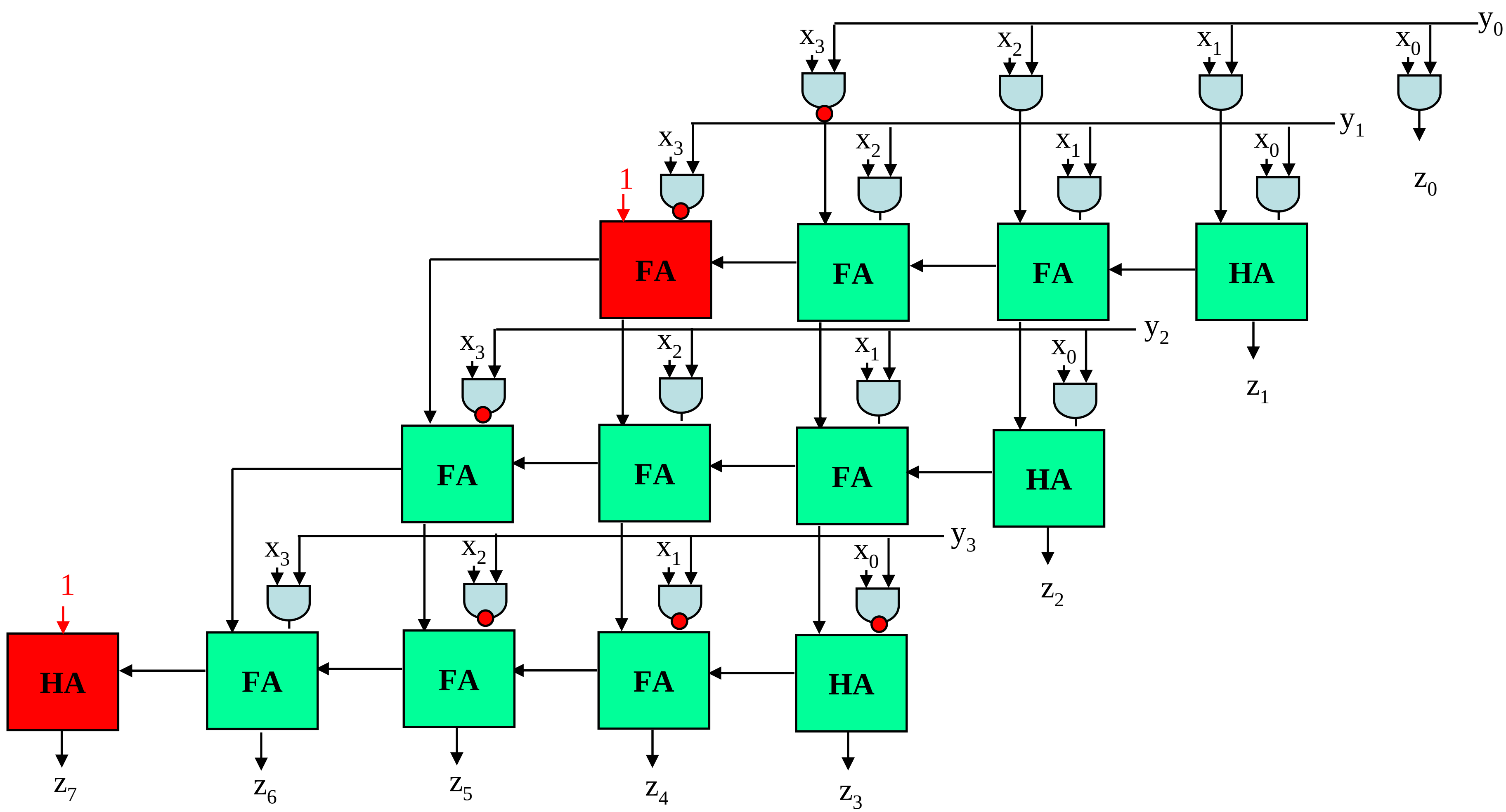
$$
\begin{array}{ccccccccc}
 & & & & & \overline{\texttt{X3Y0}} & \texttt{X2Y0} & \texttt{X1Y0} & \texttt{X0Y0} \\
+ & & & & \overline{\texttt{X3Y1}} & \texttt{X2Y1} & \texttt{X1Y1} & \texttt{X0Y1} & \\
+ & & & \overline{\texttt{X2Y2}} & \texttt{X1Y2} & \texttt{X0Y2} & & & \\
+ & & \texttt{X3Y3} & \overline{\texttt{X2Y3}}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

Step 4: finish computing the constants…

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

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

2's Complement Multiplication



Multiplication in Verilog

You can use the "*" operator to multiply two numbers:

```
wire [9:0] a,b;
wire [19:0] result = a*b;   // unsigned multiplication!
```

If you want Verilog to treat your operands as signed two's complement numbers, add the keyword `signed` to your `wire` or `reg` declaration:

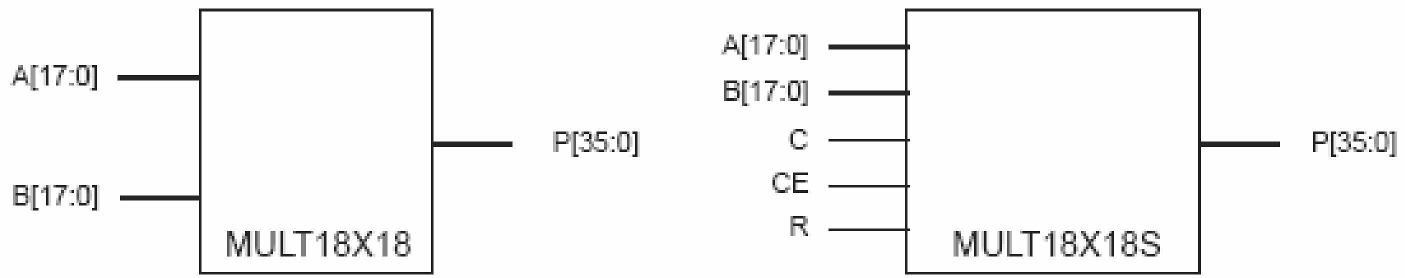
```
wire signed [9:0] a,b;
wire signed [19:0] result = a*b;  // signed multiplication!
```

Remember: unlike addition and subtraction, you need different circuitry if your multiplication operands are signed vs. unsigned. Same is true of the >>> (arithmetic right shift) operator. *To get signed operations all operands must be signed.*

To make a signed constant: 10'sh37C

Multipliers in the Virtex II

The Virtex FGPA has hardware multiplier circuits:



Combinatorial and Registered Multiplier Primitives

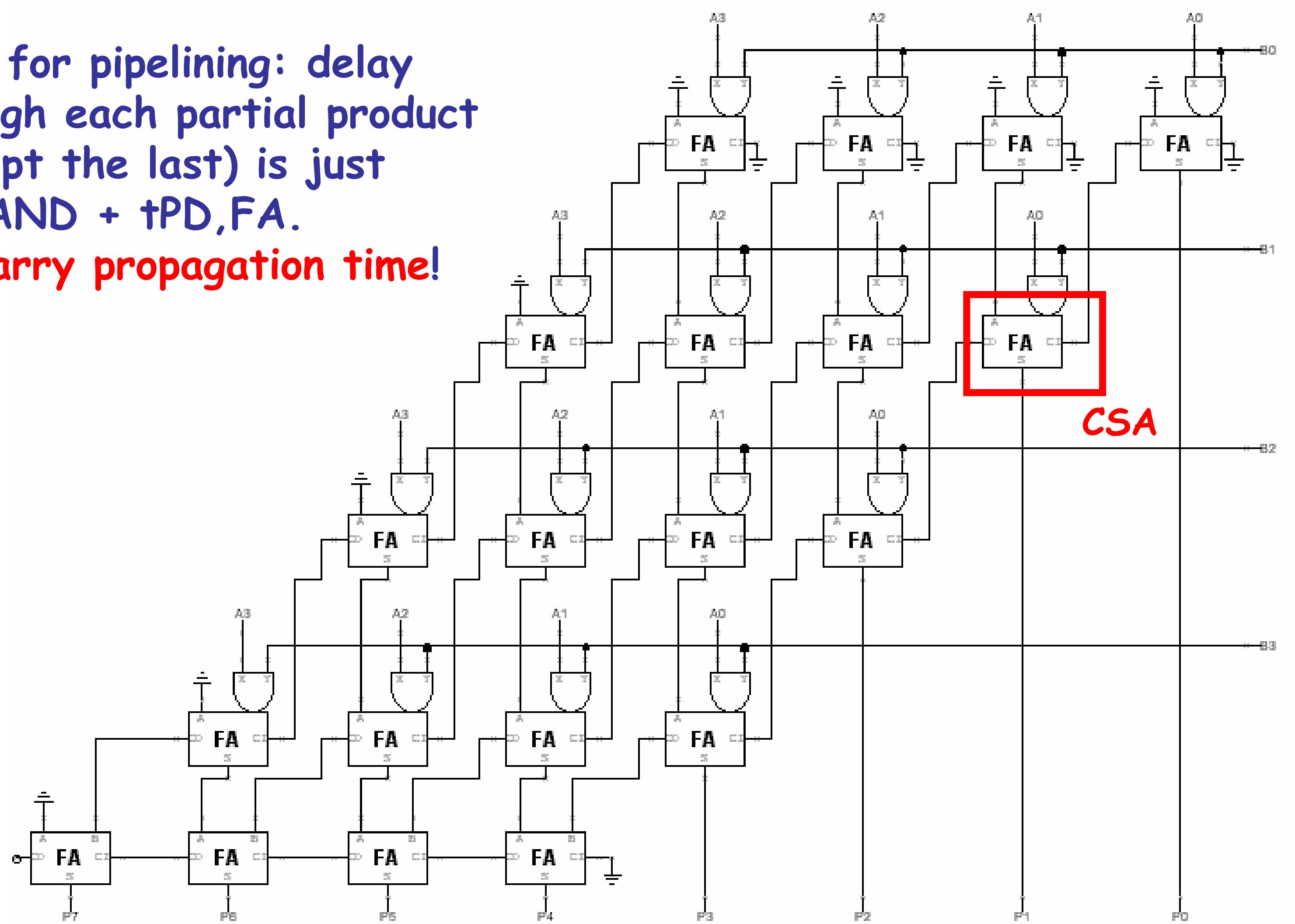
Note that the operands are signed 18-bit numbers.

The ISE tools will often use these hardware multipliers when you use the "*" operator in Verilog. Or can you instantiate them directly yourself:

```
wire signed [17:0] a,b;
wire signed [35:0] result;

MULT18X18 mymult(.A(a),.B(b),.P(result));
```

3. Faster Multipliers: Carry-Save Adder

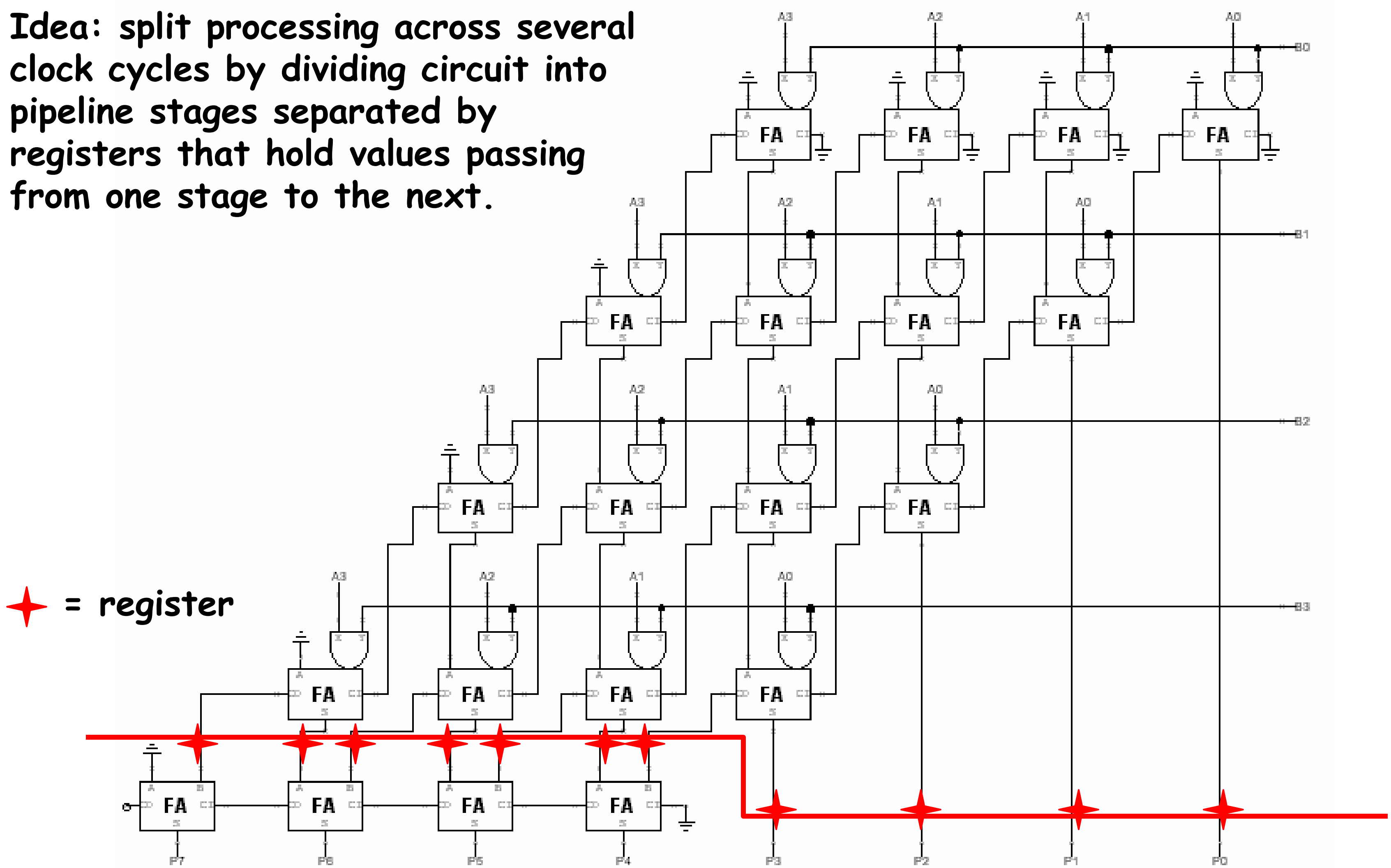
Good for pipelining: delay through each partial product (except the last) is just tPD,AND + tPD,FA.
No carry propagation time!



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

Increasing Throughput: Pipelining

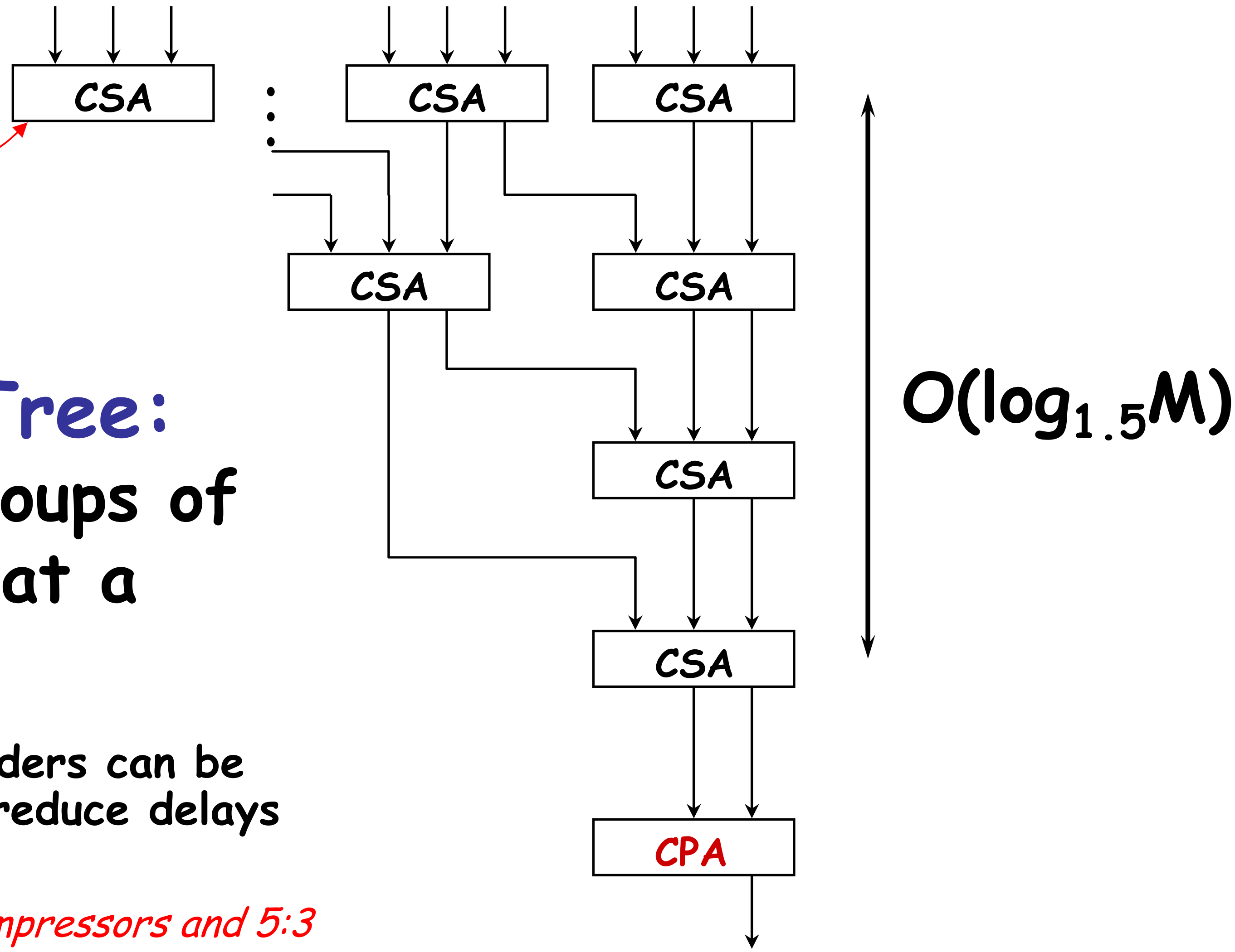
Idea: split processing across several clock cycles by dividing circuit into pipeline stages separated by registers that hold values passing from one stage to the next.



Throughput = 1 result per clock cycle (period is now $4 * t_{PD,FA}$ instead of $8 * t_{PD,FA}$)

Wallace Tree Multiplier

This is called a 3:2 counter by multiplier hackers: counts number of 1's on the 3 inputs, outputs 2-bit result.



Wallace Tree:

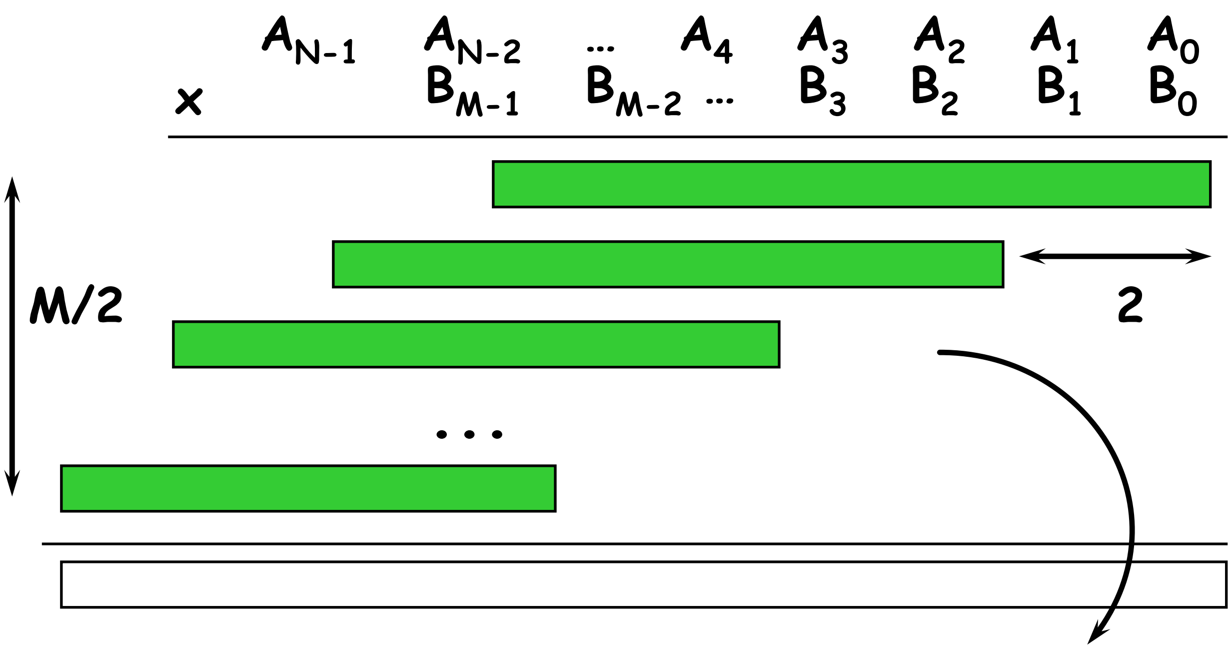
Combine groups of three bits at a time

Higher fan-in adders can be used to further reduce delays for large M.

4:2 compressors and 5:3 counters are popular building blocks.

4. Booth Recoding: Higher-radix mult.

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!

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

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

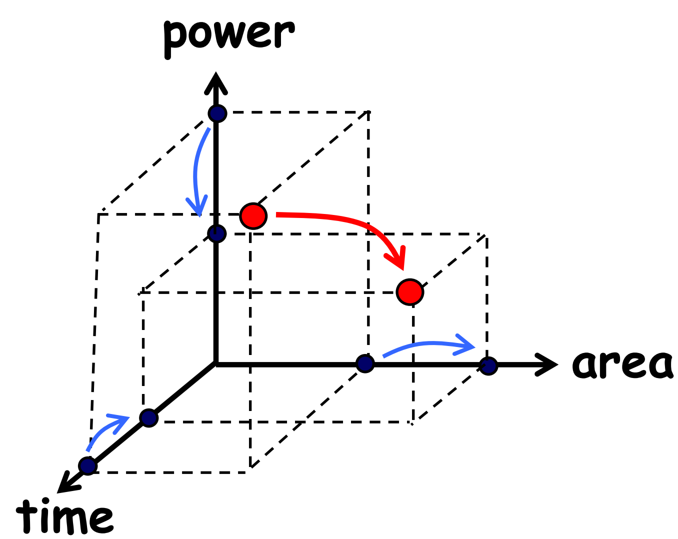
A "1" in this bit means the previous stage needed to add 4*A. Since this stage is shifted by 2 bits with respect to the previous stage, adding 4*A in the previous stage is like adding A in this stage!

5.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

$Z = X \cdot Y$

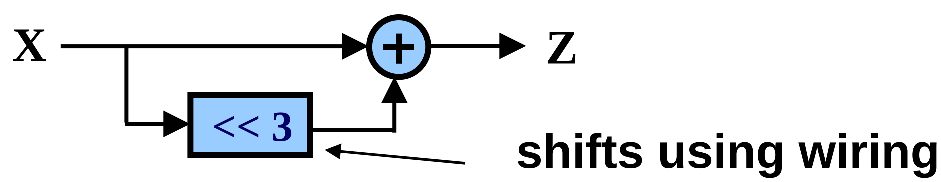
$$
\begin{array}{cccccccc}
 & & & & X_3 & X_2 & X_1 & X_0 \\
 & & & & Y_3 & Y_2 & Y_1 & Y_0 \\
\cline{5-8}
 & & & & X_3 \cdot Y_0 & X_2 \cdot Y_0 & X_1 \cdot Y_0 & X_0 \cdot Y_0 \\
 & & & X_3 \cdot Y_1 & X_2 \cdot Y_1 & X_1 \cdot Y_1 & X_0 \cdot Y_1 & \\
 & & X_3 \cdot Y_2 & X_2 \cdot Y_2 & X_1 \cdot Y_2 & X_0 \cdot Y_2 & & \\
 & X_3 \cdot Y_3 & X_2 \cdot Y_3 & X_1 \cdot Y_3 & X_0 \cdot Y_3 & & & \\
\hline
Z_7 & Z_6 & Z_5 & Z_4 & Z_3 & Z_2 & Z_1 & Z_0
\end{array}
$$

Constant multiplication (become hardwired shifts and adds)

$Z = X \cdot (1001)_2$

$$
\begin{array}{cccccccc}
 & & & & X_3 & X_2 & X_1 & X_0 \\
 & & & & 1 & 0 & 0 & 1 \\
\cline{5-8}
 & & & & X_3 & X_2 & X_1 & X_0 \\
 & X_3 & X_2 & X_1 & X_0 & & & \\
\hline
Z_7 & Z_6 & Z_5 & Z_4 & Z_3 & Z_2 & Z_1 & Z_0
\end{array}
$$

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



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

(replace 1 with 2-1)

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

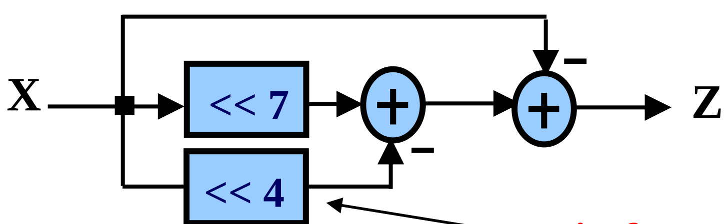
0	1	1	0	1	1	1	1

➡

0	1	1	1	0	0	0	-1

⬇

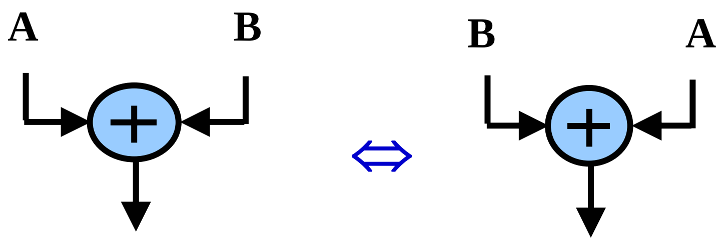
1	0	0	-1	0	0	0	-1



Shift translates to re-wiring

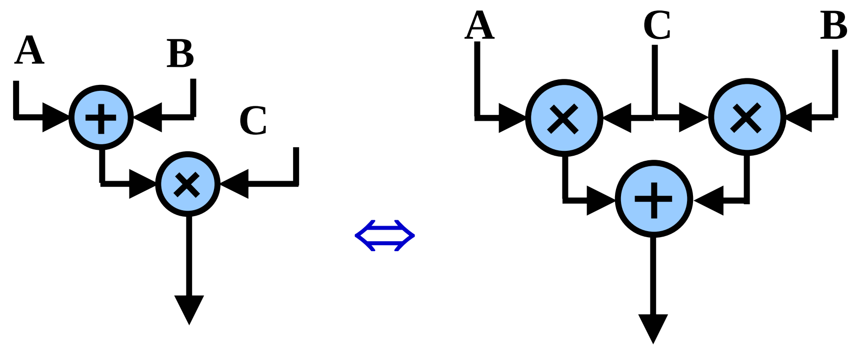
Algebraic Transformations

Commutativity



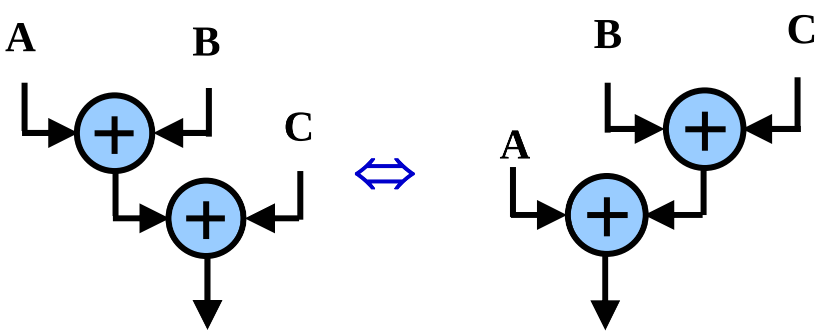
$A + B = B + A$

Distributivity



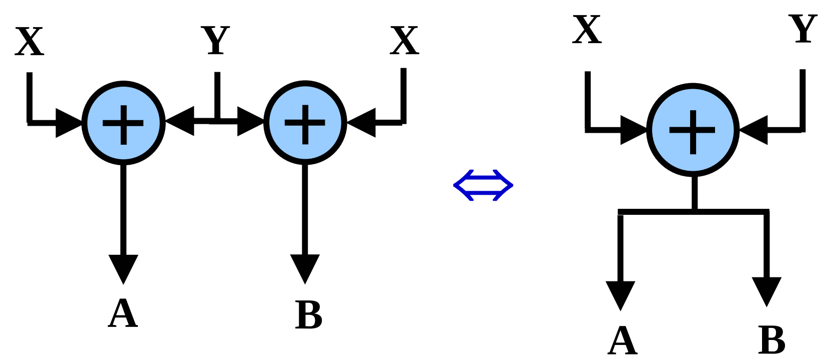
$(A + B)\ C = AB + BC$

Associativity

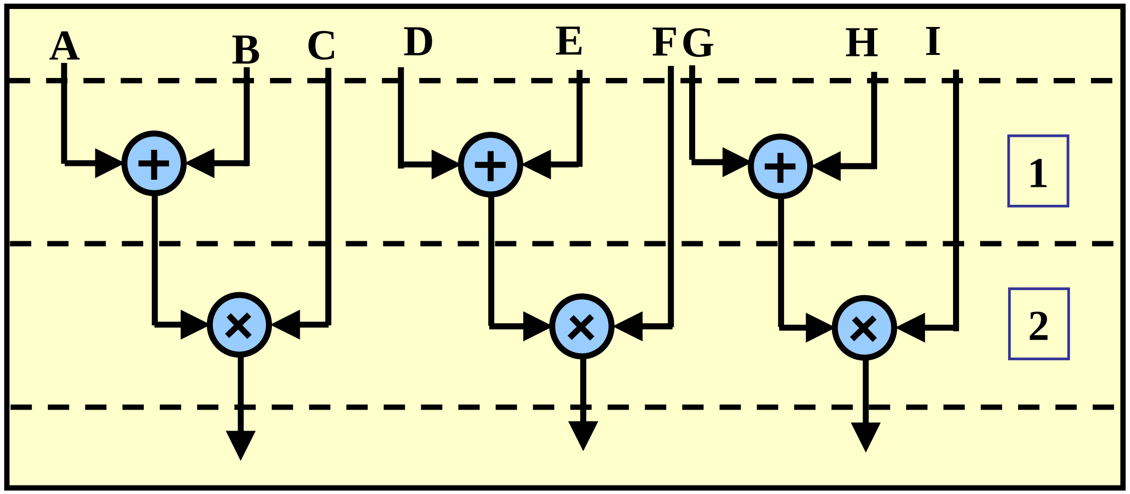


$(A + B) + C = A + (B+C)$

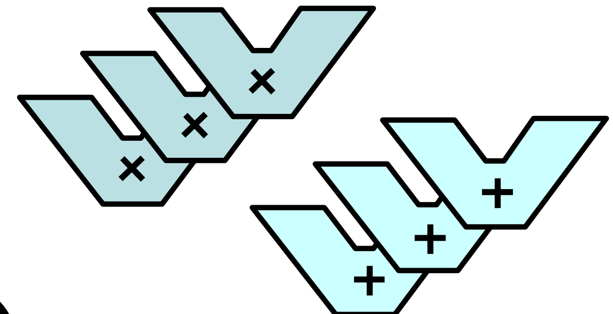
Common sub-expressions



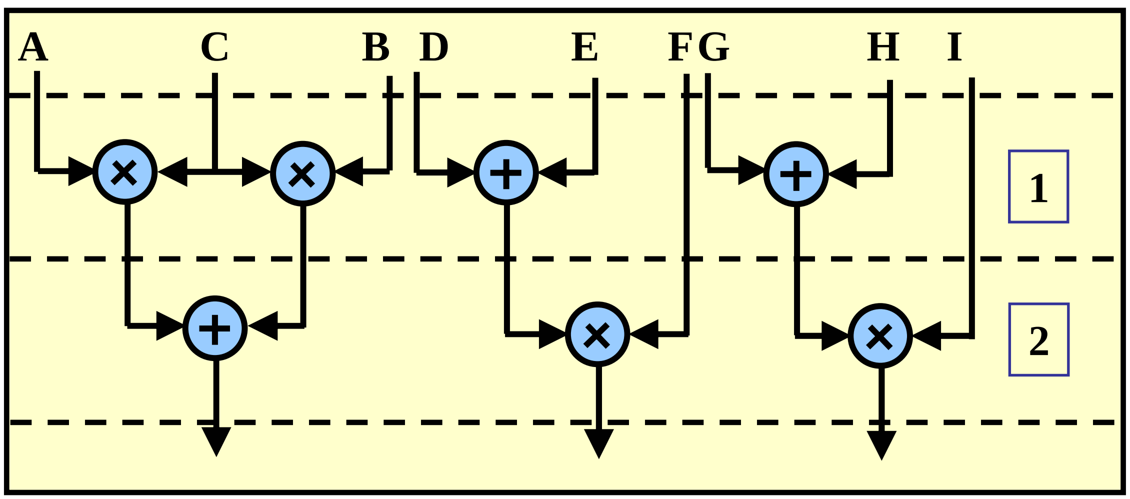
Transforms for Efficient Resource Utilization



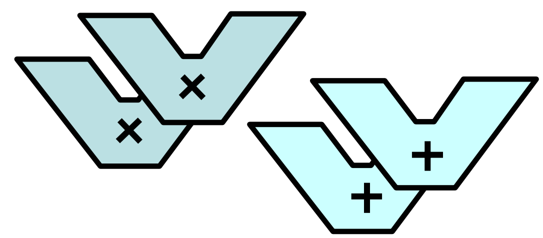
Time multiplexing: mapped to 3 multipliers and 3 adders



distributivity



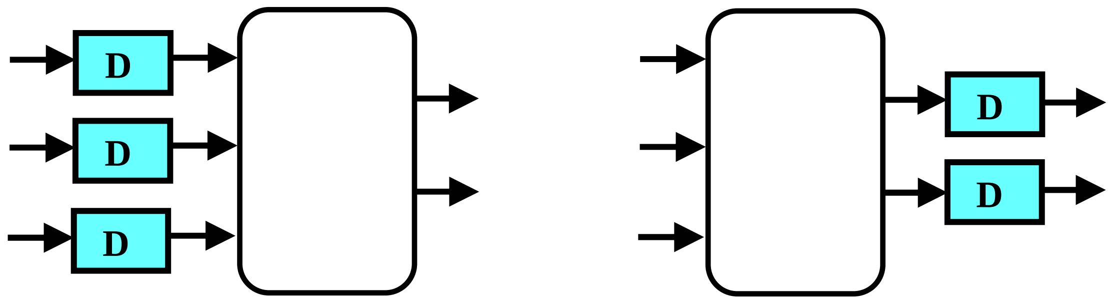
Reduce number of operators to 2 multipliers and 2 adders



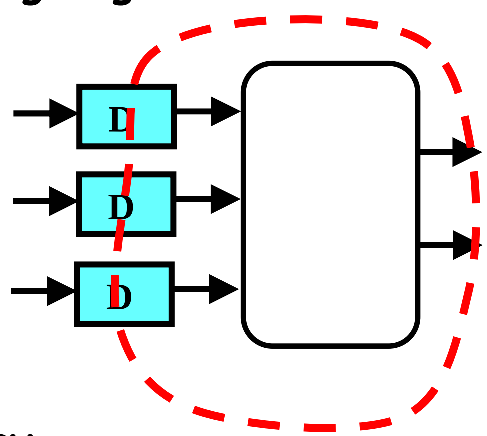
Retiming: A very useful transform

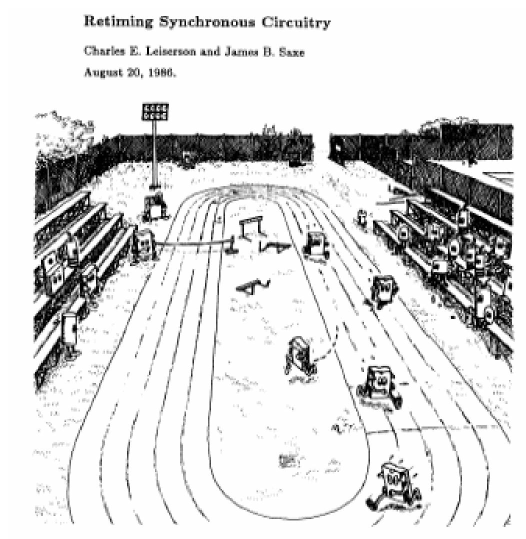
Retiming is the action of moving delay around in the systems

- 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.

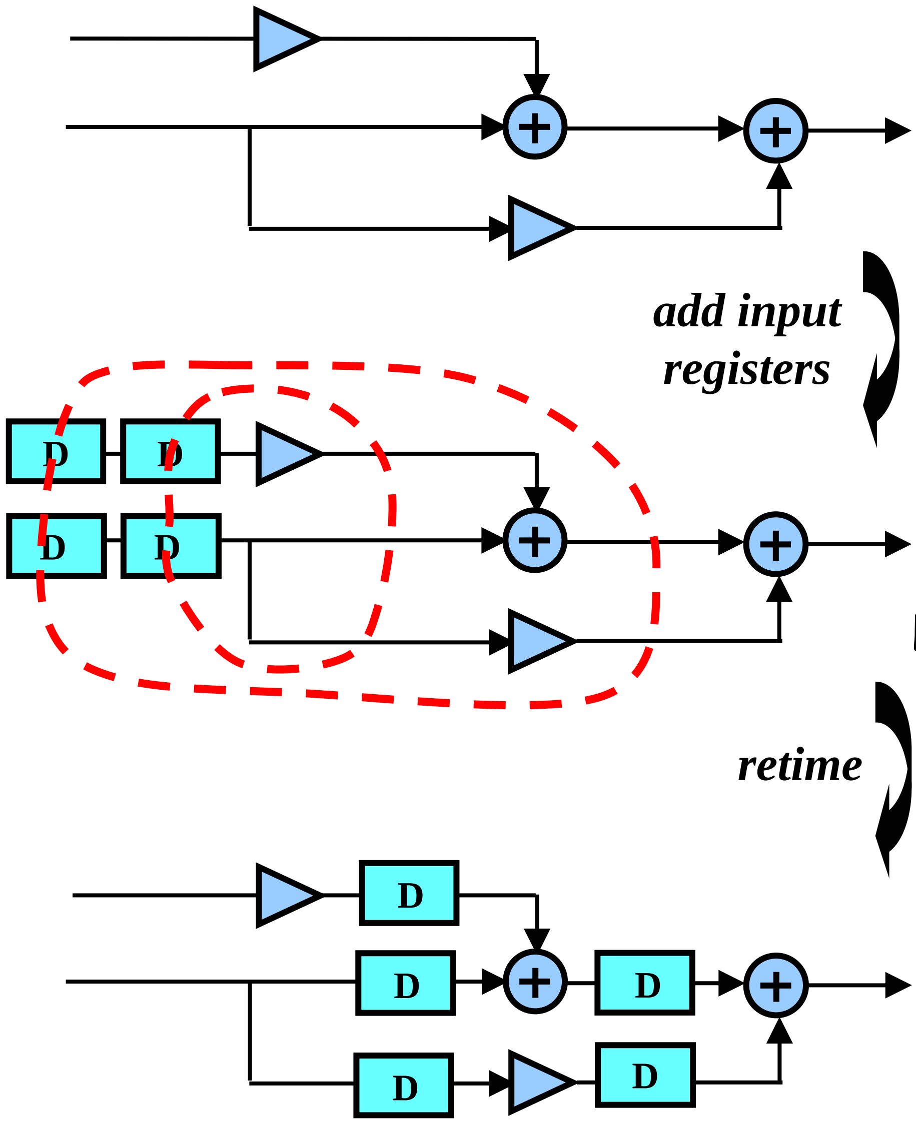




Benefits of retiming:

- Modify critical path delay
- Reduce total number of registers

Pipelining, Just Another Transformation (Pipelining = Adding Delays + Retiming)

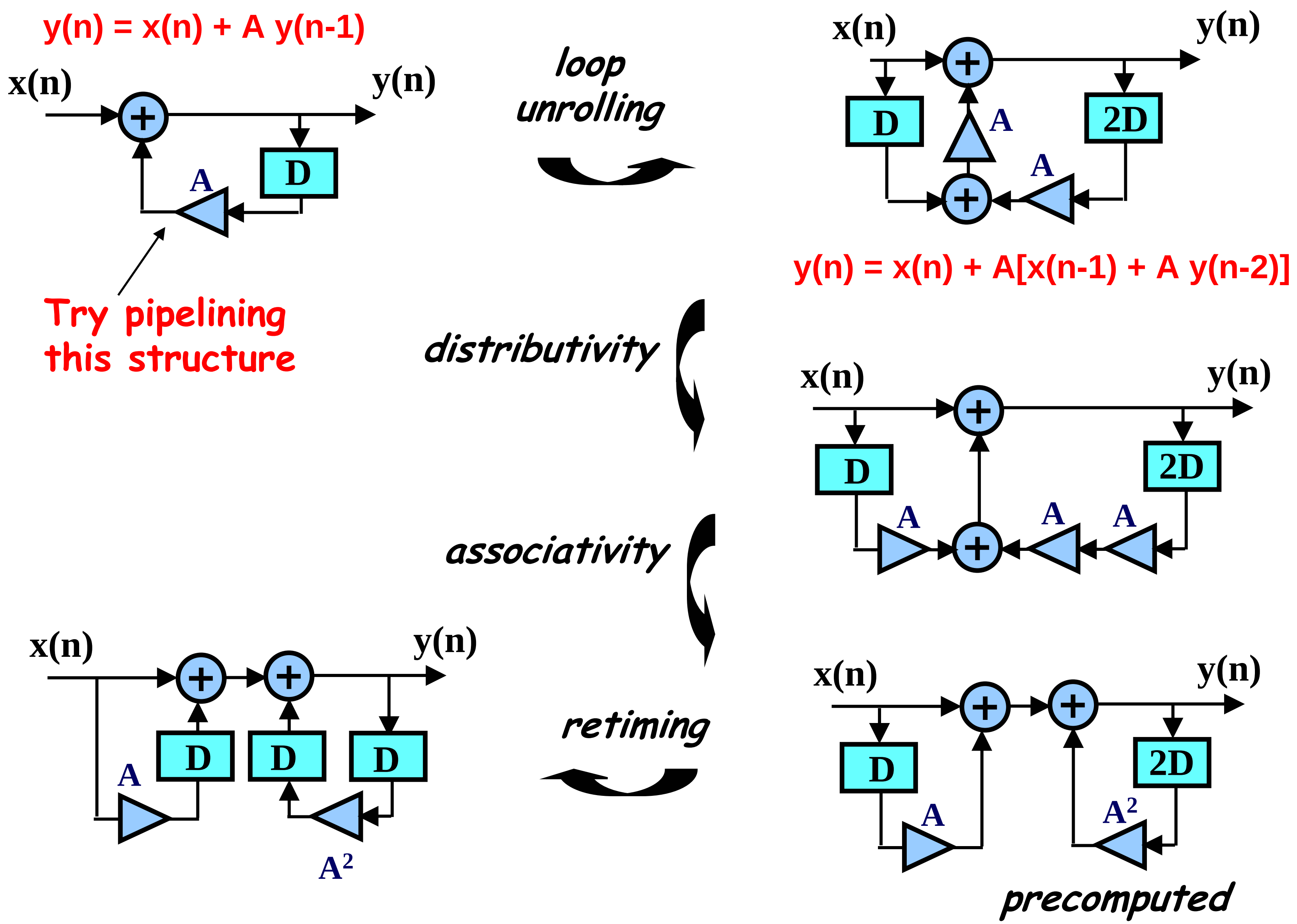


Contrary to retiming, pipelining adds extra registers to the system

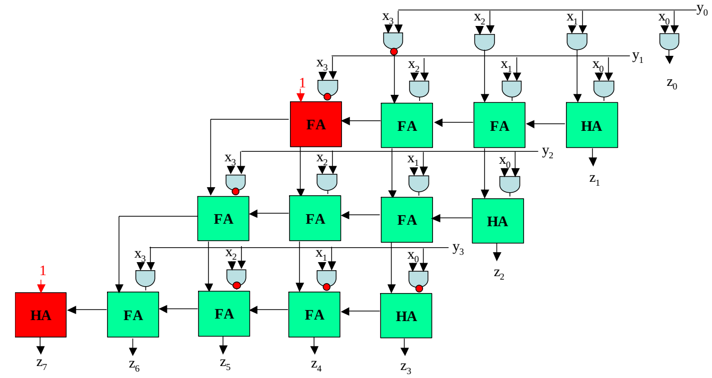
How to pipeline:

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

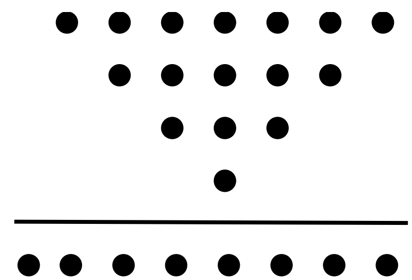
The Power of Transforms: Lookahead



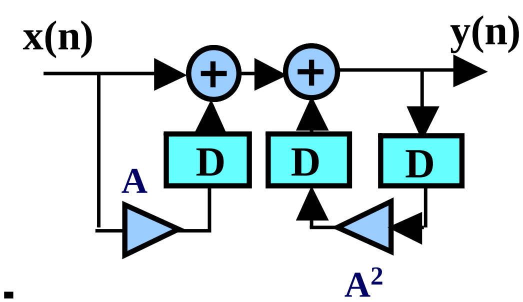
Summary



- **Simple multiplication**:
  - O(N) delay
  - Twos complement easily handled (Baugh-Wooley)
- **Faster multipliers**:
  - Wallace Tree O(log N)



- **Booth recoding**:
  - Add using 2 bits at a time



- **Behavioral Transformations**:
  - Faster circuits using pipelining and algebraic properties