



EECS151/251A
Spring 2018
Digital Design and
Integrated Circuits

Instructors:
John Wawrzynek and Nick Weaver

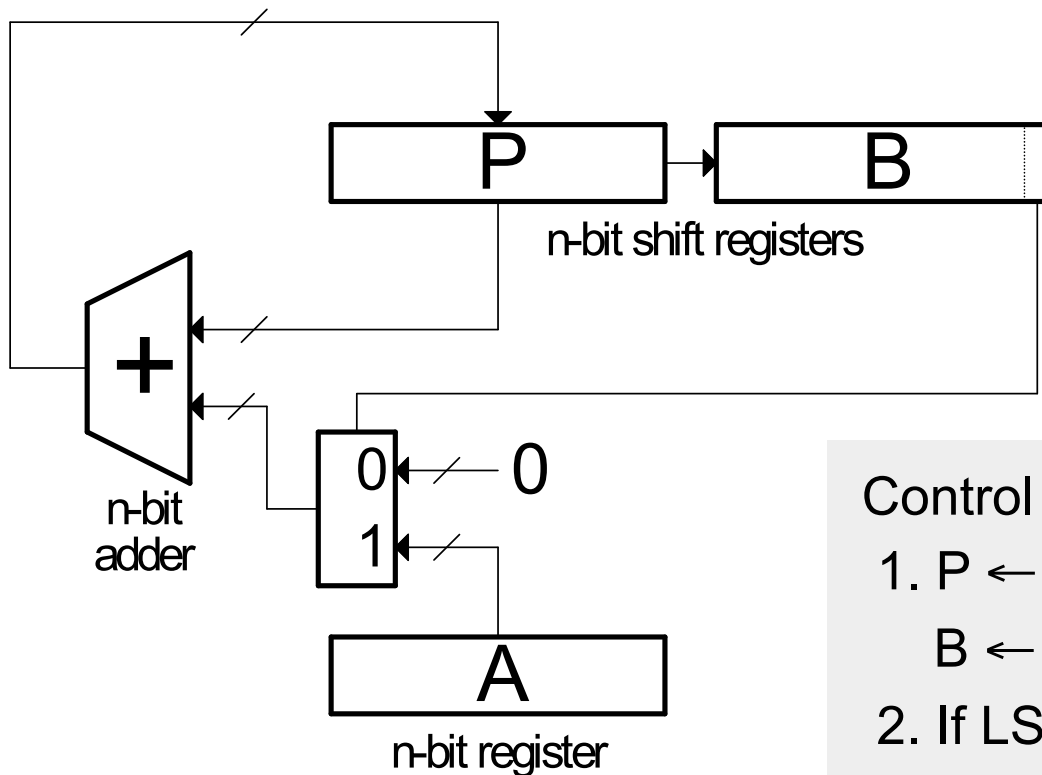
Lecture 21:
Multiplier Circuits

Multiplication

			a_3	a_2	a_1	a_0	← <i>Multiplicand</i>	
			b_3	b_2	b_1	b_0	← <i>Multiplier</i>	
		X	a_3b_0	a_2b_0	a_1b_0	a_0b_0	}	
			a_3b_1	a_2b_1	a_1b_1	a_0b_1		<i>Partial products</i>
			a_3b_2	a_2b_2	a_1b_2	a_0b_2		
			a_3b_3	a_2b_3	a_1b_3	a_0b_3		
			...		$a_1b_0 + a_0b_1$	a_0b_0	← <i>Product</i>	

Many different circuits exist for multiplication. Each one has a different balance between speed (performance) and amount of logic (cost).

“Shift and Add” Multiplier



- Sums each partial product, one at a time.
- In binary, each partial product is shifted versions of A or 0.

Control Algorithm:

1. $P \leftarrow 0$, $A \leftarrow$ multiplicand, $B \leftarrow$ multiplier
2. If LSB of $B=1$ then add A to P
else add 0
3. Shift $[P][B]$ right 1
4. Repeat steps 2 and 3 $n-1$ times.
5. $[P][B]$ has product.

- Cost $\propto n$, $T = n$ clock cycles.
- What is the critical path for determining the min clock period?

“Shift and Add” Multiplier

Signed Multiplication:

Remember for 2's complement numbers MSB has negative weight:

$$X = \sum_{i=0}^{N-2} x_i 2^i - x_{n-1} 2^{n-1}$$

$$\begin{aligned} \text{ex: } -6 &= 11010_2 = 0 \cdot 2^0 + 1 \cdot 2^1 + 0 \cdot 2^2 + 1 \cdot 2^3 - 1 \cdot 2^4 \\ &= 0 + 2 + 0 + 8 - 16 = -6 \end{aligned}$$

- Therefore for multiplication:
 - a) subtract final partial product
 - b) sign-extend partial products
- Modifications to shift & add circuit:
 - a) adder/subtractor
 - b) sign-extender on P shifter register

Outline



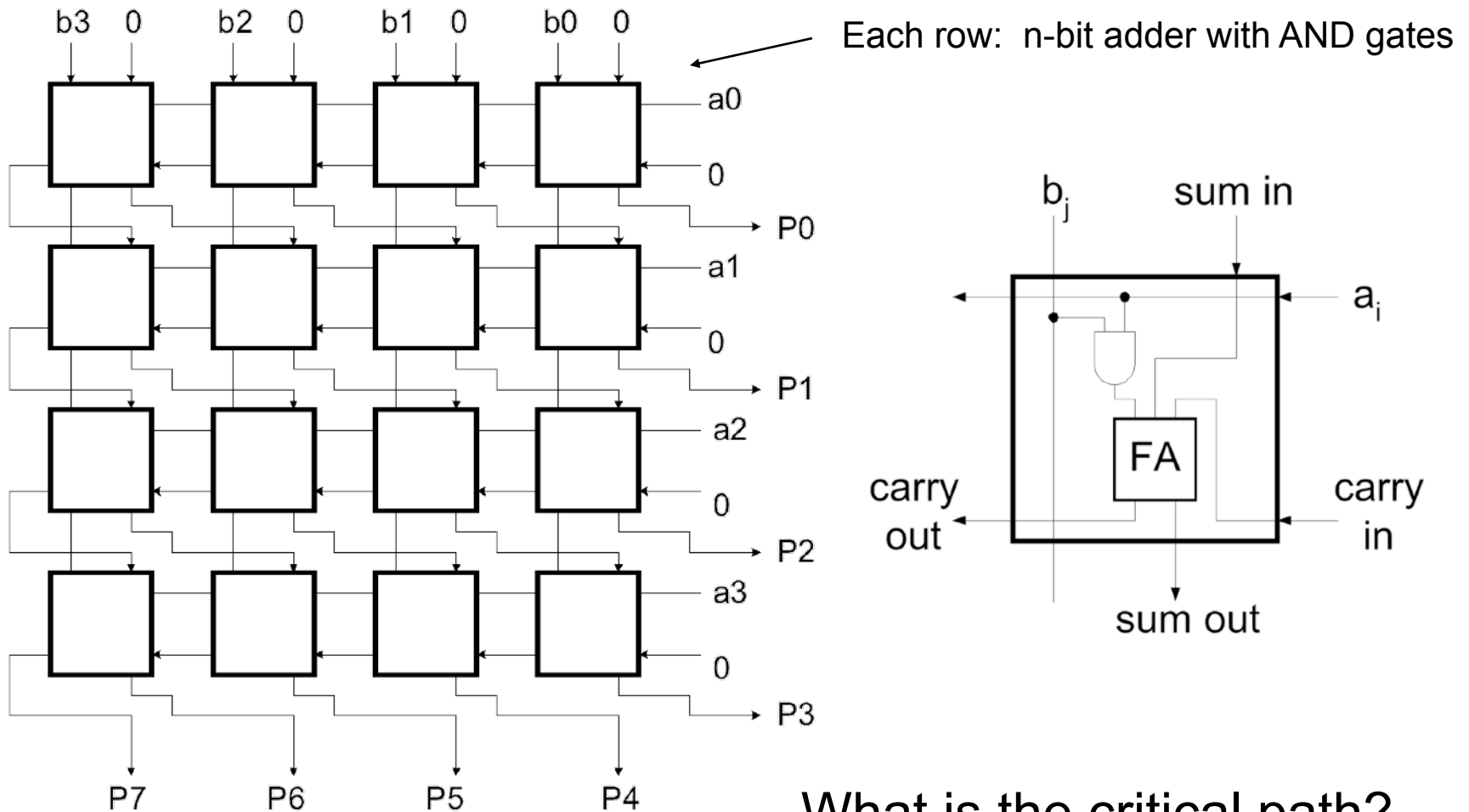
- Combinational multiplier
- Latency & Throughput
 - Wallace Tree
 - Pipelining to increase throughput
- Smaller multipliers
 - Booth encoding
 - Serial, bit-serial
- Two's complement multiplier



Unsigned Combinational Multiplier

Array Multiplier

Single cycle multiply: Generates all n partial products simultaneously.



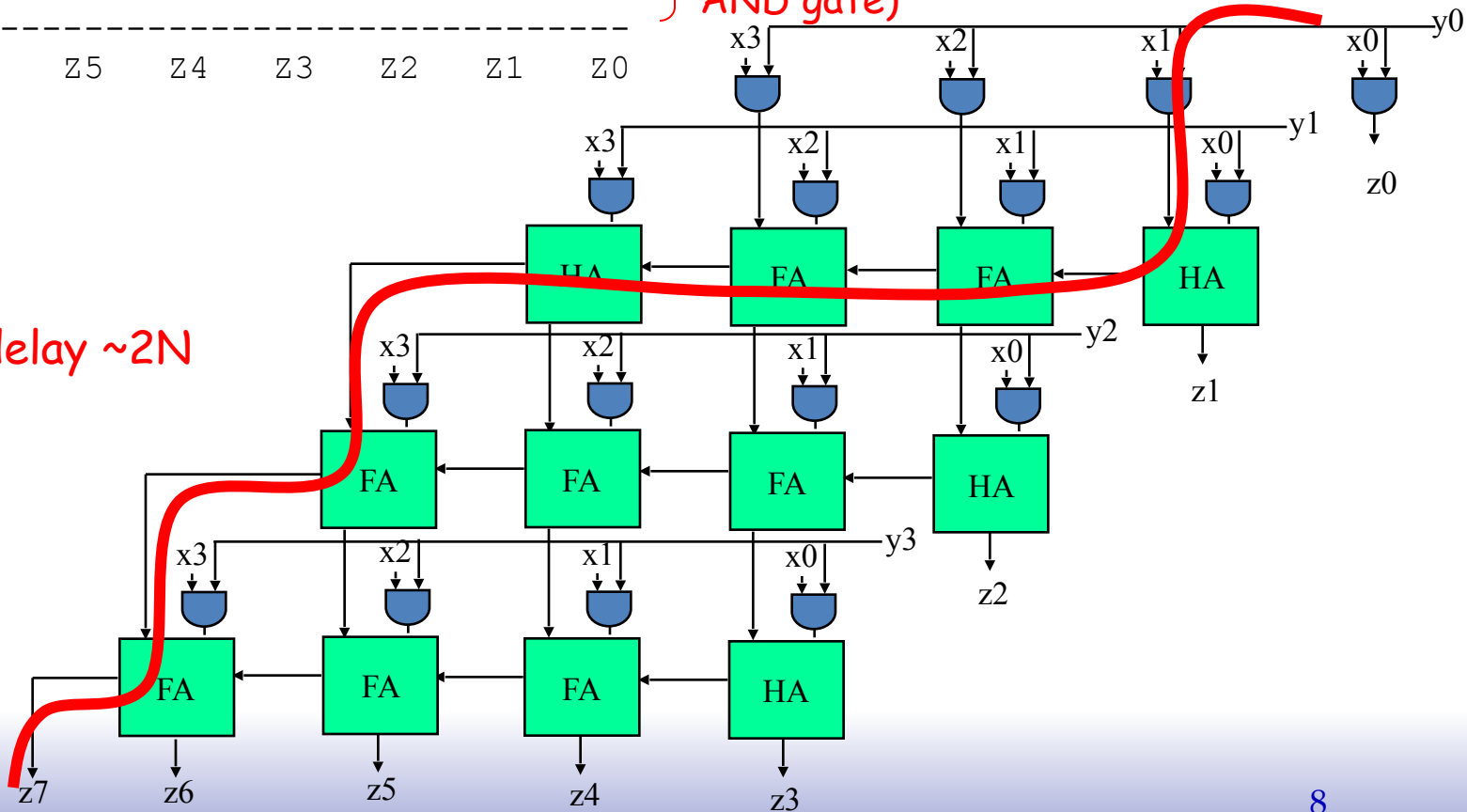
What is the critical path?

Combinational Multiplier (unsigned)

	X3	X2	X1	X0	← multiplicand
*	Y3	Y2	Y1	Y0	← multiplier

		X3Y0	X2Y0	X1Y0	X0Y0			
+		X3Y1	X2Y1	X1Y1	X0Y1			
+		X3Y2	X2Y2	X1Y2	X0Y2			
+	X3Y3	X2Y3	X1Y3	X0Y3				
	z7	z6	z5	z4	z3	z2	z1	z0

Partial products, one for each bit in multiplier (each bit needs just one AND gate)



Propagation delay $\sim 2N$

Carry-Save Addition

- Speeding up multiplication is a matter of speeding up the summing of the partial products.
- “Carry-save” addition can help.
- Carry-save addition passes (saves) the carries to the output, rather than propagating them.

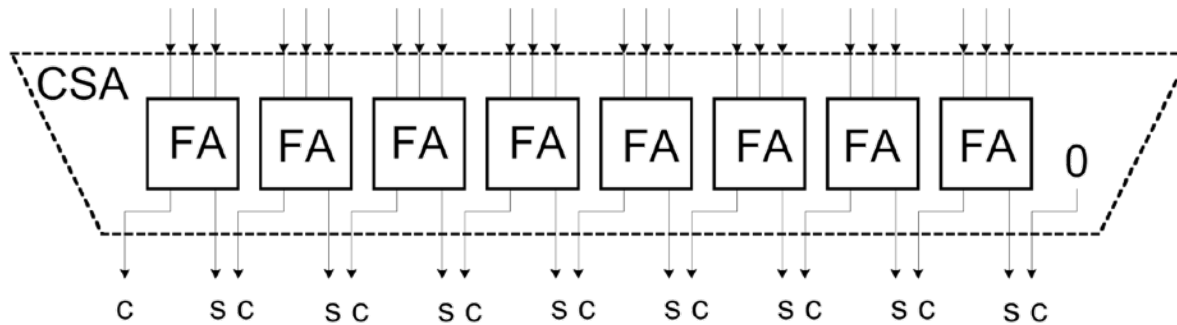
- Example: sum three numbers, $3_{10} = 0011$, $2_{10} = 0010$, $3_{10} = 0011$

$$\begin{array}{r}
 3_{10} \quad 0011 \\
 + 2_{10} \quad 0010 \\
 \hline
 c \quad 0100 = 4_{10} \\
 s \quad 0001 = 1_{10} \\
 \hline
 3_{10} \quad 0011 \\
 + \quad \quad 0010 \\
 \hline
 c \quad 0010 = 2_{10} \\
 s \quad 0110 = 6_{10} \\
 \hline
 1000 = 8_{10}
 \end{array}$$

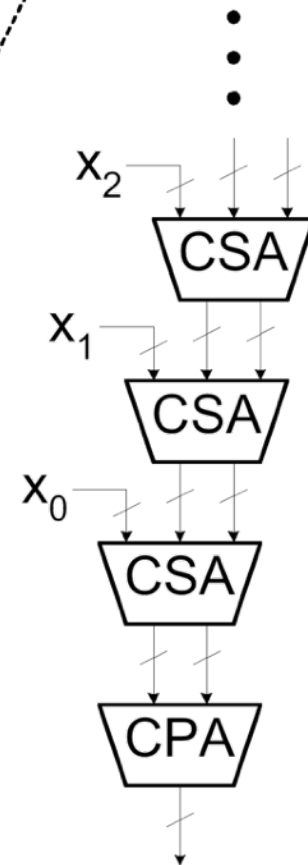
The first two additions (3₁₀ + 2₁₀) and (3₁₀ + 2₁₀) are grouped by a bracket on the right and labeled "carry-save add".
 The second two additions (3₁₀ + 3₁₀) and (3₁₀ + 3₁₀) are grouped by a bracket on the left and labeled "carry-propagate add".

- In general, *carry-save* addition takes in 3 numbers and produces 2.
- Whereas, *carry-propagate* takes 2 and produces 1.
- With this technique, we can avoid carry propagation until final addition

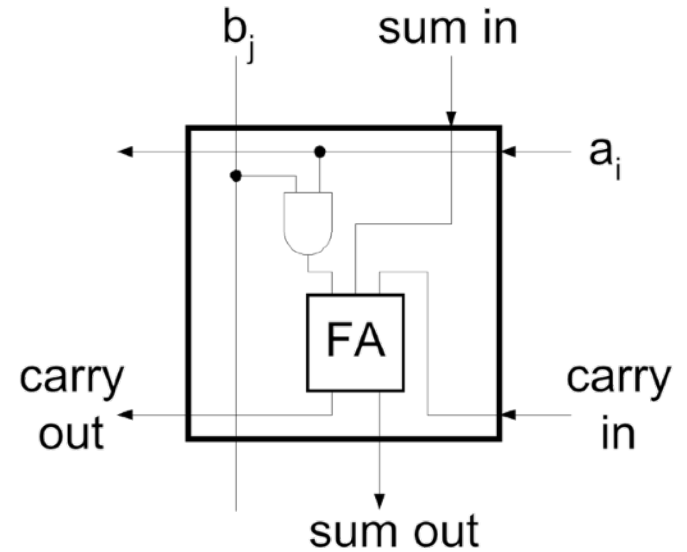
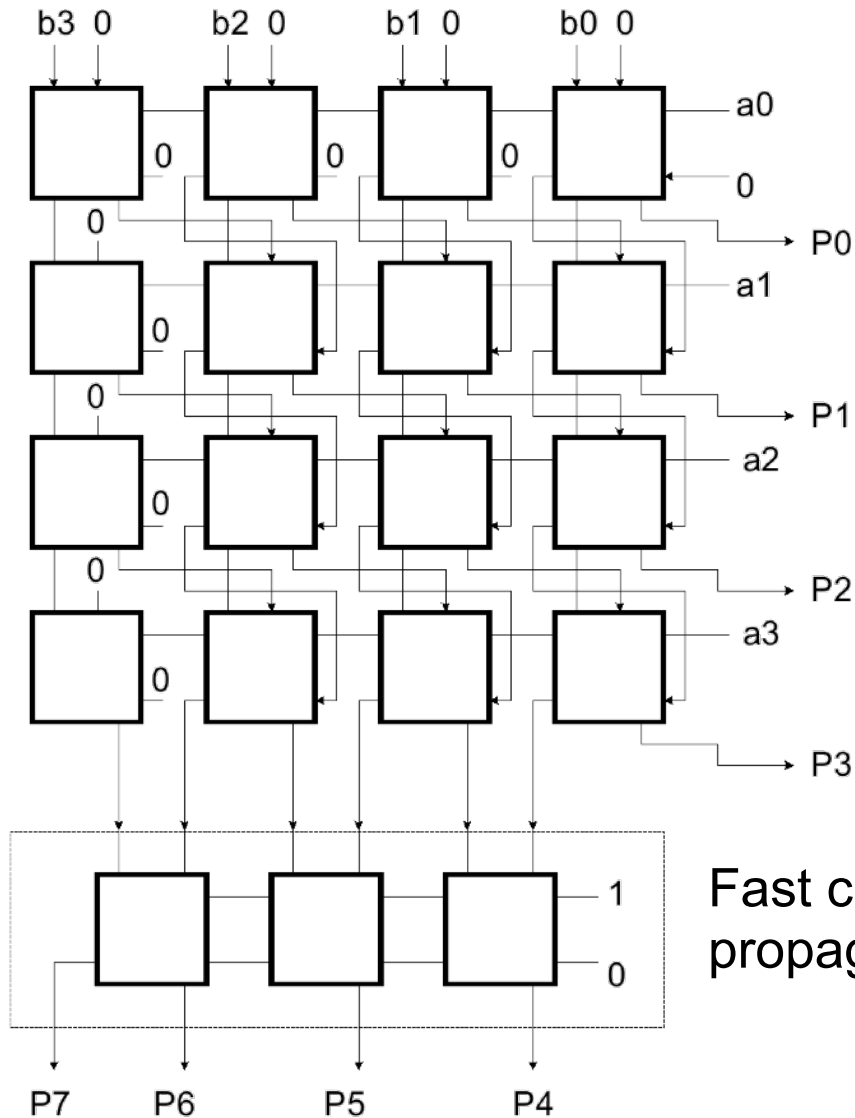
Carry-save Circuits



- When adding sets of numbers, carry-save can be used on all but the final sum.
- Standard adder (carry propagate) is used for final sum.
- Carry-save is fast (no carry propagation) and cheap (same cost as ripple adder)



Array Multiplier using Carry-save Addition

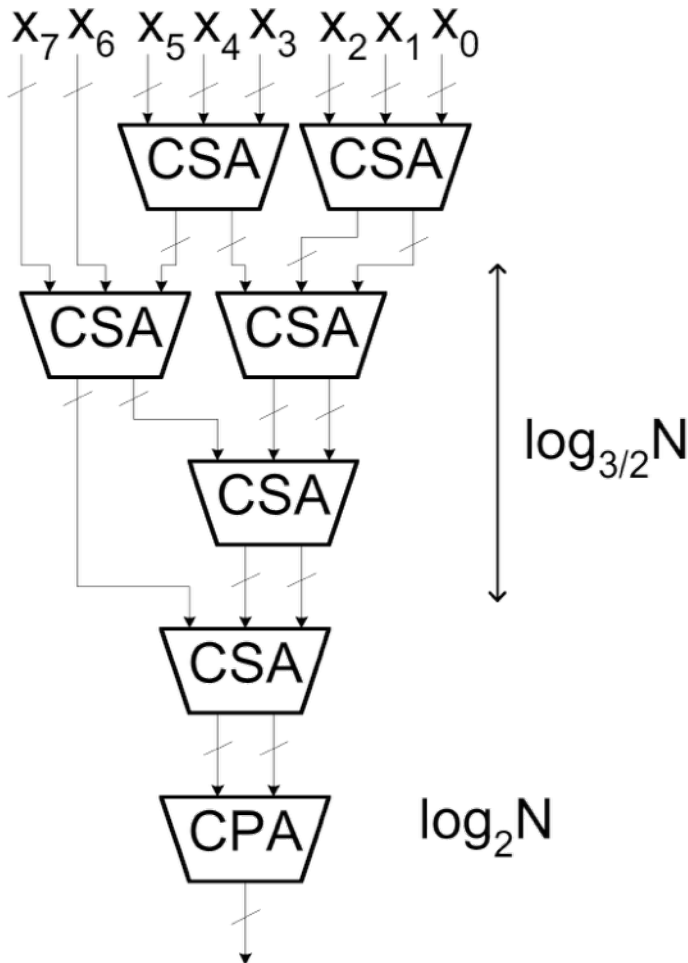


Fast carry-propagate adder

Carry-save Addition

CSA is associative and commutitive. For example:

$$(((X_0 + X_1) + X_2) + X_3) = ((X_0 + X_1) + (X_2 + X_3))$$

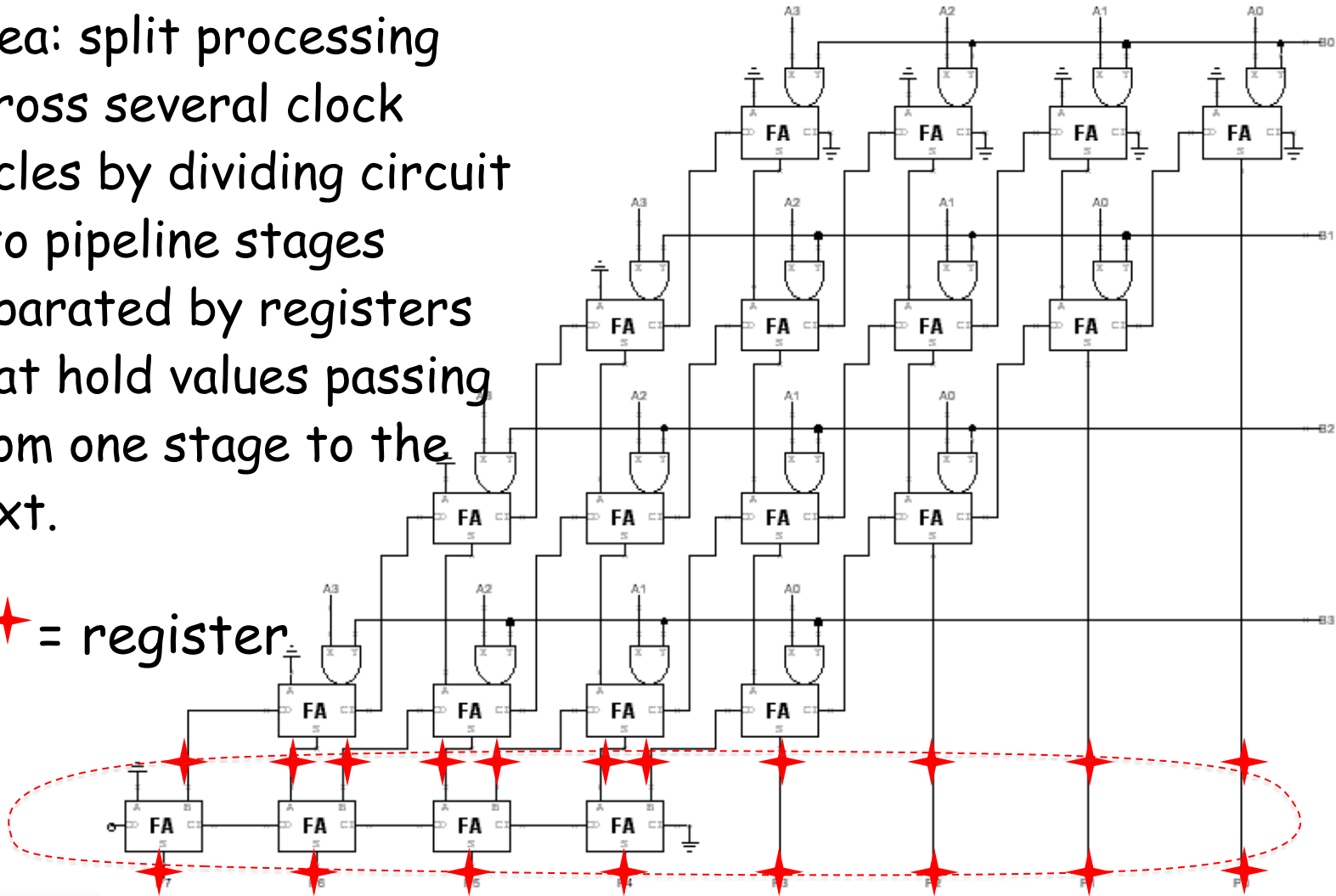


- A balanced tree can be used to reduce the logic delay.
- This structure is the basis of the **Wallace Tree Multiplier**.
- Partial products are summed with the CSA tree. Fast CPA (ex: CLA) is used for final sum.
- Multiplier delay $\propto \log_{3/2} N + \log_2 N$

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.

★ = register



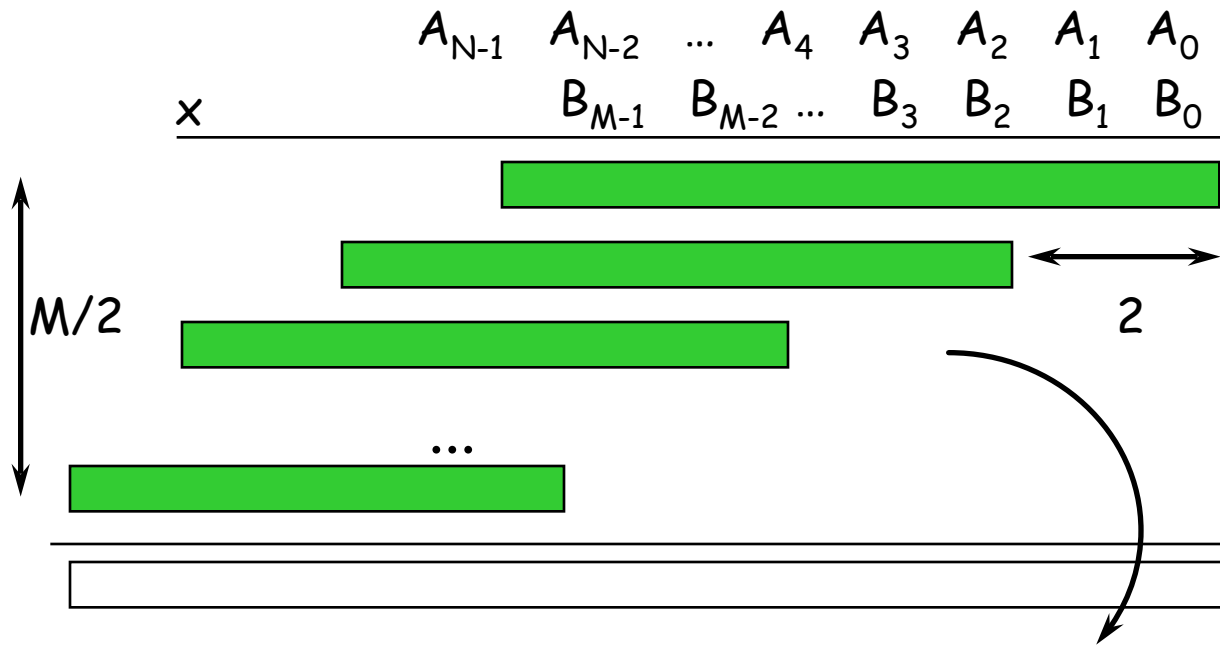
Throughput = $1/4t_{PD,FA}$ instead of $1/8t_{PD,FA}$ 13



Smaller Combinational Multipliers

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

(On-the-fly canonical signed digit encoding!)

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
1	1	0	sub A
1	1	1	add 0

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

$$\leftarrow -2^*A + A$$

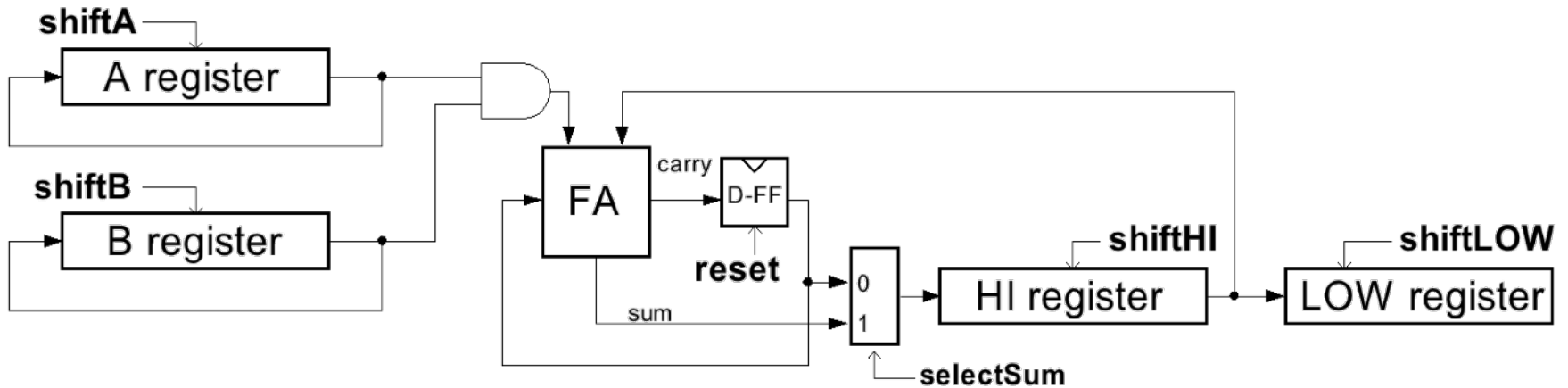
$$\leftarrow -A + A$$



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!

Bit-serial Multiplier

- Bit-serial multiplier (n^2 cycles, one bit of result per n cycles):



- Control Algorithm:

```

repeat n cycles { // outer (i) loop
    repeat n cycles{ // inner (j) loop
        shiftA, selectSum, shiftHI
    }
    shiftB, shiftHI, shiftLOW, reset
}

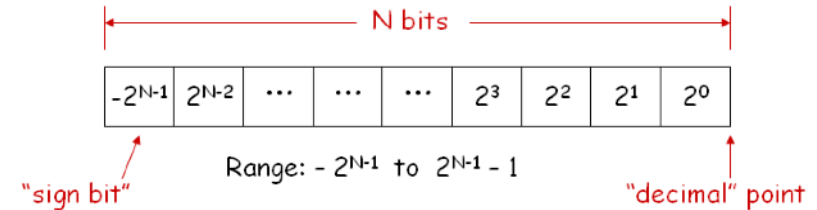
```

Note: The occurrence of a control signal x means $x=1$. The absence of x means $x=0$.



Signed Multipliers

Combinational Multiplier (signed!)



$$(-3) * (-2)$$

(-3)

(-2)

	1	0	1	(X)
*	1	1	0	(Y)

0 0 0 0 0 0

$Y_0 * X = 0$

+ **1 1 1 0 1**

$2Y_1 * X = -6$

- **1 1 0 1**

$4Y_2 * X = -12$

(+6)

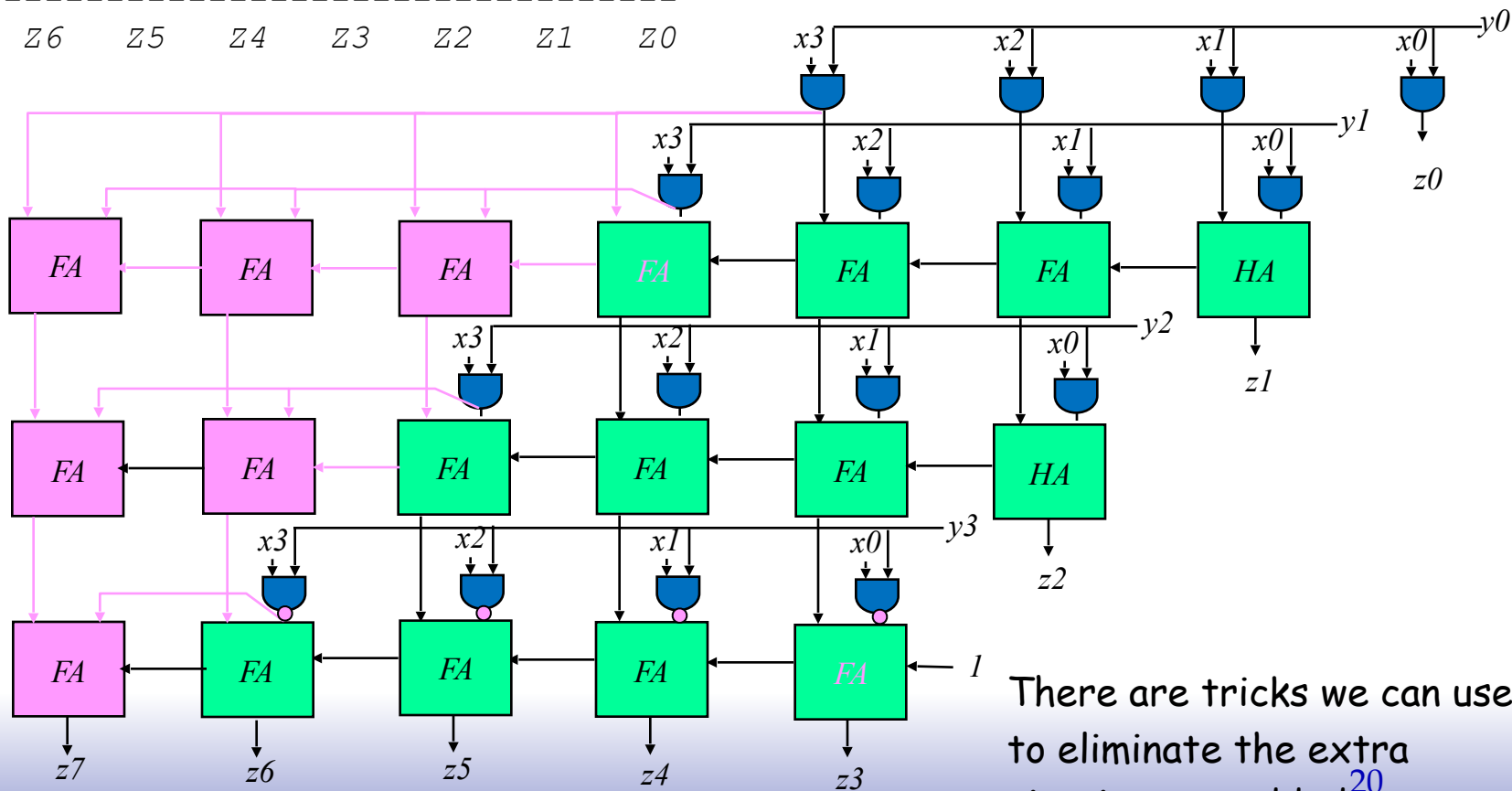
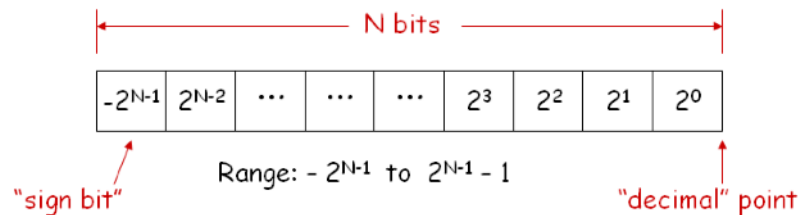
0 0 0 1 1 0

Combinational Multiplier (signed)

$$\begin{array}{r}
 X3 \quad X2 \quad X1 \quad X0 \\
 * Y3 \quad Y2 \quad Y1 \quad Y0 \\
 \hline
 \end{array}$$

$$\begin{array}{r}
 X3Y0 \quad X3Y0 \quad X3Y0 \quad X3Y0 \quad X3Y0 \quad X2Y0 \quad X1Y0 \quad X0Y0 \\
 + X3Y1 \quad X3Y1 \quad X3Y1 \quad X3Y1 \quad X2Y1 \quad X1Y1 \quad X0Y1 \\
 + X3Y2 \quad X3Y2 \quad X3Y2 \quad X2Y2 \quad X1Y2 \quad X0Y2 \\
 - X3Y3 \quad X3Y3 \quad X2Y3 \quad X1Y3 \quad X0Y3 \\
 \hline
 \end{array}$$

z7 z6 z5 z4 z3 z2 z1 z0



There are tricks we can use to eliminate the extra circuitry we added...²⁰

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).

	x3y0	x3y0	x3y0	x3y0	x3y0	x2y0	x1y0	x0y0	
+					1				
+	x3y1	x3y1	x3y1	x3y1	x2y1	x1y1	x0y1		
+				1					
+	x3y2	x3y2	x3y2	x2y2	x1y2	x0y2			
+			1						
+	x3y3	x3y3	x2y3	x1y3	x0y3				
+					1				
+		1							
-		1	1	1	1				

} **-B = ~B + 1**

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

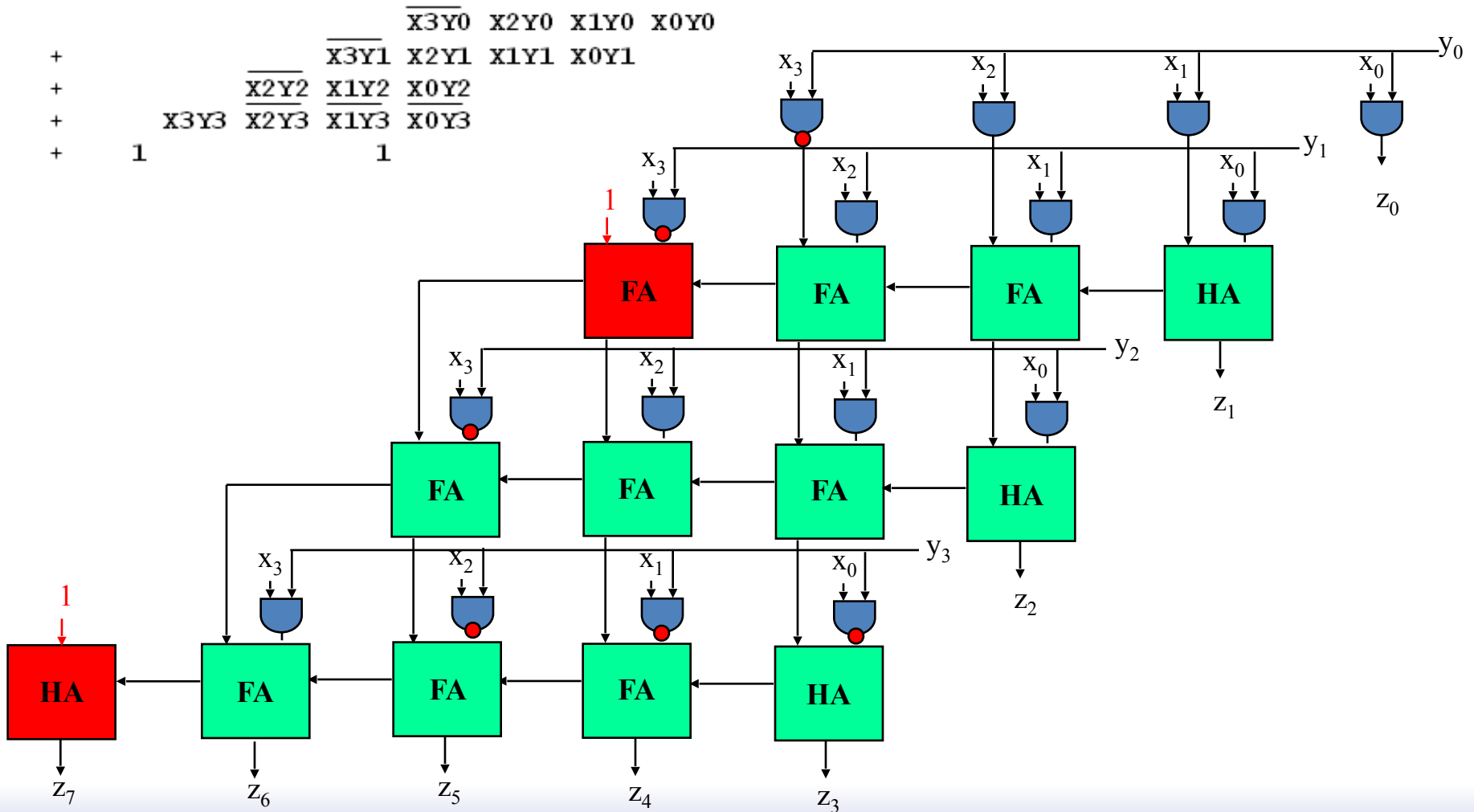
							x3y0	x2y0	x1y0	x0y0
+							x3y1	x2y1	x1y1	x0y1
+						x2y2	x1y2	x0y2		
+		x3y3	x2y3	x1y3	x0y3					
+										
+								1		
-		1	1	1	1			1		

Step 4: finish computing the constants...

								x3y0	x2y0	x1y0	x0y0
+							x3y1	x2y1	x1y1	x0y1	
+						x2y2	x1y2	x0y2			
+		x3y3	x2y3	x1y3	x0y3						
+									1		
+	1								1		

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.

```
wire signed [9:0] a;  
wire [9:0] b;  
wire signed [19:0] result = a*$signed(b);
```

To make a signed constant: 10’sh37C