

EECS151 : Introduction to Digital Design and ICs

Lecture 16 – Multipliers



Bora Nikolić and Sophia Shao



Introducing the new SiFive U8-Series Core IP

Introducing the new SiFive U8-Series Core IP

SiFive will initially offer two standard cores in the SiFive U8-Series Core IP lineup. The standard core offerings will be the SiFive U84 core, optimized for power efficiency and area efficiency, and the SiFive U87 core with vector processing.



The slide features the SiFive logo (a stylized 'S' inside a hexagon) and the text "SiFive U8-Series". Below it, a sub-headline reads "Introducing U8-Class Processor Core IP". Two sections follow: "SiFive U84 A High-Performance Scalable 8-Series Core" and "SiFive U87 A High-Performance Scalable 8-Series Core with Vector Processing". To the right is a graphic of a RISC-V processor chip.

SiFive, 10/24/2019

Review

- Binary adders are a common building block of digital systems
- Carry is in the critical path
- Carry-bypass, carry-select are usually faster than ripple-carry for lengths > 8
- Carry-lookahead, $O(\sim \log N)$ is often the fastest adder with $N > 16$



Multipliers

Warmup

- Recall long multiplication of base-10 by hand:

$$\begin{array}{r} 12 \\ \times 56 \\ \hline \end{array}$$

- In base-2 (binary), we do the same thing:

$$\begin{array}{r} 011 \\ \times 101 \\ \hline \end{array}$$

Multiplication

$$\begin{array}{r} & a_3 & a_2 & a_1 & a_0 \\ \times & b_3 & b_2 & b_1 & b_0 \\ \hline \end{array}$$

Multiplicand
Multiplier

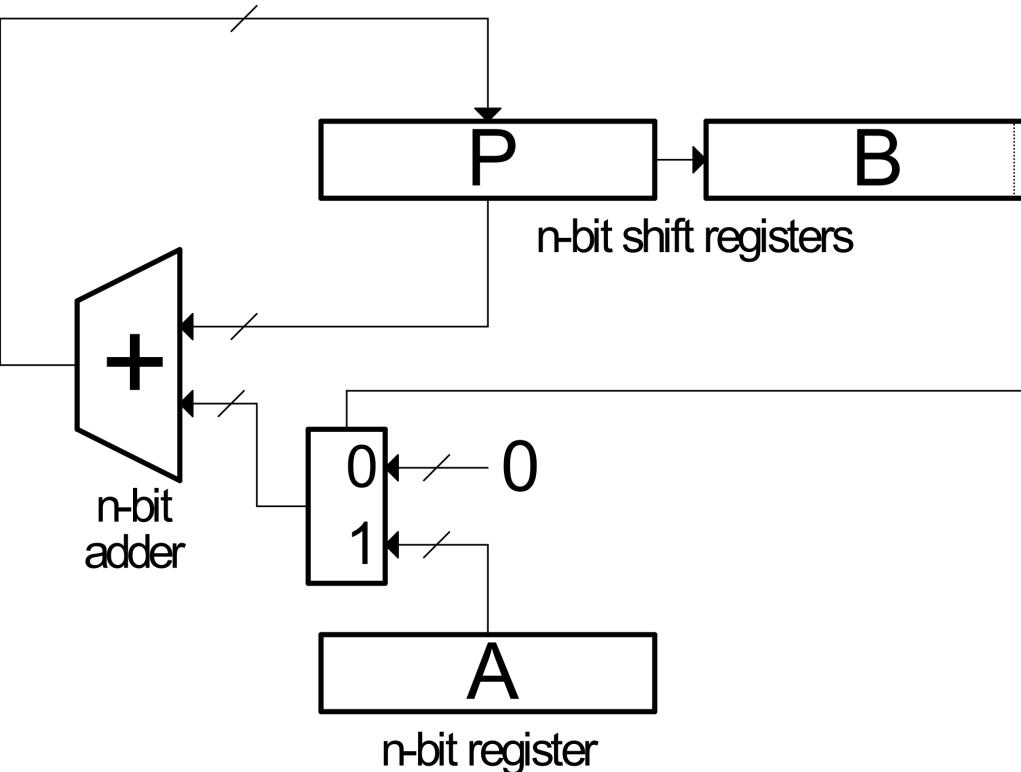
$$\begin{array}{cccc} a_3b_0 & a_2b_0 & a_1b_0 & a_0b_0 \\ a_3b_1 & a_2b_1 & a_1b_1 & a_0b_1 \\ a_3b_2 & a_2b_2 & a_1b_2 & a_0b_2 \\ a_3b_3 & a_2b_3 & a_1b_3 & a_0b_3 \\ \hline \dots & & a_1b_0 + a_0b_1 & a_0b_0 \leftarrow \text{Product} \end{array}$$

Partial products

Many different circuits exist for multiplication.

Each one has a different balance between *speed (performance)* and amount of *logic (energy, cost)*.

“Shift and Add” Multiplier



- Performance: N cycles of N-bit additions

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

Control Algorithm:

1. $P \leftarrow 0, A \leftarrow \text{multiplicand}, B \leftarrow \text{multiplier}$
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 $(n-1)$ more times.
5. $[P][B]$ has product.

“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:
 - subtract final partial product
 - sign-extend partial products
- Modifications to shift & add circuit:
 - adder/subtractor
 - sign-extender on P shifter register

Convince yourself

- What's -3×5 ?

$$\begin{array}{r} 1101 \\ \times 0101 \\ \hline \end{array}$$



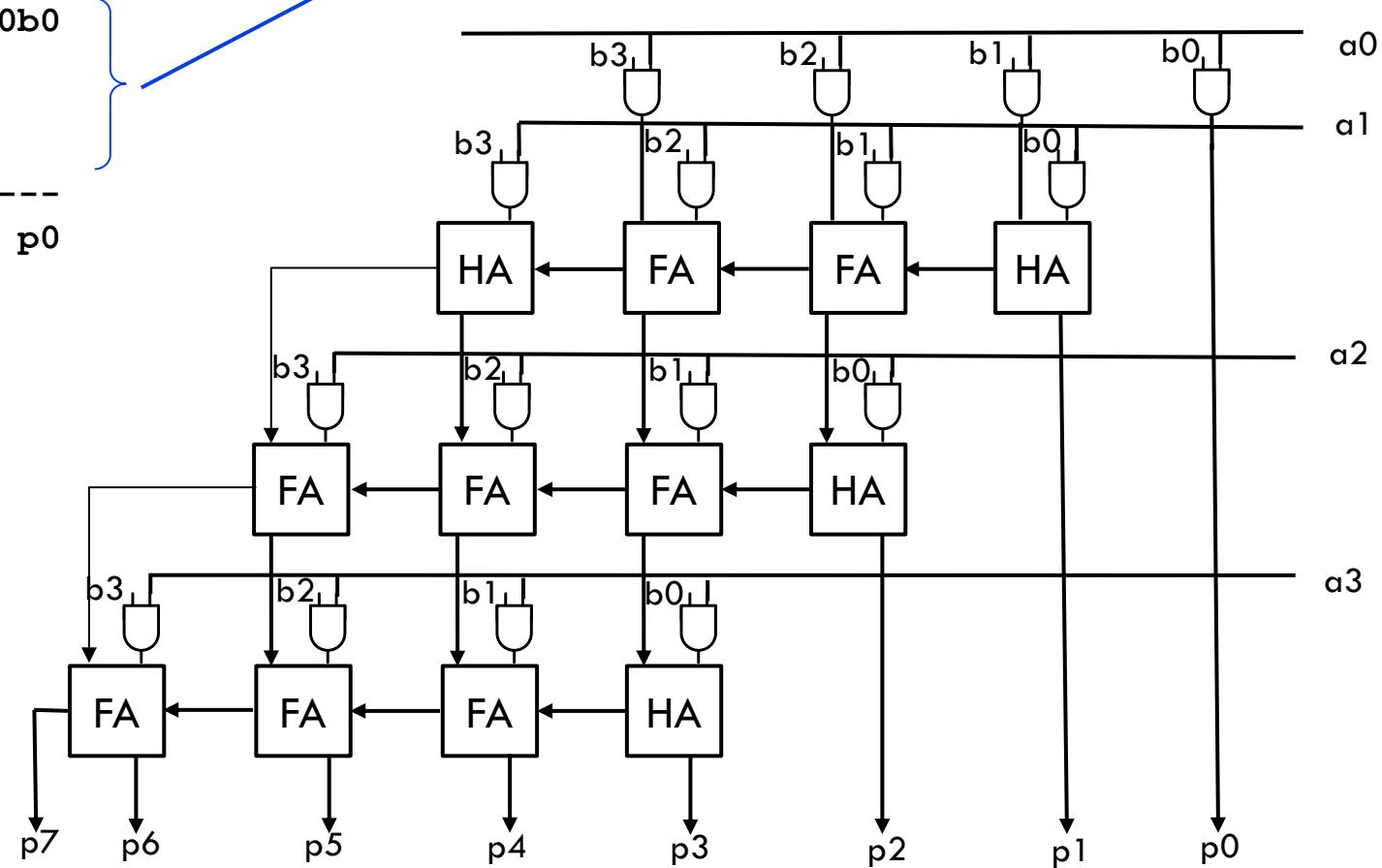
Unsigned Parallel Multiplier

Parallel (Array) Multiplier

$$\begin{array}{r} \begin{array}{cccc} & a_3 & a_2 & a_1 & a_0 \\ * & b_3 & b_2 & b_1 & b_0 \end{array} \\ \hline \begin{array}{ccccccccc} & a_3b_0 & a_2b_0 & a_1b_0 & a_0b_0 & & & & \\ & a_3b_1 & b_2a_1 & a_1b_1 & a_0b_1 & & & & \\ & a_3b_2 & a_2b_2 & a_1b_2 & a_0b_2 & & & & \\ & a_3b_3 & a_2b_3 & a_1b_3 & a_0b_3 & & & & \end{array} \\ \hline p_7 & p_6 & p_5 & p_4 & p_3 & p_2 & p_1 & p_0 \end{array}$$

multiplicand
multiplier

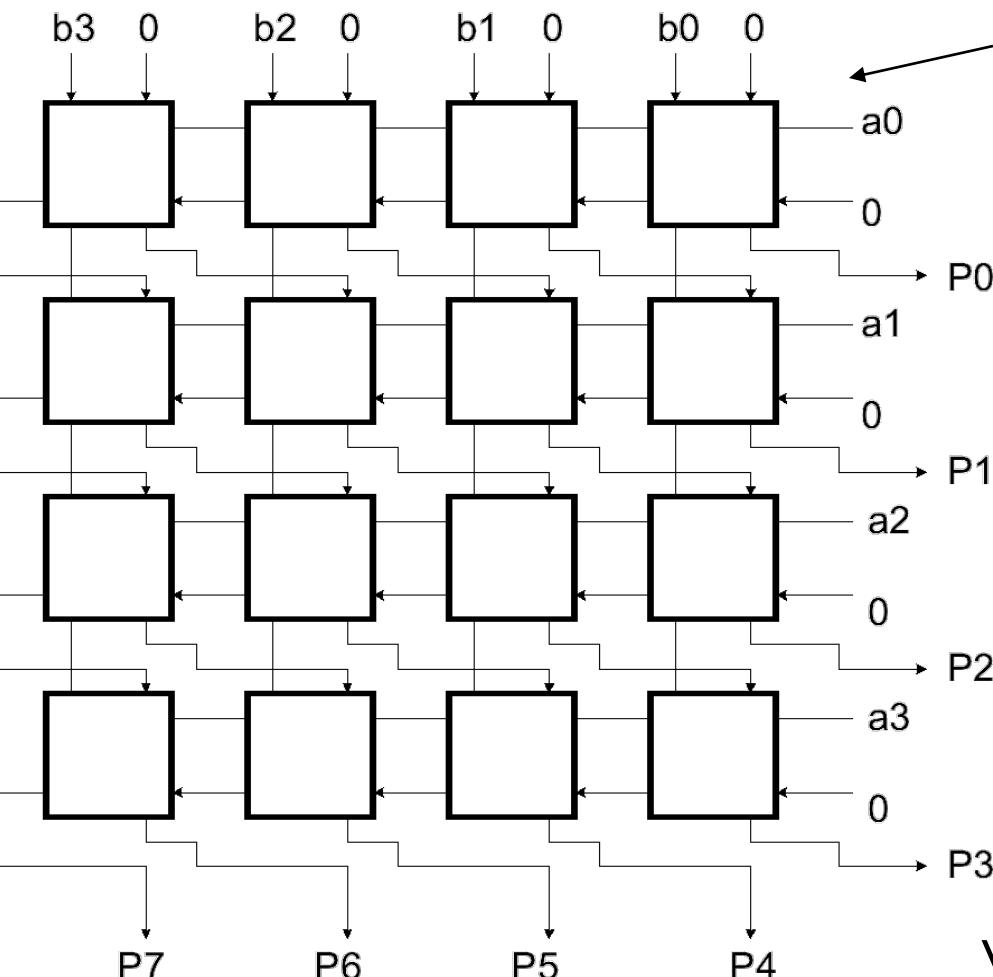
Partial products, one for each bit in multiplier
(each bit needs just one AND gate)



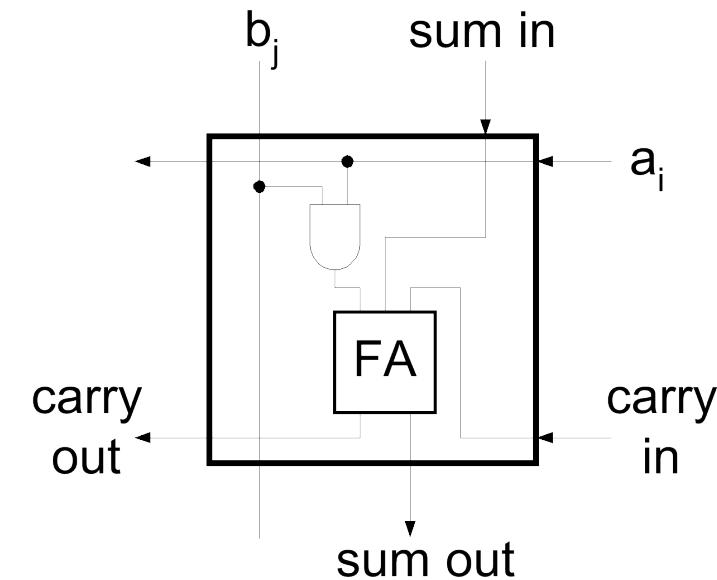
- Performance: What is the critical path?

Parallel (Array) Multiplier

Single cycle multiply: Generates all n partial products simultaneously.



Each row: n -bit adder with AND gates



What is the critical path?

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,

$$3_{10} = 0011, 2_{10} = 0010, 3_{10} = 0011$$

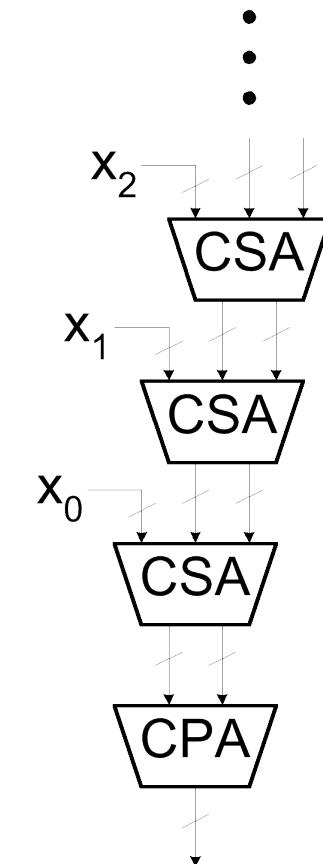
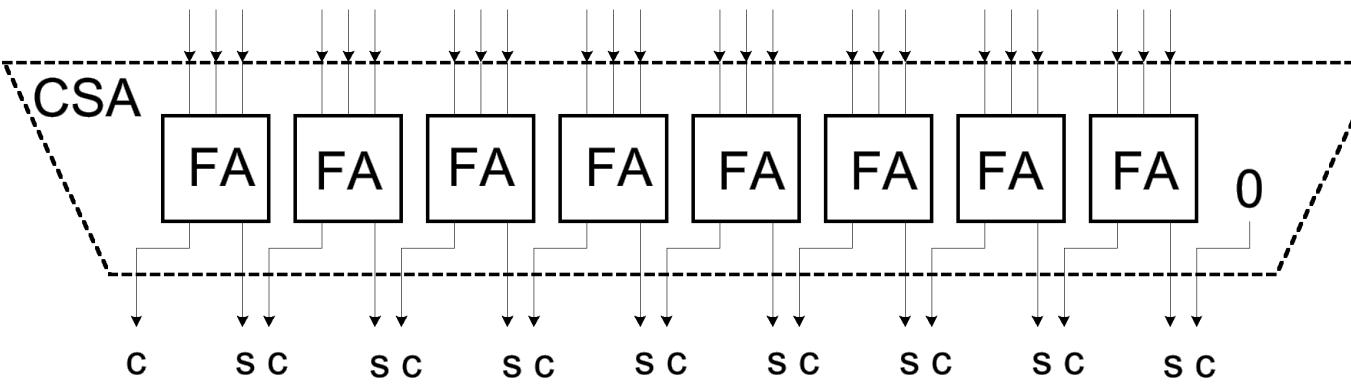
$$\left. \begin{array}{r} 3_{10} \quad 0011 \\ + 2_{10} \quad 0010 \\ \hline c \quad 0100 \\ s \quad 0001 \end{array} \right\} = 4_{10}$$

carry-save add

$$\left. \begin{array}{r} \text{carry-save add} \\ \text{carry-propagate add} \end{array} \right\} + 3_{10} \quad \begin{array}{r} 0011 \\ \hline 0010 \\ \hline 0110 \end{array} = \begin{array}{r} 2_{10} \\ 6_{10} \\ \hline 1000 = 8_{10} \end{array}$$

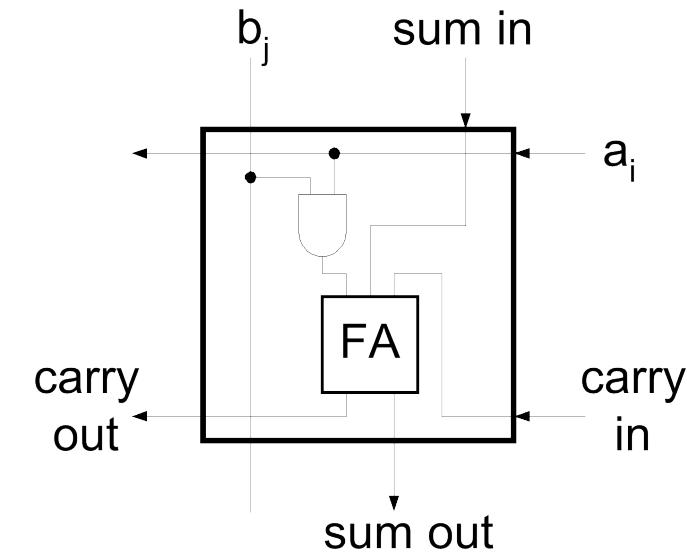
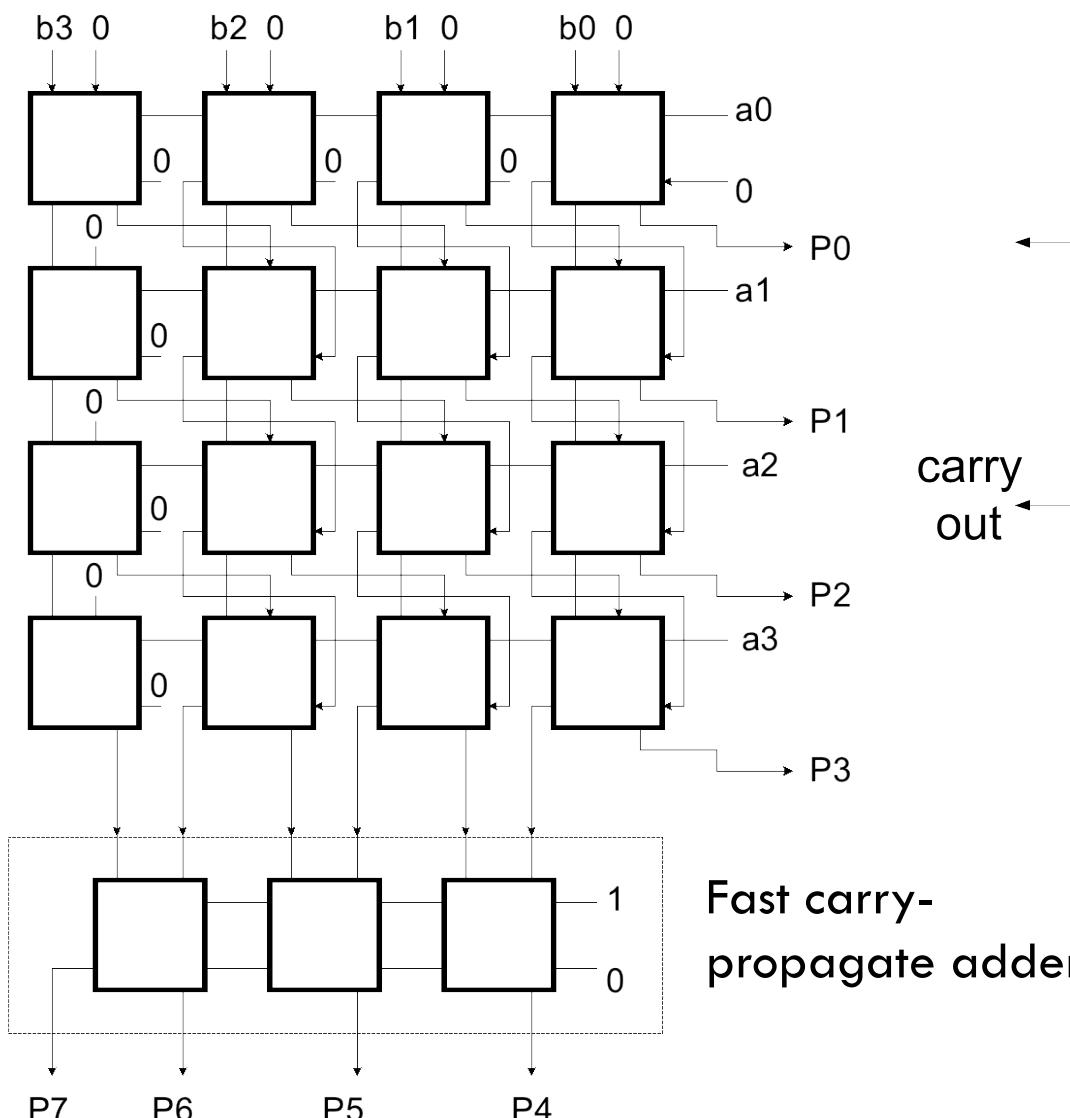
- In general, carry-save addition takes in 3 numbers and produces 2: “3:2 compressor”:
- Whereas, carry-propagate takes 2 and produces 1.
- With this technique, we can avoid carry propagation until final addition

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 inexpensive (full adders)

Array Multiplier Using Carry-Save Addition



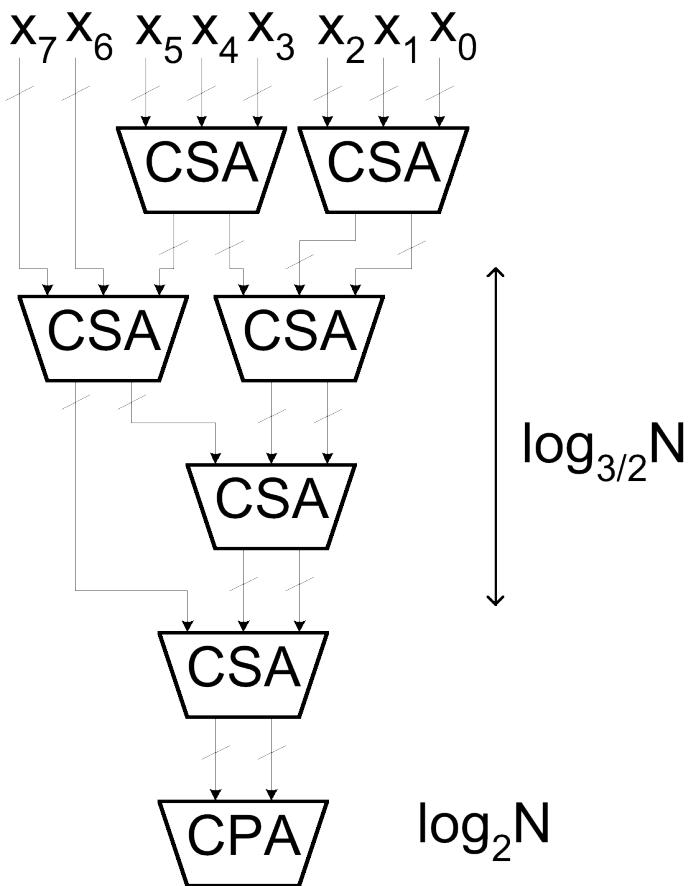
Fast carry-
propagate adder

- What is the critical path?

Carry-Save Addition

CSA is associative and commutative. 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
- It doesn't matter where you add the carries and sums, as long as you eventually do add them
- 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 $\propto \log_{3/2} N + \log_2 N$

Wallace-Tree Multiplier

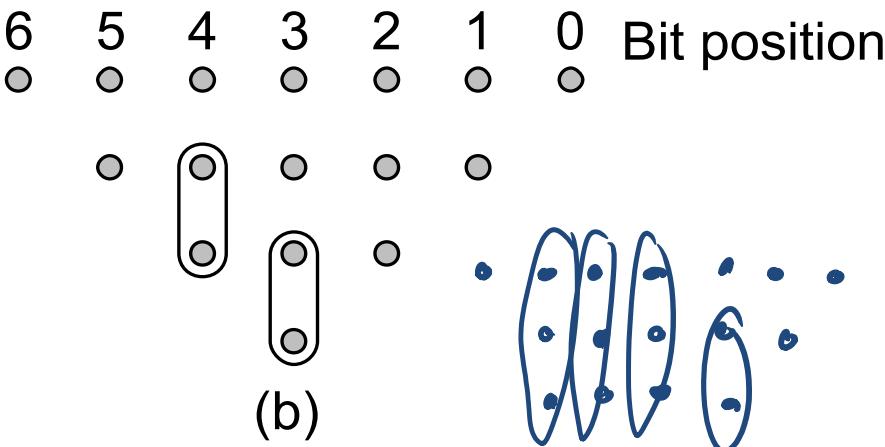
- Reduce the partial products in logic stages – 4×4 example

Partial products

6	5	4	3	2	1	0
○	○	○	○	○	○	○
○	○	○	○	○	○	○
○	○	○	○	○	○	○
○	○	○	○	○	○	○

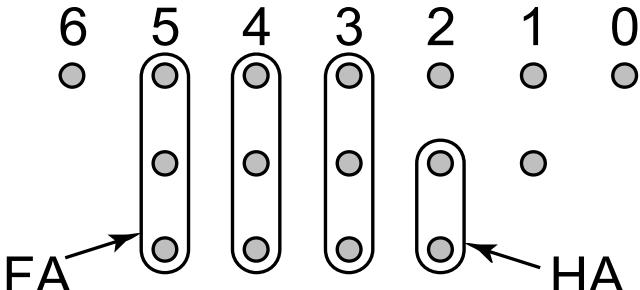
(a)

First stage



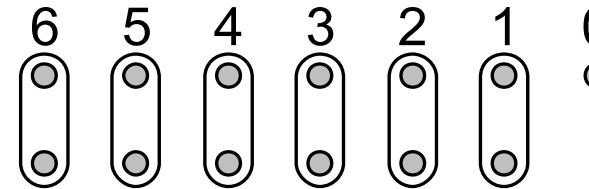
(b)

Second stage



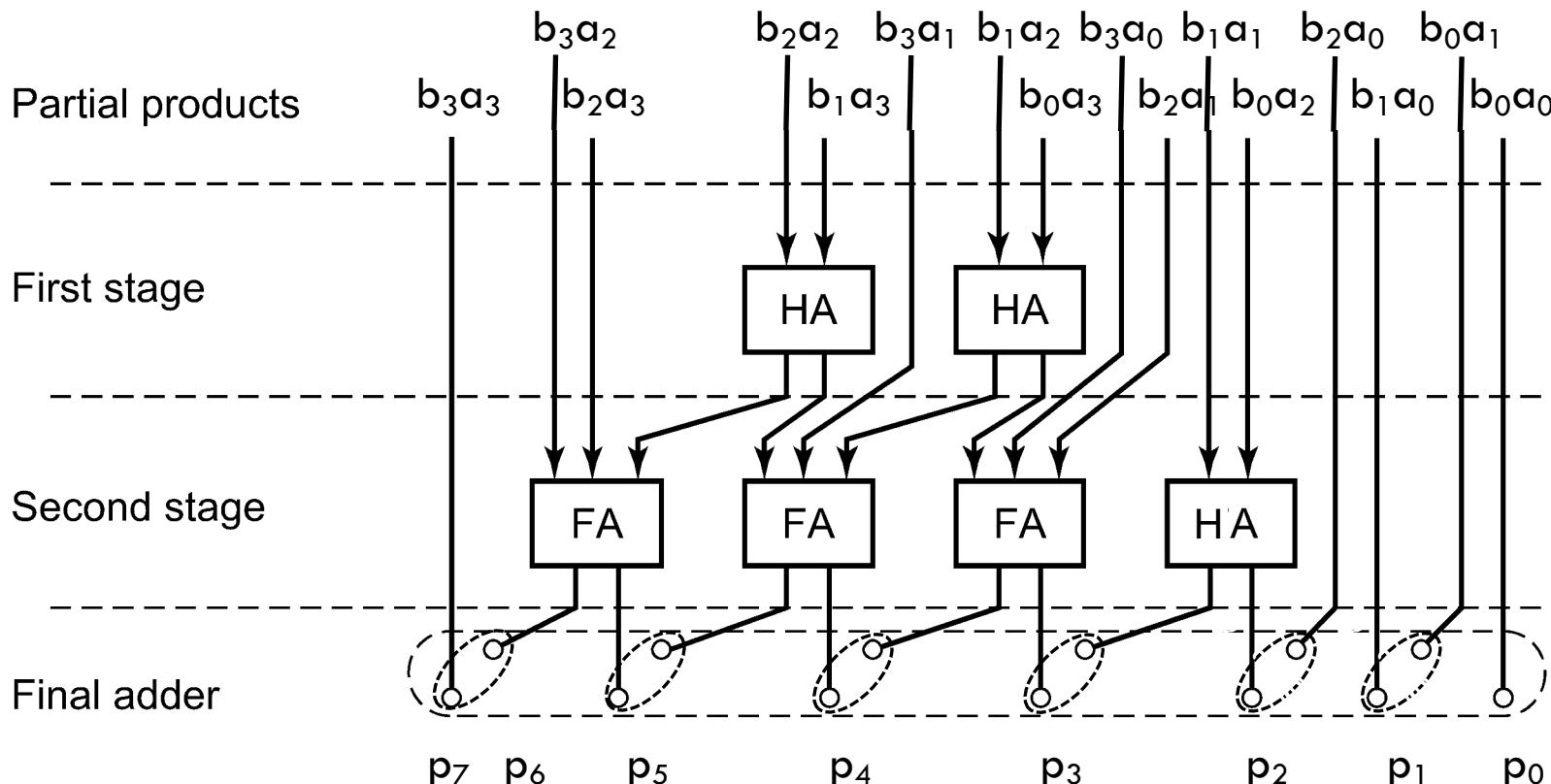
(c)

Final adder



(d)

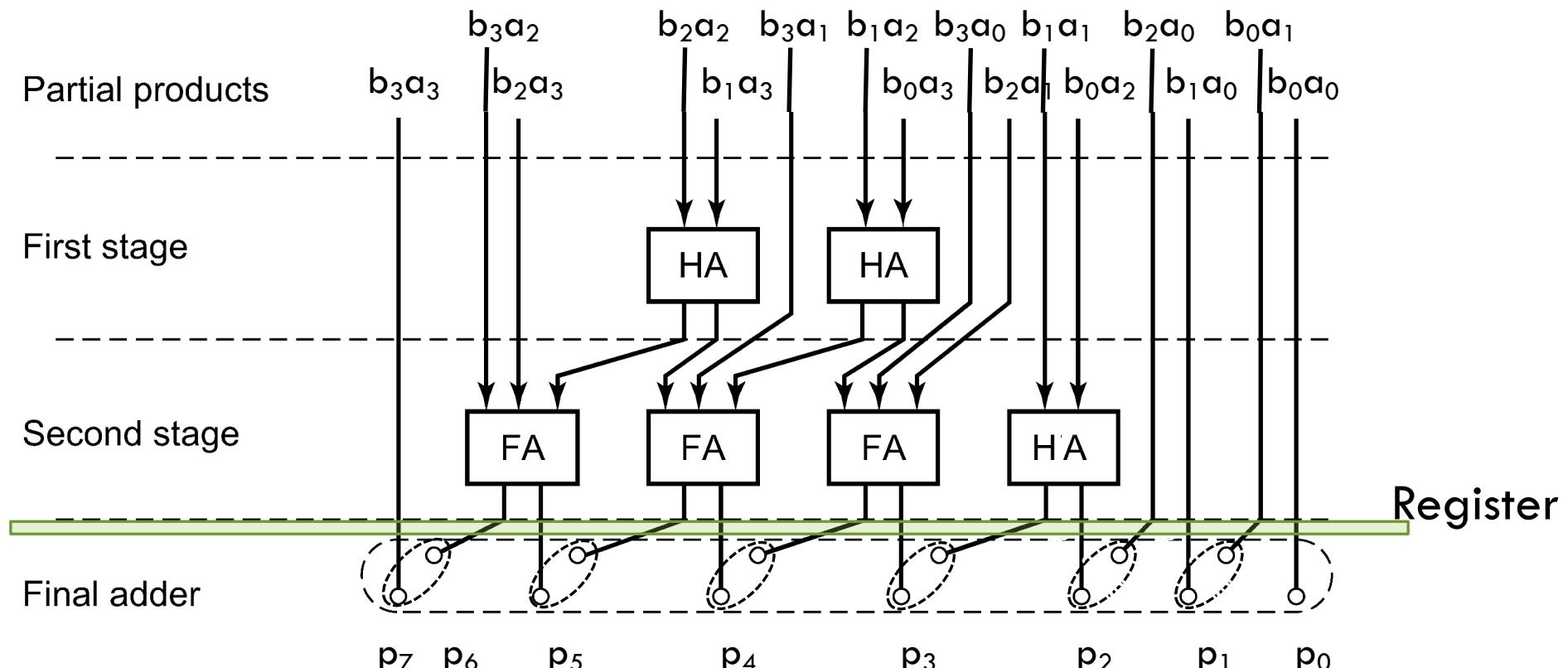
Wallace-Tree Multiplier



Note: Wallace tree is often slower than an array multiplier in FPGAs (which have optimized carry chains)

Increasing Throughput: Pipelining

- Multipliers have a long critical path: PP generation → reduction tree → final adder
 - Often pipelined before final adder (2x flip-flops for carry-save)





Booth Recoding

Booth Recoding: Motivation

$$\begin{array}{r} & a_{N-1} \dots & a_2 & a_1 & a_0 & \leftarrow & \text{Multiplicand} \\ \times & b_{N-1} \dots & b_2 & b_1 & b_0 & \leftarrow & \text{Multiplier} \\ \hline & a_{N-1}b_0 \dots & a_2b_0 & a_1b_0 & a_0b_0 & \} & \\ & a_{N-1}b_1 \dots & a_2b_1 & a_1b_1 & a_0b_1 & & \\ & a_{N-1}b_2 \dots & a_2b_2 & a_1b_2 & a_0b_2 & & \\ a_{N-1}b_3 \dots & a_2b_3 & a_1b_3 & a_0b_3 & & & \\ \hline & \dots & & a_1b_0 + a_0b_1 & a_0b_0 & \leftarrow & \text{Product} \end{array}$$

N partial products ($\times \{0, 1\}$)

How many non-zero partial products (out of N)?

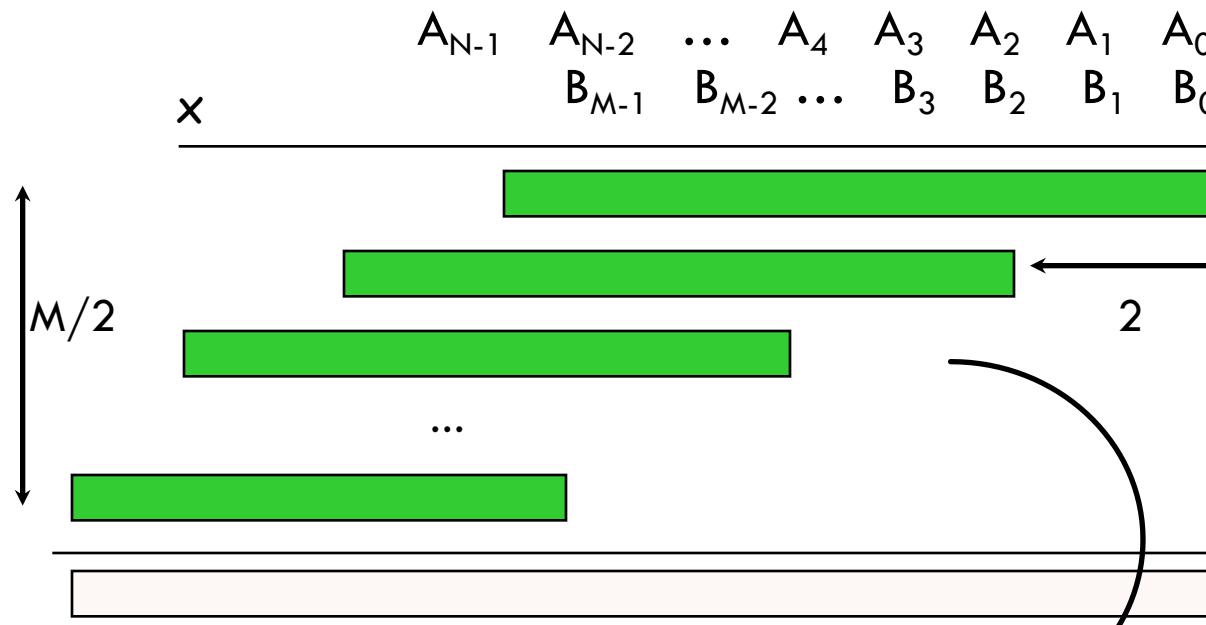
- N, if B = 000...0
- 0, if B = 111...1
- N/2 on the average

Booth Recoding: Main Idea

- Encode ...0111100... patterns:
 - $1111 = 2^3 + 2^2 + 2^1 + 2^0 = 2^4 - 2^0$
 - Only two non-zero numbers, but needs to represent +1 and -1
- Encoding method:
 - Encode pairs of bits, by looking at a window of three bits, from LSB
 - 000 is a middle of string of 0's
 - 001, 011 are the beginning of a string of 1's
 - 010 is an isolated 1
 - 100, 110 are ends of a string of 1's
 - 101 is the end of one string of 1's and the beginning of the next
 - 111 is the middle of a string of 1's
 - Worst case: ...010101... - exactly a half of non-zero partial products

Booth Recoding: Higher-radix multiplier

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



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

(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

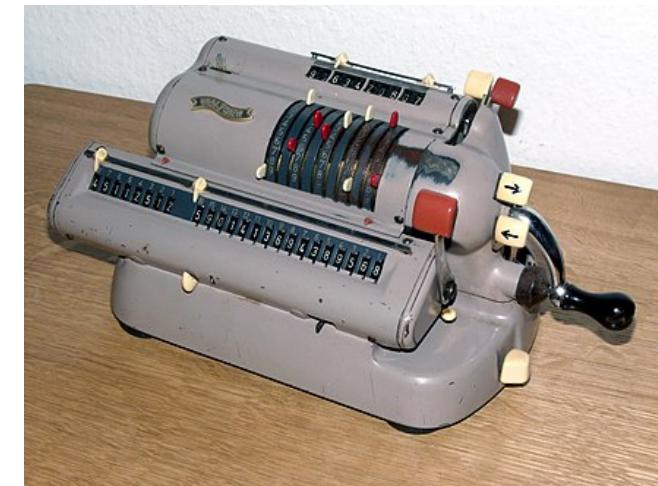
$$\begin{aligned} B_{K+1,K} * A &= 0 * A \rightarrow 0 \\ &= 1 * A \rightarrow A \\ &= 2 * A \rightarrow 2A \\ &= 3 * A \rightarrow 4A - A \end{aligned}$$

Example

- Compression tree needs to support subtraction

$$\begin{array}{r} 0111 \\ \times \quad 1010 \\ \hline -01110 \\ -00111 \\ +0111 \\ \hline 01000110 \end{array}$$

A	B
10 (0)	-2A
101	-A
001	+A



A Walther WSR160 arithmometer
(from Wikipedia)

Booth Recoding Notes

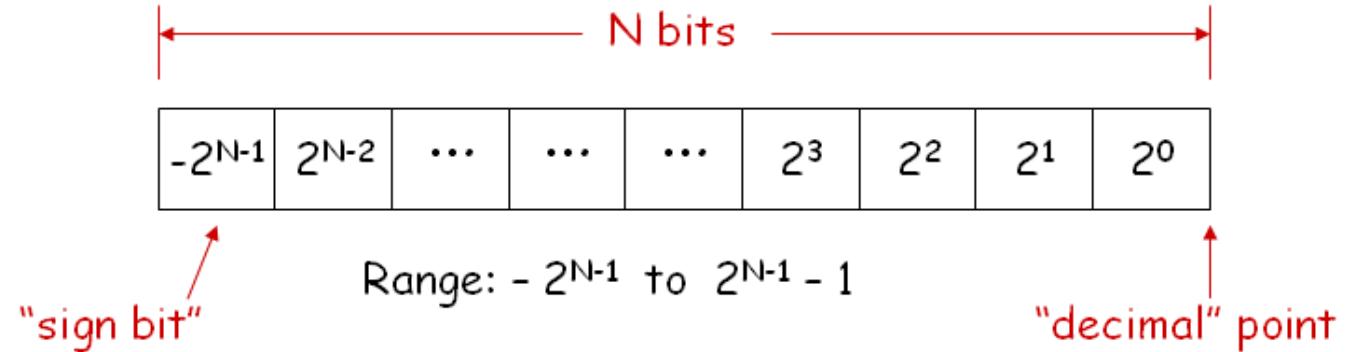
- Key advantage: Reduces the number of partial products
 - Compression tree depth becomes $\log_{3/2}[N/2]$
 - Partial product generation is slightly more complex than a NAND2
- Useful for larger multipliers
 - And some very creative solutions for repeated multiplications (FIR filters, etc)



Signed Multipliers

Signed Array Multiplier

- Two's complement



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

(-3)
(-2)

(+6)

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

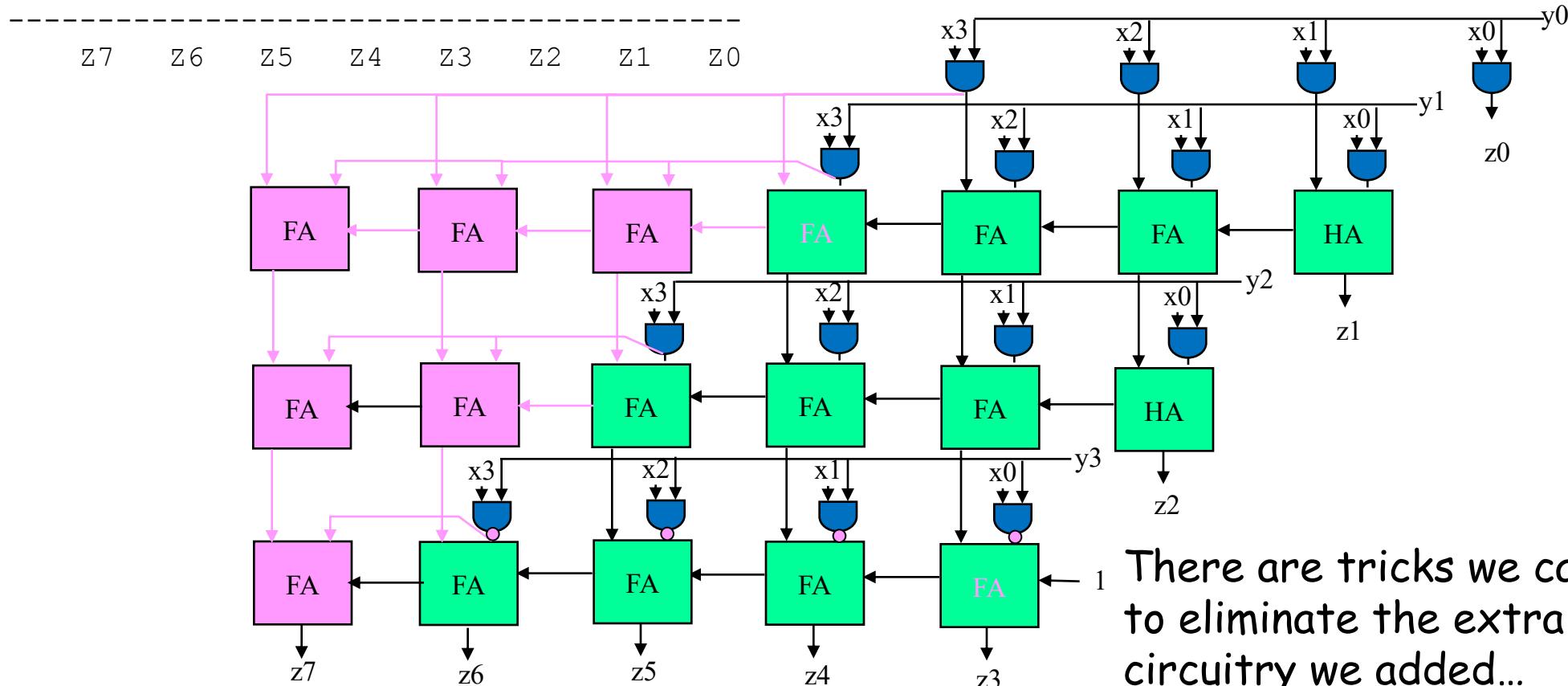
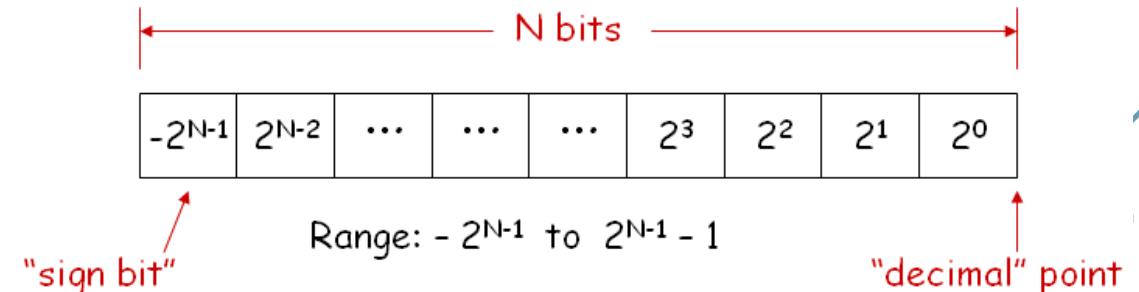
$$\begin{array}{r} 0 \ 0 \ 0 \ 0 \ 0 \ 0 \\ + 1 \ 1 \ 1 \ 0 \ 1 \\ - 1 \ 1 \ 0 \ 1 \end{array}$$

0 0 0 1 1 0

$$\begin{aligned} Y_0 * X &= 0 \\ 2Y_1 * X &= -6 \\ 4Y_2 * X &= -12 \end{aligned}$$

Combinational Multiplier (signed)

$$\begin{array}{r}
 \begin{array}{ccccccc}
 & \text{x3} & \text{x2} & \text{x1} & \text{x0} \\
 * & \text{y3} & \text{y2} & \text{y1} & \text{y0} \\
 \hline
 \end{array} \\
 \begin{array}{ccccccccc}
 \text{x3y0} & \text{x3y0} & \text{x3y0} & \text{x3y0} & \text{x3y0} & \text{x2y0} & \text{x1y0} & \text{x0y0} \\
 + \text{x3y1} & \text{x3y1} & \text{x3y1} & \text{x3y1} & \text{x2y1} & \text{x1y1} & \text{x0y1} \\
 + \text{x3y2} & \text{x3y2} & \text{x3y2} & \text{x2y2} & \text{x1y2} & \text{x0y2} \\
 - \text{x3y3} & \text{x3y3} & \text{x2y3} & \text{x1y3} & \text{x0y3}
 \end{array}
 \end{array}$$



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

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
<hr/>				
X3Y0	X3Y0	X3Y0	X3Y0	X3Y0
+ X3Y1	X3Y1	X3Y1	X3Y1	X2Y1
+ X3Y2	X3Y2	X3Y2	X2Y2	X1Y2
- X3Y3	X3Y3	X2Y3	X1Y3	X0Y3

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).

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

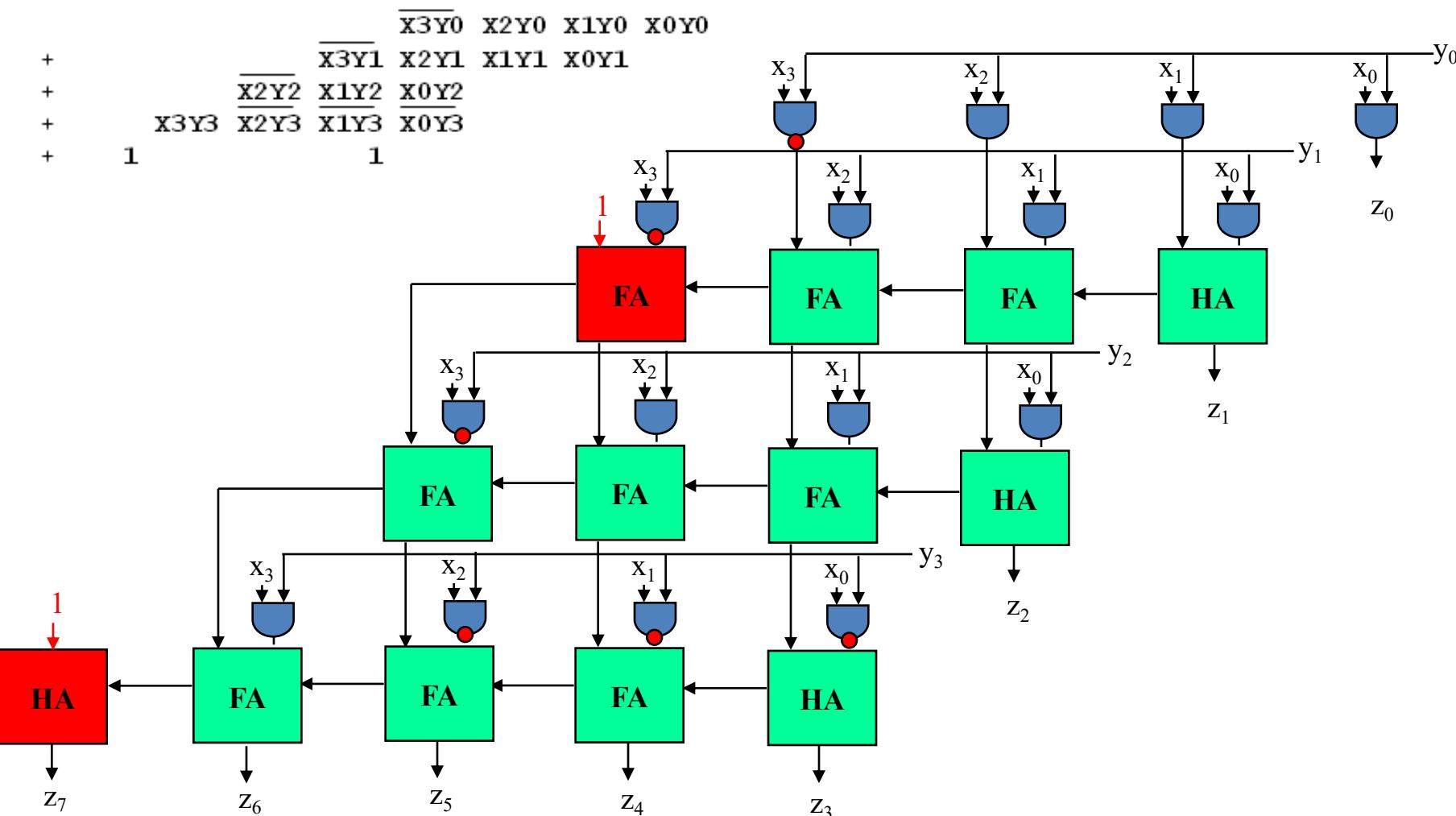
$$\begin{array}{r}
 & & \underline{\underline{x3y0}} & x2y0 & x1y0 & x0y0 \\
 + & & \underline{\underline{x3y1}} & x2y1 & x1y1 & x0y1 \\
 + & & \underline{\underline{x2y2}} & x1y2 & x0y2 & \\
 + & \underline{\underline{x3y3}} & \underline{\underline{x2y3}} & \underline{\underline{x1y3}} & \underline{\underline{x0y3}} & \\
 + & & & & & 1 \\
 - & 1 & 1 & 1 & 1 &
 \end{array}$$

Step 4: finish computing the constants...

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

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

2's Complement Multiplication (Baugh-Wooley)



Example

- What's -3×-5 ?

$$\begin{array}{r} 1101 \\ \times 1011 \\ \hline \end{array}$$

Page

31

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`

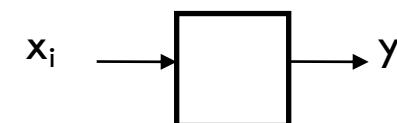


Multiplication with a Constant

Constant Multiplication

- Our multiplier circuits so far has assumed both the multiplicand (A) and the multiplier (B) can vary at runtime.
 - What if one of the two is a constant?
- $$Y = C * X$$
- “Constant Coefficient” multiplication comes up often in signal processing. Ex:

$$y_i = \alpha y_{i-1} + x_i$$

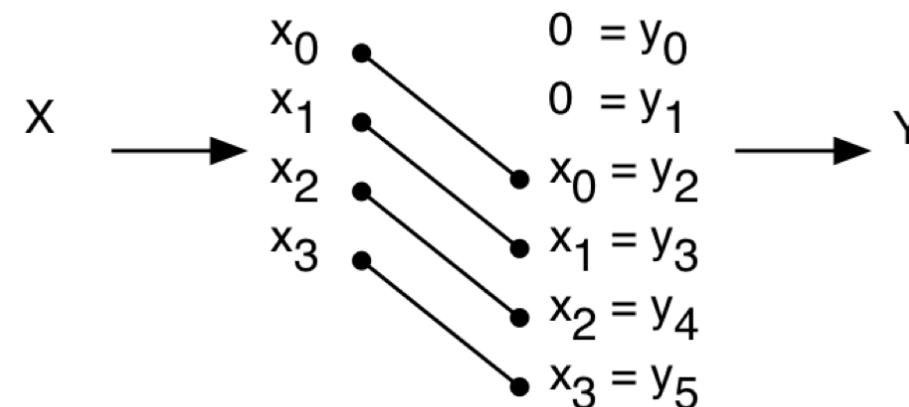


where α is an application-dependent constant that is hard-wired into the circuit.

- How do we build an array style (combinational) multiplier that takes advantage of a constant operand?

Multiplication by a Constant

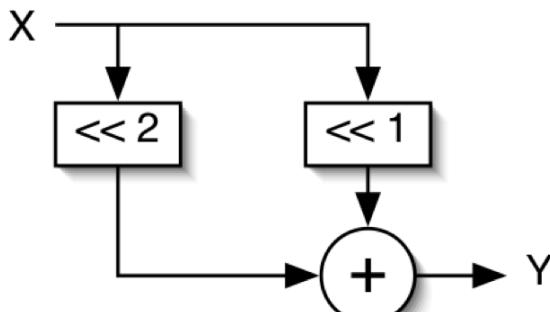
- If the constant C in $C*X$ is a power of 2, then the multiplication is simply a shift of X.
- Ex: $4*X$



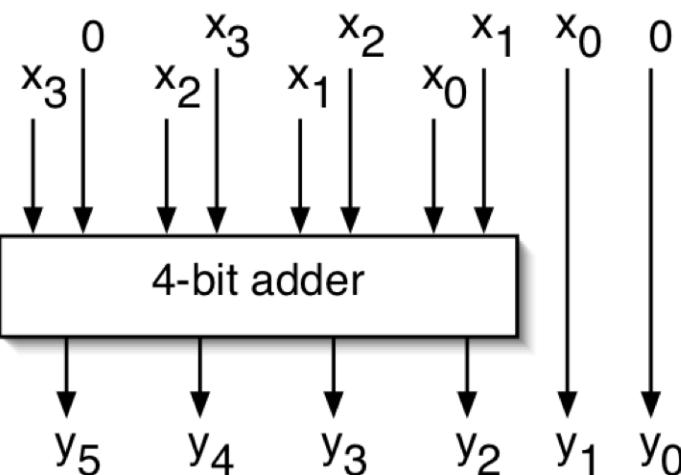
- What about division?
- What about multiplication by non-powers of 2?

Multiplication by a Constant

- In general, a combination of fixed shifts and addition:
 - Ex: $6 \times X = 0110 \times X = (2^2 + 2^1) \times X = 2^2 X + 2^1 X$

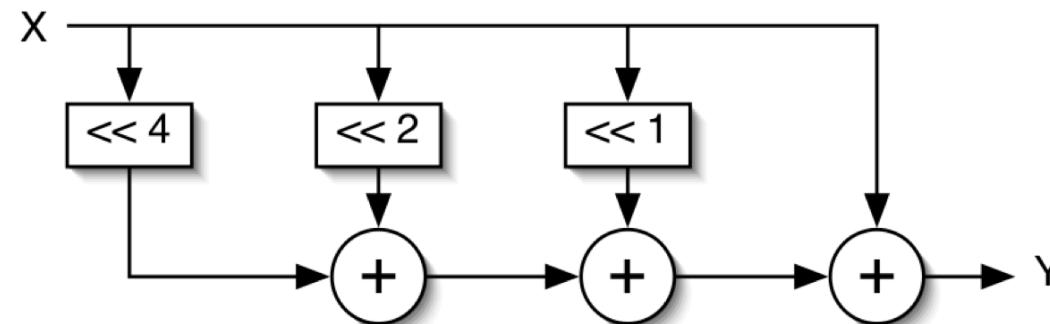


- Details:



Multiplication by a Constant

- Another example: $C = 23_{10} = 010111$



- In general, the number of additions equals one less than the number of 1's in the constant.*
- Using carry-save adders (for all but one of these) helps reduce the delay and cost, and using trees helps with delay, but the number of adders is still the number of 1's in C minus 2.
- Is there a way to further reduce the number of adders (and thus the cost and delay)?

Multiplication using Subtraction

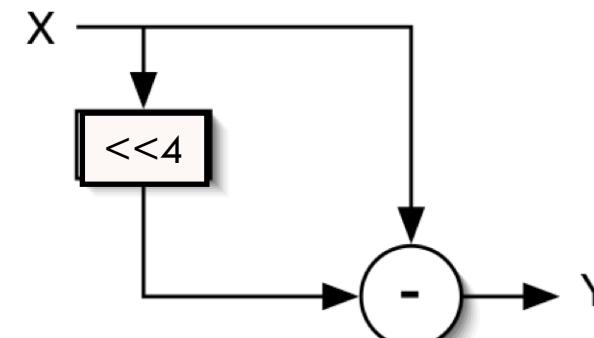
- Subtraction is the same cost and delay as addition.
- Consider $C*X$ where C is the constant value $15_{10} = 01111$.
 $C*X$ requires 3 additions.
- We can “recode” 15

$$\text{from } 01111 = (2^3 + 2^2 + 2^1 + 2^0)$$

$$\text{to } 10001 = (2^4 - \bar{2}^0)$$

where $\bar{1}$ means negative weight.

- Therefore, $15*X$ can be implemented with only one subtractor.
 - Remember Booth encoding



Canonic Signed Digit Representation

- CSD represents numbers using 1, $\bar{1}$, & 0 with the least possible number of non-zero digits.
 - Strings of 2 or more non-zero digits are replaced with a 1000.. $\bar{1}$.
 - Leads to a unique representation.
- To form CSD representation might take 2 passes:
 - First pass: replace all occurrences of 2 or more 1's:

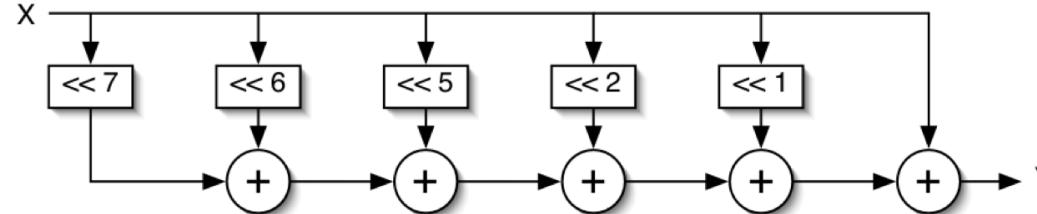
01..10 by $10..\bar{1}0$

- Second pass: same as above, plus replace $01\bar{1}0$ with 0010 and $0\bar{1}10$ with $00\bar{1}0$
- Examples:

$011101 = 29$	$0010111 = 23$	$0110110 = 54$
$100\bar{1}01 = 32 - 4 + 1$	0011001 $010\bar{1}00\bar{1} = 32 - 8 - 1$	$10\bar{1}10\bar{1}0$ $100\bar{1}0\bar{1}0 = 64 - 8 - 2$
- Can we further simplify the multiplier circuits?

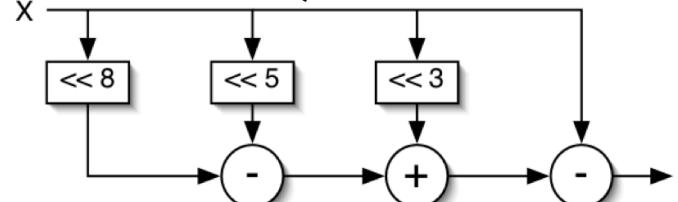
(K) Constant Coefficient Multiplication (KCM)

Binary multiplier: $Y = 231 \cdot X = (2^7 + 2^6 + 2^5 + 2^2 + 2^1 + 2^0) \cdot X$



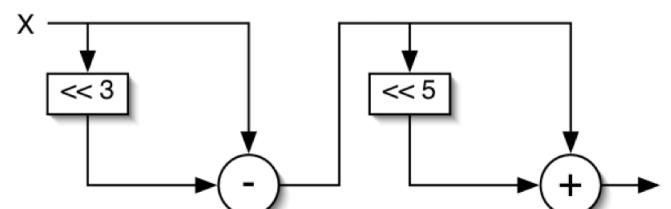
- CSD helps, but the multipliers are limited to shifts followed by adds.

- CSD multiplier: $Y = 231 \cdot X = (2^8 - 2^5 + 2^3 - 2^0) \cdot X$



- How about shift/add/shift/add ...?

- KCM multiplier: $Y = 231 \cdot X = 7 \cdot 33 \cdot X = (2^3 - 2^0) \cdot (2^5 + 2^0) \cdot X$



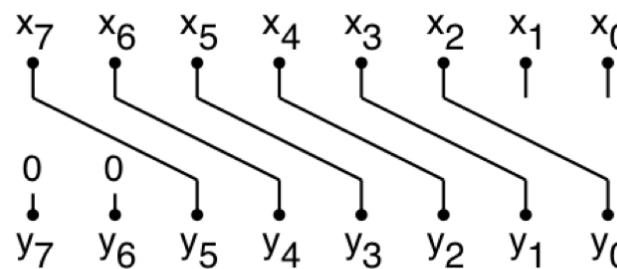
- No simple algorithm exists to determine the optimal KCM representation.
- Most use exhaustive search method.

<http://www.andraka.com/multipli.php>

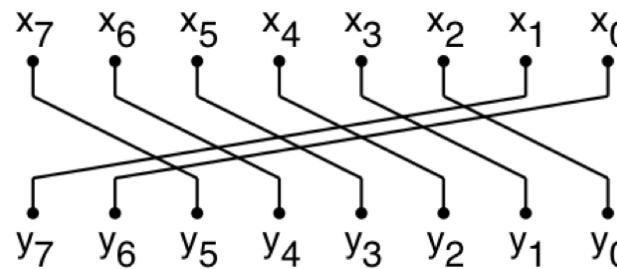


Shifters and Rotators

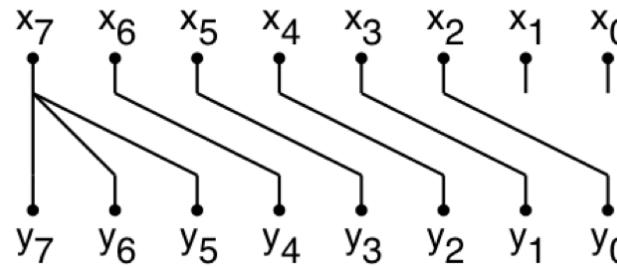
Fixed Shifters / Rotators Defined



Logical
Shift



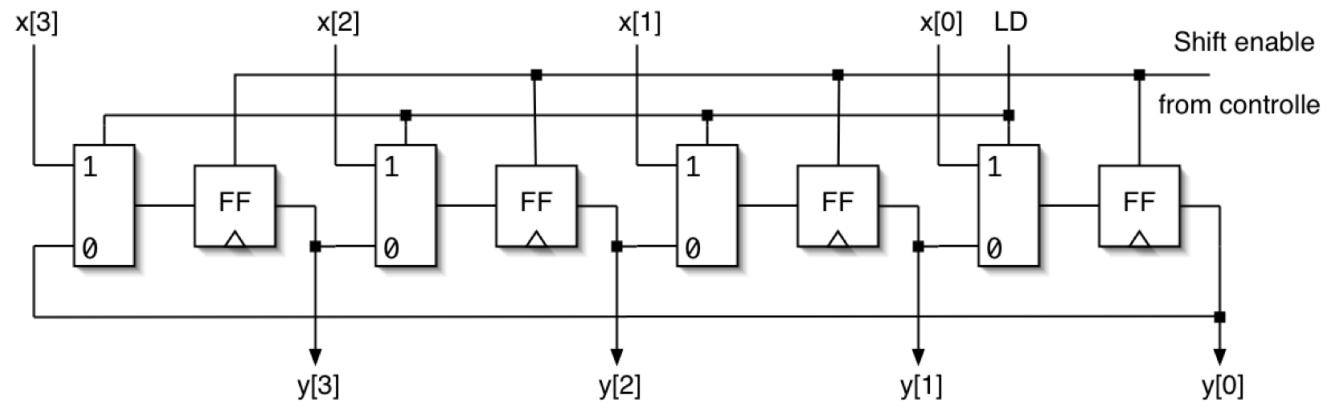
Rotate



Arithmetic
Shift

Variable Shifters / Rotators

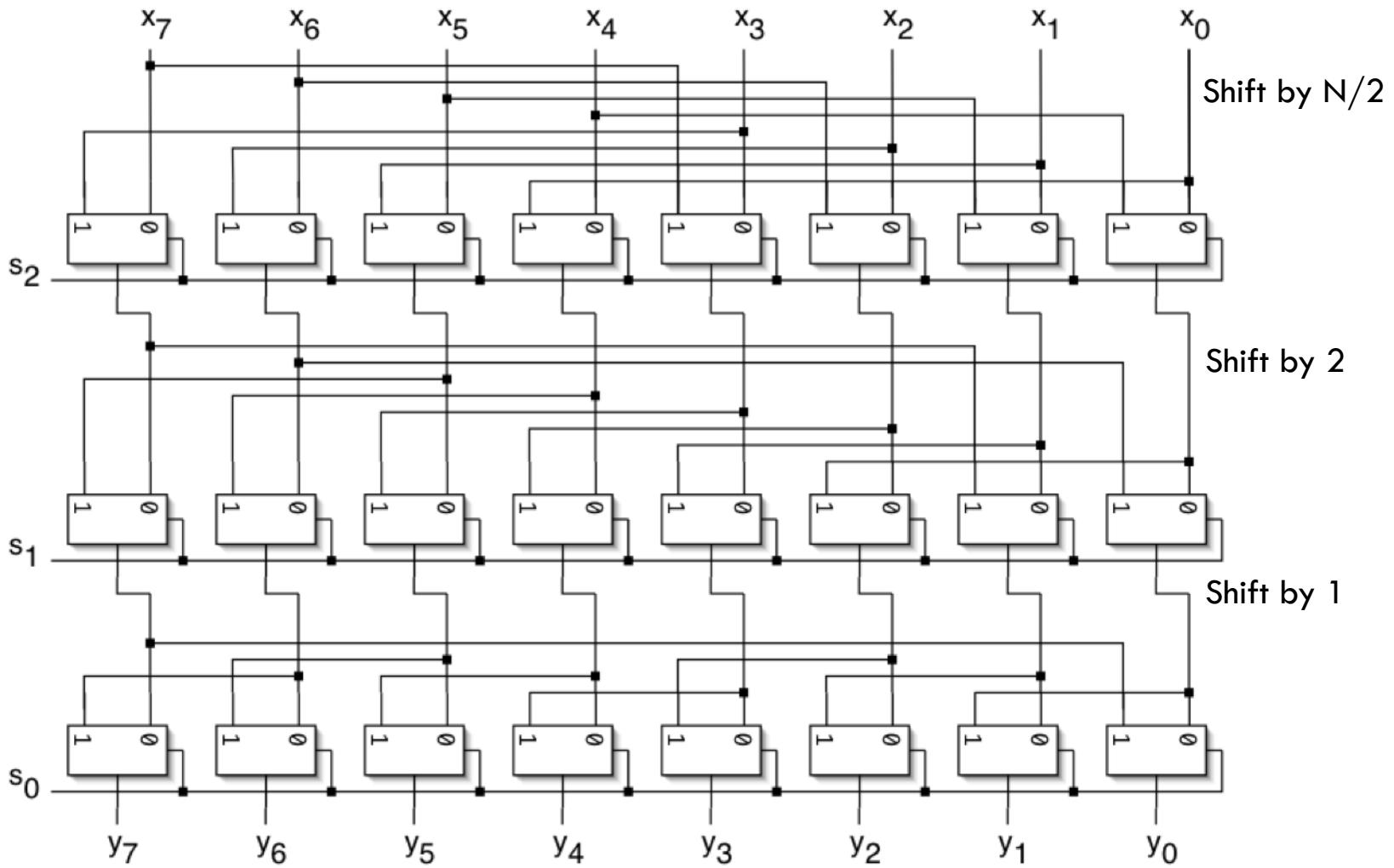
- Example: $X >> S$, where S is unknown at design time.
- Uses: Shift instruction in processors, floating-point arithmetic, division/multiplication by powers of 2, etc.
- One way to build this is a simple shift-register:
 - a) Load word,
 - b) shift enable for S cycles,
 - c) read word.



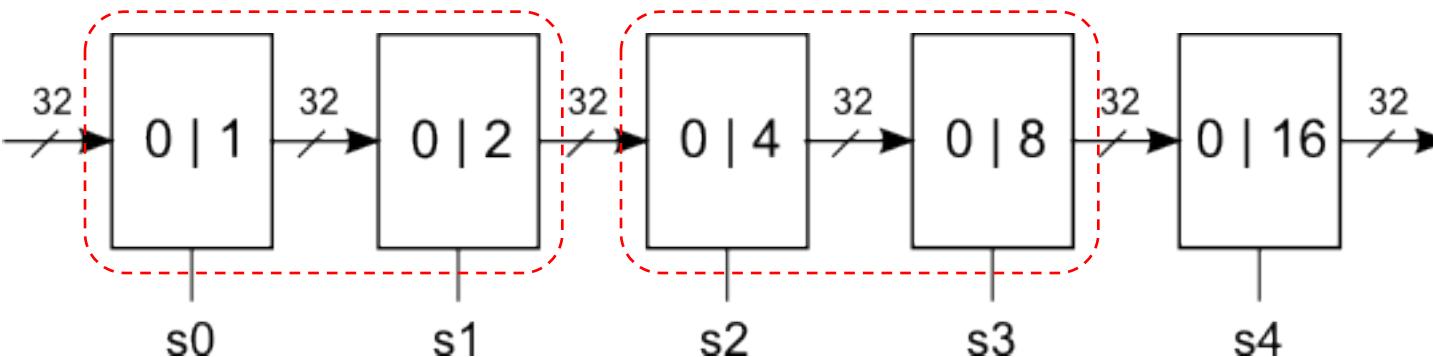
- Worst case delay $O(N)$, not good for processor design.
- Can we do it in $O(\log N)$ time and fit it in one cycle?

Log Shifter / Rotator

- Log(N) stages, each shifts (or not) by a power of 2 places, $S=[s_2; s_1; s_0]$:



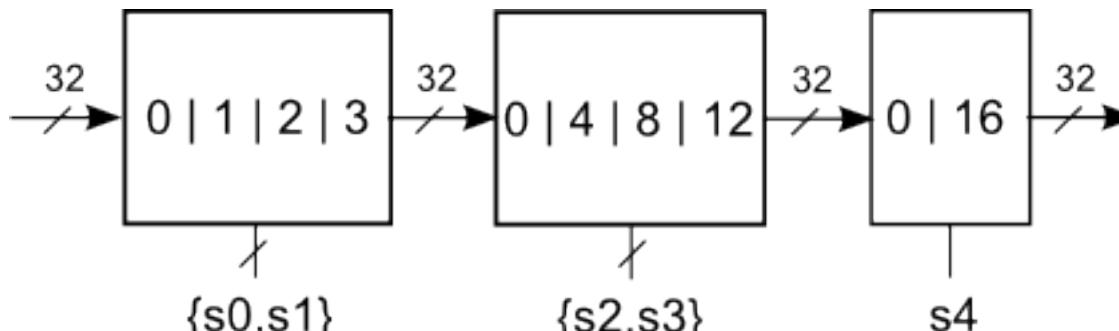
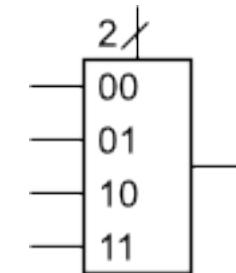
LUT Mapping of Log shifter



Efficient with 2-to-1 multiplexers, for instance, 3LUTs.

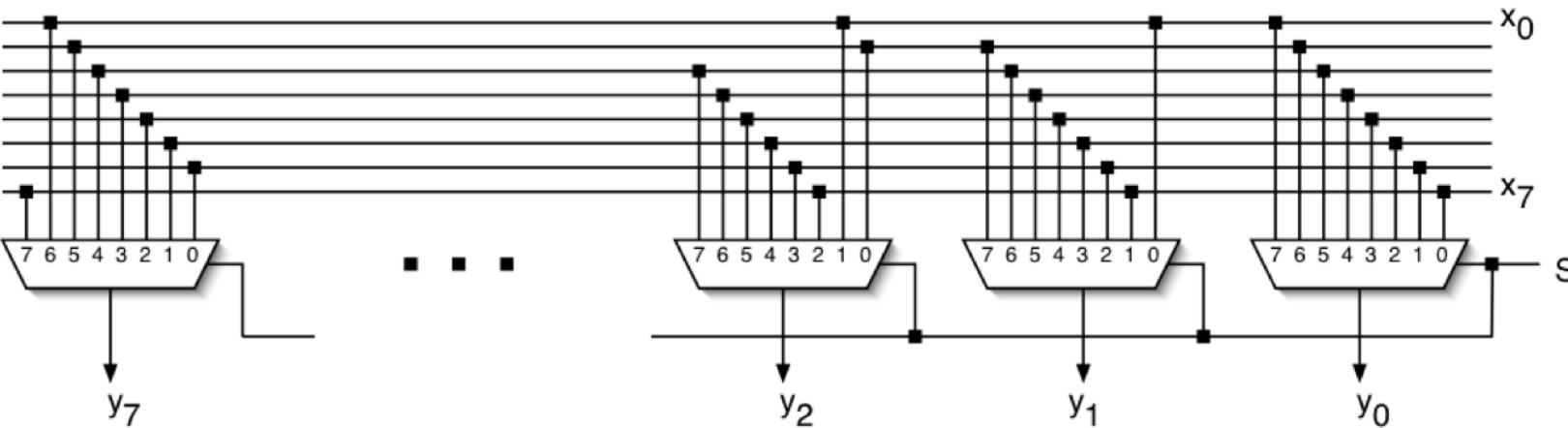
Virtex6 has 6LUTs. Naturally makes 4-to-1 muxes:

Reorganize shifter to use 4to1 muxes.



Final stage uses F7 mux

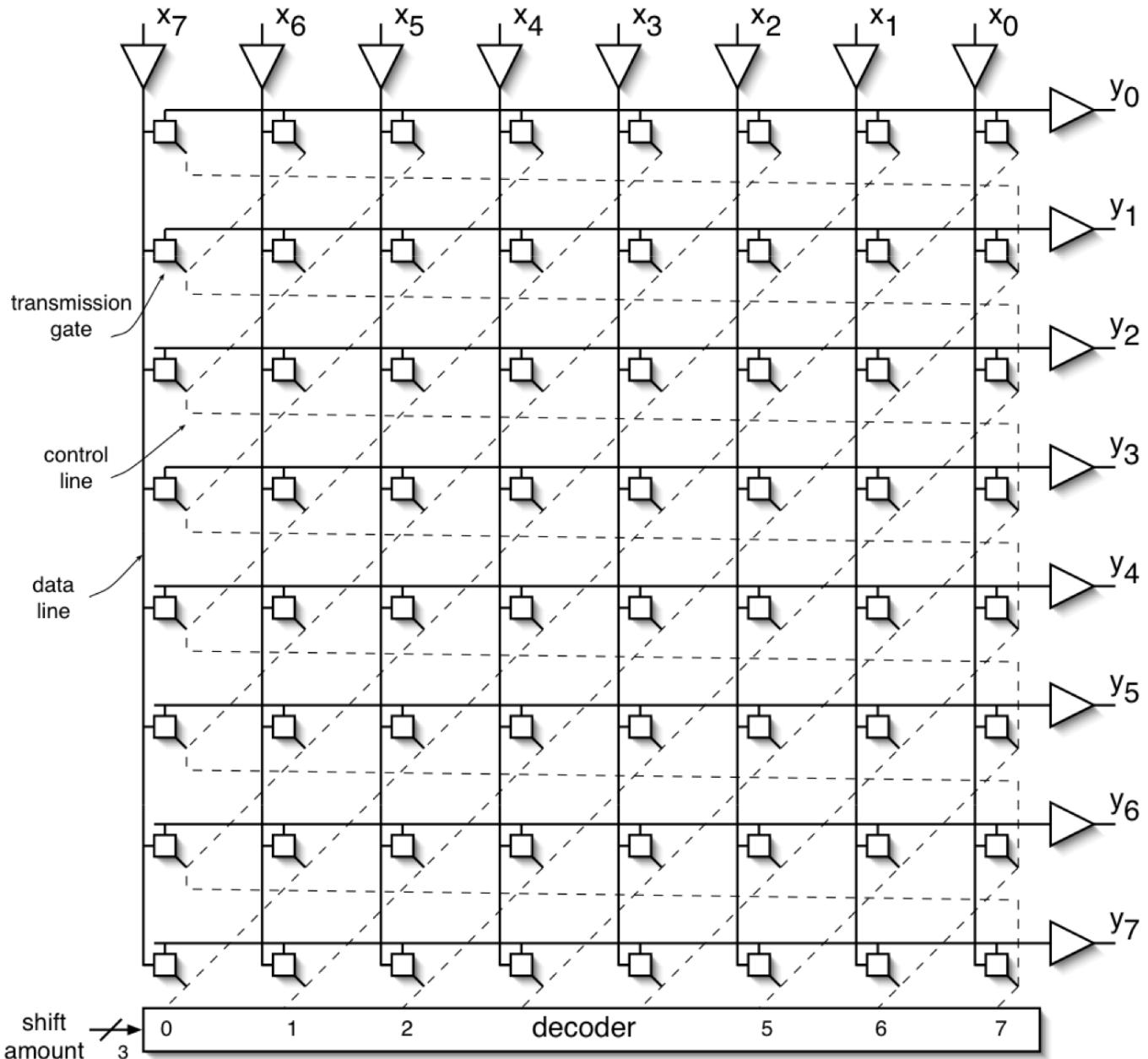
“Improved” Shifter / Rotator



- Requires careful (custom) circuit design:
 - High fanout on input signals $x_0 \dots x_7$
 - Large multiplexers are slow

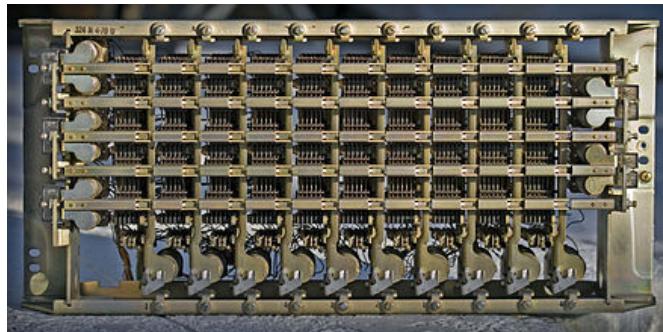
Barrel Shifter

- Cost/delay?
 - (don't forget the decoder)

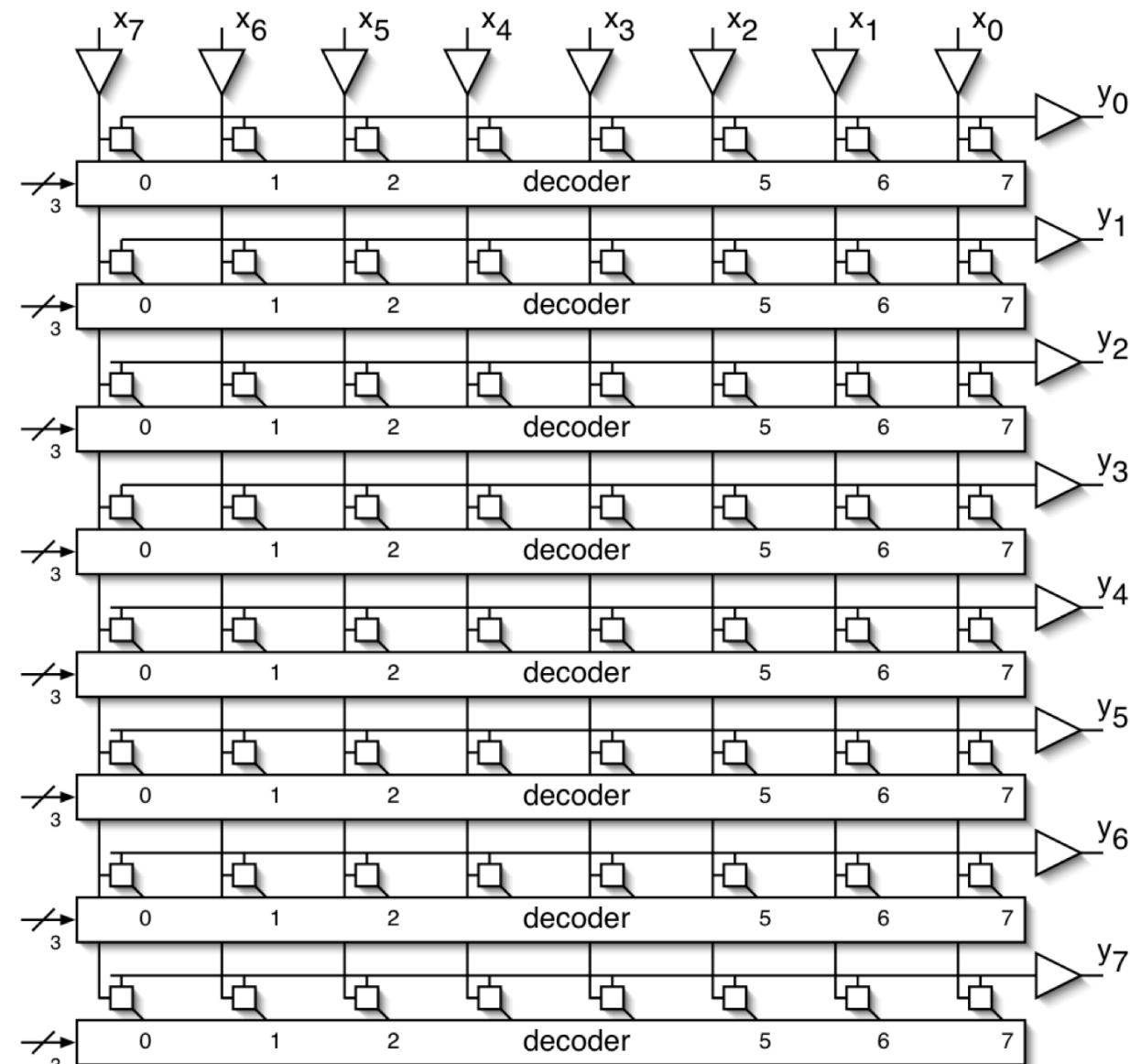


Crossbar Switch

- $N \log(N)$ control signals.
- Supports all interesting permutations
 - All one-to-one and one-to-many connections.
- Processors to memory/peripherals
- Communication hardware (switches, routers).



Western Electric 100-point six-wire Type B crossbar switch
(Wikipedia)



Review

- Binary multipliers have three blocks:
 - Partial-product generation (NAND or Booth)
 - Partial-product compression (ripple-carry array, CSA or Wallace)
 - Final adder
- Multipliers are often pipelined
- Constant multipliers can be optimized for size/speed
- Shifters and crossbars are common building blocks in digital systems
 - Often require customization