EECS151/251A
Spring 2021
Digital Design and Integrated Circuits

Instructor: John Wawrzynek

Lecture 23: Multiplier Circuits and Shifters

Announcements

- Virtual Front Row for today 4/15:
  - *Jose Rodriguez*
  - *Khashayar Pirouzmand*
  - *Jeremy Ferguson*
  - *Rahul Arya*
  - *Ian Mayle*
- **Please ask question or make comments!**
- Homework assignment 9 posted - to be due a week after posting.

Warmup

- Recall long multiplication of base-10 by hand:

```
  12
x 56
```

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

```
  011
x 101
```

Multiplication

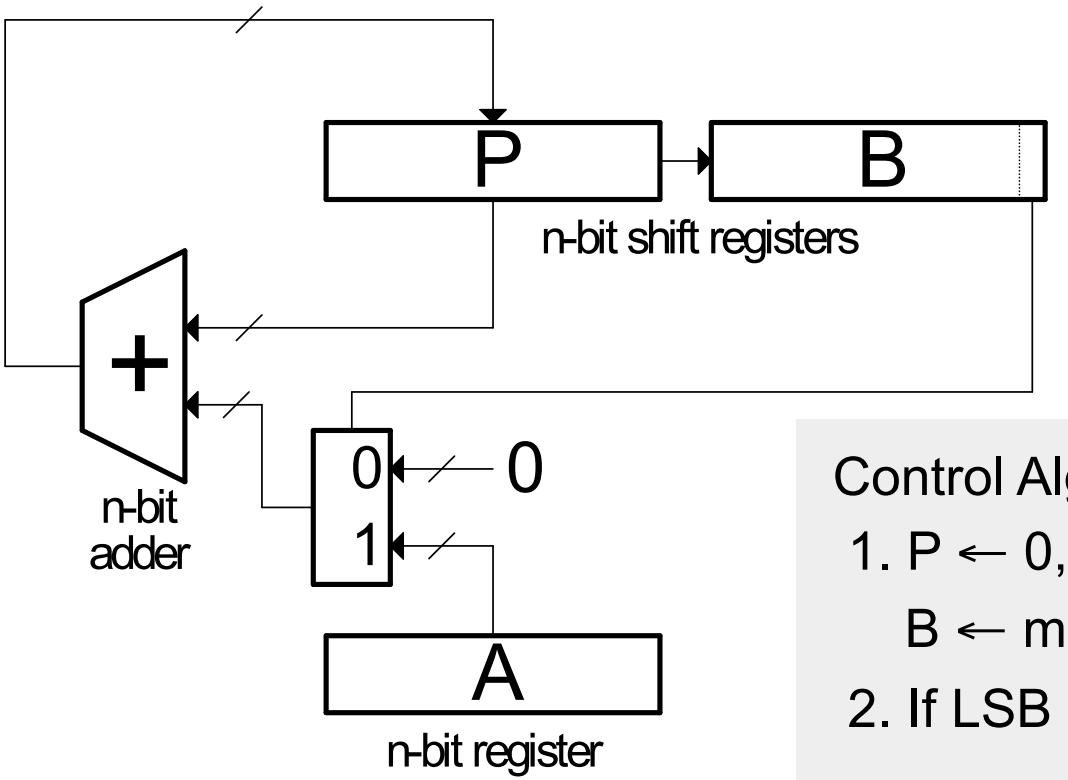
		a_3	a_2	a_1	a_0 ← *Multiplicand*
		b_3	b_2	b_1	b_0 ← *Multiplier*
	X	a_3b_0	a_2b_0	a_1b_0	a_0b_0
	a_3b_1	a_2b_1	a_1b_1	a_0b_1	
a_3b_2	a_2b_2	a_1b_2	a_0b_2		
a_3b_3	a_2b_3	a_1b_3	a_0b_3		

Partial products

. . .			$a_1b_0+a_0b_1$	a_0b_0	← *Product*

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.

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

Control Algorithm:

1. P ← 0, A ← multiplicand,
   B ← 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:

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

ex: $-6 = 11010_2 = 0 \cdot 2^0 + 1 \cdot 2^1 + 0 \cdot 2^2 + 1 \cdot 2^3 - 1 \cdot 2^4$

$= 0 + 2 + 0 + 8 - 16 = -6$

- 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

Convince yourself

- What's -3 x 5?

```
  1101
x 0101
```

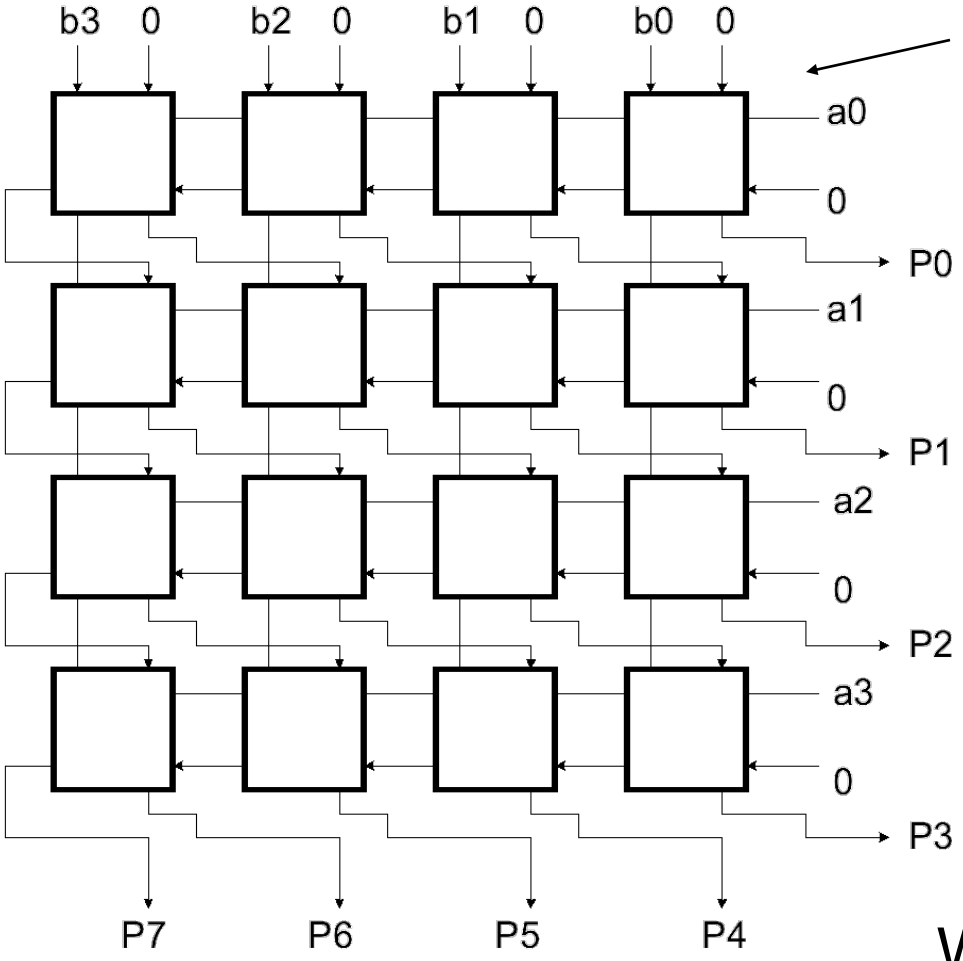
Outline for Multipliers

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

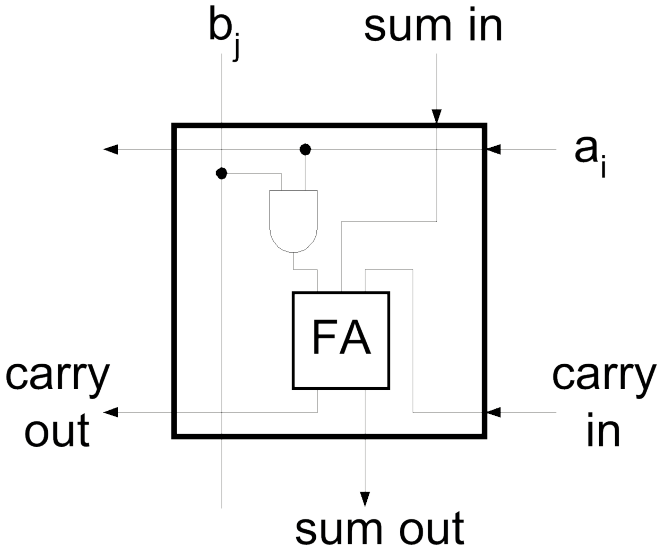
Unsigned Combinational Multiplier

Array Multiplier

Single cycle multiply: Generates all n partial products simultaneously.



Each row: n-bit adder with AND gates



What is the critical path?

Carry-Save Addition

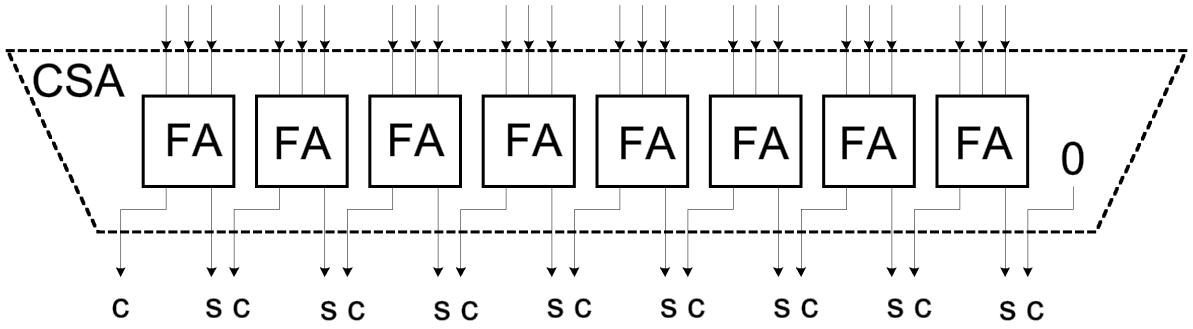
- Speeding up multiplication is a matter of speeding up the summing of the partial products.
- "Carry-save" addition can help.
- Carry-save addition passes (saves) the carries to the output, rather than propagating them.

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

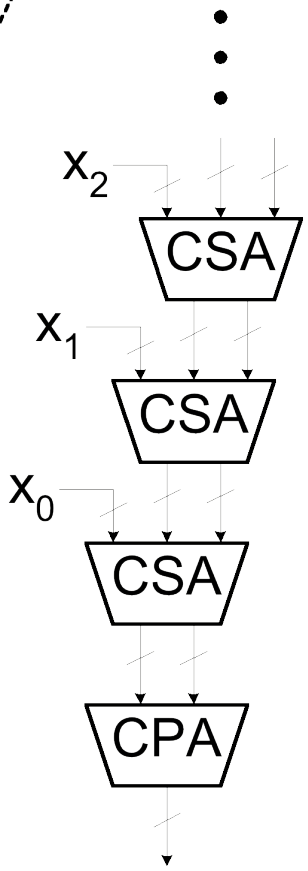
$$
\begin{array}{rll}
 & 3_{10}\ 0011 & \\
+ & 2_{10}\ 0010 & \\
\hline
 & c\ 0100 = 4_{10} & \text{carry-save add} \\
 & s\ 0001 = 1_{10} & \\
\hline
 & 3_{10}\ 0011 & \text{carry-save add} \\
 & c\ 0010 = 2_{10} & \text{carry-propagate add} \\
 & s\ 0110 = 6_{10} & \\
\hline
 & 1000 = 8_{10} &
\end{array}
$$

- In general, *carry-save* addition takes in 3 numbers and produces 2.
  - Sometimes called a "3:2 compressor": 3 input signals into 2
- Whereas, *carry-propagate* takes 2 and produces 1.
- With this technique, we can avoid carry propagation until final addition

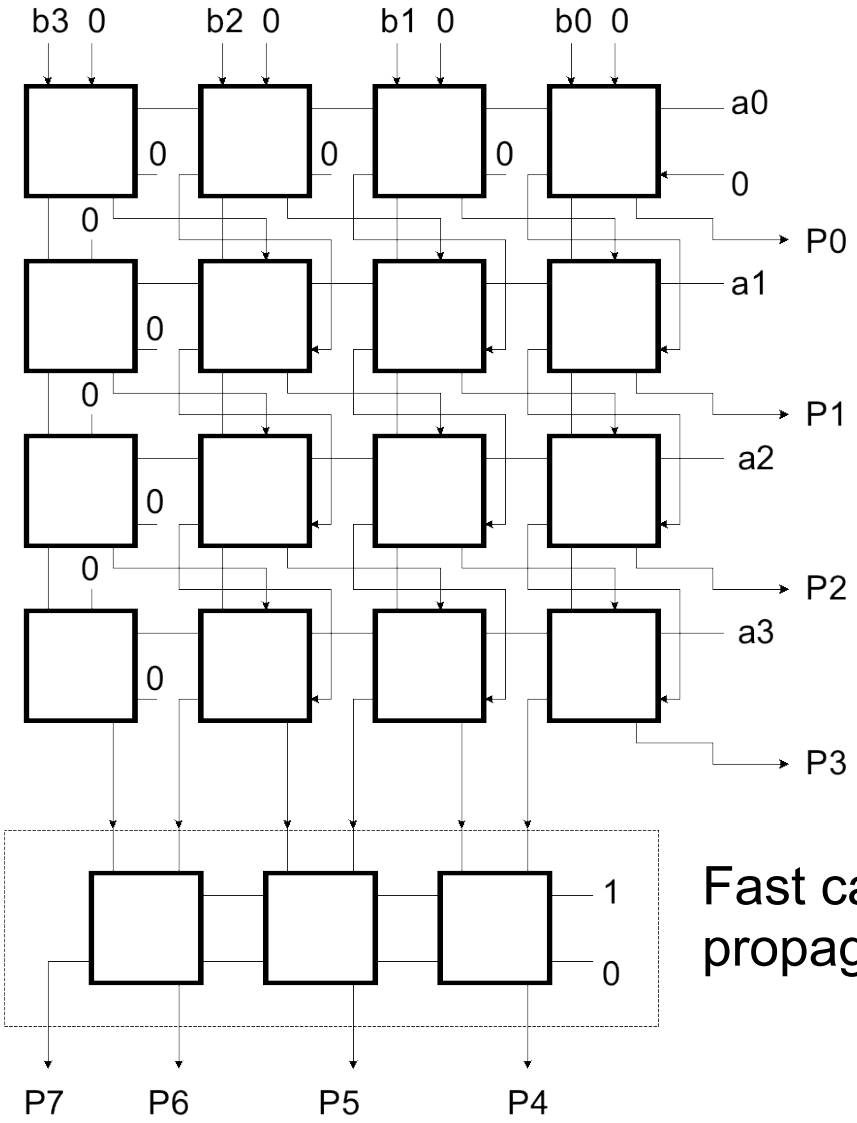
Carry-save Circuits

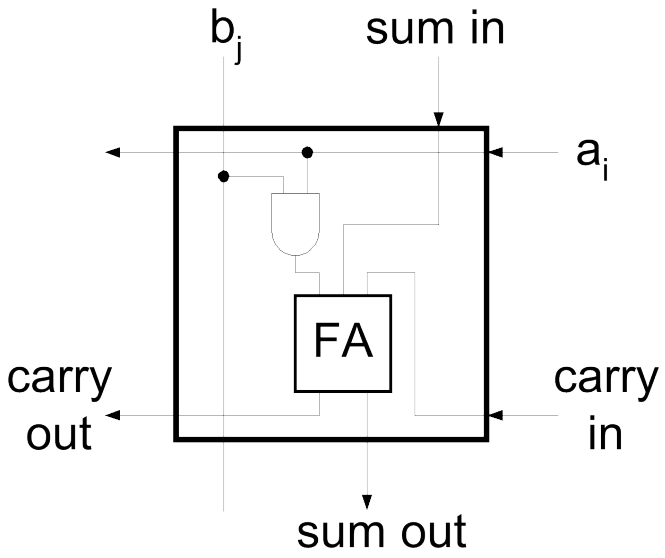


- When adding sets of numbers, carry-save can be used on all but the final sum.
- Standard adder (carry propagate) is used for final sum.
- Carry-save is fast (no carry propagation) and cheap (same cost as ripple adder)



Array Multiplier using Carry-save Addition



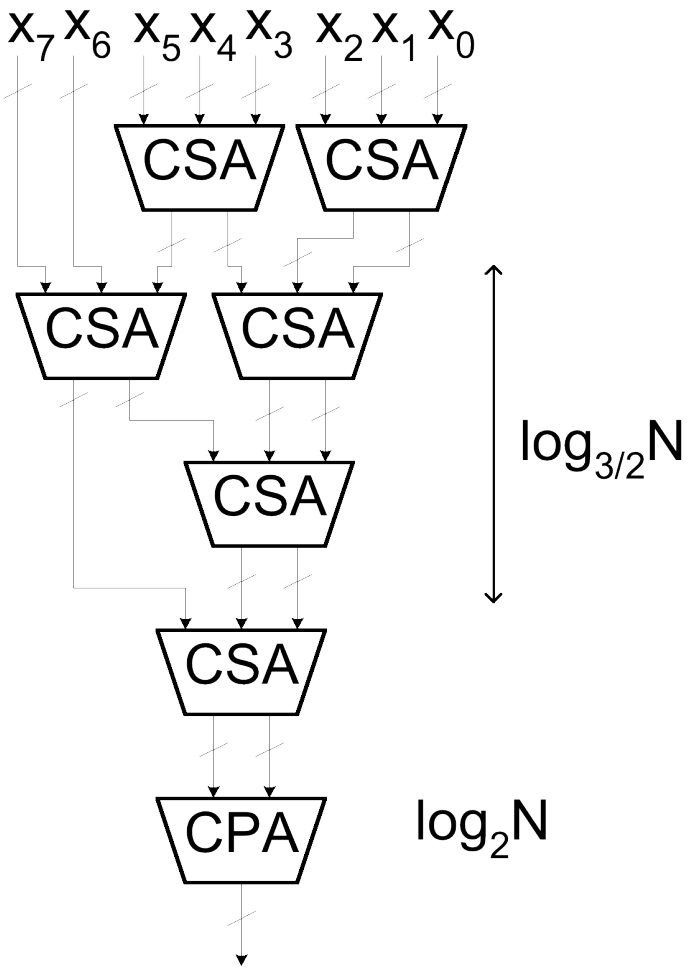


Fast carry-propagate adder

Carry-save Addition

CSA is associative and commutative. For example:

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

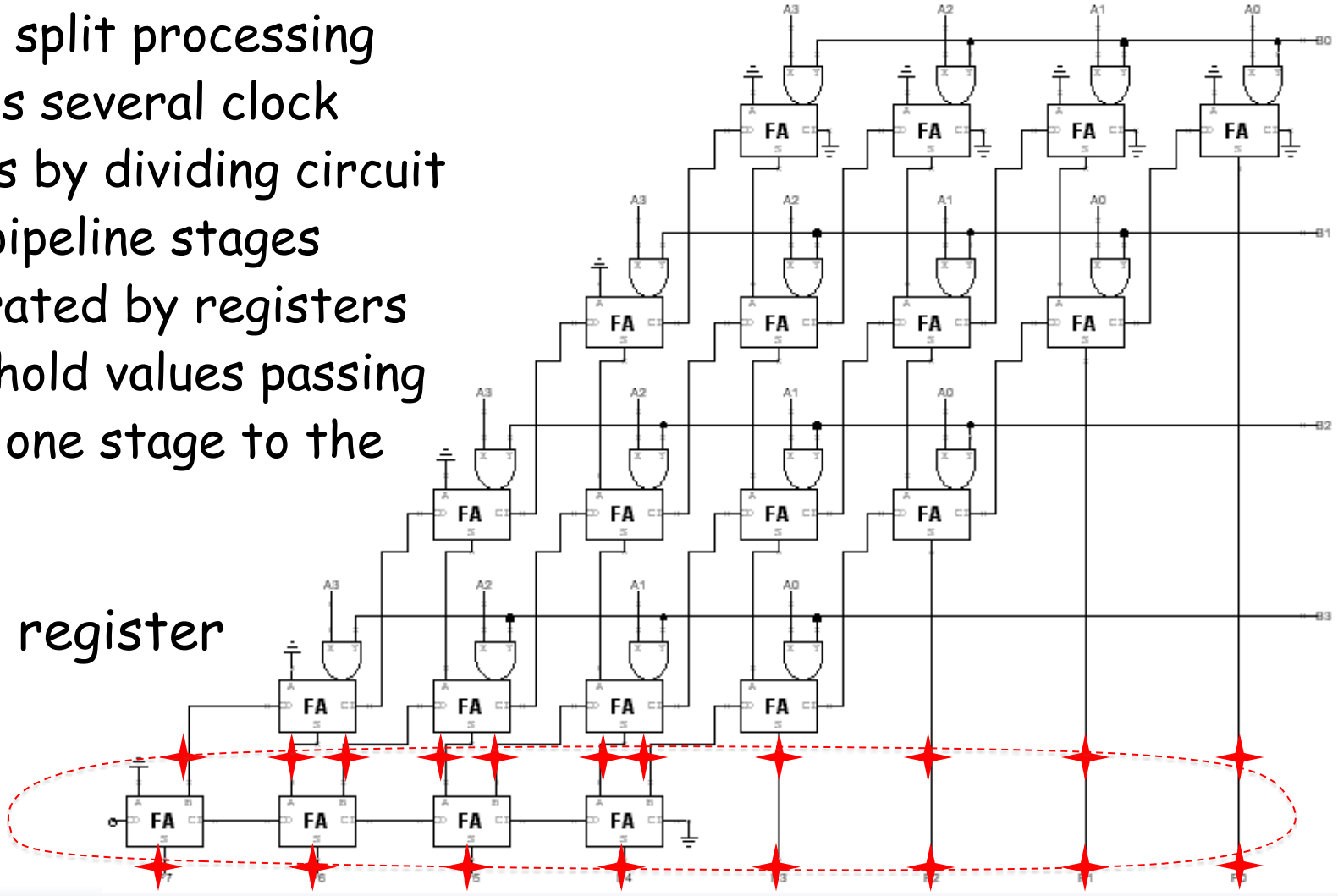


- A balanced tree can be used to reduce the logic delay.
- It doesn't matter where you add the carries and sums, as long as you eventually do add them.
- This structure is the basis of the ***Wallace Tree Multiplier***.
- Partial products are summed with the CSA tree. Fast CPA (ex: CLA) is used for final sum.
- Multiplier delay α $\log_{3/2}N$ + $\log_2 N$

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.

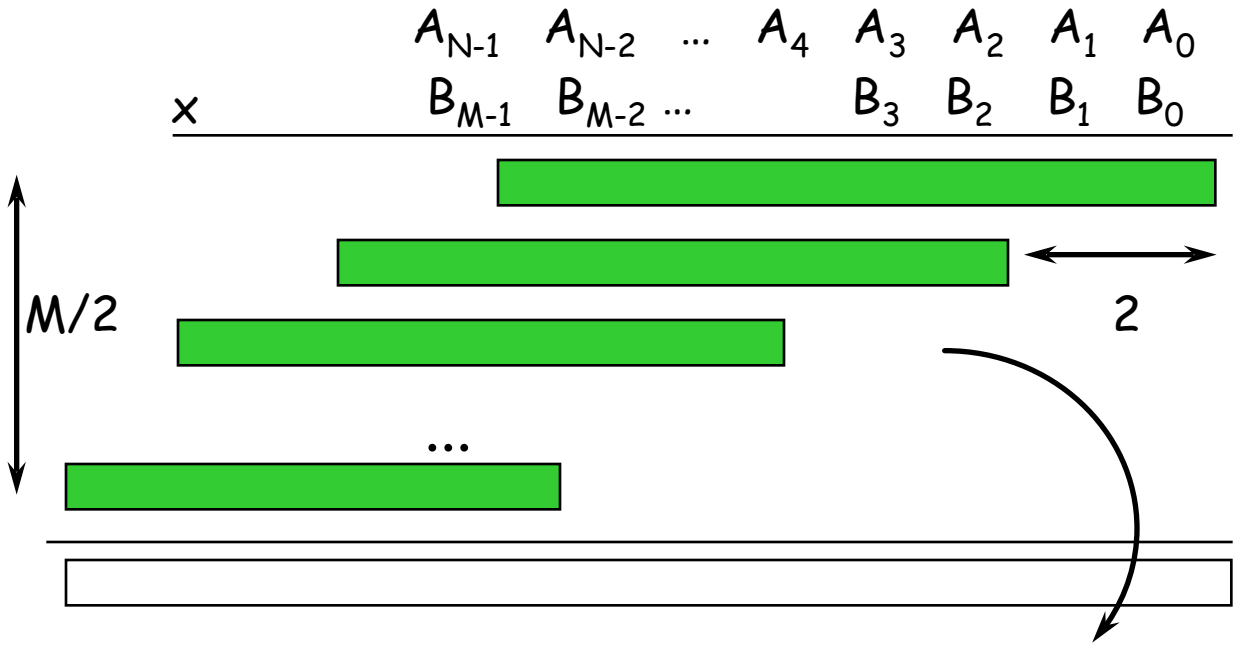
✦ = register



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 rows and halve the latency of the multiplier!



$B_{K+1,K}*A = 0*A \rightarrow 0$

$= 1*A \rightarrow A$

$= 2*A \rightarrow 4A - 2A$

$= 3*A \rightarrow 4A - A$

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

Booth recoding

current bit pair

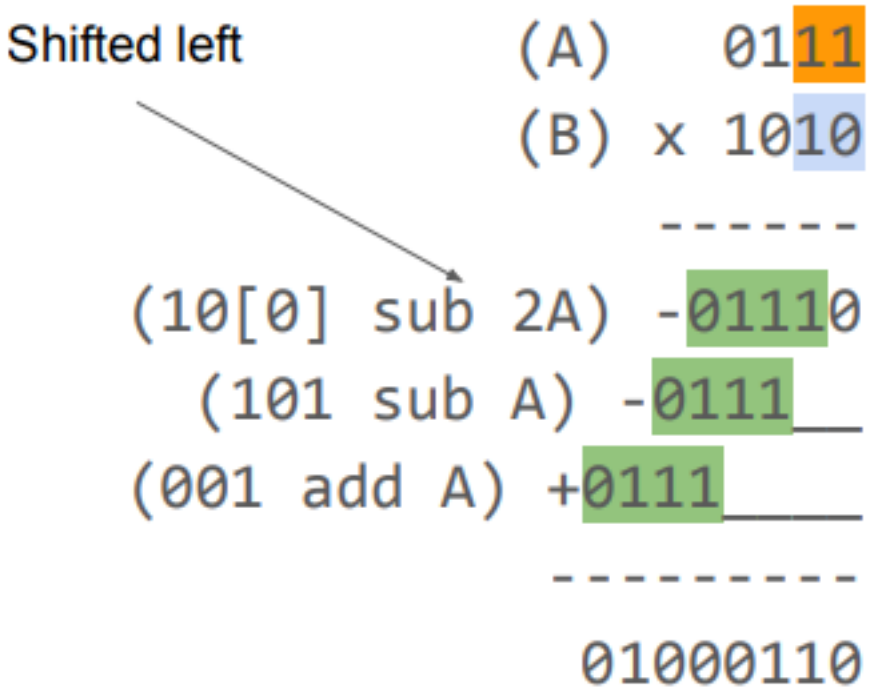
from previous bit pair

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

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

Example

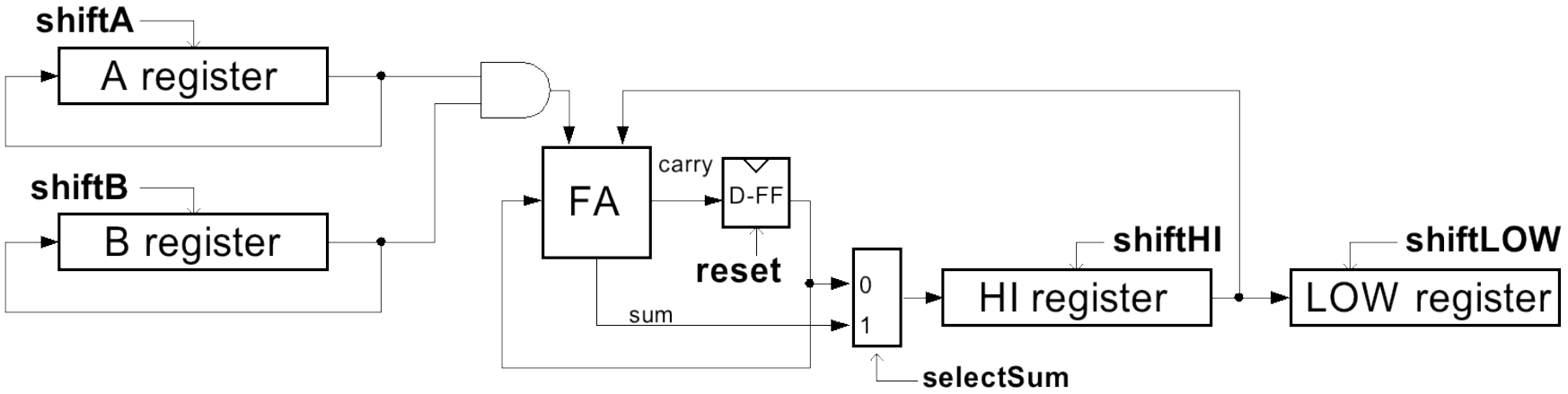

Shifted left
(A) 0111
(B) x 1010

(10[0] sub 2A) -01110
(101 sub A) -0111__
(001 add A) +0111____

01000110

Bit-serial Multiplier

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



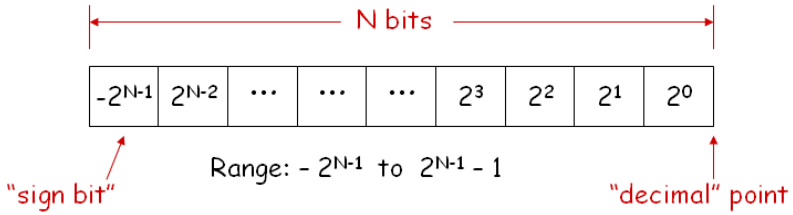
- Control Algorithm:

```
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!)

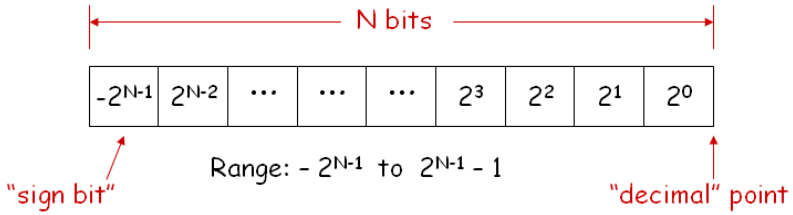


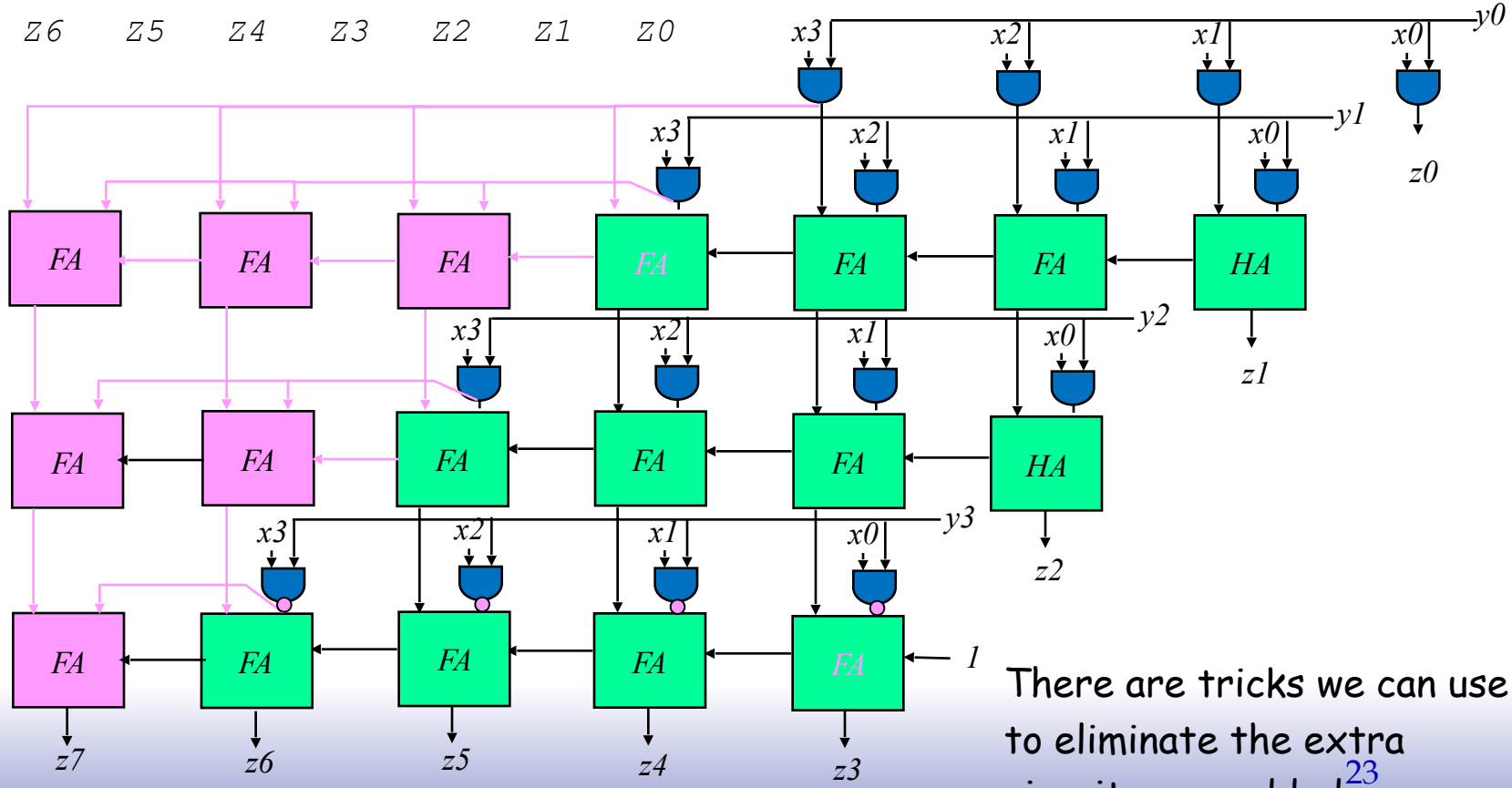
X * Y = (-3) * (-2)

```
(-3)          1   0   1        (X)
(-2)      *   1   1   0        (Y)
          ---------------------
            0 0 0 0 0 0          Y0*X =  0
          + 1 1 1 0 1            2Y1*X = -6
          - 1 1 0 1              4Y2*X = -12
          ----------------------
(+6)        0 0 0 1 1 0
```

Combinational Multiplier (signed)

```
                      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
```





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
                      ------------------------
  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
```

$-B = \sim B + 1$

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

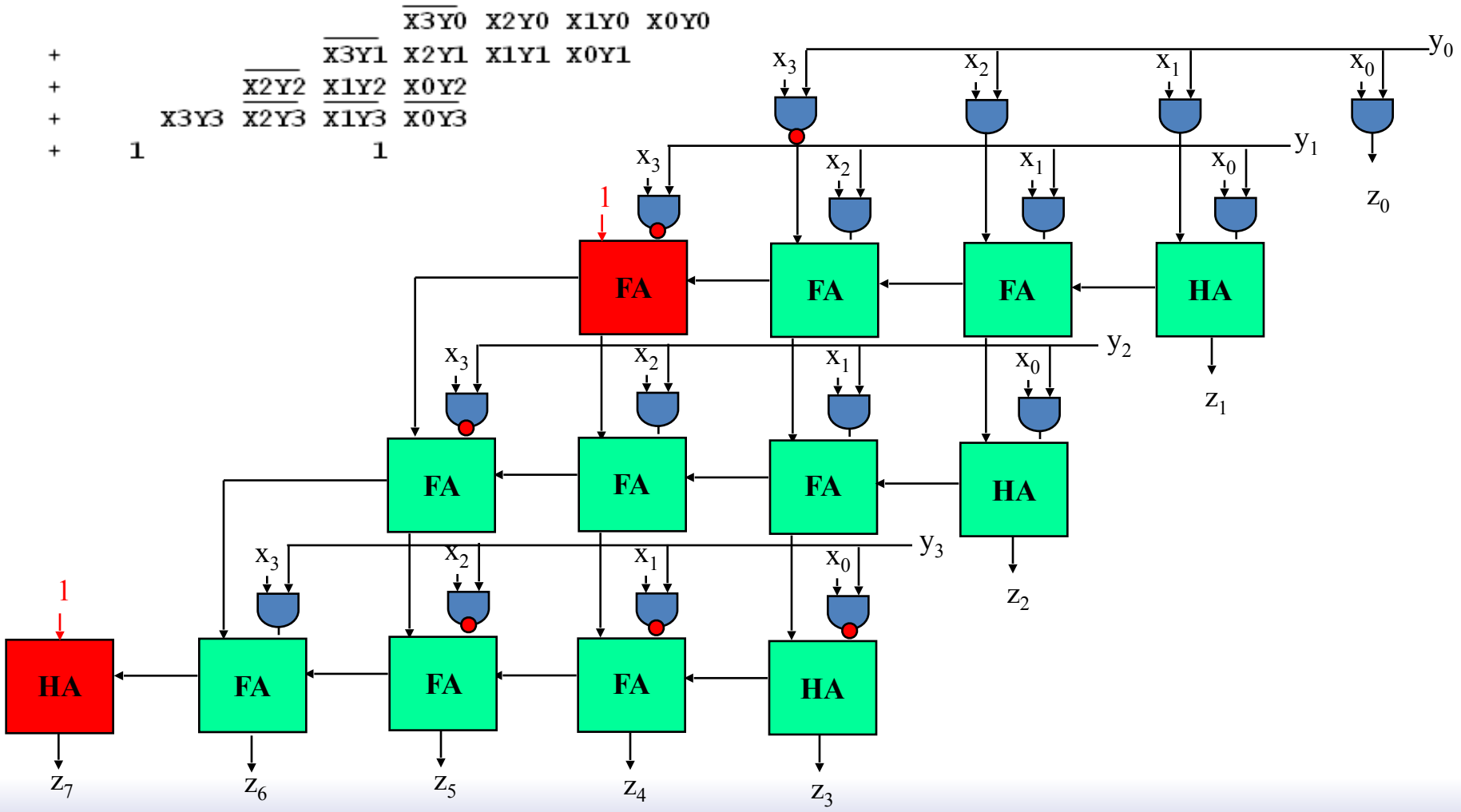
```
                          X3Y0  X2Y0  X1Y0  X0Y0
+               X3Y1  X2Y1  X1Y1  X0Y1
+         X2Y2  X1Y2  X0Y2
+   X3Y3  X2Y3  X1Y3  X0Y3
+
+                           1
-       1     1     1     1
```

Step 4: finish computing the constants…

```
                          X3Y0  X2Y0  X1Y0  X0Y0
+               X3Y1  X2Y1  X1Y1  X0Y1
+         X2Y2  X1Y2  X0Y2
+   X3Y3  X2Y3  X1Y3  X0Y3
+ 1                   1
```

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

2's Complement Multiplication



Example

- What's -3 x -5?

```
  1101
x 1011
```

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

Outline

- *Constant Coefficient Multiplication*
- *Shifters*
- *Counters*

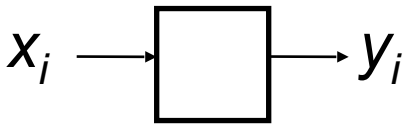
Constant Multiplication

- *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 and other hardware. Ex:

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

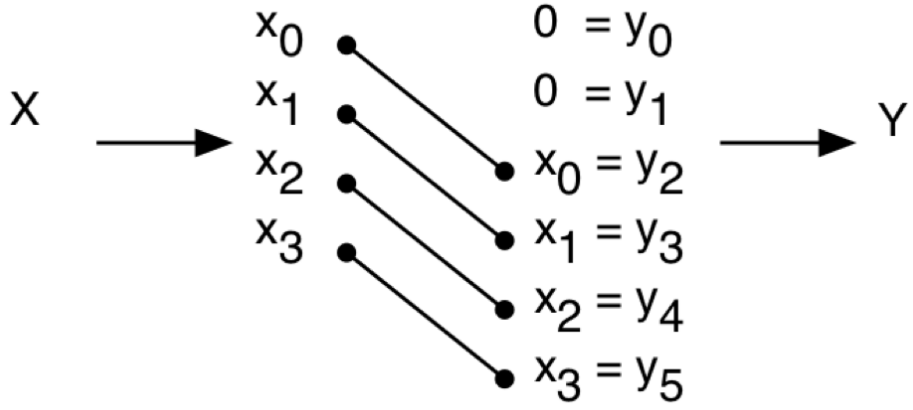


where α is an application dependent constant that is hard-wired into the circuit.

- How do we build and array style (combinational) multiplier that takes advantage of the constancy of one of the operands?

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

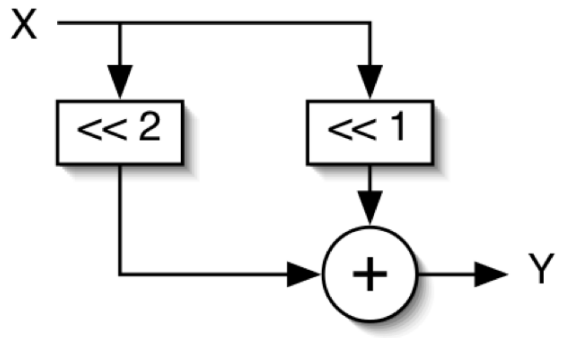


- What about division?

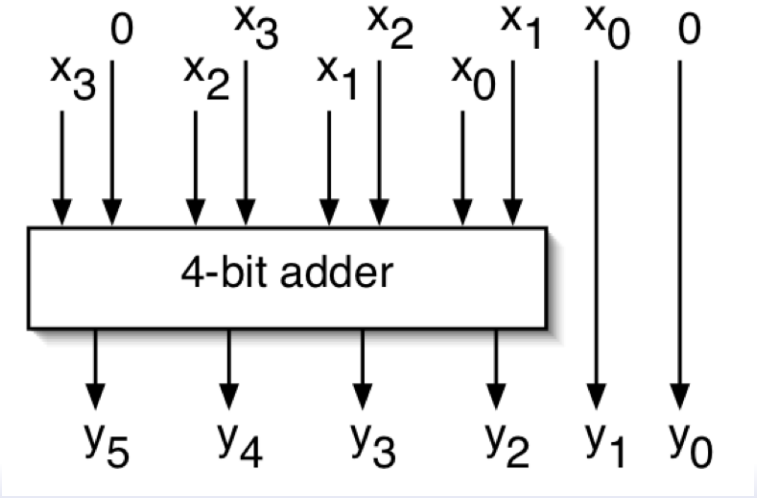
- What about multiplication by non- powers of 2?

Multiplication by a Constant

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

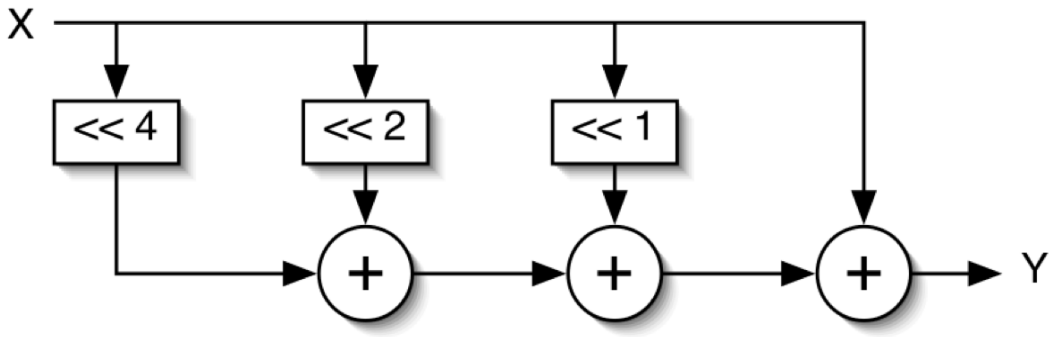


  - Details:



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 balanced 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

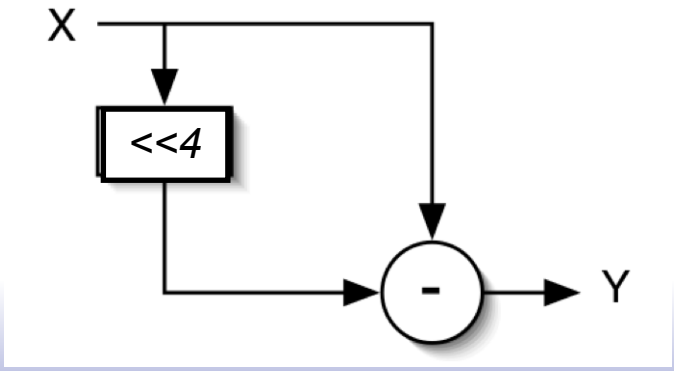
- *Subtraction is approximately the same cost and delay as addition.*
- Consider C*X where C is the constant value 15_{10} = 01111.

  C*X requires 3 additions.
- We can "recode" 15

  from 01111 = $(2^3 + 2^2 + 2^1 + 2^0)$

  to 1000$\bar{1}$ = $(2^4 - 2^0)$

  where $\bar{1}$ means negative weight.
- Therefore, 15*X can be implemented with only one subtractor.



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.
  - 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}$0 by 0010 and 0$\bar{1}$10 by 00$\bar{1}$0
- Examples:

011101 = 29
100$\bar{1}$01 = 32 - 4 + 1

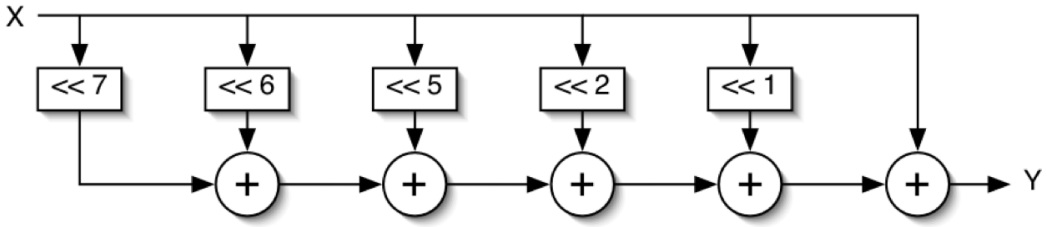
0010111 = 23
0011001$\bar{1}$
0101001$\bar{1}$ = 32 - 8 - 1

0110110 = 54
10$\bar{1}$10$\bar{1}$0
100$\bar{1}$0$\bar{1}$0 = 64 - 8 - 2

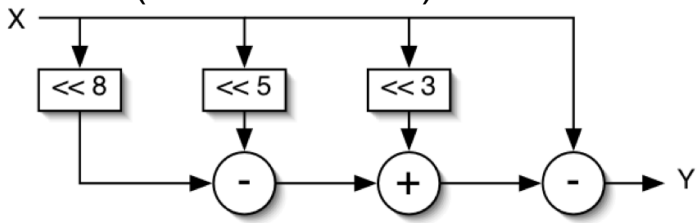
- Can we further simplify the multiplier circuits?

"Constant Coefficient Multiplication" (KCM)

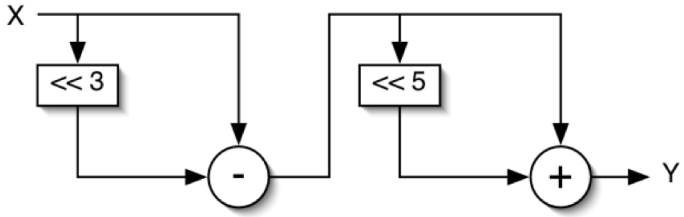
Binary multiplier: $Y = 231*X = (2^7 + 2^6 + 2^5 + 2^2 + 2^1 + 2^0)*X$



- CSD helps, but the multipliers are limited to shifts followed by adds.
  - CSD multiplier: $Y = 231*X = (2^8 - 2^5 + 2^3 - 2^0)*X$



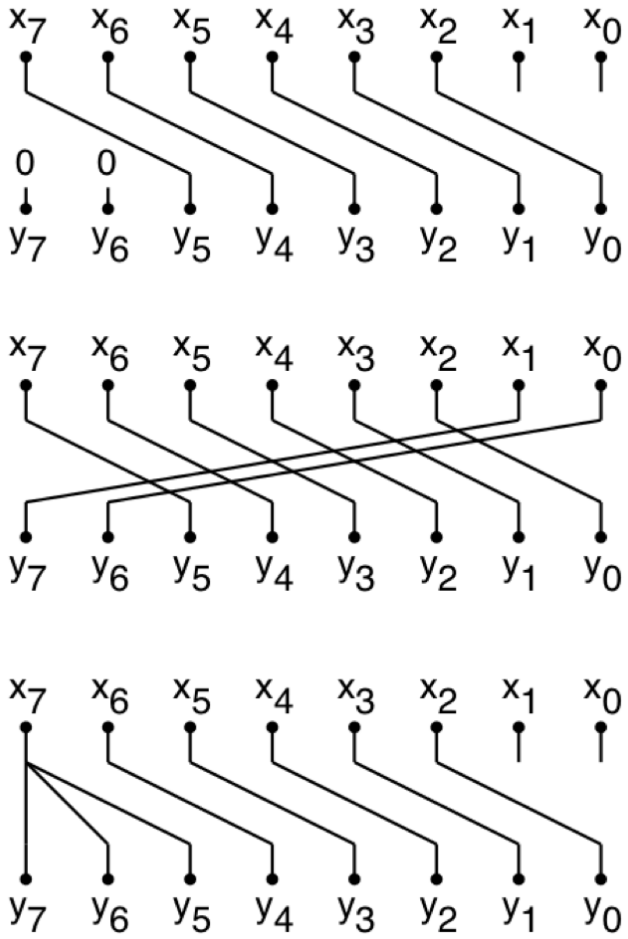
- How about shift/add/shift/add …?
  - KCM multiplier: $Y = 231*X = 7*33*X = (2^3 - 2^0)*(2^5 + 2^0)*X$



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

Shifters

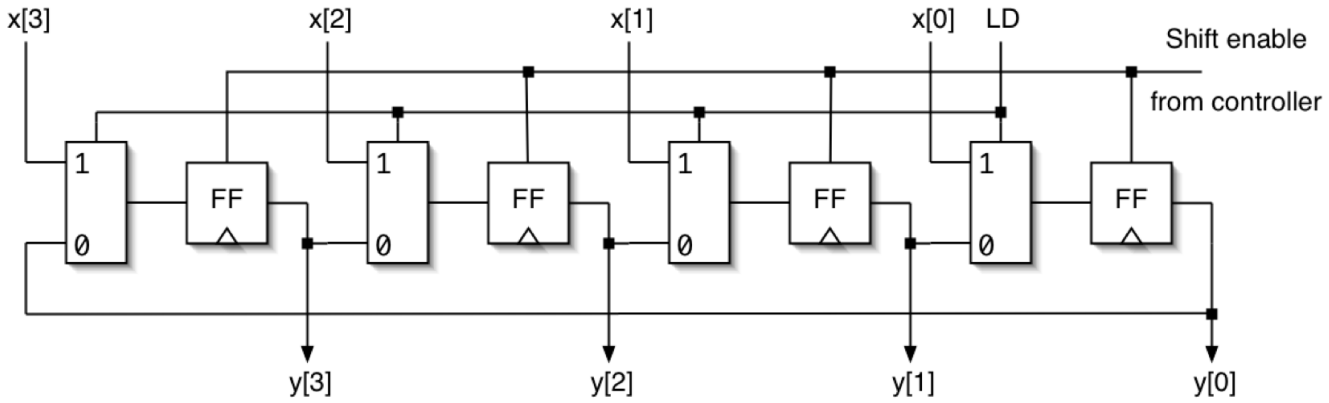
Fixed Shifters / Rotators Defined



Variable Shifters / Rotators

- Example: X >> S, where S is unknown when we synthesize the circuit.
- Uses: shift instruction in processors (ARM includes a shift on every instruction), floating-point arithmetic, division/multiplication by powers of 2, etc.
- One way to build this is a simple shift-register:

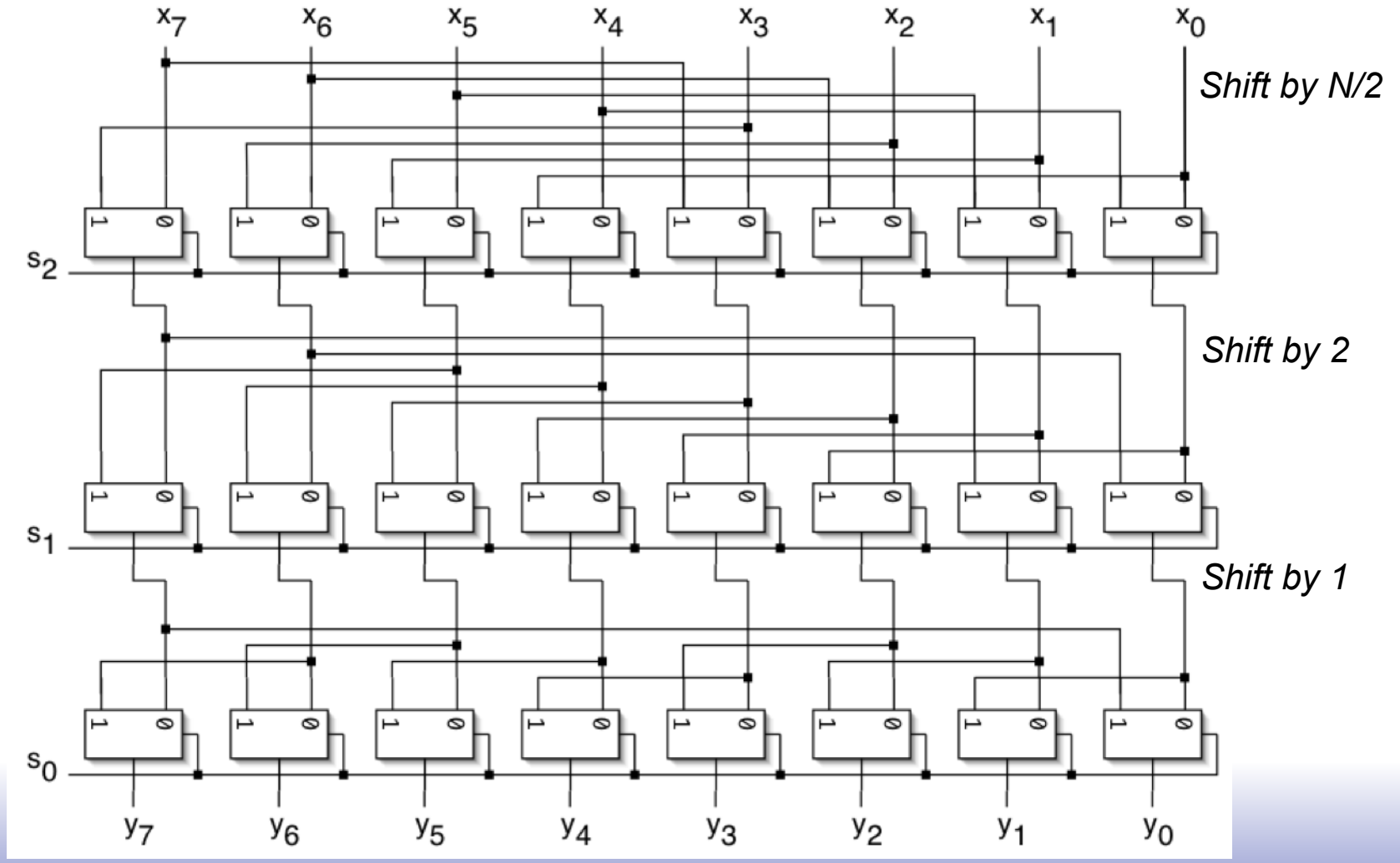
  a) Load word, b) shift enable for S cycles, c) read word.



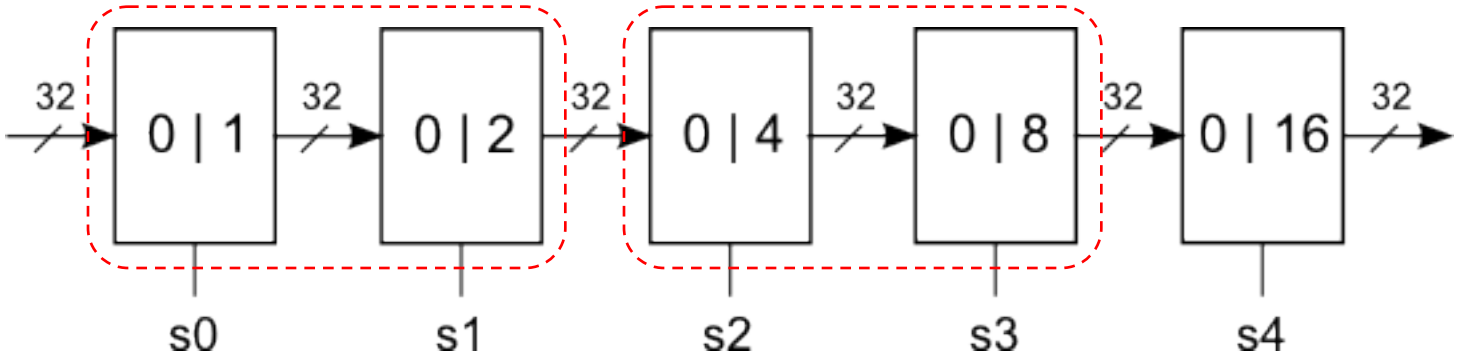
- Worst case delay O(N) , not good for processor design.
- Can we do it in O(logN) time and fit it in one cycle?

Log Shifter / Rotator

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

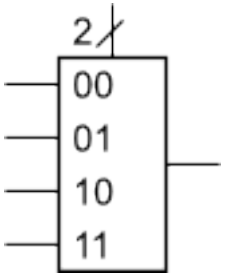


LUT Mapping of Log shifter

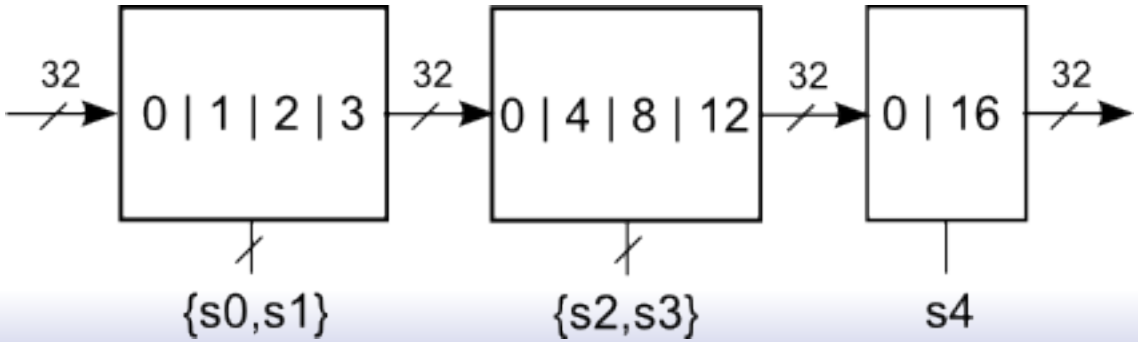


Efficient with 2to1 multiplexors, for instance, 3LUTs.

Virtex6 has 6LUTs. Naturally makes 4to1 muxes:



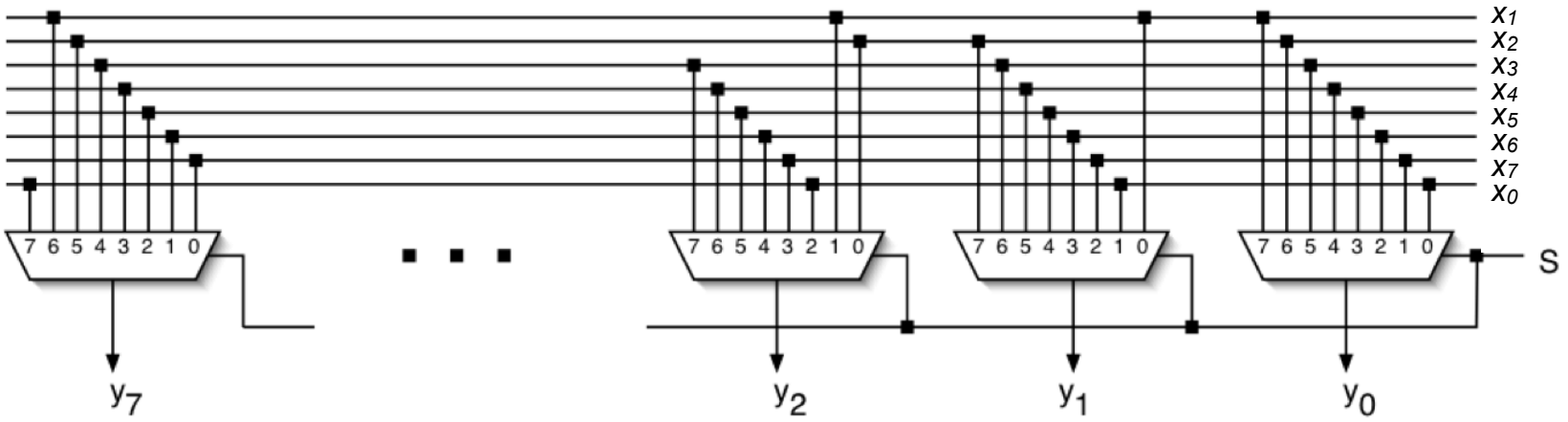
Reorganize shifter to use 4to1 muxes.



Final stage uses F7 mux

"Improved" Shifter / Rotator

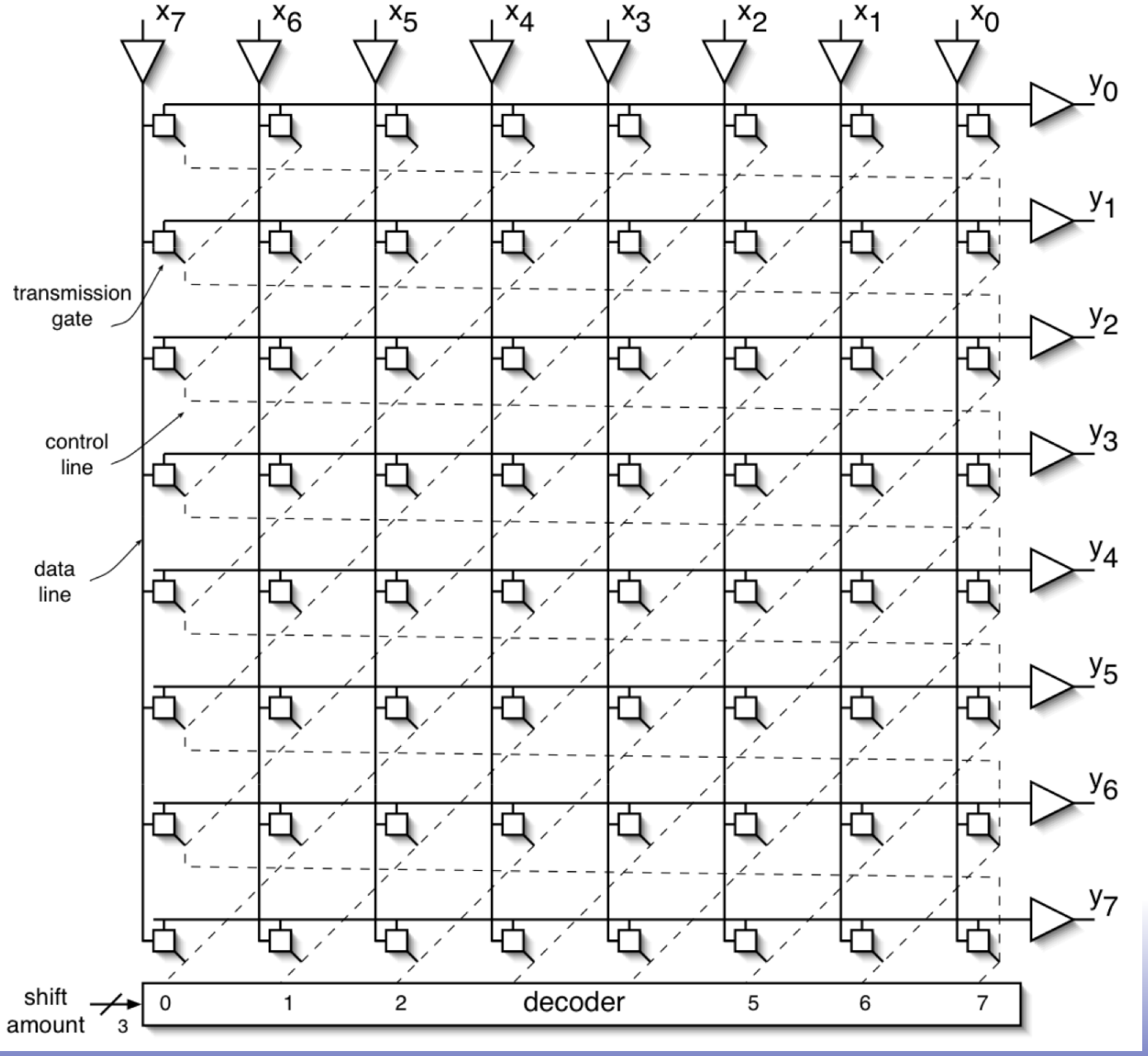
- How about this approach? Could it lead to even less delay?



Left-shift with rotate

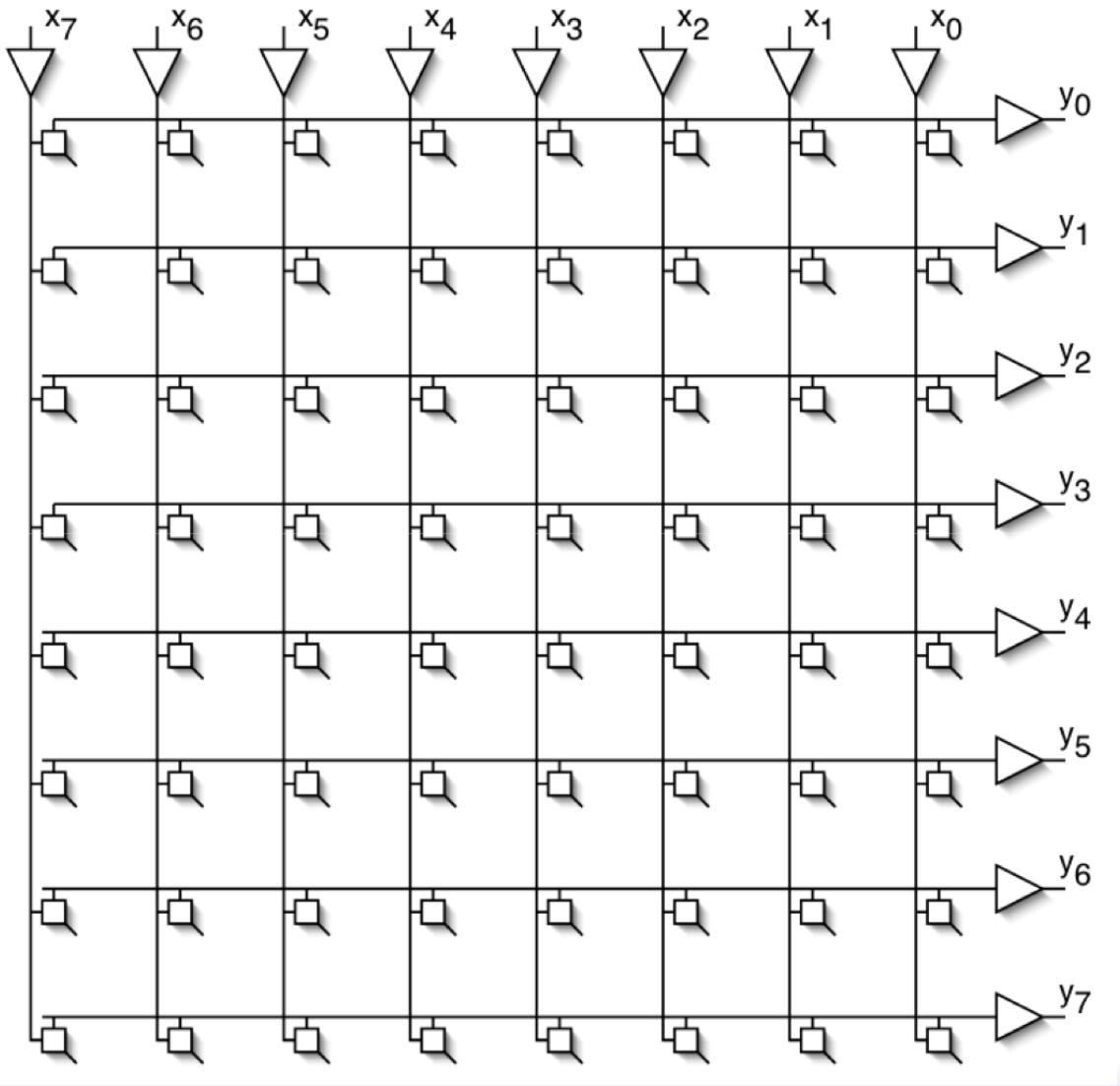
- What is the delay of these big muxes?
- Look a transistor-level implementation?

Barrel Shifter



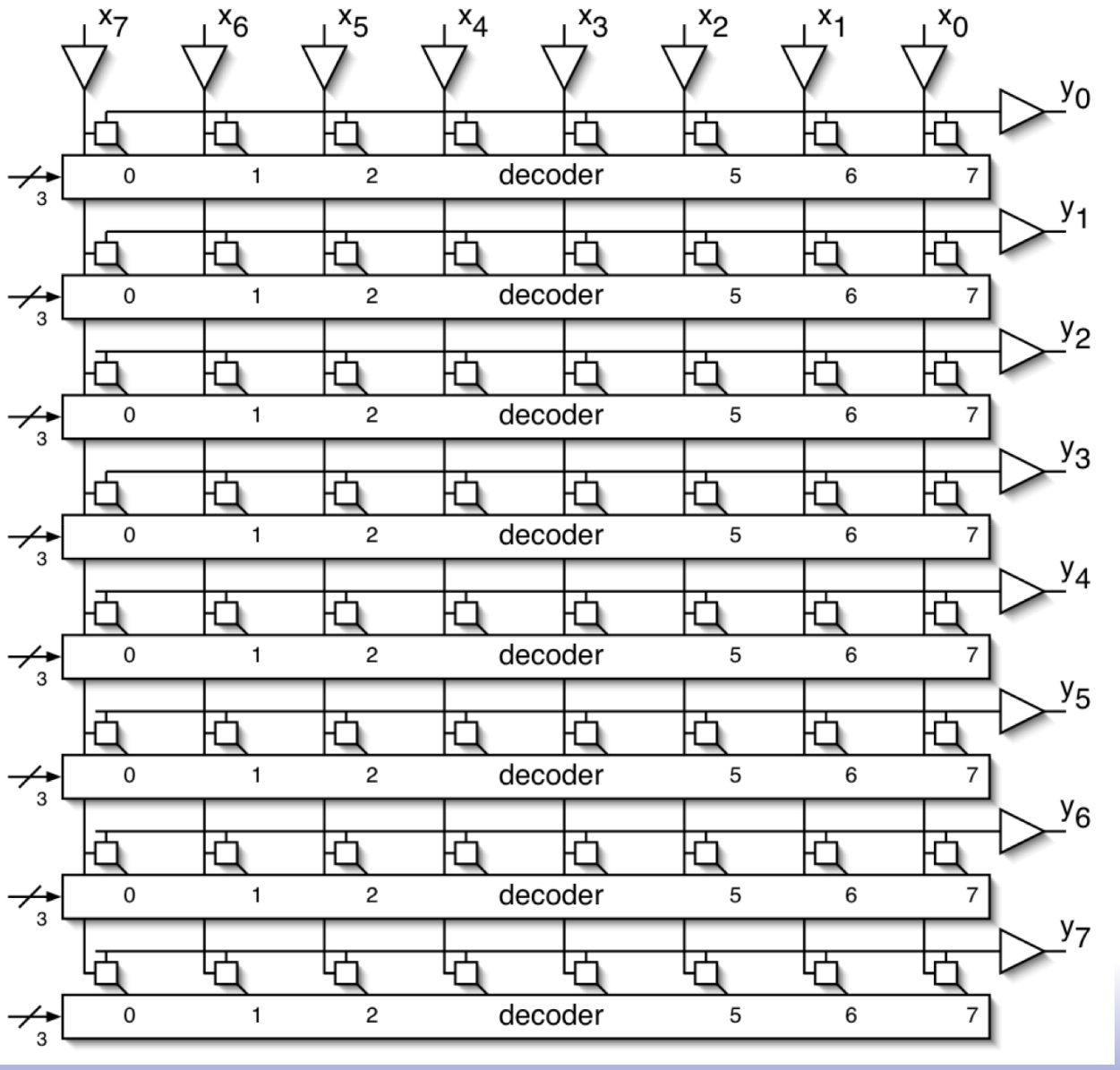
Cost/delay?

Connection Matrix



- Generally useful structure:
  - N^2 control points.
  - What other interesting functions can it do?

Cross-bar Switch



- Nlog(N) control signals.
- Supports all interesting permutations
  - All one-to-one and one-to-many connections.
- Commonly used in communication hardware (switches, routers).