

EECS150 - Digital Design

Lecture 21 - Multipliers & Shifters

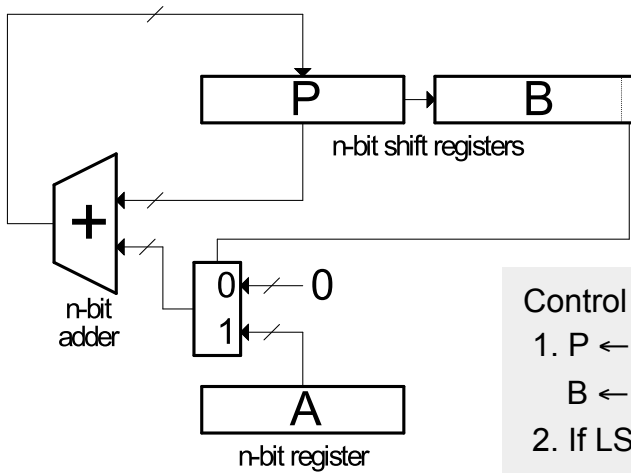
April 9, 2013
John Wawrzynek

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	}	<i>Partial products</i>	
		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					
					\dots	$a_1b_0 + a_0b_1$	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:

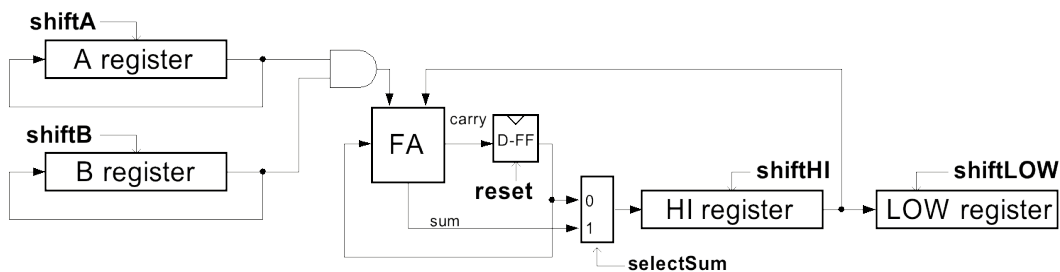
$$X = \sum_{i=0}^{N-2} x_i 2^i - x_{n-1} 2^{n-1}$$

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

- 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

Bit-serial Multiplier

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



- Control Algorithm:

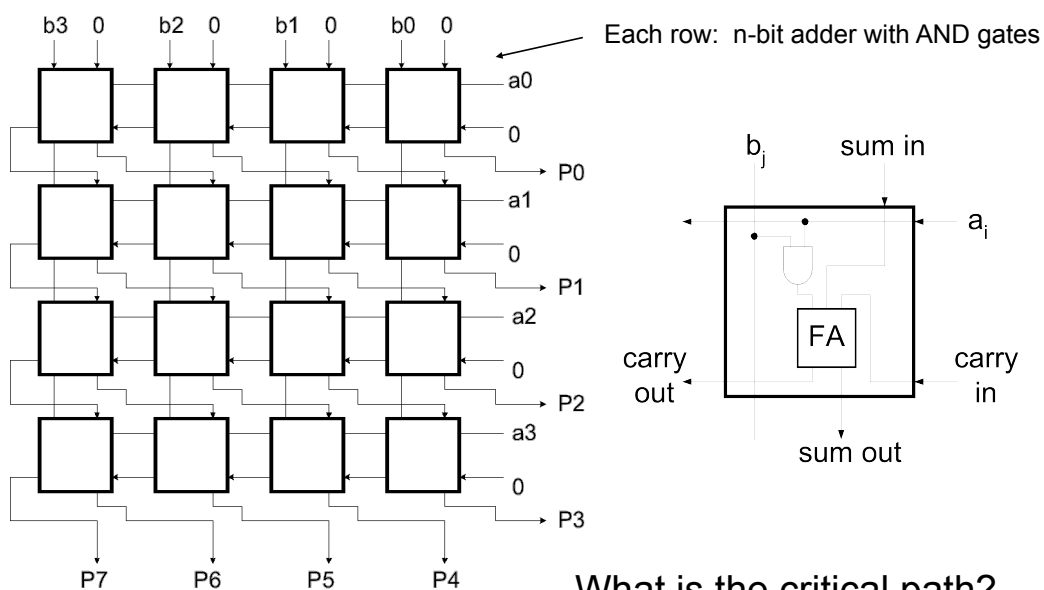
```

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 $x=0$.

Array Multiplier

Single cycle multiply: Generates all n partial products simultaneously.



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

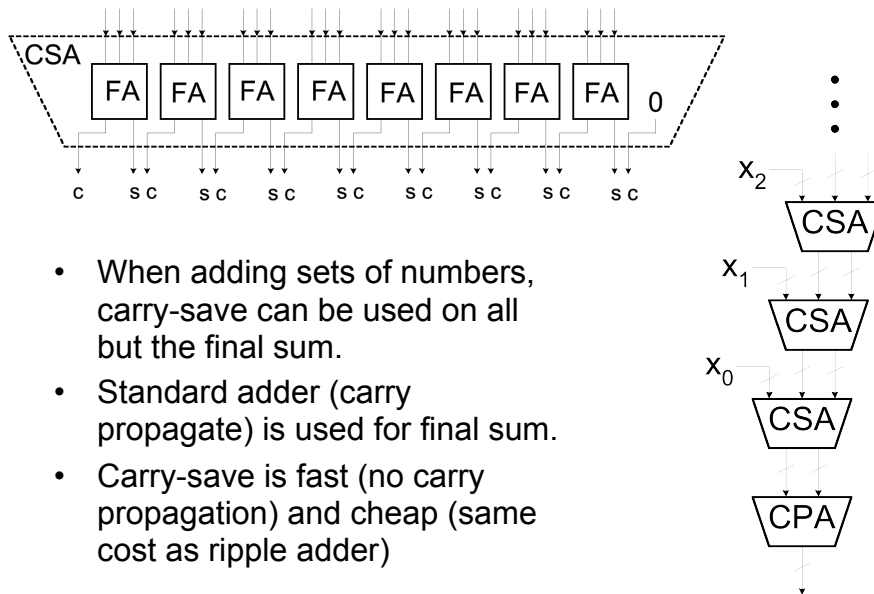
$$\begin{array}{r}
 3_{10} \quad 0011 \\
 + 2_{10} \quad 0010 \\
 \hline
 c \quad 0100 = 4_{10} \\
 s \quad 0001 = 1_{10} \\
 \hline
 3_{10} \quad 0011 \\
 + c \quad 0010 = 2_{10} \\
 \hline
 s \quad 0110 = 6_{10} \\
 \hline
 1000 = 8_{10}
 \end{array}$$

} carry-save add

} carry-propagate add

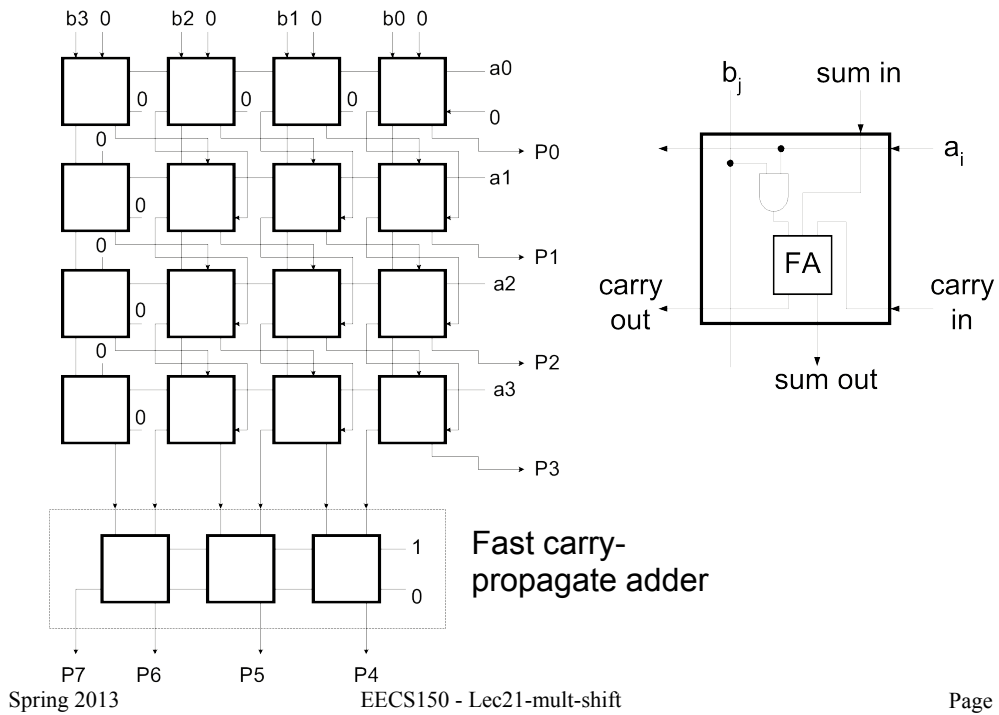
- In general, *carry-save* addition takes in 3 numbers and produces 2.
- Whereas, *carry-propagate* takes 2 and produces 1.
- With this technique, we can avoid carry propagation until final addition

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)

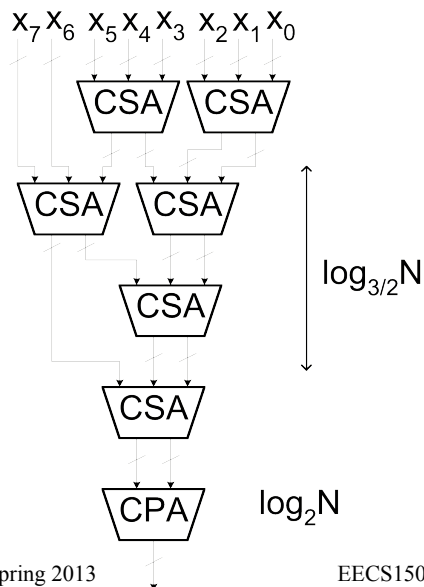
Array Multiplier using Carry-save Addition



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.
- 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 $\propto \log_{3/2} N + \log_2 N$

Constant Multiplication

- Our discussion 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:

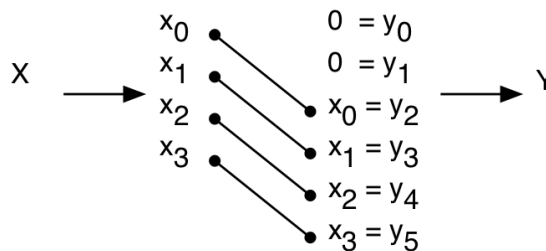
$$y_i = \alpha y_{i-1} + x_i$$


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

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

Multiplication by a Constant

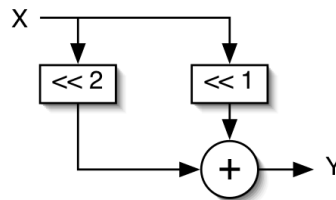
- If the constant C in $C * X$ is a power of 2, then the multiplication is simply a shift of X.
- Ex: $4 * X$



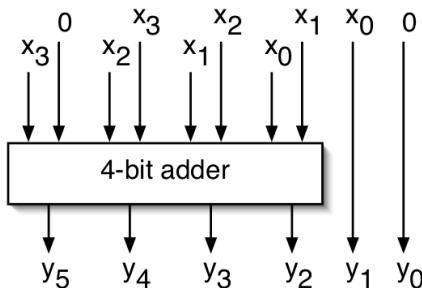
- What about division?
- What about multiplication by non-powers of 2?

Multiplication by a Constant

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

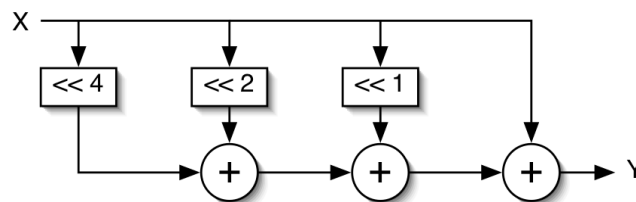


– Details:



Multiplication by a Constant

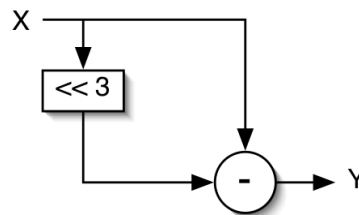
- Another example: $C = 23_{10} = 010111$



- In general, the number of additions equals the number of 1's in the constant minus one.*
- Using carry-save adders (for all but one of these) helps reduce the delay and cost, 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 $C \cdot X$ where C is the constant value $15_{10} = 01111_2$.
 $C \cdot X$ 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, $15 \cdot X$ can be implemented with only one subtractor.



Canonic Signed Digit Representation

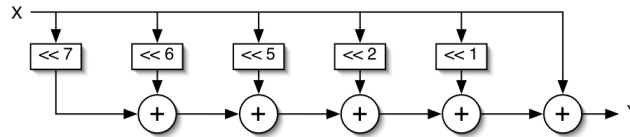
- CSD represents numbers using 1, $\bar{1}$, & 0 with the least possible number of non-zero digits.
 - Strings of 2 or more non-zero digits are replaced.
 - Leads to a unique representation.
- To form CSD representation might take 2 passes:
 - First pass: replace all occurrences of 2 or more 1's:
 $01..10$ by $10..\bar{1}0$
 - Second pass: same as a above, plus replace $0\bar{1}10$ by $00\bar{1}0$
- Examples:

$011101 = 29$	$0010111 = 23$	$0110110 = 54$
$100\bar{1}01 = 32 - 4 + 1$	$001100\bar{1}$	$10\bar{1}10\bar{1}0$
	$010\bar{1}00\bar{1} = 32 - 8 - 1$	$100\bar{1}0\bar{1}0 = 64 - 8 - 2$

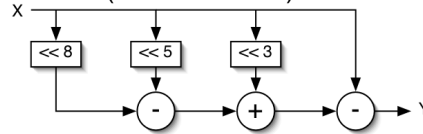
- Can we further simplify the multiplier circuits?

“Constant Coefficient Multiplication” (KCM)

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

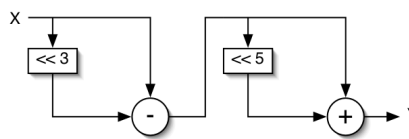


- CSD helps, but the multipliers are limited to shifts followed by adds.
 - CSD multiplier: $Y = 231 * X = (2^8 - 2^5 + 2^3 - 2^0) * X$



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

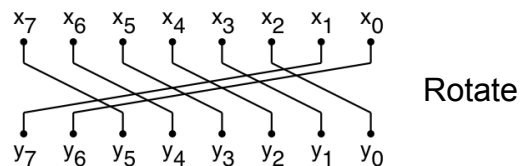
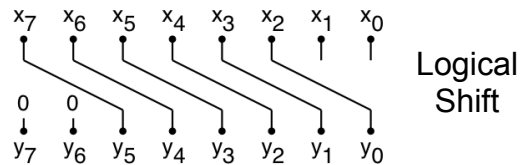
- KCM multiplier: $Y = 231 * X = 7 * 33 * X = (2^3 - 2^0) * (2^5 + 2^0) * X$



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

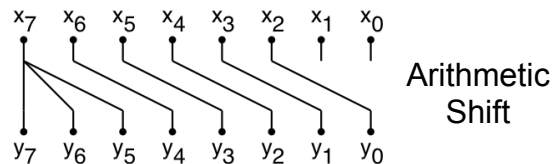
Fixed Shifters / Rotators

- “fixed” shifters
“hardwire” the shift amount into the circuit.
- Ex: verilog: $X \gg 2$
- (right shift X by 2 places)



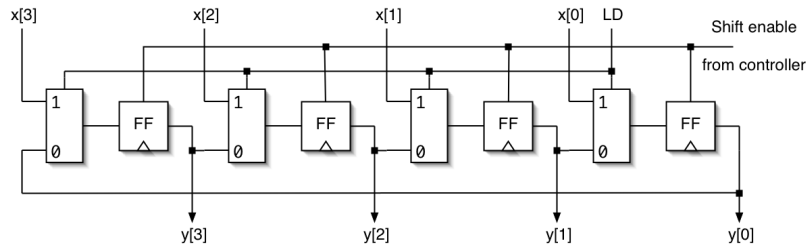
- Fixed shift/rotator is nothing but wires!

So what?



Variable Shifters / Rotators

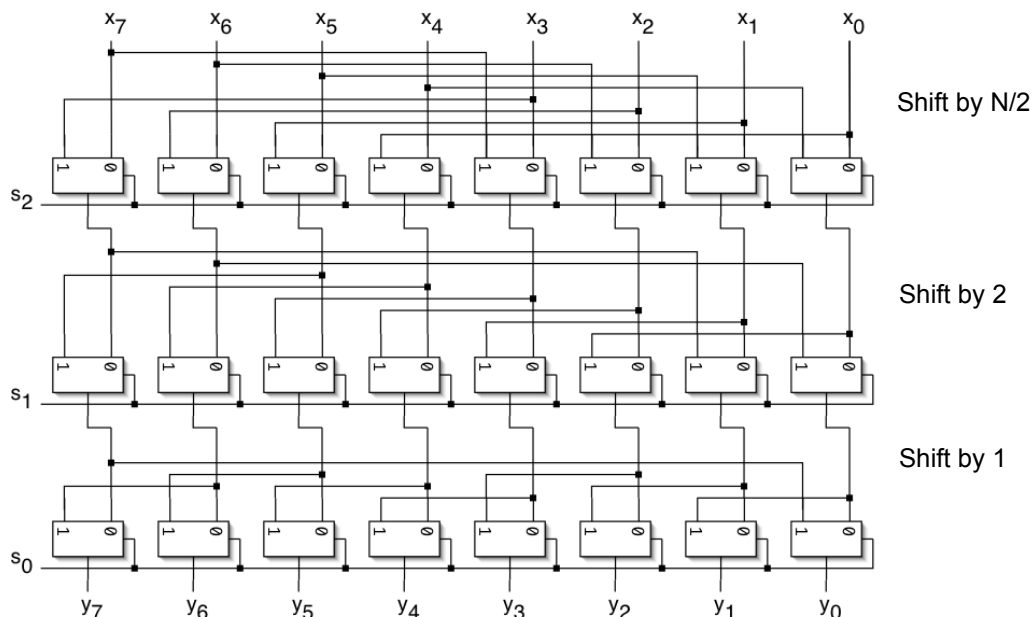
- Example: $X \gg S$, where S is unknown when we synthesize the circuit.
- Uses: shift instruction in processors (ARM includes a shift on every instruction), floating-point arithmetic, division/multiplication by powers of 2, etc.
- One way to build this is a simple shift-register:
 - a) Load word, b) shift enable for S cycles, c) read word.



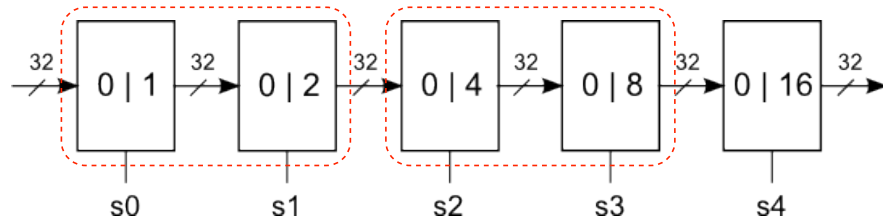
- Worst case delay $O(N)$, not good for processor design.
- Can we do it in $O(\log N)$ time and fit it in one cycle?

Log Shifter / Rotator

- $\log(N)$ stages, each shifts (or not) by a power of 2 places,

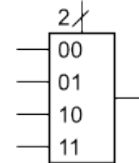


LUT Mapping of Log shifter

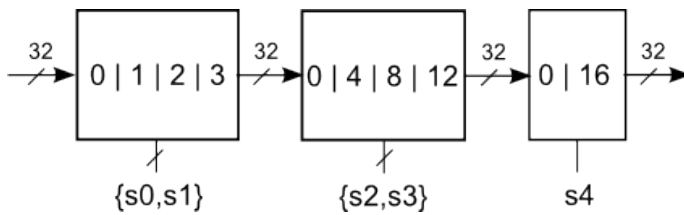


Efficient with 2to1 multiplexors, for instance, 3LUTs.

Virtex6 has 6LUTs. Naturally makes 4to1 muxes:



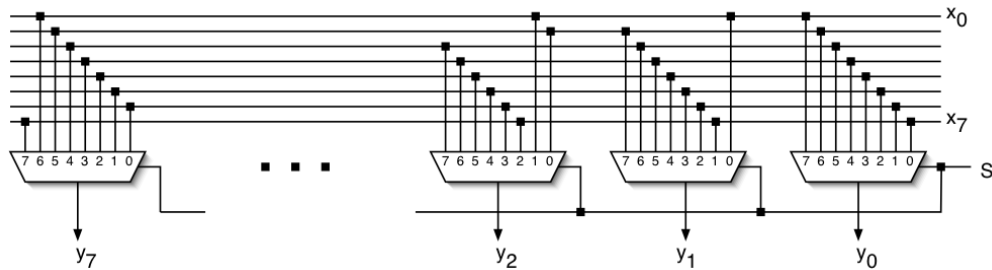
Reorganize shifter to use 4to1 muxes.



Final stage uses F7 mux

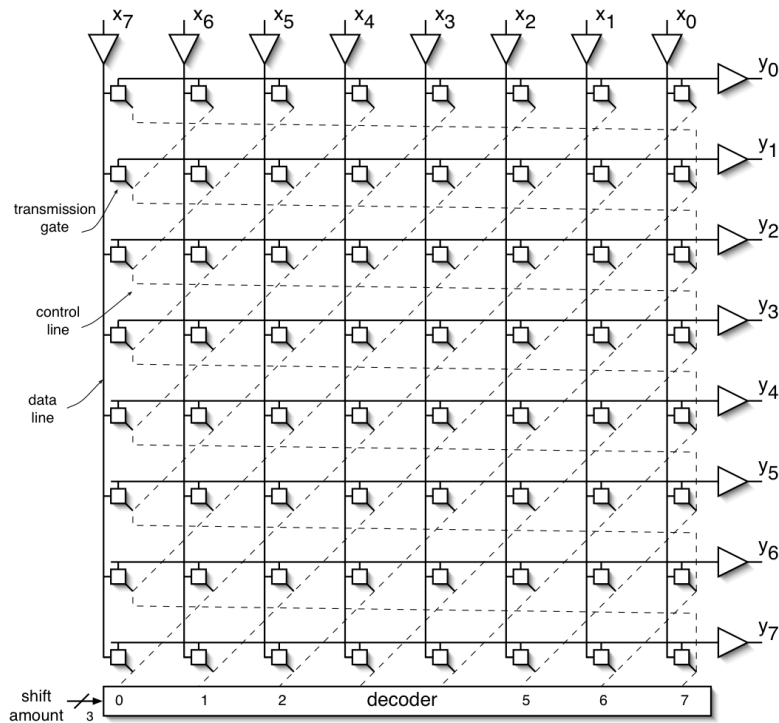
“Improved” Shifter / Rotator

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



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

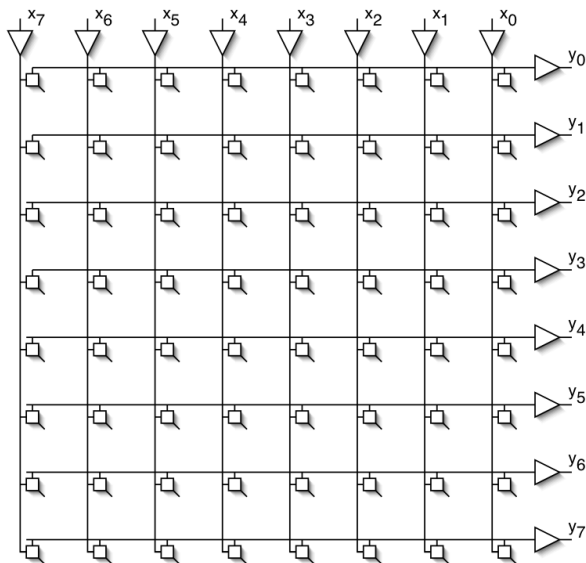
Barrel Shifter



Cost/delay?

- (don't forget the decoder)

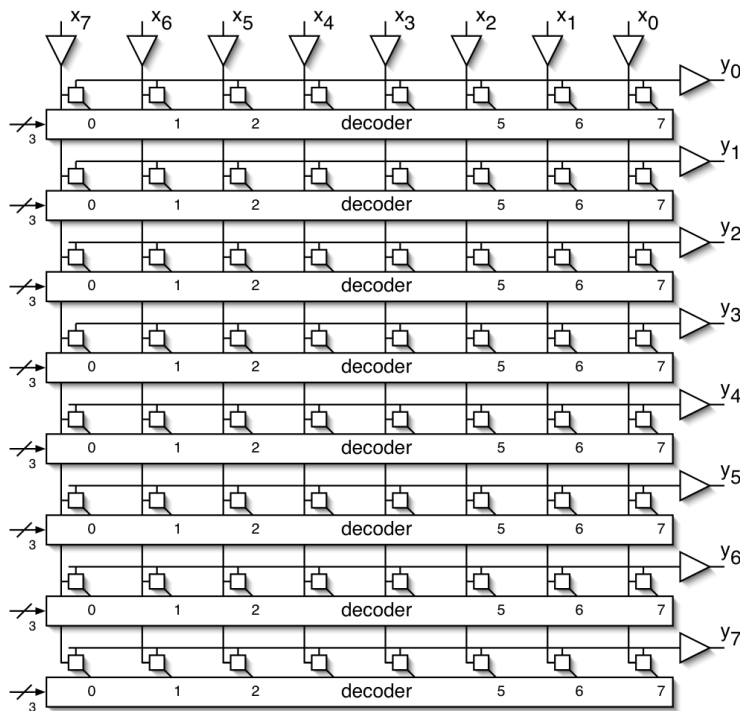
Connection Matrix



Generally useful structure:

- N^2 control points.
- What other interesting functions can it do?

Cross-bar Switch



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