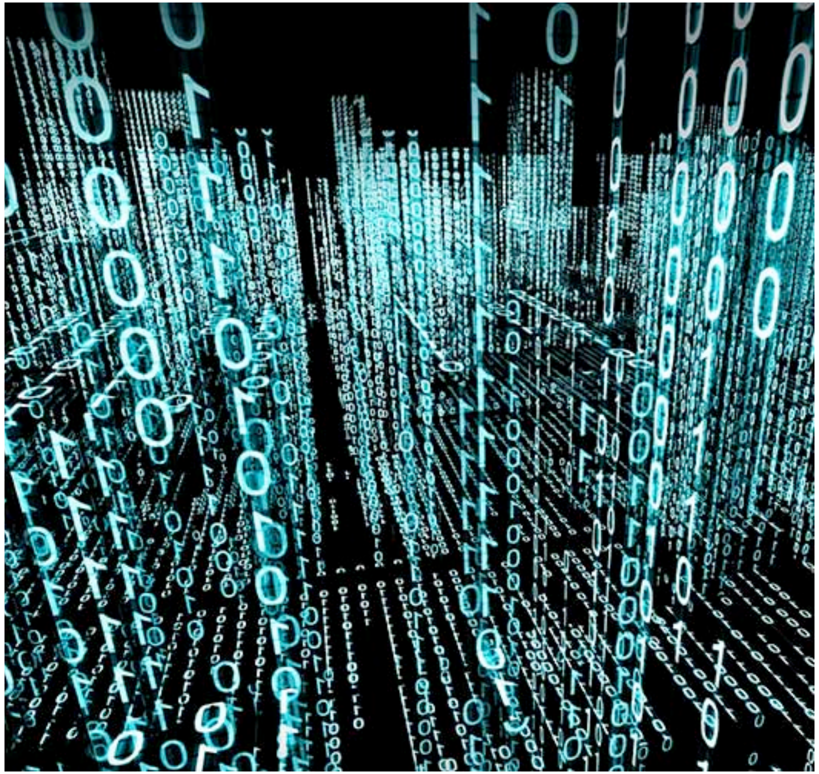


EECS151/251A

Spring 2024

Digital Design and Integrated Circuits

Instructor: John Wawrzynek

Lecture 22: Multiplier Circuits and Shifters

Announcements

- ❑ Homework 10 posted - <u>due next Wednesday</u>
- ❑ 2 more weeks of lecture (including this week)
- ❑ Next week Monday - guest lecture: Sandesh Bharadwaj, from Apple, *Hardware Verification*
- ❑ 1 more homework exercise

Warmup

- Recall long multiplication of base-10 by hand:

```
   56
x  12
 ----
```

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

```
  011
x 101
 ----
```

Multiplication

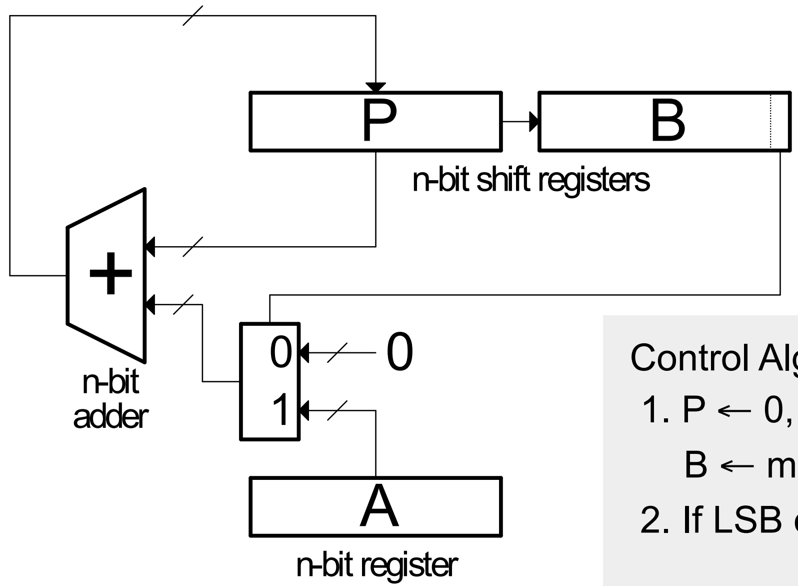
$$\begin{array}{ccccccc}
 & & & a_3 & a_2 & a_1 & a_0 \\
 & & \text{X} & b_3 & b_2 & b_1 & b_0 \\ \cline{4-7}
\end{array}
\begin{array}{l}
\leftarrow \textit{Multiplicand} \\
\leftarrow \textit{Multiplier} \\
\end{array}$$

$$\left.\begin{array}{ccccccc}
 & & & a_3b_0 & a_2b_0 & a_1b_0 & a_0b_0 \\
 & & a_3b_1 & a_2b_1 & a_1b_1 & a_0b_1 & \\
 & a_3b_2 & a_2b_2 & a_1b_2 & a_0b_2 & & \\
a_3b_3 & a_2b_3 & a_1b_3 & a_0b_3 & & & \\ \hline
\end{array}\right\} \textit{Partial products}$$

$$\begin{array}{ccccc}
 & \ldots & & a_1b_0+a_0b_1 & a_0b_0 \leftarrow \textit{Product}
\end{array}$$

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 ← 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.

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

Signed Multiplication

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

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

ex: $-6 = 11010_2 = 0\cdot2^0 + 1\cdot2^1 + 0\cdot2^2 + 1\cdot2^3 - 1\cdot2^4$

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

- Therefore for multiplication:
  - a) <u>subtract final partial product</u> (multiplier is signed)
  - b) <u>sign-extend partial products</u> (multiplicand is signed)
- Modifications to shift & add circuit:
  - a) adder/subtractor
  - b) sign-extender on P shifter register

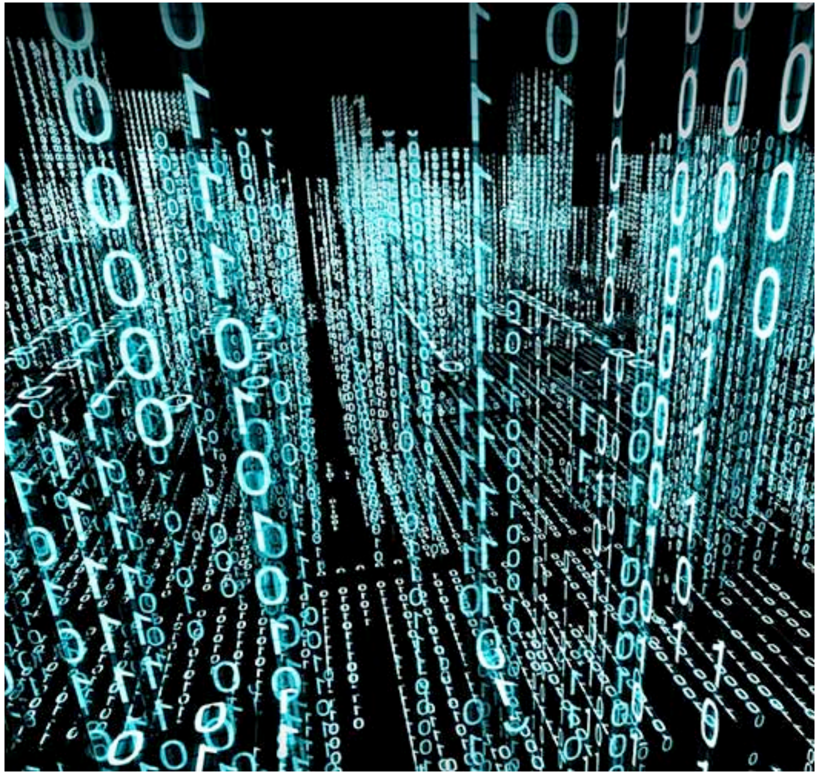
Convince yourself

- What's -3 x 5?

```
    1101
  x 0101
  ------
```

Outline for Multipliers

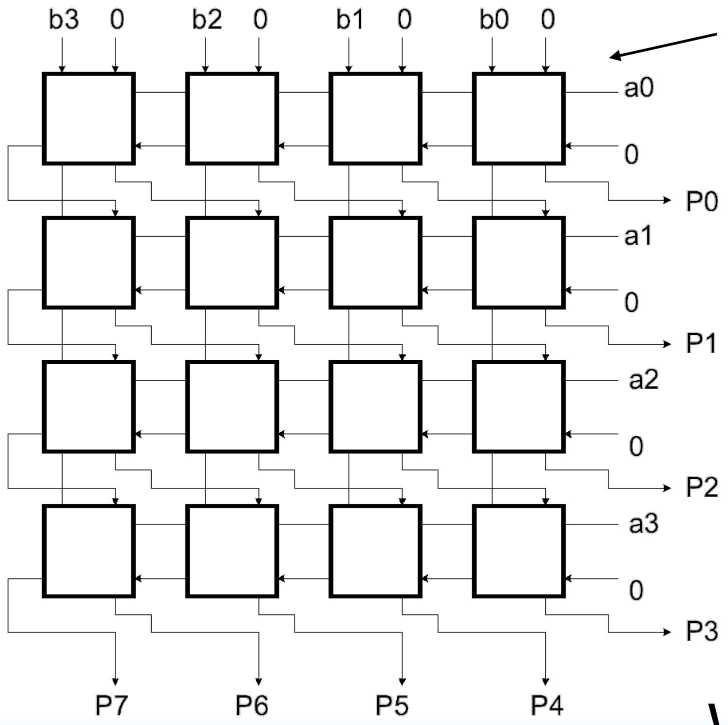
- Combinational multiplier
- Latency & Throughput
  - Wallace Tree
  - Pipelining to increase throughput
- Smaller multipliers
  - ~~Booth encoding~~
  - Serial, bit-serial
- Two's complement multiplier



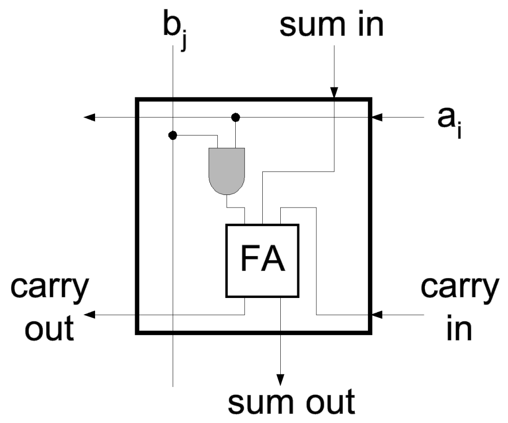
Unsigned
Combinational Multiplier

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.
- Carry-save addition takes in 3 numbers and produces 2.
- (Sometimes called a "3:2 compressor")

- Example: sum four numbers, $1_{10} = 0001$, $3_{10} = 0011$, $2_{10} = 0010$, $3_{10} = 0011$

$$\left.\begin{array}{rrcl}
 & 1_{10} & 0001 & \\
 & 3_{10} & 0011 & \\
+ & 2_{10} & \underline{0010} & \\
 & c & 0110 & = 6_{10} \\
 & s & 0000 & = 0_{10}
\end{array}\right\}\text{carry-save add}$$

carry-save add

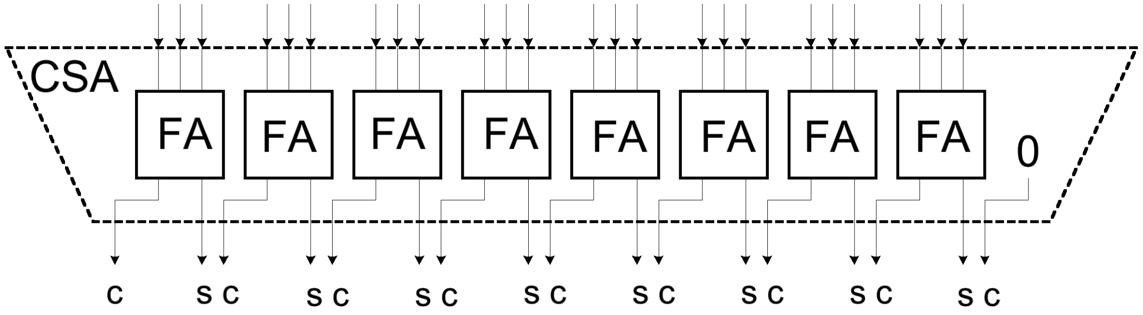
$$\begin{array}{rcl}
3_{10} & \underline{0011} & \\
c & 0100 & = 2_{10} \\
s & \underline{0101} & = 6_{10} \\
 & 1001 & = 8_{10}
\end{array}$$

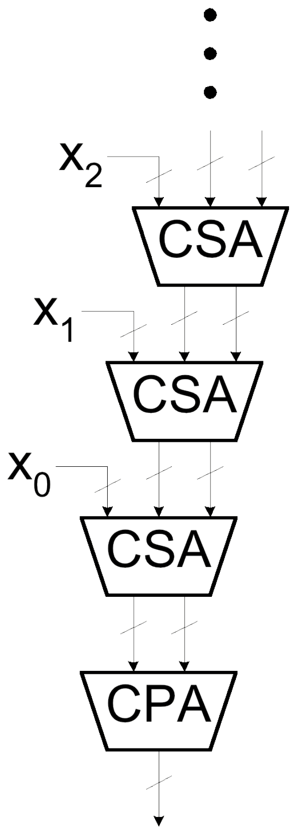
carry-propagate add

With this technique, we can avoid carry propagation until final addition!

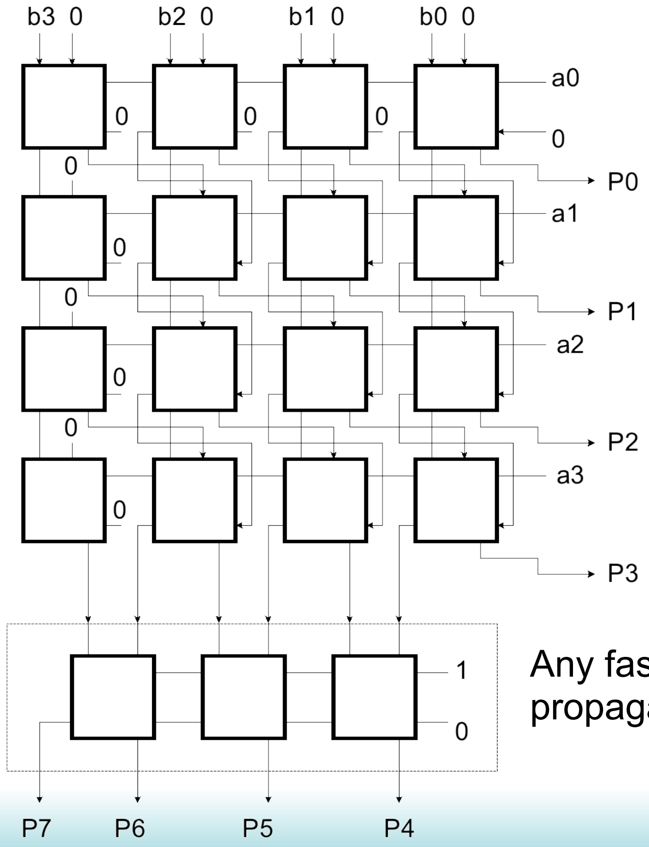
Carry-save Circuits

- 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)

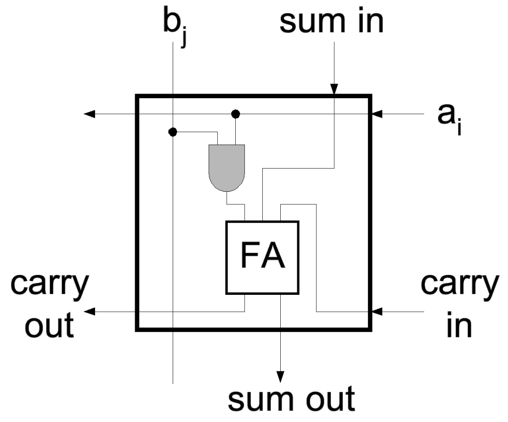




Array Multiplier using Carry-save Addition



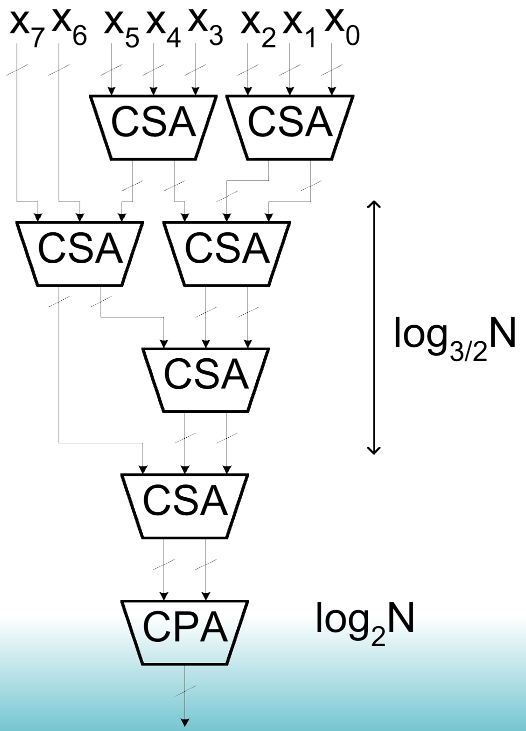
Any fast carry-propagate adder



Carry-save Addition

CSA is associative and commutative. For example:

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

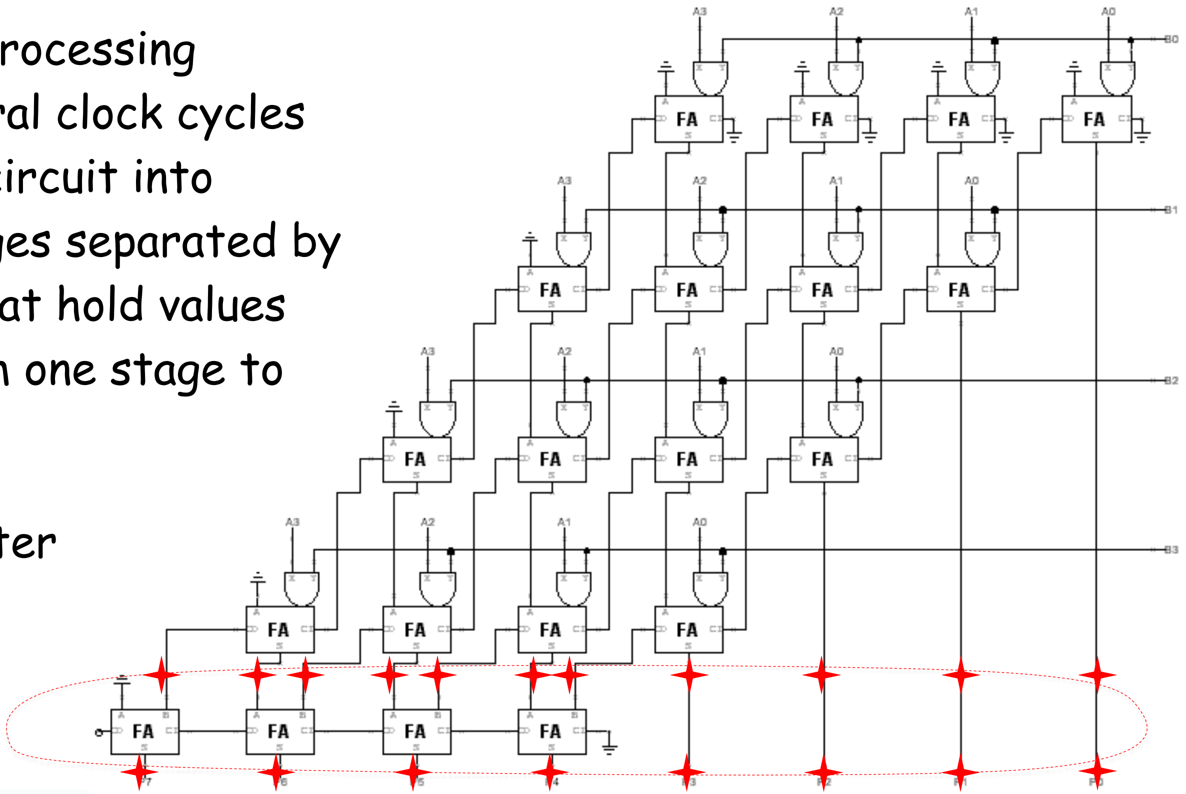


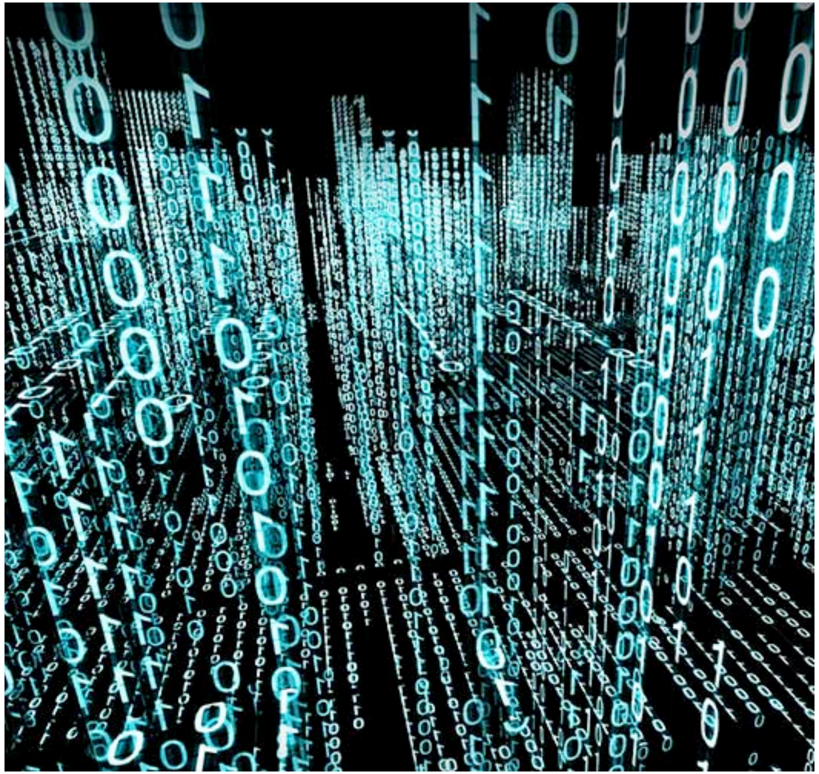
- A balanced tree can be used to reduce the logic delay.
- It doesn't matter where you add the carries and sums, as long as you eventually do add them.
- This structure is the basis of the ***Wallace Tree Multiplier***.
- Partial products are summed with the CSA tree. Fast CPA (ex: CLA) is used for final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

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

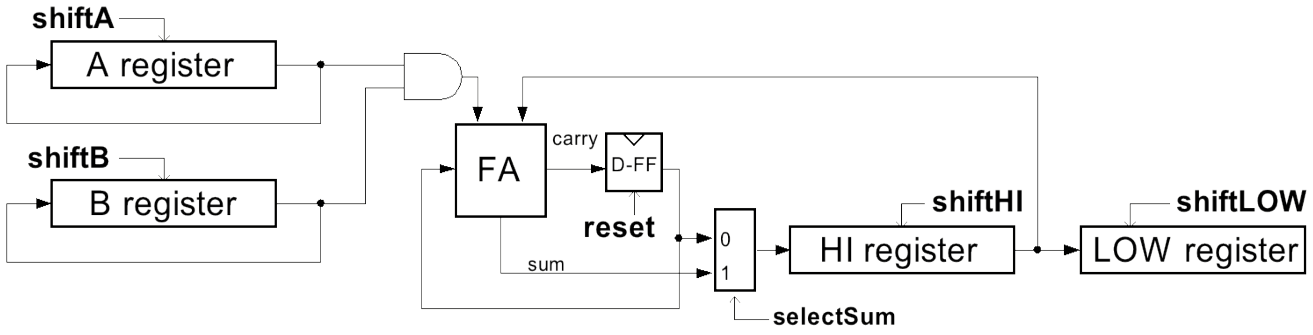




Smaller Combinational Multipliers

Bit-serial Multiplier

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

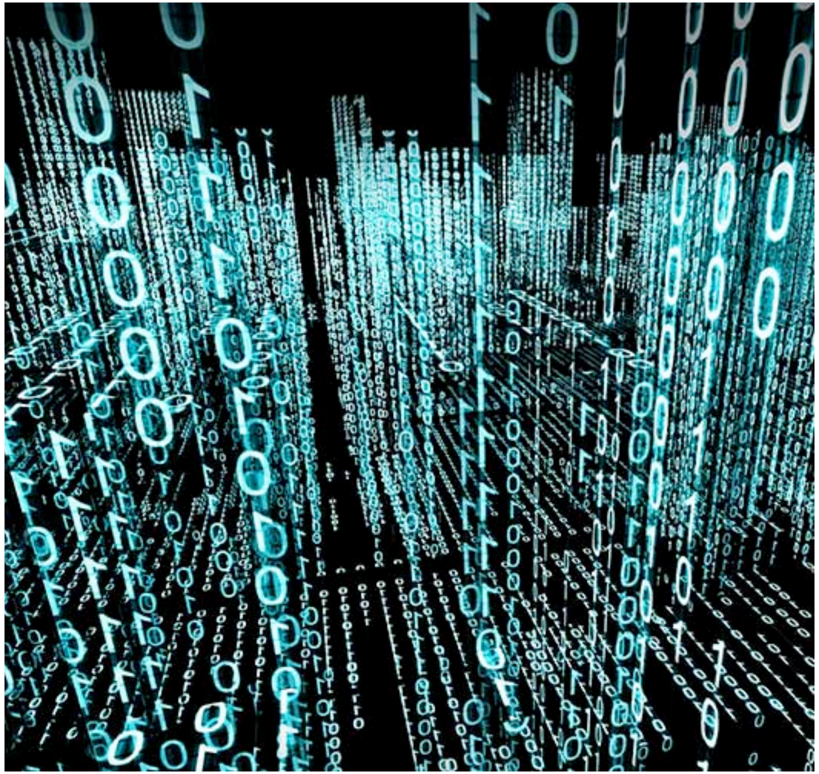


- Control Algorithm:

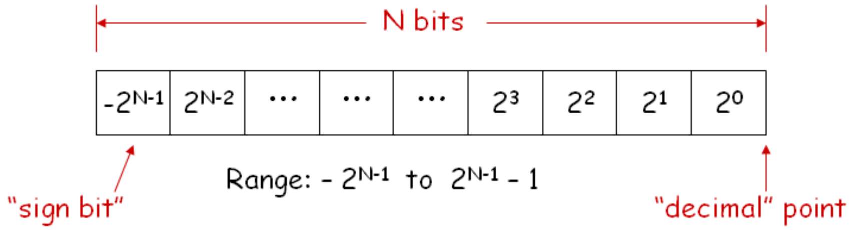
```
repeat n cycles {   // outer (i) loop
          repeat n cycles{    // inner (j) loop
                    shiftA, selectSum, shiftHI
          }
          shiftB, shiftHI, shiftLOW, reset
}
```

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

Signed Multipliers



Combinational Multiplier (signed!)

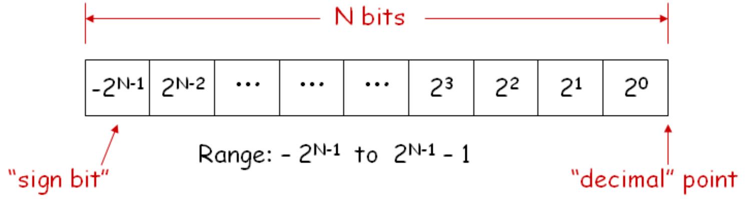


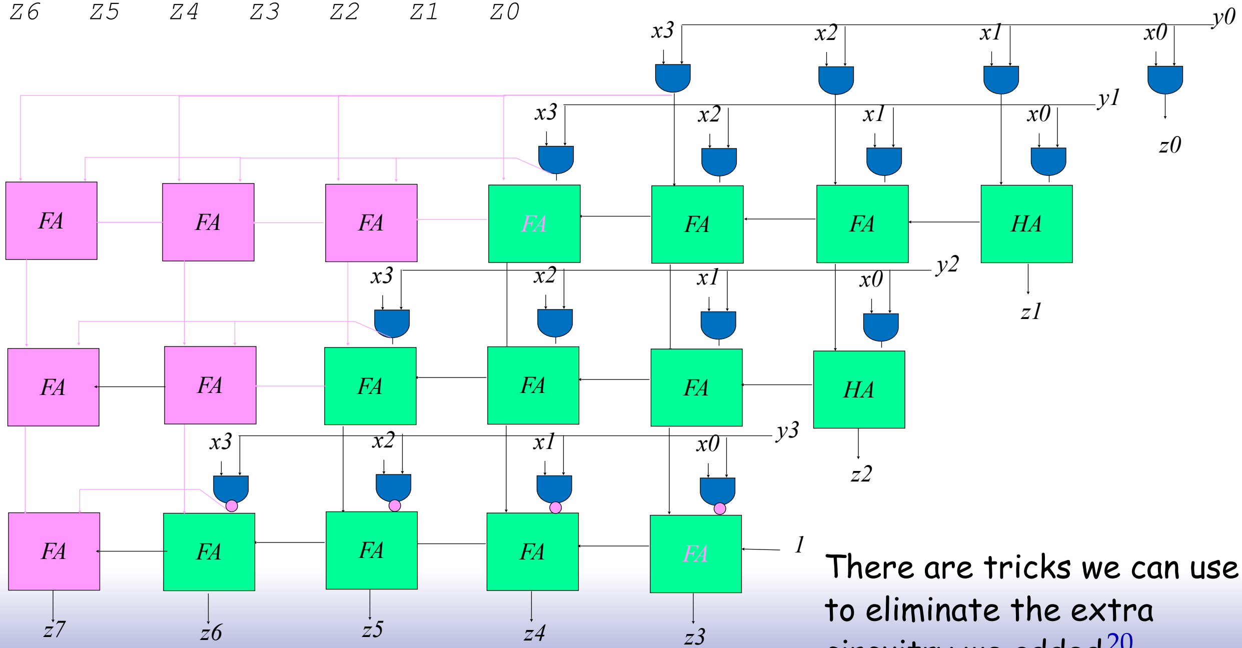
```
                              X * Y = (-3) * (-2)

(-3)                                    101           (X)
(-2)                                    110           (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

$$\begin{array}{ccccccccc}
 & & & & & X3 & X2 & X1 & X0 \\
 & & & & * & Y3 & Y2 & Y1 & Y0 \\
\hline
 & X3Y0 & X3Y0 & X3Y0 & X3Y0 & X3Y0 & X2Y0 & X1Y0 & X0Y0 \\
+ & X3Y1 & X3Y1 & X3Y1 & X3Y1 & X2Y1 & X1Y1 & X0Y1 & \\
+ & X3Y2 & X3Y2 & X3Y2 & X2Y2 & X1Y2 & X0Y2 & & \\
- & X3Y3 & X3Y3 & X2Y3 & X1Y3 & X0Y3 & & & \\
\hline
 & Z7 & Z6 & Z5 & Z4 & Z3 & Z2 & Z1 & Z0
\end{array}$$

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).

$$\begin{array}{cccccccccl}
 & X3Y0 & X3Y0 & X3Y0 & X3Y0 & X3Y0 & X2Y0 & X1Y0 & X0Y0 & \\
+ & & & & & 1 & & & & \\
+ & X3Y1 & X3Y1 & X3Y1 & X3Y1 & X2Y1 & X1Y1 & X0Y1 & & \\
+ & & & & 1 & & & & & \\
+ & X3Y2 & X3Y2 & X3Y2 & X2Y2 & X1Y2 & X0Y2 & & & \\
+ & & & 1 & & & & & & \\
+ & \overline{X3Y3} & \overline{X3Y3} & \overline{X2Y3} & \overline{X1Y3} & \overline{X0Y3} & & & & \left.\right\} -B = \sim B + 1 \\
+ & & & & & 1 & & & & \\
+ & & 1 & & & & & & & \\
- & & 1 & 1 & 1 & 1 & & & &
\end{array}$$

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

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

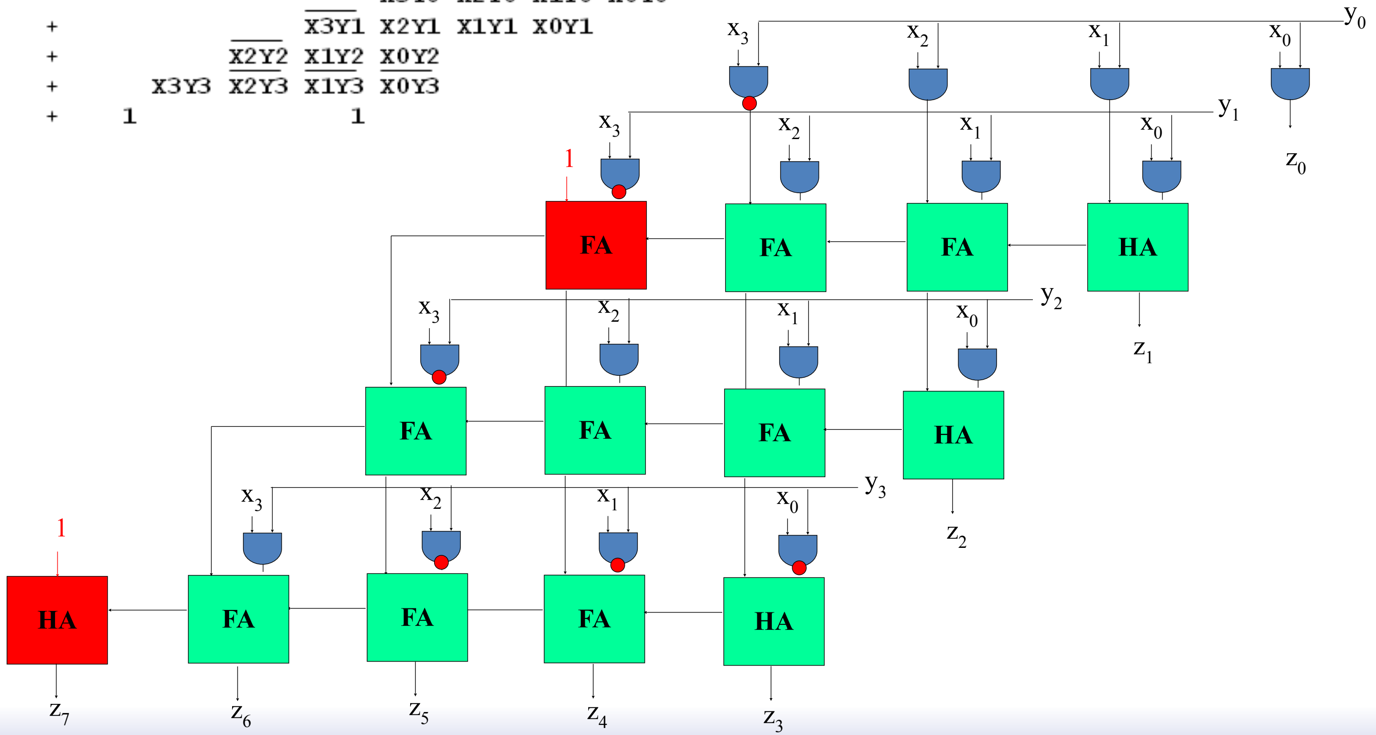
Step 4: finish computing the constants…

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

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

2's Complement Multiplication

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



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*

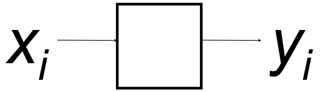
Constant Multiplication

- *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:

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

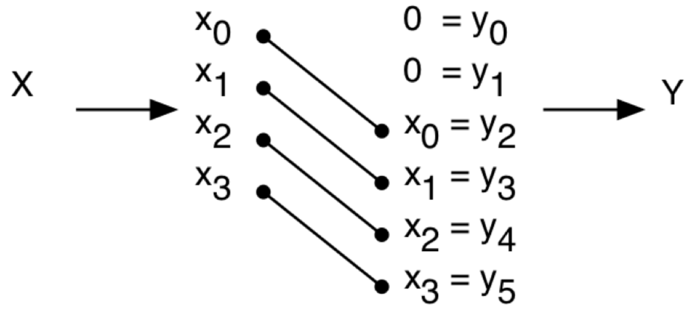


where α is an application dependent constant that is hard-wired into the circuit.

- 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

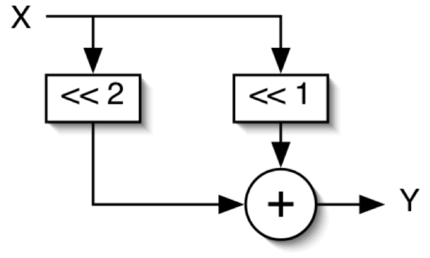


- What about division?

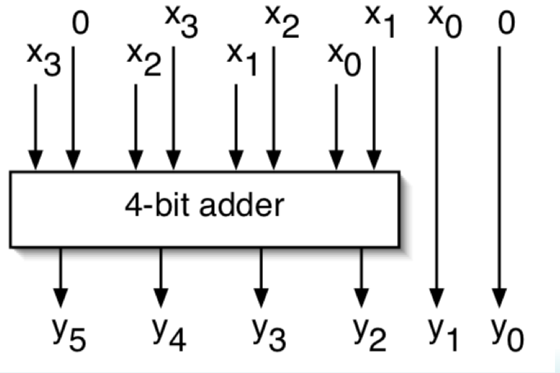
- What about multiplication by non- powers of 2?

Multiplication by a Constant

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

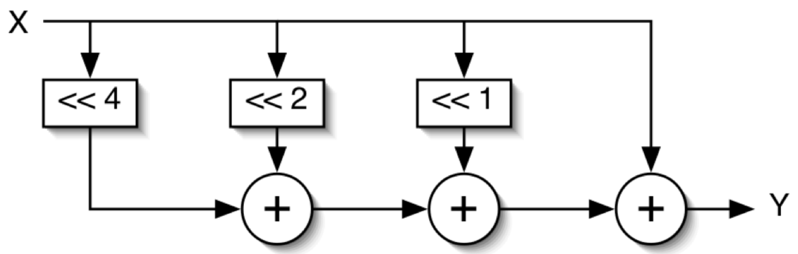


  - Details:



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 addition) helps reduce the delay and cost, and using balanced trees helps with delay.

- Is there a way to further reduce the number of adders (and thus the cost and delay)?

Multiplication using Subtraction

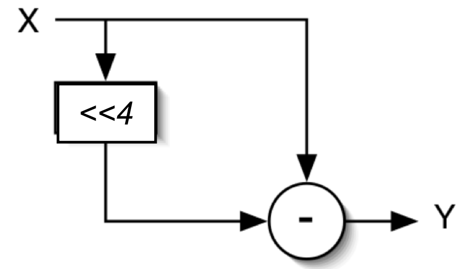
- *Subtraction is approximately the same cost and delay as addition.*
- Consider C*X where C is the constant value 15_{10} = 01111.

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

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

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

  where $\overline{1}$ 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:

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

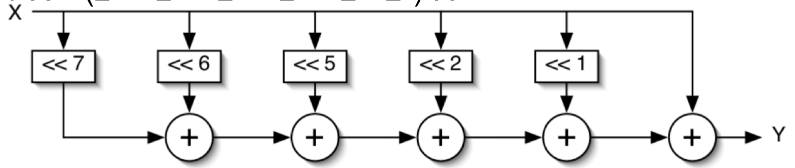
  $0010111 = 23$
  $001100\bar{1}$
  $010\bar{1}00\bar{1} = 32 - 8 - 1$

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

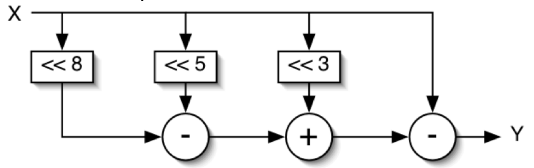
- Can we further simplify the multiplier circuits?

"Constant Coefficient Multiplication" (KCM)

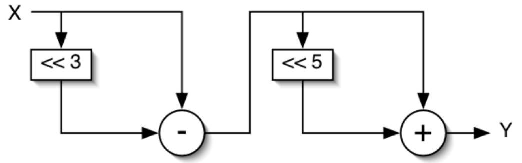
Binary multiplier: $Y = 231*X = (2^7 + 2^6 + 2^5 + 2^2 + 2^1+2^0)*X$



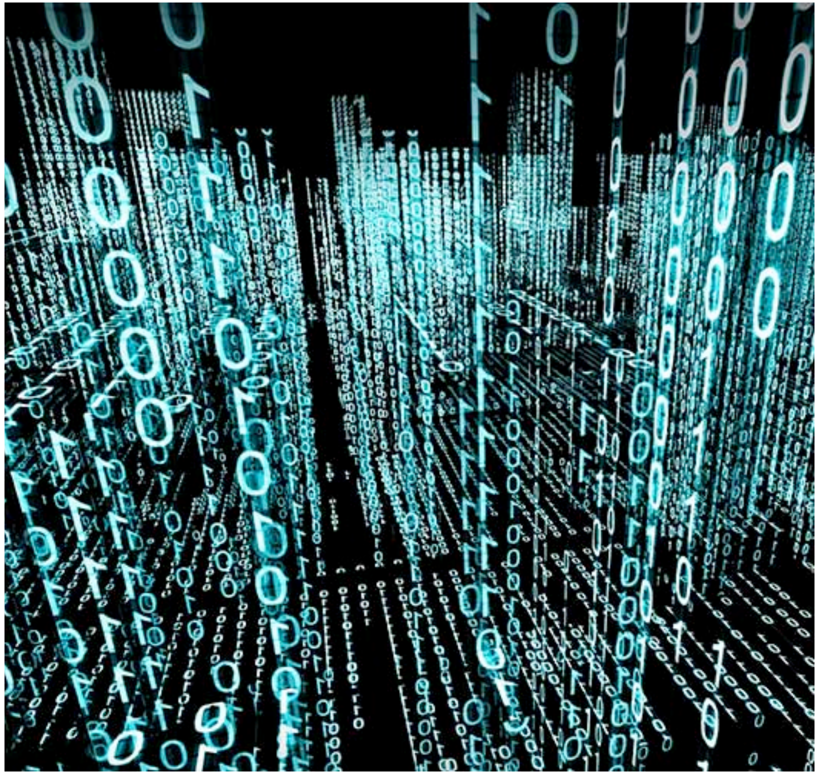
- CSD helps, but the multipliers are limited to shifts followed by adds.
  - CSD multiplier: $Y = 231*X = (2^8 - 2^5 + 2^3 - 2^0)*X$



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

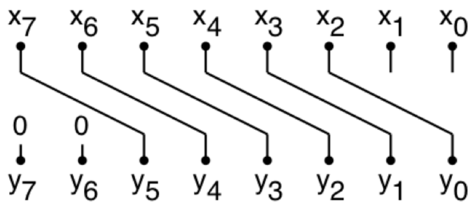


- No simple algorithm exists to determine the optimal KCM representation.
- Most use exhaustive search method.

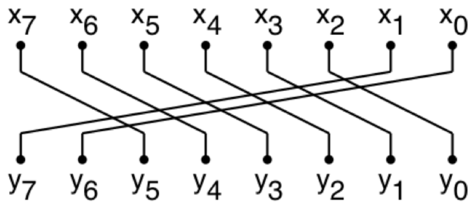


Shifters

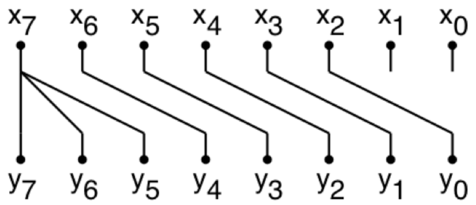
Fixed Shifters / Rotators Defined


x_7 x_6 x_5 x_4 x_3 x_2 x_1 x_0
0 0
y_7 y_6 y_5 y_4 y_3 y_2 y_1 y_0

Logical Shift


x_7 x_6 x_5 x_4 x_3 x_2 x_1 x_0
y_7 y_6 y_5 y_4 y_3 y_2 y_1 y_0

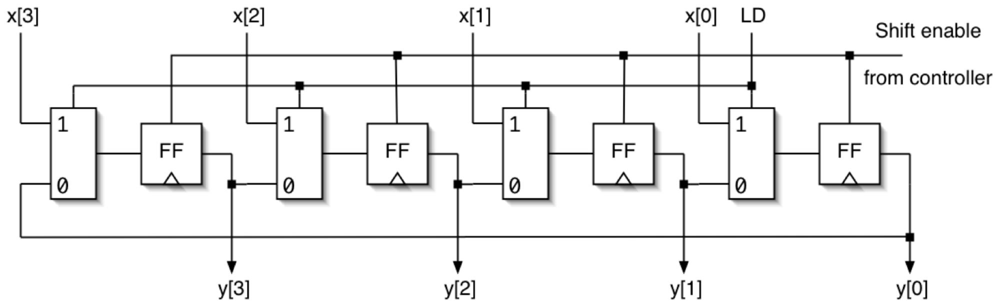
Rotate


x_7 x_6 x_5 x_4 x_3 x_2 x_1 x_0
y_7 y_6 y_5 y_4 y_3 y_2 y_1 y_0

Arithmetic Shift

Variable Shifters / Rotators

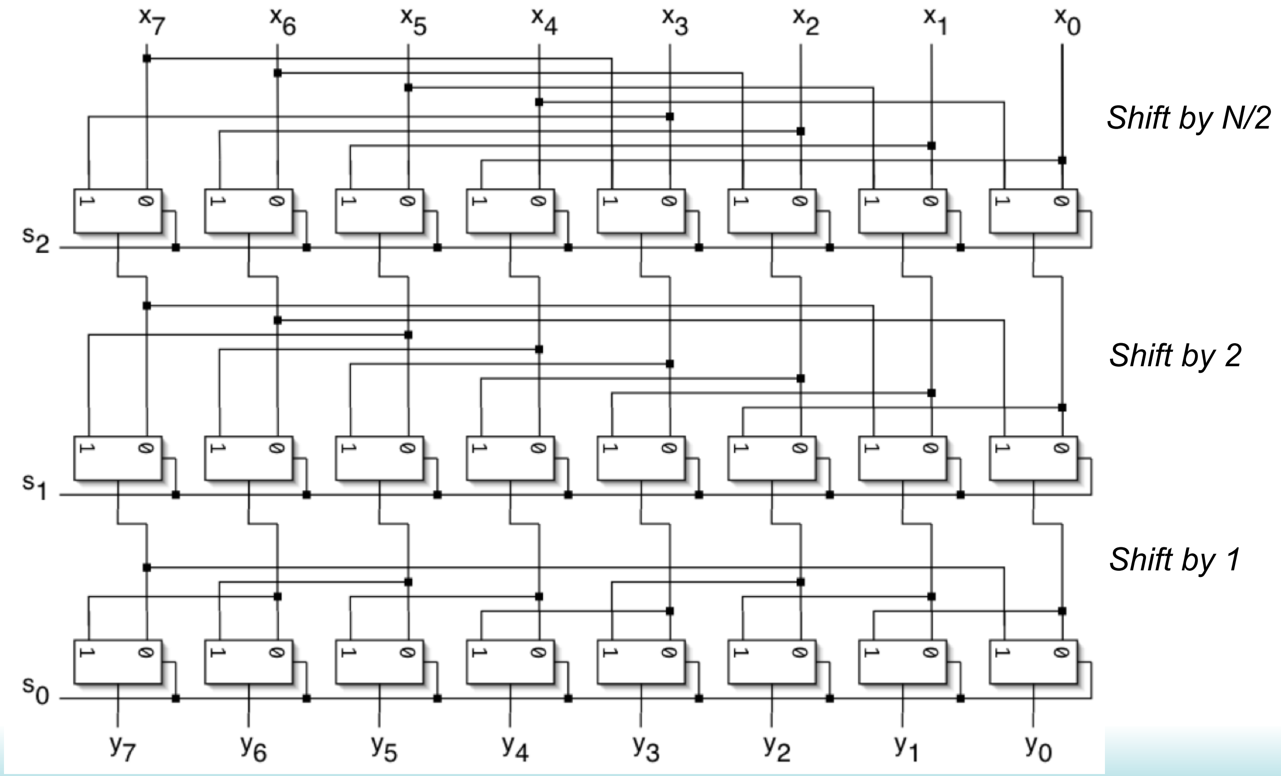
- Example: X >> S, where S is unknown when we synthesize the circuit.
- Uses: shift instruction in processors (ARM includes a shift on every instruction), floating-point arithmetic, division/multiplication by powers of 2, etc.
- One way to build this is a simple shift-register:
  a) Load word, b) shift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

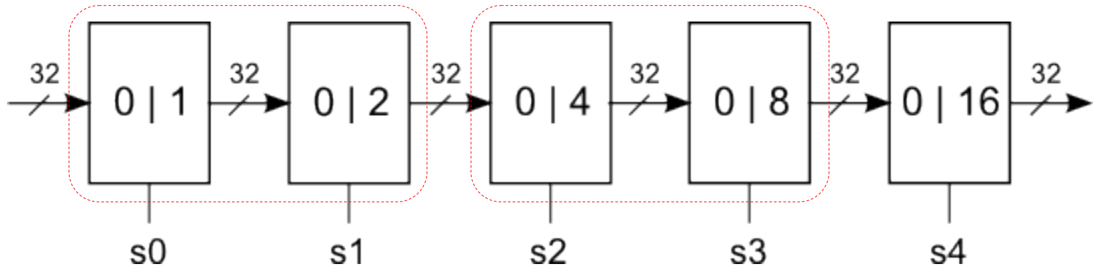
Log Shifter / Rotator

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



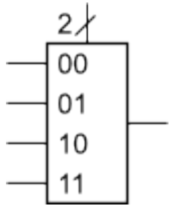
x_7 x_6 x_5 x_4 x_3 x_2 x_1 x_0
Shift by N/2
s_2
Shift by 2
s_1
Shift by 1
s_0
1
0
y_7 y_6 y_5 y_4 y_3 y_2 y_1 y_0

LUT Mapping of Log shifter

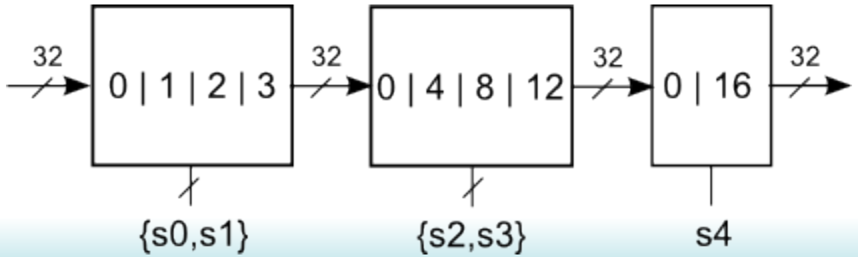


Efficient with 2to1 multiplexors, for instance, 3LUTs.

Virtex6 has 6LUTs. Naturally makes 4to1 muxes:



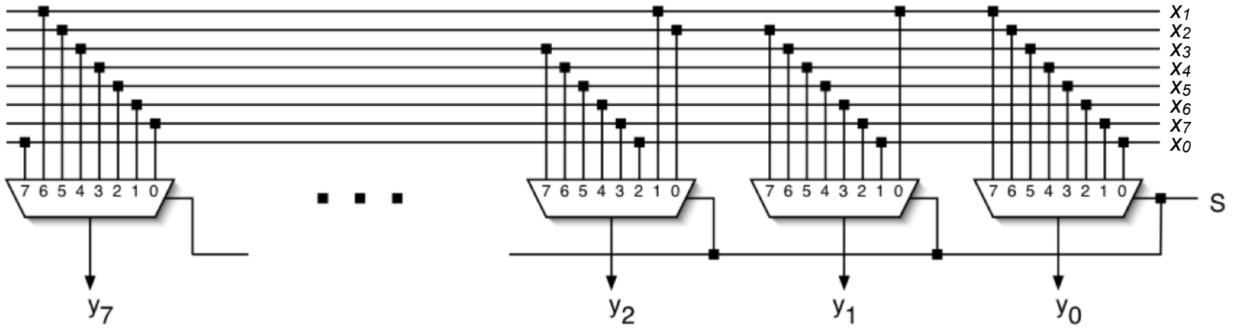
Reorganize shifter to use 4to1 muxes.



Final stage uses F7 mux

"Improved" Shifter / Rotator

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

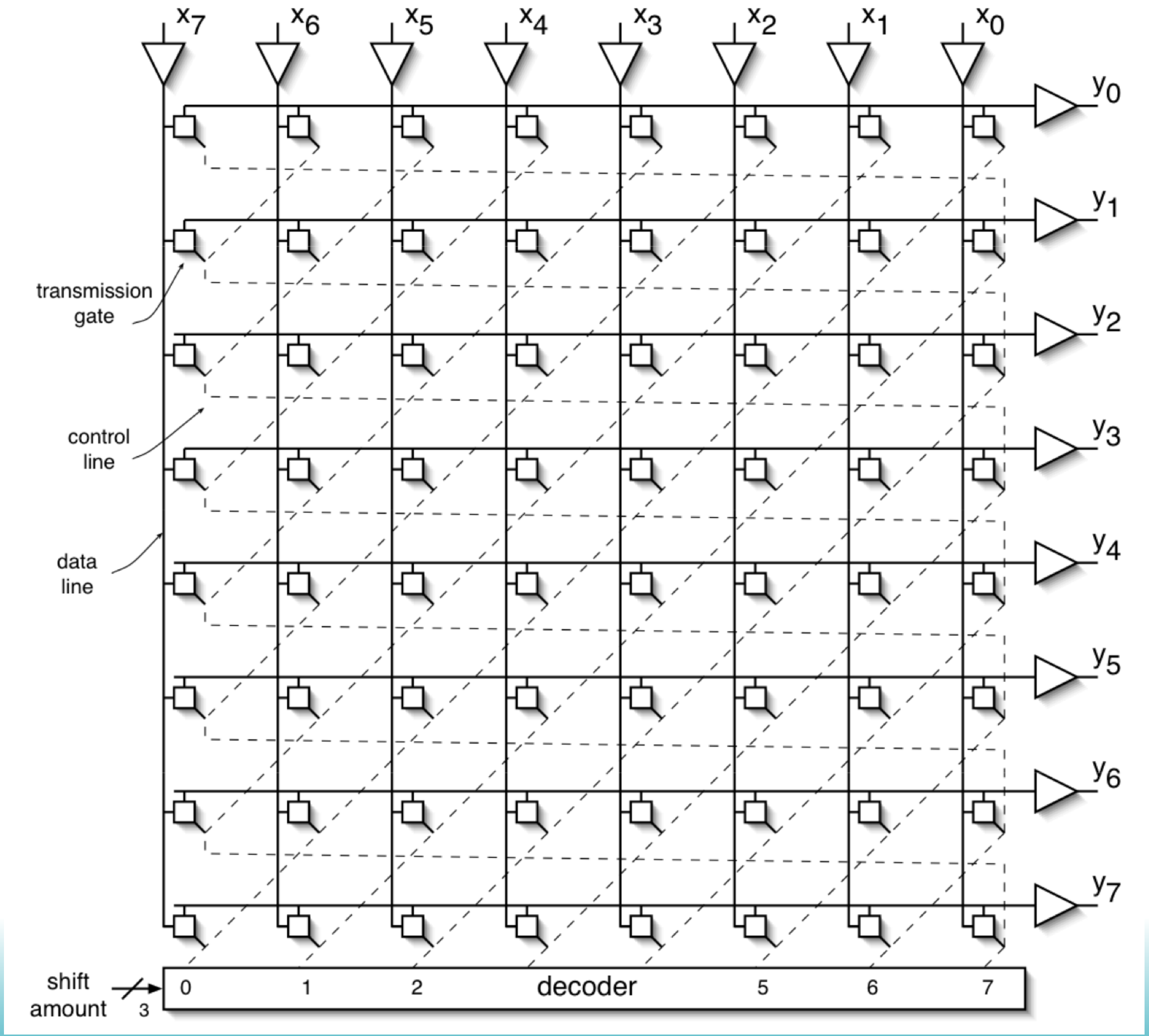


- What is the delay of these big muxes?
- Look a transistor-level implementation?

Left-shift with rotate

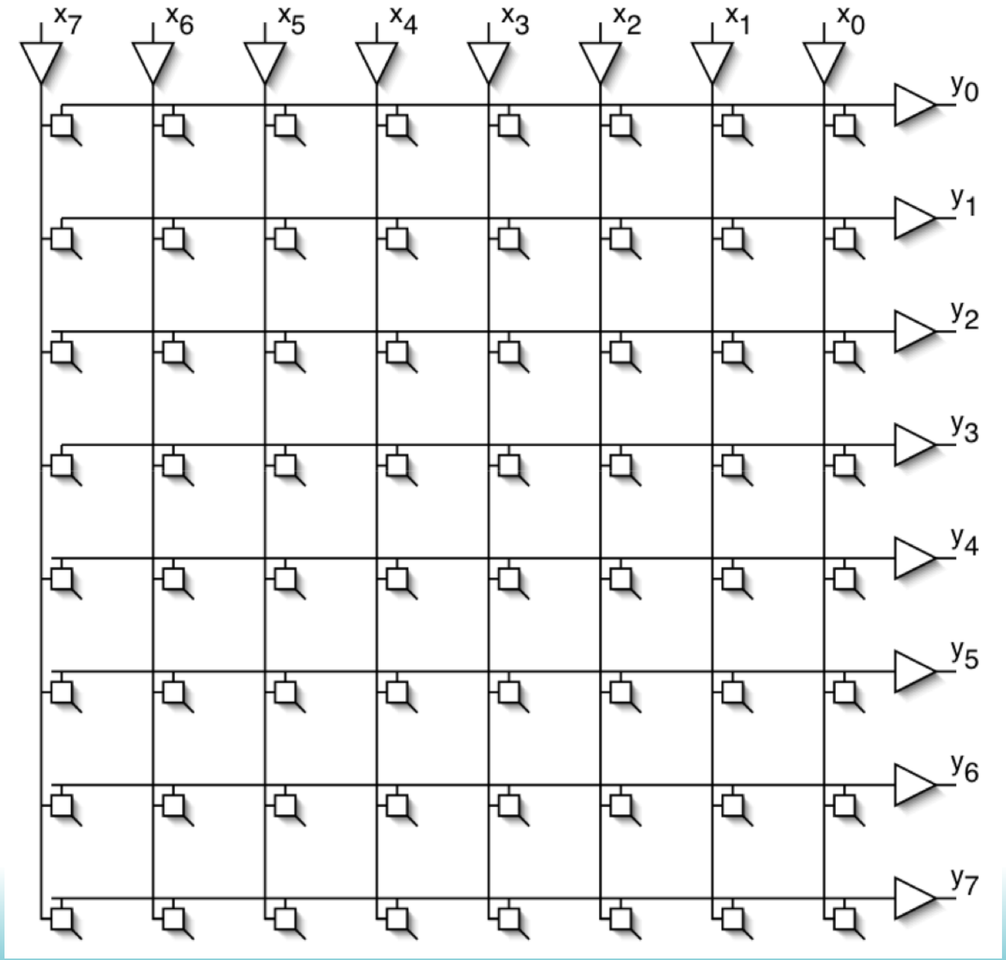
Barrel Shifter

- Cost/delay?



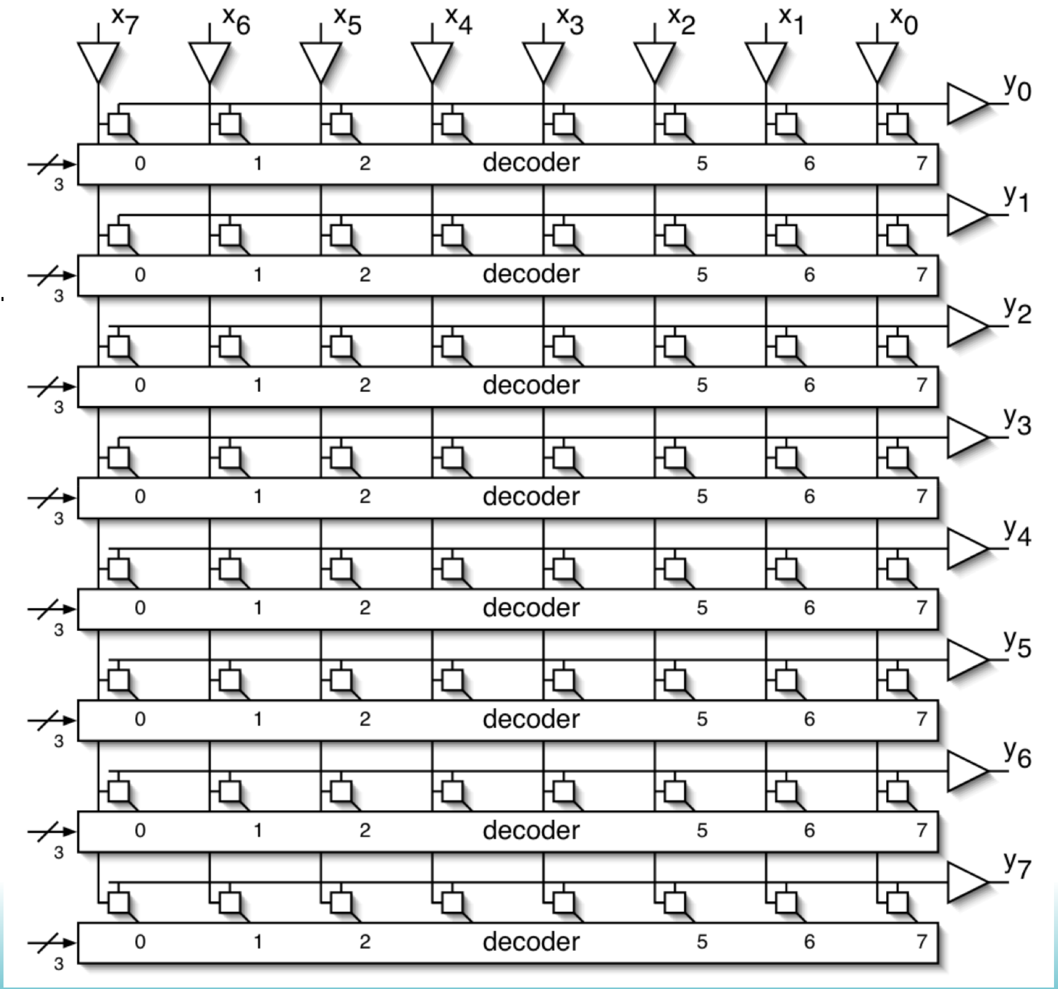
Connection Matrix

- 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-many connections.
- Commonly used in communication hardware (switches, routers).



x_7
x_6
x_5
x_4
x_3
x_2
x_1
x_0
y_0
y_1
y_2
y_3
y_4
y_5
y_6
y_7
3
0
1
2
decoder
5
6
7