EECS151 : Introduction to Digital Design and ICs

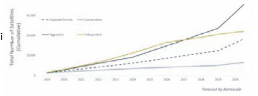
Lecture 19 – Multipliers, Shifters

Bora Nikolić

Space Jam: Efforts Launched to Corral Orbital Junk

October 28, 2021, EETimes - The quickening pace of satellite launches into low-earth orbit for applications such as global internet coverage is creating a growing space congestion and debris problem. Satellite operators are now required to include additional propellent to de-orbit satellites once their lifetime expires.

Still, decades-worth of upper stages used to push payloads to intended orbits continue to coast through the solar system. In one recent example, the expended thi stage of the Apollo Saturn V—most likely from Apollo 12 launched in October 1969—showed up near Earth in its endless heliocentric orbit.



Growth in satellites

Image: Astroscale

Review

- Adders
  - Carry is in the adder critical path
  - Mirror adders cells are commonly found in libraries
  - Ripple-carry adder is the least complex, lowest energy
  - Carry-bypass, carry-select are usually faster than ripple-carry for bitwidths > 8
- Multipliers
  - Shift-and-add is the most compact

Administrivia

- Homework 7 due this week
- Homework 8 due next week
  - In scope for midterm
- All labs need to be checked off by this week!
- Projects (ASIC and FPGA) started, first check point this week
- Midterm 2 is on November 4 at 7pm
  - Review session tonight at 7pm

Multipliers

"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?

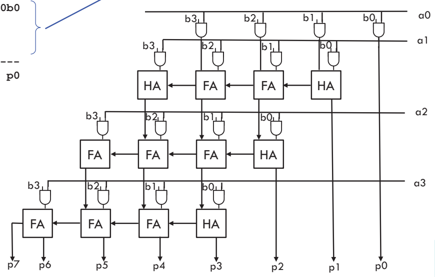
```
  1101
x 0101
```

Unsigned Parallel Multiplier

Parallel (Array) Multiplier

```
                         a3   a2   a1   a0     multiplicand
                       * b3   b2   b1   b0     multiplier
                       --------------------
                        a3b0 a2b0 a1b0 a0b0
+                  a3b1 b2a1 a1b1 a0b1
+             a3b2 a2b2 a1b2 a0b2
+        a3b3 a2b3 a1b3 a0b3
-----------------------------------------
   p7   p6   p5   p4   p3   p2   p1   p0
```

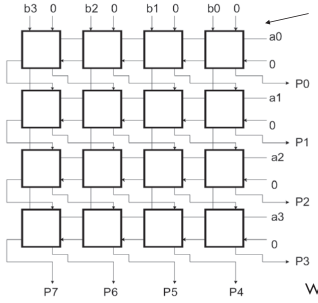
Partial products, one for each bit in multiplier (each bit needs just one AND gate)



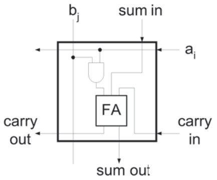
- Performance: What is the critical path?

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

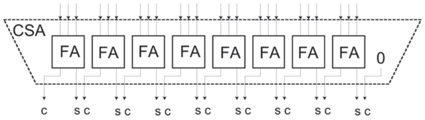
- Speeding up multiplication is a matter of speeding up the summing of the partial products.
- "Carry-save" addition can help.
- Carry-save addition passes (saves) the carries to the output, rather than propagating them.

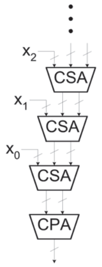
Example: sum three numbers,
$3_{10} = 0011$, $2_{10} = 0010$, $3_{10} = 0011$

```
     3₁₀  0011
  +  2₁₀  0010
   c      0100  =  4₁₀    } carry-save add
   s      0001  =  1₁₀
  +  3₁₀  0011             carry-save add
   c      0010  =  2₁₀
   s      0110  =  6₁₀    carry-propagate add
          1000  =  8₁₀
```

- 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

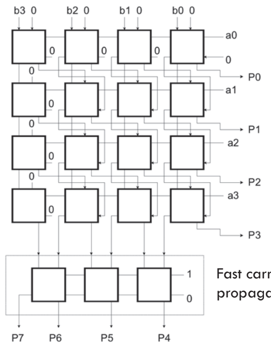
Carry-Save Circuits



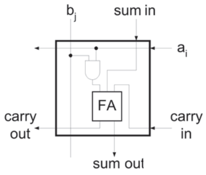


- When adding sets of numbers, carry-save can be used on all but the final sum.
- Standard adder (carry propagate) is used for final sum.
- Carry-save is fast (no carry propagation) and inexpensive (full adders)

Array Multiplier Using Carry-Save Addition



Fast carry-propagate adder

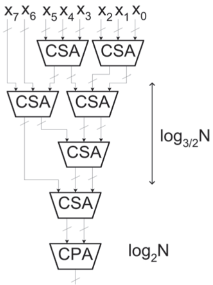


- What is the critical path?

Carry-Save Addition

CSA is associative and commutative. For example:

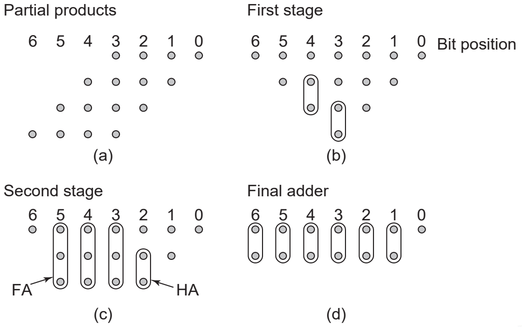
$$(((X_0 + X_1) + X_2) + X_3) = ((X_0 + X_1) + (X_2 + X_3))$$



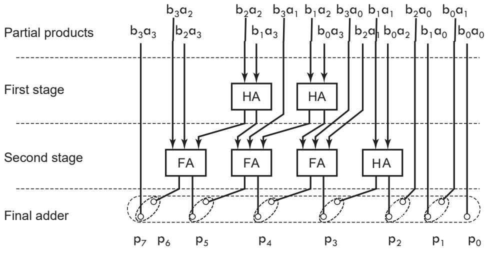
- A balanced tree can be used to reduce the logic delay
- It doesn't matter where you add the carries and sums, as long as you eventually do add them
- This structure is the basis of the ***Wallace Tree Multiplier***
- Partial products are summed with the CSA tree. Fast adder (ex: CLA) is used for final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

Wallace-Tree Multiplier

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



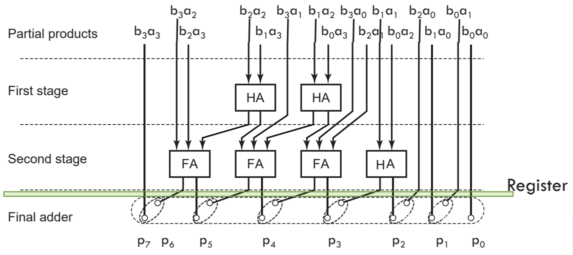
Wallace-Tree Multiplier

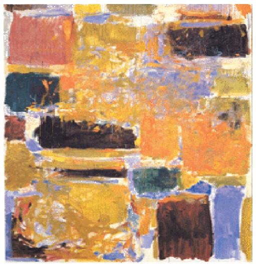


Note: Wallace tree is often slower than an array multiplier in FPGAs (which have optimized carry chains)

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

$$\begin{array}{rrrrrrrr}
 & & & a_{N-1}\ldots & a_2 & a_1 & a_0 & \leftarrow \textit{Multiplicand} \\
 & & \text{X} & b_{N-1}\ldots & b_2 & b_1 & b_0 & \leftarrow \textit{Multiplier} \\
\hline
 & & & a_{N-1}b_0\ldots & a_2b_0 & a_1b_0 & a_0b_0 & \\
 & & a_{N-1}b_1\ldots & a_2b_1 & a_1b_1 & a_0b_1 & & \textit{N partial} \\
 & a_{N-1}b_2\ldots & a_2b_2 & a_1b_2 & a_0b_2 & & & \textit{products (x \{0, 1\}} \\
a_{N-1}b_3\ldots & a_2b_3 & a_1b_3 & a_0b_3 & & & & \\
\hline
 & & \ldots & & & a_1b_0+a_0b_1 & a_0b_0 & \leftarrow \textit{Product}
\end{array}$$

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

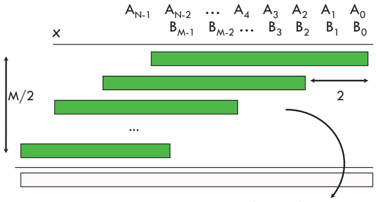
- N, if B = 000…0
- 0, if B = 111…1
- N/2 on the average

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 beginnings 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!


A_{N-1} A_{N-2} … A_4 A_3 A_2 A_1 A_0
x B_{M-1} B_{M-2} … B_3 B_2 B_1 B_0
M/2
2
…

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

$B_{K+1,K}$*A = 0*A → 0
= 1*A → A
= 2*A → 2A (or 4A – 2A)
= 3*A → 4A – A

Booth recoding

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

current bit pair: B_{K+1} B_K — from previous bit pair: B_{K-1}

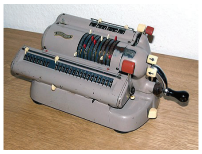
B_{K+1}	B_K	B_{K-1}	action	
0	0	0	add 0	
0	0	1	add A	
0	1	0	add A	
0	1	1	add 2*A	
1	0	0	sub 2*A	
1	0	1	sub A	← -2*A+A
1	1	0	sub A	
1	1	1	add 0	←

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

Example

- Compression tree needs to support subtraction

```
    0111                 A
 x   1010                B
---------
   -01110        10(0) -2A
  -00111         101    -A
 +0111           001    +A
---------
 01000110
```



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

N bits

-2^{N-1}	2^{N-2}	…	…	…	2^3	2^2	2^1	2^0

"sign bit"

Range: -2^{N-1} to $2^{N-1}-1$

"decimal" point

```
                (-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
```

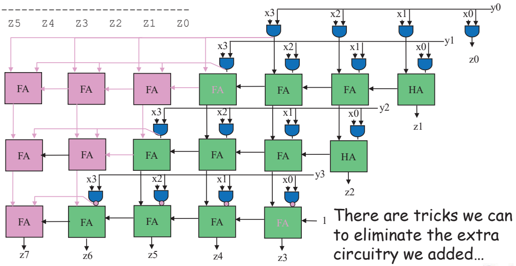
N bits

-2^{N-1}	2^{N-2}	…	…	…	2^3	2^2	2^1	2^0

"sign bit"

Range: -2^{N-1} to $2^{N-1}-1$

"decimal" point



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).

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

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

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

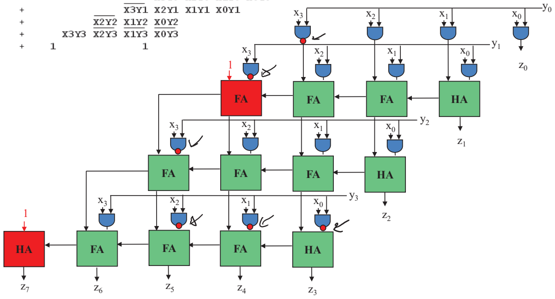
Step 4: finish computing the constants…

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

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

2's Complement Multiplication (Baugh-Wooley)

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



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!
```

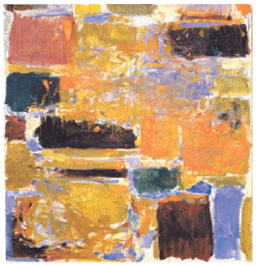
If you want Verilog to treat your operands as signed two's complement numbers, add the keyword signed to your wire or reg declaration:

```
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



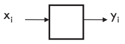
Multiplication with a Constant

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:

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

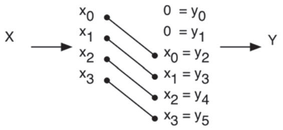


where α is an application-dependent constant that is hard-wired into the circuit.

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

Multiplication by a Constant

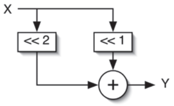
- If the constant C in C*X is a power of 2, then the multiplication is simply a shift of X.
- Ex: 4*X



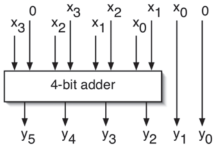
- What about division?
- What about multiplication by non- powers of 2?

Multiplication by a Constant

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

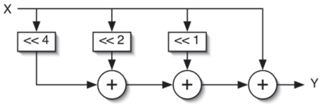


  - Details:



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 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

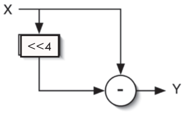
- *Subtraction is the same cost and delay as addition.*
- Consider C*X where C is the constant value $15_{10} = 01111$.

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

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

  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:

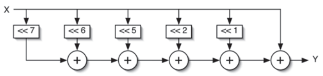
  $011101 = 29$
  $100\bar{1}01 = 32 - 4 + 1$

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

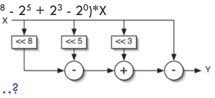
  $0110110 = 54$
  $10\bar{1}10\bar{1}0$
  $100\bar{1}0\bar{1}0 = 64 - 8 - 2$
- Can we further simplify the multiplier circuits?

(K) Constant Coefficient Multiplication (KCM)

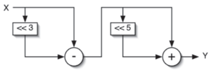
Binary multiplier: $Y = 231*X = (2^7 + 2^6 + 2^5 + 2^2 + 2^1+2^0)*X$



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

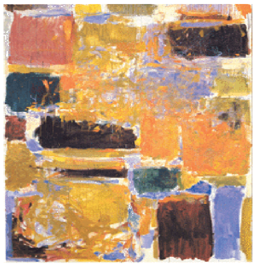


- How about shift/add/shift/add …?
  - KCM multiplier: $Y = 231*X = 7*33*X = (2^3 - 2^0)*(2^5 + 2^0)*X$



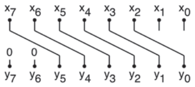
- No simple algorithm exists to determine the optimal KCM representation.
- Most use exhaustive search method.

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

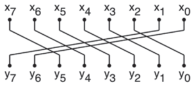


Shifters and Rotators

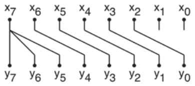
Fixed Shifters / Rotators Defined


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

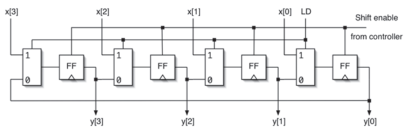

x_7 x_6 x_5 x_4 x_3 x_2 x_1 x_0
y_7 y_6 y_5 y_4 y_3 y_2 y_1 y_0

Arithmetic Shift

Variable Shifters / Rotators

- Example: X >> S, where S is unknown 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:

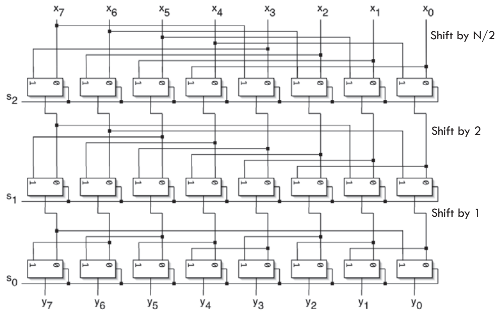
  a) Load word, b) shift enable for S cycles, c) read word.



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

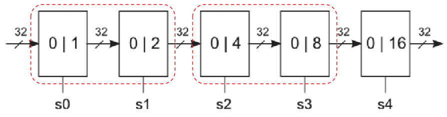
Log Shifter / Rotator

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



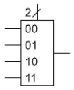
x_7 x_6 x_5 x_4 x_3 x_2 x_1 x_0
Shift by N/2
s_2
Shift by 2
s_1
Shift by 1
s_0
y_7 y_6 y_5 y_4 y_3 y_2 y_1 y_0

LUT Mapping of Log shifter

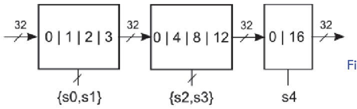


Efficient with 2-to-1 multiplexers, for instance, 3LUTs.

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

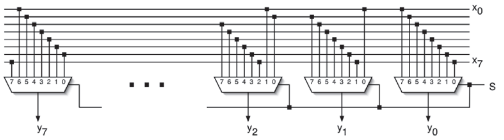


Reorganize shifter to use 4to1 muxes.



Final stage uses F7 mux

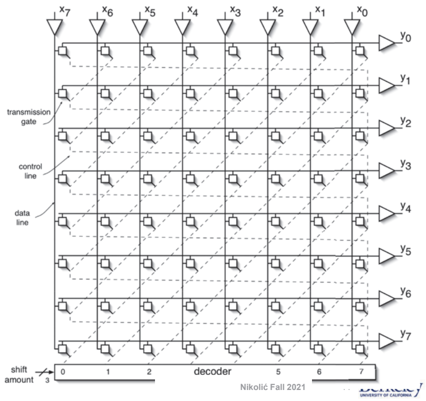
"Improved" Shifter / Rotator



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

Barrel Shifter

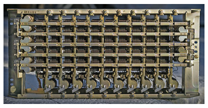
- Cost/delay?
  - (don't forget the decoder)



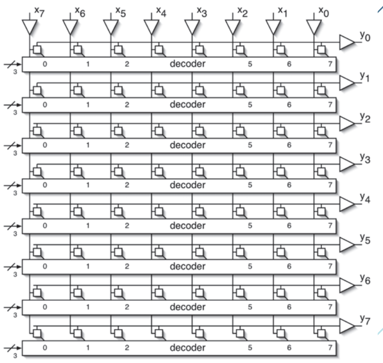
x_7 x_6 x_5 x_4 x_3 x_2 x_1 x_0
y_0 y_1 y_2 y_3 y_4 y_5 y_6 y_7
transmission gate
control line
data line
shift amount
3
0 1 2 decoder 5 6 7

Crossbar Switch

- Nlog(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)



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

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