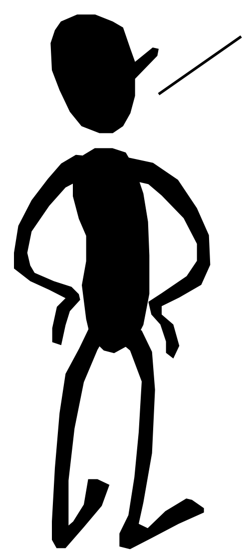
Binary Multipliers

The key trick of multiplication is memorizing a digit-to-digit table…
Everything else was just adding

×	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	8	9
2	0	2	4	6	8	10	12	14	16	18
3	0	3	6	9	12	15	18	21	24	27
4	0	4	8	12	16	20	24	28	32	36
5	0	5	10	15	20	25	30	35	40	45
6	0	6	12	18	24	30	36	42	48	54
7	0	7	14	21	28	35	42	49	56	63
8	0	8	16	24	32	40	48	56	64	72
9	0	9	18	27	36	45	54	63	72	81

×	0	1
0	0	0
1	0	1



Reading: Study Chapter 3.

Have We Forgotten Something?

Our ALU can add, subtract, shift, and perform Boolean functions. But, *even* rabbits know how to multiply…

But, it is a huge step in terms of logic… Including a multiplier unit in an ALU doubles the number of gates used.

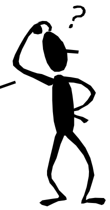
A *good* (compact and high performance) multiplier can also be tricky to design. Here we will give an overview of some of the tricks used.

Binary Multiplication

The "Binary" Multiplication Table

X	0	1
0	0	0
1	0	1

Hey, that looks like an AND gate



Binary multiplication is implemented using the same basic longhand algorithm that you learned in grade school.

$$
\begin{array}{cccccccc}
 & & & & A_3 & A_2 & A_1 & A_0 \\
 & & & \times & B_3 & B_2 & B_1 & B_0 \\
\hline
 & & & & A_3B_0 & A_2B_0 & A_1B_0 & A_0B_0 \\
 & & & A_3B_1 & A_2B_1 & A_1B_1 & A_0B_1 & \\
 & & A_3B_2 & A_2B_2 & A_1B_2 & A_0B_2 & & \\
+ & A_3B_3 & A_2B_3 & A_1B_3 & A_0B_3 & & & \\
\hline
\end{array}
$$

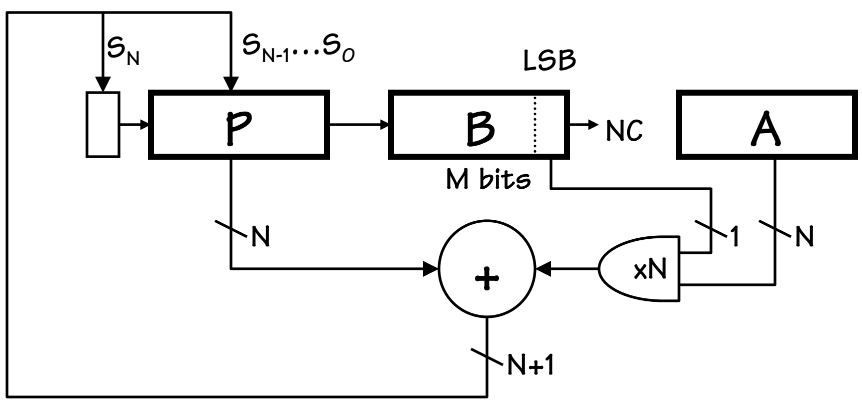
A_jB_i is a "partial product"

Multiplying N-digit number by M-digit number gives (N+M)-digit result

Easy part: forming partial products (just an AND gate since B_i is either 0 or 1)
Hard part: adding M, N-bit partial products

Sequential Multiplier

Assume the multiplicand (A) has N bits and the multiplier (B) has M bits. If we only want to invest in a single N-bit adder, we can build a sequential circuit that processes a single partial product at a time and then cycle the circuit M times:



Init: $P \leftarrow 0$, load A&B

Repeat M times {
  $P \leftarrow P + (B_{LSB}{=}{=}1 \; ? \; A : 0)$
  shift P/B right one bit
}

Done: (N+M)-bit result in P/B

Simple Combinational Multiplier

$t_{PD} = 10 * t_{PD,FA}$

not 16

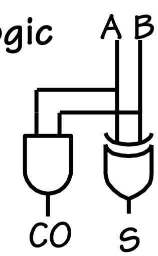
$t_{PD} = (2*(N-1) + N) * t_{PD,FA}$

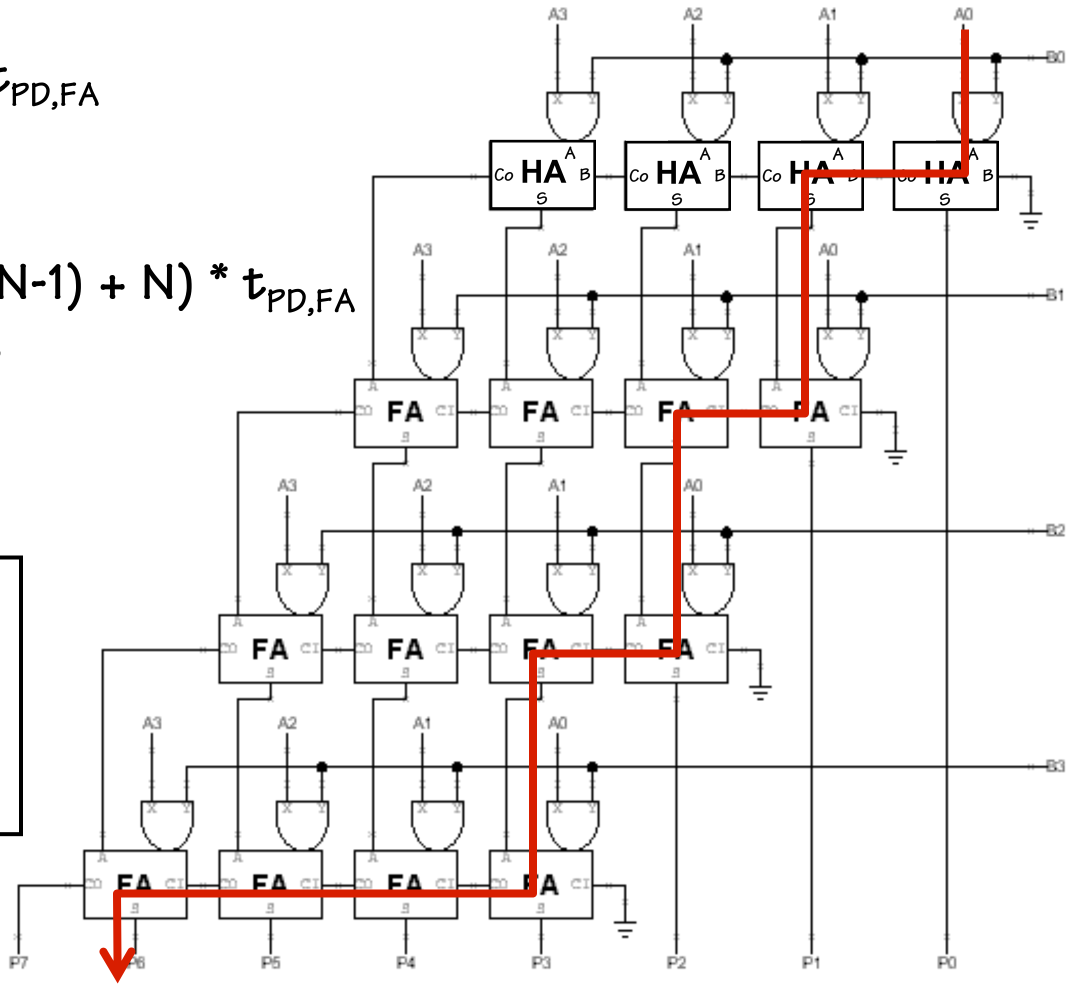
Components

N * HA

N(N-1) * FA

The Logic of a Half-Adder





NB: this circuit only works for nonnegative operands

Carry-Save Combinational Multiplier

Observation: Rather than propagating the sums across each row, the carries can instead be forwarded onto the next column of the following row

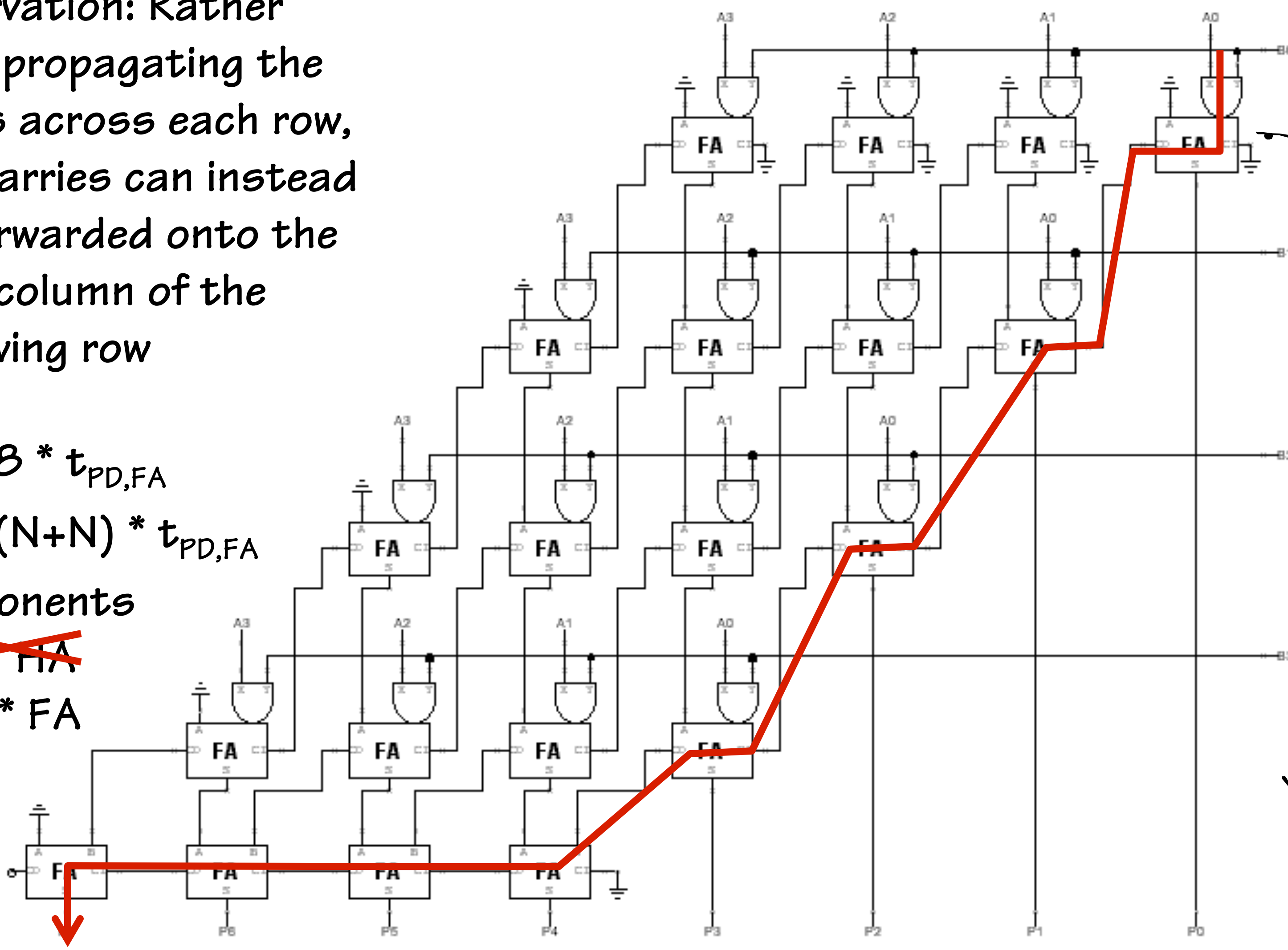
$t_{PD} = 8 * t_{PD,FA}$

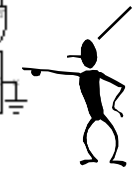
$t_{PD} = (N+N) * t_{PD,FA}$

Components

~~N * HA~~

N^2 * FA



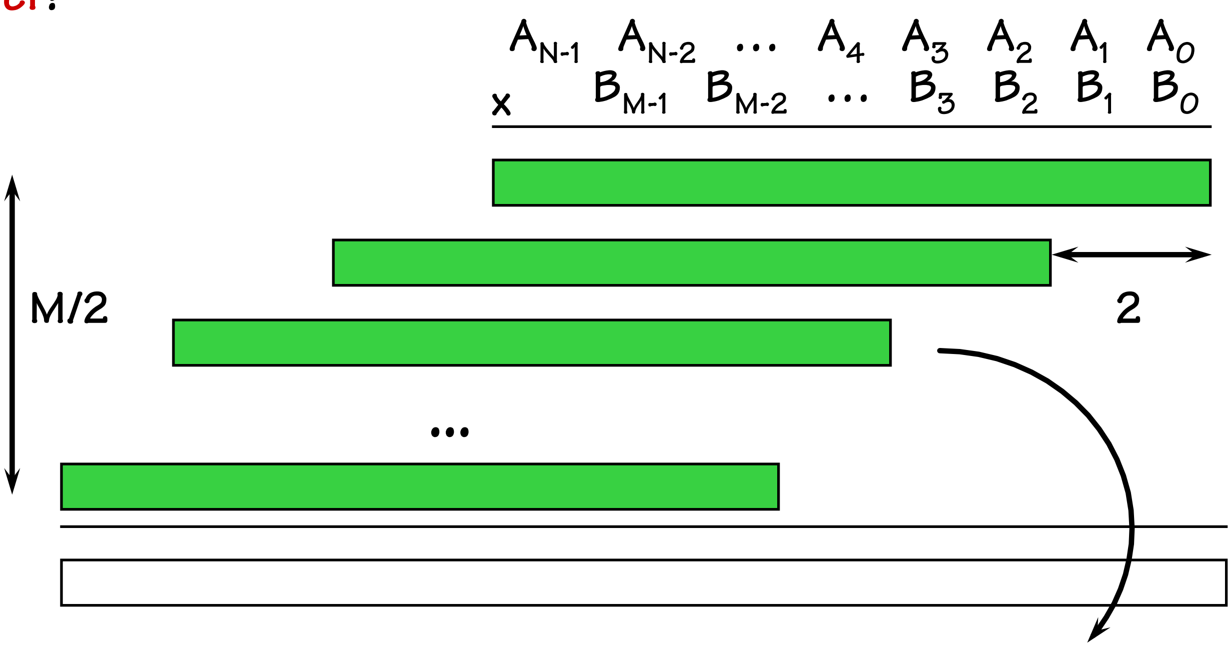


These Adders can be removed, and the AND gate outputs tied directly to the Carry inputs of the next stage.

This small improvement in performance hardly seems worth the effort, however, this design is easier to pipeline.

Higher-Radix Multiplication

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



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

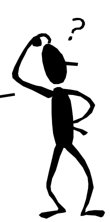
from previous bit pair

B_{2K+1}	B_{2K}	B_{2K-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

$-89 = \boxed{1\ 0}\boxed{1}\boxed{0\ 0}\boxed{1}\boxed{1}\ 1.0$

$$
\begin{array}{lr}
= -1 * 2^0 & (-1) \\
+ 2 * 2^2 & (8) \\
+ (-2) * 2^4 & (-32) \\
+ (-1) * 2^6 & (-64) \\
\hline
 & -89
\end{array}
$$

Hey, isn't that a negative number?



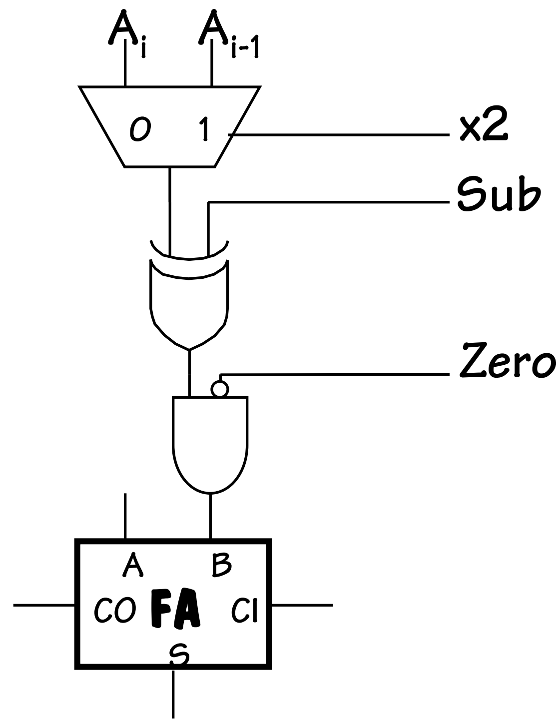
A "1" in this bit means the previous stage needed to add 4*A. Since this stage is shifted by 2 bits with respect to the previous stage, adding 4*A in the previous stage is like adding A in this stage!

Each bit can be considered to have the following weights:

$$W(B_{2K+1}) = -2$$
$$W(B_{2K}) = 1$$
$$W(B_{2K-1}) = 1$$

Booth Recoding

Logic surrounding each basic adder:



- Control lines (x2, Sub, Zero) are shared across each row
- Must handle the "+1" when Sub is 1 (extra half adders in a carry save array)

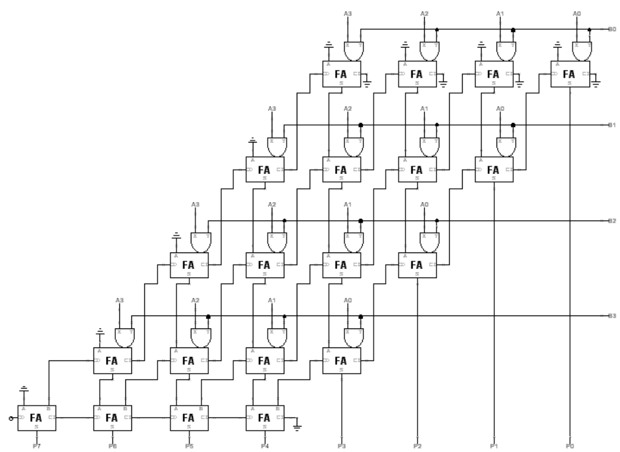
B_{2K+1}	B_{2K}	B_{2K-1}	x2	Sub	Zero
0	0	0	X	X	1
0	0	1	0	0	0
0	1	0	0	0	0
0	1	1	1	0	0
1	0	0	1	1	0
1	0	1	0	1	0
1	1	0	1	1	0
1	1	1	X	X	1

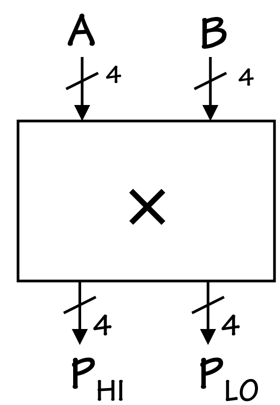
NOTE:

- Booth recoding can be used to implement signed multiplications

Bigger Multipliers

- Using the approaches described we can construct multipliers of arbitrary sizes, by considering every adder at the "bit" level
- We can also, build bigger multipliers using smaller ones





- Considering this problem at a higher-level leads to more "non-obvious" optimizations

Can We Multiply With Less?

- How many operations are needed to multiply 2, 2-digit numbers?

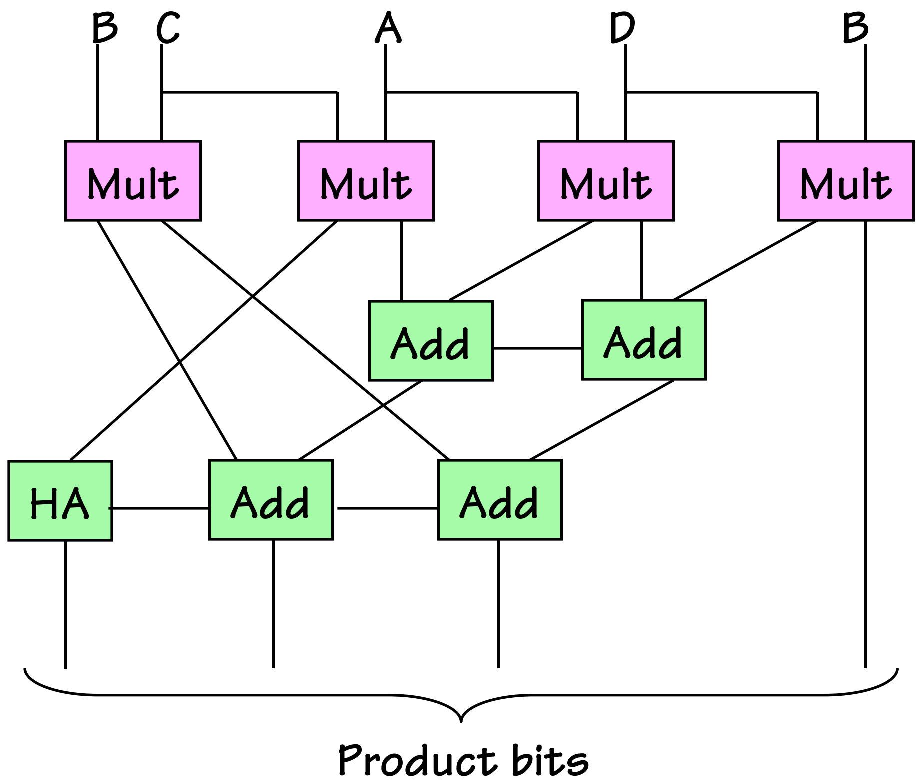
- 4 multipliers
  4 Adders

- This technique generalizes
  - You can build an 8-bit multiplier using 4 4-bit multipliers and 4 8-bit adders
  - $O(N^2 + N) = O(N^2)$

```
      A B
X     C D
---------
      DB
      +
     DA
     + +
     CB
     +
    CA
---------
```

An $O(N^2)$ Multiplier In Logic

The functional blocks would look like



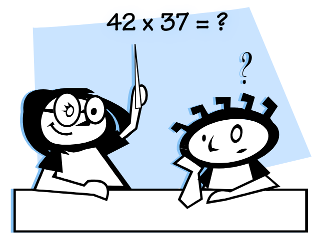
$$
\begin{array}{r}
A\ B \\
\times\quad C\ D \\ \hline
DB \\
DA \\
C\ B \\
CA \\ \hline
\end{array}
$$

A Trick

- The two middle partial products can be computed using a single multiplier and other partial products
- DA + CB = (C + D)(A + B) – (CA + DB)
- 3 multipliers
  8 adders
- This can be applied recursively
  (i.e. applied within each partial product)
- Leads to $O(N^{1.58})$ adders
- This trick is becoming more popular as N grows. However, it is less regular, and the overhead of the extra adders is high for small N

```
     A B
X    C D
--------
      DB
    DA
    C B
  CA
--------
```

Let's Try it By Hand



1) Choose 2, 2 digit numbers to multiply $\quad ab \times cd$

$$42 \times 37$$

2) Multiply $p_1 = a \times c$, $p_2 = b \times d$, $p_3 = (c + d)(a + b)$

$$p_1 = 4 \times 3 = 12,\ p_2 = 2 \times 7 = 14,$$

$$p_3 = (4+2)(3+7) = 60$$

3) Find partial subtracted sum, $SS = p_3 - (p_1 + p_2)$

$$SS = 60 - (12 + 14) = 34$$

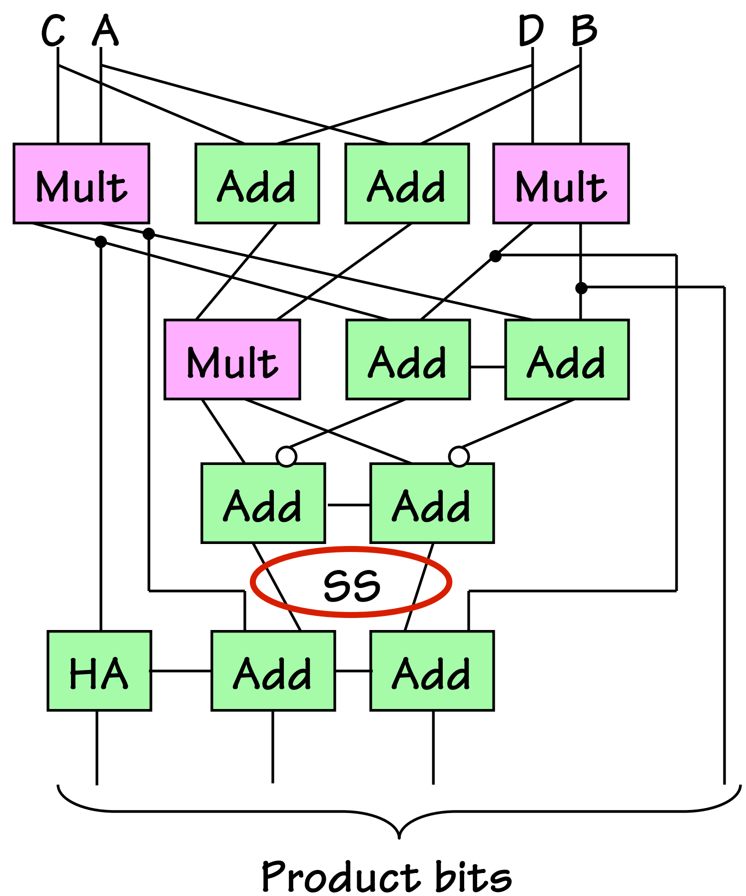
4) Add to find product, $p = 100 {*} p_1 + 10 {*} SS + p_2$

$$p = 1200 + 340 + 14 = 1554 = 42 \times 37$$

An O($N^{1.58}$) Multiplier In Logic

The functional blocks would look like

$$
\begin{array}{r}
A\ B \\
\underline{\times \quad C\ D} \\
DB \\
SS \\
\underline{CA}
\end{array}
$$

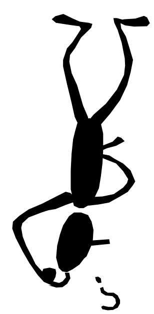


Where

$$SS = (C+D)(A+B) - (CA+DB)$$

Binary Division

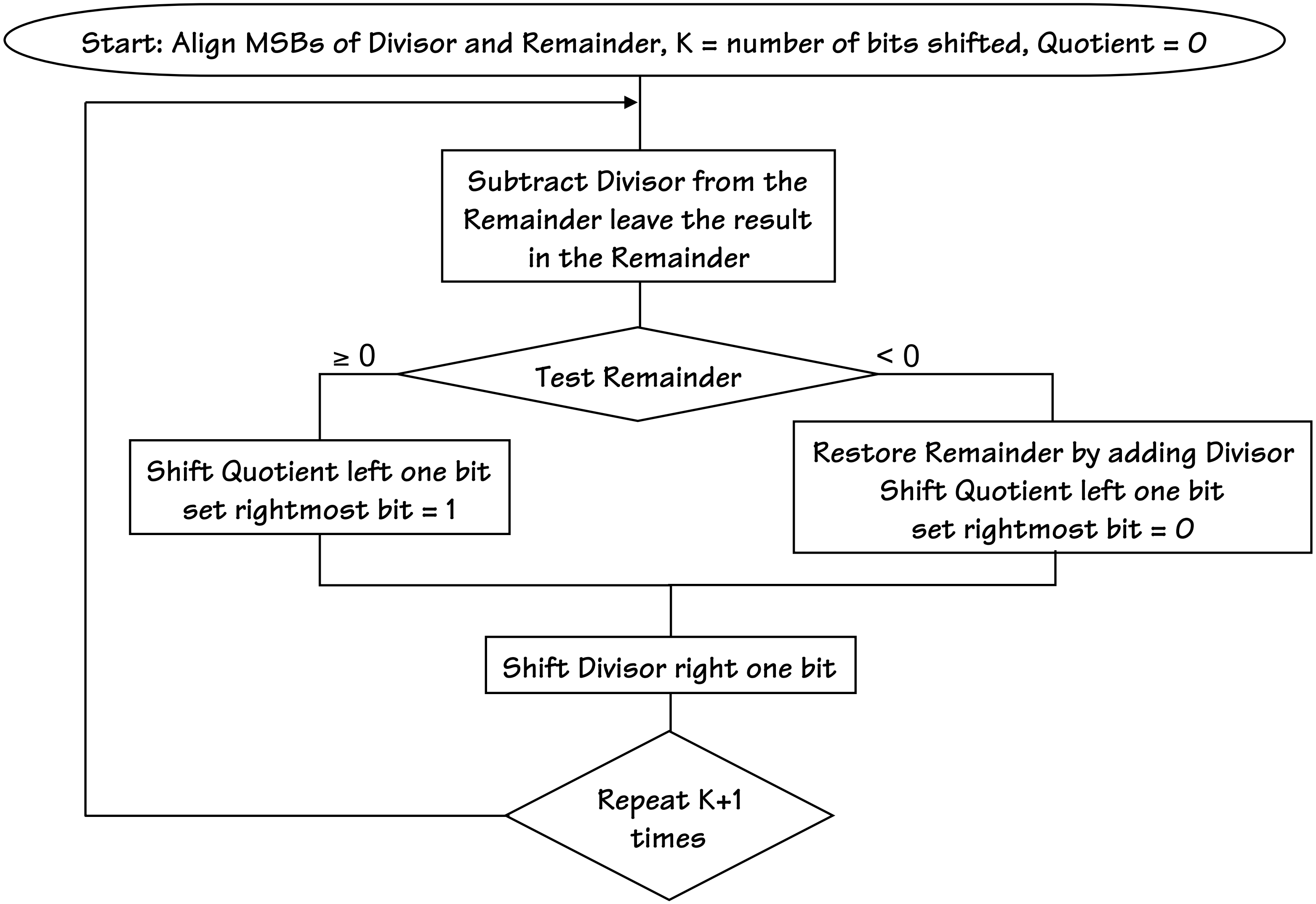
- Division merely reverses the process
  - Rather than adding successively larger partial products, subtract successively smaller divisors
  - When multiplying, we knew which partial products to actually add (based on the whether the corresponding bit was a 0 or a 1)
  - In division, we have to try *both ways*



Multiplication
Upside-down

$$\begin{array}{ccccccccccl}
 & P & P & P & P & P & P & P & P & & \\
- & & D & D & D & D & & & & & Q_3 = 0 \text{ or } 1? \\
- & & & D & D & D & D & & & & Q_2 = 0 \text{ or } 1? \\
- & & & & D & D & D & D & & & Q_1 = 0 \text{ or } 1? \\
- & & & & & D & D & D & D & & Q_0 = 0 \text{ or } 1? \\
\hline
 & & & & & R & R & R & R & &
\end{array}$$

Restoring Division



Division Example

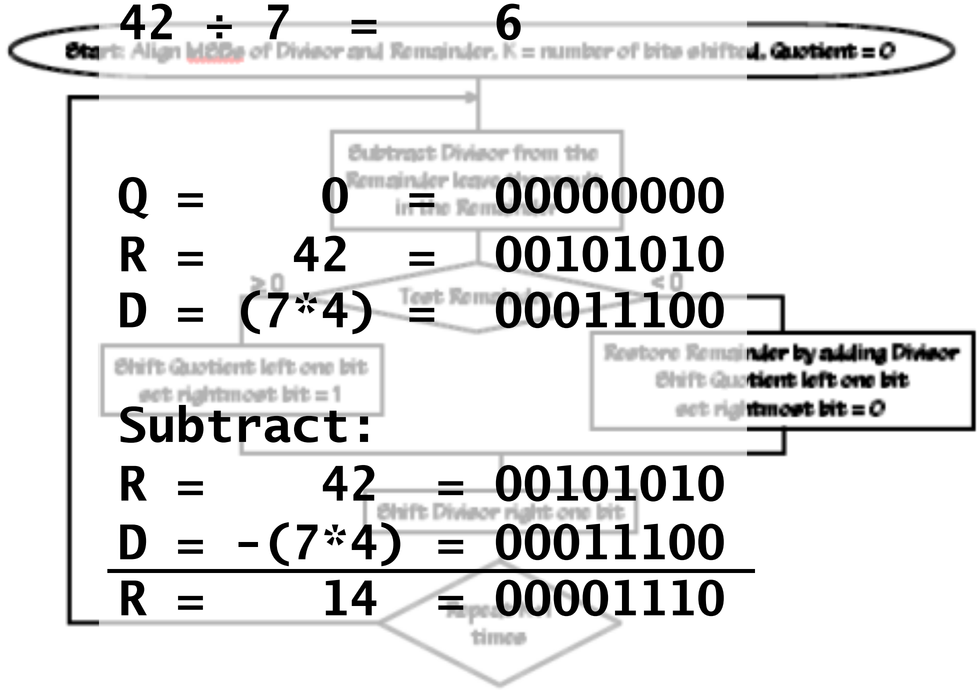
Step 1:

```
 R   D        Q
42 ÷ 7  =     6

Start:
Q =     0   =  00000000
R =    42   =  00101010
D = (7*8)   =  00111000
```

Note: K = 3, so repeat 4 times

```
Subtract:
R =     42   = 00101010
D = -(7*8)   = 00111000
-----------------------
       -14   = 11110001
Restore:
R =     42   = 00101010

Shifts:
             Q = 00000000
             D = 00011100
```

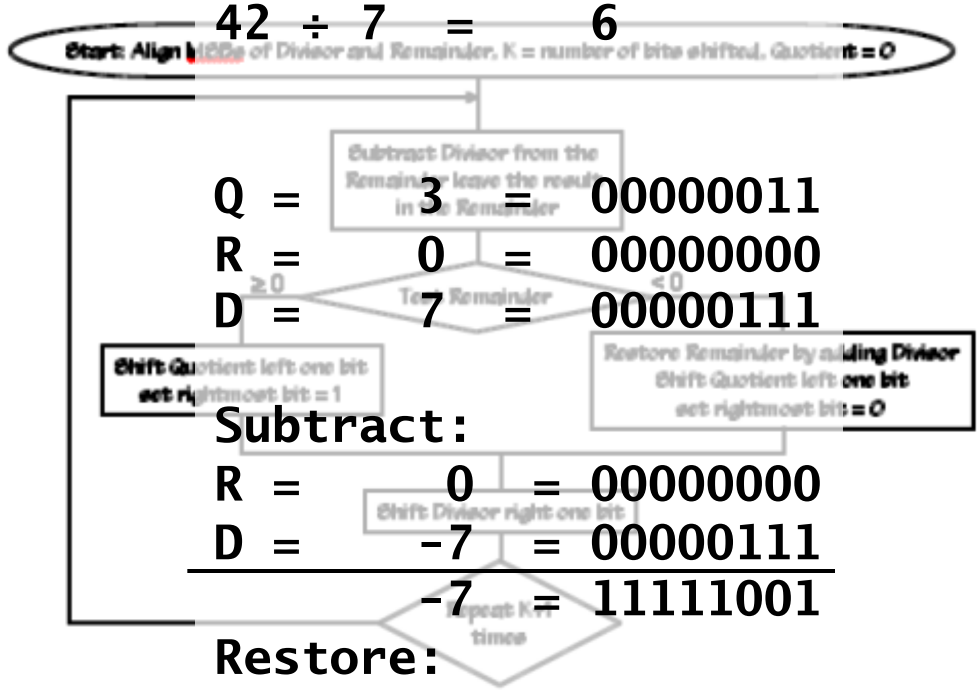
Step 2:




```
 R   D        Q
42 ÷ 7  =     6

Q =     0   =  00000000
R =    42   =  00101010
D = (7*4)   =  00011100

Subtract:
R =     42   = 00101010
D = -(7*4)   = 00011100
-----------------------
R =     14   = 00001110

Shifts:
         Q = 00000001
         D = 00001110
```

Division Example (cont)

```
Step 3:
 R   D       Q
42 ÷ 7  =    6


Q =     1   =  00000001
R =    14   =  00001110
D = (7*2)   =  00001110

Subtract:
R =     14   = 00001110
D = -(7*2)   = 00001110
--------------------------
            0 = 00000000

No Restore
Shifts:
            Q = 00000011
            D = 00000111
```



```
Step 4:
 R   D       Q
42 ÷ 7  =    6


Q =       3  =  00000011
R =       0  =  00000000
D =       7  =  00000111

Subtract:
R =        0  = 00000000
D =       -7  = 00000111
--------------------------
          -7  = 11111001
Restore:
R =        0  = 00000000
Shifts:
            Q = 00000110
            D = 00000011
            R = 00000000
```

Next Time

- We dive into floating point arithmetic