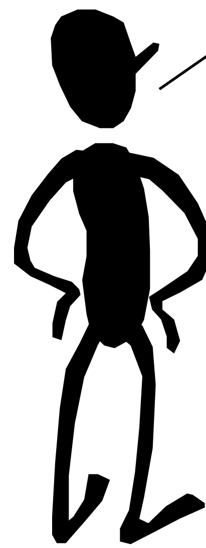
Binary Multipliers

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

×	0	1
0	0	0
1	0	1

×	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



Have We Forgotten Something?

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

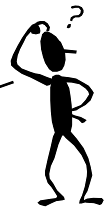
But, it is a huge step in terms of logic… Including a multiplier unit in an ALU doubles the number of gates used.

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



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

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

A_jB_i is a "partial product"

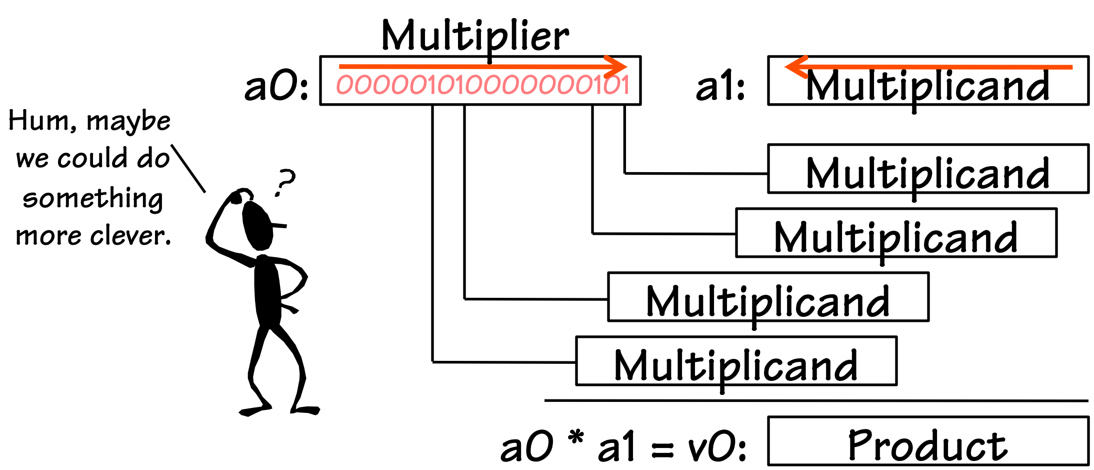
Multiplying N-digit number by M-digit number gives (N+M)-digit result

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

Multiplying in Assembly

One can use this “Shift and Add” approach to write a multiply function in assembly language

```
#   Multiplies unsigned arguments in $a0 and $a1
#   and returns value in $v0 ignoring overflows
multu:  addiu    $v0,$0,0       # zero product register
loop:   andi     $t1,$a0,1      # check low-order bit
        beq      $t1,$0,noadd   # do we need to add?
        add      $v0,$v0,$a1    # add multiplicand to product
noadd:  srl      $a0,$a0,1      # multiplier / 2
        sll      $a1,$a1,1      # 2 * multiplicand
        bne      $a0,$0,loop    # keep adding if there are
        jr       $31
```



Multiplier Unit-Block

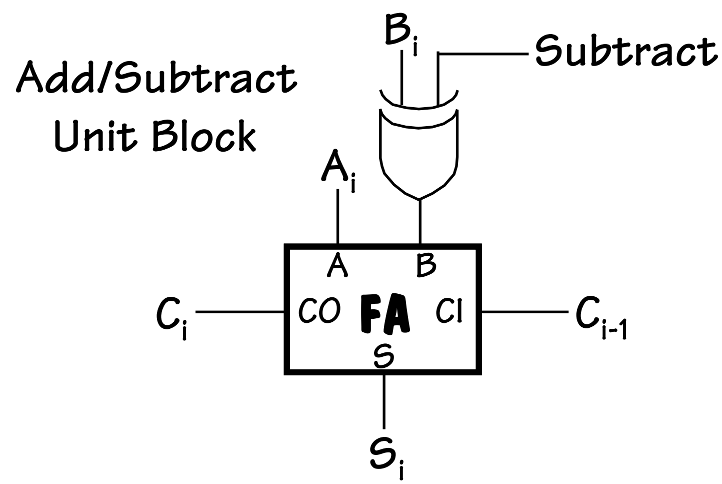
We introduce a new abstraction to aid in the construction of multipliers called the "Unsigned Multiplier Unit-block"

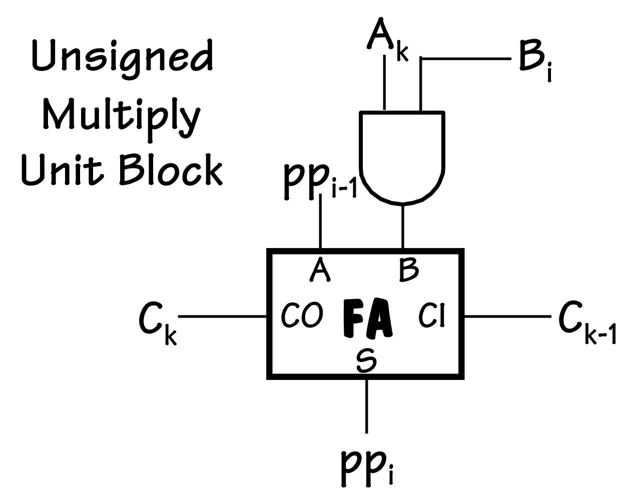
We did a similar thing last lecture when we converted our adder to an add/subtract unit.

A_k are bits of the Multiplicand and B_i are bits of the Multiplier.

The PP inputs and outputs represent "partial products" which are partial results from adding together shifted instances of the Multiplicand.

The initial PP_0 is zero.





Simple Combinational Multiplier

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

not 16

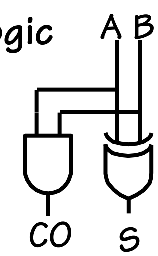
$t_{PD} = (2*(N-1) + N) * t_{PD}$

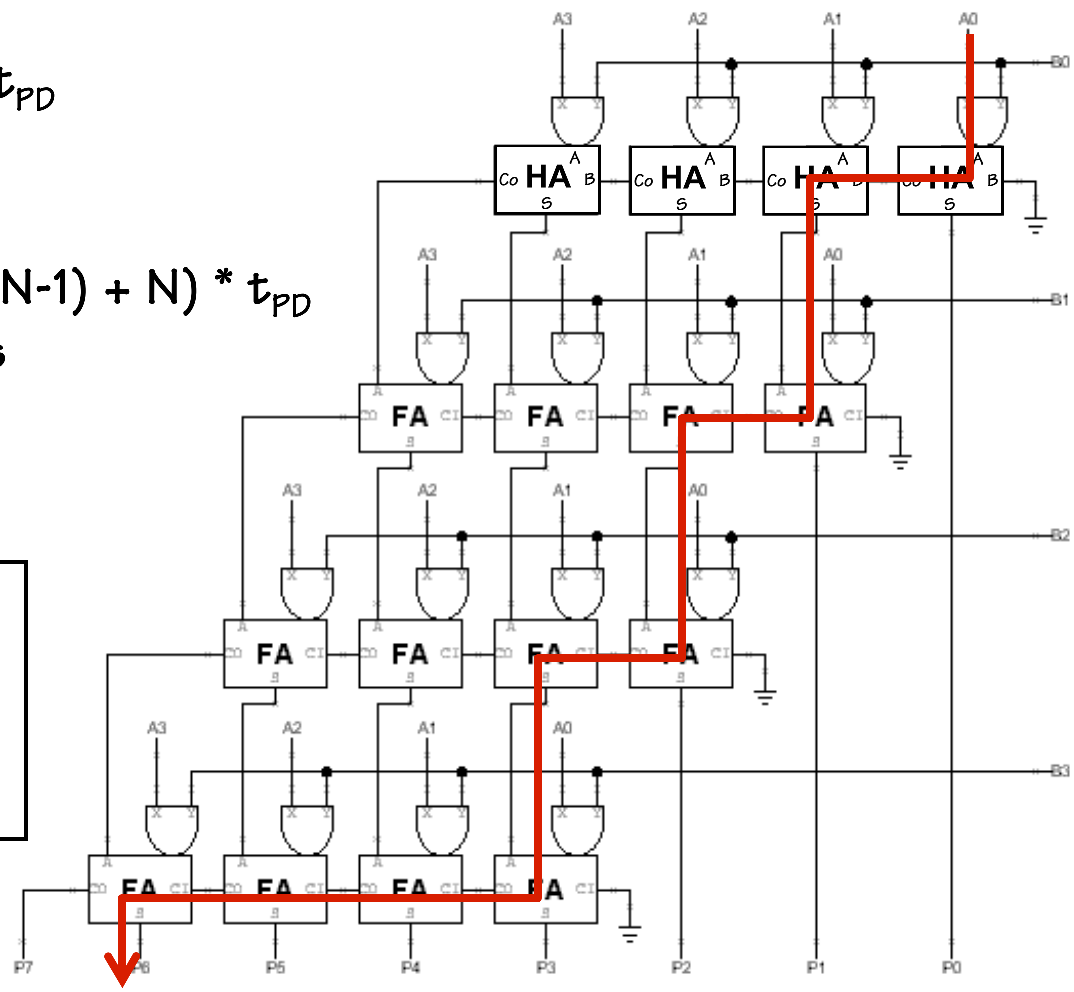
Components

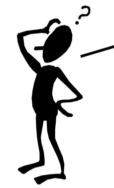
N * HA

N(N-1) * FA

The Logic of a Half-Adder

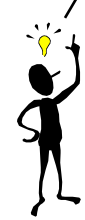






Is this faster than our assembly code?

To determine the timing specification of a composite combinational circuit we find the worst -case path for every output to any input.



NB: this circuit only works for nonnegative operands

"Carry-Save" Combinational Multiplier

Observation: Rather than propagating the carries to the next adder in each row, they can instead be forwarded to the next column of the following row

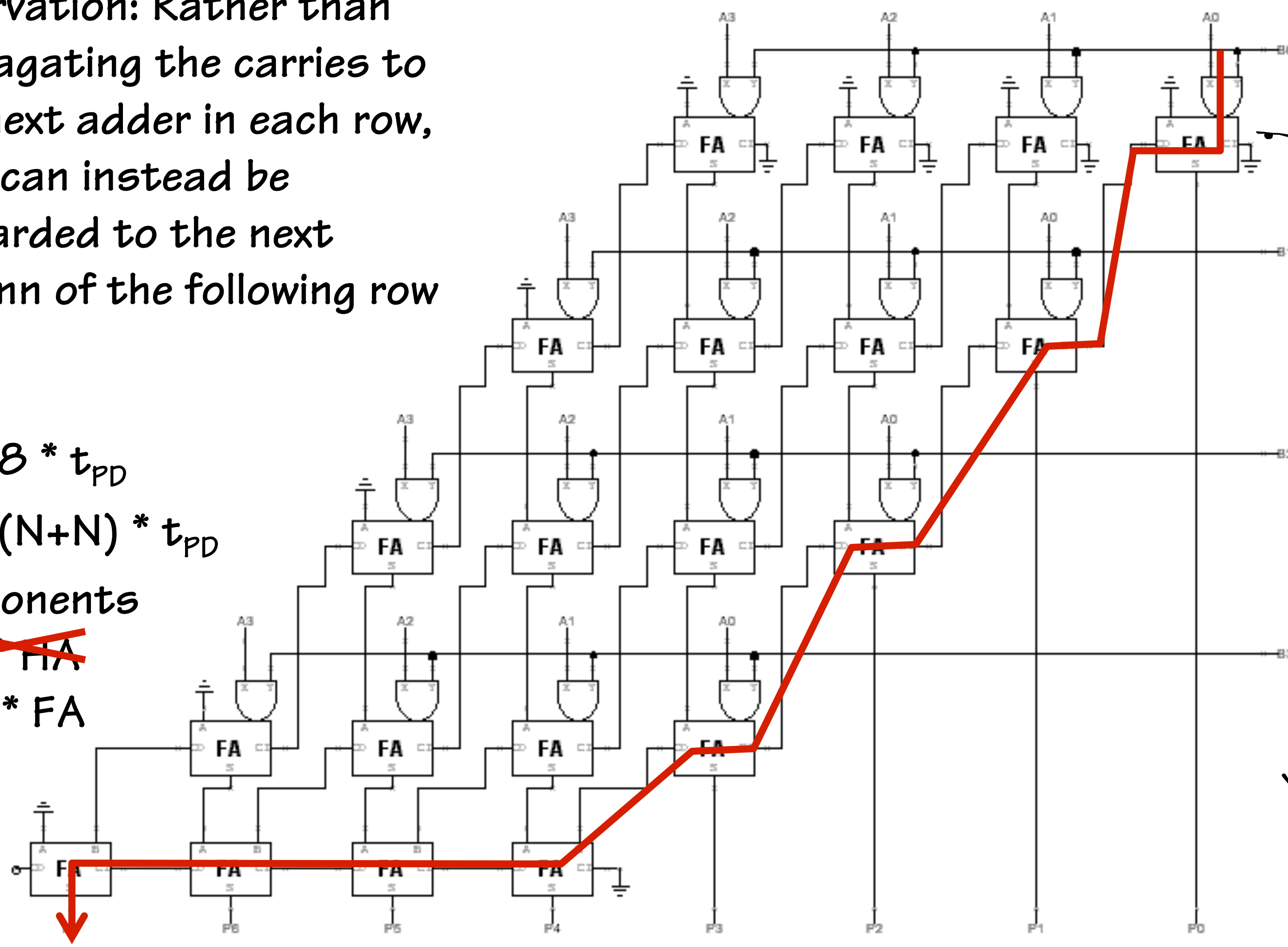
$t_{PD} = 8 * t_{PD}$

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

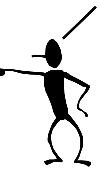
Components

~~N * HA~~

N^2 * FA



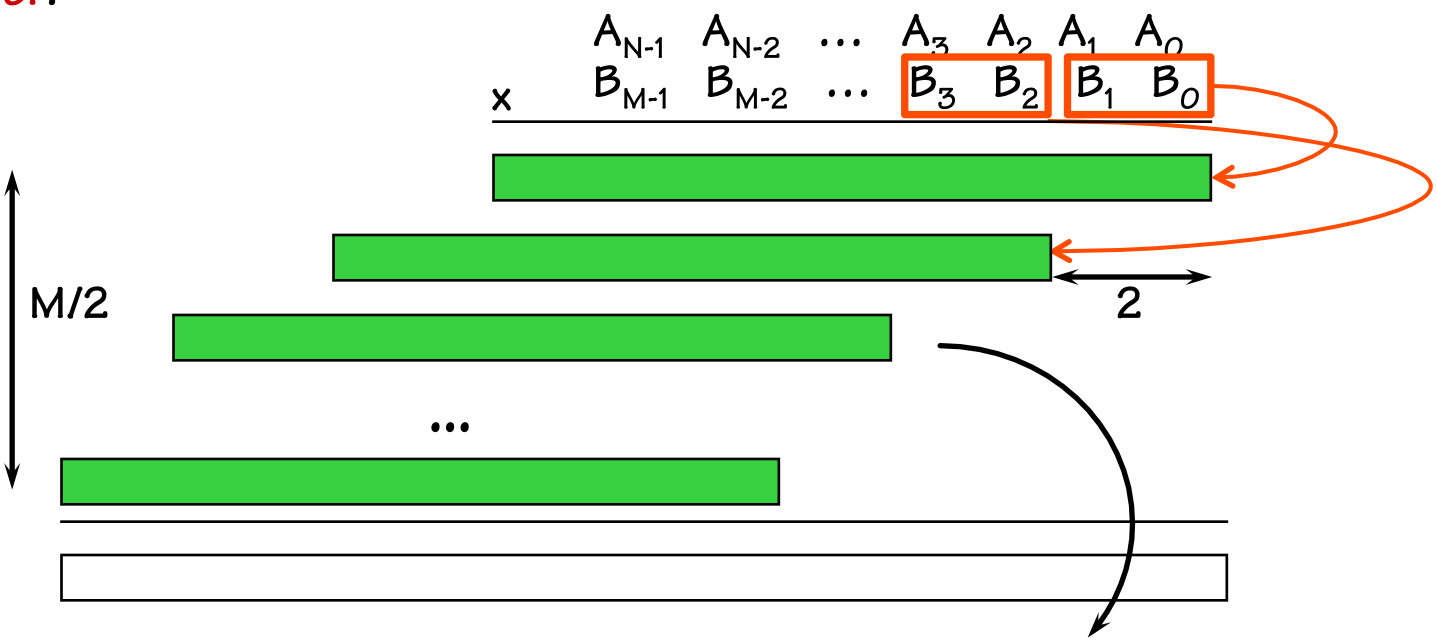
These Adders can be removed, and the AND gate outputs tied directly to the Carry inputs of the next stage.



This small performance improvement 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 rows and halve the latency of the multiplier!



$$
\begin{aligned}
B_{K+1,K} * A &= 0*A \Rightarrow 0 \\
&= 1*A \Rightarrow A \\
&= 2*A \Rightarrow 2A \text{ or } 4A - 2A \\
&= 3*A \Rightarrow 4A - A
\end{aligned}
$$

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

Booth Recoding of Multiplier

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

An encoding where each bit has the following weights:

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

$W(B_{2K}) = 1 * 2^{2K}$

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

-89 = 1 0 1 0 0 1 1 1.0

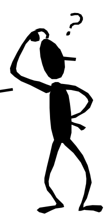
$= -1 * 2^0 \quad (-1)$

$+ 2 * 2^2 \quad (8)$

$+ (-2) * 2^4 \quad (-32)$

$+ (-1) * 2^6 \quad (-64)$

-89



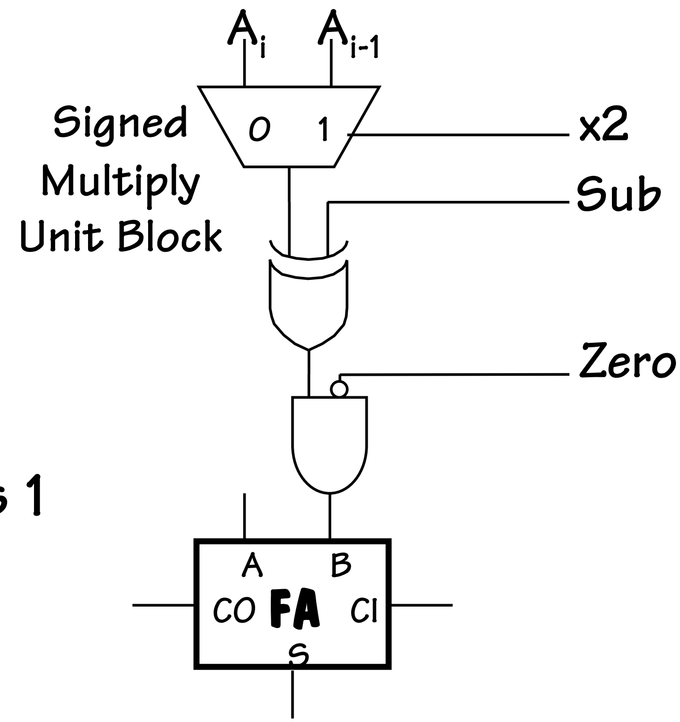
Hey, isn't that a negative number?

Yep! Booth recoding works for 2-Complement integers, now we can build a signed multiplier.

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!

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)

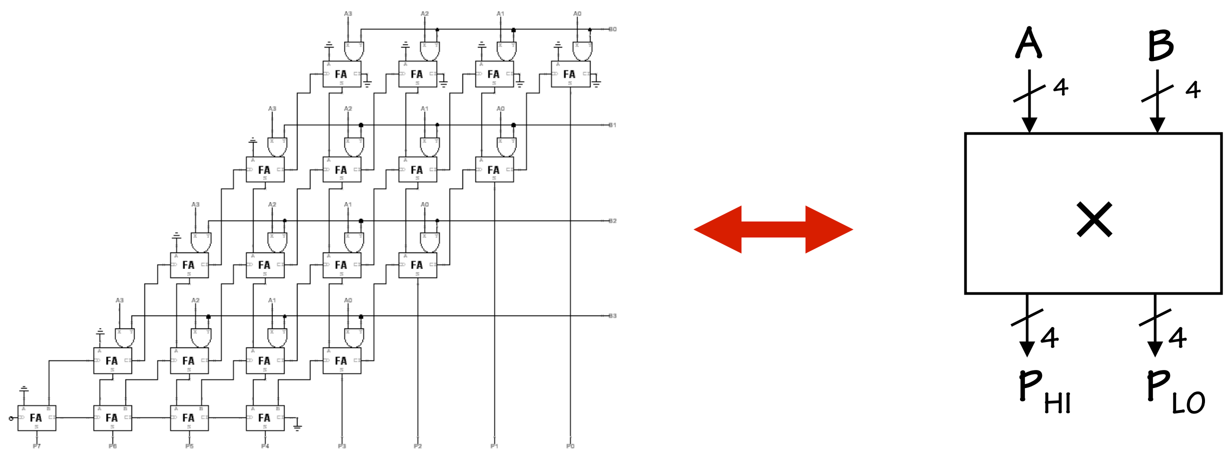
B_{2K+1}	B_{2K}	B_{2K-1}	x2	Sub	Zero
0	0	0	X	X	1
0	0	1	0	0	0
0	1	0	0	0	0
0	1	1	1	0	0
1	0	0	1	1	0
1	0	1	0	1	0
1	1	0	1	1	0
1	1	1	X	X	1

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?

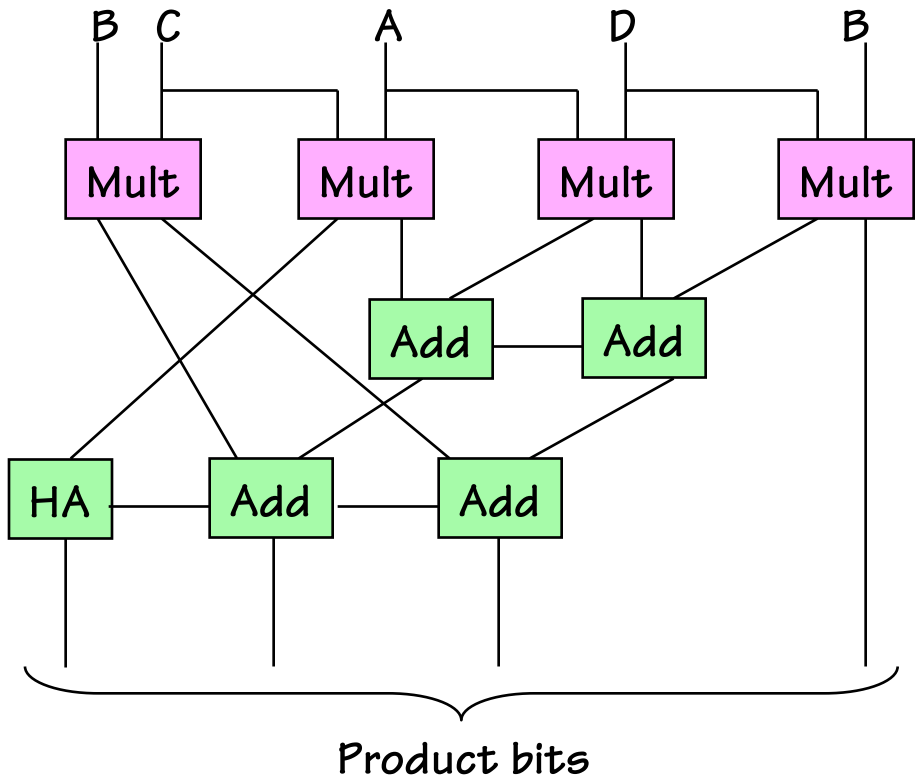
- 4 multipliers
  4 Adders

- 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




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

A Trick

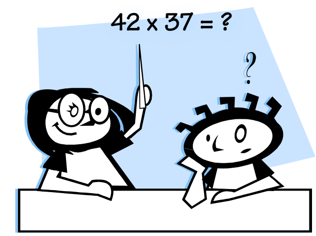
- The two middle partial products can be computed using a single multiplier and other partial products
- DA + CB = (C + D)(A + B) – (CA + DB)
- 3 multipliers
  8 adders
- This can be applied recursively
  (i.e. applied within each partial product)
- Leads to O($N^{1.58}$) adders
- This trick is becoming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

$$
\begin{array}{rrr}
 & A & B \\
\times & C & D \\
\hline
 & D & B \\
 D & A & \\
 C & B & \\
C & A & \\
\hline
\end{array}
$$

Let's Try it By Hand

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

$$42 \times 37$$



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, $SS = p_3 - (p_1 + p_2)$

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

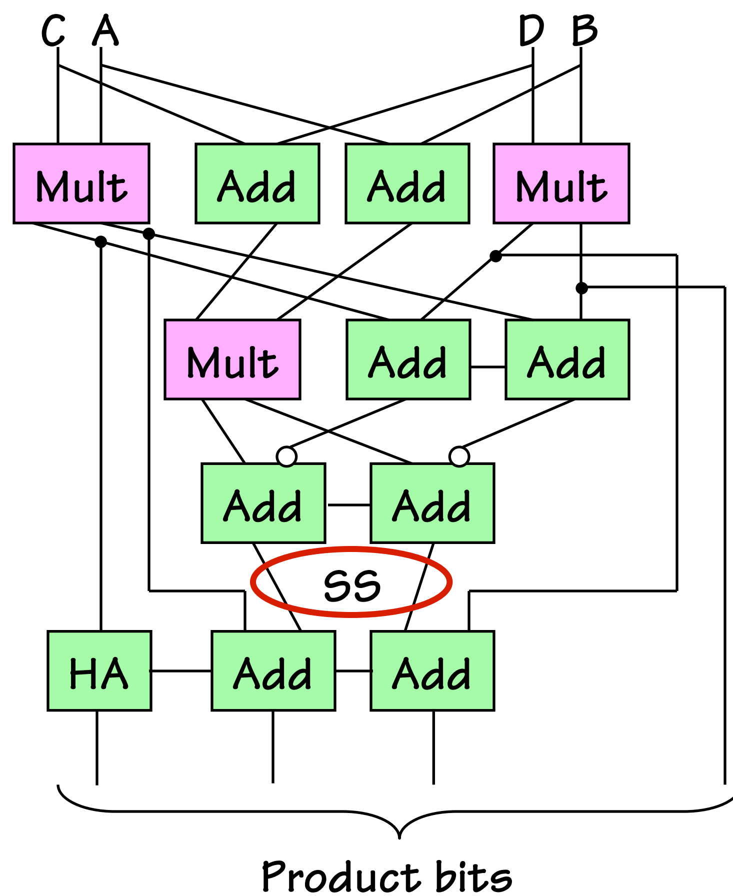
4) Add to find product, $p = 100{*}p_1 + 10{*}SS + p_2$

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

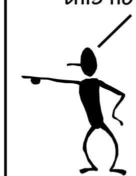
An O(N$^{1.58}$) Multiplier In Logic

The functional blocks would look like

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



Note: Adders with a bubble on one of their inputs becomes a subtractor in this notation.

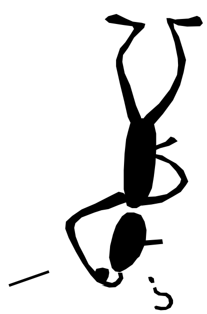


Where

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

Binary Division

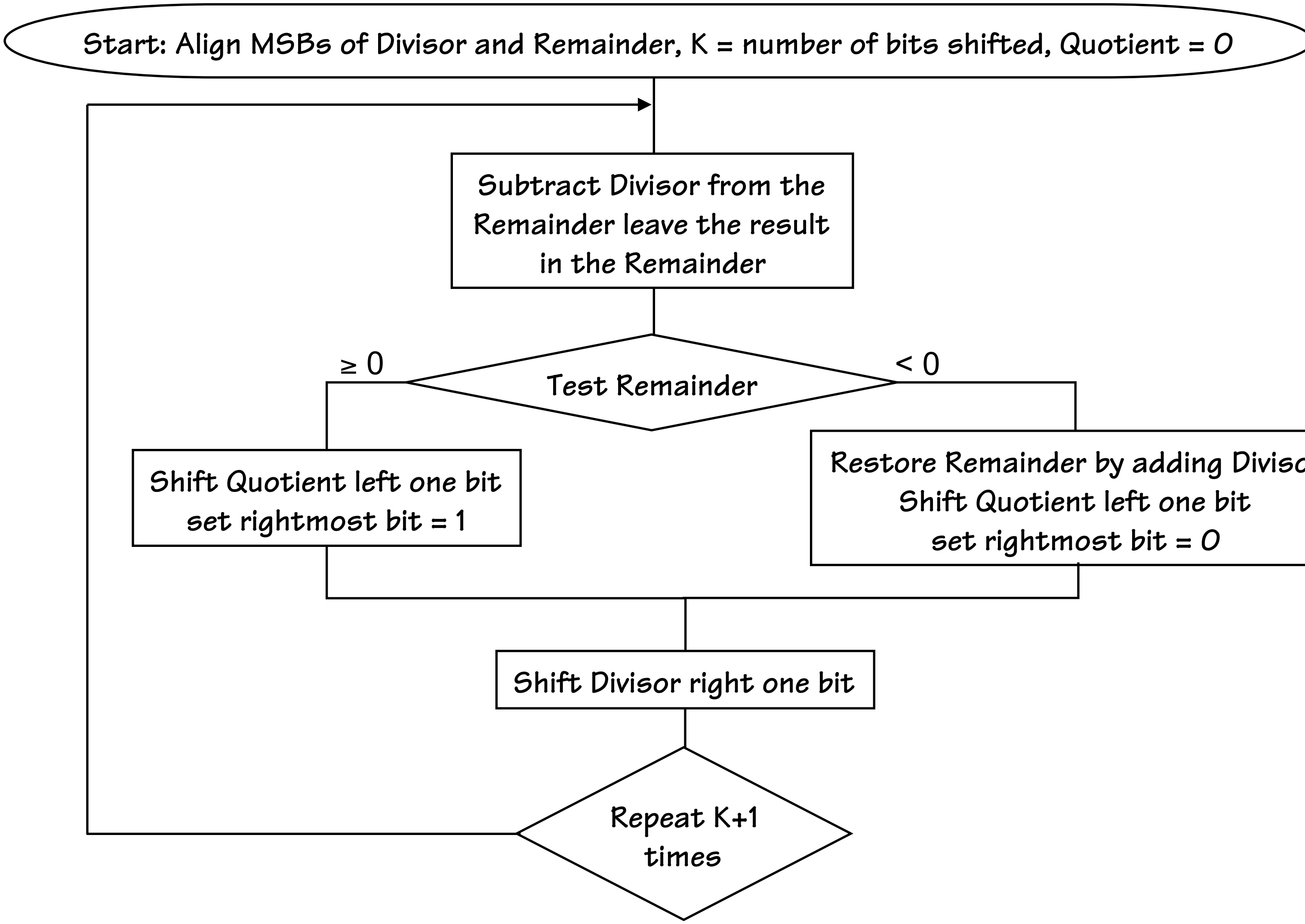
- Division merely reverses the process
  - Rather than adding successively larger partial products, subtract successively smaller divisors
  - When multiplying, we knew which partial products to actually add (based on the whether the corresponding bit was a 0 or a 1)
  - In division, we have to try *both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



Multiplication Upside-down

```
      P P P P P P P P
 -      D D D D          Q3 = 0 or 1?
 -        D D D D        Q2 = 0 or 1?
 -          D D D D      Q1 = 0 or 1?
 -            D D D D    Q0 = 0 or 1?
 ---------------------
              R R R R
```

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:

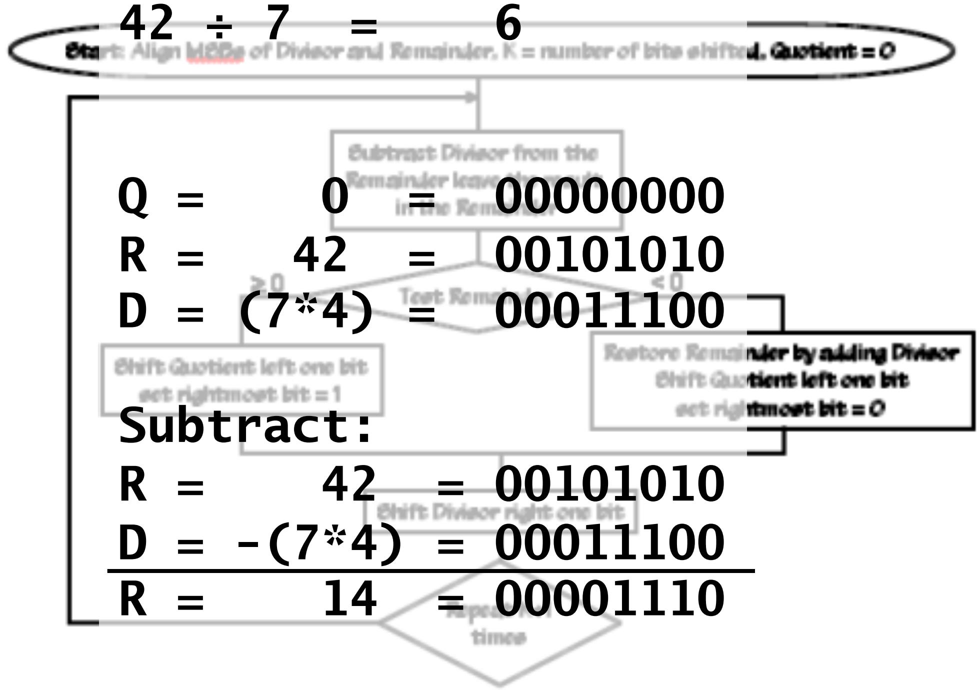
R = 42 = 00101010

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:

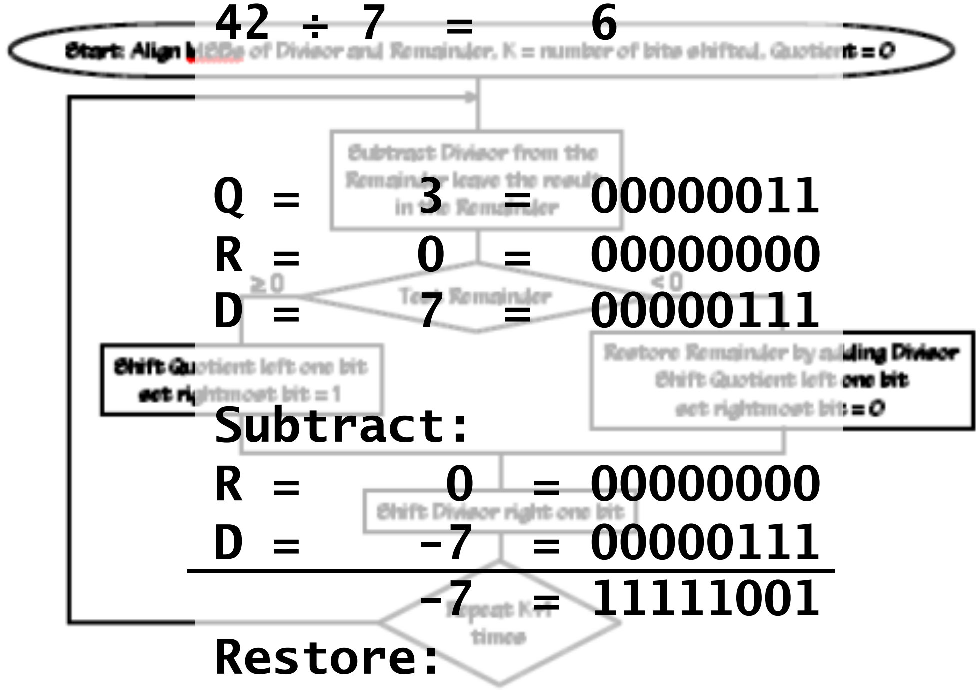
Q = 00000001
D = 00001110

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:



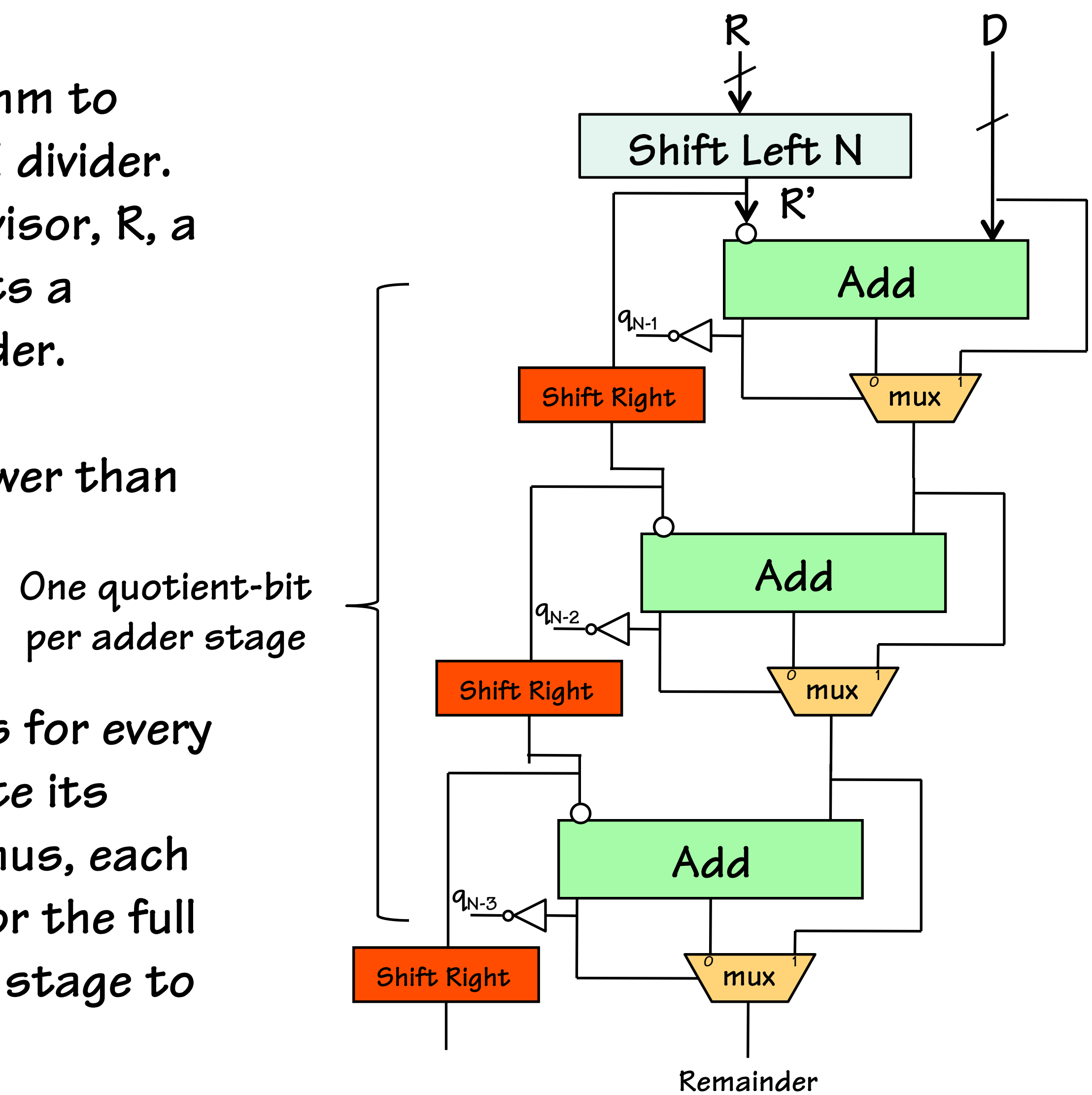
```
 R   D        Q
42 ÷ 7   =    6


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
```

Division Big Boxes

We can use this algorithm to design a combinational divider. It takes as inputs a divisor, R, a dividend, D, and outputs a quotient and a remainder.

Dividing is generally slower than multiplication.

The worst case propagation delay waits for every adder stage to generate its most significant bit, thus, each stage has to waiting for the full sum from the previous stage to complete.



Next Time

- We dive into floating point arithmetic