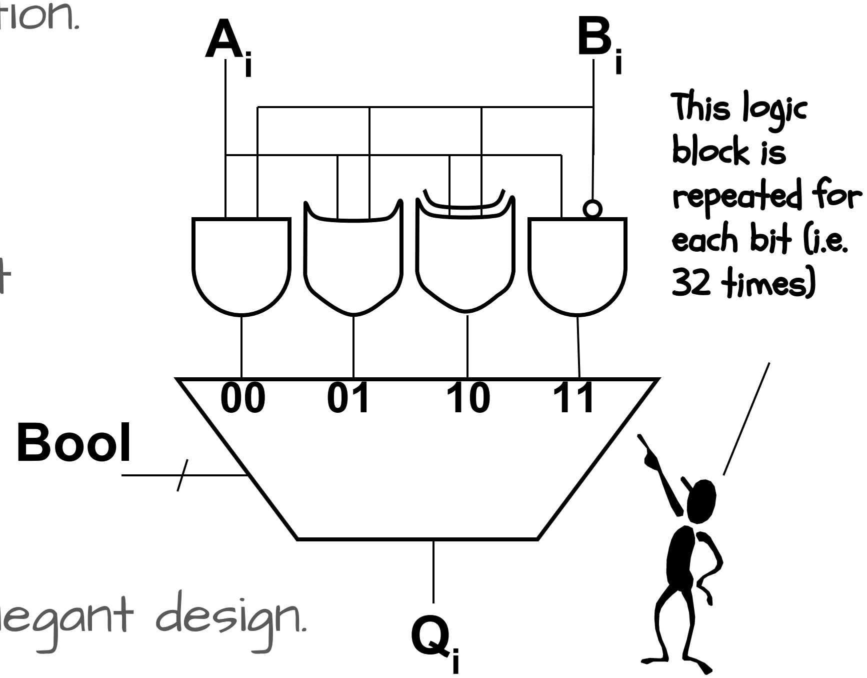
Boolean Unit (The obvious way)

It is simple to build up a Boolean unit using primitive gates and a mux to select the function.

Since there is no interconnection between bits, this unit can be simply replicated at each position. The cost is about 7 gates per bit. One for each primitive function, and approx 3 for the 4-input mux.

This is a straightforward, but not elegant design.



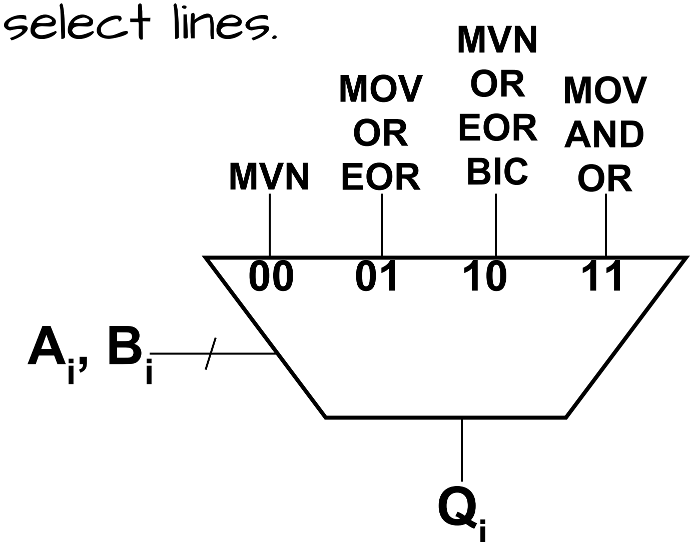
Cooler Bools

We can better leverage a MUX's capabilities in our Boolean unit design, by connecting the bits to the select lines.

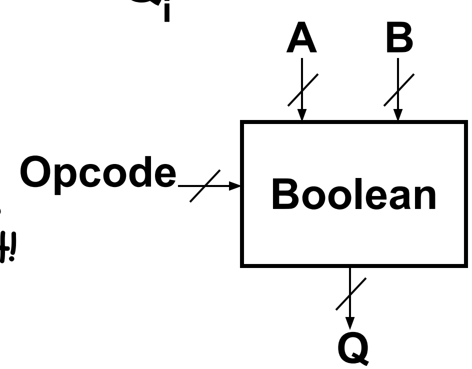
Why is this better?

While it might take a little logic to decode the truth table inputs, you only have to do it once, independent of the number of bits.

BTW, it also handles the MOV and MVN cases.



Which ever way makes the most sense to you. Let's get a box around it!



Decoding the Booleans and others

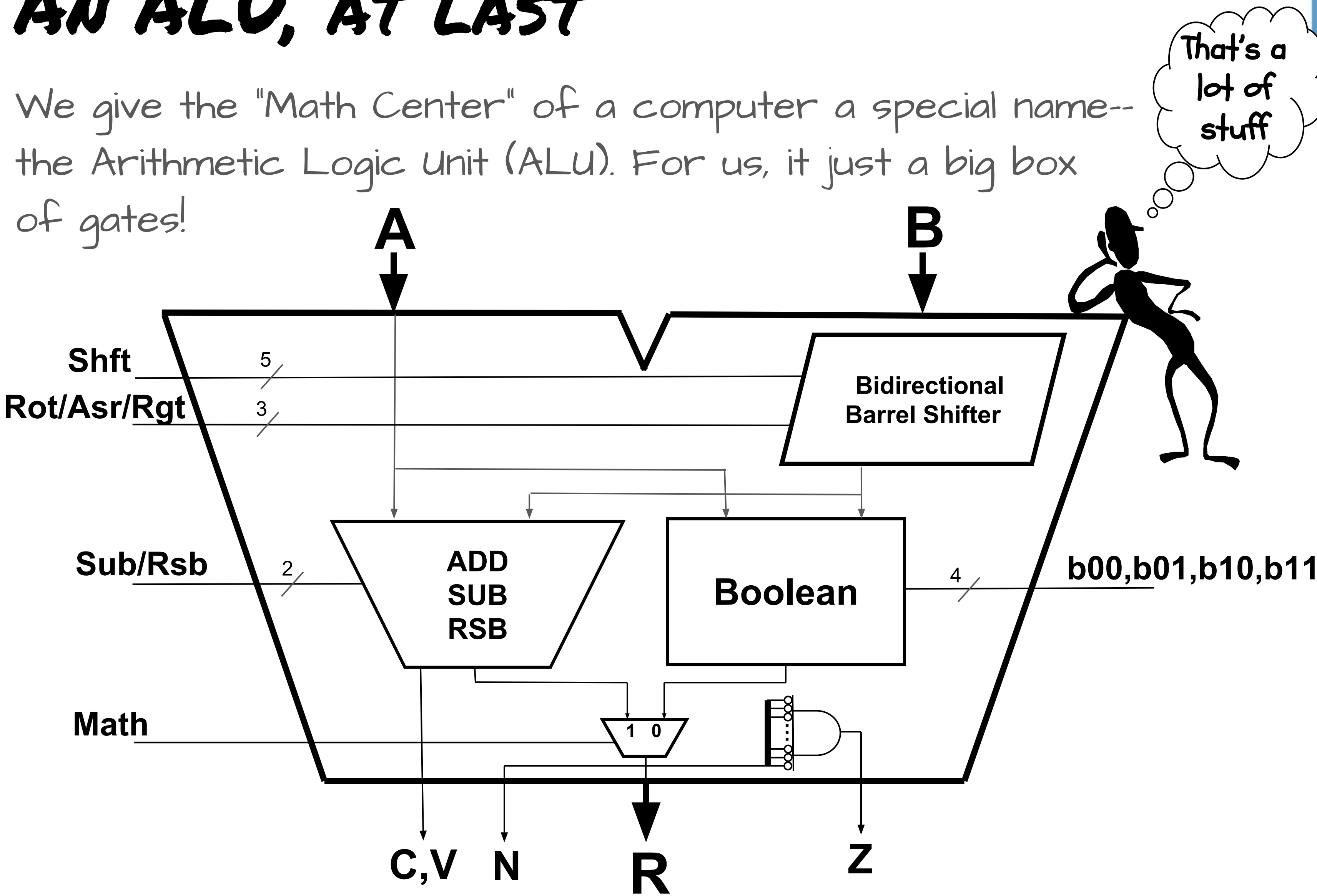
It may seem a little tedious, but all the controls that we need can be derived from the ARM OpCode encodings.

The 'X's in the truth table are "don't cares" they provide flexibility in the implementation.

Opcode	Code				00	01	10	11	Sub	Rsb	Math
AND	0	0	0	0	0	0	0	1	X	X	0
EOR	0	0	0	1	0	1	1	0	X	X	0
SUB	0	0	1	0	X	X	X	X	1	0	1
RSB	0	0	1	1	X	X	X	X	0	1	1
ADD	0	1	0	0	X	X	X	X	0	0	1
ADC	0	1	0	1	X	X	X	X	0	0	1
SBC	0	1	1	0	X	X	X	X	1	0	1
RSC	0	1	1	1	X	X	X	X	0	1	1
TST	1	0	0	0	0	0	0	1	X	X	0
TEQ	1	0	0	1	0	1	1	0	X	X	0
CMP	1	0	1	0	X	X	X	X	1	0	1
CMN	1	0	1	1	X	X	X	X	0	0	1
ORR	1	1	0	0	0	1	1	1	X	X	0
MOV	1	1	0	1	0	1	0	1	X	X	0
BIC	1	1	1	0	0	0	1	0	X	X	0
MVN	1	1	1	1	1	0	1	0	X	X	0

AN ALU, AT LAST

We give the "Math Center" of a computer a special name-- the Arithmetic Logic Unit (ALU). For us, it just a big box of gates!

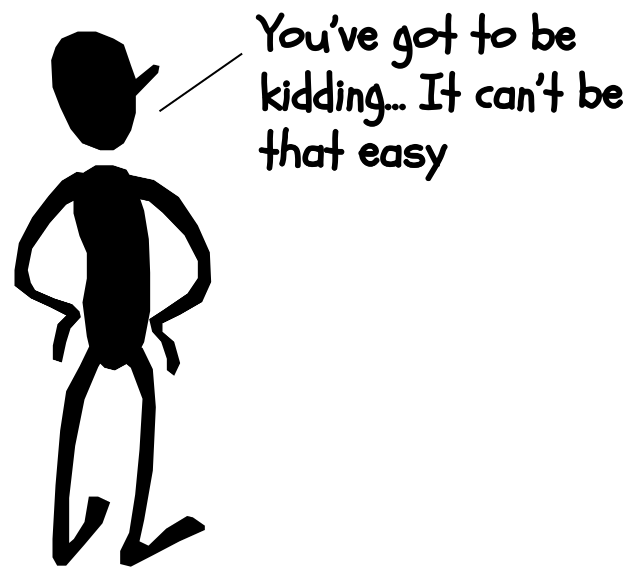


Binary Multiplication

The key to multiplication was memorizing a digit-by-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

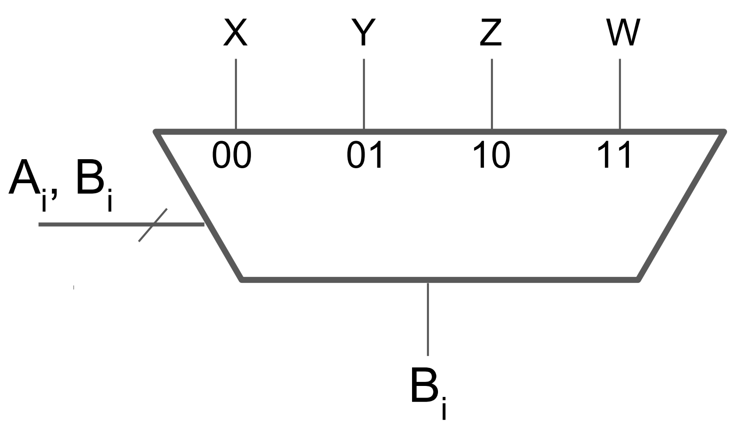
×	0	1
0	0	0
1	0	1



Warm up / Review

Suppose that you wanted to extend the ARM ISA to include a **nor** instruction like MIPS, how would the mux inputs of the BOOL functional block shown on the right be set?

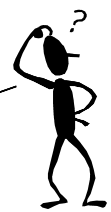
A) X, Y, Z = 1, W = 0
B) X = 0, Y, Z, W = 1
C) X= NOT(OR(Ai,Bi)), Y, Z, W = 0
D) X = 1, Y, Z, W = 0
E) A NOR cannot be implemented with this functional block



Digit by digit = bit by bit

The "Binary" Multiplication Table

Hey, that looks like an AND gate



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.

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

A_jB_i is a "partial product"

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

Multiplying in Assembly

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

```
; multiplies r0 and r1
mult:   mov     r3,#0           ; zero product
part:   tst     r1,#1           ; check if least significant bit=1
        addne   r3,r3,r0        ; add multiplicand to product
        mov     r0,r0,lsl #1    ; multiplicand *= 2
        movs    r1,r1,lsr #1    ; multiplier /= 2
        bne     part            ; continue while multiplier is not 0
        mov     r0,r3           ; copy product to return value
        bx      lr
```

```
          Multiplier                      Multiplicand
r1: 0000 0000 0010 1010       r0: 0000 0000 0100 1000

    0000 0000 0010 1010           0000 0000 0000 0000
    0000 0000 0001 0101           0000 0000 1001 000_
    0000 0000 0000 1010           0000 0000 0000 00__
    0000 0000 0000 0101           0000 0010 0100 0___
    0000 0000 0000 0010           0000 0000 0000 ____
    0000 0000 0000 0001           0000 1001 000_ ____
                                  0000 1011 1101 0000
```

Multiplier Unit-Block

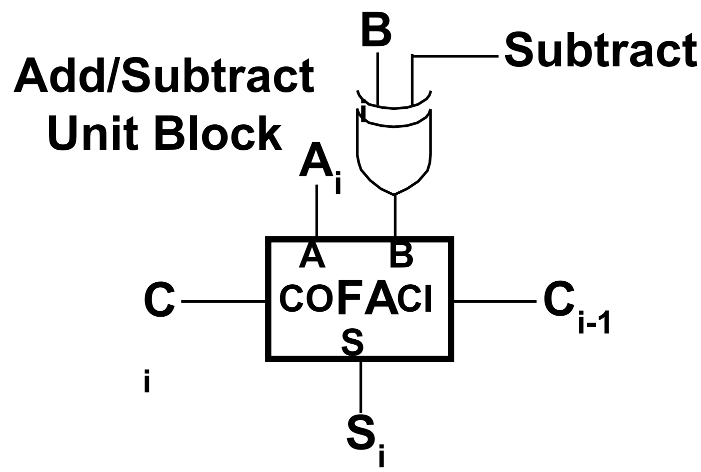
We introduce a new abstraction to aid in the construction of multipliers called the "Unsigned Multiplier Unit-block"

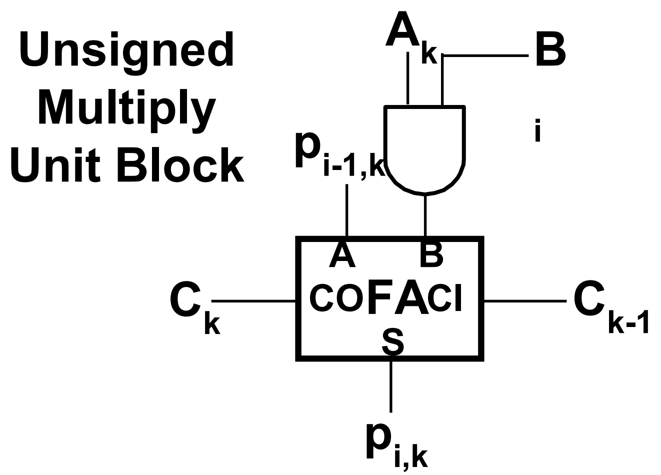
We did a similar thing last lecture when we converted our adder to an add/subtract unit.

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

The $P_{i,k}$ inputs and outputs represent "partial products" which are partial results from adding together shifted instances of the Multiplicand.

The initial $P_{0,k}$ is zero.





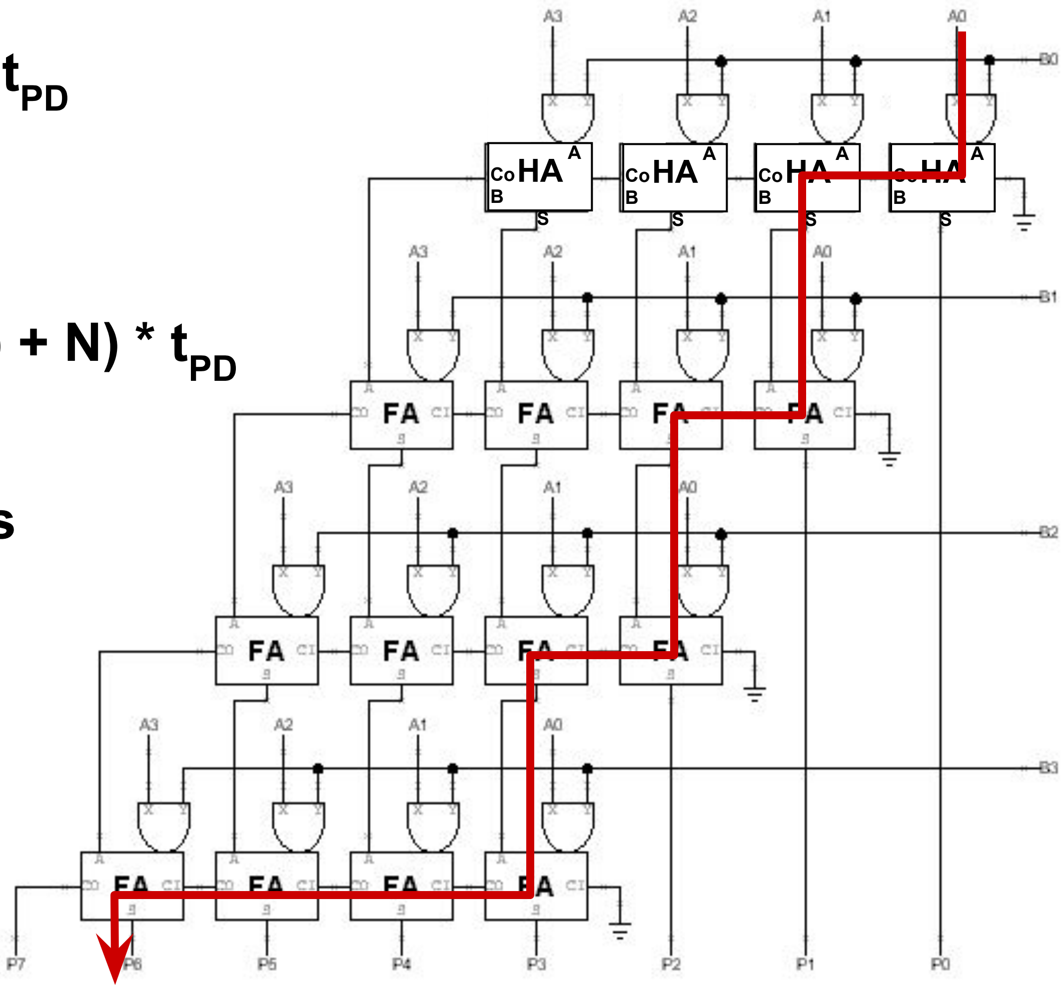
Simple Combinational Multiplier

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

not 16

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

Components
N * HA
N(N-1) * FA



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" 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

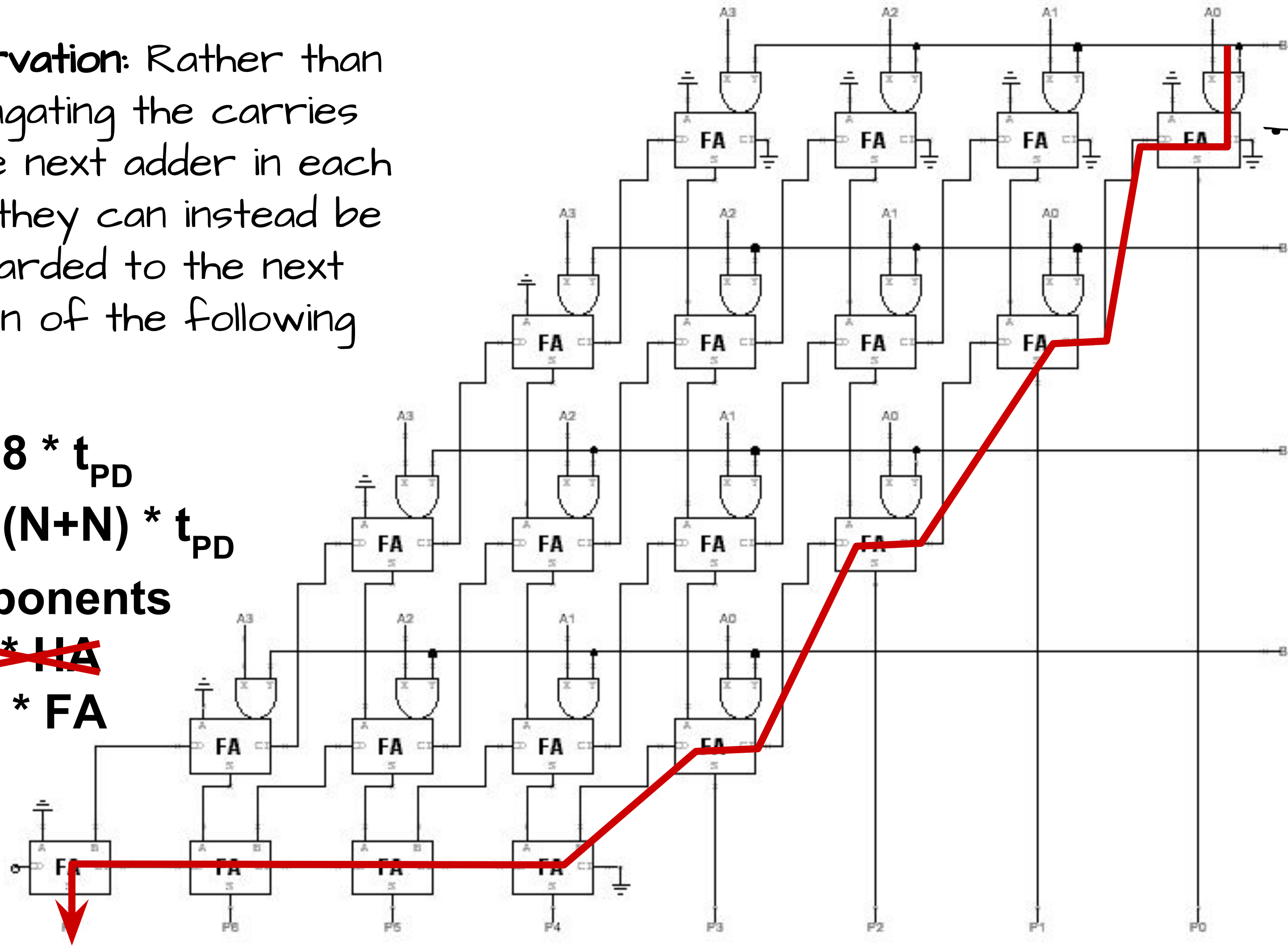
$t_{PD} = 8 * t_{PD}$

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

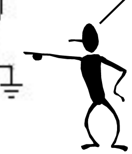
Components

~~N * HA~~

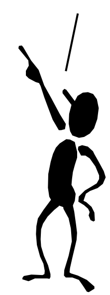
N^2 * FA



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

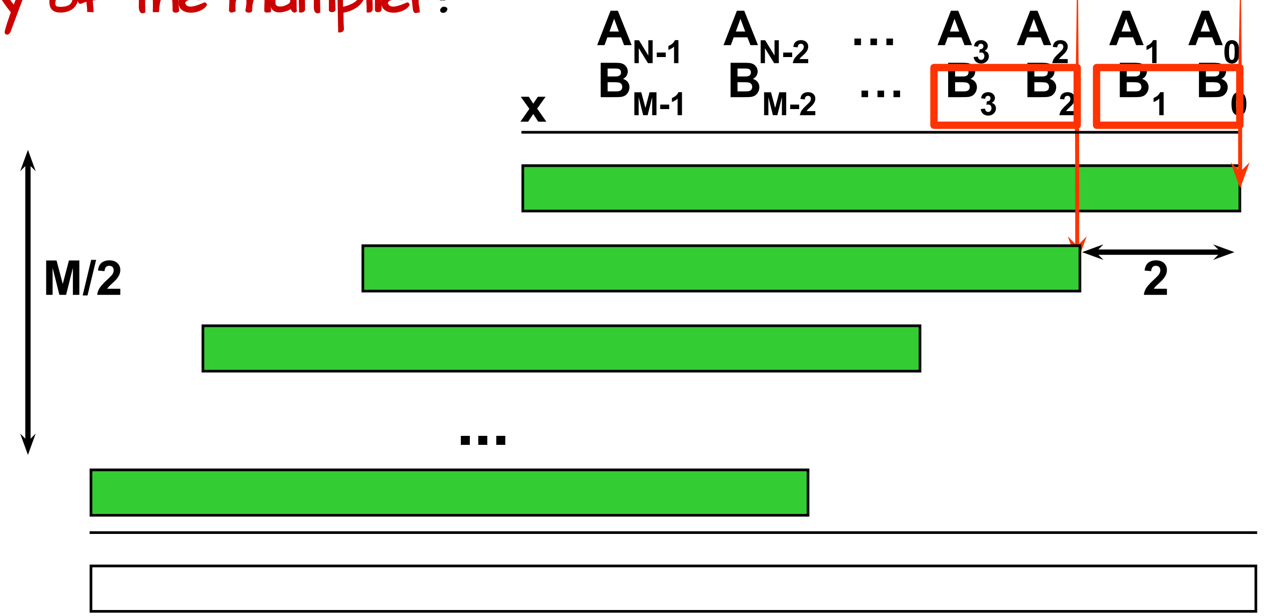


This small performance improvement hardly seems worth the effort, however, this design is easier to "pipeline".



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!



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

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

Booth 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
1	1	0	sub A ← -2*A+A
1	1	1	add 0 ← -A+A

-89 = 1 0 1 0 0 1 1 1 .0

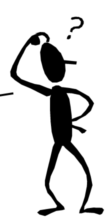
$= -1 * 2^0$ (-1)

$+ 2 * 2^2$ (8)

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

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

-89



Hey, isn't that a negative number?

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

An encoding where each bit has the following weights:

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

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

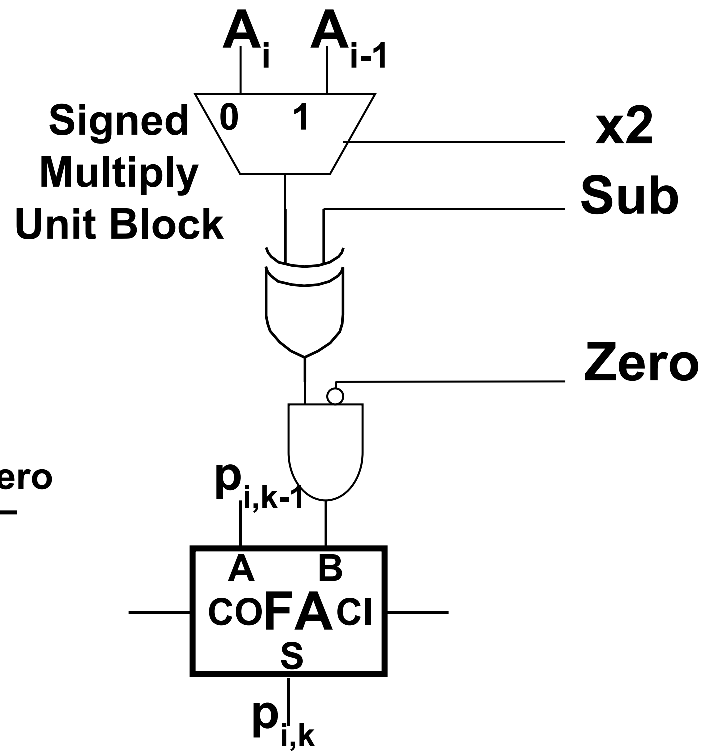
$W(B_{2K-1}) = 1 * 2^{2K}$

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 Multiplier unit block

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)



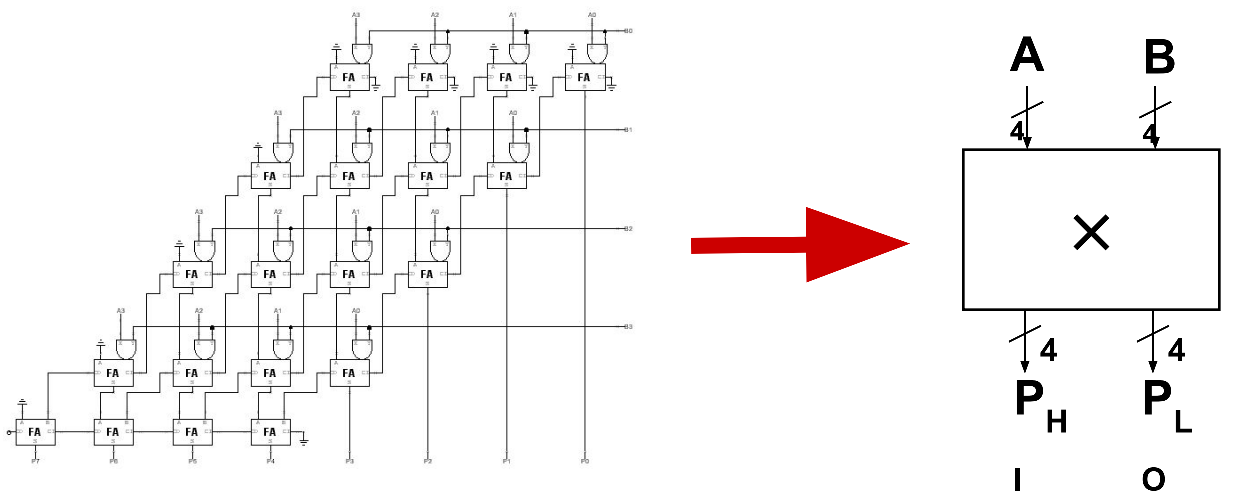
NOTE:

- Booth recoding can be used to implement signed multiplications

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	0	1	0
1	1	1	X	X	1

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?

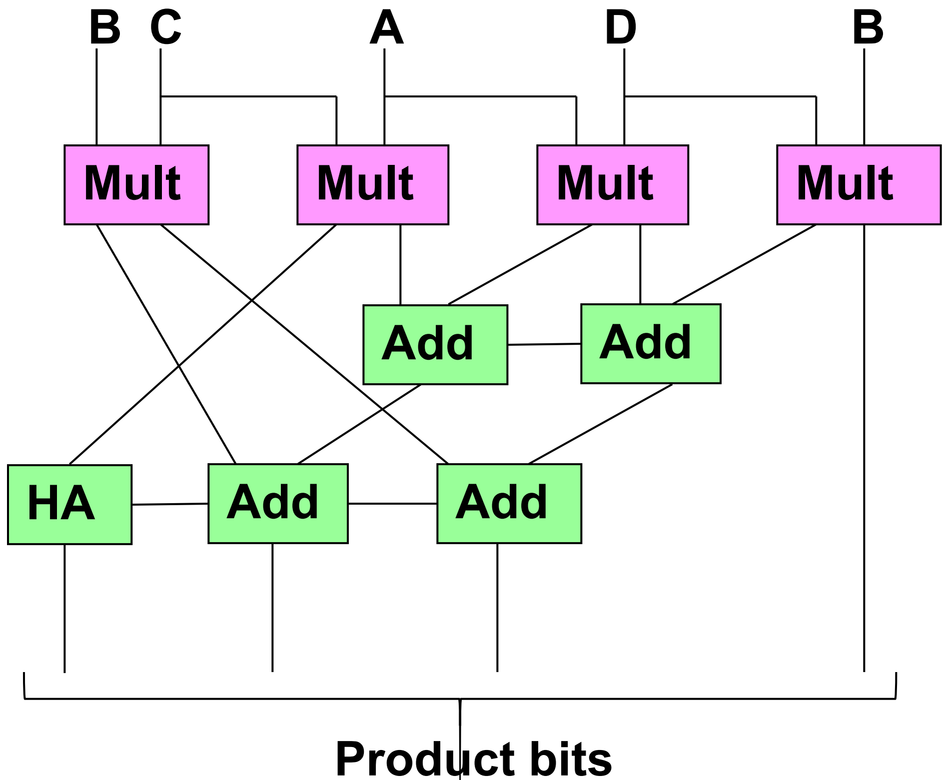
- 4 multipliers
  4 Adders

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

```
   AB
x  CD
   DB
  DA
  CB
 CA
```

O(N²) MULTIPLIER LOGIC

The functional blocks look like




```
   AB
x  CD
   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

```
      AB
x     CD
--------
      DB
     DA
     CB
    CA
```

Let's Try it By Hand

1) Choose 2, 2 digit numbers to multiply: ab x cd

42 x 37

2) Multiply digits: p1 = a x c, p2 = b x d, p3 = (c + d)(a + b)

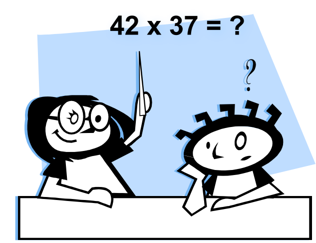
p1 = 4 x 3 = 12, p2 = 2*7 = 14, p3 = (4+2)x(3+7) = 60

3) Compute partial subtracted sum, SS = p3 - (p1 + p2)

SS = 60 - (12 + 14) = 34

4) Add as follows: p = 100 x p1 + 10 x SS + p2

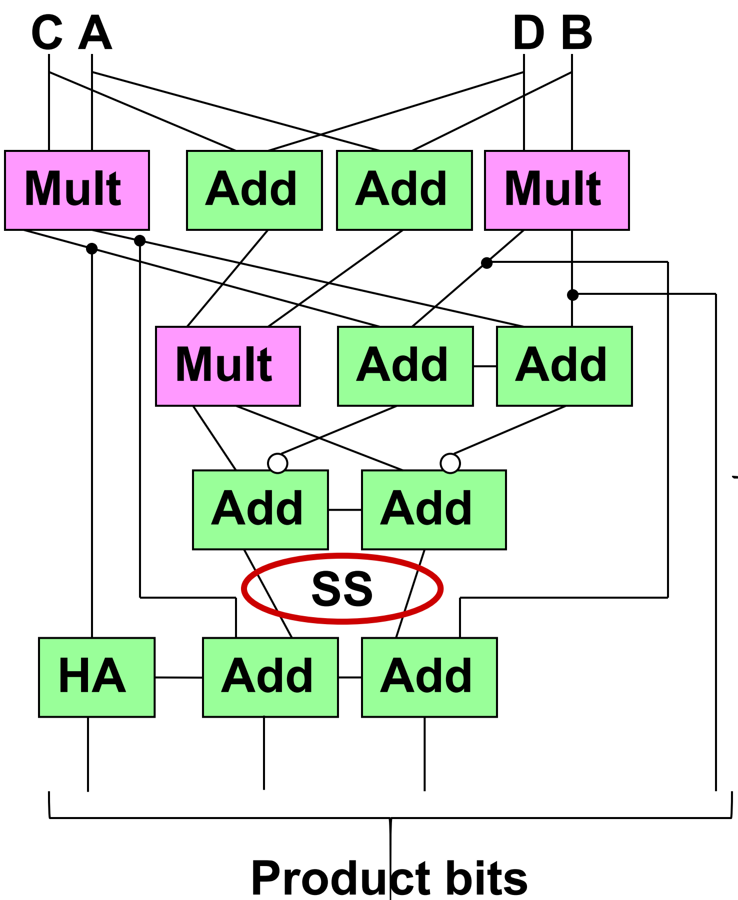
P = 1200 + 340 + 14 = 1554 = 42 x 37



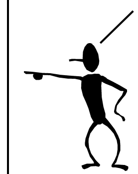
An O($N^{1.58}$) Multiplier

The functional blocks would look like:

```
      AB
x     CD
      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)