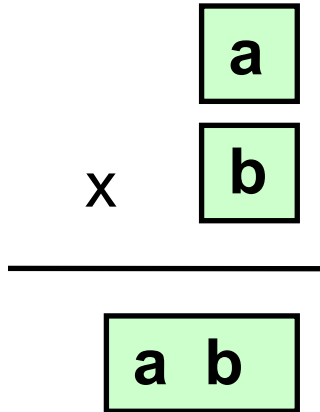


Cost/Performance Tradeoffs: a case study

Digital Systems Architecture I.

Binary Multiplication



n bits

n bits

2n bits

since $(2^n - 1)^2 < 2^{2n}$

EASY PROBLEM: design combinational circuit to multiply tiny (1-, 2-, 3-bit) operands...

HARD PROBLEM: design circuit to multiply BIG (32-bit, 64-bit) numbers

We can make *big* multipliers out of *little* ones!

Engineering Principle:

Exploit STRUCTURE in problem.

Making a 2n-bit multiplier using n-bit multipliers

Given n-bit multipliers:

$$\begin{array}{c}
 \boxed{a} \\
 \text{n bits}
 \end{array}
 \times
 \begin{array}{c}
 \boxed{b} \\
 \text{n bits}
 \end{array}
 =
 \begin{array}{c}
 \boxed{ab} \\
 \text{2n bits}
 \end{array}$$

Synthesize 2n-bit multipliers:

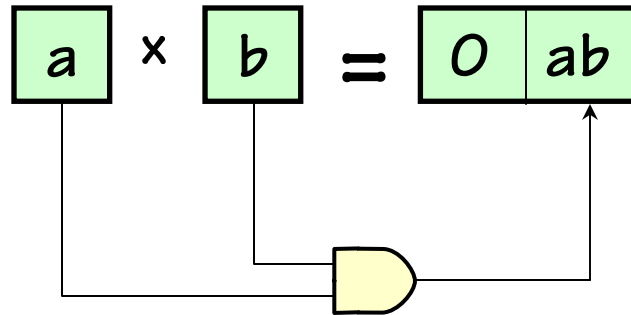
$$\begin{array}{c}
 \boxed{a} \\
 \text{2n bits}
 \end{array}
 \times
 \begin{array}{c}
 \boxed{b} \\
 \text{2n bits}
 \end{array}
 =
 \begin{array}{c}
 \boxed{ab} \\
 \text{4n bits}
 \end{array}$$

$$\begin{array}{r}
 \begin{array}{|c|c|} \hline a_H & a_L \\ \hline \end{array} \\
 \times \\
 \begin{array}{|c|c|} \hline b_H & b_L \\ \hline \end{array} \\
 \hline
 \\
 \begin{array}{|c|c|} \hline & a_L b_L \\ \hline \end{array} \\
 \begin{array}{|c|c|} \hline a_L b_H & \\ \hline \end{array} \\
 \begin{array}{|c|c|} \hline a_H b_L & \\ \hline \end{array} \\
 \begin{array}{|c|c|} \hline a_H b_H & \\ \hline \end{array} \\
 \hline
 \\
 \begin{array}{|c|c|} \hline & & & ab \\ \hline \end{array}
 \end{array}$$

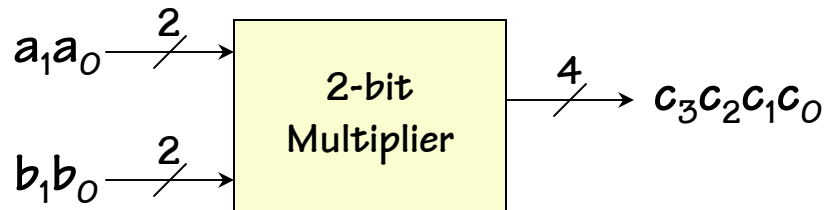
Our Basis:

$n=1$: minimalist starting point

Multiplying two 1-bit numbers is pretty simple:



Of course, we could start with optimized combinational multipliers for larger operands; e.g.

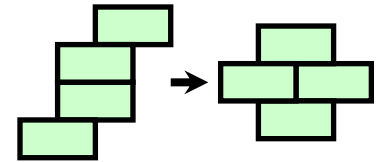


the logic gets more complex, but some optimizations are possible...

Our induction step:

2n-bit by 2n-bit multiplication:

1. Divide multiplicands into n-bit pieces
2. Form 2n-bit partial products, using n-bit by n-bit multipliers.
3. Align appropriately
4. Add.



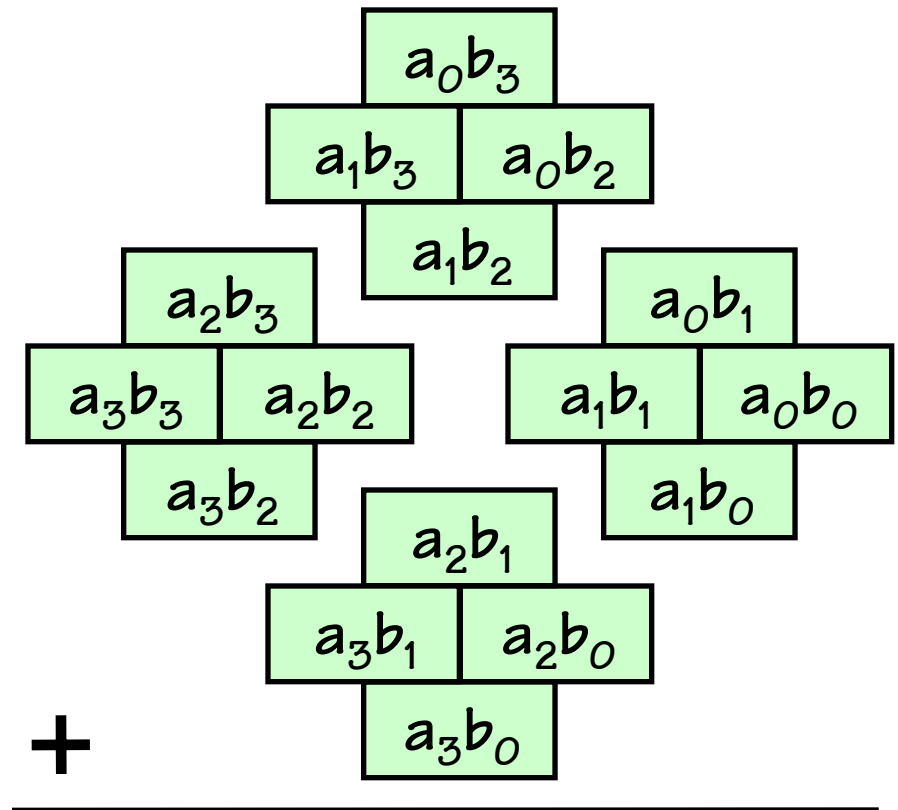
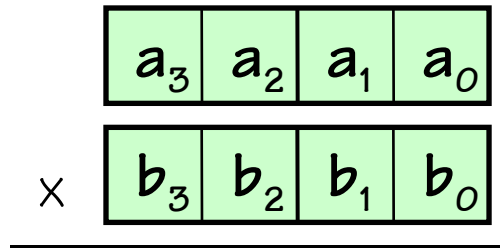
REGROUP partial products -
2 additions rather than 3!

$$\begin{array}{r}
 \boxed{a_H} \boxed{a_L} \times \boxed{b_H} \boxed{b_L} = + \begin{array}{r}
 \boxed{a_L b_H} \\
 \boxed{a_H b_H} \quad \boxed{a_L b_L} \\
 \boxed{a_H b_L} \\
 \hline
 \boxed{a \cdot b}
 \end{array}
 \end{array}$$

Induction: we can use the same structuring principle to build a 4n-bit multiplier from our newly-constructed 2n-bit ones...

Brick Wall view of partial products

Making 4n-bit multipliers from n-bit ones: 2 “induction steps”



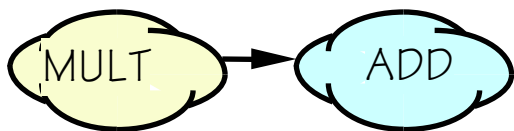
Multiplier Cookbook: Chapter 1

Given problem:

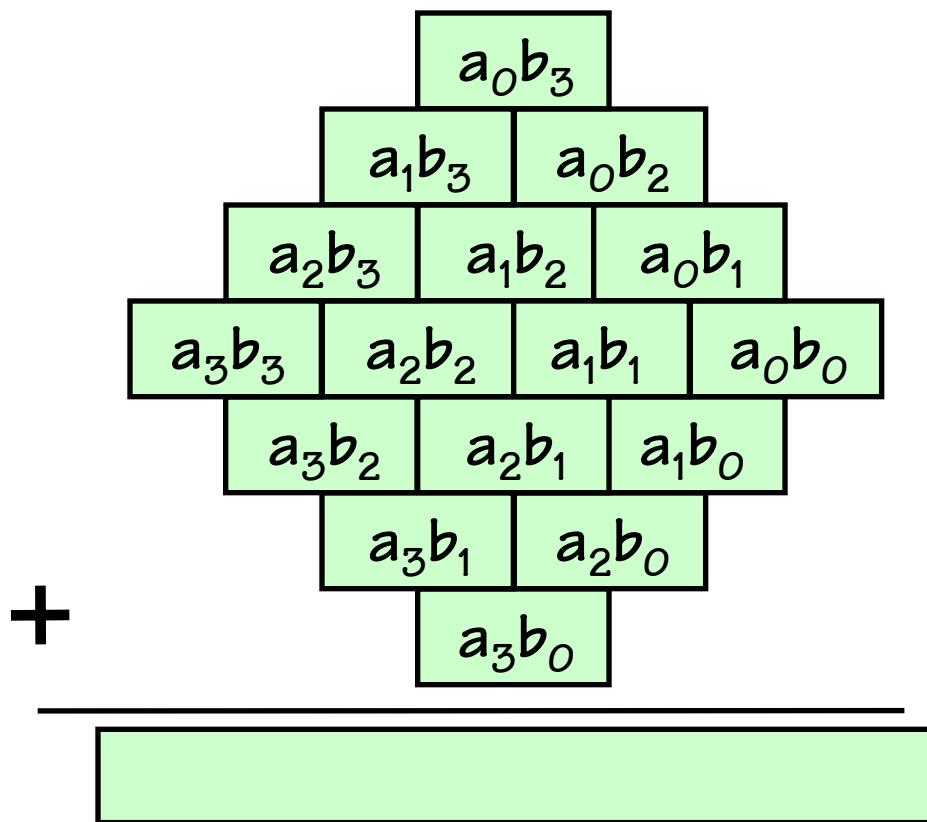
$$\begin{array}{r}
 \boxed{a_3} \boxed{a_2} \boxed{a_1} \boxed{a_0} \\
 \times \boxed{b_3} \boxed{b_2} \boxed{b_1} \boxed{b_0} \\
 \hline
 \end{array}$$

Subassemblies:

- Partial Products
- Adders



Step 1: Form (& arrange)
Partial Products:




Step 2: Sum

Performance/Cost Analysis

"Order Of" notation:

"g(n) is of order f(n)" $g(n) = \Theta(f(n))$

$g(n) = \Theta(f(n))$ if there exist $C_2 \geq C_1 > 0$,
such that for all but finitely many integral $n \geq 0$

$$c_1 \cdot f(n) \leq g(n) \leq c_2 \cdot f(n)$$


$g(n) = O(f(n))$

$\Theta(\dots)$ implies
both inequalities;
 $O(\dots)$ implies only
the second.

Example:

$$n^2 + 2n + 3 = \Theta(n^2)$$

since

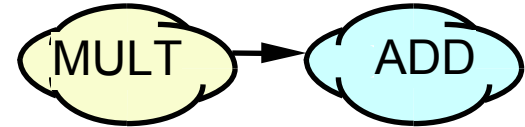
$$n^2 \leq (n^2 + 2n + 3) \leq 2n^2$$

"almost always"

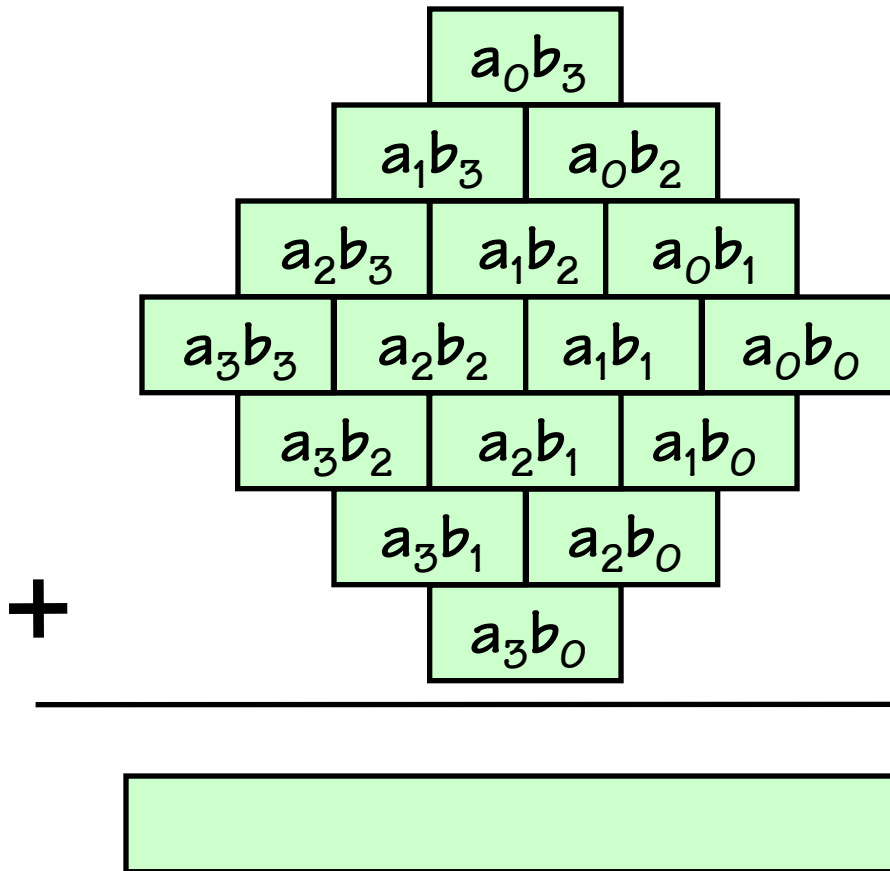
Partial Products:	n^2	=	$\Theta(n^2)$
Things to Add:	$2n-2$	=	$\Theta(n)$
Adder Width:	n	=	$\Theta(n)$
Hardware Cost:	?	=	$\Theta(n^2)$

Latency: $O(n^2) ??$

Observations:



$\Theta(n^2)$ partial products.
 $\Theta(n^2)$ full adders.
Hmmm.



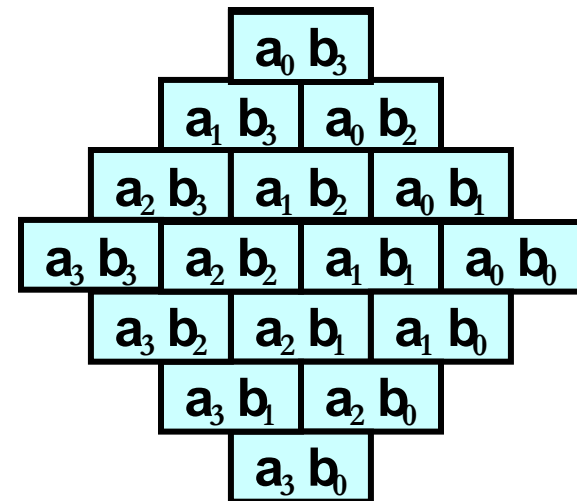
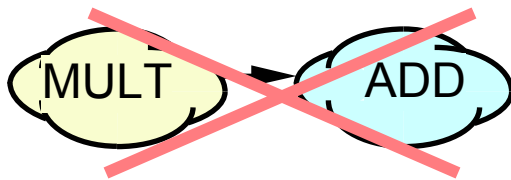
Repackaging Function

Engineering Principle #2:

Put the Solution where the Problem is.

$\Theta(n^2)$ partial products.

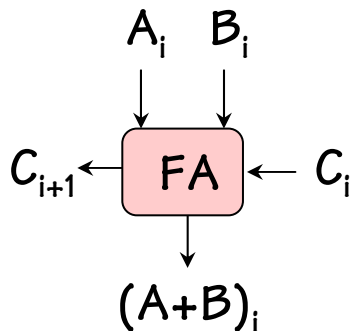
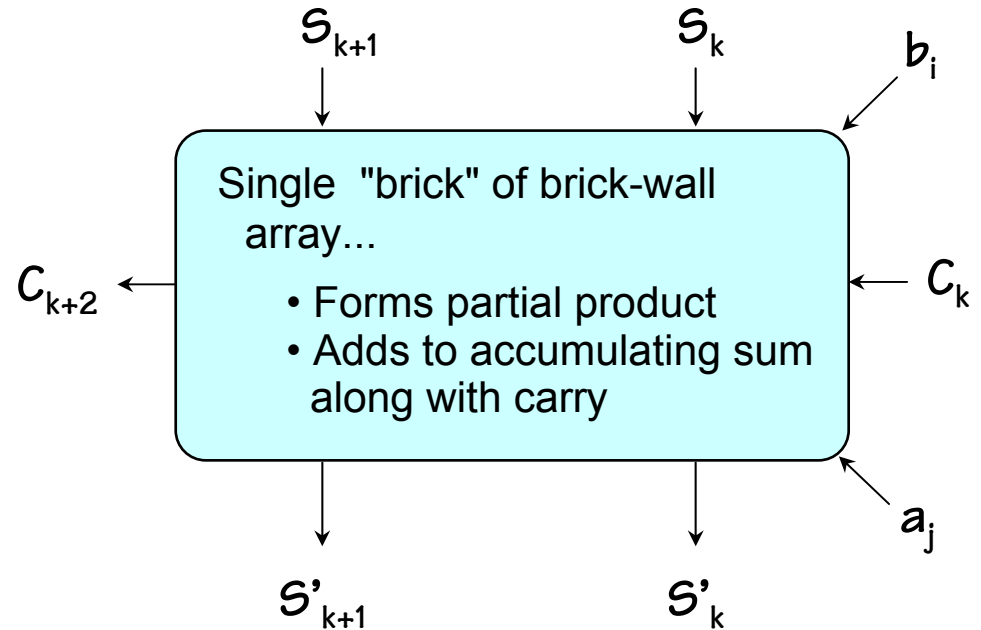
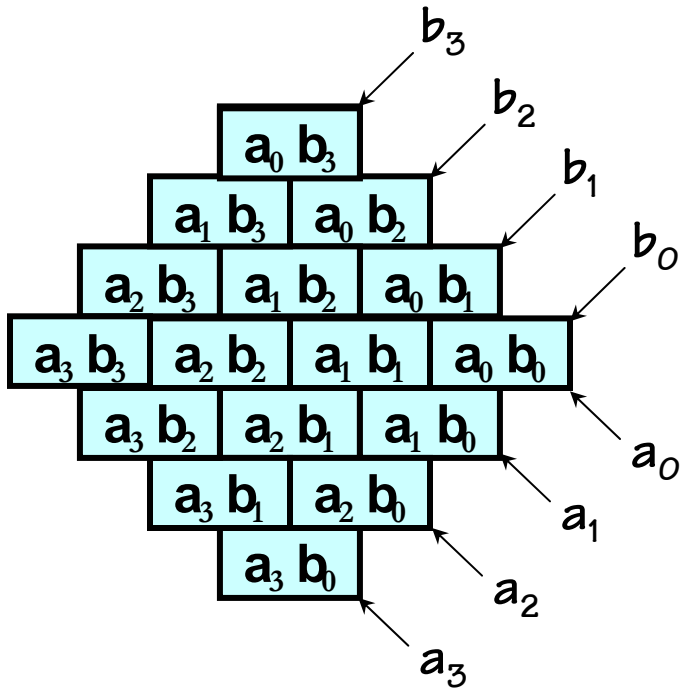
$\Theta(n^2)$ full adders.



How about n^2 blocks, each doing a little multiplication and a little addition?

Goal:

Array of Identical Multiplier Cells

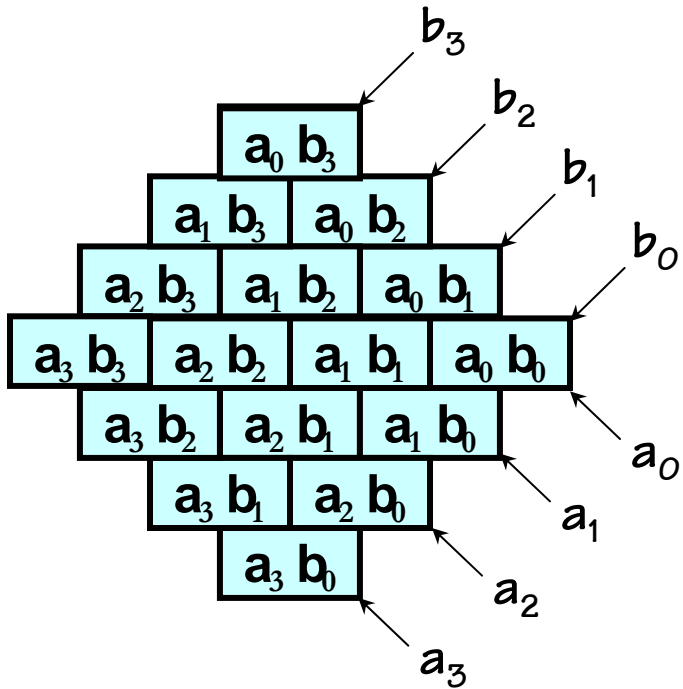


Necessary Component: Full Adder

Takes 2 addend bits plus carry bit. Produces sum and carry output bits.

CASCADE to form an n-bit adder.

Design of 1-bit multiplier "Brick":

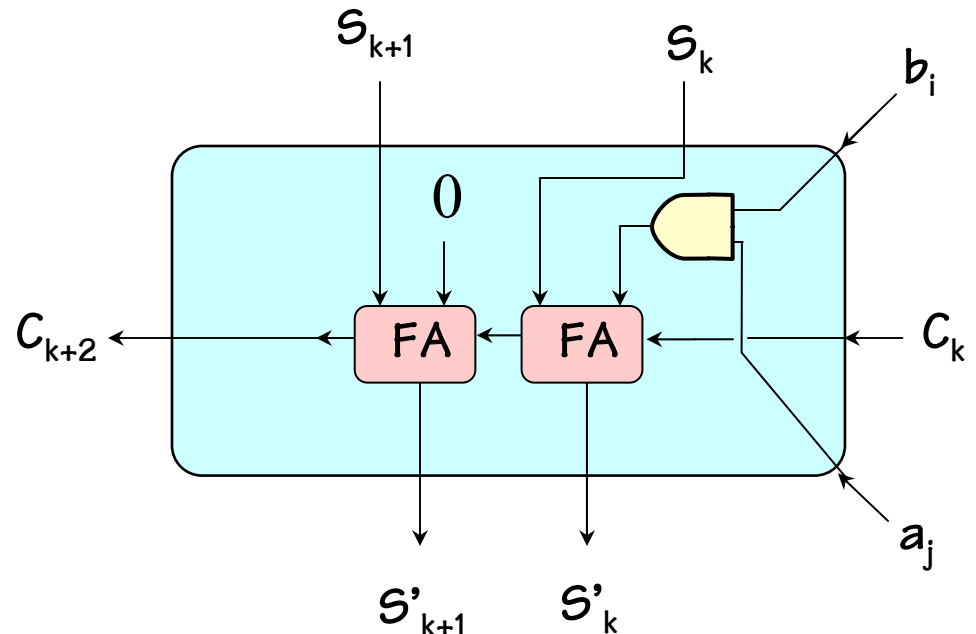


Array Layout:

- operand bits bused diagonally
- Carry bits propagate right-to-left
- Sum bits propagate down

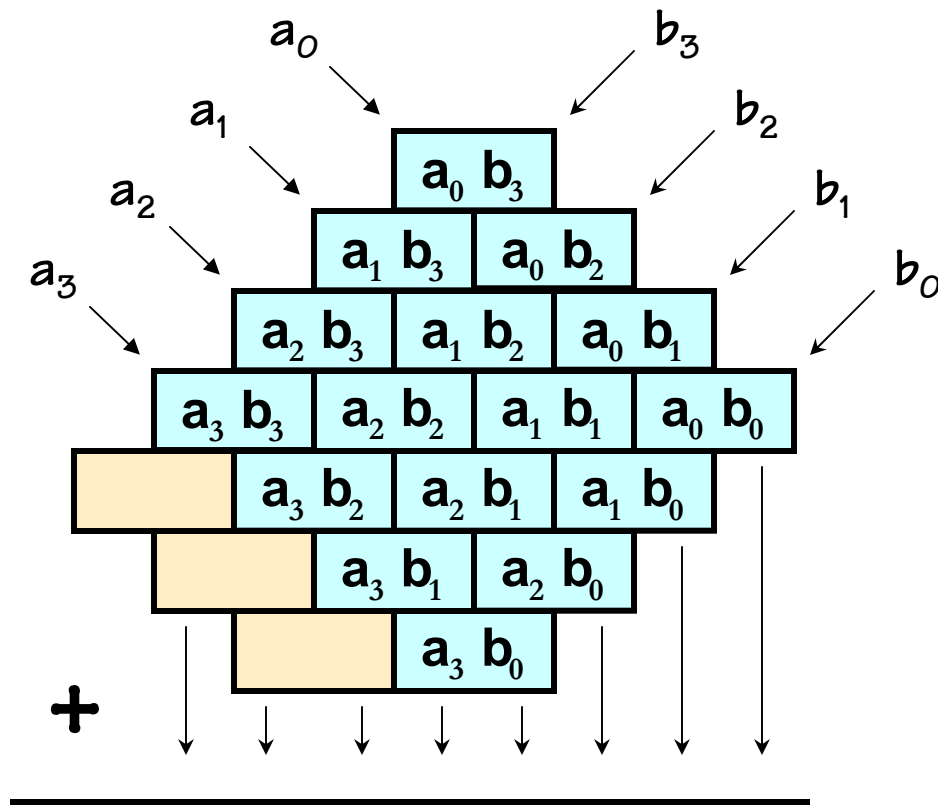
Brick design:

- AND gate forms 1x1 product
- 2-bit sum propagates from top to bottom
- Carry propagates to left



Multiplier Cookbook: Chapter 2

Combinational Multiplier:



Hardware for
n by n bits: $\Theta(n^2)$

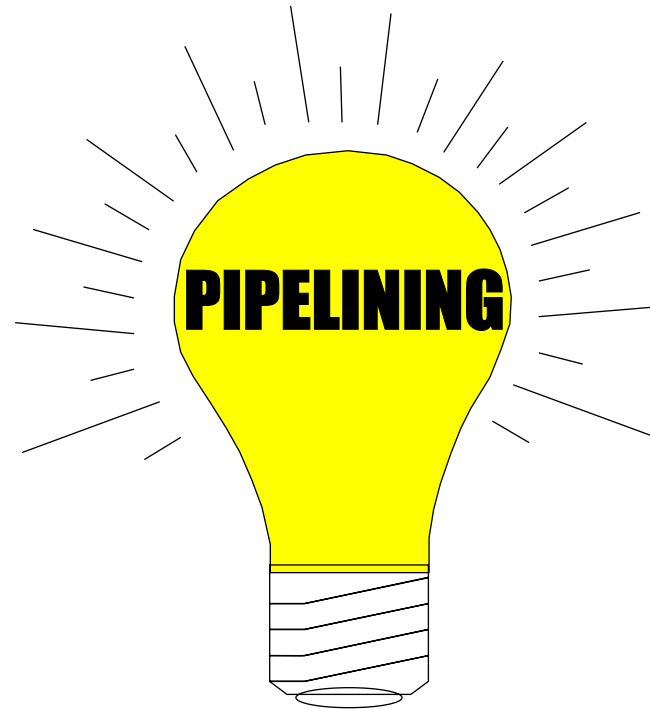
Latency: $\Theta(n)$

Throughput: $\Theta(1/n)$

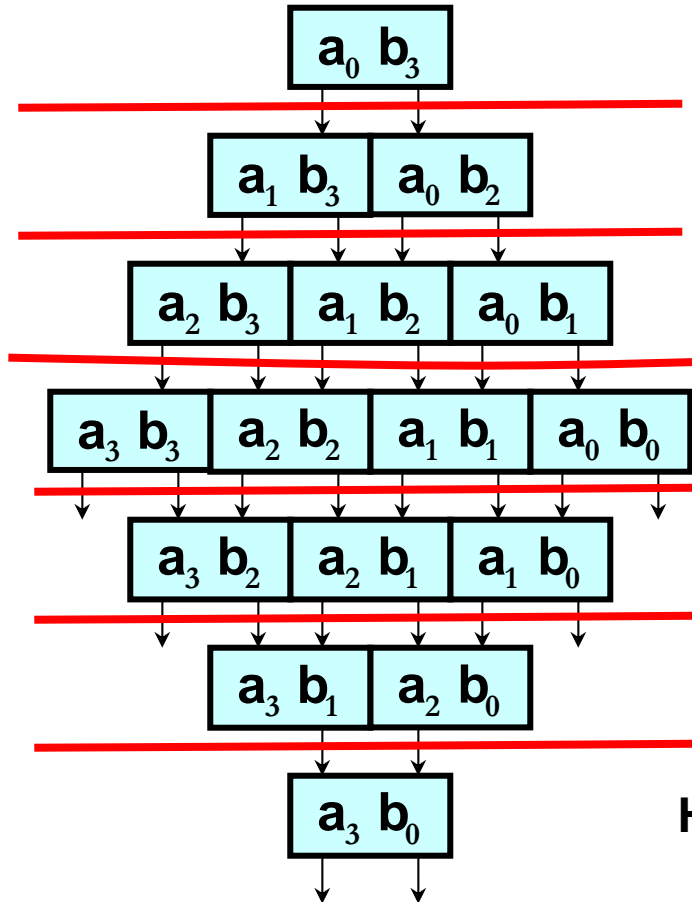
Combinational Multiplier: best bang for the buck?

Suppose we have LOTS of
multiplications.

Can we do better from a
cost/performance
standpoint?



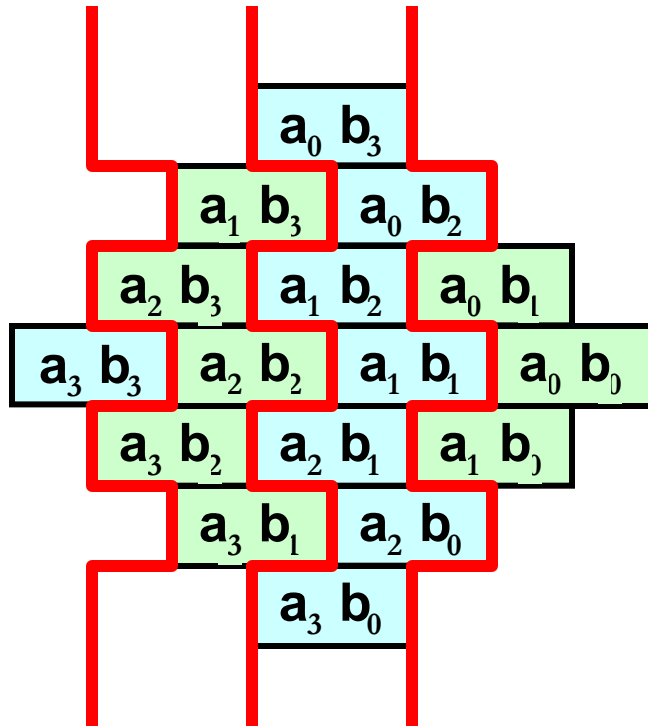
Stupid Pipeline Tricks



*gotta break
that long
carry chain!*

- Stages:** $\Theta(n)$
- Clock Period:** $\Theta(n)$
- Hardware cost for n by n bits:** $\Theta(n^2)$
- Latency:** $\Theta(n^2)$
- Throughput:** $\Theta(1/n)$

Even Stupider Pipeline Tricks



WORSE idea:

- Doesn't break long combinational paths
- NOT a well-formed pipeline...
 - ... different register counts on alternative paths
 - ... data crosses stage boundaries in *both directions!*

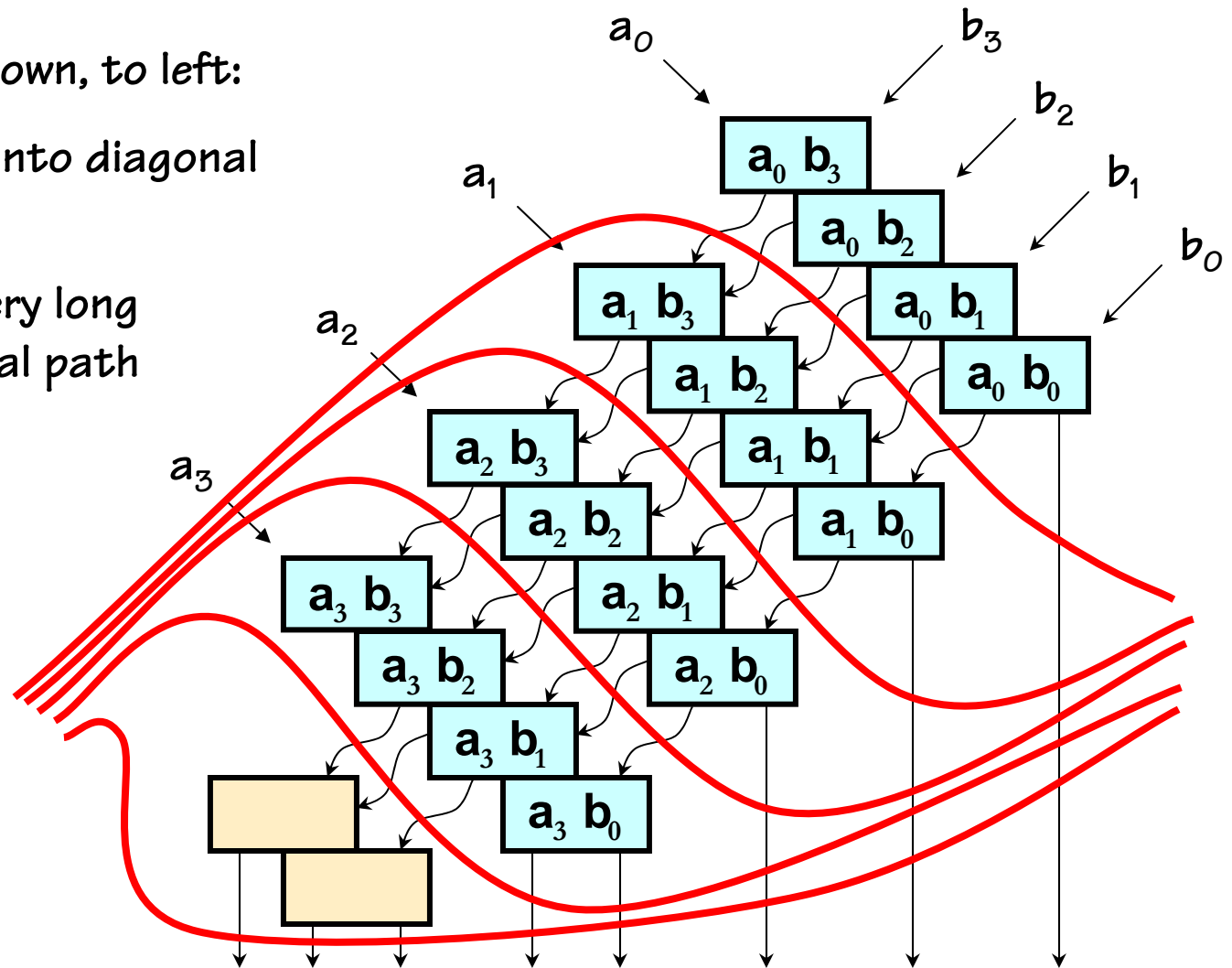
Back to basics:

what's the point of pipelining, anyhow?

Breaking $O(n)$ combinational paths

LONG PATHS go down, to left:

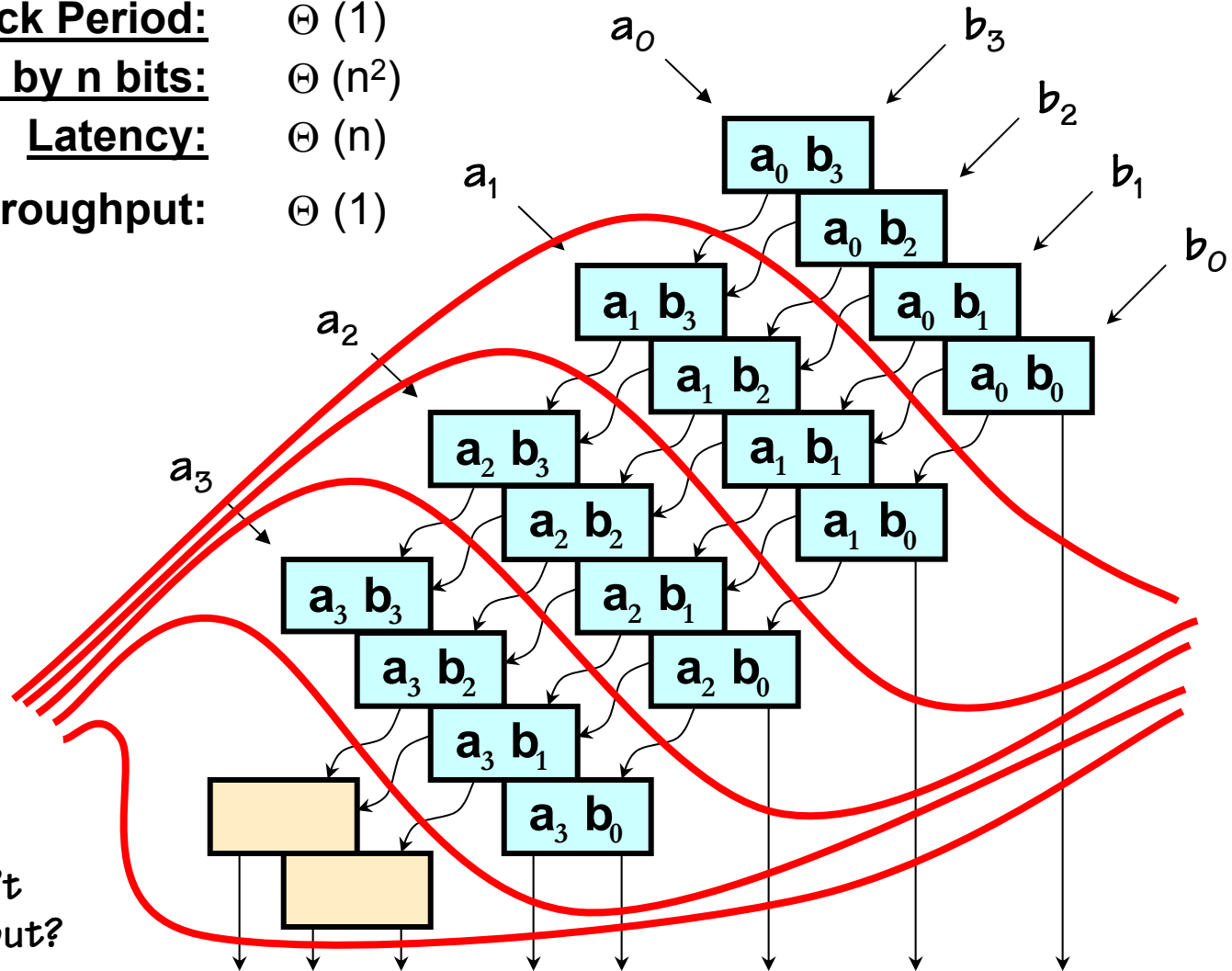
- Break array into diagonal slices
- Segment every long combinational path



GOAL: $\Theta(n)$ stages; $\Theta(1)$ clock period!

Multiplier Cookbook: Chapter 3

<u>Stages:</u>	$\Theta(n)$
<u>Clock Period:</u>	$\Theta(1)$
<u>Hardware cost for n by n bits:</u>	$\Theta(n^2)$
<u>Latency:</u>	$\Theta(n)$
<u>Throughput:</u>	$\Theta(1)$



- Well-formed pipeline (careful!)
- Constant (high!) throughput, independently of operand size.

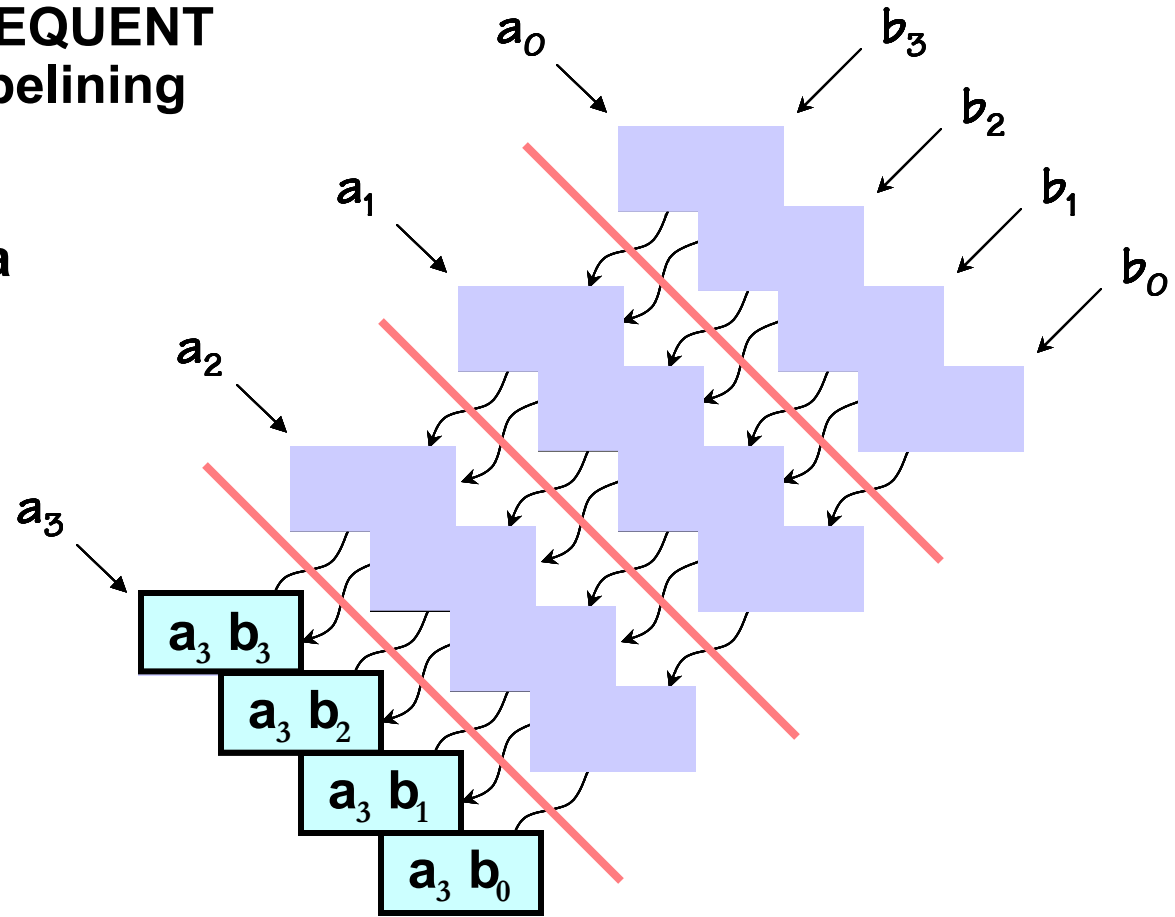
... but suppose we don't need the throughput?

Moving down the cost curve...

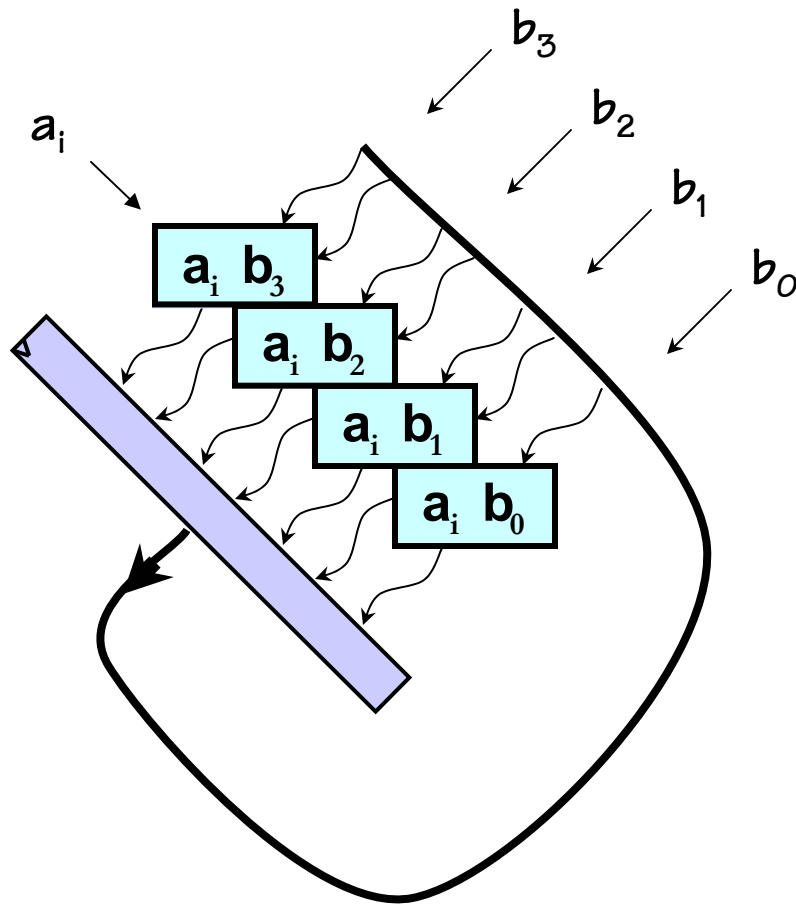
Suppose we have INFREQUENT multiplications... pipelining doesn't help us.

Can we do better from a cost/performance standpoint?

Hmmm, are all these extras really needed?



Multiplier Cookbook: Chapter 4



Sequential Multiplier:

- Re-uses a single n-bit “slice” to emulate each pipeline stage
- a operand entered serially
- Lots of details to be filled in...

Stages: 1

Clock Period: $\Theta(1)$ (constant!)

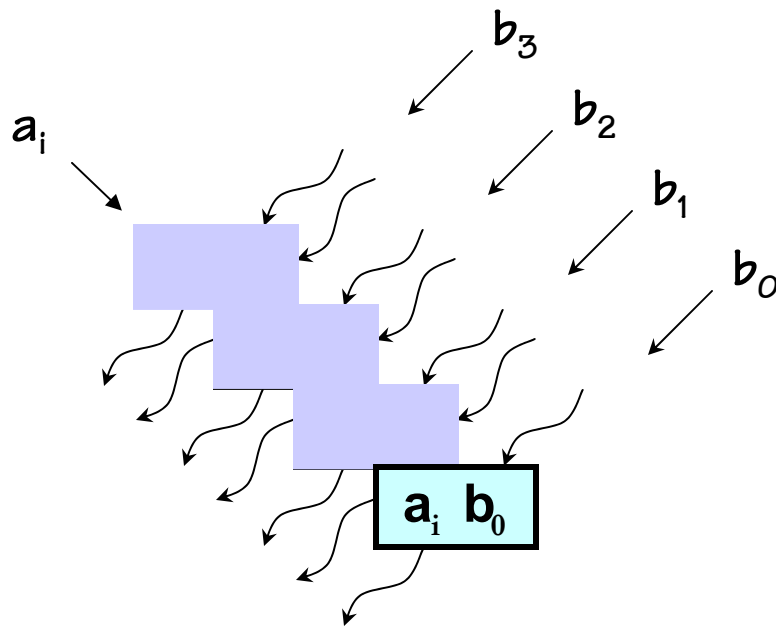
Hardware cost for n by n bits: $\Theta(n)$

Latency: $\Theta(n)$

Throughput: $\Theta(1/n)$

(Ridiculous?) Extremes Dept...

Cost minimization: how far can we go?



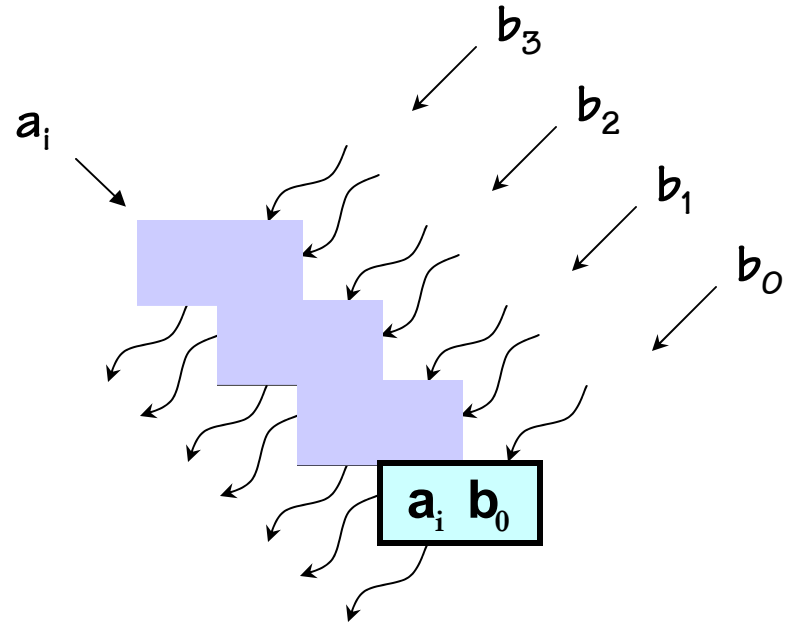
Suppose we want to minimize hardware (at any cost)...

- Consider *bit-serial*!
- Form and add 1-bit partial product per clock
- Reuse single “brick” for each bit b_j of slice;
- Re-use slice for each bit of a operand

Multiplier Cookbook: Chapter 5

Bit Serial multiplier:

- Re-uses a single brick to emulate an n-bit slice
- both operands entered serially
- $O(n^2)$ clock cycles required
- Needs additional storage (typically from existing registers)



Stages: $\Theta(1/n)$

Clock Period: $\Theta(1)$ (constant!)

Hardware cost for n by n bits: $\Theta(1) + ?$

Latency: $\Theta(n^2)$

Throughput: $\Theta(1/n^2)$

Summary:

Scheme:	\$	Latency	Thruput
Combinational	$\Theta(n^2)$	$\Theta(n)$	$\Theta(1/n)$
N-pipe	$\Theta(n^2)$	$\Theta(n)$	$\Theta(1)$
Slice-serial	$\Theta(n)$	$\Theta(n)$	$\Theta(1/n)$
Bit-serial	$\Theta(1)^*$	$\Theta(n^2)$	$\Theta(1/n^2)$

Lots more multiplier technology: fast adders, Booth Encoding, column compression, ...