Reconfigurable Computing

Customisation (Precision)

"It is the mark of an educated mind to rest satisfied with the degree of precision which the nature of the subject admits and not to seek exactness where only an approximation is possible."

– Aristotle

Philip Leong (philip.leong@sydney.edu.au)
School of Electrical and Information Engineering

http://phwl.org/talks

THE UNIVERSITY OF SYDNEY

Overview

- Number systems
  - Unsigned
  - Two's complement
  - Two's complement fractions
  - Floating Point
  - Logarithmic number system
- Case study
  - CORDIC algorithm

Number Systems

Number Systems - Unsigned

Unsigned integers are used to represent the nonnegative integers. An N-bit unsigned integer has a range $[0, 2^N - 1]$ and can be described in binary form, with u_i being the i'th binary digit:

$$U = (u_{N-1}u_{N-2}\ldots 0),\ u_i \in \{0, 1\}.$$

This represents the number

$$U = \sum_{i=0}^{N-1} u_i 2^i.$$

Number Systems – 2's Complement

$$X = (x_{N-1}x_{N-2}\ldots 0),\ x_i \in \{0,1\}.$$

X has a range of $[-2^{N-1}, 2^{N-1}-1]$ and represents

$$X = -x_{N-1}2^{N-1} + \sum_{i=0}^{N-2} x_i 2^i$$

Number systems – 2's Complement Fraction

The most significant $N-F$ bits of the number represent the integer part and the remaining F bits are the fractional part of the number

$$Y = (\overbrace{a_{N-1} \ldots a_F}^{\text{integer}} \overbrace{a_{F-1} \ldots a_0}^{\text{fraction}}).$$

This corresponds to a scaling of the two's complement integer representation by the factor $S = 2^{-F}$ and the two's complement fraction number Y represents

$$Y = 2^{-F} \times (-x_{N-1} 2^{N-1} + \sum^{N-2} x_i 2^i)$$

Note that the two's complement fraction $(N, 0)_{\mathcal{I}}$ corresponds to the two's complement integer case and $(N, N)_{\mathcal{I}}$ has a range of $[-1, 1)$.

Arithmetic Operations on 2's Complement Fractions

- If we wish to perform arithmetic on two (N,F) format 2's complement fractions
- Addition and subtraction
  - Normal addition
- Multiplication
  - An (N,F) multiplication gives a (2N, 2F) result so you need to do an arithmetic right shift by F bits from the 2N multiplier output
  - E.g. for (4,3) 0.75*0.75 = 0.110*0.110=00.100100 >> 3=0.100=0.5
- Question: how about division?

Number systems – Floating Point

$$Z = (\overbrace{a_0}^{\text{A}}\ \overbrace{b_{J-1}\ldots b_0}^{\text{B}}\ \overbrace{c_{F-1}\ldots b_0}^{\text{C}}).$$

A represents the sign S where

$$S = \begin{cases} +1 & \text{if } a_0 = 0 \\ -1 & \text{if } a_0 = 1 \end{cases}$$

The unsigned integers B and C are encoded representations of the exponent and mantissa respectively. The exponent E, is stored in a biased representation with $E = B - (2^{J-1} - 1)$. For normalized numbers, $B \neq 0$ and the significand is represented by $M = 1 + C \times 2^{-F}$. This is a two's complement fraction $(F+1, F)_{\mathcal{I}}$ with the most significant bit being implicitly set to 1. If $B = 0$, it is called a denormalized number, and there is no implicit 1 in the $(F, F)_{\mathcal{I}}$ fraction.

Number systems – Floating Point

$$Z = \begin{cases} S \times 2^E \times M & \text{if } (0 < B < 2^J - 1) \\ S \times 2^E \times (M - 1) & \text{if } (B = 0) \\ S \times \infty & \text{if } (B = 2^J - 1 \text{ and } C = 0) \\ NaN & \text{if } B = 2^J - 1 \text{ and } C \neq 0). \end{cases}$$

Number systems – Logarithmic Number System

The logarithmic number system (LNS) is a special case of floating point in which the mantissa is always 1 (i.e. only the sign and exponent fields are used). It has the advantages of simplified implementation at the expense of reduced precision. For an N bit LNS number, $(N, F)_{\mathcal{L}}$, the most significant bit is a zero flag, Z. Z is zero if the number is zero (since there is no log of zero), otherwise set. The next most significant bit is used for a sign bit and the rest of the number is the base 2 logarithm of the magnitude of the number to be represented in $(N-2, F)_{\mathcal{I}}$ two's complement fraction format. If E is the value of this two's complement fraction and S is defined as for floating point, then

$$L = \begin{cases} 0 & \text{if } Z = 0 \\ L = S \times 2^{E} & \text{if } Z = 1 \end{cases}$$

Reconfigurable Computing

The CORDIC algorithm

"Simplicity is the ultimate sophistication" - daVinci

Philip Leong (philip.leong@sydney.edu.au)
School of Electrical and Information Engineering

http://www.ee.usyd.edu.au/~phwl

THE UNIVERSITY OF SYDNEY

THE UNIVERSITY OF SYDNEY

Introduction

- COordinate Rotation DIgital Computer
- Efficient method to compute sin, cos, tan, $\sin^{-1}$, $\cos^{-1}$, $\tan^{-1}$, multiplication, division, $\sqrt{}$, sinh, cosh, tanh
  - Only uses shifts, additions and a very small lookup table

Theory

Rotations

Rotating [x y] by ϕ

$$
\begin{aligned}
x' &= x\cos(\phi) - y\sin(\phi) \\
y' &= y\cos(\phi) + x\sin(\phi).
\end{aligned}
$$

Rearranging

$$
\begin{aligned}
x' &= \cos(\phi)(x - y\tan(\phi)) \\
y' &= \cos(\phi)(y + x\tan(\phi)).
\end{aligned}
$$

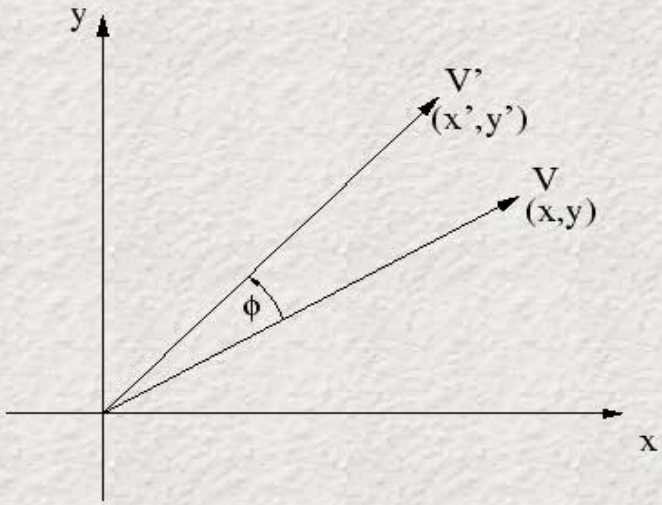


Figure 1.2: *Rotation of a vector V by the angle ϕ*

Key idea

$$\begin{aligned} x' &= \cos(\phi)(x - y\tan(\phi)) \\ y' &= \cos(\phi)(y + x\tan(\phi)). \end{aligned}$$

Can compute rotation ϕ in steps where each step is of size

$$\tan(\phi) = \pm 2^{-i}.$$

Iterative rotations

$$x_{i+1} = K_i(x_i - (y_i d_i 2^{-i}))$$
$$y_{i+1} = K_i(y_i + (x_i d_i 2^{-i})).$$

where $d_i = \pm 1$ and $K_i = \cos(\tan^{-1} 2^{-i})$

Choose d_i so that after n iterations the rotated angle is ϕ

K_i values

$$\cos(\tan^{-1} 2^{-i}) = 1/\sqrt{(1 + 2^{-2i})}.$$

$$K = \prod_{i=1}^{n} \frac{1}{\sqrt{(1 + 2^{-2i})}},$$

As n $\to \infty$, K $\to$ 0.6073 (constant factor which needs to be corrected for)

Actually it's easier to omit it and fix it later!

d_i decision (rotation mode)

Z_i is introduced to keep track of the angle that has been rotated (z0 = ϕ)

$$x_{i+1} = x_i - (y_i d_i 2^{-i})$$

$$y_{i+1} = y_i + (x_i d_i 2^{-i})$$

$$z_{i+1} = z_i - d_i \tan^{-1}(2^{-i})$$

$$d_i = \begin{cases} -1 & if\ z_i < 0 \\ +1 & otherwise \end{cases}$$

Notice we dropped the K! Rotated value is hence (Kx_n,Ky_n)

After n iterations

$$
\begin{aligned}
x_n &= \frac{1}{K}(x_0 cos(z_0) - y_0 sin(z_0)) \\
y_n &= \frac{1}{K}(y_0 cos(z_0) + x_0 sin(z_0)) \\
z_n &\approx 0.
\end{aligned}
$$

Question: What is the procedure to compute sin and cos?

THE UNIVERSITY OF SYDNEY

Computing sin(a) and cos(a)

1. Initialize (x,y,z)=(1,0,a)
2. Iterate through cordic
3. cos(a)=Kx and sin(a)=Ky

An easier way for this example is to change

step 1 to (x,y,z)=(k,0,a)

Sin(75)

0: xi=1.000000 yi=0.000000 zi=1.308997 k=1.000000 kx=1.000000 ky=0.000000

1: xi=1.000000 yi=1.000000 zi=0.523599 k=0.707107 kx=0.707107 ky=0.707107

2: xi=0.500000 yi=1.500000 zi=0.059951 k=0.632456 kx=0.316228 ky=0.948683

3: xi=0.125000 yi=1.625000 zi=-0.185027 k=0.613572 kx=0.076696 ky=0.997054

4: xi=0.328125 yi=1.609375 zi=-0.060673 k=0.608834 kx=0.199774 ky=0.979842

5: xi=0.428711 yi=1.588867 zi=0.001746 k=0.607648 kx=0.260505 ky=0.965472

6: xi=0.379059 yi=1.602264 zi=-0.029494 k=0.607352 kx=0.230222 ky=0.973138

7: xi=0.404094 yi=1.596342 zi=-0.013870 k=0.607278 kx=0.245397 ky=0.969423

d_i decision (vectoring mode)

$$d_i = \begin{cases} +1 & if\ y_i < 0 \\ -1 & otherwise. \end{cases}$$

$$\begin{aligned} x_n &= \frac{1}{K}\sqrt{(x_0^2 + y_0^2)} \\ y_n &\approx 0 \\ z_n &= z_0 + \tan^{-1}(y_0/x_0). \end{aligned}$$

- y_n minimized use to compute $\tan^{-1}$ and magnitude

Linear functions instead of trig

$$
\begin{aligned}
x_{i+1} &= x_i - 0(y_i d_i 2^{-i}) = x_i \\
y_{i+1} &= y_i + (x_i d_i 2^{-i}) \\
z_{i+1} &= z_i - d_i 2^{-i}
\end{aligned}
$$

$$
d_i = \begin{cases} -1 & if\ z_i < 0 \\ +1 & otherwise. \end{cases}
$$

After n iterations

$$
\begin{aligned}
x_n &= x_0 \\
y_n &= y_0 + x_0 z_0 \\
z_n &= 0,
\end{aligned}
$$

No need for K_n correction.

Division

$$d_i = \begin{cases} +1 & if\ y_i < 0 \\ -1 & otherwise. \end{cases}$$

$$x_n = x_0$$

$$y_n = y_0$$

$$z_n = z_0 - y_0/x_0.$$

No need for K_n correction.

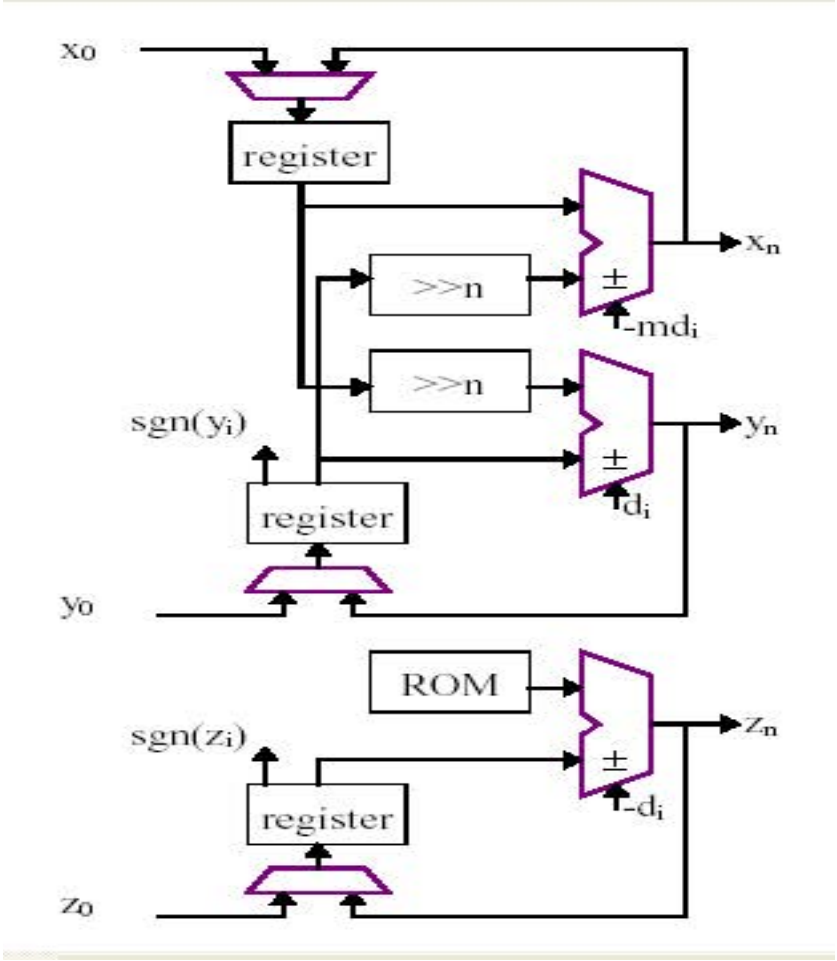
THE UNIVERSITY OF SYDNEY

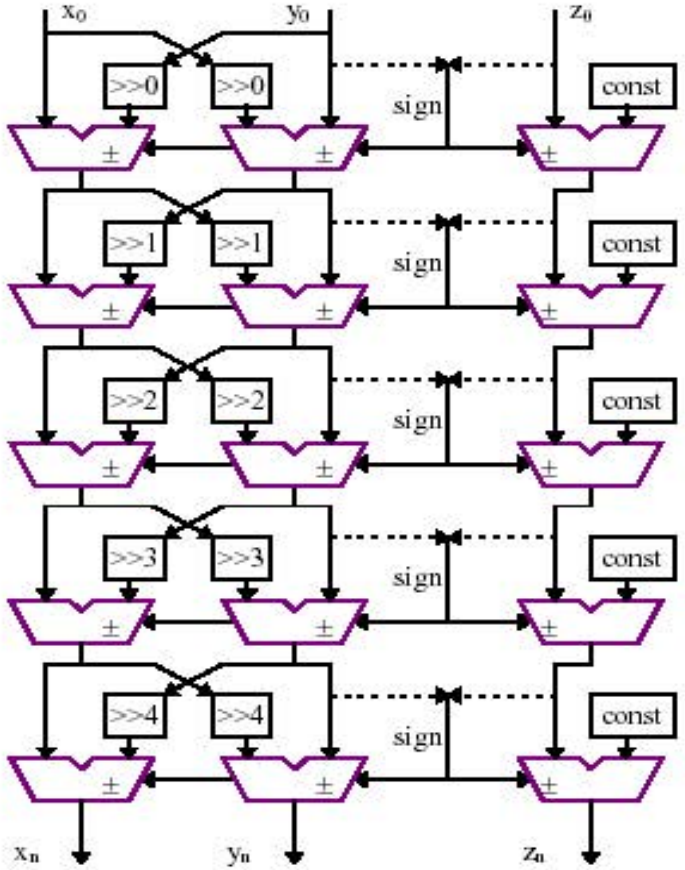
Hyperbolic functions

- Similarly, can get cosh and sinh using $\tanh^{-1}$ instead of $\tan^{-1}$
- Can also get ln and exp easily

Implementation

Andraka's iterative and unrolled cordic structure





THE UNIVERSITY OF SYDNEY

Implementation

- Can develop generalized cordic processors which can compute many different functions using similar hardware
- Implementations can be bit serial and/or pipelined as well

THE UNIVERSITY OF SYDNEY

Precision

- Need n iterations for n bits
- Converges for $-99.7 \leq z \leq 99.7$ (sum of all the angles $\tan^{-1}(2^{-i})$, $i = 0 .. n$)
  - must convert to this range first

THE UNIVERSITY OF SYDNEY

Conclusion

- CORDIC algorithms are an efficient method to compute many different functions
- Low area, high speed
- Used in calculators, DSPs, math coprocessors and supercomputers.

THE UNIVERSITY OF SYDNEY

References

- Ray Andraka, “A survey of CORDIC algorithms for FPGAs”, FPGA '98. Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays, Feb. 22-24, 1998, Monterey, CA. pp191-200 (http://www.andraka.com/cordic.htm)

THE UNIVERSITY OF SYDNEY

Review Question

- Calculate $\sqrt{2}/K$ using the CORDIC algorithm (4 iterations)
- Hint: use vectoring mode

Exploration: Multiplication

Binary Multiplication

	a	n bits
×	b	n bits
	a b	2n bits

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

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

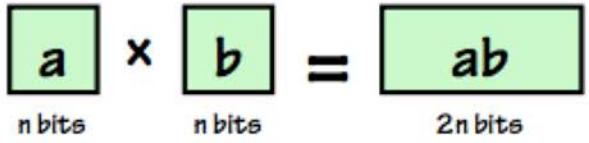
HARD PROBLEM: design circuit to multiply BIG (32-bit, 64-bit) numbers

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

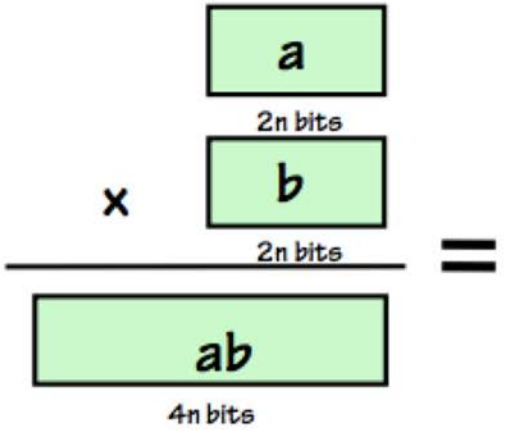
Engineering Principle:
Exploit STRUCTURE in problem.

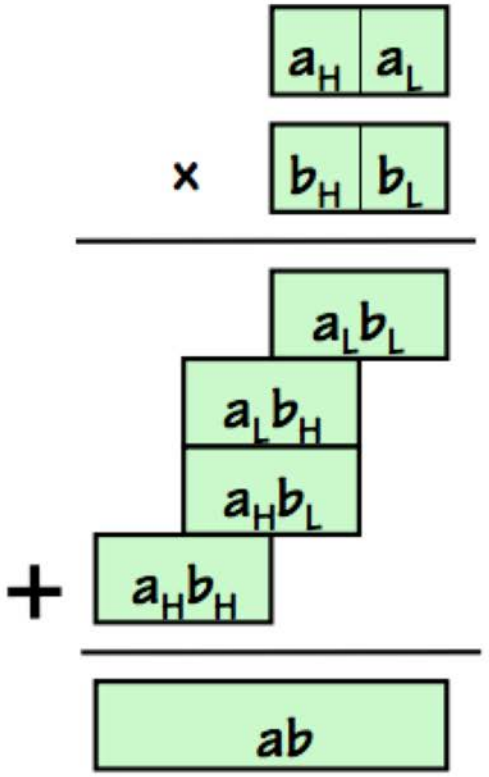
Composition of Multipliers

Given n-bit multipliers:



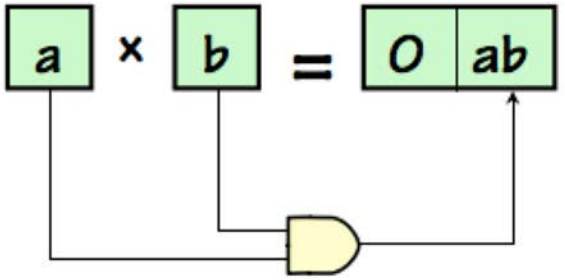
Synthesize 2n-bit multipliers:



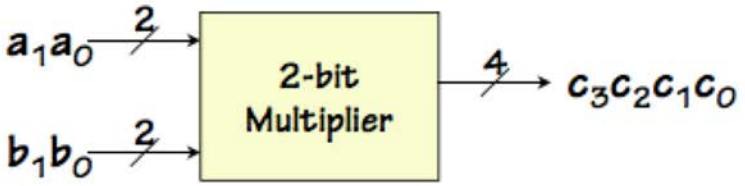


n=1: minimalist starting point

Multiplying two 1-bit numbers is pretty simple:



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

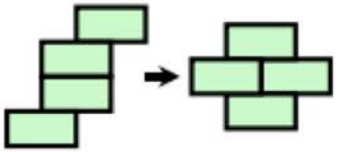


the logic gets more complex, but some optimizations are possible...

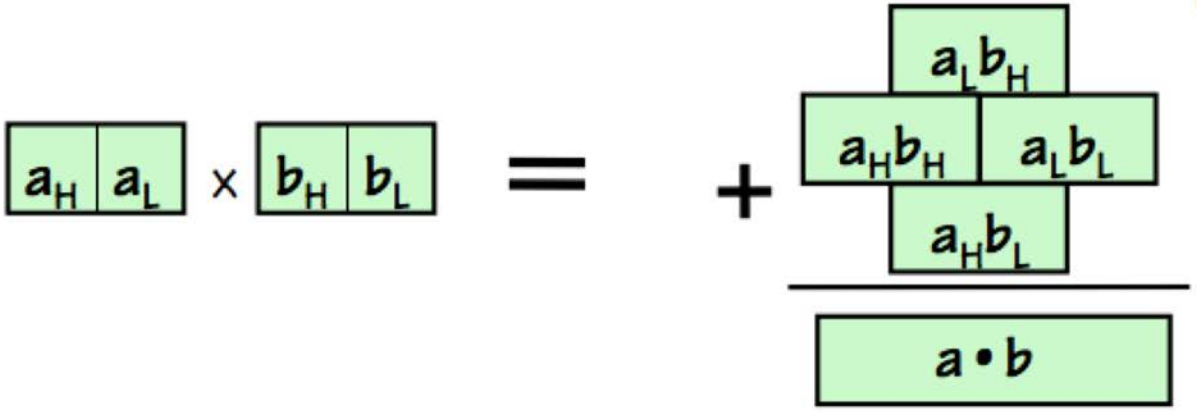
Induction Step

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.



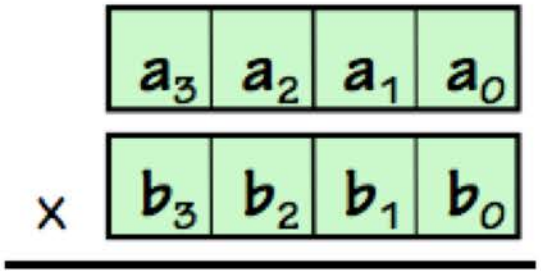
REGROUP partial products - 2 additions rather than 3!

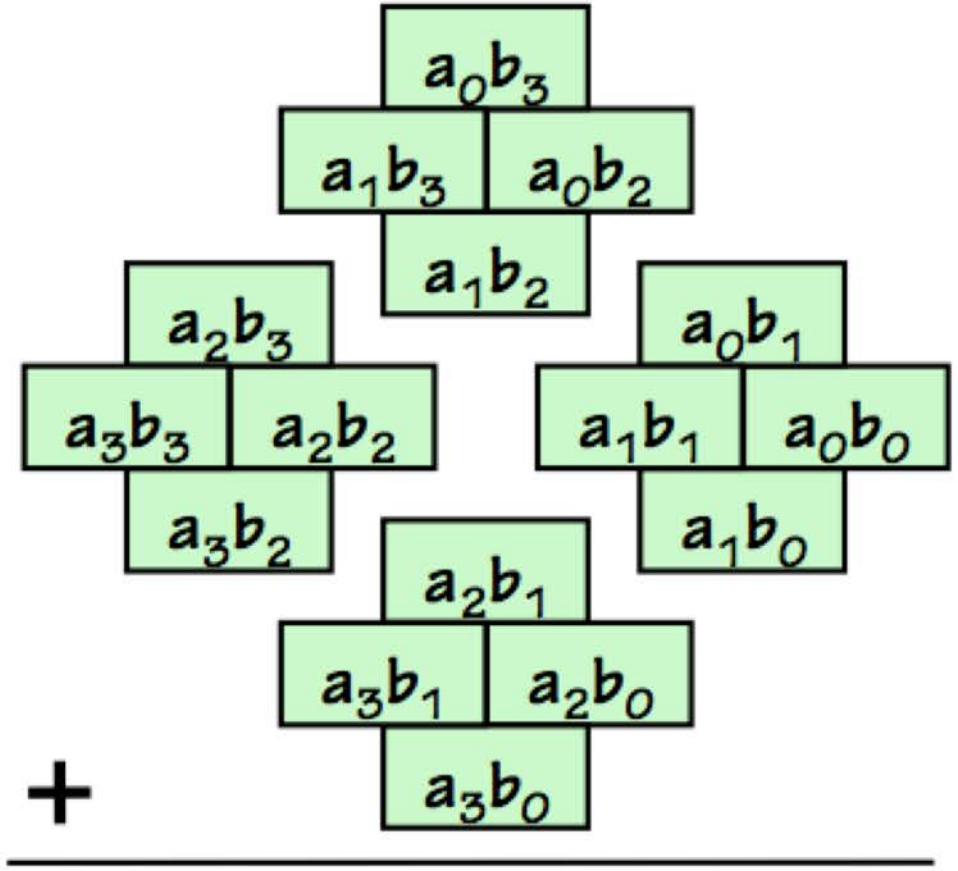


Induction: we can use the same structuring principle to build a 4n-bit multiplier from our newly-constructed 2n-bit ones...

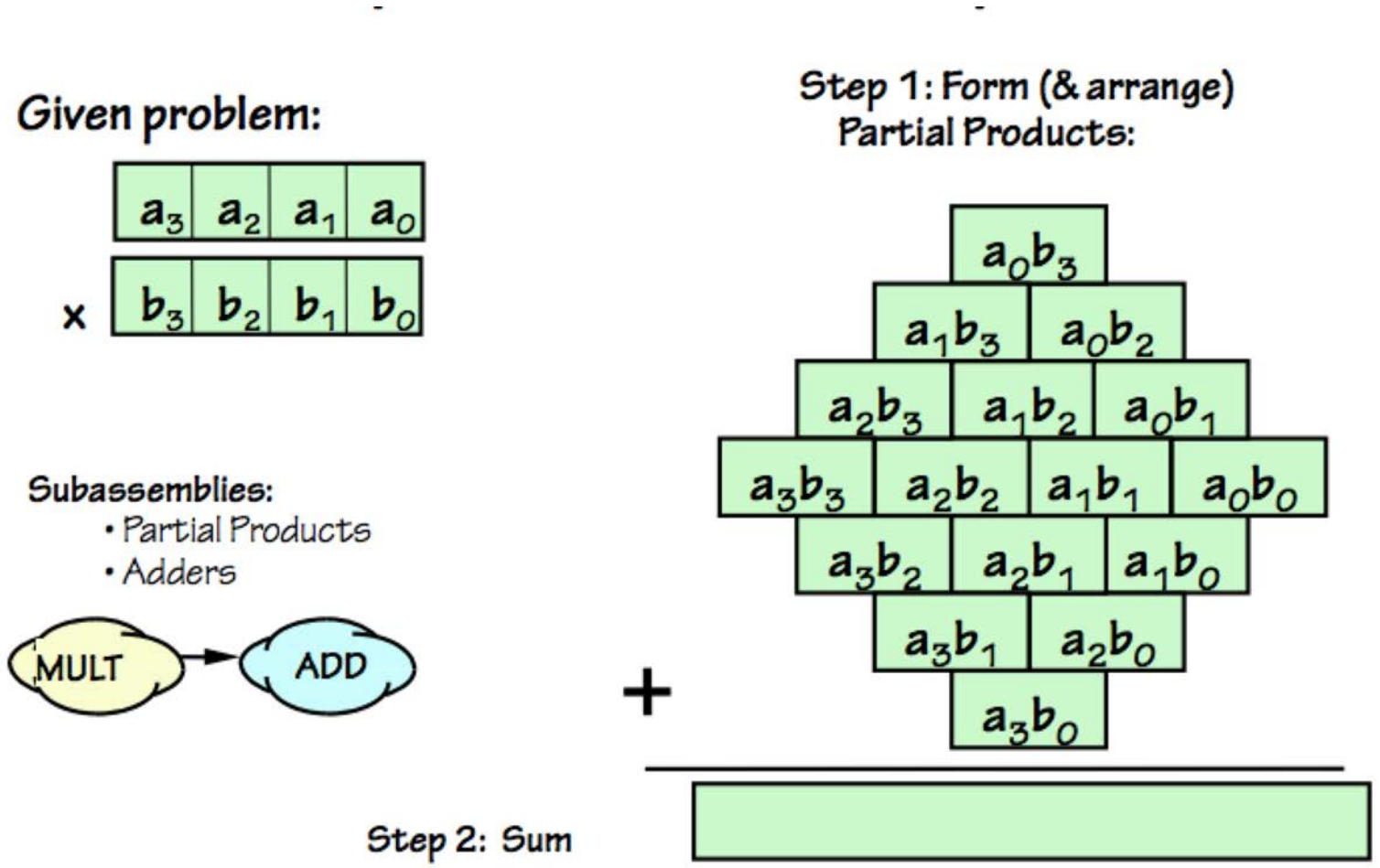
Brick Wall View of Partial Products

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





Building a Multiplier



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 $n \geq 0$

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

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

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

Example:

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

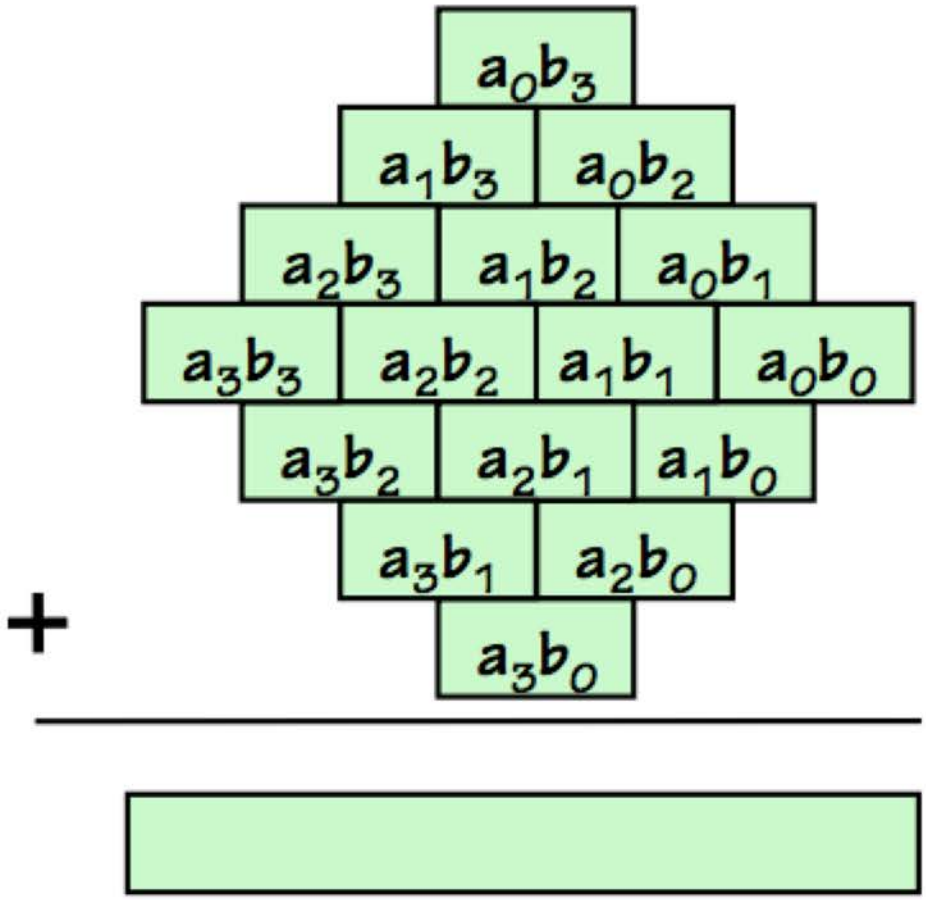
since

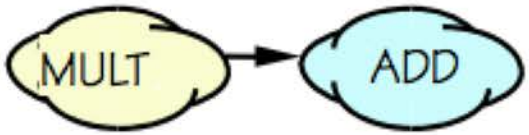
$$n^2 \leq (n^2+2n+3) \leq 2n^2$$

"almost always"

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

Observations





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

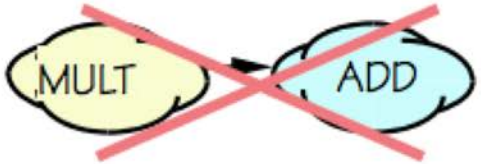
$\Theta(n^2)$ full adders.

Hmmm.

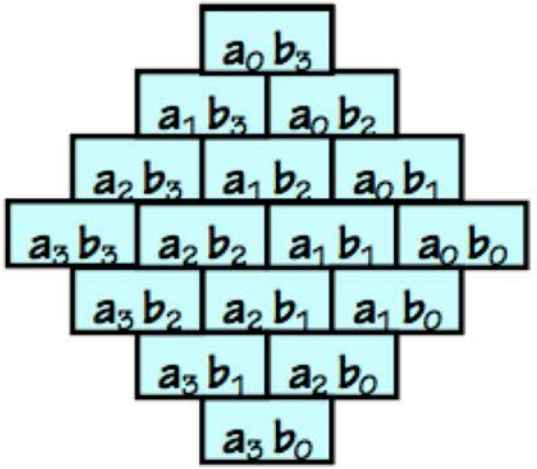
Repackaging Function

Engineering Principle #2:

Put the Solution where the Problem is.

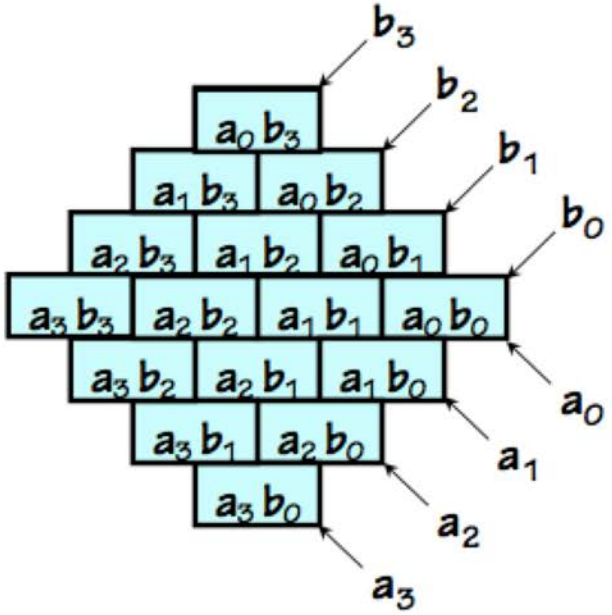


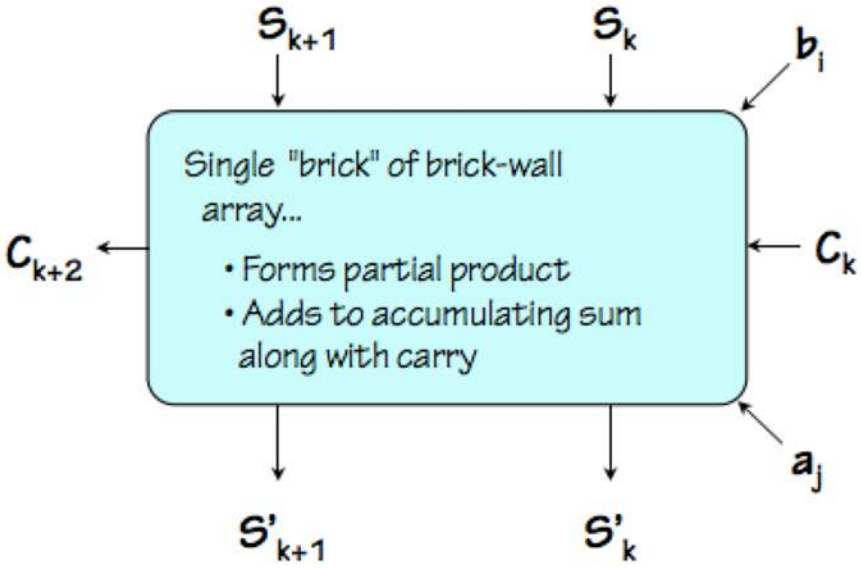
$\Theta(n^2)$ partial products.
$\Theta(n^2)$ full adders.

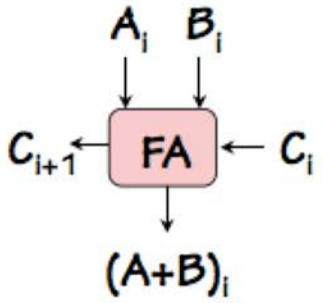


How about n^2 blocks, each doing a little multiplication and a little addition?

Goal: Array of Identical Cells





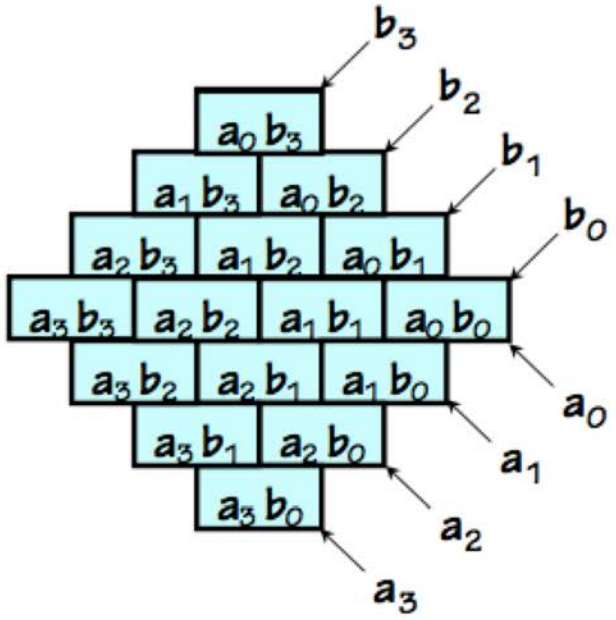


Necessary Component: Full Adder

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

CASCADE to form an n-bit adder.

1-Bit Multiplier "Brick"



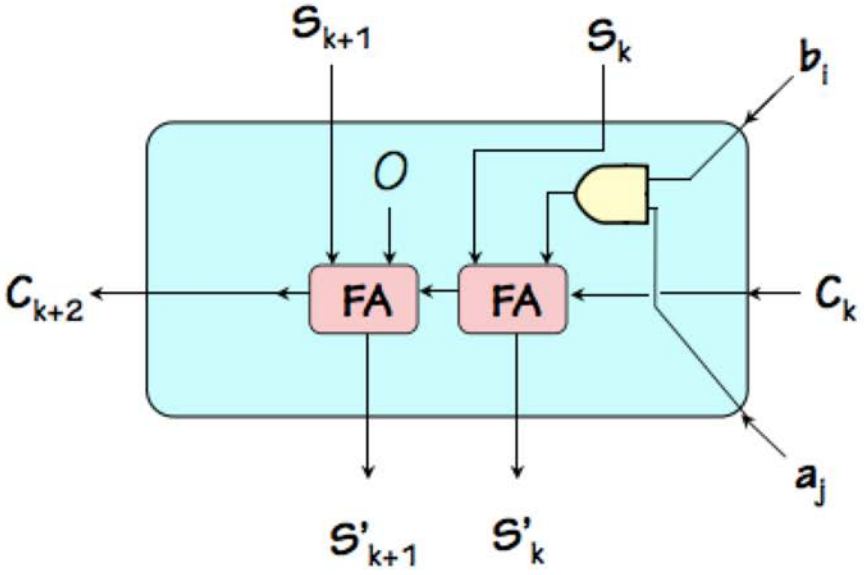
Brick design:

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

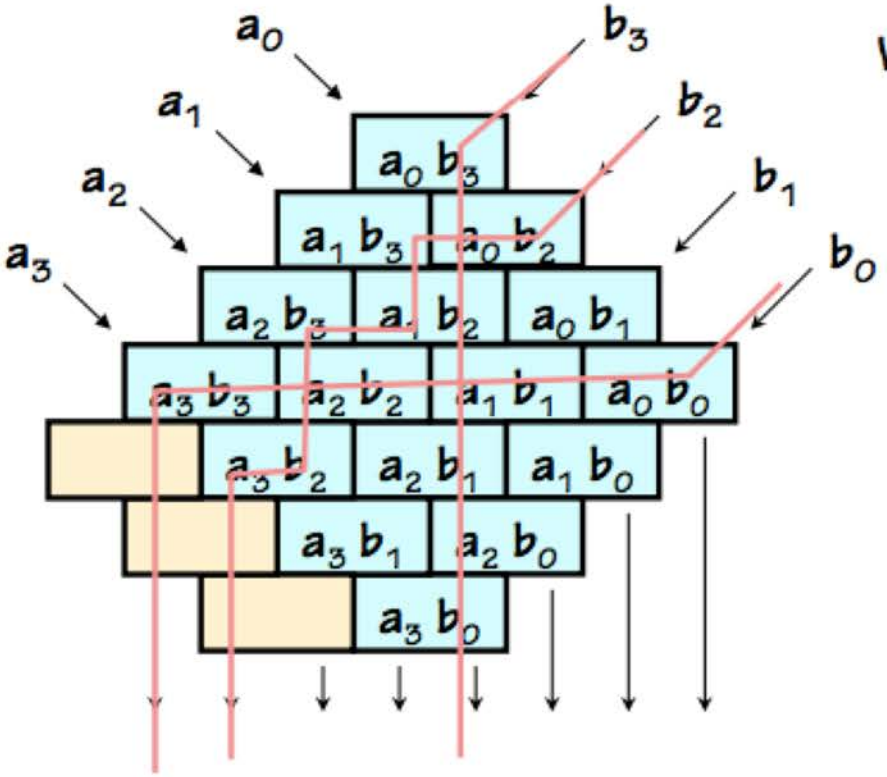
Wastes some gates… but consider (say) optimized 4x4-bit brick!

Array Layout:

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



Here's our combinational multiplier:



What's its propagation delay?

Naive (but valid) bound:

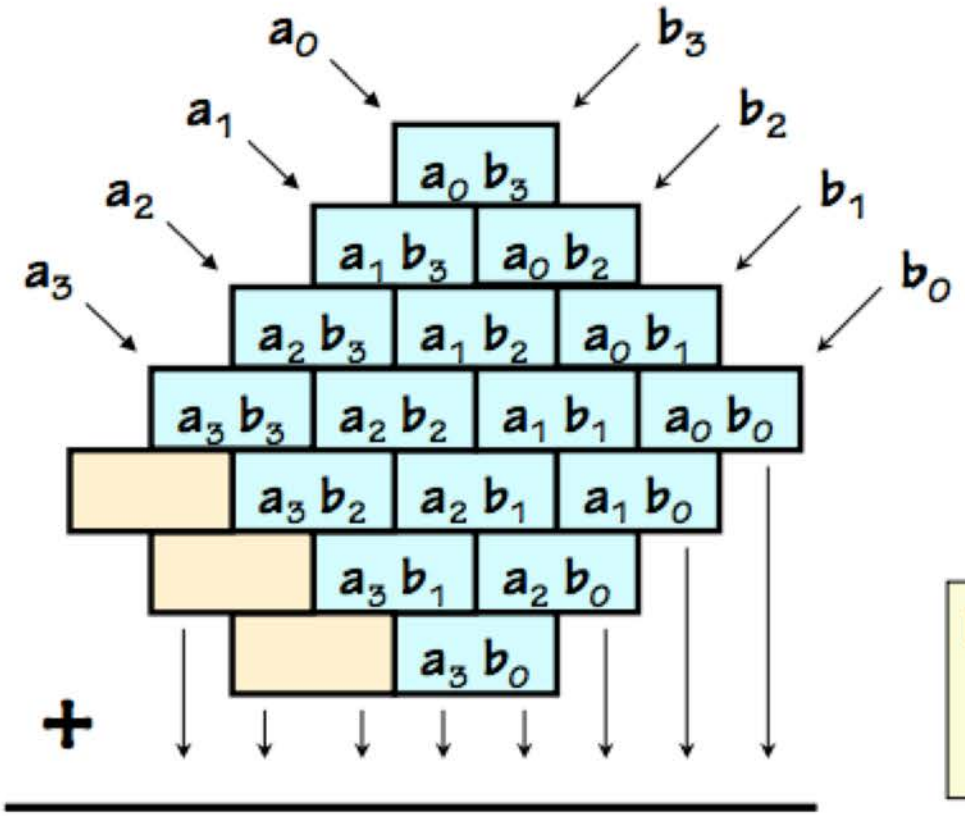
- $O(n)$ additions
- $O(n)$ time for each addition
- Hence $O(n^2)$ time required

On closer inspection:

- Propagation only toward left, bottom
- Hence longest path bounded by length + width of array: $O(n+n) = O(n)$!

Improved Multiplier

Combinational Multiplier:



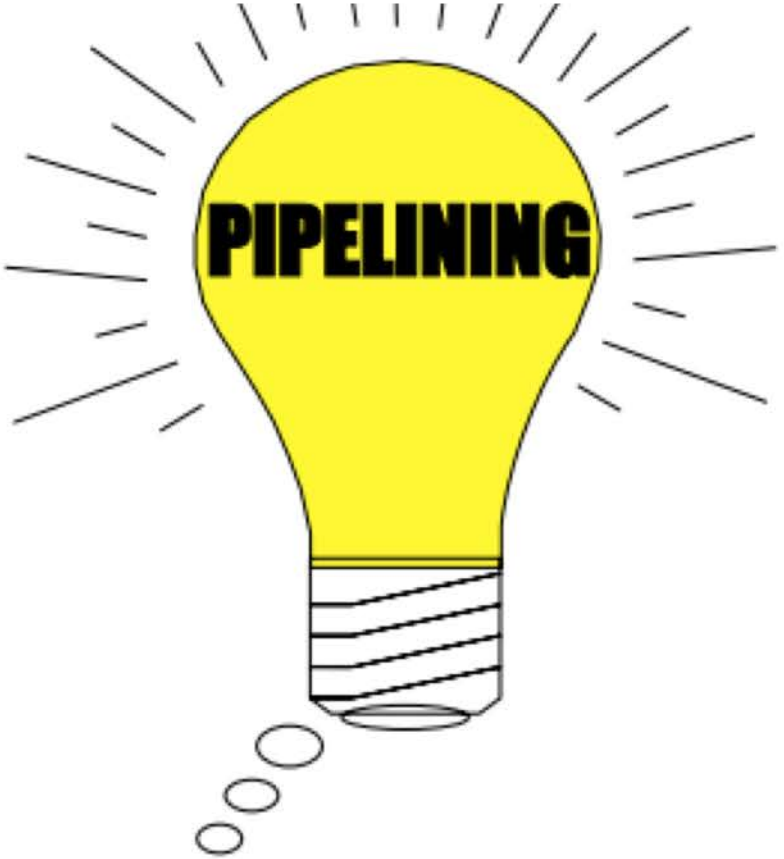
Hardware for n by n bits:	$\Theta(n^2)$
Latency:	$\Theta(n)$
Throughput:	$\Theta(1/n)$

Note: lots of tricks are available to make a faster combinational multiplier…

Combinational Multiplier Tradeoffs

Suppose we have LOTS of multiplications.

Can we do better from a cost/performance standpoint?

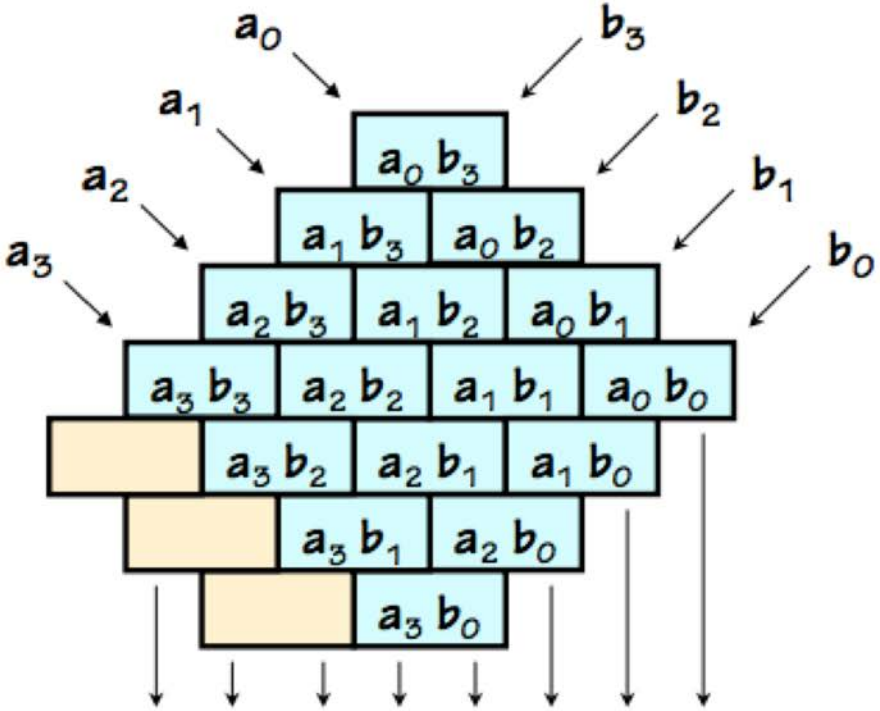


Pipelining

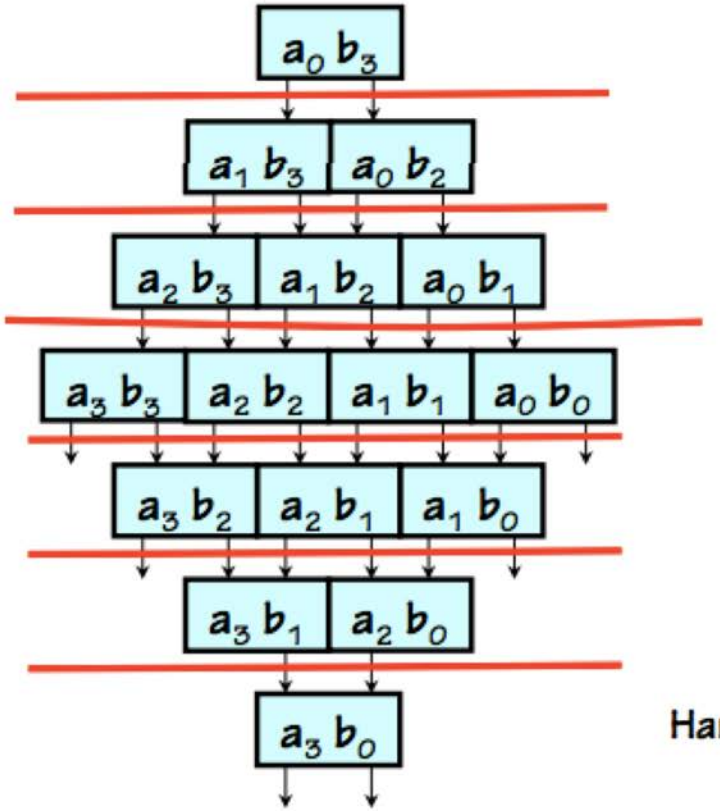
WE HAVE:

- Pipeline rules - "well formed pipelines"
- Plenty of registers
- Demand for higher throughput.

What do we do? Where do we define stages?



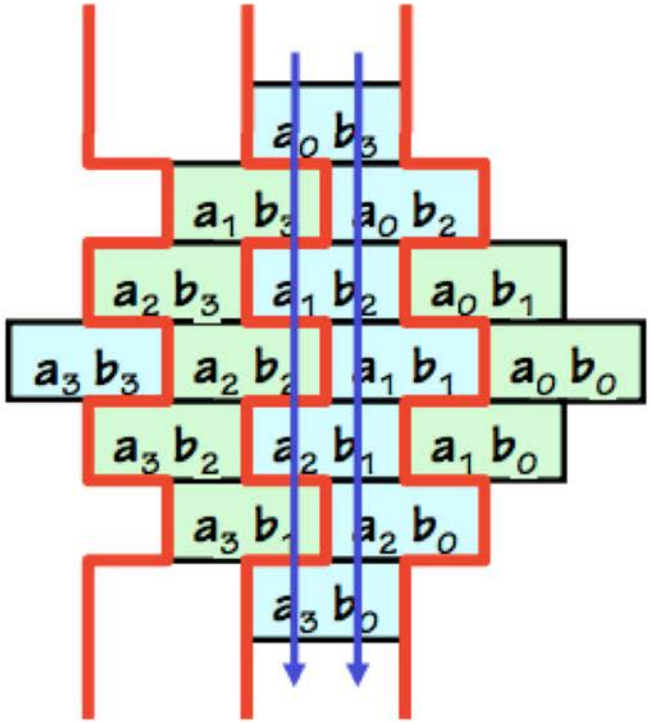
Bad Design



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:	$\Theta(1/n)$

Worse Design



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!*

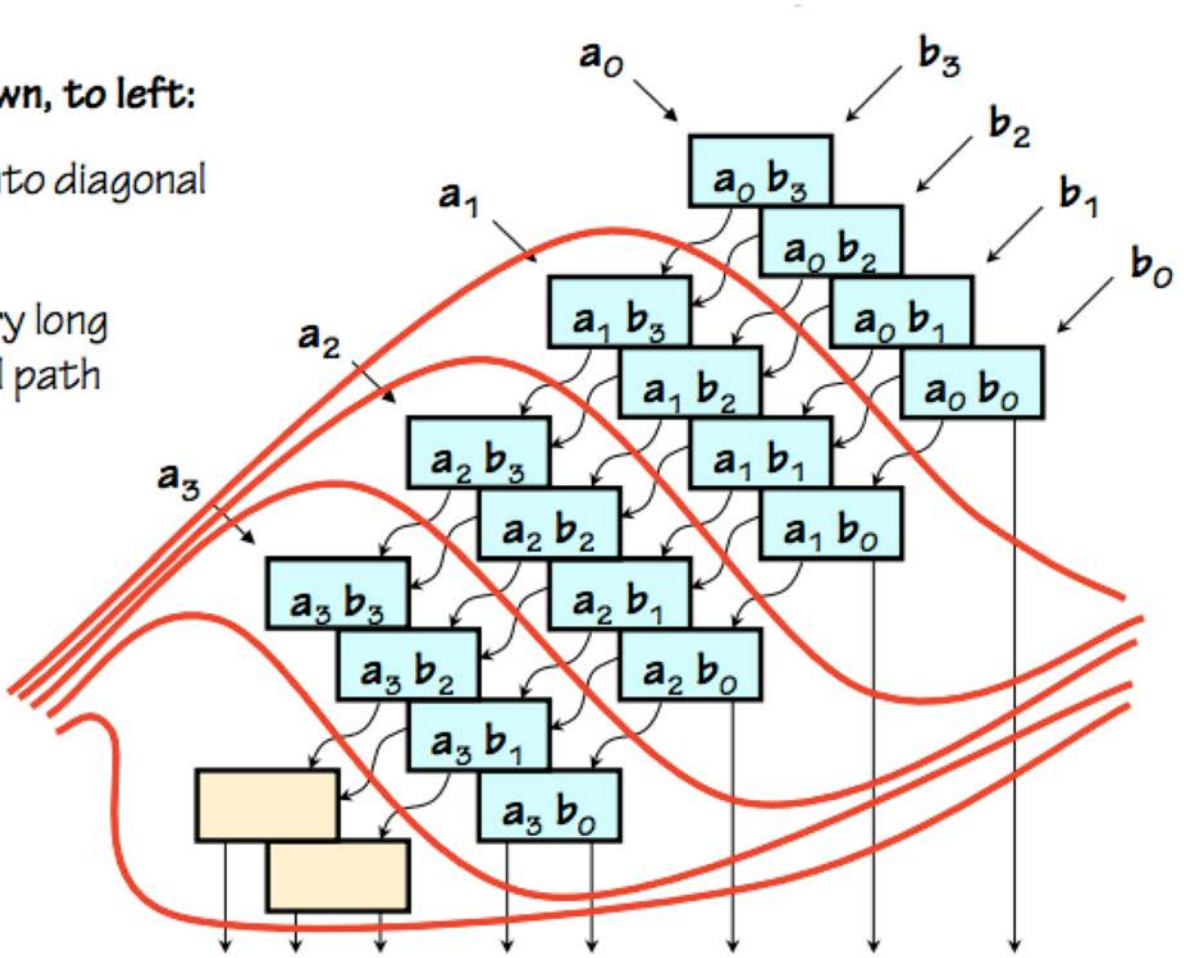
Back to basics:

what's the point of pipelining, anyhow?

Breaking O(n) Combinational Paths

LONG PATHS go down, to left:

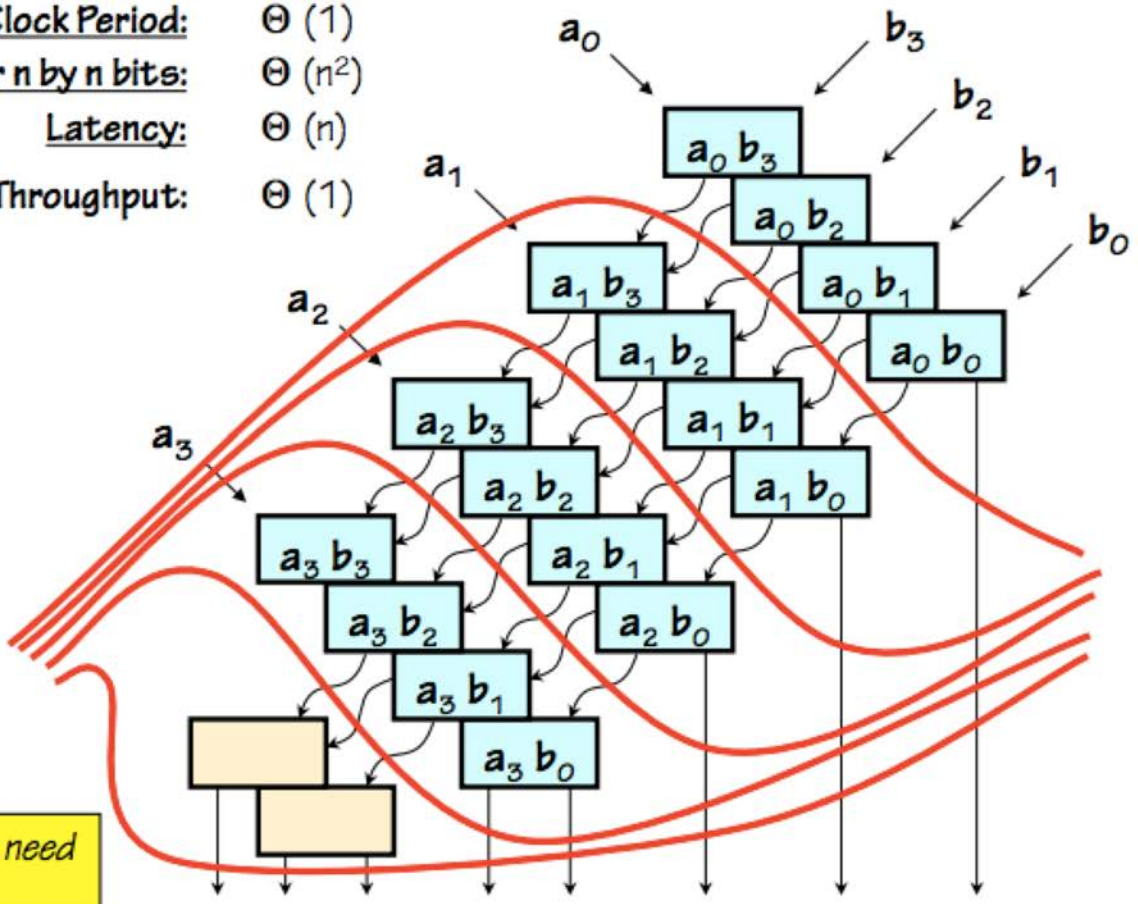
- Break array into diagonal slices
- Segment every long combinational path



GOAL: Θ (n) stages; Θ (1) clock period!

High Throughput Design

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



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

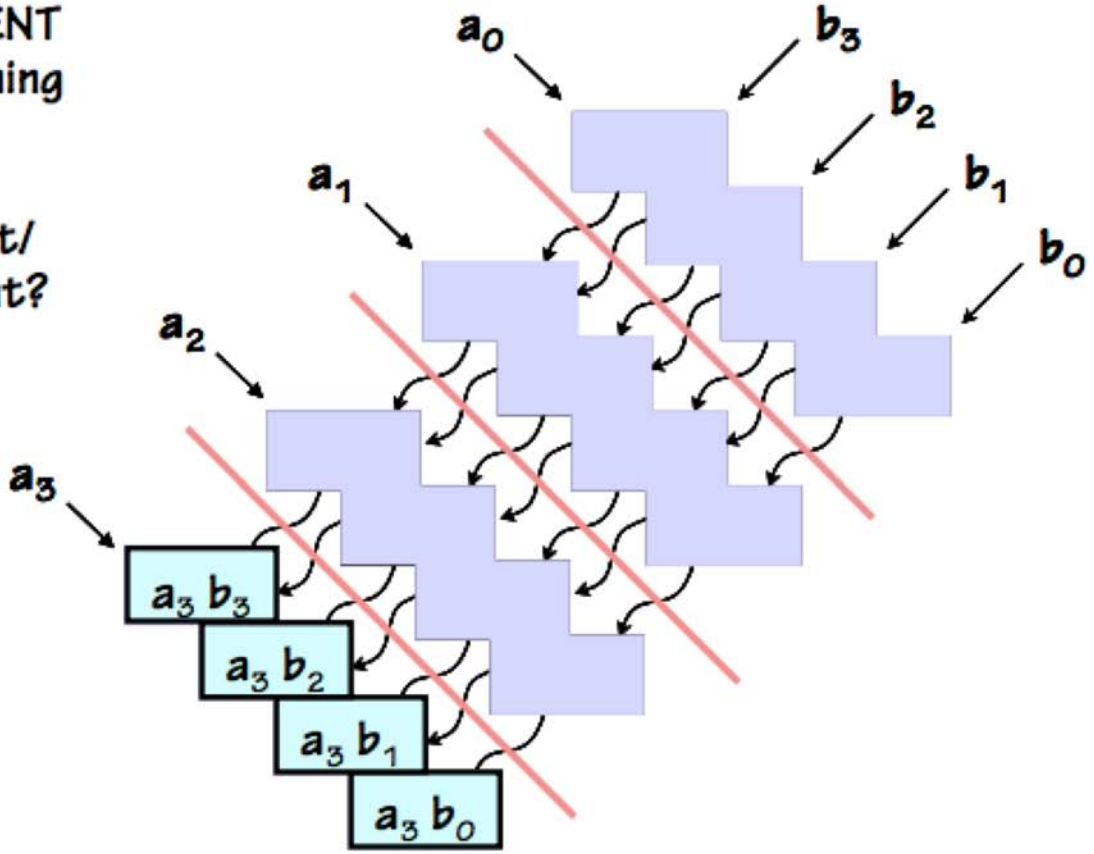
... but suppose we don't *need* the throughput?

Small Area Design

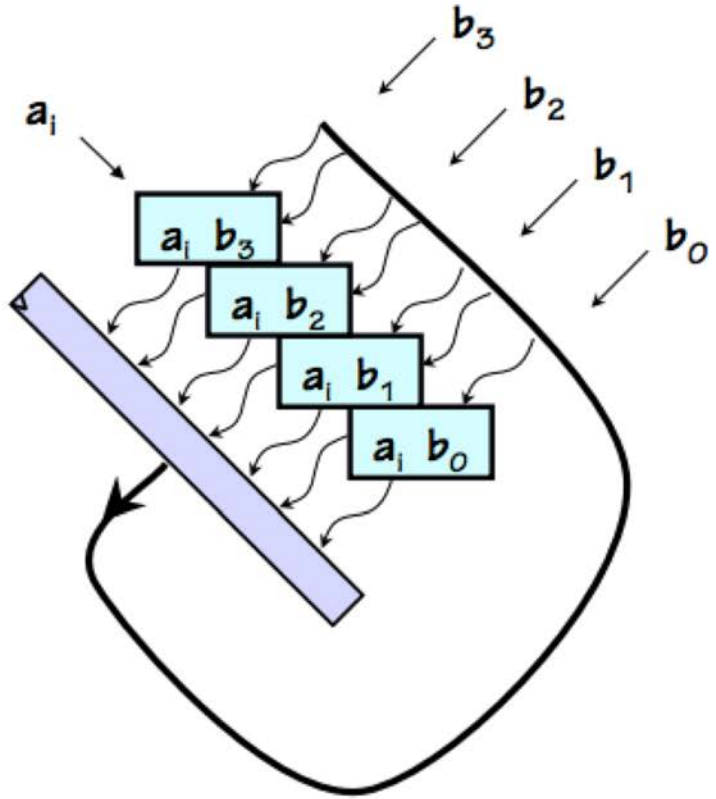
Suppose we have INFREQUENT multiplications... pipelining doesn't help us.

Can we do better from a cost/ performance standpoint?

Hmmm, do I really need all these extras?



Even Smaller Area Design



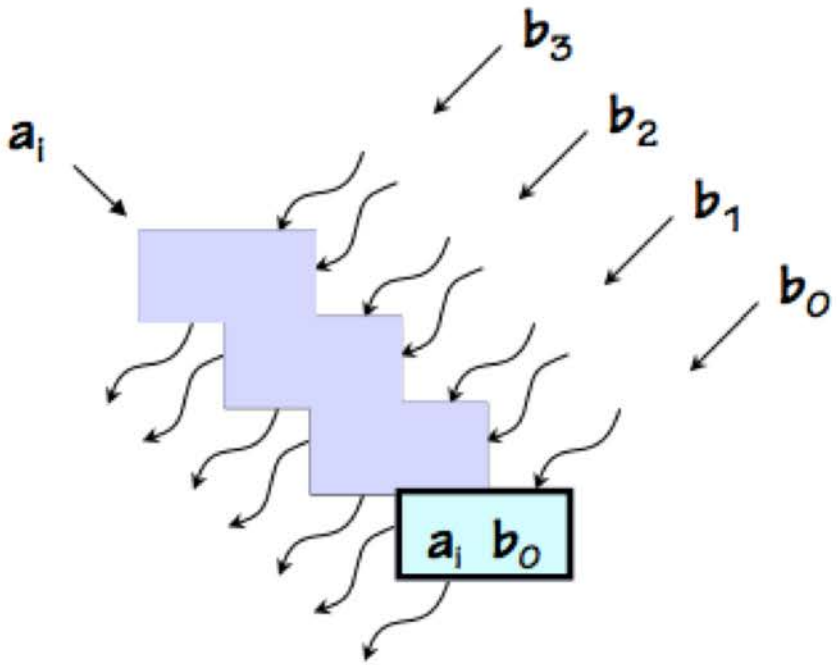
Sequential Multiplier:

- 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:	$\Theta(1/n)$

Minimum Area Design

Cost minimization: how far can we go?



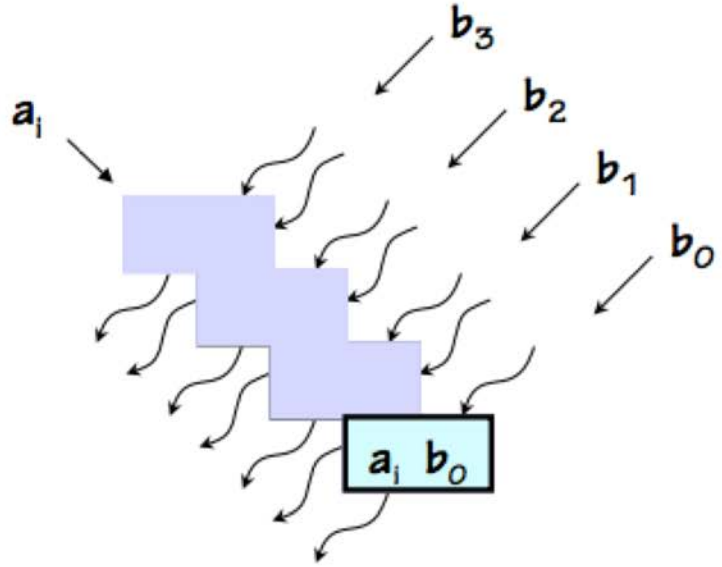
Suppose we want to minimize hardware (at any cost)…

- 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

Minimum Area Design

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, ...