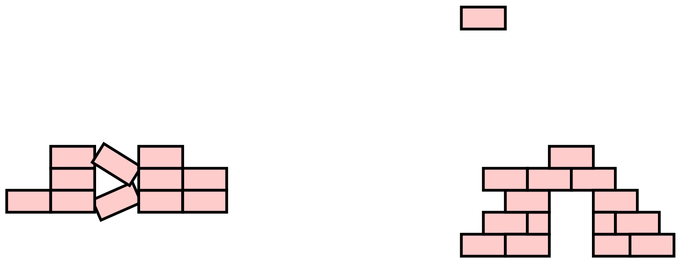
Cost/Performance Tradeoffs:

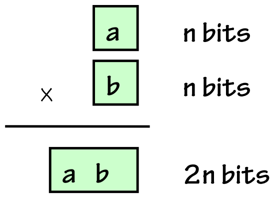
a case study

Digital Systems Architecture 1.01



Lab #3 due *tonight!*

Binary Multiplication



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

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

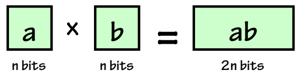
HARD PROBLEM: design circuit to multiply BIG (32-bit, 64-bit) numbers

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

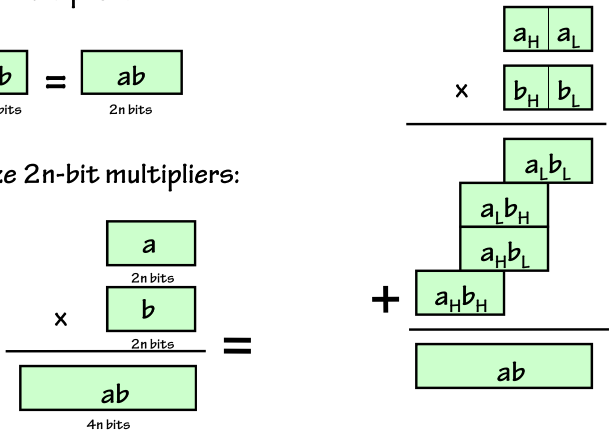
Engineering Principle:
Exploit STRUCTURE in problem.

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

Given n-bit multipliers:

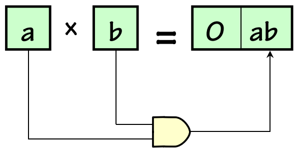


Synthesize 2n-bit multipliers:

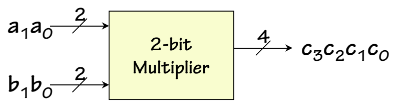


Our Basis:

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.

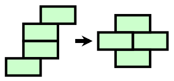


the logic gets more complex, but some optimizations are possible...

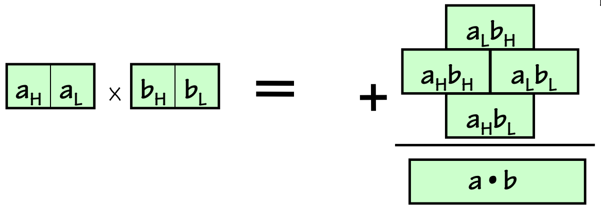
Our induction step:

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.



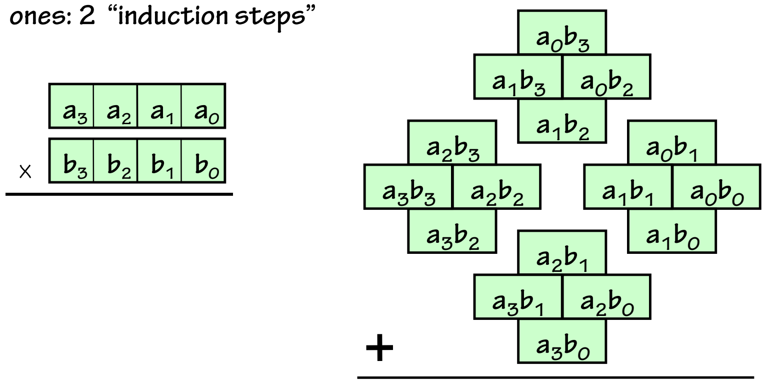
REGROUP partial products - 2 additions rather than 3!



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

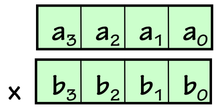
Brick Wall view
of partial products

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

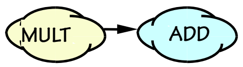


Multiplier Cookbook: Chapter 1

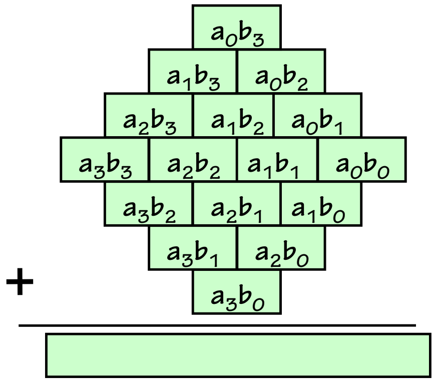
Given problem:



Subassemblies:
- Partial Products
- Adders



Step 1: Form (& arrange) Partial Products:


a_0b_3
a_1b_3 a_0b_2
a_2b_3 a_1b_2 a_0b_1
a_3b_3 a_2b_2 a_1b_1 a_0b_0
a_3b_2 a_2b_1 a_1b_0
a_3b_1 a_2b_0
a_3b_0
+

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 $n \geq 0$

$$\underbrace{c_1 \cdot f(n) \leq g(n)}_{} \underbrace{\leq c_2 \cdot f(n)}_{g(n) = O(f(n))}$$

Example:

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

since

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

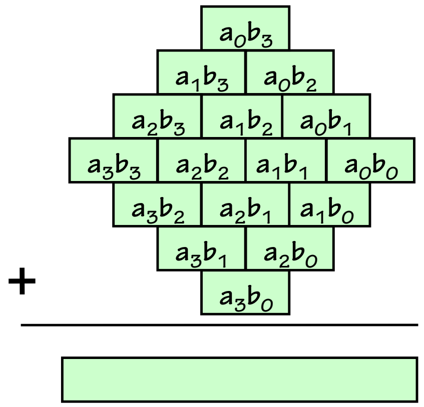
"almost always"

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

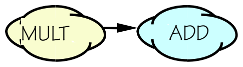
Partial Products:	n^2	=	$\Theta(n^2)$
Things to Add:	2n-2	=	$\Theta(n)$
Adder Width:	2n	=	$\Theta(n)$
Hardware Cost:	?	=	$\Theta(n^2)$

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

Observations:



a_0b_3
a_1b_3 a_0b_2
a_2b_3 a_1b_2 a_0b_1
a_3b_3 a_2b_2 a_1b_1 a_0b_0
a_3b_2 a_2b_1 a_1b_0
a_3b_1 a_2b_0
a_3b_0
+

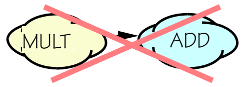


$\Theta(n^2)$ partial products.
$\Theta(n^2)$ full adders.
Hmmm.

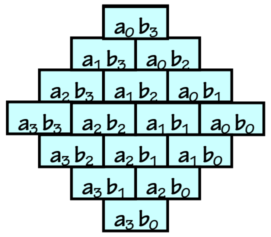
Repackaging Function

Engineering Principle #2:

Put the Solution where the Problem is.



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

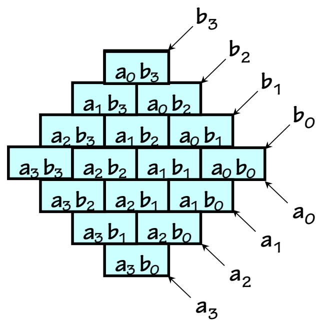


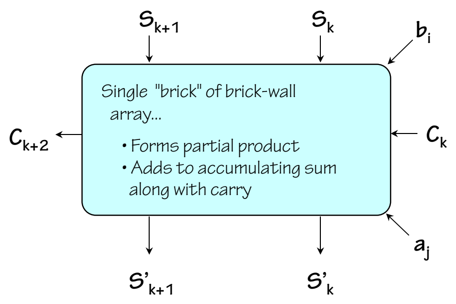
a_0b_3
a_1b_3 a_0b_2
a_2b_3 a_1b_2 a_0b_1
a_3b_3 a_2b_2 a_1b_1 a_0b_0
a_3b_2 a_2b_1 a_1b_0
a_3b_1 a_2b_0
a_3b_0

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

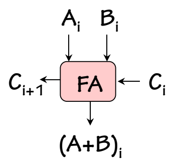
Goal:

Array of Identical Multiplier Cells





Single "brick" of brick-wall array...
- Forms partial product
- Adds to accumulating sum along with carry

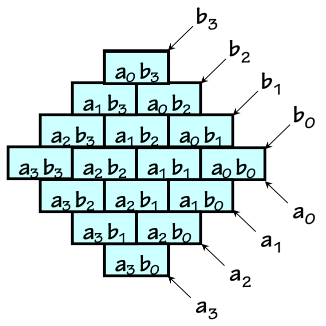


Necessary Component: Full Adder

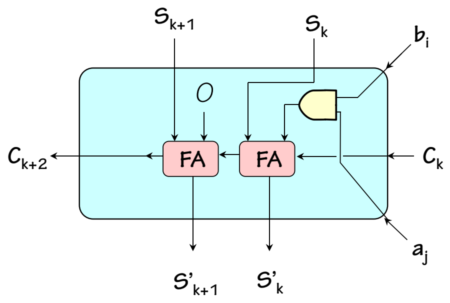
Takes 2 addend bits plus carry bit. Produces sum and carry output bits.

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*

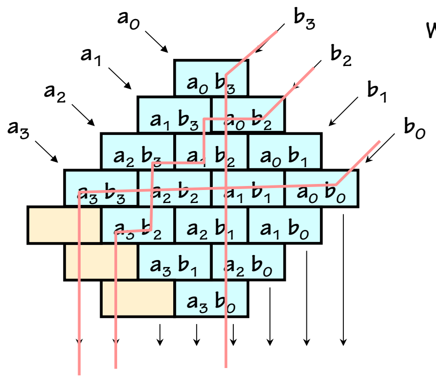


Brick design:
- AND gate forms 1x1 product
- 2-bit sum propagates from top to bottom
- Carry propagates to left

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

Latency revisited

Here's our combinational multiplier:



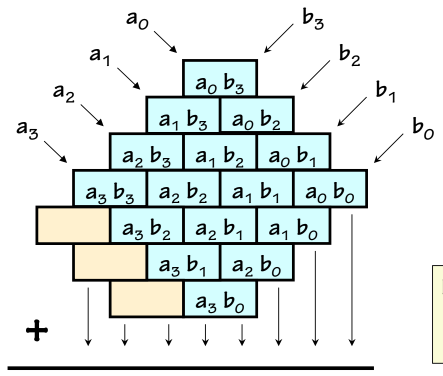
What's its propagation delay?

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: $O(n+n) = O(n)$!

Multiplier Cookbook: Chapter 2

Combinational Multiplier:



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

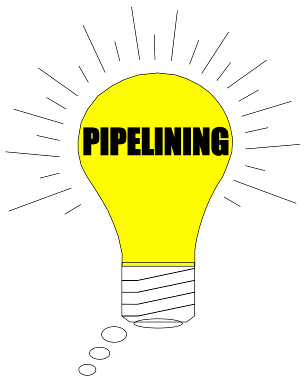
Note: lots of tricks are available to make a faster combinational multiplier...

Combinational Multiplier:

best bang for the buck?

Suppose we have LOTS of multiplications.

Can we do better from a cost/performance standpoint?

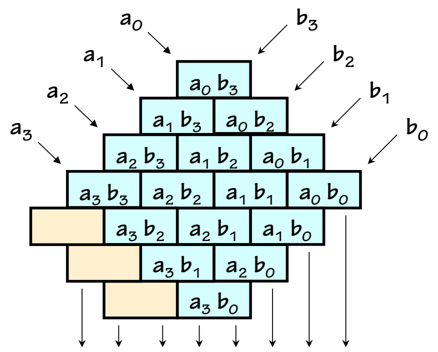


The Pipelining Bandwagon...

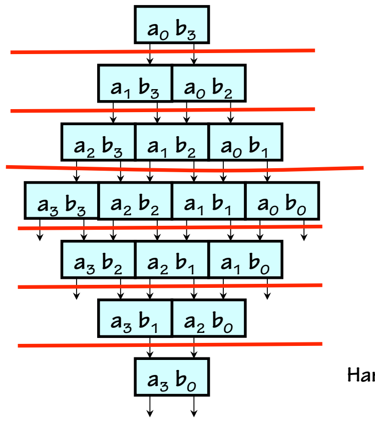
where do I get on?

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

What do we do? Where do we define stages?



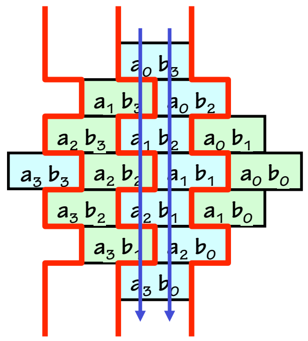
Stupid Pipeline Tricks


$a_0 b_3$
$a_1 b_3$ $a_0 b_2$
$a_2 b_3$ $a_1 b_2$ $a_0 b_1$
$a_3 b_3$ $a_2 b_2$ $a_1 b_1$ $a_0 b_0$
$a_3 b_2$ $a_2 b_1$ $a_1 b_0$
$a_3 b_1$ $a_2 b_0$
$a_3 b_0$

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


$a_0 b_3$
$a_1 b_3$ $a_0 b_2$
$a_2 b_3$ $a_1 b_2$ $a_0 b_1$
$a_3 b_3$ $a_2 b_2$ $a_1 b_1$ $a_0 b_0$
$a_3 b_2$ $a_2 b_1$ $a_1 b_0$
$a_3 b_1$ $a_2 b_0$
$a_3 b_0$

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!*

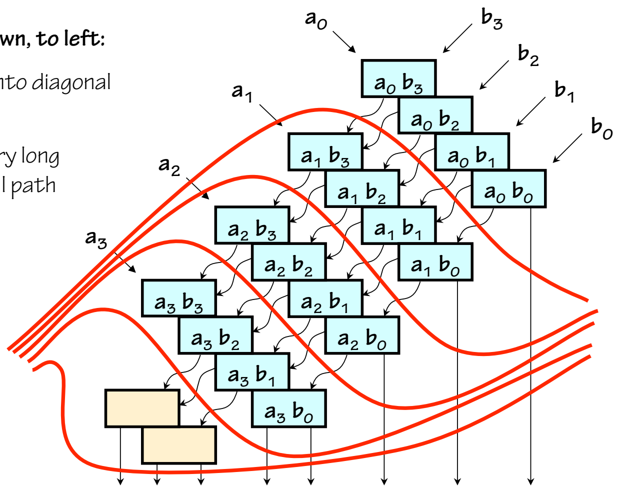
Back to basics:

what's the point of pipelining, anyhow?

Breaking O(n) combinational paths

LONG PATHS go down, to left:

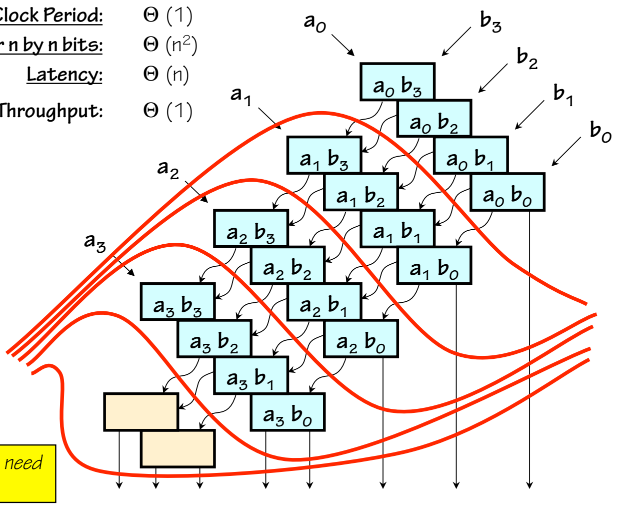
- Break array into diagonal slices
- Segment every long combinational path



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

Multiplier Cookbook: Chapter 3

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



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

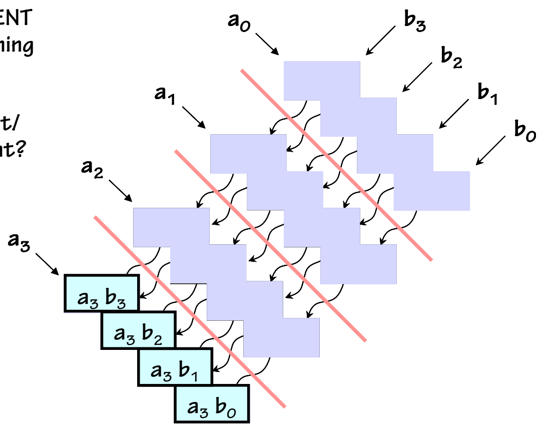
… but suppose we don't need the throughput?

Moving down the cost curve...

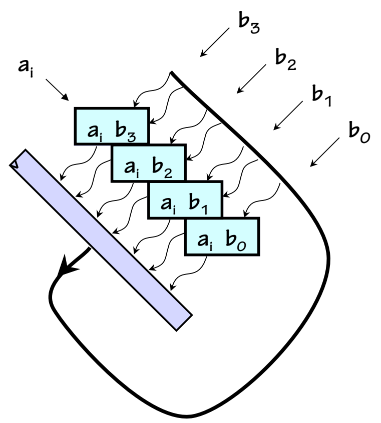
Suppose we have INFREQUENT multiplications... pipelining doesn't help us.

Can we do better from a cost/ performance standpoint?

Hmmm, do I really need all these extras?



Multiplier Cookbook: Chapter 4



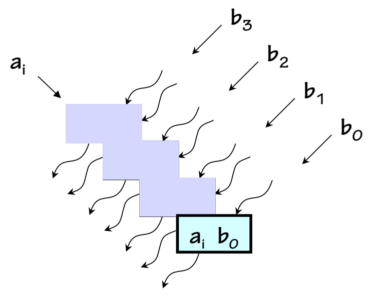
Sequential Multiplier:

- 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?



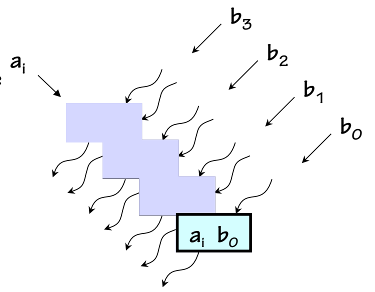
Suppose we want to minimize hardware (at any cost)...

- 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, ...