ECE4740: Digital VLSI Design

Lecture 24: Carry save adder & multipliers

A common problem in Homework 3

Switching activity and power

Recap: dynamic power consumption

- Dynamic power consumption dominated by charging/discharging capacitors

clock frequency

switching activity (per clock cycle)

$$P_{total} = V_{DD}^2 f_{clk} \sum_{k}^{K} \frac{\alpha_k}{2} C_k$$

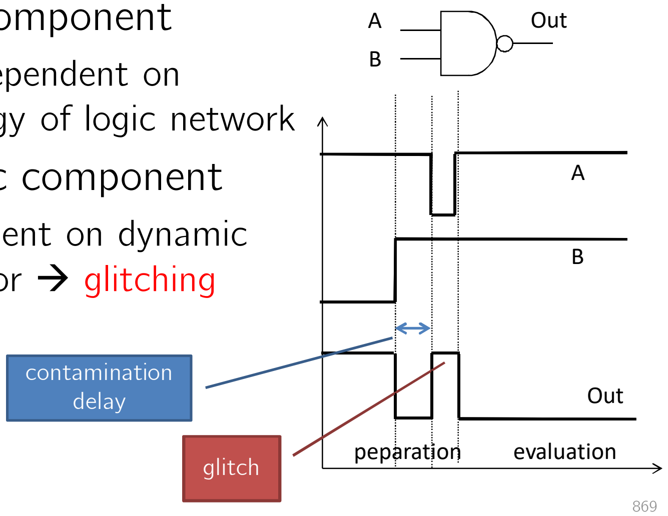
node/gate index

load capacitance for node k

- Switching activity α_k depends on logic

Dynamic power consumption

- Static component
  - only dependent on topology of logic network
- Dynamic component
  - dependent on dynamic behavior → glitching



Probabilistic model

- Depends on logic function
- Assume
  - N-input logic gate (2^N outputs)
  - inputs are i.i.d. uniformly distributed
- Transition probability is

$$\alpha = 2\frac{N_0}{2^N}\frac{N_1}{2^N}$$

#of zero entries in truth table

#of one entries in truth table

A	B	not(A+B)
0	0	1
0	1	0
1	0	0
1	1	0

Probabilistic model (cont'd)

A	B	not(A+B)
0	0	1
0	1	0
1	0	0
1	1	0

NOR2 gate with iid inputs

$$\alpha = 2\frac{N_0}{2^N}\frac{N_1}{2^N} = 0.375$$

A	B	XOR(A,B)
0	0	0
0	1	1
1	0	1
1	1	0

XOR gate with iid inputs

$$\alpha = 2\frac{N_0}{2^N}\frac{N_1}{2^N} = 0.5$$

- Only clock achieves >50% toggle rate

Non-uniform input statistics

- p_a = probability that input A is 1
- p_b = probability that input B is 1

	switching activity
AND	$2*(1-p_a*p_b)p_a*p_b$
OR	$2*(1-p_a)*(1-p_b)*[1-(1-p_a)*(1-p_b)]$
XOR	$2*[1-(p_a+p_b-2*p_a*p_b)]*(p_a+p_b-2*p_a*p_b)$

- Complicated & not very useful in practice

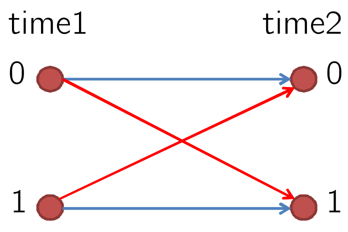
No need for a general formula!

- You only need probabilities p_a, p_b, and p_c of output being 1
  - Assume they are independent across inputs and time
- Compute probability of f(A,B,C)=1
  - $P1=(1-p_a)*p_b*(1-p_c)+p_a*p_b*p_c$
  - f(A,B,C)=0 → P0=1-P1
- Compute transition probability

A	B	C	f(A,B,C)
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

Switching probabilities?

- Consider two time steps



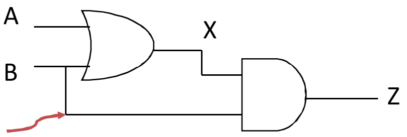
only correct if data is independent over time

- How likely is a transition going to happen?
  - $\Pr[0 \rightarrow 1] = P0*P1$
  - $\Pr[1 \rightarrow 0] = P1*P0$

$$\alpha = 2*P1*P0$$

Inter-signal correlations

- Switching activity difficult to estimate as there is correlation across space and time

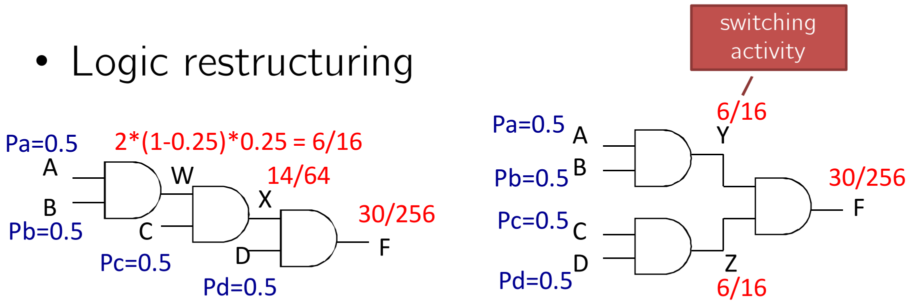


reconvergent fan-out

- One has to use conditional probabilities
- Timing does matter → glitches

Power reduction via restructuring

- Logic restructuring



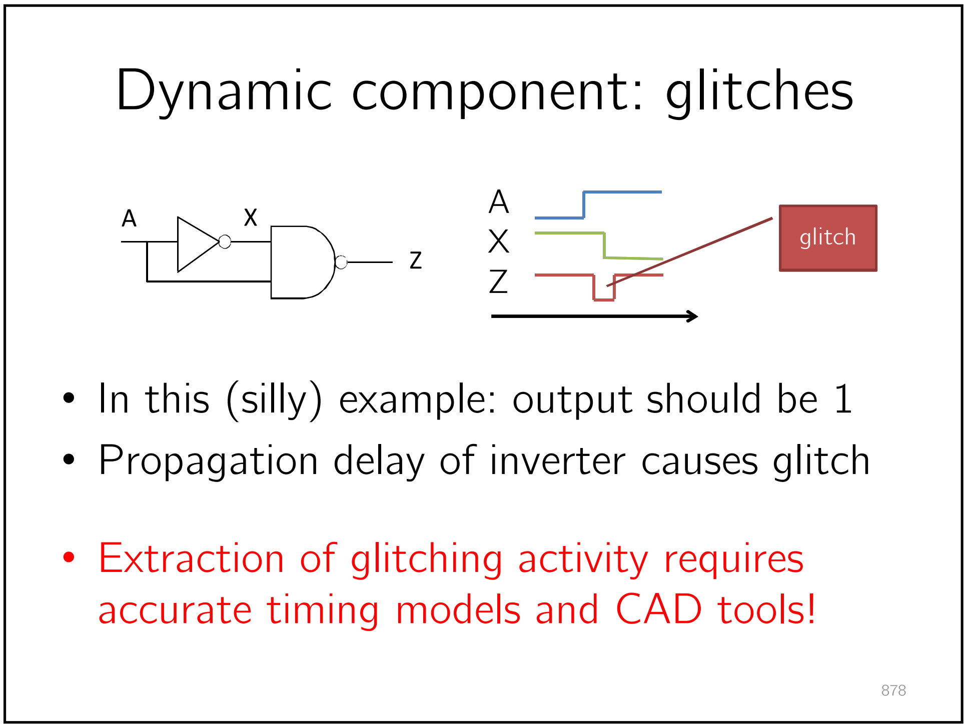
- Chain implementation: lower total switching activity than tree (for i.i.d. uniform inputs)
- Insert signals with higher transition rate at the end of the chain

Simulative static activity analysis

- Generate a series of representative inputs
- Simulate and record the logic activity
- Use these activities along with the node capacitances to estimate the power

$$P_{stat} = V_{DD}^2 f_{clk} \sum_k^K \frac{\alpha_k}{2} C_k$$

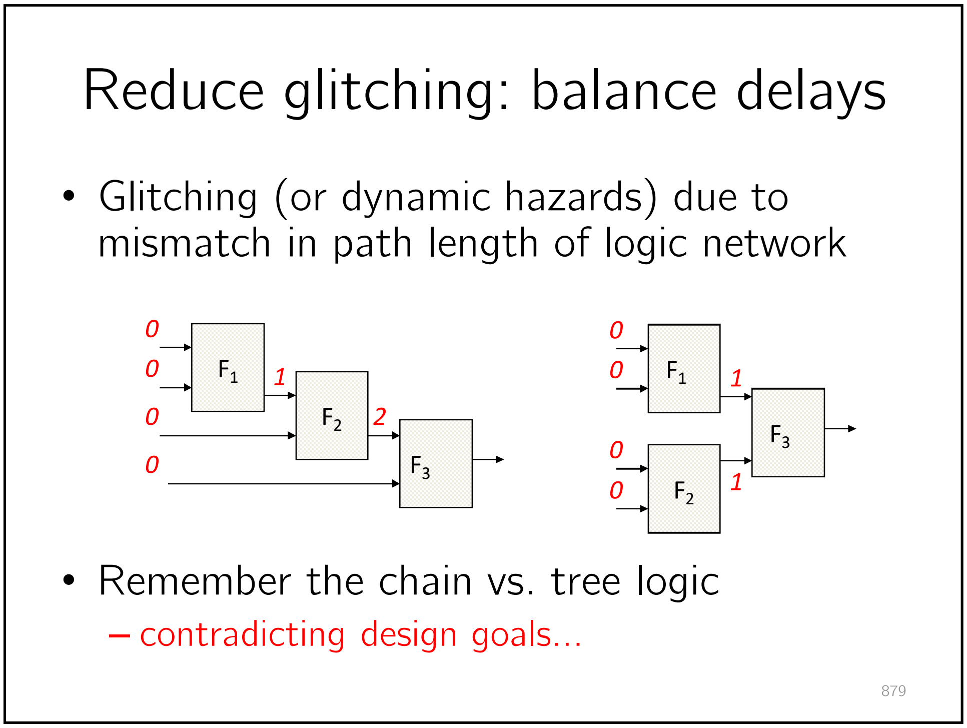
- There are tools that help you! YAY!



Dynamic component: glitches

- In this (silly) example: output should be 1
- Propagation delay of inverter causes glitch

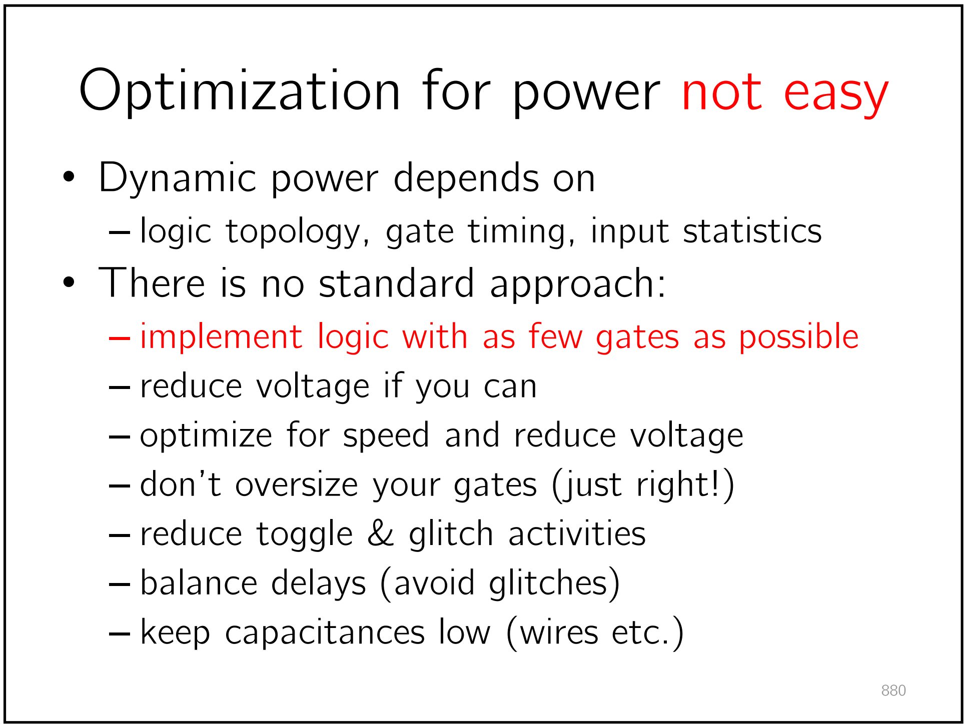
- Extraction of glitching activity requires accurate timing models and CAD tools!



Reduce glitching: balance delays

- Glitching (or dynamic hazards) due to mismatch in path length of logic network

- Remember the chain vs. tree logic
  - contradicting design goals…



Optimization for power not easy

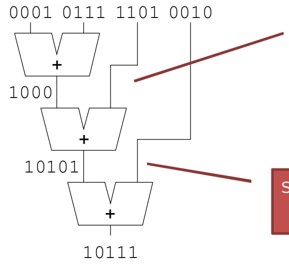
- Dynamic power depends on
  - logic topology, gate timing, input statistics
- There is no standard approach:
  - implement logic with as few gates as possible
  - reduce voltage if you can
  - optimize for speed and reduce voltage
  - don’t oversize your gates (just right!)
  - reduce toggle & glitch activities
  - balance delays (avoid glitches)
  - keep capacitances low (wires etc.)

Faster multi-operand additions

Carry save adders

Multi-operand addition

- Add four N-bit numbers: Sum = A+B+C+D
- Straightforward solution: Use 3 N-bit carry propagate adders → large and slow



tree structure can reduce critical path and reduce glitching activity

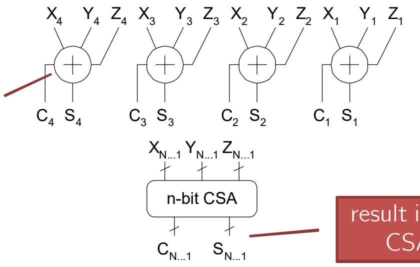
suffers from glitches!!! power inefficient

Image taken from: CMOS VLSI Design: A Circuits and Systems Perspective by Weste, Harris

Better: carry save adder (CSA)

- Remember: Full adder sums 3 inputs and produces 2 outputs (3:2 compressor)
  - Essentially adding three 1-bit numbers
- N full adders in parallel → carry save adder

constant delay: don't need to wait for carry to propagate through



$S_4S_3S_2S_1$

$C_4C_3C_2C_1 0$

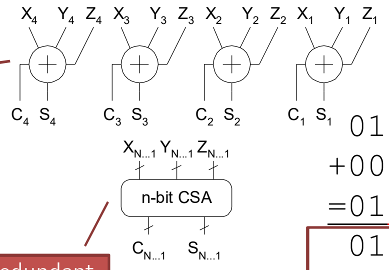
result in redundant CSA format

Image taken from: CMOS VLSI Design: A Circuits and Systems Perspective by Weste, Harris

Carry save adder (CSA)

- Main idea: Don’t propagate carry signal until last possible stage

fast and small


X_4 Y_4 Z_4 X_3 Y_3 Z_3 X_2 Y_2 Z_2 X_1 Y_1 Z_1
C_4 S_4 C_3 S_3 C_2 S_2 C_1 S_1
$X_{N...1}$ $Y_{N...1}$ $Z_{N...1}$
n-bit CSA
$C_{N...1}$ $S_{N...1}$

result in redundant CSA format

0110
+0011
=0121

0101= $S_4S_3S_2S_1$
00100=$C_4C_3C_2C_1$0
=01001

Redundant CSA format

- Assume we compute 0110 + 0011
- Carry propagate adders compute:

  0110
  +0011
  =1001

  carry propagated

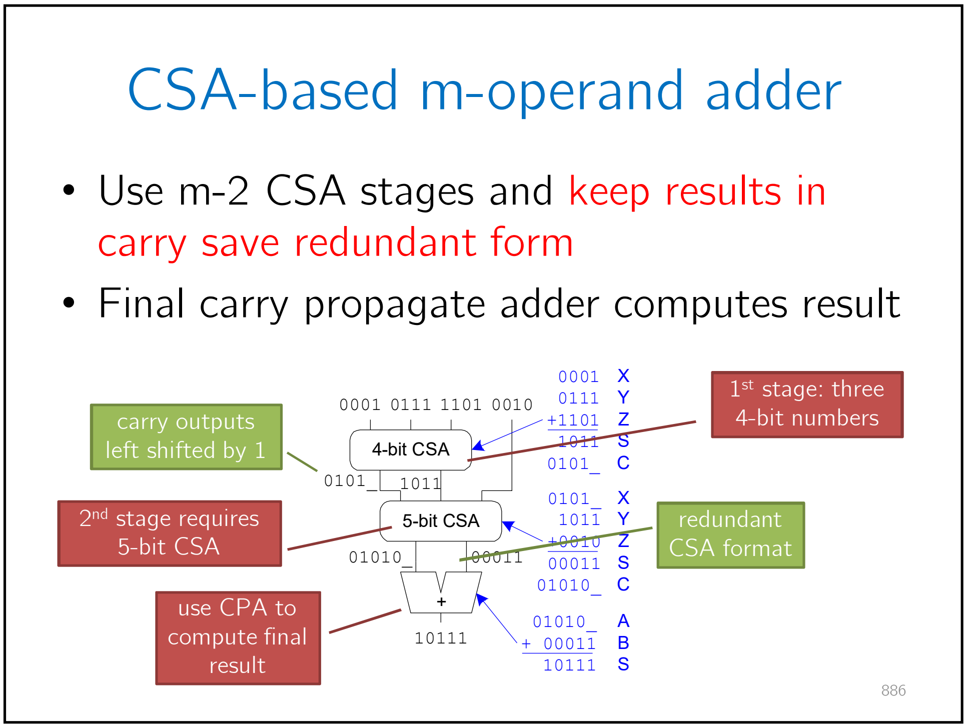
- Carry save adders compute:

  0110
  +0011
  =0121

  think non-binary, redundant number format

$S_4S_3S_2S_1$ = 0101
$C_4C_3C_2C_1$0 = 00100

adding these numbers → final result



CSA-based m-operand adder

- Use m-2 CSA stages and keep results in carry save redundant form
- Final carry propagate adder computes result

carry outputs left shifted by 1

2nd stage requires 5-bit CSA

use CPA to compute final result

0001 0111 1101 0010

4-bit CSA

0101_ 1011

5-bit CSA

01010_ 00011

+

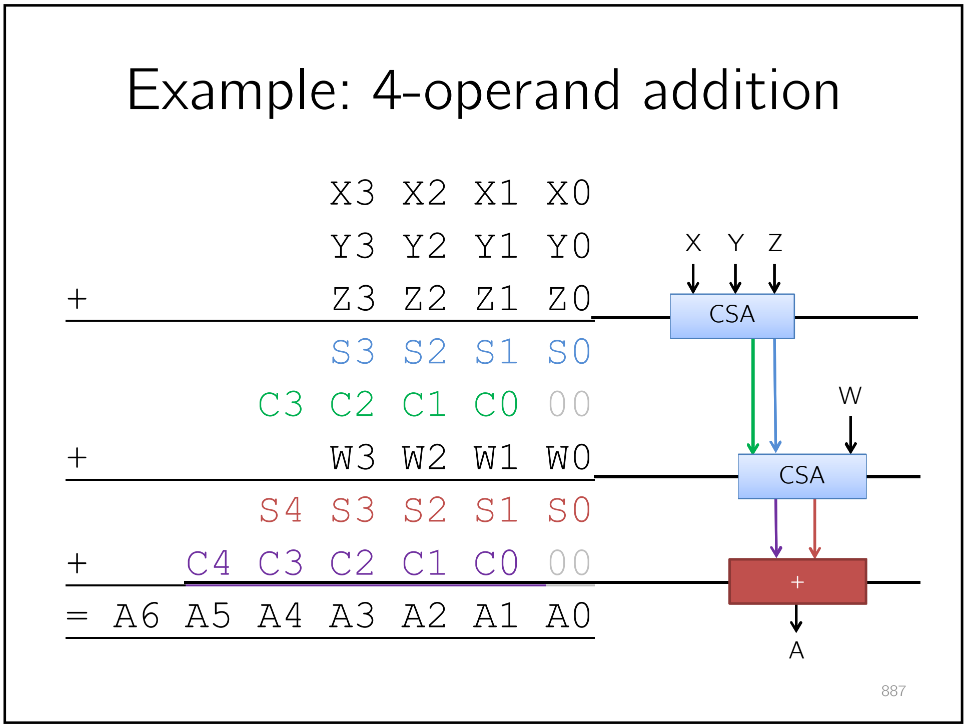
10111

```
 0001   X
 0111   Y
+1101   Z
 1011   S
 0101_  C

 0101_  X
 1011   Y
+0010   Z
00011   S
01010_  C

  01010_  A
+ 00011   B
  10111   S
```

1st stage: three 4-bit numbers

redundant CSA format

886



Example: 4-operand addition

```
           X3 X2 X1 X0
           Y3 Y2 Y1 Y0
+          Z3 Z2 Z1 Z0
           S3 S2 S1 S0
        C3 C2 C1 C0 00
+          W3 W2 W1 W0
        S4 S3 S2 S1 S0
+    C4 C3 C2 C1 C0 00
= A6 A5 A4 A3 A2 A1 A0
```

X Y Z

CSA

W

CSA

+

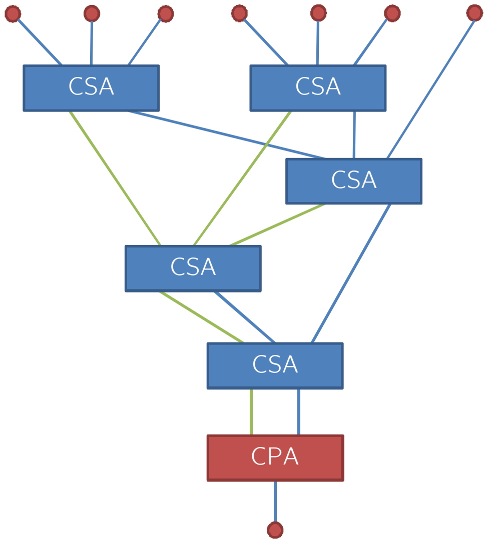
A

887

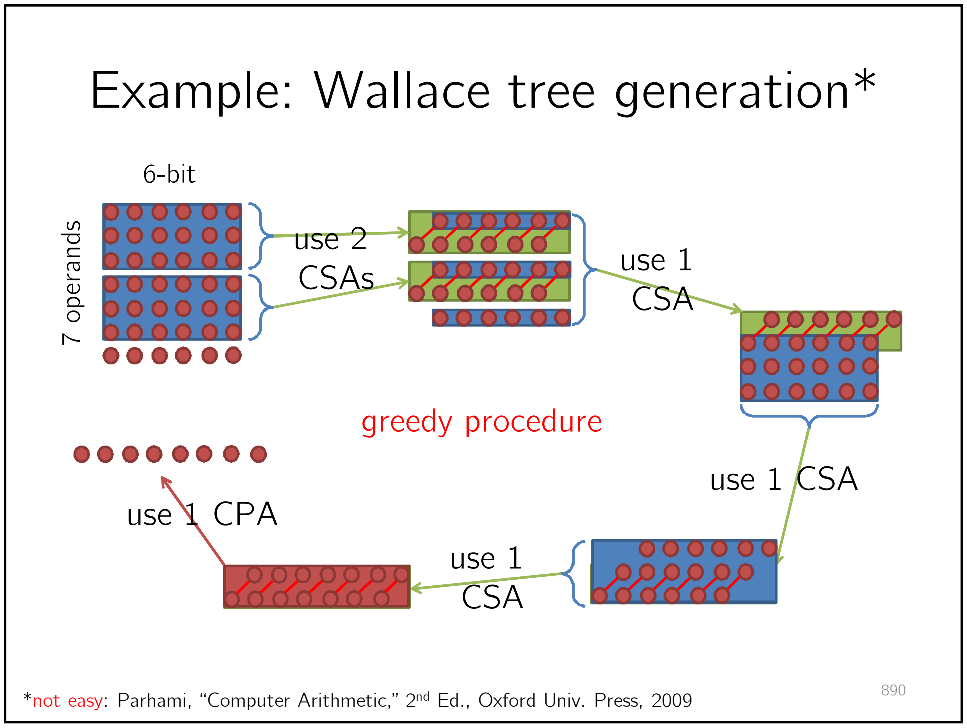
Area and delay

- Area & delay of CSA-based m-operand adders
  - $A=(m-2)A_{CSA}+A_{CPA}$
  - $T=(m-2)T_{CSA}+T_{CPA}$
- Since some inputs are 0, FA can be replaced by HA circuits (reduces area)
- There are m-operand adders that are faster → use CPA tree (Wallace or Dadda trees)
  - $A=O(m*N+N\log(N))$
  - $T=O(\log(m)+\log(N))$

Wallace tree: 7-operand example



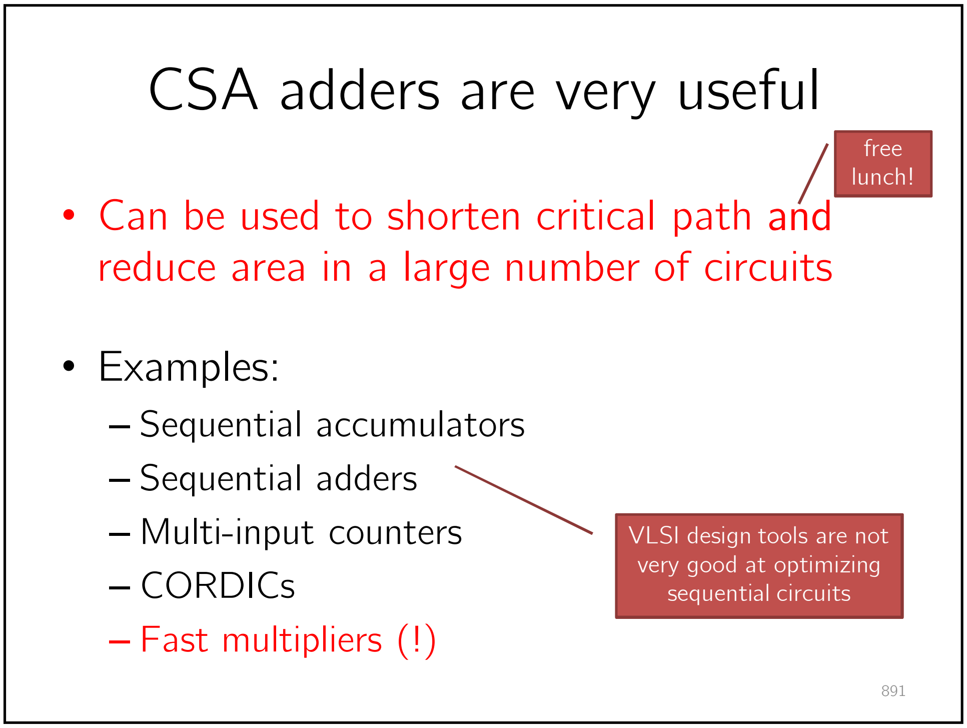
- Only 4 CSA blocks in series (compared to 7-2=5 for regular array)
- $T=O(\log(m)+\log(N))$
- $A=O(mN+N\log N)$



Example: Wallace tree generation*

*not easy: Parhami, “Computer Arithmetic,” 2^{nd} Ed., Oxford Univ. Press, 2009

890



CSA adders are very useful

- Can be used to shorten critical path and reduce area in a large number of circuits

free lunch!

- Examples:
  - Sequential accumulators
  - Sequential adders
  - Multi-input counters
  - CORDICs
  - Fast multipliers (!)

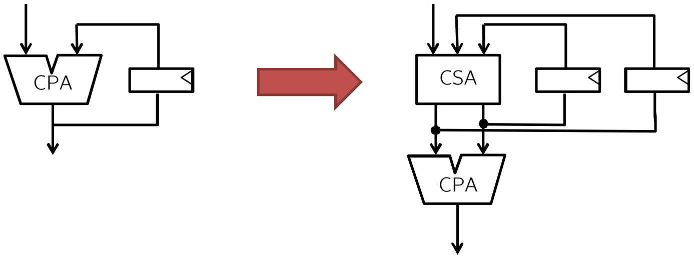
VLSI design tools are not very good at optimizing sequential circuits

891

CSA adder summary and an example

- Can shorten critical path and reduce area
- CSA optimizations often done manually (exception: multipliers or m-operand adders)

Any idea how?



Another key building block

Multiplier circuits

Why important?

- Multipliers are a key arithmetic block in a large number of applications:
  - FIR/IIR filters
  - Matrix-vector products
  - Computer graphics (games!)
  - Fast Fourier transforms (FFTs)
  - Scientific computing
  - Etc…

and they are rather slow, large, and energy inefficient…

Binary-valued multiplication

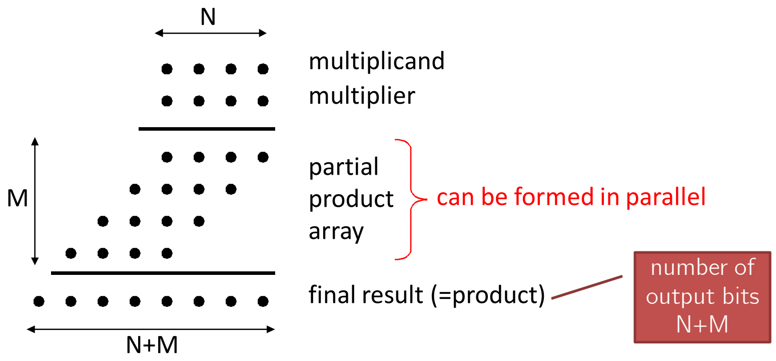
$$A = \sum_{i=0}^{M-1} a_i 2^i \qquad B = \sum_{j=0}^{N-1} b_j 2^j$$

- Coefficients a_i and b_i are in {0,1}

$$Z = A \cdot B = \sum_{i=0}^{M-1} \sum_{j=0}^{N-1} a_i b_j 2^{i+j}$$

essentially a big addition

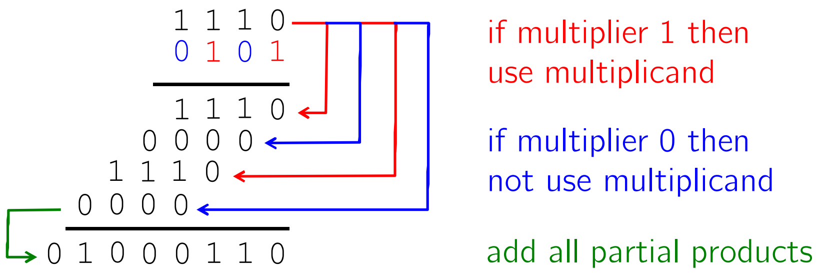
Multiplication as repeated additions



- Final result (product) is obtained through multi-operand addition

896

Example



```
         1 1 1 0
         0 1 0 1
         -------
         1 1 1 0
       0 0 0 0
     1 1 1 0
   0 0 0 0
 ---------------
 0 1 0 0 0 1 1 0
```

if multiplier 1 then use multiplicand
if multiplier 0 then not use multiplicand
add all partial products

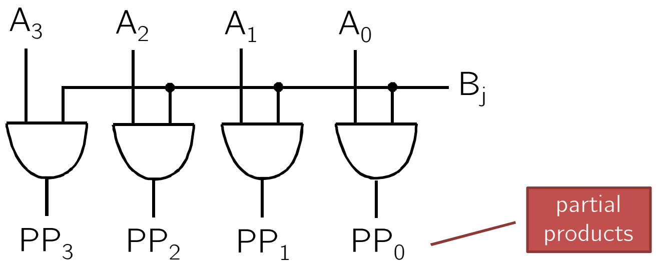
- Produce M partial products of N-bit
- Sum these to produce M+N bit product

897

Shift and add multiplication

- Right shift and add
  - Partial products array rows are accumulated from top to bottom on an N-bit adder
  - After each addition, right shift the accumulated partial product to align with next row to add
  - $T=O(N*T_{add})$ which is $O(N^2)$ for an RCA
- How to make it faster:
  - Use faster adder
  - Reduce # of partial products → Booth's recoding
  - Use carry save adders

Partial product generation



- In most cases, the partial product array has many zero rows that have no impact on result

Booth’s recoding method

- Goal: Reduce number of generated PPs
- Example:
  - Assume multiplier is 01111110
  - Generates 6 non-zero PPs
  - Recode multiplier to 100000$\underline{1}$0
  - Recoded number has only 2 non-zero PPs
- Booth’s recoding method reduces the number of non-zero PPs by half
  - Lower area and faster → but complicated ☹

$\underline{1}$ indicates -1

The array multiplier

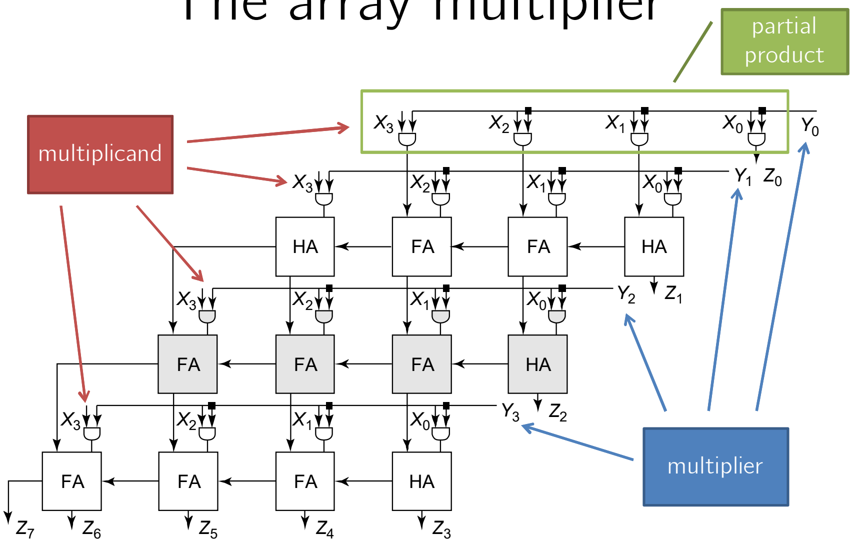


Image taken from: Digital Integrated Circuits (2nd Edition) by Rabaey, Chandrakasan, Nikolic