EECS 151/251A Discussion 9

04/20/2018

Announcements

- That extra discussion with Taehwan will be in one week
  - Location/time TBA, slides will be available if you can’t make it.
- Homework 11 out by Sunday

Agenda

- By request:
  - Booth’s recoding
  - Multipliers
  - LFSRs
  - DRAM design, latches
- Your questions

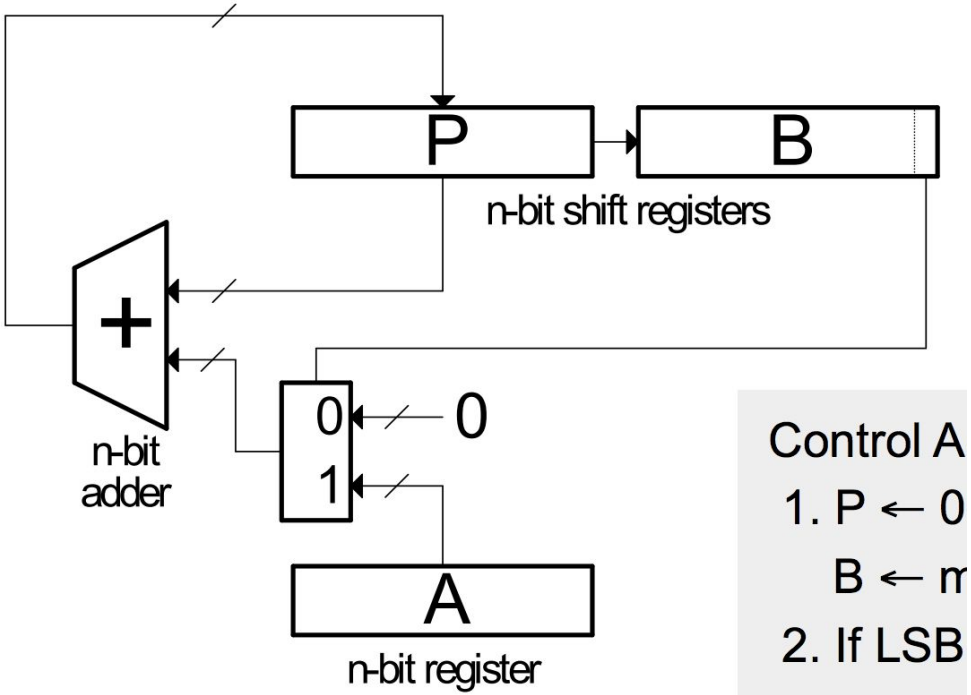
Multiplication Basics

011 x 101 = 1111

```
  011
  101
-----
  011
 000
011
-----
01111
```

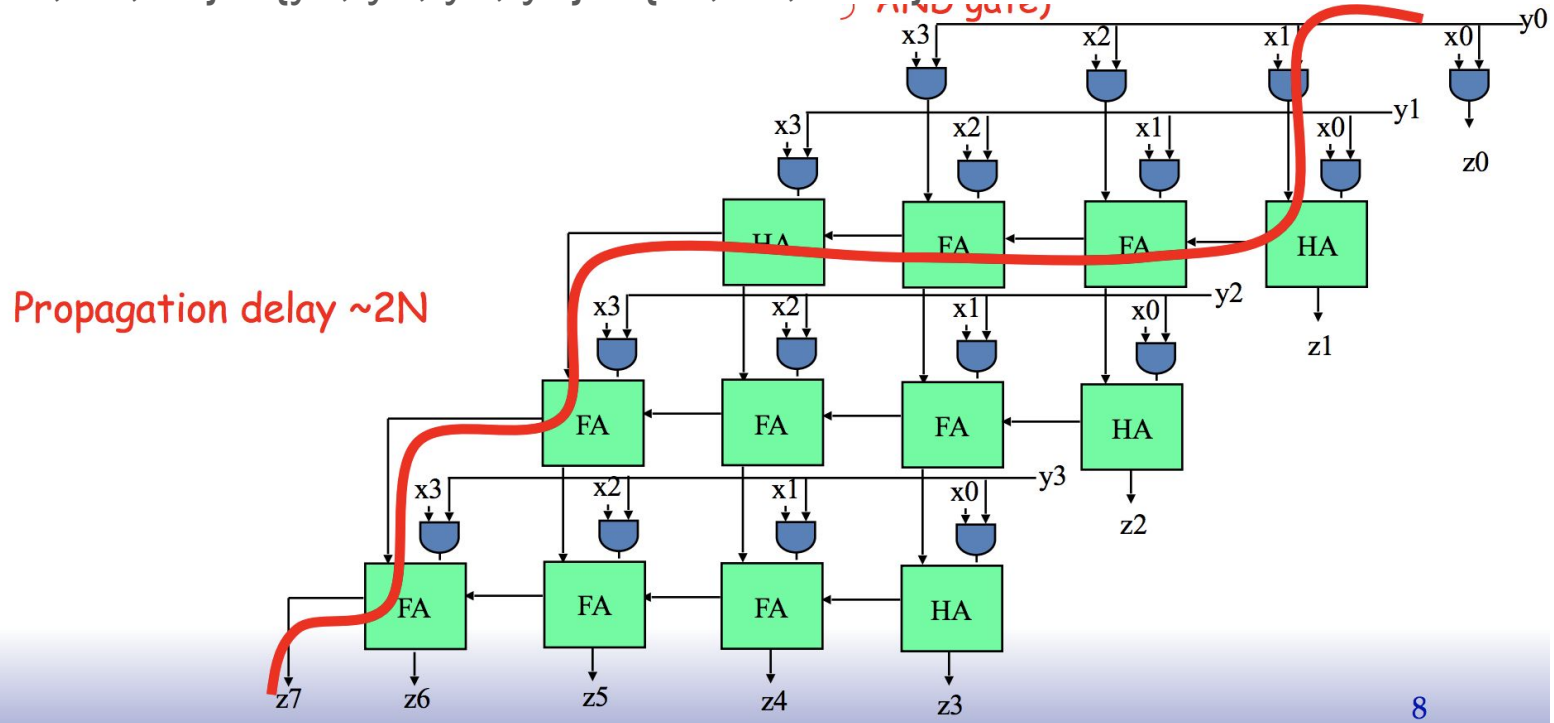
Shift-and-add multiplier

- How to deal with signed values?



Combinational (Unsigned)

{x3, x2, x1, x0} x {y3, y2, y1, y0} = {z7, z6, z5…}



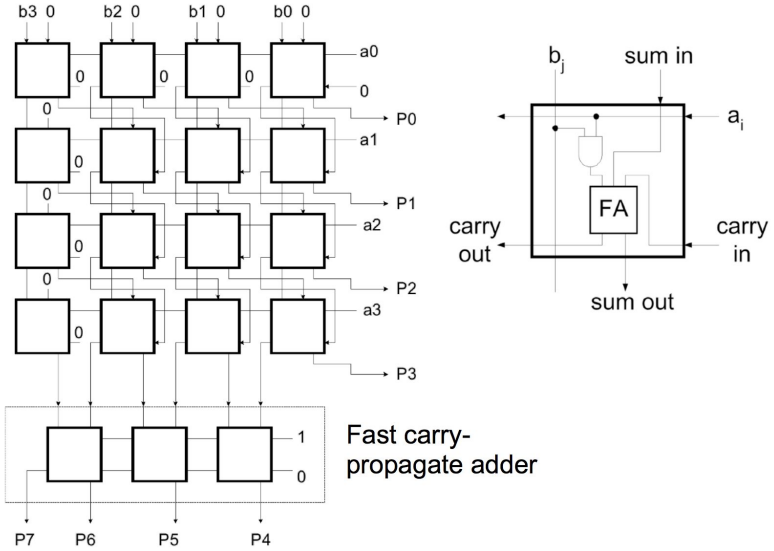
Carry-Save Adders

- Keep results for “carry” and “sum” separate until you have to
- Don’t having to add carry + sum too early: save time!
- Only useful when adding sets of numbers. Need “real” adder on final sum

- Note: An adder can be called a “3:2 *compressor*”; it turns 3 input signals into 2 output in a potentially lossy operation

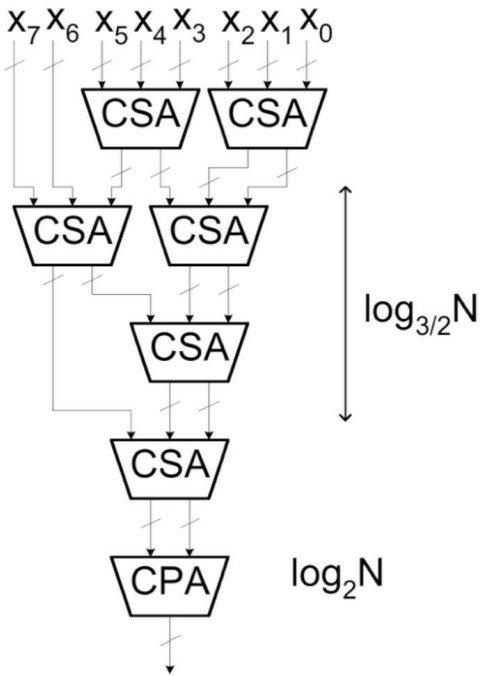
Multiplication w/ Carry-Save Adders

- Structure looks like a combinational multiplication circuit
- But intermediate steps are faster: don’t add carries to sums yet



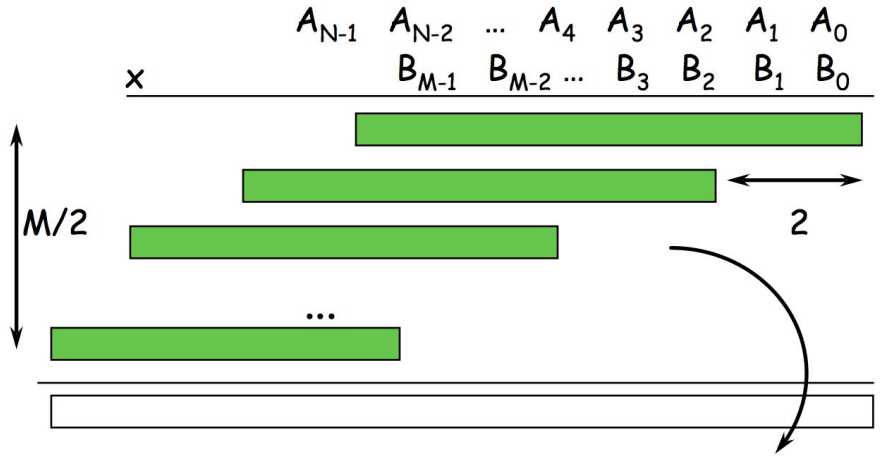
Wallace Tree

- CSA is associative + commutative
- Doesn’t matter where you add the numbers, as long as you eventually add the numbers
- This yields the structure of a “Wallace Tree”:
  - Group intermediate sum/carry signals at each stage to maximise the number of parallel adds



Booth recoding

- Add pairs at a time
- Defer part computation to next pair
- Look at previous pair to determine deferred op



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

$B_{K+1,K}$*A = 0*A → 0
= 1*A → A
= 2*A → 4A - 2A
= 3*A → 4A - A

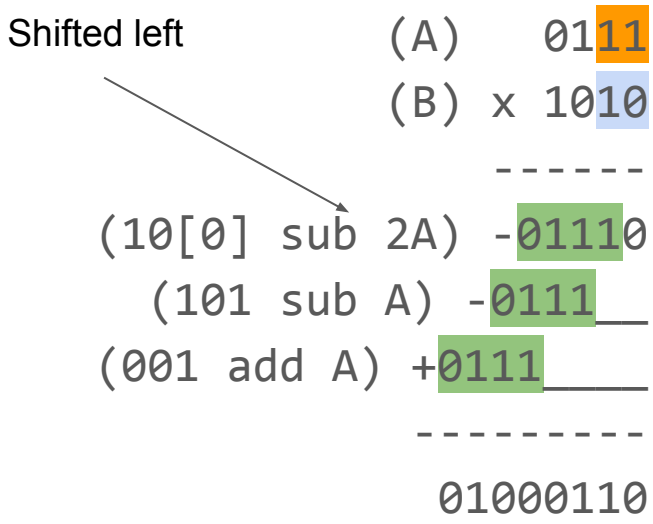
current bit pair — from previous bit pair

B_{K+1}	B_K	B_{K-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	← -2*A+A
1	1	0	sub A	
1	1	1	add 0	← -A+A

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 recoding

```
0111 x 1010
```

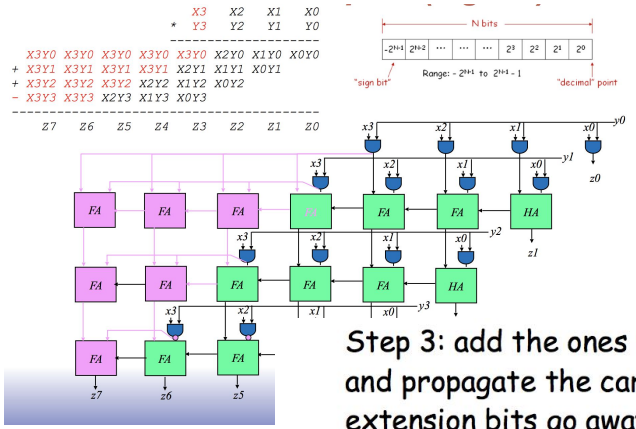


- Imagine you've separated your responsibilities per pair
- And recombined them later, more efficiently
- Operations seems unrelated because they combine separate smaller ops from previous steps

Example?

`1100 x 0101`

Combinational Multiplication (Signed)



Step 2: don't want all those extra additions, so add a carefully chosen constant, remembering to subtract it at the end. Convert subtraction into add of (complement + 1).

```
   X3Y0 X3Y0 X3Y0 X3Y0 X3Y0 X2Y0 X1Y0 X0Y0
+                    1
+ X3Y1 X3Y1 X3Y1 X3Y1 X2Y1 X1Y1 X0Y1
+                 1
+ X3Y2 X3Y2 X
+
3 X3Y3 X
   1
   1
```

Step 3: add the ones to the partial products and propagate the carries. All the sign extension bits go away!

```
                    X3Y0 X2Y0 X1Y0 X0Y0
+              X3Y1 X2Y1 X1Y1 X0Y1
+         X2Y2 X1Y2 X0Y2
+    X3Y3 X2Y3 X1Y3 X0Y3
+                        1
+
-       1    1    1    1
```

```
                         X3Y0 X2Y0 X1Y0 X0Y0
+              X3Y1 X2Y1 X1Y1 X0Y1
+         X2Y2 X1Y2 X0Y2
+    X3Y3 X2Y3 X1Y3 X0Y3
+                        1
+    1                   1
```

Result: multiplying 2's complement operands takes just about same amount of hardware as multiplying unsigned operands!

Shifters

LFSRs

References

- Discussions from EECS151/251A Fall 2017, George Alexandrov
- Homeworks from EECS151/251A Fall 2016, Spring 2017, Fall 2017
- Wikipedia