EECS 427
Lecture 9: Shifters and Multipliers
Readings: 11.5, WH 10.8, 11.4

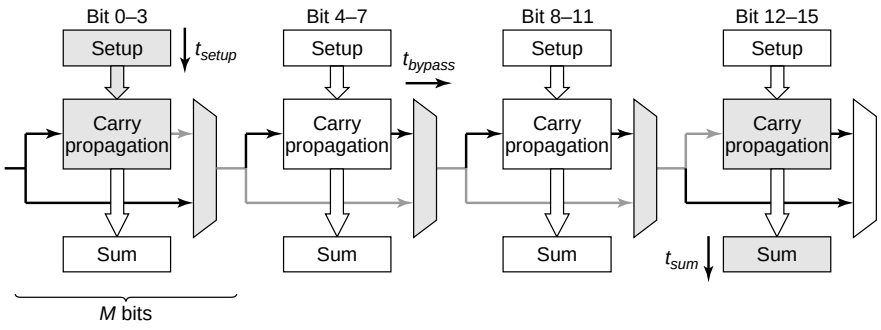
Reminders

- HW3 – project initial proposal: due tonight at 7 pm
  - Email your initial proposal (one per group) in doc format to zhengya@eecs.umich by 7 pm
  - Limit your write-up to 4 pages maximum. Keep it brief and clear.
  - Start executing your proposal in parallel with CAD assignment
  - HW4 (detailed proposal) due 11/16
- Quiz 1 on Wednesday 10/14
  - Samples (with solutions) from past terms are posted
  - Half-lecture review next Monday
  - Zhengya's office hours next week: M 3-4 pm, Tu 3:30-5:30 pm, no office hour on Wed.
- CAD4 is due Wednesday 10/14
  - You can submit it by Thursday 10/15 at noon

Last Time

- Full adder
  - Mirror adder – size it carefully
  - Transmission-gate adder – efficient XOR implementation
  - Manchester carry chain – long RC chain
- Adder topologies
  - Ripple carry – simple, small adders
  - Carry bypass – skip in groups
  - Carry select – pre-computation to shorten the critical path
  - Carry look-ahead – compute carry-in for higher order bits to shorten the critical path
- Carry look-ahead adder
  - Kogge-Stone – full tree, regular interconnect, consistent fanout, large area and high power

Carry-Bypass Adder



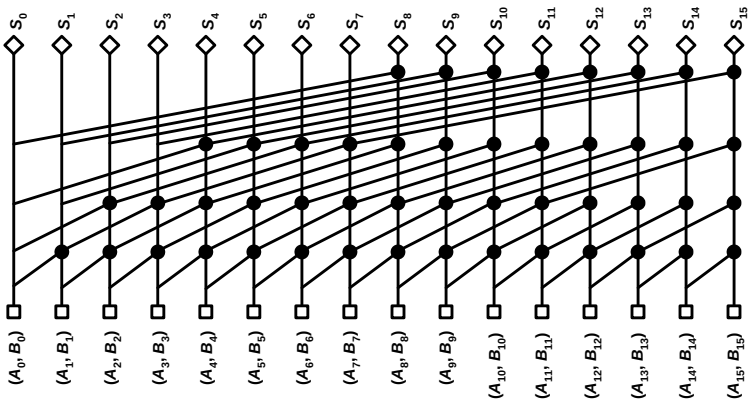
$$t_{adder} = t_{setup} + M_{tcarry} + (N/M\text{-}1)t_{bypass} + (M\text{-}1)t_{carry} + t_{sum}$$

Inner blocks do not contribute to worst-case delay since they have time to compute while bits 0-3 are propagating (assuming they have a generate or delete)

Overview

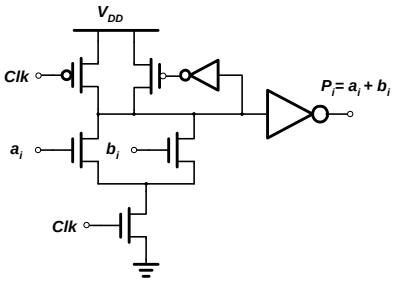
- Carry look-ahead adder
  - Kogge-Stone
  - Radix 2 or Radix 4
  - Sparse tree, Brent-Kung
- Shifter
  - Barrel shifter
  - Logarithmic shifter
- Multiplier
  - Array multiplier
  - Carry save multiplier
  - Modified Booth recoding
  - Tree multiplier

Tree Adder

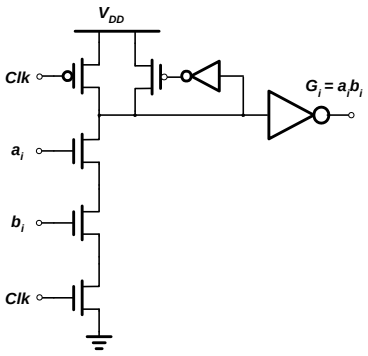


16-bit radix-2 Kogge-Stone tree

Domino Adder

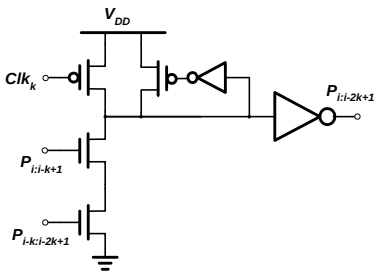


Propagate

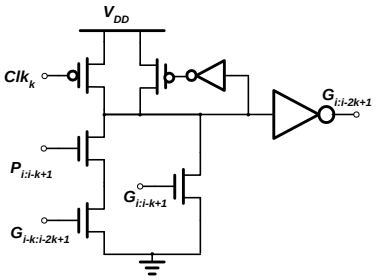


Generate

Domino Adder

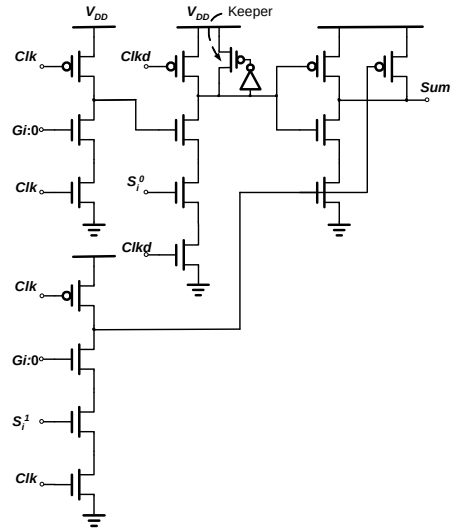


Propagate

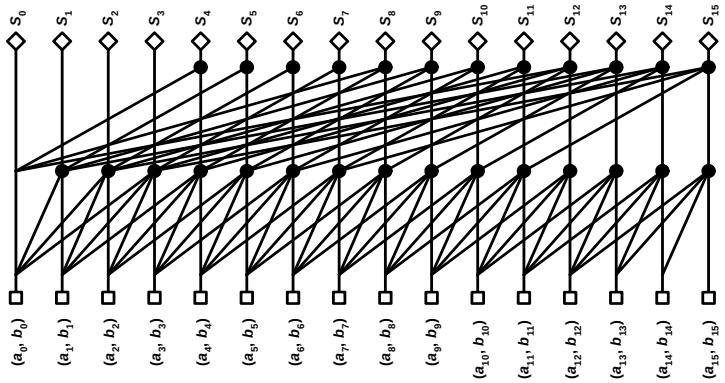


Generate

Domino Sum

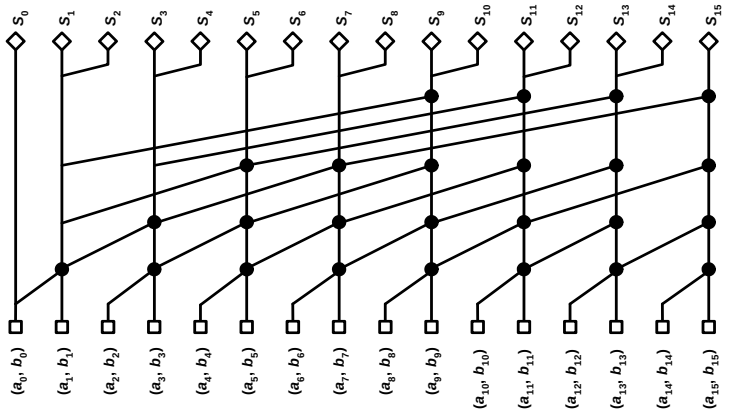


Tree Adder



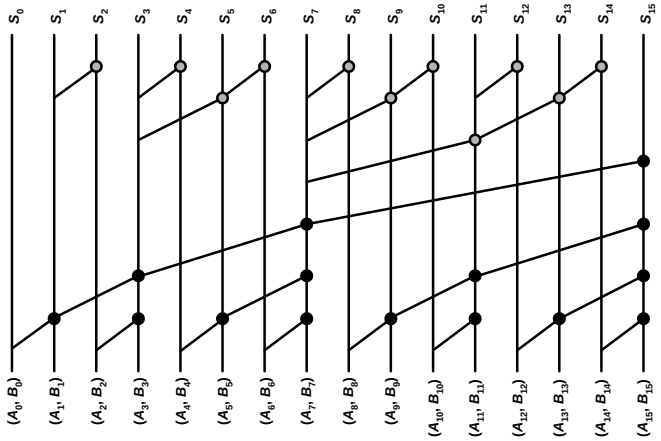
16-bit radix-4 Kogge-Stone Tree

Sparse Tree



16-bit radix-2 sparse tree with sparseness of 2

Tree Adder



Brent-Kung Tree

Radix 2 vs. Radix 4

- Fanout consideration
  - Easier to drive large fanouts with Radix-2
  - Radix-4 has fewer stages and could have speed advantage when driving low fanouts
- Wire length consideration
  - Radix-4 has longer wires (64 bits: crosses 48 bit slices vs. 32 in radix-2). Fewer logic stages precedes large wireload.
- Logic consideration
  - Radix-2 has lower stack heights

Full vs. Sparse Trees

- Not all the carries are calculated in sparse trees
  - Only every 2^{nd}, 4^{th}, etc.
  - Reduced (uneven) input capacitance
  - Fewer transistors and wires
  - Lower power
- Sparse tree adders need to recover missing carries
  - Ripple (extra gate delay)
  - Precompute (extra fanout)
  - Complex precomputation can get into the critical path

Shift Types

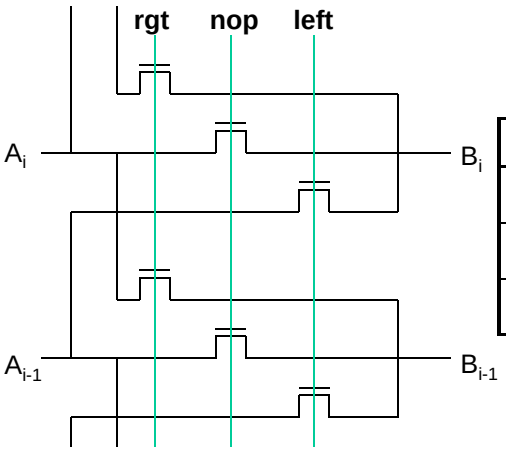
- Arithmetic vs. logical shift (Start with 1101)
  - Logical shift (baseline op for us)
    - Logical shift right by 1: 0110
    - Logical shift right by 2: 0011

    } Shift in 0s for logical right shifts
    - Logical shift left by 1: 1010
    - Logical shift left by 2: 0100

    } Shift in 0s for left shifts
  - Arithmetic shift (not baseline)
    - Arithmetic left shifts same as logical
    - Arithmetic right shift by 1: 1110
    - Arithmetic right shift by 2: 1111

    } Repeat sign of MSB for arithmetic right shifts

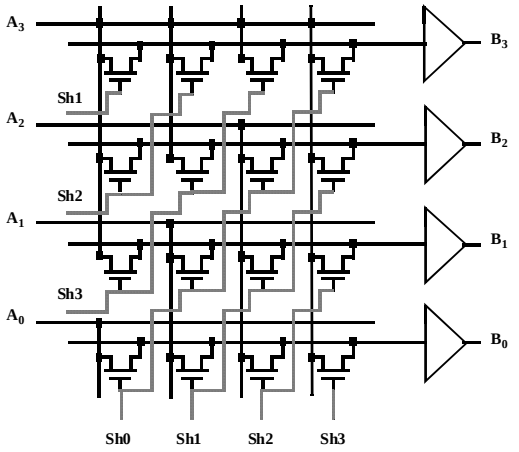
The Binary Shifter Concept



A_i	A_{i-1}	rgt	nop	left	B_i	B_{i-1}
A_1	A_0	0	1	0	A_1	A_0
A_1	A_0	1	0	0	0	A_1
A_1	A_0	0	0	1	A_0	0

Thanks to: Irwin/Narayanan

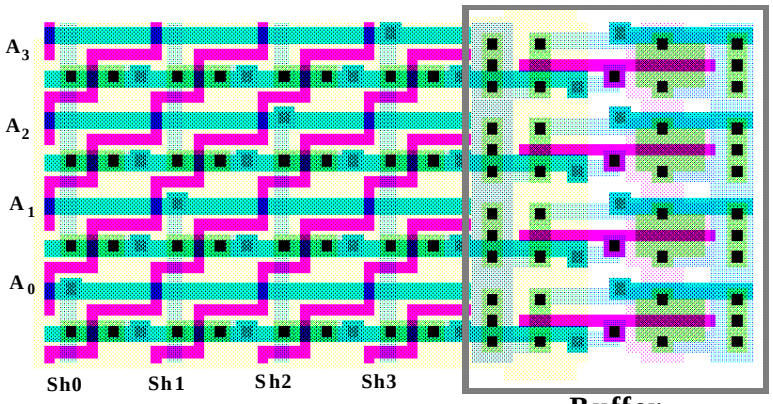
Barrel Shifter



Area Dominated by Wiring

Requires decoding of shift amount given in instruction word

4x4 Barrel Shifter

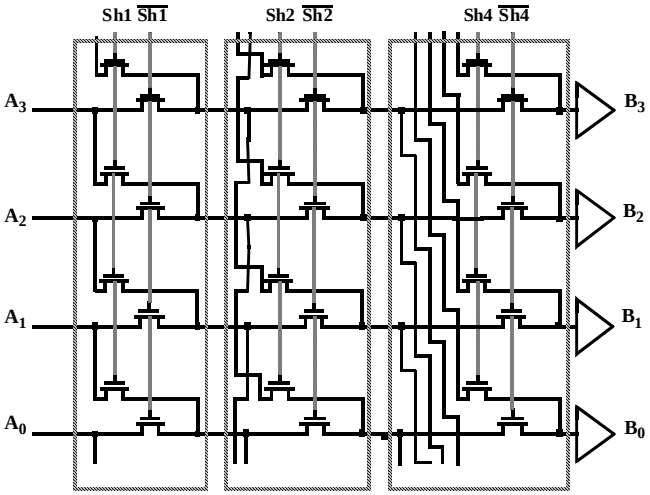


Only 1 shift bit high at any time

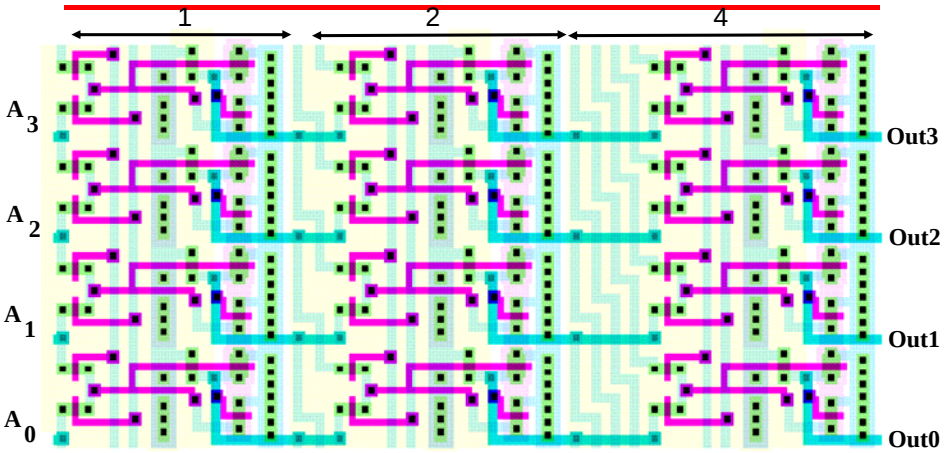
$\mathbf{Width_{barrel} \sim 2\, p_m\, N}$

N is max shift width (15 for us), p_m is metal pitch

Logarithmic Shifter



0-7 bit Logarithmic Shifter



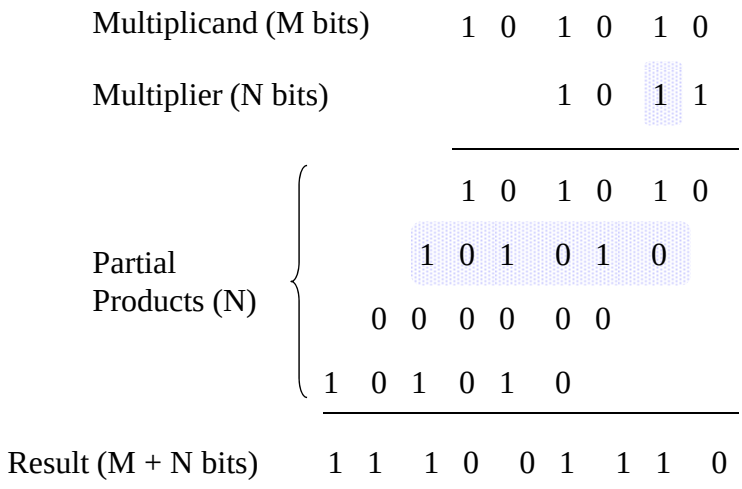
$$width_{\log} \approx p_m\left(2K + \left(1 + 2 + \ldots + 2^{K-1}\right)\right) = p_m\left(2^K + 2K - 1\right)$$

$K = \log_2 N$

Other Spins on Shifters

- Log_4 instead of log_2
  - Advantage: Fewer stages of pass transistors
  - Disadvantage: must re-encode the control bits
  - Ex: 16 bits → 4 stages of pass transistors for log_2 vs. only 2 stages for log_4
- Can use CMOS transmission gates instead of NMOS pass transistors
- Rotate instruction – could be an ISA addition
  - 11010101 → rotate right by 5 bits → 10101110
- Reverse order shifters – do the big shifts first
  - Helpful because the big shifts have larger wire capacitances
  - Elmore delay is reduced by having large caps charged through the least resistance (fewest pass transistors)

Multiplication

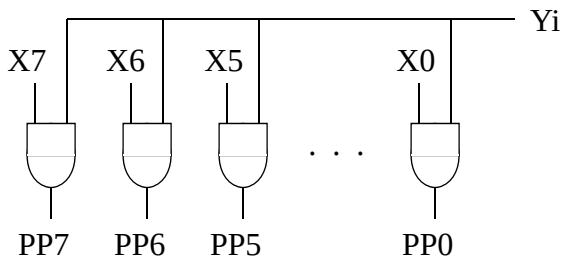


Main Steps

1. Generate partial products
2. Accumulate partial products
3. Final addition (usually using a fast carry lookahead adder)

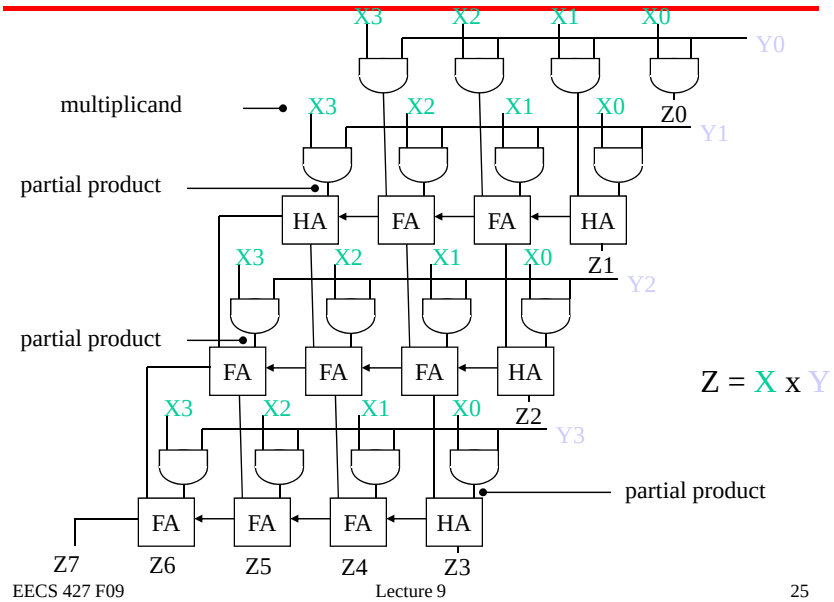
Negative numbers: Either convert to nonnegative numbers (keeping track of original signs) or use Booth multipliers

Generating Partial Products



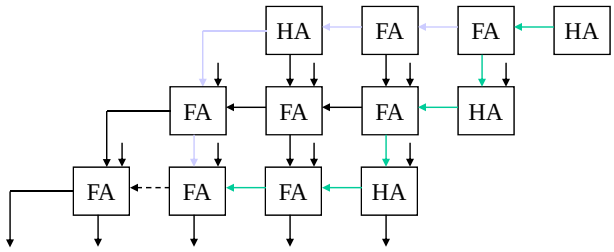
- All partial products: Bitwise AND
- Booth's algorithm reduces number of partial products

Array Multiplier



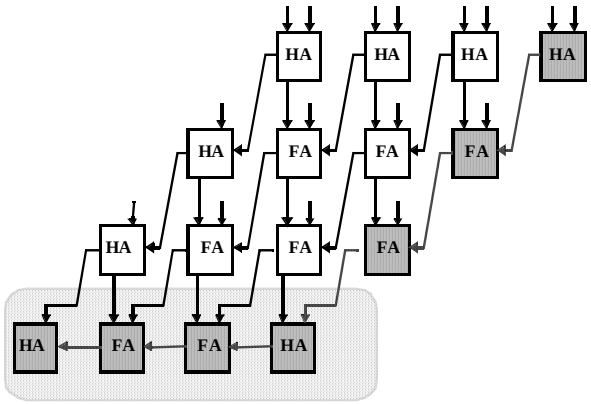
$Z = X \times Y$

Multiplier Array Critical Path



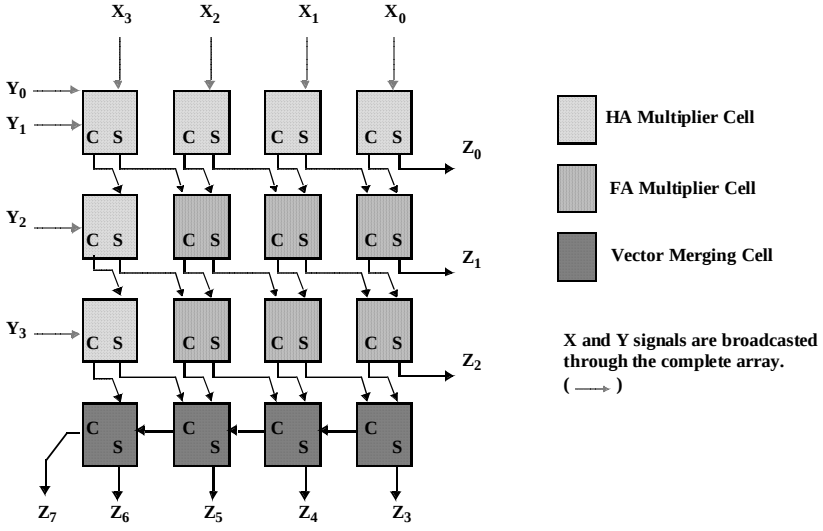
$$T \sim (M + N)\, t_{carry} + N\, t_{sum} + t_{and}$$

Carry-Save Multiplier



$$T \sim N\, t_{carry} + t_{and} + t_{merge}$$

Floorplan



Booth's Recoding

- Objective: Reduce number of partial products
- Observation: Replace a string of *n* 1s in the multiplier by a leading 1, a string of *n*-1 0s, and a trailing -1.
- Rationale: X x (2^{n-1} + … + 2^1 + 2^0) = X x (2^n – 2^0)
- Example:

```
      000111   (7)
      000110   (6)
   ----------
    11111001
   000111
   ----------
  1000101010   (42)
```

- Algorithm: For each string of *n* consecutive 1s in multiplier, subtract Y, shift *n* times to the left, and add Y.

Modified Booth Recoding

- Non-FSM hardware implementation of Booth's algorithm
- Recode multiplier in higher radix to reduce number of partial products
- For radix-4 encoding, number of partial products is N/2.

Example: Y = 0 1 1 1 0 0

1 3 0

⇨ Need to be able to multiply X by 0, 1, 2, 3 ?

Hint: Use subtraction

Recoding Table

From MSB to LSB, look at multiplier bits two at a time and in conjunction with the MSB of adjacent less-significant pair

Y_{2i+1}	Y_{2i}	Y_{2i-1}	PP_i	
0	0	0	0	
0	0	1	X	
0	1	0	X	start of string of 1s
0	1	1	2X	
1	0	0	-2X	
1	0	1	-X	why -x?
1	1	0	-X	
1	1	1	0	end of string of 1s

Overlap with next pair of bits

Example

```
                  10110010      (-78)
recode      →     10011101      (-99)
-------------------------------------
    1111111110110010
   00000001001110
    11110110010
    001001110
-------------------------------------
  10001111000101010    (7722)
```

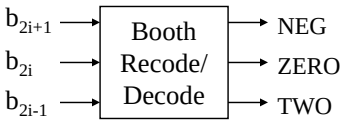
In-class example

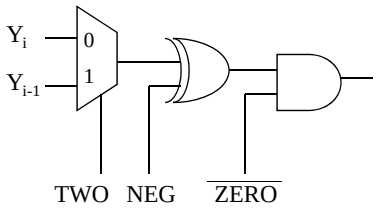
010111

X 011110

Booth Decoding and Partial Product Generation

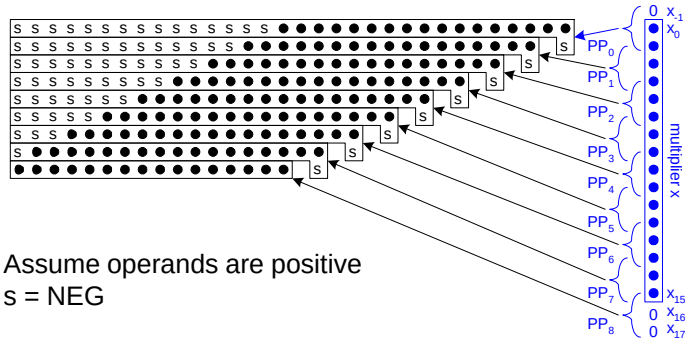
Operation	NEG	ZERO	TWO
x 0	0	1	0
x 1	0	0	0
x (-1)	1	0	0
x 2	0	0	1
x (-2)	1	0	1





Sign Extension

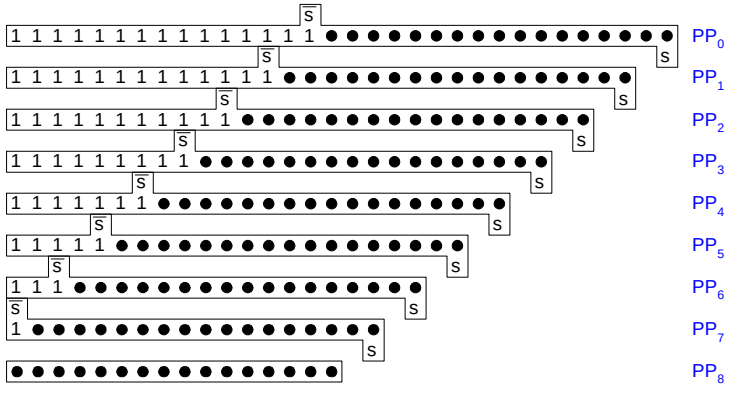
- Partial products can be negative
  - Require sign extension, which is cumbersome



Assume operands are positive
s = NEG

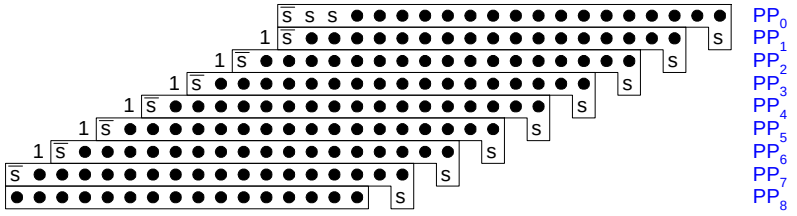
Simplified Sign Ext.

- Sign bits are either all 0's or all 1's
  - Note that all 0's is all 1's + 1 in proper column



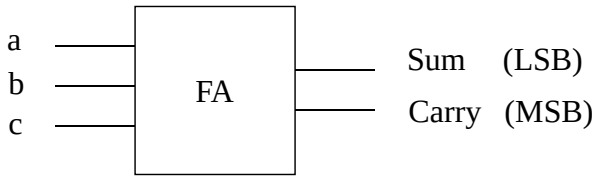
Even Simpler Sign Ext.

- No need to add all the 1's in hardware
  - Precompute the answer!



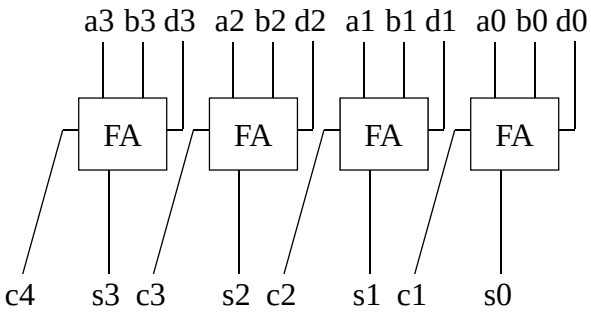
3:2 Compressor

Encode the sum of 3 bits using 2 bits

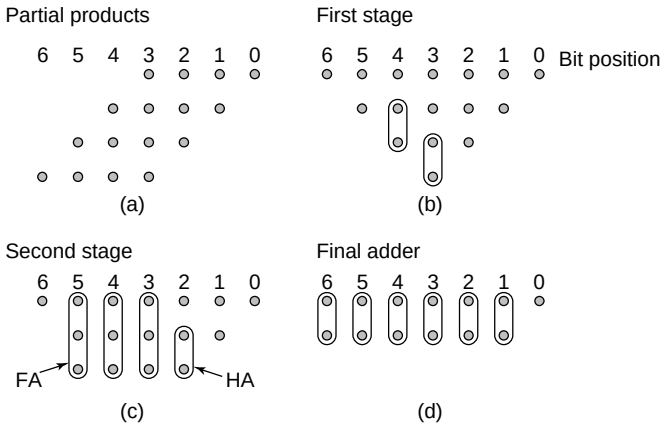


3:2 Compressor

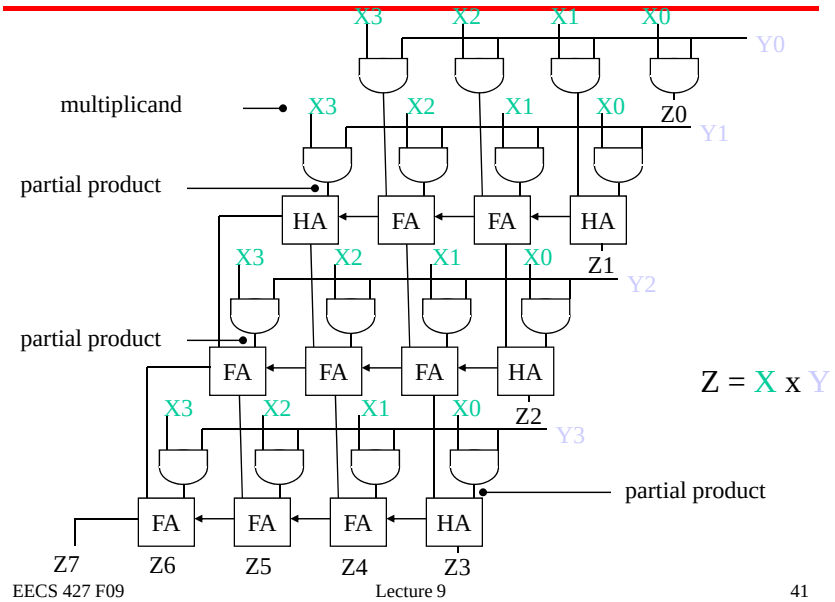
In constant time, convert the sum of 3 n-bit numbers a, b, d into a sum of 2 n-bit numbers s and c



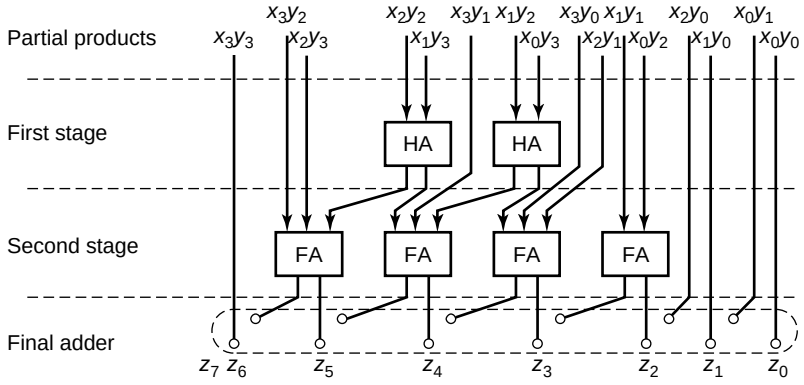
Wallace Tree Multiplier



Array Multiplier



Wallace Tree Multiplier



- In $\log_{3/2}N$ levels, tree of (3:2) compressors can transform sum of N numbers into sum of 2 numbers

Summary

- Generally speaking, multiply function consists of AND functions to generate the partial products and lots of addition
  - Carry and sum delays of adder cells can be equally critical
- Carry-save multiplier shortens critical path
  - Increases one more adder stage
- Modified Booth recoding reduces the # of partial products to be added, improves speed
  - Also suitable for 2s complement multiplication
- Tree structures reduce the # of adders needed and delay increases only logarithmically with # of bits
  - Inefficient layout structure
- Could pipeline the datapath to increase performance