EECS151/251A
Introduction to Digital Design and ICs

Lecture 20: Multipliers

Sophia Shao

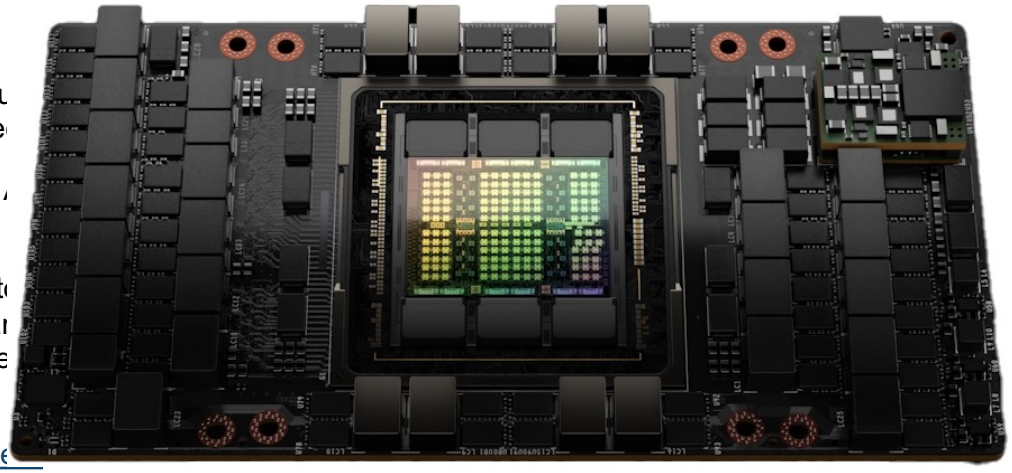
NVIDIA Announces Hopper Architecture, the Next Generation of Accelerated Computing

To power the next wave of AI data centers, NVIDIA today announced its next-generation accelerated compu platform with NVIDIA Hopper™ architecture, delivering an order of magnitude performance leap over its pre

Named for Grace Hopper, a pioneering U.S. computer scientist, the new architecture succeeds the NVIDIA A architecture, launched two years ago.

The company also announced its first Hopper-based GPU, the NVIDIA H100, packed with 80 billion transist world's largest and most powerful accelerator, the H100 has groundbreaking features such as a revolutionar Transformer Engine and a highly scalable NVIDIA NVLink® interconnect for advancing gigantic AI language deep recommender systems, genomics and complex digital twins.

https://nvidianews.nvidia.com/news/nvidia-announces-hopper-architecture-the-next-generation-of-accelerate computing



Review

- Binary adders are a common building block of digital systems
- Carry is in the critical path
- Carry-bypass, carry-select are usually faster than ripple-carry for lengths > 8
- Carry-lookahead, O(~logN) is often the fastest adder with N > 16

- **Multipliers**
  - **Basics**
  - **Parallel Multipliers**
  - **Booth Recoding**
  - **Signed Multipliers**

Warmup

- Recall long multiplication of base-10 by hand:

$$\begin{array}{r} 12 \\ \times\ 56 \\ \hline \end{array}$$

- In base-2 (binary), we do the same thing:

$$\begin{array}{r} 011 \\ \times\ 101 \\ \hline \end{array}$$

Multiplication

$$
\begin{array}{rcccc}
 & a_3 & a_2 & a_1 & a_0 \\
\times & b_3 & b_2 & b_1 & b_0 \\
\hline
 & a_3b_0 & a_2b_0 & a_1b_0 & a_0b_0 \\
 & a_3b_1 & a_2b_1 & a_1b_1 & a_0b_1 \\
 & a_3b_2 & a_2b_2 & a_1b_2 & a_0b_2 \\
a_3b_3 & a_2b_3 & a_1b_3 & a_0b_3 & \\
\hline
\ldots & & a_1b_0+a_0b_1 & a_0b_0 &
\end{array}
$$

← *Multiplicand*

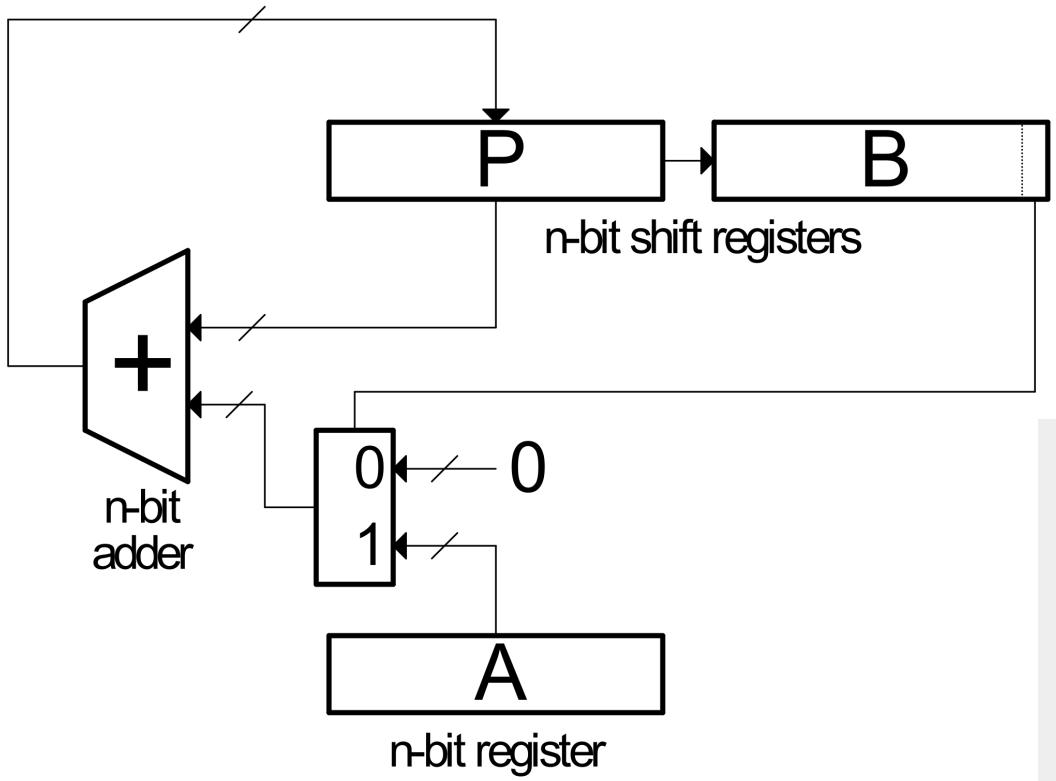
← *Multiplier*

Partial products

← *Product*

Many different circuits exist for multiplication.
Each one has a different balance between speed (performance) and amount of logic (energy, cost).

“Shift and Add” Multiplier



- Sums each partial product, one at a time.
- In binary, each partial product is shifted versions of A or 0.

Control Algorithm:

1. P ← 0, A ← multiplicand, B ← multiplier
2. If LSB of B==1 then add A to P
   else add 0
3. Shift [P][B] right 1
4. Repeat steps 2 and 3 (n-1) more times.
5. [P][B] has product.

- Performance: N cycles of N-bit additions

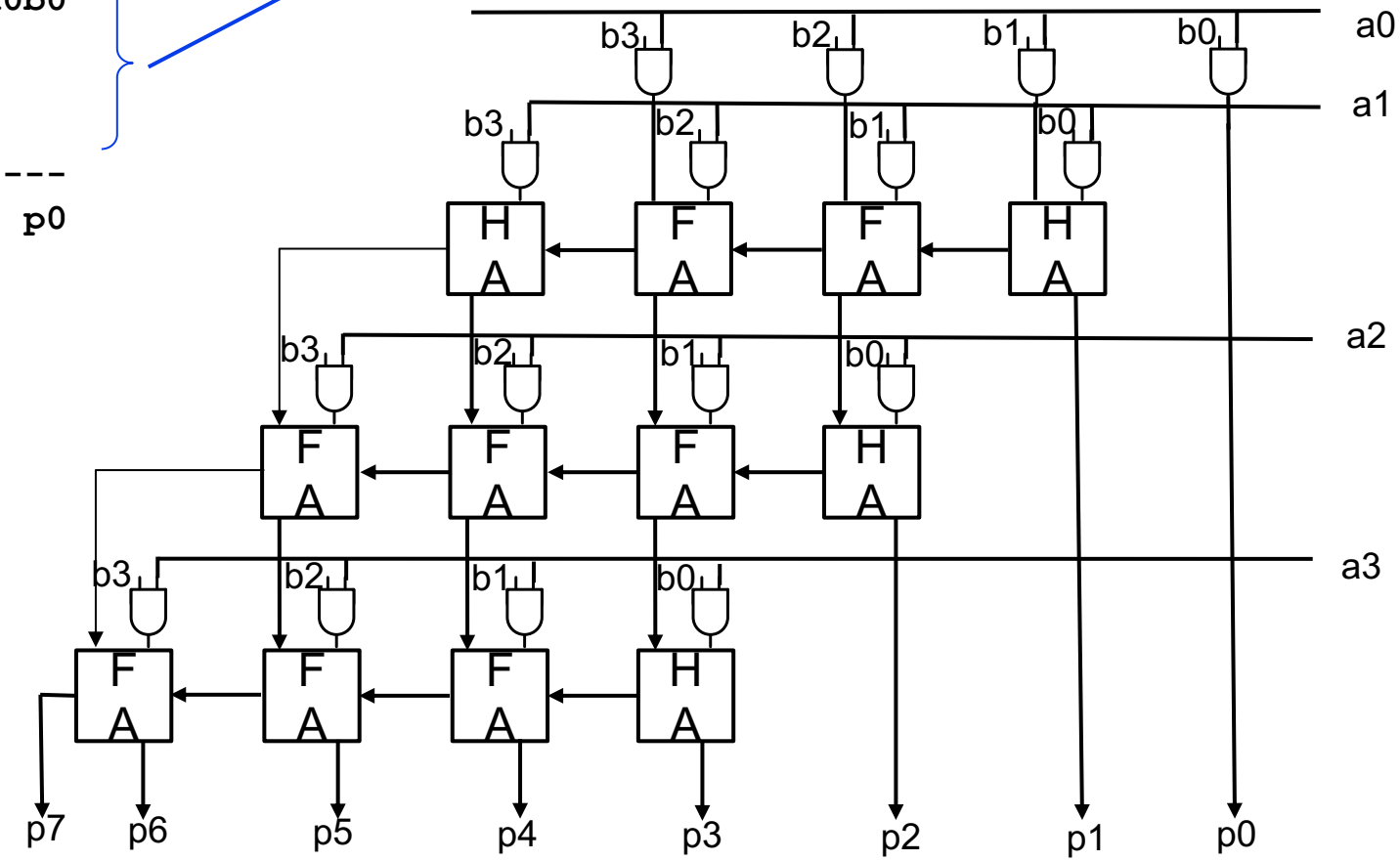
- **Multipliers**
  - **Basics**
  - **Parallel Multipliers**
  - **Booth Recoding**
  - **Signed Multipliers**

Parallel (Array) Multiplier

```
                        a3    a2    a1    a0
                    *   b3    b2    b1    b0
                    ------------------------
                       a3b0  a2b0  a1b0  a0b0
+                 a3b1  b2a1  a1b1  a0b1
+           a3b2  a2b2  a1b2  a0b2
+     a3b3  a2b3  a1b3  a0b3
---------------------------------------
   p7    p6    p5    p4    p3    p2    p1    p0
```

multiplicand
multiplier

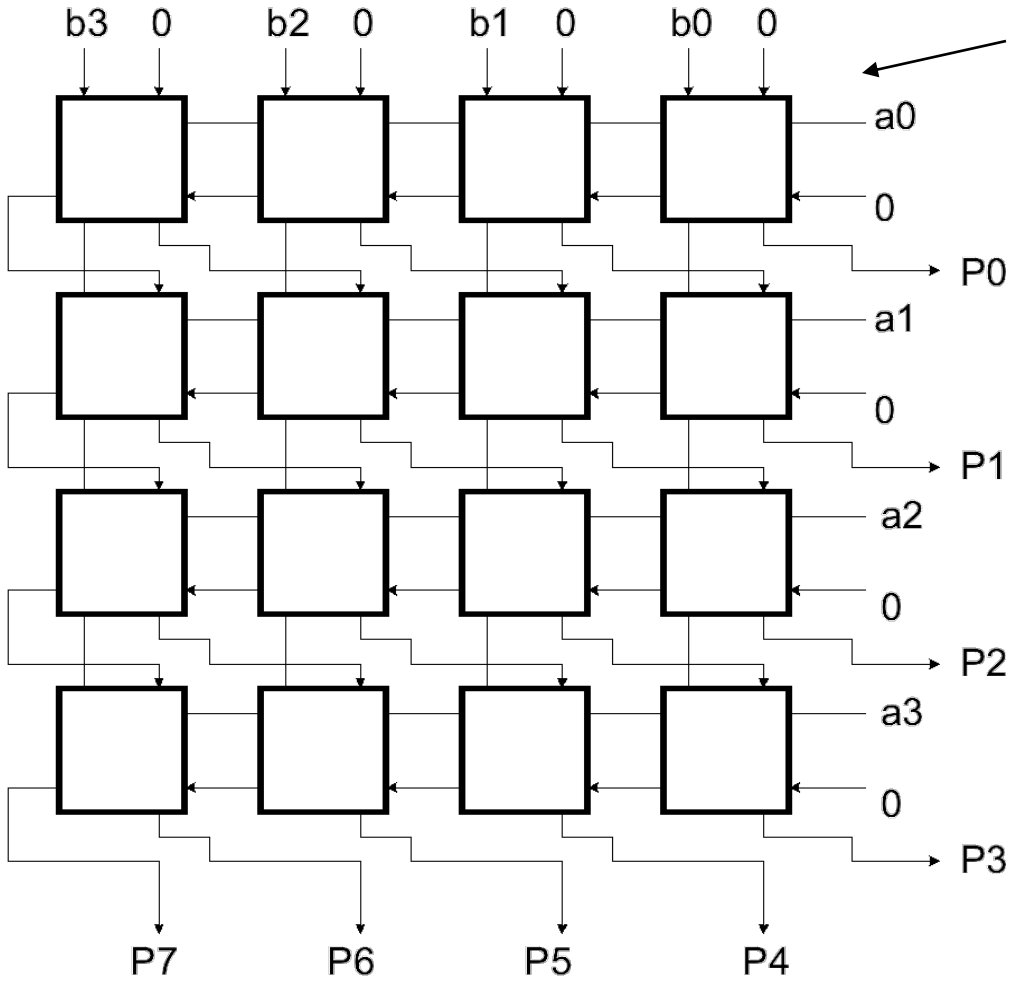
Partial products, one for each bit in multiplier
(each bit needs just one AND gate)



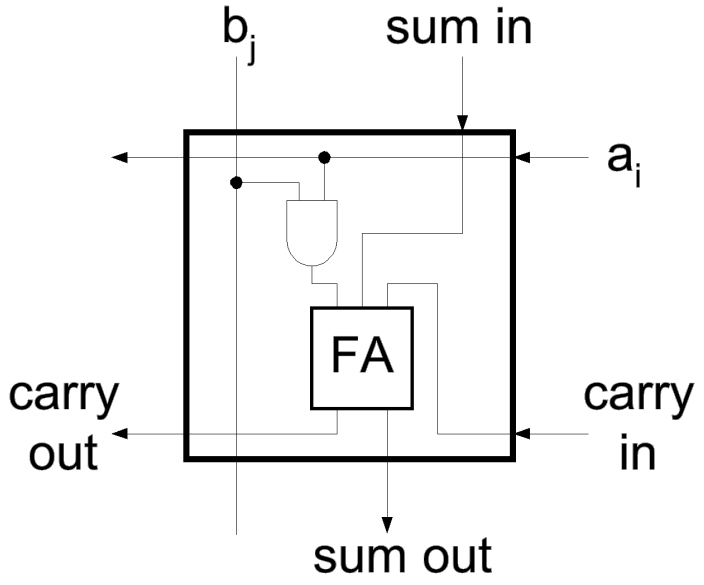
- Performance: What is the critical path?

Parallel (Array) Multiplier

Single cycle multiply: Generates all n partial products simultaneously.



Each row: n-bit adder with AND gates



Carry-Save Addition

- Speeding up multiplication is a matter of speeding up the summing of the partial products.
- "Carry-save" addition can help.
- Carry-save addition passes (saves) the carries to the output, rather than propagating them.

Example: sum three numbers,
$3_{10} = 0011$, $2_{10} = 0010$, $3_{10} = 0011$

$$
\begin{array}{rl}
 & 3_{10}\ 0011 \\
+ & 2_{10}\ 0010 \\
\hline
 & \text{c}\ 0100 = 4_{10} \\
 & \text{s}\ 0001 = 1_{10}
\end{array}
$$

carry-save add

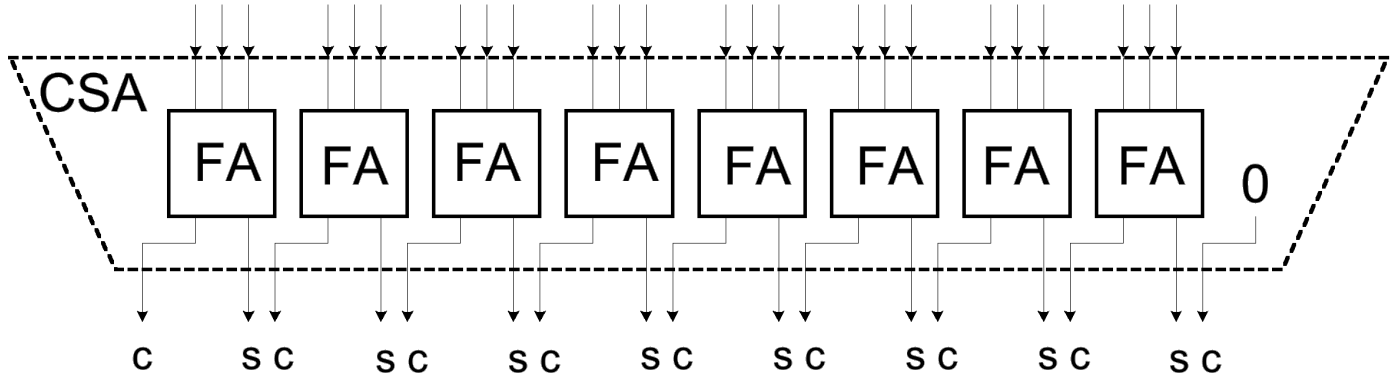
$$
\begin{array}{rl}
+ & 3_{10}\ 0011 \\
\hline
 & \text{c}\ 0010 = 2_{10} \\
 & \text{s}\ 0110 = 6_{10} \\
\hline
 & 1000 = 8_{10}
\end{array}
$$

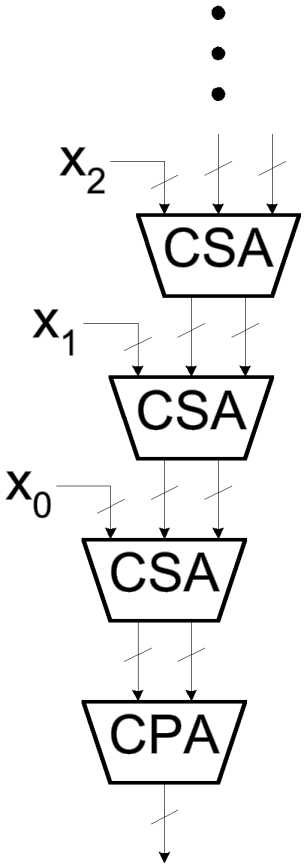
carry-save add

carry-propagate add

- *Carry-save* addition takes in 3 numbers and produces 2:"3:2 compressor"
  - Whereas, *carry-propagate* takes 2 and produces 1.
- With this technique, we can avoid carry propagation until final addition

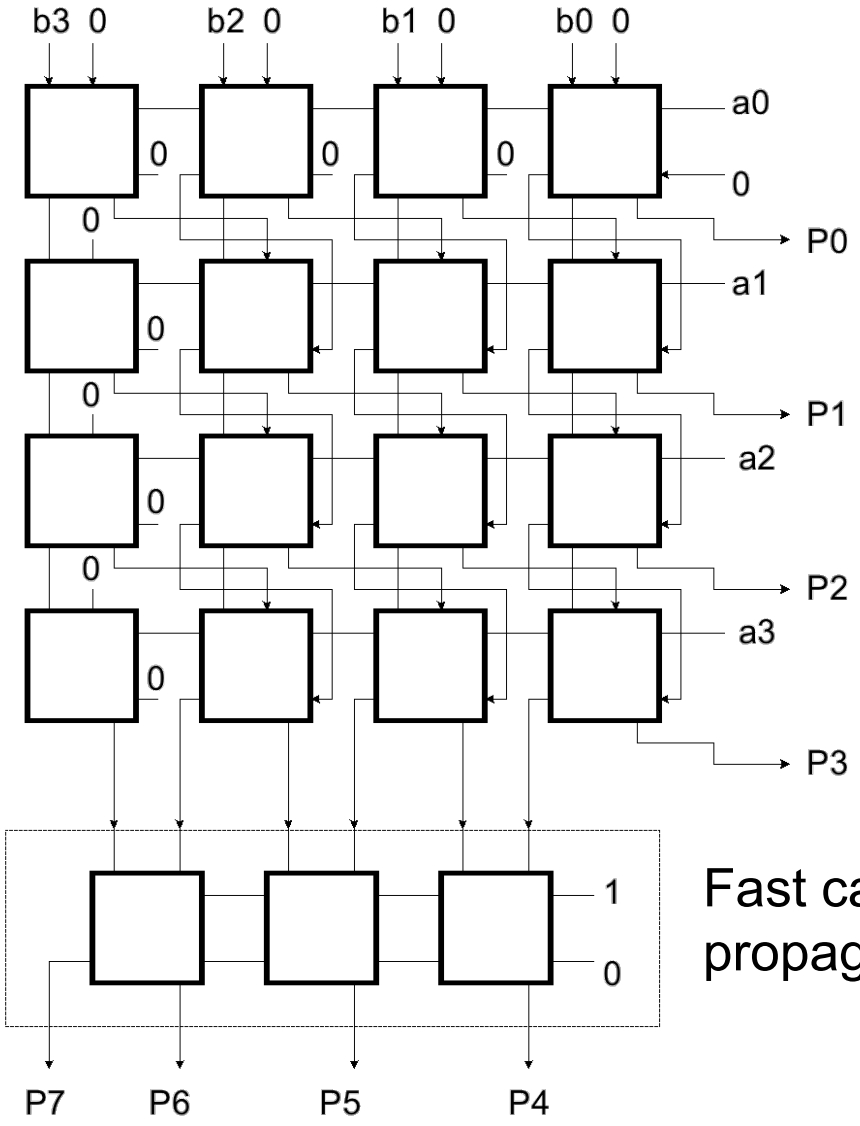
Carry-Save Circuits

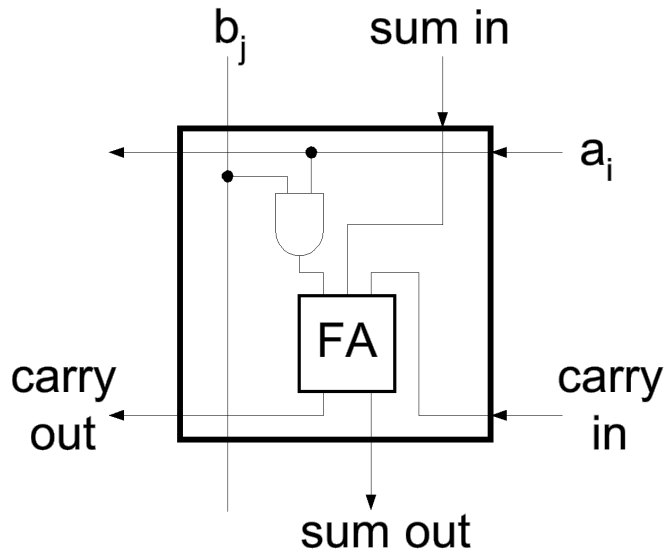




- When adding sets of numbers, carry-save can be used on all but the final sum.
- Standard adder (carry propagate) is used for final sum.
- Carry-save is fast (no carry propagation) and inexpensive (full adders)

Array Multiplier Using Carry-Save Addition





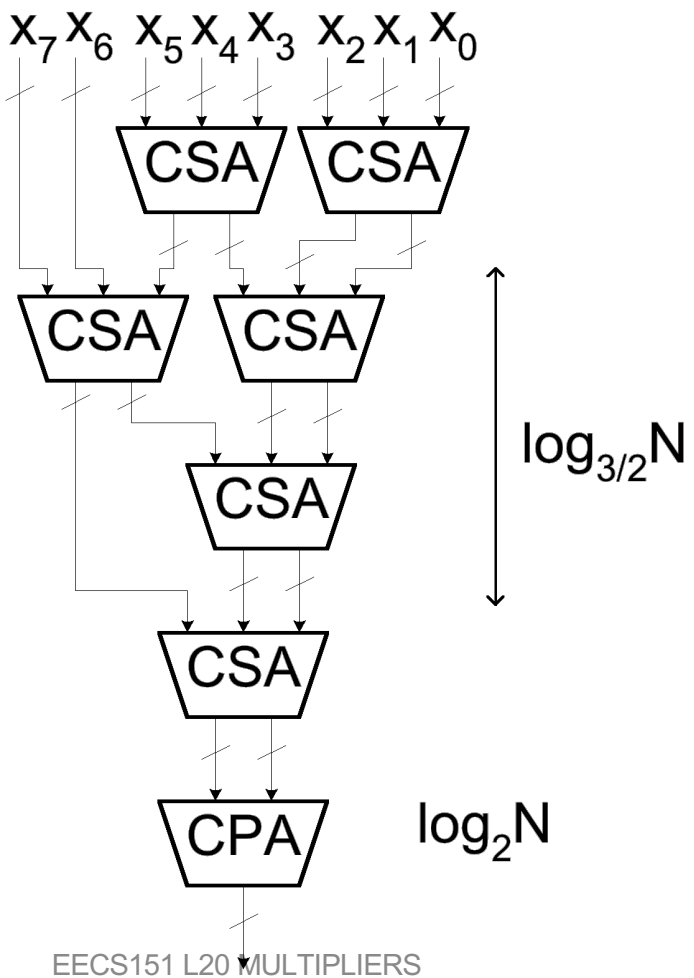
Fast carry-propagate adder

- What is the critical path?

Carry-Save Addition

CSA is associative and commutative. For example:

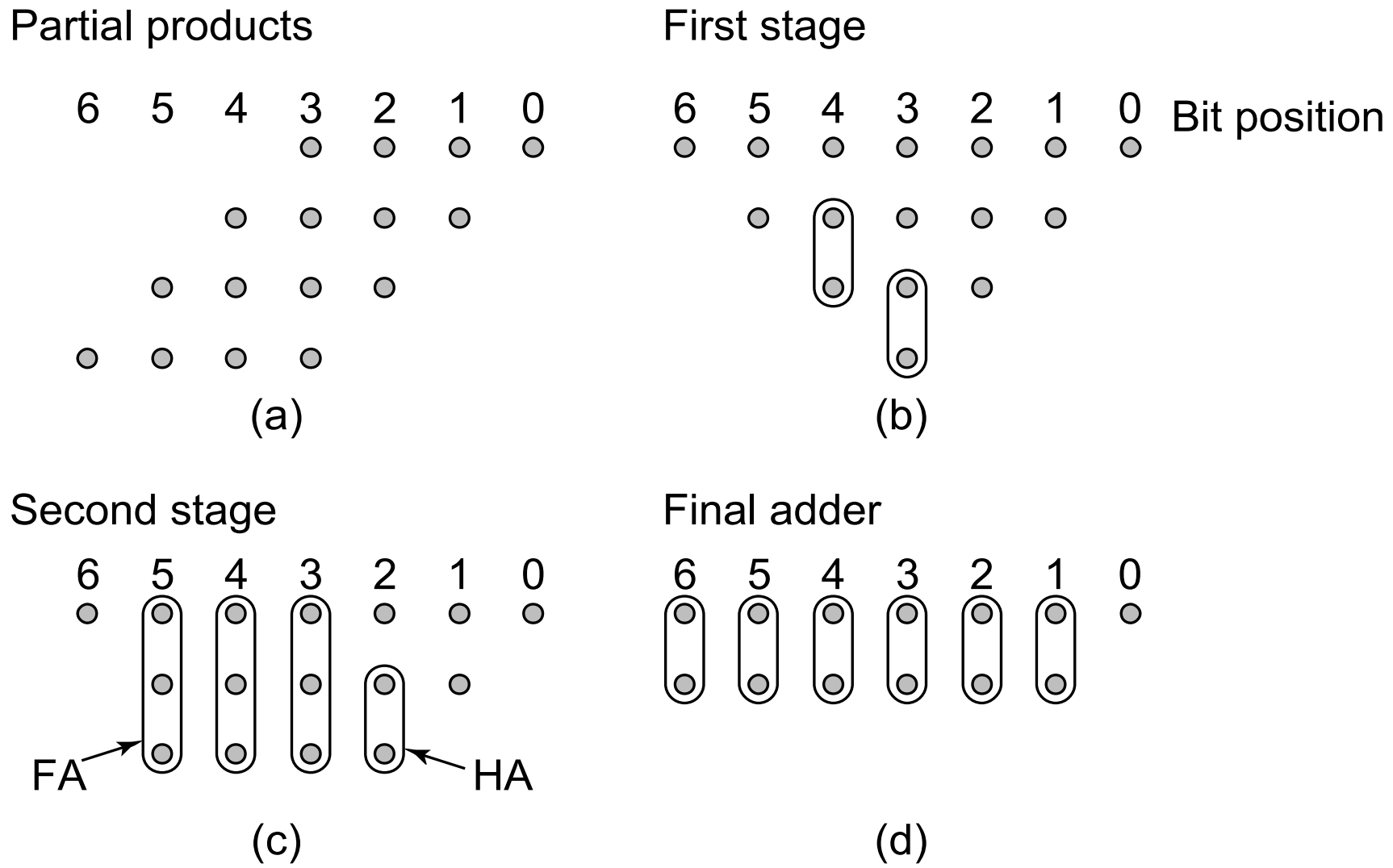
$$(((X_0 + X_1) + X_2) + X_3) = ((X_0 + X_1) + (X_2 + X_3))$$



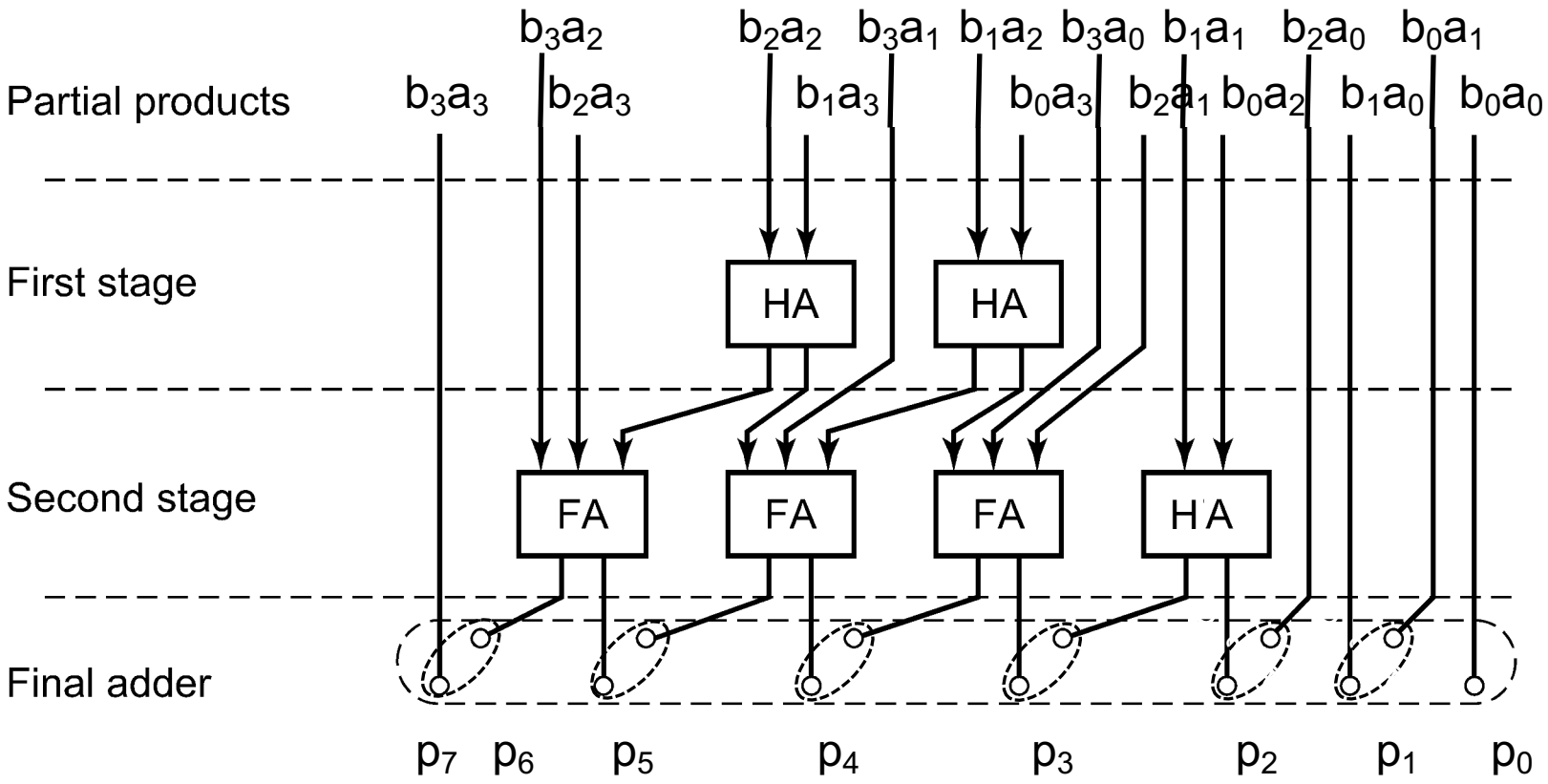
- A balanced tree can be used to reduce the logic delay
- It doesn't matter where you add the carries and sums, as long as you eventually do add them
- This structure is the basis of the ***Wallace Tree Multiplier***
- Partial products are summed with the CSA tree. Fast CPA (ex: CLA) is used for final sum
- Multiplier delay α $\log_{3/2}N + \log_2 N$

Wallace-Tree Multiplier

- Reduce the partial products in logic stages – 4 x 4 example



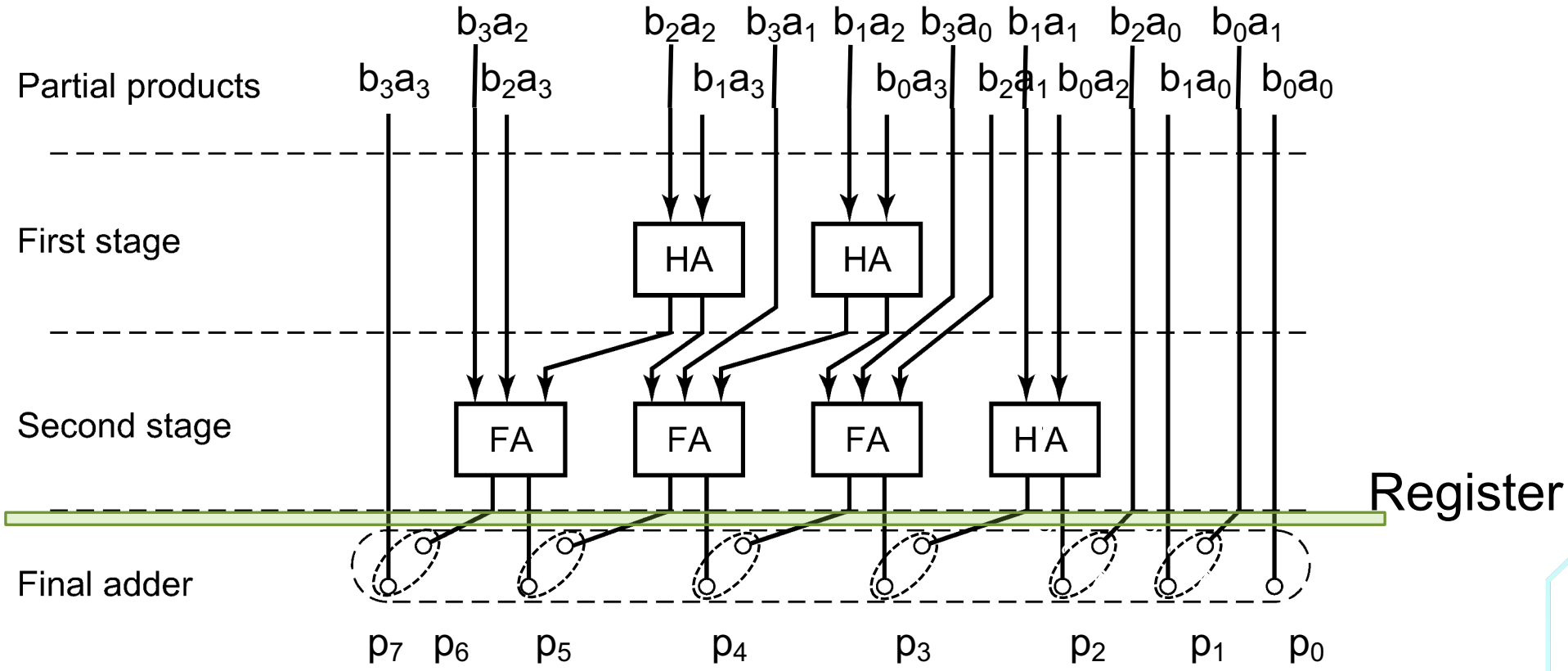
Wallace-Tree Multiplier



Note: Wallace tree is often slower than an array multiplier in FPGAs (which have optimized carry chains)

Increasing Throughput: Pipelining

- Multipliers have a long critical path: PP generation → reduction tree → final adder
  - Often pipelined before final adder (2x flip-flops for carry-save)



Administrivia

- New homework this week.
- Project, project, project!
  - Checkpoint 1 this week.

- **Multipliers**
  - **Basics**
  - **Parallel Multipliers**
  - **Booth Recoding**
  - **Signed Multipliers**

Booth Recoding: Motivation

$$
\begin{array}{rcccc}
 & a_3 & a_2 & a_1 & a_0 \\
\times & b_3 & b_2 & b_1 & b_0 \\
\hline
 & a_3b_0 & a_2b_0 & a_1b_0 & a_0b_0 \\
 & a_3b_1 & a_2b_1 & a_1b_1 & a_0b_1 \\
 & a_3b_2 & a_2b_2 & a_1b_2 & a_0b_2 \\
a_3b_3 & a_2b_3 & a_1b_3 & a_0b_3 & \\
\hline
\ldots & & a_1b_0+a_0b_1 & a_0b_0 &
\end{array}
$$

← *Multiplicand*

← *Multiplier*

Partial products

← *Product*

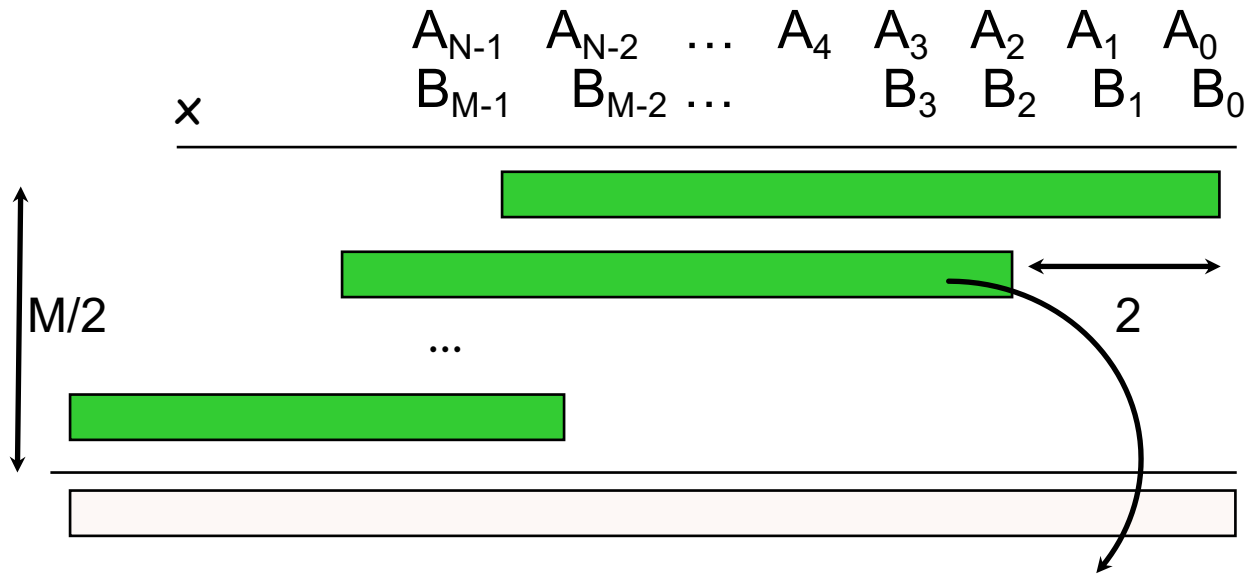
How many non-zero partial products (out of N)?

- N, if B = 000…0
- 0, if B = 111…1
- N/2 on the average

Booth Recoding: Higher-radix multiplier

- Idea: If we could use, say, 2 bits of the multiplier in generating each partial product we would halve the number of columns and speed it up!
- Encode …0111100… patterns:
  - 1111 = 2^3 + 2^2 + 2^1 + 2^0 = 2^4 – 2^0
  - Only two non-zero numbers, but needs to represent +1 and -1

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 → 2A (or 4A – 2A)
= 3*A → 4A – A

Booth recoding

(On-the-fly canonical signed digit encoding!)

current bit pair

from previous bit pair

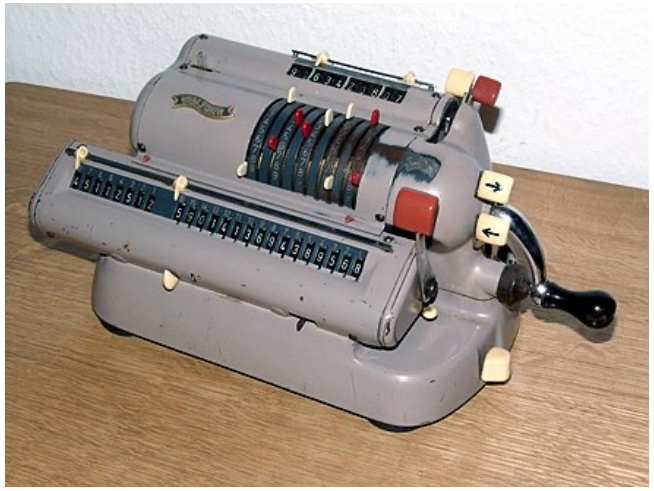
B_{K+1}	B_K	B_{K-1}	action
0	0	0	add 0
0	0	1	add A
0	1	0	add A
0	1	1	add 2*A
1	0	0	sub 2*A
1	0	1	sub A ← -2*A+A
1	1	0	sub A
1	1	1	add 0

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

Example

- Compression tree needs to support subtraction

```
      0111                    A
 x    1010                    B
----------
    -01110          10(0)  -2A
 -00111             101     -A
+0111               001     +A
----------
 01000110
```



A Walther WSR160 arithmometer (from Wikipedia)

Booth Recoding Notes

- Key advantage: Reduces the number of partial products
  - Compression tree depth becomes $\log_{3/2}[N/2]$
  - Partial product generation is slightly more complex than a NAND2
- Useful for larger multipliers
  - And some very creative solutions for repeated multiplications (FIR filters, etc)

- **Multipliers**
  - **Basics**
  - **Parallel Multipliers**
  - **Booth Recoding**
  - **Signed Multipliers**

“Shift and Add” Multiplier

Signed Multiplication:

Remember for 2’s complement numbers MSB has negative weight:

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

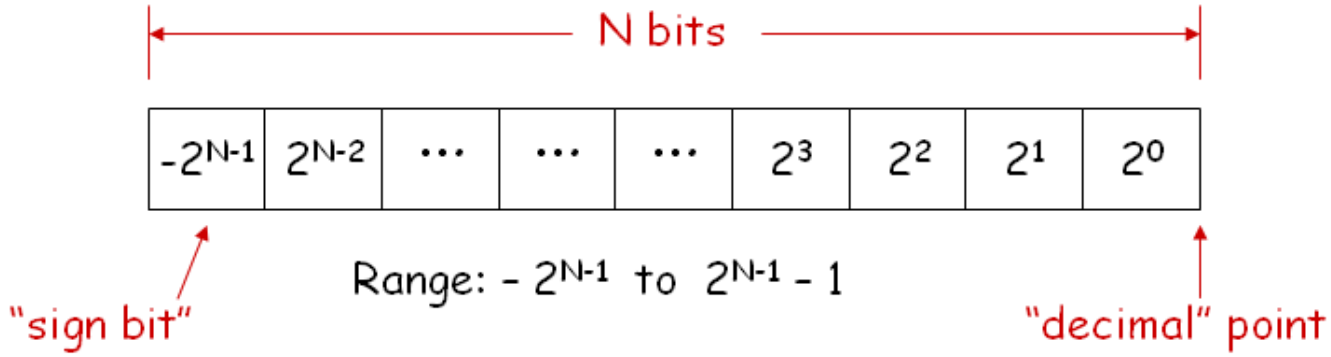
ex: $-6 = 11010_2 = 0 \cdot 2^0 + 1 \cdot 2^1 + 0 \cdot 2^2 + 1 \cdot 2^3 - 1 \cdot 2^4$

$= 0 + 2 + 0 + 8 - 16 = -6$

- Therefore for multiplication:
  - a) subtract final partial product
  - b) sign-extend partial products

Signed Array Multiplier

- Two's complement




```
                (-3) * (-2)

(-3)                1   0   1        (X)
(-2)            *   1   1   0        (Y)
                  ---------------------
                  0 0 0 0 0 0            Y0*X =  0
                + 1 1 1 0 1              2Y1*X = -6
                - 1 1 0 1                4Y2*X = -12
                ----------------------
(+6)              0 0 0 1 1 0
```

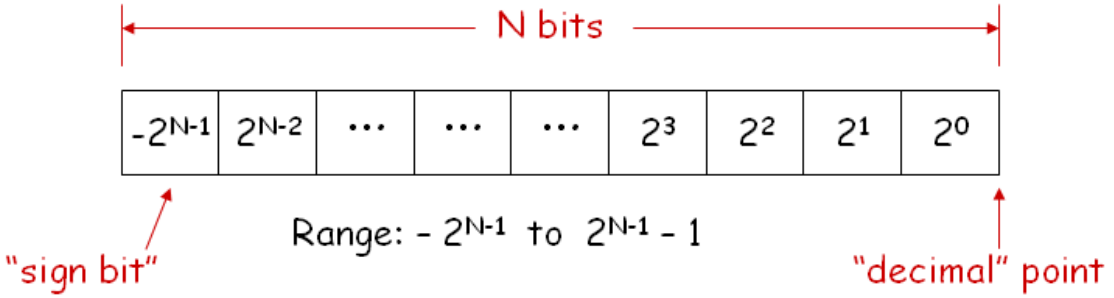
Convince yourself

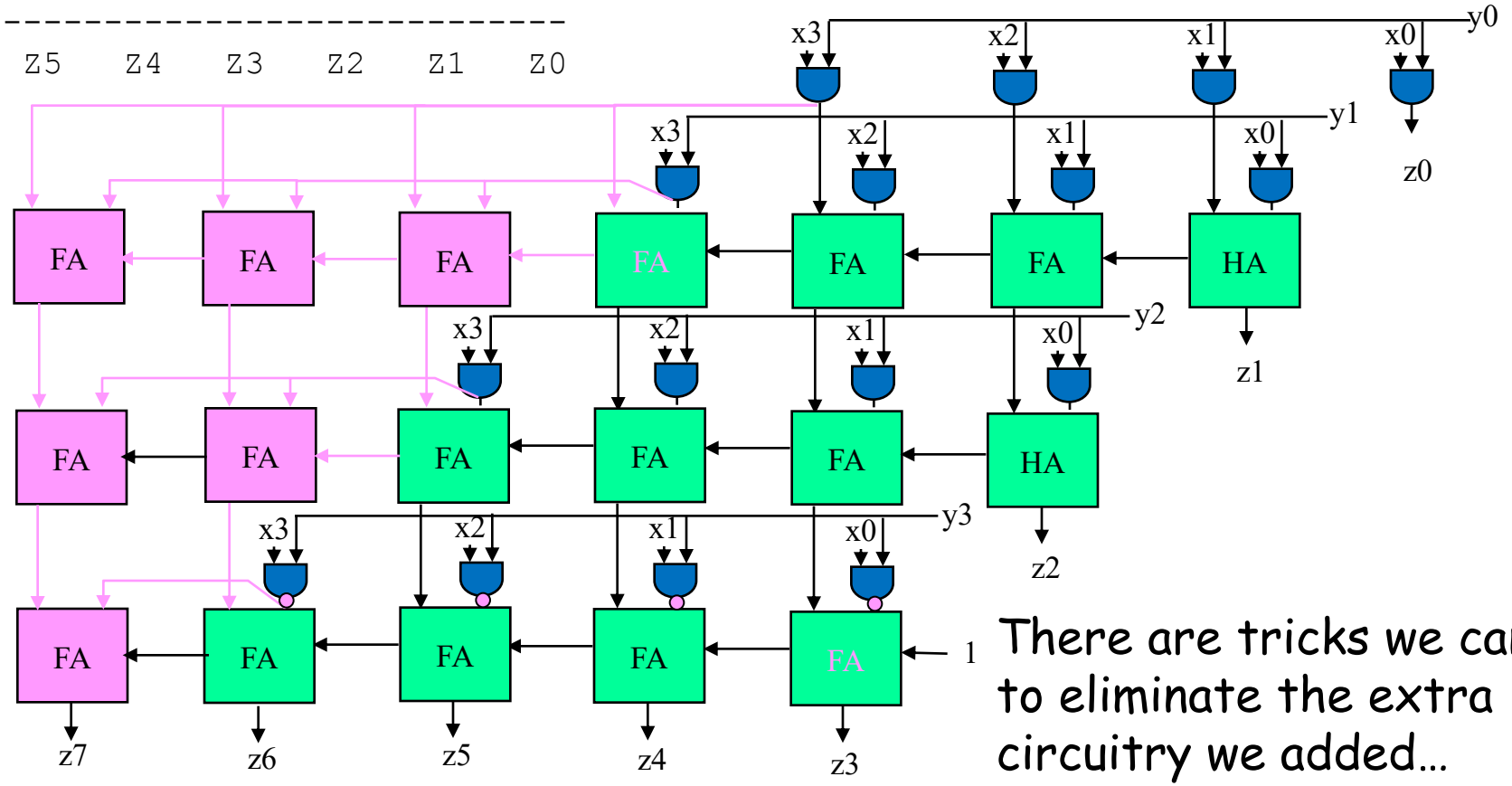
- What's -3 x 5?

$$\begin{array}{r} 1101 \\ \times\ 0101 \\ \hline \end{array}$$

Combinational Multiplier (signed)

```
                        X3    X2    X1    X0
                  *     Y3    Y2    Y1    Y0
                      ------------------------
  X3Y0 X3Y0 X3Y0 X3Y0 X3Y0 X2Y0 X1Y0 X0Y0
+ X3Y1 X3Y1 X3Y1 X3Y1 X2Y1 X1Y1 X0Y1
+ X3Y2 X3Y2 X3Y2 X2Y2 X1Y2 X0Y2
- X3Y3 X3Y3 X2Y3 X1Y3 X0Y3
-----------------------------------------
    Z7    Z6    Z5    Z4    Z3    Z2    Z1    Z0
```





There are tricks we can use to eliminate the extra circuitry we added…

2's Complement Multiplication (Baugh-Wooley)

Step 1: two's complement operands so high order bit is -2^{N-1}. Must sign extend partial products and subtract the last one

```
                          X3    X2    X1    X0
                      *   Y3    Y2    Y1    Y0
                      ------------------------
  X3Y0  X3Y0  X3Y0  X3Y0  X3Y0  X2Y0  X1Y0  X0Y0
+ X3Y1  X3Y1  X3Y1  X3Y1  X2Y1  X1Y1  X0Y1
+ X3Y2  X3Y2  X3Y2  X2Y2  X1Y2  X0Y2
- X3Y3  X3Y3  X2Y3  X1Y3  X0Y3
---------------------------------------
    Z7    Z6    Z5    Z4    Z3    Z2    Z1    Z0
```

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  X3Y2  X2Y2  X1Y2  X0Y2
+             1
+ ~X3Y3 ~X3Y3 ~X2Y3 ~X1Y3 ~X0Y3      } -B = ~B + 1
+                         1
+       1
-       1     1     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
```

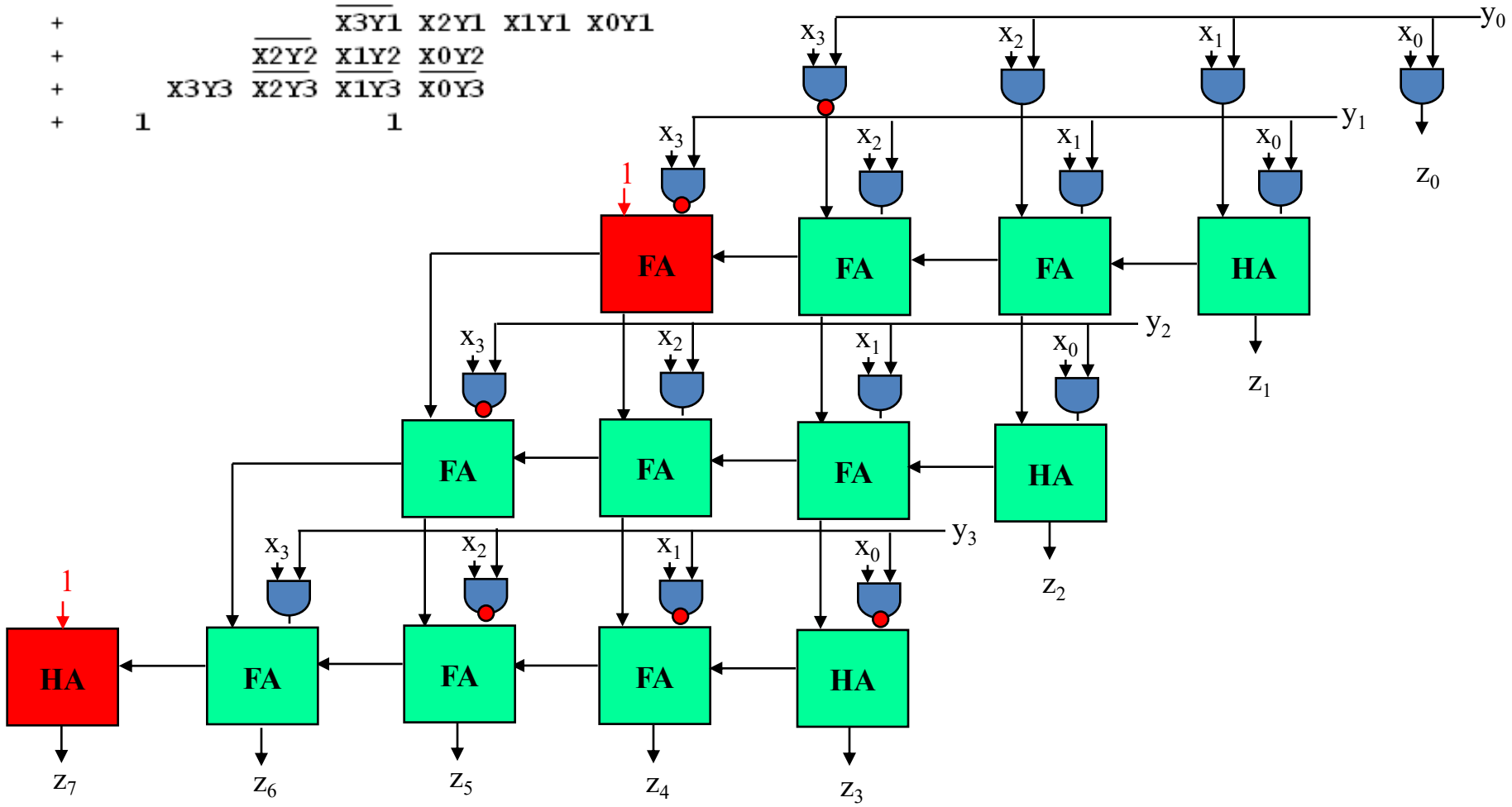
Step 4: finish computing the constants…

```
                          ~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!

2's Complement Multiplication (Baugh-Wooley)

$$
\begin{array}{llllll}
 & & & \overline{X3Y0} & X2Y0 & X1Y0 \quad X0Y0 \\
+ & & \overline{X3Y1} & X2Y1 & X1Y1 & X0Y1 \\
+ & & \overline{X2Y2} & X1Y2 & X0Y2 & \\
+ & X3Y3 & \overline{X2Y3} & \overline{X1Y3} & \overline{X0Y3} & \\
+ & 1 & & 1 & &
\end{array}
$$



Summary

- Binary number multiplications
  - Shift-and-add multiplier
- Parallel adder array
- Partial-Product accumulation
  - Carry-save adder
  - Wallace Tree multiplier
- Partial-Product Generation
  - Booth Encoding