Today: array multiplier, signed multiply, Booth encoding CO 6.3,6.

Tomorrow: Booth encoding, fast multipliers CO 6.5

Note: DL 5.6 discusses multiplication as well, I prefer CO

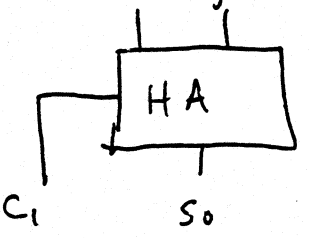
Review (CO 6.1)

grade 1 - add two 1-bit numbers: x_0, y_0

$$\begin{array}{r} x_0 \\ +\ y_0 \\ \hline c_1\ s_0 \end{array}$$

$s_0 = x_0 \oplus y_0$

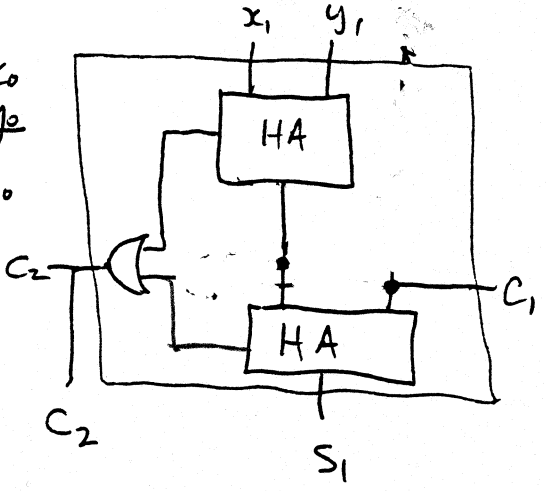
$c_1 = x_0 \cdot y_0$



half adder

grade 2 - add two 2-bit numbers:

$$\begin{array}{r} c_1 \\ x_1\ x_0 \\ +\ y_1\ y_0 \\ \hline c_2\ s_1\ s_0 \end{array}$$



"FA" full adder

HA and FA add bits of the same magnitude and generate multi-bit result, aka "counter" or "compressor" (bit)

FA is called 3:2 compressor

(3,2) counter

other compressor/counters can be built [advanced multipliers]

grade 3 - multiply two 2-bit numbers

try base 10 first

```
  13      multiplicand   M
x 11      multiplier     Q
  13    ← 13 x 1
 13     ← 13 x 10
 143      product        P
```

try binary

```
      m1 m0
    x q1 q0
    m1q0 m0q0      } form partial products using AND gates
+ m1q1 m0q1        } sum the partial products, use FA
```

3rd yr university - multiply in binary, build an array multiplier using digital logic (co. 6-3)

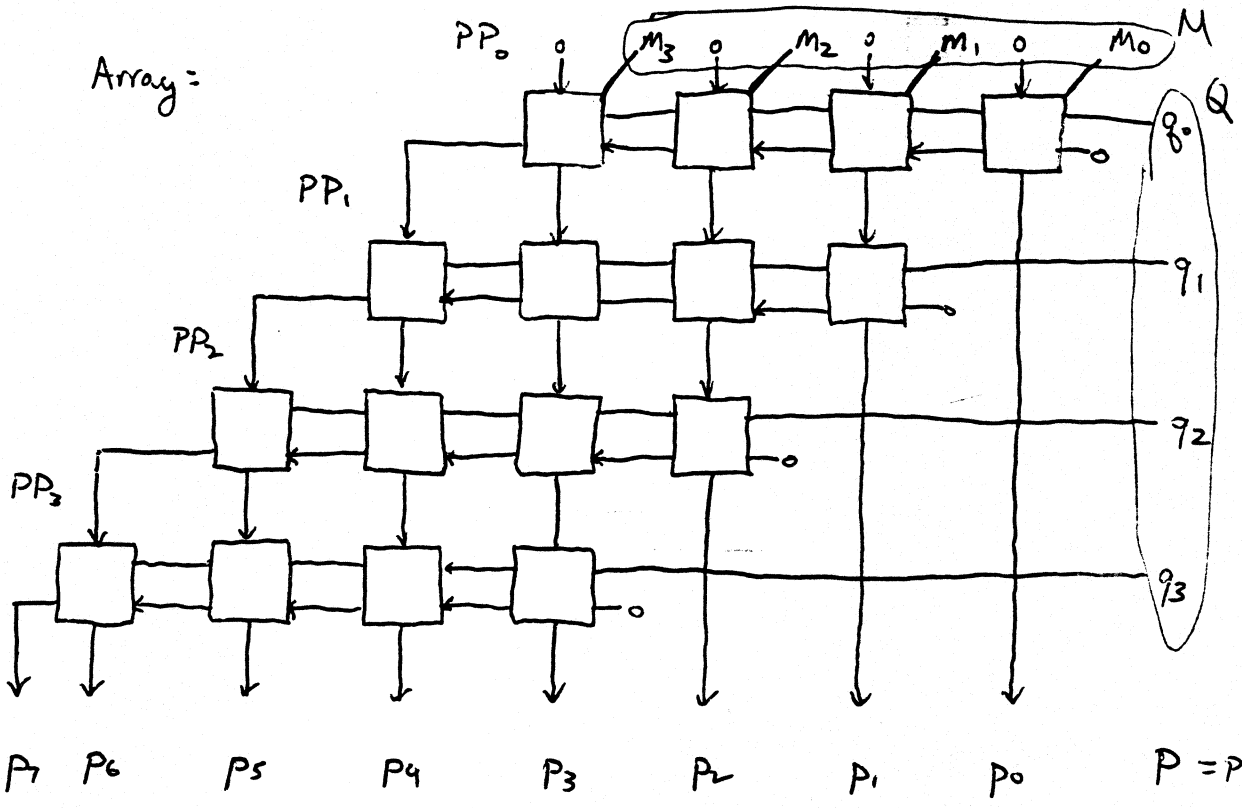
```
                1 1 0 1     (13)  M      4 bits
              x 1 0 1 1     (11)  Q      4 bits
array           1 1 0 1                    |
              1 1 0 1                      | 4+4 =
            0 0 0 0                        |
          1 1 0 1                          v
        1 0 0 0 1 1 1 1     (143 = 128+15) P   8 bits
```

- we can build an array structure that adds the partial products as we go from one row to the next
  ie, compute new partial products* as a "running total"

* these are PP_i described in the text

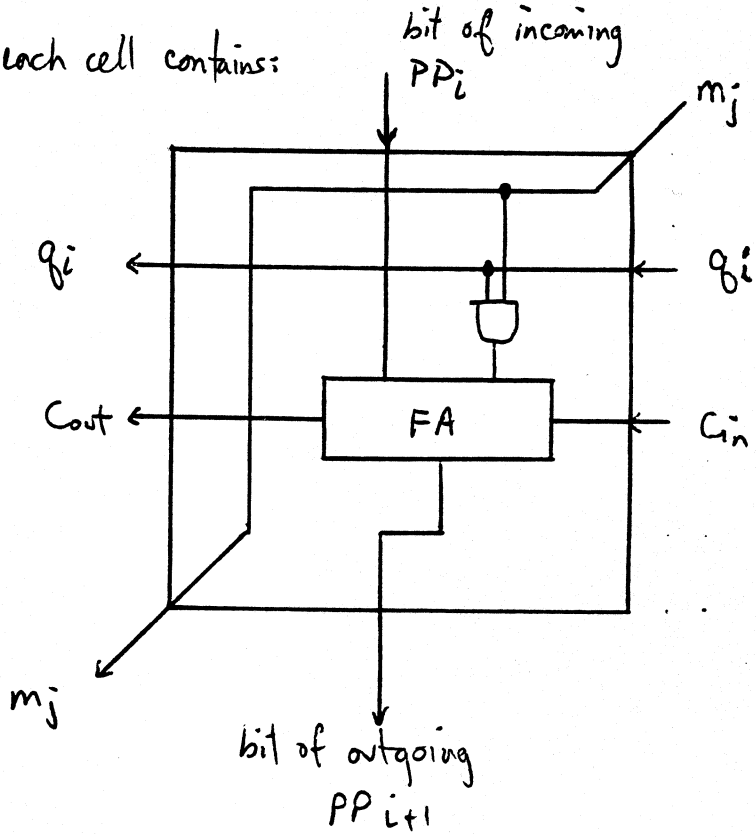
```
                PP0   0 0 0 0
                PP1   1 1 0 1
array           PP2 1 0 0 1 1
"running sum"   PP3 0 1 0 0 1
                PP4 1 0 0 0 1
              P=PP4 1 0 0 0 1 1 1 1

      1 1 0 1          PP1
    + 1 1 0 1
    1 0 0 1 1 1        PP2
  + 0 0 0 0
    1 0 0 1 1 1        PP3
+ 1 1 0 1
1 0 0 0 1 1 1 1        PP4
```

Array:



P7 P6 P5 P4 P3 P2 P1 P0 P = PP4

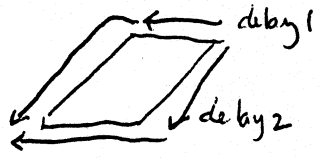
where each cell contains:



Note: large area (many adders: size(M) x size(Q))

worst-case delay roughly O(|M|) + O(|Q|)

→ we can do better (tomorrow!)



Note: signed numbers? P.i. L11

Signed Multiplication (CO 6.4, some new material here not in text)

- Recall: M × Q = P what if M<0? Q<0?
- We express <0 numbers in two's complement form: $-M = \overline{M} + 1$

01101	(13)	10011	(-13)
10010	($\overline{13}$)	01100	($\overline{-13}$)
+00001	(+1)	+00001	(+1)
10011	(-13)	01101	(13)

sign bit

- Signed arithmetic uses different logic than unsigned

Rules

(A) if Q unsigned or Q ≥ 0
multiply by adding sign extension of partial products
→ intuitive (adding signed M's)

(B) if Q < 0
we have two choices:
① note $M \cdot Q = (-M)(-Q) = (\overline{M}+1)(\overline{Q}+1)$ +'ve, do Q≥0 case
OR
② use Booth recoding (good, more useful)

(A) sign extension of partial products

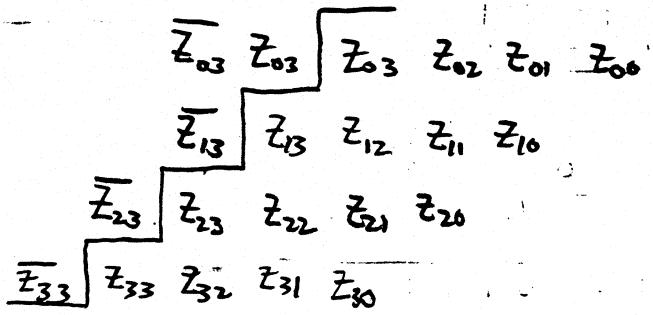
```
                1 0 0 1 1   (-13)
              x 0 1 0 1 1   (+11)
            ---------------
            1 1 1 1 1 0 0 1 1
          1 1 1 1 0 0 1 1
        0 0 0 0 0 0 0 0
      1 1 1 0 0 1 1
    0 0 0 0 0 0
  ---------------------------
  1 1 0 1 1 1 0 0 0 1   (-143)
```

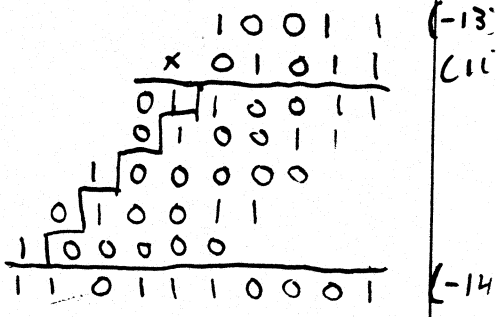
Notice: array just got bigger!

Solution 1: D.L. 5.6 shows sign extension by just 1 bit is sufficient if you do running sums of partial products (why?)

Solution 2: not in text...

Sign Extension Optimization (details later)





(B) Booth recoding (to handle Q < 0)

- easier than sign extending Q as well
- useful to build fast multipliers (tomorrow)

we can recode $Q \rightarrow Q'$ using three digits $\{\bar{1}, 0, 1\}$

$\bar{1} \equiv -1$ in text

digit q_i' formed by examining $q_i\, q_{i-1}$:

q_i	q_{i-1}	q_i'
0	0	0
0	1	1
1	0	$\bar{1}$
1	1	0

assume $q_{-1} = 0$

Example: Q 0 0 1 1 1 1 0 (30)
Q' 0 1 0 0 0 $\bar{1}$ 0 (30, Booth encoded)

Notice: 0 1 0 0 0 $\bar{1}$ 0 = 0 1 0 0 0 0 0 − 0 0 0 0 1 0
↑ ↑ (32) − (2)
+32 −2 = 0 0 1 1 1 1 0

often called skipping over 1's :

. . . 0 0 1 1 1 1 1 0 0 . . .
1 0 0 0 0 $\bar{1}$ 0 0

skips n ones

Now $M \cdot Q = M \cdot Q'$, but we need a new rule for multiplying by Q' (by $\bar{1}$ in particular)

When we mpy M by $\bar{1}$ (=-1), we take the two's complement of M : $\bar{M}+1$

Example

$$\begin{array}{rr} 0\,1\,1\,0\,1 & (13) \\ \times\ 1\,1\,0\,1\,0 & (-6) \end{array} \rightarrow \begin{array}{r} 0\,1\,1\,0\,1 \\ \times\ 0\,\bar{1}\,1\,\bar{1}\,0 \end{array}$$

Note: two's complement of 13: 10011

$$\begin{array}{r} 1\,0\,0\,0\,0\,0\,0 \\ 0\,1\,0\,0\,1\,1 \\ 1\,0\,1\,1\,0\,1 \\ 0\,1\,0\,0\,1\,1 \\ 1\,0\,0\,0\,0\,0 \\ \hline 1\,1\,1\,0\,1\,1\,0\,0\,1\,0 \end{array} \quad (-79)$$

Note: each partial product is M, or -1*M, or 0.

Note: partial products are sign-extended.

Note: Booth recoding helps us reduce a multiply by a constant to a few shifts and adds. This used to be done by compilers, called "strength reduction", because multiply instructions were very slow (or didn't exist).

Note: $-1*M = \bar{M}+1$, but +1 is costly (full add!) we can utilize the "carry-in" of the array to get the +1 for free!

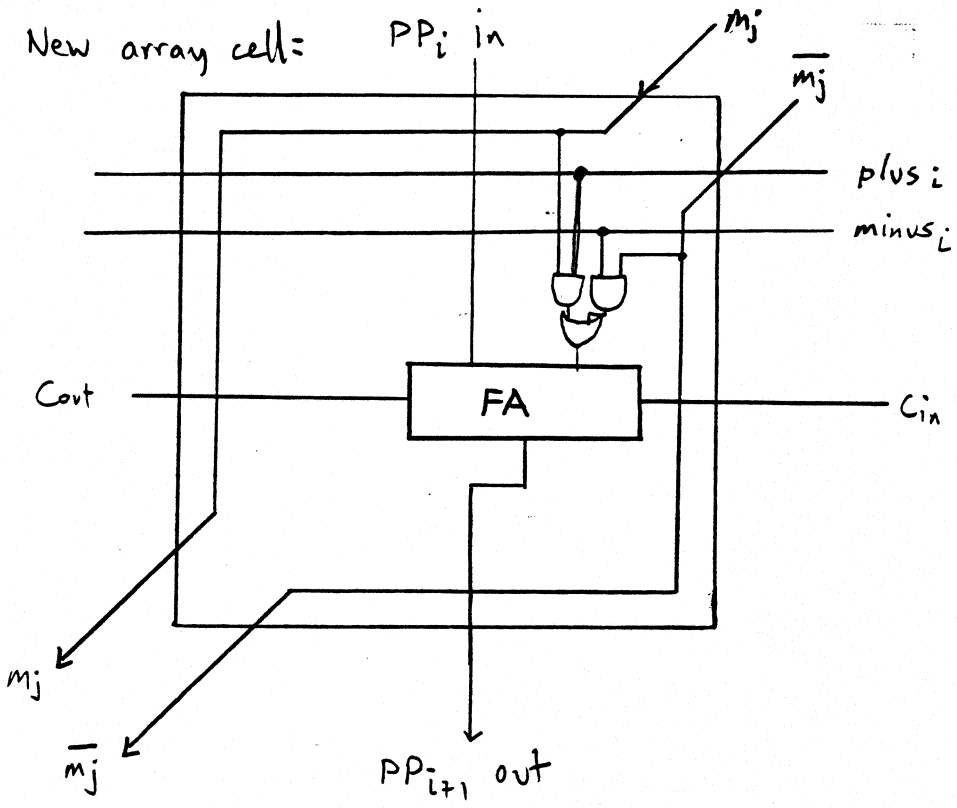
∴ distribute M, $\bar{M}$ and set Cin if using $\bar{M}$

Q 1 1 0 1 0 (-6)

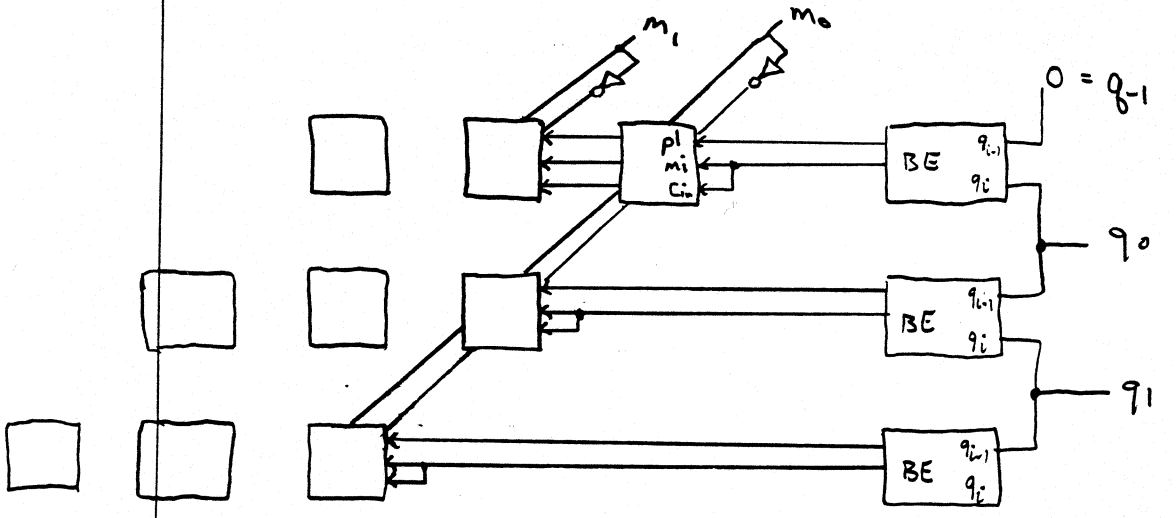
Q' 0 $\bar{1}$ 1 $\bar{1}$ 0

q_i	q_{i-1}	q_i'
0	0	0
0	1	1
1	0	$\bar{1}$
1	1	0

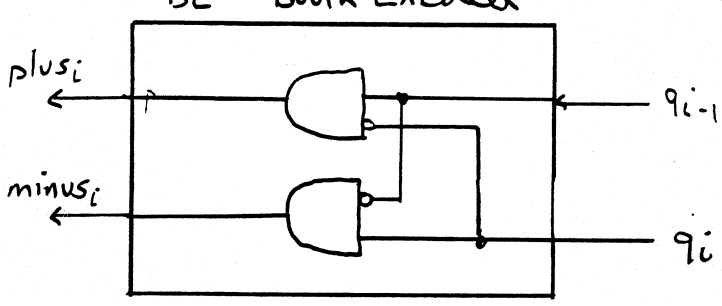
New array cell: PP_i in



1 → add
T → subtract
0 → do nothing



BE Booth Encoder



Fast Multiplication (CO 6.5)

- there are many techniques to improve delay and even area of the array multiplier
- look at 2 key techniques:
  1. Booth recoding in radix-4 (bit-pair recoding)
  2. carry save addition (Wallace trees)

1. Booth recoding in radix-4 (previously, radix 2)

compute Q'' using digits $\{\bar{2}, \bar{1}, 0, 1, 2\}$ as follows:

q_{i+1}	q_i	q_{i-1}	q_{i+1}'	q_i'	q_i''
0	0	0	0	0	0
0	0	1	0	1	1
0	1	0	1	$\bar{1}$	1
0	1	1	1	0	2
1	0	0	$\bar{1}$	0	$\bar{2}$
1	0	1	$\bar{1}$	1	$\bar{1}$
1	1	0	0	$\bar{1}$	$\bar{1}$
1	1	1	0	0	0

q_{msbs} = sign-extended

$q_{-1} = 0$

method: replace $q_{i+1}\ q_i$ (bit pair) with q_i''

Example $\underbrace{1}_{\text{sgn-ext}}\ 1\ 1\ 0\ 1\ 0\ \underbrace{0}_{q_{-1}}$ (−6)

Q' $\quad 0\ 0\ \bar{1}\ 1\ \bar{1}\ 0$ $\quad (-8+4-2 = -6)$

Q'' $\quad 0\ \ \bar{1}\ \ \bar{2}$ $\quad (0-4-2 = -6)$

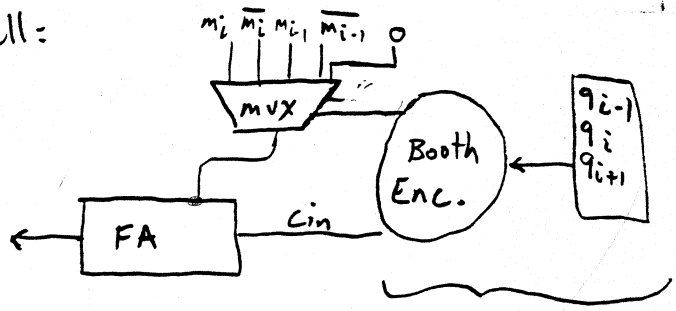
$\quad\ \ \times 2^4\ \times 2^2\ \times 2^0$

- To multiply by Q'', we form partial products $\{\bar{2}*M, \bar{1}*M, 2*M, M, 0\}$ and add them like before.
- Recall $\bar{2}*M = -2*M = 2*(-M) = 2*(\underbrace{\bar{M}+1}_{\text{two's complement of } M})$

and 2* anything is just a shift-left by 1 bit.

Key result: replacing bit-pairs in multiplicand ($Q \to Q'$) cuts # partial products in half

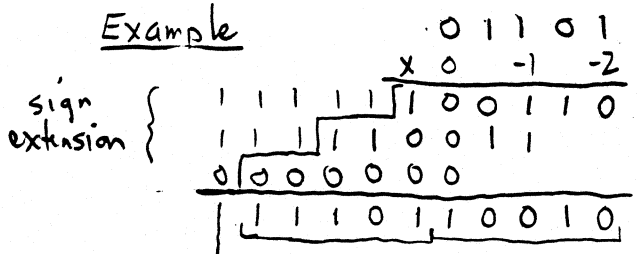
Array cell:



q_i''	mux	Cin
1	x_i	0
2	x_{i-1}	0
$\bar{2}$	$\bar{x}_{i-1}$	1
$\bar{1}$	$\bar{x}_i$	1
0	0	0

shared at right edge of array

Example



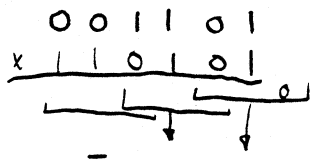
(13) 5 bits
(-6) 5 bits
6 bits

(-78) 10 bits (=5+5)

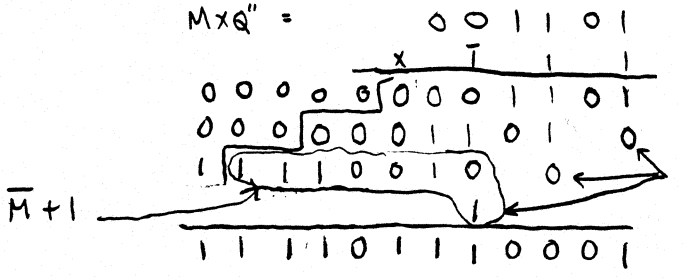
$\bar{2}\cdot M$ = 100110 6bits
$\bar{1}\cdot M$ = 110011 6bits

↳ here, we ignored these last bit because operands have only 5 bits ⇒ 10 bit product

Example



(13) M
(-11) Q
(-16+4+1 = -11) Q''



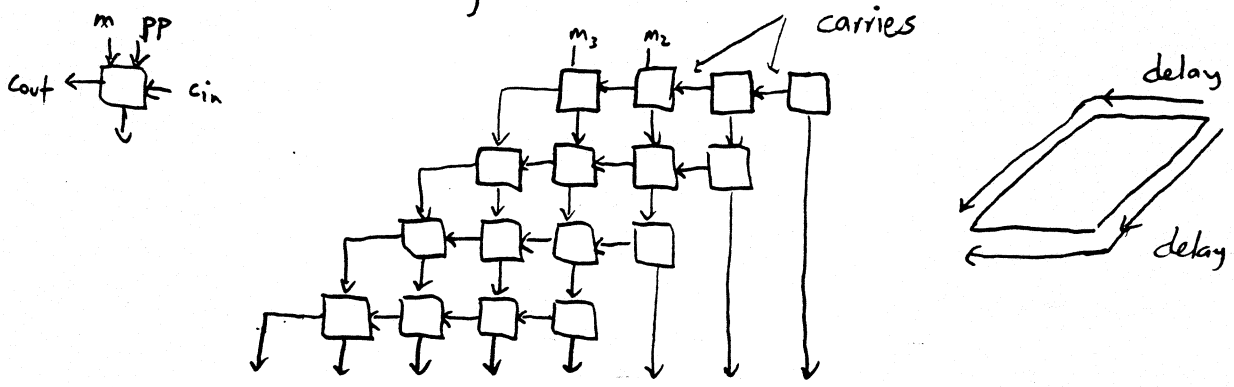
optimization: for $\{\bar{2}\cdot M = 2(\bar{M}+1)$, $\bar{1}\cdot M = \bar{M}+1\}$ use carry-ins for +1

(-143) 12-bits (=6+6)

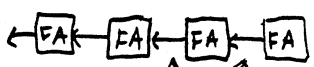
Notice, sign extension is painful again.

② Carry-Save Addition

- consider full array:



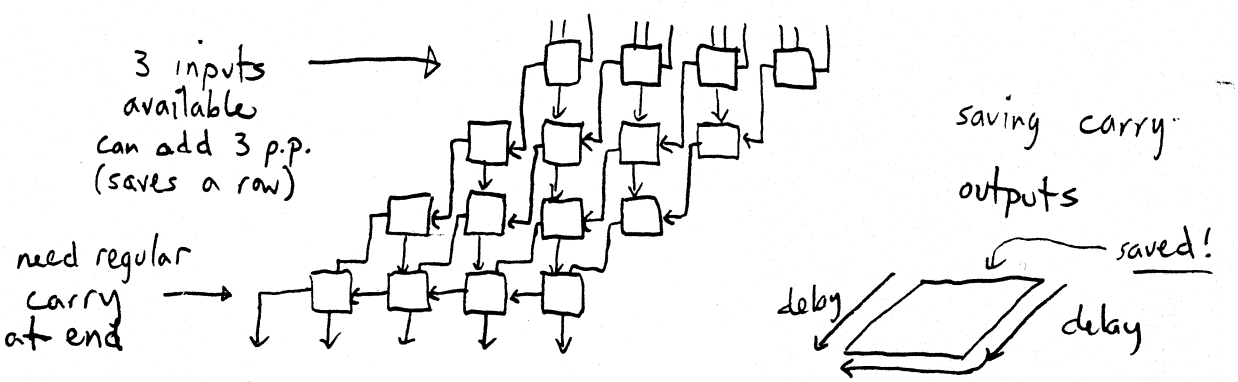
- every row contains a ripple-carry adder:



uses carry chain

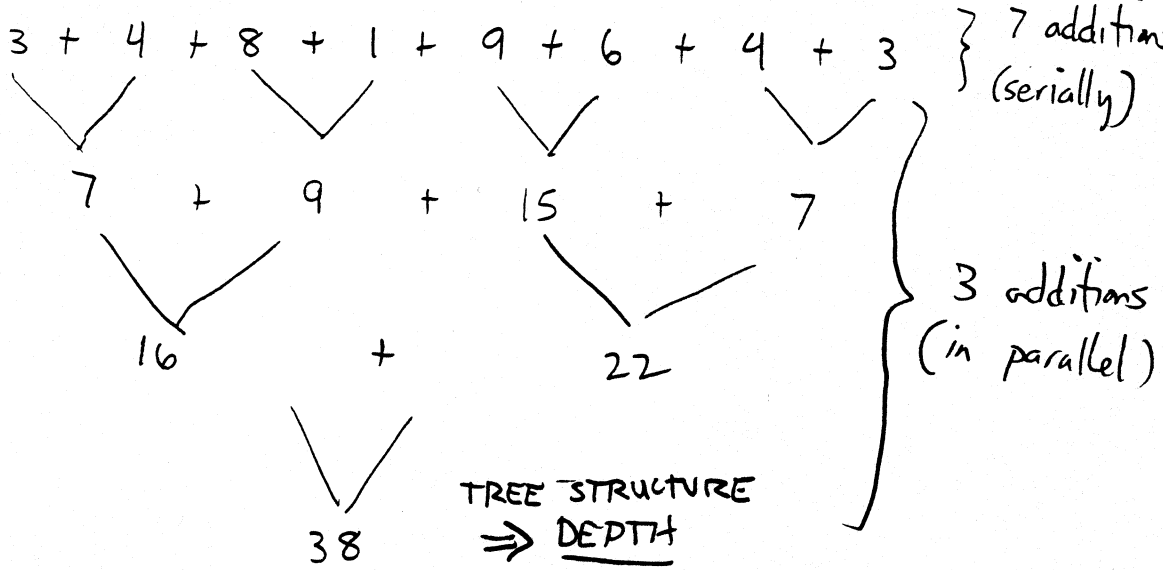
- instead of rippling-left, we can save the carry by sending it downward

note: carry-out is still sent left 1 column, since it has higher weight or magnitude ($\times 2^i$)

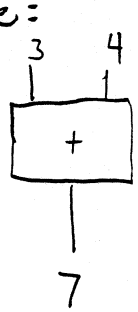


- this saves a bit, but we can do better

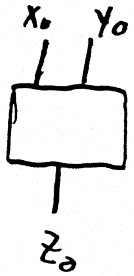
How to add 8 numbers quickly?



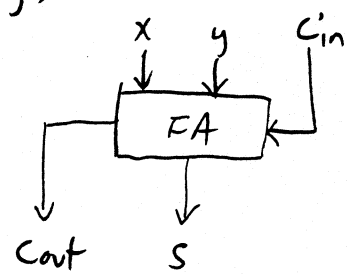
Notice structure:



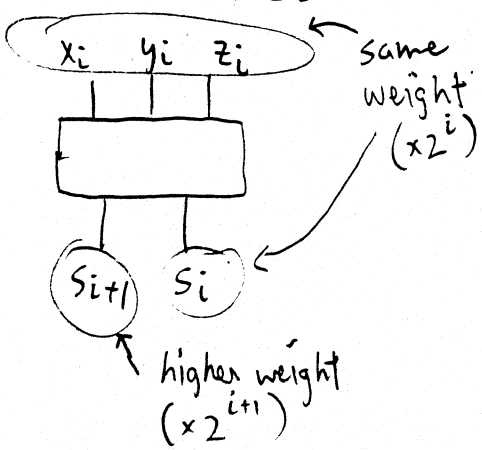
a 2:1 compressor (no carry!)



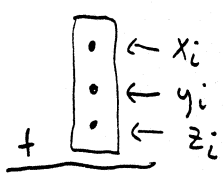
In binary, we have FA, a 3:2 compressor

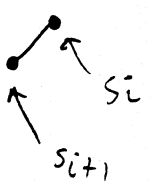


⇒ apply carry-save

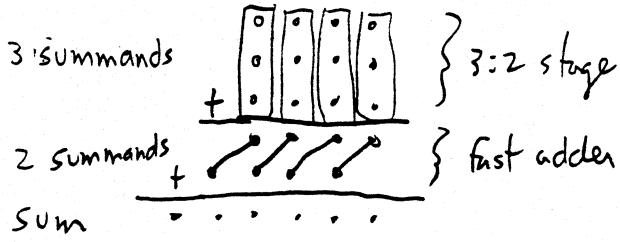


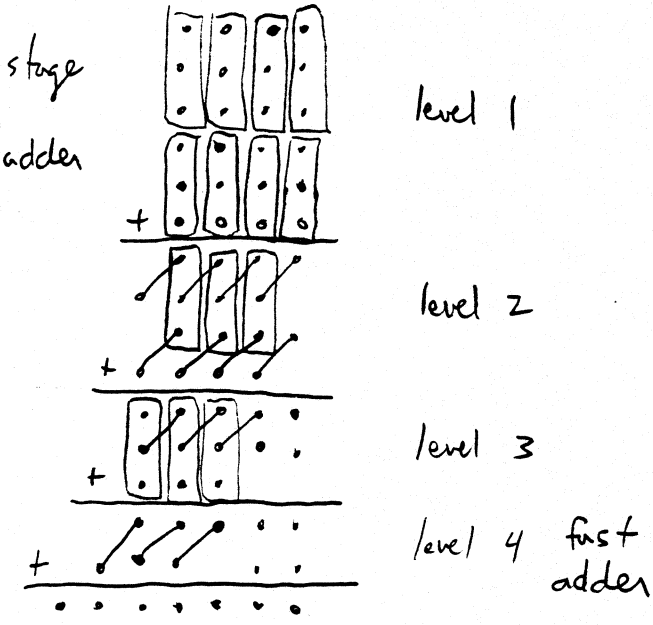
dot notation





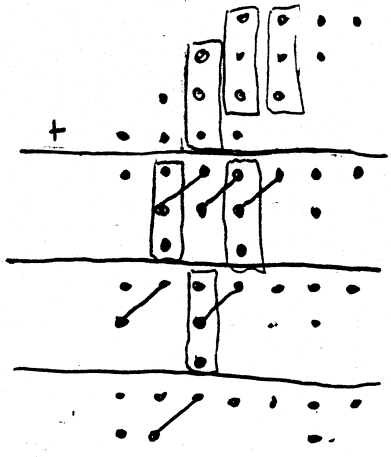
Carry-save addition of 4-bit numbers:





- reduces # levels if adding many #s : $O(\log n)$
- apply to product term array: WALLACE TREE MULTIPLIER

unsigned 4b x 4b mult:



level 1

level 2

Level 3

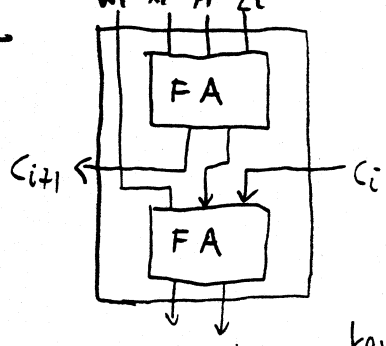
level 4 : fast adder

depth reduction ?

need <u>bigger operands</u> to save depth

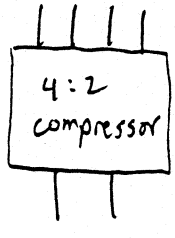
- can use different compressors, eg: 4:2

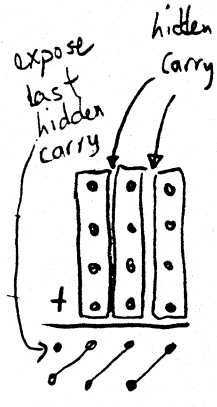
<u>4 inputs</u>



"hidden" carry bits

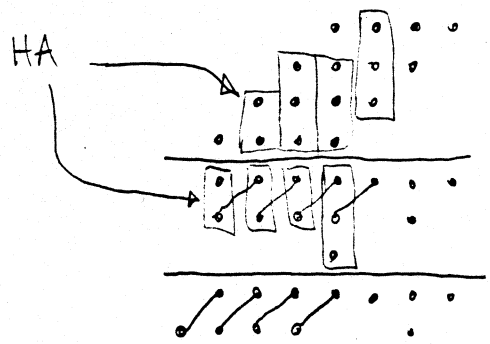
<u>2 outputs</u>





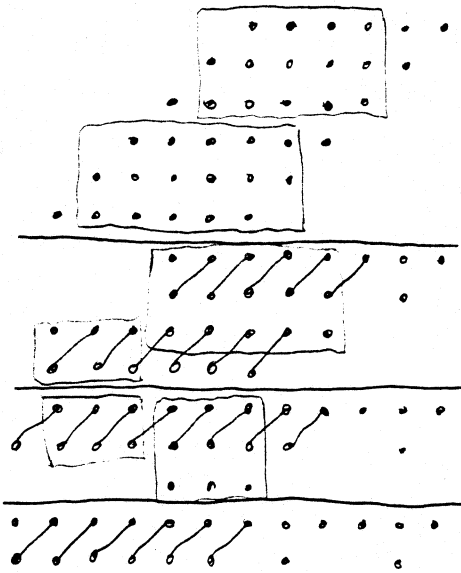
key: C_i does not affect C_{i+1}

- unsigned 46 x 46 mult
  - using FA and HA to save one level



note: always use HA to reduce bit pairs on left side

- unsigned 66 x 66 mult using only 1 more level



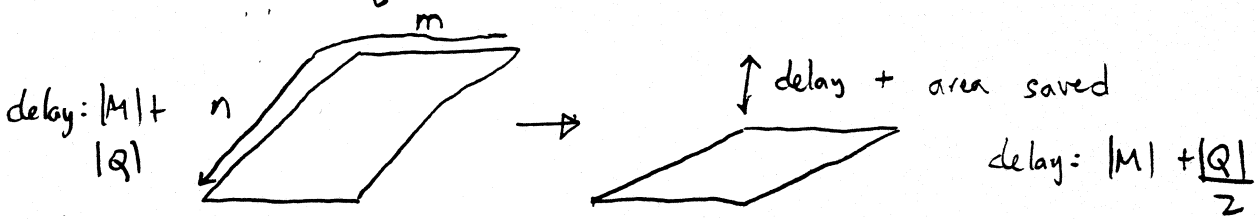
- tree height n levels $\Leftrightarrow$ $\left(\frac{3}{2}\right)^n$ inputs (# partial products)

$\log_{3/2}(n)$ levels $\Leftrightarrow$ $\underbrace{n \text{ inputs}}_{|Q|}$

- vertical delay $\sim O(\log |Q|)$
- last row : fast adder, horiz. delay $\sim O(\log |M|)$

Summary - Fast Multiplies

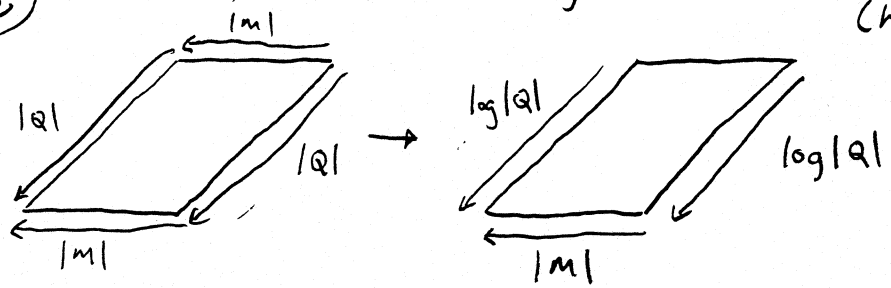
Booth encoding → reduce # partial products ($\frac{1}{2}$) w/ radix 4



Carry-save addition (Wallace tree)

tree → reduce # levels (effective height) for large # of partial products (vertical delay)

carry-save → eliminate ripple-carry in all rows (but last) (horizontal delay)



Fast adder in last row |M| → log|M|

∴ final delay ~ log|M| + log|Q|

compared to |M| + |Q| originally

→ can combine Booth radix4 and Wallace

Sign Extension Optimization Details

look carefully at sign extension in the array:

z_{03} z_{03} z_{03} z_{03} z_{03} z_{02} z_{01} z_{00}

z_{13} z_{13} z_{13} z_{13} z_{12} z_{11} z_{10}

z_{23} z_{23} z_{23} z_{22} z_{21} z_{20}

z_{33} z_{33} z_{32} z_{31} z_{30}

look at left side

z_{03} z_{03} z_{03} z_{03} 0 = 1 1 1 1 0 + 0 0 0 $\bar{z}_{03}$ 0

z_{13} z_{13} z_{13} 0 0 = 1 1 1 0 0 + 0 0 $\bar{z}_{13}$ 0 0

z_{23} z_{23} 0 0 0 = 1 1 0 0 0 + 0 $\bar{z}_{23}$ 0 0 0

z_{33} 0 0 0 0 = 1 0 0 0 0 + $\bar{z}_{33}$ 0 0 0 0

0 0 0 1 0 + $\bar{z}_{33}$ $\bar{z}_{23}$ $\bar{z}_{13}$ $\bar{z}_{03}$ 0

new array:

$\bar{z}_{03}$ z_{03} z_{02} z_{01} z_{00}

$\bar{z}_{13}$ z_{13} z_{12} z_{11} z_{10}

$\bar{z}_{23}$ z_{23} z_{22} z_{21} z_{20}

$\bar{z}_{33}$ z_{33} z_{32} z_{31} z_{30}

0 0 0 1 0

ugly to have entire row for just one '1', notice:

$$\begin{array}{r} \bar{z}_{03} \\ +\ 1 \\ \hline \bar{z}_{03}\, z_{03} \end{array}$$

FINAL ARRAY

$\bar{z}_{03}$ z_{03} z_{03} z_{02} z_{01} z_{00}

$\bar{z}_{13}$ z_{13} z_{12} z_{11} z_{10}

$\bar{z}_{23}$ z_{23} z_{22} z_{21} z_{20}

$\bar{z}_{33}$ z_{33} z_{32} z_{31} z_{30}

Example:

1 0 0 1 1 (-13)

× 0 1 0 1 1 (11)

0 1 1 0 0 1 1

0 1 0 0 1 1

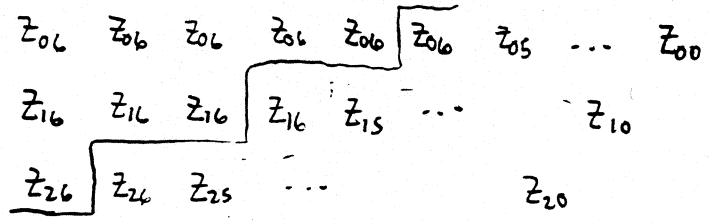
1 0 0 0 0 0

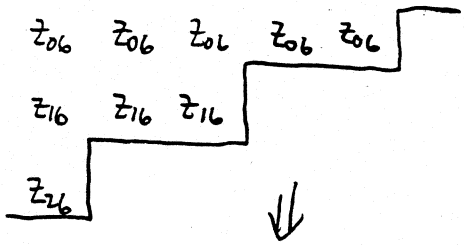
0 1 0 0 1 1

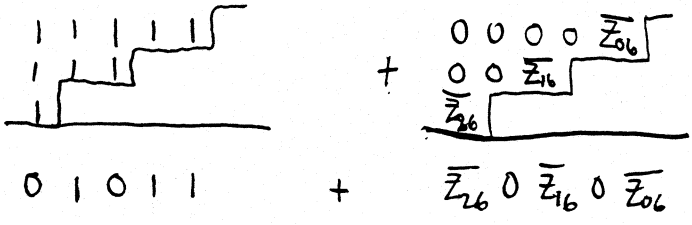
1 0 0 0 0 0

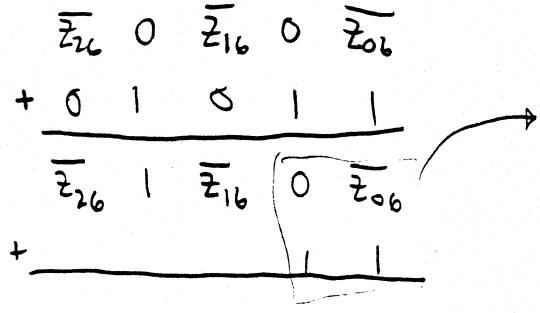
1 1 0 1 1 1 0 0 0 1 (-143)

Sign extension optimization w/ Booth radix 4 (optional)



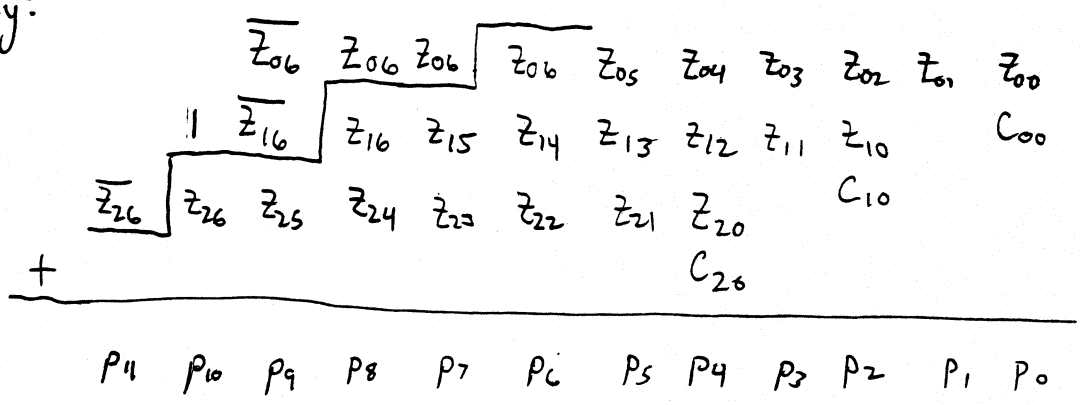






final array:

sign extension still easy!



Example