# Multiplier

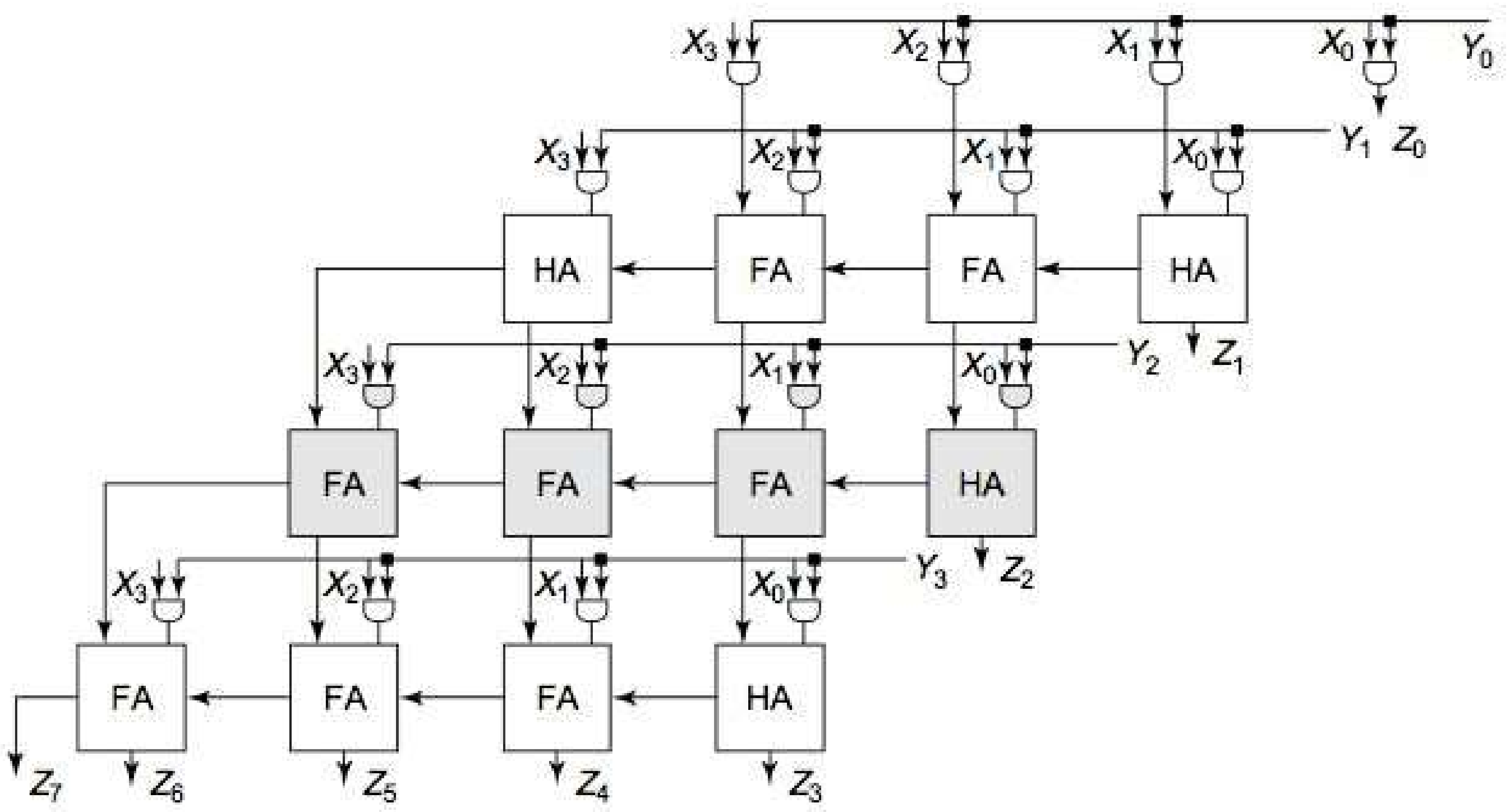
- Multiplicand: Y = $(y_{M-1}, y_{M-2}, \ldots, y_1, y_0)$
- Multiplier: X = $(x_{N-1}, x_{N-2}, \ldots, x_1, x_0)$
- Product:

$$P = \left(\sum_{j=0}^{M-1} y_j 2^j\right)\left(\sum_{i=0}^{N-1} x_i 2^i\right) = \sum_{i=0}^{N-1}\sum_{j=0}^{M-1} x_i y_j 2^{i+j}$$

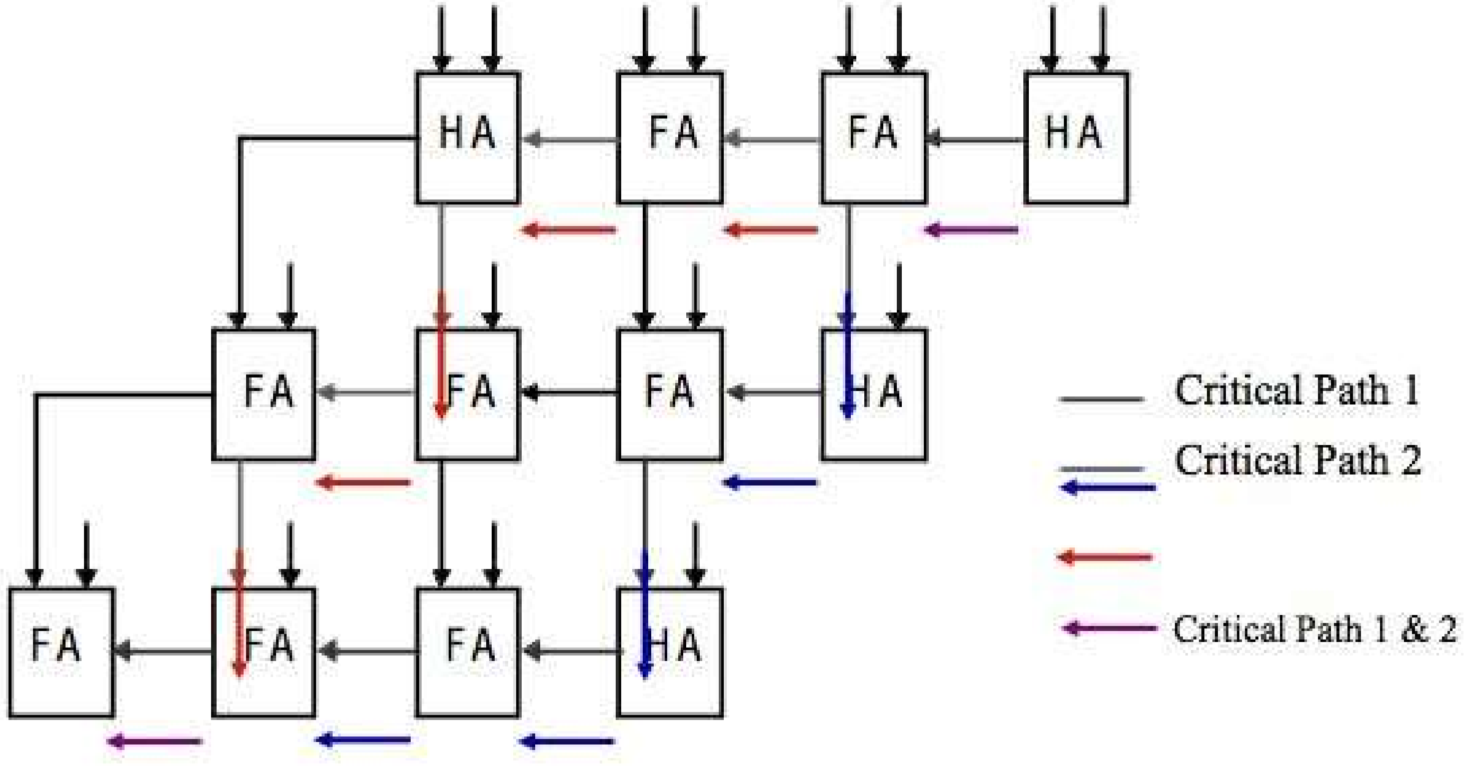
$$\begin{array}{cccccccccccc|l}
 & & & & & & y_5 & y_4 & y_3 & y_2 & y_1 & y_0 & \text{multiplicand} \\
 & & & & & & x_5 & x_4 & x_3 & x_2 & x_1 & x_0 & \text{multiplier} \\
\cline{7-12}
 & & & & & & x_0y_5 & x_0y_4 & x_0y_3 & x_0y_2 & x_0y_1 & x_0y_0 & \\
 & & & & & x_1y_5 & x_1y_4 & x_1y_3 & x_1y_2 & x_1y_1 & x_1y_0 & & \\
 & & & & x_2y_5 & x_2y_4 & x_2y_3 & x_2y_2 & x_2y_1 & x_2y_0 & & & \text{partial products} \\
 & & & x_3y_5 & x_3y_4 & x_3y_3 & x_3y_2 & x_3y_1 & x_3y_0 & & & & \\
 & & x_4y_5 & x_4y_4 & x_4y_3 & x_4y_2 & x_4y_1 & x_4y_0 & & & & & \\
 & x_5y_5 & x_5y_4 & x_5y_3 & x_5y_2 & x_5y_1 & x_5y_0 & & & & & & \\
\hline
p_{11} & p_{10} & p_9 & p_8 & p_7 & p_6 & p_5 & p_4 & p_3 & p_2 & p_1 & p_0 & \text{product}
\end{array}$$

$$
\begin{array}{cccccccccc|l}
 & & & & 1 & 0 & 1 & 0 & 1 & 0 & \text{Multiplicand} \\
\times & & & & & & 1 & 0 & 1 & 1 & \text{Multiplier} \\
\hline
 & & & & 1 & 0 & 1 & 0 & 1 & 0 & \\
 & & & 1 & 0 & 1 & 0 & 1 & 0 & & \\
 & & 0 & 0 & 0 & 0 & 0 & 0 & & & \text{Partial products} \\
+ & 1 & 0 & 1 & 0 & 1 & 0 & & & & \\
\hline
1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & & \text{Result}
\end{array}
$$

# Array Multiplier



# Critical path



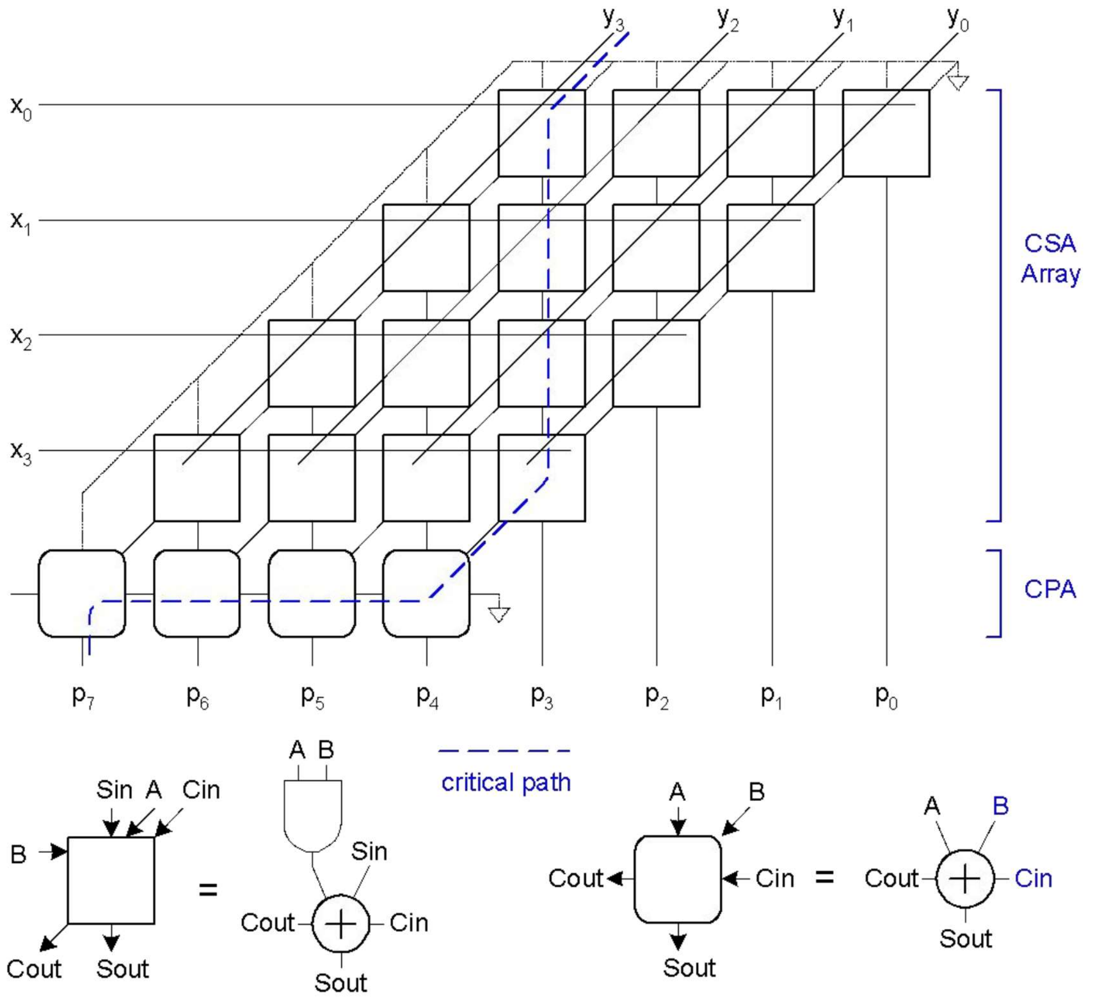
$$t_{mult} \approx [(M-1)+(N-2)]t_{carry}+(N-1)t_{sum}+(N-1)t_{and}$$

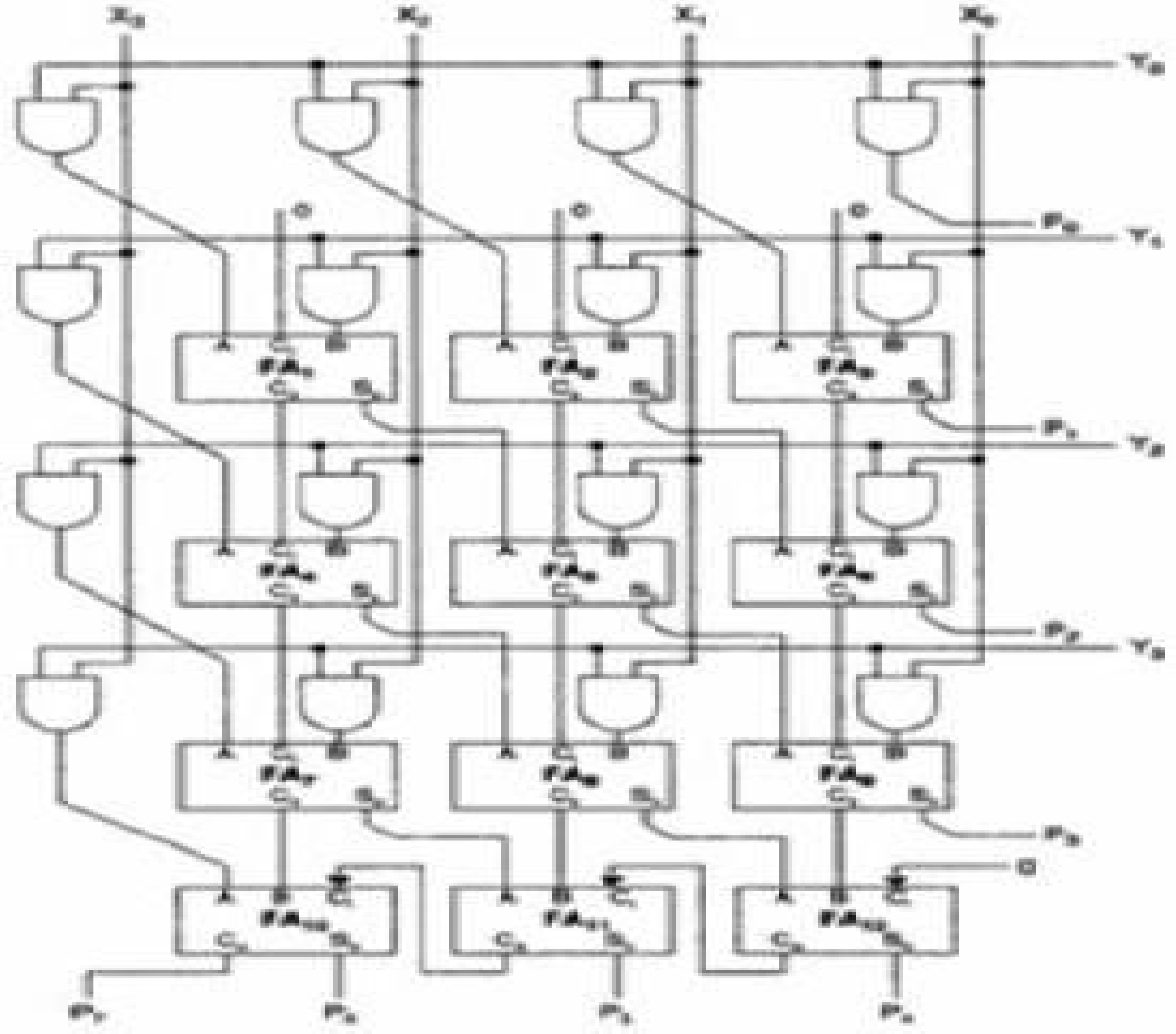
## Multiplication Algorithm

- Generation of partial products
- Adding up partial products
  - Sequentially (sequential shift and add)
  - Serially (combinational shift and add)
  - In parallel
- Speed-up techniques
  - Reduce the number of partial products
  - Accelerate addition of partial products

# BRAUN Multiplier

- Consists of an array of AND gates and CSA arranged in an iterative structure and a final CPA
- Does not require any logic registers
- Restricted to multiplication of two unsigned numbers




X3
X2
X1
X0
Y0
Y1
Y2
Y3
0
0
0
0
P0
P1
P2
P3
A
Ci
B
F.A
Co
S
P7
P6
P5
P4

- Performs well for unsigned operands that are less than 16 bits in terms of speed, power and area
- Simple and regular structure as compared to the other multipliers
- No. of components required increases quadratically with the no. of bits

- The delay of the multiplier is given by

$$t_{mult} = (N-1)t_{carry} + (N-1)t_{and} + t_{merge}$$

## Booth’s Multiplier

- Multiplies two signed binary numbers in two's compliment notation.
- Uses shifting than adding to increase their speed.

- Recall grade school trick
  When multiplying by 9:
  - Multiply by 10 (easy, just shift digits left)
  - Subtract once
- Booth’s algorithm applies same principle

## When using Booth's Algorithm:

- You will need twice as many bits in your **product** as you have in your original two **operands**.
- The **leftmost bit** of your operands (both your multiplicand and multiplier) is a SIGN bit, and cannot be used as part of the value.
- Decide which operand will be the **multiplier** and which will be the **multiplicand**
- Convert both operands to **two's complement** representation using X no. of bits

- X must be at least one more bit than is required for the binary representation of the numerically larger operand

- Begin with a product that consists of the multiplier with an additional 'X no. of leading zero bits'.

- For our example, let's multiply (**-5) x 2**

  The numerically larger operand (5) would require 3 bits to represent in binary (101). So we must use AT LEAST 4 bits to represent the operands, to allow for the sign bit.

- Let's use 5-bit 2's complement

-5 is 11011 (multiplier)

2 is 00010 (multiplicand)

- The multiplier is:
  `11011(-5 in 2’s compliment)`

- Add 5 leading zeros to the **multiplier** to get the **beginning product**:
  `00000 11011`

**STEP 1**

- Use the **LSB** and the **previous LSB** to determine the arithmetic action.
  If it is the FIRST pass, use 0 as the previous LSB.

- Possible arithmetic actions:

| | | |
|---|---|---|
| 00 | □ | no arithmetic operation |
| 01 | □ | add multiplicand to left half of product |
| 10 | □ | subtract multiplicand from left half of product |
| 11 | □ | no arithmetic operation |

## STEP 2

- Perform an **arithmetic right shift** (ASR) on the entire product.

- NOTE: For X-bit operands, Booth's algorithm requires X passes.

- Initial Product and previous LSB

```
    00000 11011 0
```

- (Note: Since this is the first pass, we use 0 for the previous LSB)

- Pass 1, Step 1: Examine the last 2 bits

```
    00000 11011 0
```

- The last two bits are 10, so we need to subtract the **multiplicand** from left half of product

Pass 1, Step 1: Arithmetic action

```
(1)  00000  (left half of product)
    -00010      (mulitplicand)
     11110    (uses a phantom borrow)
```

Place result into **left half** of product

```
        11110 11011
```

Step 2: ASR (arithmetic shift right)

Before ASR

```
        11110 11011
```

After ASR

```
        11111 01101
```

(left-most bit was 1, so a 1 was shifted in on the left)

Pass 1 is complete.

- Current Product and previous LSB
  `11111 01101 1`

- Pass 2, Step 1: Examine the last 2 bits
  `11111 01101 1`
- The last two bits are 11, so we do NOT need to perform an arithmetic action just proceed to step 2.
- Step 2: ASR (arithmetic shift right)
  Before ASR
  `11111 01101 1`
  After ASR
  `11111 10110 1`
  (left-most bit was 1, so a 1 was shifted in on the left)

- Pass 2 is complete.

- Current Product and previous LSB

  `11111 10110 1`

- Pass 3, Step 1: Examine the last 2 bits

  `11111 10110 1`

  The last two bits are 01, so we need to add the **multiplicand** to the left half of the product

Arithmetic action

```
(1)  11111  (left half of product)
    +00010      (mulitplicand)
     00001   (drop the leftmost carry)
```

Place result into **left half** of product

```
00001 10110 1
```

- Step 2: ASR (arithmetic shift right)

  Before ASR

  ```
          00001 10110 1
  ```

  After ASR

  ```
          00000 11011 0
  ```

  (left-most bit was 0, so a 0 was shifted in on the left)

- Pass 3 is complete.

## PASS 4

- Current Product and previous LSB

  ```
  00000 11011 0
  ```

- Step 1: Examine the last 2 bits

  ```
  00000 11011 0
  ```

- The last two bits are 10, so we need to subtract the **multiplicand** from the left half of the product

- Step 1: Arithmetic action

```
(1)  00000   (left half of product)
    -00010   (mulitplicand)
     11110    (uses a phantom borrow)
```

- Place result into **left half** of product

```
11110 11011 0
```

- Step 2: ASR (arithmetic shift right)

  Before ASR

  ```
          11110 11011 0
  ```

  After ASR

  ```
          11111 01101 1
  ```

  (left-most bit was 1, so a 1 was shifted in on the left)

- Pass 4 is complete.

- Current Product and previous LSB

  `11111 01101 1`
- Pass 5, Step 1: Examine the last 2 bits

  `11111 01101 1`
- The last two bits are 11, so we do NOT need to perform an arithmetic action just proceed to step 2.
- Step 2: ASR (arithmetic shift right)

  Before ASR

  `11111 01101 1`

  After ASR

  `11111 10110 1`

  (left-most bit was 1, so a 1 was shifted in on the left)

- Pass 5 is complete.

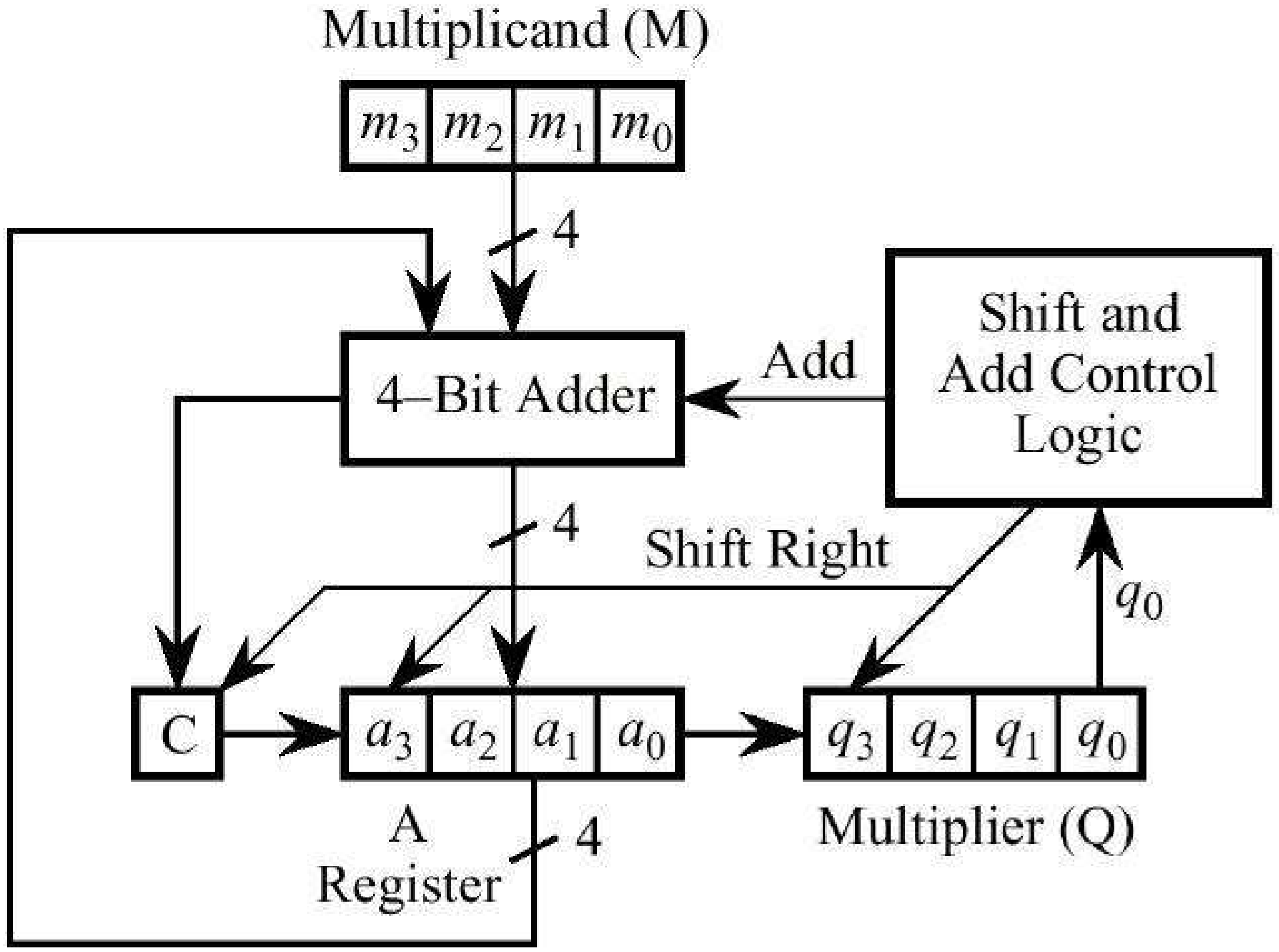
- Dropping the previous LSB, the resulting **final product** is:

  `11111 10110`

- To confirm we have the correct answer, convert the 2's complement **final product** back to decimal.
- Final product:  `11111 10110`
- Decimal value:  `-10`

  which is the CORRECT product of:

  `(-5) x 2`


Multiplicand (M)
$m_3$ $m_2$ $m_1$ $m_0$
4
4-Bit Adder
Add
Shift and Add Control Logic
4
Shift Right
$q_0$
C
$a_3$ $a_2$ $a_1$ $a_0$
$q_3$ $q_2$ $q_1$ $q_0$
A Register
4
Multiplier (Q)

## Modified Booth Multiplier (Booth Encoding)

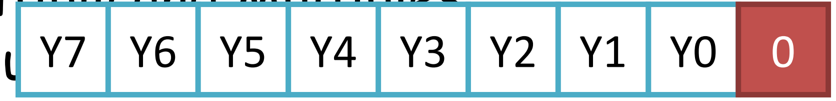
- Can encode the digits by looking at three bits at a time
- Booth recoding table:

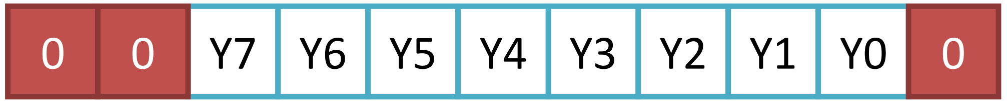
| i+1 | i | i-1 | add |
|---|---|---|---|
| 0 | 0 | 0 | 0*M |
| 0 | 0 | 1 | 1*M |
| 0 | 1 | 0 | 1*M |
| 0 | 1 | 1 | 2*M |
| 1 | 0 | 0 | –2*M |
| 1 | 0 | 1 | –1*M |
| 1 | 1 | 0 | –1*M |
| 1 | 1 | 1 | 0*M |

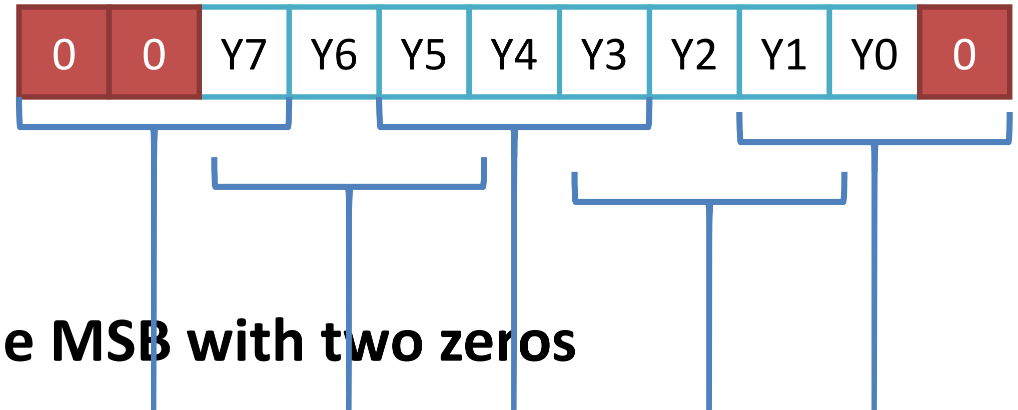
  – Must be able to add *multiplicand* times –2, -1, 0, 1 and 2

## Algorithm

1. Pad the LSB with one zero.
2. Pad the MSB with 2 zeros if n is even and 1 zero if n is odd.
3. Divide the multiplier into overlapping groups of 3-bits.
4. Determine partial product scale factor from modified booth 2 encoding table.
5. Compute the Multiplicand Multiples
6. Sum Partial Produ



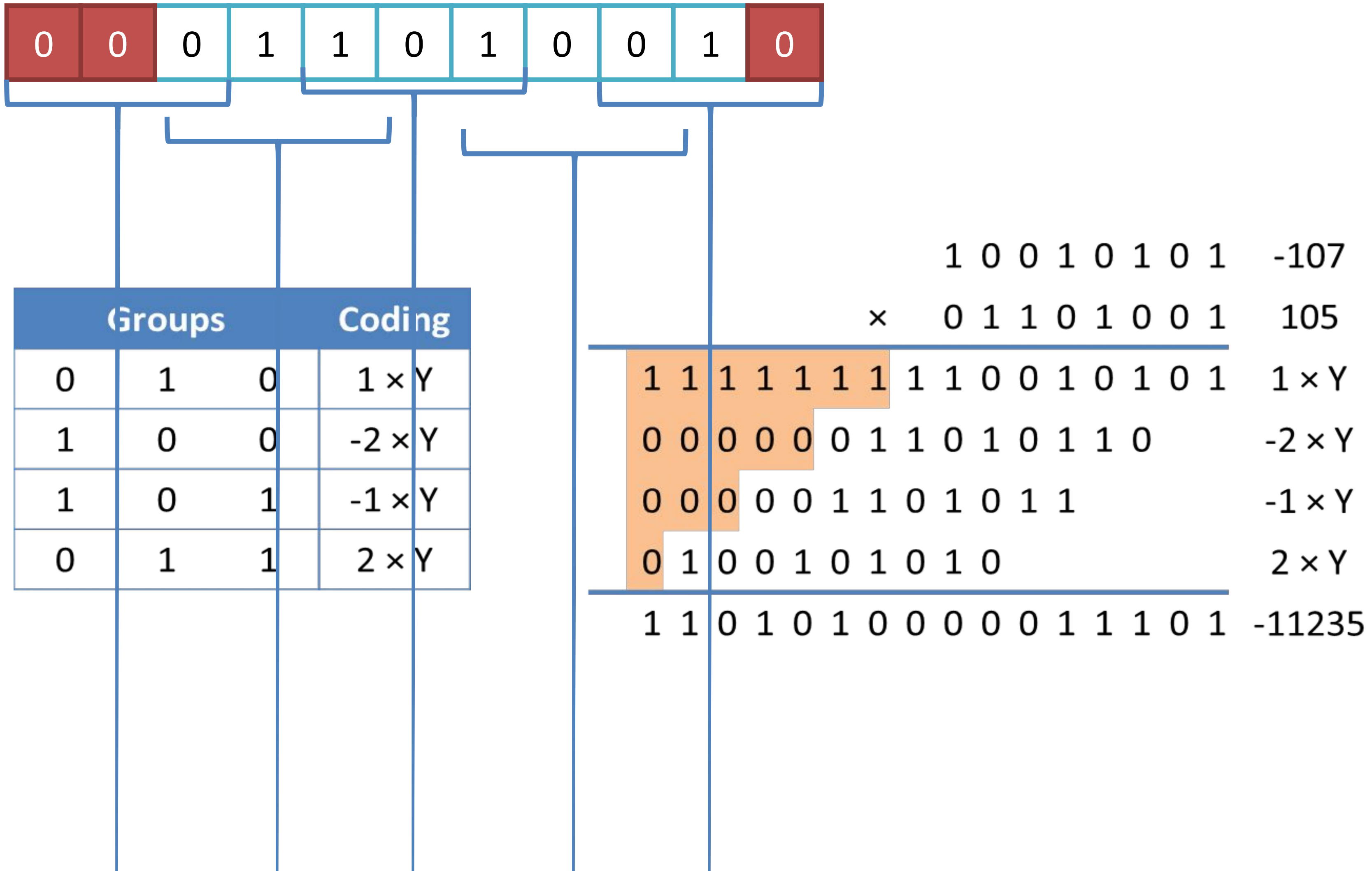




**1.Pad LSB with 1 zero**

**2.n is even then pad the MSB with two zeros**

## Example



| Groups | | | Coding |
|---|---|---|---|
| 0 | 1 | 0 | 1 × Y |
| 1 | 0 | 0 | -2 × Y |
| 1 | 0 | 1 | -1 × Y |
| 0 | 1 | 1 | 2 × Y |

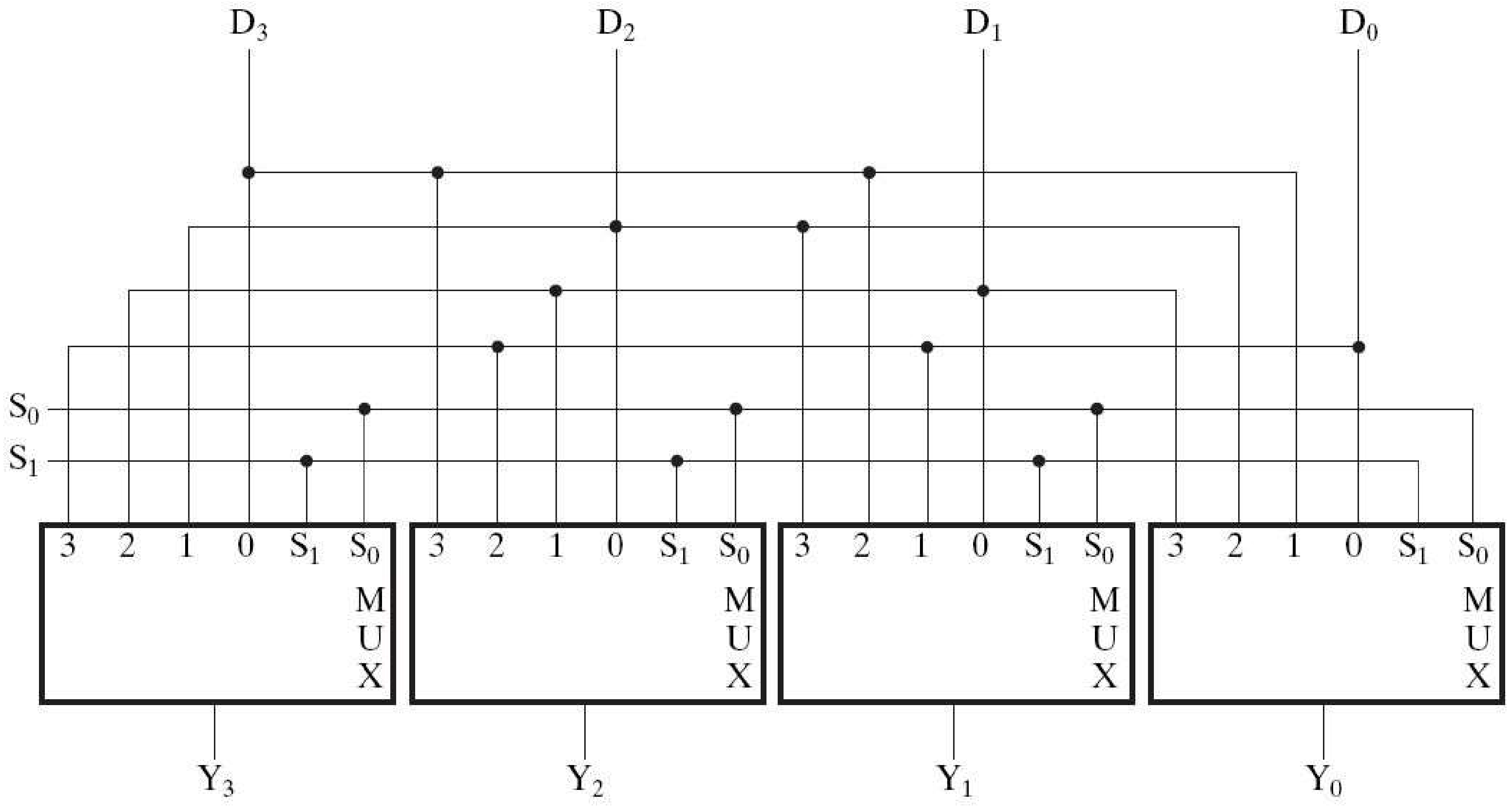
```
                        1 0 0 1 0 1 0 1     -107
                  ×     0 1 1 0 1 0 0 1      105
  -------------------------------------
  1 1 1 1 1 1 1 1 1 0 0 1 0 1 0 1          1 × Y
  0 0 0 0 0 0 1 1 0 1 0 1 1 0             -2 × Y
  0 0 0 0 0 1 1 0 1 0 1 1                 -1 × Y
  0 1 0 0 1 0 1 0 1 0                      2 × Y
  -------------------------------------
  1 1 0 1 0 1 0 0 0 0 0 1 1 1 0 1     -11235
```

# Barrel Shifter

- A barrel shifter is a digital circuit that can shift a data word by a specified number of bits in one clock cycle.
- It can be implemented with multiplexers (MUX)

**Function Table for 4-Bit Barrel Shifter**

| Select | | Output | | | | |
|---|---|---|---|---|---|---|
| $S_1$ | $S_0$ | $Y_3$ | $Y_2$ | $Y_1$ | $Y_0$ | Operation |
| 0 | 0 | $D_3$ | $D_2$ | $D_1$ | $D_0$ | No rotation |
| 0 | 1 | $D_2$ | $D_1$ | $D_0$ | $D_3$ | Rotate one position |
| 1 | 0 | $D_1$ | $D_0$ | $D_3$ | $D_2$ | Rotate two positions |
| 1 | 1 | $D_0$ | $D_3$ | $D_2$ | $D_1$ | Rotate three positions |



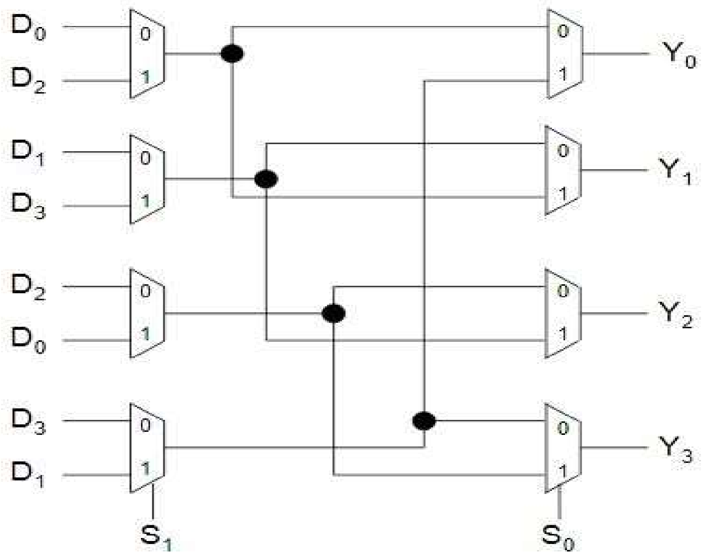
***i*** **positions of left rotation is the same as $2^n$-*i* bits of right rotation**

implementing a barrel shifter would be to use N (where N is the number of input bits) parallel N-to-1 multiplexers, one multiplexer that multiplexes all inputs into one of the outputs.
This would create an equivalent N*N to N multiplexer, which would use significant hardware resources (**multiplexer input grows in O(N$^2$).**

A more efficient implementation is possible by creating a hierarchy of multiplexers.

- 32-bit barrel shifter: can use 32 32-to-1multiplexers
- However, large fan-in undesirable. So, use layers of multiplexers

Example:
Use 2 layers of 4 2-to-1 multiplexers for 4-bit barrel shifter



For an 8 bit rotate component the multiplexers would be constructed as follows:

- the top level multiplexers rotate the data by 4 bits
- the second level rotates the data by 2 bits
- the third level rotates the data by 1 bit.

The number of multiplexers required is *n**log2(*n*), for an *n* bit word.
Four common word sizes and the number of multiplexers needed are:
64-bit — 64 * log2(64) = 64 * 6 = 384
32-bit — 32 * log2(32) = 32 * 5 = 160
16-bit — 16 * log2(16) = 16 * 4 = 64
8-bit — 8 * log2(8) = 8 * 3 = 24

```
module barrel_shifter(d,out,q,c); / Main module of 8-Bit Barrel shifter
  input [7:0]d;
  output [7:0]out,q;
  input[2:0]c;
  mux m1(q[0],d,c);
  mux m2(q[1],{d[0],d[7:1]},c);
  mux m3(q[2],{d[1:0],d[7:2]},c);
  mux m4(q[3],{d[2:0],d[7:3]},c);
  mux m5(q[4],{d[3:0],d[7:4]},c);
  mux m6(q[5],{d[4:0],d[7:5]},c);
  mux m7(q[6],{d[5:0],d[7:6]},c);
  mux m8(q[7],{d[6:0],d[7:7]},c);
  assign out=q;
endmodule
```

# Baugh-Wooley Multiplier

- Efficient way to handle sign bits
- Regular Multipliers, suited for 2's-compliment numbers
- Signed number representation

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

eg: $(1110)_2$

= (-2)

$$A = -a_{n-1}2^{n-1} + \sum_{i=0}^{n-2} a_i 2^i$$

$$B = -b_{n-1}2^{n-1} + \sum_{i=0}^{n-2} b_i 2^i$$

$$P = A \times B$$

$$= \left(-a_{n-1}2^{n-1} + \sum_{i=0}^{n-2} a_i 2^i\right) \times \left(-b_{n-1}2^{n-1} + \sum_{j=0}^{n-2} b_j 2^j\right)$$

$$= a_{n-1}b_{n-1}2^{2n-2} + \sum_{i=0}^{n-2}\sum_{j=0}^{n-2} a_i b_j 2^{i+j}$$

$$-2^{n-1}\sum_{i=0}^{n-2} a_i b_{n-1} 2^i - 2^{n-1}\sum_{j=0}^{n-2} a_{n-1} b_j 2^j$$

# The Baugh-Wooley Method and Its Modified Form

$$-a_4x_0 = a_4(1 - x_0) - a_4$$
$$= a_4x_0{'} - a_4$$

$$\begin{array}{ccc} -a_4 & a_4x_0{'} \\ \uparrow & a_4 \end{array}$$

In next column

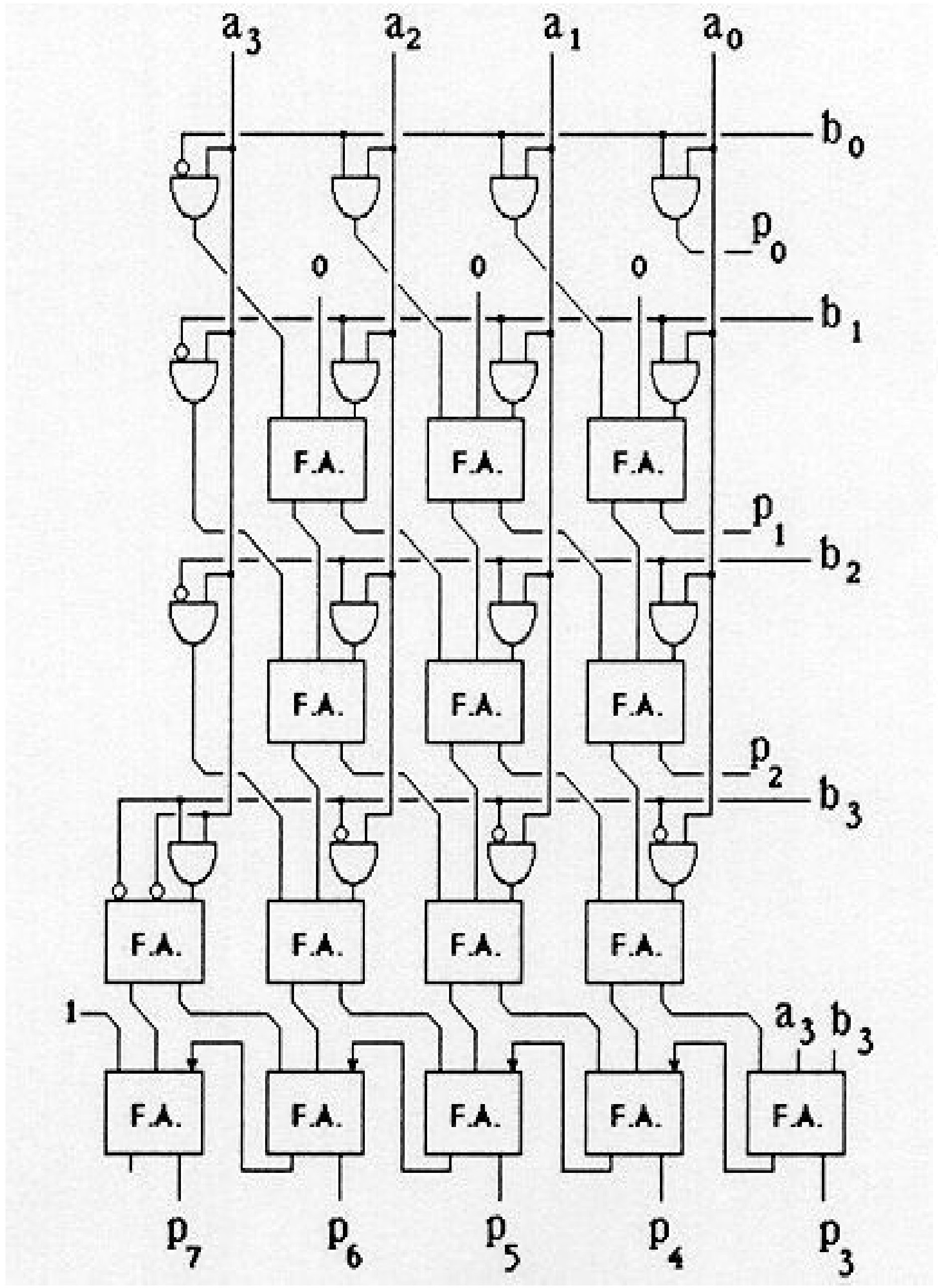
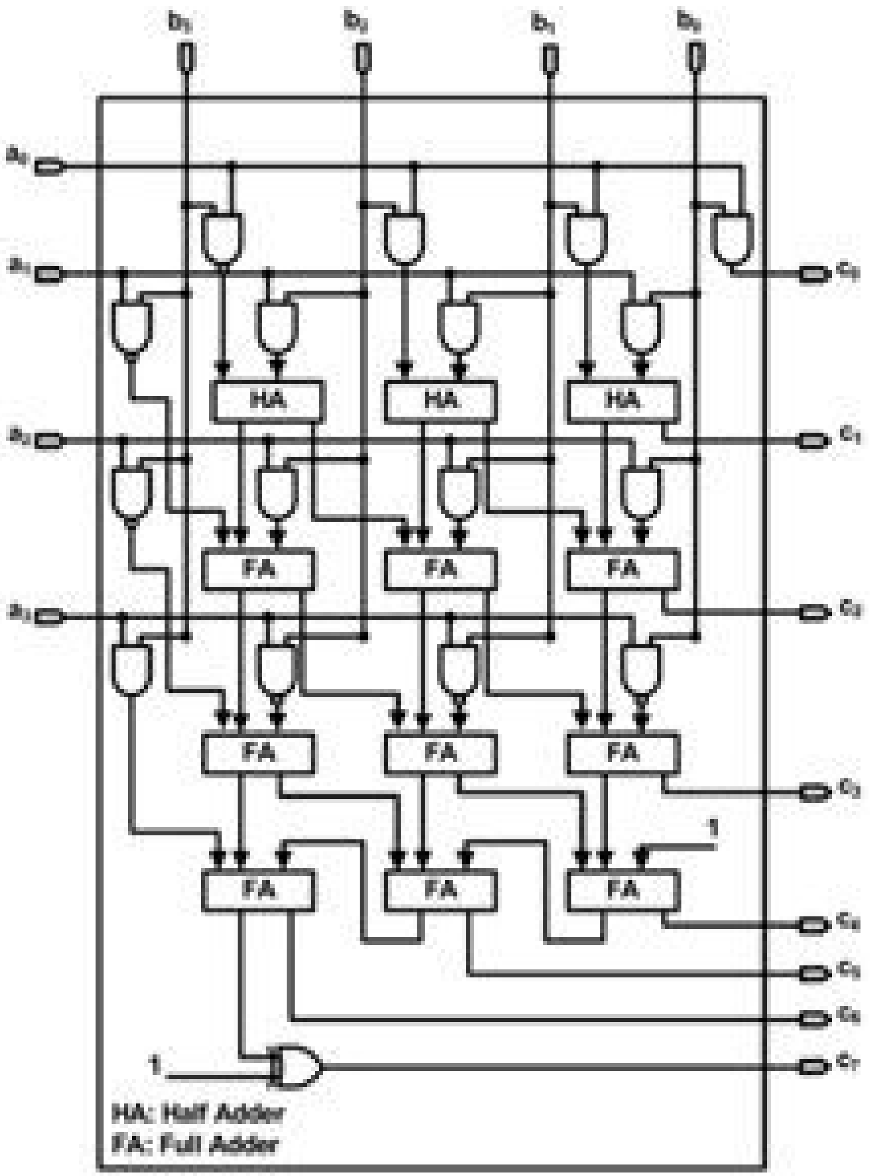
$$\begin{array}{cccccccccc}
 & & & & \times & a_4 & a_3 & a_2 & a_1 & a_0 \\
 & & & & & x_4 & x_3 & x_2 & x_1 & x_0 \\
\hline
 & & & & & a_4\bar{x}_0 & a_3x_0 & a_2x_0 & a_1x_0 & a_0x_0 \\
 & & & & a_4\bar{x}_1 & a_3x_1 & a_2x_1 & a_1x_1 & a_0x_1 & \\
 & & & a_4\bar{x}_2 & a_3x_2 & a_2x_2 & a_1x_2 & a_0x_2 & & \\
 & & a_4\bar{x}_3 & a_3x_3 & a_2x_3 & a_1x_3 & a_0x_3 & & & \\
 & a_4x_4 & \bar{a}_3x_4 & \bar{a}_2x_4 & \bar{a}_1x_4 & \bar{a}_0x_4 & & & & \\
 & \bar{a}_4 & & & & a_4 & & & & \\
1 & \bar{x}_4 & & & & x_4 & & & & \\
\hline
p_9 & p_8 & p_7 & p_6 & p_5 & p_4 & p_3 & p_2 & p_1 & p_0
\end{array}$$

$$-a_4x_0 = (1 - a_4x_0) - 1$$
$$= (a_4x_0)' - 1$$

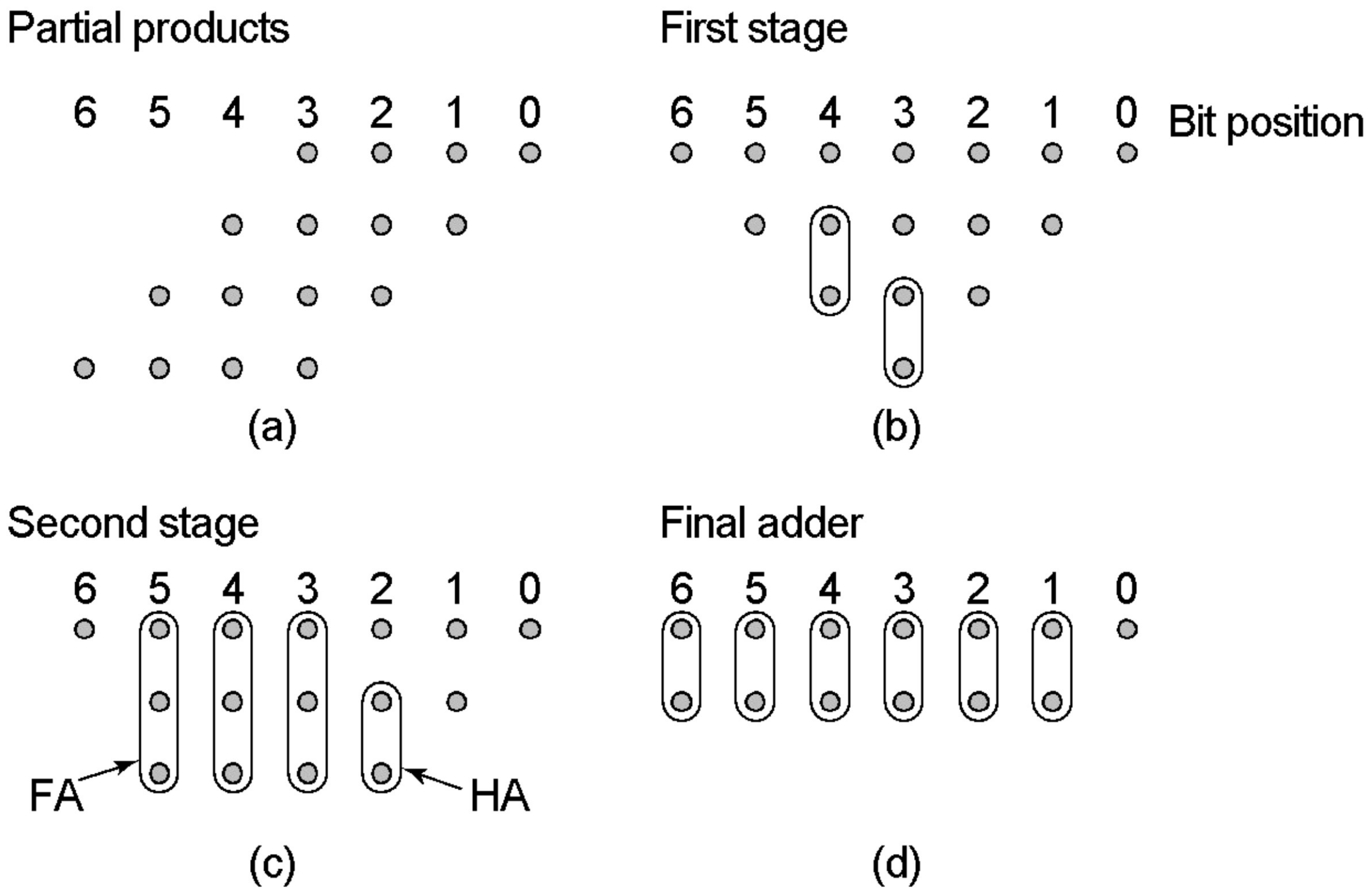
$$\begin{array}{cc} -1 & (a_4x_0)' \\ \uparrow & 1 \end{array}$$

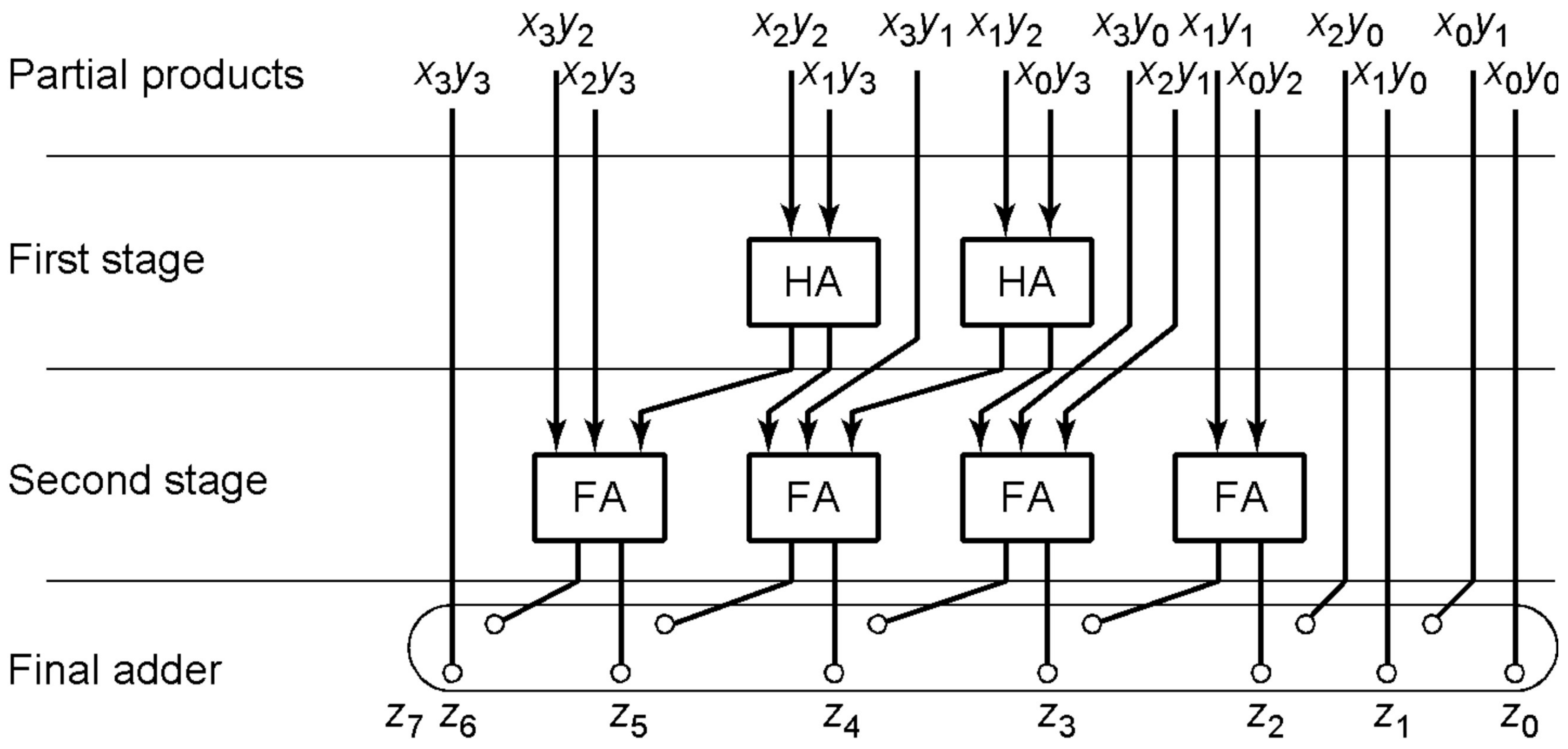
In next column

$$\begin{array}{cccccccccc}
 & & & & \times & a_4 & a_3 & a_2 & a_1 & a_0 \\
 & & & & & x_4 & x_3 & x_2 & x_1 & x_0 \\
\hline
 & & & & & \overline{a_4x_0} & a_3x_0 & a_2x_0 & a_1x_0 & a_0x_0 \\
 & & & & \overline{a_4x_1} & a_3x_1 & a_2x_1 & a_1x_1 & a_0x_1 & \\
 & & & \overline{a_4x_2} & a_3x_2 & a_2x_2 & a_1x_2 & a_0x_2 & & \\
 & & \overline{a_4x_3} & a_3x_3 & a_2x_3 & a_1x_3 & a_0x_3 & & & \\
 & a_4x_4 & \overline{a_3x_4} & \overline{a_2x_4} & \overline{a_1x_4} & \overline{a_0x_4} & & & & \\
1 & & & & 1 & & & & & \\
\hline
\bar{p}_9 & p_8 & p_7 & p_6 & p_5 & p_4 & p_3 & p_2 & p_1 & p_0
\end{array}$$


$a_3$
$a_2$
$a_1$
$a_0$
$b_0$
$p_0$
0
0
0
$b_1$
F.A.
$p_1$
$b_2$
$p_2$
$b_3$
1
$a_3$ $b_3$
$p_7$
$p_6$
$p_5$
$p_4$
$p_3$

$b_3$
$b_2$
$b_1$
$b_0$
$a_0$
$a_1$
$a_2$
$a_3$
HA
FA
1
$c_0$
$c_1$
$c_2$
$c_3$
$c_4$
$c_5$
$c_6$
$c_7$
HA: Half Adder
FA: Full Adder

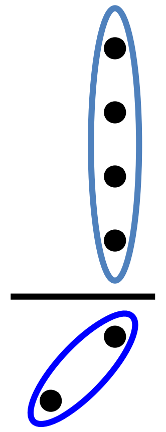
# Wallace-Tree Multiplier



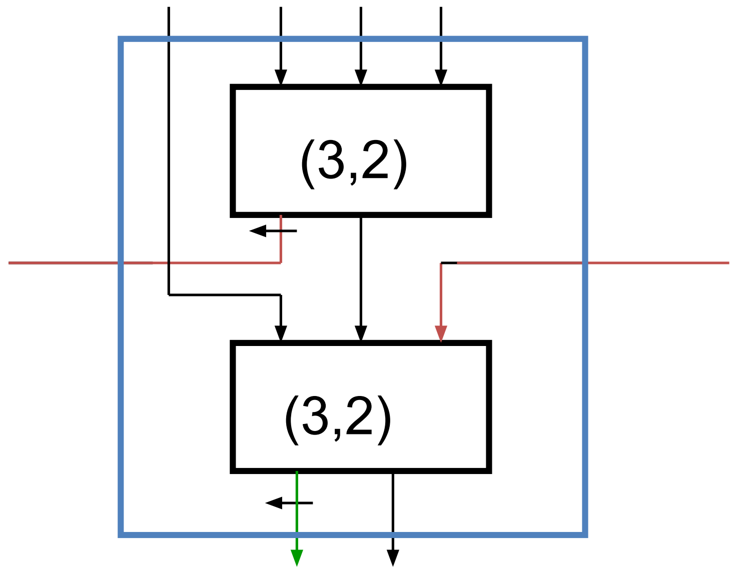


Full adder = (3,2) compressor

## (4,2) Counter

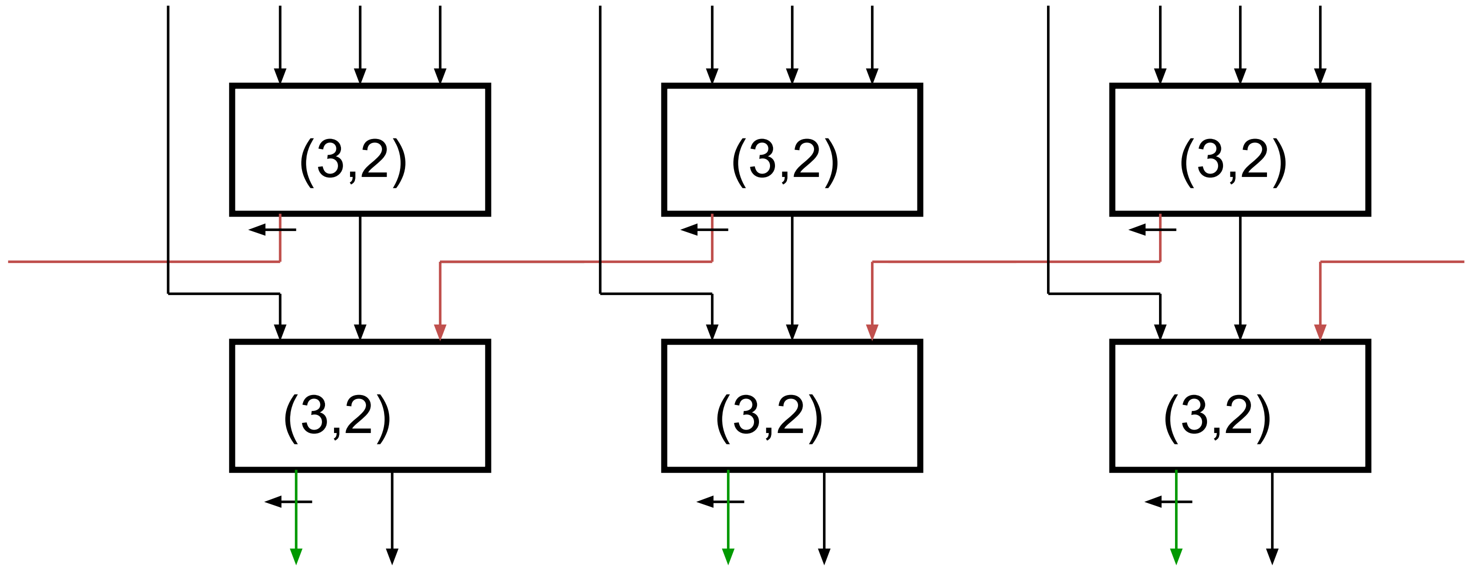


- Built out of two (3,2) counters (just FA's!)
  - all of the inputs (4 external plus one internal) have the same weight (i.e., are in the same bit position)
  - the internal carry output is fed to the next higher weight position (indicated by the )



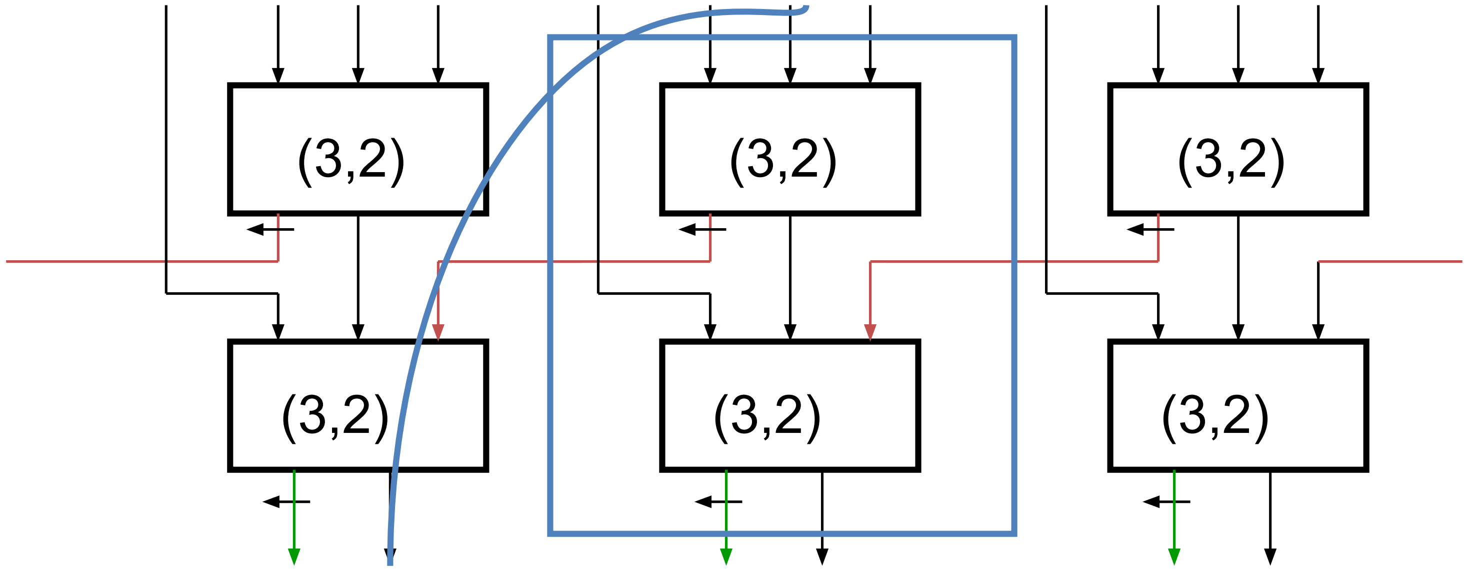
Note: Two carry outs - one "internal" and one "external"

## Tiling (4,2) Counters



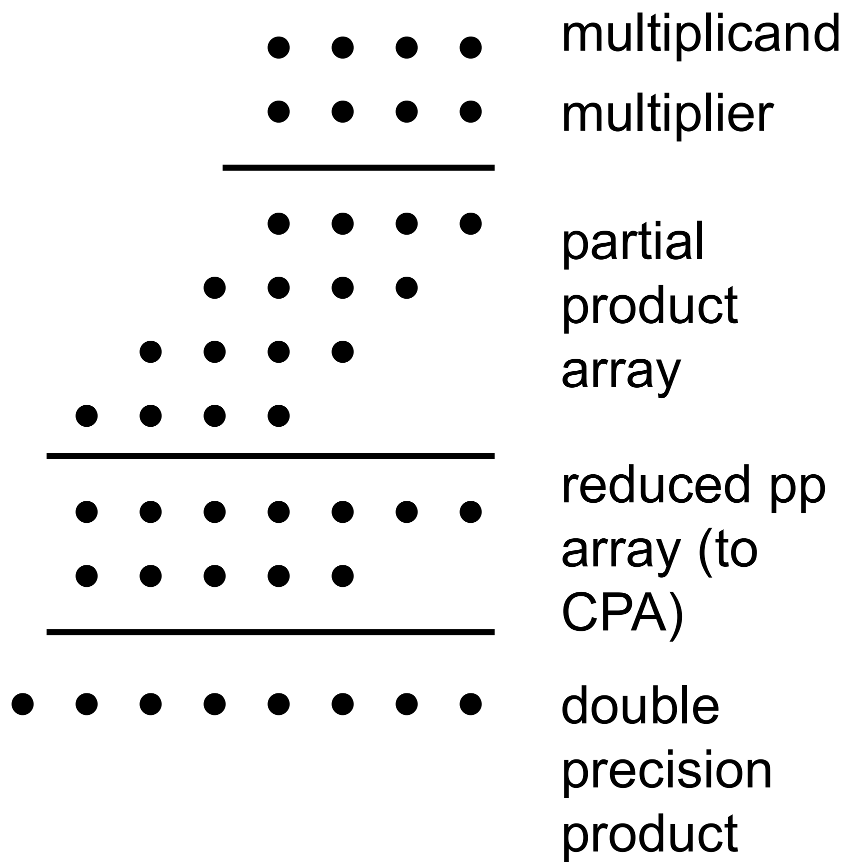
- Reduces columns four high to columns only two high
  - Tiles with neighboring (4,2) counters
  - Internal carry in at same “level” (i.e., bit position weight) as the internal carry out

# Tiling (4,2) Counters

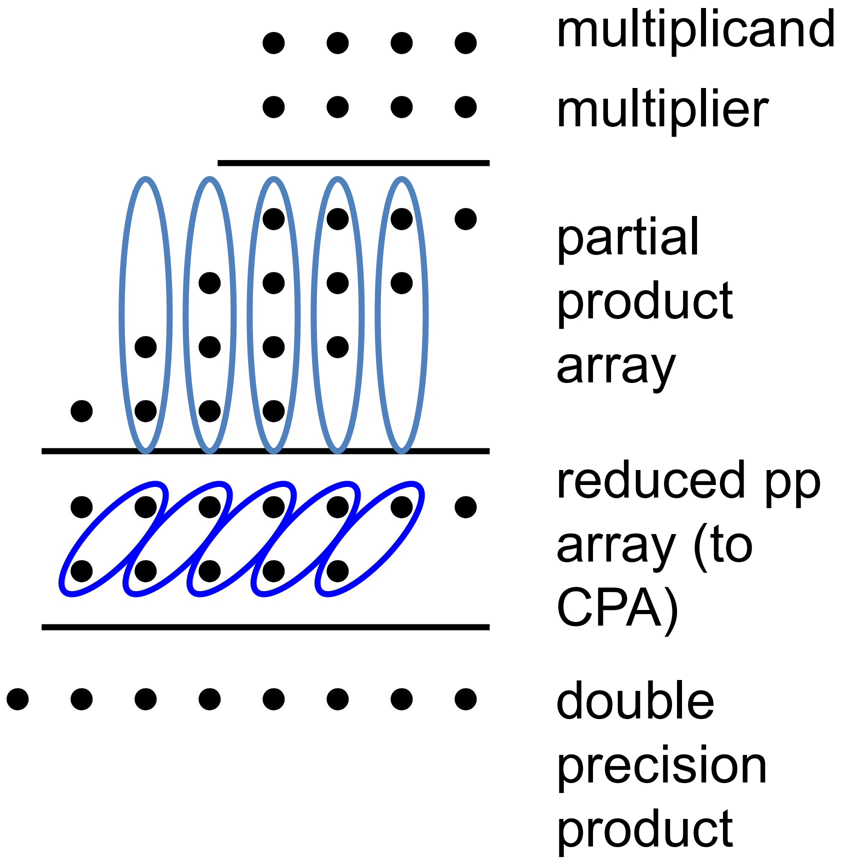


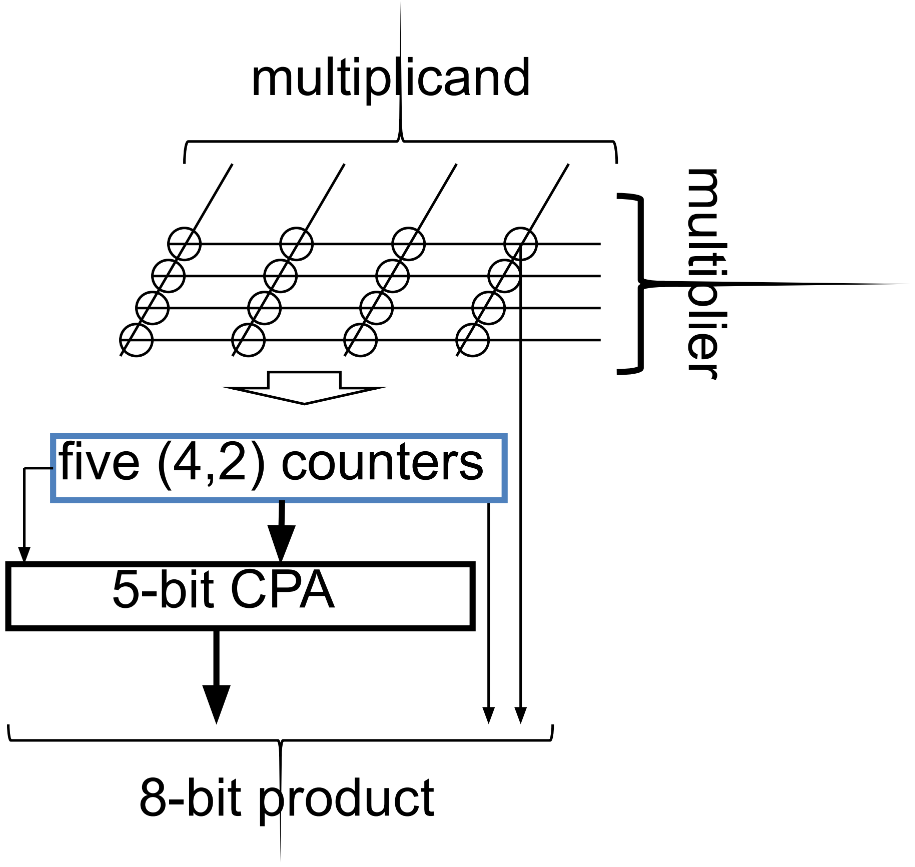
## 4x4 Partial Product Array Reduction

- Fast 4x4 multiplication using (4,2) counters



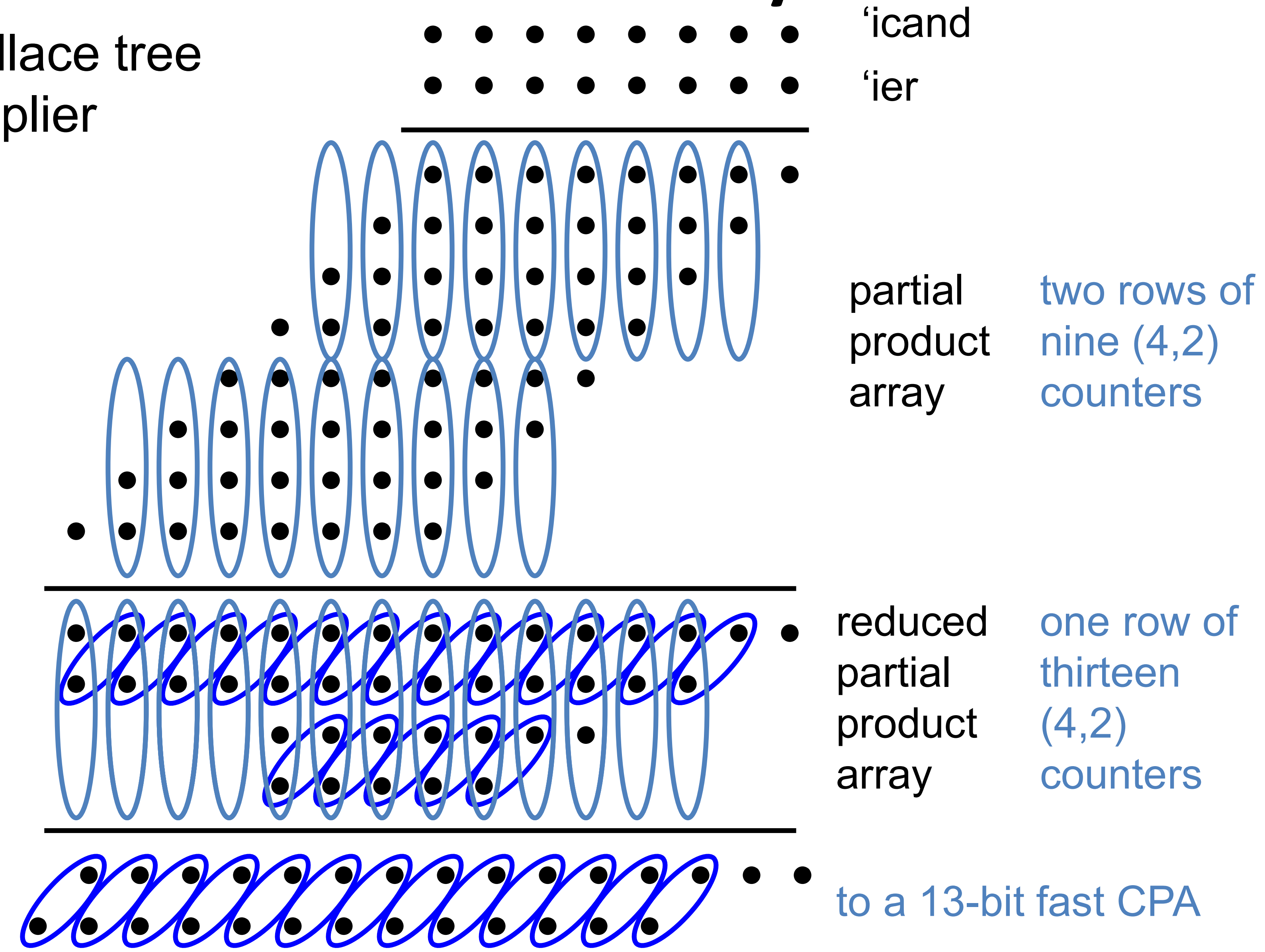
- Fast 4x4 multiplication using (4,2) counters





# 8x8 Partial Product Array Reduction

- Wallace tree multiplier


‘icand
‘ier
partial product array
two rows of nine (4,2) counters
reduced partial product array
one row of thirteen (4,2) counters
to a 13-bit fast CPA