Digital Computer Arithmetic

ECE 666

Part 6a
High-Speed Multiplication - I

Israel Koren

Spring 2004

Speeding Up Multiplication

- Multiplication involves 2 basic operations - generation of partial products + their accumulation
- 2 ways to speed up - reducing number of partial products and/or accelerating accumulation
- 3 types of high-speed multipliers:
- Sequential multiplier - generates partial products sequentially and adds each newly generated product to previously accumulated partial product
- Parallel multiplier - generates partial products in parallel, accumulates using a fast multi-operand adder
- Array multiplier - array of identical cells generating new partial products; accumulating them simultaneously
  - No separate circuits for generation and accumulation
  - Reduced execution time but increased hardware complexity

Reducing Number of Partial Products

- Examining 2 or more bits of multiplier at a time
- Requires generating A (multiplicand), 2A, 3A
- Reduces number of partial products to n/2 - each step more complex
- Several algorithm which do not increase complexity proposed - one is Booth's algorithm
- Fewer partial products generated for groups of consecutive 0's and 1's

Booth's Algorithm

- Group of consecutive 0's in multiplier - no new partial product - only shift partial product right one bit position for every 0
- Group of m consecutive 1's in multiplier - less than m partial products generated
- ...01...110... = ...10...000... - ...00...010...
- Using SD (signed-digit) notation =...100...010...
- Example:
- ...011110... = ...100000... - ...000010... = ...100010... (decimal notation: 15=16-1)
- Instead of generating all m partial products - only 2 partial products generated
- First partial product added - second subtracted - number of single-bit shift-right operations still m

Booth's Algorithm - Rules

x_i	x_{i-1}	Operation	Comments	y_i
0	0	shift only	string of zeros	0
1	1	shift only	string of ones	0
1	0	subtract and shift	beginning of a string of ones	$\bar{1}$
0	1	add and shift	end of a string of ones	1

- Recoding multiplier $x_{n-1}\, x_{n-2} \ldots x_1\, x_0$ in SD code
- Recoded multiplier $y_{n-1}\, y_{n-2} \ldots y_1\, y_0$
- x_i, x_{i-1} of multiplier examined to generate y_i
- Previous bit - x_{i-1} - only reference bit
- i=0 - reference bit $x_{-1}=0$
- Simple recoding - $y_i = x_{i-1} - x_i$
- No special order - bits can be recoded in parallel
- Example: Multiplier 0011110011(0) recoded as 0100010101 - 4 instead of 6 add/subtracts

Sign Bit

	x_{n-1}	x_{n-2}	y_{n-1}
(1)	1	0	$\bar{1}$
(2)	1	1	0

- Two's complement - sign bit x_{n-1} must be used
- Deciding on add or subtract operation - no shift required - only prepares for next step
- Verify only for negative values of X ($x_{n-1}=1$)
- 2 cases

$$A \cdot X = A \cdot \widetilde{X} - A \cdot x_{n-1} \cdot 2^{n-1} \quad \text{where} \quad \widetilde{X} = \sum_{j=0}^{n-2} x_j 2^j$$

- Case 1 - A subtracted - necessary correction
- Case 2 - without sign bit - scan over a string of 1's and perform an addition for position n-1
  - When $x_{n-1}=1$ considered - required addition not done
  - Equivalent to subtracting $A \cdot 2^{n-1}$ correction term

Example (Case-2)

A		1	0	1	1			-5
X	$\times$	1	1	0	1			-3
Y		0	$\bar{1}$	1	$\bar{1}$			recoded multiplier
Add $-A$		0	1	0	1			
Shift		0	0	1	0	1		
Add A	$+$	1	0	1	1			
		1	1	0	1	1		
Shift		1	1	1	0	1	1	
Add $-A$	$+$	0	1	0	1			
		0	0	1	1	1	1	
Shift		0	0	0	1	1	1	1

Example

A		1	0	1	1			−5		
X	×	1	0	0	1			**-7**		
Y		$\bar{1}$	0	1	$\bar{1}$			recoded multiplier		
Add $-A$		0	1	0	1					
Shift		0	0	1	0	1				
Add A	+	1	0	1	1					
		1	1	0	1	1				
Shift		1	1	1	0	1	1			
		1	**1**	**1**	**1**	**0**	**1**	**1**		
Add $-A$	+	0	1	0	1					
		0	1	0	0	0	1	1		
Shift		0	0	1	0	0	0	1	1	**35**

Booth's Algorithm - Properties

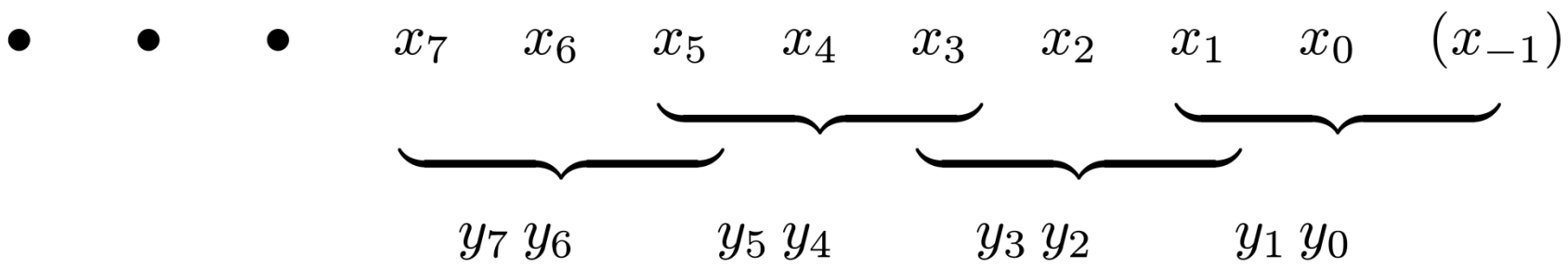
- Multiplication starts from least significant bit
- If started from most significant bit - longer adder/subtractor to allow for carry propagation
- No need to generate recoded SD multiplier (requiring 2 bits per digit)
- Booth's algorithm can handle two's complement multipliers
  - If unsigned numbers multiplied - 0 added to left of multiplier ($x_n=0$) to ensure correctness

Drawbacks to Booth's Algorithm

- Variable number of add/subtract operations and of shift operations between two consecutive add/subtract operations
  - Inconvenient when designing a synchronous multiplier
- Algorithm inefficient with isolated 1's
- Example:
- 001010101(0) recoded as $01111\bar{1}1\bar{1}\bar{1}$, requiring 8 instead of 4 operations
- Situation can be improved by examining 3 bits of X at a time rather than 2

Radix-4 Modified Booth Algorithm

- Bits x_i and x_{i-1} recoded into y_i and y_{i-1} - x_{i-2} serves as reference bit
- Separately - x_{i-2} and x_{i-3} recoded into y_{i-2} and y_{i-3} - x_{i-4} serves as reference bit
- Groups of 3 bits each overlap - rightmost being x_1 x_0 (x_{-1}), next x_3 x_2 (x_1), and so on



Radix-4 Algorithm - Rules

x_i	x_{i-1}	x_{i-2}	y_i	y_{i-1}	operation	comments
0	0	0	0	0	$+0$	string of zeros
0	1	0	0	1	$+A$	a single 1
1	0	0	$\bar{1}$	0	$-2A$	beginning of 1's
1	1	0	0	$\bar{1}$	$-A$	beginning of 1's
0	0	1	0	1	$+A$	end of 1's
0	1	1	1	0	$+2A$	end of 1's
1	0	1	0	$\bar{1}$	$-A$	a single 0
1	1	1	0	0	$+0$	string of 1's

- i=1,3,5,…
- Isolated 0/1handled
- efficiently
- If x_{i-1} is an isolated 1, $y_{i-1}=1$ - only a single operation needed
- Similarly - x_{i-1} an isolated 0 in a string of 1's -…10(1)… recoded as …11… or …01… - single operation performed
- Exercise: To find required operation - calculate $x_{i-1}+x_{i-2}-2x_i$ for odd i's and represent result as a 2-bit binary number y_iy_{i-1} in SD

Radix-4 vs. Radix-2 Algorithm

- 01|01|01|01|(0) yields 01|01|01|01| - number of operations remains 4 - the minimum
- 00|10|10|10|(0) yields 01|01̄|01̄|10̄|, requiring 4, instead of 3, operations
- Compared to radix-2 Booth's algorithm - less patterns with more partial products; Smaller increase in number of operations
- Can design n-bit synchronous multiplier that generates exactly n/2 partial products
- Even n - two's complement multipliers handled correctly; Odd n - extension of sign bit needed
- Adding a 0 to left of multiplier needed if unsigned numbers are multiplied and n odd - 2 0's if n even

Example

A			01	00	01			17
X	$\times$		11	01	11			-9
Y			$0\bar{1}$	10	$0\bar{1}$			recoded multiplier
			$-A$	$+2A$	$-A$			operation
Add $-A$	$+$		10	11	11			
2-bit Shift		1	11	10	11	11		
Add $2A$	$+$	0	10	00	10			
			01	11	01	11		
2-bit Shift			00	01	11	01	11	
Add $-A$	$+$		10	11	11			
			11	01	10	01	11	-153

- n/2=3 steps ; 2 multiplier bits in each step
- All shift operations are 2 bit position shifts
- Additional bit for storing correct sign required to properly handle addition of 2A

Radix-8 Modified Booth's Algorithm

- Recoding extended to 3 bits at a time - overlapping groups of 4 bits each
- Only n/3 partial products generated - multiple 3A needed - more complex basic step
- Example: recoding 010(1) yields $y_i\ y_{i-1}\ y_{i-2}$=011
- Technique for simplifying generation and accumulation of ±3A exists
- To find minimal number of add/subtract ops required for a given multiplier - find minimal SD representation of multiplier
- Representation with smallest number of nonzero digits - $$min \sum_{i=0}^{n-1} |y_i|$$

Obtaining Minimal Representation of X

- $y_{n-1}y_{n-2}\ldots y_0$

is a minimal representation of an SD number

if $y_i \cdot y_{i-1}=0$ for $1 \leq i \leq n-1$,

given that most significant bits can satisfy

$y_{n-1} \cdot y_{n-2} \neq 1$

Canonical Recoding

x_{i+1}	x_i	c_i	y_i	c_{i+1}	Comments
0	0	0	0	0	string of 0's
0	1	0	1	0	a single 1
1	0	0	0	0	string of 0's
1	1	0	$\bar{1}$	1	beginning of 1's
0	0	1	1	0	end of 1's
0	1	1	0	1	string of 1's
1	0	1	$\bar{1}$	1	a single 0
1	1	1	0	1	string of 1's

- Multiplier bits examined one at a time

from right; x_{i+1} - reference bit

- To correctly handle a single 0/1 in string of 1's/0's - need information on string to right
- "Carry" bit - 0 for 0's and 1 for 1's
- As before, recoded multiplier can be used without correction if represented in two's complement
- Extend sign bit x_{n-1} - $x_{n-1}x_{n-1}x_{n-2}...x_0$
- Can be expanded to two or more bits at a time
- Multiples needed for 2 bits - $\pm A$ and $\pm 2A$

Disadvantages of Canonical Recoding

- Bits of multiplier generated sequentially
- In Booth's algorithm - no "carry" propagation - partial products generated in parallel and a fast multi-operand adder used
- To take full advantage of minimum number of operations - number of add/subtracts and length of shifts must be variable - difficult to implement
- For uniforms shifts - n/2 partial products - more than the minimum in canonical recoding

Alternate 2-bit-at-a-time Algorithm

x_{i+1}	x_i	x_{i-1}	Operation	Comments
0	0	0	$+0$	string of 0's
0	0	1	$+2A$	end of 1's
0	1	0	$+2A$	a single 1
0	1	1	$+4A$	end of 1's
1	0	0	$-4A$	beginning of 1's
1	0	1	$-2A$	a single 0
1	1	0	$-2A$	beginning of 1's
1	1	1	$+0$	string of 1's

- Reducing number of partial products

But still uniform shifts of 2 bits each

- x_{i+1} reference bit for x_i x_{i-1} - i odd
- $\pm 2A, \pm 4A$ can be generated using shifts
- 4A generated when $(x_{i+1})x_i\,(x_{i-1})=(0)11$ - group of 1's - not for $(x_{i+3})(x_{i+2})x_{i+1}$ - 0 in rightmost position
  - Not recoding - cannot express 4 in 2 bits
  - Number of partial products - always n/2
  - Two's complement multipliers - extend sign bit
  - Unsigned numbers - 1 or 2 0's added to left of multiplier

Example

- Multiplier 01101110 - partial products:

$$\begin{array}{cccccc} (0) & 01 & 10 & 11 & 10 \\ & +2A & -2A & +4A & -2A \end{array}$$

- Translates to the SD number $010\overline{1}100\overline{1}0$ - not minimal - includes 2 adjacent nonzero digits
- Canonical recoding yields $0100\overline{1}00\overline{1}0$ - minimal representation

Example

A			01	00	01			17
X	$\times$	(1)	11	01	11			-9
			0	$-2A$	0			Operation
Initial $-A$			10	11	11			
Add 0	+		00	00	00			
			10	11	11			
2-bit Shift		1	11	10	11	11		
Add $-2A$	+	1	01	11	10			
		1	01	10	01	11		
2-bit Shift			11	01	10	01	11	
Add 0	+		00	00	00			
			11	01	10	01	11	-153

- Multiplier's sign bit extended in order to decide that no operation needed for first pair of multiplier bits
- As before - additional bit for holding correct sign is needed, because of multiples like -2A

Extending the Alternative Algorithm

- The above method can be extended to three bits or more at each step
- However, here too, multiples of A like 3A or even 6A are needed and
  - Prepare in advance and store
  - Perform two additions in a single step
- For example, for (0)101 we need 8-2=6, and for (1)001, -8+2=-6

Implementing Large Multipliers Using Smaller Ones

- Implementing n x n bit multiplier as a single integrated circuit - several such circuits for implementing larger multipliers can be used
- 2n x 2n bit multiplier can be constructed out of 4 n x n bit multipliers based on :

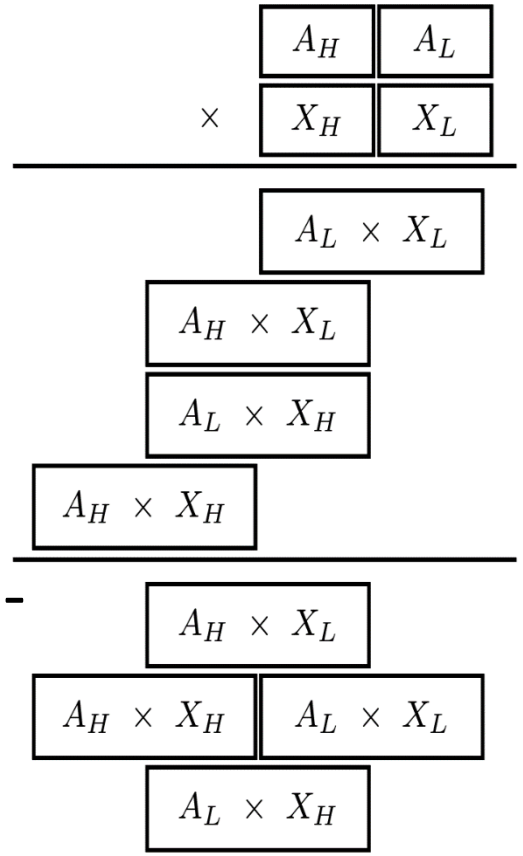
$$A \cdot X = (A_H \cdot 2^n + A_L) \cdot (X_H \cdot 2^n + X_L)$$

$$= A_H \cdot X_H \cdot 2^{2n} + (A_H \cdot X_L + A_L \cdot X_H) \cdot 2^n + A_L \cdot X_L$$

- A_H , A_L - most and least significant halves of A ; X_H , X_L - same for X

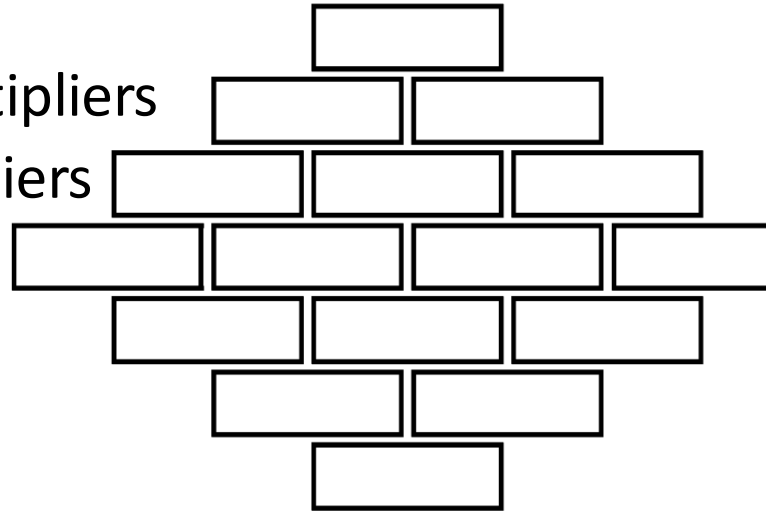
Aligning Partial Products

- 4 partial products of 2n bits - correctly aligned before adding
- Last arrangement – minimum height of matrix - 1 level of carry-save addition and a CPA
- n least significant bits - already bits of final product - no further addition needed
- 2n center bits - added by 2n-bit CSA with outputs connected to a CPA
- n most significant bits connected to same CPA, since center bits may generate carry into most significant bits - 3n-bit CPA needed



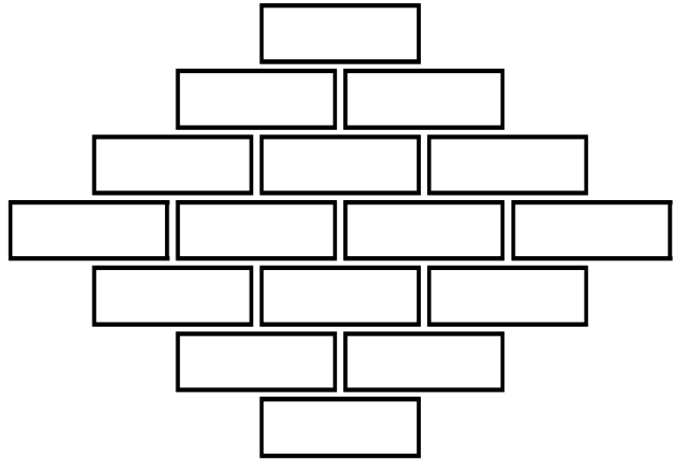
Decomposing a Large Multiplier into Smaller Ones - Extension

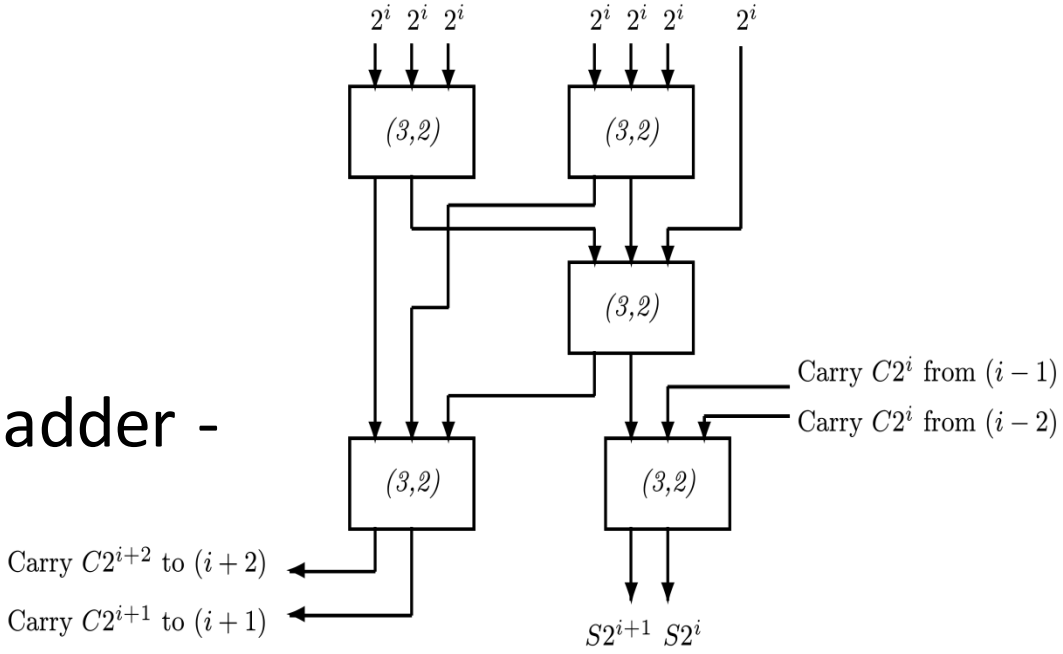
- Basic multiplier - n x m bits - $n \neq m$
- Multipliers larger than 2n x 2m can be implemented
- Example: 4n x 4n bit multiplier - implemented using n x n bit multipliers
  - 4n x 4n bit multiplier requires 4 2n x 2n bit multipliers
  - 2n x 2n bit multiplier requires 4 n x n bit multipliers
  - Total of 16 n x n bit multipliers
  - 16 partial products - aligned before being added
- Similarly - for any kn x kn bit multiplier with integer k



Adding Partial Products

- After aligning 16 products - 7 bits in one column need to be added
- Method 1: (7,3) counters - generating 3 operands added by
- (3,2) counters - generating 2 operands
- added by a CPA
- Method 2: Combining 2 sets of counters into a set of (7;2) compressors
- Selecting more economical multi-operand adder - discussed next



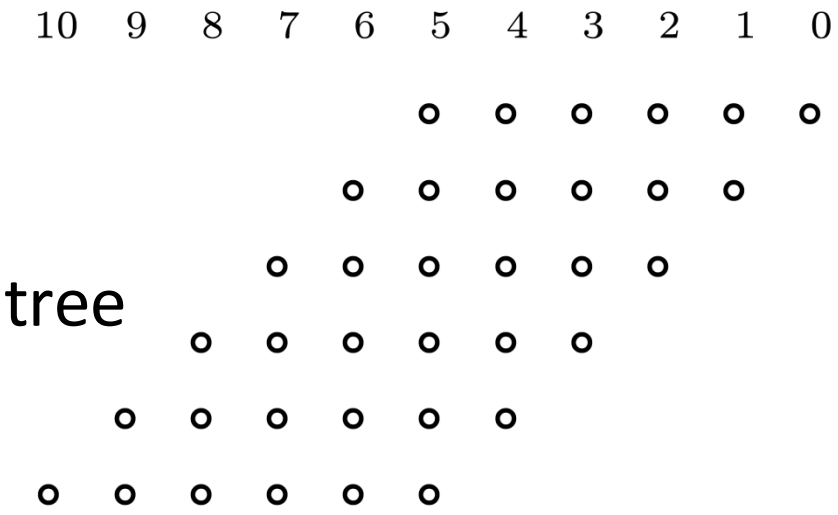


Accumulating the Partial Products

- After generating partial products either directly or using smaller multipliers
- Accumulate these to obtain final product
  - A fast multi-operand adder
- Should take advantage of particular form of partial products - reduce hardware complexity
- They have fewer bits than final product, and must be aligned before added
- Expect many columns that include fewer bits than total number of partial products - requiring simpler counters

Example - Six Partial Products

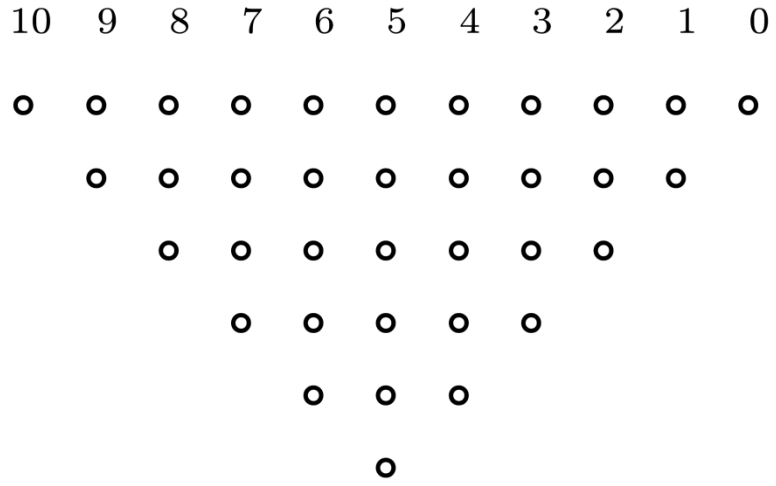
- Generated when multiplying unsigned 6-bit operands using one-bit-at-a-time algorithm
- 6 operands can be added using 3-level carry-save tree



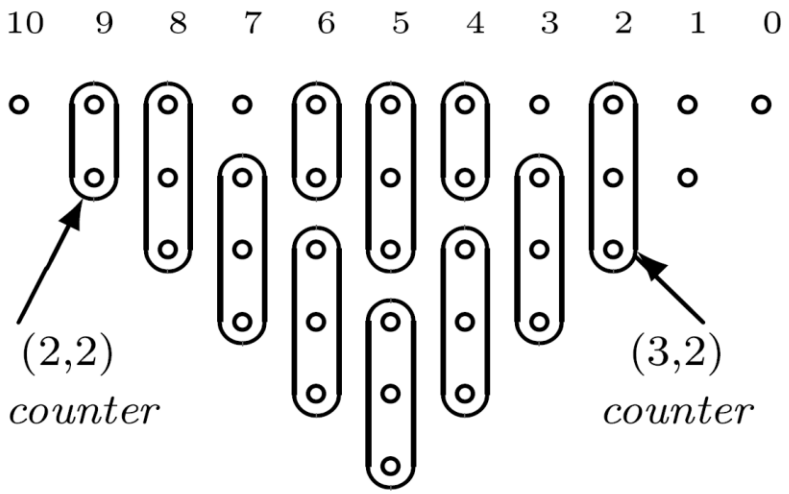
- Number of (3,2) counters can be substantially reduced by taking advantage of the fact that all

columns but 1 contain fewer than 6 bits

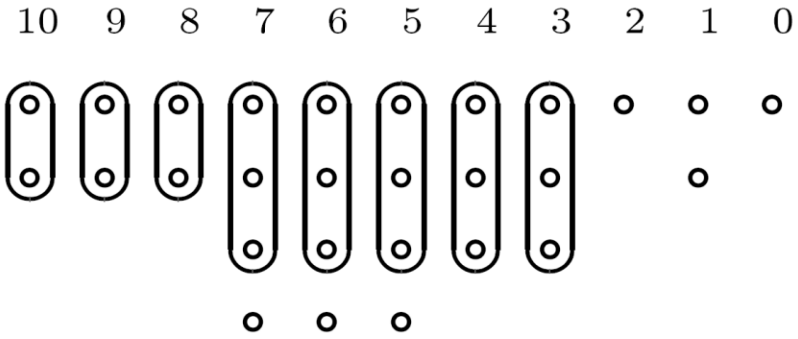
- Deciding how many counters needed - redraw matrix of bits to be added:



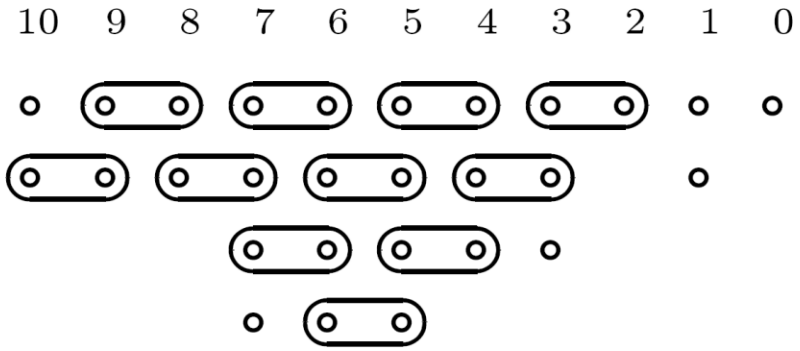
Reduce Complexity - Use (2,2) Counters (HAs)



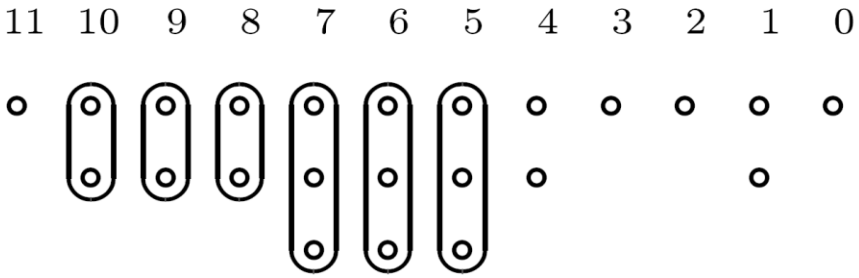
(a) Level 1 carry-save addition.



(c) Level 2 carry-save addition.



(b) Results of level 1.

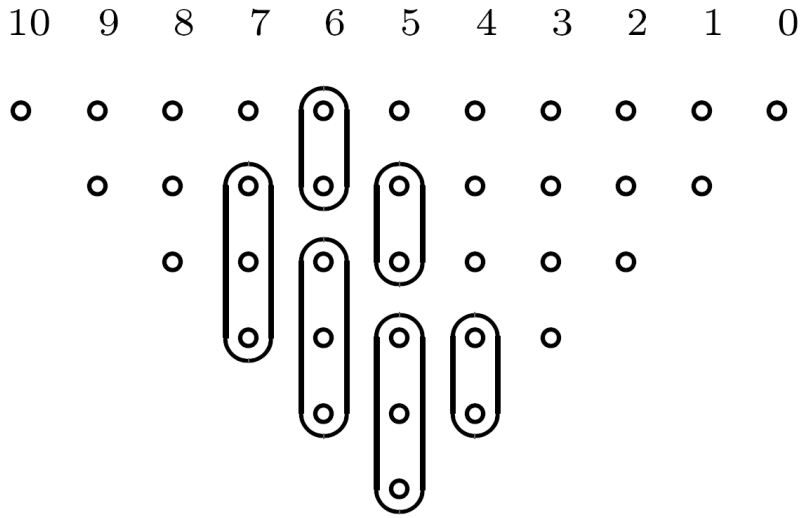


(d) Level 3 carry-save addition.

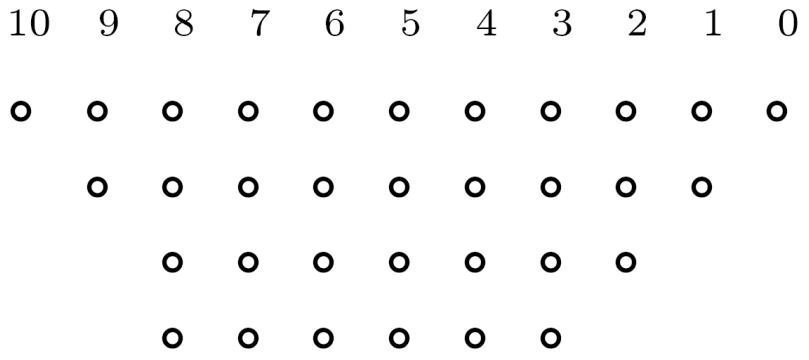
- Number of levels still 3, but fewer counters

Further reduction in number of counters

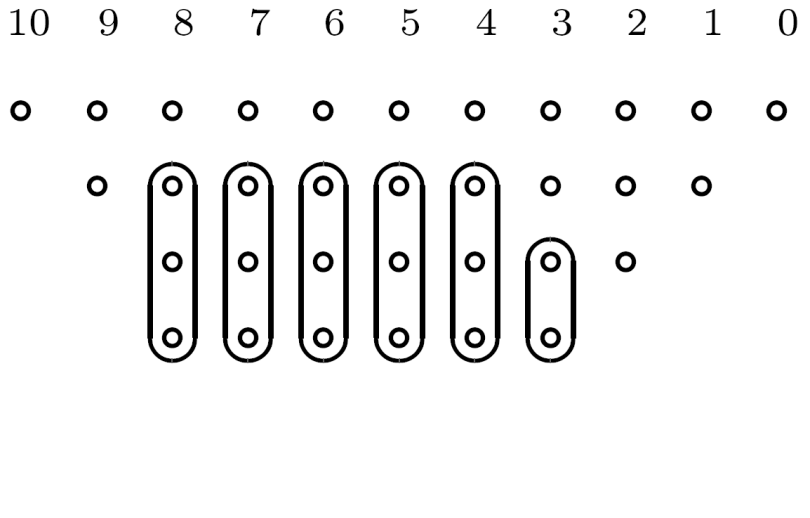
- Reduce # of bits to closest element of 3,4,6,9,13,19,…
- 15 (3,2) and 5 (2,2) vs. 16 (3,2) and 9 (2,2) counters



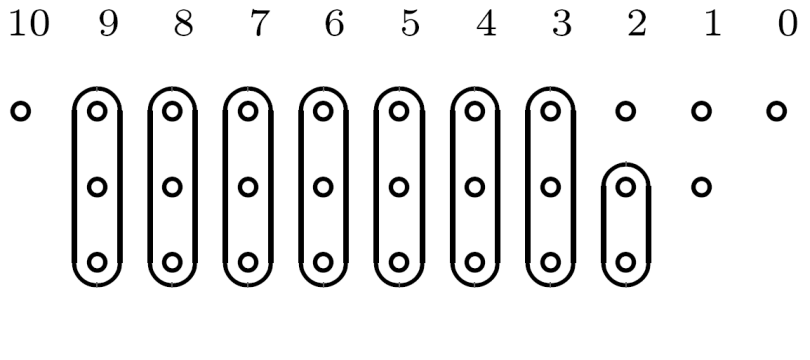
(a) Level 1 carry-save addition.



(b) Results of level 1.



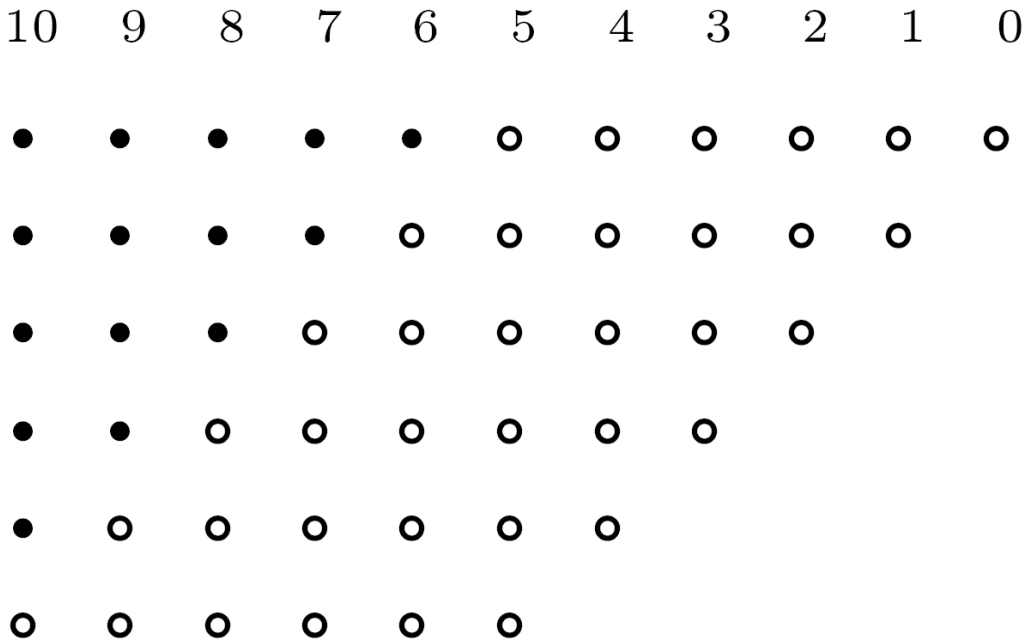
(c) Level 2 carry-save addition.



(d) Level 3 carry-save addition.

Modified Matrix for Negative Numbers

- Sign bits must be properly extended
- In row 1: 11 instead of 6 bits, and so on
- Increases complexity of multi-operand adder
- If two's complement obtained through one's complement - matrix increased even further



Reduce Complexity Increase

- Two's complement number s s s s s s z_4 z_3 z_2 z_1 z_0 with value

$$-s \cdot 2^{10} + s \cdot 2^9 + s \cdot 2^8 + s \cdot 2^7 + s \cdot 2^6 + s \cdot 2^5 + z_4 \cdot 2^4 + z_3 \cdot 2^3 + z_2 \cdot 2^2 + z_1 \cdot 2^1 + z_0$$

- Replaced by 0 0 0 0 0 (-s) z_4 z_3 z_2 z_1 z_0
- since

$$-s \cdot 2^{10} + s \cdot (2^9 + 2^8 + 2^7 + 2^6 + 2^5)$$

$$= -s \cdot 2^{10} + s \cdot (2^{10} - 2^5) = \quad -s \cdot 2^5$$

Example

- Recoded multiplier using canonical recoding

A					0	1	0	1	1	0	22
X			$\times$		0	0	1	0	1	1	11
Y					0	1	0	$\bar{1}$	0	$\bar{1}$	Recoded multiplier
$\mathbf{1}$	$\mathbf{1}$	$\mathbf{1}$	$\mathbf{1}$	$\mathbf{1}$	$\mathbf{1}$	0	1	0	1	0	
$\mathbf{0}$	$\mathbf{0}$	$\mathbf{0}$	$\mathbf{0}$	$\mathbf{0}$	0	0	0	0	0		
$\mathbf{1}$	$\mathbf{1}$	$\mathbf{1}$	$\mathbf{1}$	0	1	0	1	0			
$\mathbf{0}$	$\mathbf{0}$	$\mathbf{0}$	0	0	0	0	0				
$\mathbf{0}$	$\mathbf{0}$	1	0	1	1	0					
$\mathbf{0}$	0	0	0	0	0						
0	0	0	1	1	1	1	0	0	1	0	

Smaller Matrix for the Example

10	9	8	7	6	5	4	3	2	1	0
				0	**1**	0	1	0	1	0
				1	0	0	0	0	0	
			0	0	1	0	1	0		
		1	0	0	0	0	0			
	1	1	0	1	1	0				
1	0	0	0	0	0					
0	0	0	1	1	1	1	0	0	1	0

Using One's Complement and Carry

10	9	8	7	6	5	4	3	2	1	0
				0	**1**	0	1	0	0	1
				1	0	0	0	0	0	**1**
			0	0	1	0	0	1	**0**	
		1	0	0	0	0	0	**1**		
	1	1	0	1	1	0	**0**			
1	0	0	0	0	0	**0**				
					0					
0	0	0	1	1	1	1	0	0	1	0

Array Multipliers

- The two basic operations - generation and summation of partial products - can be merged, avoiding overhead and speeding up multiplication
- Iterative array multipliers (or array multipliers) consist of identical cells, each forming a new partial product and adding it to previously accumulated partial product
  - Gain in speed obtained at expense of extra hardware
  - Can be implemented so as to support a high rate of pipelining

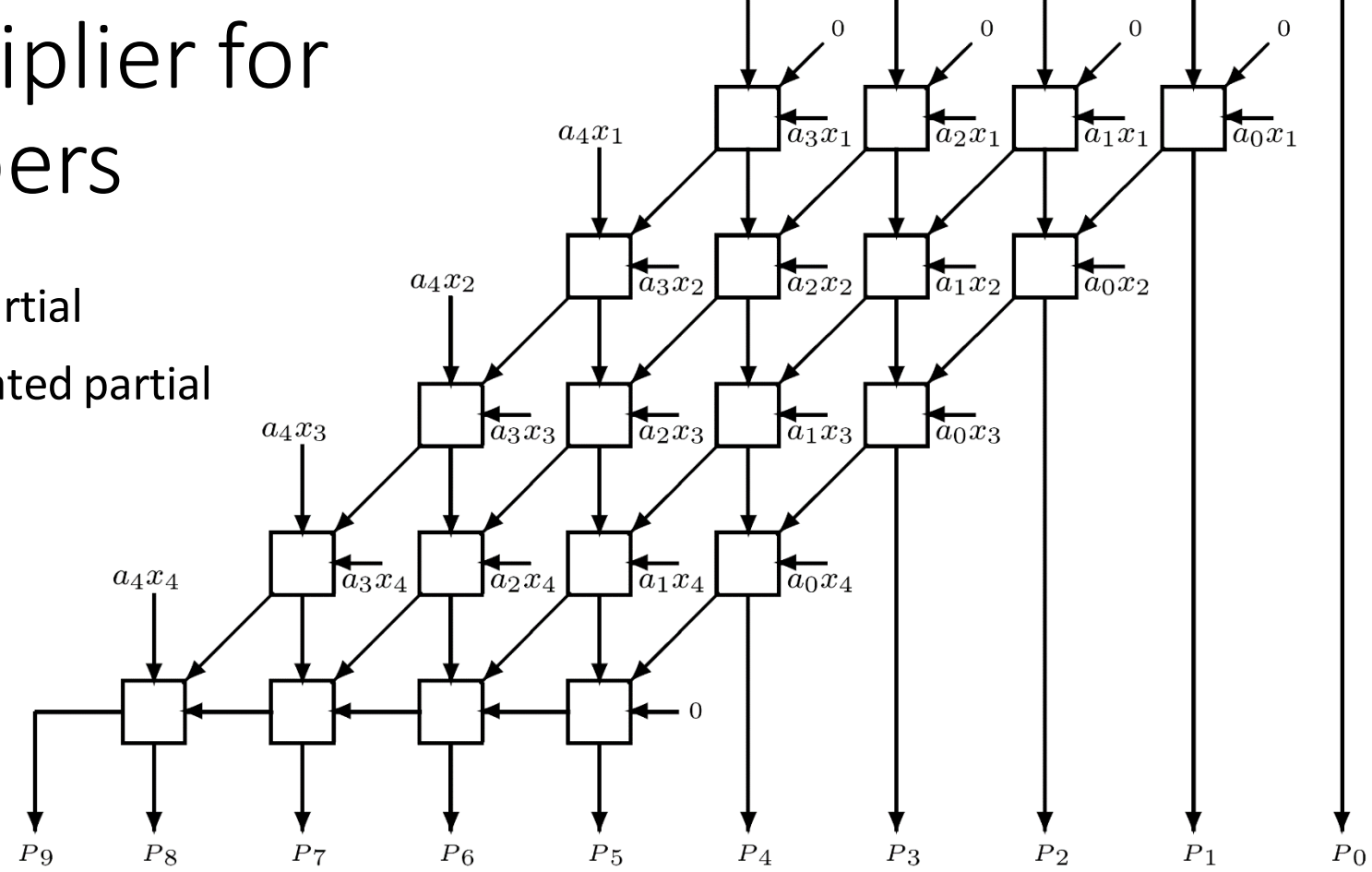
Illustration - 5 x 5 Multiplication

					a_4	a_3	a_2	a_1	a_0
				$\times$	x_4	x_3	x_2	x_1	x_0
					$a_4 \cdot x_0$	$a_3 \cdot x_0$	$a_2 \cdot x_0$	$a_1 \cdot x_0$	$a_0 \cdot x_0$
				$a_4 \cdot x_1$	$a_3 \cdot x_1$	$a_2 \cdot x_1$	$a_1 \cdot x_1$	$a_0 \cdot x_1$	
			$a_4 \cdot x_2$	$a_3 \cdot x_2$	$a_2 \cdot x_2$	$a_1 \cdot x_2$	$a_0 \cdot x_2$		
		$a_4 \cdot x_3$	$a_3 \cdot x_3$	$a_2 \cdot x_3$	$a_1 \cdot x_3$	$a_0 \cdot x_3$			
	$a_4 \cdot x_4$	$a_3 \cdot x_4$	$a_2 \cdot x_4$	$a_1 \cdot x_4$	$a_0 \cdot x_4$				
P_9	P_8	P_7	P_6	P_5	P_4	P_3	P_2	P_1	P_0

- Straightforward implementation -
  - Add first 2 partial products (a_4x_0, a_3x_0,…,a_0x_0 and a_4x_1, a_3x_1,…,a_0x_1) in row 1 after proper alignment
  - The results of row 1 are then added to a_4x_2, a_3x_2,…,a_0x_2 in row 2, and so on

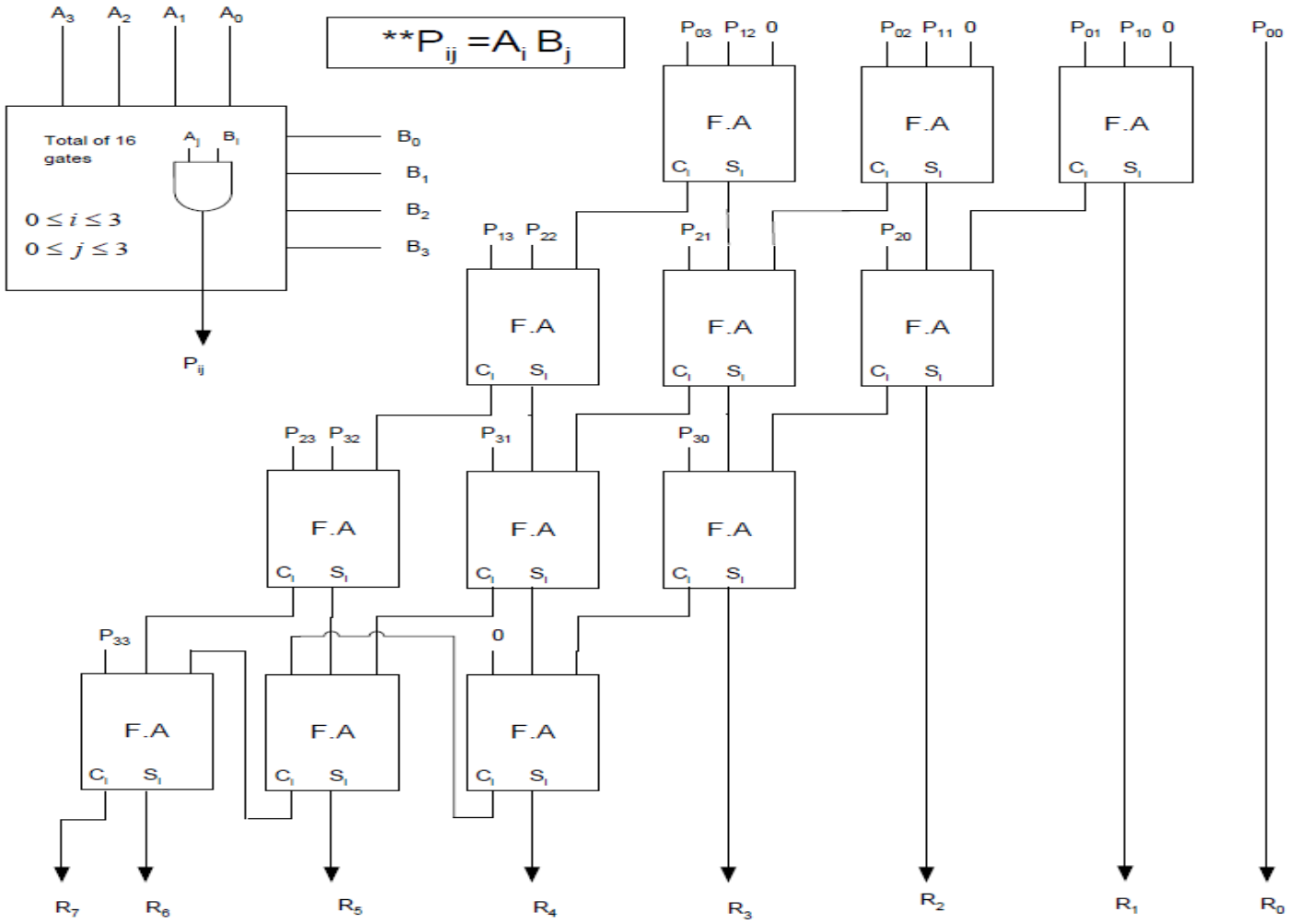
5 x 5 Array Multiplier for Unsigned Numbers

- Basic cell - FA accepting one bit of new partial product $a_i x_j$+ one bit of previously accumulated partial product + carry-in bit



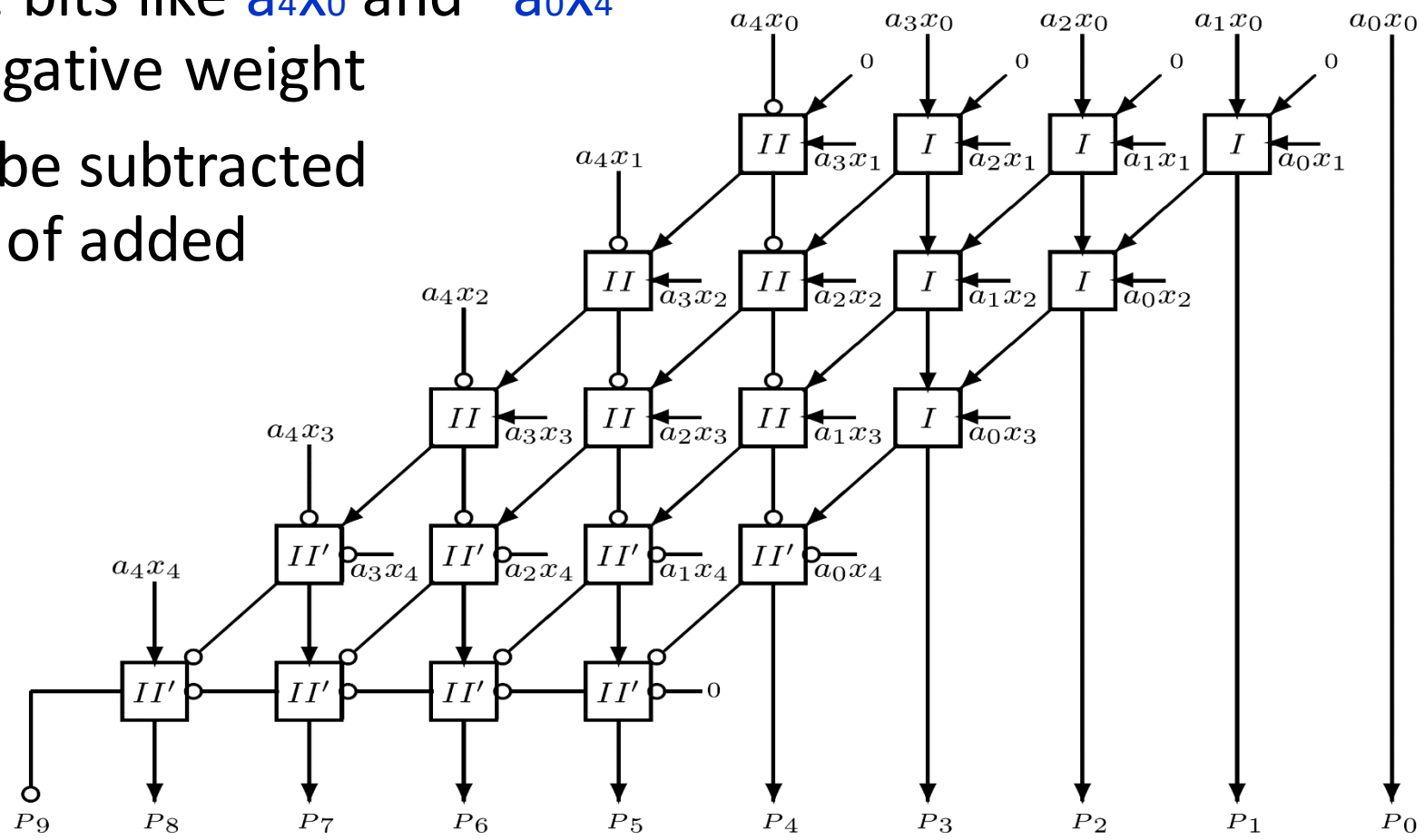
- No horizontal carry propagation in first 4 rows - carry-save type addition - accumulated partial product consists of intermediate sum and carry bits
- Last row is a ripple-carry adder - can be replaced by a fast 2-operand adder (e.g., carry-look-ahead adder)

4*4 bit Array Multiplier



Array Multiplier for Two's Complement Numbers

- Product bits like a_4x_0 and a_0x_4 have negative weight
- Should be subtracted instead of added

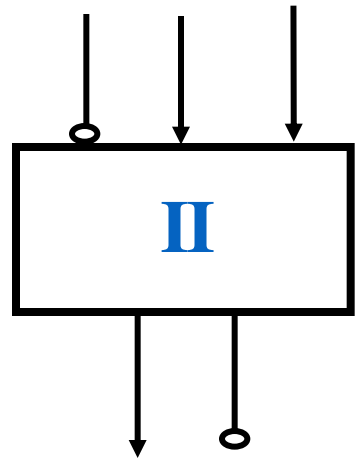


Type I and II Cells

- Type I cells: 3 positive inputs - ordinary FAs
- Type II cells: 1 negative and 2 positive inputs
- Sum of 3 inputs of type II cell can vary from -1 to 2
  - c output has weight +2
  - s output has weight -1
- Arithmetic operation of type II cell -

$$x + y - z = 2c - s$$

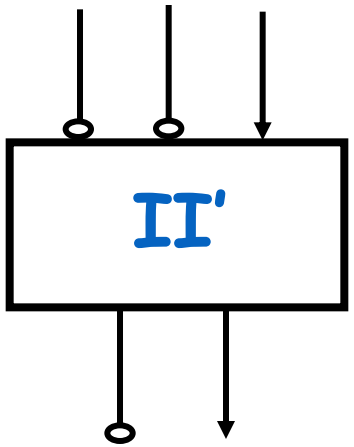
- s and c outputs given by



S= x + y –z (mod 2)

c= [(x+y-z) +s] / 2

Type I’ and II’ Cells



- Type II' cells: 2 negative inputs and 1 positive
- Sum of inputs varies from -2 to 1
  - c output has weight -2
  - s output has weight +1

- Type I' cell: all negative inputs -has negatively weighted c and s outputs
- Counts number of -1's at its inputs - represents this number through c and s outputs
- Same logic operation as type I cell - same gate implementation
- Similarly - types II and II' have the same gate implementation