Digital Computer Arithmetic

Part 6
High-Speed Multiplication

Soo-Ik Chae

Spring 2010

Speeding Up Multiplication

- Multiplication involves 2 basic operations
  - generation of partial products
  - their accumulation
- 2 ways to speed up
  - reducing number of partial products
  - accelerating accumulation

Speeding Up Multiplication

- 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...0$\bar{1}$0...
- Example:
- ...011110... = ...100000... - ...000010... = ...10$\bar{0}$0010... (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 $0100\bar{1}010\bar{1}$ - 4 instead of 6 add/subtracts

Sign Bit

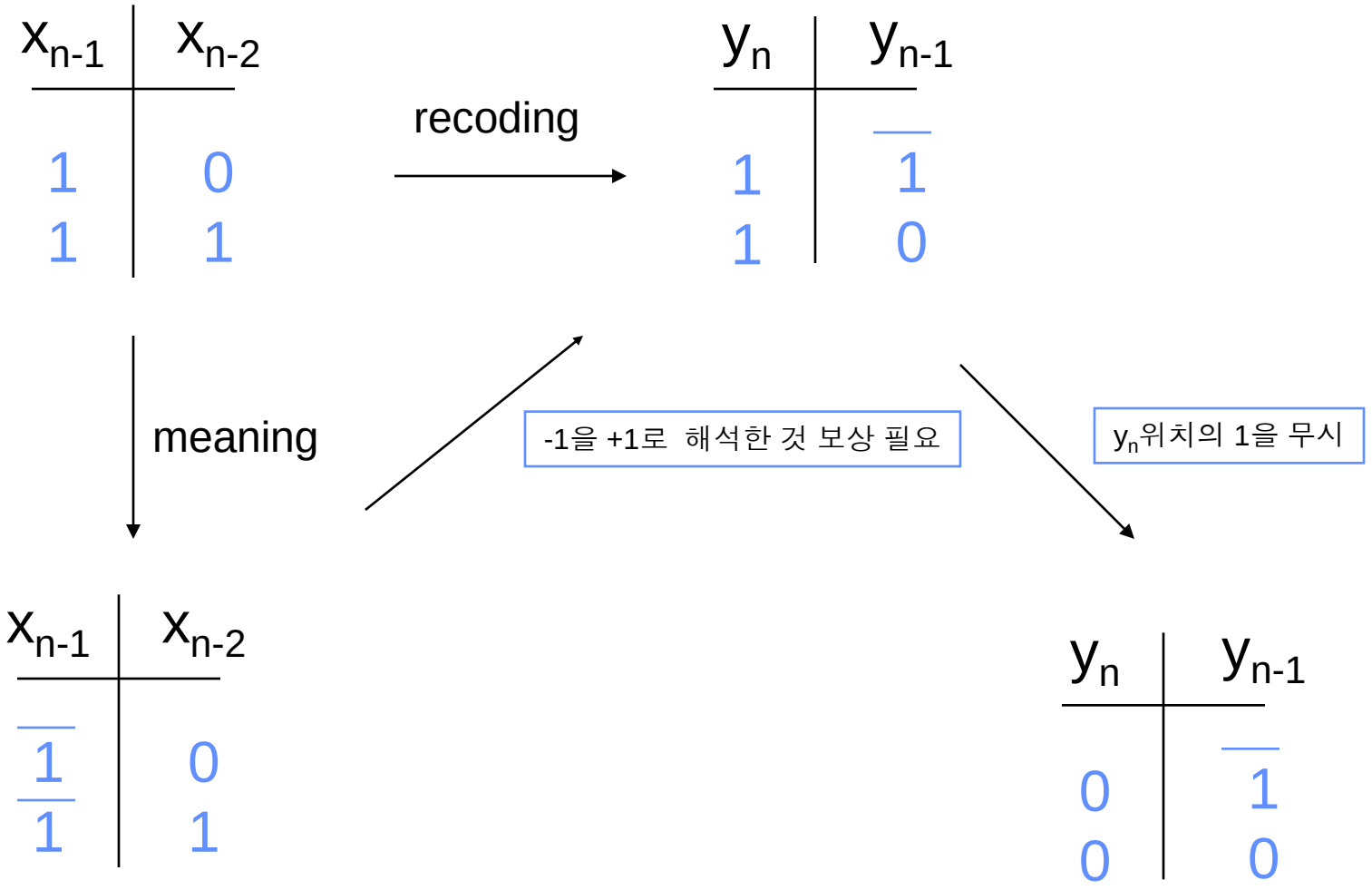
	x_{n-1}	x_{n-2}	y_{n-1}
(1)	1	0	$\bar{1}$
(2)	1	1	0

- 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

Recoding sign bit for two's complement



Example

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

Booth's Algorithm - Properties

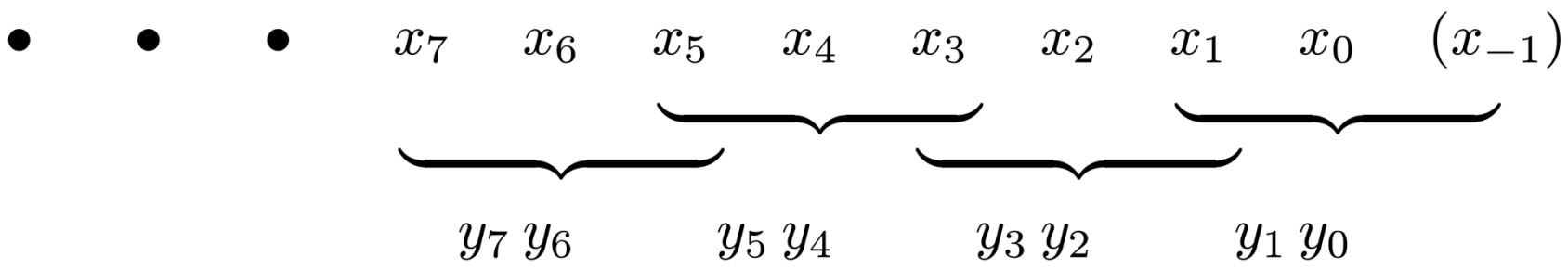
- Multiplication starts from least significant bit
- If started from most significant bit - longer adder/subtractor to allow for carry propagation
- No need to generate recoded SD multiplier (requiring 2 bits per digit)
  - Bits of original multiplier scanned - control signals for adder/subtractor generated
- 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 $01\bar{1}1\bar{1}1\bar{1}1\bar{1}$, 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 0/1 handled 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 …$1\bar{1}$… or …$0\bar{1}$… - 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|0$\bar{1}$|0$\bar{1}$|$\bar{1}$0|, 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 $1 \le i \le n-1$, given that most significant bits can satisfy $y_{n-1} \cdot y_{n-2} \neq 1$
- Example: Representation of 7 with 3 bits 111 minimal representation although $y_i \cdot y_{i-1} \neq 0$
- For any X - add a 0 to its left to satisfy above condition

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

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

- Reducing number of partial products but still uniform shifts of 2 bits each

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

- 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:

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

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

Dealing with Least significant Bit

- For the rightmost pair x_1x_0, if $x_0 = 1$ - considered continuation of string of 1's that never really started - no subtraction took place
- Example: multiplier 01110111 - partial products:

$$\begin{array}{lcccc} & 01 & 11 & 01 & 11 \\ & +2A & +0 & -2A & +0 \\ \text{instead of} & +2A & +0 & -2A & -A \end{array}$$

- Correction: when $x_0=1$ - set initial partial product to -A instead of 0
- 4 possible cases:

x_2	x_1	x_0	Operation
0	0	1	$+2A - A = A$
0	1	1	$+4A - A = 3A$
1	0	1	$-2A - A = -3A$
1	1	1	$0 - A = -A$

Example

- Previous 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

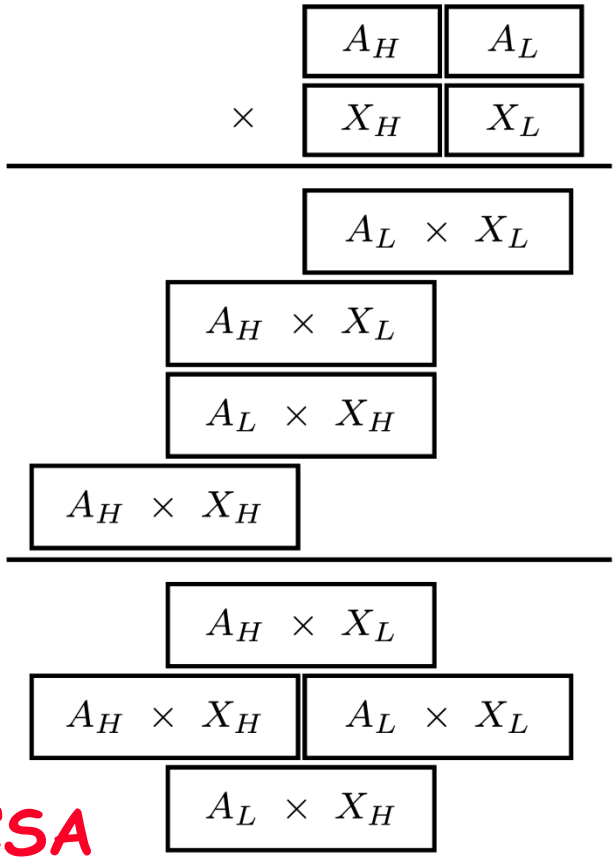
- Implementing **n x n** bit multiplier as a single integrated circuit - several such circuits for implementing larger multipliers can be used
- **2n x 2n** bit multiplier can be constructed out of **4 n x n** bit multipliers based on :

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

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

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

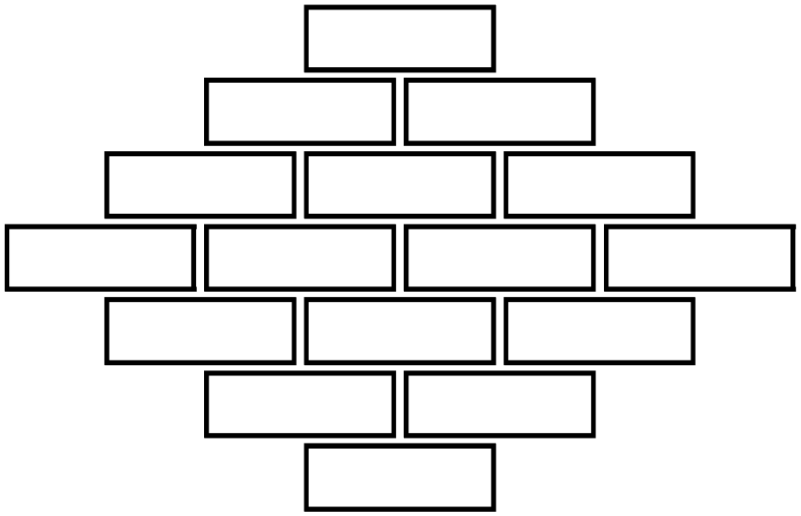
Aligning Partial Products



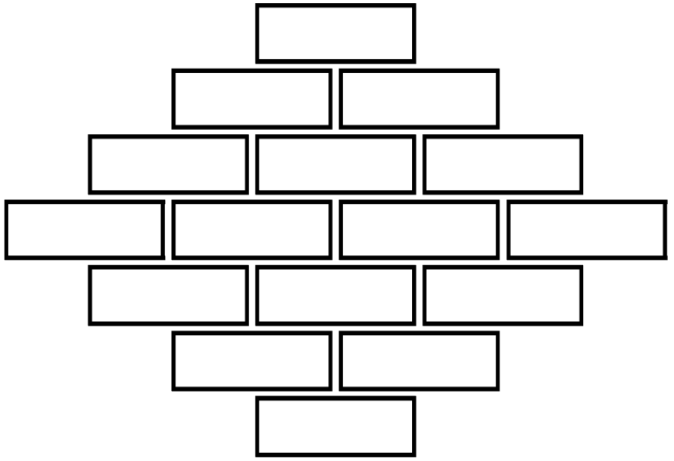
- 4 partial products of 2n bits - correctly aligned before adding
- Last arrangement - minimum height of matrix - 1 level of carry-save addition and a CPA
- n least significant bits - already bits of final product - no further addition needed
- 2n center bits - added by 2n-bit CSA with outputs connected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

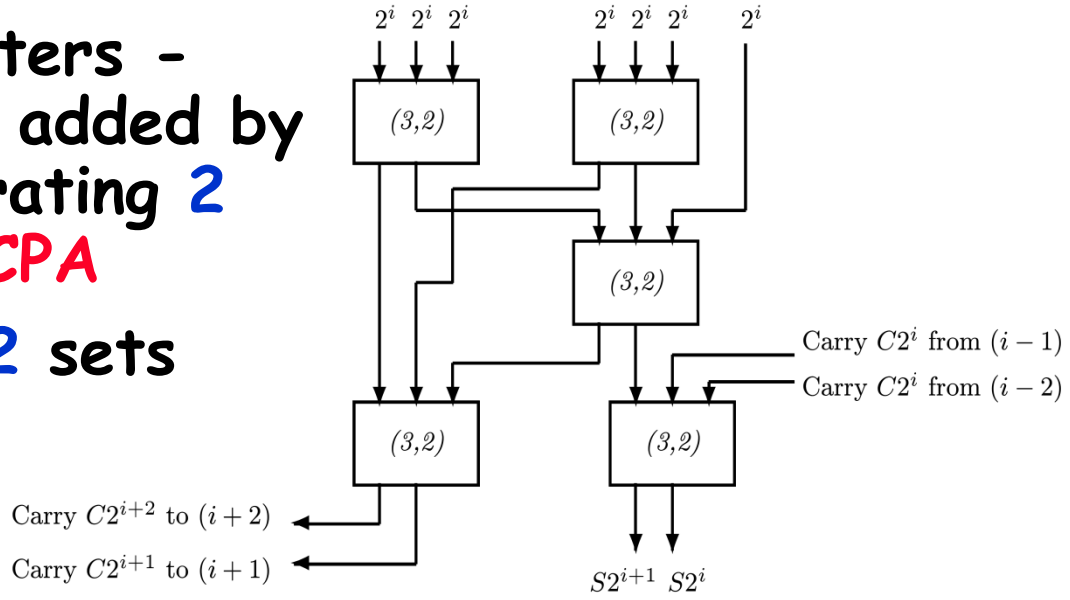
- Basic multiplier - n x m bits - $n \neq m$
- Multipliers larger than 2n x 2m can be implemented
- Example: 4n x 4n bit multiplier - implemented using n x n bit multipliers
  - 4n x 4n bit multiplier requires 4 2n x 2n bit multipliers
  - 2n x 2n bit multiplier requires 4 n x n bit multipliers
  - Total 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



Digital Computer Arithmetic

Part 6b
High-Speed Multiplication - II

Soo-Ik Chae

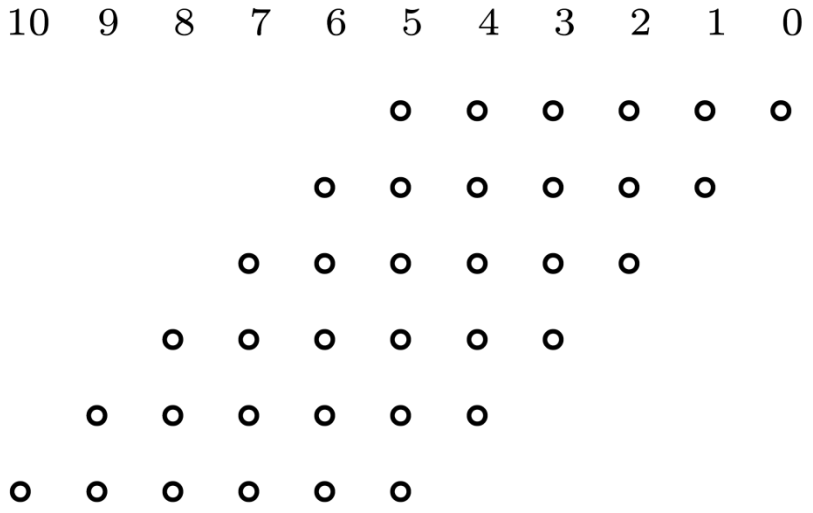
Spring 2009

Accumulating the Partial Products

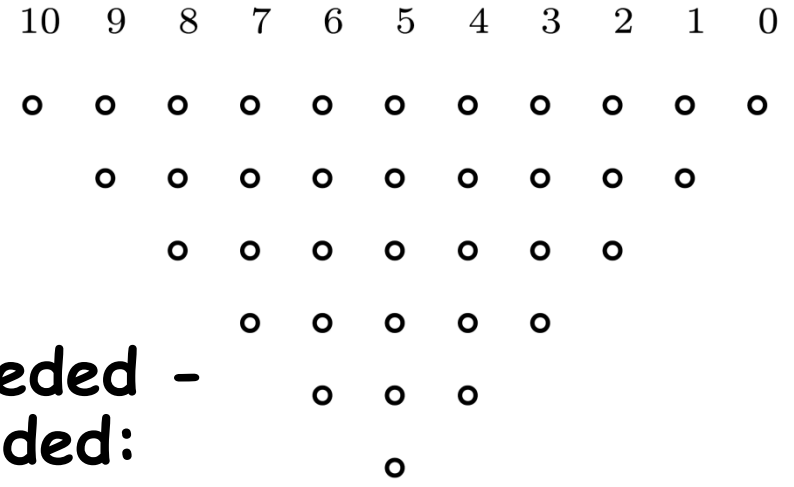
- After generating partial products either directly or using smaller multipliers
- Accumulate these to obtain final product
  - A fast multi-operand adder
- Should take advantage of particular form of partial products - reduce hardware complexity
- They have fewer bits than final product, and must be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

Example - Six Partial Products

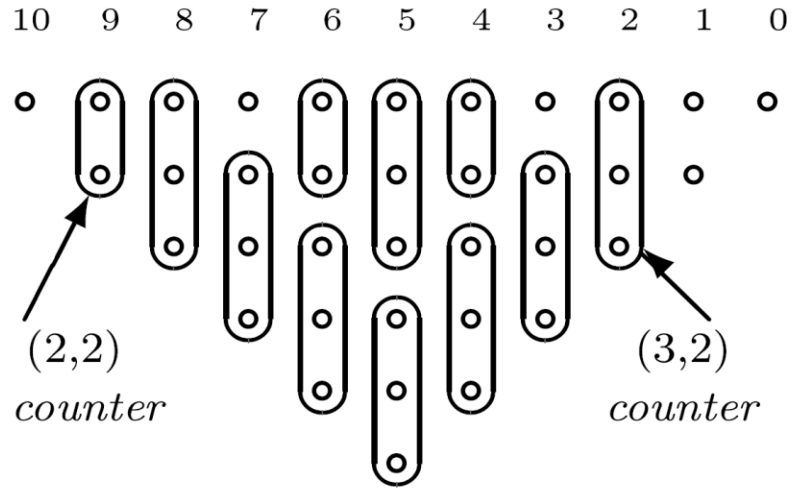
- Generated when multiplying unsigned 6-bit operands using one-bit-at-a-time algorithm
- 6 operands can be added using 3-level carry-save tree



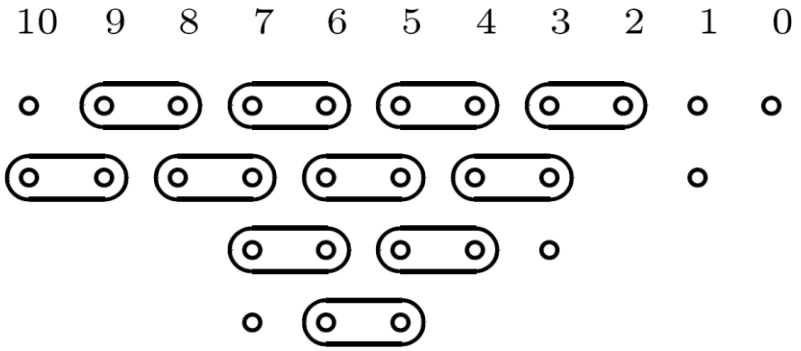
- Number of (3,2) counters can be substantially reduced by taking advantage of the fact that all columns but 1 contain fewer than 6 bits
- Deciding how many counters needed - redraw matrix of bits to be added:



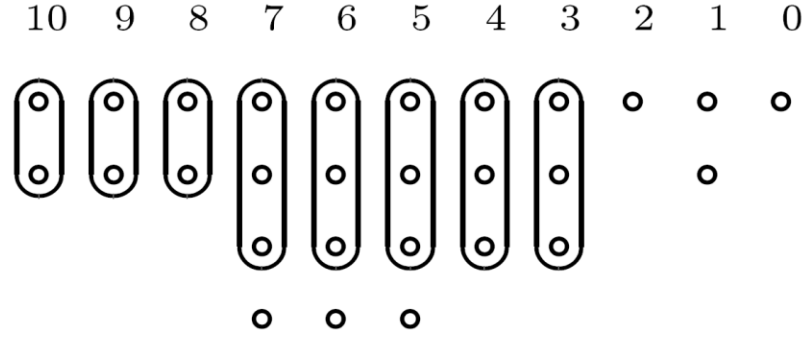
Reduce Complexity - Use (2,2) Counters (HAs)



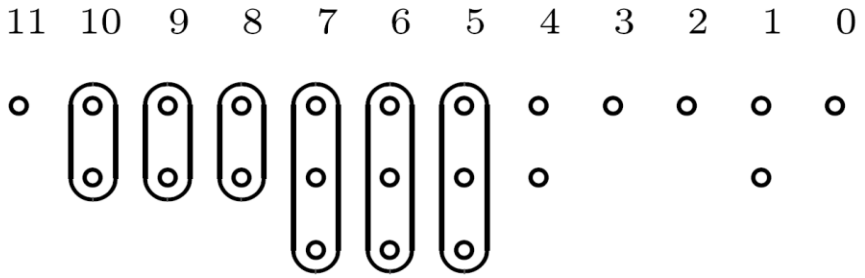
(a) Level 1 carry-save addition.



(b) Results of level 1.



(c) Level 2 carry-save addition.

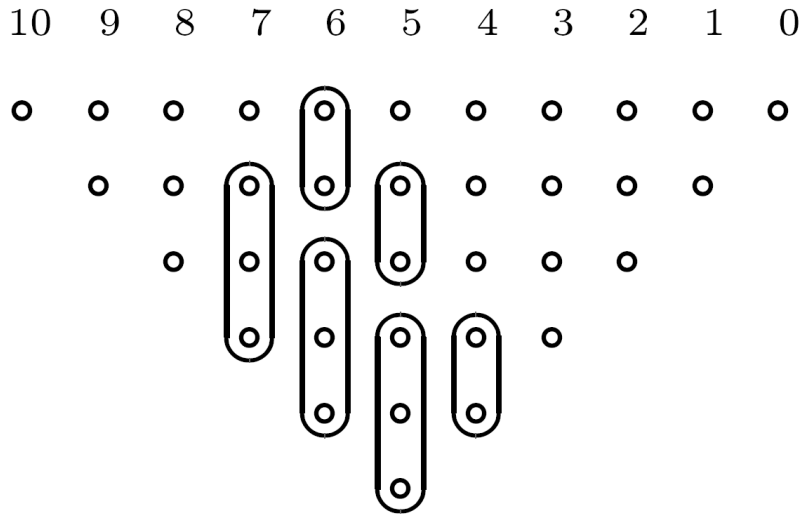


(d) Level 3 carry-save addition.

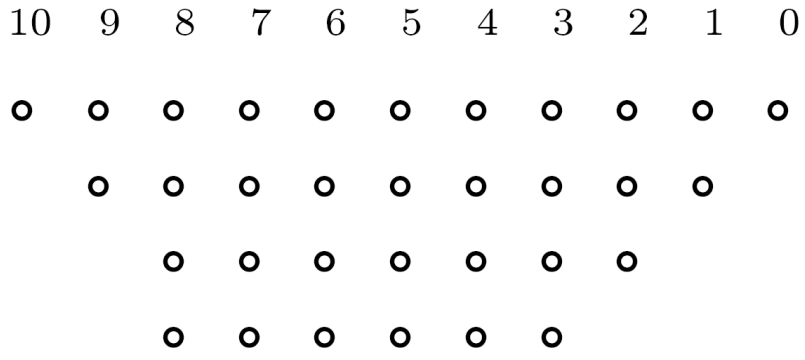
- Number of levels still 3, but fewer counters

Further reduction in number of counters

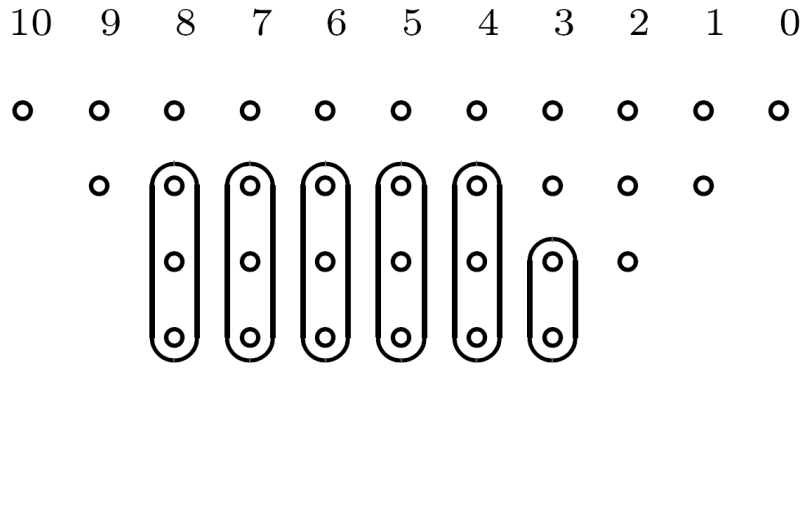
- Reduce # of bits to closest element of 3,4,6,9,13,19,…
- 15 (3,2) and 5 (2,2) vs. 16 (3,2) and 9 (2,2) counters



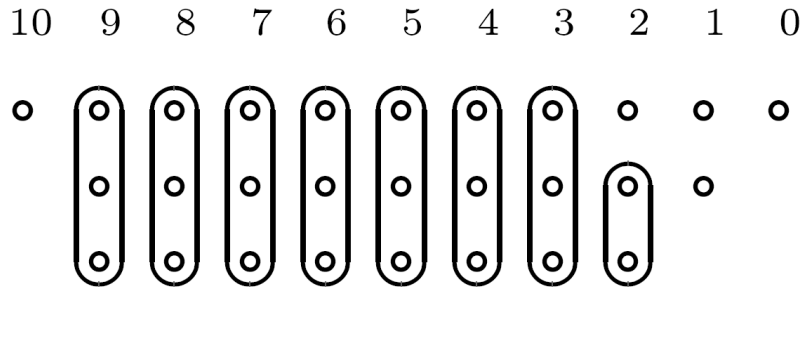
(a) Level 1 carry-save addition.



(b) Results of level 1.



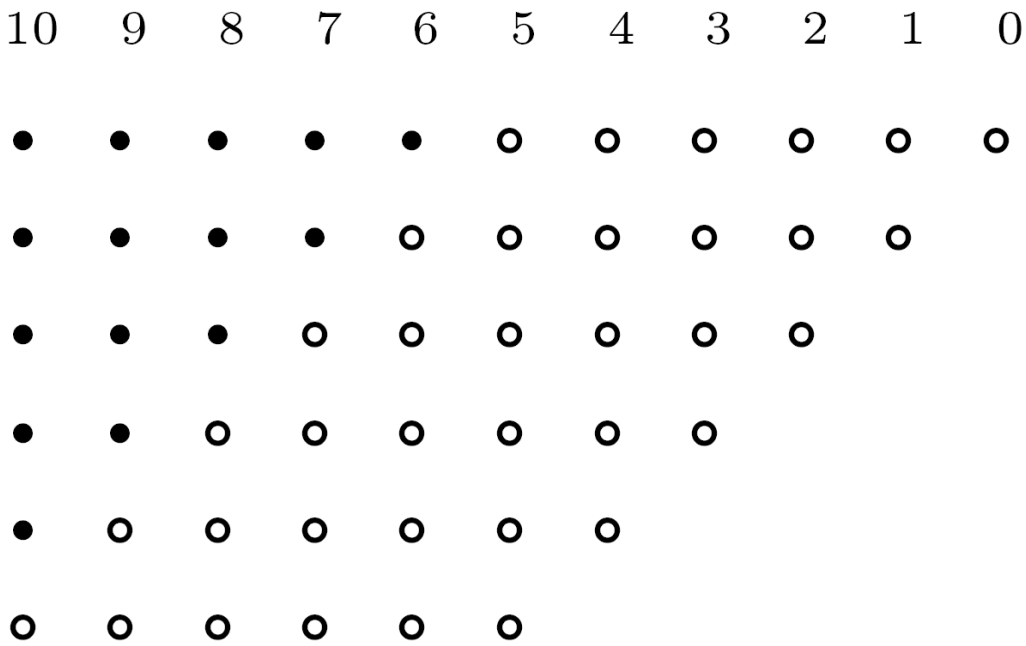
(c) Level 2 carry-save addition.



(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

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

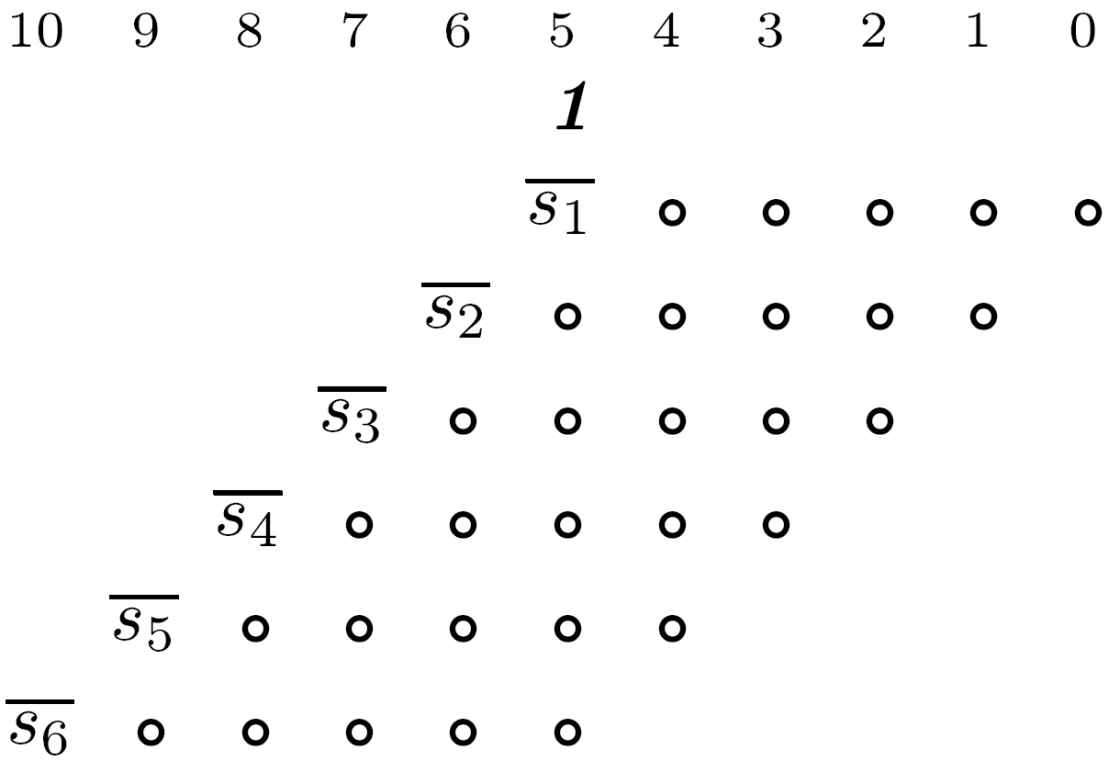
- Replaced by
  0 0 0 0 0 (-s) z_4 z_3 z_2 z_1 z_0
- since

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

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

New Bit Matrix

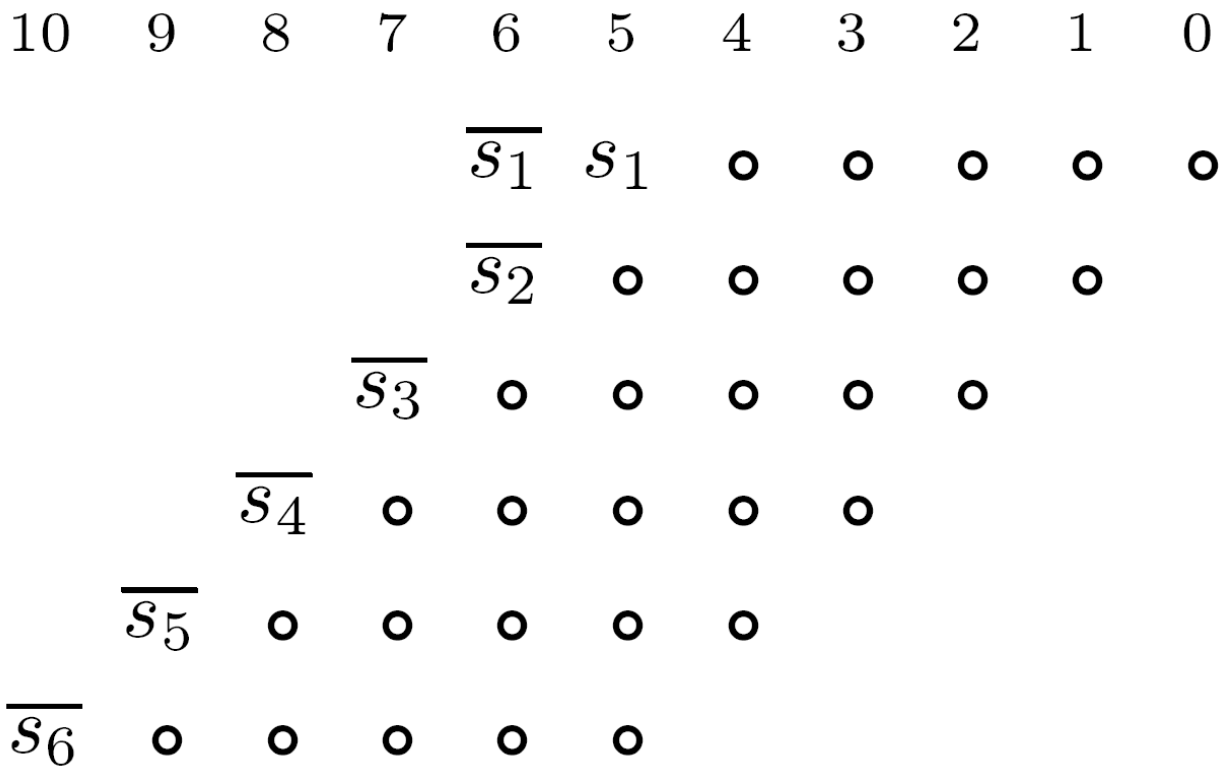
- To get -s in column 5 - complement original s to (1-s) and add 1
  - Carry of 1 into column 6 serves as the extra 1 needed for sign bit of second partial product



- New matrix has fewer bits but higher maximum height (7 instead of 6)

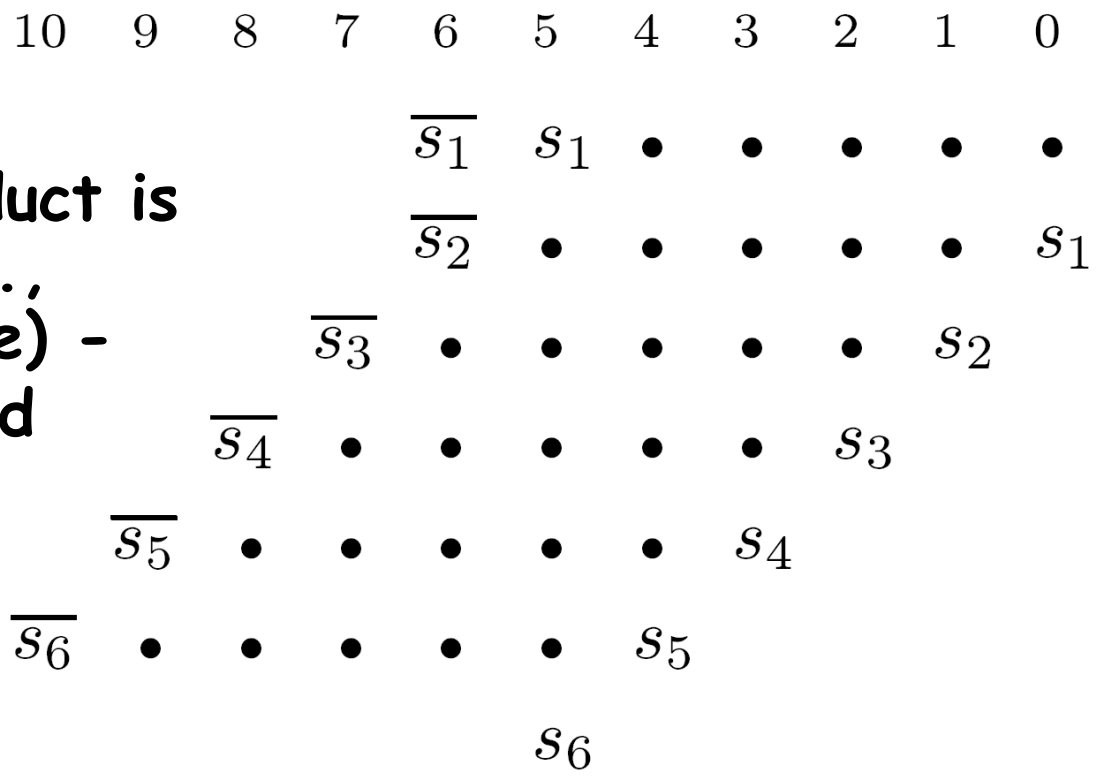
Eliminating Extra 1 in Column 5

- Place two sign bits s_1 and s_2 in same column
- $(1-s_1)+(1-s_2) = 2 - s_1 - s_2$
- 2 is carry out to next column
- Achieved by first extending sign bit s_1



Using One's Complement and Carry

- Add extra carries to matrix
- Full circles - complements of corresponding bits are taken whenever $s_i=1$
- Extra s_6 in column 5 increases maximum column height to 7
- If last partial product is always positive (i.e., multiplier is positive) - s_6 can be eliminated



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
1	**1**	**1**	**1**	**1**	**1**	0	1	0	1	0	
0	**0**	**0**	**0**	**0**	0	0	0	0	0		
1	**1**	**1**	**1**	0	1	0	1	0			
0	**0**	**0**	0	0	0	0	0				
0	**0**	1	0	1	1	0					
0	0	0	0	0	0						
0	0	0	1	1	1	1	0	0	1	0	

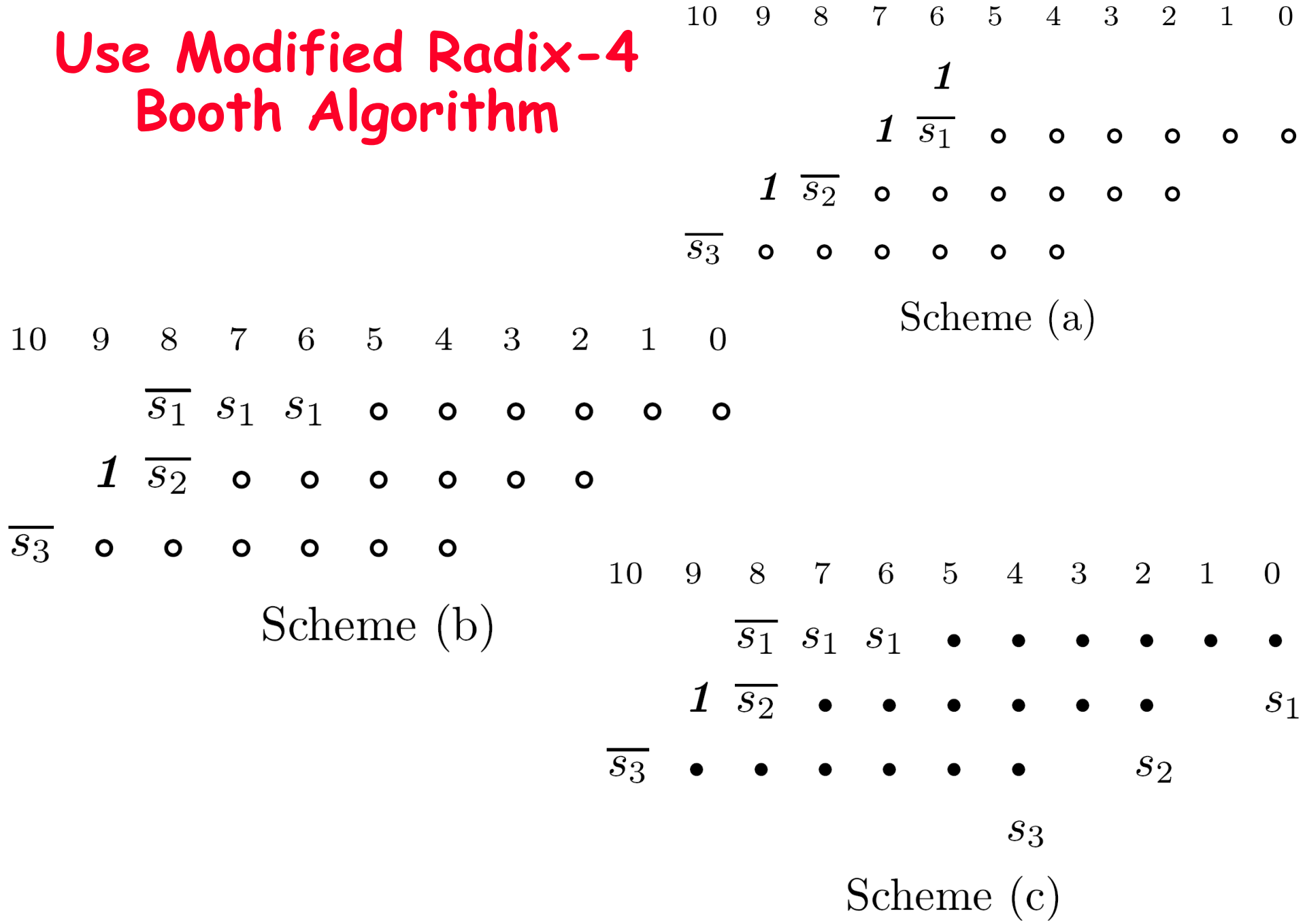
Smaller Matrix for the Example

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

Use Modified Radix-4 Booth Algorithm



Example 2: Using radix-4 modified Booth's

- Same recoded multiplier $0101\bar{1}0\bar{1}$

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

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

Alternative Techniques for Partial Product Accumulation

- Reducing number of levels in tree - speeding up accumulation
- Achieving more regular design
- Tree structures usually have irregular interconnects
  - Irregularity complicates implementation- area-inefficient layouts
- Number of tree levels can be lowered by using reduction rate higher than 3:2
- Achieve 2:1 reduction rate by using SD adders
  - SD adder also generates sum in constant time
  - Number of levels in SD adder tree is smaller
  - Tree produces a single result rather than two for CSA tree

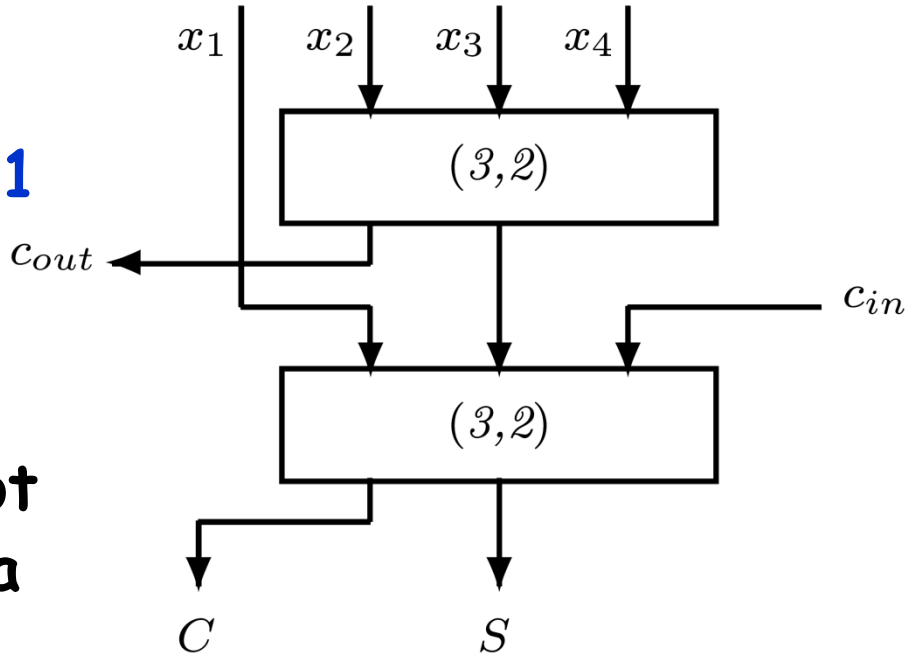
Final Result of SD Tree

- In most cases, conversion to two's complement needed
- Conversion done by forming two sequences:
- First - Z^+ - created by replacing each negative digit of SD number by 0
- Second - Z^- - replaces each negative digit with its absolute value, and each positive digit by 0
- Difference $Z^+ - Z^-$ - found by adding two's complement of Z^- to Z^+ using a CPA
- Final stage of a CPA needed as in CSA tree

SD Adder Tree vs. CSA Tree

- SD - no need for a sign bit extension when negative partial products - no separate sign bit
- Design of SD adder more complex - more gates and larger chip area - each signed digit requires two ordinary bits (or multiple-valued logic)
- Comparison between the two must be made for specific technology
- Example:
  - 32x32 Multiplier based on radix-4 modified Booth's algorithm - 16 partial products
  - CSA tree with 6 levels, SD adder tree with 4 levels
  - Sophisticated logic design techniques and layout schemes result in less area-consuming implementations

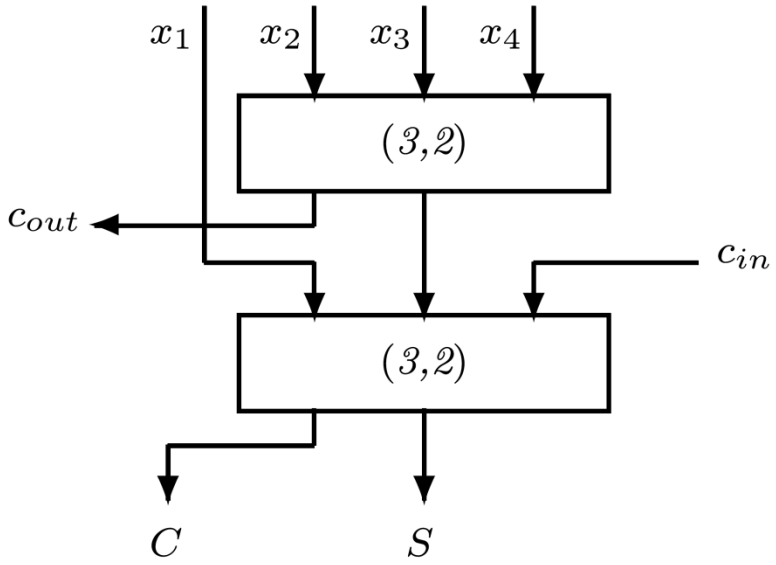
(4;2) Compressors

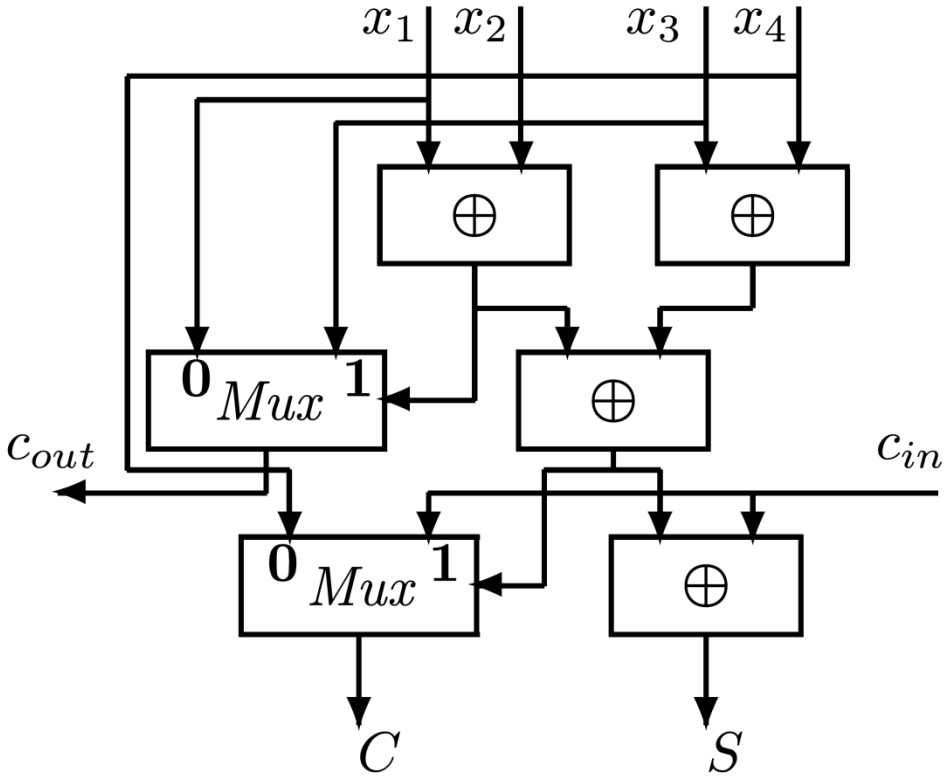


- Same reduction rate of 2:1 achieved without SD representations by using (4;2) compressors
- Designed so that Cout is not a function of Cin to avoid a ripple-carry effect
- (4;2) compressor may be implemented as a multi-level circuit with a smaller overall delay compared to implementation based on 2 (3,2) counters

Example Implementation

- Delay of 3 exclusive-or gates to output S vs. delay of 4 exclusive-or gates
  - 25% lower delay





Other Multi-Level Implementations of a (4;2) Compressor

- All implementations must satisfy

$$x_1 + x_2 + x_3 + x_4 + c_{in} = S + 2(C + c_{out})$$

- Cout should not depend on Cin to avoid horizontal rippling of carries
- Truth table : (a,b,c,d,e,f - Boolean variables)

x_1	x_2	x_3	x_4	c_{out}	C	S
0	0	0	0	0	0	c_{in}
0	0	0	1	0	c_{in}	$\overline{c_{in}}$
0	0	1	0	0	c_{in}	$\overline{c_{in}}$
0	0	1	1	a	$\bar{a}$	c_{in}
0	1	0	0	0	c_{in}	$\overline{c_{in}}$
0	1	0	1	b	$\bar{b}$	c_{in}
0	1	1	0	c	$\bar{c}$	c_{in}
0	1	1	1	1	c_{in}	$\overline{c_{in}}$
1	0	0	0	0	c_{in}	$\overline{c_{in}}$
1	0	0	1	d	$\bar{d}$	c_{in}
1	0	1	0	e	$\bar{e}$	c_{in}
1	0	1	1	1	c_{in}	$\overline{c_{in}}$
1	1	0	0	f	$\bar{f}$	c_{in}
1	1	0	1	1	c_{in}	$\overline{c_{in}}$
1	1	1	0	1	c_{in}	$\overline{c_{in}}$
1	1	1	1	1	1	c_{in}

- Previous implementation - a=b=c=1, d=e=f=0

Comparing Delay of Trees

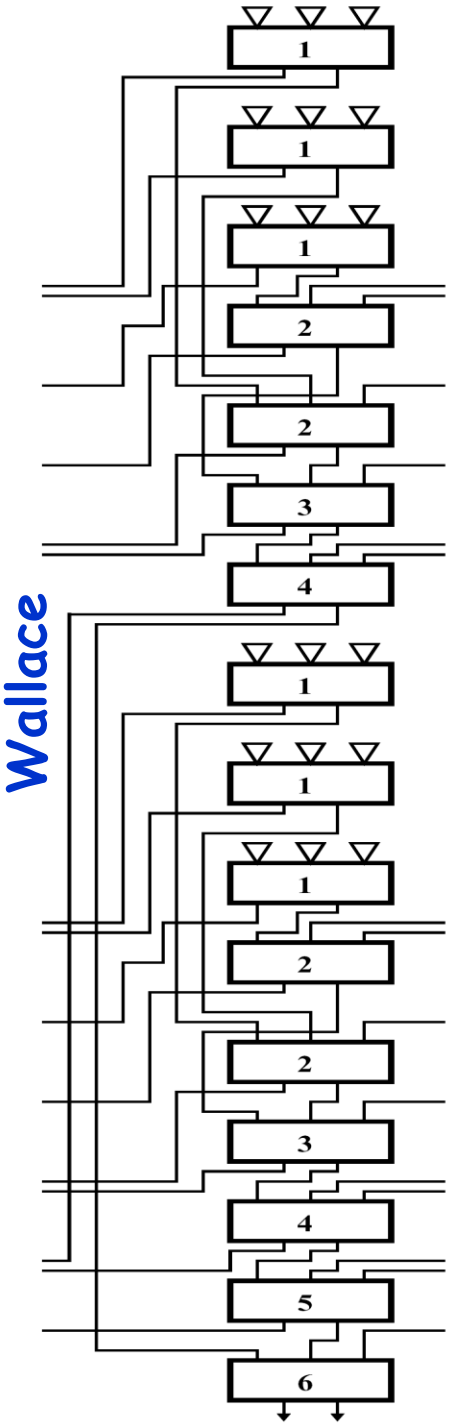
Number of operands	Number of levels using (3,2)	Number of levels using (4;2)	Equivalent delay
3	1	1	1.5
4	2	1	1.5
5 – 6	3	2	3
7 – 8	4	2	3
9	4	3	4.5
10 – 13	5	3	4.5
14 – 16	6	3	4.5
17 – 19	6	4	6
20 – 28	7	4	6
29 – 32	8	4	6
33 – 42	8	5	7.5

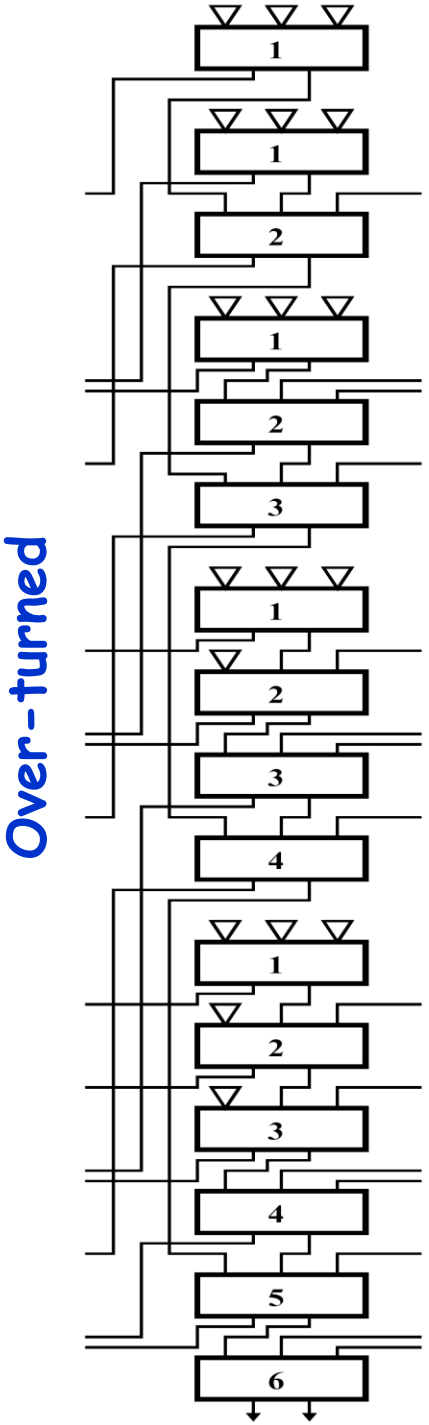
Other Implementations

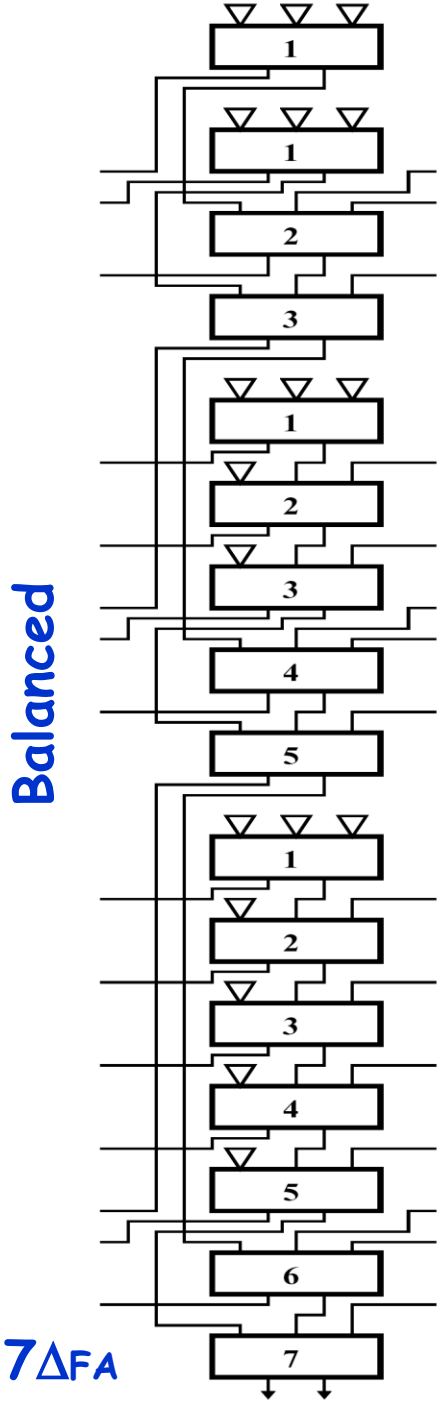
- Other counters and compressors can be used: e.g., (7,3) counters
- Other techniques suggested to modify CSA trees which use (3,2) counters to achieve a more regular and less area-consuming layout
- Such modified tree structures may require a somewhat larger number of CSA levels with a larger overall delay
- Two such techniques are:
  - Balanced delay trees
  - Overturned-stairs trees

Bit-Slices for Three Techniques

* 18 operands
* radix-4 modified Booth
* Triangles - multiplexers
* Rectangles - (3,2) counters
* 15 outgoing & incoming carries aligned
* adjacent bit-slices abut





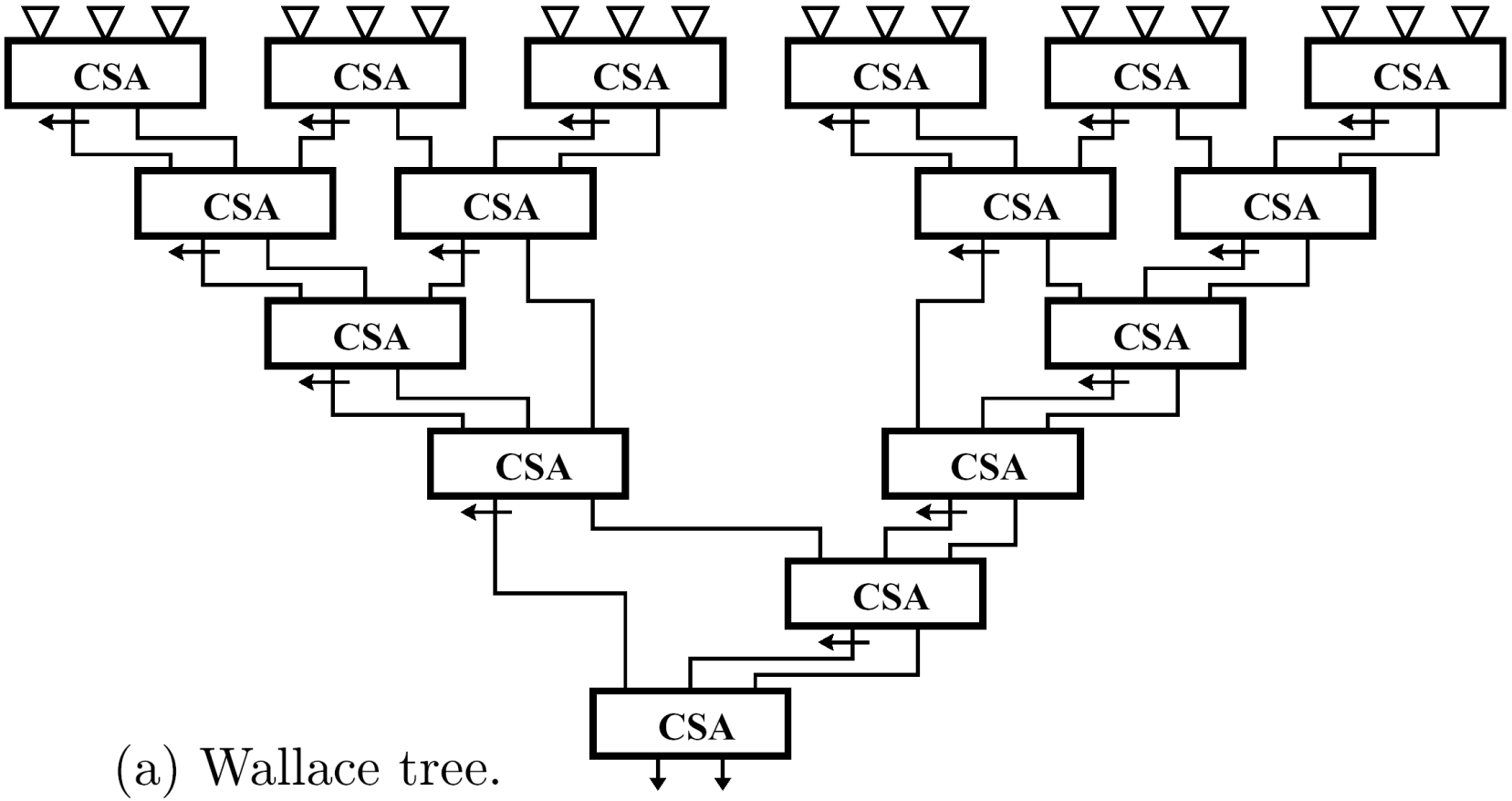


7ΔFA

Comparing the Three Trees

- Incoming carries routed so that all inputs to a counter are valid before or at necessary time
- Only for balanced tree - all 15 incoming carries generated exactly when required - all paths balanced
- In other 2 - there are counters for which not all incoming carries are generated simultaneously
  - For example, bottom counter in overturned-stairs - incoming carries with delays of $4\Delta_{FA}$ and $5\Delta_{FA}$
- Number of wiring tracks between adjacent bit-slices (affect layout area)
  - Wallace tree requires 6; overturned-stairs 3; balanced tree 2 tracks
- Tradeoff between size and speed
  - Wallace : lowest delay but highest number of wiring tracks
  - Balanced: smallest number of wiring tracks but highest delay

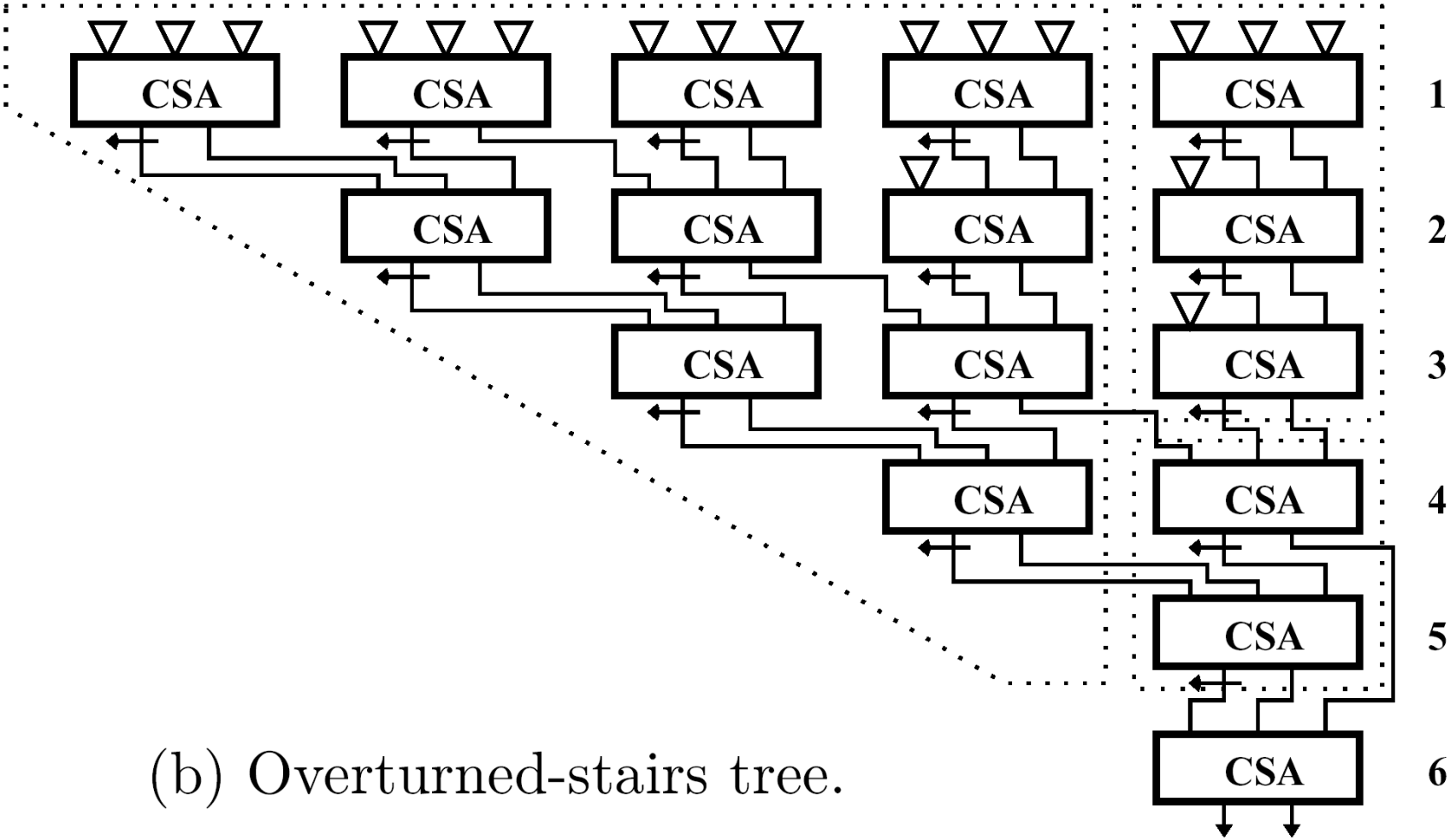
Complete Structure of Wallace Tree



(a) Wallace tree.

- Balanced and overturned-stairs have regular structure - can be designed in a systematic way

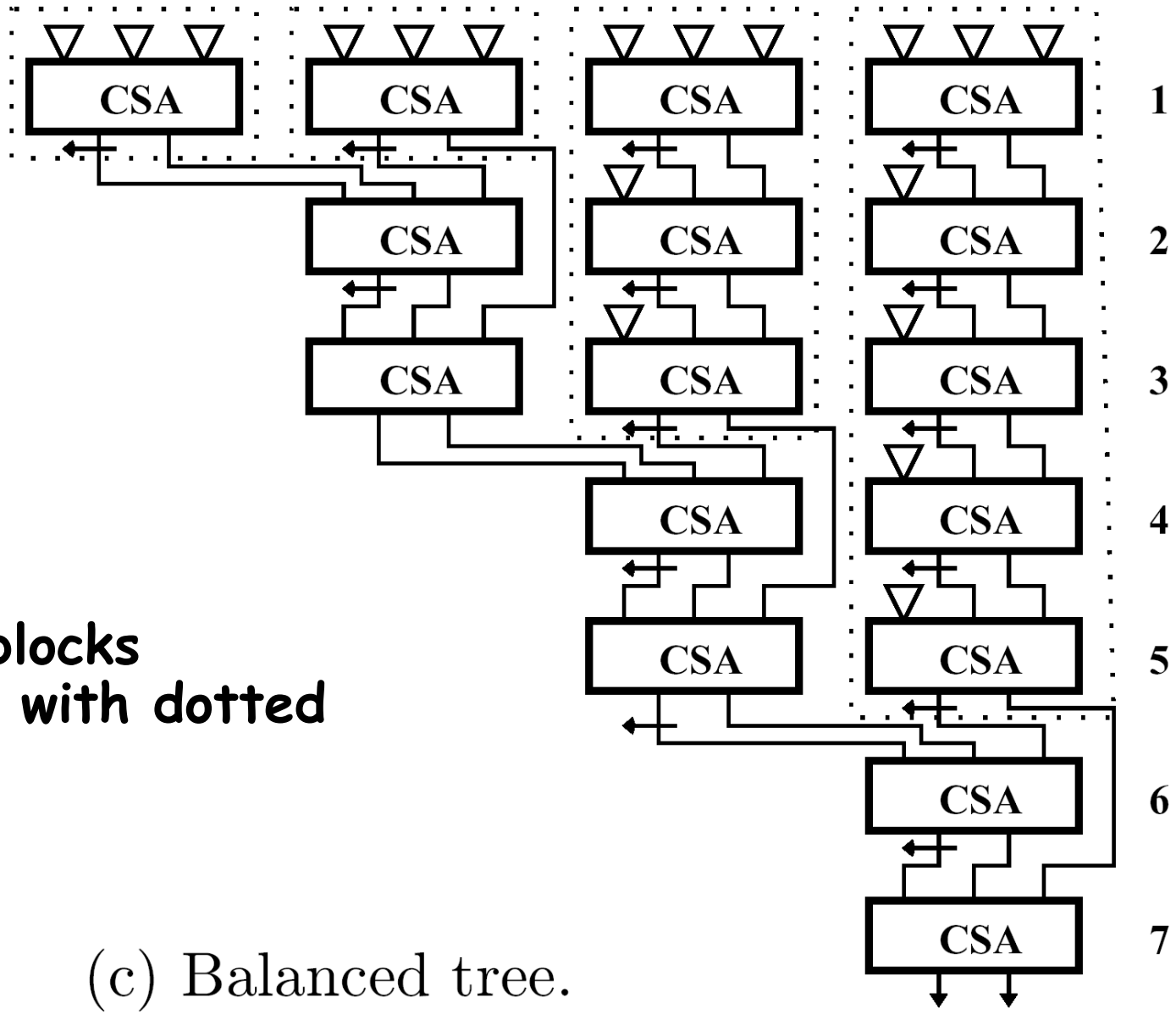
Complete Structure of Over-turned Tree



(b) Overturned-stairs tree.

- Building blocks indicated with dotted lines

Complete Structure of Balanced Tree

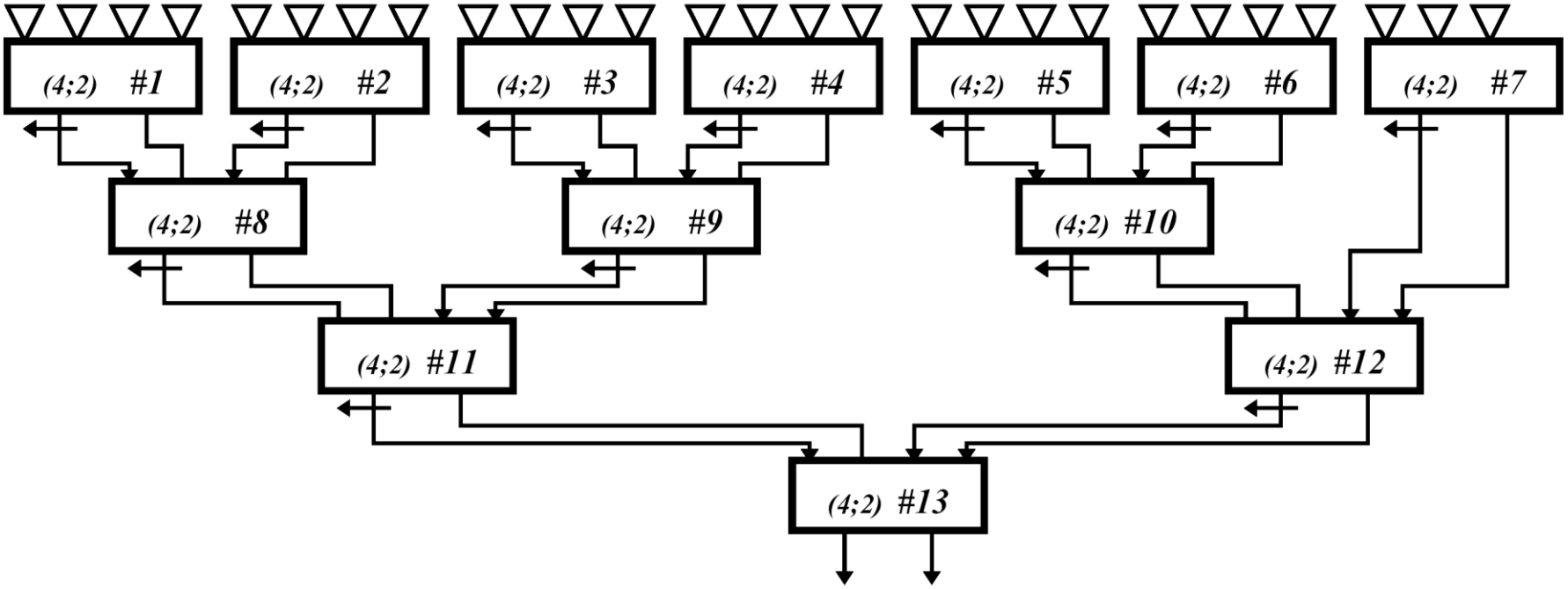


- Building blocks indicated with dotted lines

(c) Balanced tree.

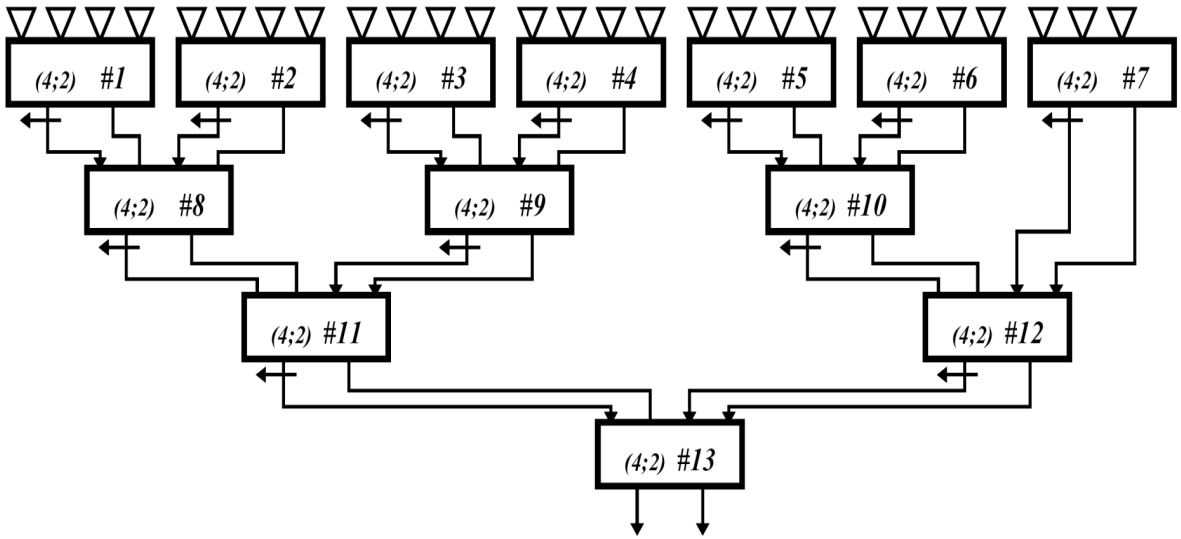
Layout of CSA Tree

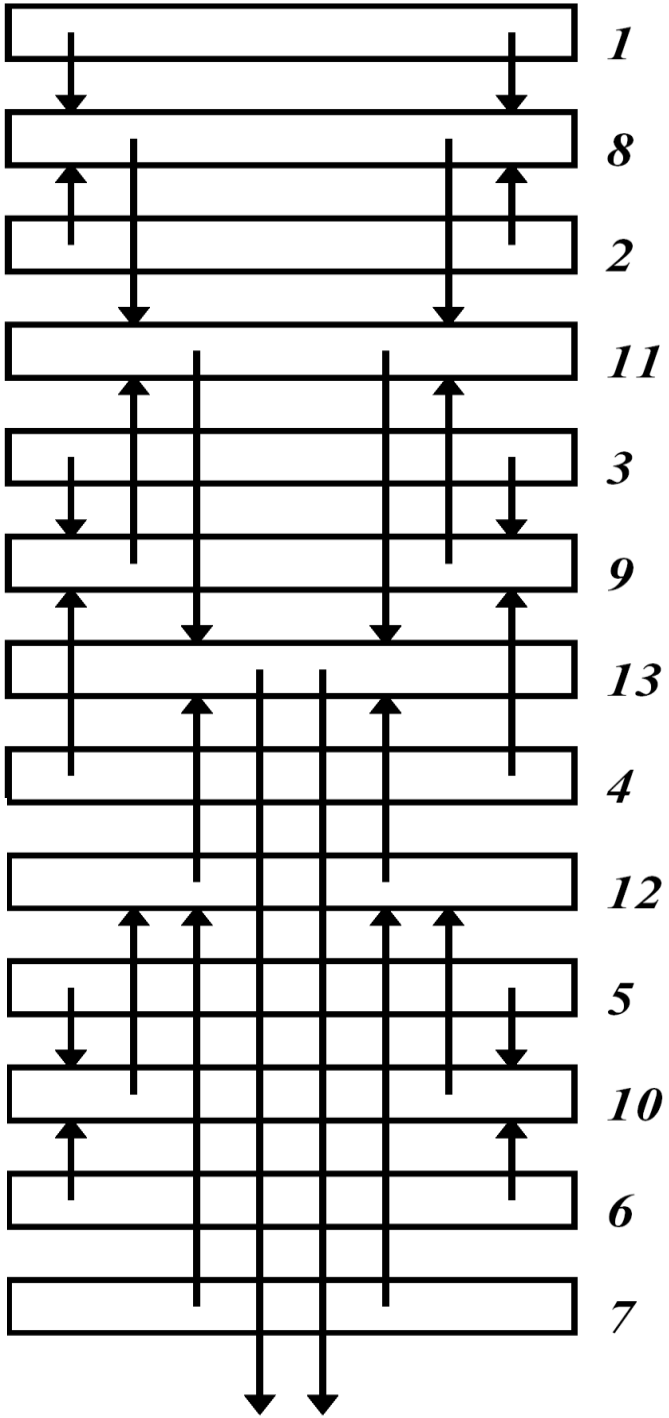
- Wires connecting carry-save adders should have roughly same length for balanced paths
- CSA tree for 27 operands constructed of (4;2) compressors



Layout of CSA Tree

- Bottom compressor (#13) is located in middle so that compressors #11 and #12 are roughly at same distance from it
- Compressor #11 has equal length wires from #8 and #9

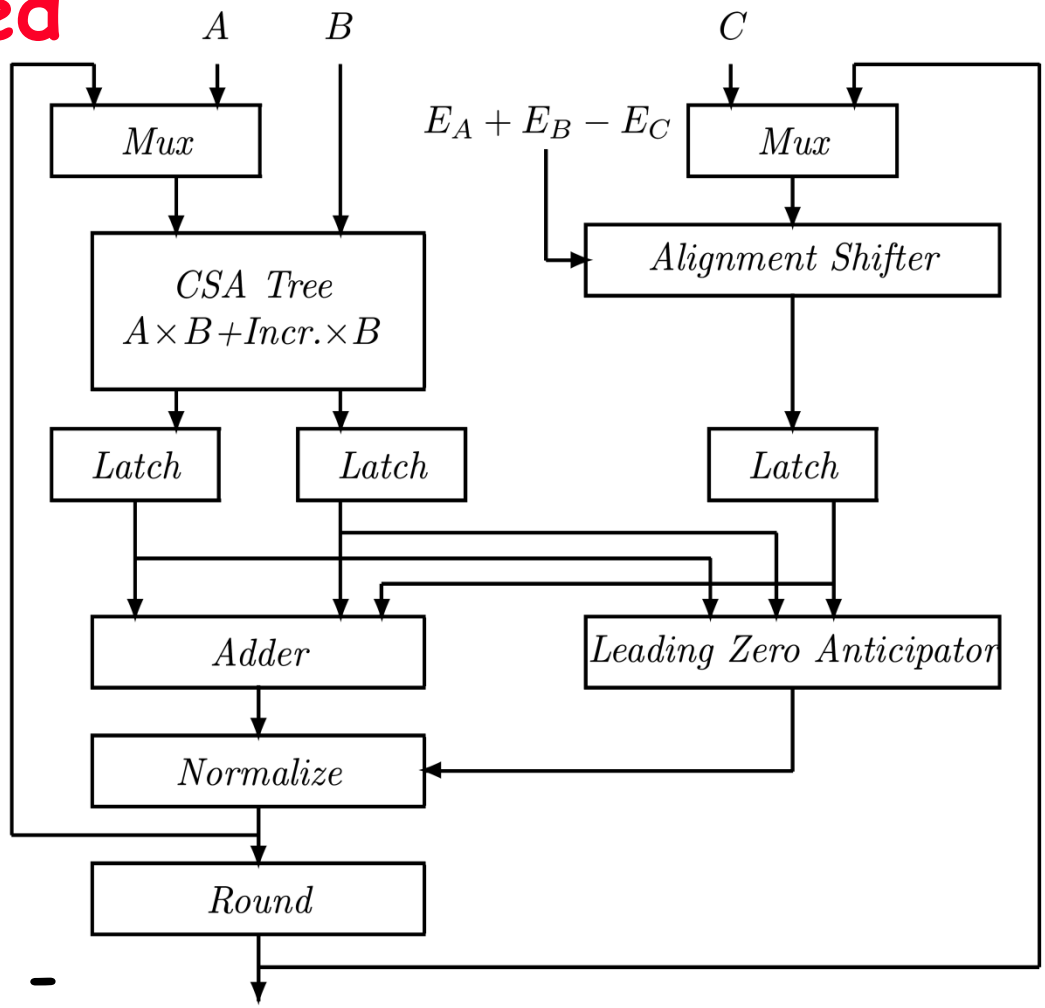




Fused Multiply-Add Unit

- Performs $A \times B$ followed by adding C
  - $A \times B + C$ done as single and indivisible operation
- Multiply only: set $C=0$; add (subtract) only: set $B=1$
  - Can reduce overall execution time of chained multiply and then add/subtract operations
- Example: Evaluation of a polynomial $a_n x^n + a_{n-1} x^{n-1} + \ldots + a_0$ through $[(a_n x + a_{n-1})x + a_{n-2}]x + \ldots$
- Independent multiply and add operations can not be performed in parallel
- Another advantage for floating-point operations - rounding performed only once for $A \times B + C$ rather then twice for multiply and add
  - Rounding introduces computation errors - reducing number of roundings reduces overall error

Implementating Fused Multiply-Add Unit



- A,B,C - significands; E_A, E_B, E_C - exponents of operands
- CSA tree generates partial products and performs carry-save accumulation to produce 2 results which are added with properly aligned C
- Adder gets 3 operands - first reduces to 2 ((3,2) counters), then performs carry-propagate addition
- Post-normalization and rounding executed next

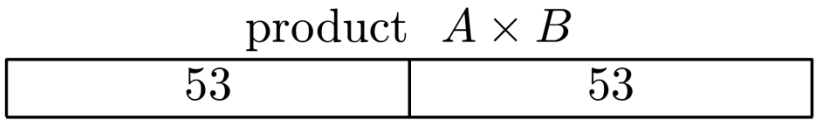
Two Techniques to reduce Execution Time

- First: leading zero anticipator circuit uses propagate and generate signals produced by adder to predict type of shift needed in post-normalization step
- It operates in parallel to addition so that the delay of normalization step is shorter
- Second (more important): alignment of significand C in $E_A+E_B-E_C$ done in parallel to multiplication
- Normally, align significand of smaller operand (smaller exponent)
- Implying: if $A \times B$ smaller than C, have to shift product after generation - additional delay

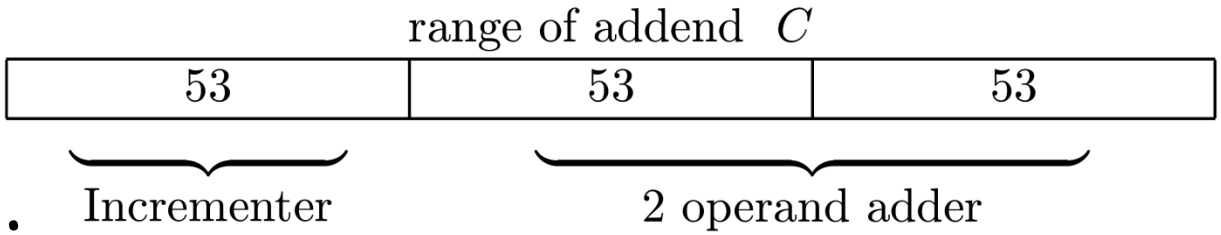
Instead - Always align C

- Even if larger than AxB - allow shift to be performed in parallel to multiplication
- Must allow C to shift either to right (traditional) or left
- Direction - $E_A+E_B-E_C$ is positive or negative
- If C shifted to left - must increase total number of bits in adder

Example



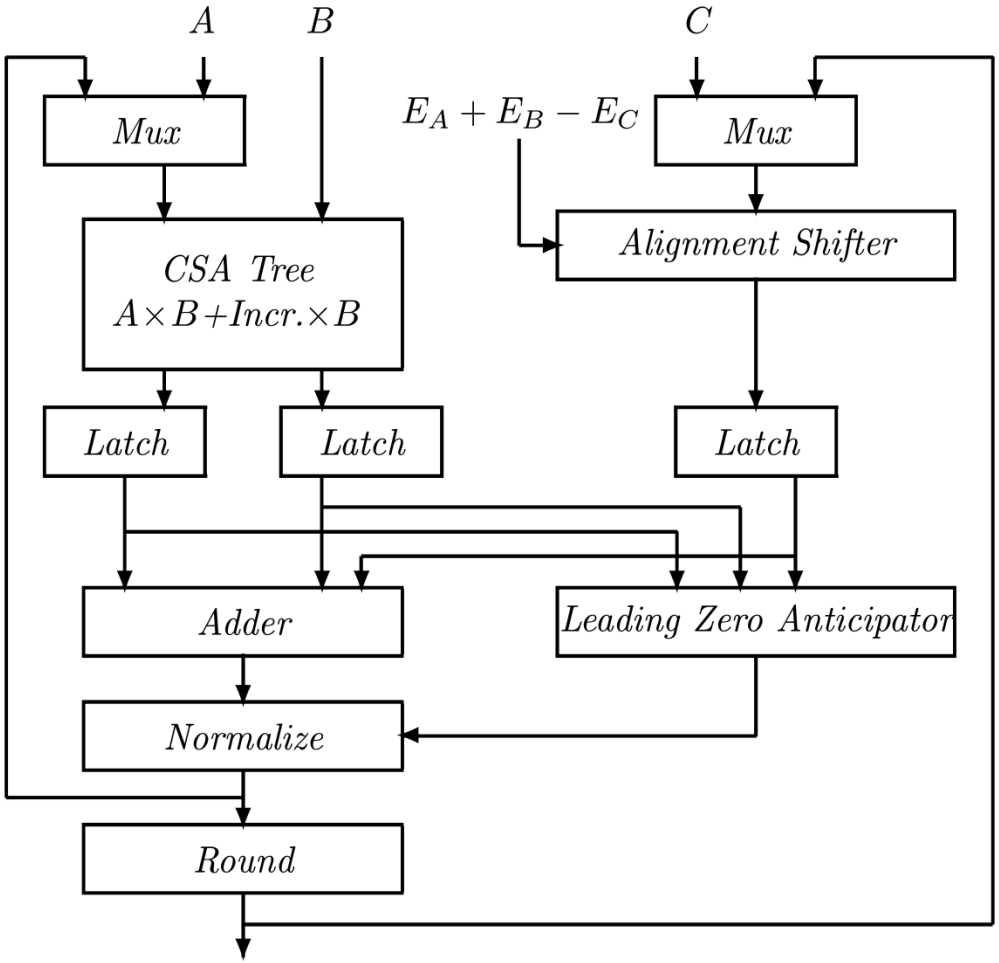
- Long IEEE operands - possible range of C relative to $A \times B$:



- $53 \geq E_A + E_B - E_C \geq -53$
- If $E_A + E_B - E_C \geq 54$, bits of C shifted further to right will be replaced by a sticky bit, and if $E_A + E_B - E_C \leq -54$, all bits of $A \times B$ replaced by sticky bit
- Overall penalty - 50% increase in width of adder - increasing execution time
- Top 53 bits of adder need only be capable of incrementing if a carry propagates from lower 106 bits

Additional Computation Paths

- Path from Round to multiplexer on right used for (XxY+Z)+AxB
- Path from Normalize to multiplexer on left used for (X x Y+Z) x B +C
- Rounding step for (X x Y+Z) is performed at same time as multiplication by B, by adding partial product Incr.x B to CSA tree



Digital Computer Arithmetic

Part 6c
High-Speed Multiplication - III

Soo-Ik Chae

Spring 2009

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

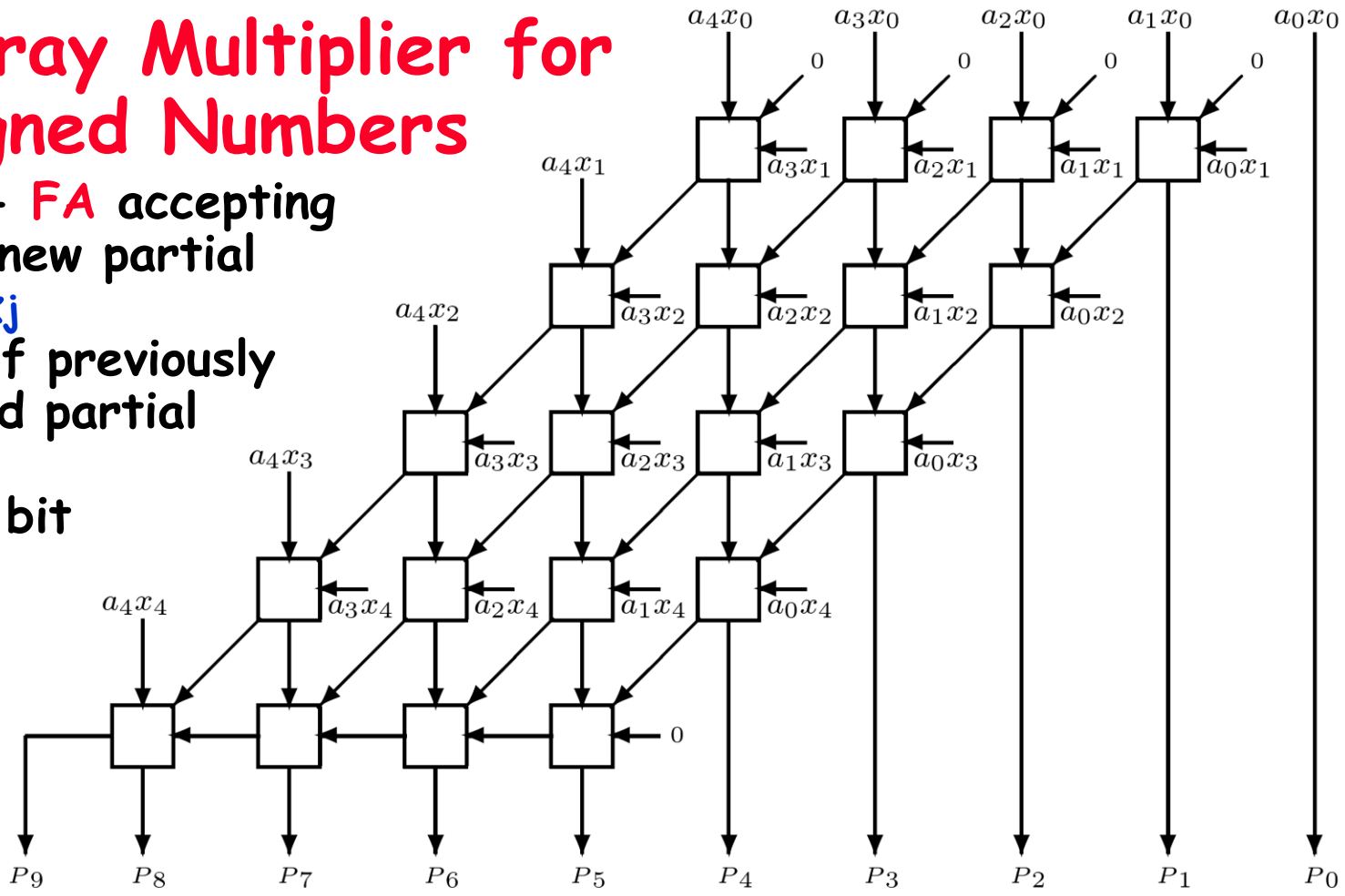
Illustration - 5 x 5 Multiplication

					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

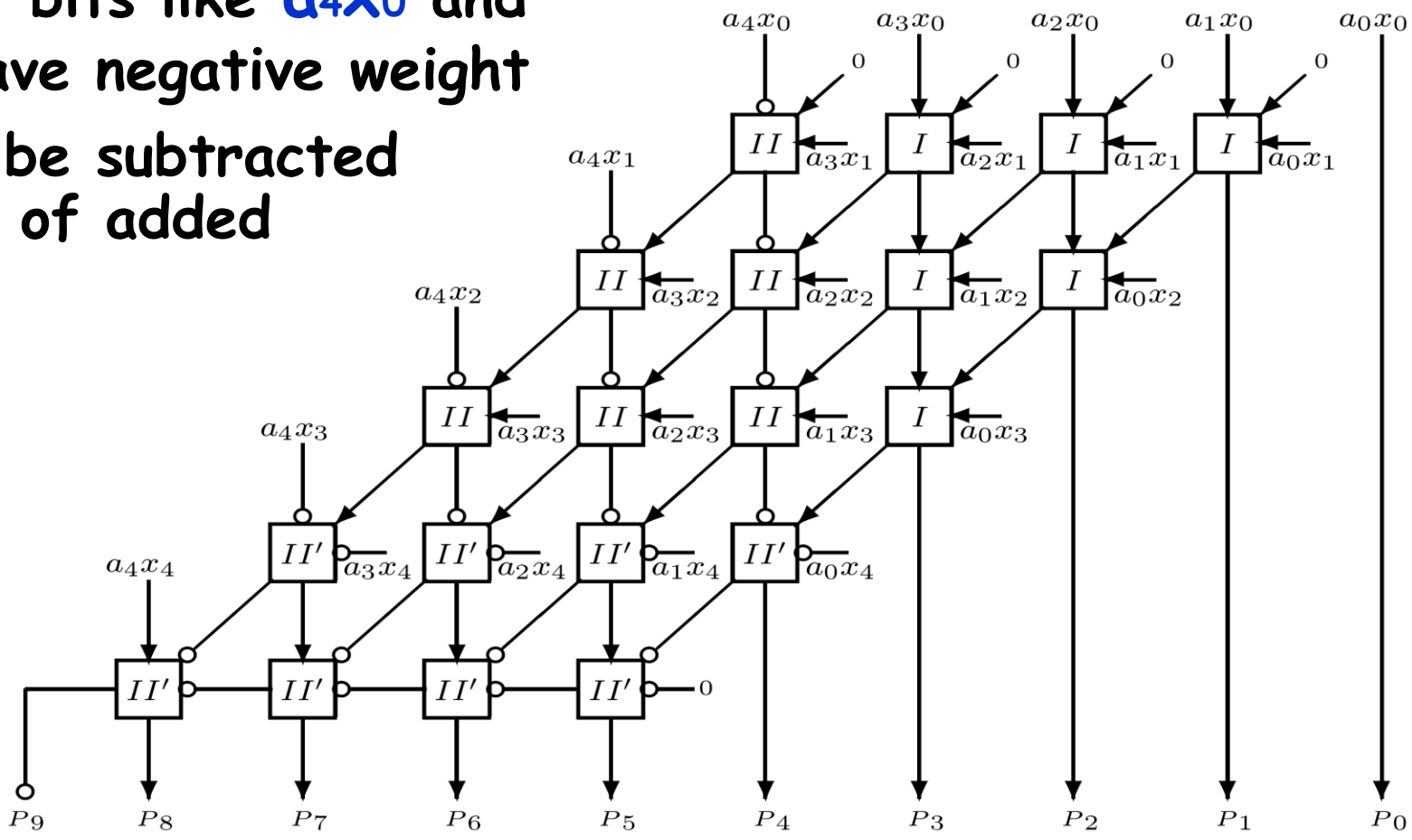
- Basic cell - FA accepting one bit of new partial product $a_i x_j$ + one bit of previously accumulated partial product + carry-in bit



- 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)

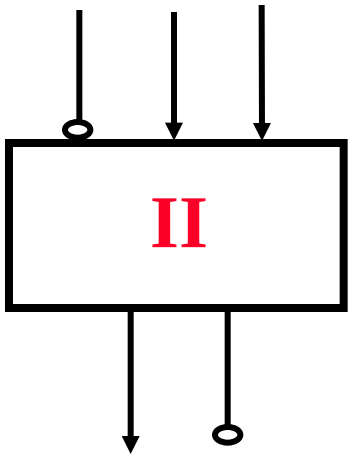
Array Multiplier for Two's Complement Numbers

- 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 -



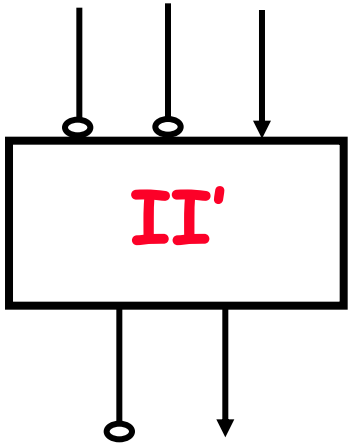
$$x + y - z = 2c - s$$

- s and c outputs given by

$$s = (x + y - z) \bmod 2 \qquad c = \frac{2}{(x + y - z) + s}$$

Type I' and II' Cells

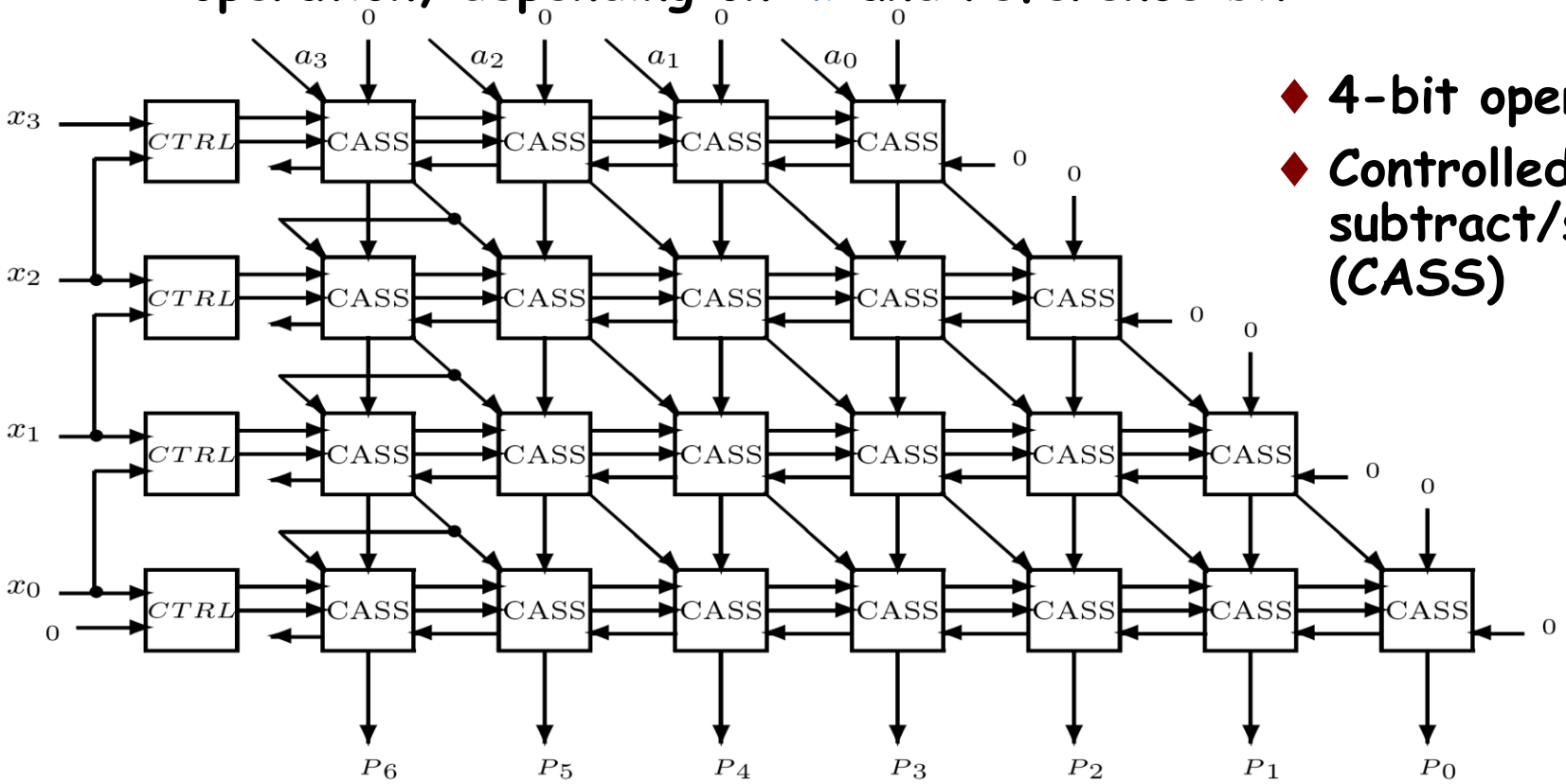
- Type II' cells: 2 negative inputs and 1 positive
- Sum of inputs varies from -2 to 1
  - c output has weight -2
  - s output has weight +1



- 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

Booth's Algorithm Array Multiplier

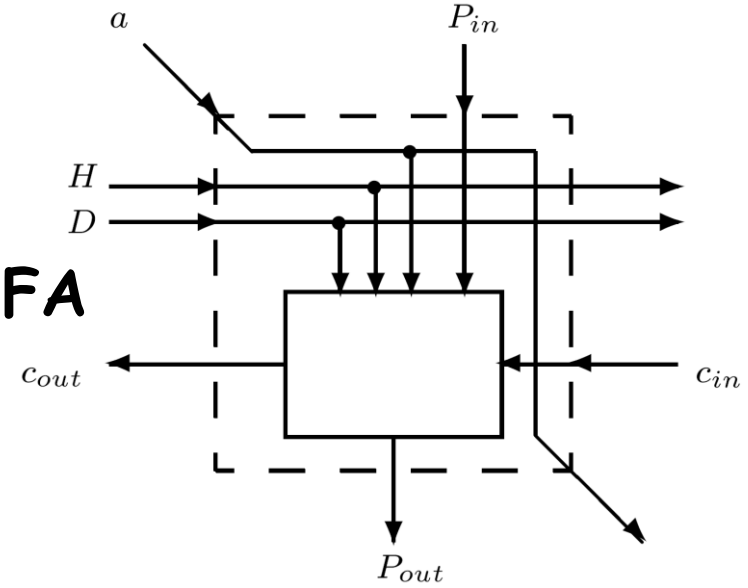
- For two's complement operands
- n rows of basic cells - each row capable of adding or subtracting a properly aligned multiplicand to previously accumulated partial product
  - Cells in row i perform an add, subtract or transfer-only operation, depending on xi and reference bit



- 4-bit operands
- Controlled add/ subtract/shift (CASS)

Controlled add/subtract/shift - CASS

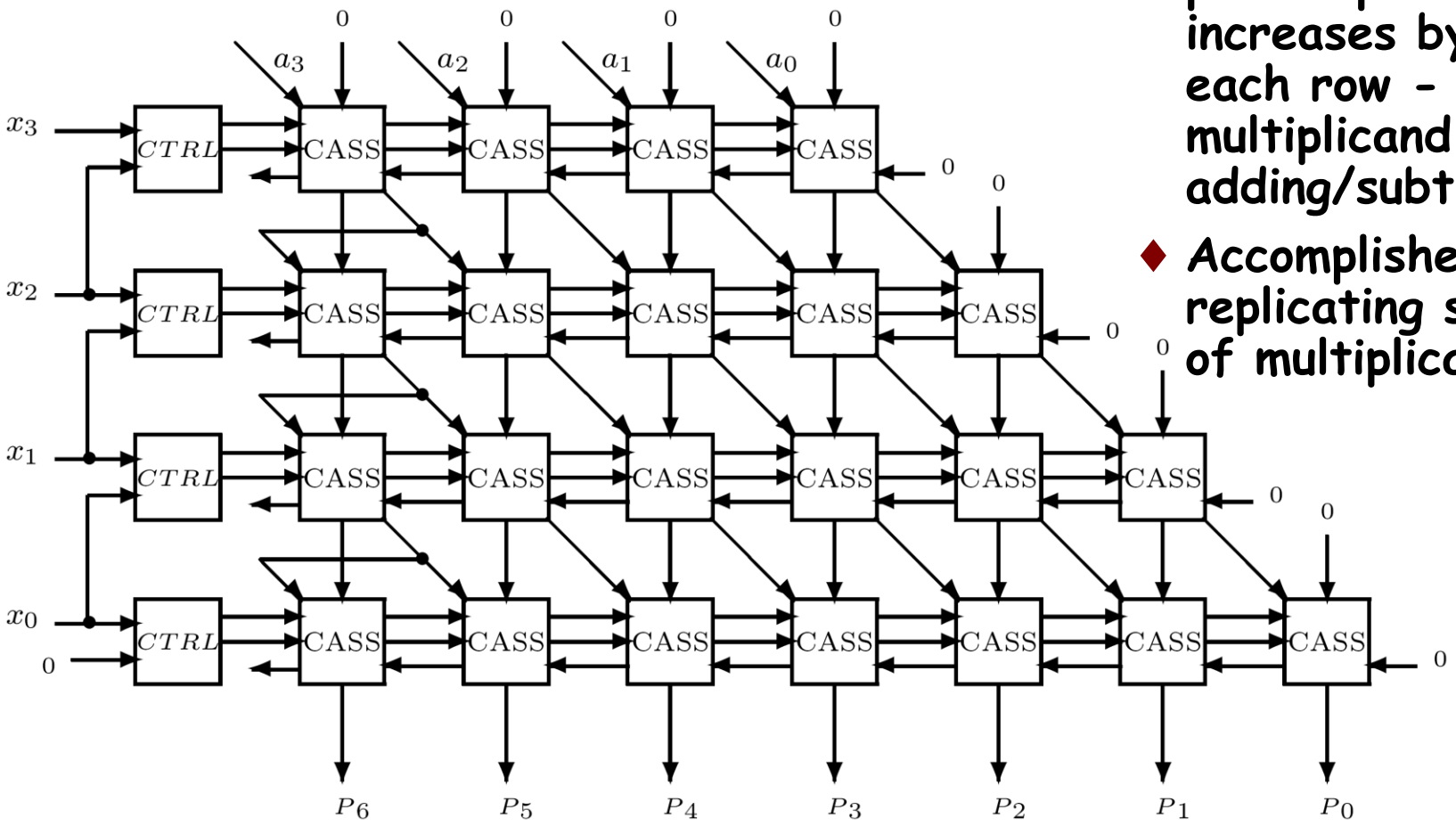
- H and D: control signals indicating type of operation
- H=0: no arithmetic operation done
- H=1: arithmetic operation performed - new Pout
  - Type of arithmetic operation indicated by D signal
  - D=0: multiplicand bit, a, added to Pin with cin as incoming carry - generating Pout and cout as outgoing carry
  - D=1: multiplicand bit, a, subtracted from Pin with incoming borrow and outgoing borrow
- Pout=Pin⊕(a H)⊕(cin H)
  cout=(Pin⊕D)(a+cin) + a cin
- Alternative: combination of multiplexer (0, +a and -a) and FA
- H and D generated by CTRL - based on xi and reference bit x{i-1}



Controlled add/subtract/shift (CASS) cell

Booth's Algorithm Array Multiplier - details

- First row - most significant bit of multiplier
- Resulting partial product need be shifted left before adding/subtracting next multiple of multiplicand
- A new cell with input Pin=0 is added
- Number of bits in partial product increases by one each row - expand multiplicand before adding/subtracting it
- Accomplished by replicating sign bit of multiplicand



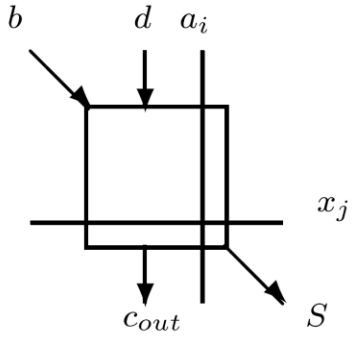
Properties and Delay

- Cannot take advantage of strings of 0's or 1's - cannot eliminate or skip rows
- Only advantage: ability to multiply negative numbers in two's complement with no need for correction
- Operation in row i need not be delayed until all upper (i-1) rows have completed their operation
- P0, generated after one CASS delay (plus delay of CTRL), P1 generated after two CASS delays, and P{2n-2}, generated after (2n-1) CASS delays
- Similarly can implement higher-radix multiplication requiring less rows
- Building block: multiplexer-adder circuit that selects correct multiple of multiplicand A and adds it to previously accumulated partial product

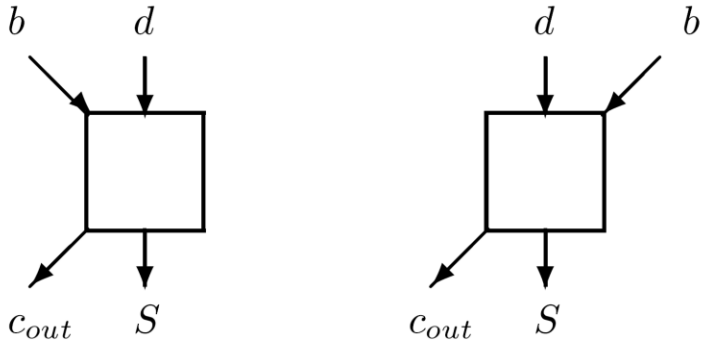
Pipelining

- Important characteristic of array multipliers - allow pipelining
- Execution of separate multiplications overlaps
- The long delay of carry-propagating addition must be minimized
- Achieved by replacing CPA with several additional rows - allow carry propagation of only one position between consecutive rows
- To support pipelining, all cells must include latches - each row handles a separate multiplier-multiplicand pair
- Registers needed to propagate multiplier bits to their destination, and propagate completed product bits

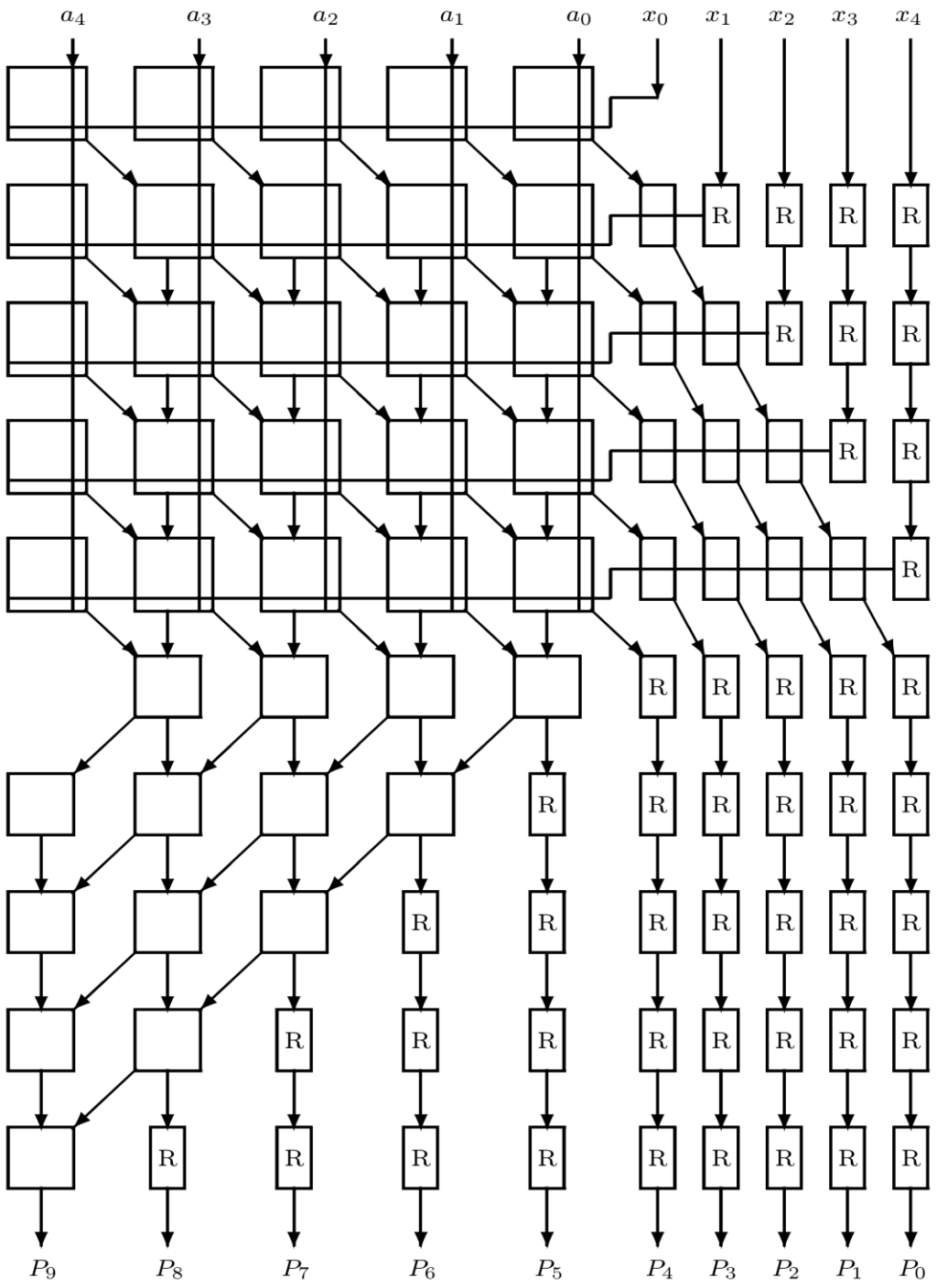
Pipelined Array Multiplier



Latched full adder with an AND gate.



Latched half adders.



Optimality of Multiplier Implementations

- Bounds on performance of algorithms for multiplication
- Theoretical bounds for multiplication similar to those for addition
- Adopting the idealized model using (f,r) gates:
- Execution time of a multiply circuit for two operands with n bits satisfies
- $T_{mult} \geq \lceil \log_f 2n \rceil$
- If residue number system is employed:
- $T_{mult} \geq \lceil \log_f 2m \rceil$
- m - number of digits needed to represent largest modulus in residue number system

Optimal Implementations

- Need to compare performance (execution time) and implementation costs (e.g., regularity of design, total area, etc.)
- Objective function like $A\,T$ can be used
- A - area and T - execution time
- A more general objective function: $A\,T^{\alpha}$
  - α can be either smaller or larger than 1

Basic Array Multiplier

- Very regular structure - can be implemented as a rectangular-shaped array - no waste of chip area
- n least significant bits of final product are produced on right side of rectangle; n most significant bits are outputs of bottom row of rectangle
- Highly regular and simple layout but has two drawbacks:
  - Requires a very large area, proportional to n^2, since it contains about n^2 FAs and AND gates
  - Long execution time T of about $2n\,\Delta_{FA}$ (Δ_{FA} - delay of FA)
- More precisely, T consists of $(n-1)\Delta_{FA}$ for first $(n-1)$ rows and $(n-1)\Delta_{FA}$ for CPA (ripple-carry adder)
- AT is proportional to n^3

Pipelined & Booth Array Multipliers

- Required area increases even further (CPA replaced)
- Latency of a single multiply operation increases
- However, pipeline period ($\Rightarrow$ pipeline rate) shorter
- Booth based array multiplier offers no advantage
  - A - order of n^2 and T - linear in n
- Radix-4 Booth can potentially be better - only $n/2$ rows - could reduce T and A by factor of two
- However, actual delay & area higher - recoding logic and, more importantly, partial product selectors, add complexity & interconnections - longer delay per row
- Also, since relative shift between adjacent rows is two bits, must allow carry to propagate horizontally
  - Can be achieved locally or in last row - then carry propagation through $2n-1$ bits (instead of $n-1$)
- Exact reduction depends on design and technology

Radix-8 Booth & CSA Tree

- Similar problems with radix-8 Booth's array multiplier
  - In addition, 3A should be precalculated
  - Reduction in delay and area may be less than expected 1/3
  - Still, may be cost-effective in certain technologies and design styles
- Partial products can be accumulated using a cascade or a tree structure with shorter execution time
- But CSA tree structures have irregular interconnects - no area-efficient layout with a rectangular shape
- Moreover - overall width 2n usually required - multiplier area of order $2n \log k$
- AT may increase as $2n \log^2 k$

Delay of Balanced Delay Tree

- **Balanced delay tree** - more regular structure
  - Increments in number of operands - 3,3,5,7,9...
  - Sum of series - order of $k=j^2$ (j - number of elements in series, k - number of operands)
- Number of levels - determines overall delay - linear in $j=\sqrt{k}$
- Compare to log k - number of levels in complete binary tree
- **Proof**: exercise
- Above expressions - theoretical, limited practical significance
- Detailed analysis of alternative designs is necessary for specific technology