



UNIVERSITY OF MASSACHUSETTS
Dept. of Electrical & Computer Engineering

Digital Computer Arithmetic

ECE 666

Part 6c
High-Speed Multiplication - III

Israel Koren
Spring 2008

ECE666/Koren Part.6c.1

Copyright 2008 Koren

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

ECE666/Koren Part.6c.2

Copyright 2008 Koren

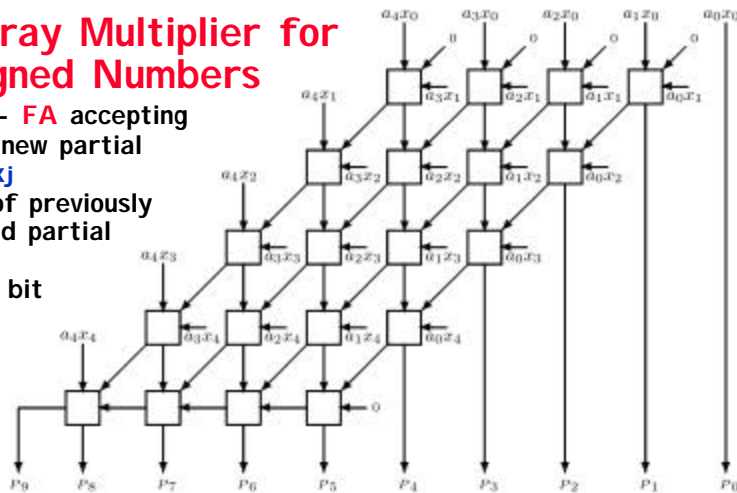
Illustration - 5 x 5 Multiplication

					a_4	a_3	a_2	a_1	a_0
	×				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
($a_4x_0, a_3x_0, \dots, a_0x_0$ and $a_4x_1, a_3x_1, \dots, a_0x_1$)
in row 1 after proper alignment
 - * The results of row 1 are then added to
 $a_4x_2, a_3x_2, \dots, a_0x_2$ in row 2, and so on

5 x 5 Array Multiplier for Unsigned Numbers

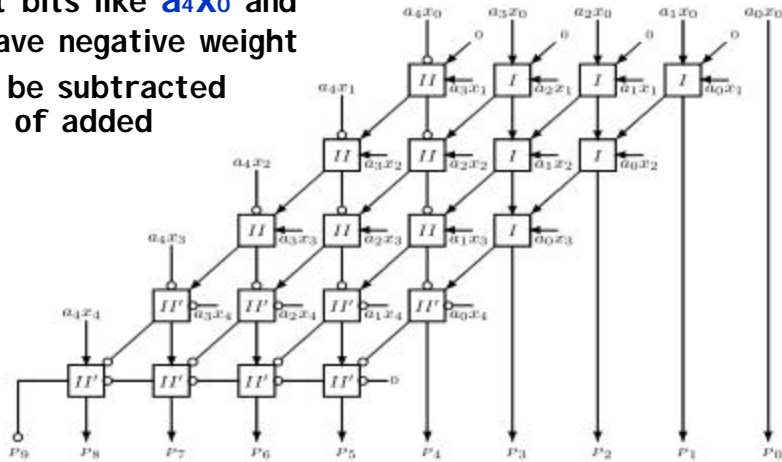
- ◆ **Basic cell - FA** accepting one bit of new partial product $a_i x_j$ + one bit of previously accumulated partial product + carry-in bit



- ◆ 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



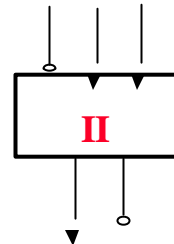
ECE666/Koren Part.6c.5

Copyright 2008 Koren

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

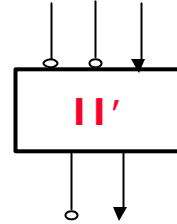
$$s = (x + y - z) \bmod 2 \quad c = \frac{(x + y - z) + s}{2}$$

ECE666/Koren Part.6c.6

Copyright 2008 Koren

Type I' and II' Cells

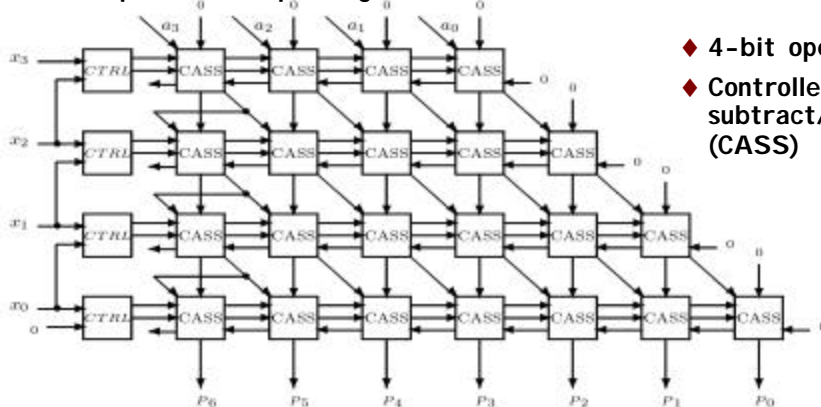
- ◆ Type II' cells: 2 negative inputs and 1 positive
- ◆ Sum of inputs varies from -2 to 1
 - * c output has weight -2
 - * s output has weight +1



- ◆ 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

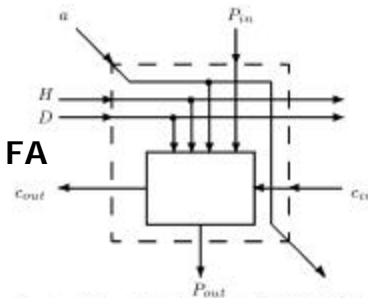
- ◆ For two's complement operands
- ◆ n rows of basic cells - each row capable of adding or subtracting a properly aligned multiplicand to previously accumulated partial product
 - * Cells in row i perform an add, subtract or transfer-only operation, depending on x_i and reference bit



- ◆ 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 **P_{out}**
 - * Type of arithmetic operation indicated by **D** signal
 - * **D=0**: multiplicand bit, **a**, added to **P_{in}** with **c_{in}** as incoming carry - generating **P_{out}** and **c_{out}** as outgoing carry
 - * **D=1**: multiplicand bit, **a**, subtracted from **P_{in}** with incoming borrow and outgoing borrow
- ◆ $P_{out} = P_{in} \hat{\Delta} (a \ H) \hat{\Delta} (c_{in} \ H)$
 $C_{out} = (P_{in} \hat{\Delta} D)(a + C_{in}) + a \ C_{in}$
- ◆ Alternative: combination of multiplexer (0, +a and -a) and FA
- ◆ **H** and **D** generated by CTRL - based on **x_i** and reference bit **x_{i-1}**

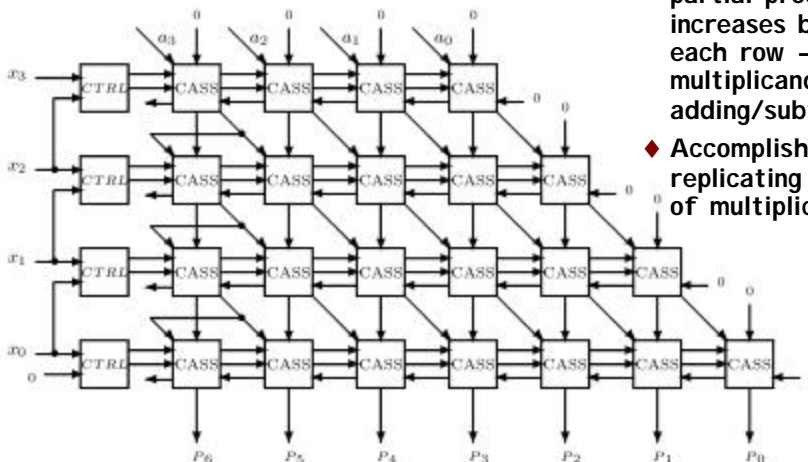


Controlled add/subtract/shift (CASS) cell

ECE666/Koren Part.6c.9

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 **P_{in}=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



light 2008 Koren

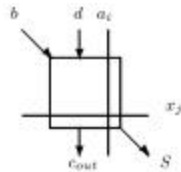
Properties and Delay

- ◆ 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
- ◆ P_0 , generated after one CASS delay (plus delay of CTRL), P_1 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

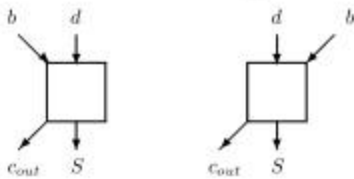
Pipelining

- ◆ 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

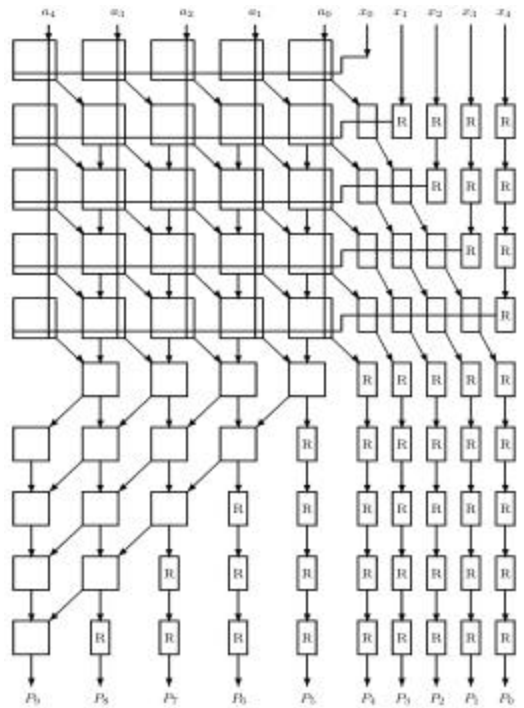


Latched full adder with an AND gate.



Latched half adders.

ECE666/Koren Part.6c.13



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
- ◆ $T_{\text{mult}} \approx \log f \cdot 2n$
- ◆ If residue number system is employed:
- ◆ $T_{\text{mult}} \approx \log f \cdot 2m$
- ◆ m - number of digits needed to represent largest modulus in residue number system

ECE666/Koren Part.6c.14

Copyright 2008 Koren

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: $A T^a$
 - * a 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 $2 n D_{FA}$ (D_{FA} - delay of FA)
- ◆ More precisely, T consists of $(n-1)D_{FA}$ for first $(n-1)$ rows and $(n-1)D_{FA}$ 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 (P 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

ECE666/Koren Part.6c.17

Copyright 2008 Koren

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 $2n \log^2 k$

ECE666/Koren Part.6c.18

Copyright 2008 Koren

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