

**INF5430**

**Digital Arithmetic HW Design**

**High-Speed Multiplication I**

# 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
- ◆  $\dots 01\dots 110\dots = \dots 10\dots 000\dots - \dots 00\dots 010\dots$
- ◆ Using SD (signed-digit) notation  $= \dots 100\dots 0\bar{1}0\dots$
- ◆ **Example:**
- ◆  $\dots 011110\dots = \dots 100000\dots - \dots 000010\dots = \dots 1000\bar{1}0\dots$  (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  $x_{n-1} x_{n-2} \dots x_1 x_0$  in **SD** code
- ◆ **Recoded** multiplier  $y_{n-1} y_{n-2} \dots y_1 y_0$
- ◆  $x_i, x_{i-1}$  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 **001110011(0)** recoded as **01000 $\bar{1}$ 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**

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

- ◆ **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

$A$		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

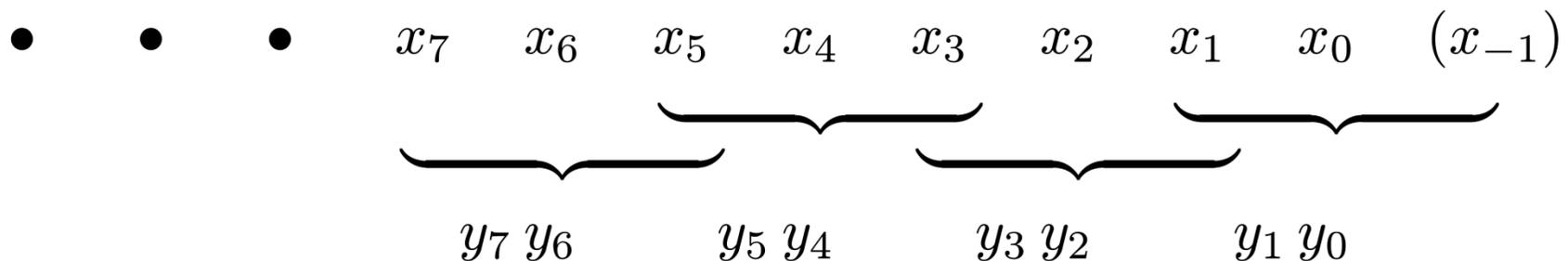
- ◆ Multiplication starts from least significant bit
- ◆ If started from most significant bit - longer adder/subtractor to allow for carry propagation
- ◆ No need to generate recoded **SD** multiplier (requiring **2** bits per digit)
  - \* Bits of original multiplier scanned - control signals for adder/subtractor 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

◆  $i=1, 3, 5, \dots$

◆ 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 ... $\bar{1}1$ ... 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  $y_i y_{i-1}$  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 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  $\pm 3A$  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

- ◆  $y_{n-1}y_{n-2}\dots y_0$  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  $y_{n-1} \cdot y_{n-2} \neq 1$

- ◆ **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

- ◆ 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  $\pm 2A$

$x_{i+1}$	$x_i$	$c_i$	$y_i$	$c_{i+1}$	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

# 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 uniform 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:

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

- ◆ 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:

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

- ◆ **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

$A$			01	00	01		17
$X$	$\times$	(1)	11	01	11		-9
			0	$-2A$	0		Operation
Initial $-A$							
	+		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

### ◆ Previous example -

- ◆ 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

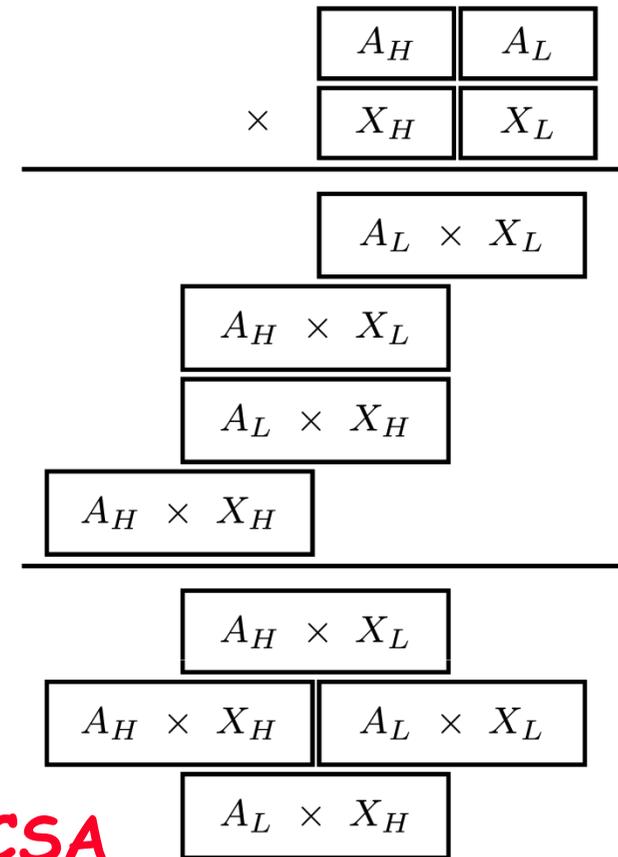
- ◆ Implementing  $n \times n$  bit multiplier as a single integrated circuit - several such circuits for implementing larger multipliers can be used
- ◆  $2n \times 2n$  bit multiplier can be constructed out of 4  $n \times n$  bit multipliers based on :

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

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

