

Lecture 7:
Tree and Array Multipliers

ECE 645—Computer Arithmetic
3/18/08

ECE 645 – Computer Arithmetic

Lecture Roadmap

- Signed Multi-Operand Addition (Two's Complement)
- Multiplication
 - Range of unsigned and signed multiplication
- Tree Multiplication
 - Unsigned Multiplication
 - Signed Multiplication (Two's Complement)
- Array Multiplication
- Squaring

Required Reading

- B. Parhami, *Computer Arithmetic: Algorithms and Hardware Design*
 - Chapter 11, Tree and Array Multipliers
- Note errata at:
 - http://www.ece.ucsb.edu/~parhami/text_comp_arit.htm#errors



Signed Multi-Operand Addition

- Thus far considered multi-operand adder structures for unsigned addition
- Now introduce nuances involved in signed addition

5

Adding Two's Complement Numbers: Avoiding or Detecting Overflow

- To avoid overflow, adding a K.L binary two's complement number to a K.L two's complement number results in a (K+1).L number. To compute, sign extend MSB, ignore c_{K+1}

Example:

$$\begin{array}{r}
 00111.01 + \leftarrow K=4, L=2 \\
 00110.10 = \\
 \hline
 01101.11
 \end{array}$$

Ignore c_{K+1} →

- If result is confined to a K.L number, need overflow detection, which is the c_K xor c_{K-1} (see previous lecture)

Example:

$$\begin{array}{r}
 0111.01 + \\
 0110.10 = \\
 \hline
 01101.11
 \end{array}$$

c_K XOR c_{K-1} indicates overflow →

6

Adding Two's Complement Numbers: Avoiding Overflow in VHDL

- Sign extend input values, then add and ignore final carryout (c_{K+1})

Example:

$$\begin{array}{r}
 00111.01 \quad + \\
 00110.10 \quad = \\
 \hline
 01101.11
 \end{array}$$

Ignore c_{K+1} →

```

use IEEE.STD_LOGIC_SIGNED.all;
entity adder_signed_carryout is
  port(
    a : in STD_LOGIC_VECTOR(1 downto 0);
    b : in STD_LOGIC_VECTOR(1 downto 0);
    sum : out STD_LOGIC_VECTOR(2 downto 0));
end adder_signed_carryout;

architecture arch of adder_signed_carryout is
  signal tempsum : std_logic_vector(2 downto 0);
begin
  tempsum <= (a(1) & a) + (b(1) & b); -- sign extend before addition
  sum <= tempsum;
end arch;

```

7

Adding Multiple Two's Complement Numbers

- When adding two numbers, must sign-extend to final result's width, then add together using adder techniques
- When adding multiple numbers, must sign-extend to final result's width, then add together using adder techniques
 - This can dramatically increase size of carry-save adders

Extended positions	Sign	Magnitude positions
$X_{k-1} X_{k-1} X_{k-1} X_{k-1} X_{k-1}$	X_{k-1}	$X_{k-2} X_{k-3} X_{k-4} \dots$
$Y_{k-1} Y_{k-1} Y_{k-1} Y_{k-1} Y_{k-1}$	Y_{k-1}	$Y_{k-2} Y_{k-3} Y_{k-4} \dots$
$Z_{k-1} Z_{k-1} Z_{k-1} Z_{k-1} Z_{k-1}$	Z_{k-1}	$Z_{k-2} Z_{k-3} Z_{k-4} \dots$

sign extend to final width (requires more adder bits)

8

Two's Complement using "Negative Weight"

$$X = -x_{k-1}2^{k-1} + \sum_{i=-1}^{k-2} x_i \cdot 2^i$$

- Recall a two's complement number can be represented by the above equation, a "negative weight" representation
- Example: $(10011)_2 = (-13)_{10}$, $(01001)_2 = (9)_{10}$

$$\begin{array}{rcccccc} -2^4 & 2^3 & 2^2 & 2^1 & 2^0 & \\ \mathbf{1} & 0 & 0 & 1 & 1 & \\ = -2^4 & + & 2^1 & + & 2^0 & = -16 + 2 + 1 = -13 \end{array}$$

$$\begin{array}{rcccccc} 0 & 1 & 0 & 0 & 1 & \\ = 2^3 & + & 2^0 & = & 8 + 1 = 9 \end{array}$$

9

Two's Complement Sign Extension Using "Negative Weight" Method

- To sign extend a k-bit number by one bit
- First step: Instead of making the MSB have a weight of -1, consider that the digit has a value of -1
 - Does not "exist" hardware, just for notational purposes
- Second step: Apply the equation $x_i = (1 - |x_i|) + 1 - 2 = x_i' + 1 - 2$, where $x_i' = \text{NOT } x_i$
 - In original representation, complement the sign bit, add one to sign bit column (column k-1), add negative one to column k
- Third step: Now remap the new sign bit such that the weight is -1

$$\begin{array}{rcccccc} 2^4 & 2^3 & 2^2 & 2^1 & 2^0 & \\ \mathbf{-1} & 0 & 0 & 1 & 1 & \\ = -2^4 & + & 2^1 & + & 2^0 & = -16 + 2 + 1 = -13 \end{array}$$

$$\begin{array}{rcccccc} 2^5 & 2^4 & 2^3 & 2^2 & 2^1 & 2^0 \\ \mathbf{-1} & 0 & 0 & 0 & 1 & 1 \\ +1 & & & & & \\ = -2^5 & + & 2^4 & + & 2^1 & + & 2^0 & = -16 + 2 + 1 = -13 \end{array}$$

$$\begin{array}{rcccccc} \mathbf{-2^5} & 2^4 & 2^3 & 2^2 & 2^1 & 2^0 \\ \mathbf{1} & \mathbf{1} & 0 & 0 & 1 & 1 \\ = -2^5 & + & 2^4 & + & 2^1 & + & 2^0 & = -16 + 2 + 1 = -13 \end{array}$$

10

Multi-Operand Sign Extension using "Negative Weight" Method

----- Extended positions -----					Sign	Magnitude positions -----				
1	1	1	1	0	x_{k-1}'	x_{k-2}	x_{k-3}	x_{k-4}	...	
					y_{k-1}'	y_{k-2}	y_{k-3}	y_{k-4}	...	
					z_{k-1}'	z_{k-2}	z_{k-3}	z_{k-4}	...	
					1					

- Apply the equation $x_i = (1 - |x_i|) + 1 - 2 = x_i' + 1 - 2$, to all three numbers
 - Complement the three sign bits
 - Add three +1 values to the k-1 column
 - Add three -1 values to the k column
- One -1 value from the k column and two +1 values from the k-1 column eliminate each other
 - This leaves two -1 values in k column, one +1 value in the k-1 column
- Two -1 values from the k column become
 - One -1 value in the k+1 column
 - One 0 value in the k column
- One -1 value in the k+1 column becomes
 - One -1 value in the k+2 column
 - One +1 value in the k+1 column
- Continue moving left until you reach the new MSB
 - When have final -1 value in the new MSB column (in this case, k+4) "re-map" the new sign bit such that it's weight is -1
 - The -1 in the MSB now becomes a 1

11

Multiplication

ECE 645 - Computer Arithmetic

“At least one good reason
 for studying multiplication and division is that
 there is an infinite number of ways
 of performing these operations
 and hence there is an infinite number of PhDs
 (or expenses-paid visits to conferences in the USA)
 to be won from inventing new forms of multiplier.”

Alan Clements
 The Principles of Computer Hardware, 1986

Multiplication Output Range

- Multiplication: $A \times B = C$, $A = K.L$ number, $B = M.N$ number
- Unsigned multiplication
 - $K.L \times M.N = (K+M).(L+N)$ number
 - $1.3 \times 1.4 = 2.7$ number
 - Example:
 - Binary: $1.101 \times 1.1001 = 10.1000101$
 - Decimal: $1.625 \times 1.5625 = 2.5390625$
- Signed multiplication (two's complement)
 - $K.L \times M.N = (K+M).(L+N)$ number
 - $1.3 \times 1.3 = 2.6$ number
 - Example:
 - Binary: $1.000 \times 1.000 = 01.000000$
 - Decimal: $-1 \times -1 = 1$
 - Binary: $0.111 \times 0.111 = 00.110001$
 - Decimal: $0.875 \times 0.875 = 0.765625$
 - Binary: $1.000 \times 0.111 = 11.001000$
 - Decimal: $-1 \times -0.875 = -0.875$
 - NOTE: Only need $K+M$ integer bits when A and B are their **most negative allowable values**
 - If A and B can be restricted such that they do not reach most negative allowable values, then only need $K+M-1$ integer bits, i.e. output results is $(K+M-1).(L+N)$
 - This is a useful trick often using in DSP systems to reduce wordlengths!

Multiplication of signed and unsigned numbers (1)

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
```

Since using both signed and unsigned data types, don't use `std_logic_unsigned/signed`. Do all conversions explicitly.

```
entity multiply is
  port(
    a : in STD_LOGIC_VECTOR(15 downto 0);
    b : in STD_LOGIC_VECTOR(7 downto 0);
    cu : out STD_LOGIC_VECTOR(23 downto 0);
    cs : out STD_LOGIC_VECTOR(23 downto 0)
  );
end multiply;
```

architecture dataflow of multiply is

```
SIGNAL sa: SIGNED(15 downto 0);
SIGNAL sb: SIGNED(7 downto 0);
SIGNAL sres: SIGNED(23 downto 0);

SIGNAL ua: UNSIGNED(15 downto 0);
SIGNAL ub: UNSIGNED(7 downto 0);
SIGNAL ures: UNSIGNED(23 downto 0);
```

VHDL and hardware does not care about the binary point. It is up to the user to keep track of where the binary point is in the input and output.

15

Multiplication of signed and unsigned numbers (2)

```
begin

-- signed multiplication
sa <= SIGNED(a);
sb <= SIGNED(b);
sres <= sa * sb;
cs <= STD_LOGIC_VECTOR(sres);

-- unsigned multiplication
ua <= UNSIGNED(a);
ub <= UNSIGNED(b);
ures <= ua * ub;
cu <= STD_LOGIC_VECTOR(ures);

end dataflow;
```

16

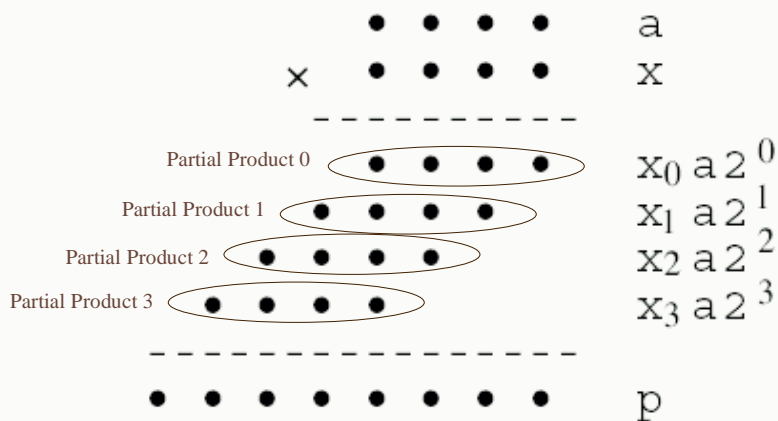
Notation

a	Multiplicand	$a_{k-1} a_{k-2} \dots a_1 a_0$
x	Multiplier	$x_{k-1} x_{k-2} \dots x_1 x_0$
p	Product (a · x)	$p_{2k-1} p_{2k-2} \dots p_2 p_1 p_0$

If multiplicand and multiplier different sizes, usually multiplier is the smaller size

17

Multiplication of Two 4-bit Unsigned Binary Numbers in Dot Notation



Number of partial products = number of bits in multiplier x
 Bit-width of each partial product = bit-width of multiplicand a

18

Basic Multiplication Equations

$$p = a \cdot x \quad x = \sum_{i=0}^{k-1} x_i \cdot 2^i$$

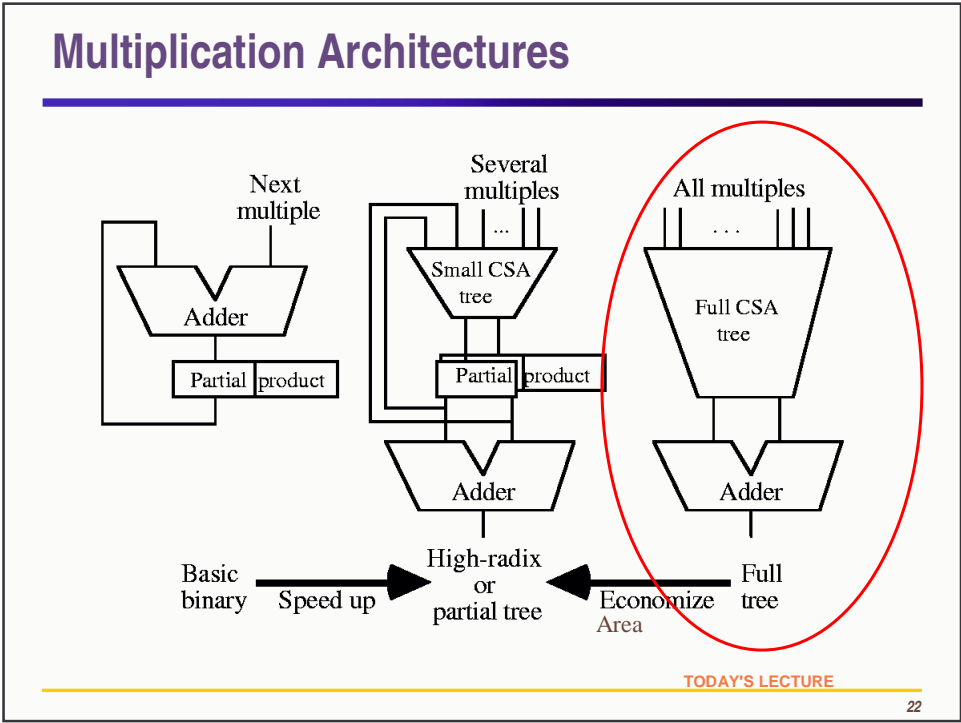
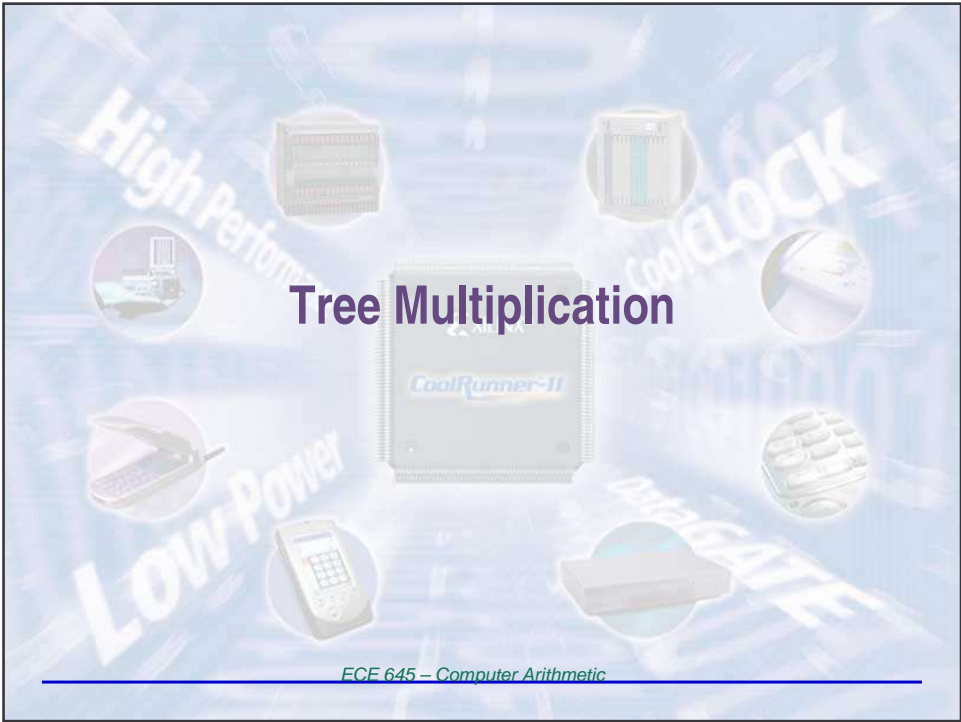
$$\begin{aligned} p = a \cdot x &= \sum_{i=0}^{k-1} a \cdot x_i \cdot 2^i = \\ &= x_0 a 2^0 + x_1 a 2^1 + x_2 a 2^2 + \dots + x_{k-1} a 2^{k-1} \end{aligned}$$

19

Unsigned Multiplication

				a_4	a_3	a_2	a_1	a_0		
			x	x_4	x_3	x_2	x_1	x_0		
$ax_0 \cdot 2^0$				$a_4 x_0$	$a_3 x_0$	$a_2 x_0$	$a_1 x_0$	$a_0 x_0$		
$ax_1 \cdot 2^1$	+			$a_4 x_1$	$a_3 x_1$	$a_2 x_1$	$a_1 x_1$	$a_0 x_1$		
$ax_2 \cdot 2^2$				$a_4 x_2$	$a_3 x_2$	$a_2 x_2$	$a_1 x_2$	$a_0 x_2$		
$ax_3 \cdot 2^3$				$a_4 x_3$	$a_3 x_3$	$a_2 x_3$	$a_1 x_3$	$a_0 x_3$		
$ax_4 \cdot 2^4$				$a_4 x_4$	$a_3 x_4$	$a_2 x_4$	$a_1 x_4$	$a_0 x_4$		
	p_9	p_8	p_7	p_6	p_5	p_4	p_3	p_2	p_1	p_0

20



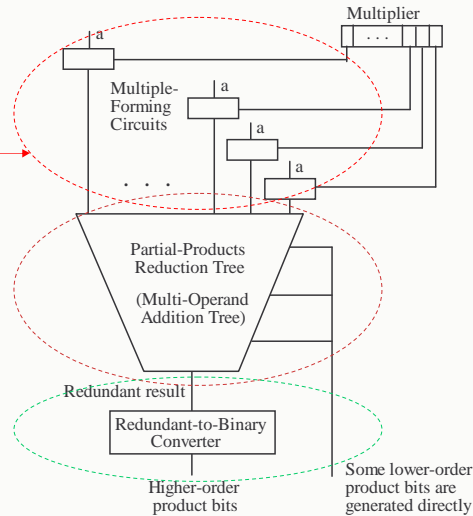
Full Tree Architecture

Designs are distinguished by variations in three elements:

1. Multiple-forming circuits

2. Partial products reduction tree

3. Redundant-to-binary converter



23

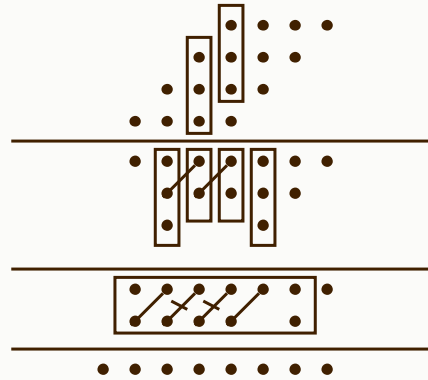
Tree Adder Components

- 1: Multiple Forming Circuits
 - In binary multipliers, these are AND gates (i.e. $a \text{ AND } x_i$)
 - In signed Booth multipliers, these are Booth recoding blocks
 - These circuits create **partial products** ready to be summed
- 2: Partial Products Reduction Tree
 - This is usually a carry-save tree (i.e. Wallace, Dadda)
 - Produces a "redundant" result (i.e. carry and save outputs)
 - Some lower bits produced directly
- 3: Redundant-to-Binary Converter
 - This is usually a fast carry-propagate adder (i.e. carry and save lines \rightarrow final output sum)

24

2. Partial Products Tree Reduction (Dadda)

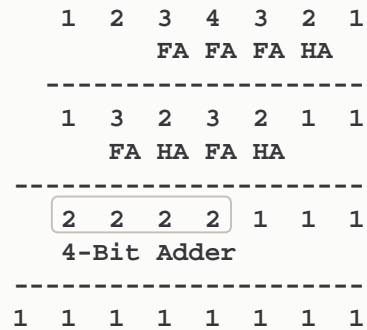
- Dadda tree: combine as late as possible while maintaining same number of carry-save levels (tree height)



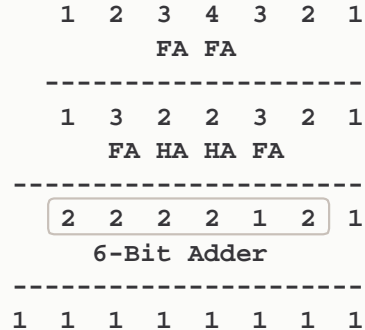
Dadda Tree: 2 carry-save levels, 4 FA, 2 HA, 6-bit CPA

Wallace versus Dadda Tree

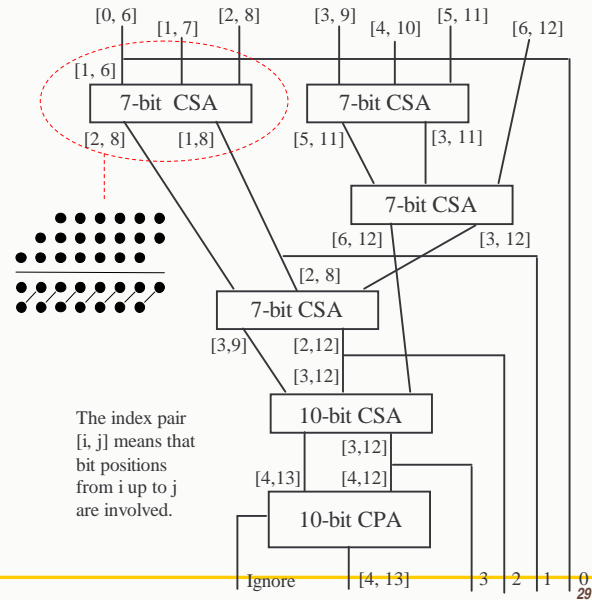
Wallace Tree
(5 FAs + 3 HAs + 4-Bit Adder)



Dadda Tree
(4 FAs + 2 HAs + 6-Bit Adder)



7 x 7 Multiplier



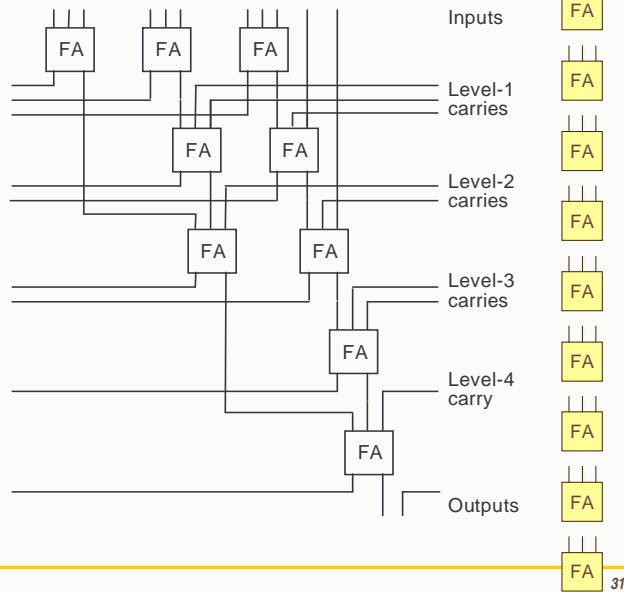
Different Reduction Architectures

- Tree architectures (Wallace, Dadda) are quite irregular and can cause VLSI implementation issues
- Seek other methods for partial product reduction which have better VLSI properties
- Two methods:
 - 11:2 compressor (counter) method occupies narrow vertical slice
 - Create one column, replicate side-by-side
 - A carry created at level i enters level $i+1$
 - Balanced delay tree
 - 11 inputs, 2 outputs
 - 4:2 binary tree

Slice of 11:2 Counter Reduction Tree

In VLSI,
one column

$11 + \psi_1 = 2\psi_1 + 3$
Therefore, $\psi_1 = 8$
carries are needed



Binary Tree of 4-to-2 Reduction Modules

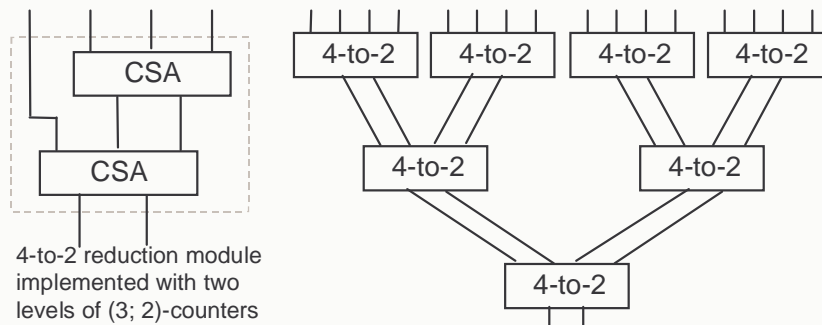


Fig. 11.5 Tree multiplier with a more regular structure based on 4-to-2 reduction modules.

Due to its recursive structure, a binary tree is more regular than a 3-to-2 reduction tree when laid out in VLSI

Example Multiplier with 4-to-2 Reduction Tree

Even if 4-to-2 reduction is implemented using two CSA levels, design regularity potentially makes up for the larger number of logic levels

Similarly, using Booth's recoding may not yield any advantage, because it introduces irregularity

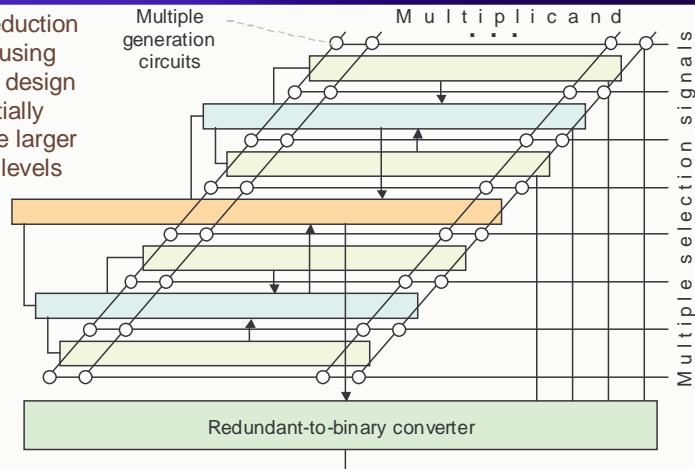


Fig. 11.6 Layout of a partial-products reduction tree composed of 4-to-2 reduction modules. Each solid arrow represents two numbers.

3. Redundant-to-Binary Converter

- Use fast adder such as:
 - Carry-select
 - Carry-lookahead
 - Conditional-sum
 - Etc.

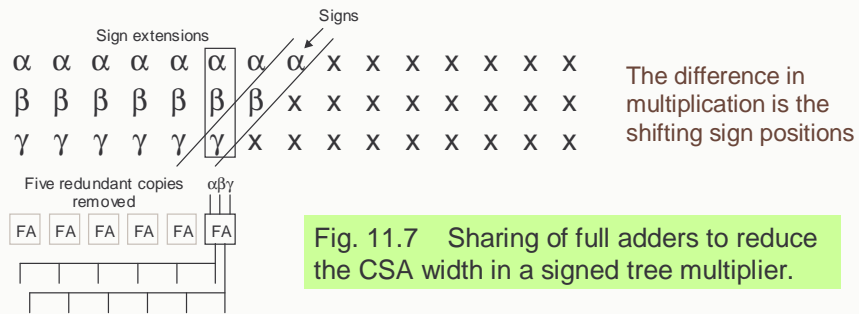


Two's Complement Multiplication

----- Extended positions -----					Sign	Magnitude positions -----				
X_{k-1}	X_{k-1}	X_{k-1}	X_{k-1}	X_{k-1}	X_{k-1}	X_{k-2}	X_{k-3}	X_{k-4}	...	
Y_{k-1}	Y_{k-1}	Y_{k-1}	Y_{k-1}	Y_{k-1}	Y_{k-1}	Y_{k-2}	Y_{k-3}	Y_{k-4}	...	
Z_{k-1}	Z_{k-1}	Z_{k-1}	Z_{k-1}	Z_{k-1}	Z_{k-1}	Z_{k-2}	Z_{k-3}	Z_{k-4}	...	

- Recall, for two's complement multi-operand addition, must first sign extend all partial products to result bit-width, then add
- This can be wasteful in terms of extra full adder cells requires
- Two solutions:
 - Remove redundant full adder cells for sign extension
 - Use "negative weight" representation → Baugh-Wooley multiplier

Remove redundant full adder cells for sign-extension

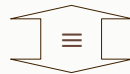


- Remove redundant copies of FA cells which produce the same result
- But also may have fanout issues for this adder

37

Two's Complement Negative Weight Representation

$$\begin{array}{rcccccc}
 & -2^4 & 2^3 & 2^2 & 2^1 & 2^0 \\
 & \mathbf{a}_4 & a_3 & a_2 & a_1 & a_0 \\
 \times & \mathbf{x}_4 & x_3 & x_2 & x_1 & x_0 \\
 \hline
 \end{array}$$



$$\begin{array}{rcccccc}
 & 2^4 & 2^3 & 2^2 & 2^1 & 2^0 \\
 & -\mathbf{a}_4 & a_3 & a_2 & a_1 & a_0 \\
 \times & -\mathbf{x}_4 & x_3 & x_2 & x_1 & x_0 \\
 \hline
 \end{array}$$

38

Implementing Partial Products

$$\bar{z} = 1 - z$$

$$z = 1 - \bar{z}$$

$$-a_j x_i = -a_j (1 - \bar{x}_i) = a_j \bar{x}_i - a_j = a_j \bar{x}_i + a_j - 2 a_j$$

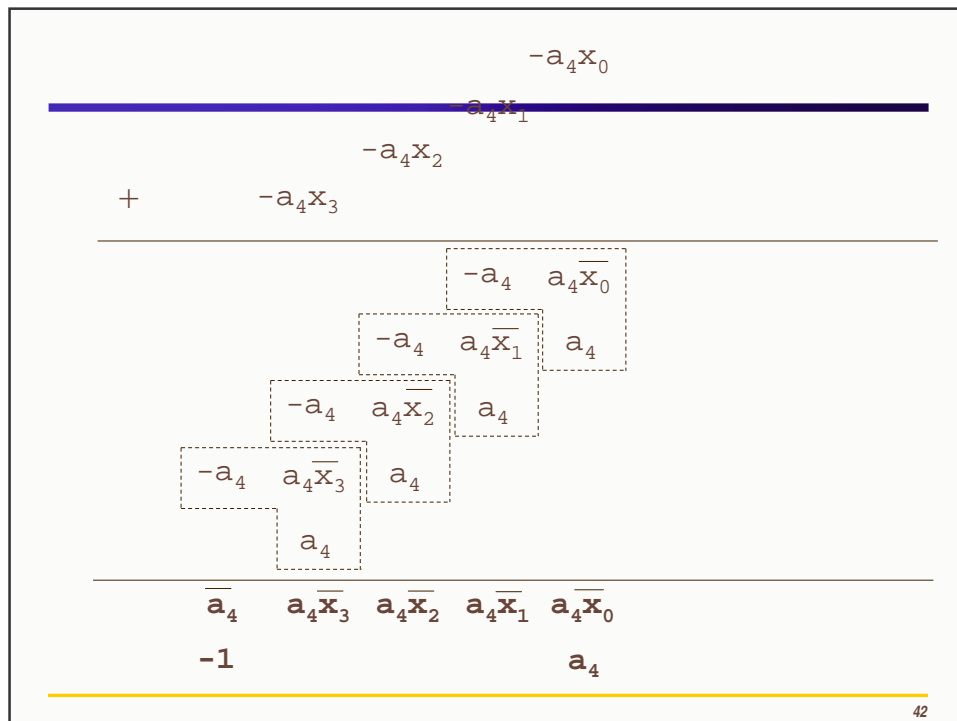
$$-a_j x_i = -(1 - \bar{a}_j) x_i = \bar{a}_j x_i - x_i = \bar{a}_j x_i + x_i - 2 x_i$$

$$-a_j x_i = -(1 - \bar{a}_j \bar{x}_i) = \bar{a}_j \bar{x}_i - 1 = \bar{a}_j \bar{x}_i + 1 - 2$$

$$-a_j = -(1 - \bar{a}_j) = \bar{a}_j - 1 = \bar{a}_j + 1 - 2$$

$$-x_i = -(1 - \bar{x}_i) = \bar{x}_i - 1 = \bar{x}_i + 1 - 2$$

41



42

Baugh-Wooley Two's Complement Multiplier

$$\begin{array}{r}
 \begin{array}{cccccc}
 & & & & -a_4 & a_3 & a_2 & a_1 & a_0 \\
 & & & & x & -x_4 & x_3 & x_2 & x_1 & x_0 \\
 \hline
 & & & & a_4\bar{x}_0 & a_3x_0 & a_2x_0 & a_1x_0 & a_0x_0 \\
 + & & & & a_4\bar{x}_1 & a_3x_1 & a_2x_1 & a_1x_1 & a_0x_1 \\
 & & & & a_4\bar{x}_2 & a_3x_2 & a_2x_2 & a_1x_2 & a_0x_2 \\
 & & & & a_4\bar{x}_3 & a_3x_3 & a_2x_3 & a_1x_3 & a_0x_3 \\
 & & & & a_4x_4 & \bar{a}_3x_4 & \bar{a}_2x_4 & \bar{a}_1x_4 & \bar{a}_0x_4 \\
 & & & & \bar{a}_4 & & & & a_4 \\
 & & & & 1 & \bar{x}_4 & & & x_4 \\
 \hline
 p_9 & p_8 & p_7 & p_6 & p_5 & p_4 & p_3 & p_2 & p_1 & p_0 \\
 \hline
 -2^9 & 2^8 & 2^7 & 2^6 & 2^5 & 2^4 & 2^3 & 2^2 & 2^1 & 2^0
 \end{array}
 \end{array}$$

45

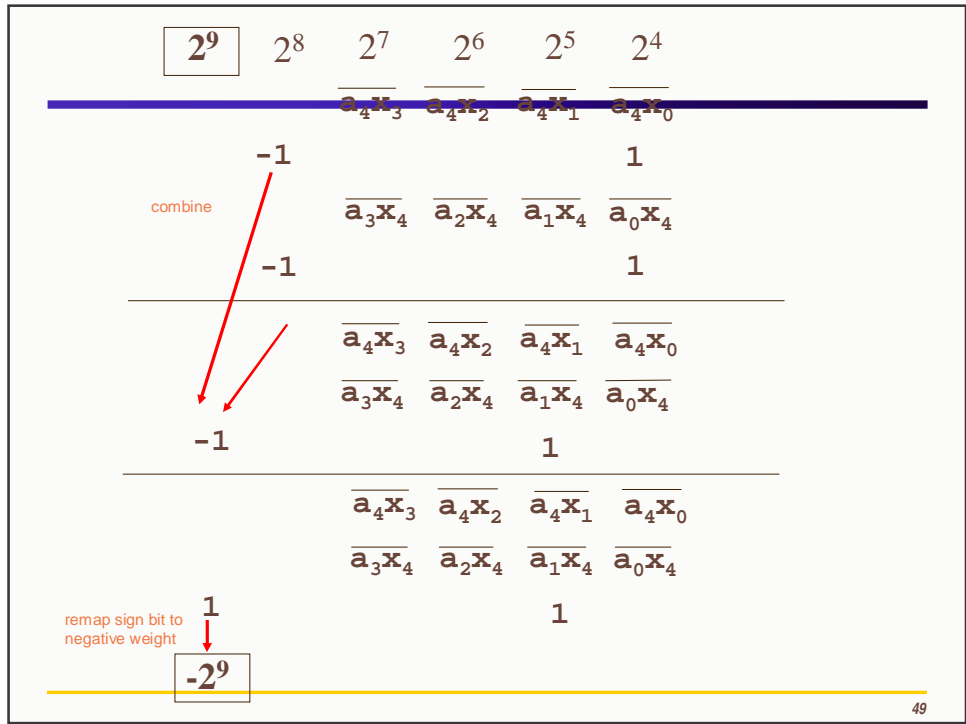
Modified Baugh-Wooley Two's Complement Multiplier

- Apply the equation $x_i = (1 - |x_i|) + 1 - 2 = x_i' + 1 - 2$, where $x_i' = \text{NOT } x_i$

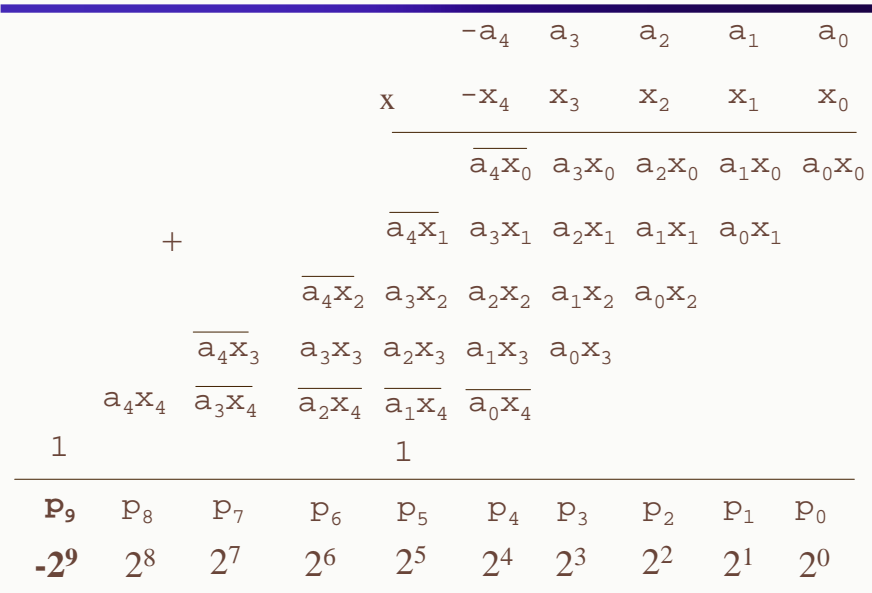
46

$$\begin{array}{r}
 -a_4x_0 \\
 \hline
 a_4x_1 \\
 -a_4x_2 \\
 + \quad -a_4x_3 \\
 \hline
 \begin{array}{r}
 -1 \quad \overline{a_4x_0} \\
 -1 \quad \overline{a_4x_1} \quad 1 \\
 -1 \quad \overline{a_4x_2} \quad 1 \\
 -1 \quad \overline{a_4x_3} \quad 1 \\
 1
 \end{array} \\
 \hline
 \overline{a_4x_3} \quad \overline{a_4x_2} \quad \overline{a_4x_1} \quad \overline{a_4x_0} \\
 -1 \qquad \qquad \qquad 1
 \end{array}$$

$$\begin{array}{r}
 -a_3x_4 \quad -a_2x_4 \quad -a_1x_4 \quad -a_0x_4 \\
 \hline
 \begin{array}{r}
 -1 \quad \overline{a_0x_4} \\
 -1 \quad \overline{a_1x_4} \quad 1 \\
 -1 \quad \overline{a_2x_4} \quad 1 \\
 -1 \quad \overline{a_3x_4} \quad 1 \\
 1
 \end{array} \\
 \hline
 \overline{a_3x_4} \quad \overline{a_2x_4} \quad \overline{a_1x_4} \quad \overline{a_0x_4} \\
 -1 \qquad \qquad \qquad 1
 \end{array}$$



Modified Baugh-Wooley Multiplier

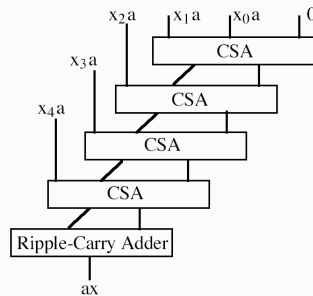




Array Multipliers

- Note that CSA-based partial product reduction trees are irregular and sometimes introduce difficulty in VLSI implementation (wiring, placement, etc.)
- One alternative is array multiplier
 - "Slower" in terms of number of logic levels
 - **But very nice in terms of VLSI implementation (wiring, placement, etc.)**

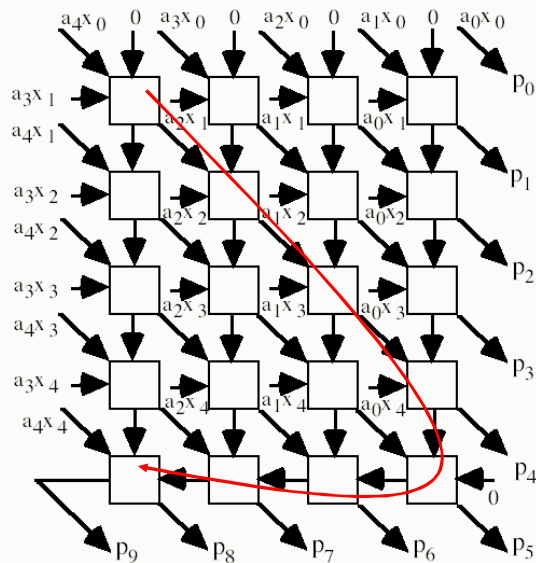
Basic 5 x 5 Unsigned Array Multiplier



- Critical path is longer
 - For M-bit multiplier and M-bit multiplicand, critical path passes through approximately $(M - 1) + (M - 1)$ full adder cells
- Also good for computing $ax + y$
 - Replace zero on top CSA with y
 - Helpful for inner product calculations and multiply-accumulate functions

53

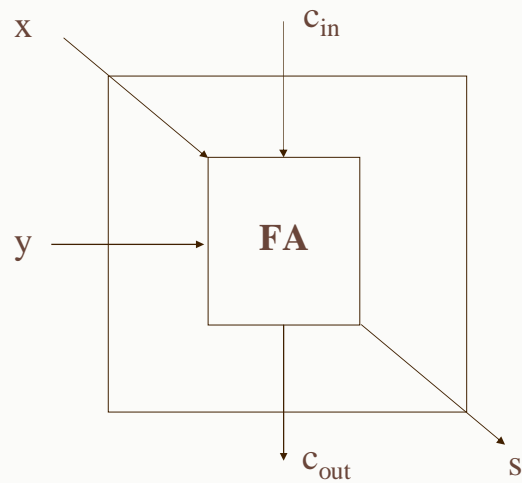
5 x 5 Array Multiplier



Critical path
(assuming sum
and carry delays
the same)

54

Array Multiplier Basic Cell



55

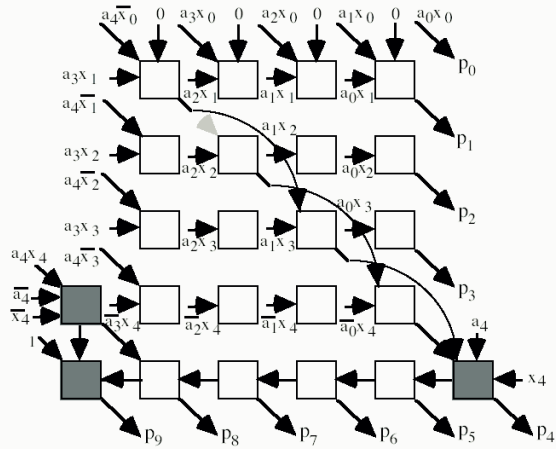
Baugh-Wooley Two's Complement Multiplier

$$\begin{array}{r}
 \begin{array}{cccccc}
 & & & & -a_4 & a_3 & a_2 & a_1 & a_0 \\
 x & & & & -x_4 & x_3 & x_2 & x_1 & x_0 \\
 \hline
 & & & & a_4\bar{x}_0 & a_3x_0 & a_2x_0 & a_1x_0 & a_0x_0 \\
 + & & & & a_4\bar{x}_1 & a_3x_1 & a_2x_1 & a_1x_1 & a_0x_1 \\
 & & & & a_4\bar{x}_2 & a_3x_2 & a_2x_2 & a_1x_2 & a_0x_2 \\
 & & & & a_4\bar{x}_3 & a_3x_3 & a_2x_3 & a_1x_3 & a_0x_3 \\
 & & & a_4x_4 & \bar{a}_3x_4 & \bar{a}_2x_4 & \bar{a}_1x_4 & \bar{a}_0x_4 & \\
 & & & \bar{a}_4 & & & & & a_4 \\
 1 & & & \bar{x}_4 & & & & & x_4 \\
 \hline
 p_9 & p_8 & p_7 & p_6 & p_5 & p_4 & p_3 & p_2 & p_1 & p_0 \\
 \hline
 -2^9 & 2^8 & 2^7 & 2^6 & 2^5 & 2^4 & 2^3 & 2^2 & 2^1 & 2^0
 \end{array}
 \end{array}$$

56

Modifications in 5 x 5 Array Multiplier

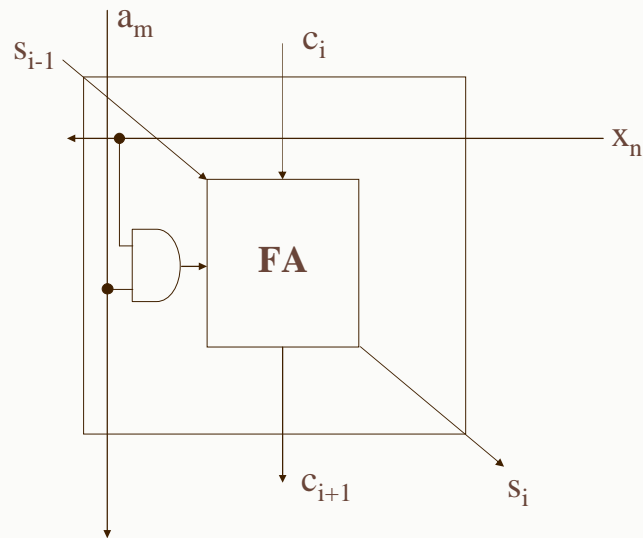
- Shaded cells indicate additional cells for Baugh-Wooley
- When sum logic longer delay than carry logic, critical path goes through diagonal then along last row
 - Can reduce this by causing some sum signals to skip rows
 - These are shown as curved arcs in figure



57

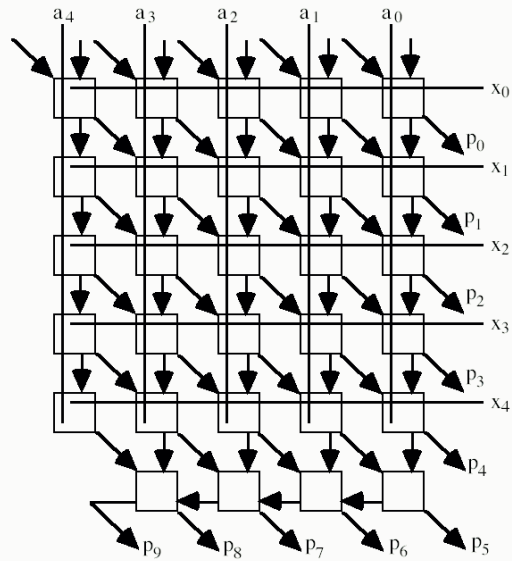
Array Multiplier – Modified Basic Cell

AND gate included in basic cell



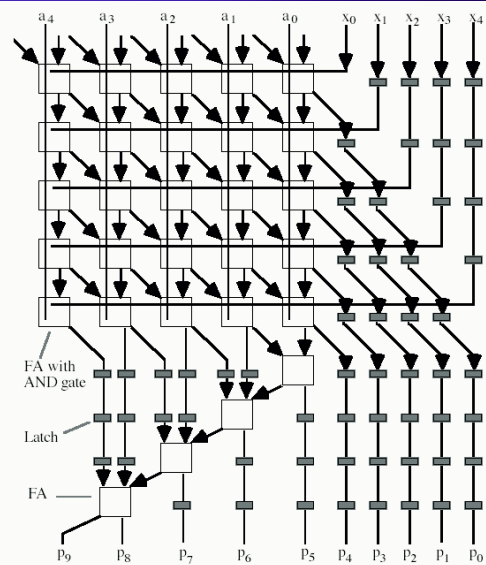
58

5 x 5 Array Multiplier with Modified Cells



59

Pipelined 5 x 5 Multiplier



60



Optimizations for Squaring (1)

Multiply x by x

					x_4	x_3	x_2	x_1	x_0	
					\times	x_4	x_3	x_2	x_1	x_0
						x_4x_0	x_3x_0	x_2x_0	x_1x_0	x_0x_0
				x_4x_1	x_3x_1	x_2x_1	x_1x_1	x_0x_1	x_0x_1	Reduce
		x_4x_2	x_3x_2	x_2x_2	x_1x_2	x_0x_2				to next
	x_4x_3	x_3x_3	x_2x_3	x_1x_3	x_0x_3					column
	x_4x_4	x_3x_4	x_2x_4	x_1x_4	x_0x_4					
p_9	p_8	p_7	p_6	p_5	p_4	p_3	p_2	p_1	p_0	

Reduce the bit matrix

					x_4	x_3	x_2	x_1	x_0	
					\times	x_4	x_3	x_2	x_1	x_0
						x_4x_3	x_4x_2	x_4x_1	x_4x_0	x_3x_0
						x_4	x_3x_2	x_3x_1	x_3x_0	x_2x_0
							x_3	x_2x_1	x_2x_0	x_1x_0
								x_2	x_1	x_0
p_9	p_8	p_7	p_6	p_5	p_4	p_3	p_2	0	x_0	

Optimizations for Squaring (2)

$$\begin{array}{r}
 x_i x_j \\
 x_j x_i \\
 \hline
 x_i x_j
 \end{array}
 \qquad
 x_i x_j + x_i x_j = 2 x_i x_j$$

$$\begin{array}{r}
 x_i x_j \\
 x_i \\
 \hline
 x_i x_j \quad x_i \bar{x}_j
 \end{array}
 \qquad
 \begin{aligned}
 x_i x_j + x_i &= 2 x_i x_j - x_i x_j + x_i = \\
 &= 2 x_i x_j + x_i (1-x_j) = \\
 &= 2 x_i x_j + x_i \bar{x}_j
 \end{aligned}$$

63

Squaring Using Lookup Tables

for relatively small values k

input=a	output=a ²
0	0
1	1
2	4
3	9
4	16
	...
i	i ²
	...
2 ^k -1	(2 ^k -1) ²

2^k words 2k-bit each

64

Multiplication Using Squaring

$$a \cdot x = \frac{(a+x)^2 - (a-x)^2}{4}$$