No	Topic
1.	Number systems
2.	Fixed point (FXP) addition: theory
3.	FXP adders: basic implementations, speed-up techniques
4.	SD number system
5.	FXP multiplication: basics
6.	FXP multiplication: speed-up techniques
7.	**FXP multiplication: implementations**
8.	FXP division: basics, speed-up techniques
9.	FXP division: speed-up techniques, implementations
10.	Floating point numbers: basics
11.	Floating point numbers: rounding
12.	Arithmetic Optimization Techniques

MULTIPLIER IMPLEMENTATIONS

- Multiplication involves two operations

  1. Generation of partial products
  2. Accumulation of partial products

- Two ways to speed up multiplication

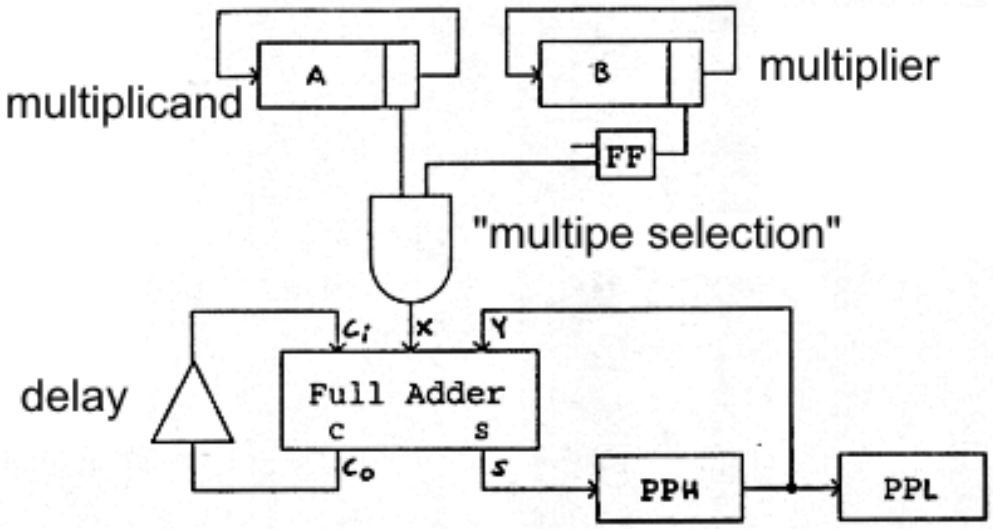
  1. Reduce the number of partial products (previous lecture)
  2. Accelerate accumulation of partial products (this lecture)

- Classification of high-speed multipliers

  1. **Sequential multipliers**: generate the partial products sequentially and add each newly generated product to the previously accumulated partial product
  2. **Parallel multipliers**: generate all partial products in parallel and then use a fast multi-operand adder for their accumulation
  3. **Cellular array multipliers**: made up of an array of identical cells that generate new partial products and accumulate them simultaneously, no separate circuits for partial product generation and their accumulation

1. SEQUENTIAL MULTIPLIERS

1.1. Sequential multiplier for serial 1-bit scanning



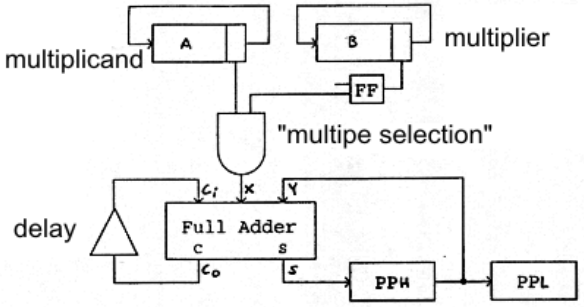
- Partial product and multiple of *A* are added up bit-serially
  → each accumulation takes *n* cycles
  → multiplier bit is held unchanged during this time

PPH = MSBs of partial product
PPL = LSBs of partial product

- Serial multiplication requires n^2 cycles for result
- Suitable for all number systems (Sam, 1's, 2's, unsigned)
- Simple implementation, low-cost

EXAMPLE **Multiply $A = 101_2 = 5_{10}$ and $B = 011 = 3_{10}$ using serial multiplier**

cycle	*A*	*B*	c_i	*X*	*y*	c_o	*s*	PPH	PPL
1.	101	011	0	1	0	0	1	000	---
								100	
2.	110	011	0	0	0	0	0		
								010	
3.	011	011	0	1	0	0	1		
								101	
Shift partial product								010	1--
4.	101	101	0	1	0	0	1		
								101	
5.	110	101	0	0	1	0	1		
								110	
6.	011	101	0	1	0	0	1		
								111	
Shift second partial product								011	11-
7.	101	110	0	0	1	0	1		
								101	
8.	110	110	0	0	1	0	1		
								110	
9.	011	110	0	0	0	0	0		
								011	
Shift last partial product								001	111

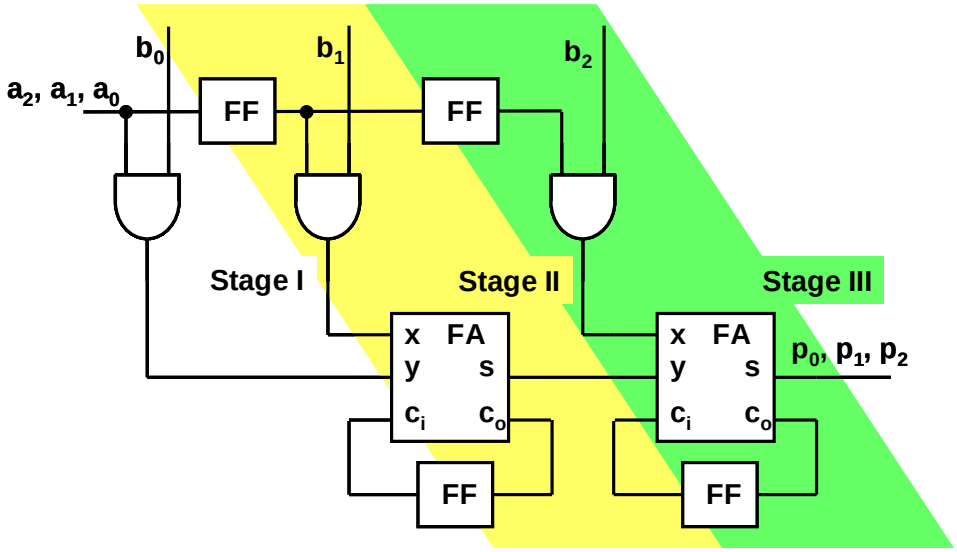


Result: $001111_2 = 15_{10}$

1.2. Serial-parallel multiplier (1-bit scanning)

$$
\begin{array}{ccccc}
 & & a_2 & a_1 & a_0 \\
 & & b_2 & b_1 & b_0 \\
\hline
 & & a_2b_0 & a_1b_0 & a_0b_0 \\
 & a_2b_1 & a_1b_1 & a_0b_1 & \\
a_2b_2 & a_1b_2 & a_0b_2 & & \\
\hline
a_2b_2 & a_2b_1+a_1b_2 & a_2b_0+a_1b_1+a_0b_2 & a_1b_0+a_0b_1 & a_0b_0
\end{array}
$$

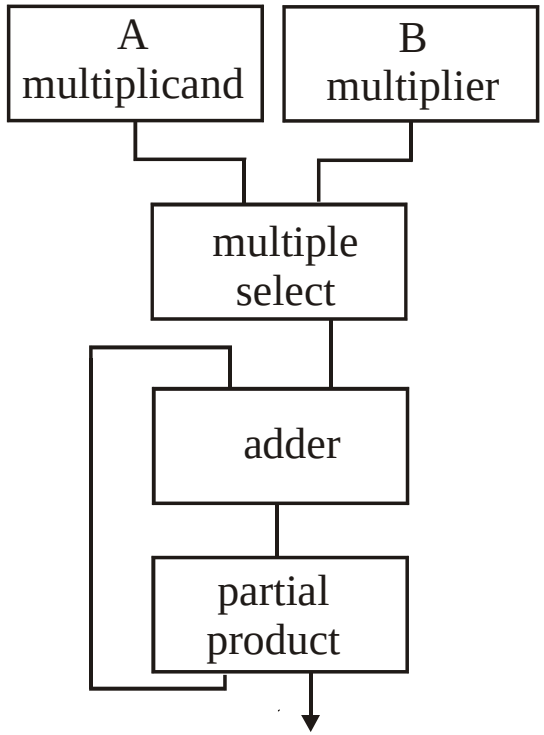
Cycle	Stage I	Stage II	Stage III
1	a_0b_0	a_0b_0	a_0b_0
2	a_1b_0	$a_1b_0+a_0b_1$	$a_1b_0+a_0b_1$
3	a_2b_0	$a_2b_0+a_1b_1$	$a_2b_0+a_1b_1+a_0b_2$
4	0	a_2b_1	$a_2b_1+a_1b_2$
5	0	0	a_2b_2



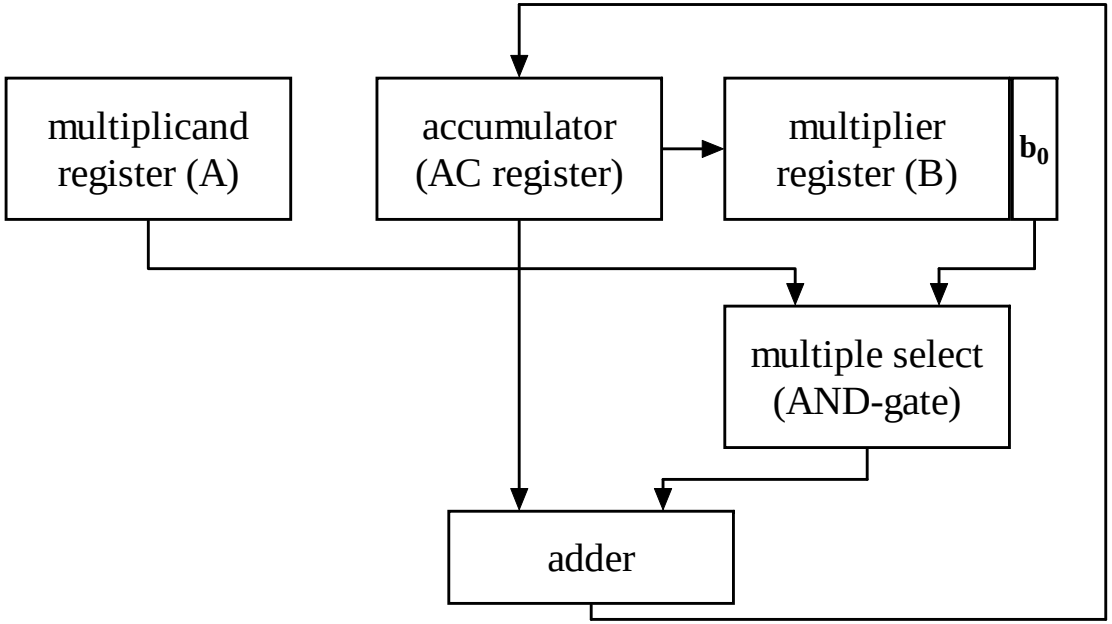
- Note: a_ib_j is called a summand

1.3 Sequential multiplier for parallel 1 –bit scanning

- Generic multiplier block diagram:



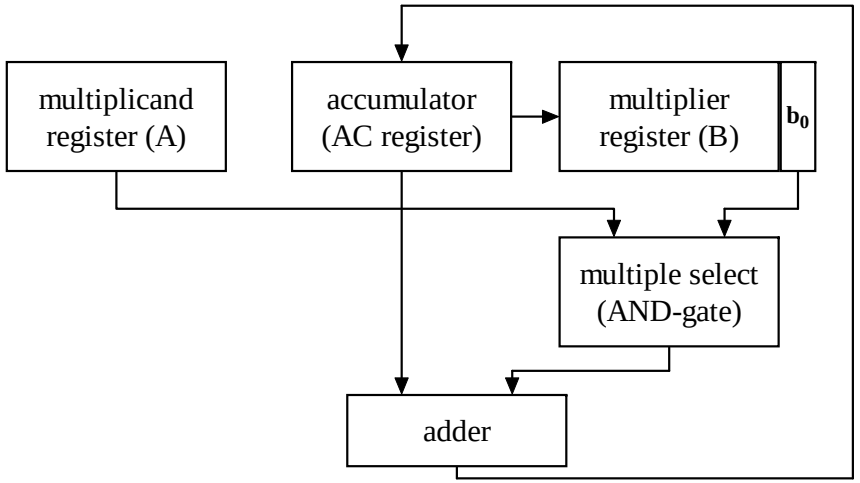
- Basic implementation for right-shift method
  - Summands are generated parallel in each cycle
  - One cycle: add multiple of *A* into partial product
  - *n* cycles needed to complete operation
  - *n*+1 –bit adder required for intermediate overflows



EXAMPLE

	Reg AC	Reg B	A = 00101
$p^{(0)}$	00000	01011	
add *A*	00101		
$2\,p^{(1)}$	00101	01011	
shift	00010	10101 1	omit 1
add *A*	00101		
$2\,p^{(2)}$	00111	10101	
shift	00011	11010 1	omit 1
add zero	00000		
$2\,p^{(3)}$	00011	11010	
shift	00001	11101 0	omit 0
add *A*	00101		
$2\,p^{(4)}$	00110	11101	
shift	00011	01110 1	omit 1
add zero	00000		
$2\,p^{(5)}$	00011	01110	
shift	00001	10111 0	omit 0
Final result	00001	10111	

- Note how the multiplier bits are shifted out and replaced by the LSB part of the partial product (and final result)
- The example applies to unsigned numbers multiplication: note the zeros shifted in to the AC-register
- More complicated with signed numbers!



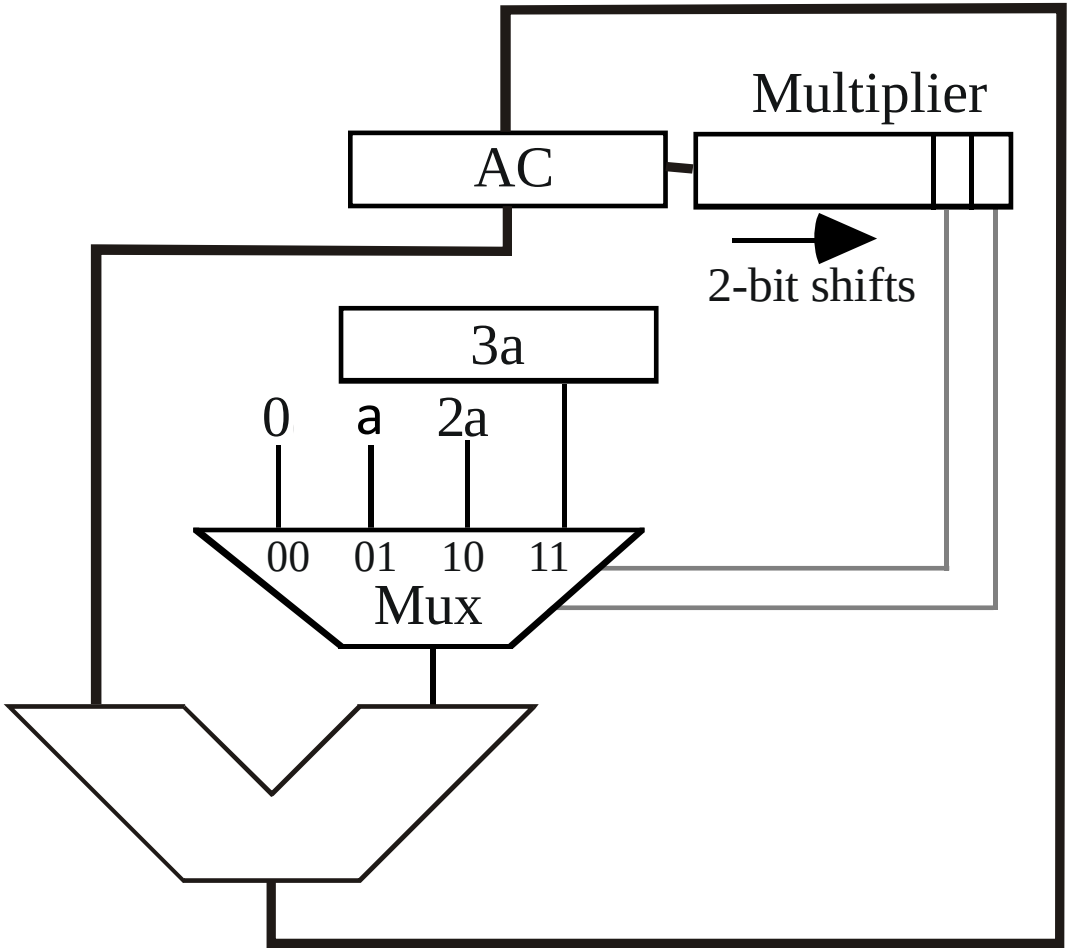
1.4 Sequential multipliers for *m* –bit scanning and *m* –bit recoding

- Scanning and recoding differ in multiple count, type and shift length
- Both require
  1. Multiple generation
  2. Selection of proper multiple
  3. Accumulation of partial product (= addition of multiple)
- Implementations differ in how above are arranged and / or combined

EXAMPLE

2-bit non-overlapping scanning (radix-4 multiplication)

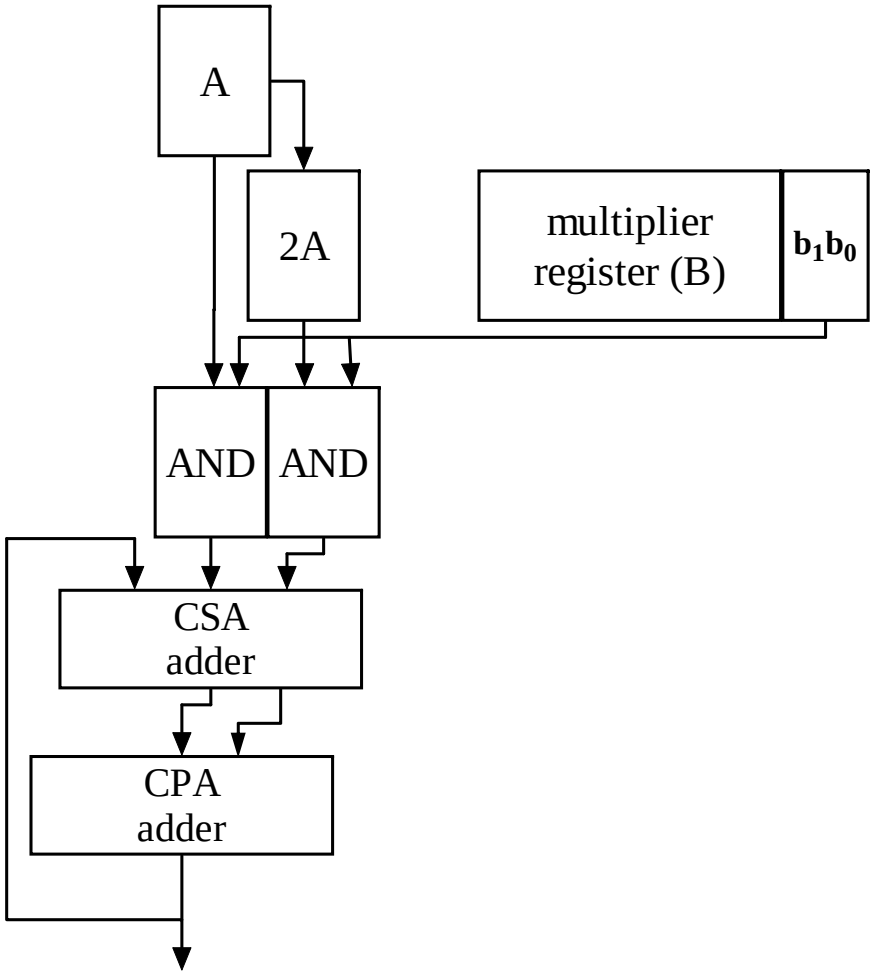
- Multiples (0, *A*, 2*A*, and 3*A*) are generated before operation
- Depending on multiplier bits, one of possible multiplies is added into partial product per cycle



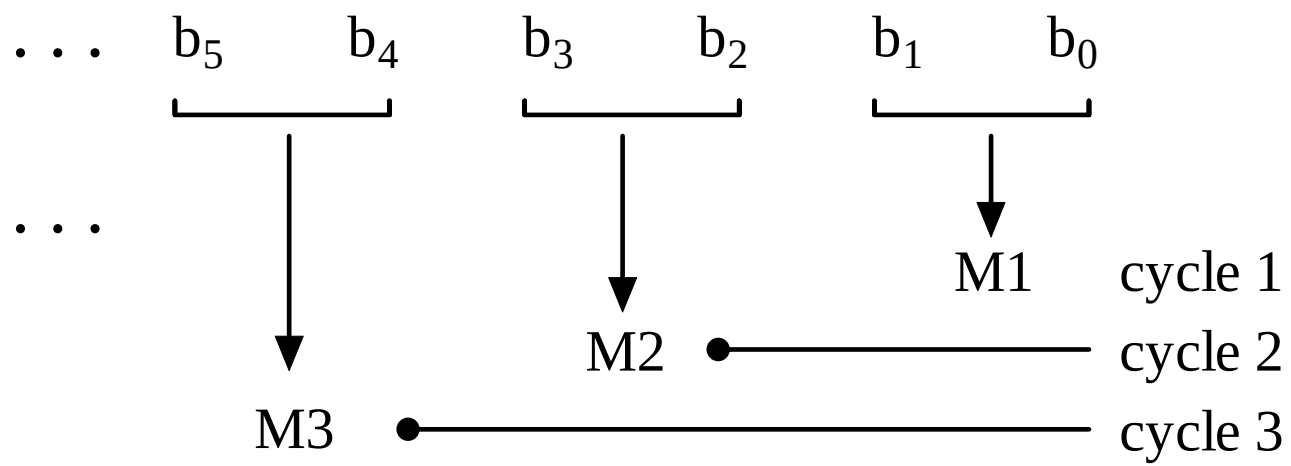
EXAMPLE

2-bit non-overlapping scanning (radix-4 multiplication)

- Multiples (0, *A*, 2*A*, 3*A*)
  - Generated on the fly
  - Selected by multiplier bits
- Overflow / sign extension handled as described earlier for 1's, 2's, SaM numbers



- One multiple is added in each cycle

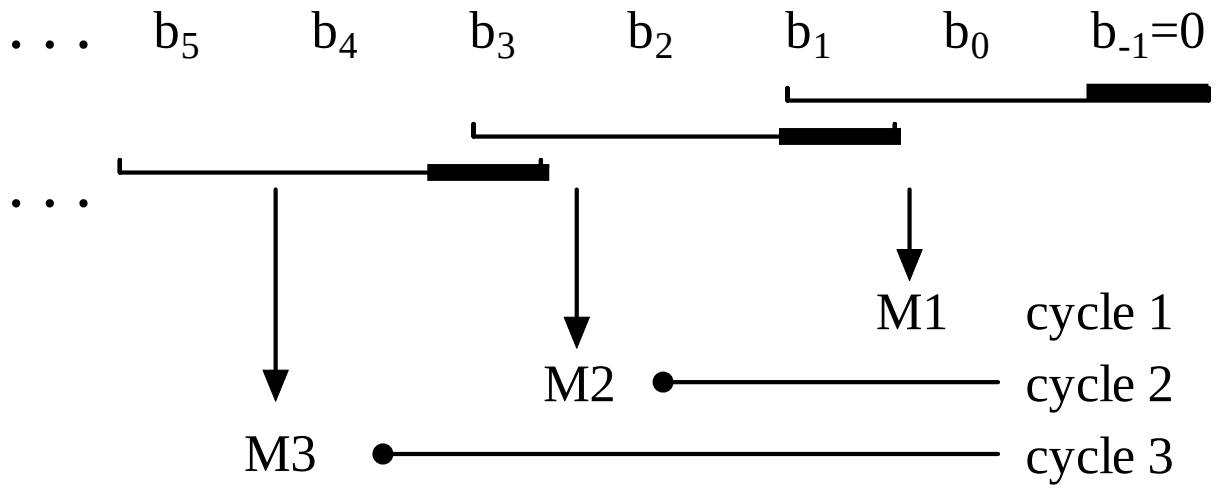


- Note: $M1$ = 0, A, $2A$ or $3A$ depending on $b_1 b_0$
  $M2$ = 0, A, $2A$ or $3A$ depending on $b_3 b_2$

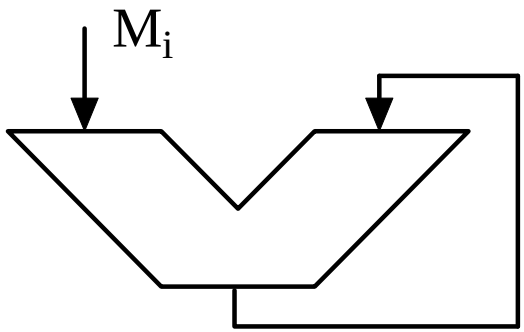
- More than one multiple can be added during one cycle, because multiples can be obtained independently

EXAMPLE

2-bit look-behind recoding



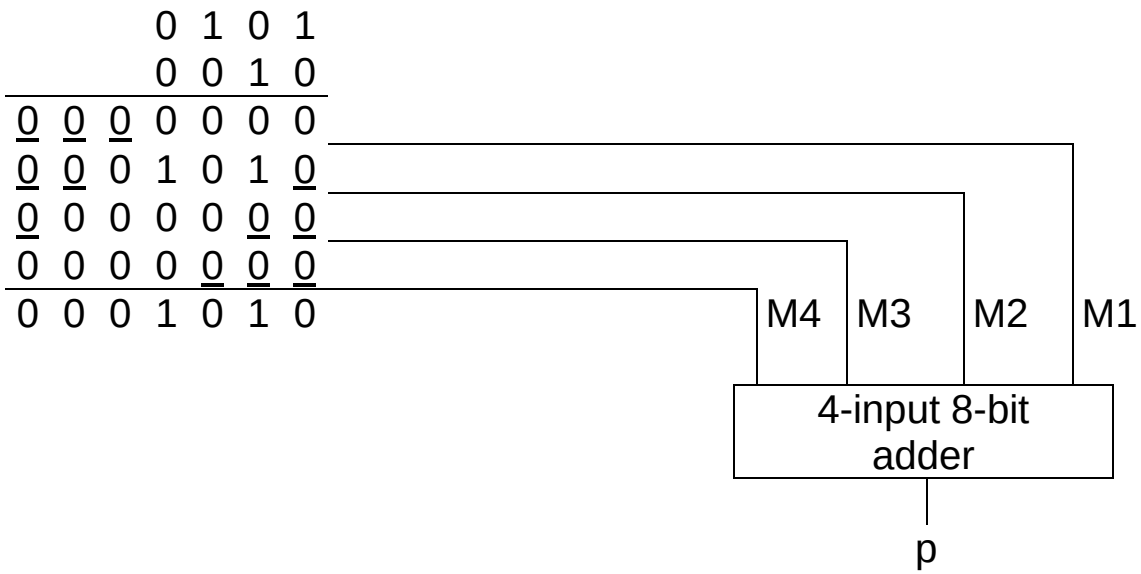
- One multiple is added in each cycle



- More than one multiple can be added during one cycle, because multiples can be obtained independently

2. PARALLEL MULTIPLIERS

2.1. Basic method



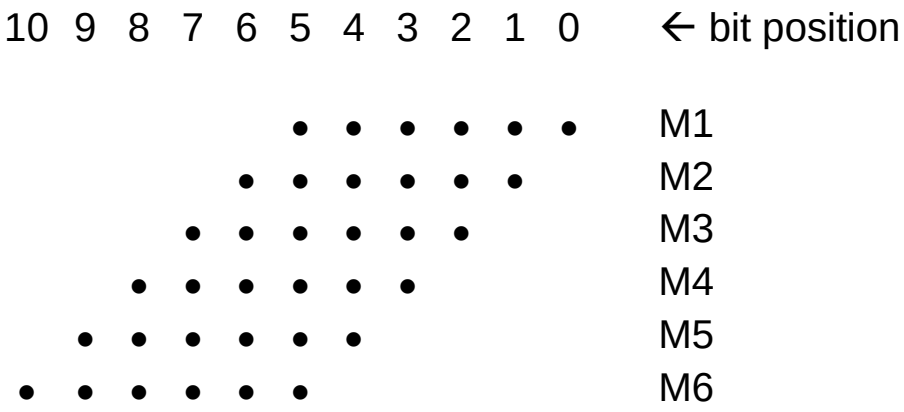
- Note sign extension to multiples
- Adder will be large and wasteful because of sign extended bits
- For parallel addition in simpler hardware, summands may be added column-wise using (*m*, *n*)-counters (CSA adders)

2.2. Parallel column-wise addition of summands

- Consider example of multiplication of A and B

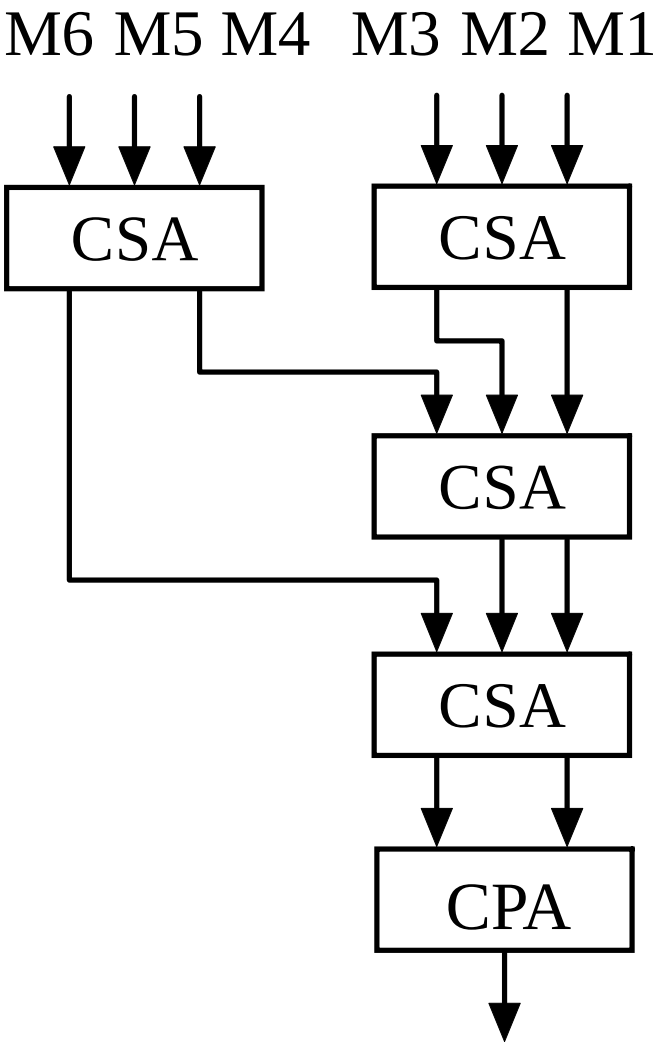
```
   a5  a4  a3  a2  a1  a0      A
x  b5  b4  b3  b2  b1  b0      B
```

- Using dot-notation, the summand matrix is following (consists of 36 summands):



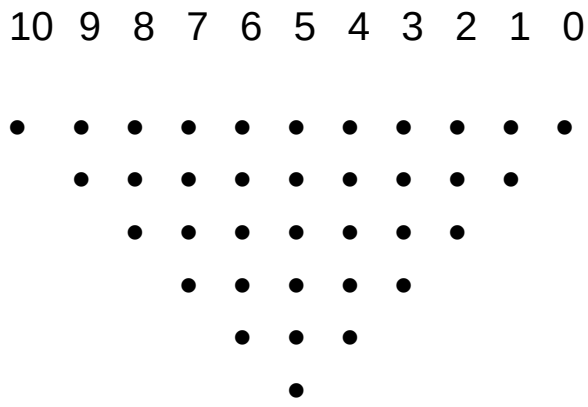
- In the dot notation, the value of the summand is not shown, but the order (weight) is highlighted

- M1, M2, …M6 can be added using e.g. six-input Wallace tree adder:



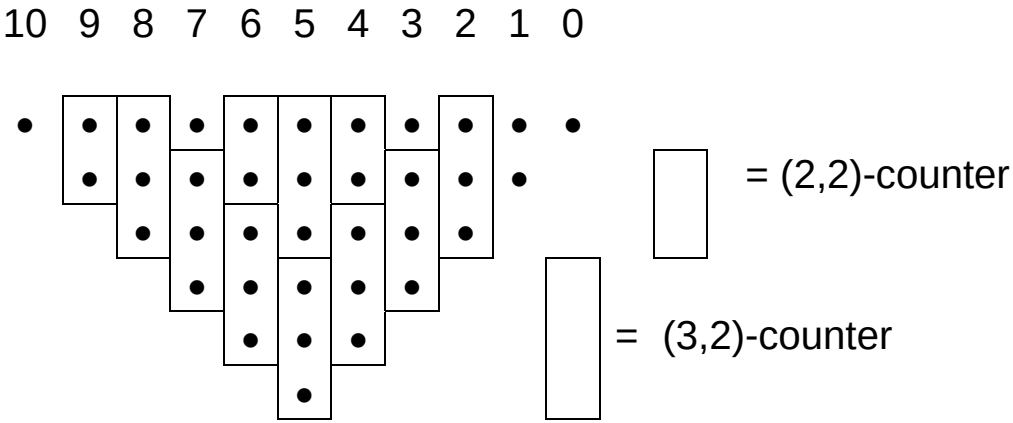
- Wallace tree
  - The bits are combined at the earliest opportunity
  - Fastest possible implementation
- Dadda tree
  - The bits are combined at the latest opportunity
  - The CSA levels are minimized
  - The tree is much simpler than Wallace
  - Speed is slower than with Wallace

- (3,2)-counters can be saved, if the original summand matrix is reorganized as follows:

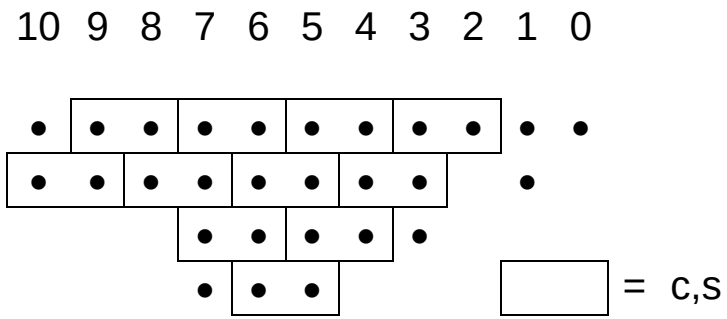


- Columns are added up in 3 carry-save stages followed by CPA merging → 4 levels of adders

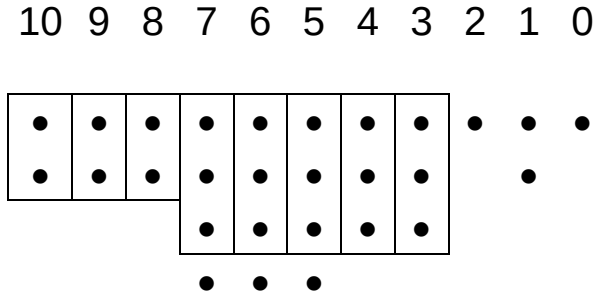
- Level 1 CSA addition:



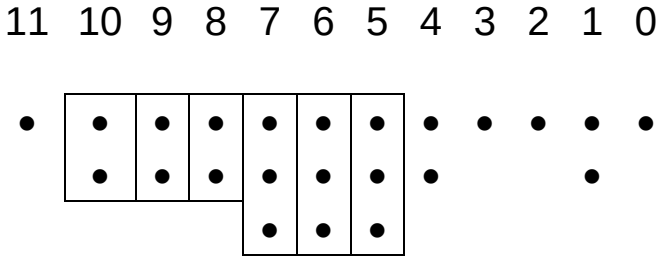
- Results of level 1 CSA addition:



- Level 2 CSA addition:



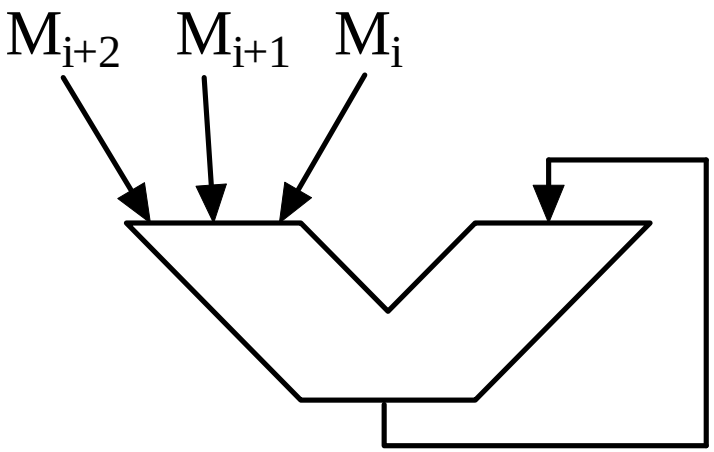
- Level 3 CSA addition:



- After that, CPA (Carry Propagate Adder) is used to merge carry and sum bits (the remaining two rows)

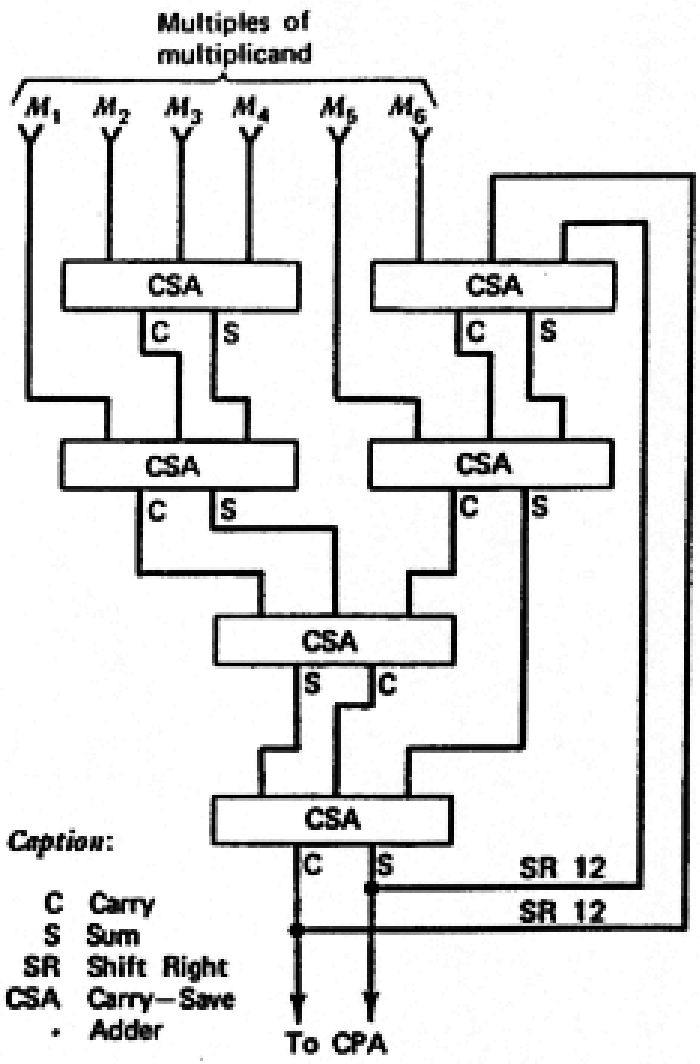
2.3. Combining techniques of sequential and parallel multipliers

- 3 multiples addition / cycle



- First cycle: M3, M2, M1 added to partial product
- Second cycle: M6, M5, M4 added to partial product, etc

- EXAMPLE: parallel addition of 6 multiples into partial product



- 6 multiples are generated e.g. out of

  a) 12 multiplier bits in 2-bit non-overlapped scan
  -> each multiple may be 0,A,2A,3A

  b) 13 multiplier bits in 2-bit look-ahead recoding
  -> each multiple may be 0,±2A, ±4A

  c) 13 multiplier bits in 2-bit look-behind recoding
  -> each multiple may be 0, ±A, ±2A

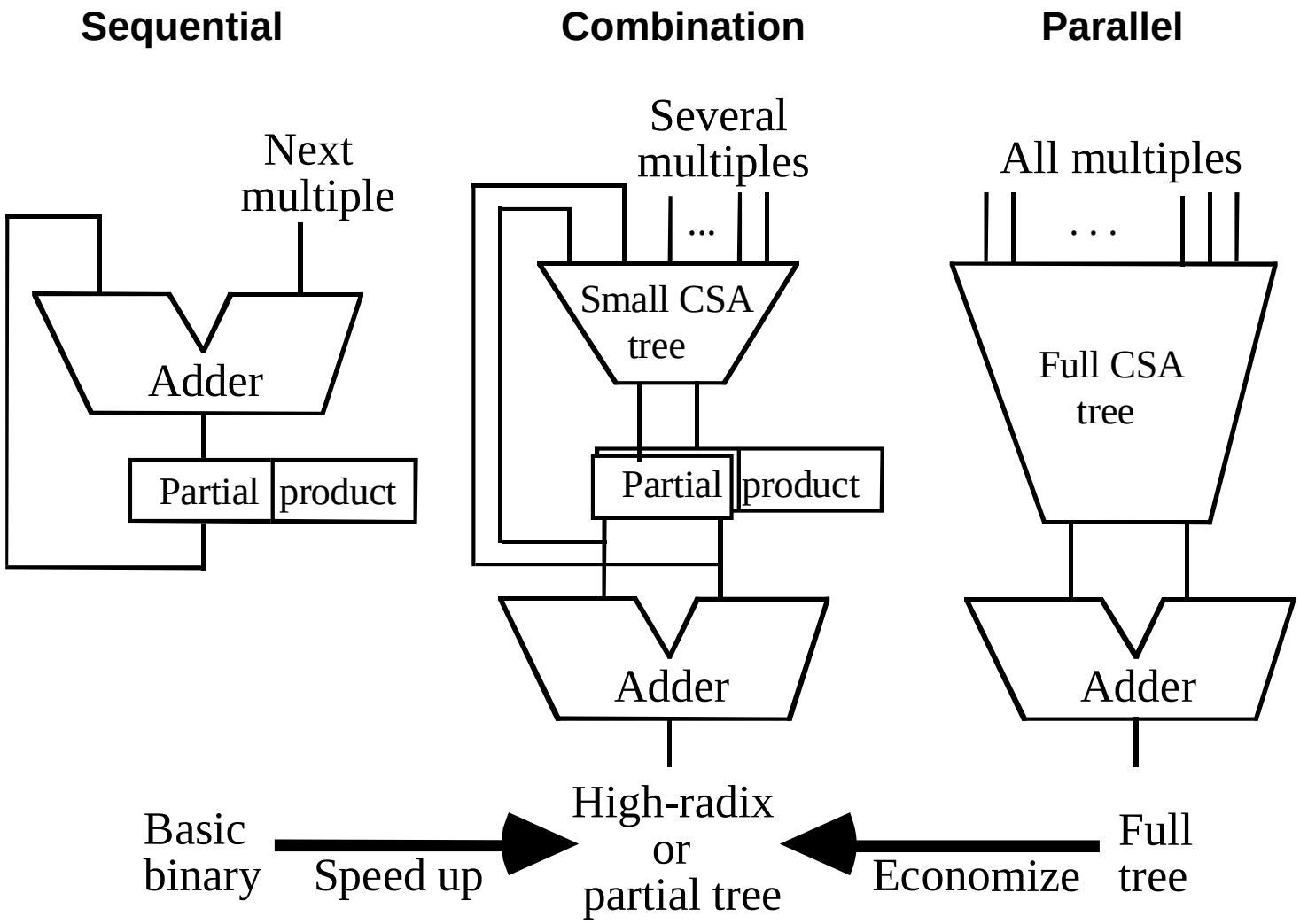
  d) 19 multiplier bits in 3-bit look-behind recoding
  -> each multiple may be 0, ±A, ±2A, ±3A, ±4A

  e) 6 multiplier bits in 1-bit scanning
  -> each multiple may be 0,A

- CSA width must be changed if more bits are scanned/recoded

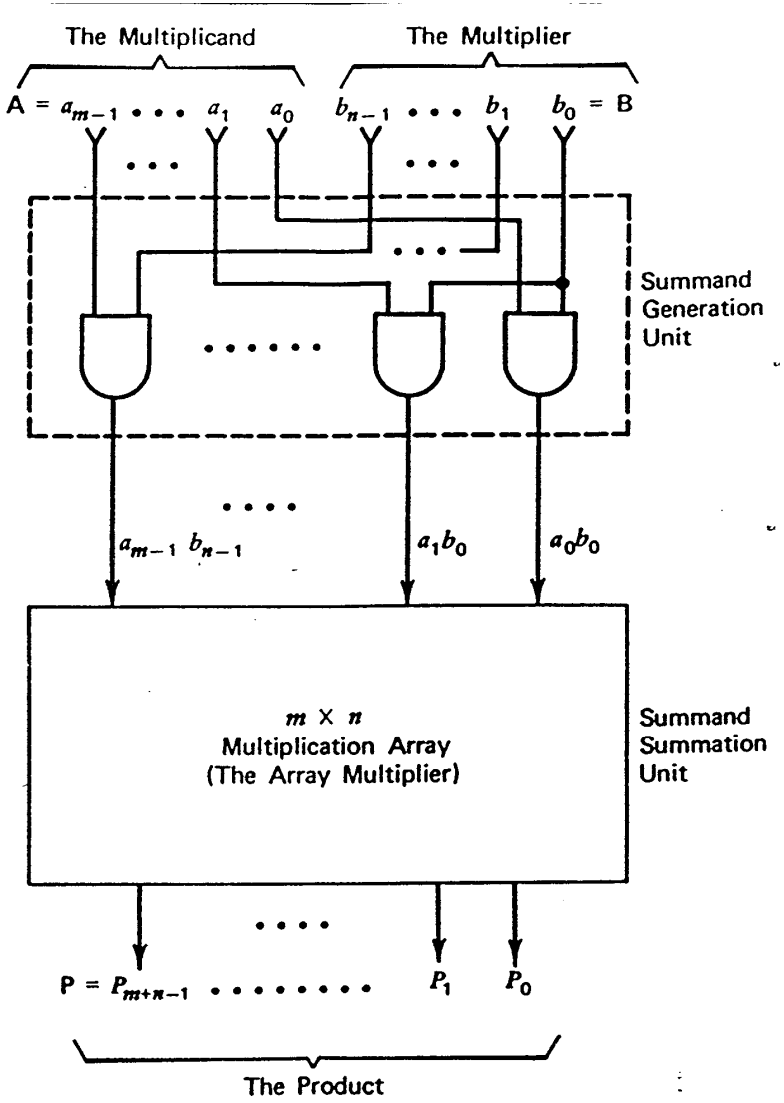
- Amount of right shift depends on amount of multiplier bits scanned in parallel

- Trade-off between complexity, speed and area



3. CELLULAR ARRAY MULTIPLIERS

- Main functional blocks
  - Summand ($a_i b_j$) generation unit
  - Summand summation unit
- In actual implementations, the blocks are merged
- Array multipliers are suitable for pipelining
  - ⇨ Typically best performance among multipliers
- Reasonable implementations
  - Unsigned
  - Indirect 1's complement and sign-magnitude
  - Direct 2's complement multiplication
- Array multiplier performs column-wise addition of summands



- Carry-out from each (whole) column can be generated partially and merged into the next column

EXAMPLE

row	col	3	2	1	0
1				1	0
2			0	1	1
3		0	1	1	0
4		0	0	1	
s_i		0	1	0	1
$c_{i+1..i}$		0	0	10	0

col	4	3	3	2	2	1	1	0
1						1		0
2				0		1		1
				0	1	0	0	1
				↓↙		↓↙		↓
				1		0		1
3		0		1		1		0
		0	1	0	0	1	0	1
		↓↙		↓↙		↓↙		
		1		0		1		
4		0		0		1		
	0	1	0	0	1	0		
	↓	↓	↓	↓	↓	↓	↓	↓
	0	1	0	0	1	0	0	1
	c	s	c	s	c	s	c	s

3.1 Braun multiplier

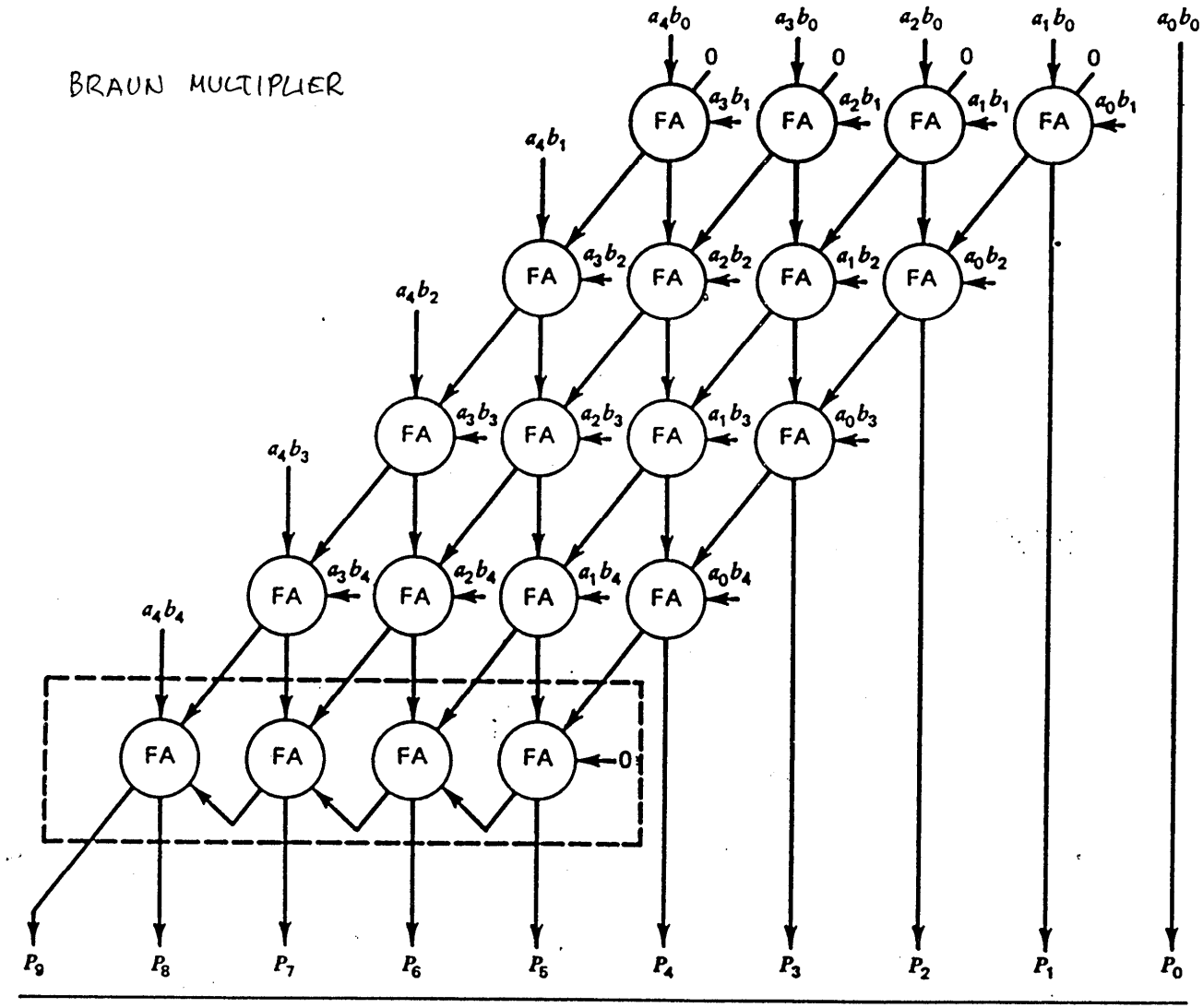
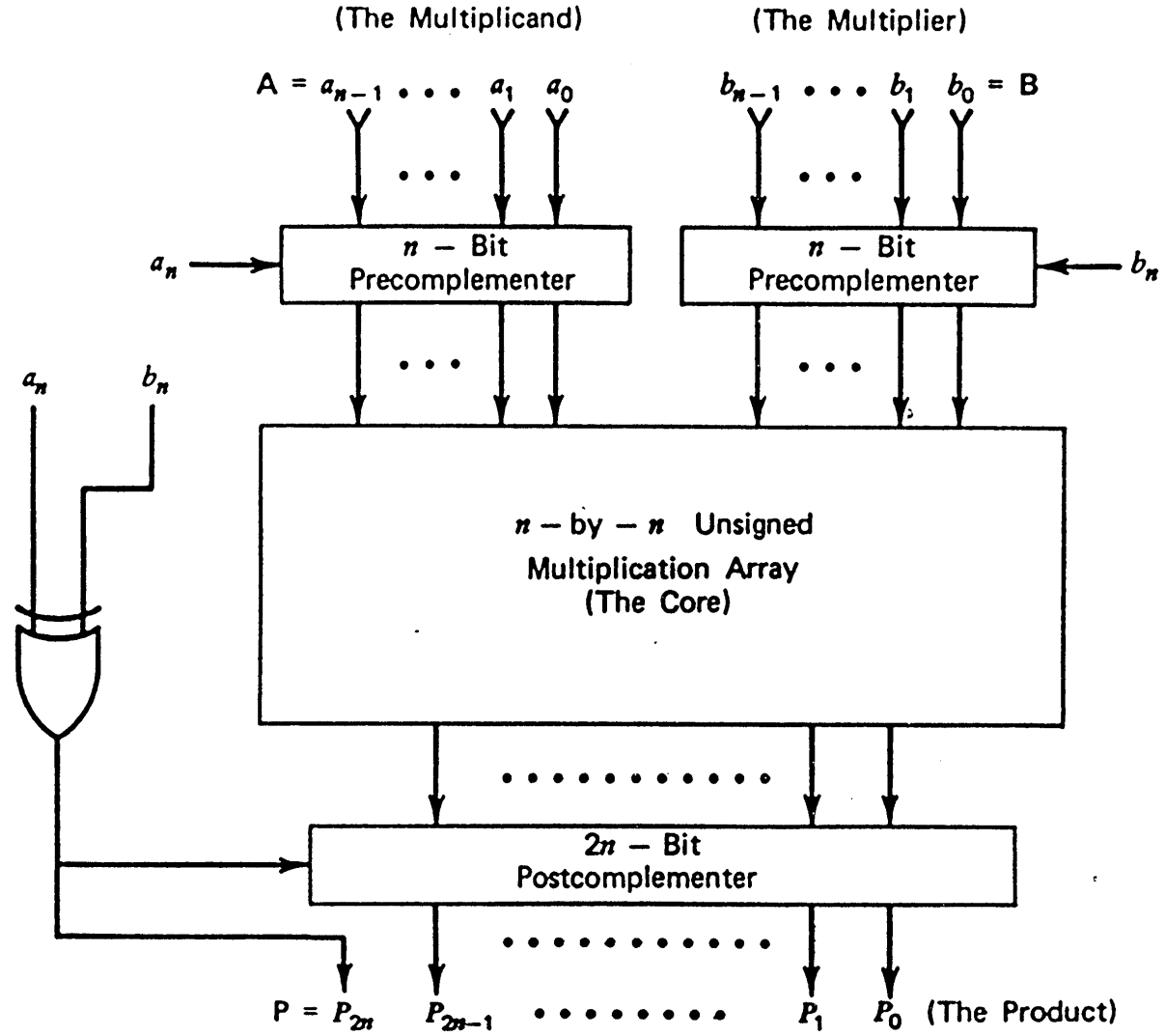


Figure 6.3 The schematic circuit diagram of a 5-by-5 unsigned array multiplier (Braun [5]).

- Above merging is directly applied in Braun multiplier
- Suitable for unsigned and SaM numbers

3.2 Indirect cellular array signed number multiplication

- Operation
  - Signed numbers are converted to unsigned
  - Multiplication is performed
  - Result is converted if needed
- Not reasonable for 2's complement numbers



3.3 Direct 2's complement array multiplication

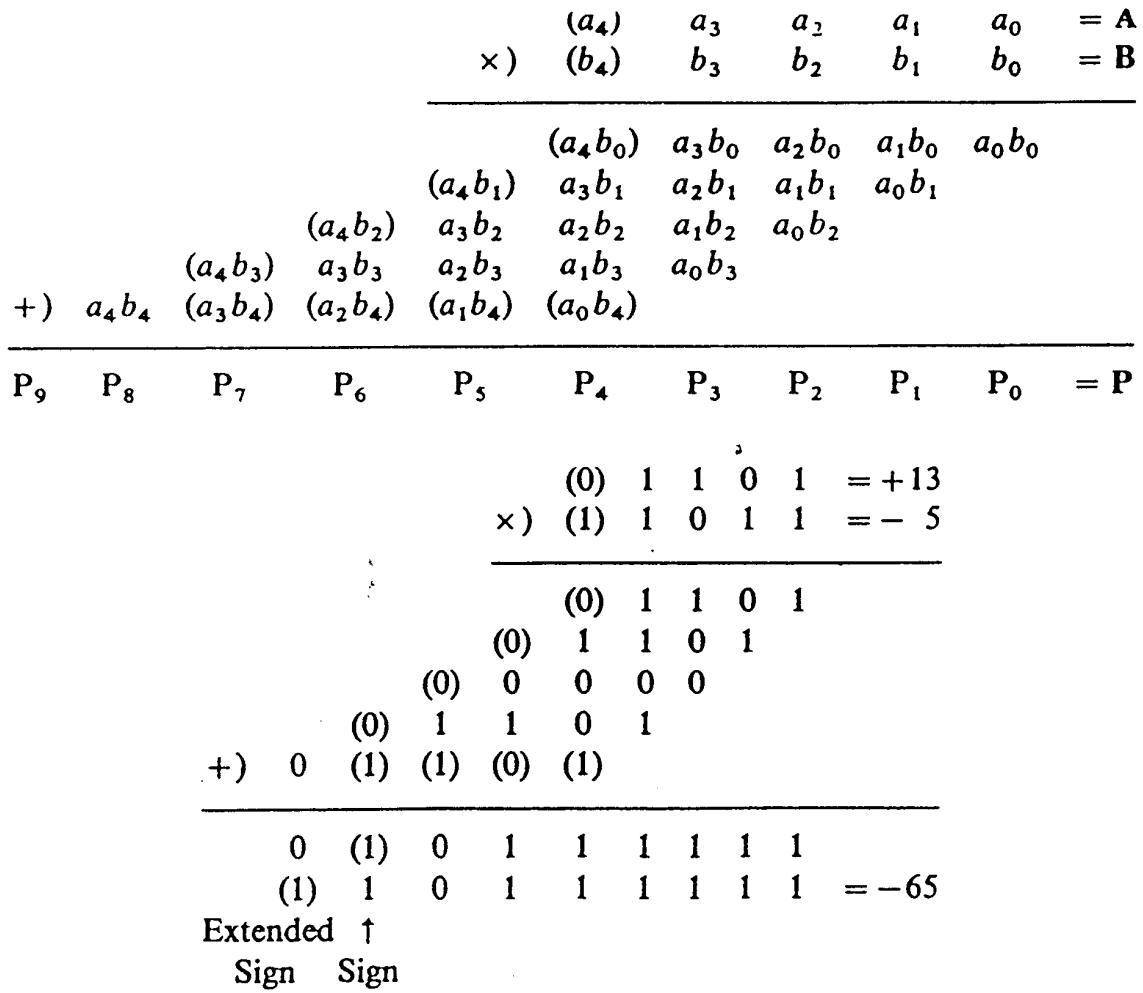
- Recall corrections required in general case
- For array multiplication, corrections can be included to carry and sum bits merging during summand additions
- Based on 2's complement number representation

$$X_v = -x_{n-1}r^{n-1} + \sum_{i=0}^{n-2} x_i r^i$$

- Negative operands causes negative summands
  - MSB digit is considered negative

EXAMPLE

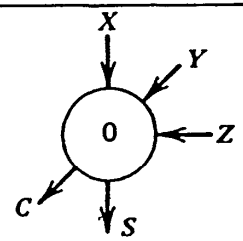
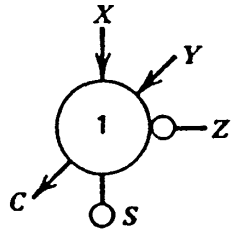
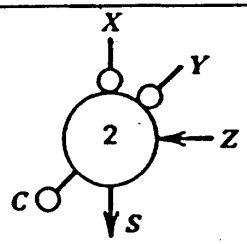
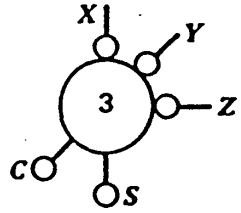
- Multiplication of +13 and -5
- Each single summand can be negative or positive
- Negative summands are marked with ()
- In the last line of summands, the complement of *A* is added: corresponds to the correction made in "cumulating partial products" multiplication



- Previous example implies that addition of summands requires also negatively weighted inputs for adders
- Generalized full adder can be used for additions
- Equations:

$$\text{Type 0 or Type 3} \begin{cases} S = \bar{X}\bar{Y}Z + \bar{X}Y\bar{Z} + X\bar{Y}\bar{Z} + XYZ \\ C = XY + YZ + ZX \end{cases}$$

$$\text{Type 1 or Type 2} \begin{cases} S = \bar{X}\bar{Y}Z + \bar{X}Y\bar{Z} + X\bar{Y}\bar{Z} + XYZ \\ C = XY + X\bar{Z} + Y\bar{Z} \end{cases}$$

Type	Logic Symbol	Operation
Type 0 Full Adder		X Y +) Z CS
Type 1 Full Adder		X Y +) −Z C(−S)
Type 2 Full Adder		−X −Y +) Z (−C) S
Type 3 Full Adder		−X −Y +) −Z (−C) (−S)

3.4 Pezaris multiplier

- Direct application of generalized full adders for direct 2's complement array multiplication leads to Pezaris multiplier

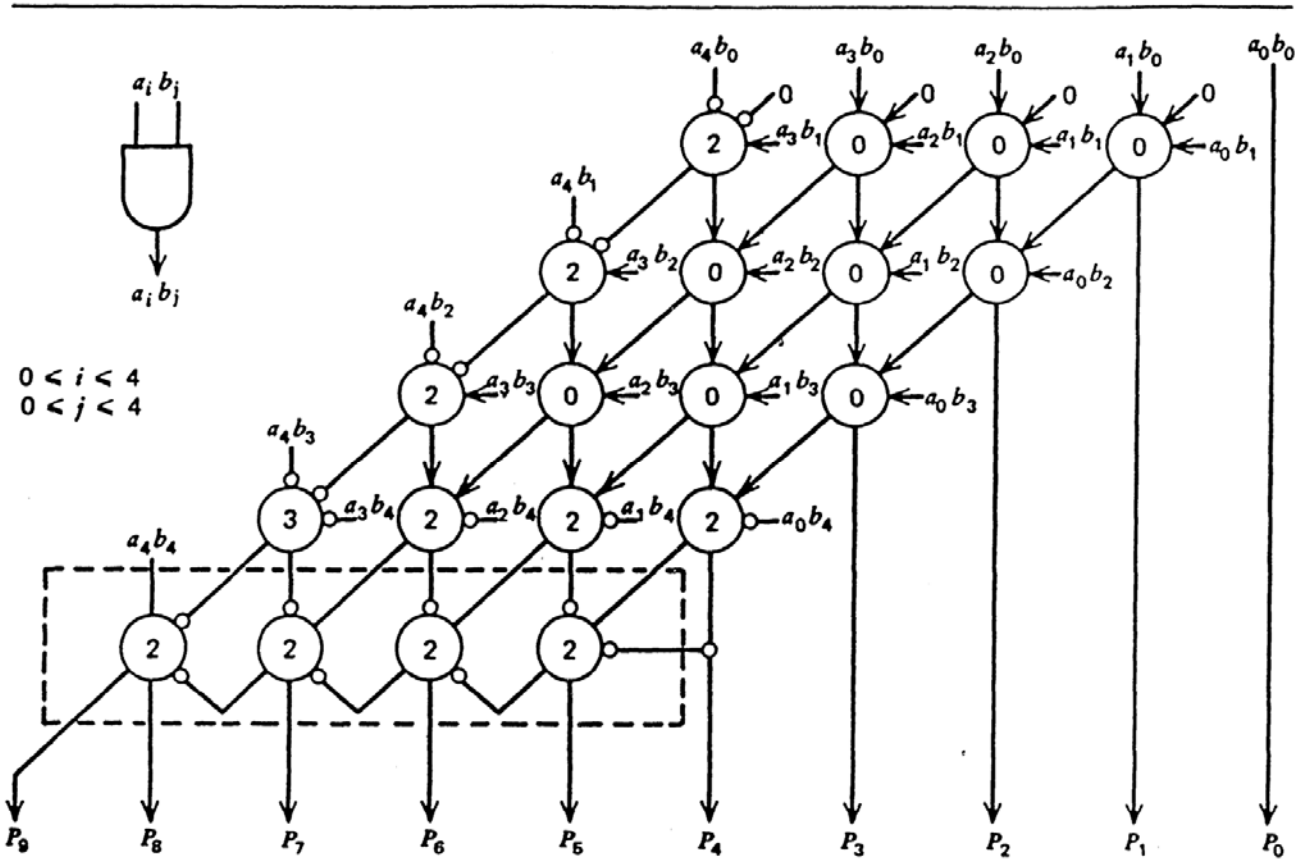


Figure 6.10 The schematic circuit diagram of a 5-by-5 Pezaris array multiplier.

- Pezaris multiplier has one large section which contains three types of full adders
- Improvement can be made by rearranging the summand matrix elements (without affecting column-wise additions)
- Tri-section multiplier has three different sections, each section uses only one type of adder
  - 3 FA types required
- Bi-section multiplier uses only 2 types of FA
  - First section adds all positive summands
  - Second section adds all negative summands

Bi-section

					(a_4)	a_3	a_2	a_1	a_0	$= A$
				$\times)$	(b_4)	b_3	b_2	b_1	b_0	$= B$
Positive section						a_3b_0	a_2b_0	a_1b_0	a_0b_0	
					a_3b_1	a_2b_1	a_1b_1	a_0b_1		
				a_3b_2	a_2b_2	a_1b_2	a_0b_2			
	a_4b_4	0	a_3b_3	a_2b_3	a_1b_3	a_0b_3				
		(a_4b_3)	(a_4b_2)	(a_4b_1)	(a_4b_0)					Negative
		(a_3b_4)	(a_2b_4)	(a_1b_4)	(a_0b_4)					section
	(P_9)	P_8	P_7	P_6	P_5	P_4	P_3	P_2	P_1	P_0 $= P$

$P = A \times B$

A, B, and P are all negatively signed tow's complement numbers.

Figure 6.12 The segregation of positive and negative summands in a 5-by-5 bisection array multiplier for two's complement numbers.

3.5 Baugh-Wooley's multiplier

- In all above cases, direct subtraction is performed (not addition of complement)
- Only normal FA:s can be used, if summands are arranged according to two's complement operation
  - ➔ Baugh-Wooley's multiplier

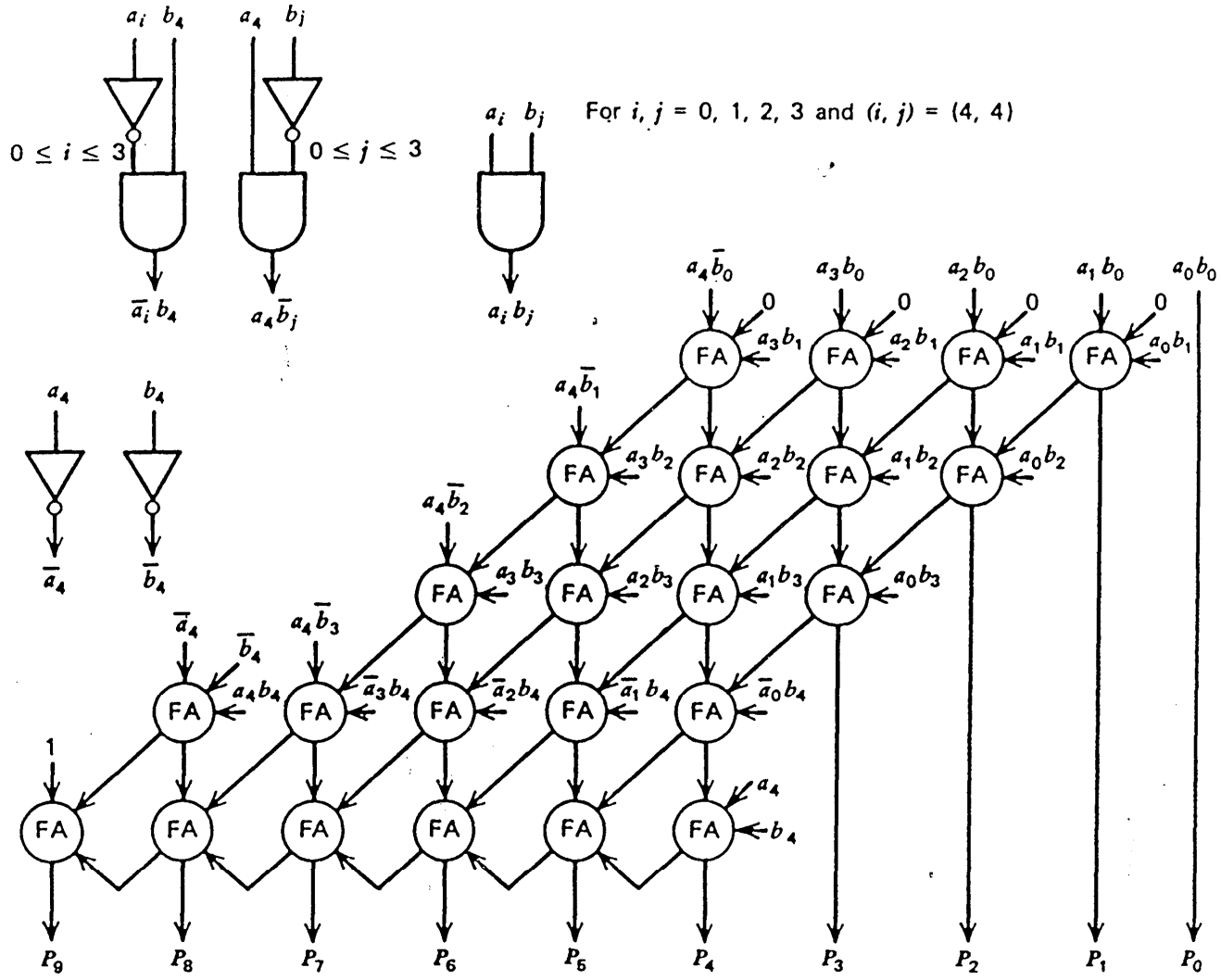


Figure 6.17 The schematic logic circuit diagram of a 5-by-5 Baugh–Wooley two's complement array

3.6 Recoded multiplier cellular array multiplication

- Partial product should be accumulated or subtracted depending on recoded multiplier bits
- For string of ones or zeros, only shifting is performed: unit should also be able to shift current row of summands
- Implementation requires controlled add-subtract-shift (CASS) cell

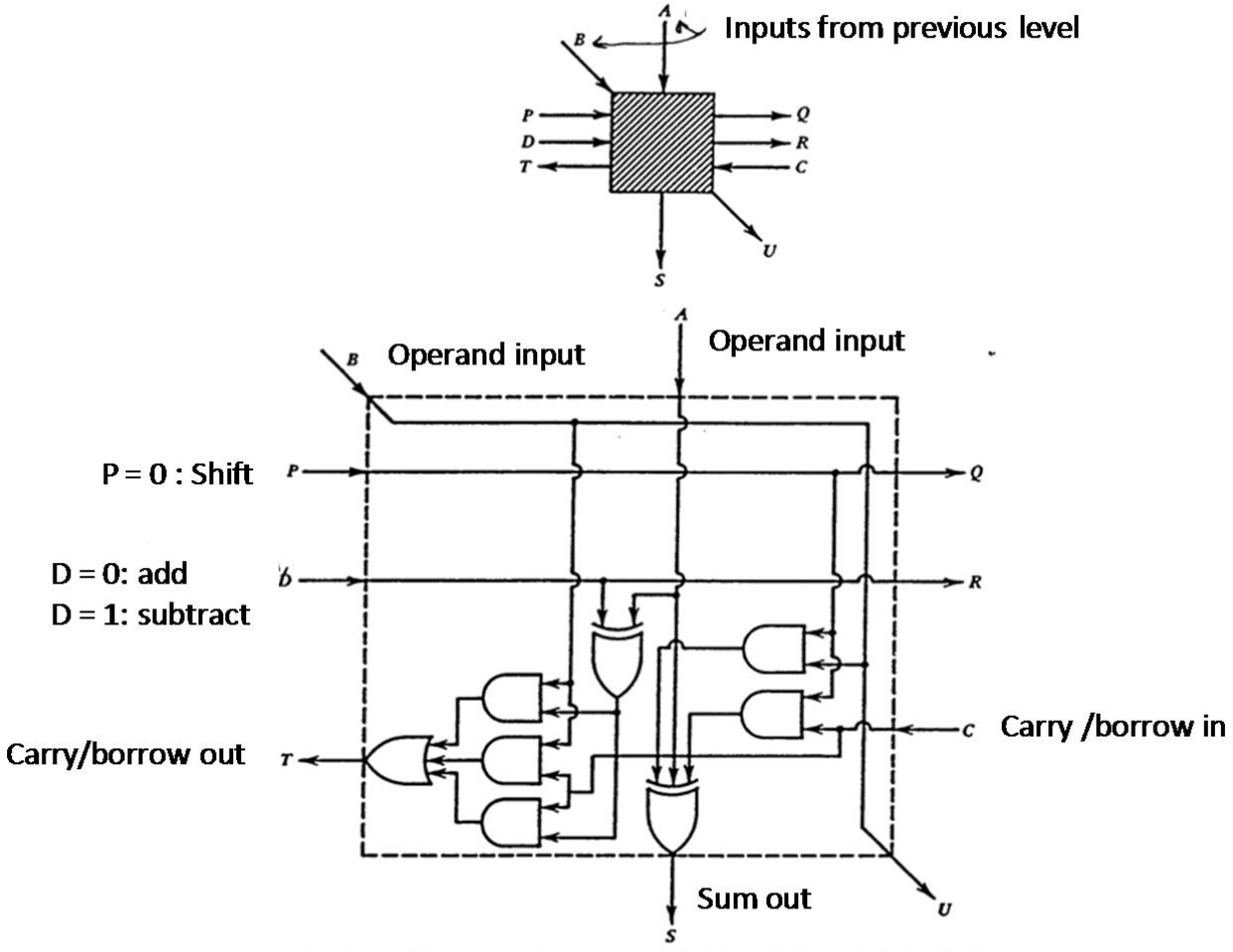
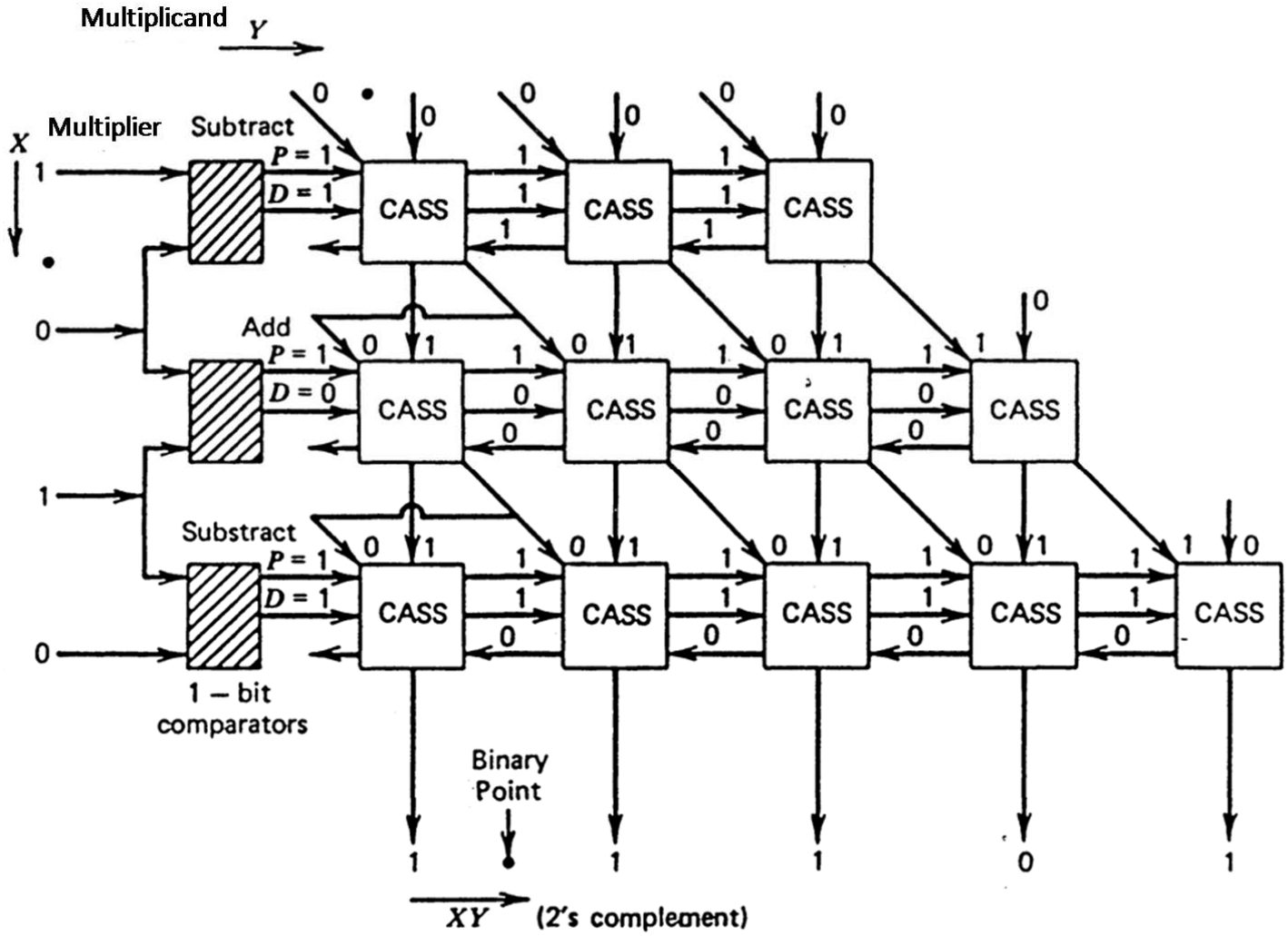


Figure 6.28 A schematic logic circuit diagram of the *Controlled Add-Subtract-Shift* (CASS) cell.

- 3x3 recoded cellular array multiplier



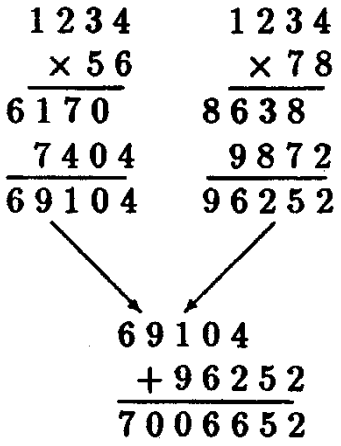
4. MODULAR MULTIPLICATION

- Large multipliers can be constructed from smaller

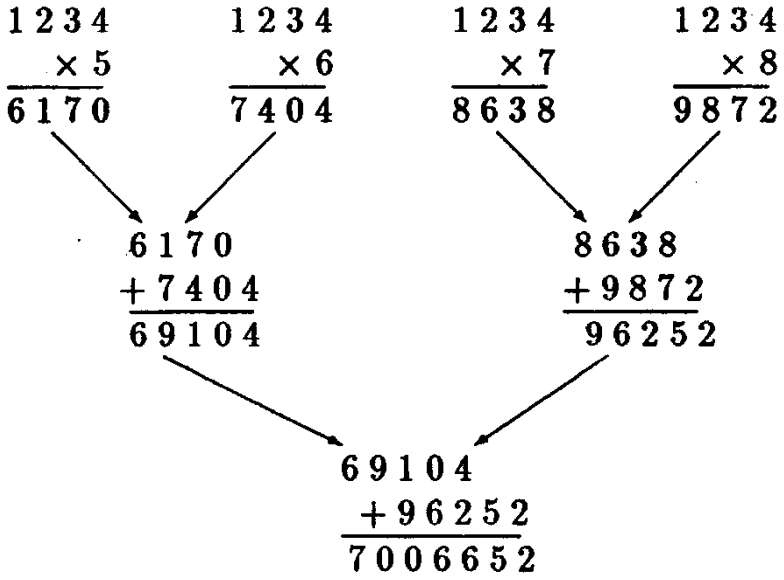
EXAMPLE

Consider multiplication of A = 1234 and B = 5678 performing addition of several multiples simultaneously (this is the same principle as in previous carry-save adder based multipliers)

Multiplier sliced in two



Multiplier sliced in four



EXAMPLE

Construct $2n \times 2n$ bit multiplier using *n*-bit multipliers

Let $A = a_{2n-1}, ..., a_0$

$B = b_{2n-1}, ..., b_0$

A_L, B_L be the least significant parts of operands (*n*-bits)

A_H, B_H be the most significant parts of operands

Then

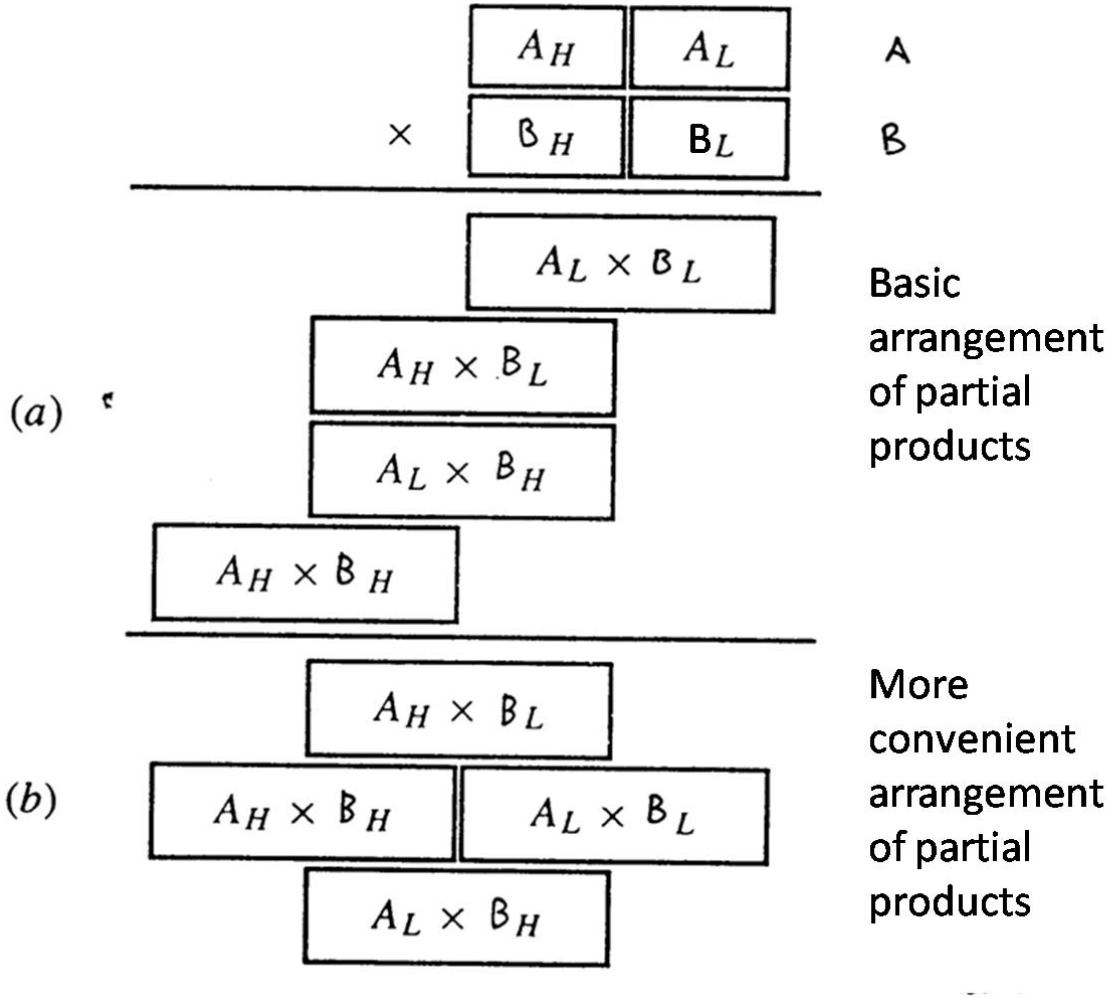
$$AB = (A_H 2^n + A_L)(B_H 2^n + B_L)$$

$$= A_H B_H 2^{2n} + A_H B_L 2^n + A_L B_H 2^n + A_L B_L$$

$$= A_H B_H 2^{2n} + (A_H B_L + A_L B_H) 2^n + A_L B_L$$

- Block diagram

$$AB = A_H B_H 2^{2n} + (A_H B_L + A_L B_H)2^n + A_L B_L$$



- Subproduct columns are added up for final result
- Max 3-input adders required for final stage and 4 multipliers to produce subproducts
- Each subproduct can be generated by some multiplier, e.g. cellular array multiplier (Braun multiplier may be called Non-additive Multiplier Module NMM)
- $2n \times 2n$ bit multiplier using $2^2 = 4$ *n*-bit multipliers can be extended to any power of two
- $kn \times kn$ bit multiplier uses k^2 *n*-bit multipliers
- Partial products can be arranged as follows (Wallace)

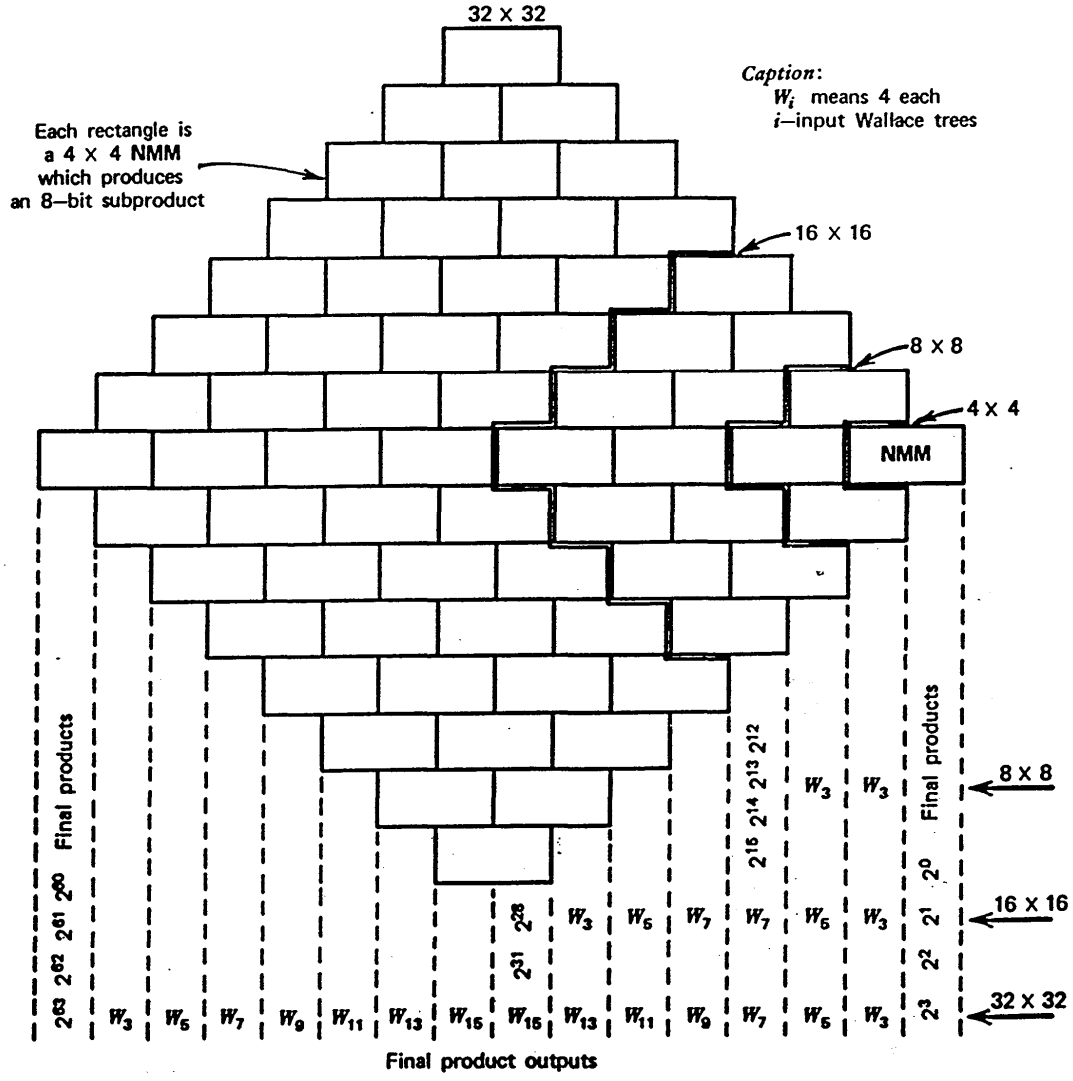


Figure 6.7 Array arrangement for various multipliers from size 4-by-4 to 32-by-32.

5. LOOK-UP TABLE MULTIPLICATION

- Very fast, if large memory is available
- In multiplication, the table size grows quite rapidly with the width of the operands

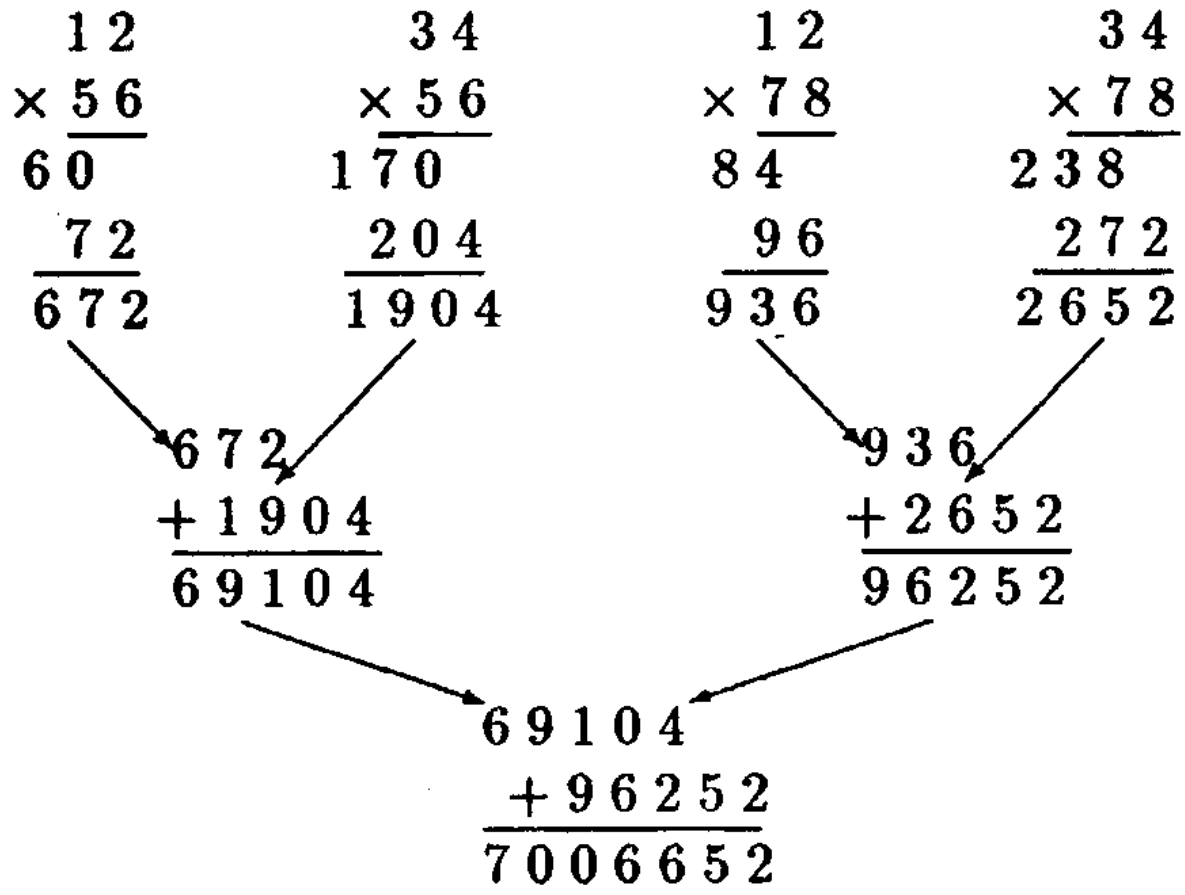
EXAMPLE

8x8 bit multiplier requires 64k x 16bit memory

- Practical multiplier implementation based on lookup table idea combines a number of small tables and small adders
  - Multiplicand and/or multiplier is split
  - Lookup table is used to obtain the summands
  - Tree of adders is used to add the summands

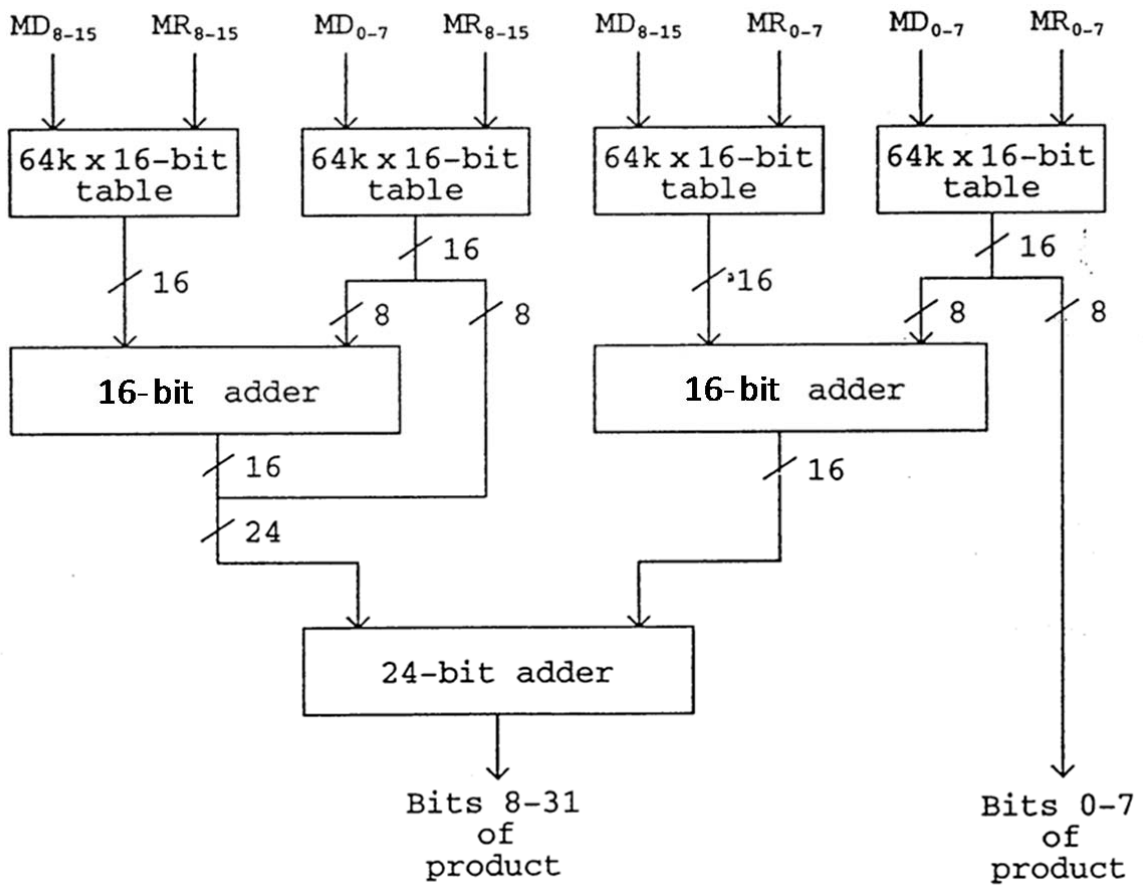
EXAMPLE

Consider multiplication of A = 1234 and B = 5678 (A and B sliced in two)



EXAMPLE

Implementation of 16x16 bit multiplier with four 8x8 bit look-up table multipliers and adders



MD = multiplicand
MR = multiplier

6.IMPLEMENTING MULTIPLICATION ON FPGA

- Two possibilities
  - Use hard macros for multiplication
  - Implement LUT-based multipliers

6.1 Hard macros for multiplication

- Many FPGA vendors incorporate special hard-wired multiplier blocks on FPGA
  - Increase processing speed
  - Reduce area
- Features of hard macros differ between FPGA vendors, vendor-specific examples of hard macros for multiplication
  - Xilinx: Embedded 18x18 bit signed multipliers
  - Altera: Embedded DSP blocks
  - Lattice: Booth multiplication logic
- Recommended to use

6.2 LUT-based multipliers

- LUT-based multipliers are inherently slow
  - Large number of programmable logic blocks are connected together
- Sometimes LUT-based multiplication may still be needed
  - Design incorporates more multipliers than FPGA provides
  - Hard macro does not support the desired special multiplication