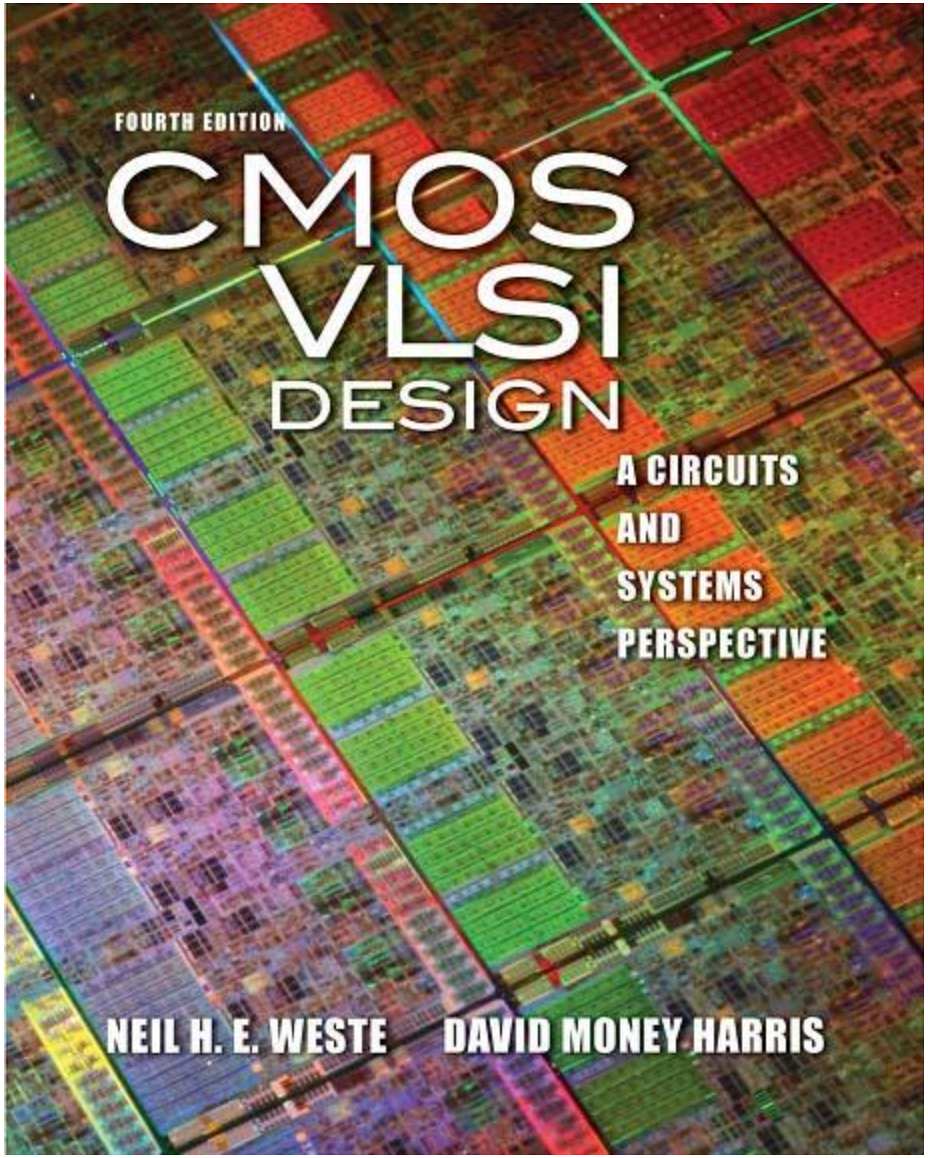
CPE 110408423
VLSI Design
Chapter 11: Datapath Subsystems

Bassam Jammil

[Computer Engineering Department,

Hashemite University]



Lecture 2: Datapath Functional Units

Outline

- What is datapath
- Multi-input Adders
- 1's & 0's Detectors
- Comparators
- Shifters
- Multipliers

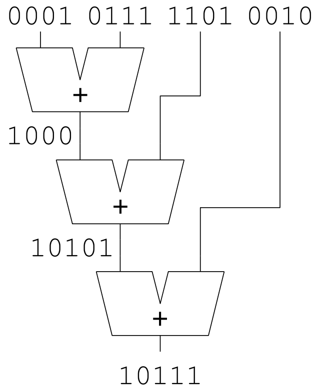
Datapath

- CPU consists mainly of:
  - Datapath: functional units which perform operations on data
  - Control: sequences datapath, memory, ...
  - Memory elements
  - Special purpose cells
    - I/O
    - Power distribution
    - Clock generation and distribution
    - Analog and RF

11.2.4 Multi-input Adders

- Suppose we want to add ***k*** N-bit words
  - Ex: 0001 + 0111 + 1101 + 0010 = 10111
- Straightforward solution: ***k-1*** N-input cascaded carry-propagation adders (CPAs).
  - The main problems: Large and slow

Adding 4 4-bit numbers
using three cascaded CPAs.
Note: ***k***=4



Carry-Save Adder (CSA)

- CSA is a full adder sums 3 inputs and produces 2 outputs
  - Carry output has twice *weight* of sum output
- N full adders in parallel are called *carry save adder*
  - Produce N sums and N carry outs

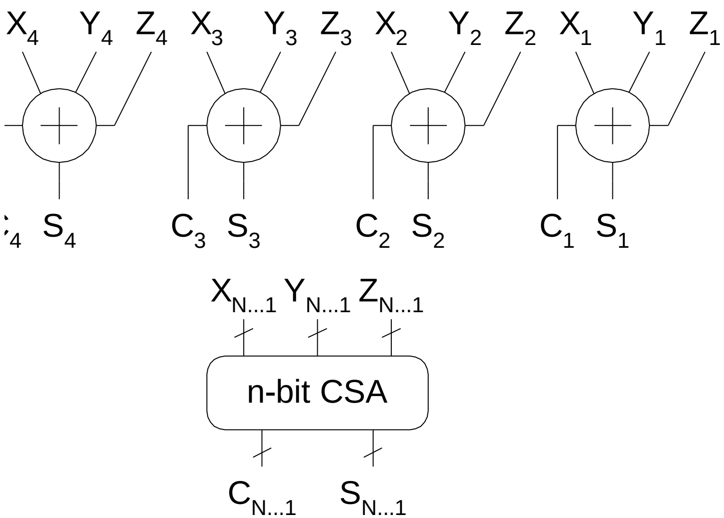
The weight of X_i , Y_i , Z_i and S_i is 2^{i-1}.

The weight of Ci is 2^i.

$X+Y+Z = S + 2\,C$

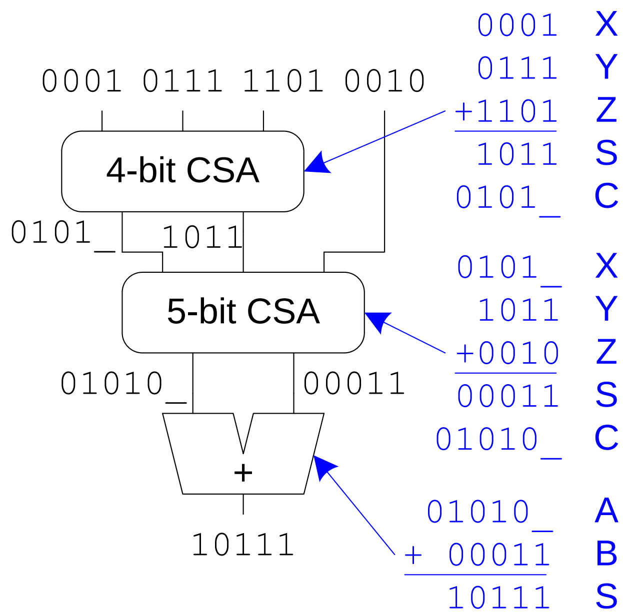
So, the carry is not propagated, rather it is saved. Hence, the name carry save adder.

When one of the inputs to a CSA is a constant, the hardware can be reduced further. If a bit of the input is 0, the CSA column reduces to a half-adder. If the bit is 1, the CSA column simplifies to $S=A\oplus B$ and $C = A+B$.



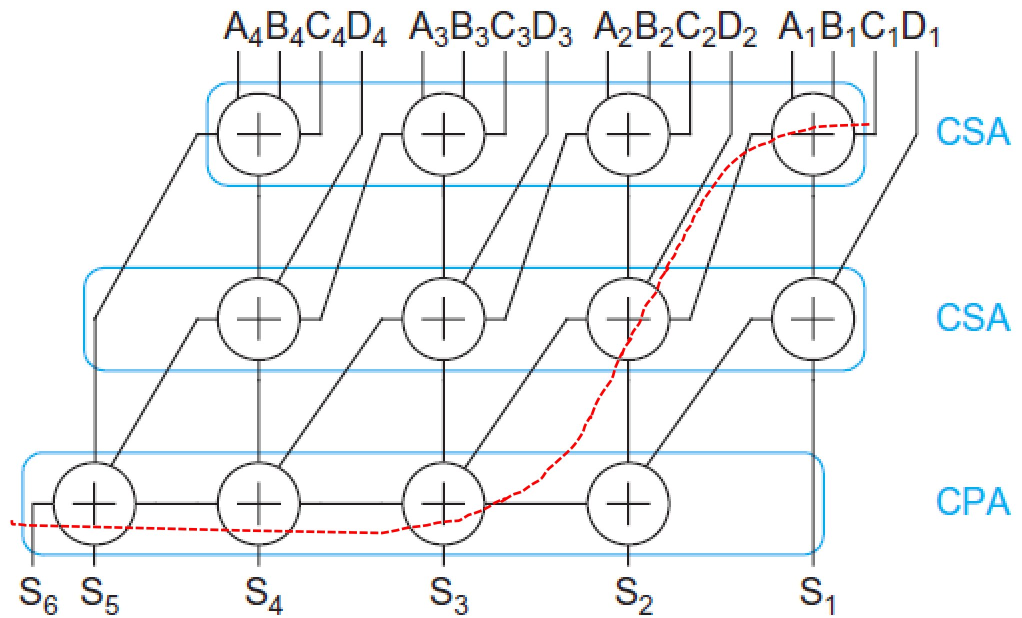
CSA Application

- Use k-2 stages of CSAs
  - – Keep result in carry-save redundant form
- Final CPA computes actual result



The critical path of adding k numbers

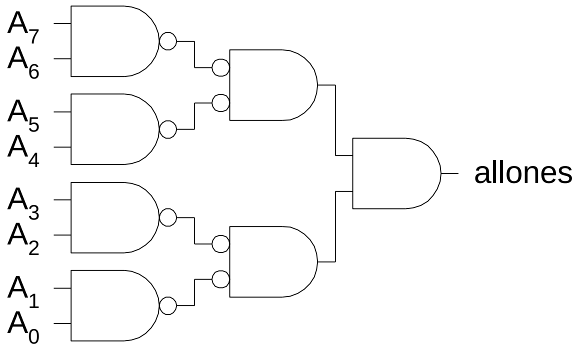
- In general *k* numbers can be summed with *k – 2 [3:2] CSAs and only one CPA.*
- This is much faster compared with *k-1* CPAs.

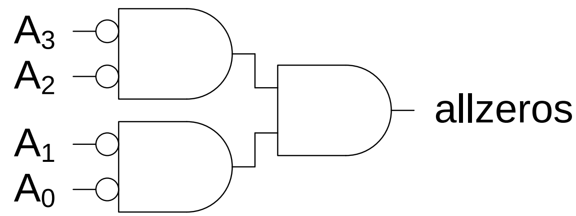


11.3 1's & 0's Detectors

- 0's detector: A = 00…000
  - It is a function which outputs 1 when all N inputs are zeros.
  - Designed using N-input AND gate
- 1's detector: A = 11…111
  - It is a function which outputs 1 when all N inputs are ones.
  - Designed with
    - NOR gate
    - NOTs + 1's detector

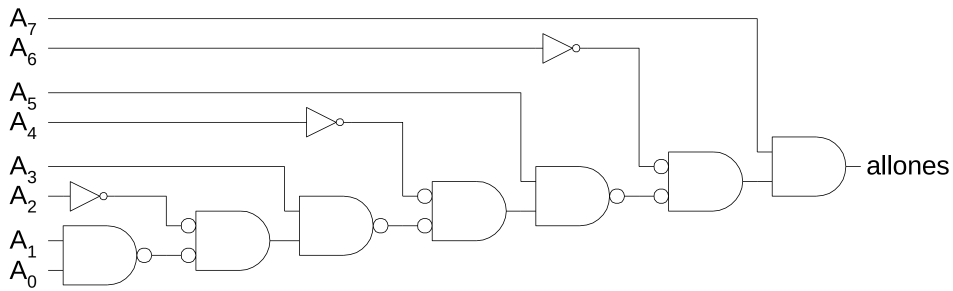
11.3 1's & 0's Detectors





The gates can be built in a binary tree style. The timing path has log *N* stages.

If inputs arrives in different times, then build the design in a sequential /cascaded style. The timing path is *N-1* stages.

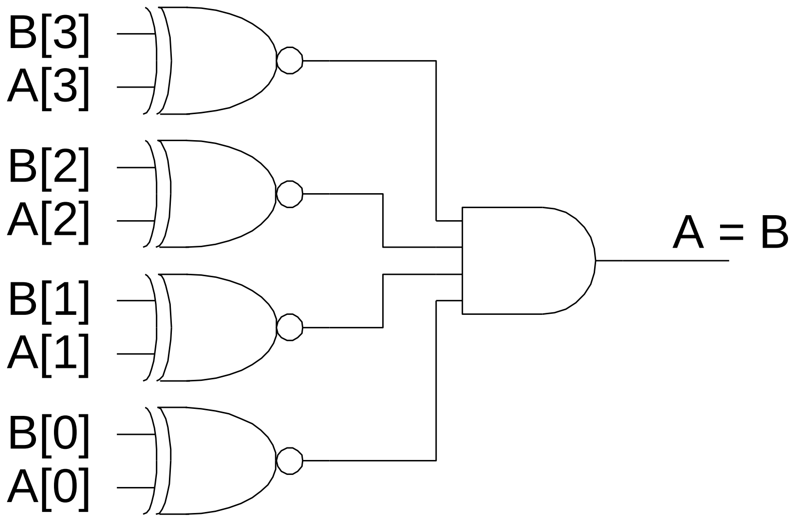


11.4 Comparators: unsigned numbers

- Naming convention
  - 0's detector: A = 00…000
  - 1's detector: A = 11…111
  - Equality comparator: A = B
  - Magnitude comparator: A < B
- A *magnitude comparator* determines the larger of two binary numbers.
- *To compare two* **unsigned** numbers *A and B, compute* $B - A = B + \sim A + 1$.
  - *If there is a carry-out,* $A \le B$*;* otherwise, $A > B$.
  - *A zero detector indicates that the numbers are equal.*

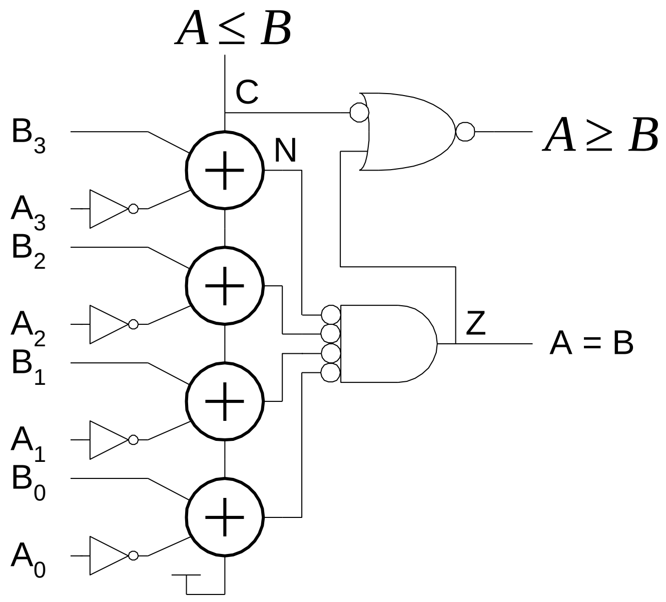
Equality Comparator

- Check if each bit is equal (XNOR, aka equality gate)
- 1's detect on bitwise equality



Magnitude Comparator

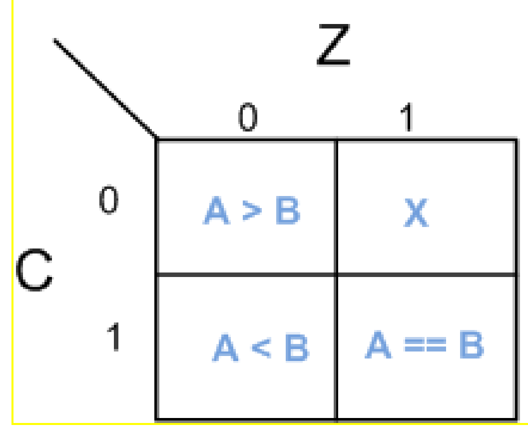
- Compute B – A and look at sign
- B – A = B + ~A + 1
- For unsigned numbers, carry out is sign bit



Unsigned Magnitute Comparator

Relation	Unsigned Comparison
$A = B$	Z
$A \neq B$	$\overline{Z}$
$A < B$	$C \cdot \overline{Z}$
$A > B$	$\overline{C}$
$A \leq B$	C
$A \geq B$	$\overline{C} + Z$

Unsigned comparator Flags



Comparators: signed vs. unsigned

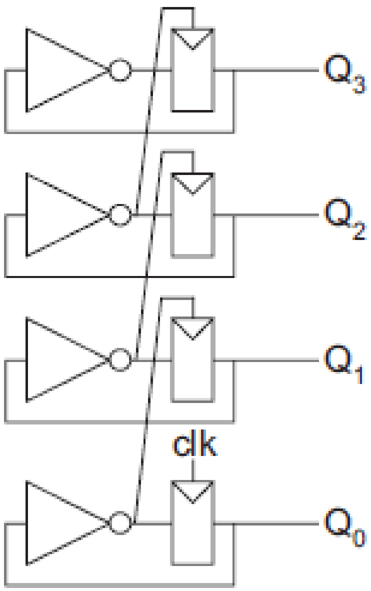
- Reading only
- For signed numbers, comparison is harder and depends on the following flags.
  - C: carry out
  - Z: zero (all bits of B – A are 0)
  - N: negative (MSB of result)
  - V: overflow (inputs had different signs, output sign ≠ B)
  - S: N xor V (sign of result)

11.5 Counters

- Two commonly used types of counters are binary counters and **linear-feedback shift registers**.
- An N-bit binary counter sequences through 2^N outputs in binary order.
- An N-bit linear-feedback shift register sequences through up to $2^N - 1$ outputs in pseudo-random order
- Some of the common features of counters include the following: *Resettable, Loadable, Enabled Reversible(UP/DOWN input), Terminal Count (TC output asserted when counter overflows* or underflows).

11.5.1 Binary Counters

- Asynchronous Counter
  - It is composed of *N registers connected in toggle configuration, where the falling* transition of each register clocks the subsequent register.
  - Therefore, the delay can be quite long.
  - It has no reset signal, making it difficult to test.
  - It is good frequency divider.



Binary Counters

- Shown below:
  - (a) up/down counter
  - (b) up counter (incrementer)
  - (c) Fully featured counter with reset/load/ up/dn.

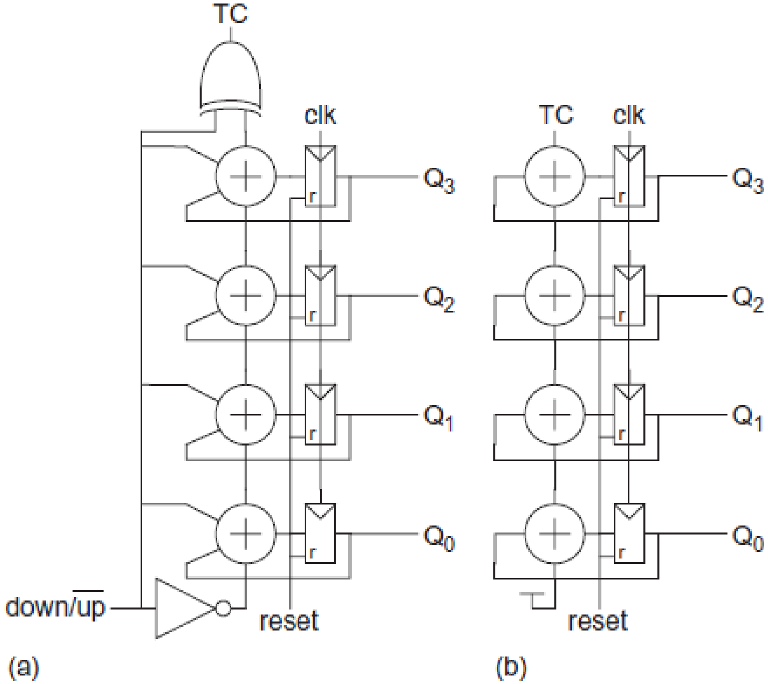


FIGURE 11.49 Synchronous counters

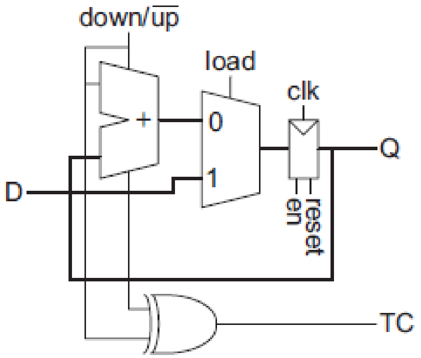


FIGURE 11.50 Synchronous up/down counter with reset, load, and enable

11.5.4 Linear-Feedback Shift Register

- A linear-feedback shift register (LFSR) consists of *N registers configured as a shift register.* The input to the shift register comes from the XOR of particular bits of the register, as shown in the Figure for a 3-bit LFSR. On reset, the registers must be initialized to a nonzero value (e.g., all 1s). The pattern of outputs for the LFSR is shown in the Table.
- The output *Y follows the 7-bit sequence [1110010].* This is an example of a *pseudorandom bit sequence (PRBS).*

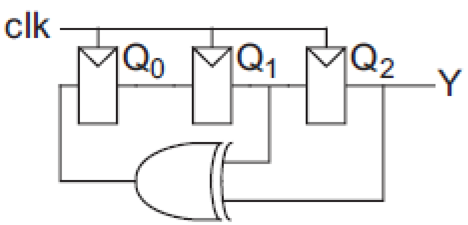


FIGURE 11.54 3-bit LFSR

TABLE 11.7 LFSR sequence

Cycle	Q_0	Q_1	Q_2 / Y
0	1	1	1
1	0	1	1
2	0	0	1
3	1	0	0
4	0	1	0
5	1	0	1
6	1	1	0
7	1	1	1
Repeats forever			

Linear-Feedback Shift Register

- **Maximal-length shift register** outputs sequences through all $2^n - 1$ combinations (excluding all 0s).
  - Such as the example in the previous slide.
- The inputs fed to the XOR are called the tap sequence and are often specified with a **characteristic polynomial**.
- For example, this 3-bit LFSR has the characteristic polynomial $1 + x^2 + x^3$ because the taps come after the second and third registers.

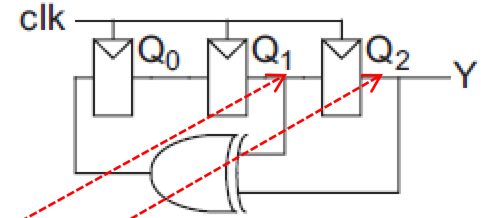


FIGURE 11.54 3-bit LFSR

LFSR Example

Example 11.1

Sketch an 8-bit linear-feedback shift register. How long is the pseudo-random bit sequence that it produces?

SOLUTION: Figure 11.55 shows an 8-bit LFSR using the four taps after the 1st, 6th, 7th, and 8th bits, as given in Table 11.7. It produces a sequence of $2^8 - 1 = 255$ bits before repeating.

Characteristic Polynomial is:
$1 + x^1 + x^6 + x^7 + x^8$

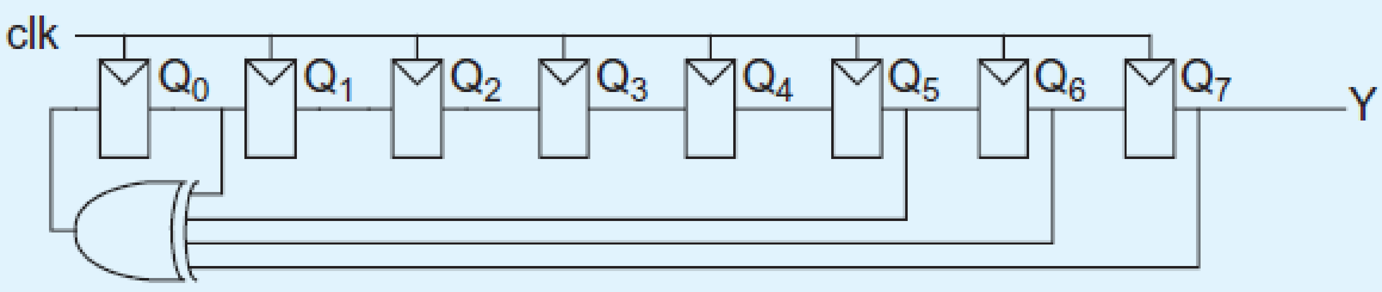


FIGURE 11.55 8-bit LFSR

11.8 Shifters

- Shifts can either be performed by a constant or variable amount.
  - Constant shifts are trivial in hardware, requiring only wires. They are also an efficient way to perform multiplication or division by powers of two.
  - A variable shifter takes an *N-bit input, A, a shift amount, k,* and control signals indicating the shift type and direction. It produces an *N-bit output, Y.*

Shifters

- There are three common types of variable shifts, each of which can be to the left or right:

1. Logical Shift:

- Shifts number left or right and fills with 0's
  - 1011 LSR 1 = $\boxed{0101}$ 1011 LSL1 = $\boxed{0110}$

2. Arithmetic Shift:

- Shifts number left or right. Right shift sign extends
  - 1011 ASR1 = $\boxed{1101}$ 1011 ASL1 = $\boxed{0110}$

3. Rotate:

- Shifts number left or right and fills with lost bits
  - 1011 ROR1 = $\boxed{1101}$ 1011 ROL1 = $\boxed{0111}$

Implementing Rotate Operation

- Array shifter
  - Involves an array of N N-input multiplexers to select each of the outputs from each of the possible input positions.
  - It requires a decoder to produce the 1-of-N-hot shift amount.
  - In practice, multiplexers with more than 4–8 inputs have excessive parasitic capacitance
  - , so they are faster
- Logarithmic shifter
  - Uses $\log_v$ N levels of v-input multiplexers
  - For example, in a radix-2 logarithmic shifter, the first level shifts by N/2, the second by N/4, and so forth until the final level shifts by 1.
  - In a logarithmic shifter, no decoder is necessary.

Implementing Rotate Operation

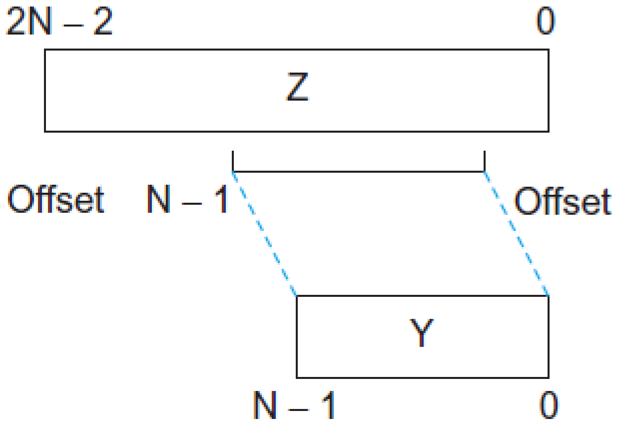
- A left rotate by *k bits is equivalent to a right rotate by* ***N – k*** *bits.*
  - *Computing* ***N – k*** requires a subtracter in the critical path.
  - ***N – k*** = *N + k + 1 = k + 1. Thus, the left* rotate can be performed by preshifting right by 1, then doing a right rotate by the complemented shift amount.
- Logical and arithmetic shifts are similar to rotates, but must replace bits at one end or the other with a *kill value (either 0 or the sign bit).*

Shift Architectures

- The two major shifter architectures are **funnel shifters** and **barrel shifters**.
- Funnel shifter
  - the kill values are incorporated at the beginning,
- Barrel shifter
  - the kill values are chosen at the end
- Comparison:
  - Both barrel and funnel shifters can use array or logarithmic implementations.
  - For general-purpose shifting, both architectures are comparable in energy and delay.
  - If only shift operations (but not rotates) are required, the funnel architecture is simpler, while if only rotates (but not shifts) are required, the barrel is simpler.

11.8.1 Funnel Shifter

- The funnel shifter creates a 2N – 1-bit input word ***Z*** from ***A*** and/or the kill values, then selects an N-bit field from this input word
- A funnel shifter can do all six types of shifts
- Selects N-bit field Y from 2N–1-bit input
  - Shift by k bits ($0 \leq k < N$)
  - Logically involves N N:1 multiplexers

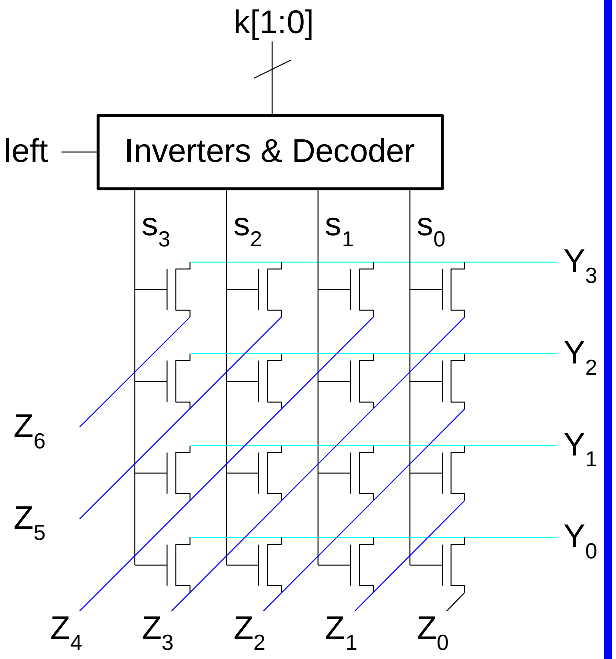


Funnel Source Generator

Shift Type	$Z_{2N-2:N}$	Z_{N-1}	$Z_{N-2:0}$	Offset
Rotate Right	$A_{N-2:0}$	A_{N-1}	$A_{N-2:0}$	k
Logical Right	0	A_{N-1}	$A_{N-2:0}$	k
Arithmetic Right	sign	A_{N-1}	$A_{N-2:0}$	k
Rotate Left	$A_{N-1:1}$	A_0	$A_{N-1:1}$	$\bar{k}$
Logical/Arithmetic Left	$A_{N-1:1}$	A_0	0	$\bar{k}$

Array Funnel Shifter

- N N-input multiplexers
  - Use 1-of-N hot (one multiplexer for each output bit) select signals for shift amount
  - nMOS pass transistor design (V_t drops!)
- The shift amount is conditionally inverted (for left shifts) and decoded into select signals that are fed vertically across the array. The outputs are taken horizontally.
- Each row of transistors attached to an output forms one of the multiplexers. The 2N – 1 inputs run diagonally to the appropriate mux inputs.



Array Funnel Shifter

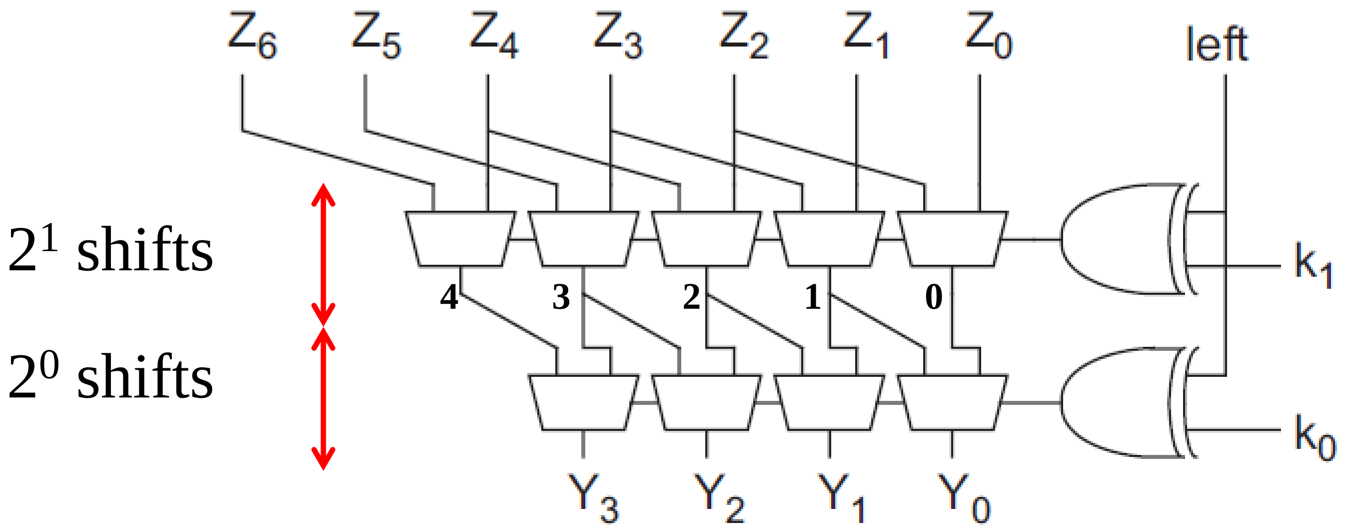
Shift Right	k	Z6	Z5	Z4	Z3	Z2	Z1	Z0
0	0	0	0	0	A3	A2	A1	A0
1	1	0	0	0	A3	A2	A1	A0
2	2	0	0	0	A3	A2	A1	A0
3	3	0	0	0	A3	A2	A1	A0

Shift Left	k	Z6	Z5	Z4	Z3	Z2	Z1	Z0
0	3	A3	A2	A1	A0	0	0	0
1	2	A3	A2	A1	A0	0	0	0
2	1	A3	A2	A1	A0	0	0	0
3	0	A3	A2	A1	A0	0	0	0

Shifter output

Logarithmic Funnel Shifter

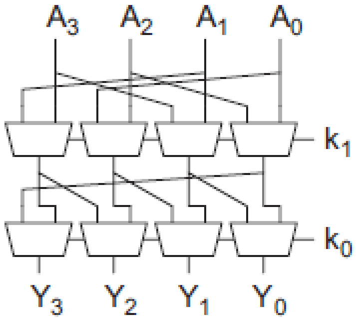
- Log N stages of 2-input muxes
  - No select decoding needed
  - The XOR gates on the control inputs conditionally invert the shift amount for left shifts.



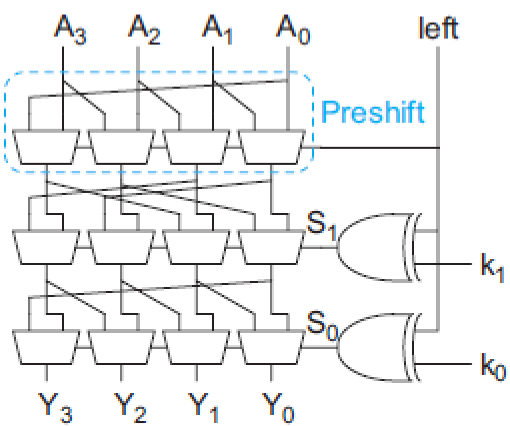
11.8.2 Barrel Shifter

- Barrel shifters perform right rotations using wrap-around wires.
  - Unlike funnel shifters, barrel shifters contain long wrap-around wires. In a large shifter, it is beneficial to upsize or buffer the drivers for these wires.
- Left rotations are right rotations by $N - k = \overline{k} + 1$ bits.
- Shifts are rotations with the end bits masked off.
- Barrel shifters come in array and logarithmic forms; we focus on logarithmic barrel.

Logarithmic Barrel Shifter

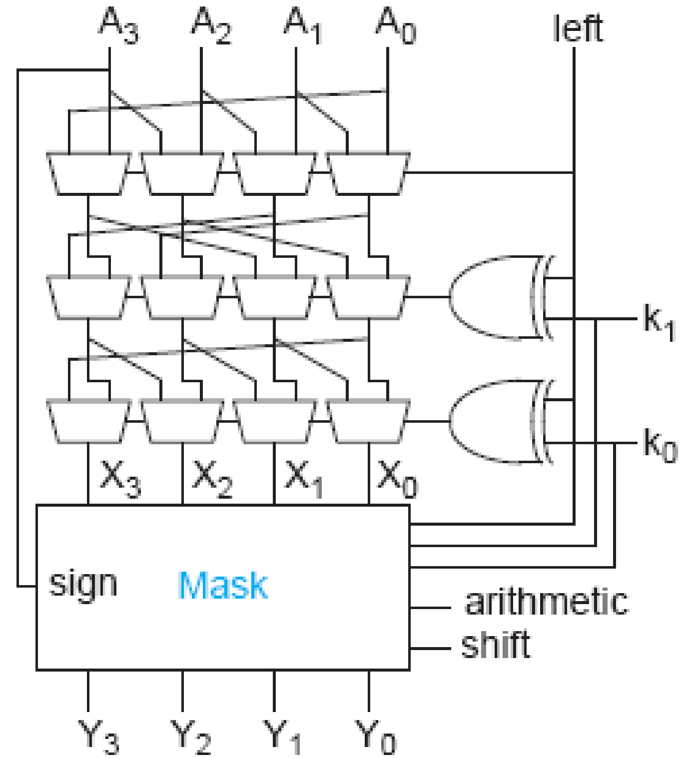


Right shift only



Right/Left shift

Rotate left by prerotating right by 1, then rotating right by *~k*.

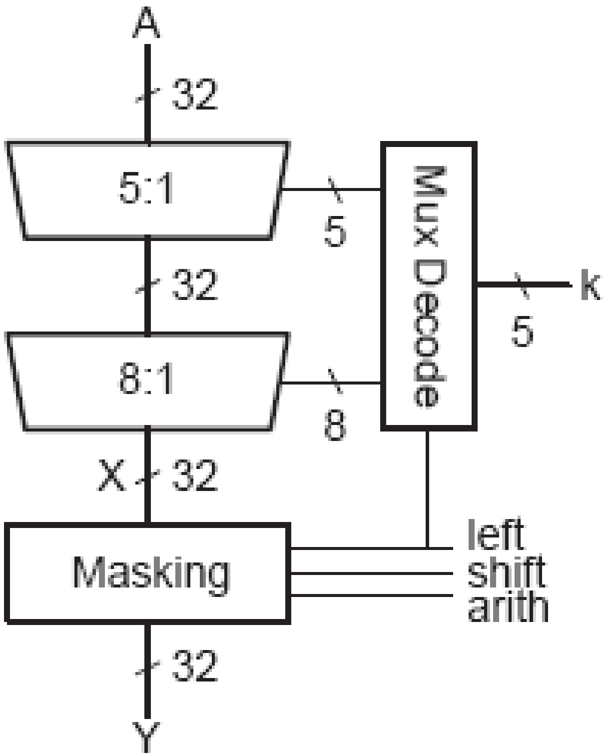


Right/Left Shift & Rotate

Performing logical or arithmetic shifts on a barrel shifter requires a way to mask out the bits that are rotated off the end of the shifter

32-bit Logarithmic Barrel (read)

- Datapath never wider than 32 bits
- First stage preshifts by 1 to handle left shifts and for rotate rights by 0, 1, 2,3 or 4 bits.
- Second stage rotate rights by by 0, 4, 8, 12 16, 20, 24 or 28 bits
- The masking unit generates an N-bit mask with ones.



11.9 Multiplication

- Example:

```
        011001 : 25₁₀        multiplicand
      × 100111 : 39₁₀        multiplier
        011001
       011001
      011001                 partial
     000000                  products
    000000
  +011001
  001111001111 : 975₁₀       product
```

- M x N-bit multiplication
  - Produce N partial products, each partial product is M-bit .
  - Sum these to produce (M+N)-bit product

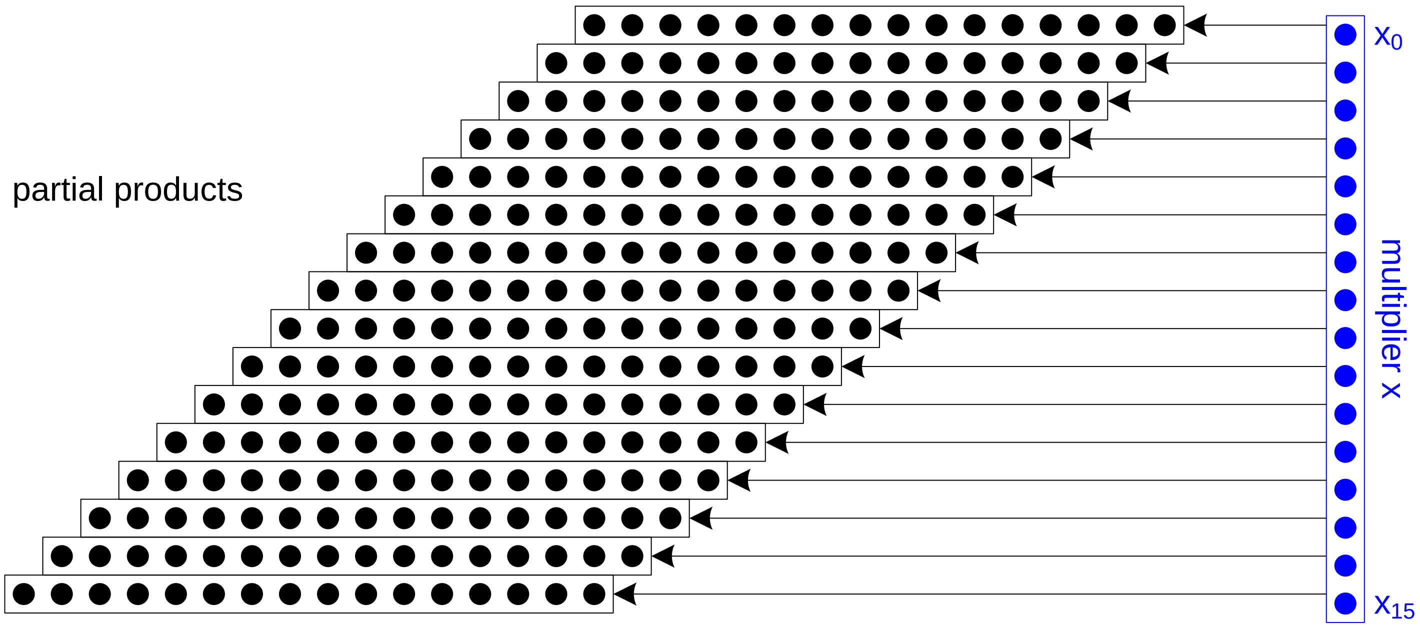
General Form

- Multiplicand: $Y = (y_{M-1}, y_{M-2}, \ldots, y_1, y_0)$
- Multiplier: $X = (x_{N-1}, x_{N-2}, \ldots, x_1, x_0)$
- Product: $$P = \left(\sum_{j=0}^{M-1} y_j 2^j\right)\left(\sum_{i=0}^{N-1} x_i 2^i\right) = \sum_{i=0}^{N-1}\sum_{j=0}^{M-1} x_i y_j 2^{i+j}$$

						y_5	y_4	y_3	y_2	y_1	y_0	multiplicand
						x_5	x_4	x_3	x_2	x_1	x_0	multiplier
						x_0y_5	x_0y_4	x_0y_3	x_0y_2	x_0y_1	x_0y_0	
					x_1y_5	x_1y_4	x_1y_3	x_1y_2	x_1y_1	x_1y_0		
				x_2y_5	x_2y_4	x_2y_3	x_2y_2	x_2y_1	x_2y_0			partial products
			x_3y_5	x_3y_4	x_3y_3	x_3y_2	x_3y_1	x_3y_0				
		x_4y_5	x_4y_4	x_4y_3	x_4y_2	x_4y_1	x_4y_0					
	x_5y_5	x_5y_4	x_5y_3	x_5y_2	x_5y_1	x_5y_0						
p_{11}	p_{10}	p_9	p_8	p_7	p_6	p_5	p_4	p_3	p_2	p_1	p_0	product

Dot Diagram

- Large multiplications can be more conveniently illustrated using *dot diagrams.*
- Each dot represents a bit
- The partial products are represented by a horizontal boxed row of dots, shifted according to their weight.
- The multiplier bits used to generate the partial products are shown on the right.

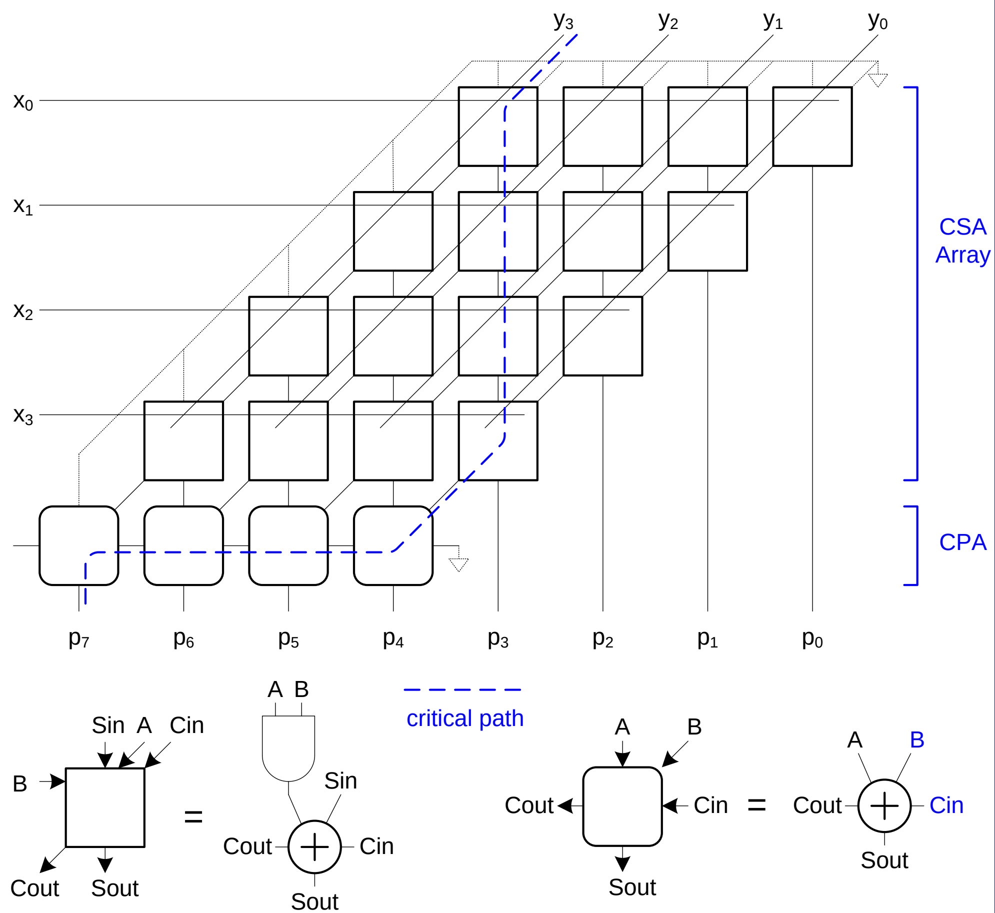


Multiplication Techniques

- There are a number of techniques that can be used to perform multiplication. The choice is based upon factors such as latency, throughput, energy, area, and design complexity.
- One approach is to use an *M + 1-bit* carry-propagate adder (CPA) :
  - Add the first two partial products, then another CPA to add the third partial product to the running sum, and so forth.
  - Such an approach requires *N – 1 CPAs and is slow, even if a* fast CPA is employed.
- More efficient **parallel** approaches use some sort of **array or tree** of full adders to sum the partial products.

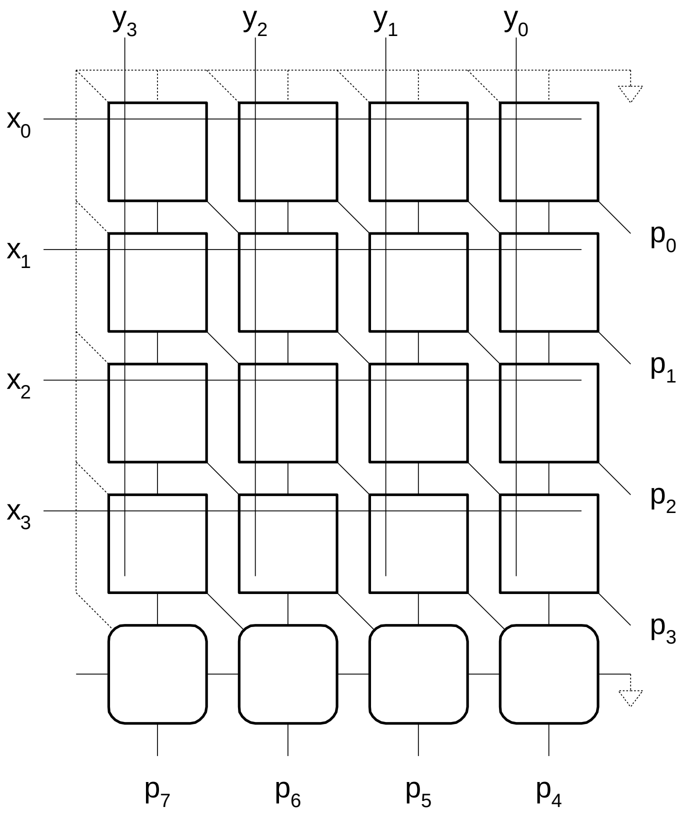
11.9.1 Array Multiplier

- The first row converts the first partial product into carry-save redundant form.
- Each later row uses the CSA to add the corresponding partial product to the carry-save redundant result of the previous row and generate a carry-save redundant result.
- Note that the first row of CSAs adds the first partial product to a pair of 0s.
- The least significant N output bits are available as sum outputs directly from CSAs. The most significant output bits arrive in carry-save redundant form and require an M-bit carry-propagate adder to convert into regular binary form.
- The first row of CSAs can be used to add the first three partial products together. **Critical path: *N–2 CSAs and a CPA.***
- To improve speed: replace the bottom row with a faster CPA such as a lookahead or tree adder.



Rectangular Array

- Squash array to fit rectangular floorplan
- circuits are assigned rectangular blocks in the floorplan so the parallelogram shape wastes space.



11.9.3 Fewer Partial Products

- Array multiplier requires N partial products
- If we looked at groups of r bits, we could form N/r partial products.
  - Faster and smaller?
  - Called radix-2^r encoding
- Ex: r = 2: look at pairs of bits
  - Form partial products of 0, Y, 2Y, 3Y
  - First three are easy, but 3Y requires adder ☹

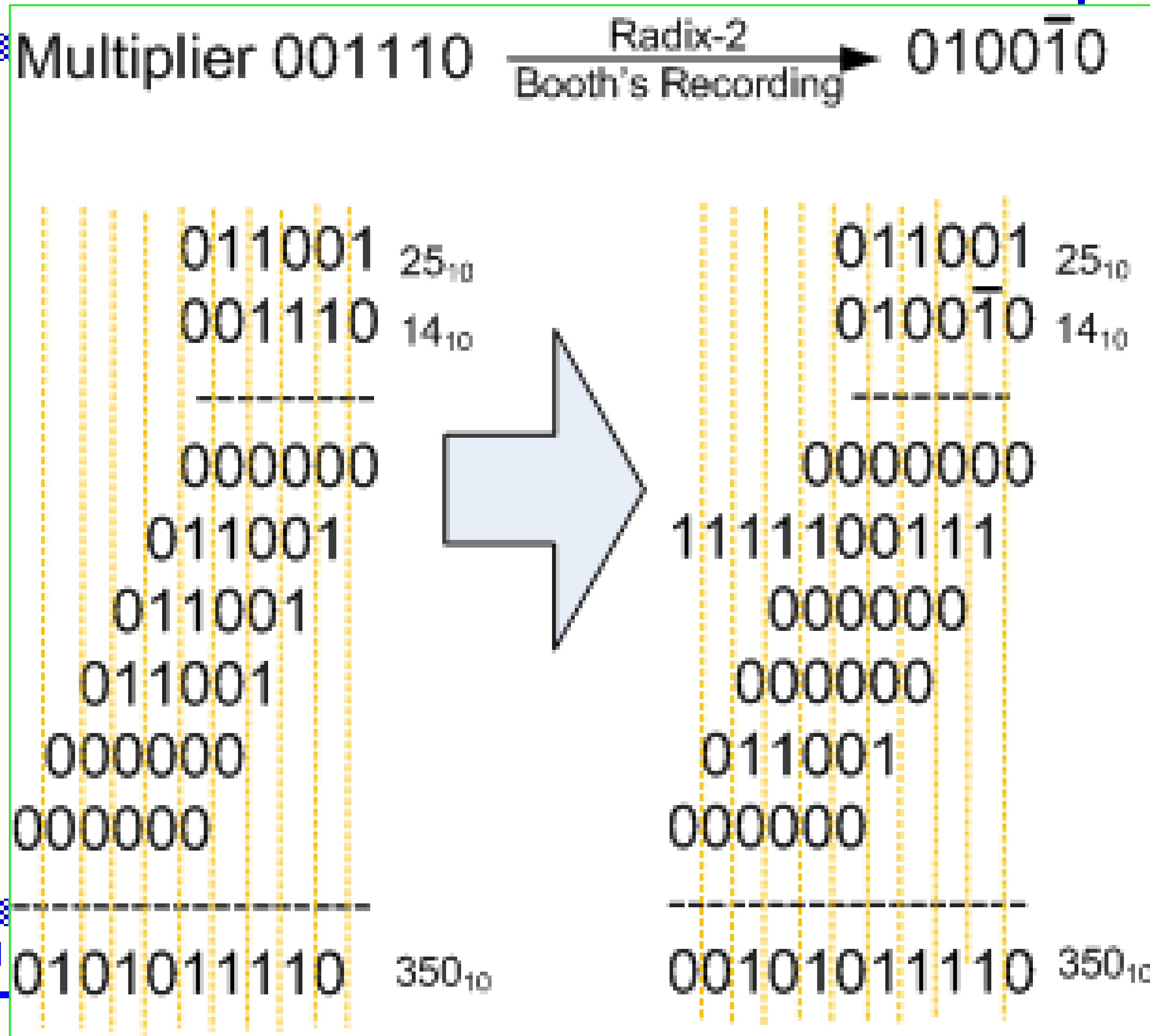
Radix-2 Booth's Recording

- ❑ In old (**serial**)multipliers, addition operations were very slow.
- ❑ Booth attempted to reduce the addition operations, because addition is slow.
  - – He observed that when there are a large number of consecutive 1's in multiplier, the multiplication can be speeded up.
  - – $2^j + 2^{j-1} + \ldots + 2^i = 2^{j+1} - 2^i$
  - – Example:
    - • 01110 = 10000 - 000010 = $2^6 - 2^1$
    - • 14 = 16 - 2
  - – The larger the sequence of 1's, the larger the savings
- ❑ Booth Recording converts the binary number represented by the set [0,1] to binary signed-digit number using the digit set [-1, 1, **0**].
- ❑ Radix-2 booth encoding examines x_i and x_{i-1} to generate the encoding y_i.

Xi	**Xi-1**	**Yi**	**Explanation**
0	0	0	No string of one's insight
0	1	1	End of string of ones
1	0	-1	Beginning of string of ones
1	1	0	Continuation of string of ones

Radix-2 Booth's Recording Example

- **Regular multiplication: 3 additions**
- **Booth: one addition and one subtraction**



11.9.3 Booth Encoding: Radix-4

- Radix-4 multiplier produces N/2 partial products.
- Each partial product is 0, Y, 2Y, or 3Y, depending on a pair of bits of X.
  - Computing 2Y is a simple shift,
  - but 3Y is a hard multiple requiring a slow carry propagate addition of Y + 2Y before partial product generation begins.
- Booth encoding was originally proposed to accelerate **serial multiplication**.
- Modified Booth encoding allows higher radix parallel operation without generating the hard 3Y multiple by instead using negative partial products.
- Observe that
  - 3Y = 4Y – Y
  - 2Y = 4Y – 2Y
  - 4Y in a radix-4 multiplier array is equivalent to Y in the next row of the array that carries four times the weight.

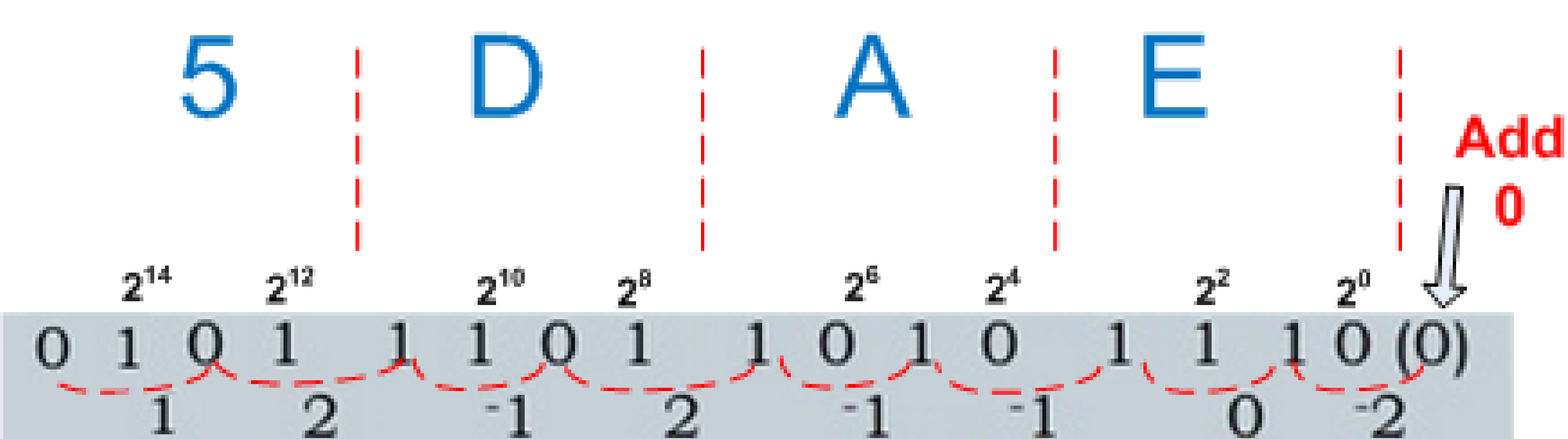
Radix-4 Modified Booth

Inputs			Partial Product	
x_{2i+1}	x_{2i}	x_{2i-1}	PP_i	**Explanation**
0	0	0	0	No string of 1's
0	0	1	Y	End of string of 1's
0	1	0	Y	Isolated 1
0	1	1	$2Y$	End of string of 1's
1	0	0	$-2Y$	Beginning of string of 1's
1	0	1	$-Y$	End string, begin new string
1	1	0	$-Y$	Beginning of string of 1's
1	1	1	$-0\ (=0)$	Continuation of string of 1's

Radix-4 Booth Encoding Example

- ❑ Compute the Booth's Encoding for the 16-bit unsigned number: 5DAE

Inputs			Partial Product
x_{2i+1}	x_{2i}	x_{2i-1}	PP_i
0	0	0	0
0	0	1	Y
0	1	0	Y
0	1	1	$2Y$
1	0	0	$-2Y$
1	0	1	$-Y$
1	1	0	$-Y$
1	1	1	$-0\ (=0)$



$=1\times2^{14} + 2\times2^{12} + -1\times2^{10} + 2\times2^{8} + -1\times2^{6} + -1\times2^{4} + 0\times2^{2} + -2\times2^{0}$

$=23982$

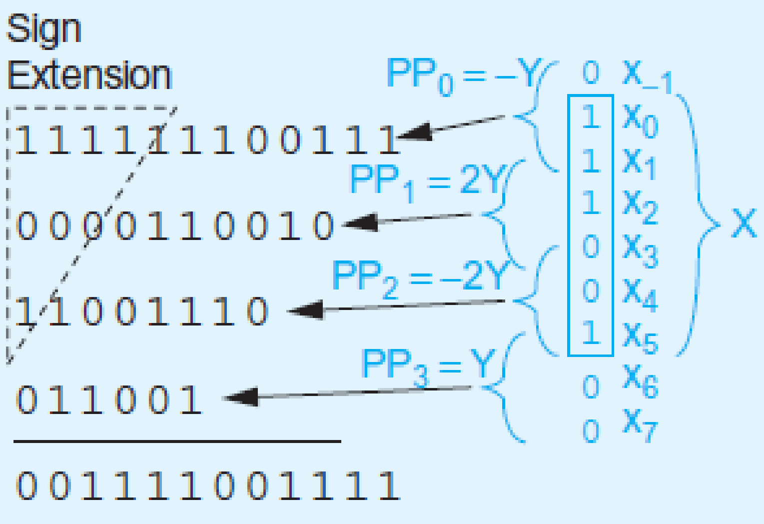
Radix-4 Booth Encoding Example

Example 11.3

Repeat the multiplication of $P = Y \times X = 011001_2 \times 100111_2$ from Figure 11.71, applying Booth encoding to reduce the number of partial products.

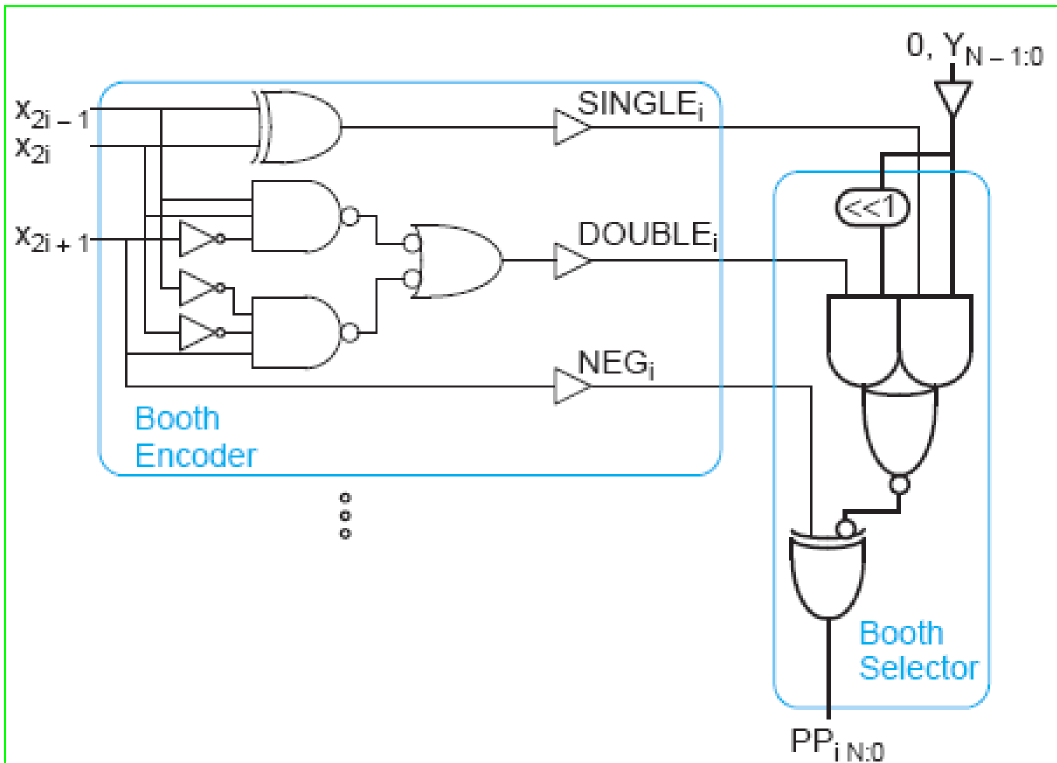
```
      011001 : 25₁₀    multiplicand
    × 100111 : 39₁₀    multiplier
      011001
     011001
    011001             partial
   000000              products
  000000
+011001
001111001111 : 975₁₀   product
```

SOLUTION: Figure 11.79 shows the multiplication. X is written vertically and the bits are used to select the four partial products. Each partial product is shifted two columns left of the previous one because it has four times the weight. The upper bits are sign-extended with 1s for negative partial products and 0s for positive partial products. The partial products are added to obtain the result.



Booth Hardware

- Booth encoder generates control lines for each PP
  - Booth selectors choose PP bits



Inputs			Partial Product	Booth Selects		
x_{2i+1}	x_{2i}	x_{2i-1}	PP_i	$SINGLE_i$	$DOUBLE_i$	NEG_i
0	0	0	0	0	0	0
0	0	1	Y	1	0	0
0	1	0	Y	1	0	0
0	1	1	$2Y$	0	1	0
1	0	0	$-2Y$	0	1	1
1	0	1	$-Y$	1	0	1
1	1	0	$-Y$	1	0	1
1	1	1	-0 (= 0)	0	0	1