Digital Logic Circuits
'Common Combinational Circuits'
ELEC2200
Summer 2009

David J. Broderick
brodedj@auburn.edu
http://www.auburn.edu/~brodedj
Office: Broun 360

Modular Design

- Common combinational circuits are often lumped together into components
- A collection of gates is described by a block with inputs and outputs
- The behavior of the block is defined to meet a common purpose
- Used to make common operations easier/faster to implement

Adders

- We can use a combinational circuit to add binary values
- Recall:

Carries-->

1	1	1	0	0
	1	0	1	0
	1	1	1	1
	1	0	0	1

<--Result

Adders

- Each column can be thought of as a group of circuits
- Look at inputs and outputs

Carries-->

1	1	1	0	0
	1	0	1	0
	1	1	1	1
	1	0	0	1

<--Result

Adders

- Each column shares these set of inputs
- Look at inputs and outputs

Carries-->

1	1	1	0	0
	1	0	1	0
	1	1	1	1
	1	0	0	1

<--Result

Adders

- Each column shares these set of inputs
- Look at inputs and outputs

Carries-->

1	1	1	0	0
	1	0	1	0
	1	1	1	1
	1	0	0	1

<--Result

Adders

- Each column shares these set of inputs
- Look at inputs and outputs

Carries-->

1	1	1	0	0
	1	0	1	0
	1	1	1	1
	1	0	0	1

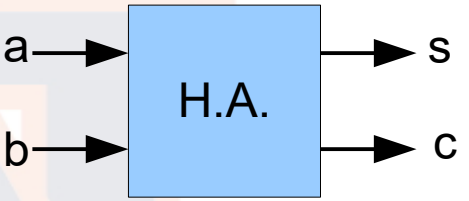
<--Result

Half-Adder

- Ignoring carry-in:

```
 c
     a
+    b
-----
     s
```

a	b	s	c
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



- s=a(+)b
- c=a·b

Full-Adder

- With carry-in:

```
 co  ci
       a
 +     b
-------
       s
```

a	b	ci	s	co
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

s

a\b ci	00	01	11	10
0	0	1	0	1
1	1	0	1	0

co

a\b ci	00	01	11	10
0	0	0	1	0
1	0	1	1	1

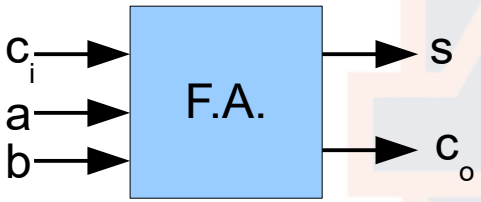
- s=a(+)b
- c=a·b

Full-Adder

- $s=a\cdot\bar{b}\cdot\bar{c_i}+\bar{a}\cdot\bar{b}\cdot c_i+a\cdot b\cdot c_i+\bar{a}\cdot b\cdot\bar{c_i}$

  $s=a(+)b(+)c_i$

- $c_o=a\cdot c_i+b\cdot c_i+a\cdot b$



s

a\b ci	00	01	11	10
0	0	1	0	1
1	1	0	1	0

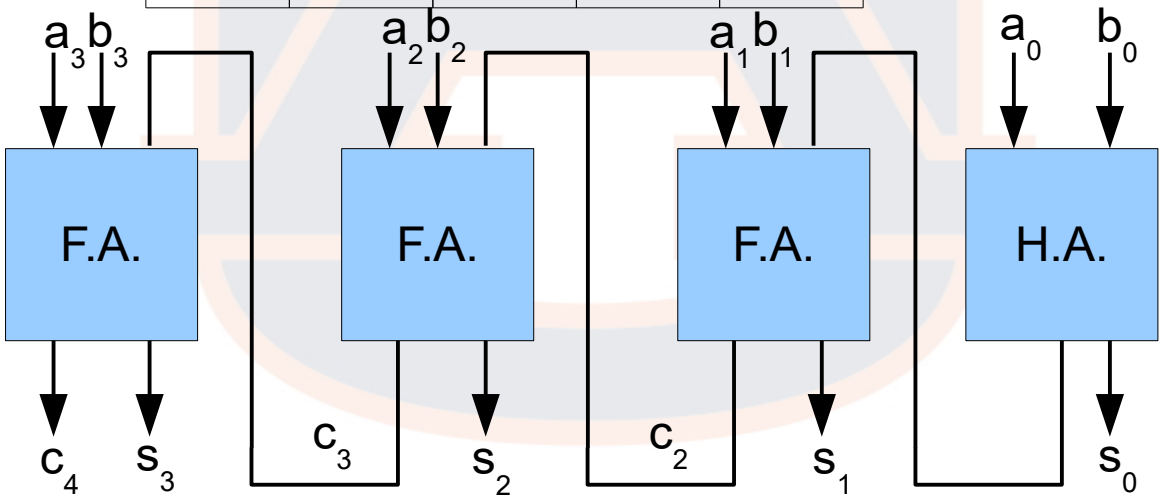
CO

a\b ci	00	01	11	10
0	0	0	1	0
1	0	1	1	1

Ripple-Carry Adder

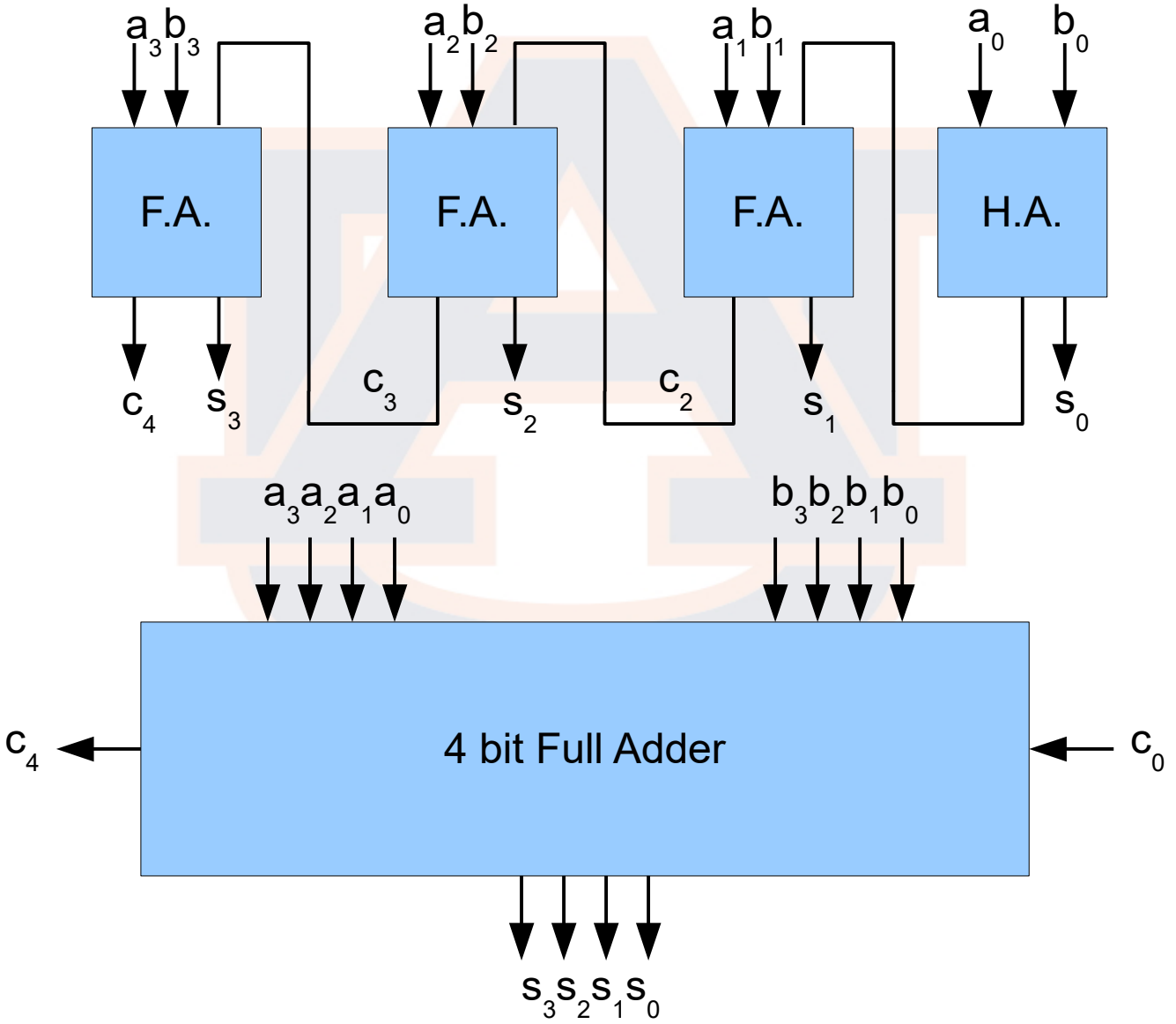
- A slow (but small) method of adding larger values:

c3	c2	c1	c0	
	a3	a2	a1	a0
+	b3	b2	b1	b0
	s3	s2	s1	s0



Ripple-Carry Adder

- Often redrawn as:

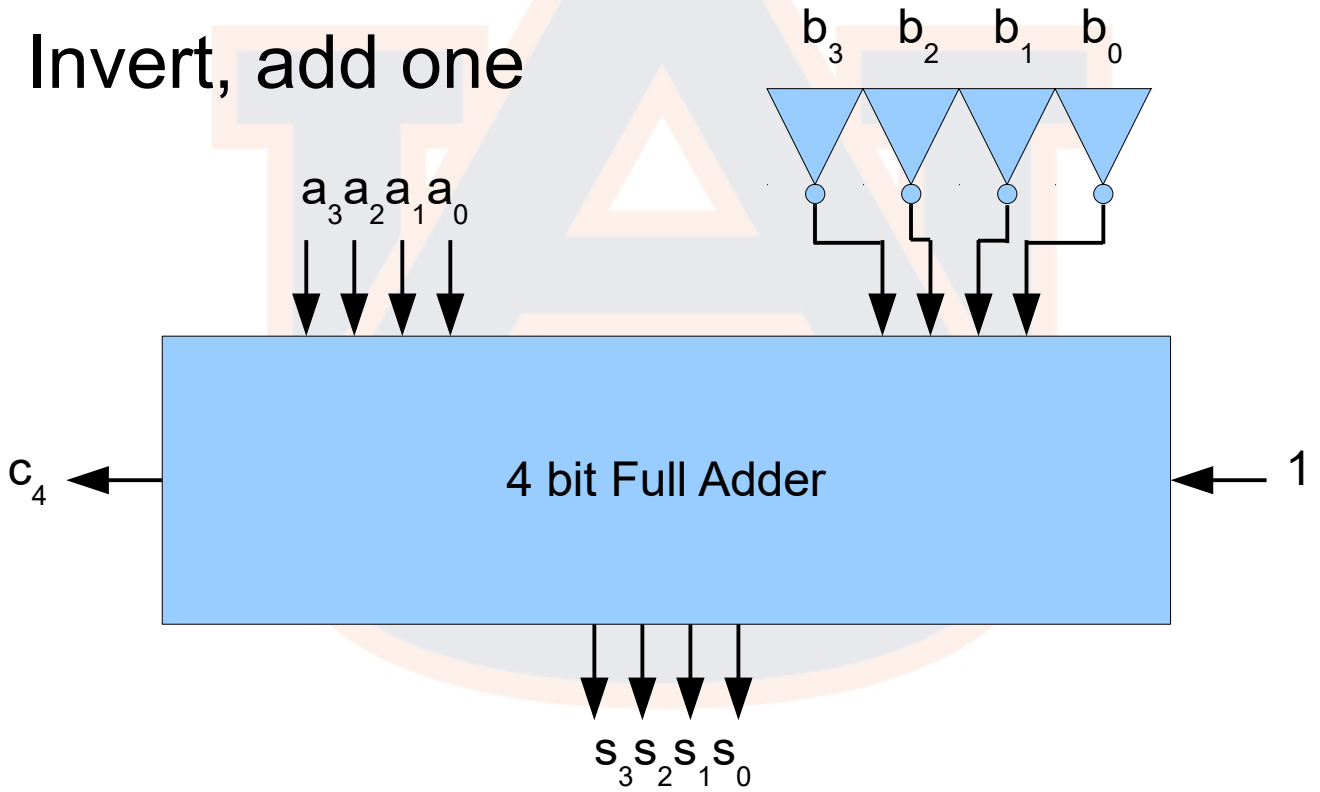


Fully Parallel Adders

- If all inputs are known (9 inputs) expressions can be written for all outputs (5 outputs)
- Creates a faster adder but does not scale well
    - 9 input K-map!
- Intermediate solutions exist
    - Carry look-ahead adders
    - Carry-save adders
    - etc...

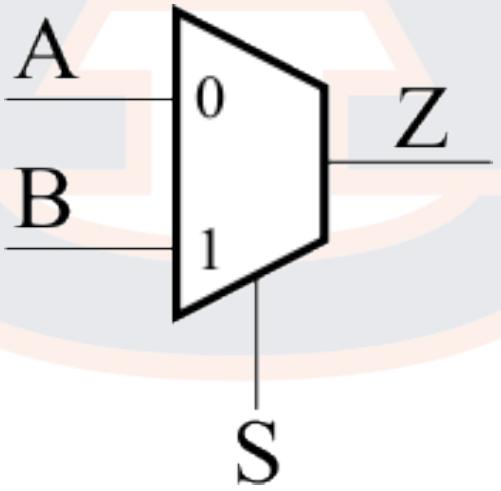
Subtractors

- Using 2's complement, subtraction looks like addition
  - Invert, add one



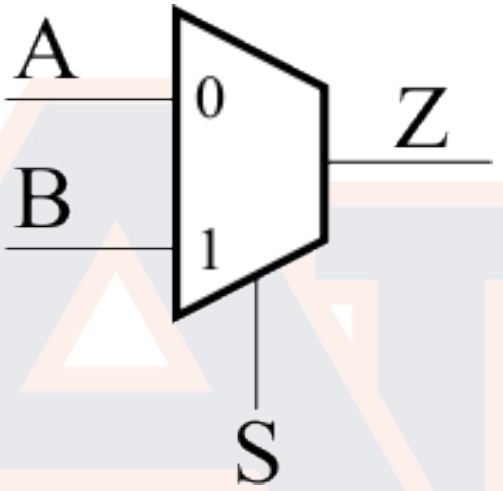
Multiplexers

- Selects between multiple inputs to pass through to output
- n control signals are used to select up to 2^n inputs to pass to the single output
- A 2-to-1 MUX:



Multiplexers

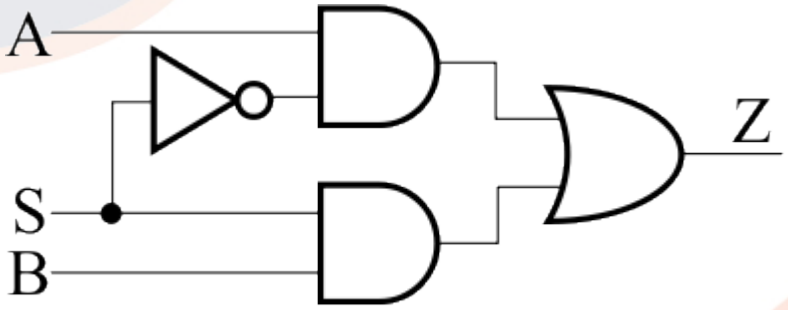
- If S=0, Z=A
- If S=1, Z=B
- 3 inputs, 1 output



A	B	S	Z
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

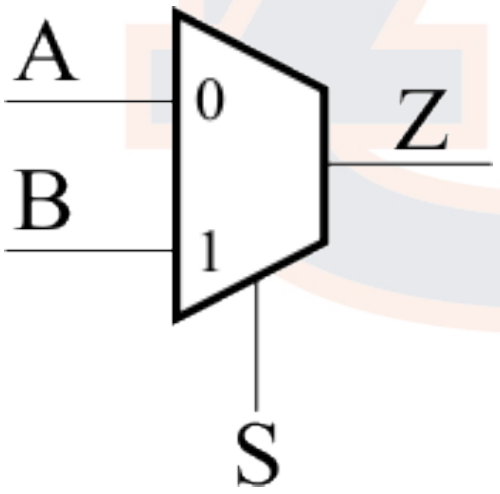
AB\S	0	1
00	0	0
01	0	1
11	1	1
10	1	0

$$Z=A\cdot\overline{S}+B\cdot S$$



Multiplexers

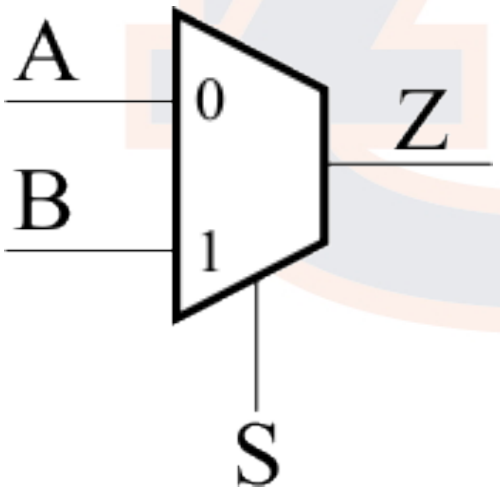
- The multiplexer is a functionally complete set
- Need to show AND, OR, and NOT
- AND: A=0, Z=B·S



A	B	S	Z
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

Multiplexers

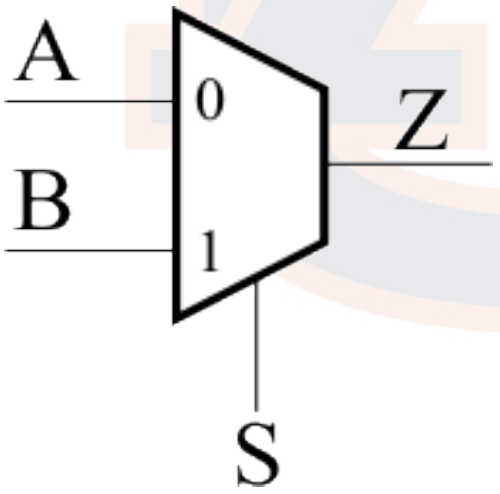
- The multiplexer is a functionally complete set
- Need to show AND, OR, and NOT
- OR: B=1, Z=A+S



A	B	S	Z
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

Multiplexers

- The multiplexer is a functionally complete set
- Need to show AND, OR, and NOT
- NOT: A=1, B=0, Z=$\overline{S}$



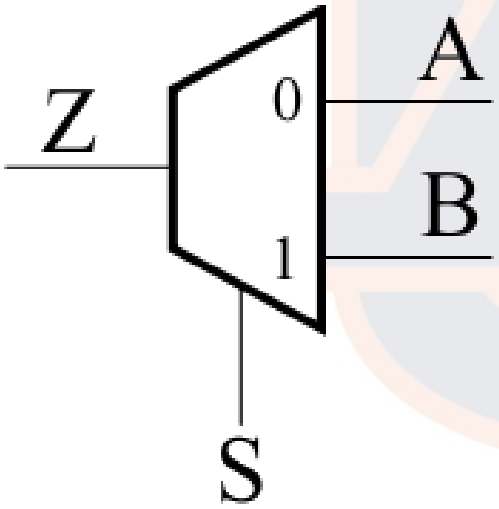
A	B	S	Z
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

Adder/Subtractor

Adder/Subtractor

Demultiplexers

- Opposite of a multiplexer
- Input passed to selected output, all other outputs are zero



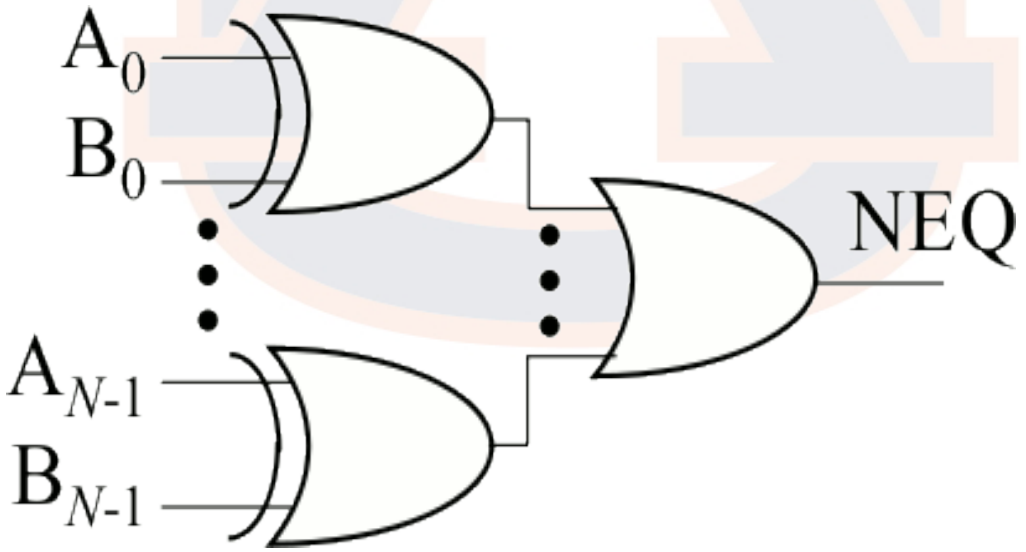
S	Z	A	B
0	0	0	0
0	1	1	0
1	0	0	0
1	1	0	1

Decoders/Encoders

- Decoders
    - N inputs produce M outputs (typically M>N)
    - BCD to 7-segment decoder is a good example
- Encoder
    - N inputs produce M outputs (typically N>M)
    - We could just as easily have taken the 7-segment values, (A-G) and encode to BCD or HEX
- Convertors
    - N inputs produce M outputs (typically N=M)
    - Binary to Gray-Code convertor is a good example

Comparators

- Equal-to comparators us XOR function
    - XOR produces 1 when inputs differ
    - Do bit-wise compare of N bit number
- Outputs of XORs then ORed together.
- If any bit differs, output =1

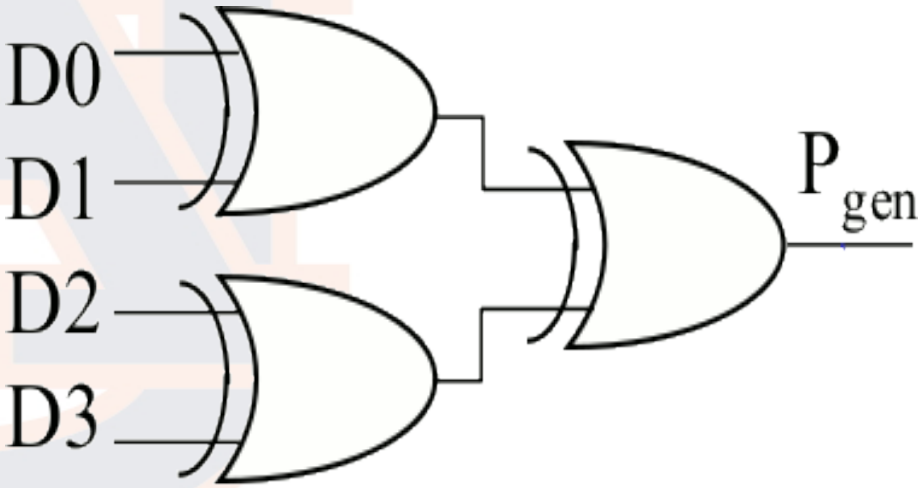


Comparators

- Greater than/Less than performed with 2's comp subtraction
- Examine sign (MSB) of result to determine result

Parity Circuits

- Parity makes use of the XOR function
  - XOR produces a 1 for an odd # of 1s
  - Implements even parity
  - An XNOR on the output can be used for odd parity



Parity Circuits

- Check for correct parity by comparing P_{gen} with incoming parity bit P_{in}
- One more XOR gate for comparison