CSEE 3827: Fundamentals of Computer Systems, Spring 2011

3. Combinational Circuit Design

Prof. Martha Kim (martha@cs.columbia.edu)
Web: http://www.cs.columbia.edu/~martha/courses/3827/sp11/

Outline (H&H 2.8, 5.2)

- Standard combinational circuits
  - Decoder
  - Encoder / priority encoder (bonus, not in text)
  - Code converter (bonus, not in text)
  - Multiplexer
- Addition
  - Half and full adders
  - Ripple carry adder
  - Carry lookahead adder
- Subtraction
- Comparator (“!=” in lecture, “<“ in text)
- ALU (in text, not covering yet)
- Shifter

Combinational circuits

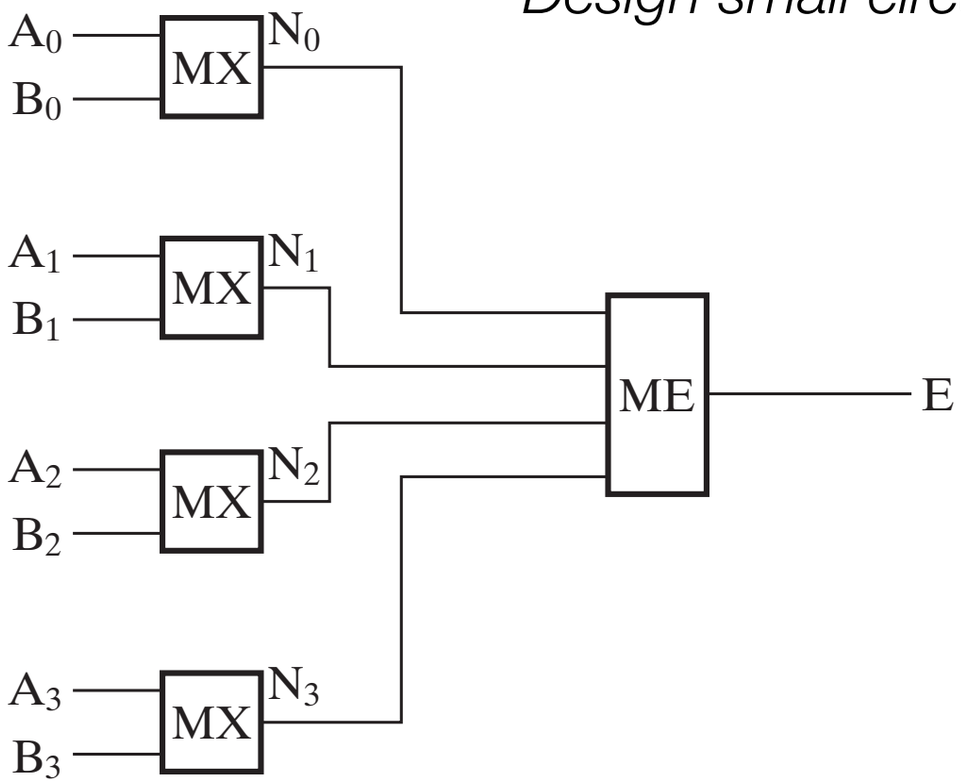
- Combinational circuits are stateless

- The outputs are functions only of the inputs

Hierarchical design

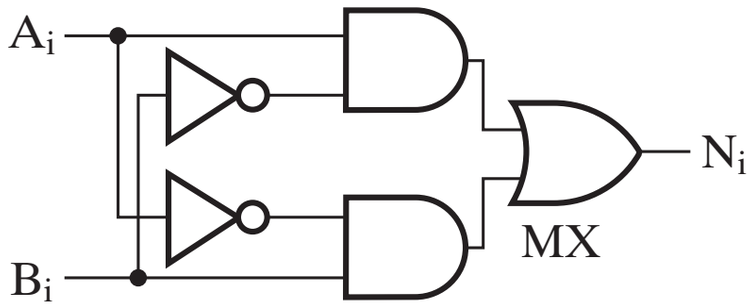
Design small circuits to be used in a bigger circuit

"Big"Circuit

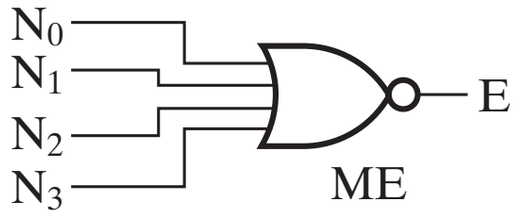


(a)

Smaller Circuits



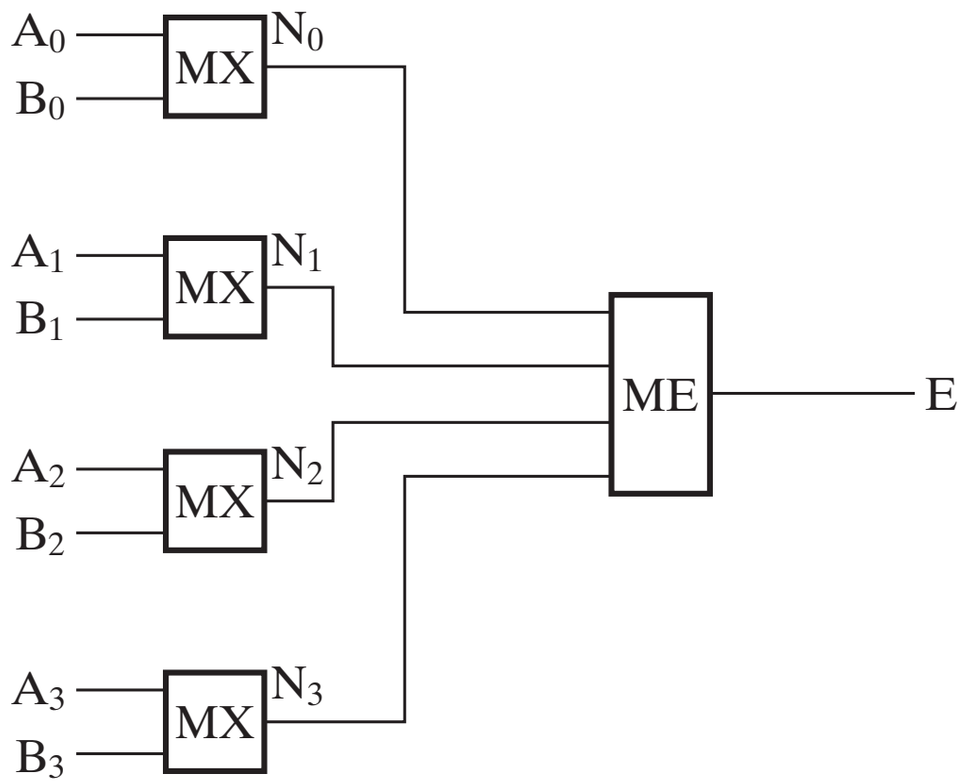
(b)



(c)

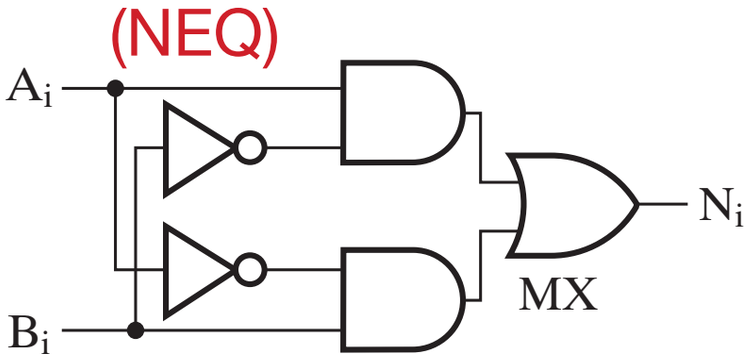
Hierarchical design (It's a comparator!)

(4-bit equality comparator)

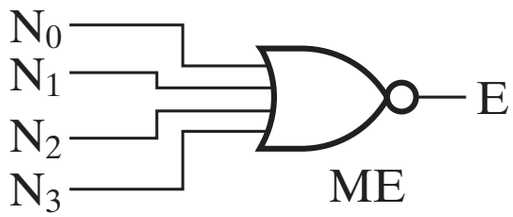


(a)

(NEQ)



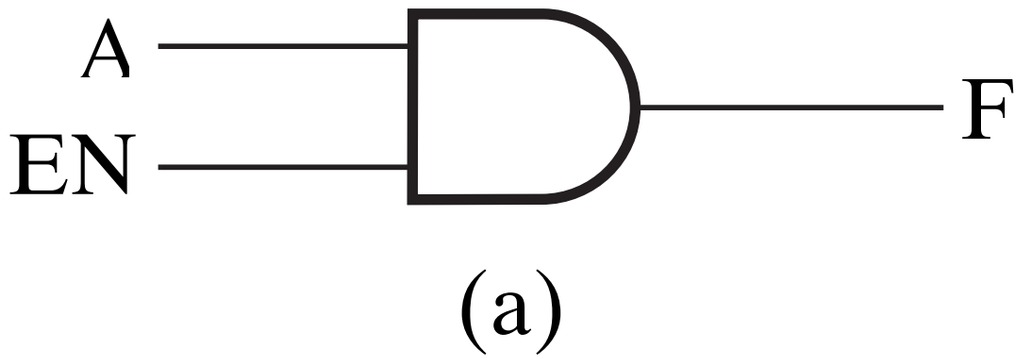
(b)



(c)

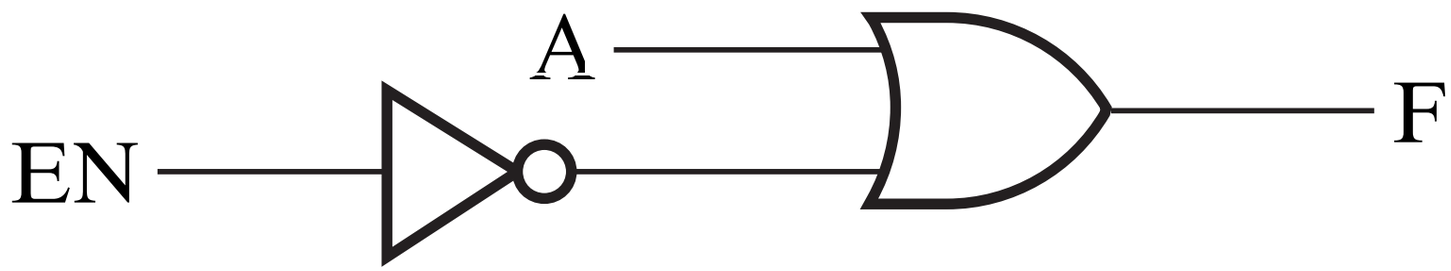
Enabler circuits

Output is "enabled" (F=A) only when input 'ENABLE' signal is asserted (EN=1)



(a)

EN	F
0	0
1	A

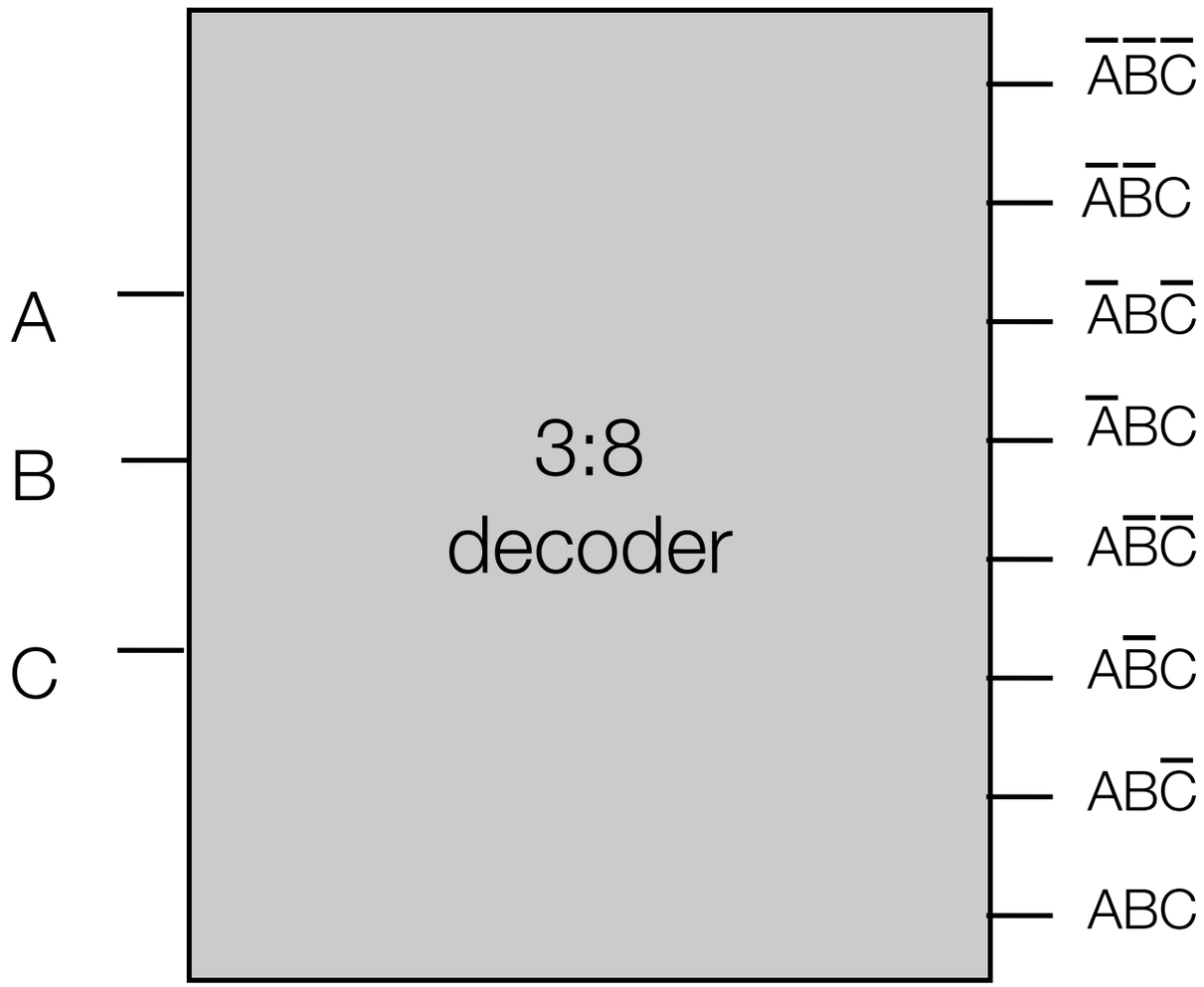


(b)

EN	F
0	1
1	A

Decoder-based circuits

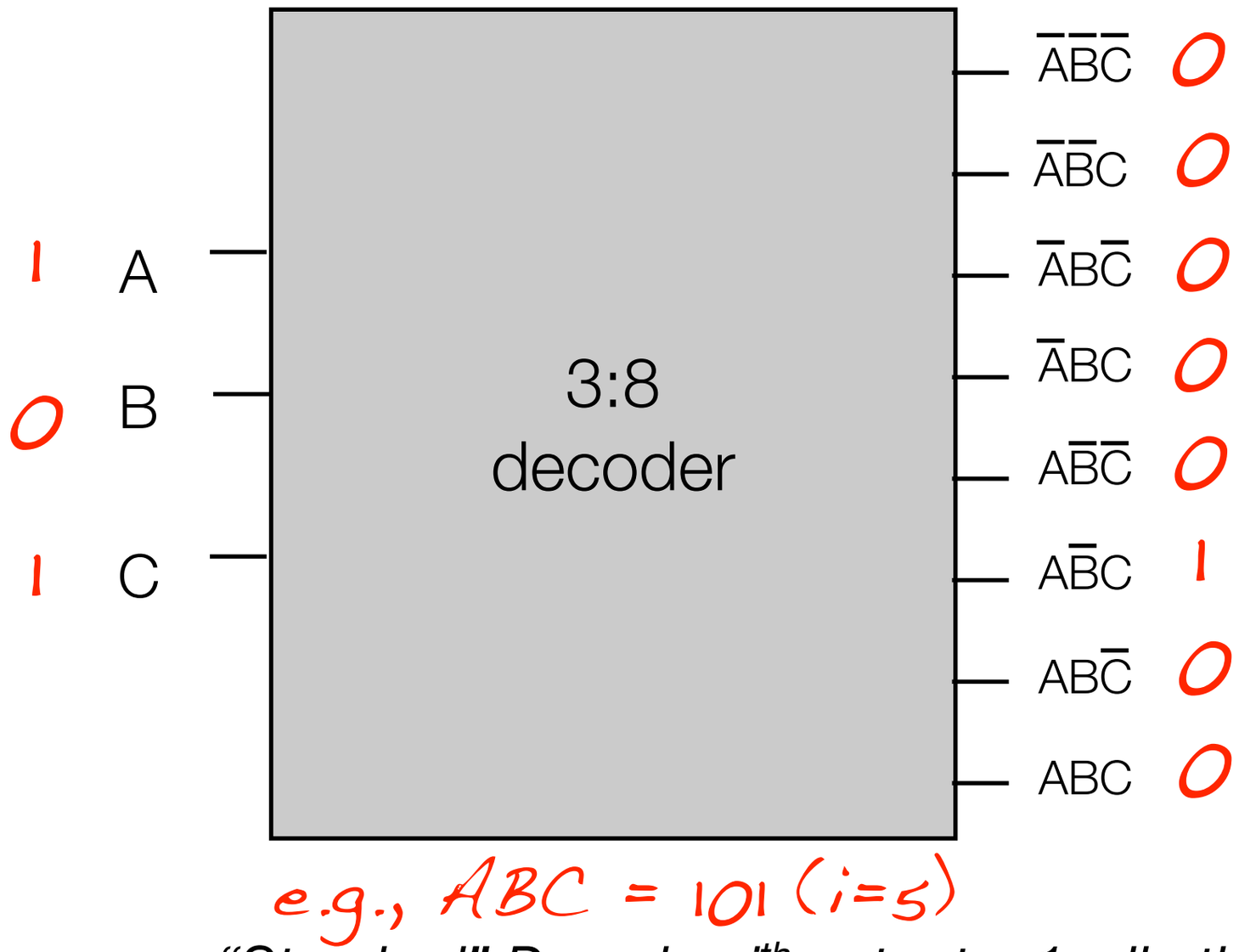
Converts n-bit input to m-bit output, where n <= m <= 2^n



"Standard" Decoder: i^{th} output = 1, all others = 0,
where i is the binary representation of the input (ABC)

Decoder-based circuits

Converts n-bit input to m-bit output, where n <= m <= 2^n



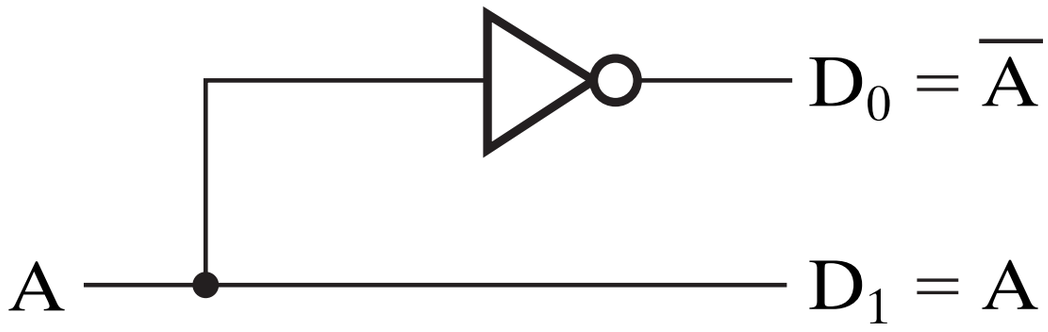
"Standard" Decoder: i^{th} output = 1, all others = 0,

where i is the binary representation of the input (ABC)

Internal design of 1:2 decoder

A	D_0	D_1
0	1	0
1	0	1

(a)

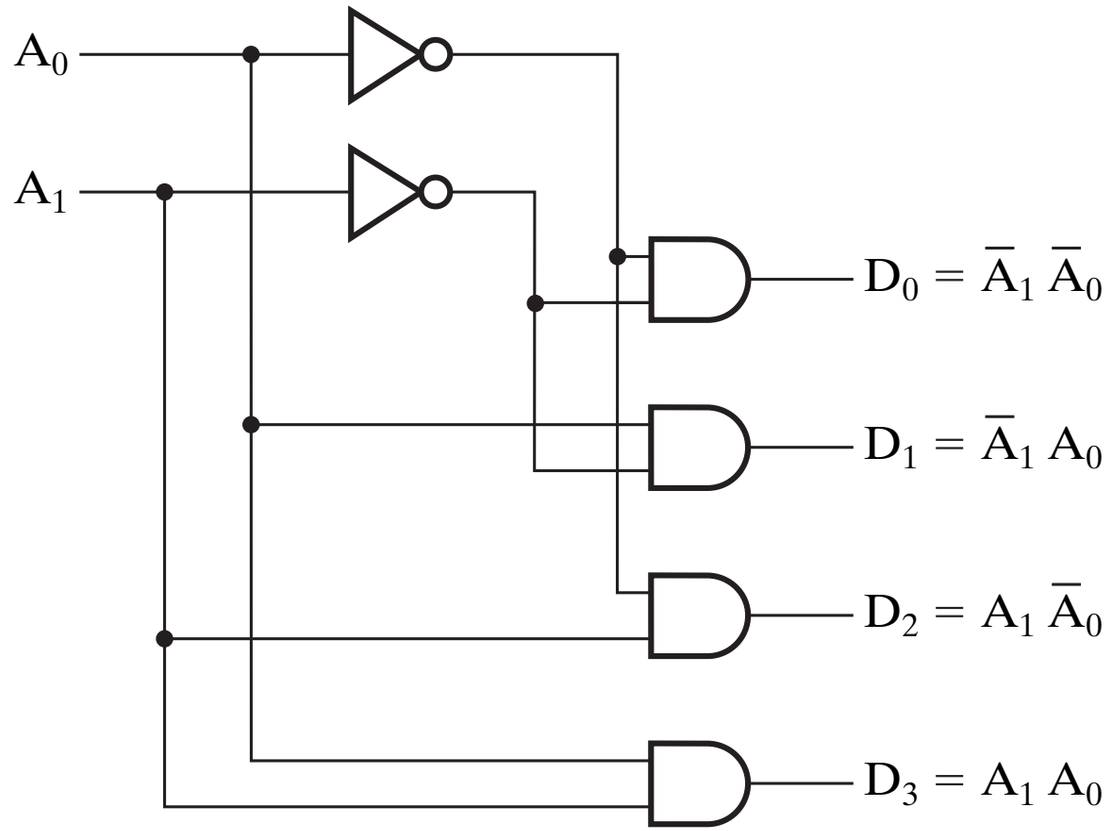


(b)

Internal design of 2:4 decoder

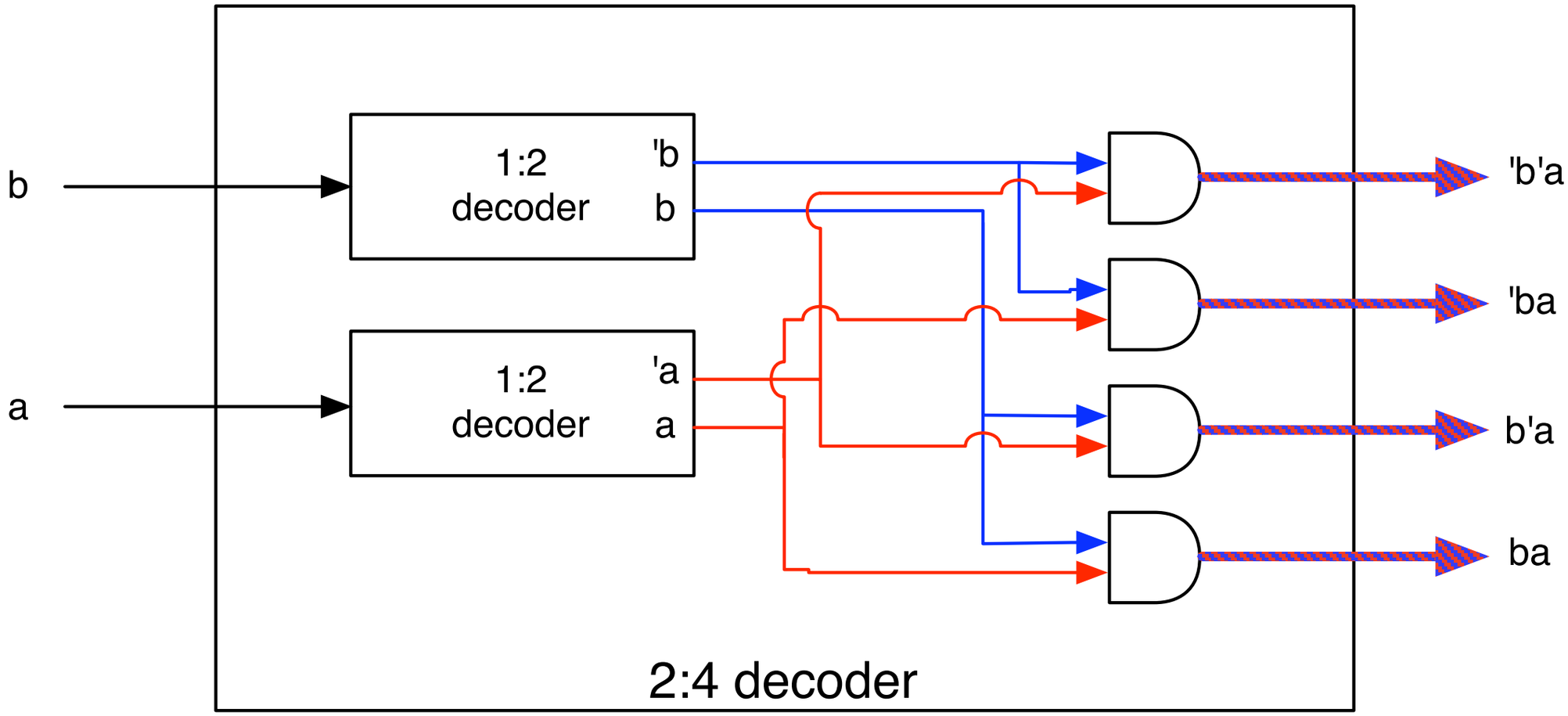
A_1	A_0	D_0	D_1	D_2	D_3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

(a)



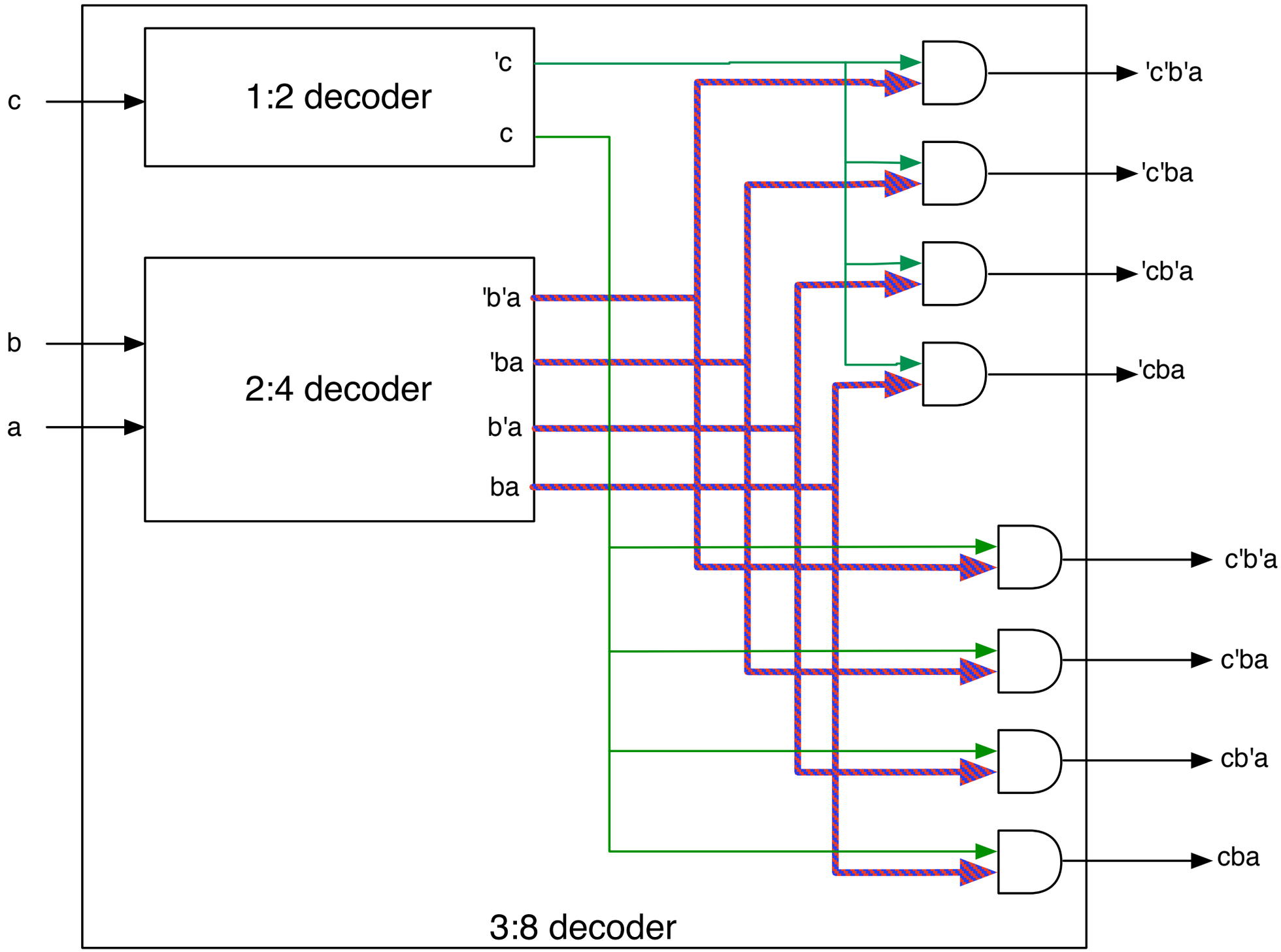
(b)

Hierarchical design of 2:4 decoder



Can build 2:4 decoder out of two 1:2 decoders
(and some additional circuitry)

Hierarchical design of 3:8 decoder



Encoders

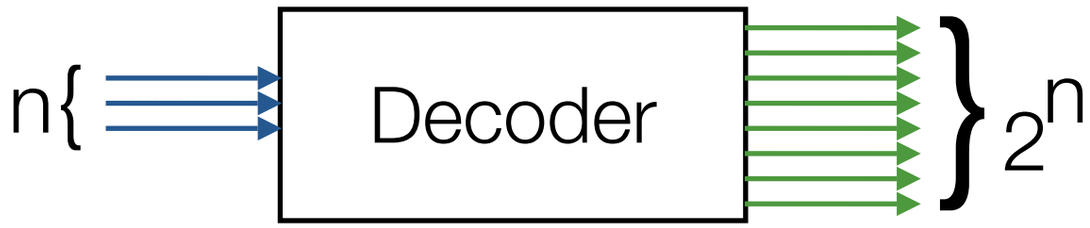
T 3-7

Inverse of a decoder: converts m-bit input to n-bit output, where n <= m <= 2^n

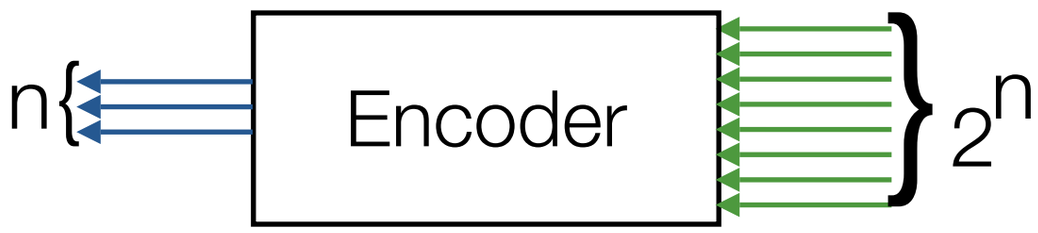
TABLE 3-7
Truth Table for Octal-to-Binary Encoder

Inputs								Outputs		
D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	A_2	A_1	A_0
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1

Decoder and encoder summary



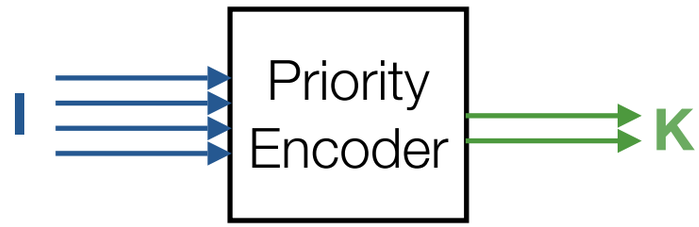
BCD values			One-hot encoding							
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0



Note: for Encoders - input is assumed to have just one 1, the rest 0's

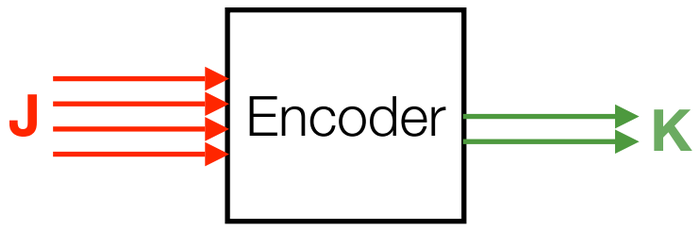
In class design: priority encoder

A priority encoder takes **2^n bit input (I)** and produces **n bits of output (K)** indicating in BCD the position of the most significant 1 on the input.



I3	I2	I1	I0	K1	K0
0	0	0	1	0	0
0	0	1	x	0	1
0	1	x	x	1	0
1	x	x	x	1	1

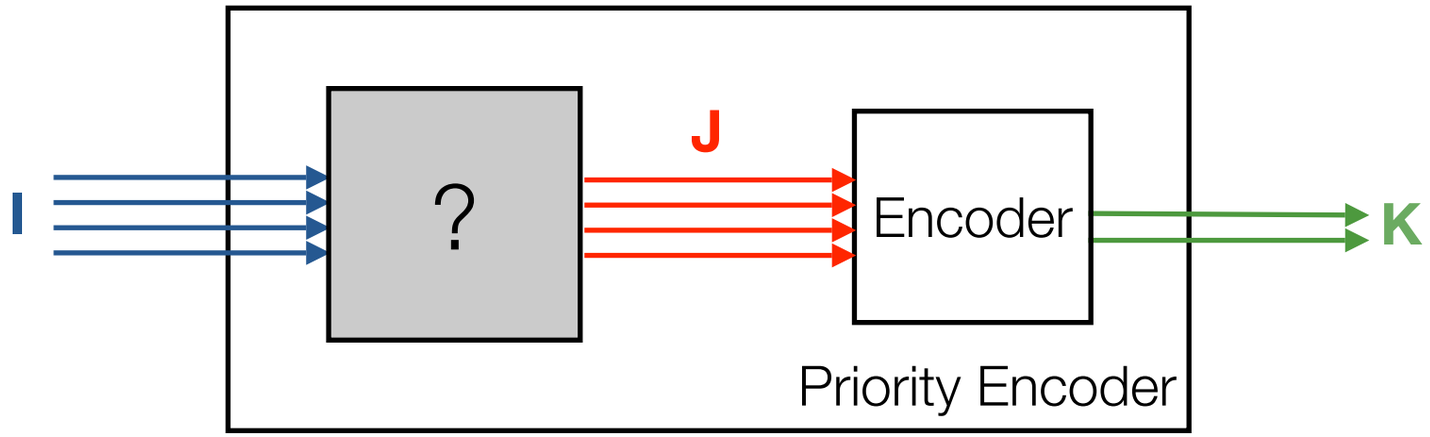
We will leverage a regular encoder which takes **2^n bit one-hot encoded input (J)** and produces **n bits of output (K)** indicating in BCD the position of the 1 on the input.



J3	J2	J1	J0	K1	K0
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

In class design: priority encoder (2)

This gets us part of the way there, leaving us with a simpler problem of translating **I** into **J:**



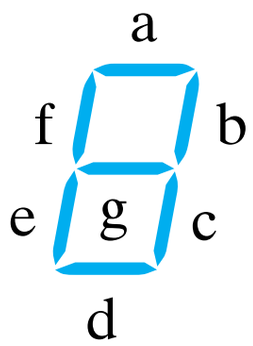
I3	I2	I1	I0	J3	J2	J1	J0
0	0	0	1	0	0	0	1
0	0	1	x	0	0	1	0
0	1	x	x	0	1	0	0
1	x	x	x	1	0	0	0

NB: An input i = x is still a don't care, it means "for all possible values of i". So here input 1xxx means any 4-bit input starting with a 1, i.e., 1000, 1001, 1010, 1011, 1100, ...

From inspection of the truth table we can see the following definitions of Jx. Could also have used k-maps.

J3 = I3
J2 = I2`I3
J1 = I1`I2`I3
J0 = I0`I1`I2`I3

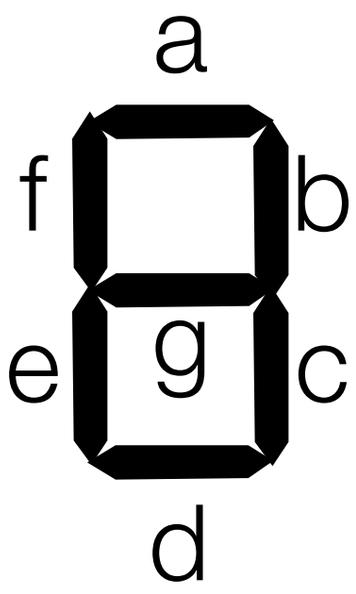
General code conversion



(a) Segment designation

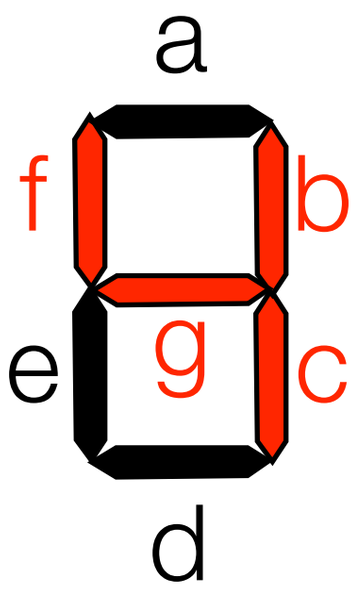
(b) Numeric designation for display

Code conversion



Input					Output						
Va	W	X	Y	Z	a	b	c	d	e	f	g
0	0	0	0	0	1	1	1	1	1	1	0
1	0	0	0	1	0	1	1	0	0	0	0
2	0	0	1	0	1	1	0	1	1	0	1
3	0	0	1	1	1	1	1	1	0	0	1
4	0	1	0	0	0	1	1	0	0	1	1
5	0	1	0	1	1	0	1	1	0	1	1
6	0	1	1	0	1	0	1	1	1	1	1
7	0	1	1	1	1	1	1	0	0	0	0
8	1	0	0	0	1	1	1	1	1	1	1
9	1	0	0	1	1	1	1	0	0	1	1
X	1	0	1	0	X	X	X	X	X	X	X
X	1	0	1	1	X	X	X	X	X	X	X
X	1	1	0	0	X	X	X	X	X	X	X
X	1	1	0	1	X	X	X	X	X	X	X
X	1	1	1	0	X	X	X	X	X	X	X
X	1	1	1	1	X	X	X	X	X	X	X

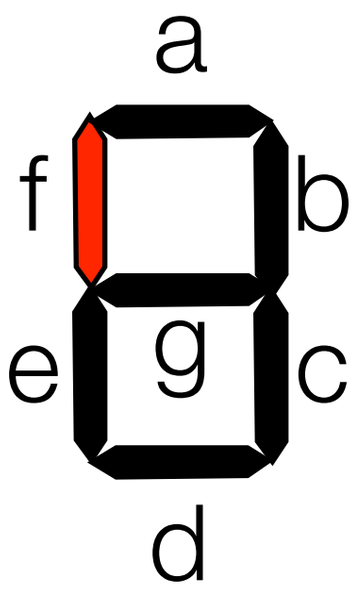
Code conversion



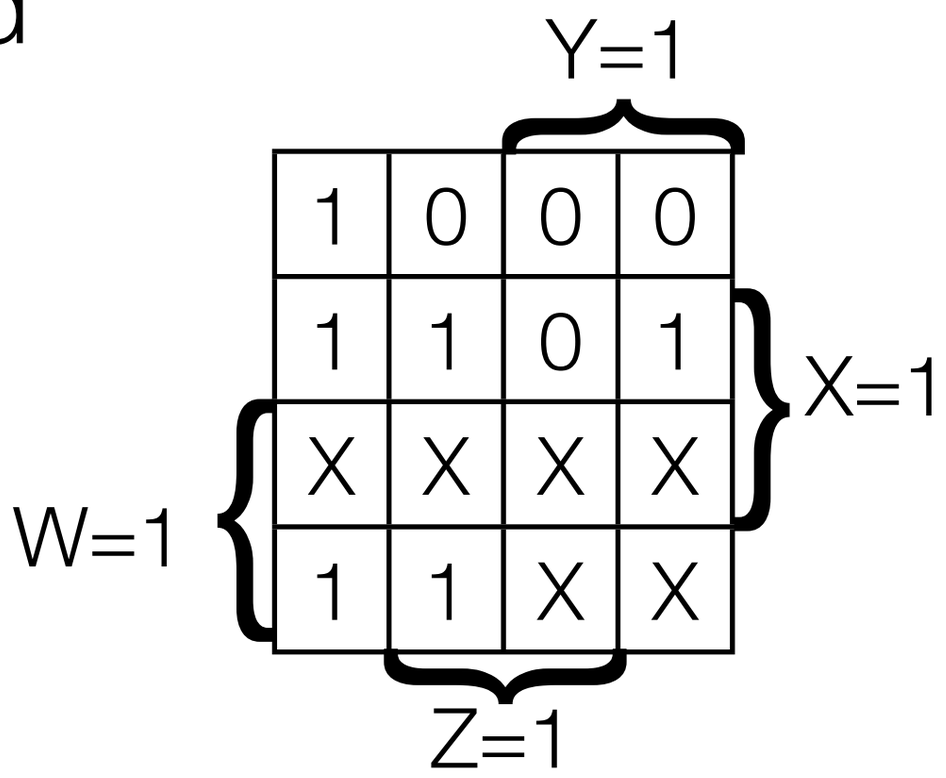
e.g., what outputs "lights up" when input V=4?

	Input				Output						
Va	W	X	Y	Z	a	b	c	d	e	f	g
0	0	0	0	0	1	1	1	1	1	1	0
1	0	0	0	1	0	1	1	0	0	0	0
2	0	0	1	0	1	1	0	1	1	0	1
3	0	0	1	1	1	1	1	1	0	0	1
4	0	1	0	0	0	1	1	0	0	1	1
5	0	1	0	1	1	0	1	1	0	1	1
6	0	1	1	0	1	0	1	1	1	1	1
7	0	1	1	1	1	1	1	0	0	0	0
8	1	0	0	0	1	1	1	1	1	1	1
9	1	0	0	1	1	1	1	0	0	1	1
X	1	0	1	0	X	X	X	X	X	X	X
X	1	0	1	1	X	X	X	X	X	X	X
X	1	1	0	0	X	X	X	X	X	X	X
X	1	1	0	1	X	X	X	X	X	X	X
X	1	1	1	0	X	X	X	X	X	X	X
X	1	1	1	1	X	X	X	X	X	X	X

Code conversion

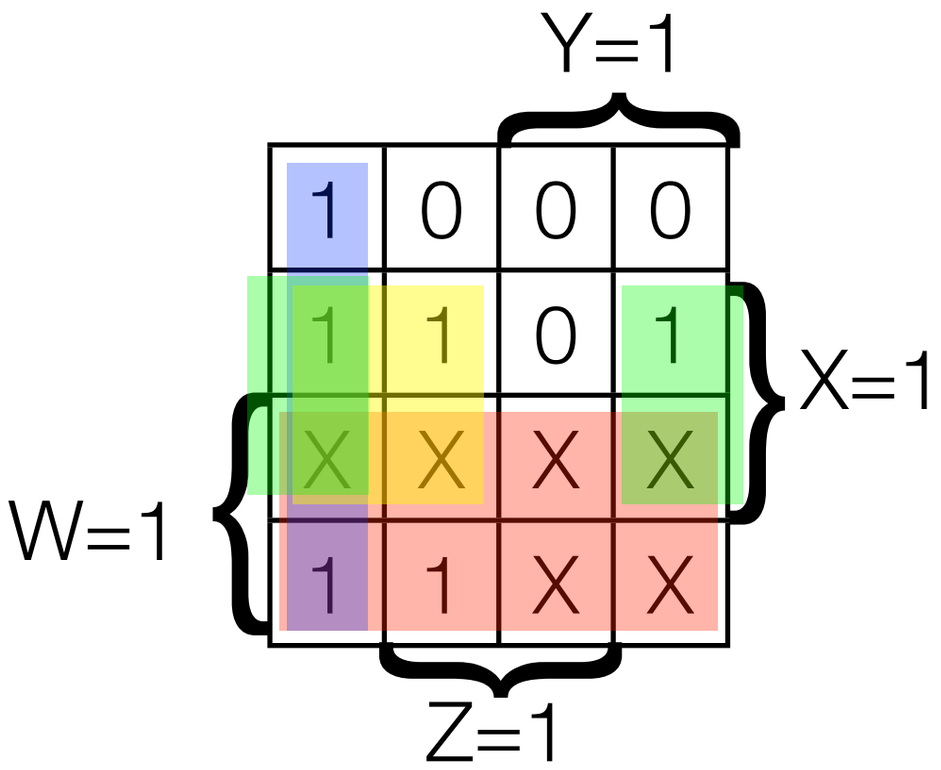


For what values does output f “light up” for?

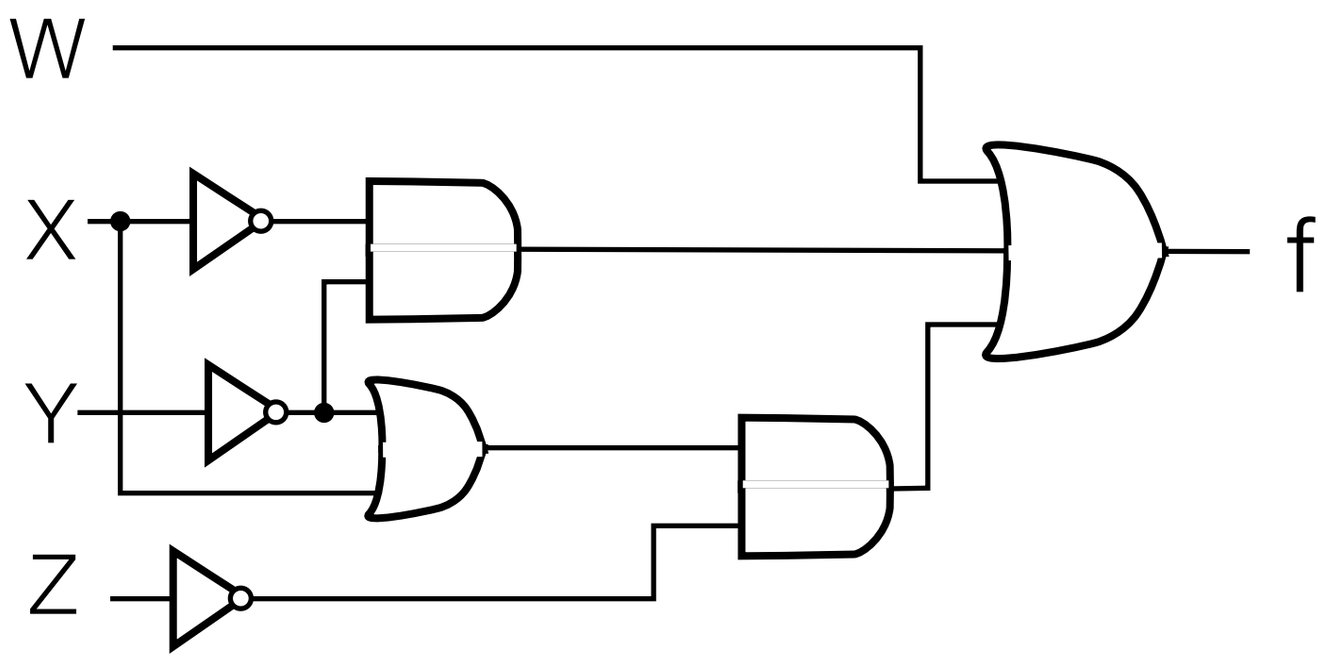


	Input				Output						
Va	W	X	Y	Z	a	b	c	d	e	f	g
0	0	0	0	0	1	1	1	1	1	1	0
1	0	0	0	1	0	1	1	0	0	0	0
2	0	0	1	0	1	1	0	1	1	0	1
3	0	0	1	1	1	1	1	1	0	0	1
4	0	1	0	0	0	1	1	0	0	1	1
5	0	1	0	1	1	0	1	1	0	1	1
6	0	1	1	0	1	0	1	1	1	1	1
7	0	1	1	1	1	1	1	0	0	0	0
8	1	0	0	0	1	1	1	1	1	1	1
9	1	0	0	1	1	1	1	0	0	1	1
X	1	0	1	0	X	X	X	X	X	X	X
X	1	0	1	1	X	X	X	X	X	X	X
X	1	1	0	0	X	X	X	X	X	X	X
X	1	1	0	1	X	X	X	X	X	X	X
X	1	1	1	0	X	X	X	X	X	X	X
X	1	1	1	1	X	X	X	X	X	X	X

Algebra and Circuit for “f”

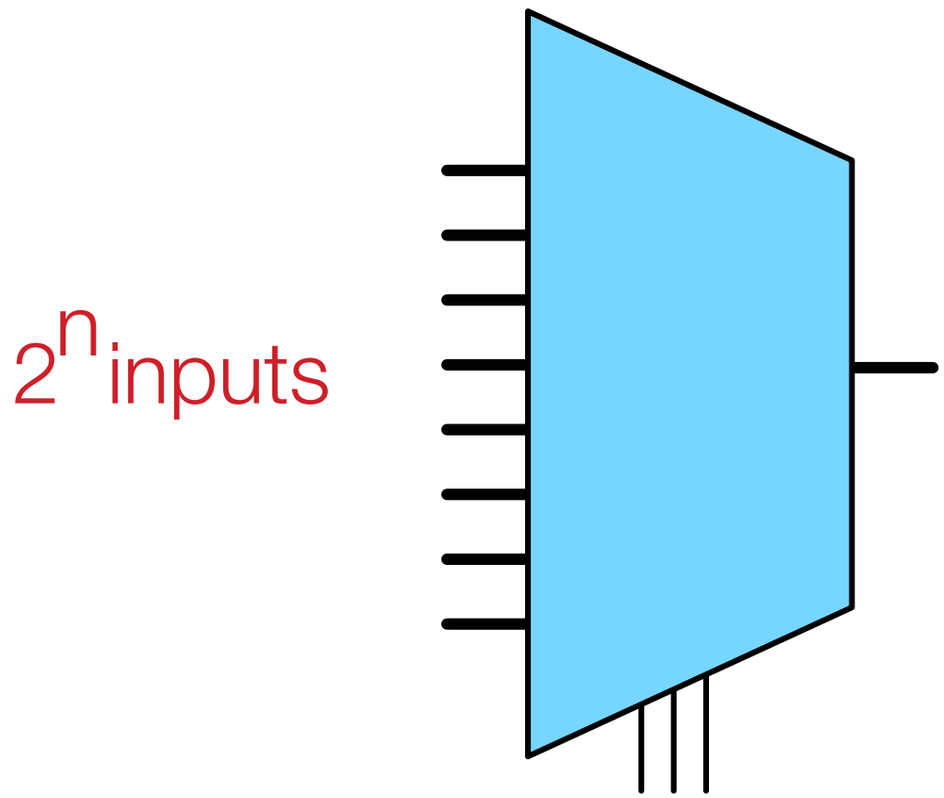


$$f = W + \overline{Y}\overline{Z} + X\overline{Z} + \overline{X}\overline{Y} = W + (X+\overline{Y})\overline{Z} + \overline{X}\overline{Y}$$



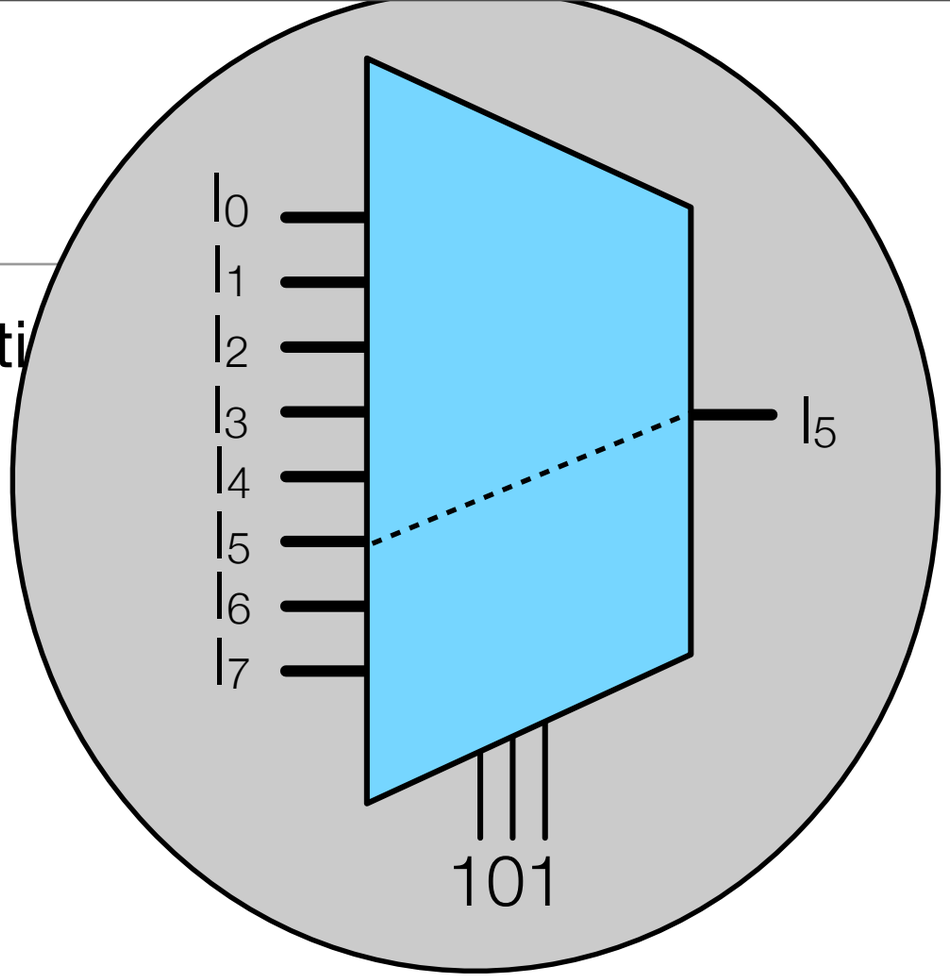
Multiplexers (or Muxes)

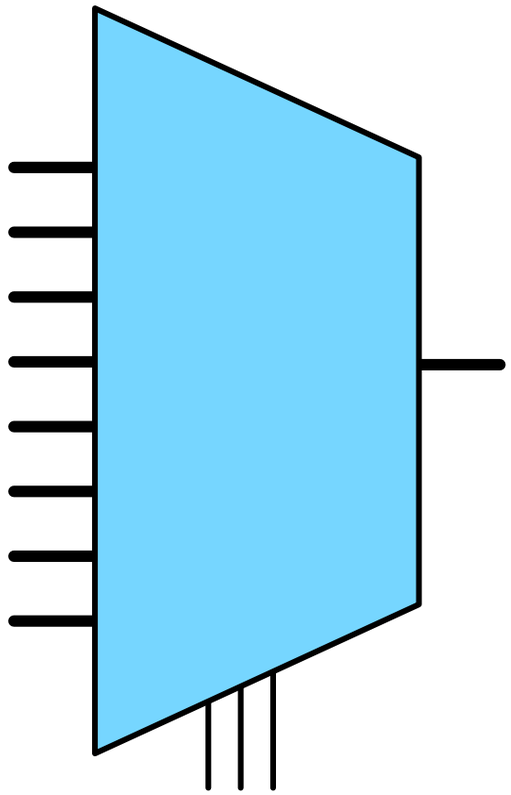
- Combinational circuit that **selects** binary information from one of many input lines and directs it to one output line



Multiplexers (or Muxes)

- Combinational circuit that **selects** binary informati
  lines and directs it to one output line





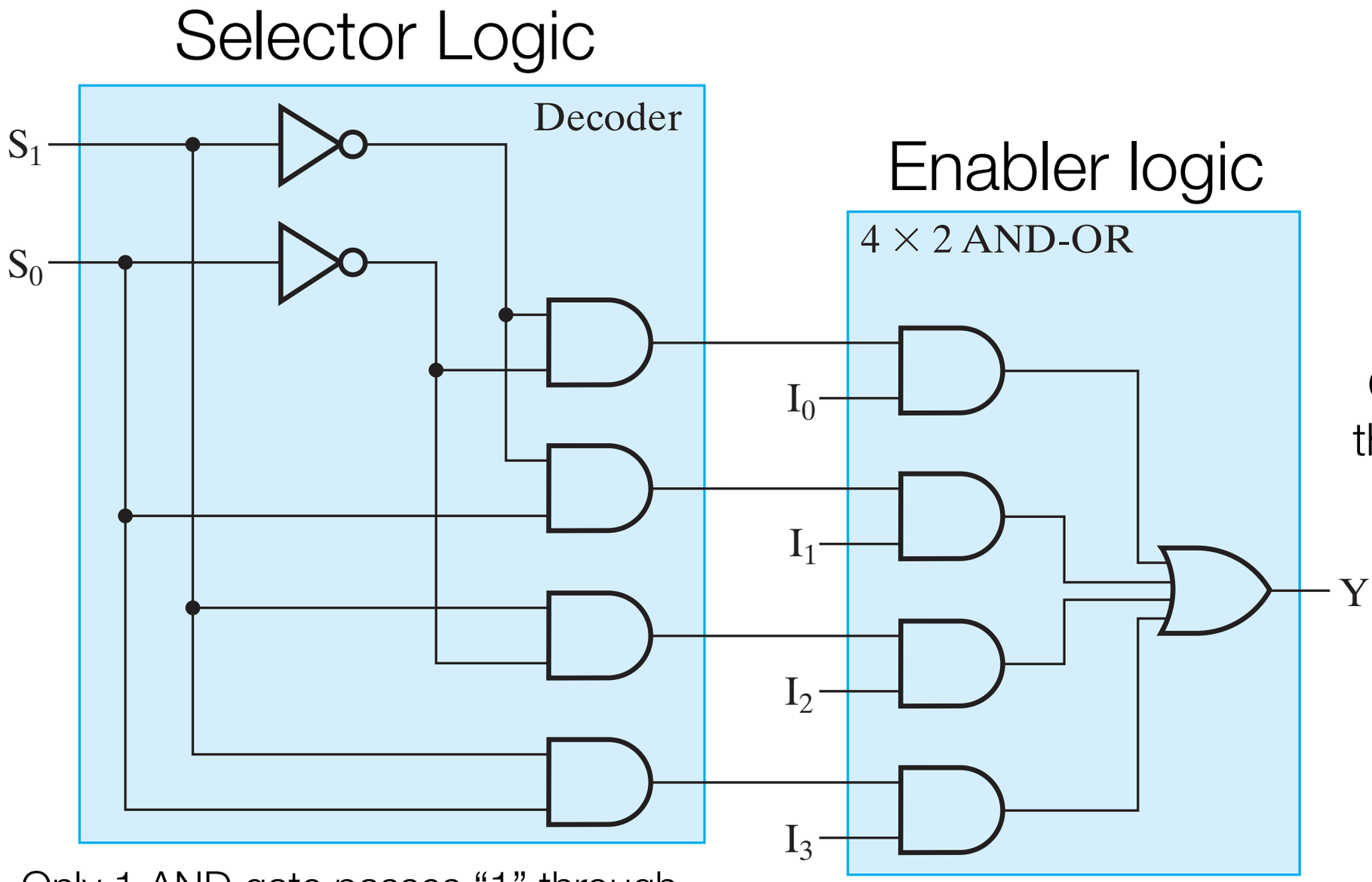
2^n inputs

1 output

n selection bits

indicate (in binary) which input feeds to the output

Internal mux organization



Or gate "passes through" the non-zeroed out I_i

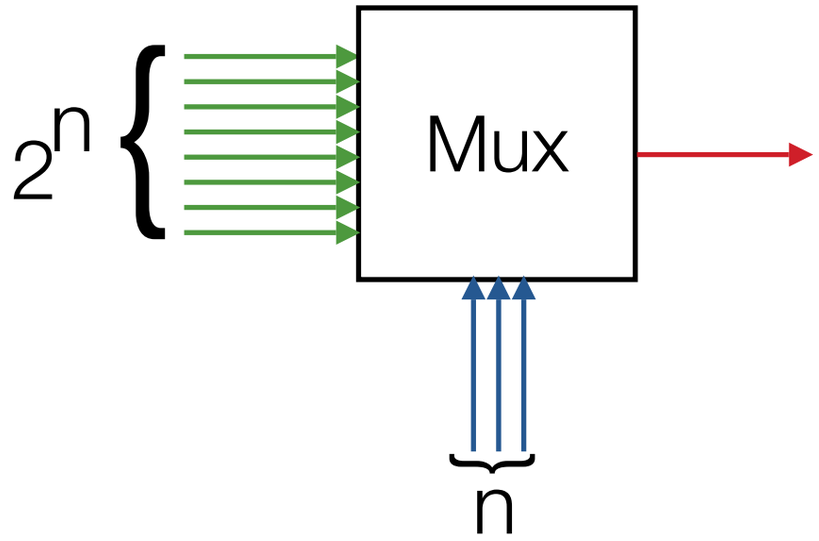
Only 1 AND gate passes "1" through

AND gates "zero out" unselected I_i

In class exercise

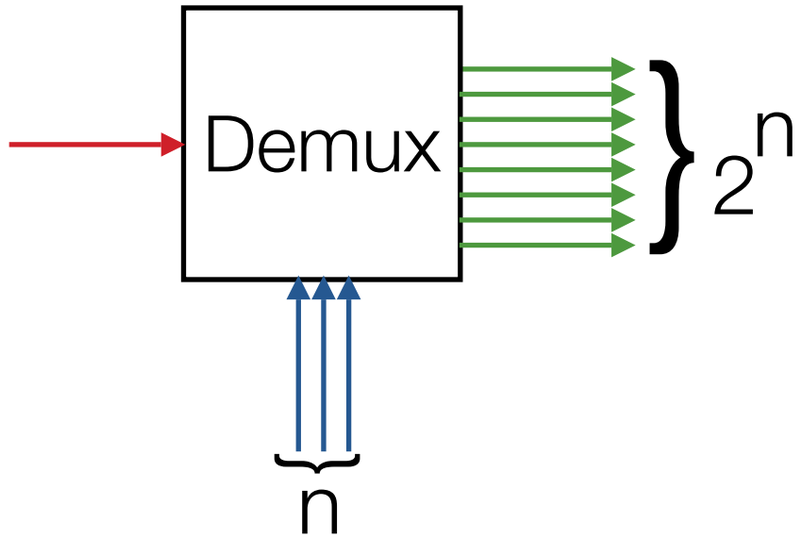
How would you implement an 8:1 mux using two 4:1 muxes?

Multiplexer truth table



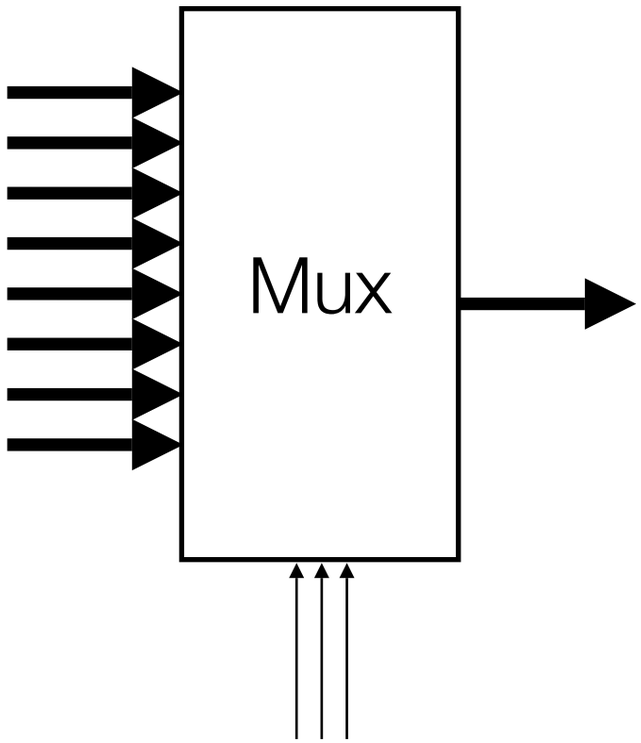
2^n inputs								n-bit BCD value			1 output
a	x	x	x	x	x	x	x	0	0	0	a
x	b	x	x	x	x	x	x	0	0	1	b
x	x	c	x	x	x	x	x	0	1	0	c
x	x	x	d	x	x	x	x	0	1	1	d
x	x	x	x	e	x	x	x	1	0	0	e
x	x	x	x	x	f	x	x	1	0	1	f
x	x	x	x	x	x	g	x	1	1	0	g
x	x	x	x	x	x	x	h	1	1	1	h

Demultiplexers (Demuxes)

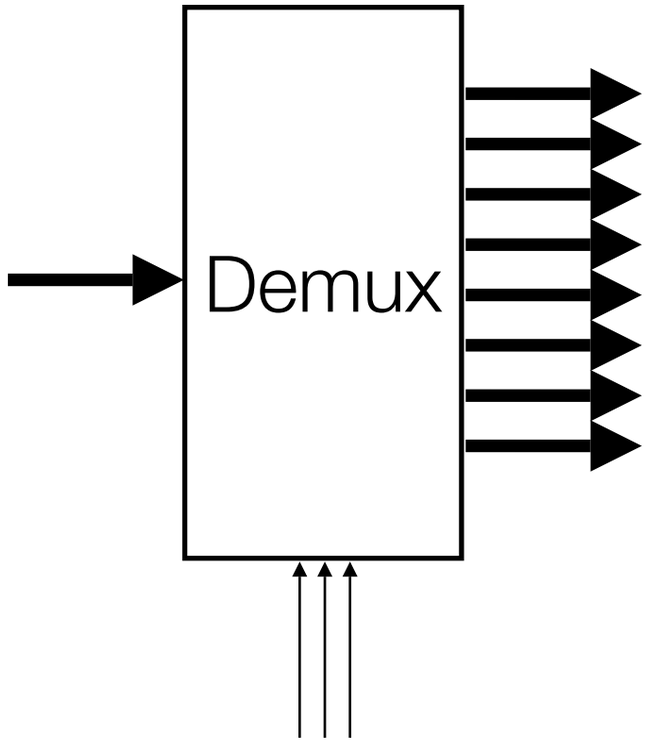


1 input	n-bit BCD value			2^n outputs							
a	0	0	0	a	0	0	0	0	0	0	0
b	0	0	1	0	b	0	0	0	0	0	0
c	0	1	0	0	0	c	0	0	0	0	0
d	0	1	1	0	0	0	d	0	0	0	0
e	1	0	0	0	0	0	0	e	0	0	0
f	1	0	1	0	0	0	0	0	f	0	0
g	1	1	0	0	0	0	0	0	0	g	0
h	1	1	1	0	0	0	0	0	0	0	h

Muxes and demuxes called “steering logic”



“merge”

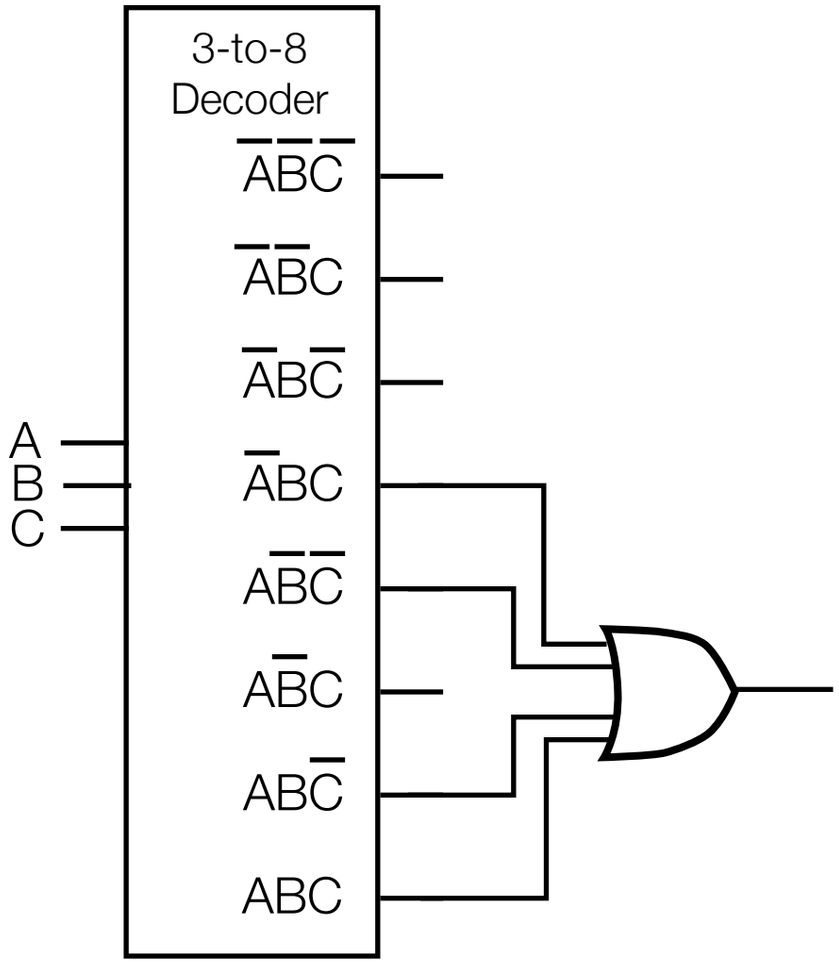


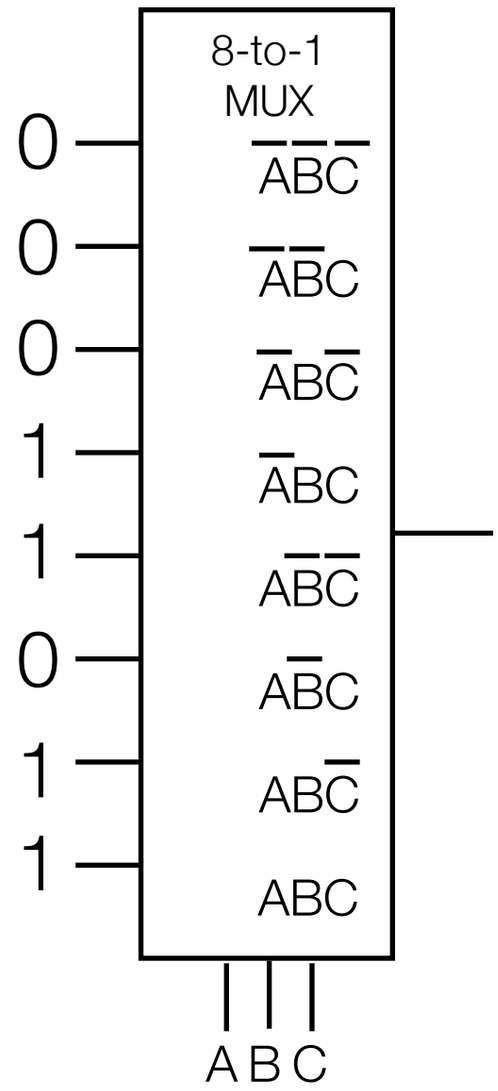
“fork”

Representing Functions with Decoders and MUXes

- e.g., $F = A\bar{C} + BC$

A	B	C	minterm	F
0	0	0	$\bar{A}\bar{B}\bar{C}$	0
0	0	1	$\bar{A}\bar{B}C$	0
0	1	0	$\bar{A}B\bar{C}$	0
0	1	1	$\bar{A}BC$	1
1	0	0	$A\bar{B}\bar{C}$	1
1	0	1	$A\bar{B}C$	0
1	1	0	$AB\bar{C}$	1
1	1	1	ABC	1





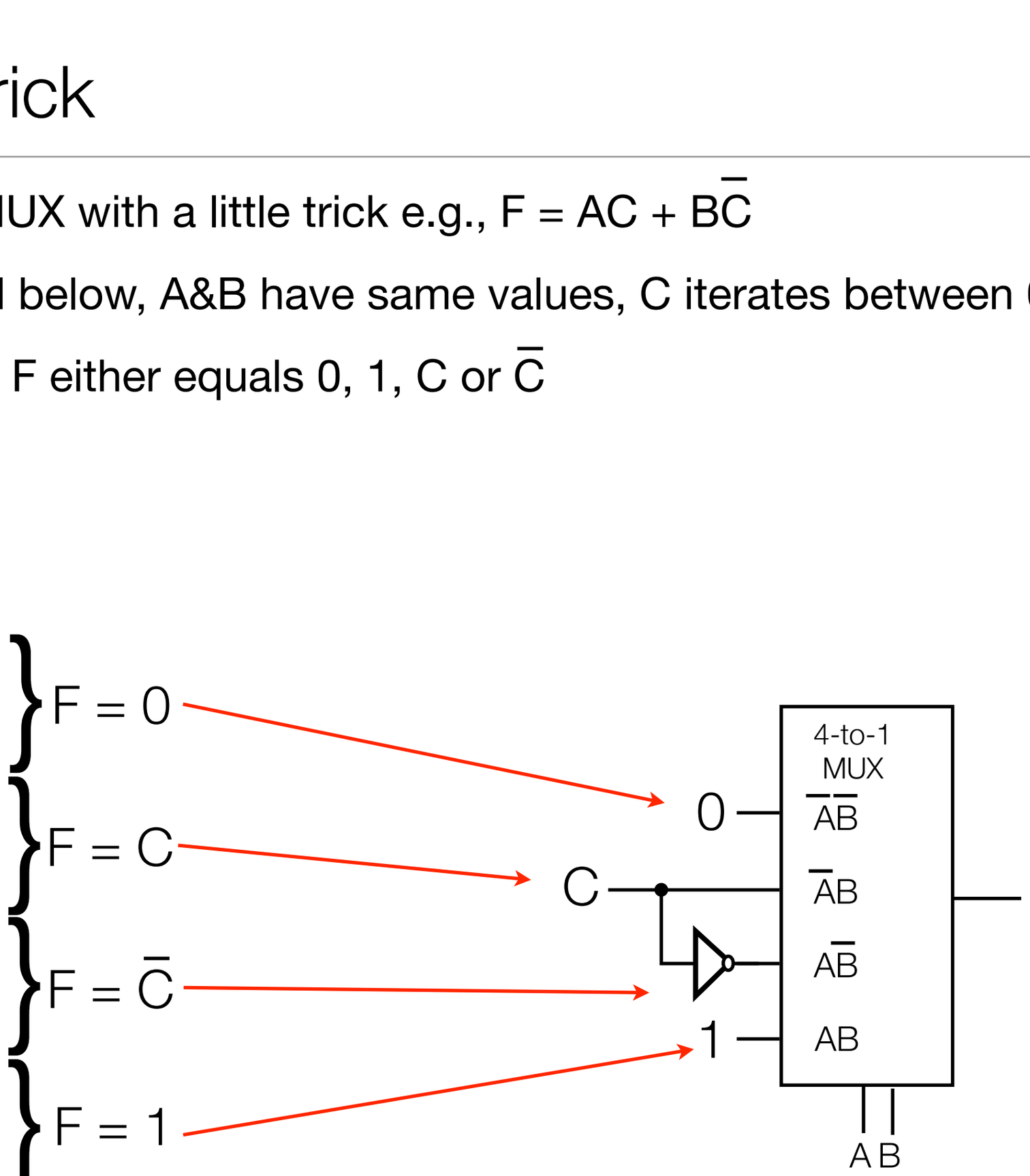
- Decoder: OR minterms for which F should evaluate to 1

- MUX: Feed in the value of F for each minterm

A Slick MUX trick

- Can use a smaller MUX with a little trick e.g., $F = AC + B\bar{C}$
- Note for rows paired below, A&B have same values, C iterates between 0&1
- For the pair of rows, F either equals 0, 1, C or $\bar{C}$

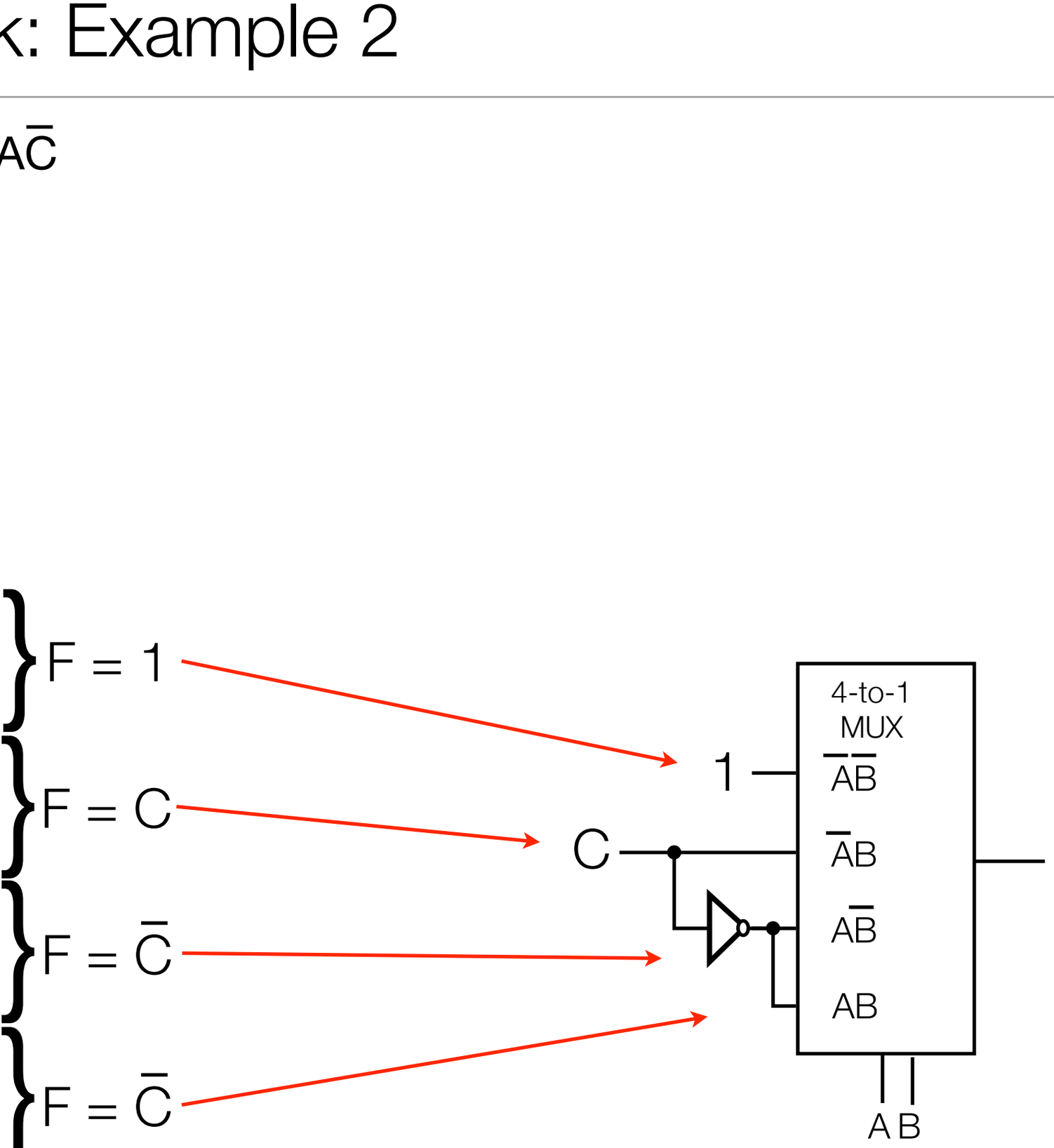
A	B	C	minterm	F
0	0	0	$\bar{A}\bar{B}\bar{C}$	0
0	0	1	$\bar{A}\bar{B}C$	0
0	1	0	$\bar{A}B\bar{C}$	0
0	1	1	$\bar{A}BC$	1
1	0	0	$A\bar{B}\bar{C}$	1
1	0	1	$A\bar{B}C$	0
1	1	0	$AB\bar{C}$	1
1	1	1	ABC	1



Slick MUX trick: Example 2

- e.g., $F = \bar{A}C + \bar{B}\bar{C} + A\bar{C}$

A	B	C	minterm	F	
0	0	0	$\bar{A}\bar{B}\bar{C}$	1	$F = 1$
0	0	1	$\bar{A}\bar{B}C$	1	
0	1	0	$\bar{A}B\bar{C}$	0	$F = C$
0	1	1	$\bar{A}BC$	1	
1	0	0	$A\bar{B}\bar{C}$	1	$F = \bar{C}$
1	0	1	$A\bar{B}C$	0	
1	1	0	$AB\bar{C}$	1	$F = \bar{C}$
1	1	1	ABC	0	



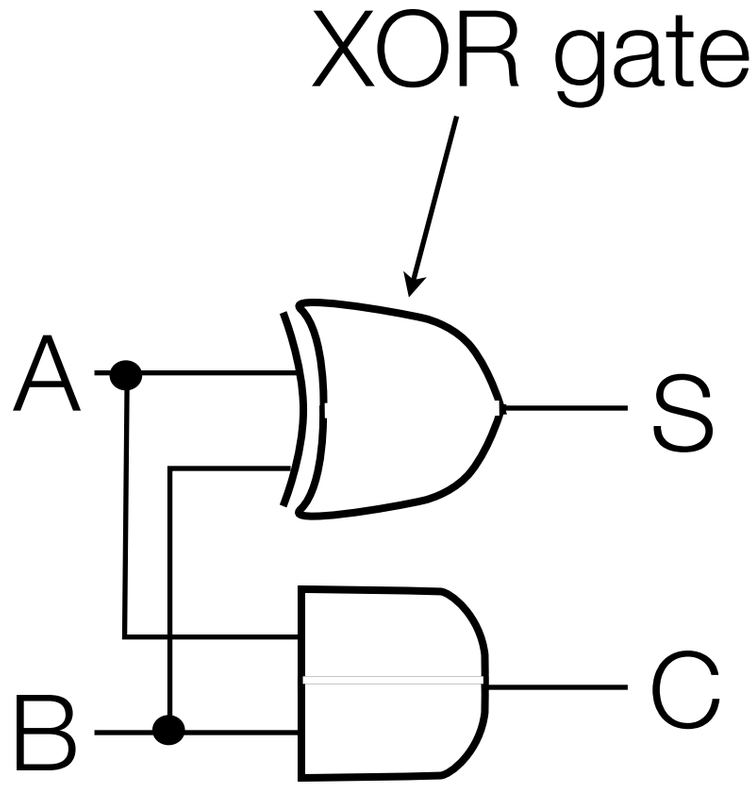
Addition: The Half-Adder

- Addition of 2 bits: A & B produces a summand (S) and carry (C)

A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

$$S = A \oplus B$$

$$C = AB$$



- But to do addition, we really need to add 3 bits at a time (to account for carries), e.g.,

$$\begin{array}{rr} & 011 \\ & \text{carry bits} \\ + & 1011 \\ & 1001 \\ \hline & 10101 \end{array}$$

The Full Adder

- Takes as input 2 digits (A&B) and a previous carry (P)

P	A	B	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

S:

P \ B	0	1	0	1
1	0	1	0	1

A

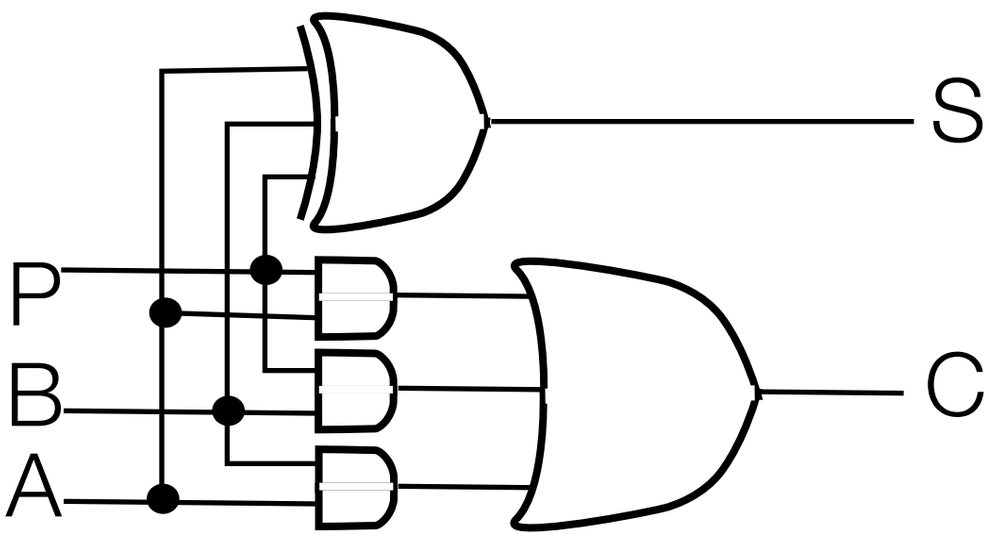
C:

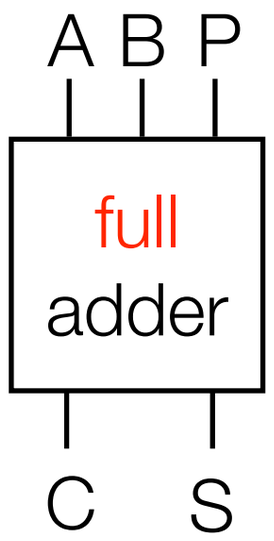
P \ B	0	0	1	0
0	0	1	1	1

A

$$S = A \oplus B \oplus P$$

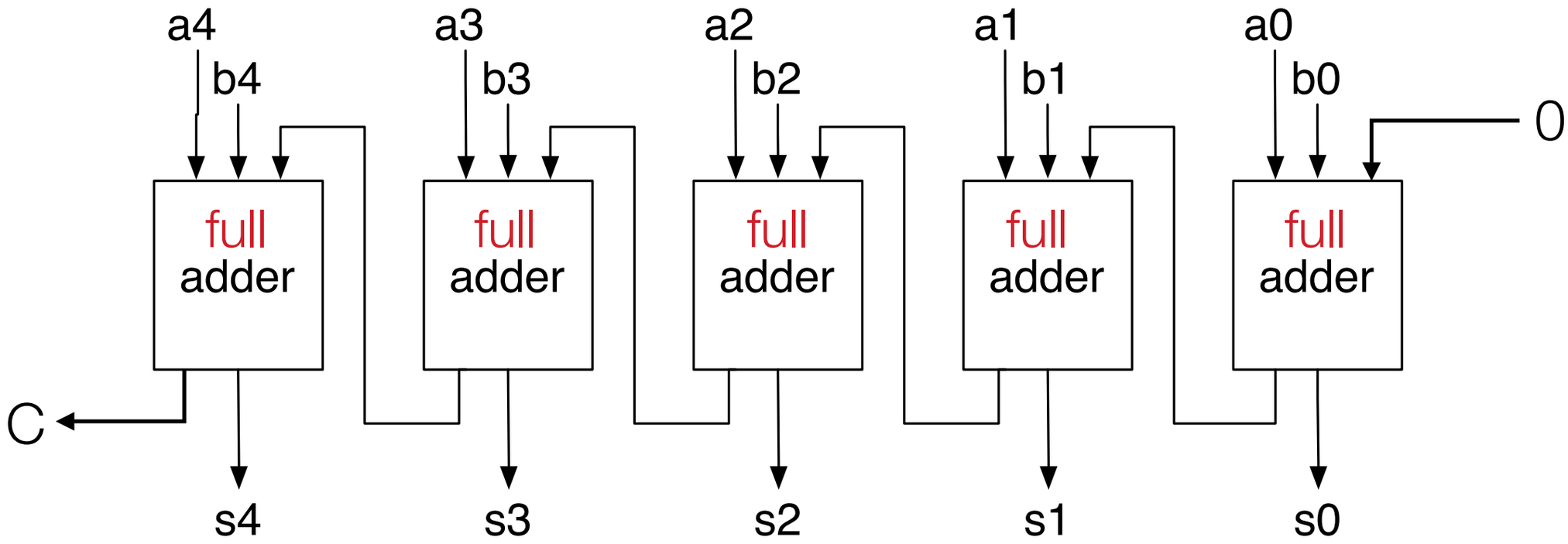
$$C = AB + AP + BP$$





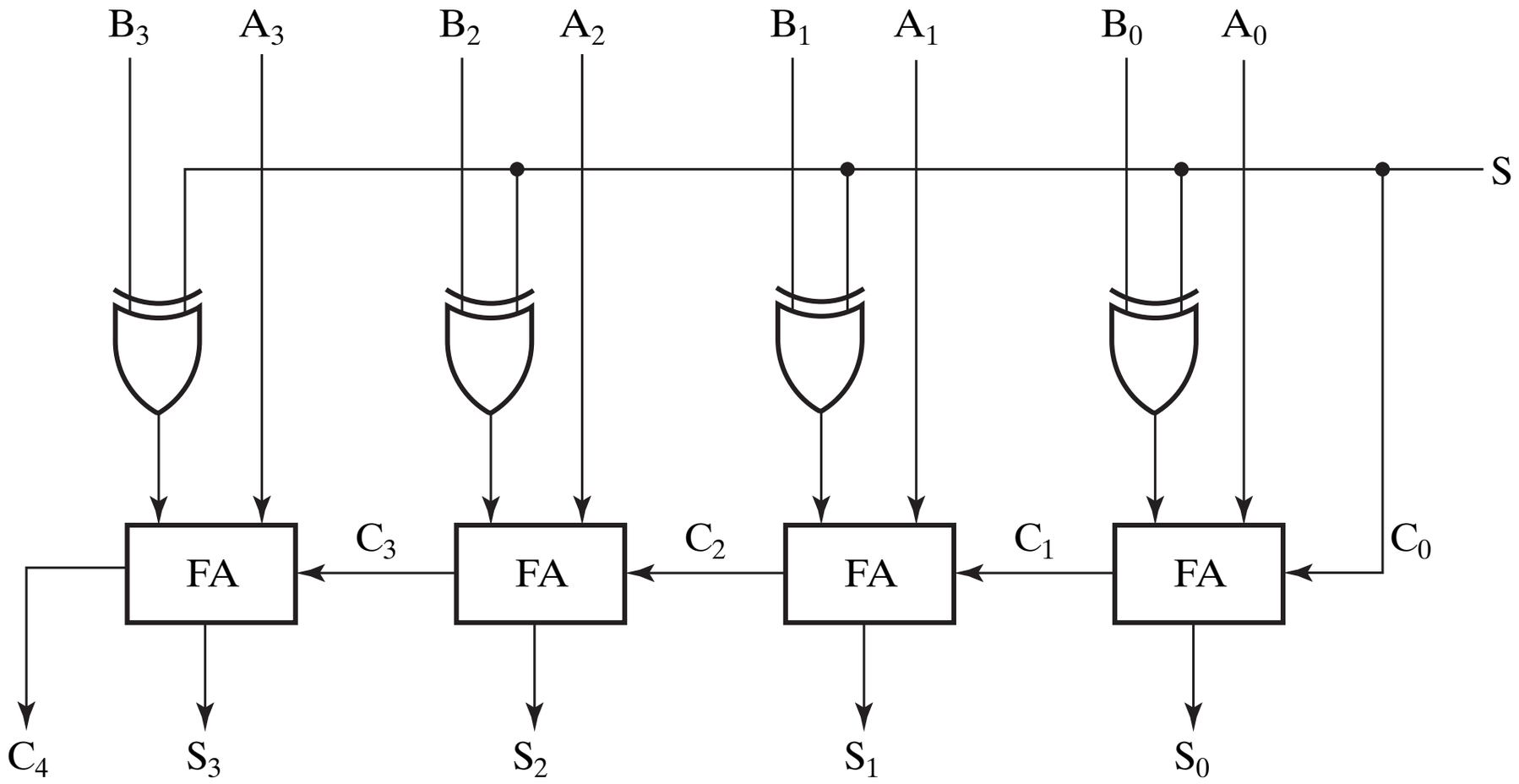
5-bit ripple carry adder

Computes $a_4a_3a_2a_1a_0 + b_4b_3b_2b_1b_0$



- Note how computation "ripples" through adders from left to right
  - Each full adder's has depth 2 (inputs pass through 2 gates to reach output)
  - Full adder that computes s_i cannot "start" its computation until previous full adder computes carry
  - The longest depth in a k-bit ripple carry adder is 2k

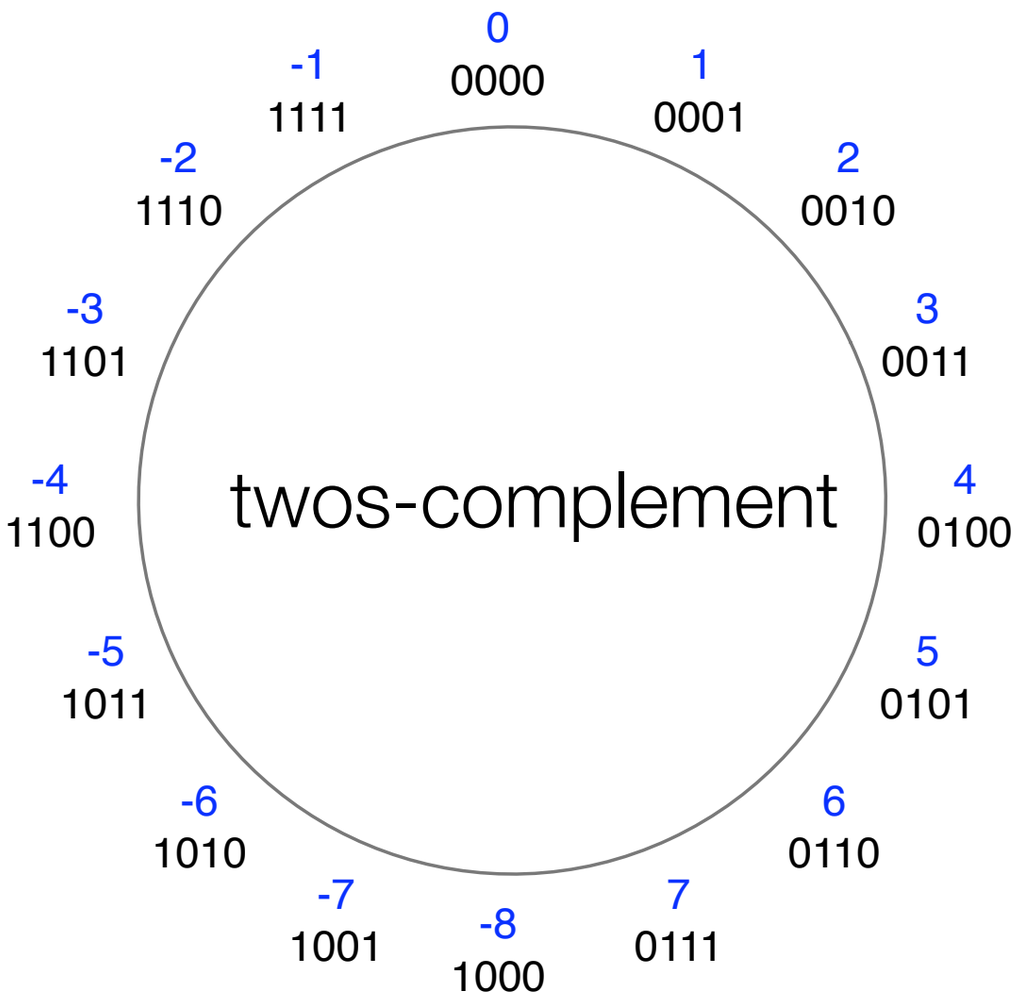
Adder/subtractor for #'s in 2's complement form



S=0: B unchanged, C_0=0: add

S=1: B complemented, C_0=1 (bits flipped and 1 added): subtract

Handling overflow




```
  0111
(5) 0101
(3) 0011
----------
  1000   (-8)
```

```
   1111
(-5) 1011
(-3) 1101
----------
  1000   (-8)
```

```
  1000
(-6) 1010
(-3) 1101
----------
  0111   (7)
```

```
   0010
(-6) 1010
(3)  0011
----------
  1101   (-3)
```

Handling overflow

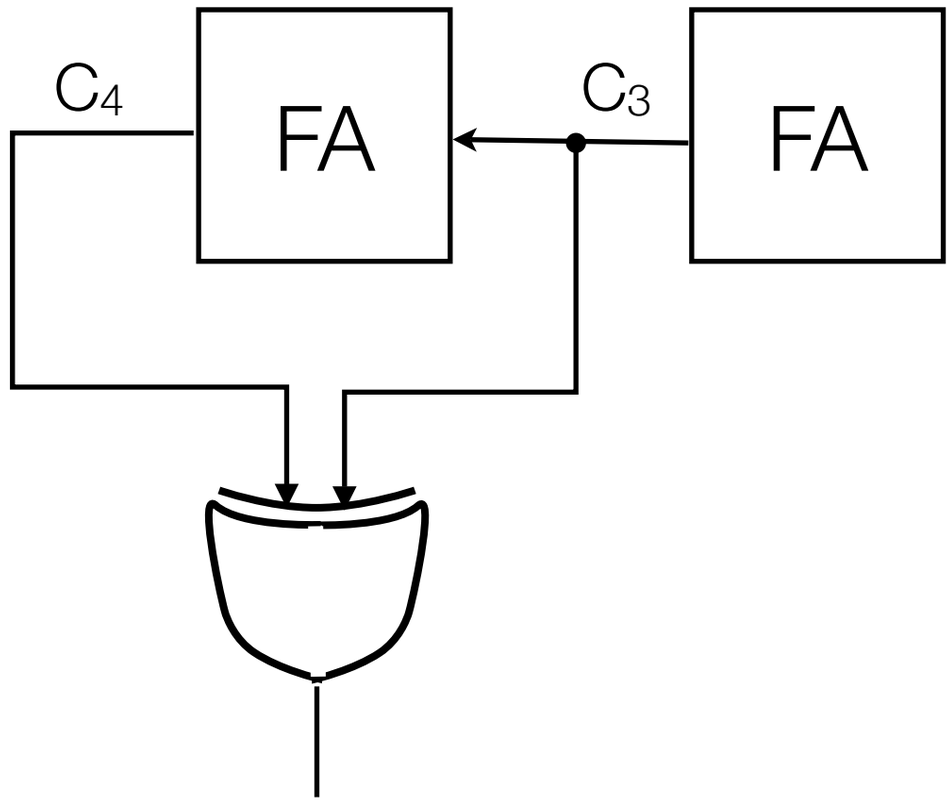
```
c4  c3  c2  c1  c0
    a3  a2  a1  a0
    b3  b2  b1  b0
    ---------------
    s3  s2  s1  s0
```

a3	b3	c3	c4	s3	overflow	
0	0	0	0	0	0	sum of two pos is pos
0	0	1	0	1	1	
0	1	0	0	1	0	
0	1	1	1	0	0	
1	0	0	0	1	0	
1	0	1	1	0	0	
1	1	0	1	0	1	sum of two negs is neg
1	1	1	1	1	0	

n bit two's comp: -2^n <---> 2^n - 1
split into pos and neg ranges and find smallest and largest possible results. show that they're in range for twos comp.

Overflow computation in adder/subtractor

For 2's complement, overflow if 2 most significant carries differ



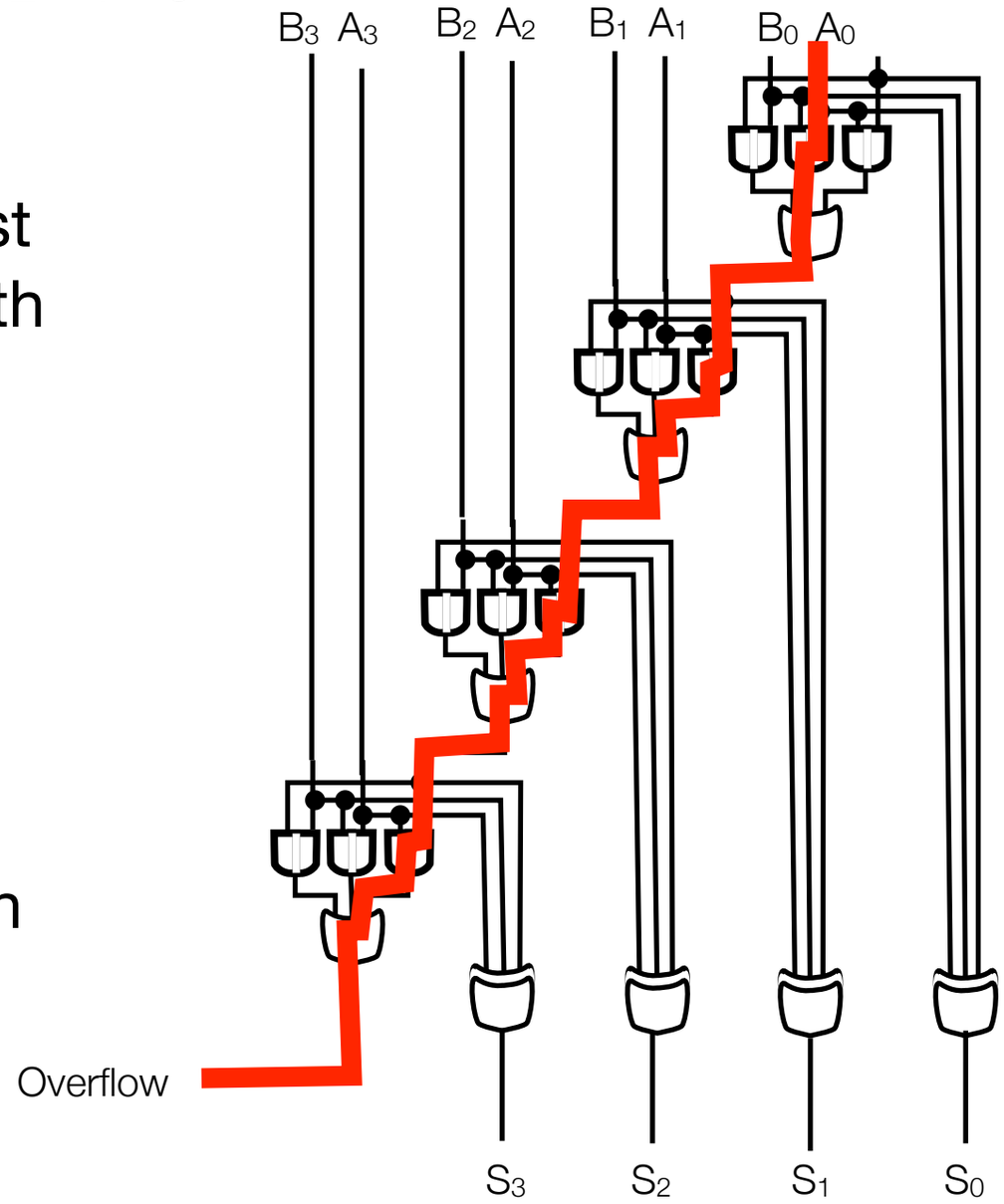
= 0 then no overflow,

= 1 then overflow

Ripple-Carry adder circuit depth

$A_3A_2A_1A_0 + B_3B_2B_1B_0 = S_3S_2S_1S_0$

- Depth of a circuit is the longest (most gates to go through) path
- Overflow has depth 8
- S_3 has depth 7
- In general, S_i has depth 2i+1 in Ripple-Carry Adder



Carry lookahead adder (CLA)

- Goal: produce an adder of shorter circuit depth

- Start by rewriting the carry function

$$c_{i+1} = a_i b_i + a_i c_i + b_i c_i$$

$$c_{i+1} = a_i b_i + c_i (a_i + b_i)$$

$$c_{i+1} = g_i + c_i (p_i)$$

carry generate

$g_i = a_i b_i$

carry propagate

$p_i = a_i + b_i$

Carry lookahead adder (CLA) (2)

- Can recursively define carries in terms of propagate and generate signals

$$c_1 = g_0 + c_0 p_0$$

$$c_2 = g_1 + c_1 p_1$$

$$= g_1 + (g_0 + c_0 p_0) p_1$$

$$= g_1 + g_0 p_1 + c_0 p_0 p_1$$

$$c_3 = g_2 + c_2 p_2$$

$$= g_2 + (g_1 + g_0 p_1 + c_0 p_0 p_1) p_2$$

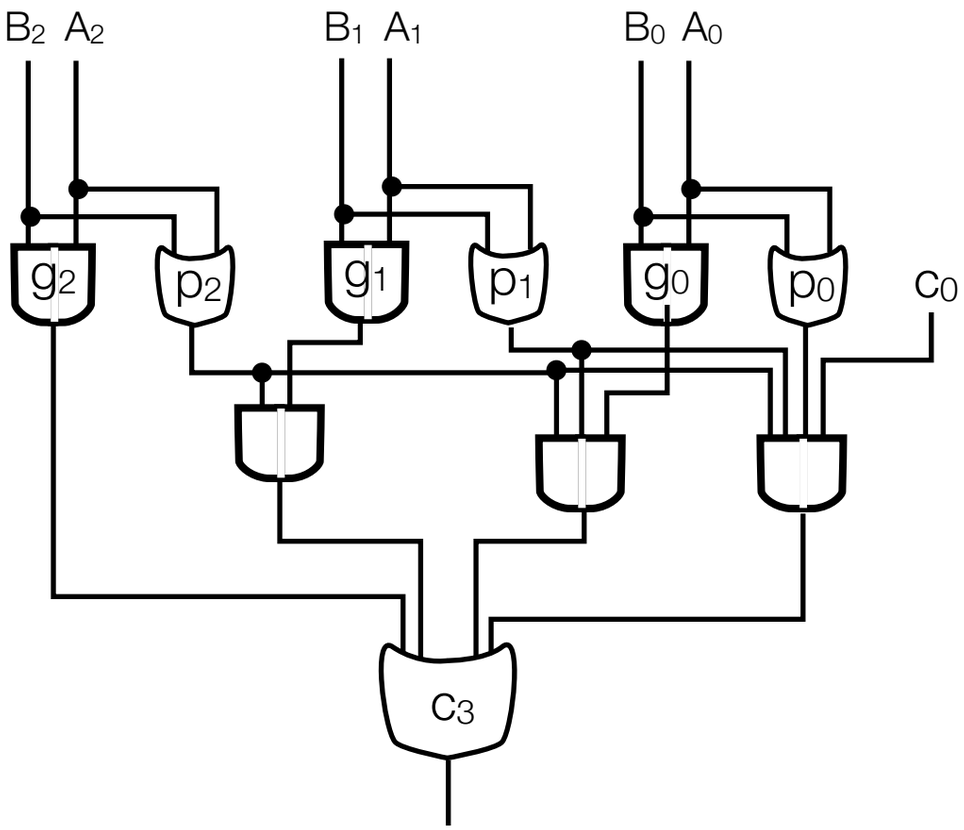
$$= g_2 + g_1 p_2 + g_0 p_1 p_2 + c_0 p_0 p_1 p_2$$

- ith carry has i+1 product terms, the largest of which has i+1 literals
- If AND, OR gates can take unbounded inputs: total circuit depth is 2 (SoP form)
- If gates take 2 inputs, total circuit depth is $1 + \log_2 k$ for k-bit addition

Carry lookahead adder (CLA) (3)

$$c_0 = 0$$
$$c_1 = g_0 + c_0p_0$$
$$c_2 = g_1 + g_0p_1 + c_0p_0p_1$$
$$c_3 = g_2 + g_1p_2 + g_0p_1p_2 + c_0p_0p_1p_2$$
$$c_4 = g_3 + g_2p_3 + g_1p_2p_3 + g_0p_1p_2p_3 + c_0p_0p_1p_2p_3$$

$$s_0 = a_0 \oplus b_0 \oplus c_0$$
$$s_1 = a_1 \oplus b_1 \oplus c_1$$
$$s_2 = a_2 \oplus b_2 \oplus c_2$$
$$s_3 = a_3 \oplus b_3 \oplus c_3$$
$$s_4 = a_4 \oplus b_4 \oplus c_4$$



Depth of 3 for all c_i

Depth of 4 for all s_i, i>0

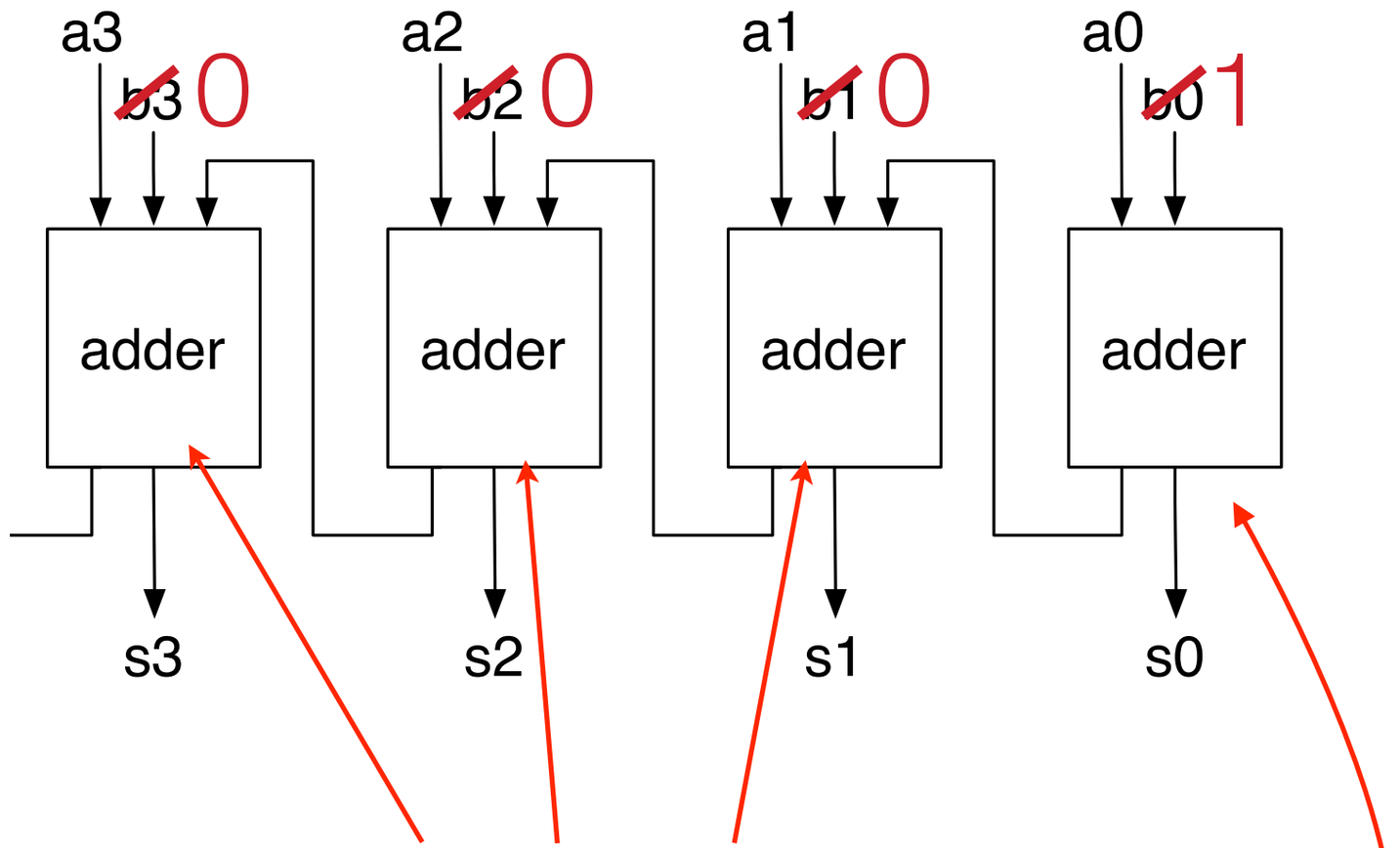
Note: gates take only 2 inputs: depth increases by a log2 factor: still much less than linear of ripple-carry adder

Contraction

Contraction is the simplification of a circuit through constant input values.

Contraction example: adder to incrementer

- What is the hardware and delay savings of implementing an incrementer using contraction?



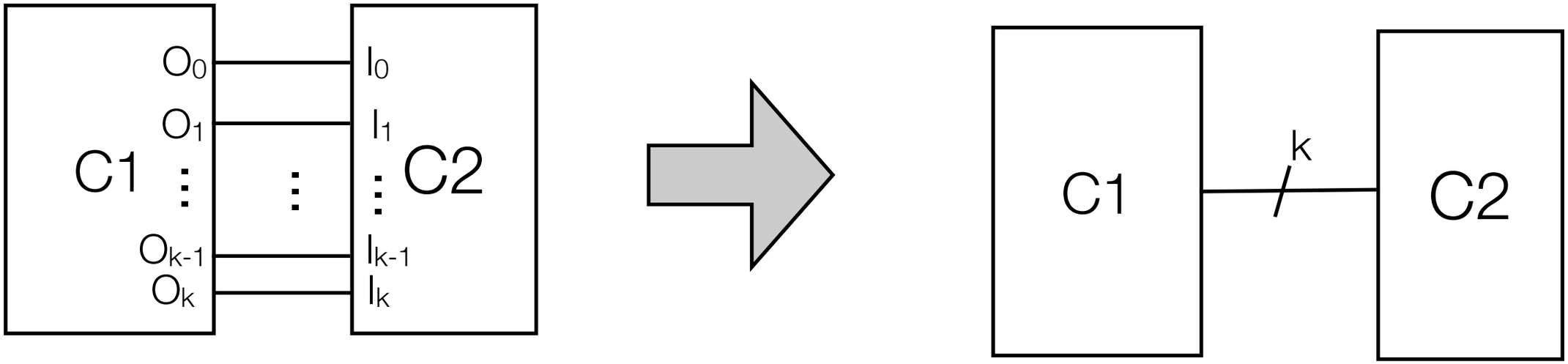
Incrementer circuit

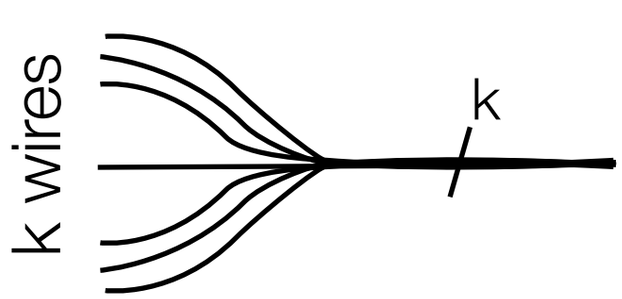
a_0	S	C
0	1	0
1	0	1

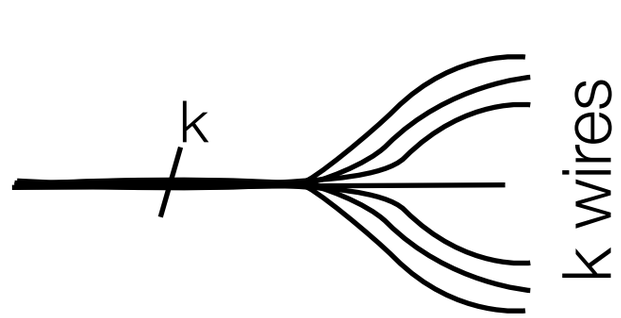
$S_0=\overline{a}_0$, $C_0=a_0$

Multi-wire notation

- Useful when running a bunch of bits in parallel to the same (similar place)

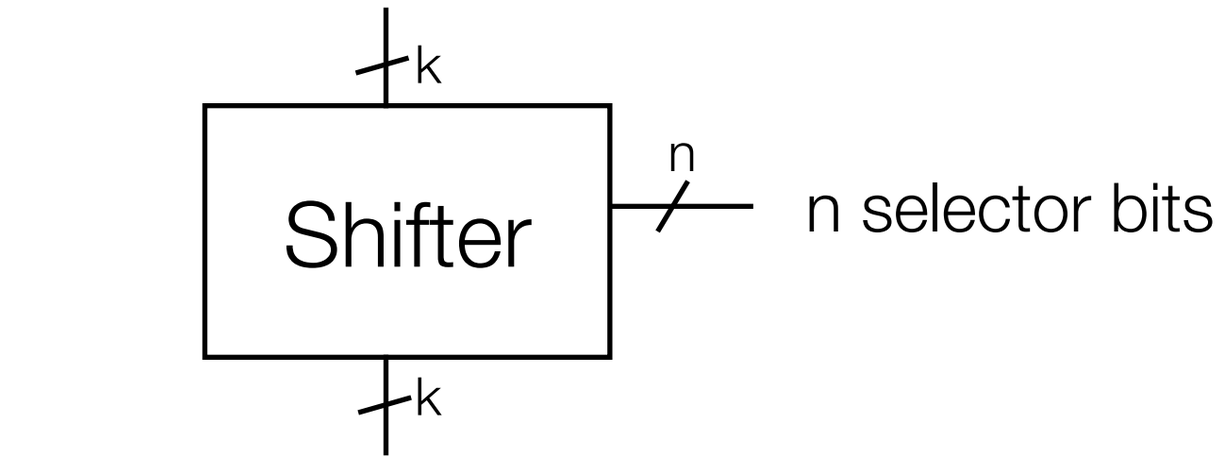






Shifter Circuit

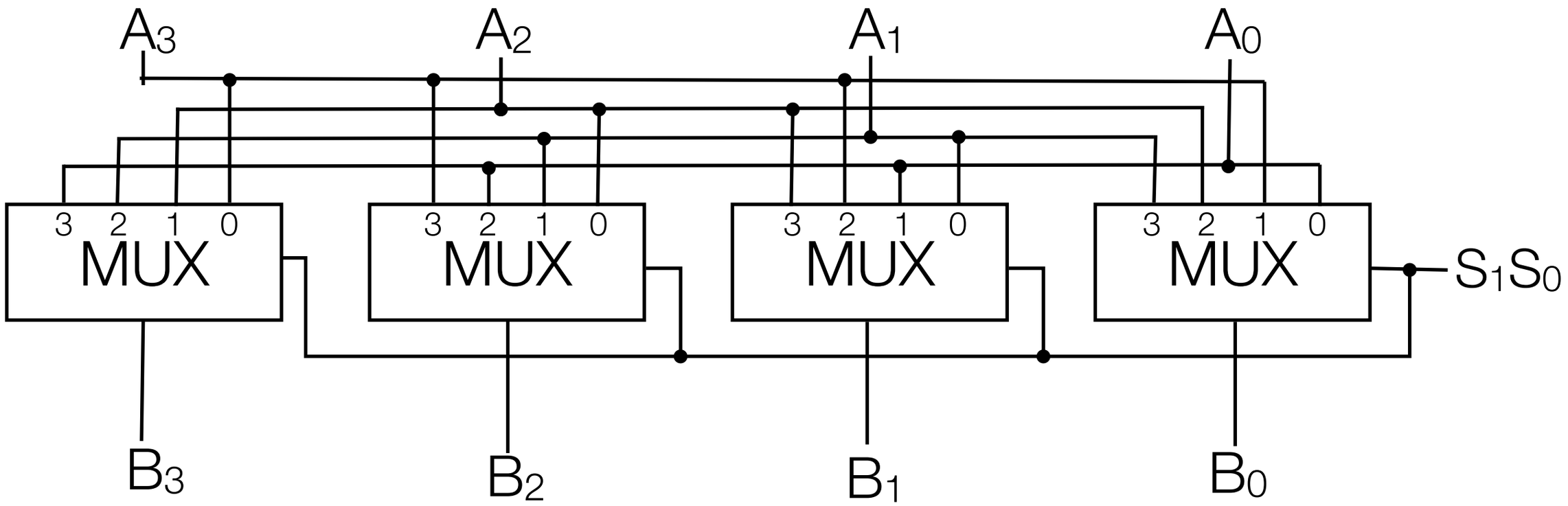
- Shifts bits of a word: $A_{k-1}A_{k-2}\ldots A_2A_1A_0$



$B_{k-1}B_{k-2}\ldots B_2B_1B_0$

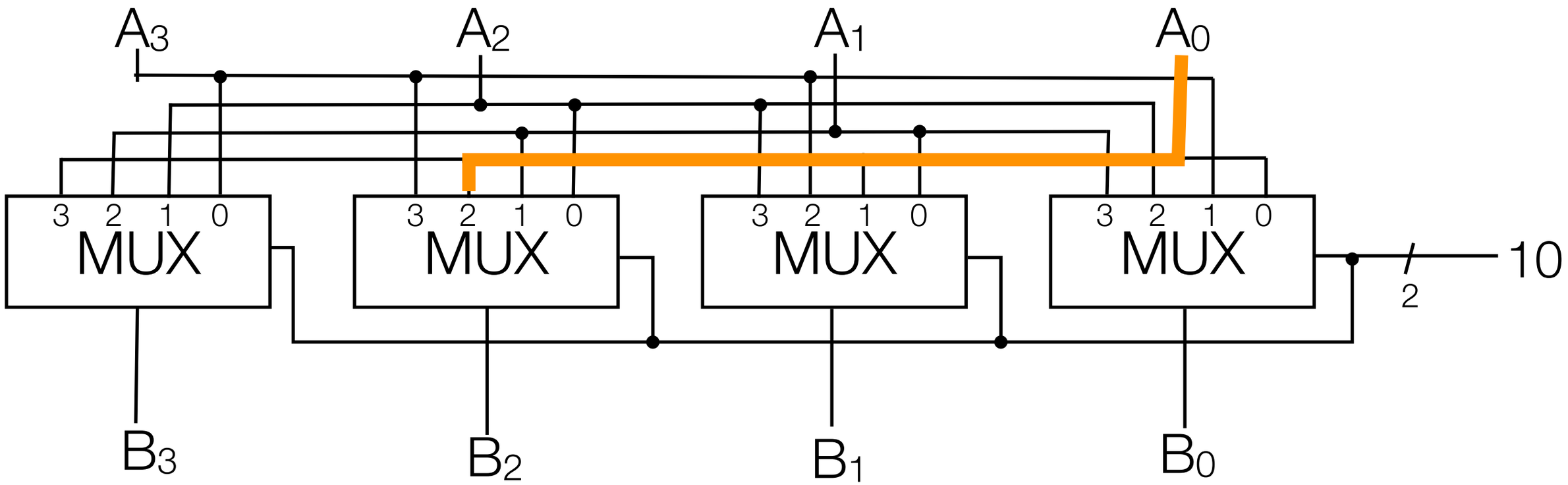
- Various types of shifters
  - Barrel: selector bits indicate (in binary) how "far" bits shift
    - selector value = j, then $B_i = A_{i-j}$
    - bits can "wraparound" $B_i \pmod{2^n} = A_{i-j} \pmod{2^n}$ or rollout (B_i=0 for i<j)
  - L/R with enable: n=2, high bit enables, low bit indicates direction (e.g., 0=left [$B_i = A_{i-1}$], 1=right [$B_i = A_{i+1}$])

Barrel Shifter Design with wraparound (using MUXs)



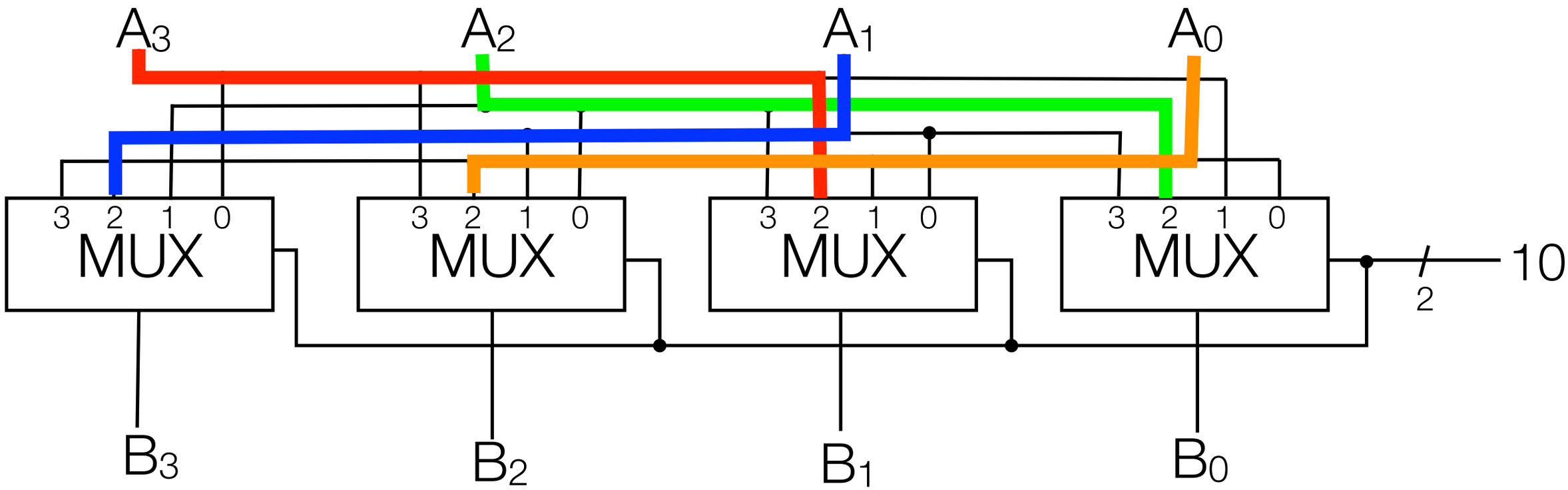
- Basic form of design: Each A_i feeds into each MUX connecting to B_j into input (j-i) mod 4

Barrel Shifter Design with wraparound (using MUXs)



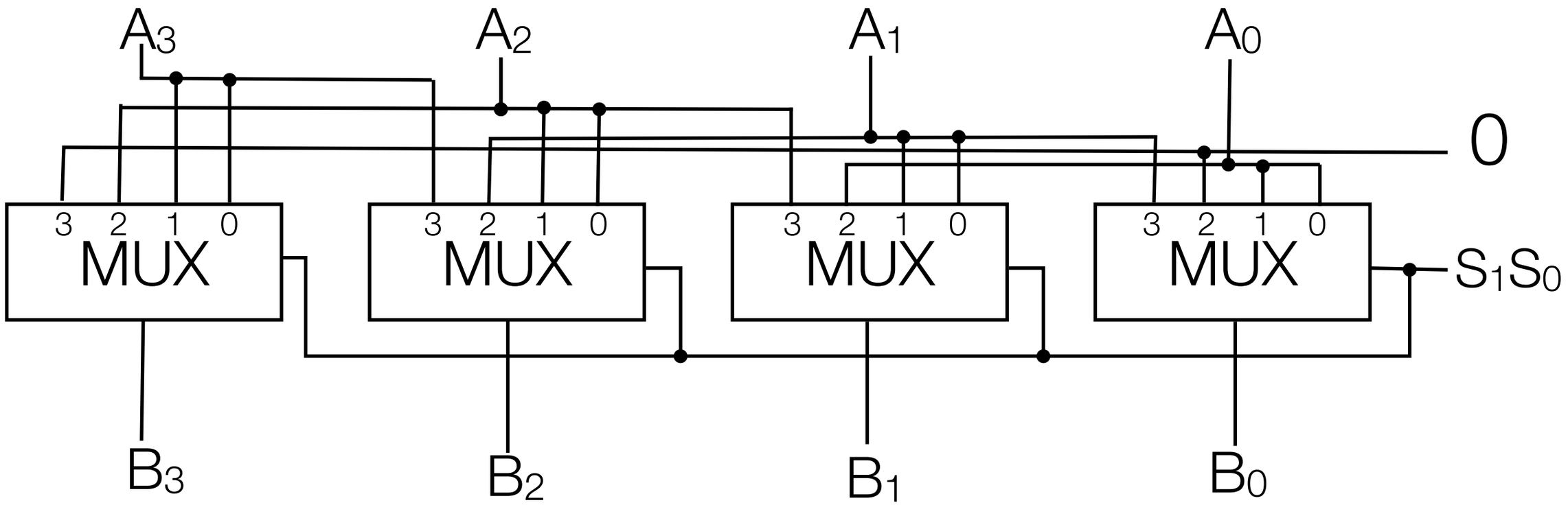
- Basic form of design: Each A_i feeds into each MUX connecting to B_j into input (j-i) mod 4
- Selector is 10 (i.e., 2 binary):
- each MUX entry 2 is selected
  - A_0 flows into the ‘2’ input of the MUX whose output is B_2

Barrel Shifter Design with wraparound (using MUXs)



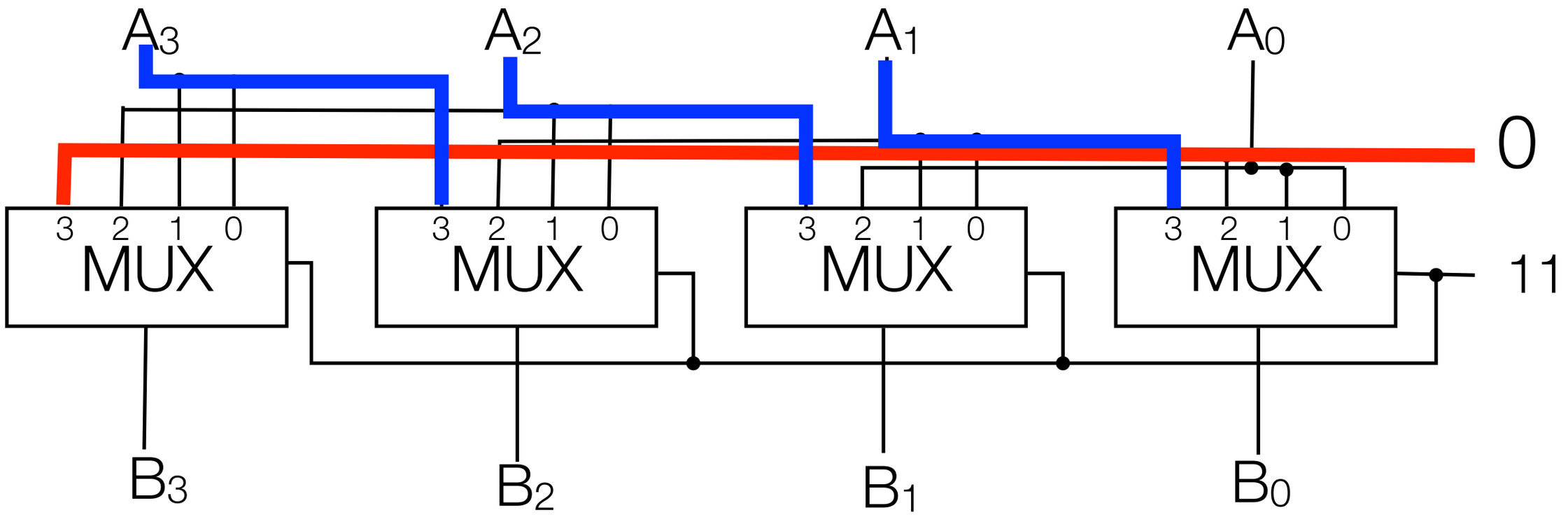
- Basic form of design: Each A_i feeds into each MUX connecting to B_j into input (j-i) mod 4
- Selector is 10 (i.e., 2 binary):
- each MUX entry 2 is selected
  - $B_3\ B_2\ B_1\ B_0 = A_1\ A_0\ A_3\ A_2$

L/R Shift w/ Rollout



- Basic form of design:
  - 0 & 1 MUX selectors ($S_1 = 0$) feed A_i to B_i
  - 2 MUX selector feeds from left ($B_i = A_{i-1}$), 3 MUX from right ($B_i = A_{i+1}$)
  - Note 0 feeds (0's roll in when bits rollout)

L/R Shift w/ Rollout



- Basic form of design:
  - 0 & 1 MUX selectors ($S_1 = 0$) feed A_i to B_i
  - 2 MUX selector feeds from left ($B_i = A_{i-1}$), 3 MUX from right ($B_i = A_{i+1}$)
  - Note 0 feeds (0's roll in when bits rollout)