CSEE 3827: Fundamentals of Computer Systems

Combinational Circuits

Outline (M&K 3.1, 3.3, 3.6-3.9, 4.1-4.2, 4.5, 9.4)

- **Combinational Circuit Design**
    - Standard combinational circuits
        - enabler
        - decoder
        - encoder / priority encoder
        - Code converter
        - MUX (multiplexer) & DeMux
    - Addition / Subtraction
        - half and full adders
        - ripple carry adder
        - carry lookahead adder
    - Shifter

Combinational circuits

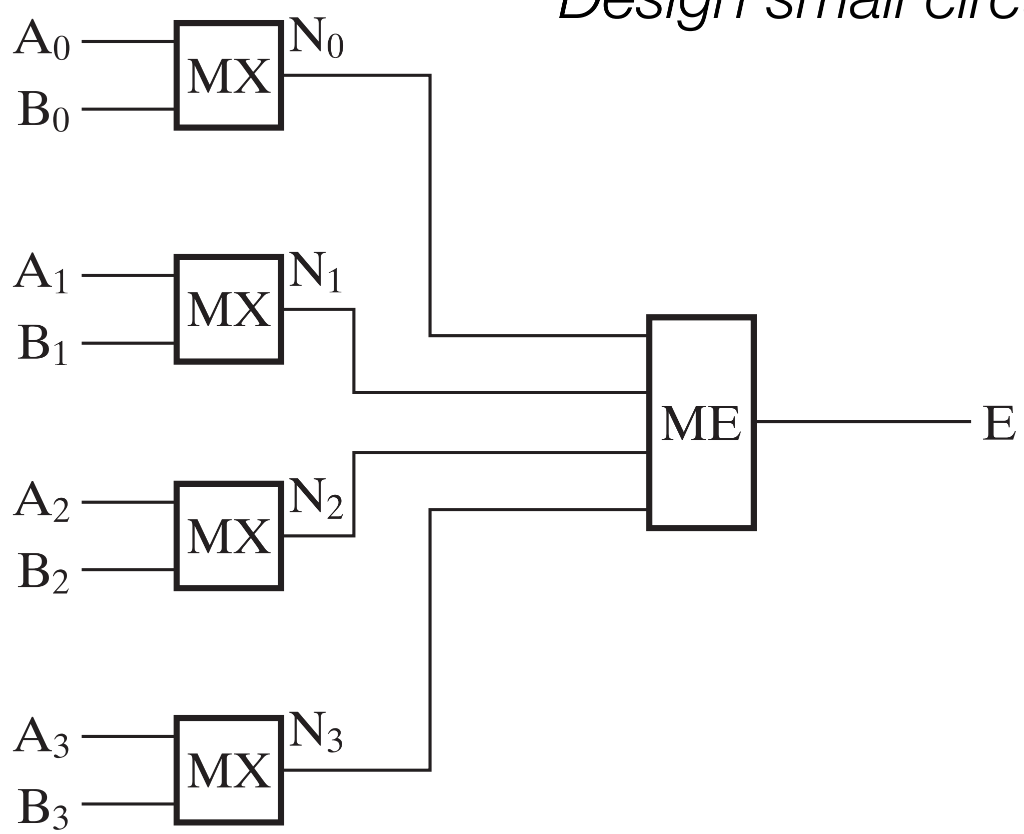
- Combinational circuits are stateless
- The outputs are functions only of the inputs

Hierarchical design

3-4

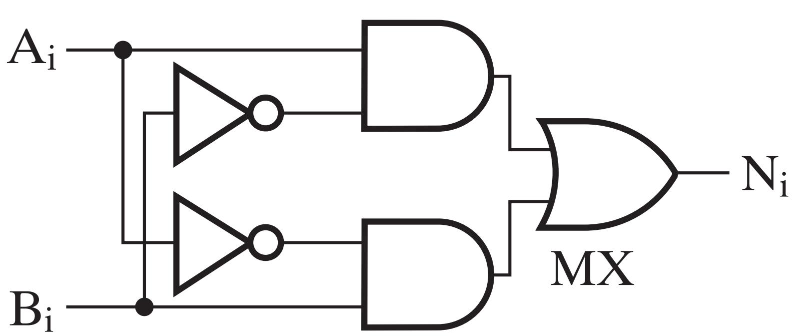
Design small circuits to be used in a bigger circuit

"Big" Circuit

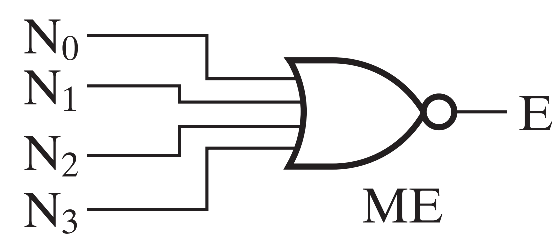


(a)

Smaller Circuits



(b)

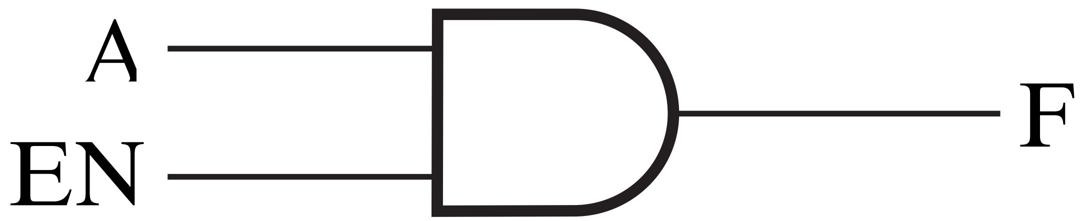


(c)

Enabler circuits

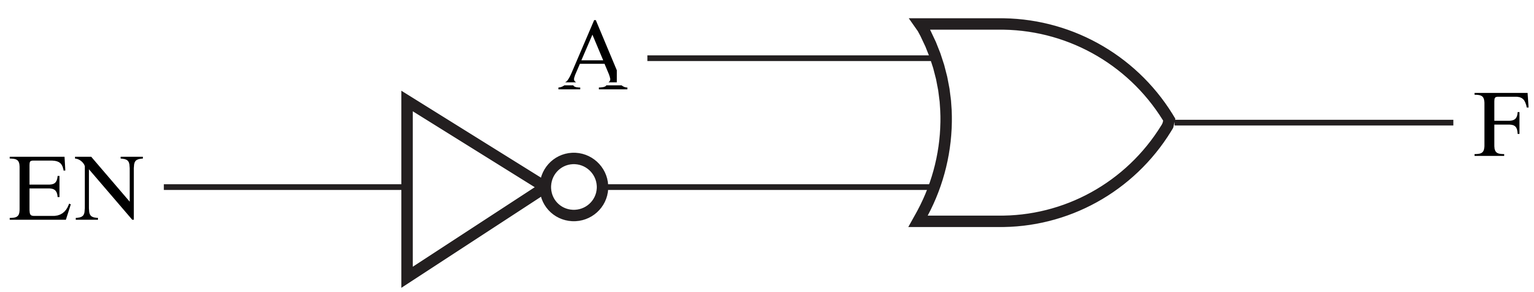
3-15

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



(a)

EN	F
0	0
1	A

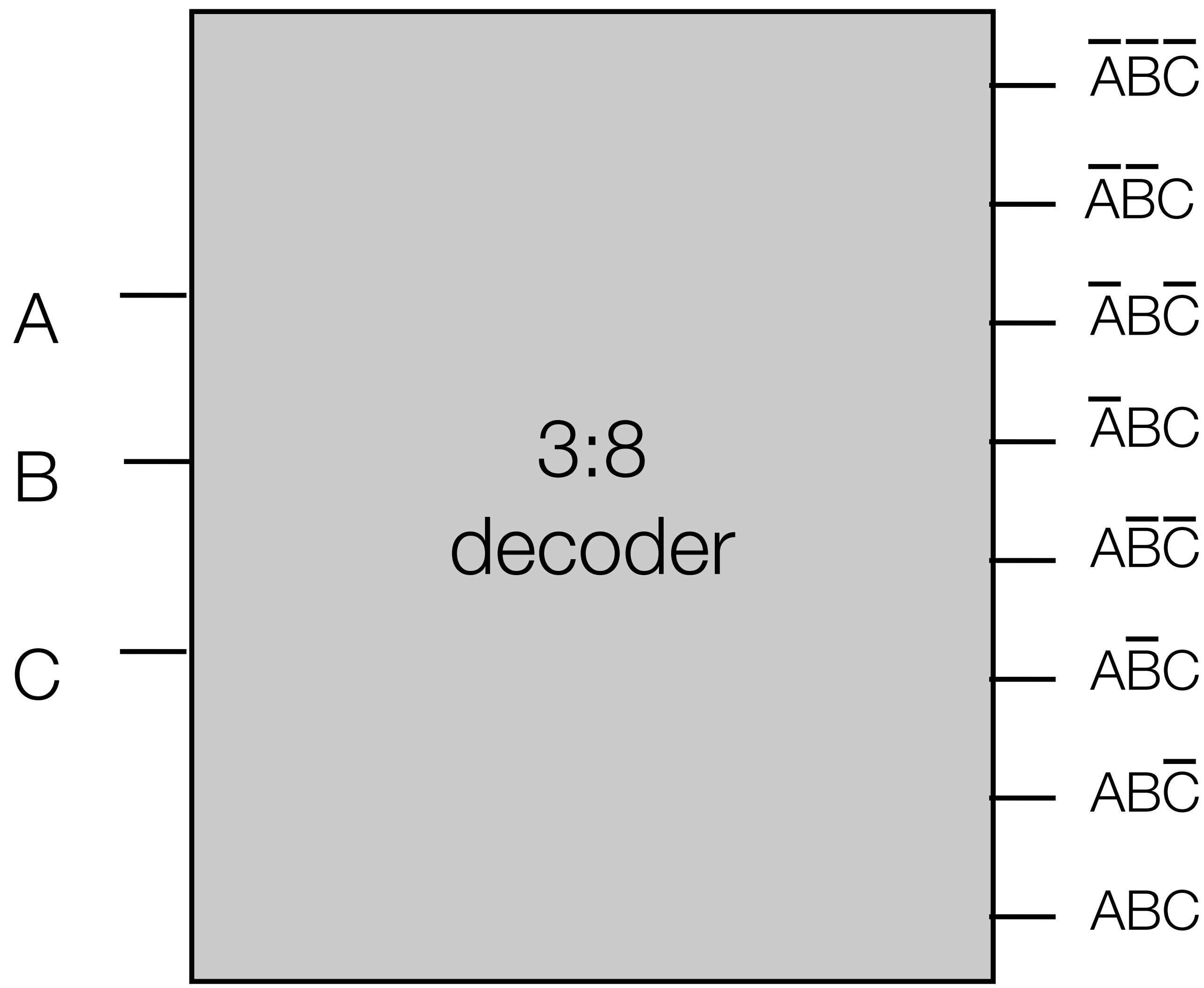


(b)

EN	F
0	1
1	A

Decoder-based circuits

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

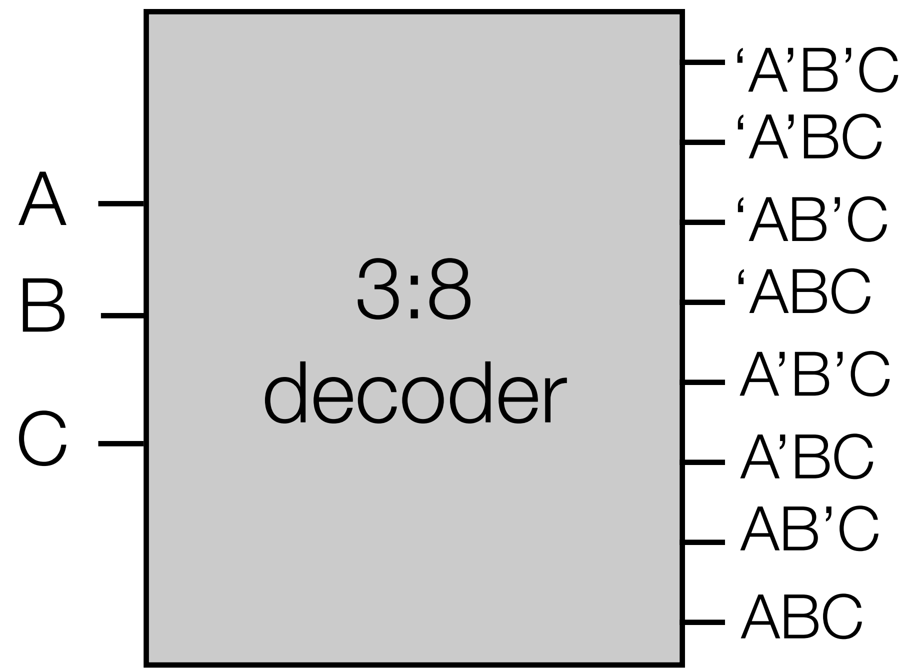


"Standard" Decoder: i^{th} output = 1, all others = 0,

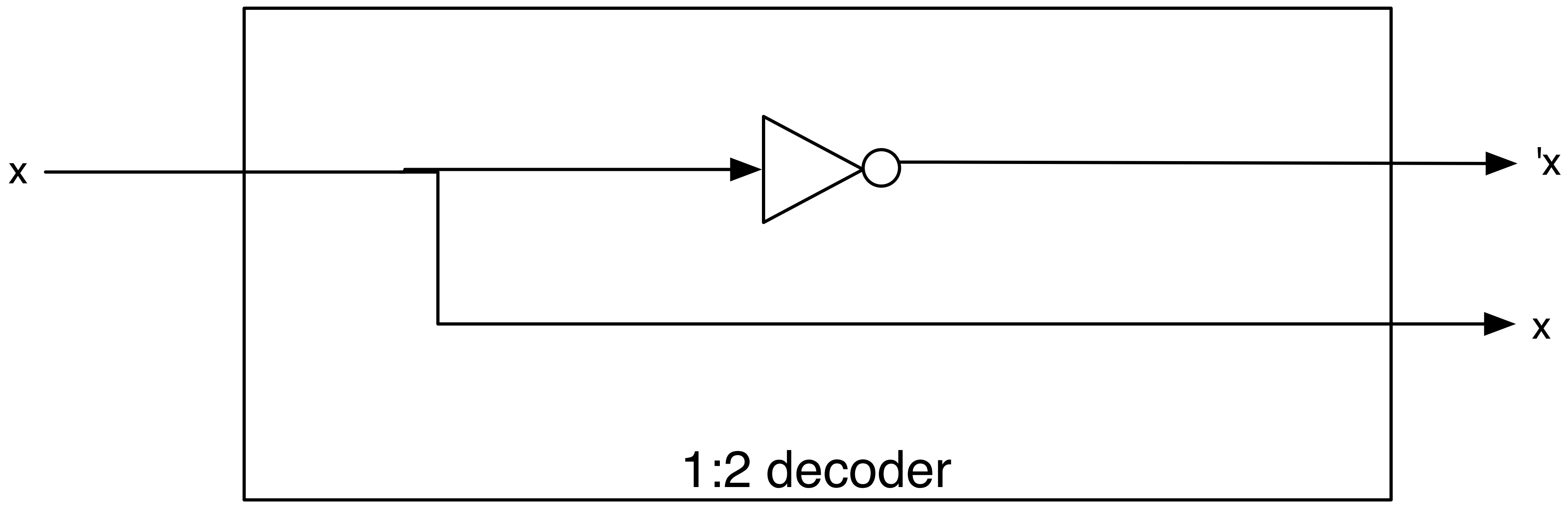
where i is the binary representation of the input (ABC)

Decoder-based circuits

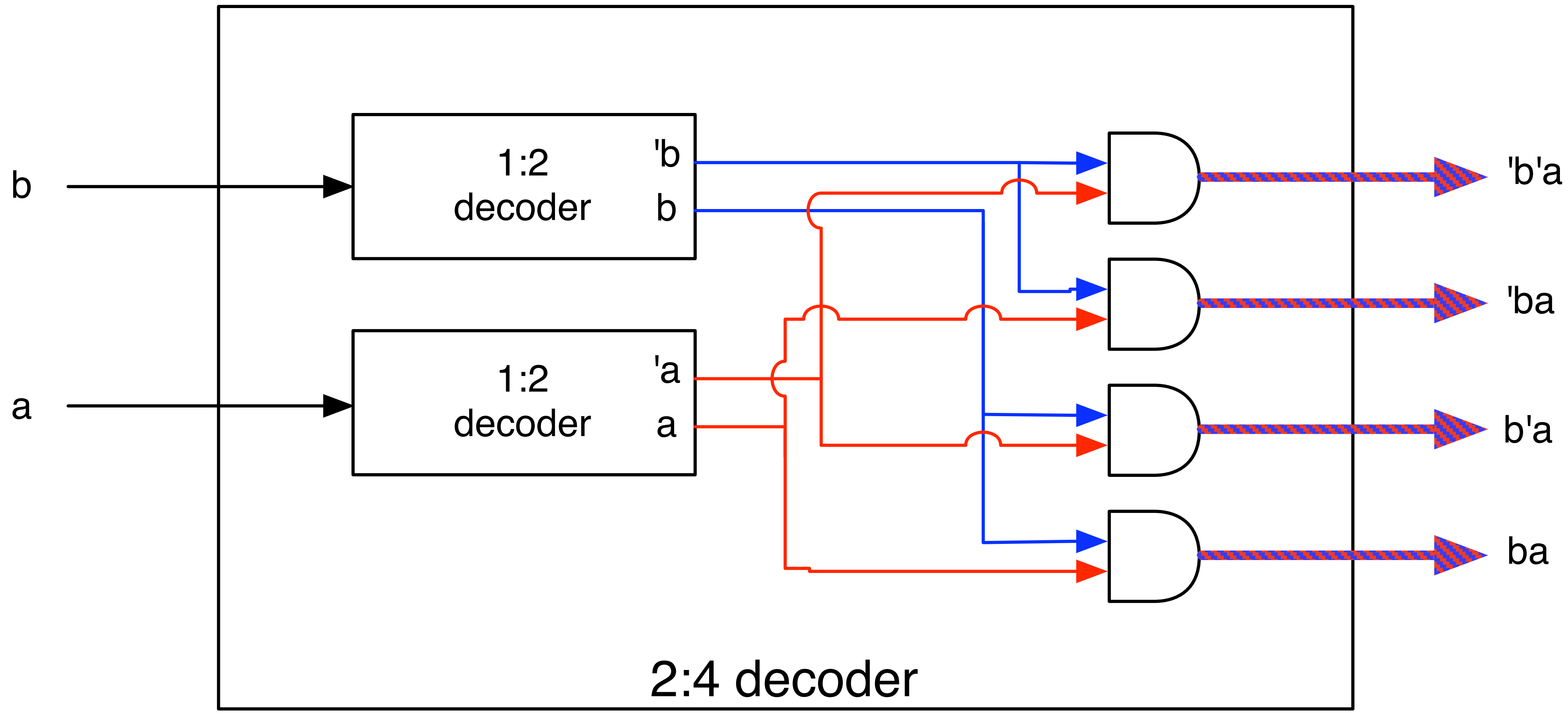
So, if decoders produce minterms. . .



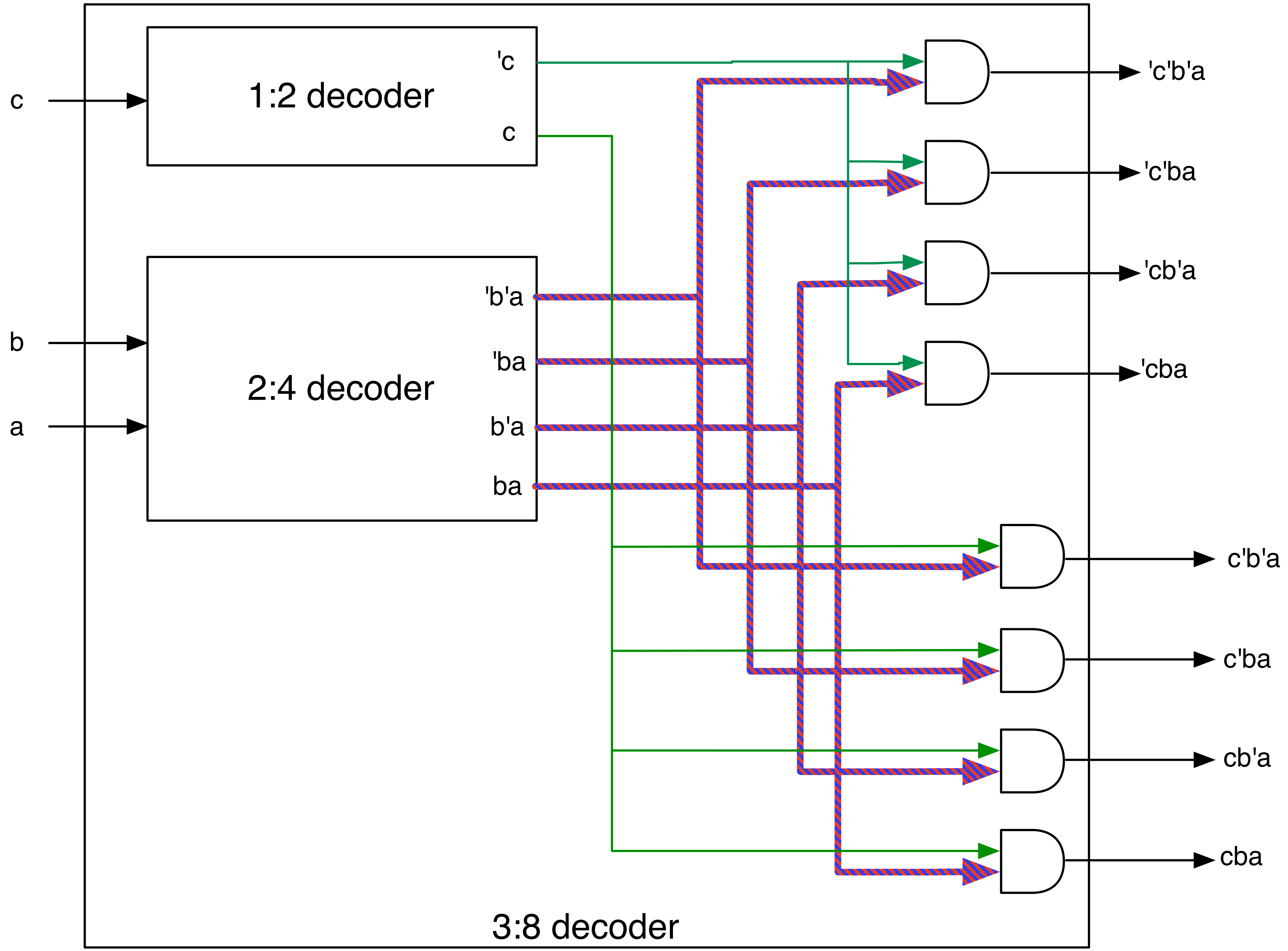
Internal design of 1:2 decoder



Hierarchical design of decoder (2:4 decoder)



Hierarchical design of decoder (3:8 decoder)



Encoders

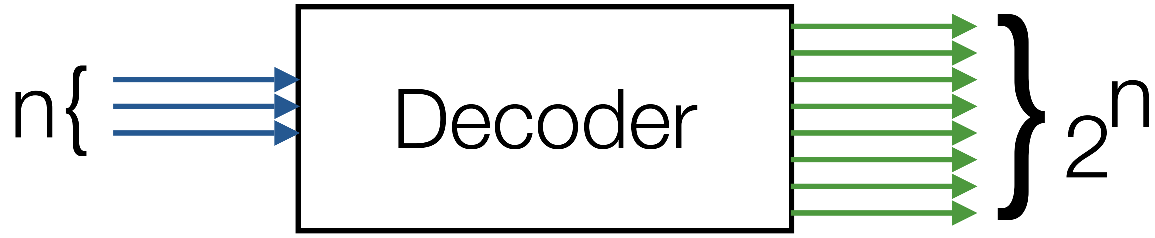
T 3-7

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

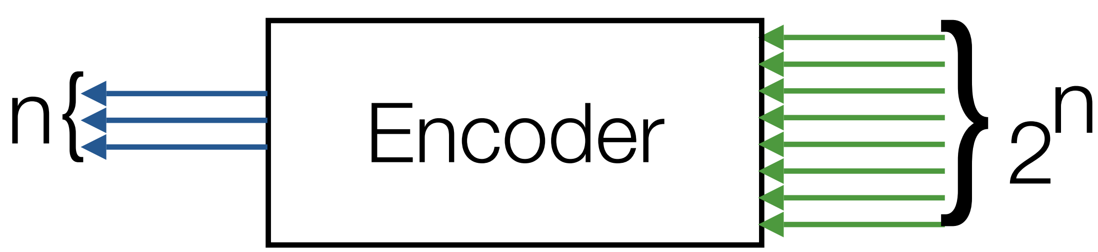
TABLE 3-7
Truth Table for Octal-to-Binary Encoder

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 be just one 1, the rest 0's

Priority Encoder

T 3-8

Like a regular encoder, but designed for any combination of inputs.

TABLE 3-8
Truth Table of Priority Encoder

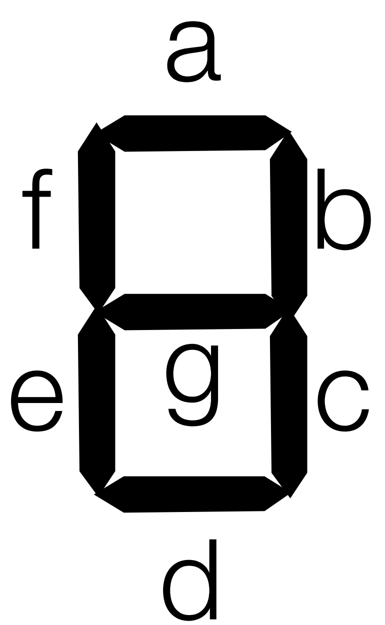
Inputs				Outputs		
D_3	D_2	D_1	D_0	A_1	A_0	V
0	0	0	0	X	X	0
0	0	0	1	0	0	1
0	0	1	X	0	1	1
0	1	X	X	1	0	1
1	X	X	X	1	1	1

General code conversion

3-3

(a) Segment designation (b) Numeric designation for display

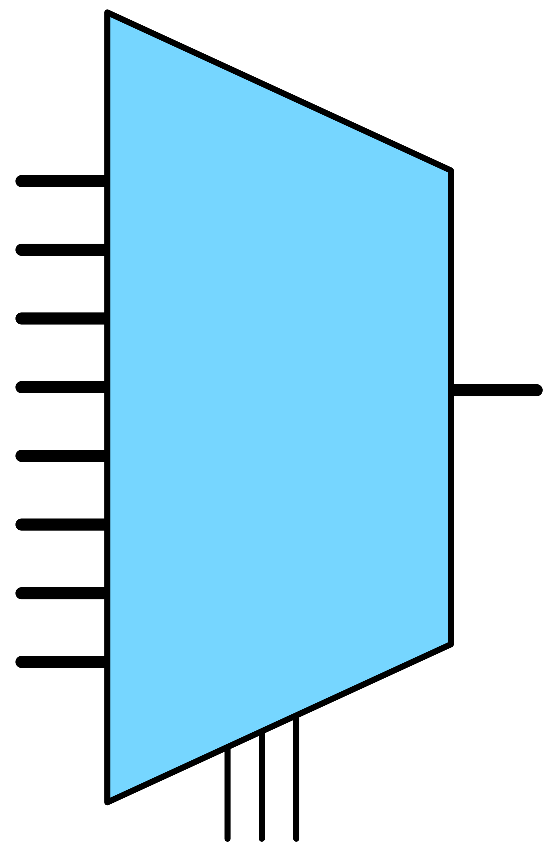
Code conversion for the "a"



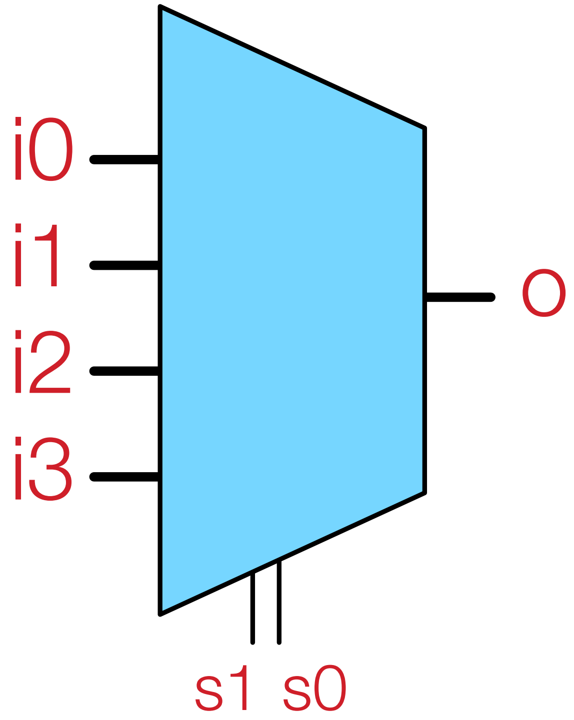
	Input				Output						
Val	W	X	Y	Z	a	b	c	d	e	f	g
0	0	0	0	0	1	1	1	1	1	1	0
1	0	0	0	1	0	1	1	0	0	0	0
2	0	0	1	0	1	1	0	1	1	0	1
3	0	0	1	1	1	1	1	1	0	0	1
4	0	1	0	0	0	1	1	0	0	1	1
5	0	1	0	1	1	0	1	1	0	1	1
6	0	1	1	0	1	0	1	1	1	1	1
7	0	1	1	1	1	1	1	0	0	0	0
8	1	0	0	0	1	1	1	1	1	1	1
9	1	0	0	1	1	1	1	0	0	1	1
A	1	0	1	0	1	1	1	0	1	1	1
b	1	0	1	1	0	0	1	1	1	1	1
C	1	1	0	0	1	0	0	1	1	1	0
d	1	1	0	1	0	1	1	1	1	0	1
E	1	1	1	0	1	0	0	1	1	1	1
F	1	1	1	1	1	0	0	0	1	1	1

Multiplexers

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



Truth table for a 4:1 mux

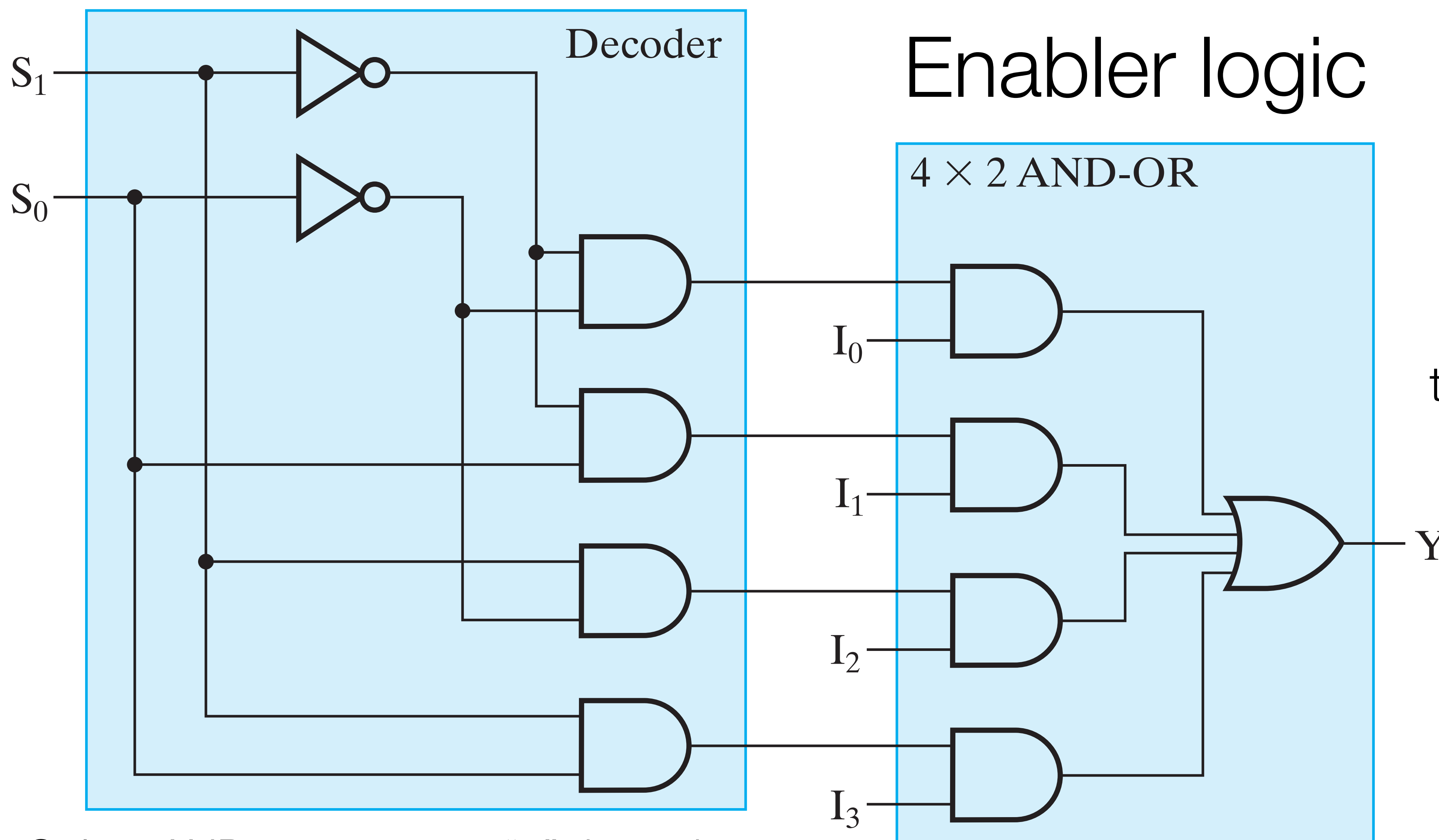


Internal mux organization

3-26

Selector Logic

Enabler logic



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

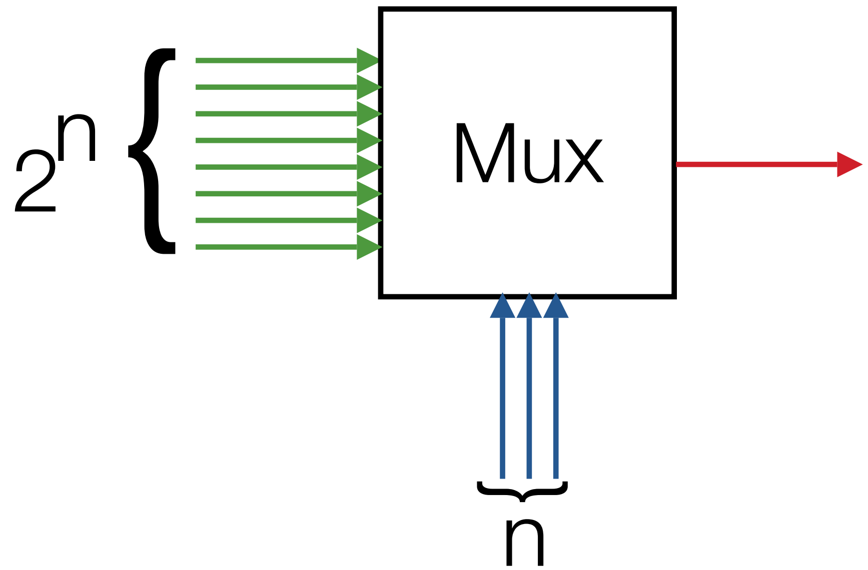
Only 1 AND gate passes "1" through

AND gates "zero out" unselected I_i

In class exercise

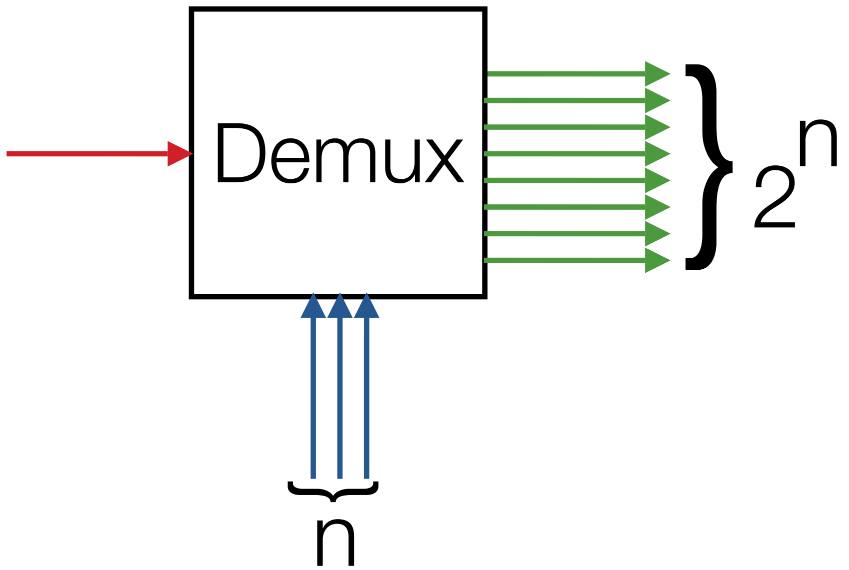
How would you implement an 8:1 mux using two 4:1 muxes?

Multiplexer truth table



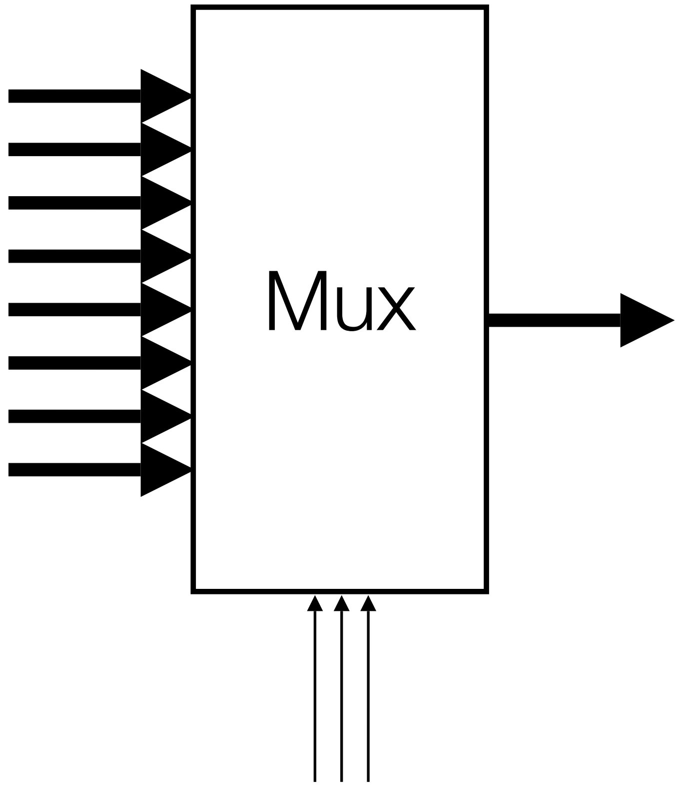
2^n inputs								n-bit BCD value			1 output
a	x	x	x	x	x	x	x	0	0	0	a
x	b	x	x	x	x	x	x	0	0	1	b
x	x	c	x	x	x	x	x	0	1	0	c
x	x	x	d	x	x	x	x	0	1	1	d
x	x	x	x	e	x	x	x	1	0	0	e
x	x	x	x	x	f	x	x	1	0	1	f
x	x	x	x	x	x	g	x	1	1	0	g
x	x	x	x	x	x	x	h	1	1	1	h

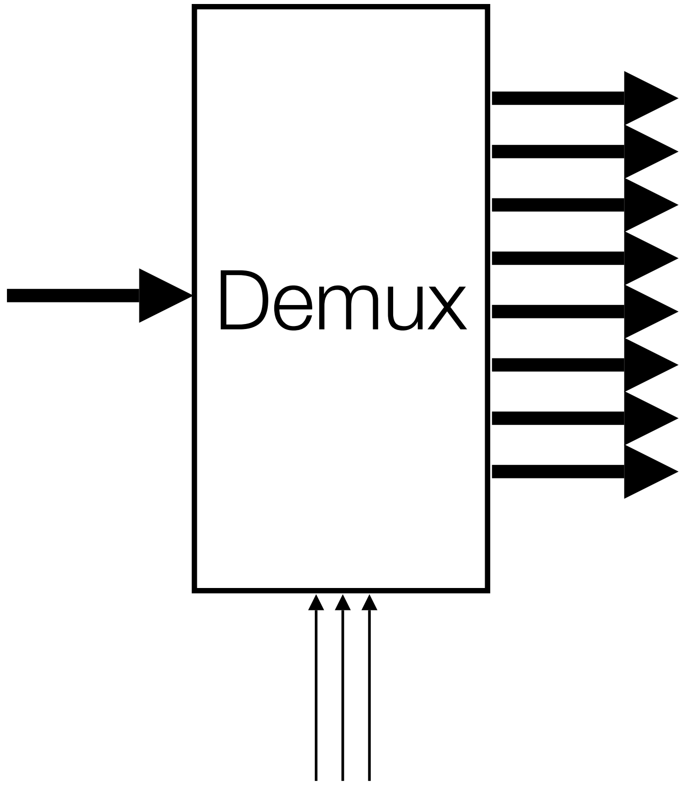
Demultiplexers



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”



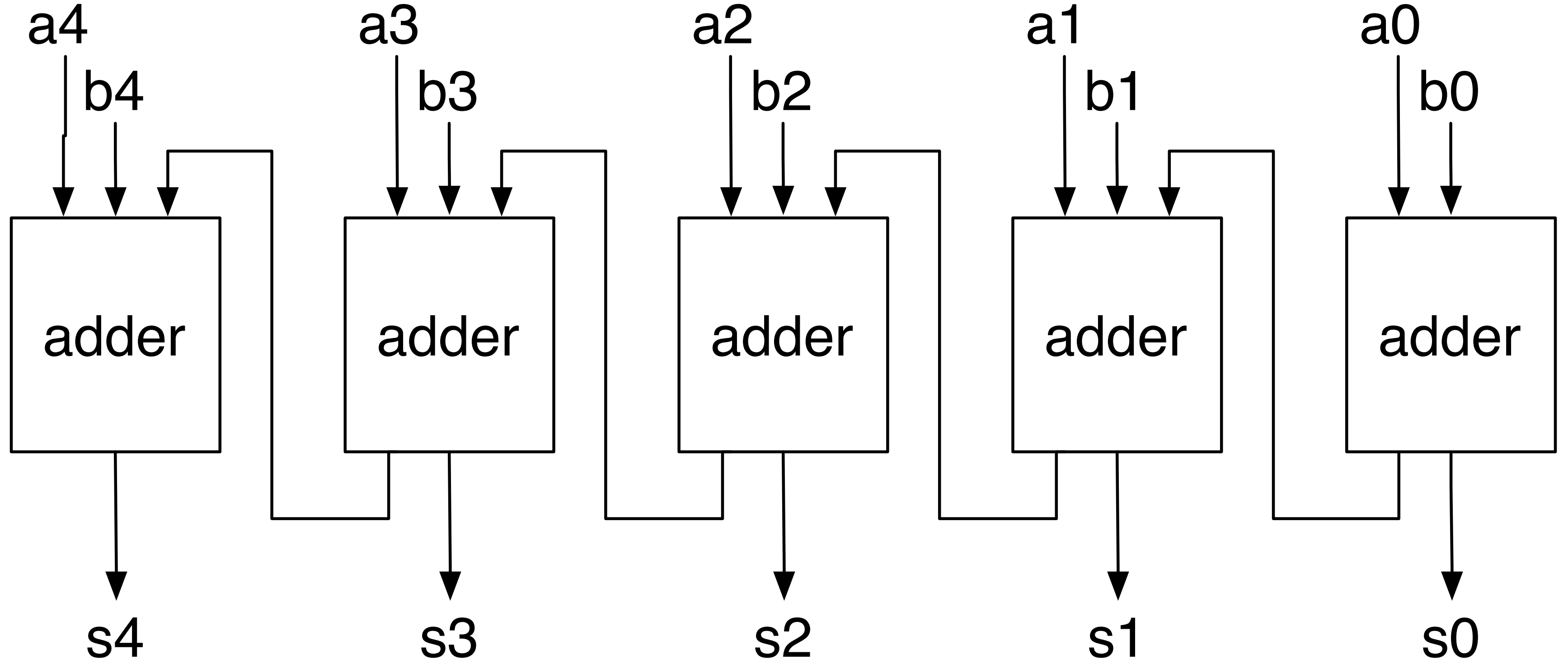


“merge”

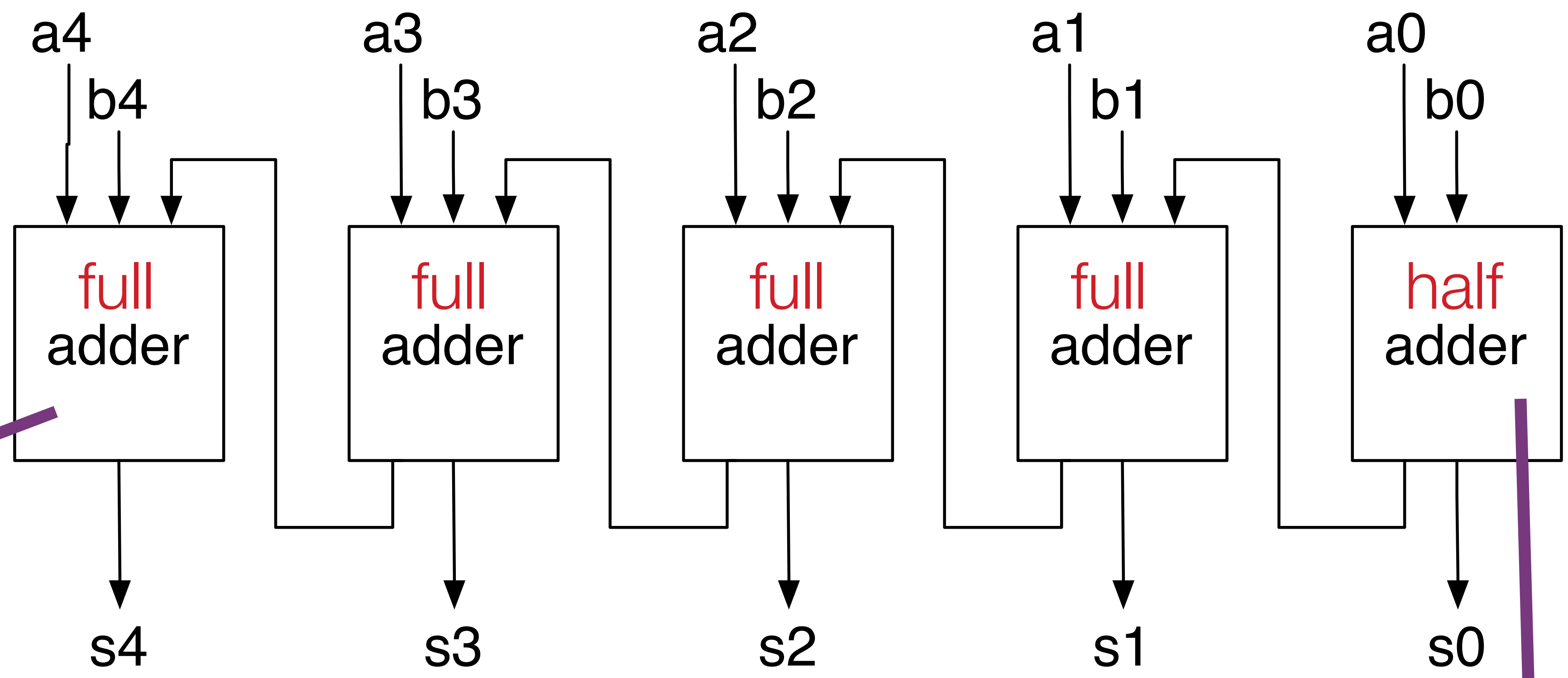
“fork”

Decimal v. binary addition

	Decimal	Binary
carries	1 1	1 1 1 1
	4 3 5 8 2	0 1 0 1 1
+	2 2 5 7 3	0 0 1 0 1
sum	6 6 1 5 5	1 0 0 0 0



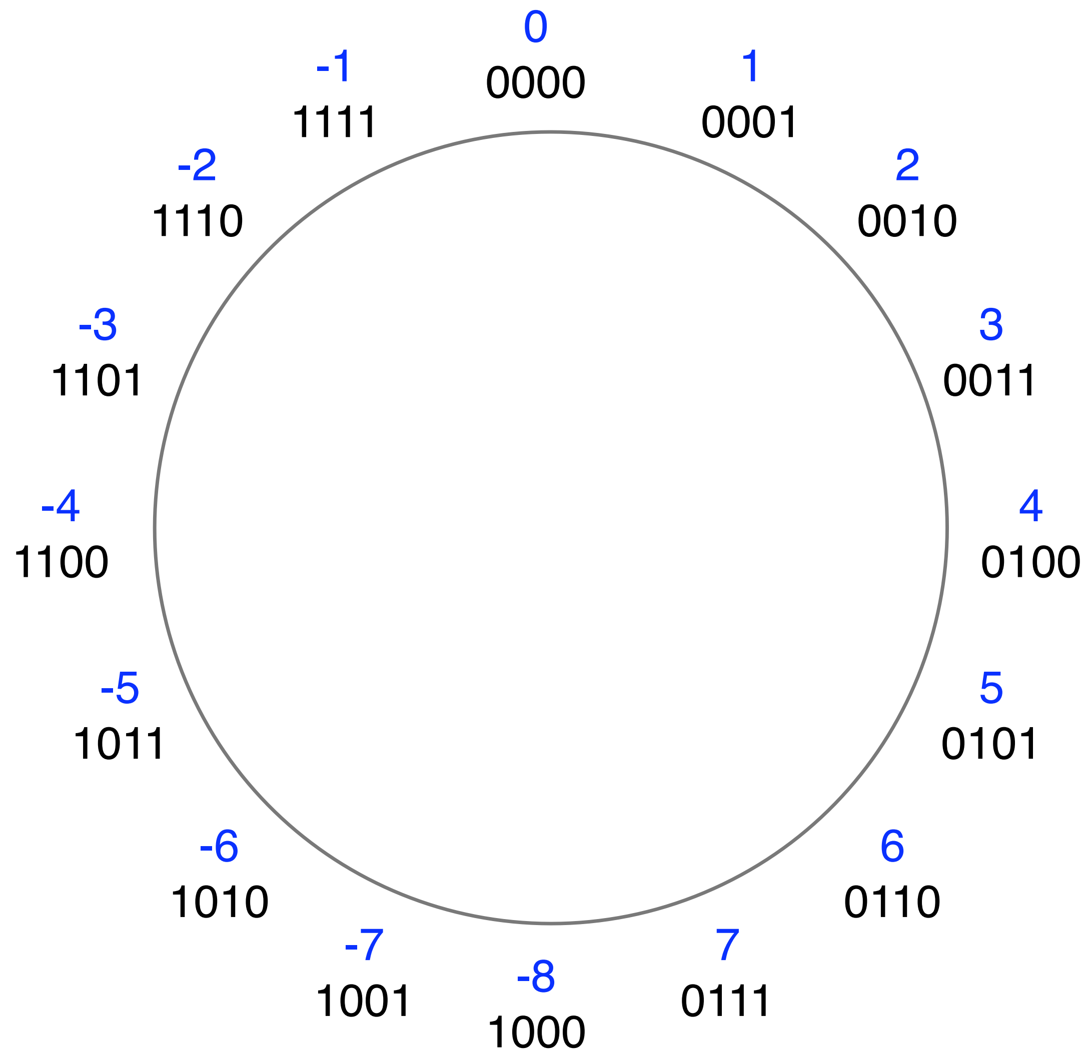
Ripple carry adder



a	b	cin	cout	s
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

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

Subtraction w. twos complement representation

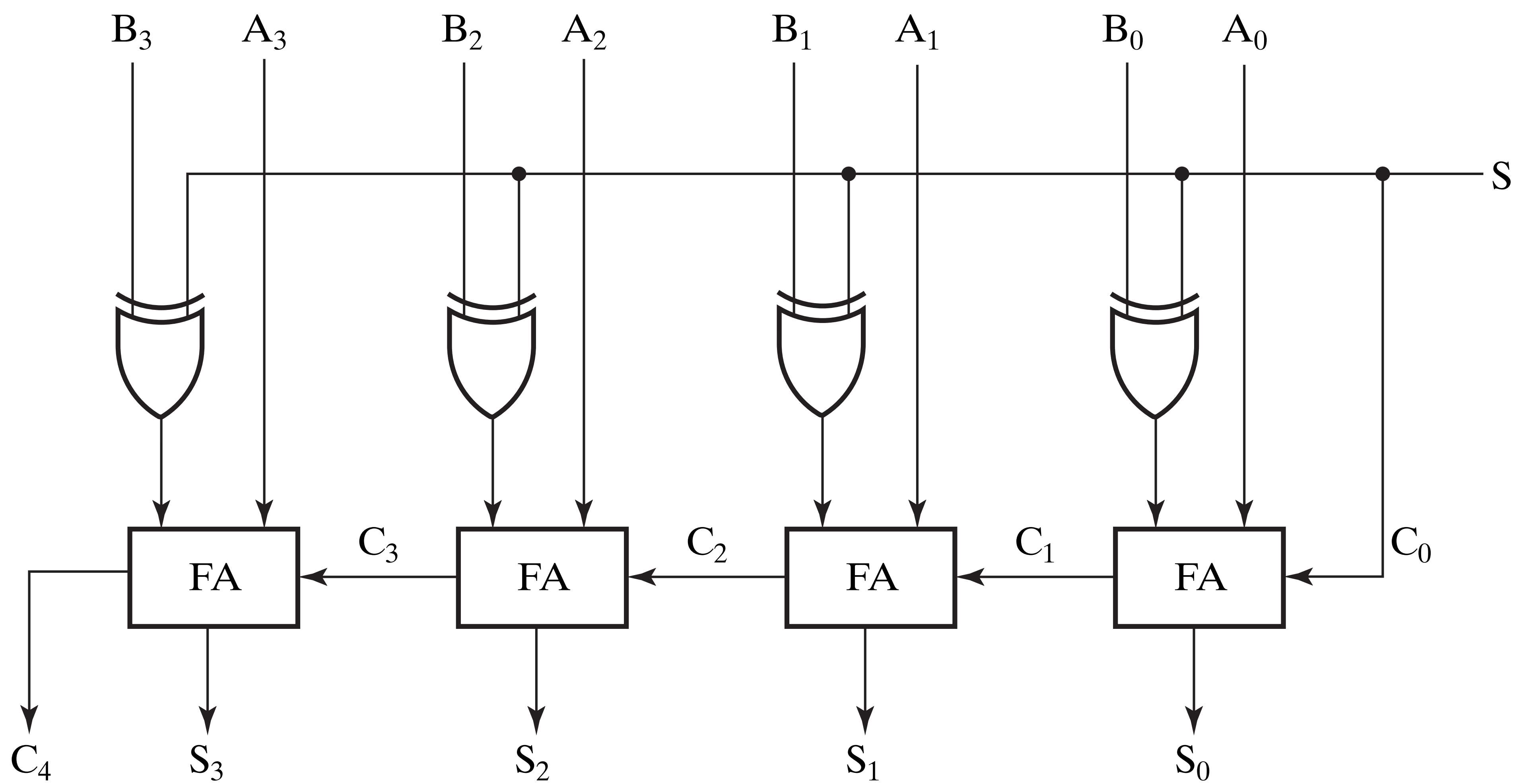


Can be accomplished with a **twos-complementor** and an **adder**

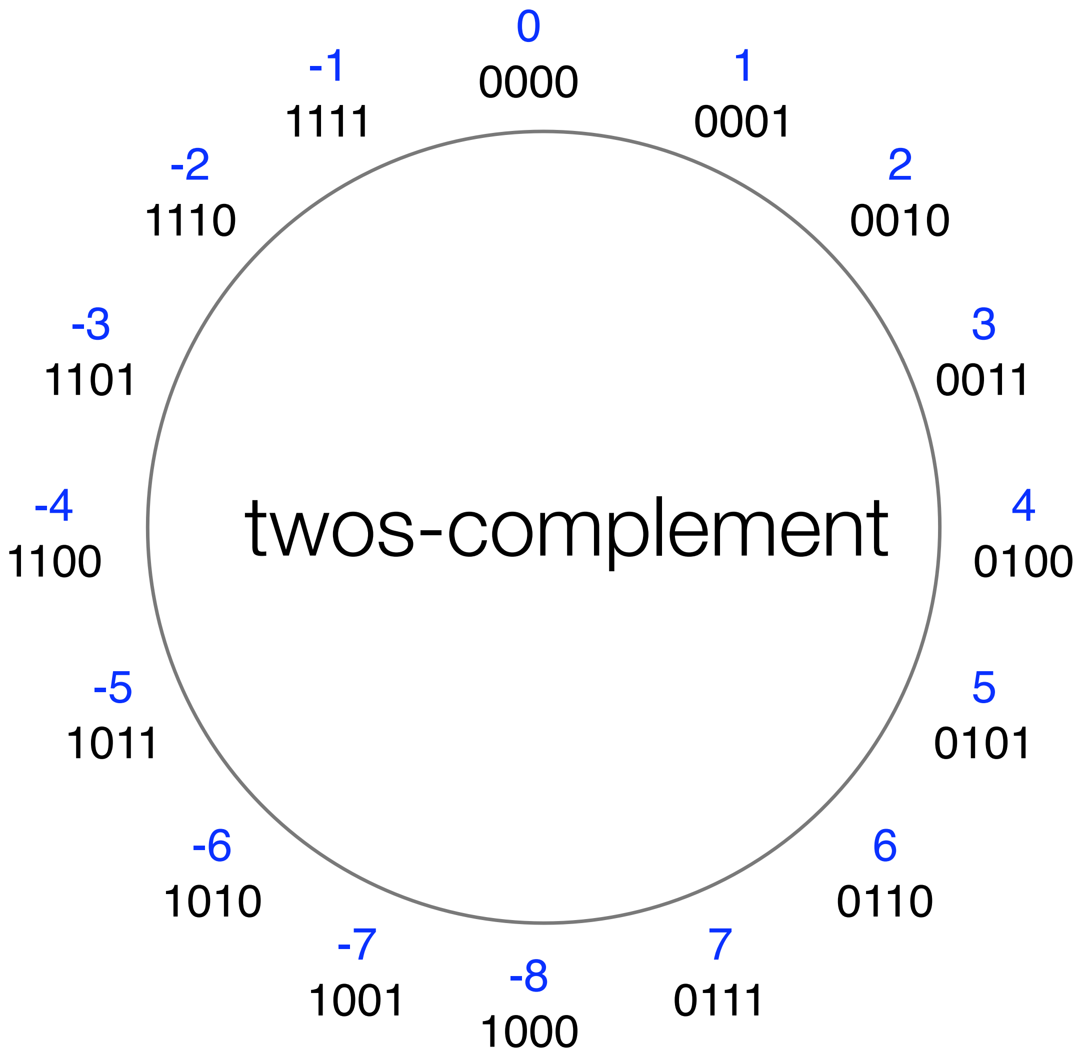
In class exercise: designing an adder-subtractor

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

4-7



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)
```

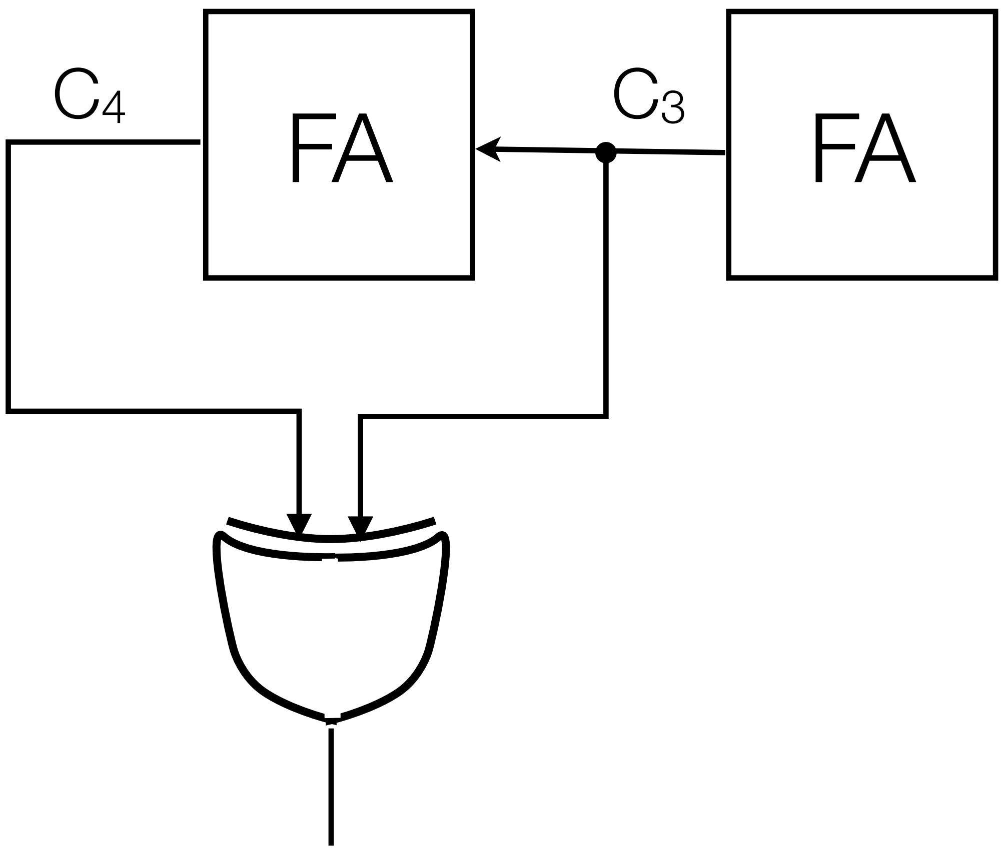
Handling overflow

```
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	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	

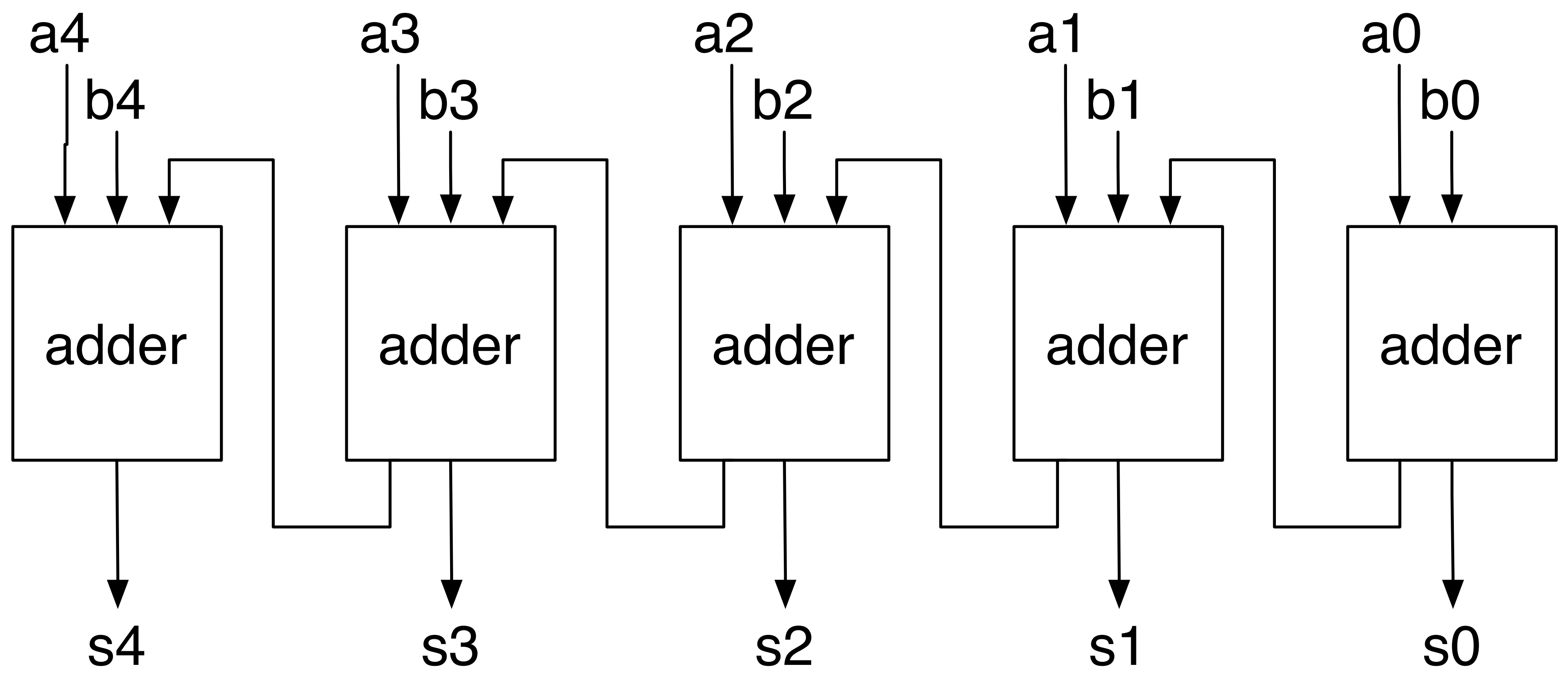
Overflow computation in adder/subtractor

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



Ripple carry adder delay analysis

- Assume unit delay for all gates
  - S = A ⊕ B ⊕ Cin
    - [S ready _______ units after A,B and Cin ready]
  - Cout = AB + ACin + BCin
    - [Cout ready _______ units after A,B and Cin ready]



Carry lookahead adder (CLA)

- Goal: produce an adder of less 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

$$
\begin{aligned}
c_1 &= g_0 + c_0p_0 \\
c_2 &= g_1 + c_1p_1 \\
&= g_1 + (g_0 + c_0p_0)p_1 \\
&= g_1 + g_0p_1 + c_0p_0p_1 \\
c_3 &= g_2 + c_2p_2 \\
&= g_2 + (g_1 + g_0p_1 + c_0p_0p_1)p_2 \\
&= g_2 + g_1p_2 + g_0p_1p_2 + c_0p_0p_1p_2
\end{aligned}
$$

- 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)

$c_0 = 0$

$c_1 = g_0 + c_0p_0$

$c_2 = g_1 + g_0p_1 + c_0p_0p_1$

$c_3 = g_2 + g_1p_2 + g_0p_1p_2 + c_0p_0p_1p_2$

$s_0 = a_0 \oplus b_0 \oplus c_0$

$s_1 = a_1 \oplus b_1 \oplus c_1$

$s_2 = a_2 \oplus b_2 \oplus c_2$

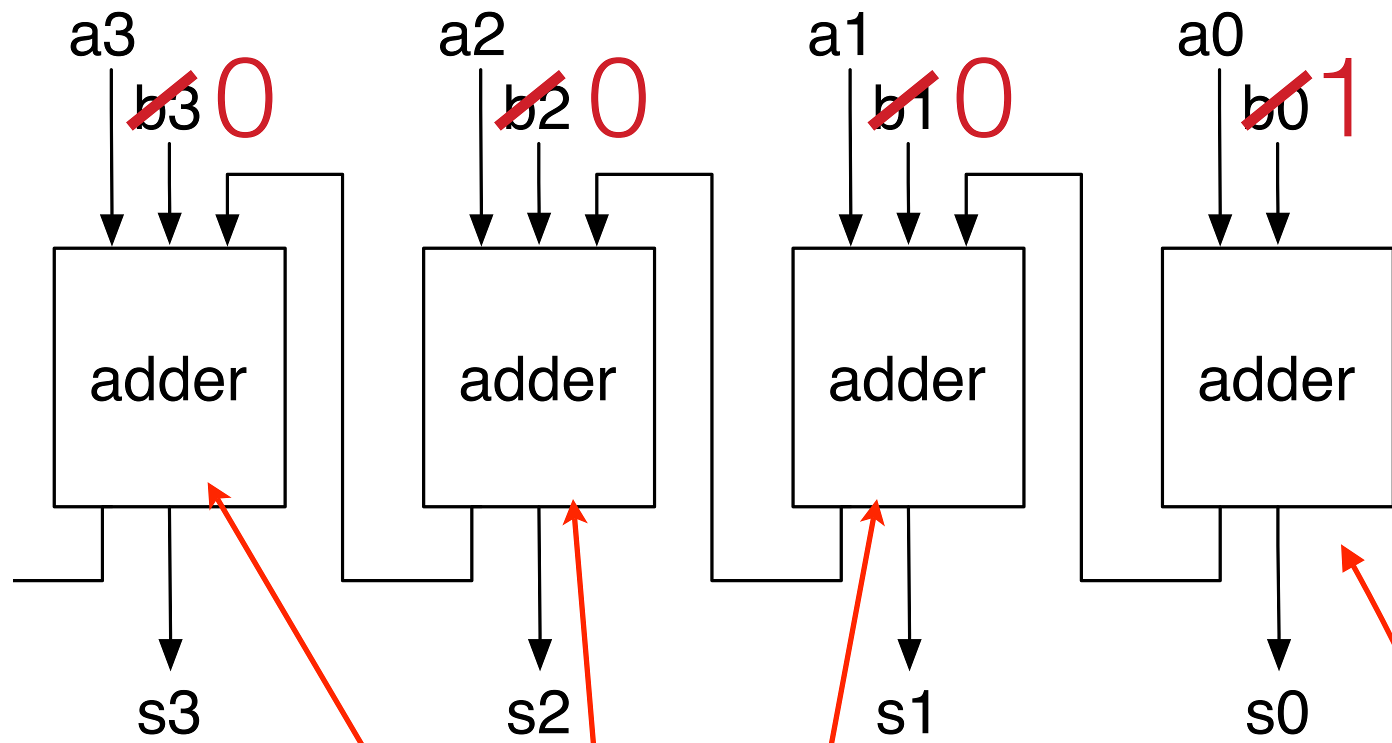
$s_3 = a_3 \oplus b_3 \oplus c_3$

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?



Can be reduced to half-adders

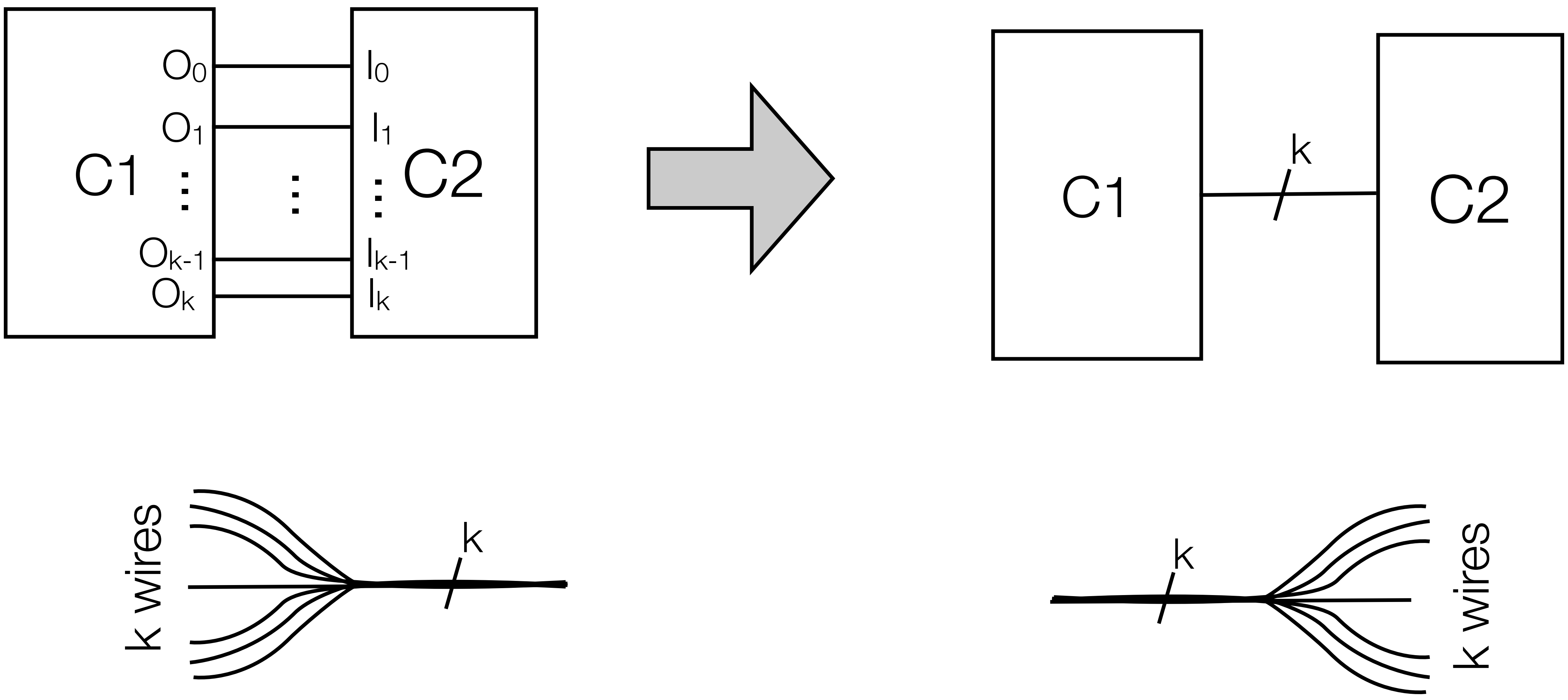
Incrementer circuit

a_0	S	C
0	1	0
1	0	1

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

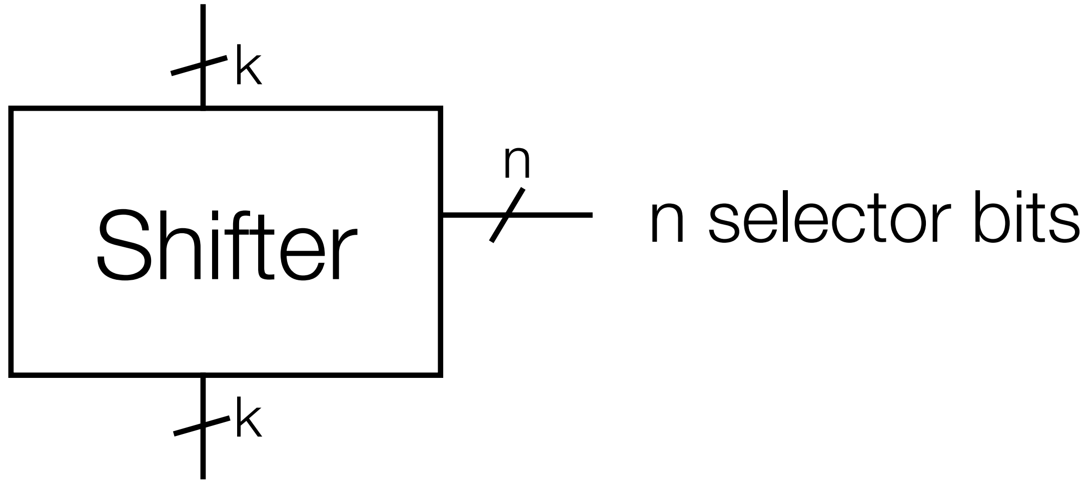
Multi-wire notation

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



Shifter Circuit

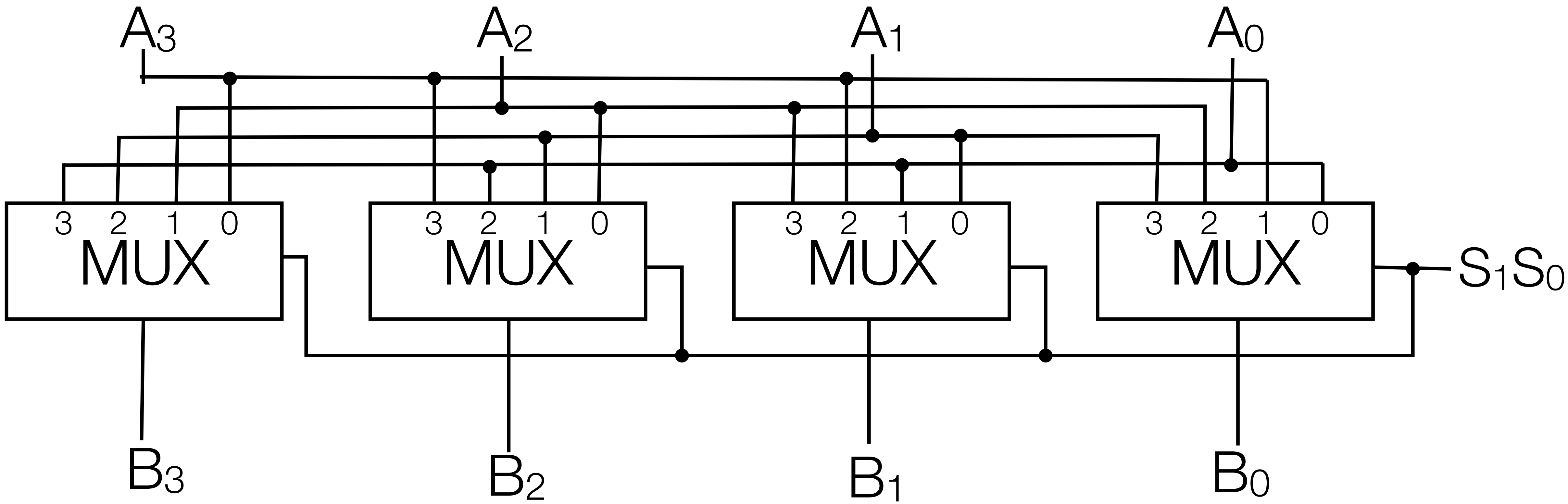
- Shifts bits of a word: $A_{k-1}A_{k-2}...A_2A_1A_0$



$B_{k-1}B_{k-2}...B_2B_1B_0$

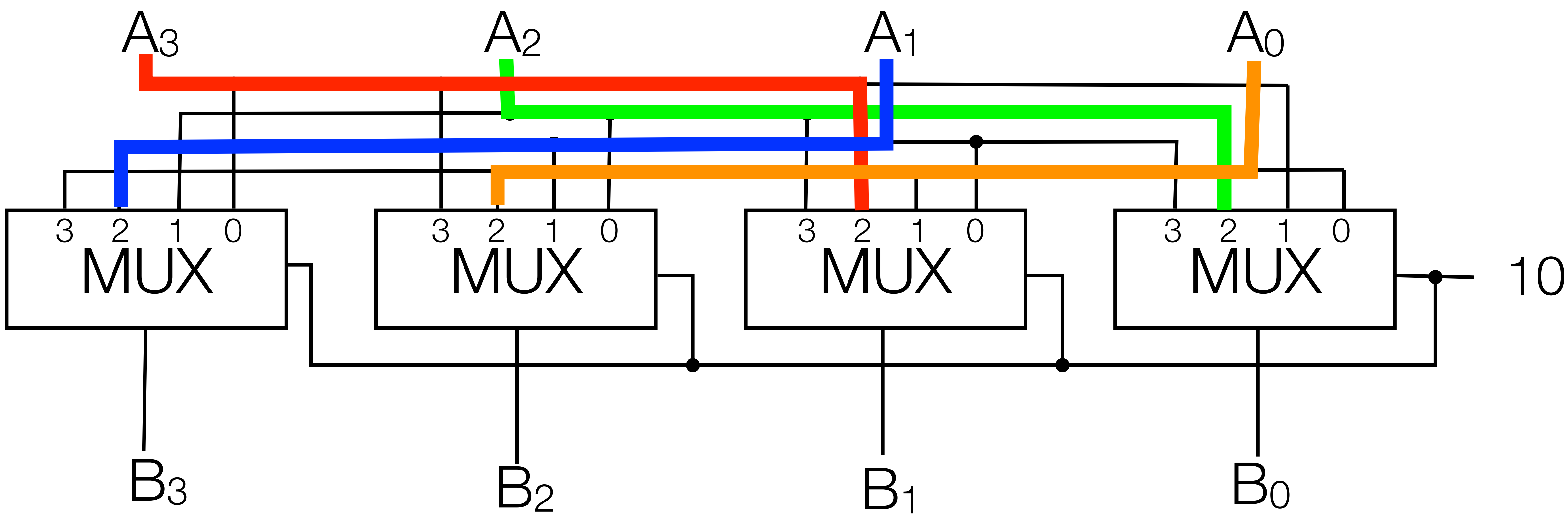
- Various types of shifters
  - Barrel: selector bits indicate (in binary) how “far” bits shift
    - selector value = j, then $B_i = A_{i-j}$
    - bits can “wraparound” $B_i \pmod{2^n} = A_{i-j} \pmod{2^n}$ or rollout (B_i=0 for i<j)
  - L/R with enable: n=2, high bit enables, low bit indicates direction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

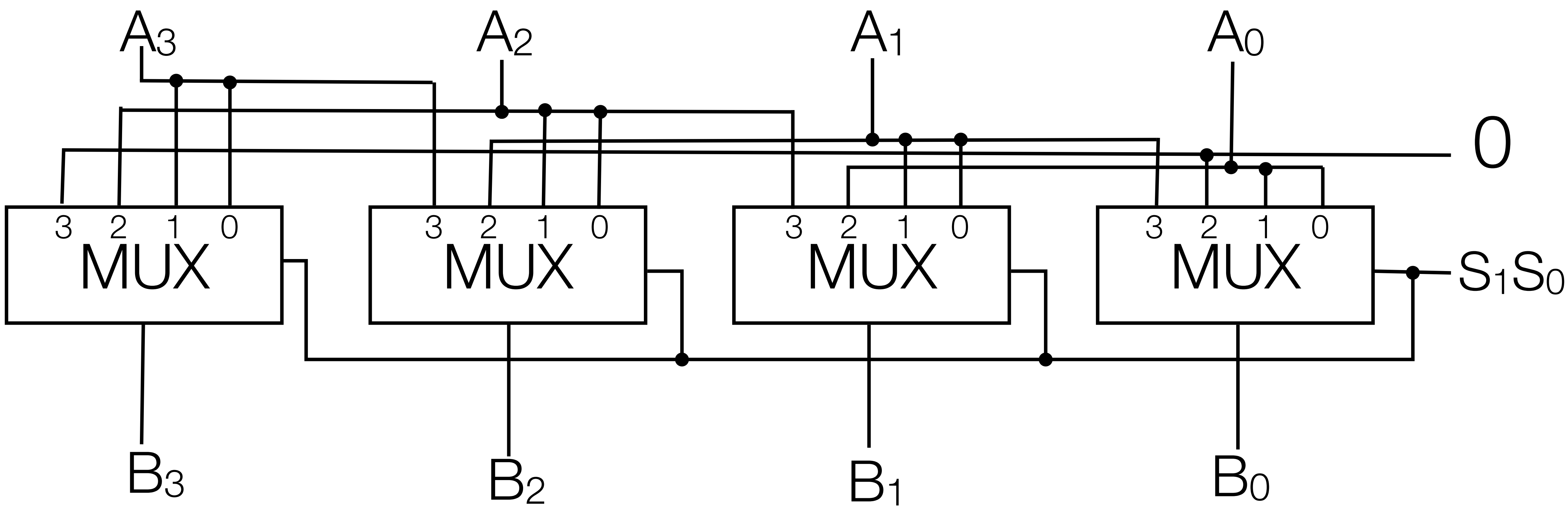
Barrel Shifter Design with wraparound (using MUXs)



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

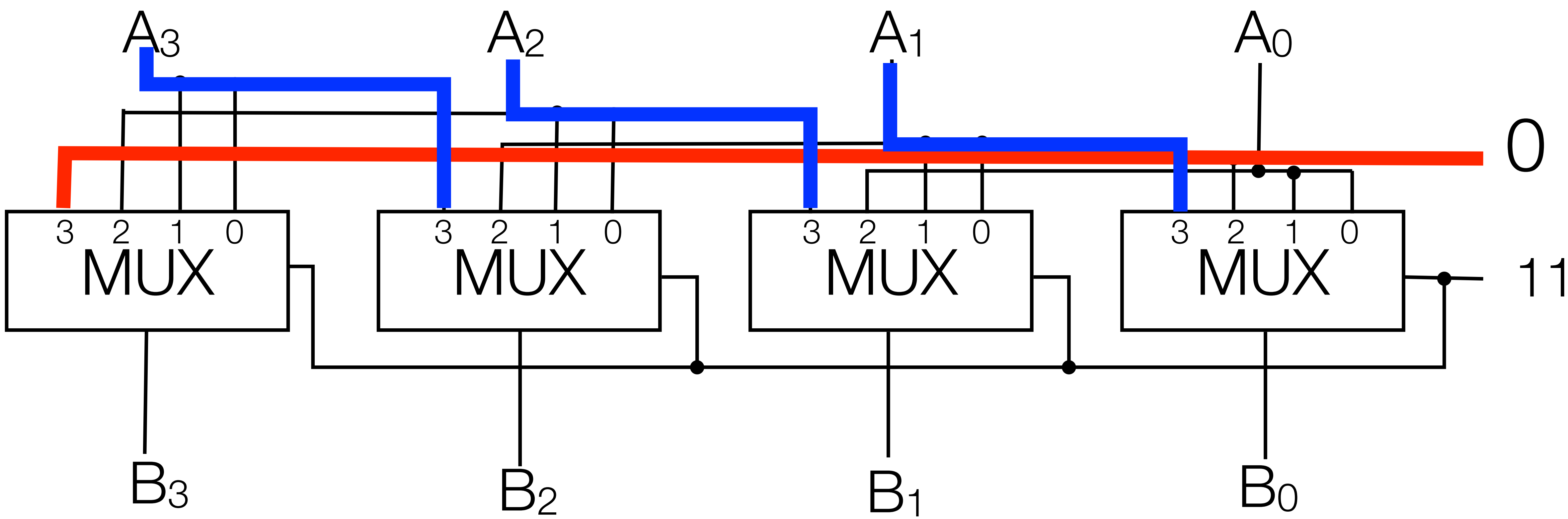
- Selector is 10 (i.e., 2 binary): each MUX entry 2 is selected

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)