# CSEE 3827: Fundamentals of Computer Systems

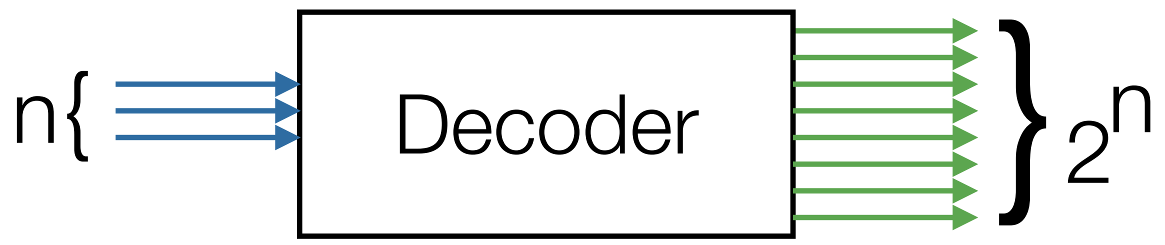
Lecture 7

February 11, 2009

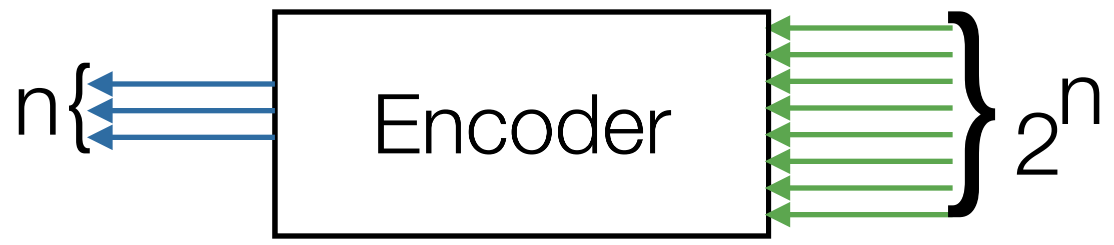
Martha Kim
martha@cs.columbia.edu

## Decoders and encoders

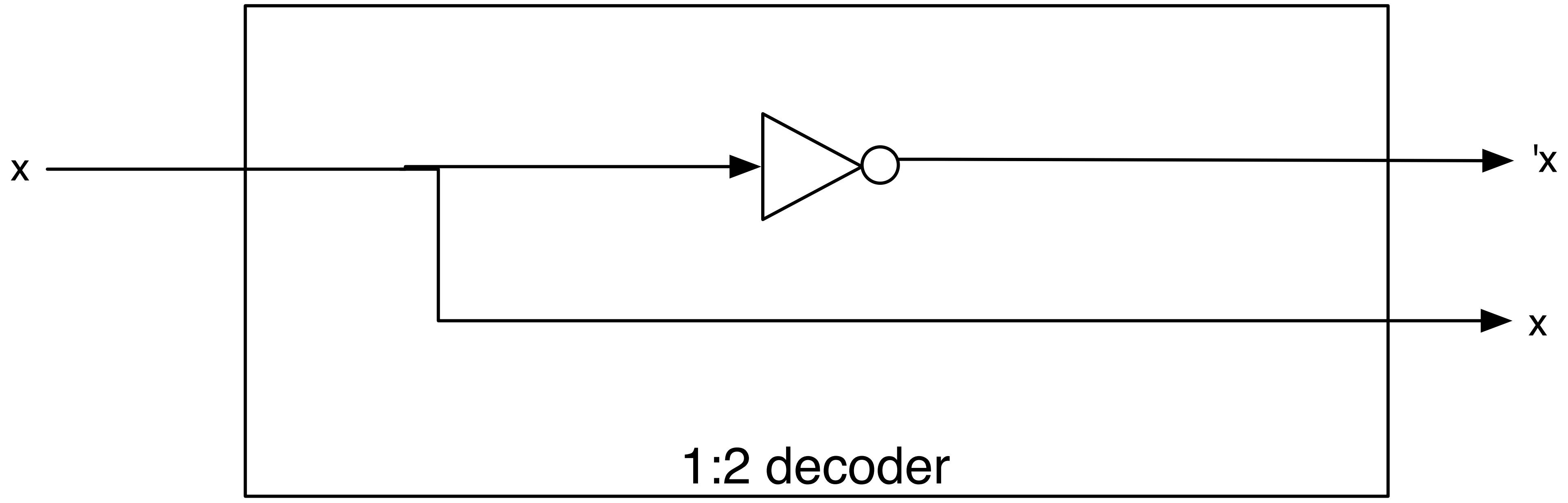
# Decoders and encoders



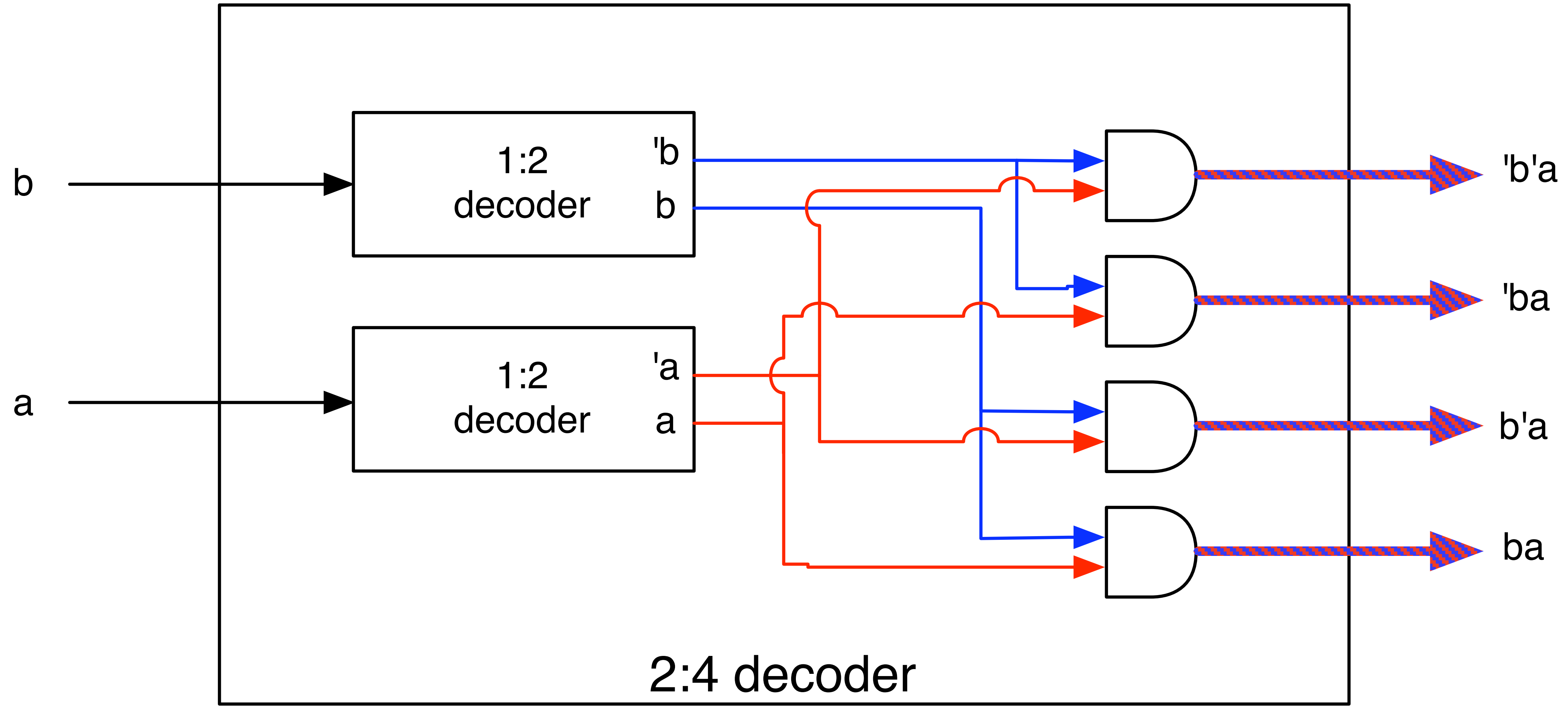
| BCD values | | | One-hot encoding | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0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



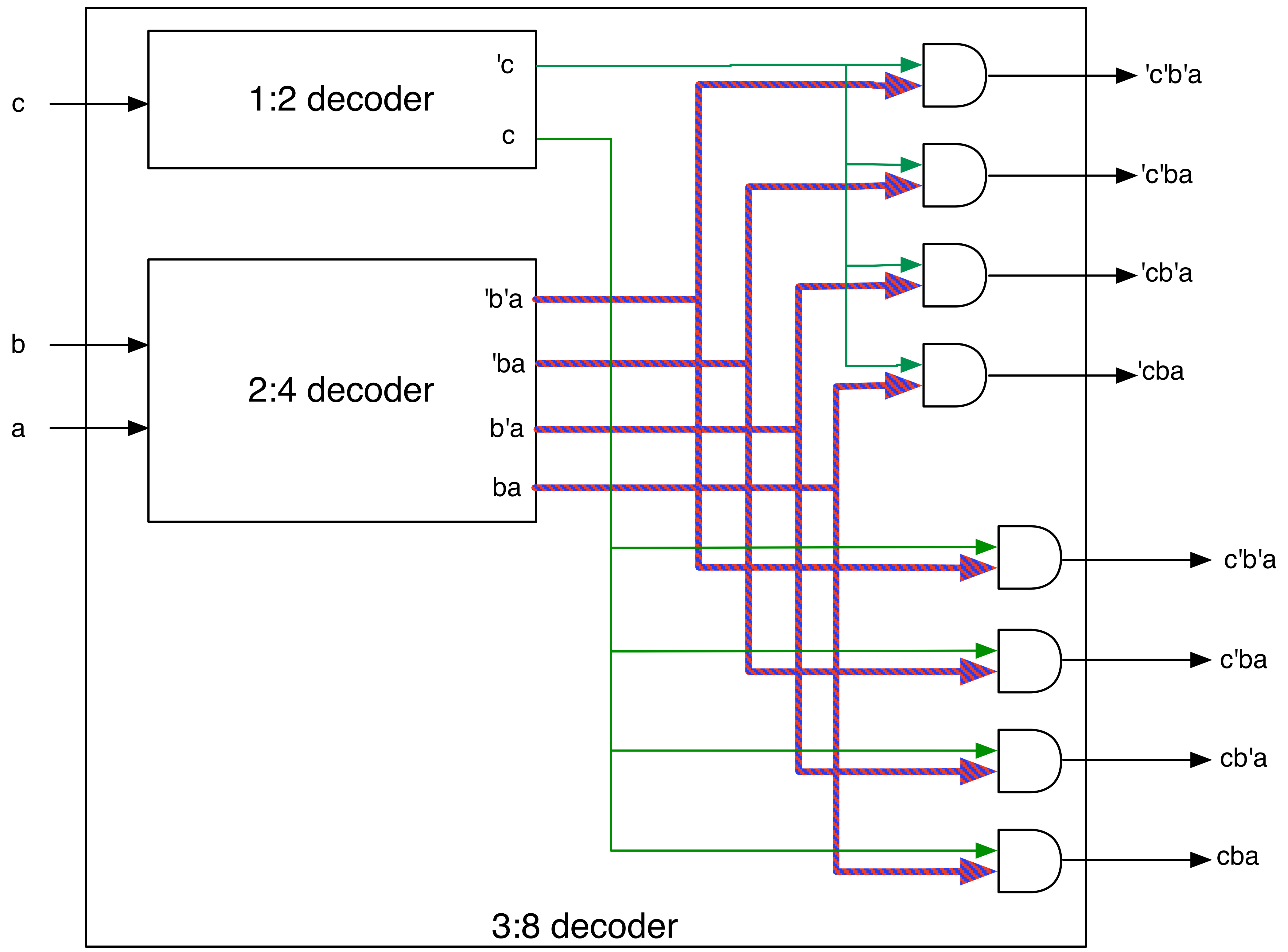
# Internal design of 1:2 decoder



## Hierarchical design of decoder (2:4 decoder)

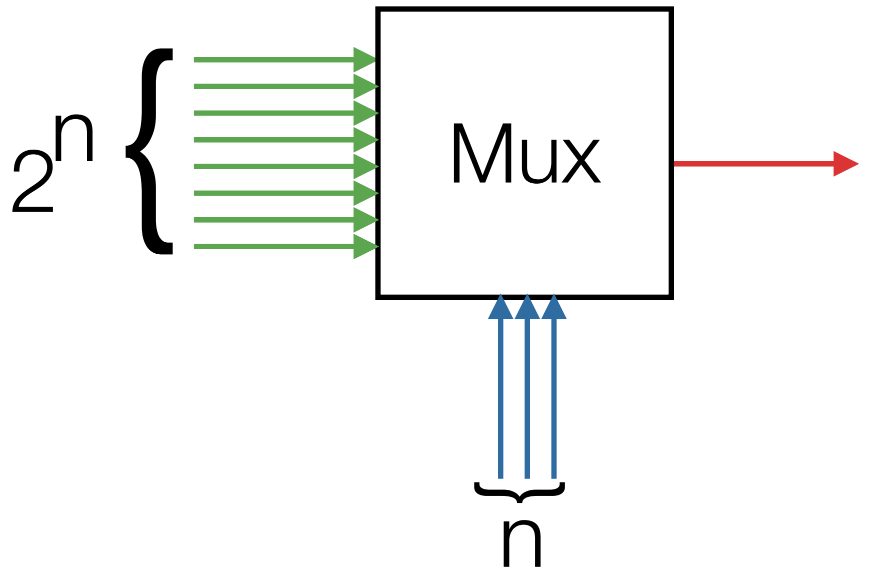


# Hierarchical design of decoder (3:8 decoder)



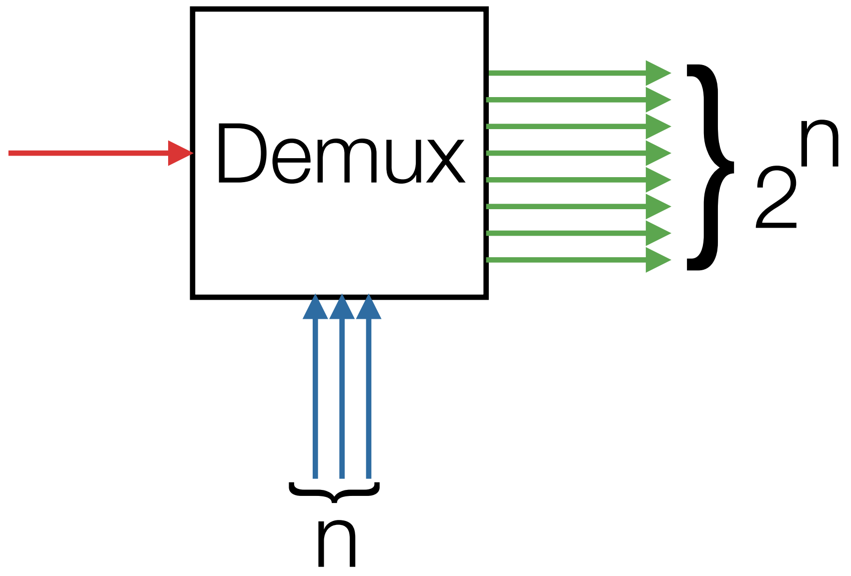
## Multiplexers and demultiplexers

# Multiplexers & Demultiplexers



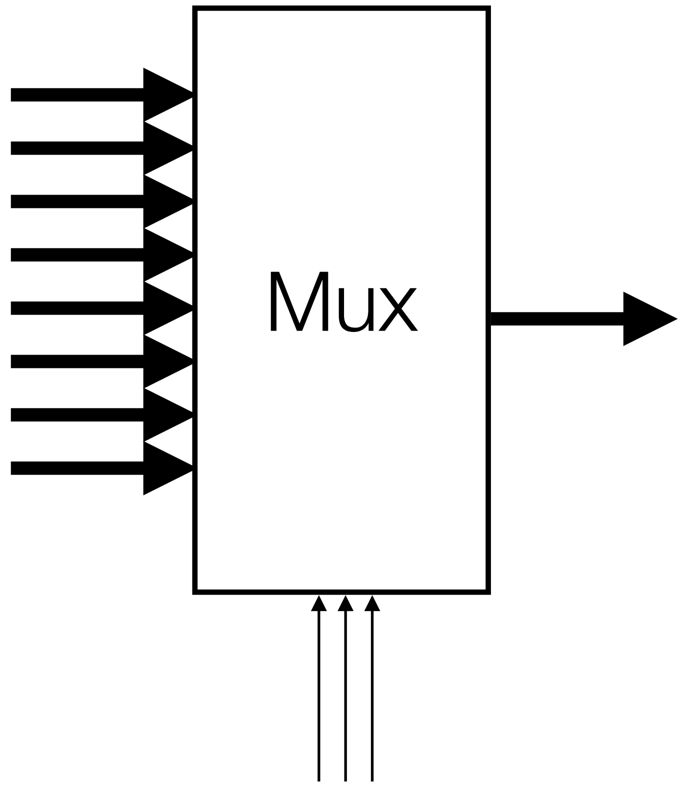
| 2^n inputs | | | | | | | | n-bit BCD value | | | 1 output |
|---|---|---|---|---|---|---|---|---|---|---|---|
| a | x | x | x | x | x | x | x | 0 | 0 | 0 | a |
| x | b | x | x | x | x | x | x | 0 | 0 | 1 | b |
| x | x | c | x | x | x | x | x | 0 | 1 | 0 | c |
| x | x | x | d | x | x | x | x | 0 | 1 | 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

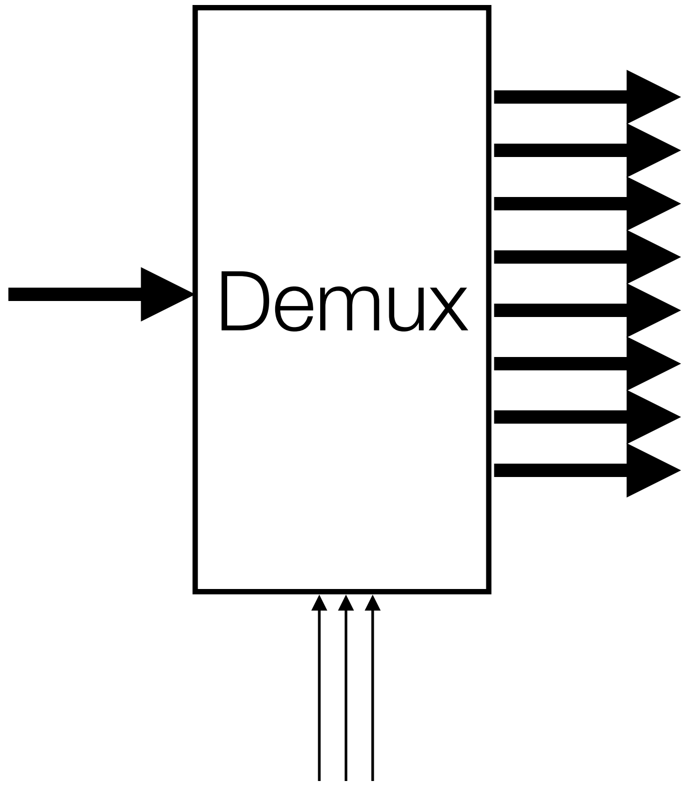


| 1 input | n-bit BCD value | | | 2^n outputs | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| a | 0 | 0 | 0 | a | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| b | 0 | 0 | 1 | 0 | b | 0 | 0 | 0 | 0 | 0 | 0 |
| c | 0 | 1 | 0 | 0 | 0 | c | 0 | 0 | 0 | 0 | 0 |
| d | 0 | 1 | 1 | 0 | 0 | 0 | d | 0 | 0 | 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”

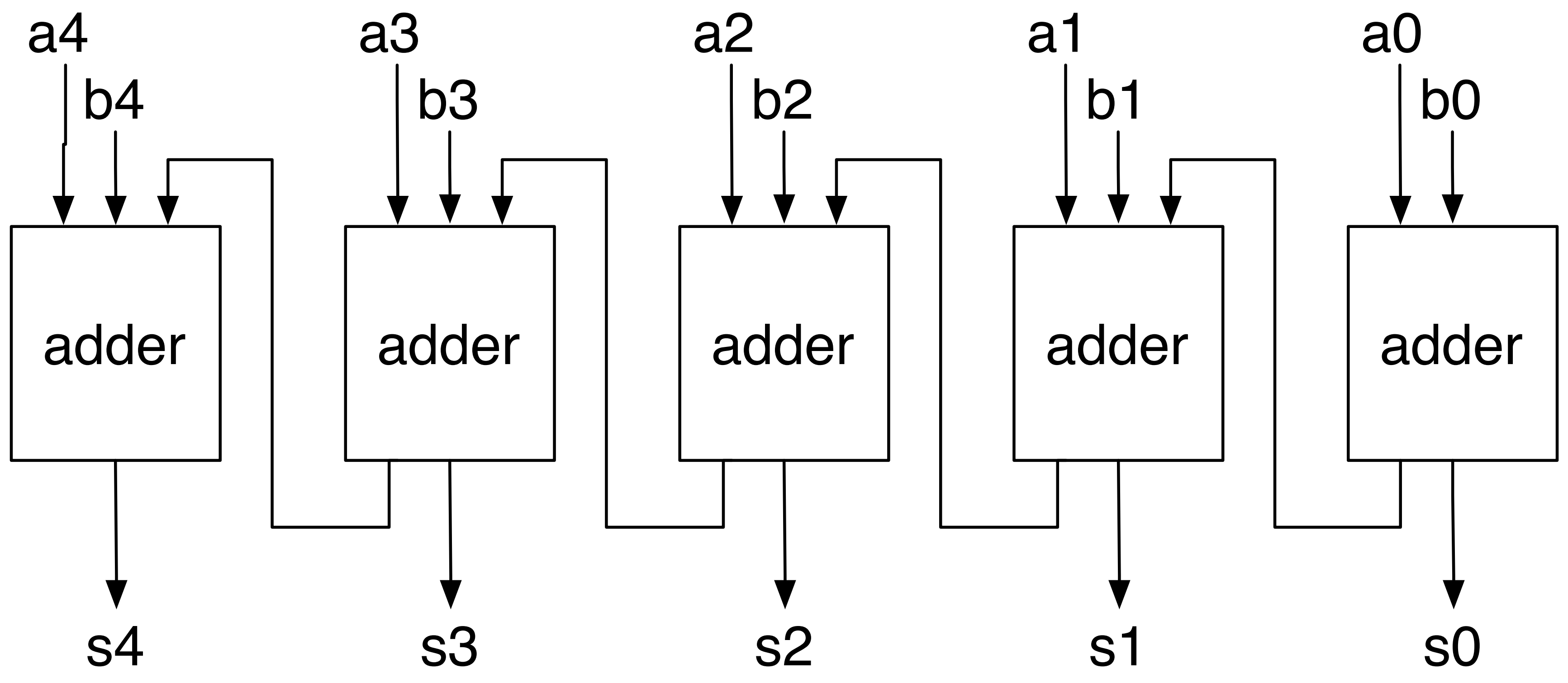
# Mini-homework

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

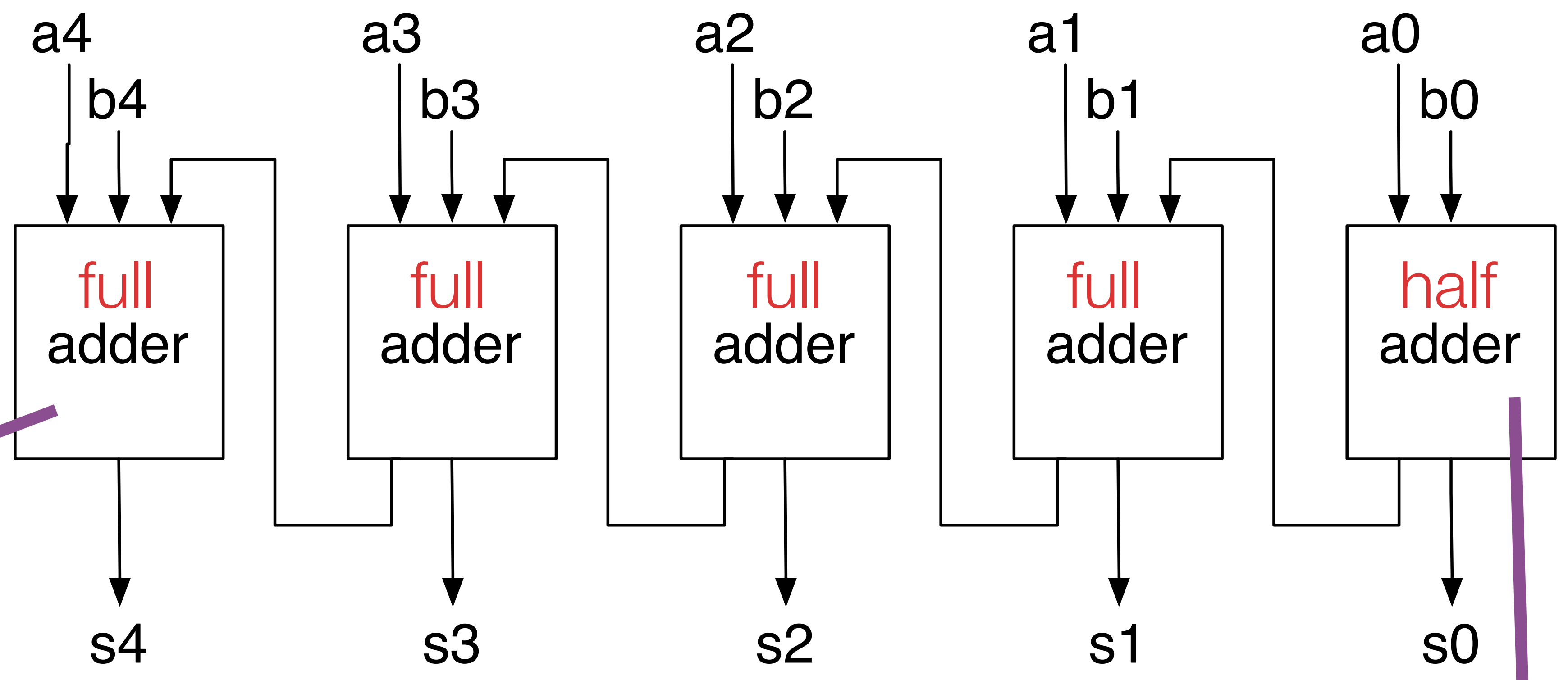
## Binary addition / Ripple carry adder

# Decimal v. binary addition

$$\begin{array}{rccccc}
 & & 1 & 1 & & \\
 & 4 & 3 & 5 & 8 & 2 \\
+ & 2 & 2 & 5 & 7 & 3 \\
\hline
 & 6 & 6 & 1 & 5 & 5
\end{array}
\qquad\qquad
\begin{array}{rccccc}
 & 1 & 1 & 1 & 1 & \\
 & 0 & 1 & 0 & 1 & 1 \\
+ & 0 & 0 & 1 & 0 & 1 \\
\hline
 & 1 & 0 & 0 & 0 & 0
\end{array}$$



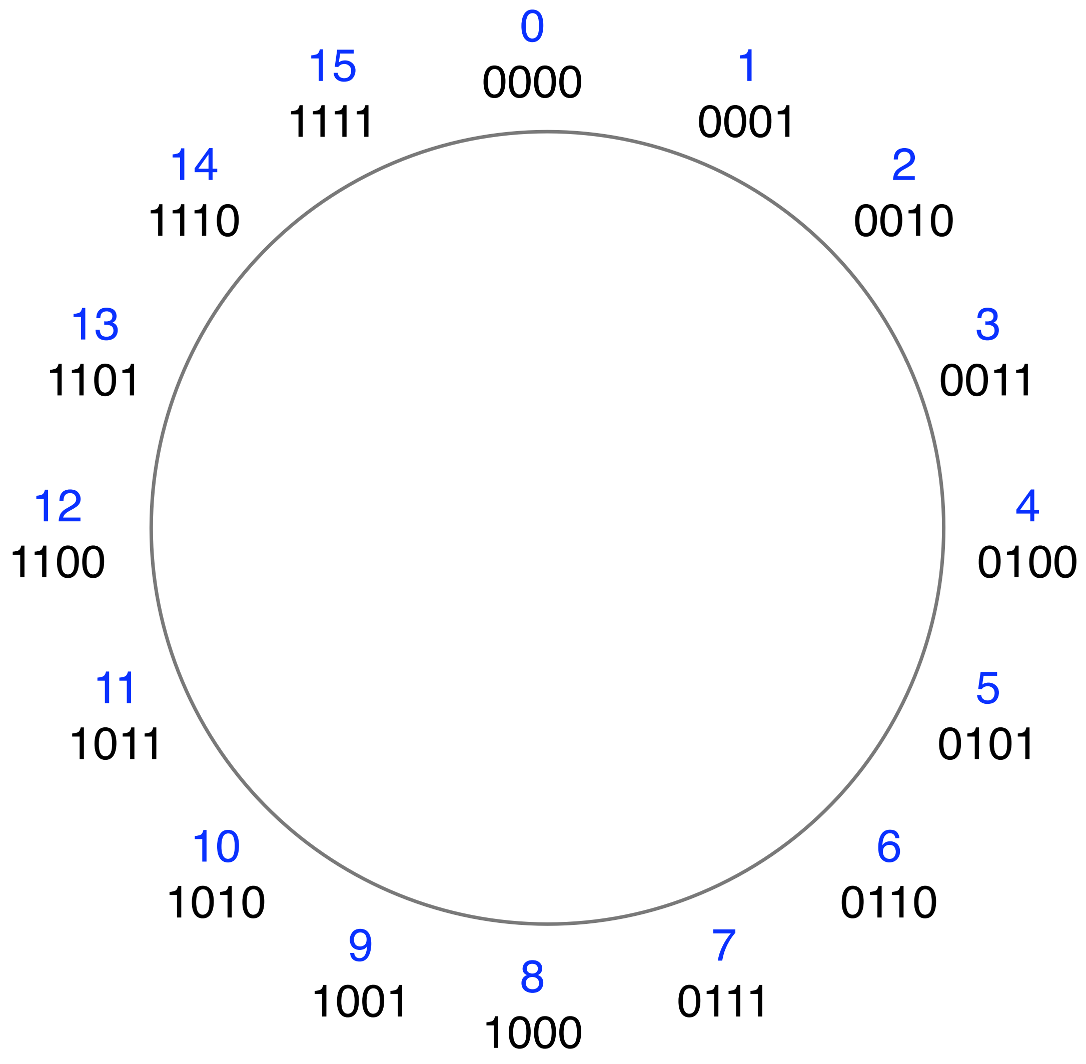
## Ripple carry adder



| 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 | | |

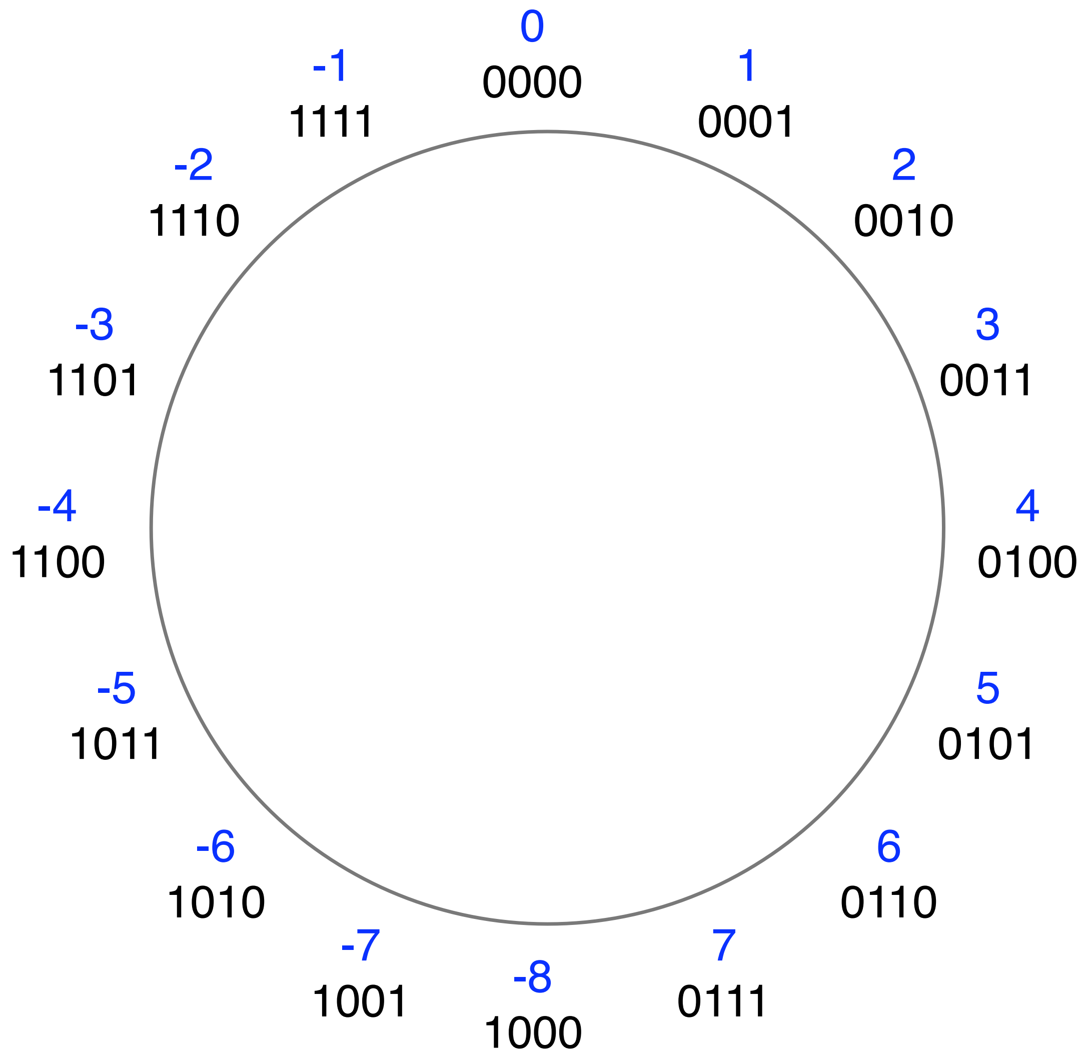
| a | b | c | s |
|---|---|---|---|
| 0 | 0 | | |
| 0 | 1 | | |
| 1 | 0 | | |
| 1 | 1 | | |

## Subtraction



Can be accomplished with a **comparator** and a **subtractor**

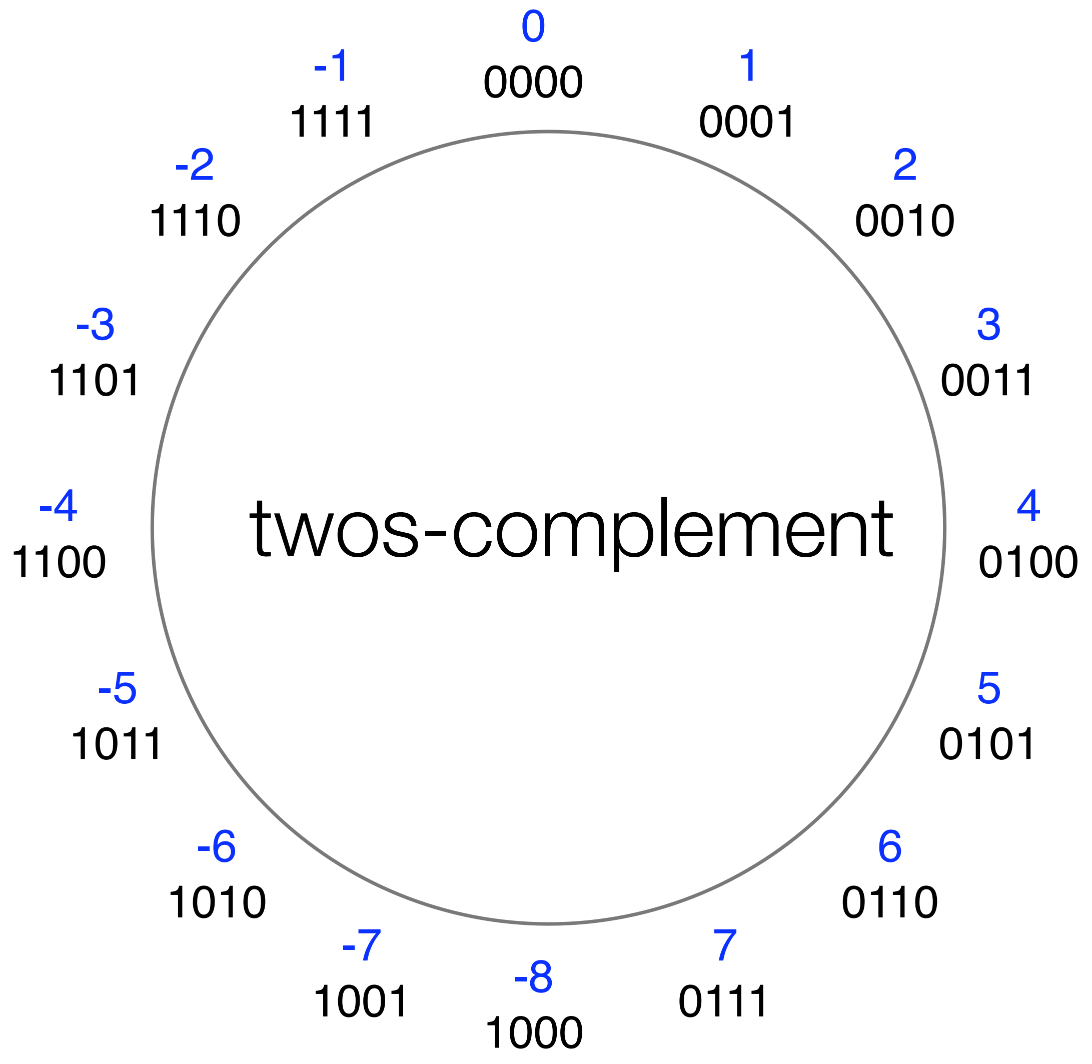
## Subtraction w. twos complement representation



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

## In class exercise: designing an adder-subtractor

## Handling overflow



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

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

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

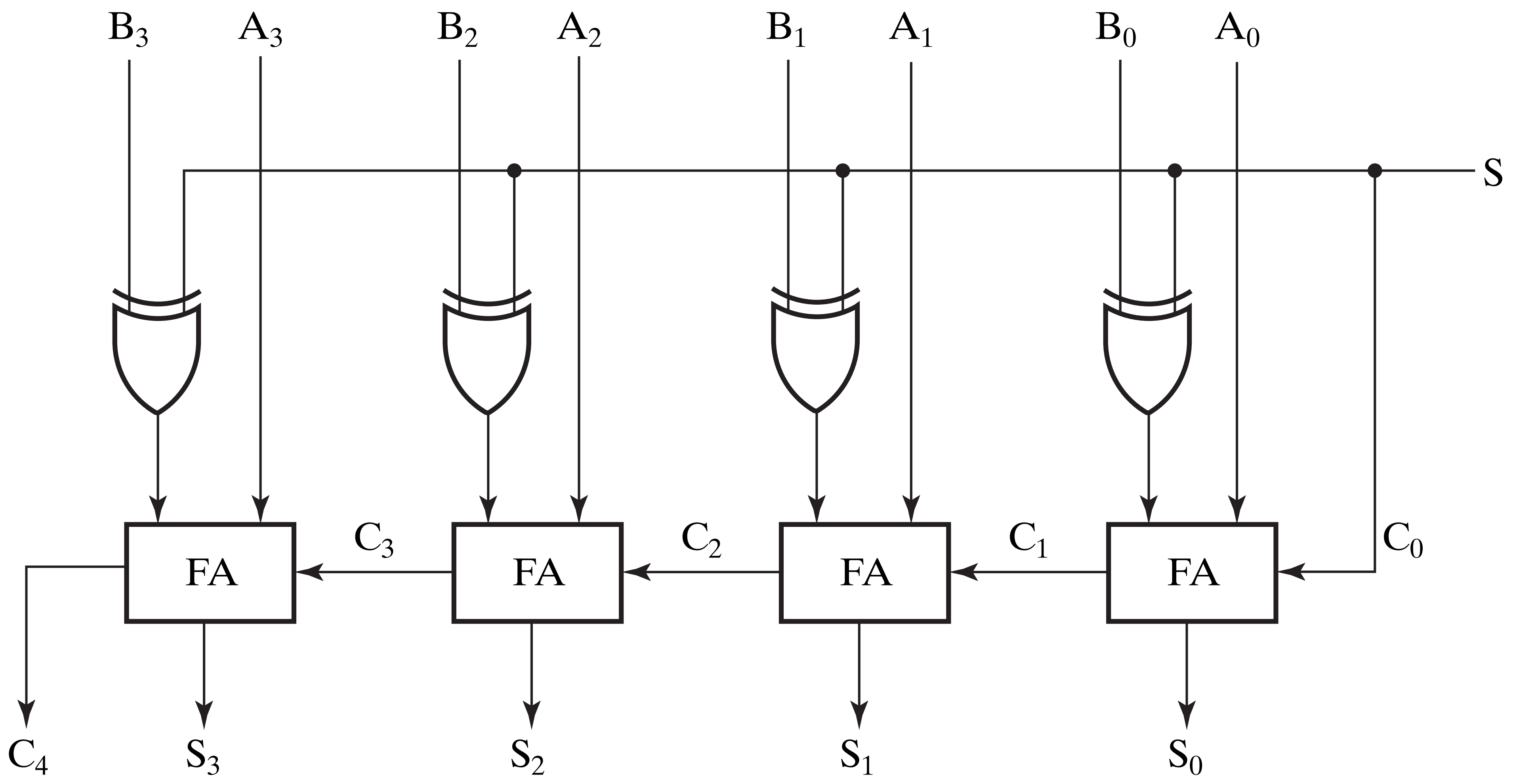
```
      0010
(-6)  1010
(3)   0011
      ----
      1101    (-3)
```

## Handling overflow (2)

| a4 | b4 | c4 | c5 | s4 | 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 | |

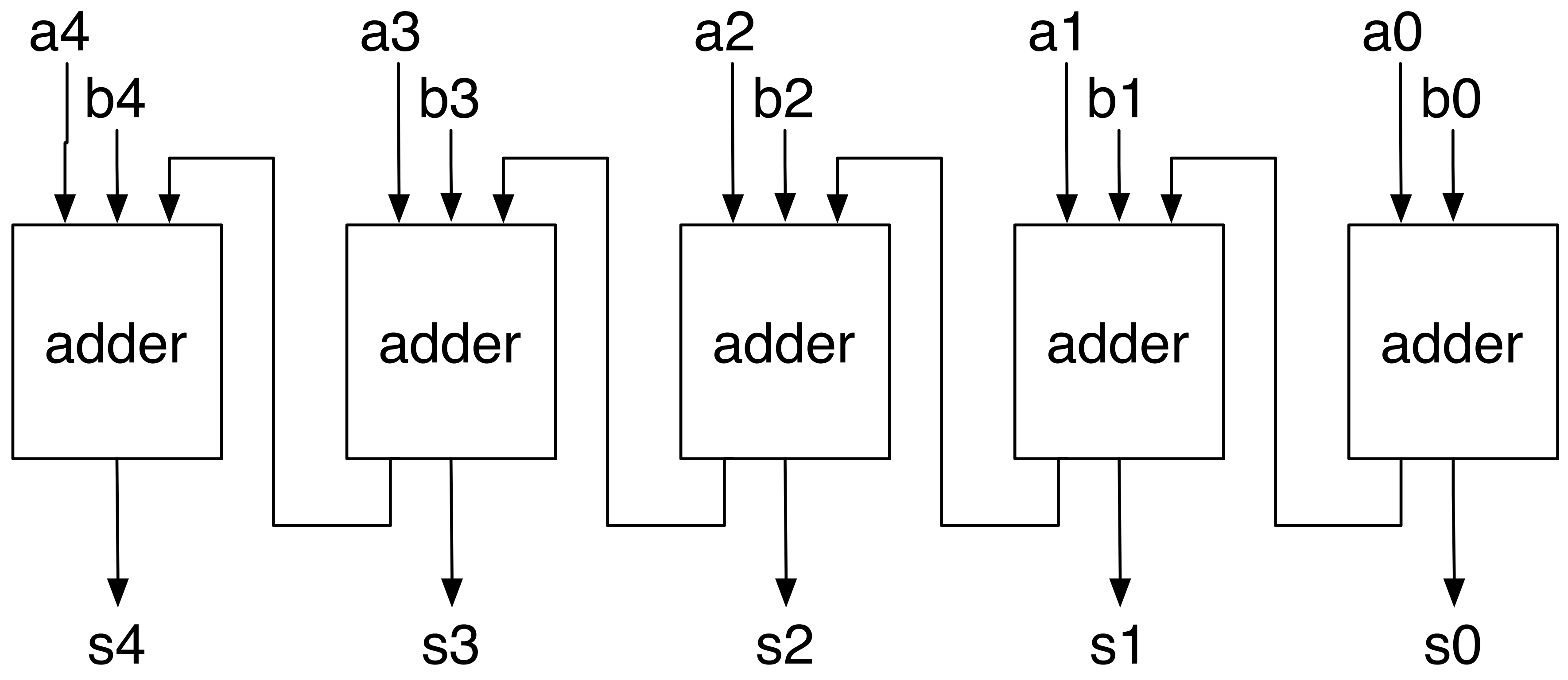
# Adder/subtractor

4-7



# 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]



## Faster adders

- Since an adder is just combinational logic, we know it can be implemented in two levels of logic, but with more hardware.
- This area/delay tradeoff is very common.

## Carry lookahead adder (CLA)

- Start by rewriting the carry function

$$c_{i+1} = a_ib_i + a_ic_i + b_ic_i$$

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

$$c_{i+1} = \underbrace{g_i}_{\text{carry generate}} + c_i\,(\underbrace{p_i}_{\text{carry propagate}})$$

carry generate

$g_i = a_ib_i$

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

## Carry lookahead adder (CLA) (3)

$c_0 = 0$

$c_1 = g_0 + c_0p_0$

$c_2 = g_1 + g_0p_1 + c_0p_0p_1$

$c_3 = g_2 + g_1p_2 + g_0p_1p_2 + c_0p_0p_1p_2$

$s_0 = a_0 \oplus b_0 \oplus c_0$

$s_1 = a_1 \oplus b_1 \oplus c_1$

$s_2 = a_2 \oplus b_2 \oplus c_2$

$s_3 = a_3 \oplus b_3 \oplus c_3$

## CLA delay analysis

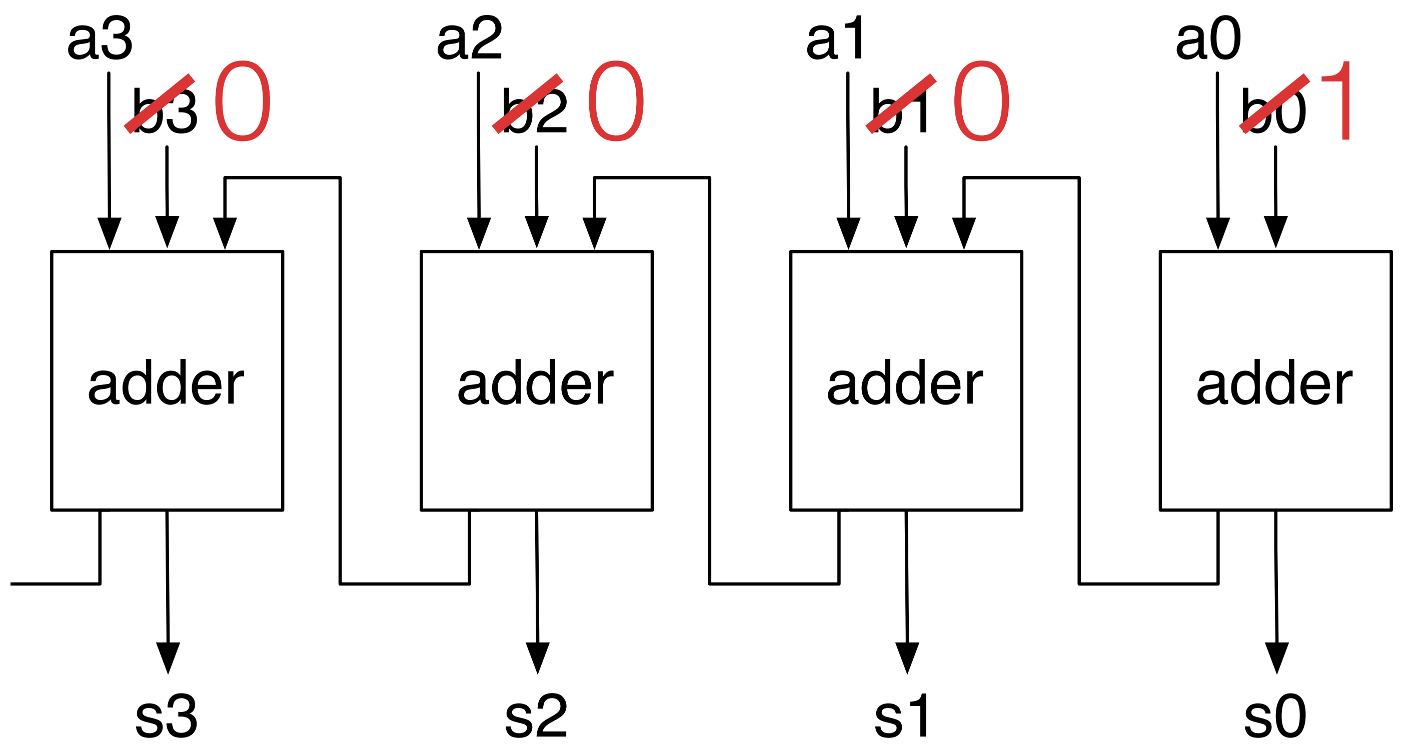
- Delay to produce any $p_i$ or $g_i$?
- Delay to produce $S_0$?
- Delay to produce $S_1$?
- Delay to produce $S_2$?
- Delay to produce $S_3$?

## Contraction

- Simplification of a circuit through constant input values.

# In class exercise

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


a3
~~b3~~ 0
a2
~~b2~~ 0
a1
~~b1~~ 0
a0
~~b0~~ 1
adder
adder
adder
adder
s3
s2
s1
s0