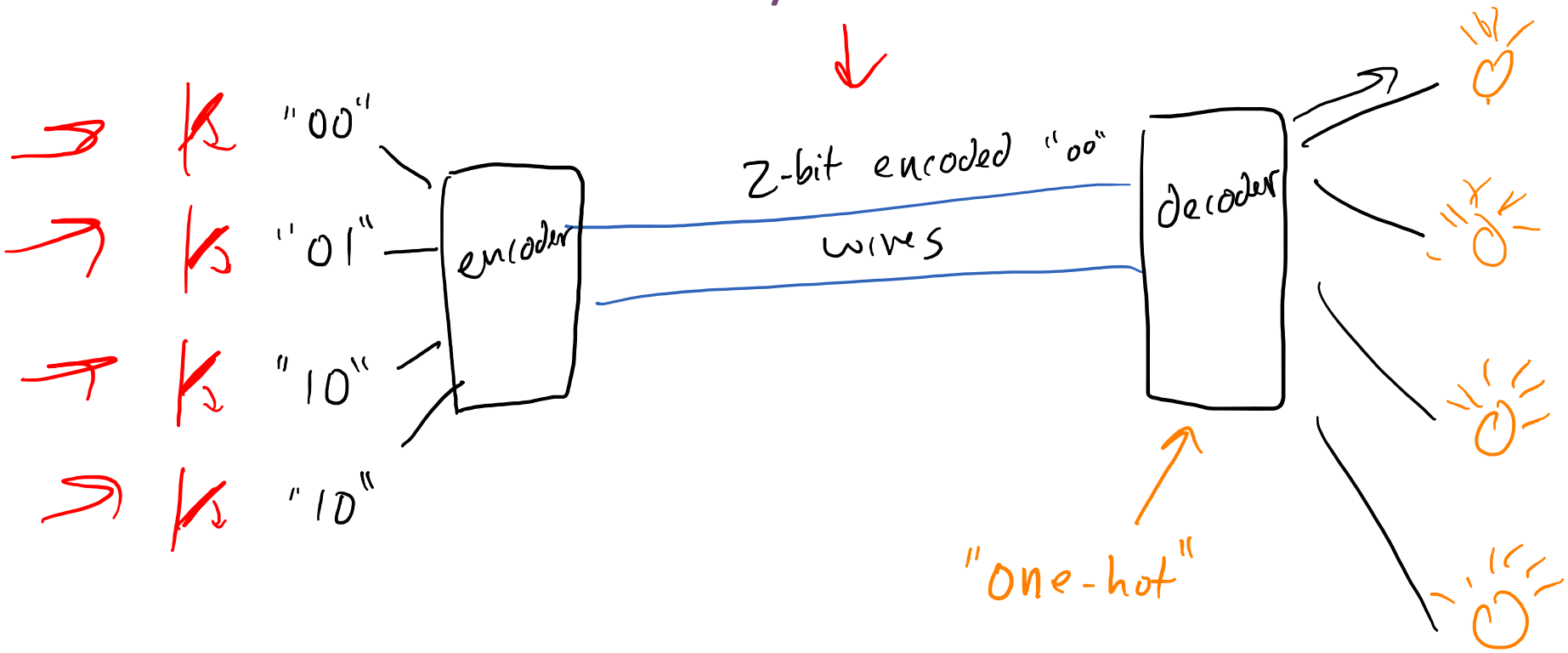
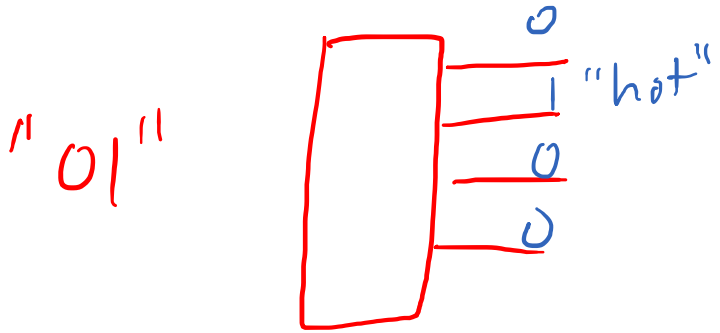
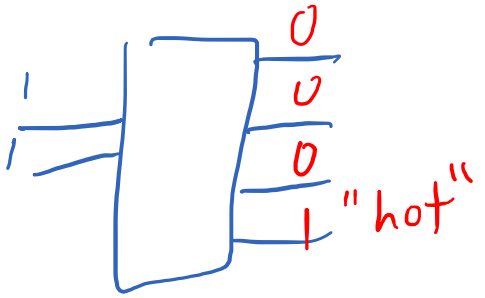
Multiplexers
Demultiplexers

Andrew Lukefahr

Last Time: Encoder / Decoder



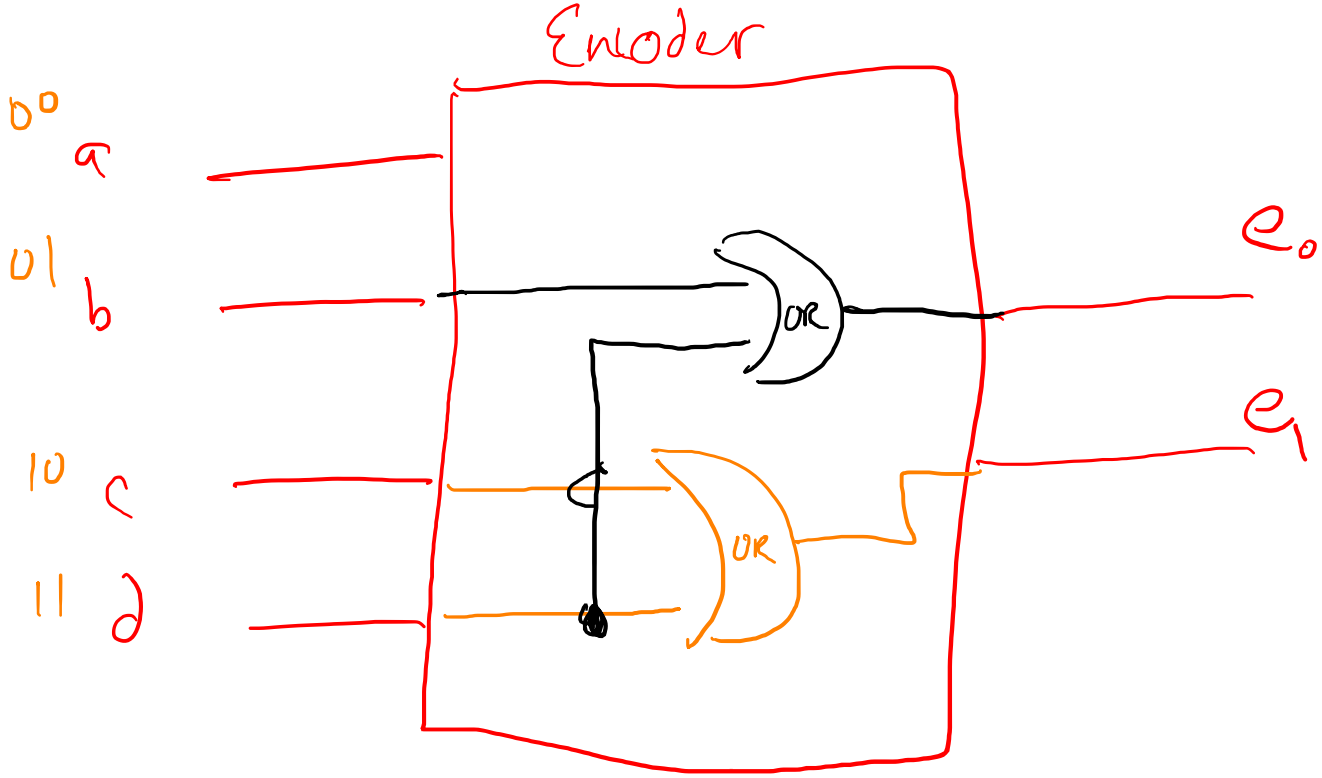




Encoders

- Have: 4 input wires
- Want: Encoded data to 2 output wires

Inputs				Outputs		
a	b	c	d	e_1	e_0	dec
1	0	0	0	0	0	0
0	1	0	0	0	1	1
0	0	1	0	1	0	2
0	0	0	1	1	1	3



$e_1 = c \mid d;$

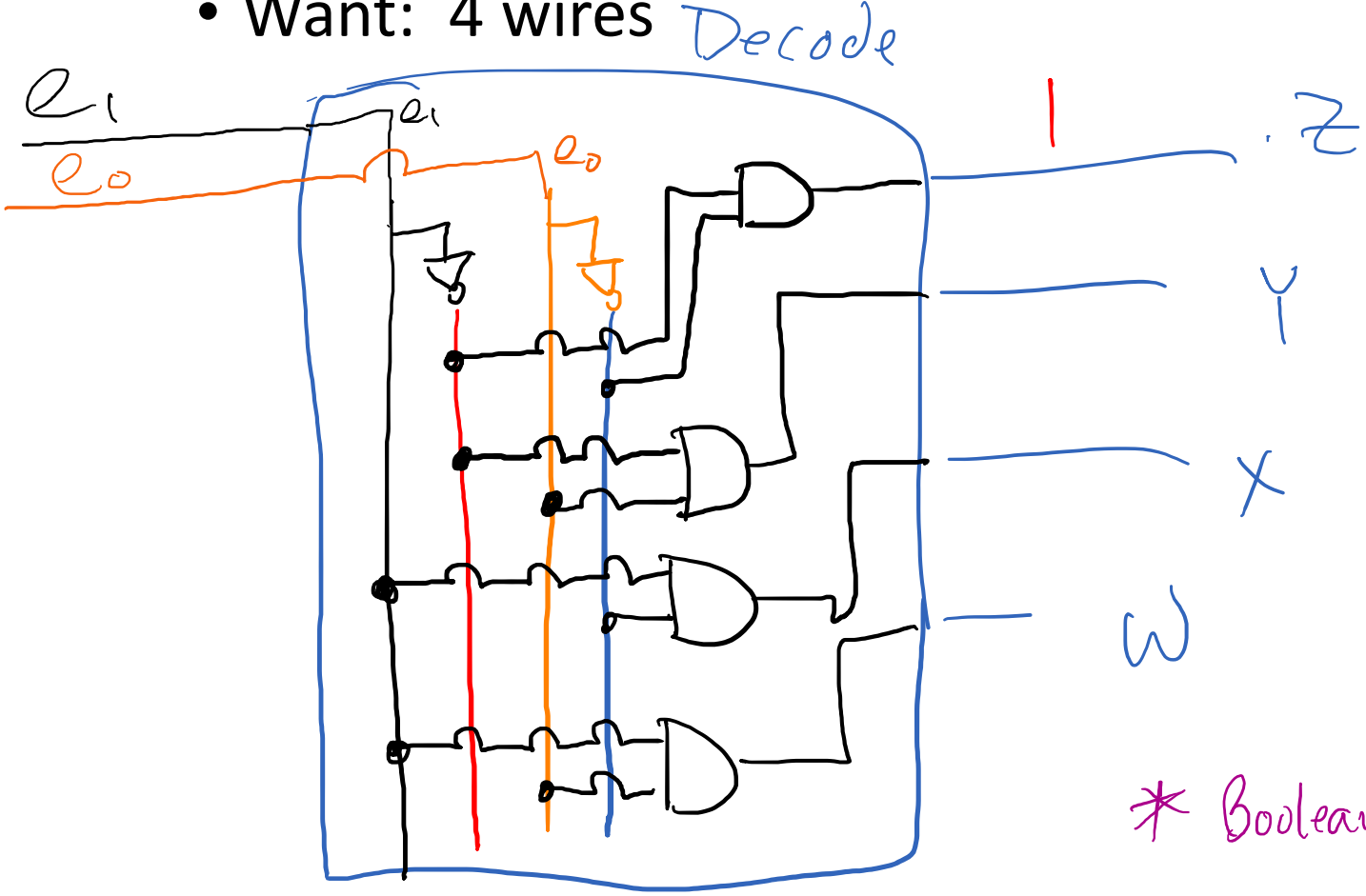
$e_0 = b \mid d;$

Decoders

- Have: 2-bit encoded data
- Want: 4 wires

e_1	e_0
0	0
0	1
1	0
1	1

W	X	Y	Z
0	0	0	1
0	0	1	0
0	1	0	0
1	0	0	0



$Z = \sim e_1 \,\&\, \sim e_0$

$Y = \sim e_1 \,\&\, e_0$

$X = e_1 \,\&\, \sim e_0$

$W = e_1 \,\&\, e_0$

* Boolean optimizations possible

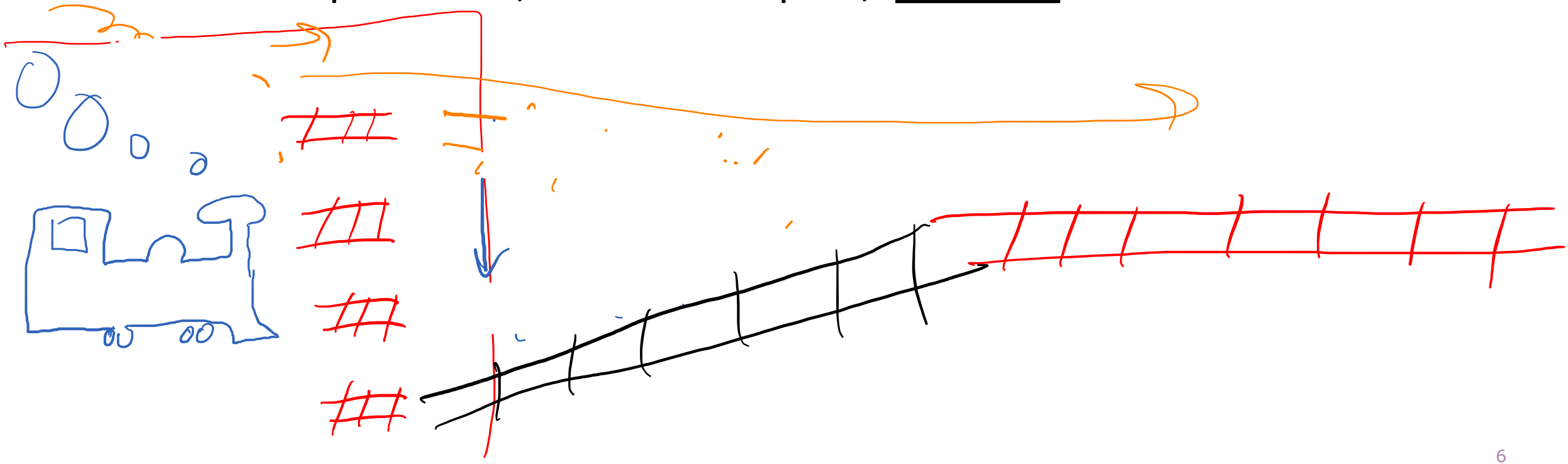
This time

- Multiplexers
- Demultiplexers
- Verilog arrays
- Binary Arithmetic

Multiplexer

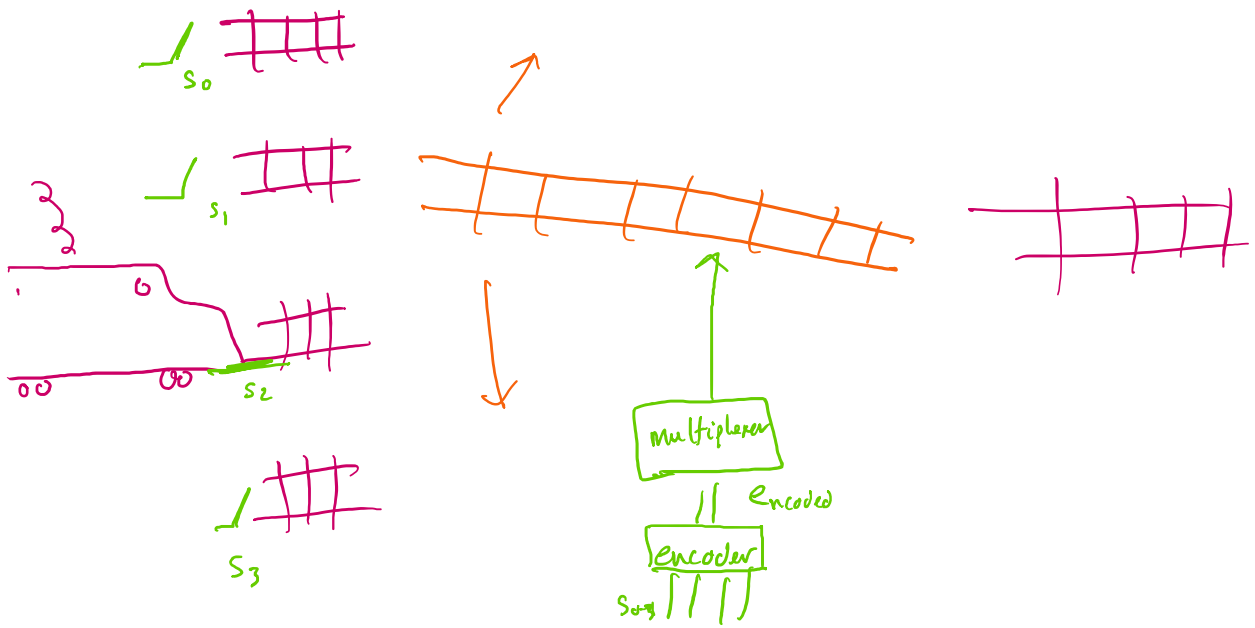
- Input “Selector”: Lets one input through at a time

- 2^N data input wires , N “select” inputs, <u>1 output</u>

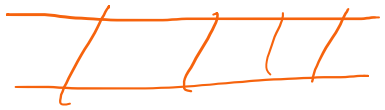


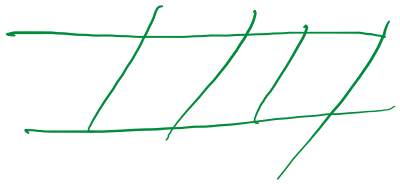
Multiplexer

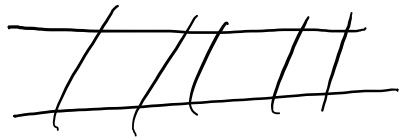
- 2^N data input wires , N “select” inputs, 1 output,



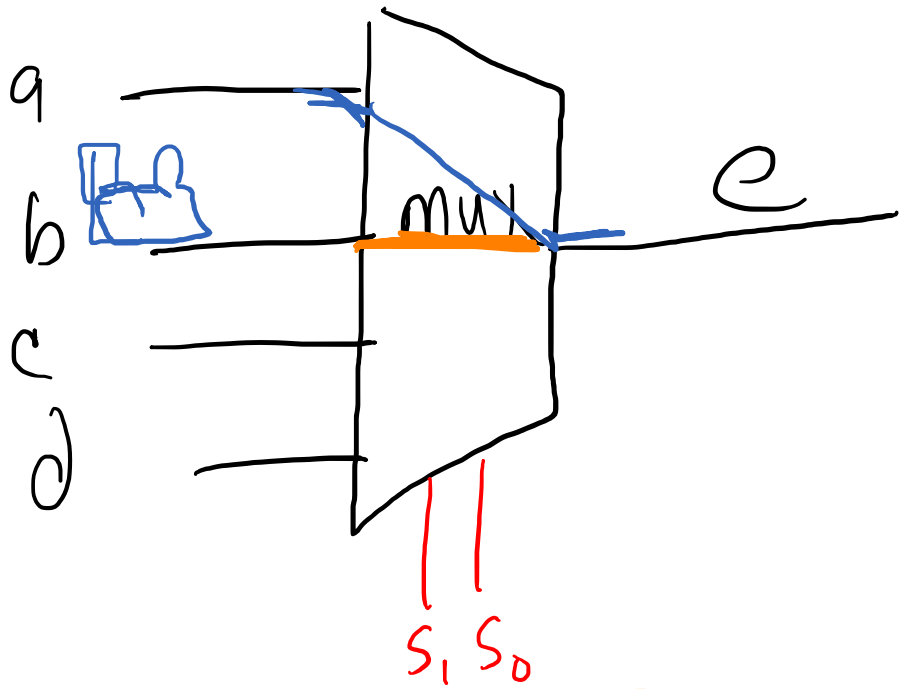
Multiplexer: Train Examples







Multiplexer



S_1	S_0	a	b	c	d	e	
0	0	0	0	0	0	0	(a)
0	0	1	X	X	X	1	(a)
0	1	X	0	X	X	0	(b)
0	1	X	1	X	X	1	(b)
1	0	X	X	c	X	c	
1	1	X	X	X	d	d	

$e_a = \overline{S_1} \cdot \overline{S_0} \cdot a$

$e_b = \overline{S_1} \cdot S_0 \cdot b$

$e_c = S_1 \cdot \overline{S_0} \cdot c$

$e_d = S_1 \cdot S_0 \cdot d$

encoder

$e = e_a + e_b + e_c + e_d$

Multiplexer

S_1	S_0	a	b	c	d	e
0	0	0	0	0	0	0
0	0	0	0	0	1	0
0	0	0	x	x	x	0 (a)
0	0	1	x	x	x	1 (a)
0	1	x	0	x	x	0 (b)
0	1	x	1	x	x	1 (b)
1	0	x	x	c	x	c
1	1	x	x	x	d	d

$e = e_a \mid e_b \mid e_c \mid e_d$

$e_a = \overline{S_1} \,\&\, \overline{S_0} \,\&\, a$

$e_b = \overline{S_1} \,\&\, S_0 \,\&\, b$

$e_c = S_1 \,\&\, \overline{S_0} \,\&\, c$

$e_d = S_1 \,\&\, S_0 \,\&\, d$

Multiplexer

$$e = e_a \mid e_b \mid e_c \mid e_d$$

$$e_a = \overline{S_1} \,\&\, \overline{S_0} \,\&\, a$$

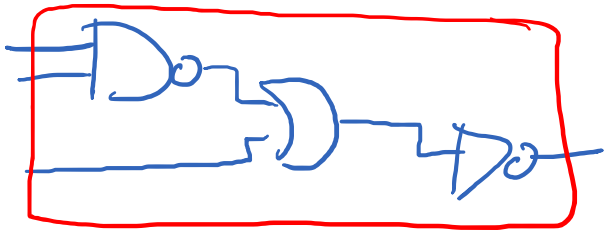
$$e_b = \overline{S_1} \,\&\, S_0 \,\&\, b$$

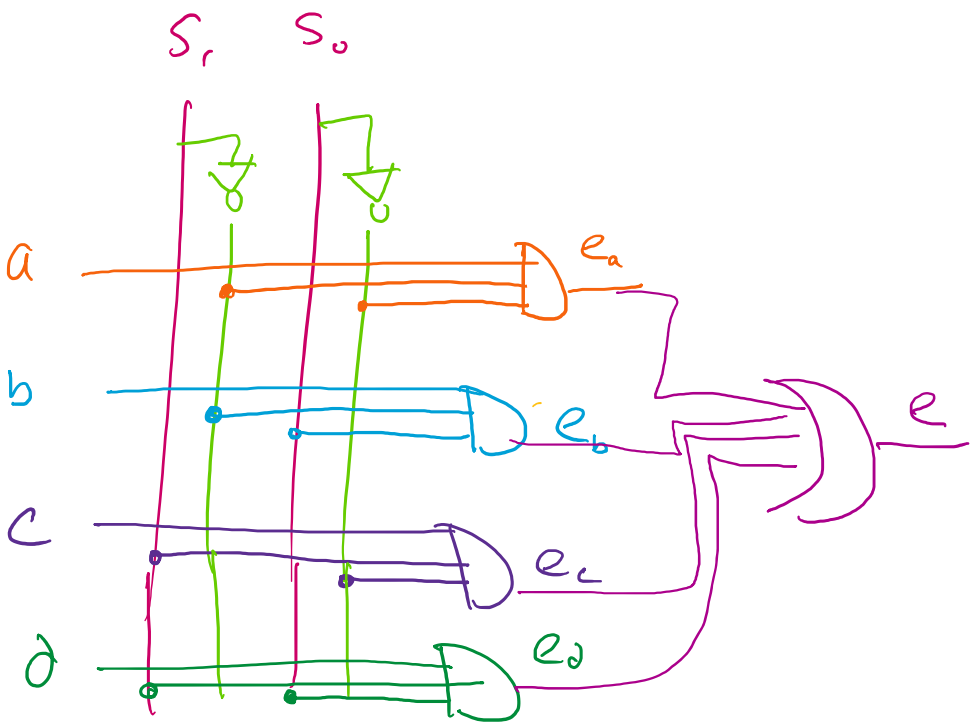
$$e_c = S_1 \,\&\, \overline{S_0} \,\&\, c$$

$$e_d = S_1 \,\&\, S_0 \,\&\, d$$

Multiplexer

2 NAND
2 NAND





$e = e_a \mid e_b \mid e_c \mid e_d$

$e_a = \overline{S_1} \,\&\, \overline{S_0} \,\&\, a$

$e_b = \overline{S_1} \,\&\, S_0 \,\&\, b$

$e_c = S_1 \,\&\, \overline{S_0} \,\&\, c$

$e_d = S_1 \,\&\, S_0 \,\&\, d$

Multiplexer Verilog

```
module mux4(
    input a,b,c,d,
    input s1, s0,
    output m
    );

  wire

endmodule
```

Multiplexer Verilog

fixme!

note: skipped assign stmts.

```
module mux4(
    input a,b,c,d,
    input s1, s0,
    output m
    );

    wire ao = ~s1 & ~s0 & a;
    wire bo = ~s1 &  s0 & b;
    wire co =  s1 & ~s0 & c;
    wire do =  s1 &  s0 & d;

    assign m = ao | bo | co | do;
endmodule
```

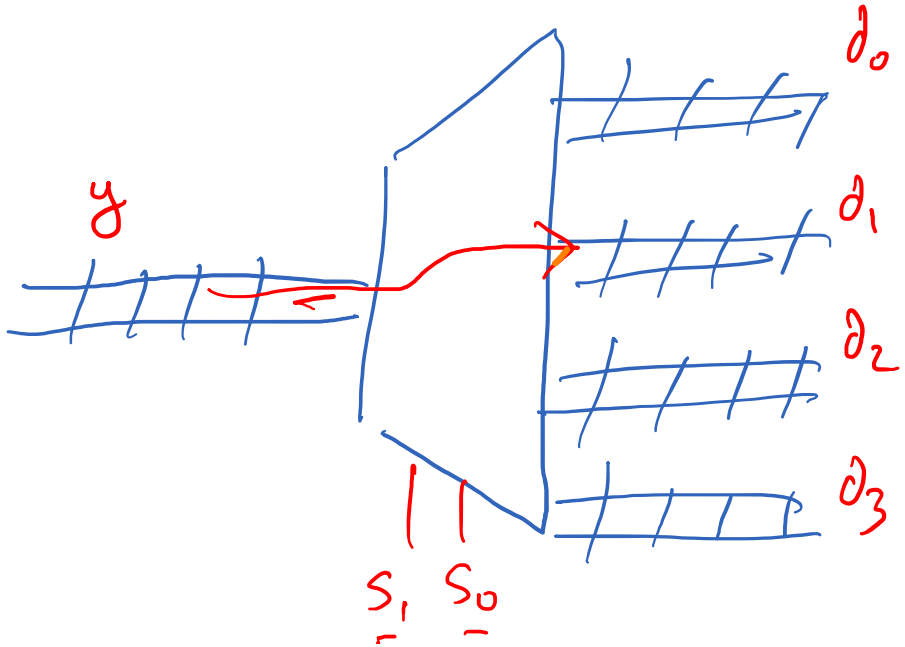
e_a
e_b
e_c
e_d

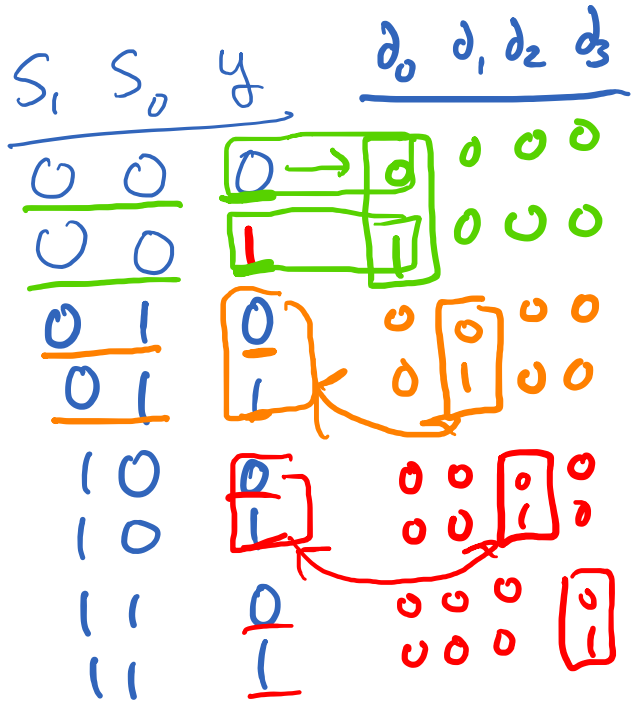
Demultiplexers

- Opposite of a “multiplexer”
- 1 data input, N “select” inputs, 2^N outputs

Demultiplexers

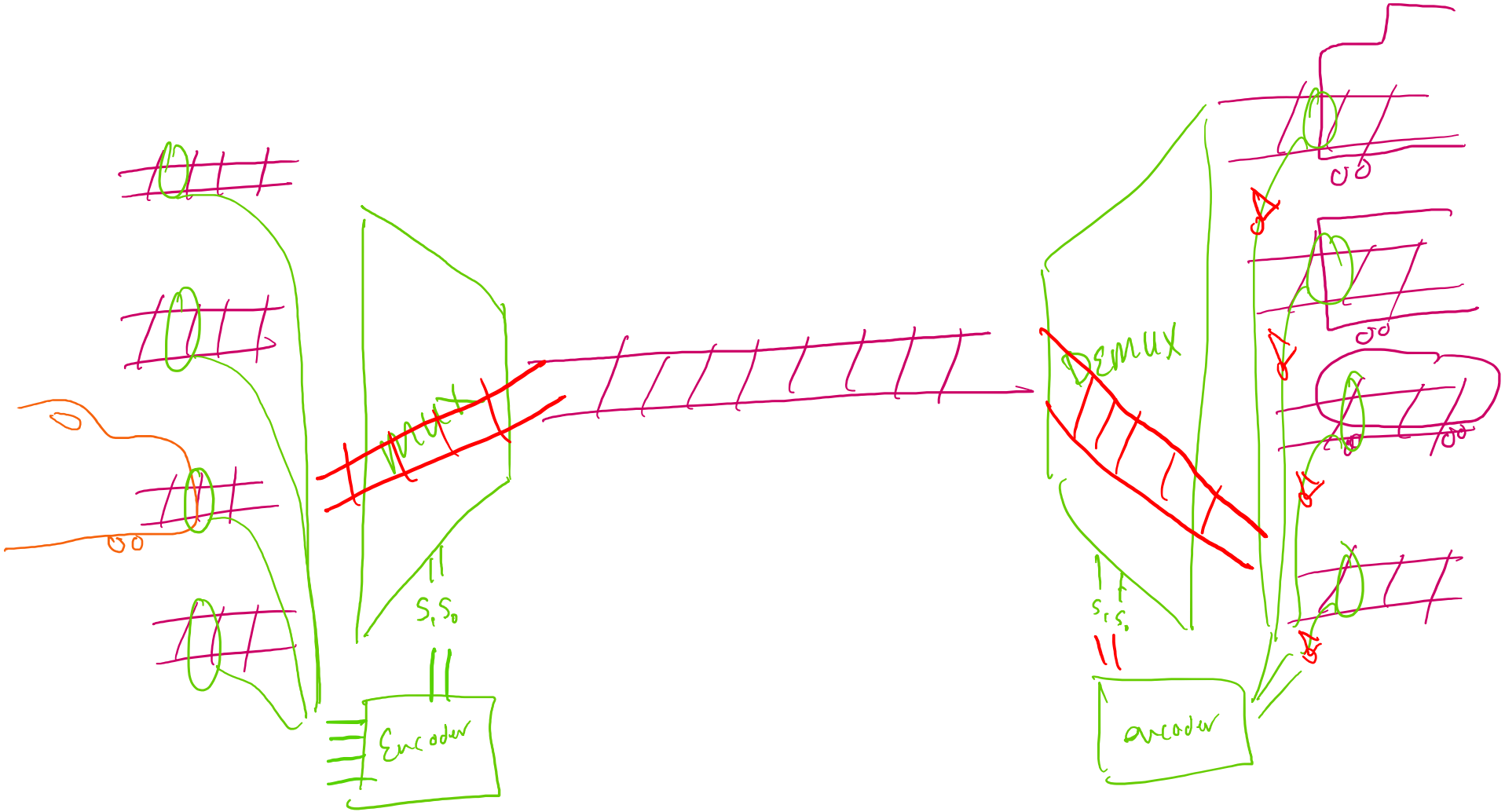
- Opposite of a “multiplexer”
- 1 data input, N “select” inputs, 2^N outputs





Train Example

Train Example



Arrays in Verilog

C- int foo [5];

wire [15:0] foo;

- Bundle multiple wires together to form an array.

```
type [mostSignificantIndex:leastSignificantIndex] name;
```

- Examples

flipped C →
C-like →
pythonlike →

- `wire[15:0] x; //declare 16-bit array`
- `x[2]  // access wire 2 within x`
- `x[5:2] //access wire 5 through 2`
- `x[5:2]= {1,0,y,z}; //concatenate 4 signals`

wire [15:0] x

x = y

wire [0:15] y

5 4 3 2

x[2:5] = {z, y, 0, 1};

Arrays in Verilog

- Can also be used in module definitions

```
module multiply (
   input    [7:0]     a,    //8-bit signal
   input    [7:0]     b,    //8-bit signal
   output  [15:0]     c     //16-bit signal
   );
   //stuff
endmodule
```

Constants in Verilog

- A `wire` only needs `1` or `0`
- Arrays need more bits, how to specify?

{ 42, 26 }

wire foo = 1;

wire [15:0] bar = 42;

0x42;

bits ' radix value

8'h0 0000 0000

8'hff; 1111 1111

9'd3 = 0 0000 0011

3'b2 = 0 10

- RULE: Always specify the bit-widths and notation!

Constants in Verilog

- A `wire` only needs `1` or `0`
- Arrays need more bits, how to specify?

- `8'h0  = 0000 0000` //using hex notation
- `8'hff  = 1111 1111`
- `8'b1  = 0000 0001` // using binary notation
- `8'b10 = 0000 0010`
- `8'd8 = 0000 1000` //using decimal notation

- RULE: Always specify the bit-widths and notation!

Constants in Verilog

```
module mtest;
    reg      [7:0]   aa = {1'b0,1'b1,1'b0,1'b0,
                           1'b1,1'b0,1'b0,1'b0};
    reg      [7:0]   bb = 8'b01001000;
    wire     [15:0]  cc ;
    reg      [7:0]   yy = {8{1'b1}};
    reg      [7:0]   zz = 8'hff;

    multiply m0(.a(aa), .b(8'h1), .c(cc));

endmodule
```

Binary Arithmetic

→ Questions on Mux / Demux? / Arrays?

→ Is this review?

→ Who has seen binary arithmetic before?

→ Two's Complement

$5 - 2 = 5 + -2$

Binary Addition

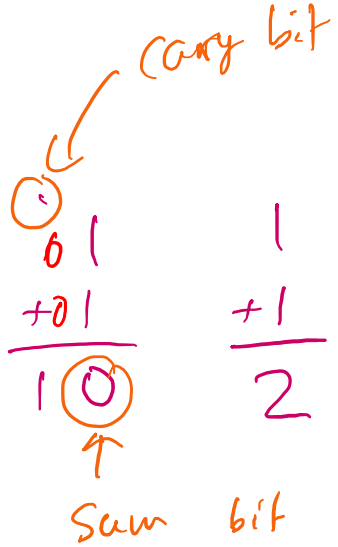
- How do we add two binary numbers, x & y?

Binary Addition

Add two binary numbers, x & y.

$x = \{0, 1\}$ $y = \{0, 1\}$

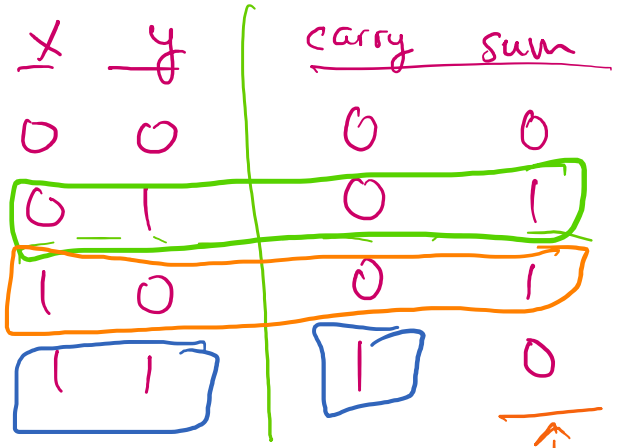
x	y	sum	carry	
0	0	0	0	
0	1	1	0	
1	0	1	0	
1	1	0	1	(10)

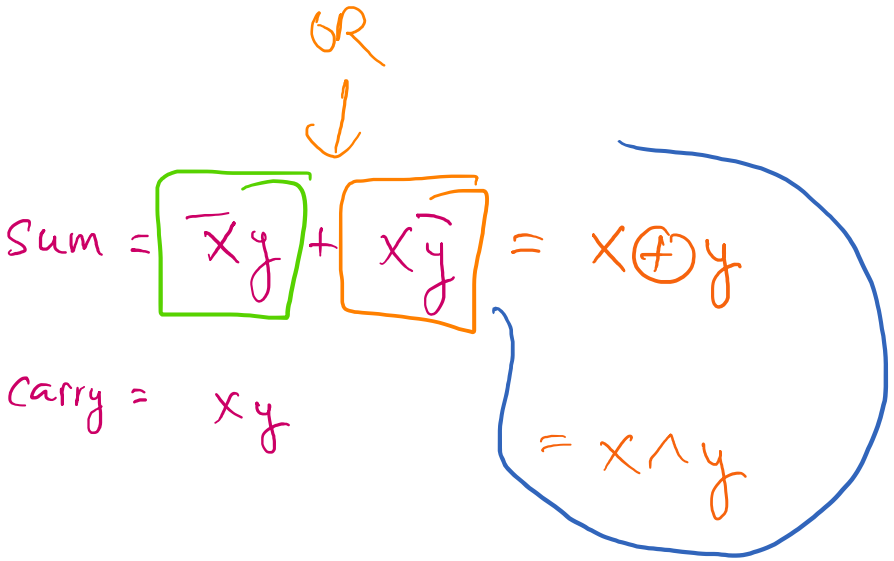


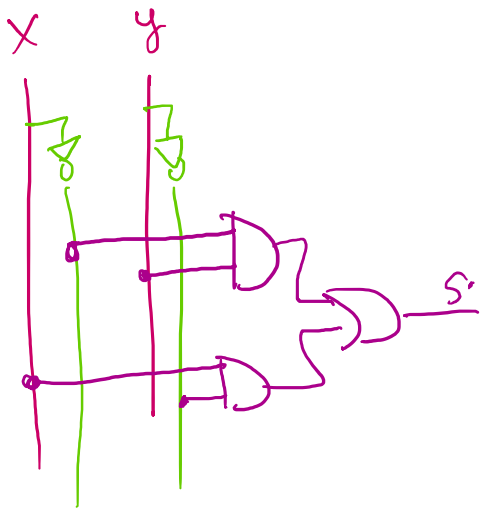
Half Adder

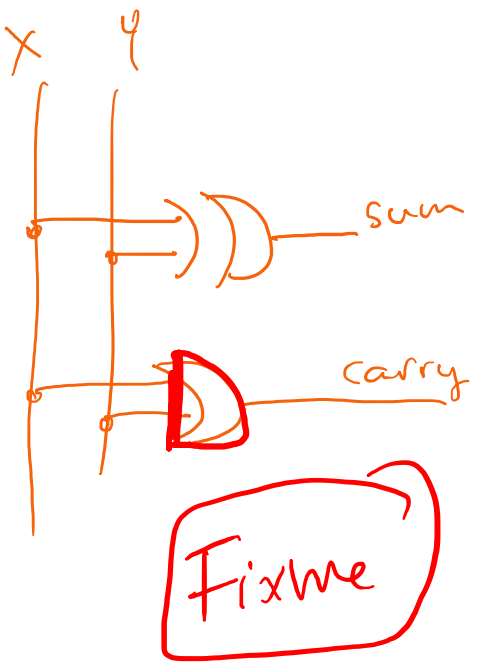
x	y	carry	sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Half Adder


x y carry sum
0 0 0 0
0 1 0 1
1 0 0 1
1 1 1 0
also xor?

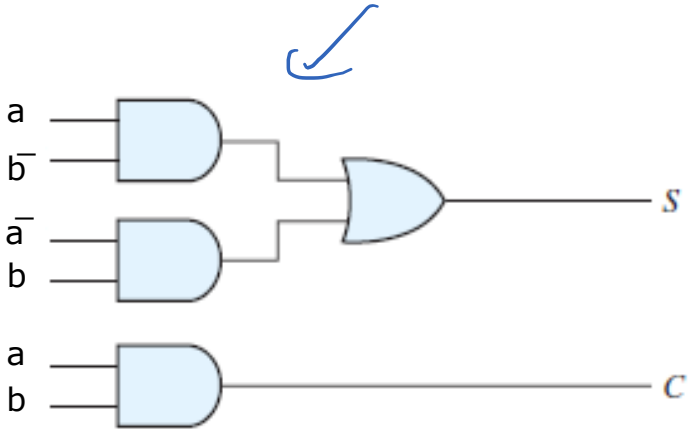

OR
Sum = $\bar{x}y + x\bar{y}$ = $x \oplus y$
carry = xy
= $x \wedge y$


x
y
S


x
y
sum
carry
Fixme

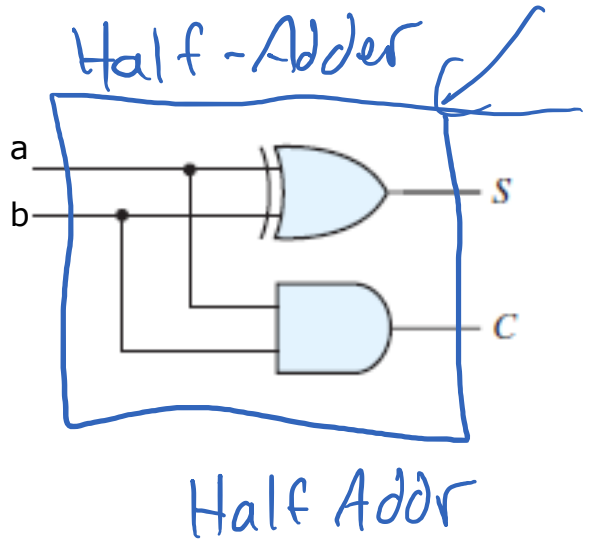
Half Adder

i	a	b	C	S
0	0	0	0	0
1	0	1	0	1
2	1	0	0	1
3	1	1	1	0



$S = a\,\bar{b} + \bar{a}\,b = a \oplus b$

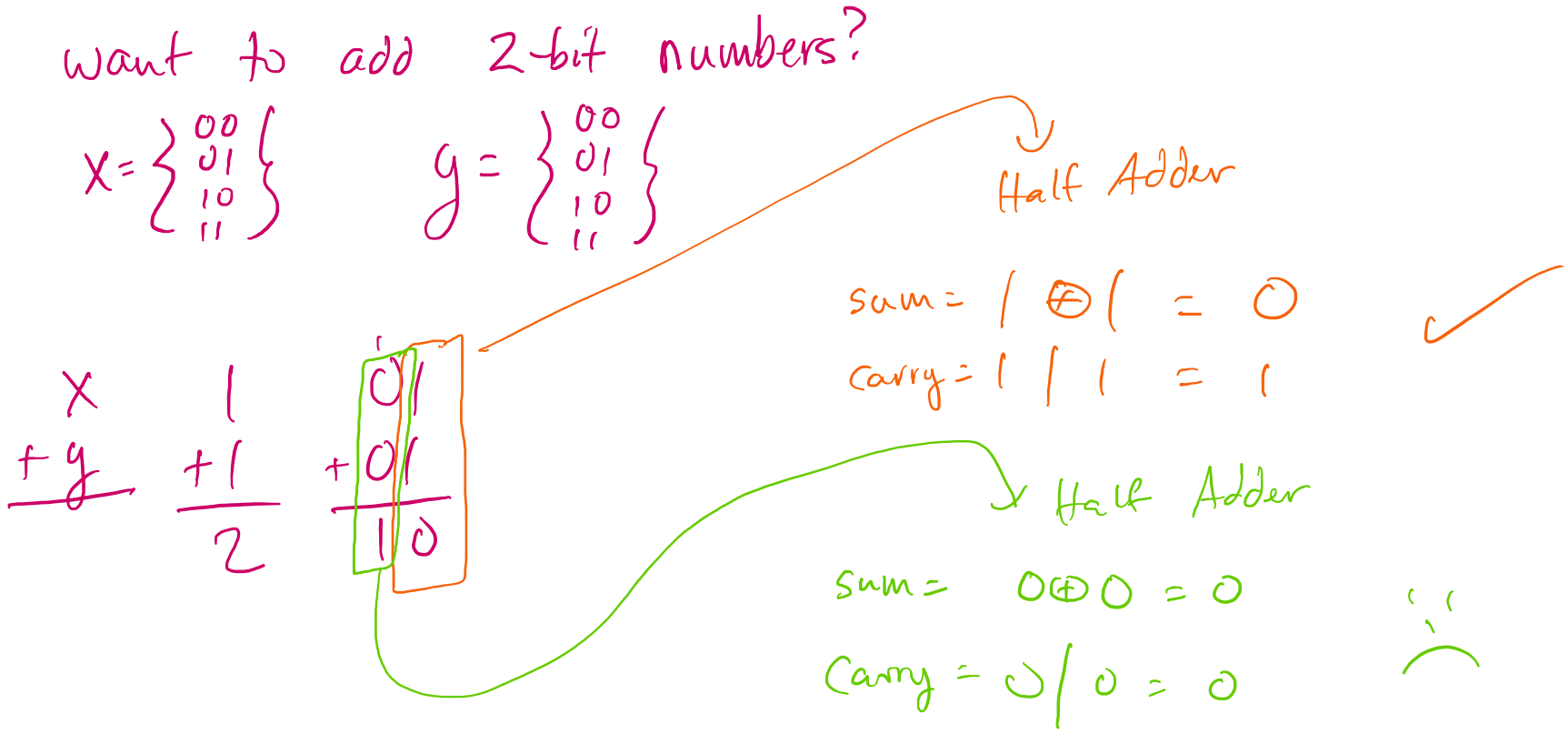
$C = a\,b$



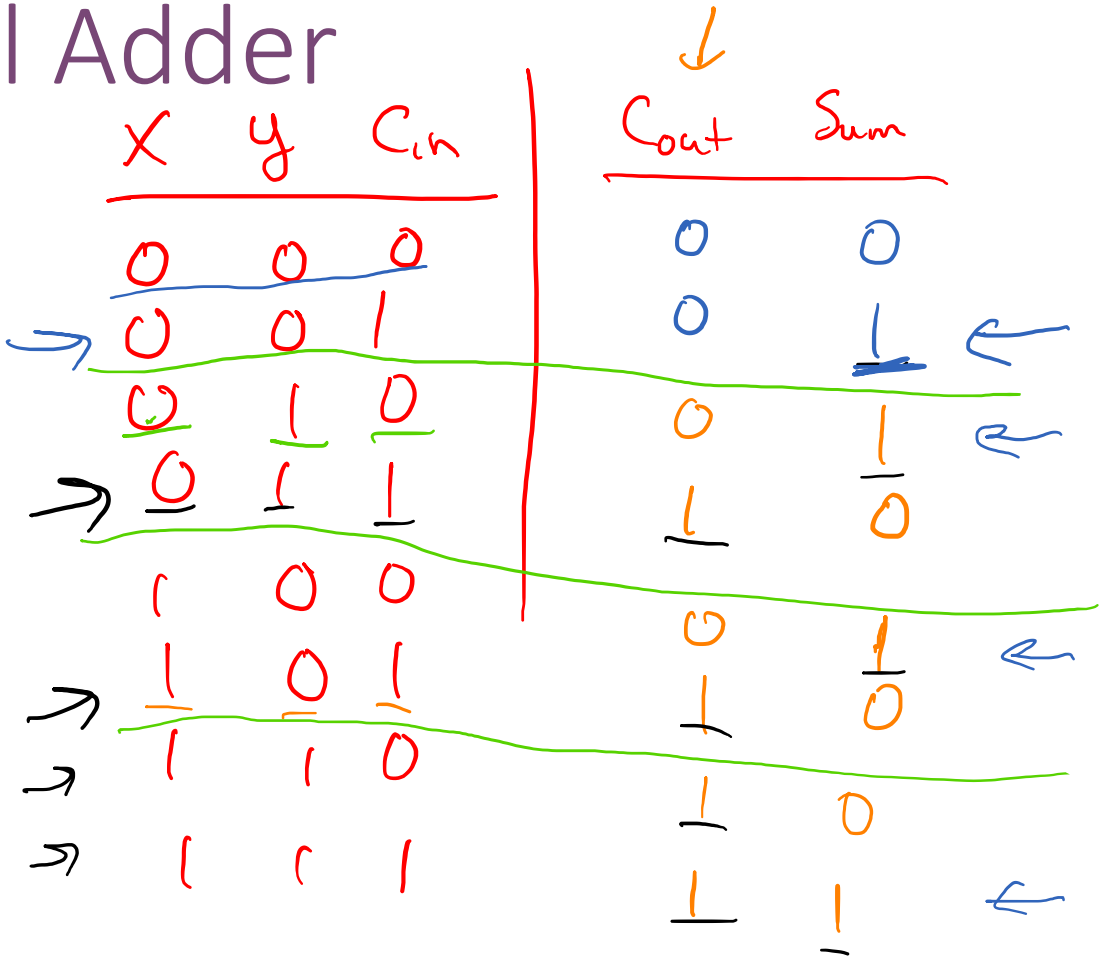
Binary Addition

- What if x and y are 2-bits each?

Binary Addition



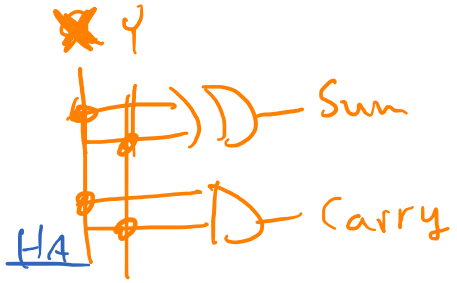
Full Adder



$$C_{out} = \bar{X}\,Y\,C_{in} + X\,\bar{Y}\,C_{in} + X\,Y\,\bar{C_{in}} + X\,Y\,C_{in}$$

$$Sum = \bar{X}\,\bar{Y}\,C_{in} + \bar{X}\,Y\,\bar{C_{in}} + X\,\bar{Y}\,\bar{C_{in}} + X\,Y\,C_{in}$$

Full Adder



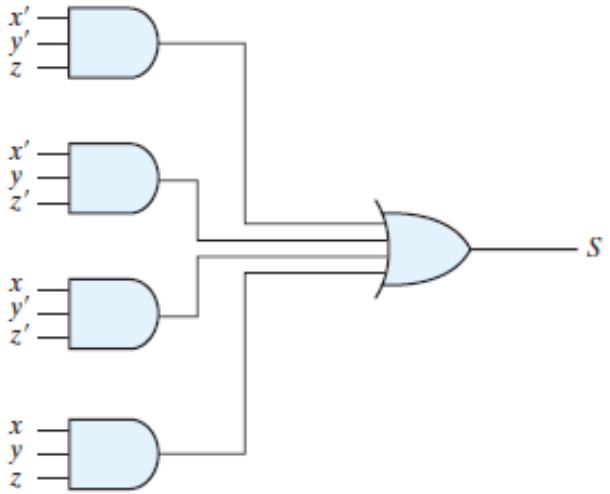
X	Y	C_{in}	Cout	Z	sum
0	0	0	HA	0	0
0	0	1		0	1
0	1	0		1	1
0	1	1		1	0
1	0	0		1	1
1	0	1		1	0 X
1	1	0		0	+y
1	1	1		0	Z
					1 + Cin
					sum

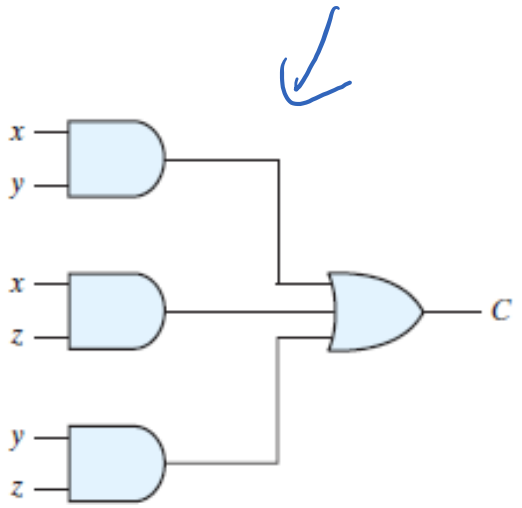
$$Sum = (\cancel{Sum} = X \oplus y) \oplus C_{in} = X \oplus y \oplus C_{in}$$

Sum = Sum(Sum(a,b), Cin)

Full Adder

i	x	y	z	C	S
0	0	0	0	0	0
1	0	0	1	0	1
2	0	1	0	0	1
3	0	1	1	1	0
4	1	0	0	0	1
5	1	0	1	1	0
6	1	1	0	1	0
7	1	1	1	1	1





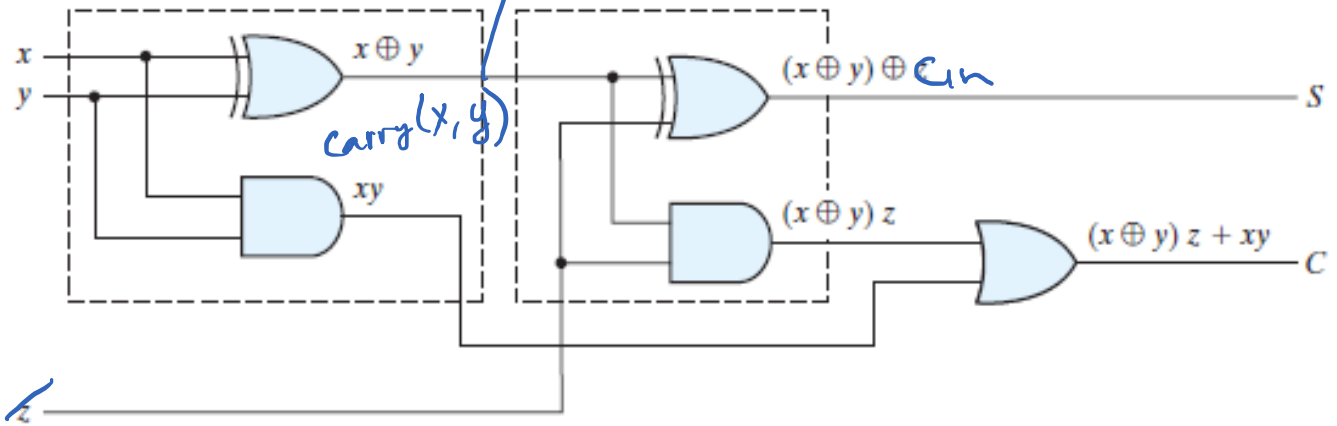
X+y = "10"

"x+y" = 01 + carry = "10"

sum(x,y)

X=1 = C=1
y=1

X=0 } Cout=1
y=1
Cin=1
t=1 } Cout=1
y=0
cin=1 } Cin

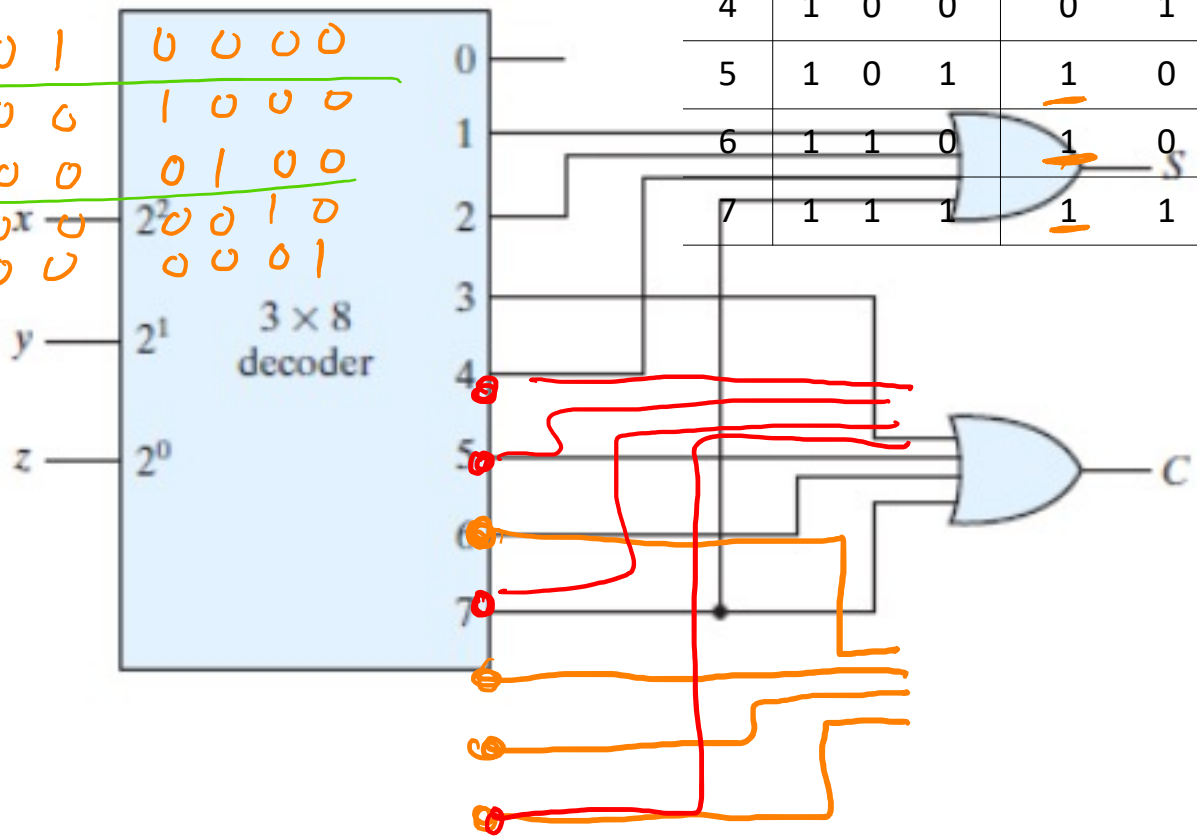


Full Adder

$S(x, y, z) = \{1, 2, 4, 7\}$

$C(x, y, z) = \{3, 5, 6, 7\}$

i	x	y	z	C	S
0	0	0	0	0	0
1	0	0	1	0	1
2	0	1	0	0	1
3	0	1	1	1	0
4	1	0	0	0	1
5	1	0	1	1	0
6	1	1	0	1	0
7	1	1	1	1	1

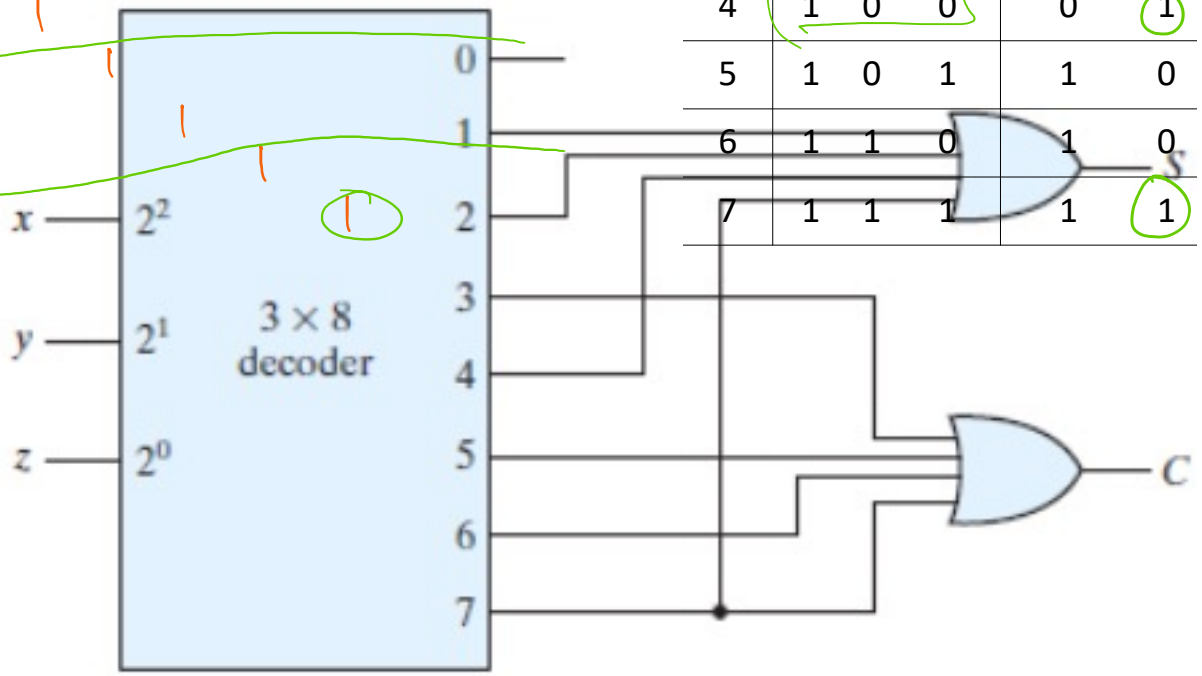


Full Adder

i	*x*	*y*	*z*	*C*	*S*
0	0	0	0	0	0
1	0	0	1	0	1
2	0	1	0	0	1
3	0	1	1	1	0
4	1	0	0	0	1
5	1	0	1	1	0
6	1	1	0	1	0
7	1	1	1	1	1

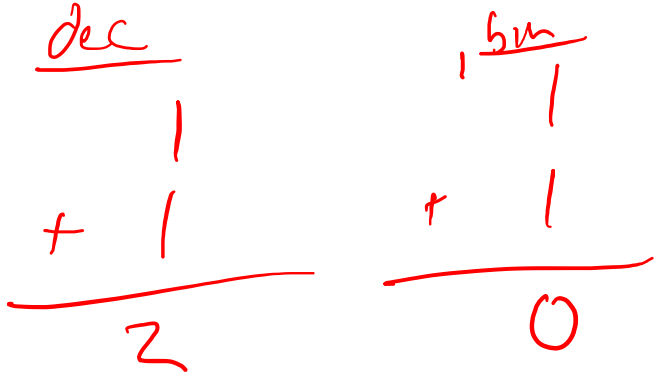
$S(x, y, z) = \{1, 2, 4, 7\}$

$C(x, y, z) = \{3, 5, 6, 7\}$



Multi-Bit Addition

- How do we build an adder if x & y are 4-bit numbers?



Multi-Bit Addition

have them do?

x & y are 4-bit numbers?

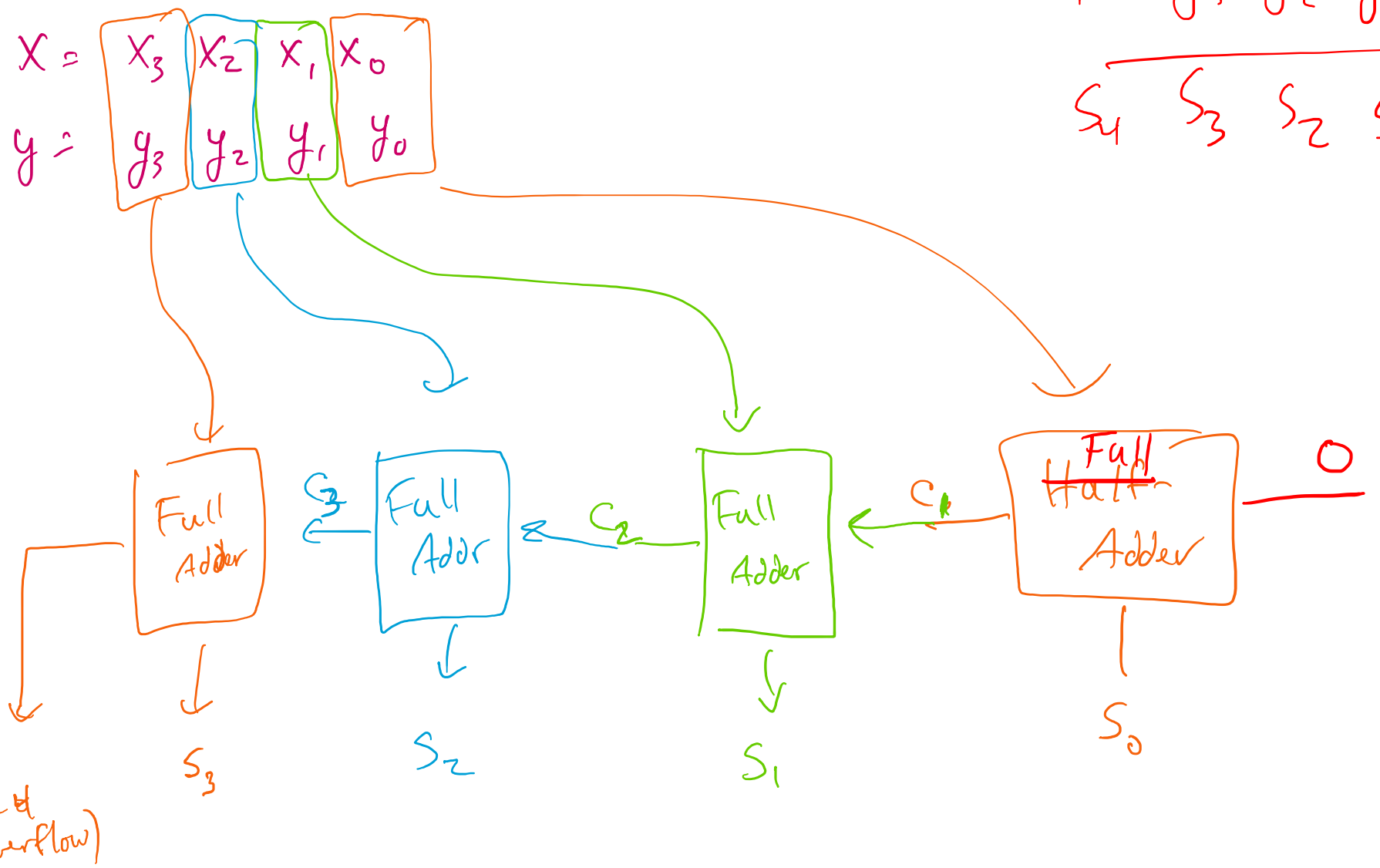
x	10	1010
+ y	+11	+1011
	21	10101

16 + 0 + 4 + 0 + 1 = 21

Ripple-Carry Adder

Ripple-Carry Adder

$X + Y = X_3\ X_2\ X_1\ X_0 + Y_3\ Y_2\ Y_1\ Y_0 = S_4\ S_3\ S_2\ S_1\ S_0$

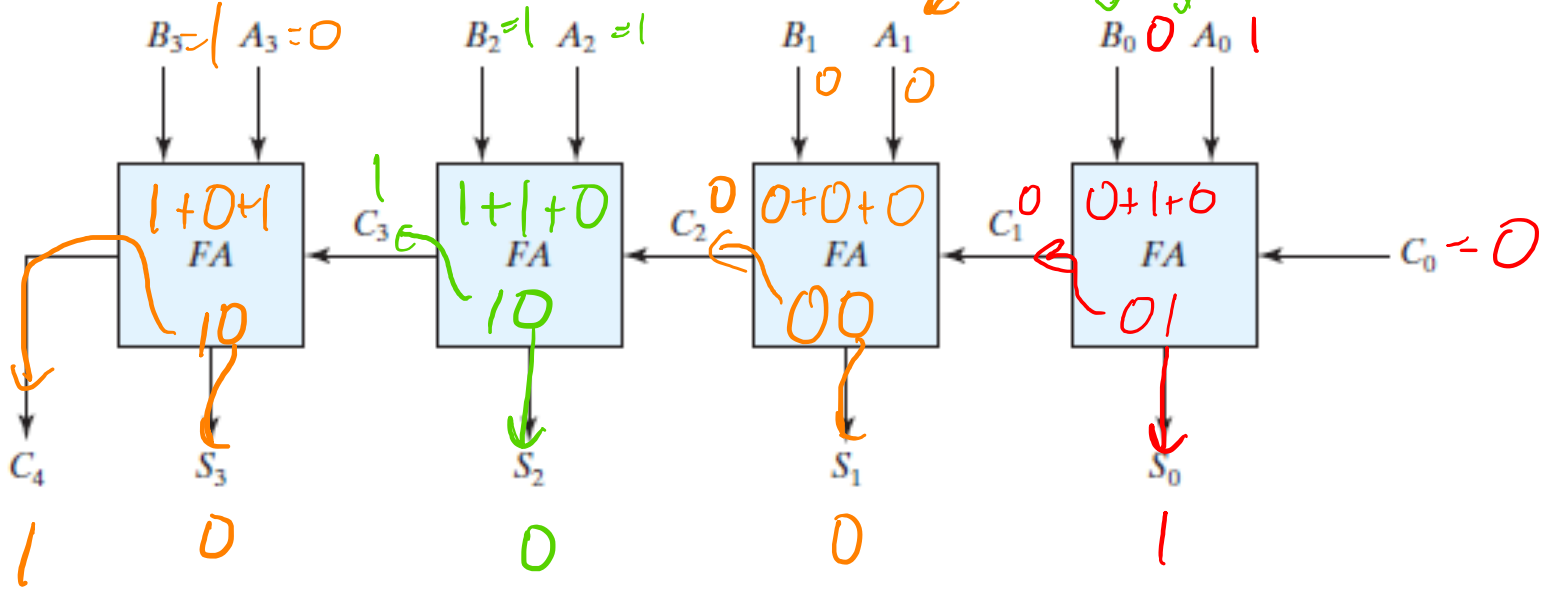


Ripple-Carry Adder

A = 4'b0101 = 5

B = 4'b0100 = 12

10001 = 17



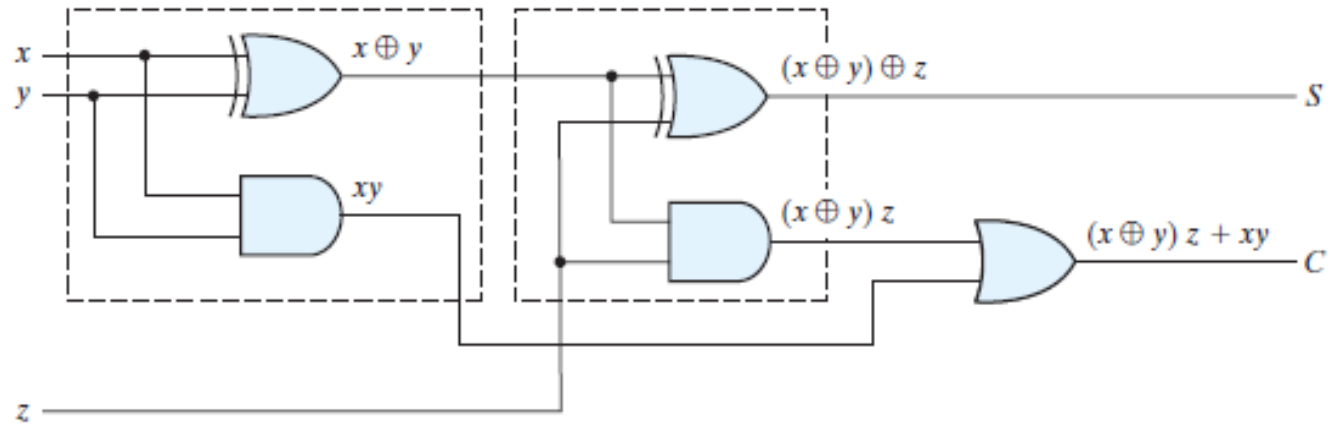
Gate Delay

- Gates are not magic, they are physical
- Takes time for changes flow through
- Assume 5ps / gate

- How fast can we update our adder?

Full Adder Gate Delay

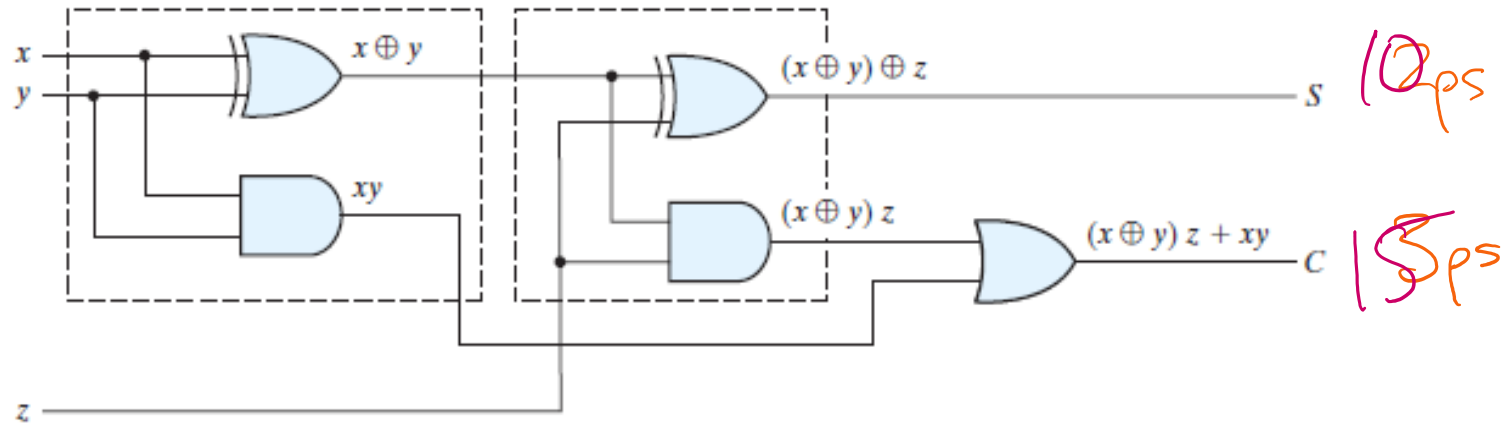
- Assume 5ps/gate



- What is the total delay on `s`? on `c`?

Full Adder Gate Delay

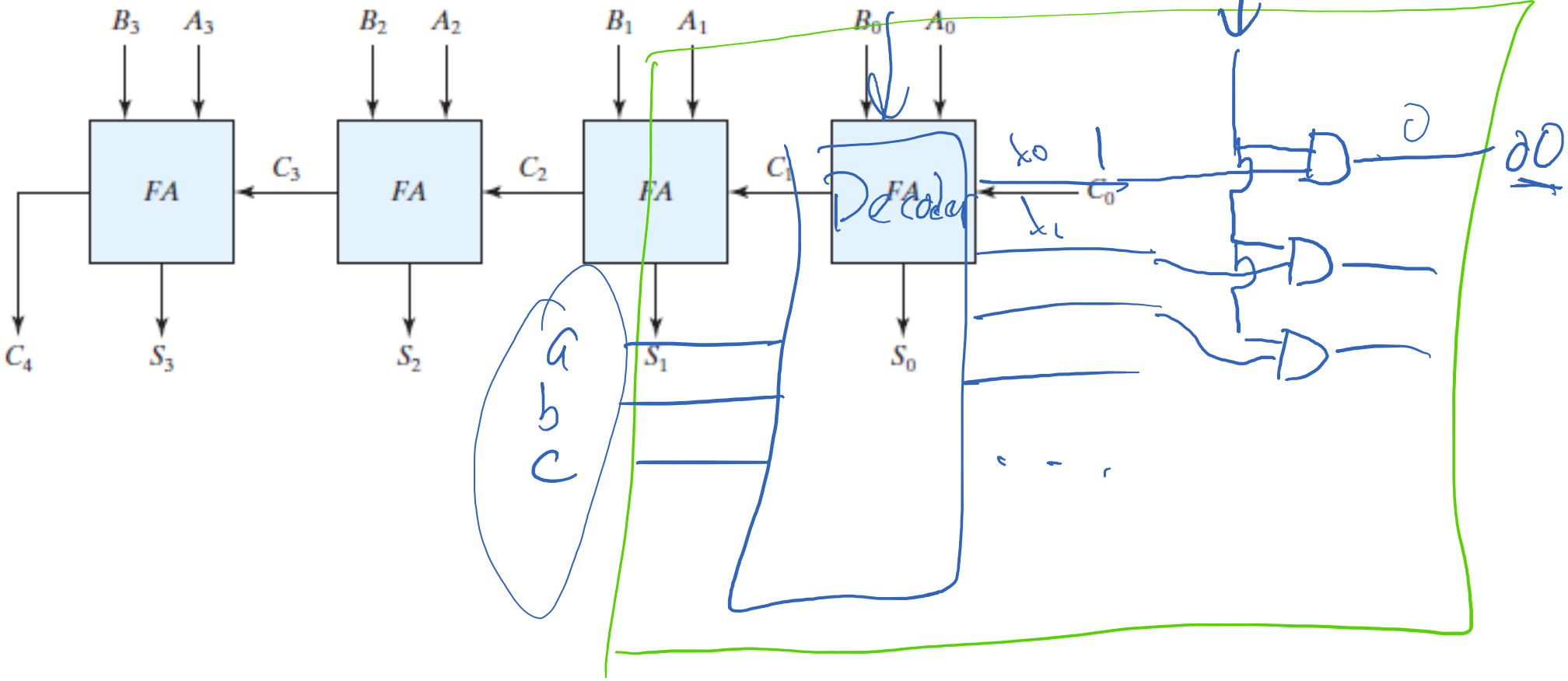
- Assume 1ps/gate



- What is the total delay on `s`? on `c`?

Ripple-Carry Gate Delays

- What is the total delay here?



Adder Gate Delays

- What is the total delay for:

- 1-bit addition:
- 4-bit addition:
- 8-bit addition:
- 16-bit addition:
- 32-bit addition:
- 64-bit addition:

Adder Gate Delays

- What is the total delay for:

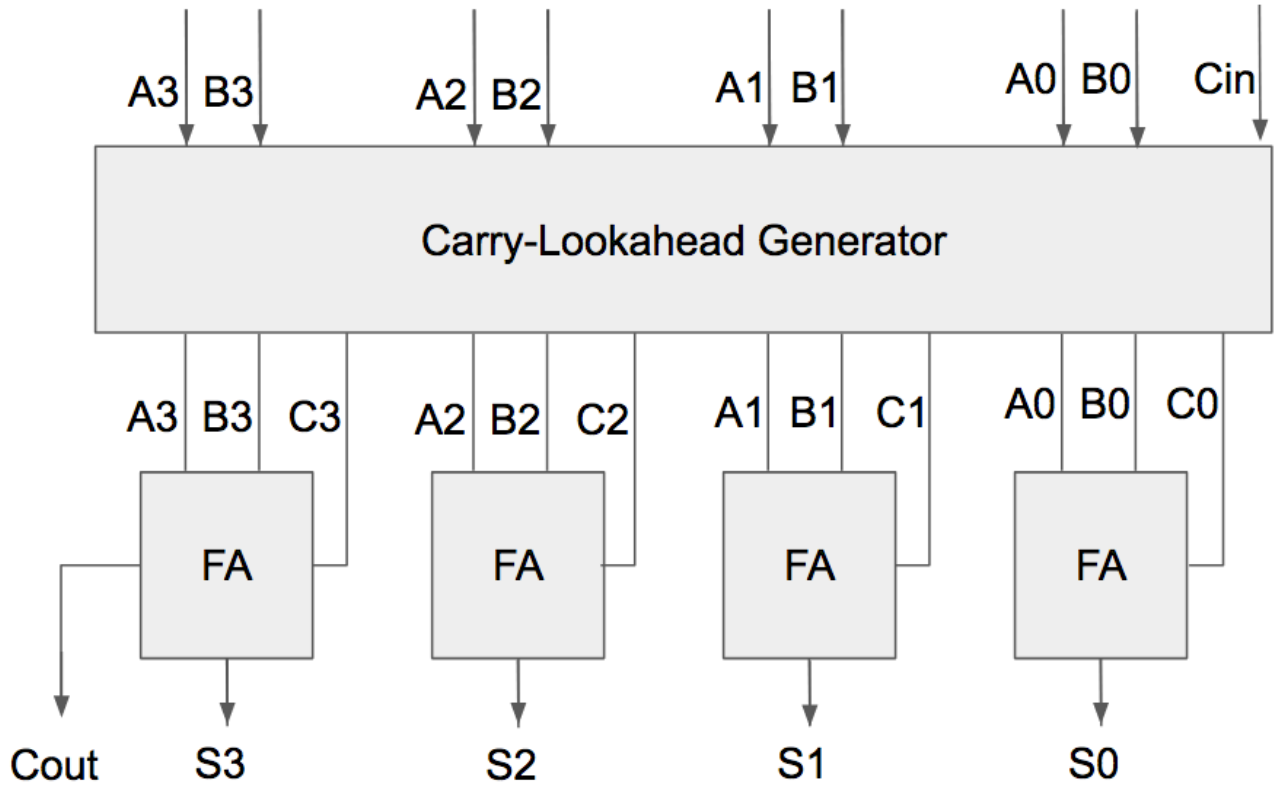
- 1-bit addition: 15ps
- 4-bit addition: 60ps
- 8-bit addition: 120ps
- 16-bit addition: 240ps
- 32-bit addition: 480ps
- 64-bit addition: 960ps = ~1 GHz

Faster Adder Options?

- What can be done to build a faster 64-bit adder?

How to break carry dependencies?

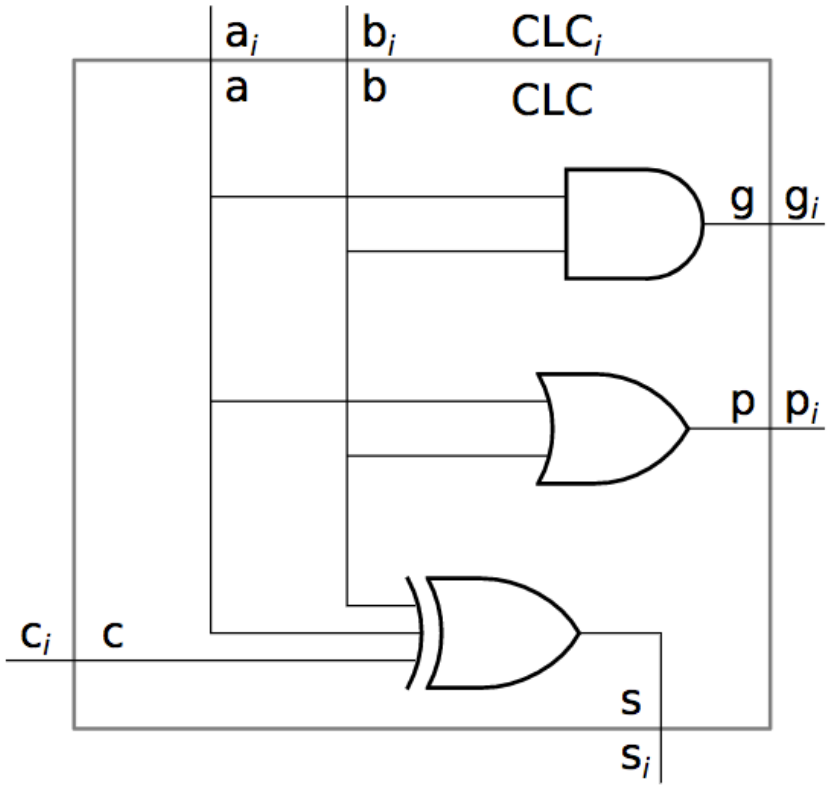
“Fast” Adder Concept



- Compute all carry bits in parallel
- Feed A,B, Carry directly into full adders in parallel
- Doesn’t work in practice. ☹

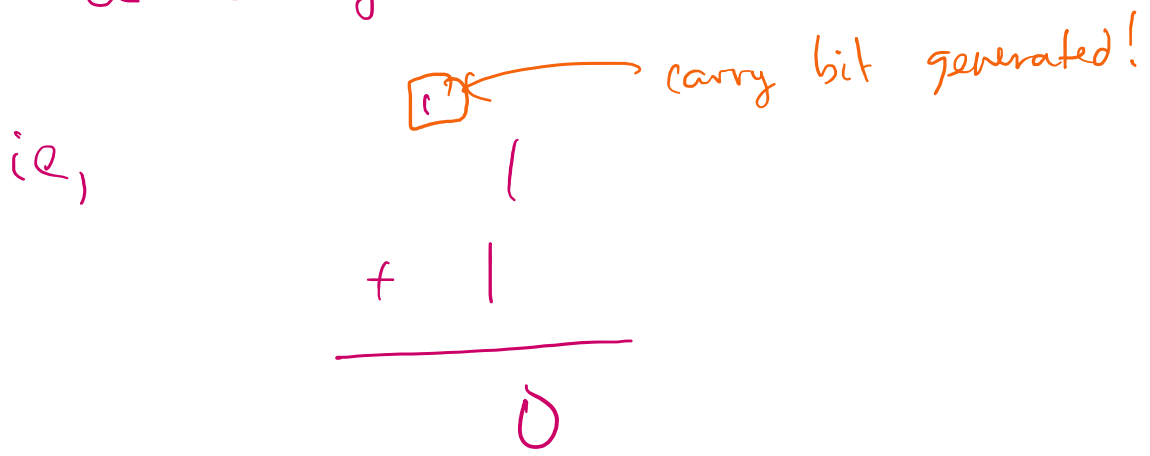
Rethinking Half-Adders

Rethinking Half-Adders



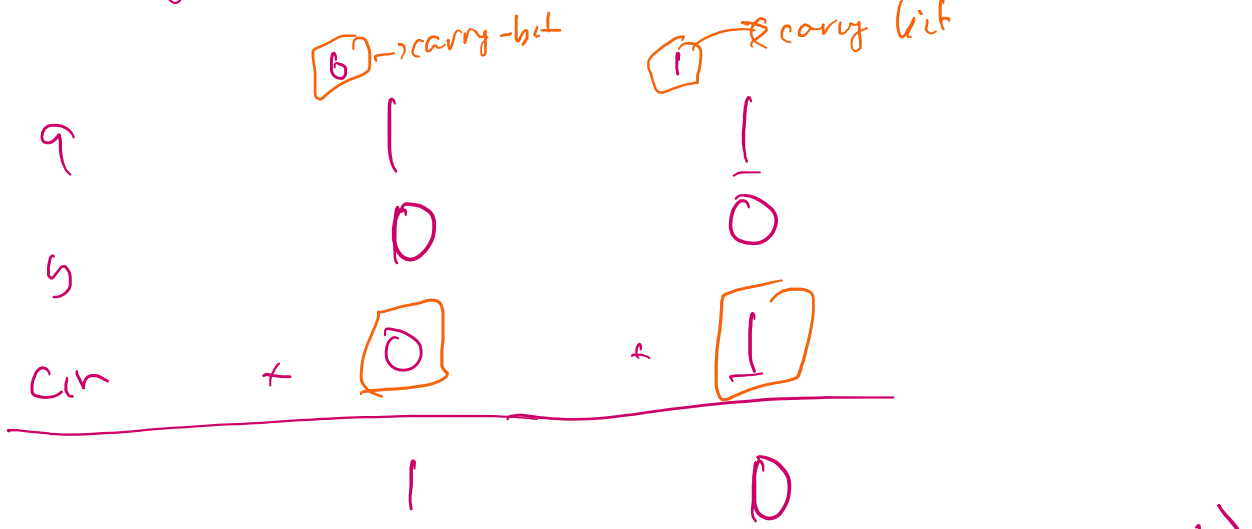
Generate

a_i & b_i are both 1, so they "generate" a carry-bit



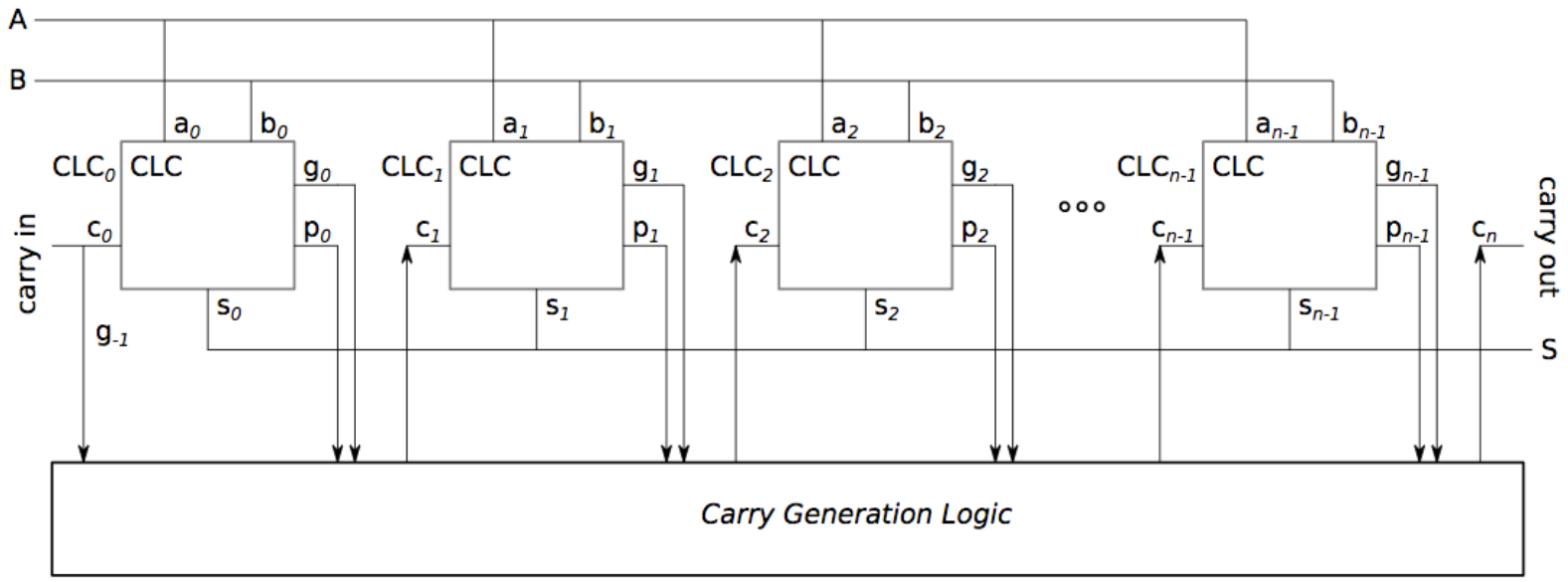
Propagate

a_i or b_i is 1, so will "propagate" carry-bit if c_{in} is also 1

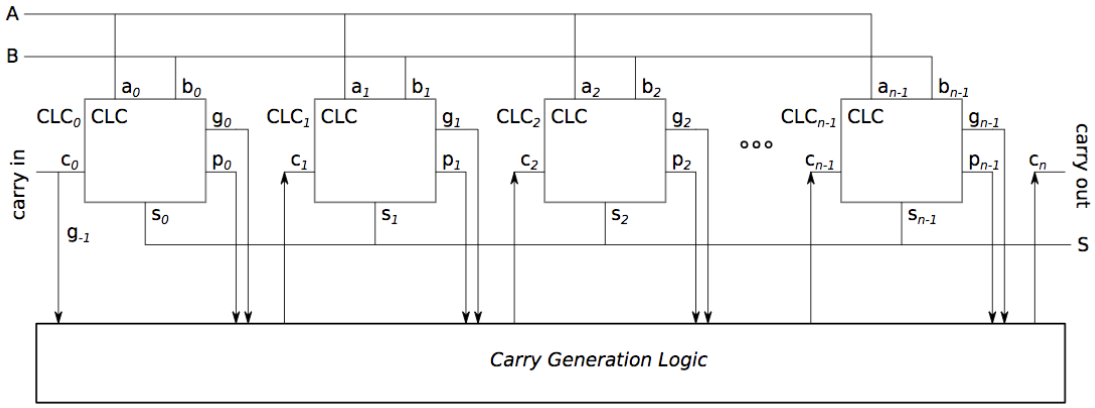


carry-bit is propigated (or passed through) by the "prop" signal

“Fast Adder” Idea



Computing Carries



Next Time

- Arithmetic Logic Units (ALUs)