Testing

ENGR 210 / CSCI B441

Addition / Subtraction

Andrew Lukefahr

Course Website

fangs-bootcamp.github.io

Write that down!

Announcements

- Work on P2

A simple testbench

```
`timescale 1ns/1ps
module BeltAlarm_tb();

logic k, p, s;
wire alarm;
BeltAlarm dut0( .k(k), .p(p), .s(s),.alarm(alarm) );

initial
begin
    k = 'h0; p = 'h0; s= 'h0;
    $monitor ("k:%b p:%b s:%b a:%b", k, p, s, alarm);
    #10
    assert(alarm == 'h0) else $fatal(1, "bad alarm");
    $display("@@@Passed");
end
endmodule
```

```
module BeltAlarm(
     input k, p, s,
     output alarm
);

      assign alarm = k & p & ~s;
endmodule
```

Submodule Example

```
`timescale 1 ns/1 ns

module BeltAlarm(
      input k, p, s,
      output alarm
);

    assign alarm = k & p & ~s;

endmodule
```

```
`timescale 1 ns/1 ns

module TwoBeltAlarm(
      input k, st_pas, sb_pas,
      input st_drv, sb_drv
      output alarm
);
      wire al_pas, al_drv; //intermediate wires

      //submodules, two different examples
      BeltAlarm ba_drv(k, st_drv, sb_drv, al_drv); //no named arguments
      BeltAlarm ba_pas(.k(k), .p(st_pas),
            .s(sb_pas), .alarm(al_pas)); // with named arguments

      assign alarm = al_pas | al_drv;
endmodule
```

2-BeltAlarm Task

```
task checkAlarm(
    input kV, stPasV, sbPasV,
    input stDrvV, sbDrvV,
    input alarmV
    );

    k = kV; stPas=stPasV, sbPas=sbPasV;
    stDrv = stDrvV; sbDrv = sbDrvV;
    #10
    assert(alarm == alarmV) else
        $fatal (1, "bad alarm, expected:%b got:%b",
                    alarmV, alarm);
endtask
```

```
module TwoBeltAlarm(
     input k, st_pas, sb_pas,
     input st_drv, sb_drv
     output alarm
);
     wire al_pas, al_drv;

     BeltAlarm ba_drv(k, st_drv, sb_drv, al_drv);
     BeltAlarm ba_pas(.k(k), .p(st_pas),
          .s(sb_pas), .alarm(al_pas));

     assign alarm = al_pas | al_drv;
endmodule
```

```
initial begin
    k = 0; st_pas = 'b0; sb_pas = 'b0;
    st_drv = 'b0; sb_drv = 'h0;
    #10
    assert(alarm == 'h0) else $fatal(1, "bad alarm");
    #10
    checkAlarm(0,'b0,'h0, 'h0, 'h0, 'h0);
    for (int i = 0; i < 32; ++i) begin
        $display("i:%d [%b]", i, i[4:0]);

        if ( (i == 18) | (i == 22) | (i == 30)) // driver
            checkAlarm( i[4], i[3], i[2], i[1], i[0], 'h1);
        else if ( (i == 24 ) | (i == 25) | (i==27)) //passenger
            checkAlarm( i[4], i[3], i[2], i[1], i[0], 'h1);
        else if ( (i==26) ) //both
            checkAlarm( i[4], i[3], i[2], i[1], i[0], 'h1);
        else
            checkAlarm( i[4], i[3], i[2], i[1], i[0], 'h0);
    end

    $display("@@@Passed");
end
```

```
task checkAlarm(
    input kV, stPasV, sbPasV,
    input stDrvV, sbDrvV,
    input alarmV
    );

    k = kV; stPas=stPasV, sbPas=sbPasV;
    stDrv = stDrvV; sbDrv = sbDrvV;
    #10
    assert(alarm == alarmV) else
        $fatal (1, "bad alarm, expected:%b got:%b",
                alarmV, alarm);
endtask
```

For Loops in Testbenches

- You <u>can</u> write for-loops in your testbenches

```
module for_loop_simulation ();
  logic [7:0] r_Data; // Create 8 bit value

  initial begin
      for (int ii=0; ii<6; ii=ii+1) begin
        r_Data = ii;
        $display("Time %d: r_Data is %b", $time, r_Data);
        #10;
      end
  end
endmodule
```

- Please ***<u>no for-loops in your synthesizable code (yet)!</u>***

- https://www.nandland.com/vhdl/examples/example-for-loop.html

Arrays in Verilog

- Bundle multiple wires together to form an array.

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

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

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?

- `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

Constants in Verilog

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

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

endmodule
```

`wire` vs `logic`

- ***`wire`***
  - Only used with '`assign`' and module outputs
  - Boolean combination of inputs
  - Can never hold state

- ***`logic`***
  - Used with '`always`' and module outputs
  - Can be Boolean combination of inputs
  - Can hold state (but doesn't have to)

always_comb Blocks

```
wire foo;
assign foo = x & y | z;
```

… is equivalent to …

```
wire foo = x & y | z;
```

… is equivalent to …

```
logic foo;
always_comb //comb-inational
     foo = x & y | z;
```

always_comb blocks with if

```
module decoder (
        input [1:0] sel,
        output logic [3:0] out
        );
    always_comb begin
        if (sel == 2'b00) begin
                out = 4'b0001;
        end else if (sel == 2'b01) begin
                out = 4'b0010;
        end else if (sel == 2'b10) begin
                out = 4'b0100;
        end else if (sel == 2'b11) begin
                out = 4'b1000;
        end
    end
endmodule
```

always_comb with case

```
module decoder (
        input [1:0] sel,
        output logic [3:0] out
        );

    always_comb begin
        case(sel)
                2'b00: out=4'b0001;
                2'b01: out=4'b0010;
                2'b10: out=4'b0100;
                2'b11: out=4'b1000;
        endcase
    end

endmodule
```

always_comb with case

```
module decoder (
        input [1:0] sel,
        output logic [3:0] out
        );

    always_comb begin
        case(sel)
                2'b00: out=4'b0001;
                2'b01: out=4'b0010;
                2'b10: out=4'b0100;
                                    // what about sel==2'b11?
        endcase
    end


endmodule
```

always_comb with case

```
module decoder (
        input [1:0] sel,
        output logic [3:0] out
        );

    always_comb begin
        out = 4'b0000; //default
        case(sel)
                2'b00: out=4'b0001;
                2'b01: out=4'b0010;
                2'b10: out=4'b0100;
                                    // what about sel==2'b11?
        endcase
    end

endmodule
```

Always specify defaults for **always_comb**!

* add logic after class

always_ff
next time

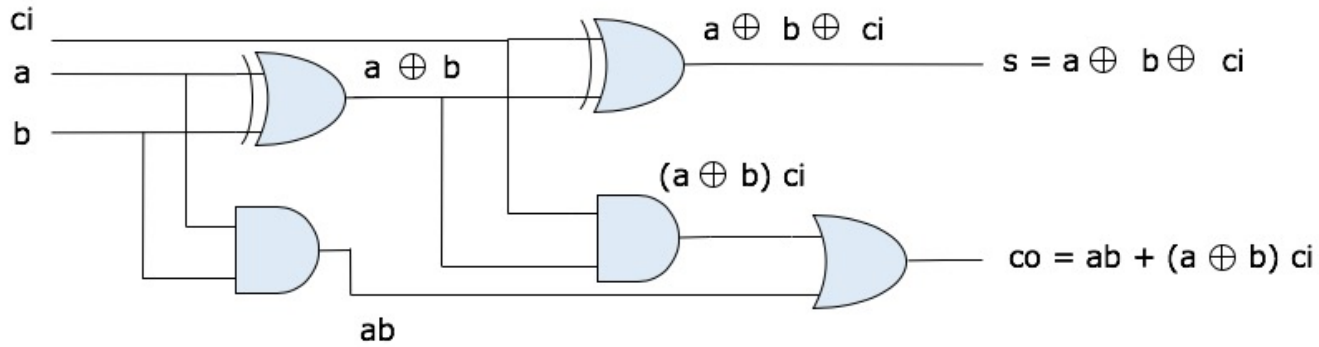
always_comb with case

```
module decoder (
        input [1:0] sel,
        output logic [3:0] out
        );

    always_comb begin
        out = 4'b0000; //default
        case(sel)
                2'b00: out=4'b0001;
                2'b01: out=4'b0010;
                2'b10: out=4'b0100;
                default: out = 4'b0000; // what about sel==2'b11?
        endcase
    end

endmodule
```

Always specify defaults for **always_comb**!

"Warning: Inferred Latch"

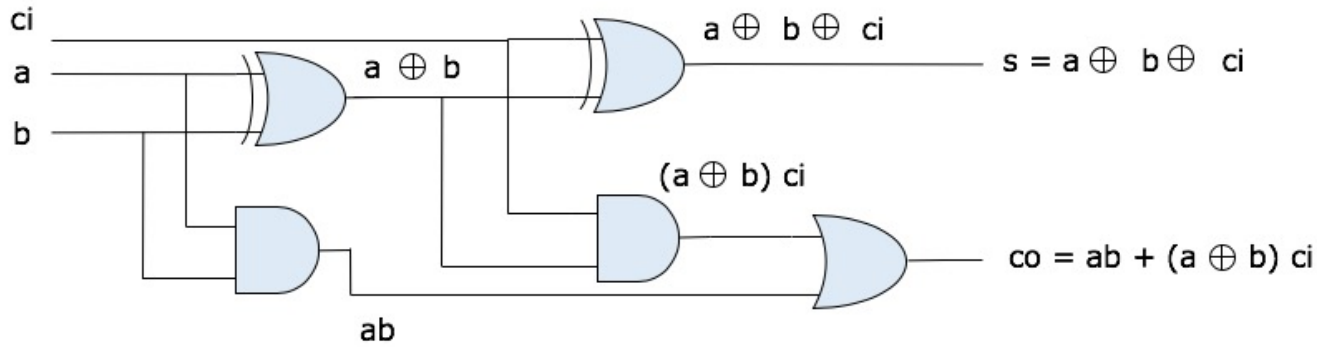
1-Bit “Full” Adder to ‘always_comb’




```
module FullAddr (
   input a,b,ci,
   output s, co
   );


   assign s = a ^ b ^ ci;
   assign co = (a & b) |
               (( a ^ b) & ci);


endmodule
```

```
module FullAddr (









endmodule
```

1-Bit “Full” Adder to ‘always_comb’

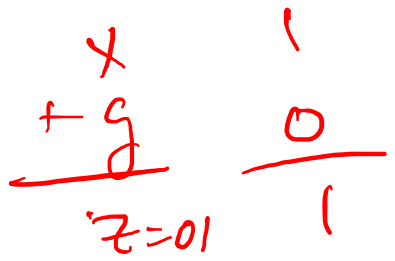



```
module FullAddr (
   input a,b,ci,
   output s, co
   );


   assign s = a ^ b ^ ci;
   assign co = (a & b) |
               (( a ^ b) & ci);


endmodule
```

```
module FullAddr (
    input a,b,ci,
    output logic s, co
    );

   always_comb begin
       s = a ^ b ^ ci;
       co = (a & b) |
          (( a ^ b) & ci);
   end


endmodule
```

Addition / Subtraction

Binary Addition



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

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

x		y	math (Dec)	Carry	Sum
0	+	0	0 + 0 = 0	0	0
0	+	1	0 + 1 = 1	0	1
1	+	0	1 + 0 = 1	0	1
1	+	1	1 + 1 = 2	1	0

Half Adder

x	+ y	= carry	sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

XOR

carry = x · y // boolean

x & y // verilog

sum = $\bar{x} \cdot y + x \cdot \bar{y}$ // boolean

(~x & y) | (x & ~y) // verilog

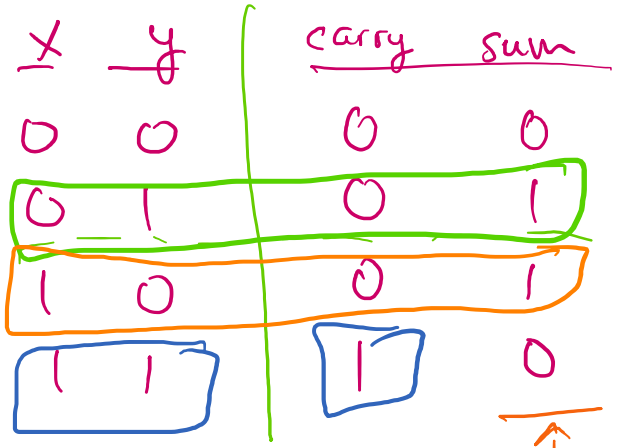
↑ (correct)

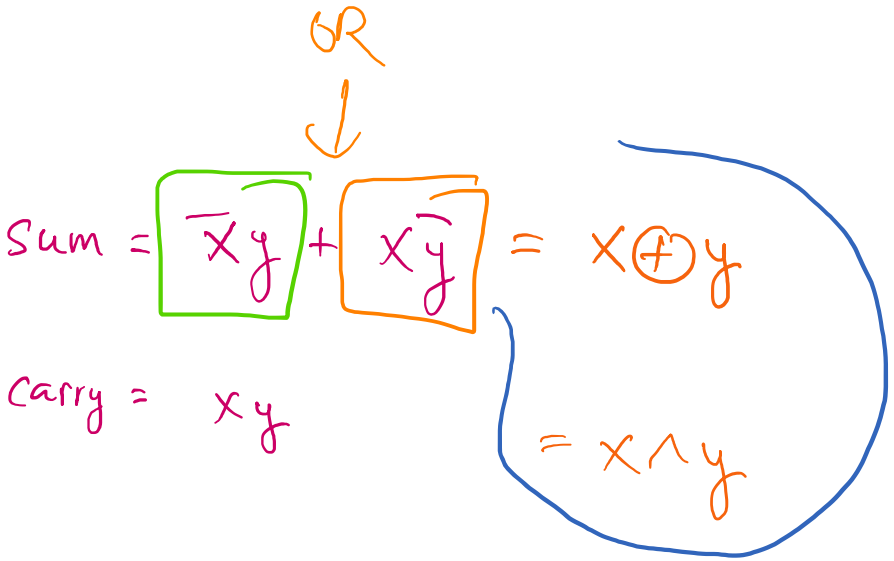
simplify

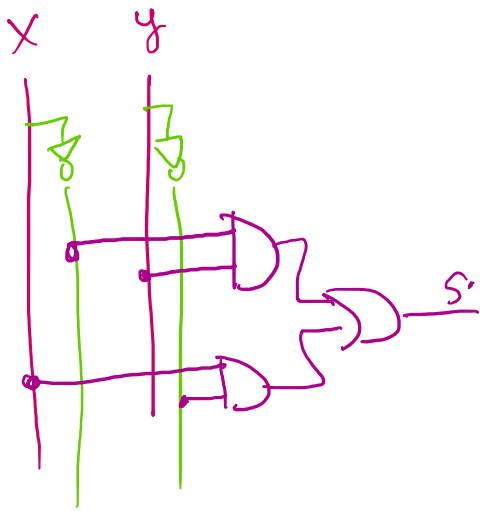
Sum = x ⊕ y // Boolean

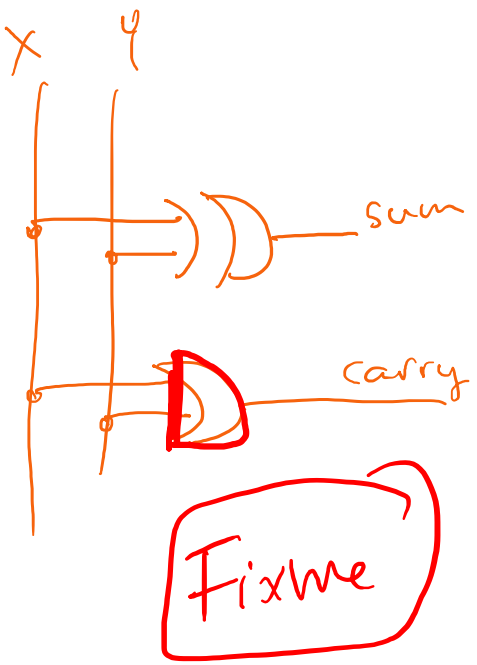
Sum = x ^ y; // verilog

Half Adder


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

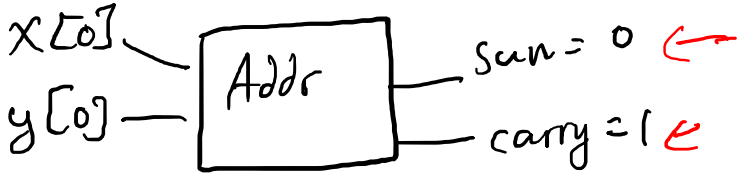

OR
Sum = $\bar{x}y + x\bar{y}$ = $x \oplus y$
carry = xy
= $x \wedge y$


x
y
S


x
y
sum
carry
Fixme

Binary Addition

x	y	carry	sum
1	1	1	0



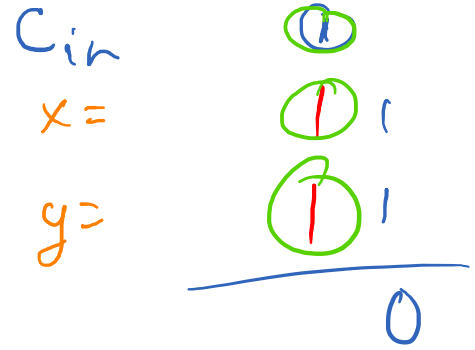
- What if x and y are 2-bits each?

x = 3, y = 3

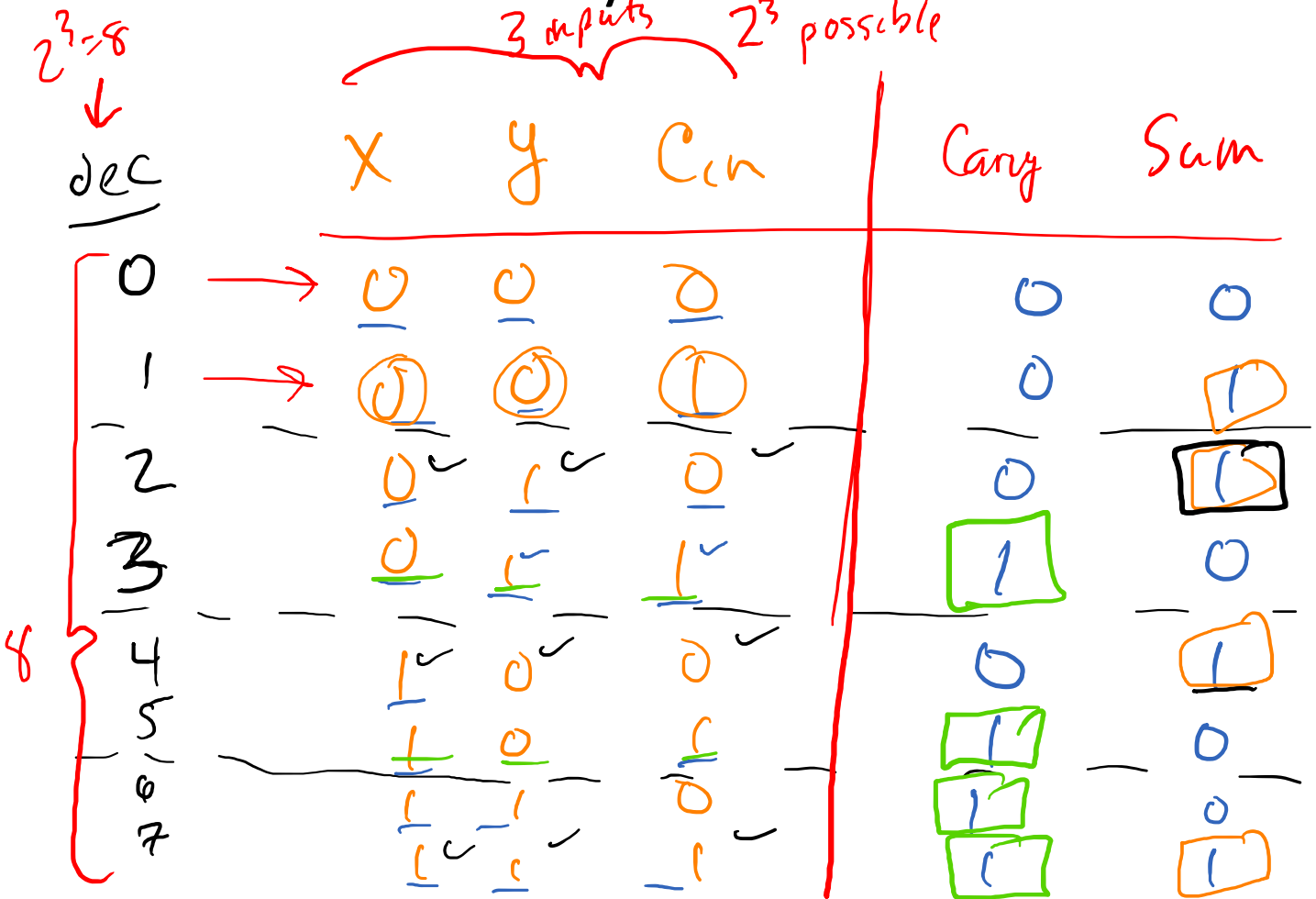
```
  3  =   01
+ 3      +11
-----    ---
  6      110
```

```
    1 (1)
  0  1 1
+ 0  1 1
---------
  1 1 0
```

Binary Addition



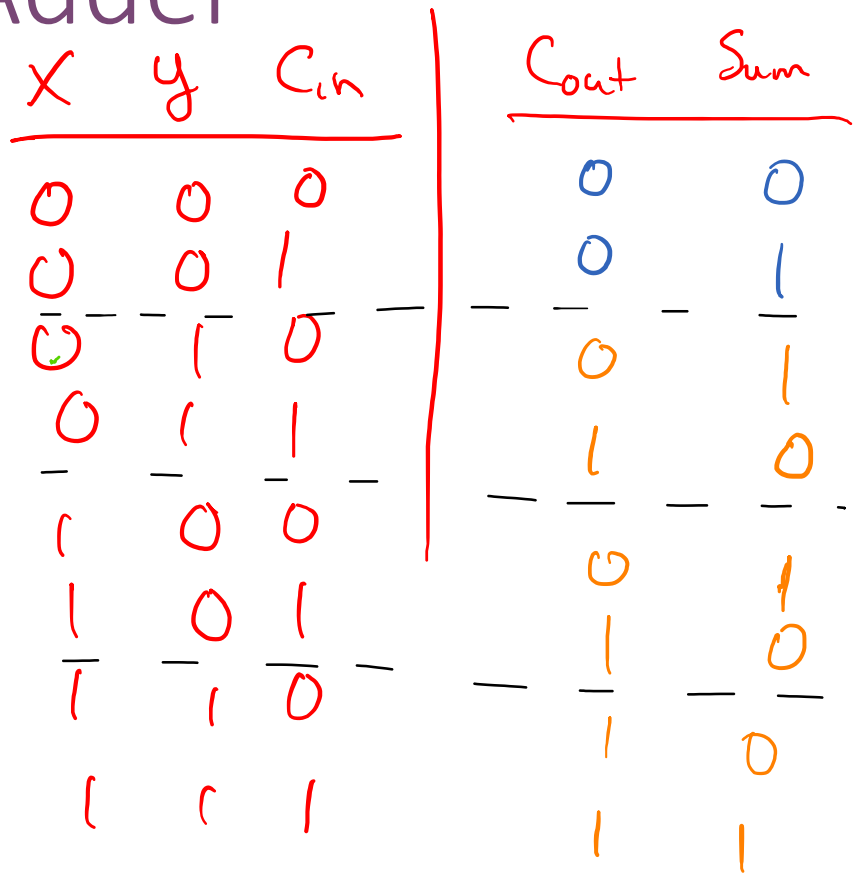
- What if x and y are 2-bits each?



$$Carry = \bar{x} \cdot y \cdot C_{in} + x \cdot \bar{y} \cdot C_{in} + x \cdot y \cdot \overline{C_{in}} + x \cdot y \cdot C_{in}$$

$$Sum = \bar{x} \cdot \bar{y} \cdot C_{in} + \bar{x} \cdot y \cdot \overline{C_{in}} + x \cdot \bar{y} \cdot \overline{C_{in}} + x \cdot y \cdot C_{in}$$

Full Adder

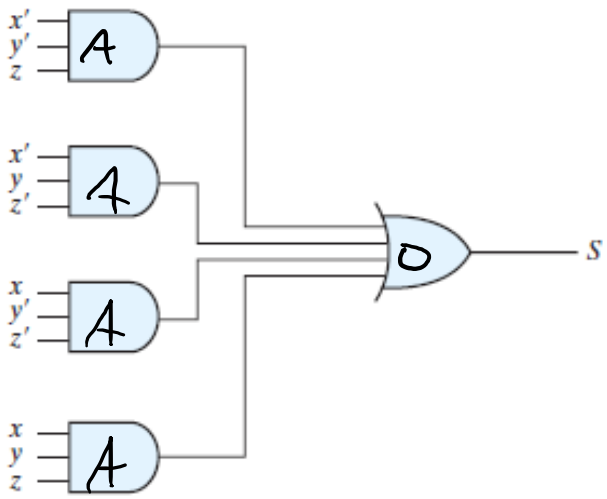

X y Cin
Cout Sum
0 0 0 | 0 0
0 0 1 | 0 1
0 1 0 | 0 1
0 1 1 | 1 0
1 0 0 | 0 1
1 0 1 | 1 0
1 1 0 | 1 0
1 1 1 | 1 1

P2: use "+" in Verilog

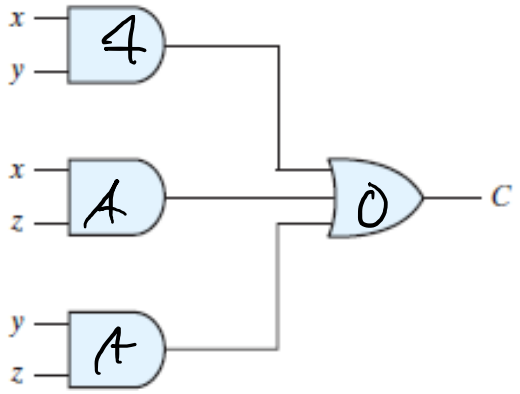
Full Adder

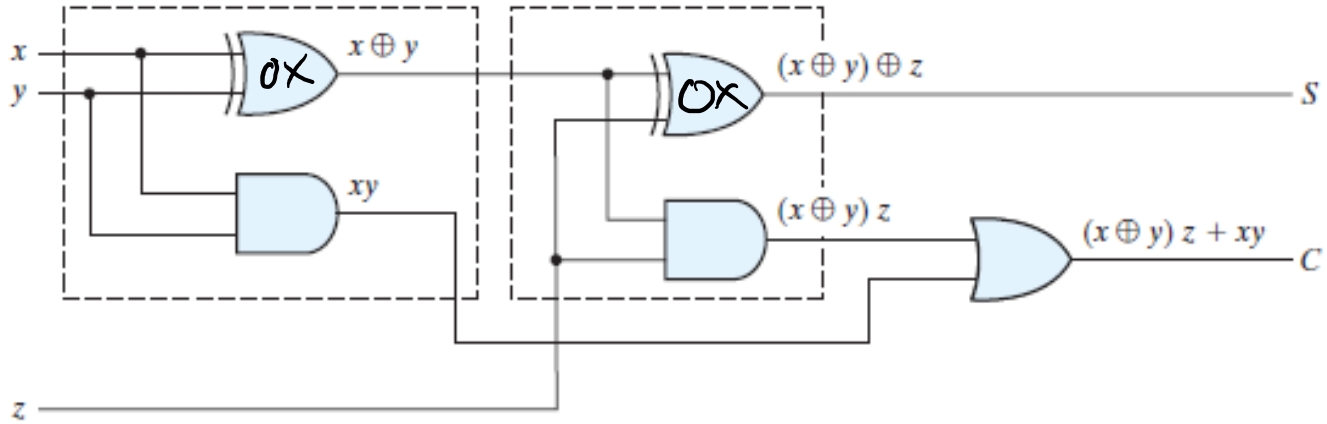
i	x	y	z	C	S
0	0	0	0	0	0
1	0	0	1	0	1
2	0	1	0	0	1
3	0	1	1	1	0
4	1	0	0	0	1
5	1	0	1	1	0
6	1	1	0	1	0
7	1	1	1	1	1

Sum

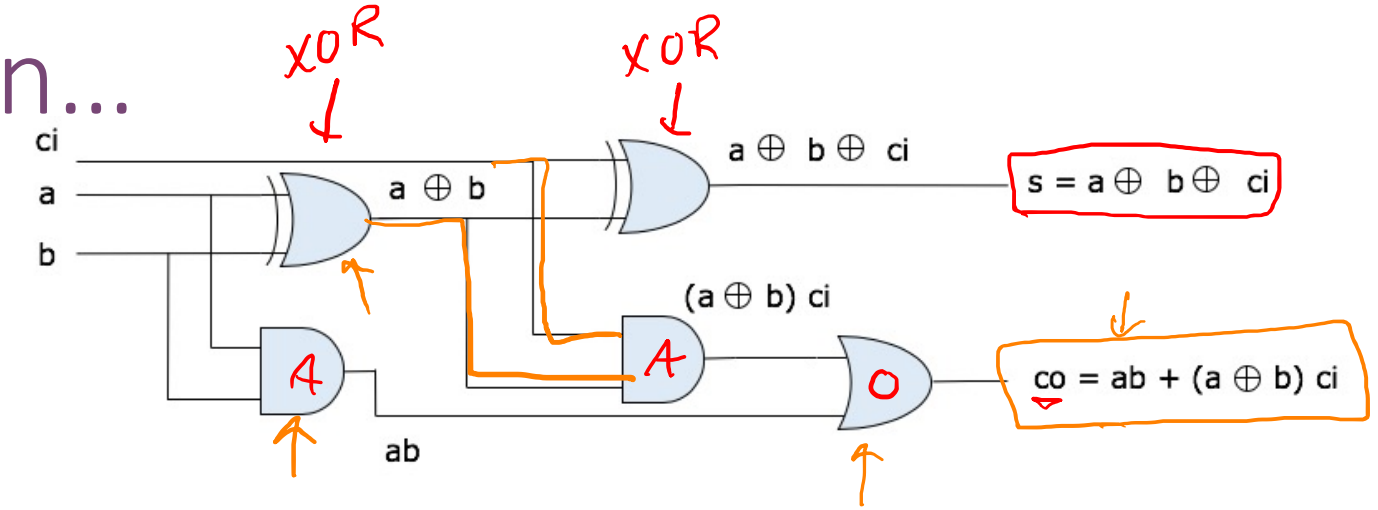


Carry





Your Turn…



Implement the circuit above in Verilog.

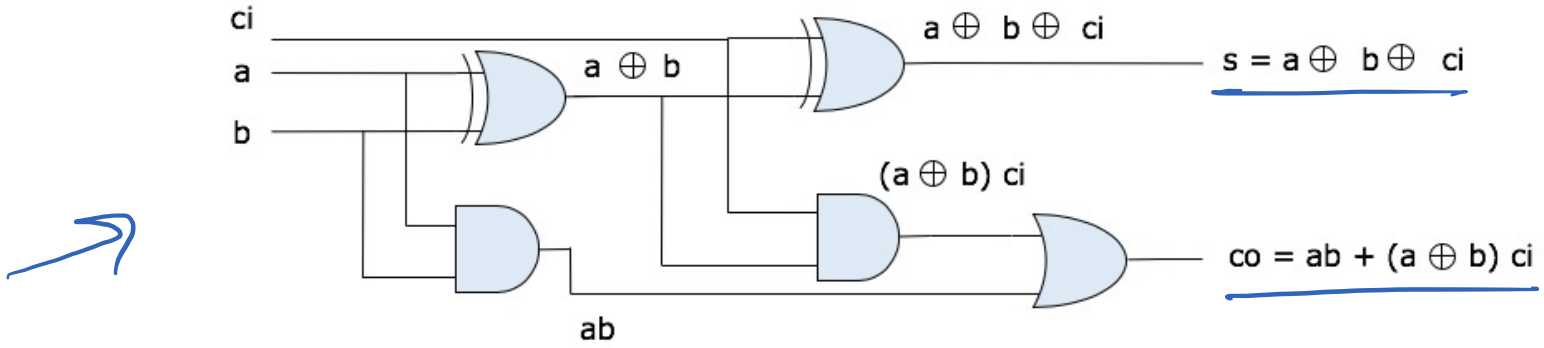
- Hint: XOR in Verilog is '^'

```
module FullAdder (
    input a,b,ci,
    output s, co
    );

    // Your Code Here!


endmodule
```

Ci = Carry_in
Co = Carry out

assign s = a ^ b ^ Ci ;
assign Co = (a & b) | ((a ^ b) & ci);

1-Bit “Full” Adder




```
module FullAddr (
   input a,b,ci,
   output s, co
   );

   assign s = a ^ b ^ ci;
   assign co = (a & b) | ( a ^ b) & ci;

endmodule
```

Ripple-Carry Adder

C_4 C_3 C_2 C_1

$X = \quad X_3 \; X_2 \; X_1 \; X_0$

$y = + \quad y_3 \; y_2 \; y_1 \; y_0$

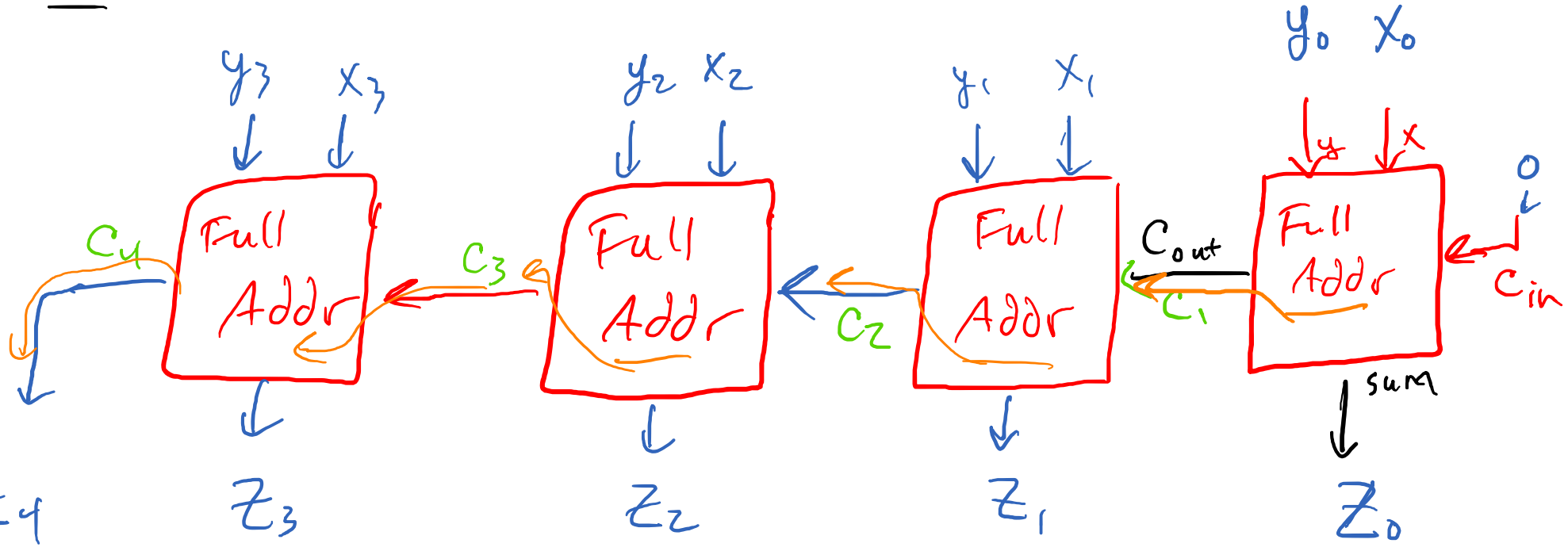
$z_4 \; z_3 \; z_2 \; z_1 \; z_0$

$C_1 \quad 1$

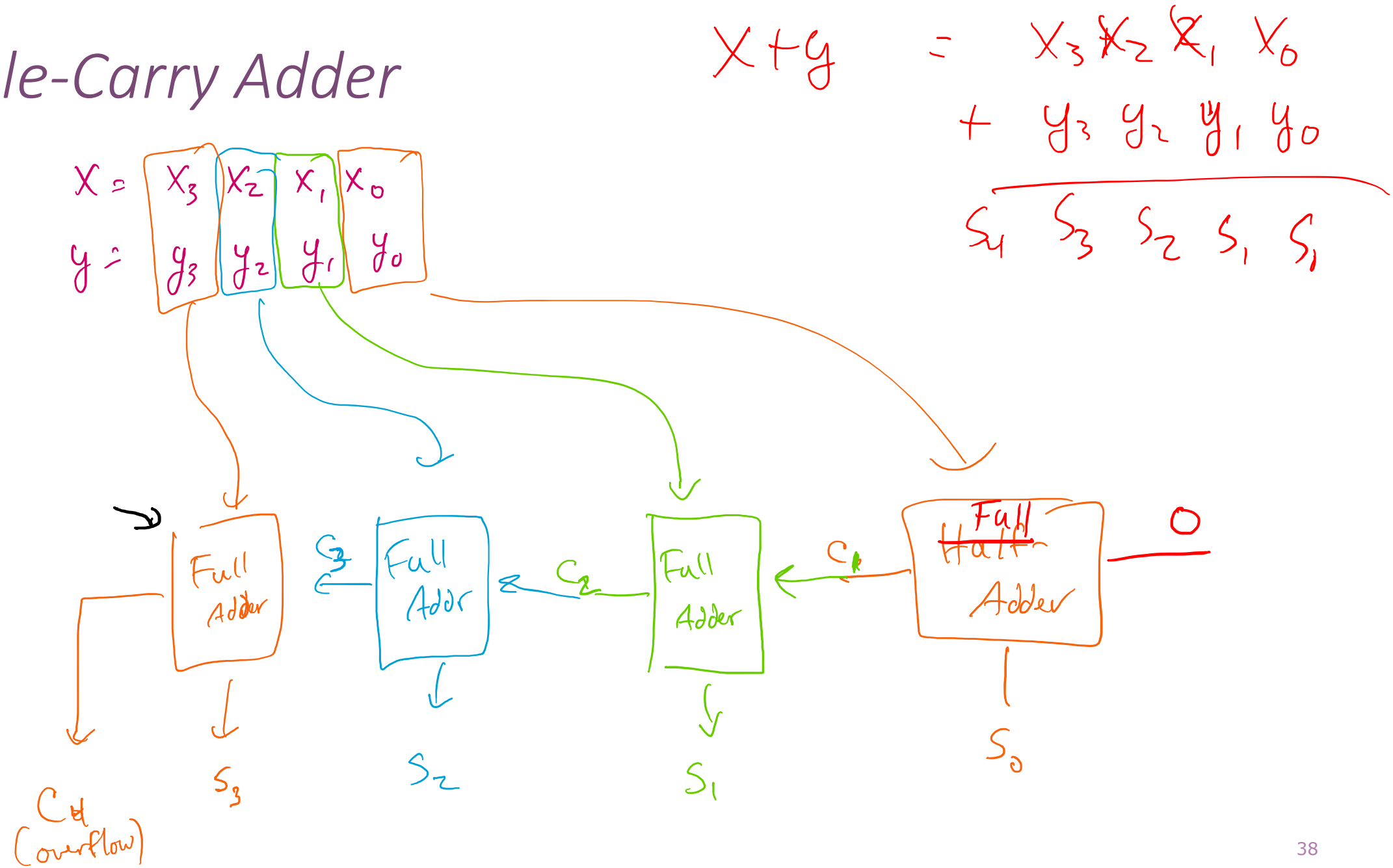
$X_1 = \quad 1$

$y_1 = + \quad 0$

$1 \; 0$



Ripple-Carry Adder

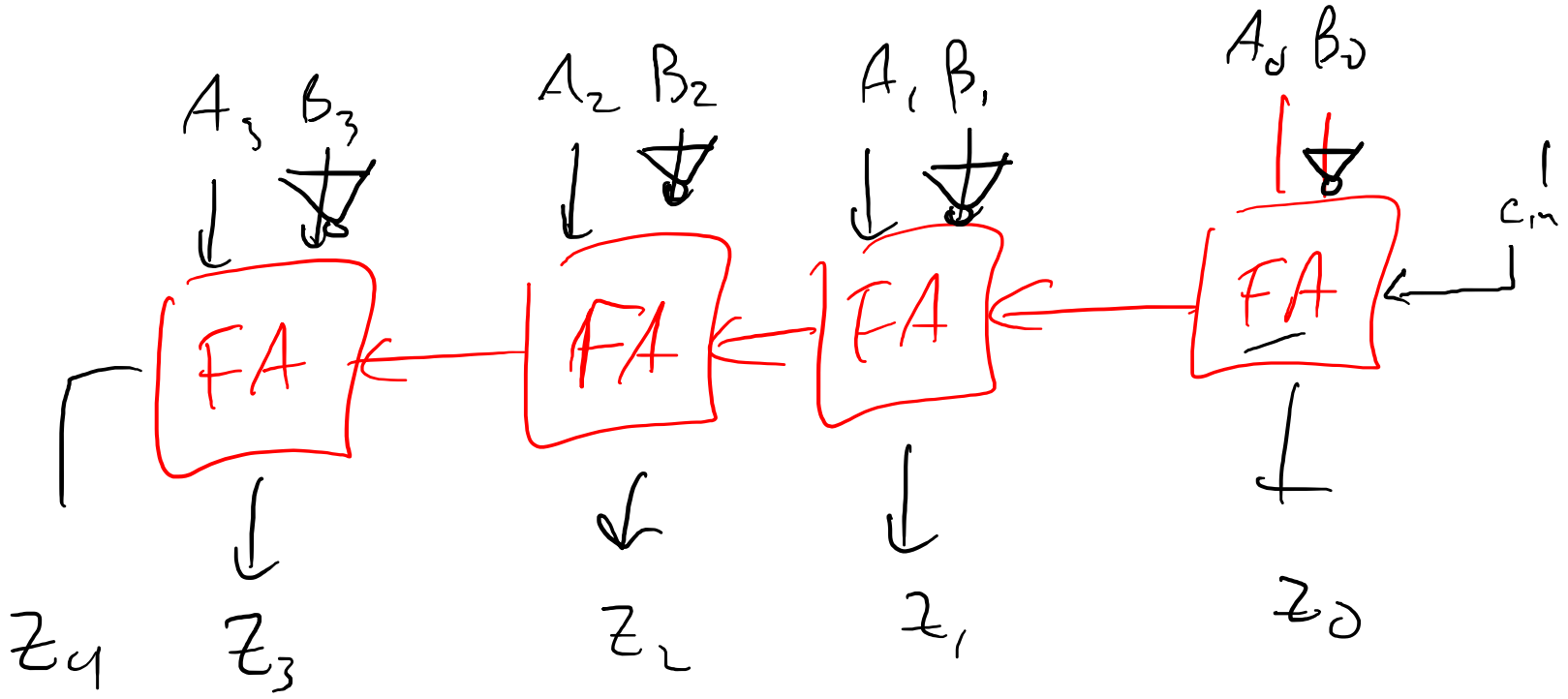


Subtraction with Adders?

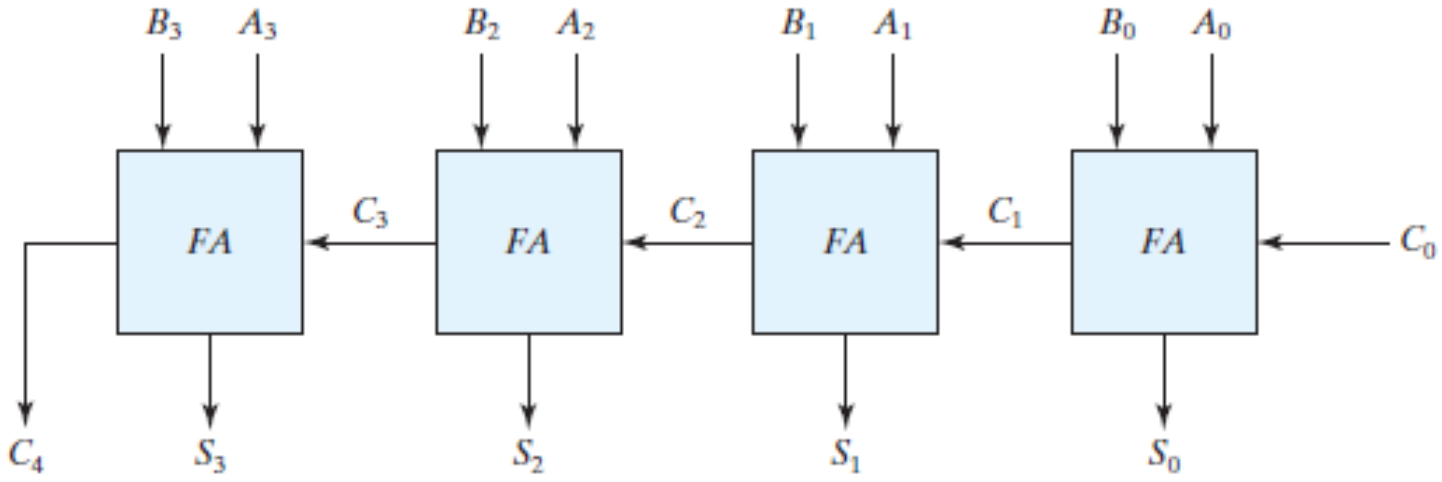
$A - B = A + (-B) =$

$A + \sim B + 1$

- We've done `A+B`, what about `A-B`?

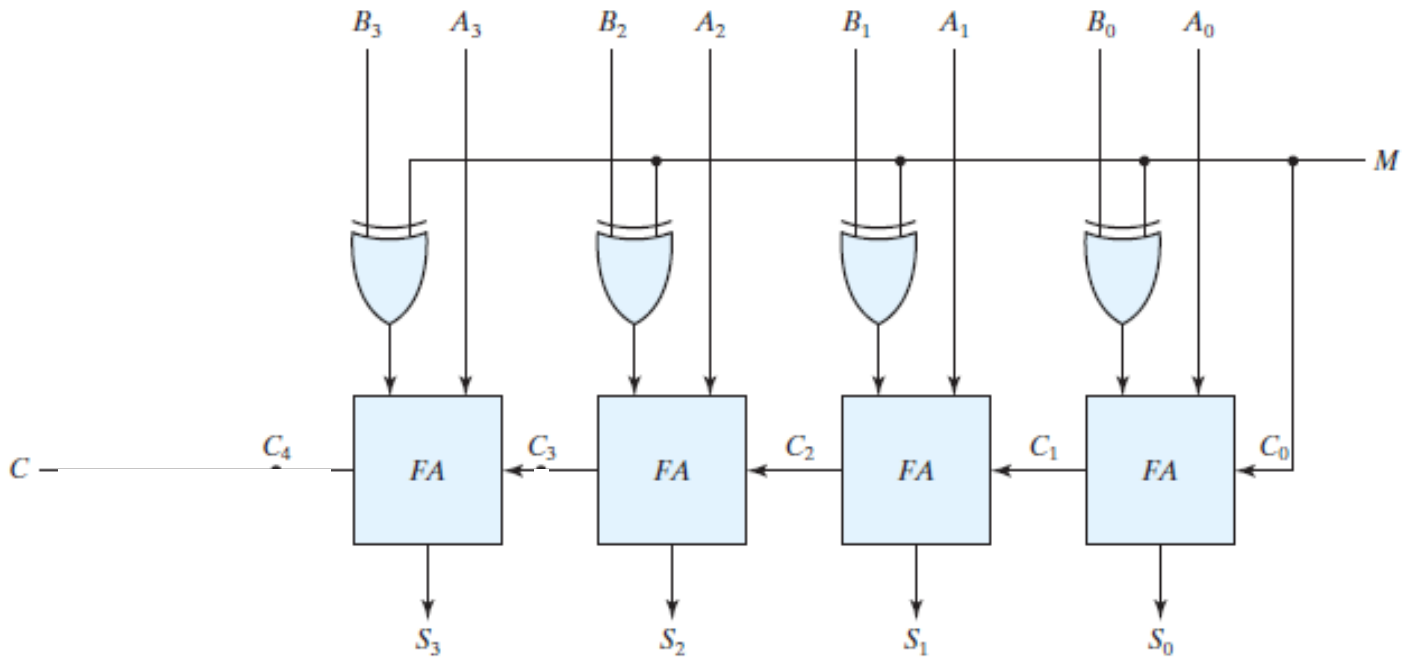


Subtraction with Adders?



Adder/Subtractor

- Mode input:
  - If $M = 0$, then $S = A + B$, the circuit performs addition
  - If $M = 1$, then $S = A + \overline{B} + 1$, the circuit performs subtraction



Overflow for signed numbers?

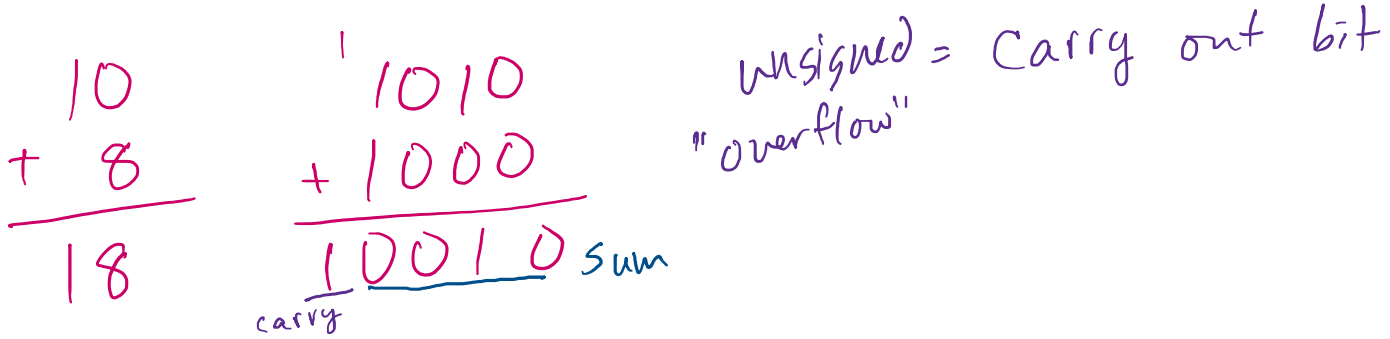
- Unsigned

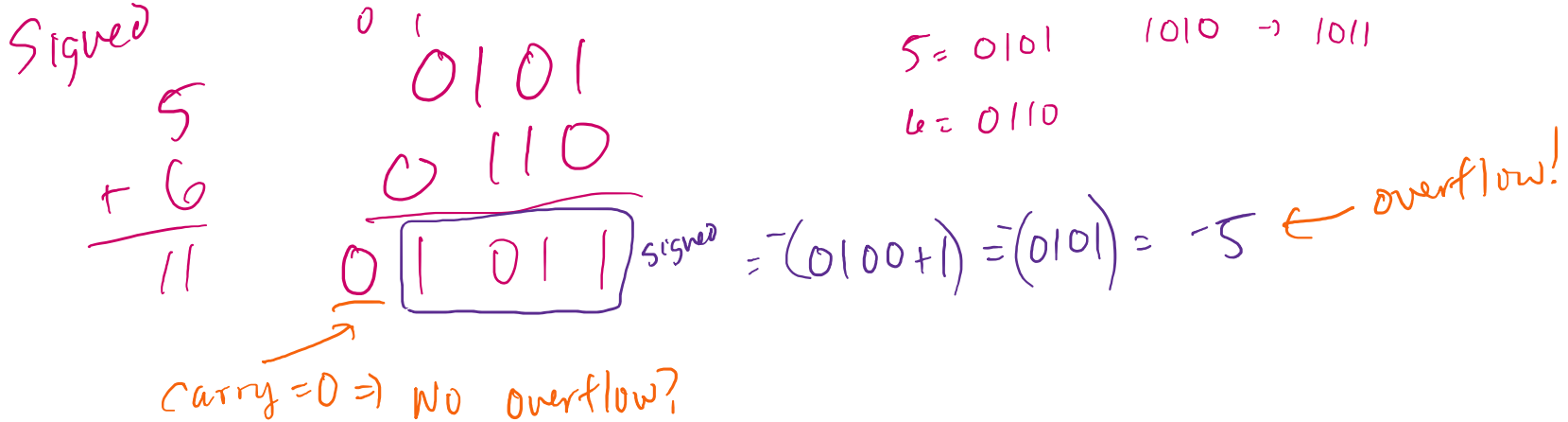
$$\begin{array}{r} 10 \\ +\ 8 \\ \hline 18 \end{array}$$

- Signed

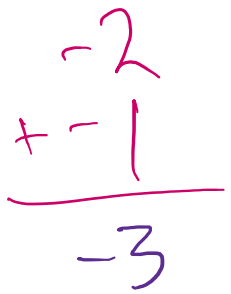
$$\begin{array}{r} 5 \\ +\ 6 \\ \hline 11 \end{array}$$

Overflow for signed numbers?



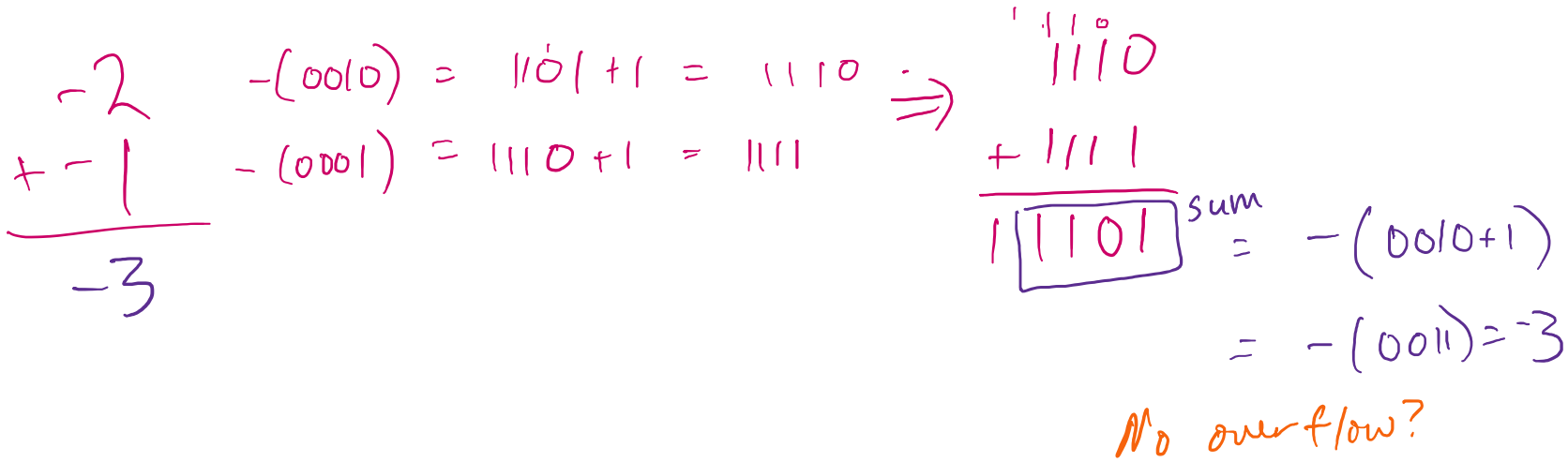


Overflow for signed numbers?




```
   +2
+  -1
-------
   +1
```

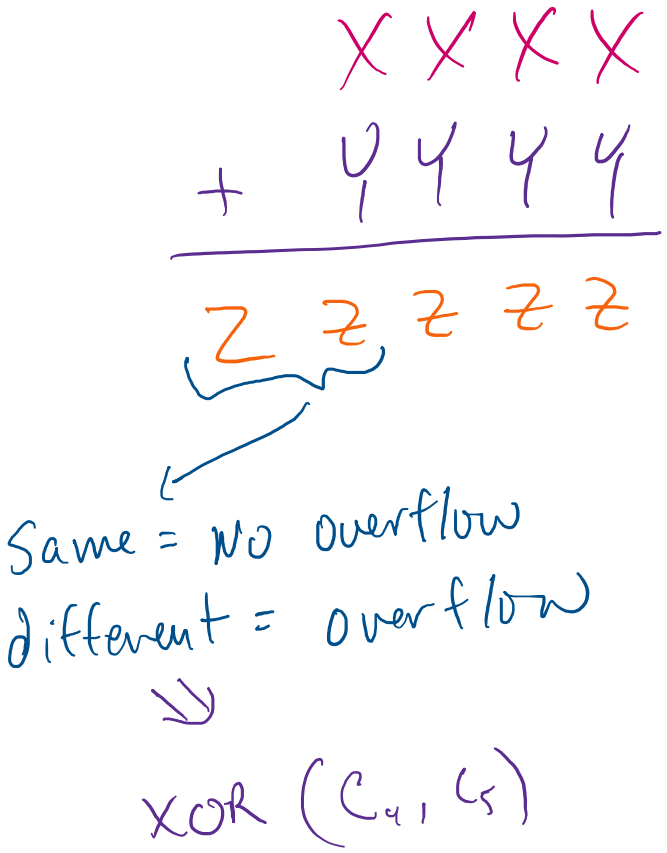
Overflow for signed numbers?




```
  +2.      0010.    0010
+ -1.  +-(0001) + 1111
-------  ------    ----
   +1              10001 <- no overflow
```

Overflow for signed numbers

Stopped here!



Signed numbers Only!

unsigned = regular carry

Overflow detection

- When two numbers with *n* digits each are added and the sum is a number occupying *n* + 1 digits, we say that an overflow occurred.
- The detection of an overflow after the addition of two binary numbers depends on whether the numbers are considered to be signed or unsigned.
- When two unsigned numbers are added, an overflow is detected from the end carry out of the most significant position.
- In case of signed numbers, two details are important:
  the leftmost bit always represents the sign,
  negative numbers are in 2's-complement form.
- When two signed numbers are added:
  the sign bit is treated as part of the number
  the end carry does not indicate an overflow.

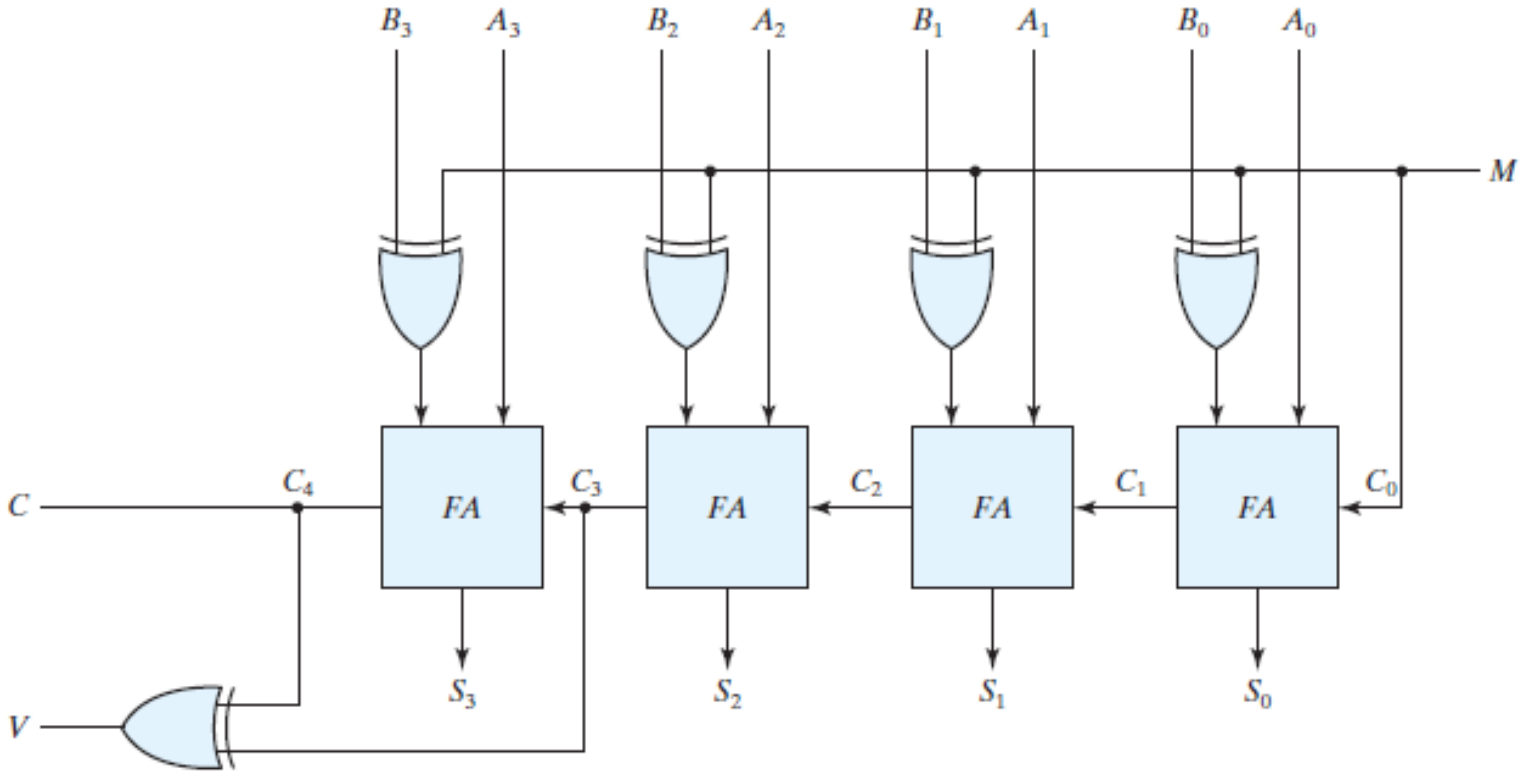
Overflow detection

- An overflow cannot occur after an addition if one number is positive and the other is negative, since adding a positive number to a negative number produces a result whose magnitude is smaller than the larger of the two original numbers.
- An overflow may occur if the two numbers added are both positive or both negative.
- An overflow condition can be detected by observing the carry into the sign bit position and the carry out of the sign bit position.

  If these two carries are equal, there was no overflow.

  If these two carries are not equal, an overflow has occurred.
- If the two carries are applied to an exclusive-OR gate, an overflow is detected when the output of the gate is equal to 1.

Adder with overflow detection



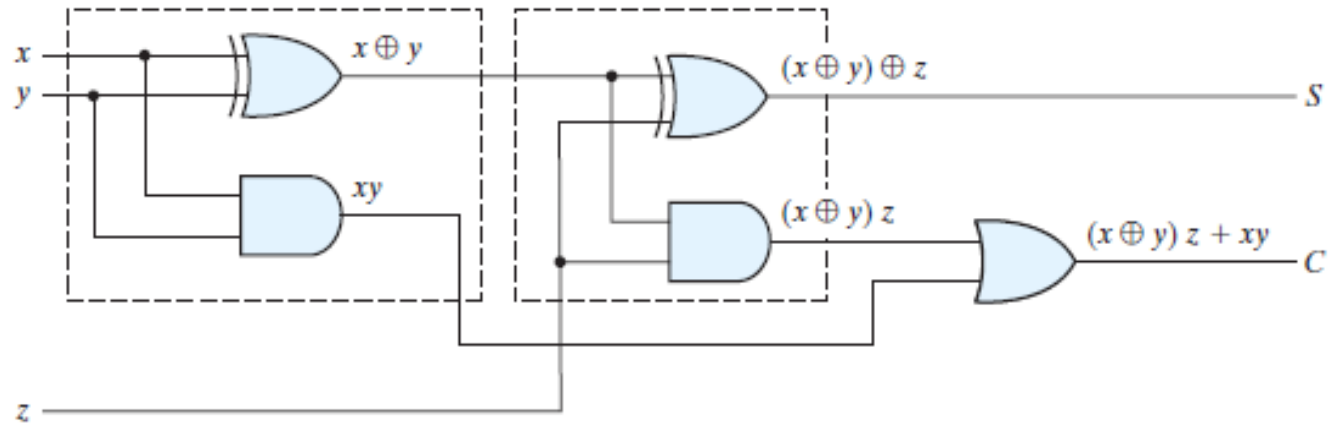
Gate Delay

- Gates are not magic, they are physical
- Takes time for changes flow through
- Assume 5ps (5E-12) / gate

- How fast can we update our adder?

Full Adder Gate Delay

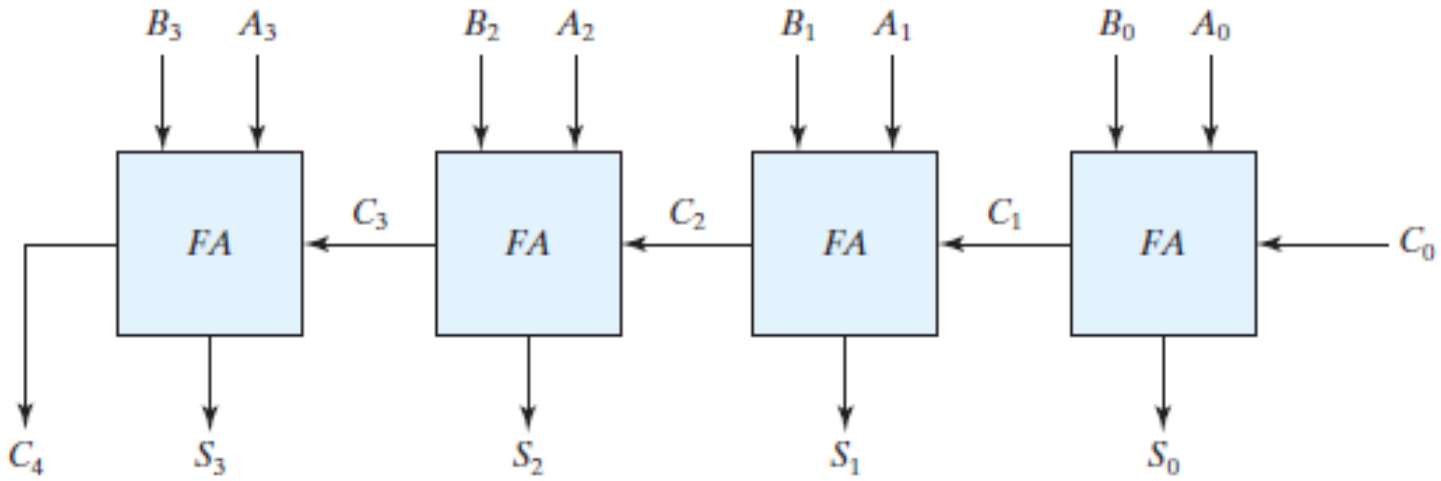
- Assume 5ps/gate



- 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?

Next Time

- Latches / Flip-Flops