ENGR 210 / CSCI B441

Testbenches/
Addition / Subtraction

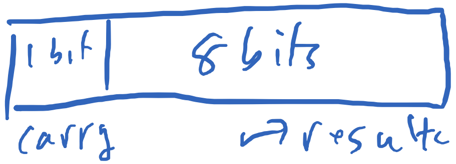
Andrew Lukefahr

Announcements

- ~~P5 is due Friday~~
- P6 is ~~out~~ due Friday

→ carry / overflow

→ carry tip: 9-bit addition



→ overflow tip: check the book,
book online for free.

A simple testbench

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

logic k, p, s;
logic alarm; //or ‘wire’
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
```

BeltAlarm Testing

```
initial
begin
    k = 'h0; p = 'h0; s= 'h0;
    $monitor ("k:%b p:%b s:%b a:%b",
                   k, p, s, alarm);
    checkAlarm('h0,'h0,'h0, 'h0);
    checkAlarm('h0,'h0,'h1, 'h0);
    checkAlarm('h0,'h1,'h0, 'h0);
    checkAlarm('h0,'h1,'h1, 'h0);
    checkAlarm('h1,'h0,'h0, 'h0);
    checkAlarm('h1,'h0,'h1, 'h0);
    checkAlarm('h1,'h1,'h0, 'h1);
    checkAlarm('h1,'h1,'h1, 'h0);
    $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
```

Submodule Example

```
 ‘timescale 1 ns/1 ns

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

```
 ‘timescale 1 ns/1 ns

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

    assign alarm = k & p & ~s;

endmodule
```

2-BeltAlarm testbench

```
`timescale 1ns/1ps
module tb();
logic k, st_pas, sb_pas, st_drv, sb_drv, alarm;

TwoBeltAlarm dut0(    .k(k), .st_pas(st_pas), .sb_pas(sb_pas),
 .st_drv(st_drv), .sb_drv(sb_drv),    .alarm(alarm)    );

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

```
module TwoBeltAlarm(
        input k, st_pas, sb_pas,
        input st_drv, sb_drv,
        output alarm
);
        logic 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
```

2-BeltAlarm Task

```
module TwoBeltAlarm(
     input k, st_pas, sb_pas,
     input st_drv, sb_drv
     output alarm
);
     logic 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
```

```
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
```

```
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

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

logic k, st_pas, sb_pas, st_drv, sb_drv;
logic alarm;

TwoBeltAlarm dut0(
   .k(k), .st_pas(st_pas), .sb_pas(sb_pas),
   .st_drv(st_drv), .sb_drv(sb_drv),
   .alarm(alarm)
   );

task checkAlarm(
   input kV, st_pasV, sb_pasV, st_drvV, sb_drvV,
   input alarmV
   );
   #1
   k = kV; st_pas = st_pasV; sb_pas = sb_pasV;
   st_drv = st_drvV; sb_drv = sb_drvV;
   #1
   assert(alarm == alarmV) else
      $fatal (1, "bad alarm, expected:%b got:%b", alarmV, alarm);
   #1;
endtask
```

```
initial
begin
   k = 0; st_pas = 'b0; sb_pas = 'b0;
   st_drv = 'b0; sb_drv = 'h0;
   $monitor ("k:%b stPas:%b sbPas:%b stDrv:%b sbDrv:%b, a:%b",
      k, st_pas, sb_pas, st_drv, sb_drv, alarm);
   #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) )
         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

endmodule
```

```
Vivado Simulator 2020.2
Time resolution is 1 ps
run -all
i:           0 [00000]
k:0 stPas:0 sbPas:0 stDrv:0 sbDrv:0, a:0
i:           1 [00001]
k:0 stPas:0 sbPas:0 stDrv:0 sbDrv:1, a:0
i:           2 [00010]
k:0 stPas:0 sbPas:0 stDrv:1 sbDrv:0, a:0
i:           3 [00011]
k:0 stPas:0 sbPas:0 stDrv:1 sbDrv:1, a:0
i:           4 [00100]
k:0 stPas:0 sbPas:1 stDrv:0 sbDrv:0, a:0
i:           5 [00101]
```

```
k:1 stPas:1 sbPas:1 stDrv:0 sbDrv:0, a:0
i:          29 [11101]
k:1 stPas:1 sbPas:1 stDrv:0 sbDrv:1, a:0
i:          30 [11110]
k:1 stPas:1 sbPas:1 stDrv:1 sbDrv:0, a:1
i:          31 [11111]
k:1 stPas:1 sbPas:1 stDrv:1 sbDrv:1, a:0
@@@Passed
exit
```

Launch with:

```
$ xvlog -sv TwoBeltAlarm_tb.sv
            TwoBeltAlarm.sv BeltAlarm.sv
$ xelab  tb -debug typical -s tb.snap
$ xsim tb.snap --R
```

03_Code Demo

‘`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)

‘wire’ vs ‘logic’

Verilog (OLD) Rules:

- Use **reg (or logic)** for left hand side (LHS) of signals assigned inside in **always** blocks
- Use Verilog **wire** for LHS of signals assigned outside **always** blocks

Much of the Internet still uses this!

This works for E210/B441!

‘wire’ vs ‘logic’

SystemVerilog (NEW) Rules:

Just use ‘`logic`’*

* ***EXCEPT***

`logic foo = ‘h42;` **(BAD)**

`wire foo = ‘h42;` **(OK)**

`logic foo;`
assign `foo = ‘h42;` **(OK)**

‘`wire`’ vs ‘`logic`’

SystemVerilog (NEW) Rules:

Just use ‘`logic`’*

<- Also works in E210/B441

* EXCEPT

```
logic foo = ‘h42;  (BAD)

wire foo = ‘h42;   (OK)

logic foo;
assign foo = ‘h42; (OK)
```

UPDATE: ‘`wire`’ vs ‘`logic`’

SystemVerilog (NEW) Rules:

Just use ‘`logic`’*

* ***EXCEPT***

`logic foo = ‘h42;` ~~**(BAD)**~~ **(OK)**

`logic foo = a & b;` **(BAD – Initial a & b only)**

`wire foo = a & b;` **(OK)**

```
logic foo;
assign foo = a & b;
```
(OK)

Arrays in Verilog

- Bundle multiple wires together to form an array.

`type [mostSignificantIndex:leastSignificantIndex] name;`

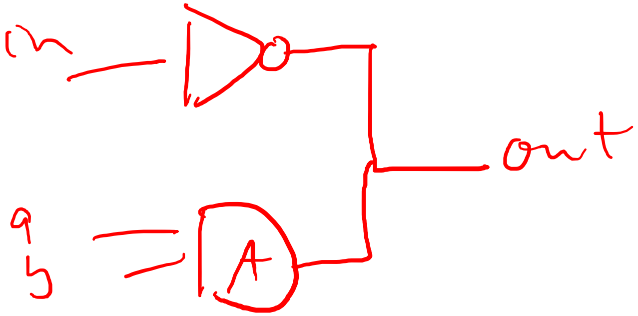
- Examples
  - `logic [15:0] x; //declare 16-bit array`
  - `x[2]   // access wire 2 within x`
  - `x[5:2] //access wires 5 through 2`
  - `x[5:2]= {1,0,y,z}; //concatenate 4 signals`

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
```

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  logic [15:0]    c     //16-bit signal
   );
   //stuff
endmodule
```

Can always add ‘`logic`’ when needed

Constants in Verilog

- A `logic` can only be a `1` or `0`
- Arrays need more bits, how to specify?

- `8’h0   = 0000 0000`   //using <u>h</u>ex notation
- `8’hff   = 1111 1111`
- `8’b1   = 0000 0001`   // using <u>b</u>inary notation
- `8’b10 = 0000 0010`
- `8’d8  = 0000 1000` //using <u>d</u>ecimal notation

Constants in Verilog

```
logic      [7:0]  aa ;
aa = {1'b0,1'b1,1'b0,1’b0,
        1'b1,1'b0,1'b0,1'b0};
aa = 8'b01001000;
aa = {8{1'b1}}; //concat
aa = 'hff; //inferred

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

`always_comb` Blocks

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

OLD Verilog

… is equivalent to …

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

New SystemVerilog

… is equivalent to …

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

New SystemVerilog Syntax

always_comb adds **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` adds **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

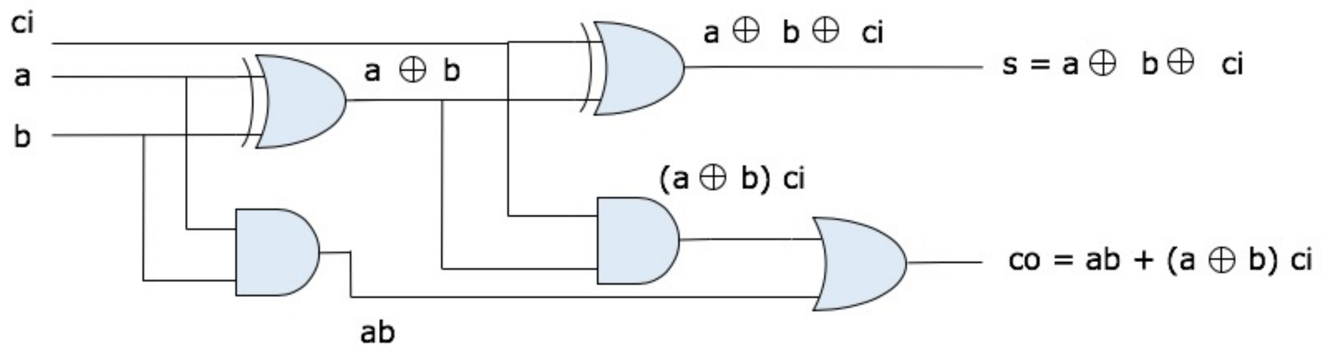
```
module decoder (
        input [1:0] sel,
        output logic [3:0] out
        );

    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`**!

Always specify defaults for **`always_comb`**!

Always specify defaults for **`always_comb`!**

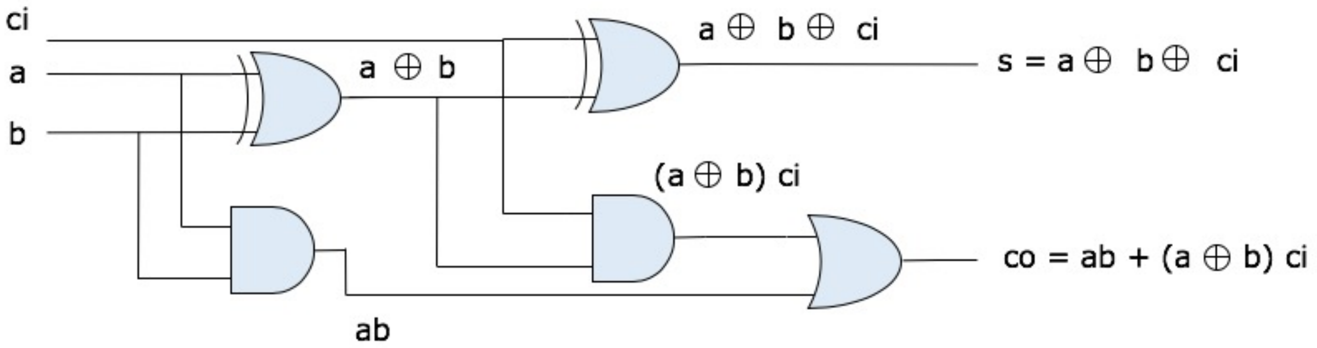
1-Bit “Full” Adder to ‘always_comb’



```
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,  cc
);
always_comb  begin
    s = 0; co = 0;  // defaults
    s = a ^ b ^ ci;
   co = (a & b) | ((a ^ b) & ci);
end

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

Half Adder

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

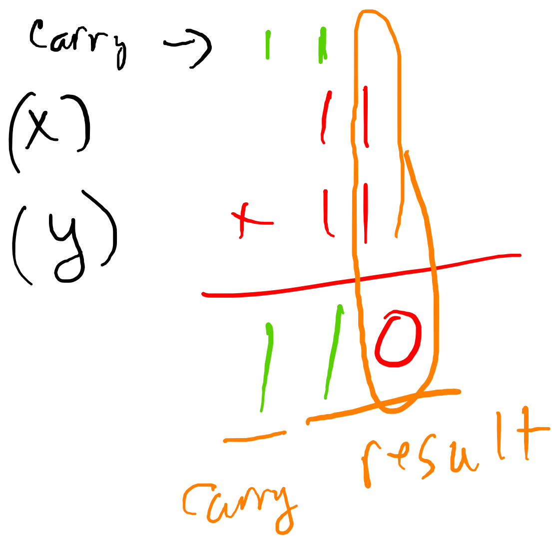
decimal

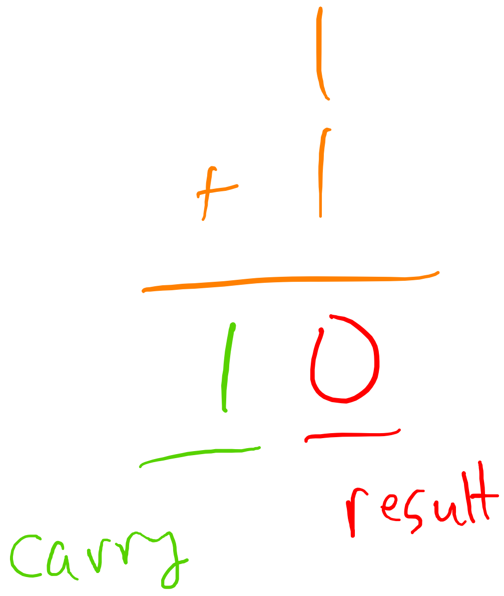
0 + 0 = 0
0 + 1 = 1
1 + 0 = 1
1 + 1 = 2

Binary Addition

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

x = 3, y = 3





Full Adder

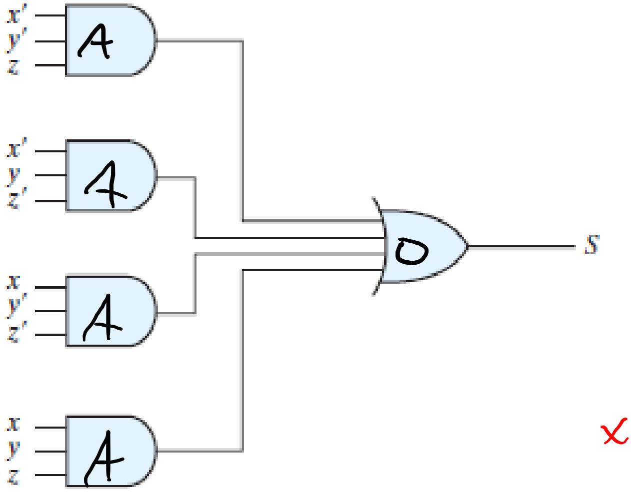
X	y	C_{in}	C_{out}	Sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

P2: use "+" in Verilog

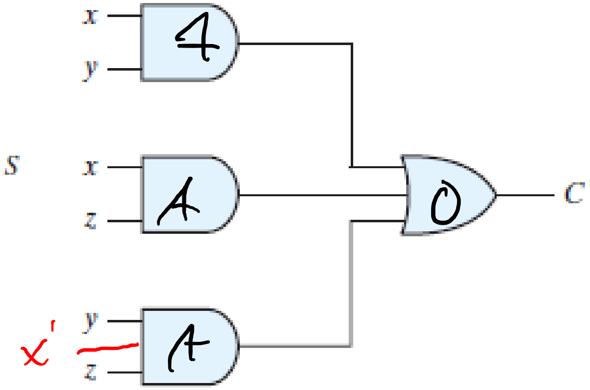
Full Adder

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

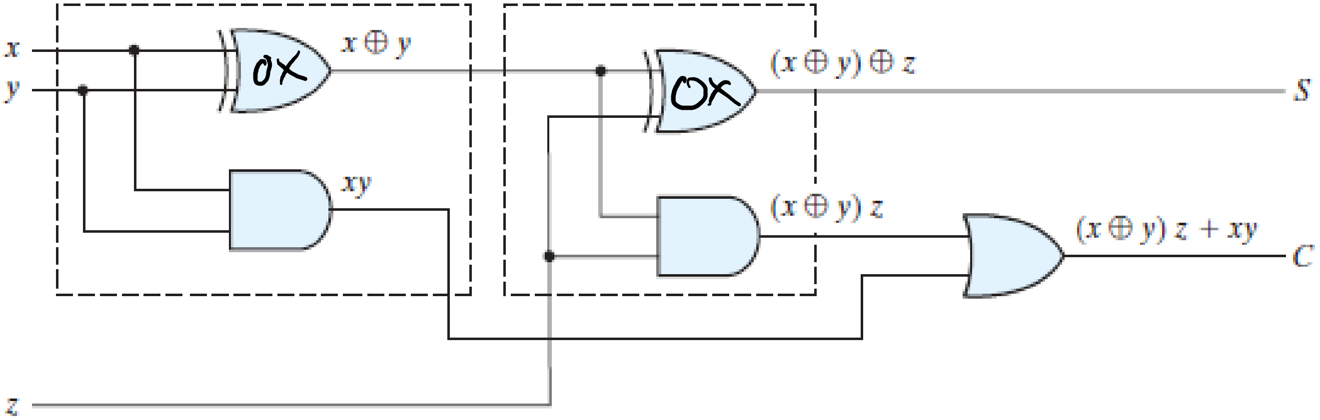
Sum



Carry

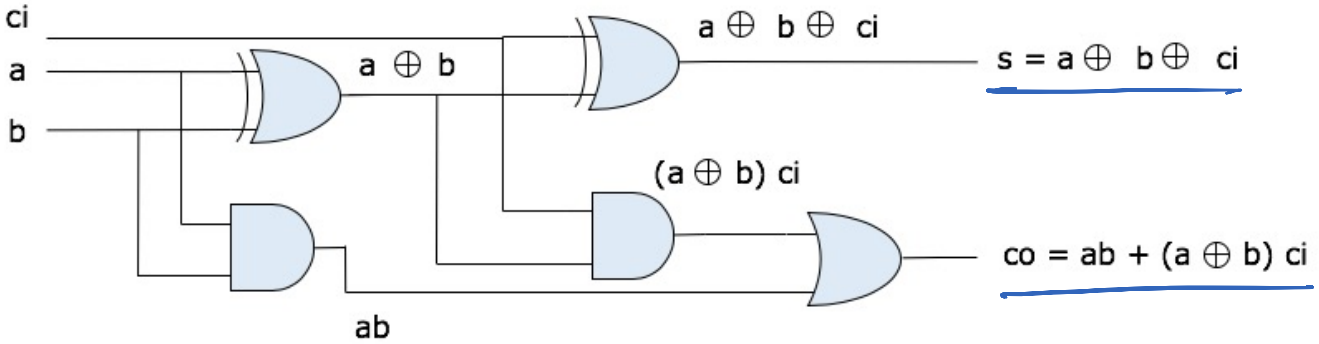


✓



logic [7:0] x = a + b;

1-Bit “Full” Adder



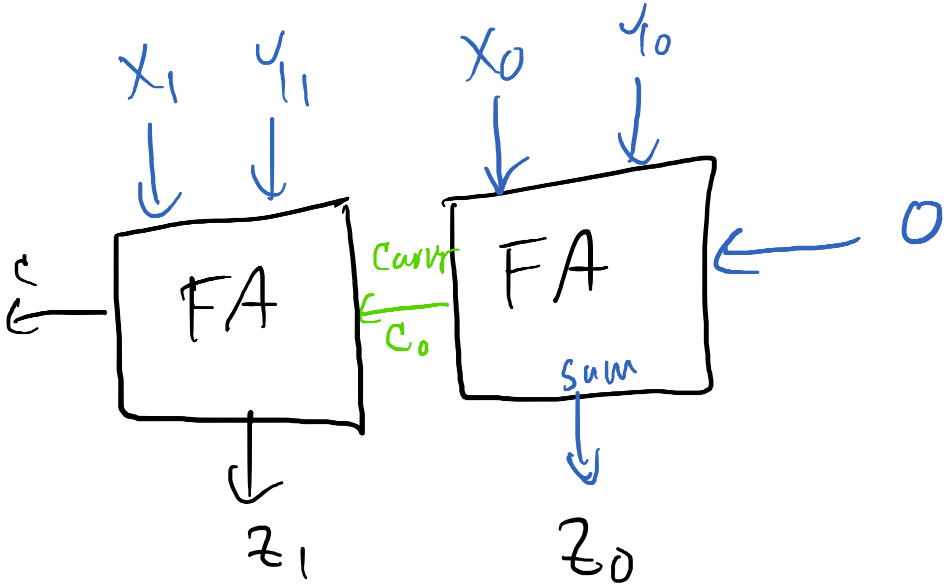
```
module FullAddr (
    input a,b,ci,
    output s, co
    );

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

endmodule
```

logic [1:0] tmp

tmp = {1'b0,a} + {1'b0,b} +
{1'b0,c};

Ripple-Carry Adder

$$\begin{array}{cccccc}
 & c_3 & c_2 & c_1 & c_0 & 0 \\
 & & x_3 & x_2 & x_1 & x_0 \\
+ & & y_3 & y_2 & y_1 & y_0 \\
\hline
 & z_4 & z_3 & z_2 & z_1 & z_0
\end{array}$$

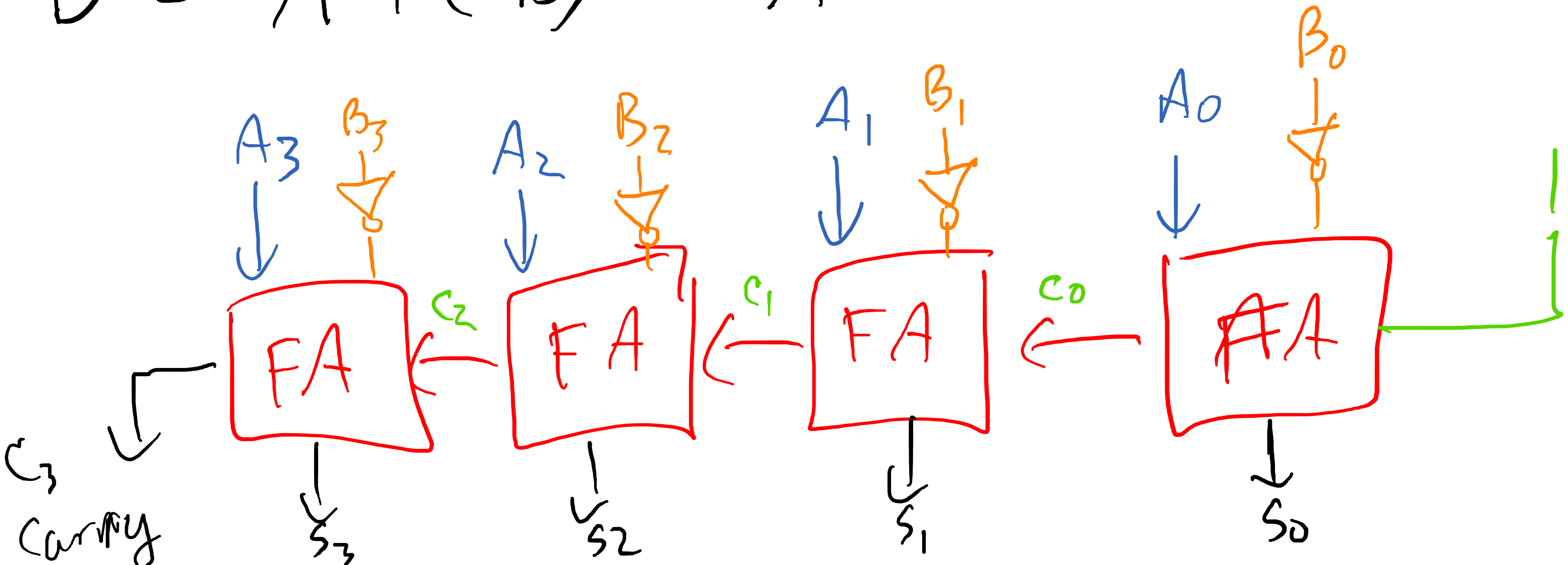


Subtraction with Adders?

- We've done `A+B`, what about `A−B`?

$$-B = \sim B + 1$$

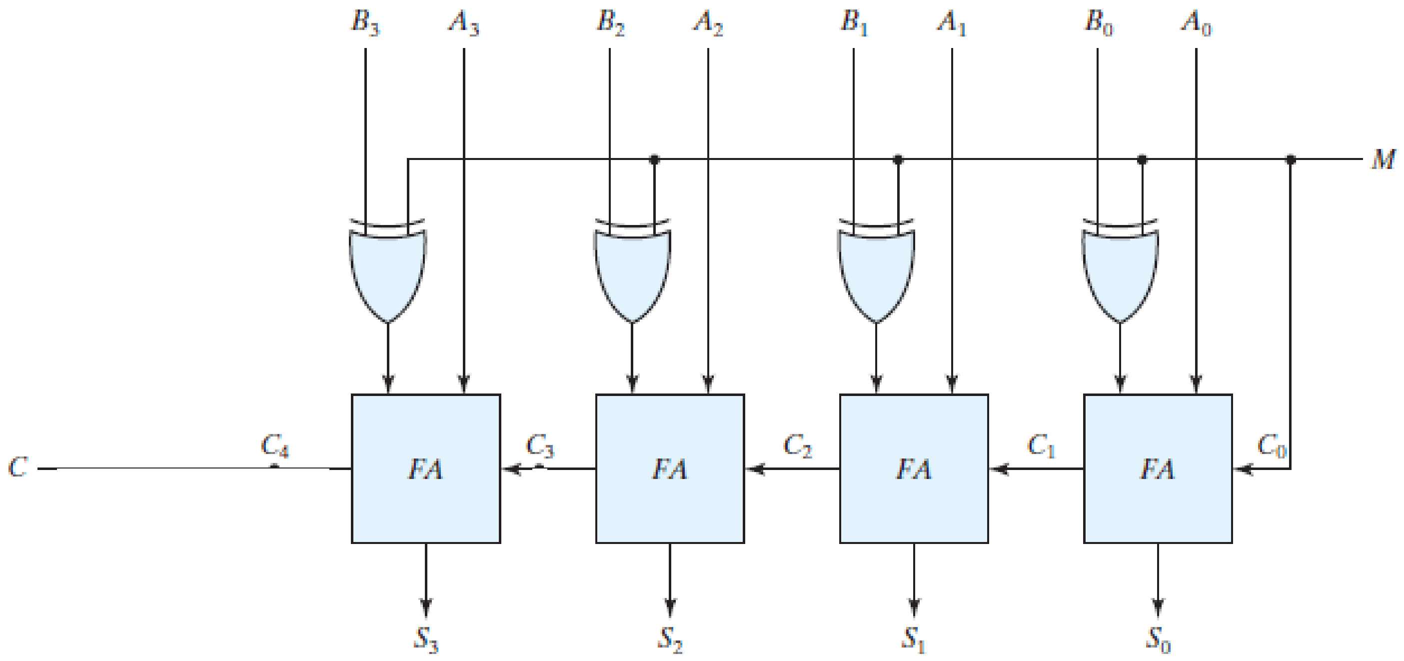
$$A - B = A + (-B) = A + \sim B + 1$$



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

B_0	M	Z
0	0	0
0	1	1
1	0	1
1	1	0



Overflow

Assume 4-bit addition

Stopped

- Unsigned

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

$$\begin{array}{r} 1\ 0\ 0\ 0\ \ \\ 1010 \\ +\ 1000 \\ \hline 1\boxed{0010} \end{array} \Rightarrow \underline{\underline{2}}$$

- Signed

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

$$\begin{array}{r} \boxed{0\ 1}\ 0\ 0\ \ \\ 0101 \\ +\ 0110 \\ \hline 0\boxed{1011} \end{array} \rightarrow 0100 + 1 = 0101 = +5$$

$\rightarrow -5$

Overflow

$$\begin{array}{r} 10 \\ +\ 8 \\ \hline 18 \end{array} \qquad \begin{array}{r} {}^{1}\ \ 1010 \\ +\ 1000 \\ \hline \underbrace{1}_{\text{carry}}\underbrace{0010}_{\text{sum}} \end{array}$$

unsigned = carry out bit

"overflow"

Signed

$$\begin{array}{r} 5 \\ +\ 6 \\ \hline 11 \end{array} \qquad \begin{array}{r} {}^{0\ 1}\ \ 0101 \\ 0110 \\ \hline \underline{0}\ \boxed{1011} \end{array} \text{ signed } = -(0100+1) = -(0101) = -5 \leftarrow \text{overflow!}$$

$5 = 0101$ $\quad 1010 \to 1011$

$6 = 0110$

carry = 0 ⇒ no overflow?

Overflow for signed numbers?

```
   -2
+ -1
-------
   -3
```

```
   -7
+  -7
-------
   -14
```

Overflow for signed numbers?

$\boxed{1\,1}\,1\,0$ carry is same ⇒ no overflow

$-2 \quad -(0010) = 1101+1 = 1110$

$+\,-1 \quad -(0001) = 1110+1 = +\,1111$

$\underline{1}0001$

$\boxed{1\,1}\,1\,0$ carry is same ⇒ no overflow

$+2 = 0010$

$+\,-1 = +\,1111$

10001

$\boxed{10}01$ → carry is different

⇒ overflow

$$\begin{array}{rlll} & -7 & -(0111) = 1000+1 = & 1001 \\ + & \underline{-7} & -(0111) = 1000+1 = & \underline{+1001} \\ & & & 10010 \end{array}$$

$\boxed{01}11$ → carry is different

→ overflow

$$\begin{array}{rll} & +7 & 0111 \\ + & \underline{+7} & \underline{+0111} \\ & & 01110 \end{array}$$

Overflow for signed numbers

$$\begin{array}{r} x\,x\,x\,x \\ +\quad y\,y\,y\,y \\ \hline z\,z\,z\,z\,z \end{array}$$

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.

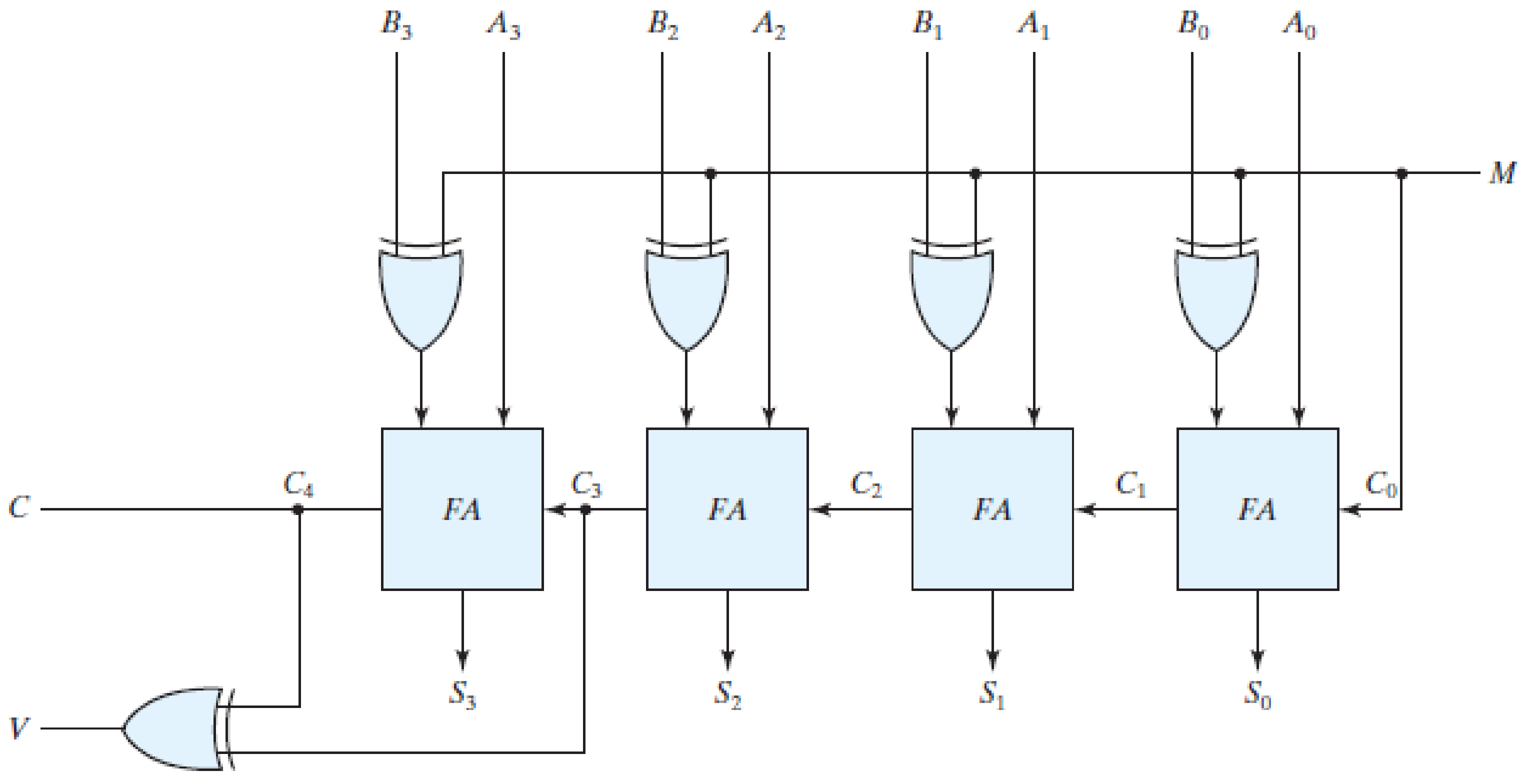
Overflow detection

- 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



P3 Tips

Next Time

- Latches / Flip-Flops

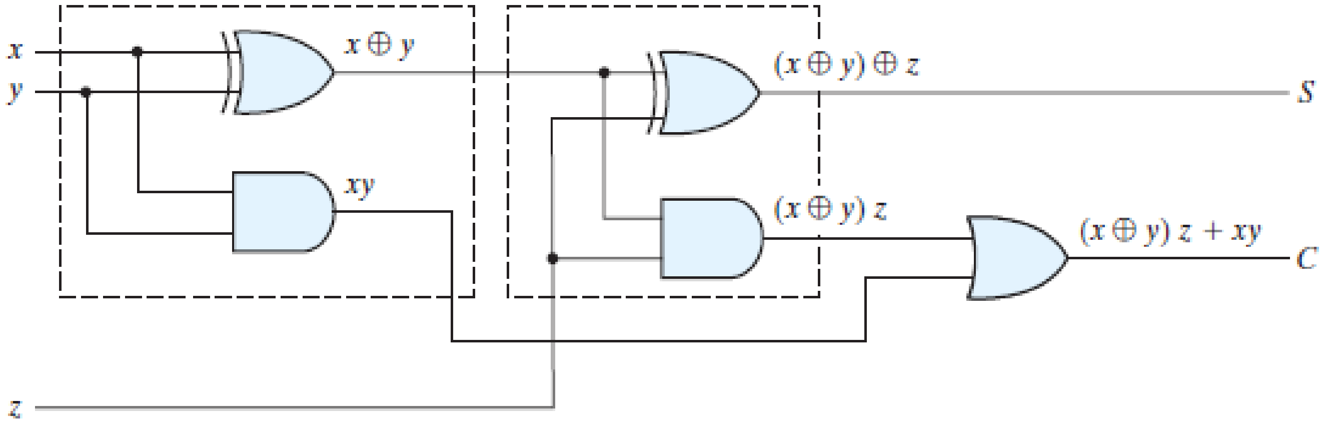
Gate Delay

- 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

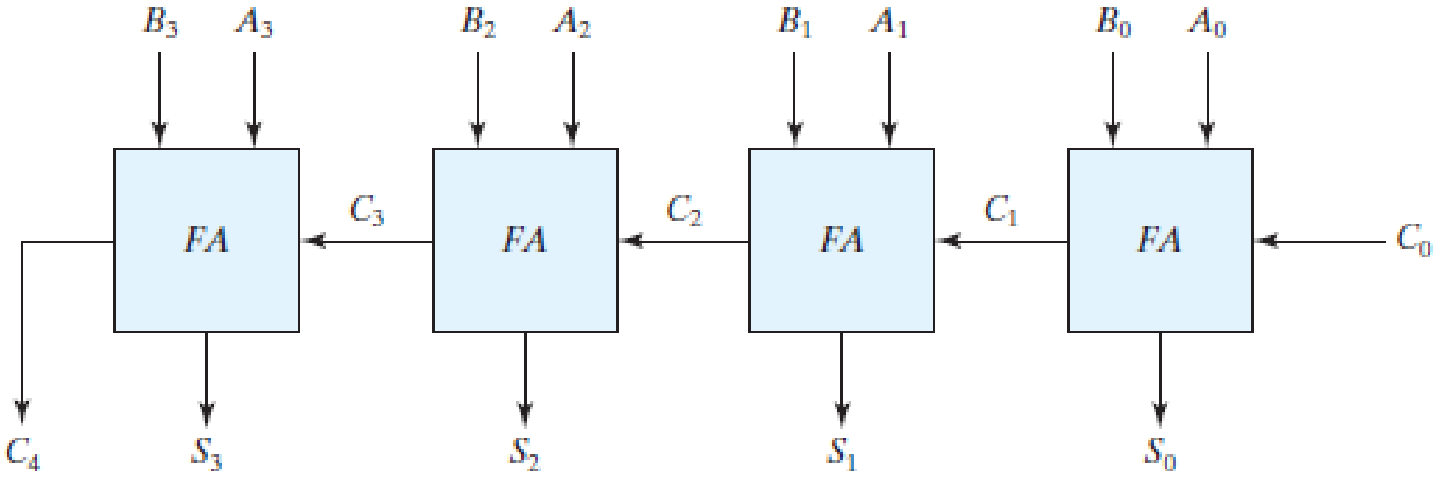
- Assume 5ps/gate



- 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