Encoders / Decoders

Andrew Lukefahr

Last Time

Review

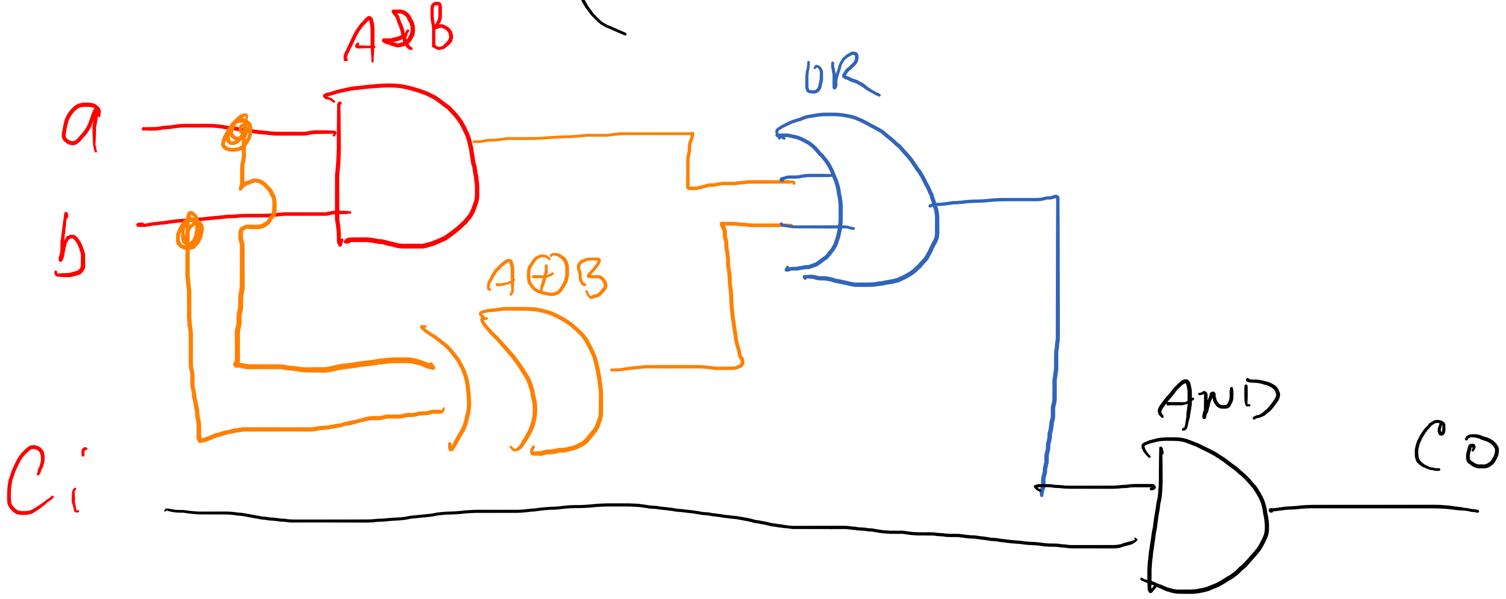
- Boolean Logic / Schematic / Truth Tables
- Verilog
  - Submodules

new

- Verilog
  - Testbenches

Review: Logic

- Draw this circuit: **co = a & b | (a ^b) & ci;**



Review: Submodules

- `modules` are basic building block in Verilog
- Group modules together to form more complex structure

Review: Submodules

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

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

endmodule
```

```
module TwoBitAddr(
   input a0, a1, b0, b1, ci,
   output s0, s1, co
   );

   //code me!
   wire R; //caRry
   FullAddr ad0 ( .a(a0), .b(b0), .ci(ci),
                  .s(s0), .co(r));
   FullAddr ad1 ( .a(a1), .b(b1), .ci(r),
                  .s(s1), .co(co));
endmodule
```

$$\begin{array}{rrr} & \overset{r}{a_1} & a_0 \\ + & b_1 & b_0 \\ + & & C_{in} \\ \hline C_{out} & S_1 & S_0 \end{array} \qquad \begin{array}{rr} & \overset{1}{1} \\ + & 1 \\ + & 1 \\ \hline & 0 \end{array}$$

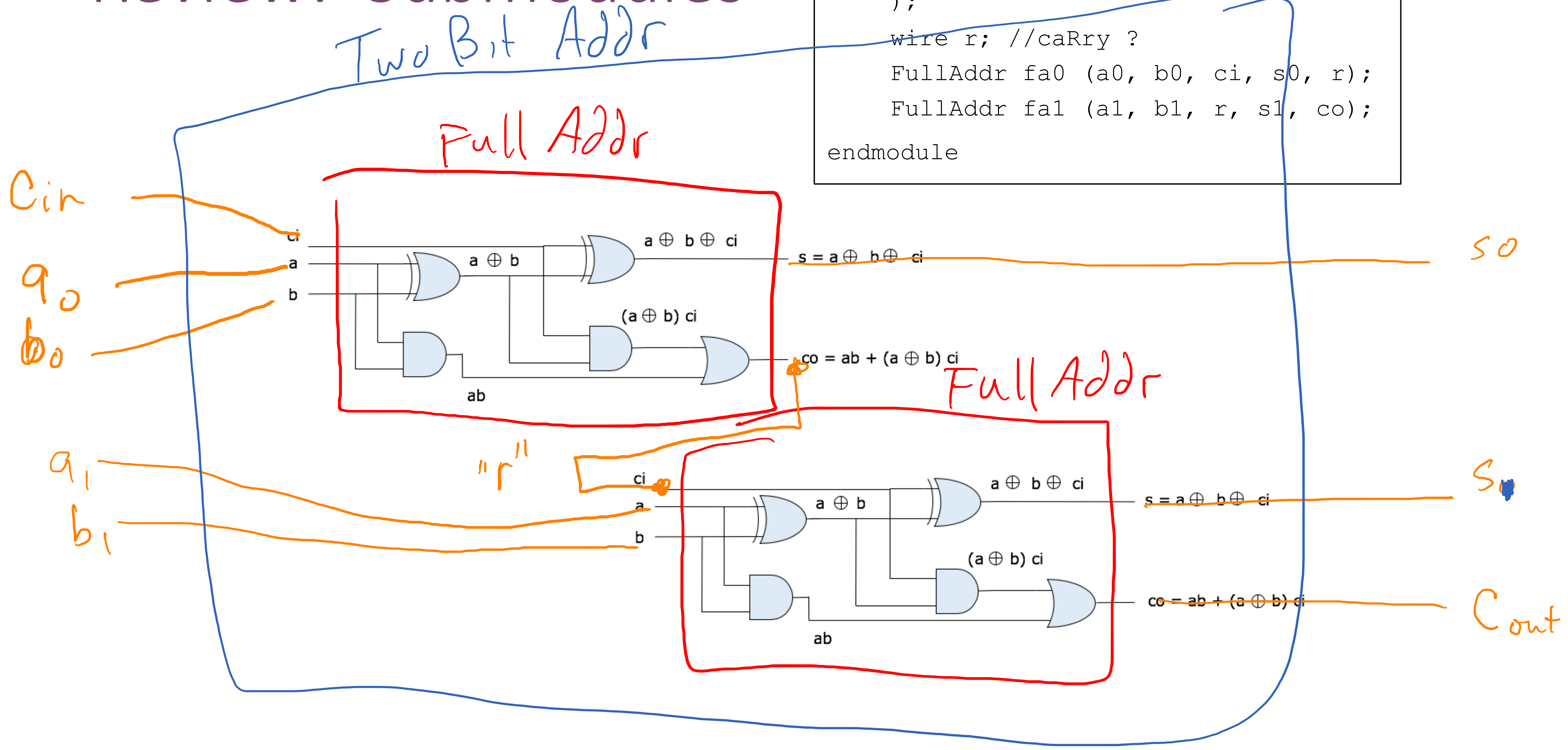
Review: Submodules

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

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

endmodule
```

```
module TwoBitAddr(
   input a0, a1, b0, b1, ci,
   output s0, s1, co
   );

   wire r; //caRry ?

   FullAddr fa0 (.a(a0), .b(b0), .ci(ci), .s(s0), .co(r)  );
   FullAddr fa1 (.a(a1), .b(b1), .ci(r) , .s(s1), .co(co) );

endmodule
```

Review: Submodules

```
module TwoBitAddr(
     input a0, a1, b0, b1, ci,
     output s0, s1, co
     );
     wire r; //caRry ?
     FullAddr fa0 (a0, b0, ci, s0, r);
     FullAddr fa1 (a1, b1, r, s1, co);
endmodule
```



Verilog Testbenches

- Synthesize-able vs. Simulate-able

- Synthesis: Real circuit on real hardware
  - Only “synthesizable” Verilog allowed

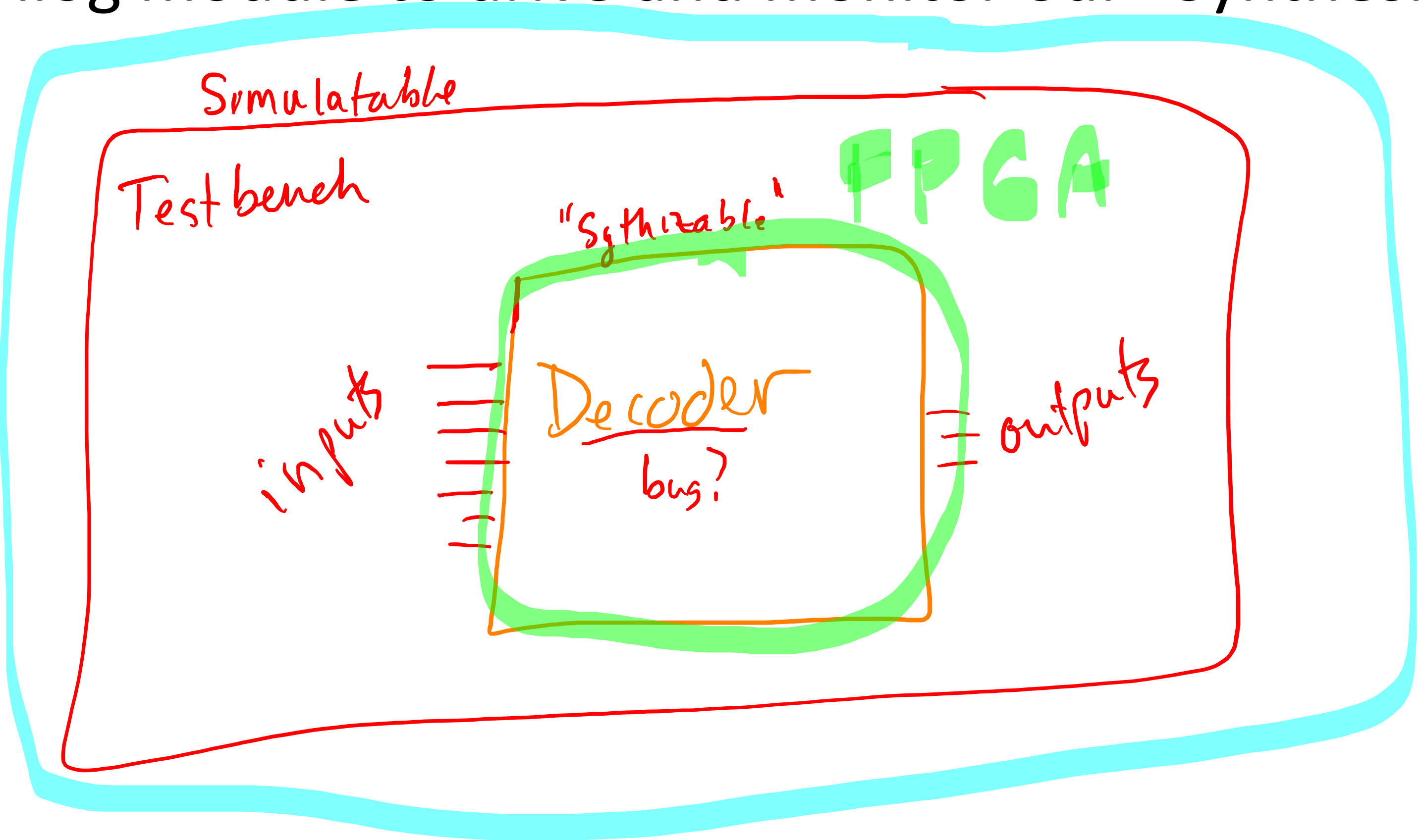
- Simulation: Test our design with software simulations
  - ”Non-synthesizable” Verilog allowed (System Verilog)
  - Often simulation-only commands start with a dollar sign (`$`)

TestBenches

- Another Verilog module to drive and monitor our “Synthesizable” module

TestBenches

- Another Verilog module to drive and monitor our “Synthesizable” module



“initial” statement

C

for(int i = 0; i<10, ++i){

}

- Simulation only!
- Starts at simulation time 0, executes exactly once, and then does nothing.
- Group multiple statements with `begin` and `end`.
  - `begin/end` are the ‘`{`’ and ’`}`’ of Verilog.

```
initial
begin
    a = 1;
    b = 0;
end
```

`timescale 1ns/1ps

Delayed execution

#10.0005 ← appear the same

#10.0001 ←

- If a delay `#<delay>` is seen before a statement, the statement is executed `<delay>` time units after the previous statement.

10 ns

```
initial
begin
    #10 a = 1; // executes at 10 time units
    #25 b = 0;// executes at 35 time units
end
```

- We can use this to test different inputs over time on our circuits

$monitor

- `$monitor` prints a new line every time it's output changes
- C printf-like format

-
  ```
  $monitor($time,
      “K= %b, P= %b, S= %b, A= %b\n",
       K,P,S,A);
  ```

%b = binary

%h = hexa

%d = decimal

Example Output:

```
0  K= 0, P= 0, S= 0, A= 0
5  K= 1, P= 0, S= 0, A= 0
10 K= 1, P= 1, S= 0, A= 1
```

Testing a Full Adder

```
`timescale 1ns / 1ps

/// initialize FullAddr

initial
begin

     //$monitor optional

     #1 //wait 1ns
     a = 1; b = 0; ci = 0;
     #0.001 // 1ps
     assert( s == 1) else $fatal(1, ”s”);
     assert( co == 0) else $fatal(1, “co”);

     #1 //wait 1ns
     a = 1; b = 1; ci = 0;
     #0.001 // 1ps
     assert( s == 0) else $fatal(1, ”s”);
     assert( co == 1) else $fatal(1, “co”);

     $finish;
end
```

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

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

endmodule
```

return

print

$$\begin{array}{r} 1 \\ +\ 0 \\ +\ 0 \\ \hline 0\ 1 \end{array}$$

$$\begin{array}{r} 1 \\ +\ 1 \\ +\ 0 \\ \hline 1\ 0 \end{array}$$

Testing a Full Adder

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

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

endmodule
```

```
`timescale 1ns / 1ps

/// initialize FullAddr

initial
begin

     //$monitor optional

     #1 //wait 1ns
     a = 1; b = 0; ci = 0;
     #0.001 // 1ps
     assert( s == 1) else $fatal(1, ”s”);
     assert( co == 0) else $fatal(1, “co”);

     #1 //wait 1ns
     a = 1; b = 1; ci = 0;
     #0.001 // 1ps
     assert( s == 0) else $fatal(1, ”s”);
     assert( co == 1) else $fatal(1, “co”);

     $finish;
end
```

Tasks in Verilog

- A task in a Verilog simulation behaves similarly to a C function call.

```
task taskName;
    input localVariable1;
    input localVariable2;

    #1 //1 ns delay
    globalVariable1 = localVariable1;
    #1 // 1ns delay
    assert( globalVariable2 == localVariable2)
        else $fatal(1, “failed!”);

endtask
```

inputs

what to do

what to check

Tasks in Testing

```
module FullAddr (
     input a,b,ci,
     output s, co
     );
   s = a ^ b ^ ci;
   co = a & b | ( a ^ b) & ci;
endmodule
```

```
`timescale 1ns / 1ps

// declare a,b,ci, s, & co

FullAddr fa0 (.a(a), .b(b), .ci(ci), .s(s), .co(co));

task TestOne; //set module signals to T(est) values
     input aT, bT, ciT, sT, coT;

     #1
     a = aT; b= bT; ci = ciT;
     #1
     assert( s  == sT ) else $fatal(1, "s  failed");
     assert( co == coT) else $fatal(1, "co failed");
endtask

initial
begin
     TestOne(.aT(1), .bT(0), .ciT(1), .sT(1), .coT(0));
     TestOne(.aT(1), .bT(1), .ciT(0), .sT(0), .coT(1));
     $finish;
end
```

$$\begin{array}{r} 1 \\ +\ 0 \\ +\ 1 \\ \hline 10 \end{array}$$

a=1 b=0 cin=1 s=1 co=0 ✓

a=1 b=1 cin=0 s=0 co=1

Tasks in Testing

```
module TwoBitAddr(
    input a0, a1, b0, b1, ci,
    output s0, s1, co
    );
    wire r;
    FullAddr fa0 (a0,b0,ci,s0,r);
    FullAddr fa1 (a1,b1,r,s1,co);
endmodule
```

```
/// module definition
// declare a0,a1,b0,b1,ci, s0,s1,& co

TwoBitAdder tba0 (a0,a1,b0,b1,ci,s0,s1,co);

task TestTwo;
    input a1T, a0T, b1T, b0T, cinT, CoutT, s1T, s0T;
    #1
    a1 = a1T; a0 = a0T; b1 = b1T; b0 = b0T, cin = cinT;
    #1
    assert( co == coT) else $fatal(1, "cout failed\n");
    assert((s0 == s0T) && (s1 = s1T)) else $fatal(1, "sum failed\n");
endtask

                a1 a0 b1 b0 Ci  Cout s1 s0
initial begin   ↓  ↓  ↓  ↓  ↓   ↓    ↓  ↓
    TestTwo( 0,0,0,0,0, 0,0,0);  // a=00 + b=00 + ci=0 => s=00 co=0
    TestTwo( 0,0,0,0,1, 0,0,1);  // a=00 + b=00 + ci=1 => s=01 co=0
    //more tests + $finish
end
```

Tasks in Testing

```
module TwoBitAddr(
    input a0, a1, b0, b1, ci,
    output s0, s1, co
    );
    wire r;
    FullAddr fa0 (a0,b0,ci,s0,r);
    FullAddr fa1 (a1,b1,r,s1,co);
endmodule
```

```
/// module definition
// declare a0,a1,b0,b1,ci, s0,s1,& co

TwoBitAdder tba0 (a0,a1,b0,b1,ci,s0,s1,co);

task TestTwo;
    input a1T, a0T, b1T, b0T, ciT;
    input coT, s0T, s1T;

    #1
    a0 = a0T; a1 = a1T; b0 = b0T; b1=b1T, ci = ciT;
    #1
    assert( (s0  == s0T) && (s1 == s1T)) else $fatal(1, “s  failed”);
    assert( co == coT) else $fatal(1, “co failed”);
endtask

initial begin
    TestTwo( 0,0,0,0,0, 0,0,0);  // 00 + 00 + 0 = 000
    TestTwo( 0,0,0,0,1, 0,0,1);  // 00 + 00 + 1 = 001
    //more tests + $finish
end
```

`Tasks` in Testing

- `tasks` are very useful for quickly testing Verilog code

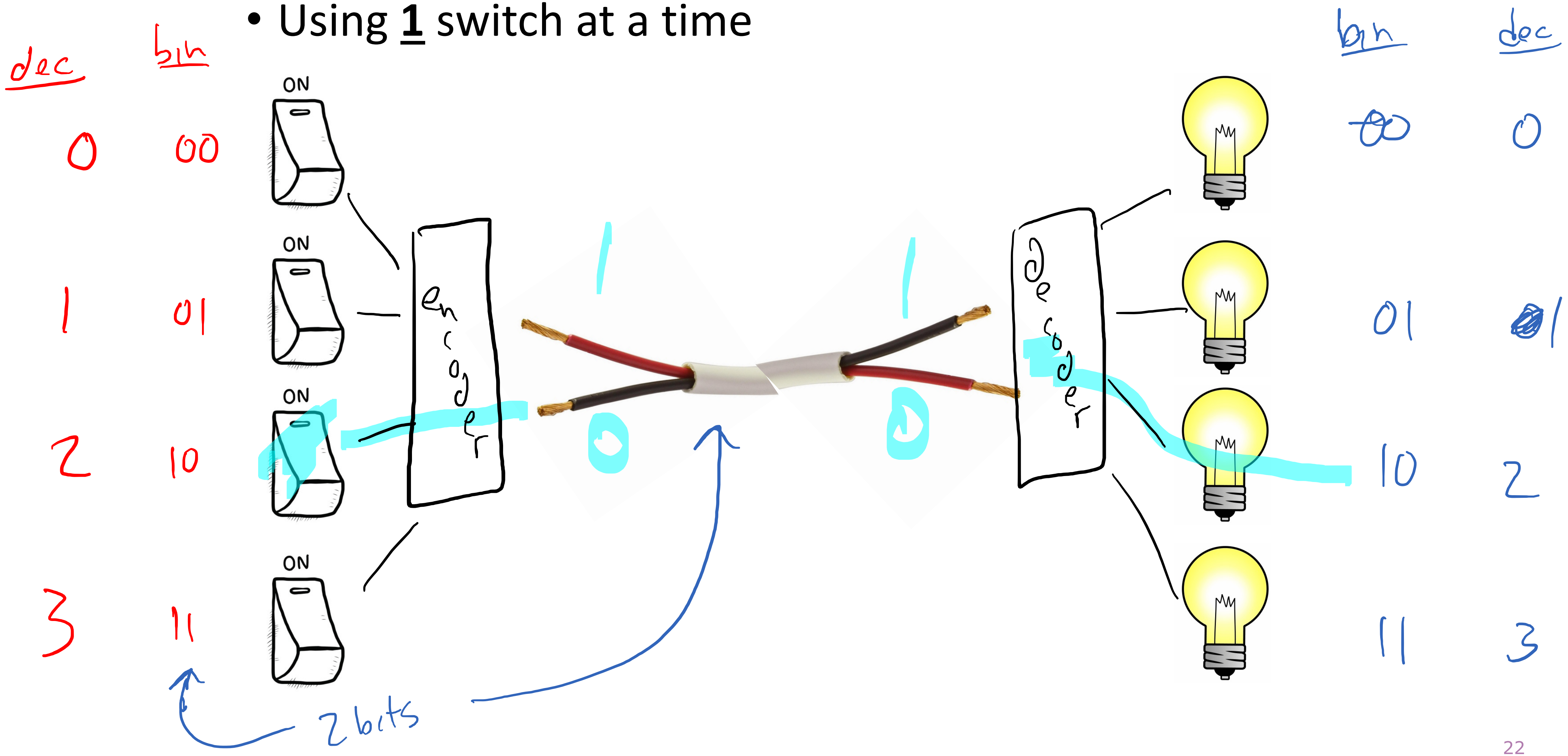
- Call a `task` to quickly change + check things
- A `task` can call another `task`

- There is a `function` in Verilog.
- We don’t use it.

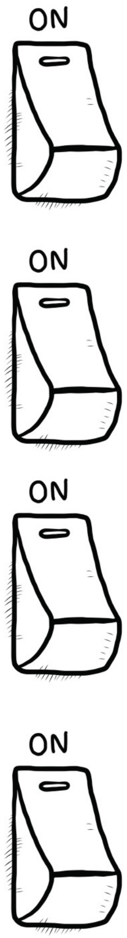
Topic Change: Encoders / Decoders

Encoding/Decoding

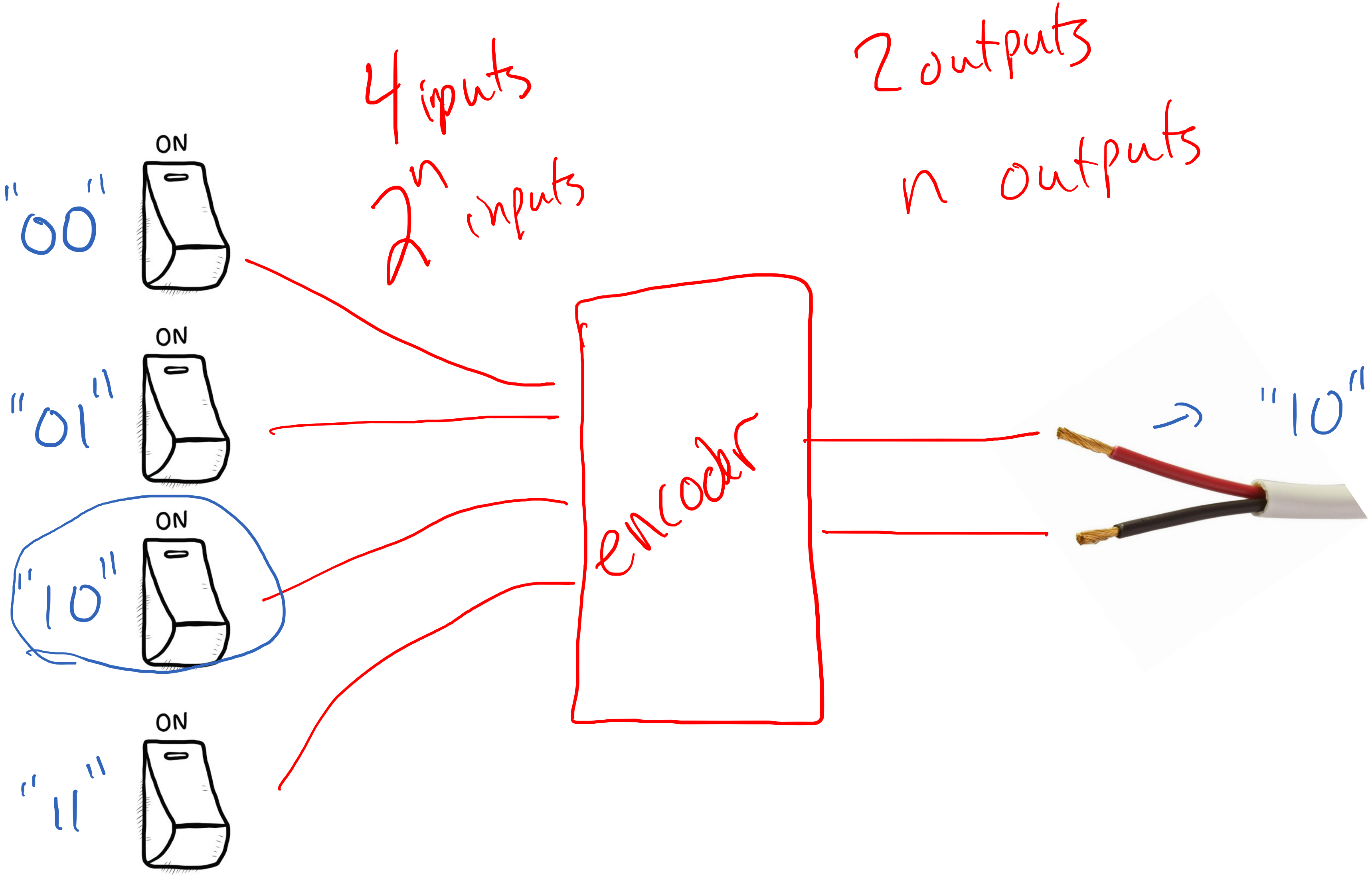
- Using **1** switch at a time



Encoding



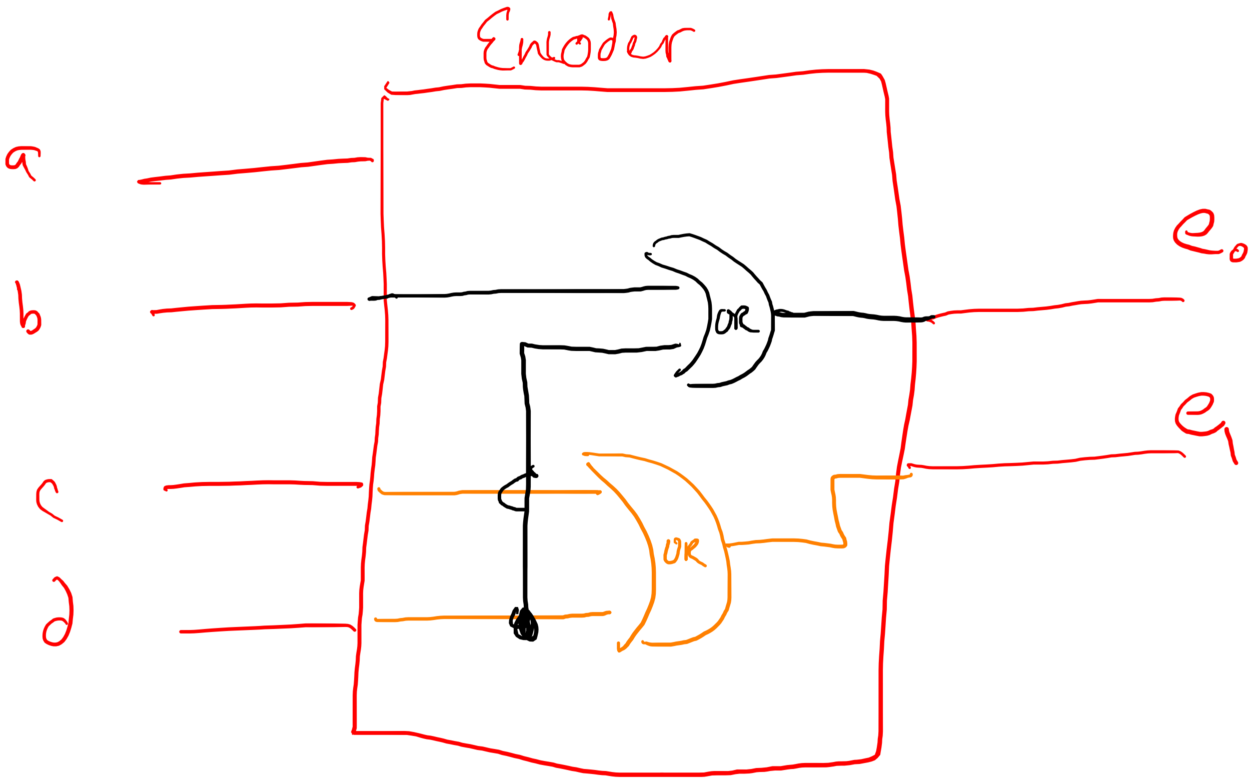
Encoding


4 inputs
2^n inputs
2 outputs
n outputs
"00"
ON
"01"
ON
"10"
ON
"11"
ON
encoder
→ "10"

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



$e_1 = c \mid d;$

$e_0 = b \mid d;$

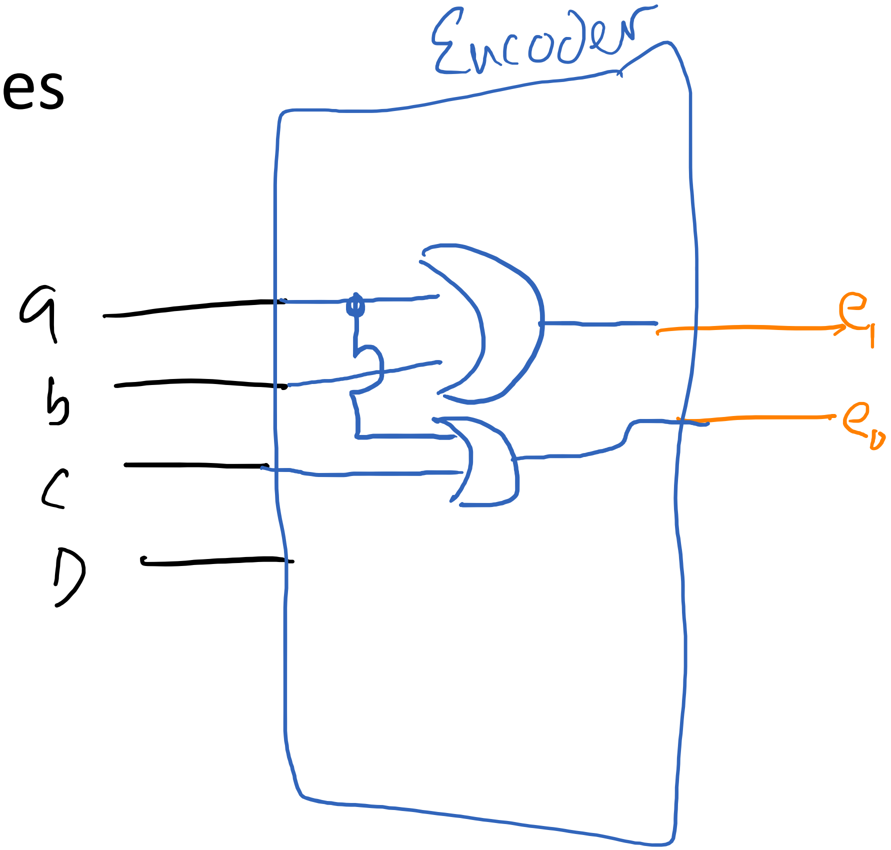
Encoders

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

inputs				outputs	
a	b	c	d	e_1	e_0
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

$$e_1 = a \mid b;$$

$$e_0 = a \mid c;$$



Encoder in Verilog

e1 = c | d;
e0 = b | d;

```
module encoder (
    input a,b,c,d,
    output e1, e0
    );

    assign e1 = c | d;
    assign e0 = b | d;

endmodule
```

Encoder in Verilog

```
module encoder (
    input a,b,c,d,
    output e1, e0
    );

    assign e0 = c | d;
    assign e1 = b | d;

endmodule
```

Encoder w/Valid

- How would you add a `valid` output signal?

```
module encoder_valid (
     input a,b,c,d,
     output e1, e0,
     output valid
     );
      //code me!
```

encoder enc0 (.a(a), .b(b), .c(c), .d(d), .e1(e1), .e0(e0));

assign valid = a | b | c | d;

```
endmodule
```

a	b	c	d	v
1	0	0	0	1
0	1	0	0	1
0	0	1	0	1
0	0	0	1	1
0	0	0	0	0

```
module encoder (
     input a,b,c,d,
     output e1, e0
     );

                  c|d
     assign e0 = ~~a | c;~~
     assign e1 = ~~b | a;~~
                  b|d;

endmodule
```

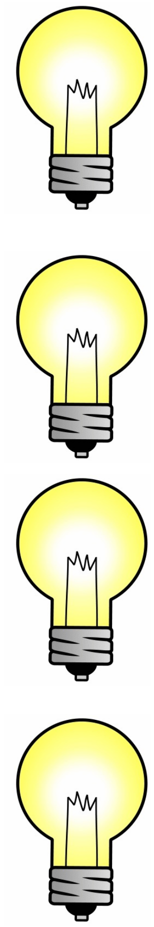
Encoder w/Valid

- How would you add a `valid` output signal?

```
module encoder (
    input a,b,c,d,
    output e1, e0
    );

    assign e0 = a | c;
    assign e1 = b | a;

endmodule
```

```
module encoder_valid (
    input a,b,c,d,
    output e1, e0,
    output valid
    );

    encoder e0 (.a(a),.b(b),.c(c), .d(d), .e1(e1), .e0(e0) );
    assign valid = a | b | c | d; // e0|e1|d

endmodule
```

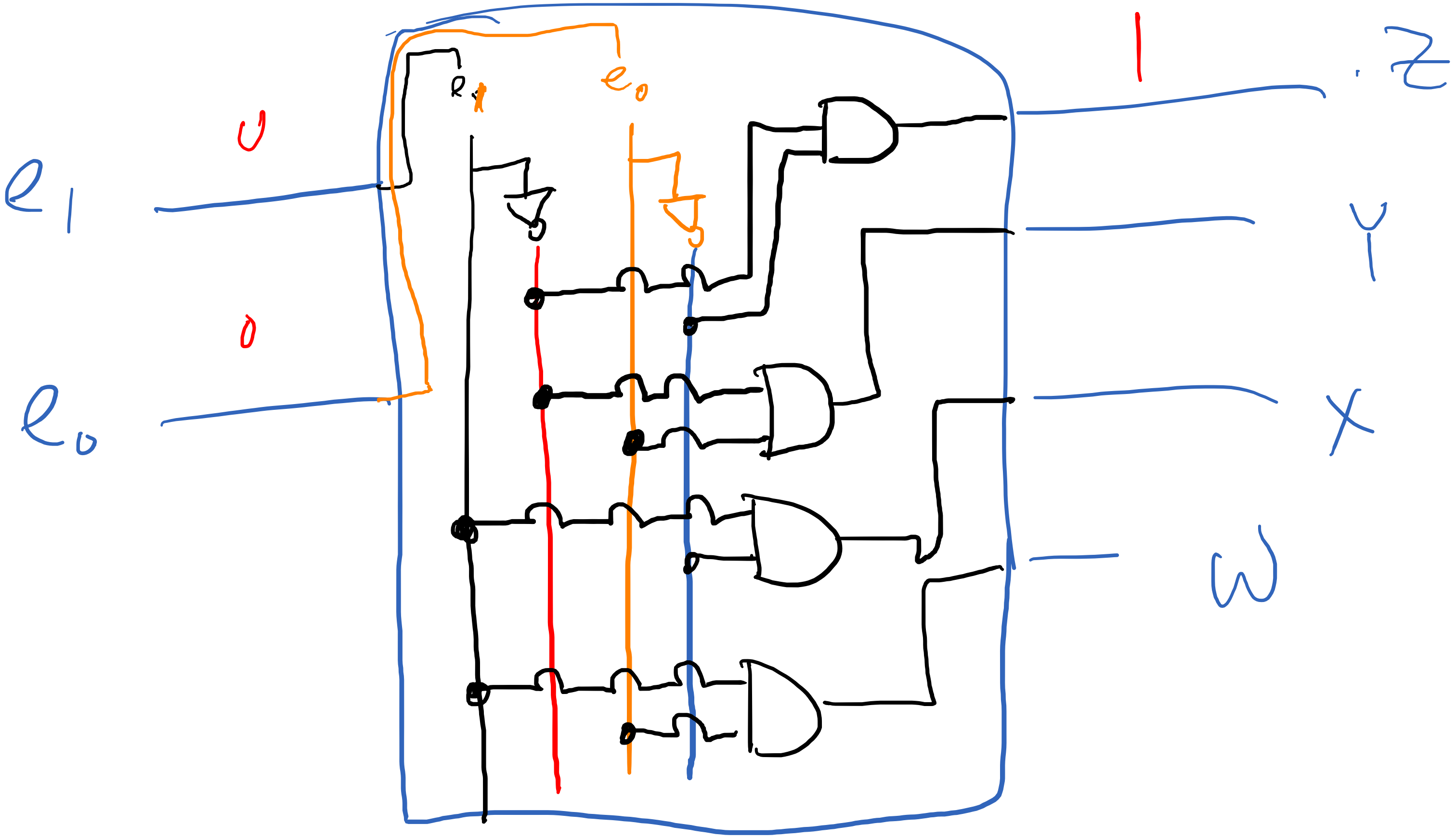
Decoding



Decoders

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

e_1	e_0	W	X	Y	Z
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0



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

$= (\sim(e_1 | e_0))$

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

$X = e_1 \ \& \sim e_0$

$W = e_1 \ \& \ e_0$

Decoders

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

Decoders

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

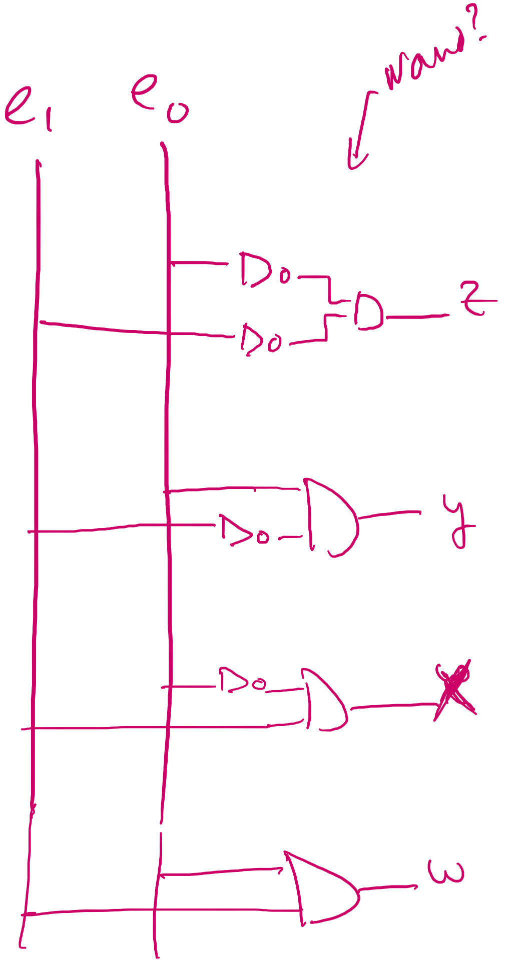
e_1	e_0	w	x	y	z
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

$z = \bar{e}_1 \bar{e}_0$

$y = \bar{e}_1 e_0$

$x = e_1 \bar{e}_0$

$w = e_1 e_0$



Decoder Verilog

```
module decoder (
    input e1, e0,
    output w,x,y,z
    );

    assign w = ~e1 & ~e0;
    assign x = ~e1 &  e0;
    assign y =  e1 & ~e0;
    assign z =  e1 &  e0;

endmodule
```

Encoders / Decoders

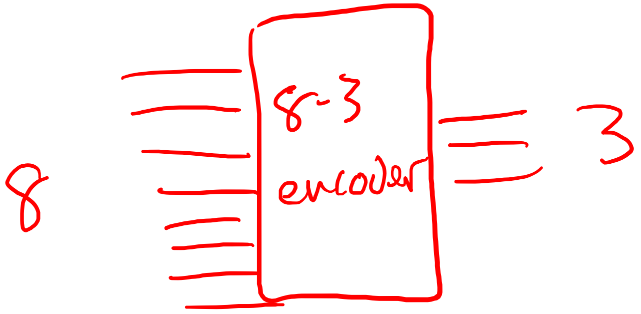
- Encoder:
  - Takes *2^n* single-bit signals, emits an *n-bit* encoded signal

- Decoder:
  - Takes an *n-bit* encoded signal, emits *2^n* single-bit signals

Bigger Encoders

- What would a 8-to-3 *encoder* look like?
- Write the Truth Table & Boolean Equation

8-3 Encoder



hierarchially

d_0	d_1	d_2	d_3	d_4	d_5	d_6	d_7	e_2	e_1	e_0
							1	0	0	0
						1		0	0	1
					1			0	1	0
				1				0	1	1
			1					1	0	0
		1						1	0	1
	1							1	1	0
1								1	1	1

$e_2 = d_0 | d_1 | d_2 | d_3 ;$

$e_1 = d_0 | d_1 | d_4 | d_5 ;$

$e_0 = d_0 | d_2 | d_4 | d_6$

Next Time

- Multiplexer
- Demultiplexer
- Adders
- ALUs