Lecture #4

* So, today, we discuss the prop. delay of combinational circuits.

- When the input to a gate changes, and causes the output to change, the change cannot occur instantly. (There is a time constant associated with the $(R_{eq} \cdot C_{eq})$ seen at that node.)
- Our first-order model:

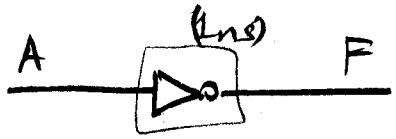
associates a delay with each gate. (This is, of course, not entirely realistic: delays are associated w/ wires, too) - but simple model for now

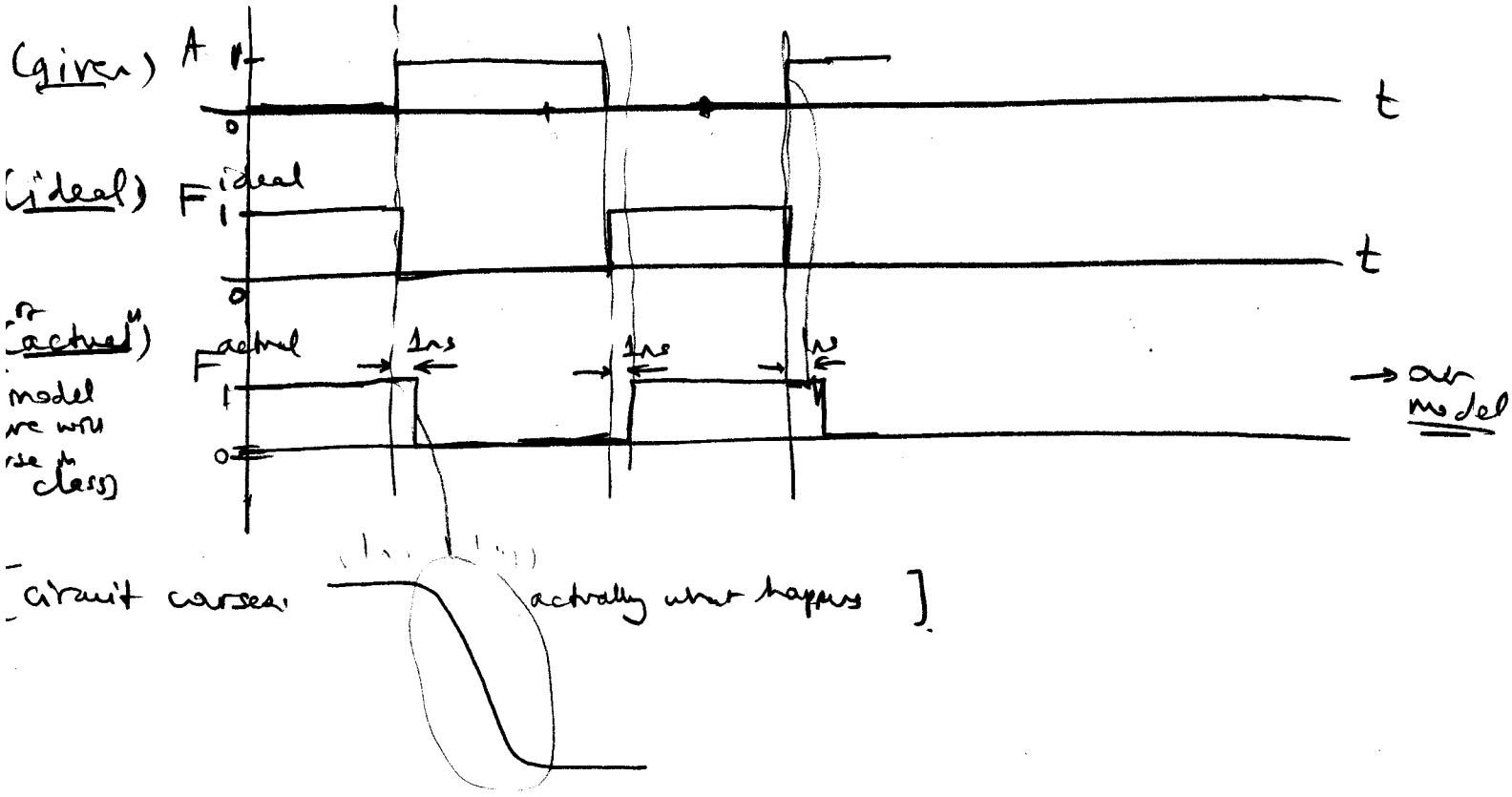
Lecture #3

outline: 1) Delay of comb. circuits
2) levels for " "

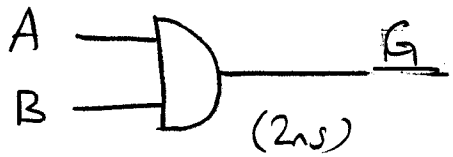
Timing diagram:

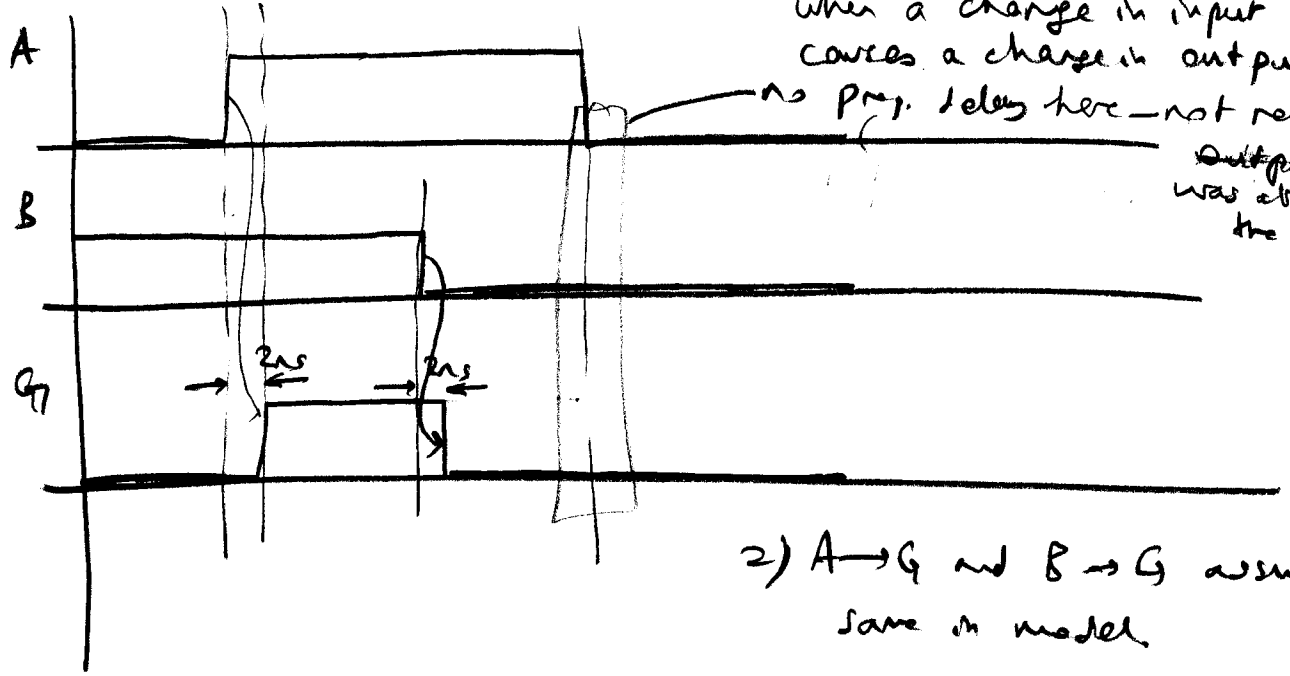
Ex1





Ex2



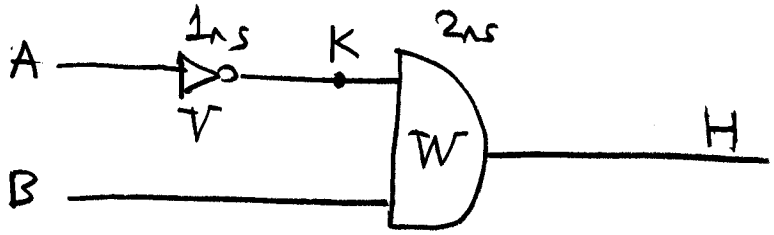


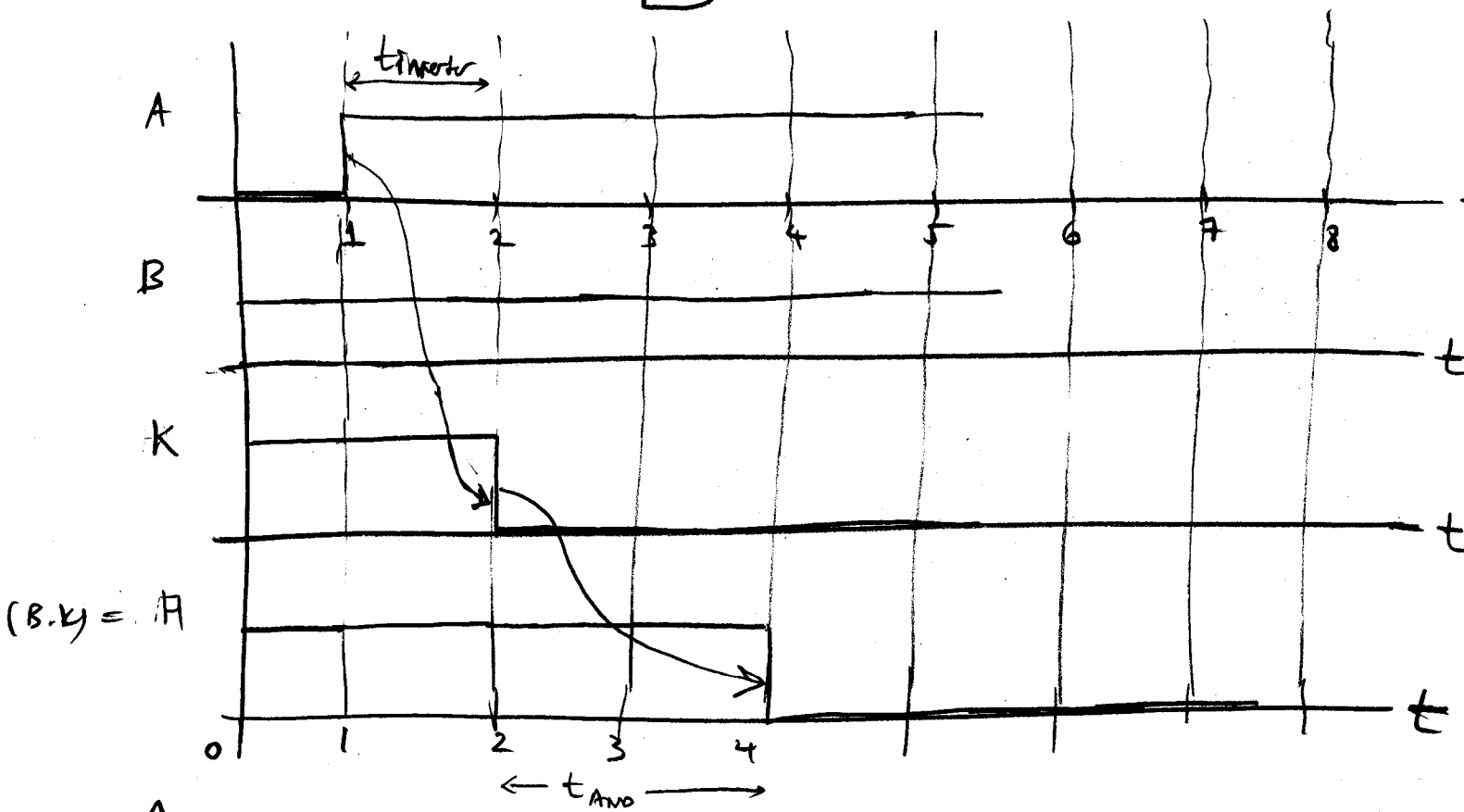
Note that:

1) Prop. delay is relevant only when a change in input causes a change in output.
→ no prop. delay here — not relevant
Output is was already the same

2) A → G and B → G assumed same in model

Ex3





- Assume that at time $t=0$, the circuit is in steady-state. ($A=0$, $B=1$, and H has already settled to $H=1$). and $K=1$.

[* The above are timing diagrams under our model.]

- You see that it took 3ns after $A: 0 \rightarrow 1$ that $H: 1 \rightarrow 0$.

- Similarly, it is easy to see that: starting w/ same initial conditions, $B: 1 \rightarrow 0$ to $H: 1 \rightarrow 0$ would take 2ns

- The maximum propagation delay of a combinational circuit is very important because it tells you the worst-case delay of that combinational circuit.
- Above, the max prop. delay is 3 ns.
- The path along which the max prop. delay occurs is called the critical path.

Above: critical path: A → K → H

or simply: A, K, H.

or: A — (V) 1ns — (W) 2ns — H

gates

- An input transition that causes the max. prop. delay is called a critical transition.

Above: A critical transition is:

Steady state	transition
in: A = 0	A = 1
in: B = 1	
out: H = 1	H = 0

Another one:

A = 1	A = 0
B = 1	
H = 0	H = 1

2 critical transitions.

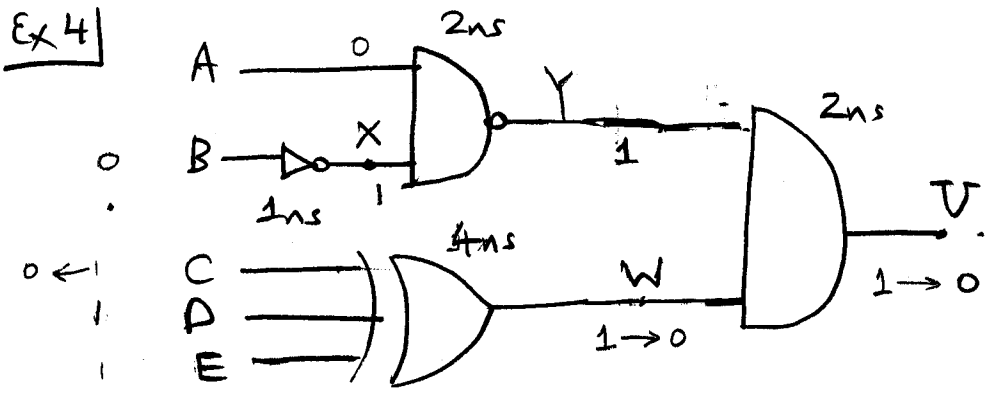
- The minimum delay of the circuit is 2ns above.

B → (W) → H.

- The number of levels of this circuit is 2.

(||| maximum # of gates that an input signal has to reach the output)

Ex 4



- fan-in of a gate: # inputs into the gate
- max fan-in in this circuit = 3.
- fan-out of a gate is the number of gates it drives.
- max fanout in this circuit = 1.

- # of Levels = 3.
- max prop. delay = max {5ns, 6ns} = 6ns
- critical path: C, D, E — XOR — W — AND — U.

(Note that critical path/max delay may be diff. from # when the # of Levels is max)

- minimum delay: min {4ns, 6ns} = 4ns

Minimum delay path: A — NAND — Y — AND — U

- critical transition(s): [stay in old state w/ (U=1)].

Ex

steady state	new state
A=0, B=0	
C=1, D=1	C=0
E=1, U=1	U=0

Readings 2.10, Appendix A.1–A.10
6.6.1–6.6.2, 6.6.5.

[redacted]:

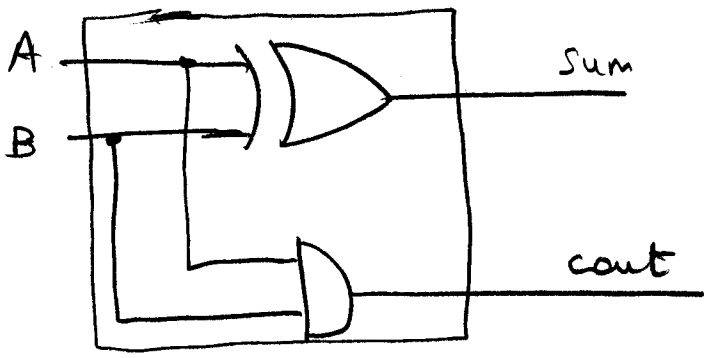
~~Outline~~

- ~~Introduction to Verilog (for comb. logic)~~
- ~~Timing of comb. circuits~~

Verilog: a concurrent programming (hardware description) language to simulate & verify digital circuits. Further with synthesis tools (such as Xilinx tools, & Synopsis), a Verilog description ~~of a circuit~~ can be mapped to hardware by automated tools.

Ex) (Half-adder).

interface



implementation

We want to write statements that describe this hardware.

```
module halfAdder (A, B, sum, cout);
    input A, B;
    output sum, cout;

    assign sum = A ^ B;    // XOR.
    assign cout = A & B;   // AND.

endmodule.
```

adder.v

Verilog code

adder.v
test.v
stimulus.v

→ Verilog compiler → Verilog simulator (modelsim)

→ Synthesis tools (Xilinx) (logic synthesis / optimization)

→ netlist (hardware) (exact description of circuit)

Key things to note:

- Continuous assignment.
  - → if A or B changes, the assign statement will update sum accordingly.

- Concurrent: the assign statements are concurrent.
  - i.e. one does not execute after the other
  - you can swap these statements in code, does not matter: produces same hardware

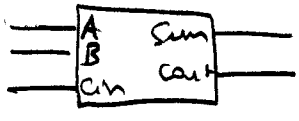
- A good way to think about Verilog is to always keep in mind what it is going to be synthesized into.
- It describes hardware. (~~not sequential execution~~)

- An alternative way to write assigns above:

```
[ xor (sum, A, B);    //instantiates an Xor gate
  and (cout, A, B);   //instantiates an AND gate ]
```

↑ a Verilog primitive.

Ex2 (Full-adder). [Ask class.]




```
module fullAdder (A, B, cin, sum, cout);
    input A, B, Cin;
    output sum, cout;

    assign sum = A ^ B ^ Cin;
    assign cout = (cin & (A ^ B)) | (A & B);
endmodule
```

(order does not matter. (but make sure that when you call fullAdder, you have it in this order))

(or) (do not use +)

Operator precedence:

& (high)

^

| (low)

Therefore:

```
assign cout = cin & (A^B) | A&B;
```

is another way to write the last statement.

+ Hierarchical design:
(Modular)
Given a task/hardware to build, we decompose it into smaller modules.

Ex) (Ripple-carry adder): 4-bit (Chapter #5).

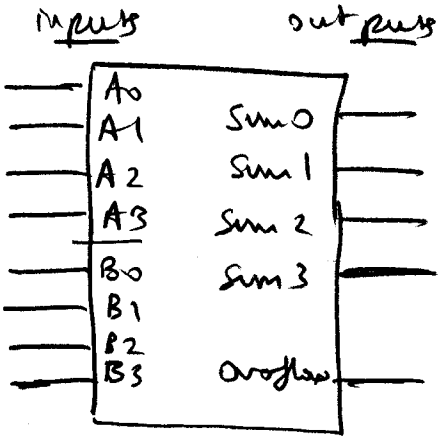
Variable types in Verilog:

- `wire a;`
  - represents a wire.
  - Variables are wires by default.
    (e.g. above example, all vars are implicitly wire.)
- We will talk about other types later.

— (Hierarchical) design:
Modular

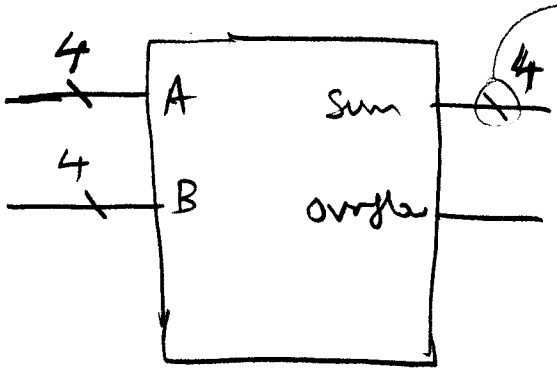
• Implement a large piece of hardware by dividing it up into modules.

Ex3) (Ripple-carry adder): (4-bit unsigned #'s)



module / interface

A vector representation for interface:

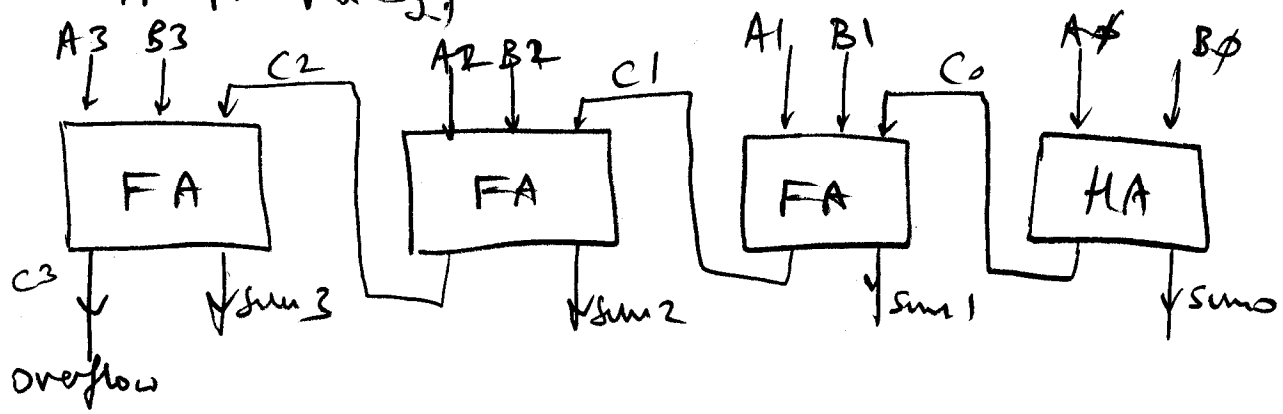


called a vector or bus

(has two meanings: two is one of them)

Implementation: (we saw in lecture #2)

(implement it in verilog)

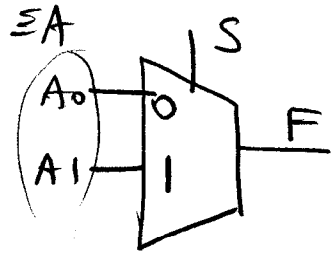


```
//adds 4-bit unsigned numbers.
module rippleCarryAdder (A, B, sum, overflow);
    input [3:0] A;
    input [3:0] B;
    output [3:0] sum;
    output overflow;

    // need internal wires;
    wire [3:0] carry;
    (need to declare every variable.)

    // Key: simply wire up the hardware in picture

    halfAdder M0(A[0], B[0], sum[0], carry[0]);
    fullAdder M1(A[1], B[1], carry[0], sum[1], carry[1]);
    fullAdder M2(A[2], B[2], carry[1], sum[2], carry[2]);
    fullAdder M3(A[3], B[3], carry[2], sum[3], carry[3]);

    assign overflow = carry[3];

endmodule
```

selection operator

order of these statements does not matter!

Ex 4) (2:1 mux):



#1:

```
module mux2_1 (A, S, F);
    input [1:0] A;
    input S;
    output F;

    assign F = S & A[1] | ~S & A[0];

endmodule
```

precedence: ~ & ^ |

complement ( )

#2 replaced by:

```
assign F = S ? A[1] : A[0];
```

"conditional operator"

same semantics as ternary operator in C.

continuous assignment.

[Write Verilog to implement a 4:1 mux using two 2:1 muxes).

- The always construct (procedural.)

Ex 5) (2:1 mux): → reg

instead

```
always @(A or S)
    if (S == 1)
        F = A[1];
    else
        F = A[0];
```

Sensitivity/activation list.

equality operator

executed if f at any least one variable in sensitivity list changes value.

Procedural assignments: (assign based on a sensitivity list).

Ex5] (2:1 mux).

```
module mux2_1 (A, S, F);
    input [1:0] A;
    input S;
    output F;
    reg F;
    always @ (A or S)
        if (S == 1)
            F = A[1];
        else
            F = A[0];
endmodule
```

reg → register type variable.

always @ (A or S) → sensitivity list

if (S == 1) → equality operator

{ procedural block.

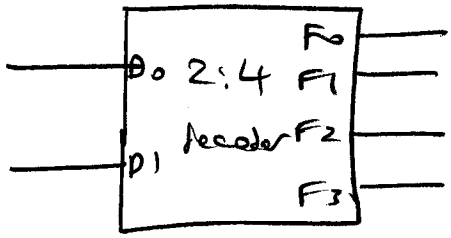
[executes iff at least one variable in sensitivity list changes value.

LHS var must be of type reg.
(does not mean that a register will be synthesized.)

* In general we will use continuous assignments (assign) for comb. logic. It is possible to use always blocks (as above) for comb. logic but there are common pitfalls if more experienced (we will talk more later)

Ex| DECODER:

2:4 decoder:



D1	D0	F3	F2	F1	F0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

~~(Not that hard to do 1 to 1 mapping)~~

~~(Any 1 to 1 mapping in which the output reset~~

- Write the 2:4 decoder in Verilog: using a behavioral description with if else:

```
module decoder2_4 (D, F);
    input [1:0] D;
    output [3:0] F;
    reg [3:0] F;
```

```
always @(D)
    if (D == 2'b00)
        F = 4'b0001;
    else if (D == 2'b01)
        F = 4'b0010;
    else if (D == 2'b10)
        F = 4'b0100;
    else
        F = 4'b1000;
endmodule
```

width, binary

A better way to write this is with case statement

```
case (D)
    2'b00 : F = 4'b0001;
    2'b01 : F = 4'b0010;
    2'b10 : F = 4'b0100;
    2'b11 : F = 4'b1000;
endcase
```