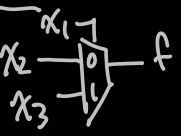
Verilog Review & Lab 3 Prep.

- to describe this mux: 

- one method:

```
module mux2to1 (input x1,x2,x3,
    output f);
    assign f = (!x1 & x2)|(x1 & x3);
endmodule
```

- Another version:

```
module mux2to1 (input x1,x2,x3, output f);
    reg f;
    always @(x1,x2,x3)
      if (!x1)
        f = x2;
      else
        f = x3;
endmodule
```

always block: to use statement such as if-else, case, etc. we must put these stmts. inside an always block.

@(x1,x2,x3) is called the sensitivity list. Any signal that can directly affect an output signal from the always block (f in this case) must be in the sensitivity list.

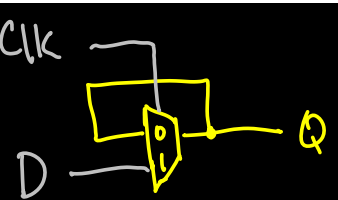
reg f : any signal that is assigned a value inside an always block must be declared as type reg.

- also could write:

```
module mux2to1 (...);
  reg f;
  always @(*)
  begin *
    if (!x) f = x2;
    else f = x3;
  end *
end module
```

* begin-end is needed whenever the always block has more than one Verilog statement (if-else is just one stmt)

Recall




```
module D_latch (input D, Clk, output reg Q);
  always @ (D, Clk)
    if (Clk == 1'b1)
      Q = D;
end module
```

* Note: if there is no else, then this means "stay unchanged". i.e., it implies

```
else
  Q = Q;
```

Flip-flops

```
module D-flipflop (input D, Clock,
    output reg Q);
    always@ (posedge Clock)
      Q <= D;
end module
```

* this convention of using posedge in the sensitivity list causes the Verilog compiler to synthesize a FF for any signal that is assigned a value in this always block.

Note: <= assignment is used for FF; it is called a non-block assignment (later)

Midterm practice

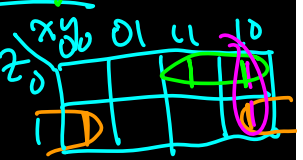
- Given a SOP expression for a function, f, derive the POS using B. algebra. (in minimal form)

$f = \bar{x}\bar{z} + yz$ SOP f

$\therefore \bar{f} = \overline{(\bar{x}\bar{z} + yz)}$

$\bar{f} = (\overline{\bar{x}\bar{z}}) \cdot (\overline{yz}) = (x+z) \cdot (\bar{y}+\bar{z})$ POS $\bar{f}$

$\therefore \bar{f} = x\bar{y} + x\bar{z} + z\bar{y} + z\bar{z}$



or

$\bar{f} = x\bar{z} + z\bar{y}$ (consensus)

$\bar{f} = x\bar{y}(z+\bar{z}) + x\bar{z} + z\bar{y}$

$\bar{f} = x\bar{y}z + x\bar{y}\bar{z} + x\bar{z} + z\bar{y}$

$\bar{f} = x\bar{y}z + z\bar{y} + x\bar{y}\bar{z} + x\bar{z}$ (Absorption)

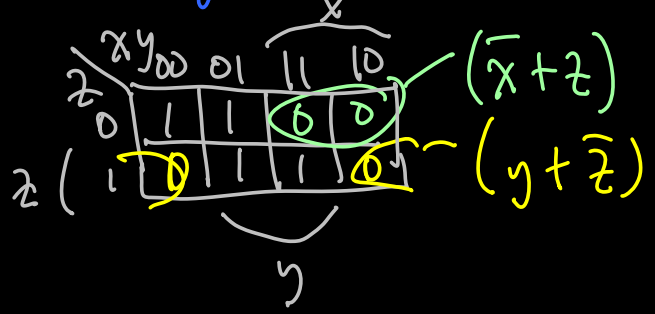
$\bar{f} = z\bar{y} + x\bar{z}$ SOP $\bar{f}$

$\therefore f = \overline{\bar{f}} = \overline{z\bar{y} + x\bar{z}}$

$= (\bar{z}+y)\cdot(\bar{x}+z)$ POS

Check

$f = \bar{x}\bar{z} + yz$

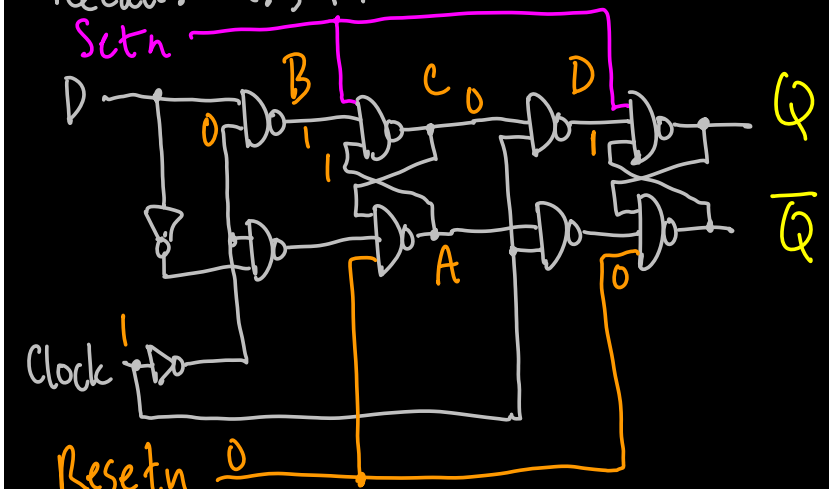


Note: the minimal POS of f is just the "Demorgans equivalent" of the minimal SOP for $\bar{f}$.

Midterm results: Avg = 74%
Lowest = 11%
Highest = 100% (7 students)

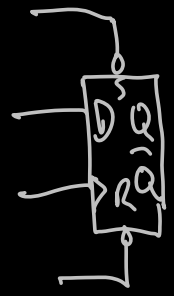
D FF (con't)

Recall: m/s FF



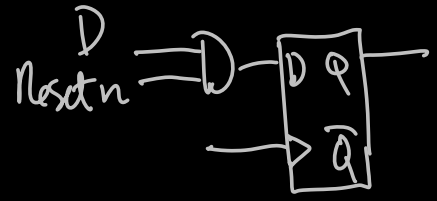
- if Resetn = 0, then $\bar{Q} = 1$. If Clock = 0, then D = 1, so Q = 0 (✓). If Clock = 1, then B = 1. Also, A = 1, ∴ C = 0 and D = 1. So, Q = 0 (✓)
- This is called asynchronous Reset, or asynchronous Clear. The word asynchronous means "independent of the clock signal").

- The Setn is an active-low asynchronous set, or asynchronous preset input.



Note: R might be labelled Clr, or Clear, or Clrn, etc. (see a FF Data Sheet).

Q. How would we design an active-low synchronous reset? (it only performs the reset on a clock edge).



Recall:

```
module D-flipflop (input D, Clock,
                   output reg Q);
  always@ (posedge Clock)
    Q <= D;
end module
```

Asynchronous Reset

```
module D-FF (input D, Clock, Resetn,
             output reg Q);
  always@ (negedge Resetn, posedge Clock)
    if (Resetn==0)
      Q <=0;
    else
      Q <= D;
end module
```

Note: because Resetn is in the sensitivity list, it can directly affect Q, and is therefore an asynchronous input.

```
module D-FF (...);
  always @ (posedge Clock)
    if (Resetn ==0)
      Q <= 0;
    else
      Q <=D;
end module
```

Note: since Resetn is not in the sensitivity

list, it is a synchronous input.

Blocking vs. Non-Blocking Assignments

```
Q = D;   // blocking
Q <= D;  // non-blocking
```

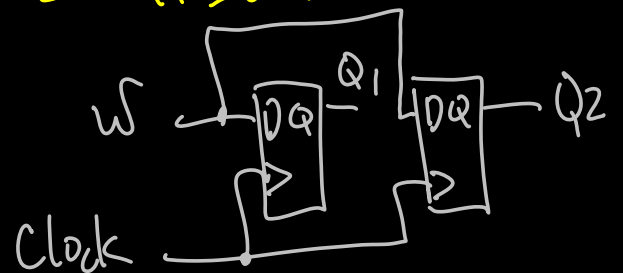
Summary: multiple assignments inside an always block using = (blocking) are evaluated by the compiler in order. But multiple assignments using <= (non-blocking) are evaluated concurrently.

Examples

```
module blah (input w, clock, output
             reg Q1, Q2);
always @(posedge clock)
begin
    Q1 = w;    ①
    Q2 = Q1;   ②
end
endmodule
```

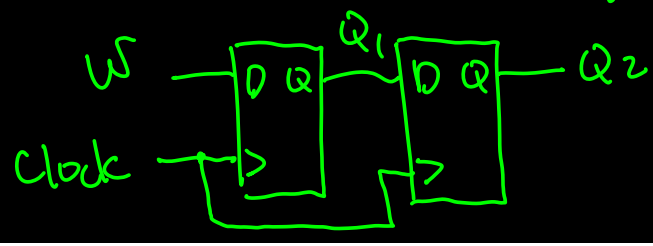
- Since ① is a blocking assignment, then ② "sees" the result of ①, which is Q1 = w.

∴ Q2 = Q1 = w.




```
module blah (input w, clock, output
             reg Q1, Q2);
always @(posedge clock)
begin
    Q1 <= w;   ①
    Q2 <= Q1;  ②
end
endmodule
```

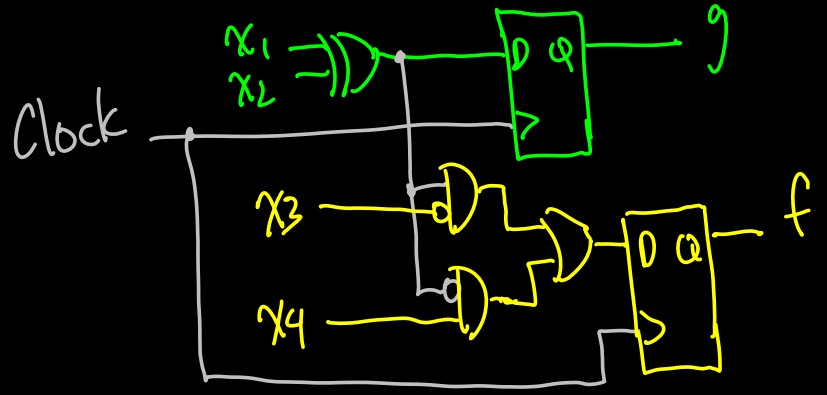
now, ① & ② are evaluated at the same time, which means:



Moral of the story: allows use <= (non-blocking) for sequential circuits. Allowing use = (blocking) for combinational ckts.

Examples

```
module oh_my (input x1, x2, x3, x4, clock,
              output reg f, g);
  always @ (posedge clock)
  begin
    g = x1 ^ x2;
    f = (g & ~x3) | (~g & x4);
  end
endmodule
```



- if we change to <= assignments: