module myxor (c, a, b); // Note: List interface signals

// Declare interface signals
output c;
input a, b;

// declare internal wires
wire nota, notb, anotb, bnota;

// Describe internal circuitry
not(nota, a); // nota = not a, using not() primitive module
not(notb, b); // notb = not b
and(anotb, a, notb); // anotb = a and notb
and(bnota, b, nota); // bnota = b and nota
or(c, anotb, bnota);
endmodule
// Module name is “top level design entity” from our Quartus project
// Interface signals are real board signals
module Lectures(LEDG, KEY);
    // Still have to declare the interface signals
    output [3:0] LEDG; // Note: only using LEDG 3..0
    input [3:0] KEY;
    // Now use our new XOR gate
    myxor(LEDG[0],KEY[0],KEY[1]);
    myxor(LEDG[1],KEY[2],KEY[3]);
endmodule

• Works, but we get some critical warnings
  – It wants us to name instances of “myxor”
  – It wants a timing constraints file (we’ll deal with this later)
// Module name is “top level design entity” from our Quartus project
// Interface signals are real board signals
module Lectures(LEDG, KEY);
    // Still have to declare the interface signals
    output [3:0] LEDG; // Note: only using LEDG 3..0
    input  [3:0] KEY;
    // Now use our new XOR gate
    myxor x1(LEDG[0],KEY[0],KEY[1]);
    myxor x2(LEDG[1],KEY[2],KEY[3]);
endmodule

• Warning: not driving output pins
  – Didn’t tell it what to do with LEDG[3:2]
// Module name is “top level design entity” from our Quartus project
// Interface signals are real board signals
module Lectures(LEDG, KEY);
  // Still have to declare the interface signals
  output [1:0] LEDG;
  input [3:0] KEY;
  // Now use our XOR module
  myxor x1(LEDG[0], KEY[0], KEY[1]);
  myxor x2(LEDG[1], KEY[2], KEY[3]);
endmodule
We’re not telling Quartus what to do with device pins we are not using, e.g. LEDR, SW, etc. 
  – Won’t happen if you follow the QuickStart instructions

This could be dangerous
  – E.g. if we drive a value on an input pin

Fix it
  – Go to Assignments -> Settings
  – Click on “Device” on the left, then the “Device and Pin Options” in the dialog
  – Click on the “Unused Pins” tab
  – Select “As inputs tri-stated”
  – This tells it not to drive any value on any unused pin
module Lectures(LEDG, KEY);
    output [0:0] LEDG;
    input  [3:0] KEY;
    wire xor1,xor2;
    myxor x1(xor1,KEY[0],KEY[1]);
    myxor x2(xor2,KEY[2],KEY[3]);
    myxor x3(LEDG[0],xor1,xor2);
endmodule
Using “assign”

• The compiler can make gates and wires for us:
  module myxor (c, a, b); // Same as before
    output c;
    input a, b;
    assign c = (!b && a) || (!a && b); // Or even “assign c=b^a;”
  endmodule

• See what the compiler made for us
  – R-Click a myxor instance entity in Project Navigator
  – Locate -> Locate in RTL viewer
  – Double-click the box, see your circuit
  – Once there, can browse the entire Lectures circuit
module myxor (OUT, IN1, IN2);
    output OUT;
    input IN1, IN2;
    assign OUT = IN1 ^ IN2;
endmodule

• Why? So you can also do this:
  myxor x3(.OUT(LEDG[0]),.IN1(xor1),.IN2(xor2));
• Or this:
  myxor x3(.IN1(xor1),.IN2(xor2),.OUT(LEDG[0]));
• Either way, now it’s a easier to understand what
  myxor x3 is doing w/o looking at myxor code
Sequential Circuits

• We only designed combinatorial circuits
  – Output is f(inputs) with some delay

• Sequential circuits also have state
  – Output depends on both inputs and state
  – State can also change depending on inputs

• Example KEY[3] turns LEDG[0] on/off
  – One push turns it on, another turns it off, etc.
  – This is not “assign LEDG[0]=KEY[3];”

• We need a flip-flop to store current on/off state
• Could make flip-flop modules and wire them up
  – Gets very tedious very quickly
• Or we can declare a register 😊
  reg state; // One-bit register “state”
• And tell Verilog when and how to update it
  always @(posedge KEY[3]) state <= !state;
• What does this do
  – Whenever there is a positive (rising) edge in KEY[3]
    make new state be equal to ! of previous state
module Lectures(LEDG, KEY);
    output [0:0] LEDG;
    input    [3:0] KEY;
    wire flip = ! KEY[3]; // Let’s give a name to “! KEY[3]”
    reg state;
    always @(posedge flip) state <= !state;
    assign LEDG[0]=state;
endmodule

• Why !KEY[3]?
  – Because it is 0 when pressed and 1 when not
  – Try it!
  – We could use “always @(negedge flip) …” instead
And we get...
• Light that switches on either KEY[2] or KEY[3]
• Easy:
  module Lectures(LEDG, KEY);
    output [0:0] LEDG;
    input  [3:0] KEY;
    wire flip1 = ! KEY[2];
    wire flip2 = ! KEY[3];
    reg state;
    always @(posedge flip1 or posedge flip2) state <= !state;
    assign LEDG[0]=state;
  endmodule
• Does not work!
• Nothing!
• But Verilog compiler can only figure out so much!
• We did get a Critical Warning
  Verilog HDL warning at Class.v(7): can't infer register for assignment in edge-triggered always construct because the clock isn't obvious. **Generated combinational logic instead**
• “I could not figure out the exam question about caching, so I wrote down all I knew about pipelining instead, hoping for some partial credit”
Verilog determines which signal will be used to clock the flip-flop using the following rule:

1. In the “always” @ block, find an edge-triggered signal that is not explicitly mentioned in the body of the block, e.g. in:
   ```verilog
   always @(posedge flip) state <= !state;
   ```
2. It gets confused if it finds more than one, e.g.:
   ```verilog
   always @(posedge flip1 or posedge flip2) state <= !state;
   ```
3. Note that this is not a problem for processor design:
   - All latches use just one clock signal (the clock)
• OK, do “flip1 || flip2”, use to trigger state changes
  – Not exactly the same:
  – Press one (e.g. SW[3]), then press the other, then release them both (in any order)

• Correct operation would be to
  – Turn on (when SW[3] pressed)
  – Turn off (when SW[2] pressed)
  – Do nothing when they are released

• But our “flip1 || flip2” circuit does this:
  – Turn on (when SW[3] pressed)
  – Do nothing after that
• OK, do “flip1 ^ flip2”, use to trigger state changes
  – The XOR approach will fix that!
• Not quite:
  – Let’s say light is off (0) initially
  – Turns on when SW[3] pressed, 1 ^ 0 is 1, and 0 -> 1 transition triggers FF state change
  – **Does nothing** when SW[2] pressed, 1 ^ 1 is 0, but 1 -> 0 transition does not trigger state change
  – **Turns off** when button (e.g. W[3]) released!
  – Does nothing when the other button released
What is really needed is TWO flip-flops
One switched by flip1, the other by flip2
Now XOR those
reg state1, state2;
always @(posedge flip1) state1 <= !state1;
always @(posedge flip2) state2 <= !state2;
assign LEDG[0]=state1 ^ state2;
That’s why the Verilog compiler was confused
  – We asked the poor thing to do it with just one FF!
• Let’s say the light should initially be on
• Ha, we can just flip the result!
  assign LEDG[0]=!state;
• But what if we have something more complex
  (like the instruction counter of a processor)
• We can set initial conditions:
  initial state=1;
Begin-End

- For **always** and **initial**, can have more than one statement inside, just use begin-end, e.g.

```verbatim
initial begin
  state1=0;
  state2=1;
end

always @(posedge KEY[3]) begin
  state1 <= state2;  // Extremely important: These happen simultaneously!
  state2 <= state1;
end
```
• Can do things conditionally, e.g.
  wire click = ! KEY[3];
  wire reset = ! KEY[2];
  reg state;
  initial state=0;
  always @(posedge click) begin
    if(reset)
      state <= 0;
    else
      state <= !state;
  end
  assign LEDG[0]=state;
always @(posedge click) begin
    if(reset)
        state <= 0;
    else
        state <= !state;
end

always @(posedge click or posedge reset) begin
    if(reset)
        state <= 0;
    else
        state <= !state;
end
always @(click or reset) begin
  if(reset)
    state <= 0;
  else
    state <= !state;
end

always @(posedge click or reset) begin
  if(reset)
    state <= 0;
  else
    state <= !state;
end
• Already seen these, e.g. “input [3:0] KEY;” is a 4-bit input whose least significant bit is KEY[0];
• We can use multi-bit signals in expressions, e.g. output [3:0] LEDG;
  input  [3:0] KEY;
  assign LEDG=KEY;
• Careful with operators:
Operators

Logical

! (NOT)
&& (AND)
|| (OR)

Bitwise

~ (NOT)
& (AND)
| (OR)
^ (XOR)

Relational

== (equal)
!= (not equal)
< (less than)

Arithmetic (oh, yes!)

assign LEDG = ~KEY[1:0] + ~KEY[3:2];
Let’s count key presses

always @(posedge !KEY[0])
    state<=state+1; // Note - creates a 4-bit adder
assign LEDG=state;
endmodule

• Now you wish you had Verilog for your LogicWorks assignments in CS 2110, do you 😊
Let’s test the board!

• 10 switches (SW), 10 red LEDs (LEDR)
  – Let’s control each LEDR with its SW
• 4 keys (KEY), 8 green LEDs (LEDG)
  – Let’s control LEDG[0] and LEDG[4] with KEY[0], etc.
• Easy so far:

```verilog
input [9:0] SW;
input [3:0] KEY;
output [9:0] LEDR;
output [7:0] LEDG;
assign LEDR = SW;
assign LEDG[3:0] = ~KEY;
assign LEDG[7:4] = ~KEY;
```
Let’s test the board!

• Four-digit seven-segment display
  – Digits are named HEX0, HEX1, HEX2, HEX3, each is a 7-bit output (one bit per segment)
  – Let’s turn on one segment at a time, each stays on for ½ seconds

• How do we measure ½ second?
  – How do we even know how much time has elapsed?
  – Need some sort of clock!
  – There are several on the board, we’ll use CLOCK_50
    • It’s a 50MHz clock, one tick every 20ns
    • We need to change our HEX displays every 25,000,000 ticks!
input  CLOCK_50;
reg [31:0] Cnt;
initial Cnt=32'd0;
always @(posedge CLOCK_50) begin
  Cnt<=Cnt+32'd1;
  if(Cnt==32'd25000000) begin
    // Change HEX display here
    Cnt<=32'd0;
  end
end
• Can’t just have “assign HEX0=??”
  – HEX display must retain value for ½ second
  – It’s not just a combinatorial function of some input(s)!
• Need a register to store the value to display
  output [6:0] HEX0, HEX1, HEX2, HEX3;
  reg [27:0] HexState;
  assign HEX0=~HexState[ 6: 0];
  assign HEX1=~HexState[13: 7];
  assign HEX2=~HexState[20:14];
  assign HEX3=~HexState[27:21];

• The last 4 lines can be written as
  assign {HEX3,HEX2,HEX1,HEX0}=~HexState;
Changing the display

output [6:0] HEX0, HEX1, HEX2, HEX3;
reg [27:0] HexState;
assign {HEX3,HEX2,HEX1,HEX0}=~HexState;
initial HexState=28'b1;
reg [31:0] Cnt;
initial Cnt=32'd0;
always @(posedge CLOCK_50) begin
  Cnt<=Cnt+32'd1;
  if(Cnt==3'd25000000) begin
    HexState<=={HexState[26:0],HexState[27]};
    Cnt<=32'd0;
  end
end
What do we finally get?