CS 3220
Verilog “always”
module Lectures(LEDG, KEY);
    output [0:0] LEDG;
    input   [3:0] KEY;
    wire flip = ! KEY[3];
    reg state;
    always @(posedge flip)
        state <= !state;
    assign LEDG[0]=state;
endmodule
Which events can be @

- Edges, e.g. “@(posedge signal)”
  - “Execute” when rising or falling edge occurs
  - Results in flip-flop kind of behavior
- Value of signal(s), e.g. “@(signal1 or signal2)”
  - “Execute” continuously as these signals change
  - Results in combinatorial-logic kind of behavior
- Must be "synthesizable"
  - Compiler must know how to translate your code into the kinds of gates/FFs available on the board!
• 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

“state” is a flip-flop triggered by two different clocks
(the FFs on the board have one clock input)
always @(posedge click or posedge reset) begin
  if(reset)
    state <= 0;
  else
    state <= !state;
end

• Our FFs have asynchronous set and reset inputs
  – Works only when used to set FF to 1 or to 0 regardless of what the clock value is
  – Because that’s the only thing that set/reset inputs do!
Edge-triggered “always”

• Use this for all sorts of state
  – Flip-Flops, Registers, Memories, etc.

• Using multiple “always” blocks
  – All statements that (might) write to the same state bit must be in the same always block
  – Different always blocks can write to different bits
  – Will see an example of this

• Always use “<=“ for writing to state bits
  – Writes with “=“ are possible but you should not do it until you know exactly what “<=“ and “=“ do!
always @(posedge clock) begin
  count <= count + 16'd1;
  if(count == 16'd29)
    count <= 16'd0;
end

always @(posedge clock) begin
  if(count == 16'd30)
    count <= 16'd0;
  end
always @(posedge clock) begin
  count <= count + 16'd1;
end

Result is the same as
always @(posedge clock)
  if(count == 16'd29)
    count <= 16'd0;
  else
    count <= count + 16'd1;

Error – can’t synthesize this
reg [15:0] cnt;
always @(posedge clock)
cnt <= cnt + 16'd1;

What does this translate into?

- Creates a 16-bit register named “cnt”
- Creates a combinatorial adder
- Connects the output of the “cnt” register to one input of the adder
- Connects a 16-bit constant 1 (0001 in hex) to the second adder input
- Connects the output of the adder to the D input of register “cnt”
- Connects signal “clock” to the clock input of the “cnt” register

What if we said @(negedge clock) instead?
reg [15:0] upcnt,dncnt;
always @(posedge clock) begin
    upcnt <= upcnt + 16'd1;
    dncnt <= dncnt - 16'd1;
end

- What does this translate into?
  - For upcnt, same as what we had
  - For dncnt, a separate 16-bit register, a subtracter, and wiring to feed dncnt and 16-bit one to it, then feed the result to input of dncnt

- Remember this!
  - Each Verilog statement creates more hardware!
  - Which hardware? Whatever you put in the statement!
• If the highest clock frequency is 100MHz for this:
  
  ```verilog
  reg [15:0] cnt;
  always @(posedge clock)
  cnt <= cnt + 16'd1;
  ```

• What is the highest clock frequency for this:

  ```verilog
  reg [15:0] upcnt,dncnt;
  always @(posedge clock) begin
  upcnt <= upcnt + 16'd1;
  dncnt <= dncnt - 16'd1;
  end
  ```
always @(posedge clock) begin
    if(cnt == 2'd2) begin
        cnt <= 2'd0;
    end else
        cnt <= cnt + 2'd1;
end

- What does this translate into?
always @(posedge clock) begin
  case(cnt)
    0: cnt <= 2;
    2: cnt <= 3;
    3: cnt <= 1;
    1: cnt <= 0;
    default: // Just in case
      cnt <= 0;
  endcase
end
module Class(KEY,HEX0,HEX1);
  input [0:0] KEY;
  output [6:0] HEX0, HEX1;
  wire clock = ! KEY[0];
  reg [3:0] cnthi, cntlo;
  always @(posedge clock) begin
    if(cntlo == 4'd9) begin
      cntlo <= 4'd0;
      if(cnthi == 4'd9)
        cnthi <= 4'd0;
      else
        cnthi <= cnthi + 4'd1;
    end else begin
      cntlo <= cntlo + 4'd1;
    end
  end
  disp dlo(HEX0,cntlo);
endmodule

module disp(sseg,num);
  output [6:0] sseg;
  input [1:0] num;
  assign sseg =
    (num == 2'd0) ? 7'b1000000 :
    (num == 2'd1) ? 7'b1111001 :
    (num == 2'd2) ? 7'b0100100 :
    ...
endmodule
Two types of state assignment in Verilog
- "=" is called a “blocking” assignment
- "<=" is called a non-blocking assignment

We only used "<=" until now
- Computes RHS expression during the cycle
- Latches value into LHS at end of cycle (e.g. posedge)
- All "<=" happen “simultaneously” (at clock edge)

Multiple "<=" to the same reg?
- Same @always block? Last one wins!
- Different @always blocks? Conflict (can’t synthesize)!
What about “=”

- Two big differences
  - Changes value of LHS as soon as RHS computed
  - Delays subsequent “=” until the current one is done

- What does this do?
```
always @(posedge clock) begin
  stage1 <= stage2 + 1;
  stage2 <= stage1;
end
```

- What does this do?
```
always @(posedge clock) begin
  stage1 = stage2 + 1;
  stage2 = stage1;
end
```
Using “=“

- Using it for state is dangerous
  - Does not necessarily create actual state
  - Our stage1 optimized out (reg converted to a wire)!

- What then is it good for?
  - Describing combinatorial logic!

```plaintext
reg [3:0] state,tmp1; // tmp1 is optimized out
always @(state) begin // When state changes, change tmp1
    if(state == 4'd0)
        tmp1=0;
    else
        tmp1=state + 4'd1;
end
always @(posedge clock) state=tmp1; // On clock, state=tmp1
```
parameter S_BITS=5;
parameter [(S_BITS-1):0]
  S_ZERO ={(S_BITS){1'b0}},
  S_ONE ={{(S_BITS-1){1'b0}},1'b1},
S_FETCH1=S_ZERO, // 00000
S_FETCH2=S_FETCH1+S_ONE, // 00001
S_FETCH3=S_FETCH2+S_ONE, // 00010
S_ALU1  =S_FETCH3+S_ONE, // 00011
...
reg [(S_BITS-1):0] state=S_FETCH1,next_state;
always @(state or opcode1 or rsrc1 or rsrc2 or rdst or opcode2 or ALUzero) begin
  // Default next-state is state+1
  next_state=state+5'd1;
  // Set all signals to zero (not shown), this will be the default
  case(state)
    S_FETCH1: {DrPC,LdMAR}={1'b1,1'b1};
    S_FETCH2: {DrMem,LdIR,IncPC}={1'b1,1'b1,1'b1};
    S_FETCH3: begin
      case(opcode1)
        OP1_ALU: begin
          next_state=S_ALU1; // Override the default!
          ...
        endcase
      end
    end
  endcase
end
always @(posedge clk)
state<=next_state;

Note how “parameter” is used here to give a name to a constant. This makes the code below a lot easier to read.

This entire “always” block only creates combinational logic!
module ALU(A,B,CTL,OUT);
  parameter BITS;
  parameter CBITS;
  parameter CMD_ADD = 0,
               CMD_SUB = 1,
               CMD_LT = 2,
               CMD_LE = 3,
               CMD_AND = 4,
               CMD_OR = 5,
               CMD_XOR = 6,
               CMD_NAND = 7,
               CMD_NOR = 8,
               CMD_NXOR = 9;
  input [(CBITS-1):0] CTL;
  input [(BITS-1):0] A,B;
  output [(BITS-1):0] OUT;
  reg [(BITS-1):0] tmpout;
always @(A or B or CTL) begin
  case(CTL)
    CMD_ADD:  tmpout = A+B;
    CMD_SUB:  tmpout = A-B;
    CMD_LT:   tmpout = (A<B);
    CMD_LE:   tmpout = (A<=B);
    CMD_AND:  tmpout = A&B;
    CMD_OR:   tmpout = A|B;
    CMD_XOR:  tmpout = A^B;
    CMD_NAND: tmpout = ~(A&B);
    CMD_NOR:  tmpout = ~(A|B);
    CMD_NXOR: tmpout = ~(A^B);
    default:  tmpout = Something;
  endcase
end
assign OUT = tmpout;
endmodule

tmpout is optimized out!  But only if tmpout is always assigned
module ALU(A,B,CTL,OUT);
  parameter BITS;
  parameter CBITS;
  parameter
    CMD_ADD, CMD_SUB, CMD_LT, CMD_LE, CMD_AND, CMD_OR, CMD_XOR, CMD_NAND, CMD_NOR, CMD_NXOR;
  input [(CBITS-1):0] CTL;
  input [(BITS-1):0] A,B;
  output [(BITS-1):0] OUT;
  reg [(BITS-1):0] tmpout;
  always @(A or B or CTL) begin
    case(CTL)
      CMD_ADD:  tmpout = A+B;
      CMD_SUB:  tmpout = A-B;
      CMD_LT:   tmpout = (A<B);
      CMD_LE:   tmpout = (A<=B);
      CMD_AND:  tmpout = A&B;
      CMD_OR:   tmpout = A|B;
      CMD_XOR:  tmpout = A^B;
      CMD_NAND: tmpout = ~(A&B);
      CMD_NOR:  tmpout = ~(A|B);
      CMD_NXOR: tmpout = ~(A^B);
      default:  tmpout = {CBITS{1'bX}};
    endcase
  end
  assign OUT=tmpout;
endmodule
module ALU(A,B,CTL,OUT);
    parameter BITS;
    parameter CBITS;
    parameter CMD_ADD, CMD_SUB, CMD_LT, CMD_LE, CMD_AND, CMD_OR, CMD_XOR, CMD_NAND, CMD_NOR, CMD_NXOR;
    input [(CBITS-1):0] CTL;
    input [(BITS-1):0] A,B;
    output [(BITS-1):0] OUT;
    wire signed [(BITS-1):0] A,B;
    reg signed [(BITS-1):0] tmpout;
    always @(A or B or CTL) begin
        case(CTL)
            CMD_ADD: tmpout = A+B;
            CMD_SUB: tmpout = A-B;
            CMD_LT: tmpout = (A<B);
            CMD_LE: tmpout = (A<=B);
            CMD_AND: tmpout = A&B;
            CMD_OR: tmpout = A|B;
            CMD_XOR: tmpout = A^B;
            CMD_NAND: tmpout = ~(A&B);
            CMD_NOR: tmpout = ~(A|B);
            CMD_NXOR: tmpout = ~(A^B);
            default: tmpout = {CBITS{1'bX}};
        endcase
    end
    assign OUT=tmpout;
endmodule
Parametrized modules

- Example: code similar for 2-bit, 4-bit, etc. counter
  - Want to write one module for all of these

```verilog
module counter(IN,OUT);
  parameter BITS,INC;
  input IN;
  output [(BITs-1):0] OUT;
  reg [(BITs-1):0] state;
  always @(posedge IN)
    state<=state+INC;
  assign OUT=state;
endmodule

counter #(4,1) cnt1(clock,count);
counter #(INC(3),BITs(8)) cnt2(clock,count);
```
For loops

• Example: sign-extender module

```verilog
module SXT(IN,OUT);
    parameter IBITS;
    parameter OBITS;
    input [(IBITS-1):0] IN;
    output [(OBITS-1):0] OUT;
    reg [(OBITS-1):0] tmpout;
    integer i;
    always @(IN) begin
        tmpout[(IBITS-1):0]=IN;
        for(i=IBITS;i<OBITS;i=i+1) begin
            tmpout[i]=IN[IBITS-1];
        end
    end
    assign OUT=tmpout;
endmodule
```

// This is how you can use this module:
SXT #( .IBITS(4), .OBITS(8)) sxt1(SW[3:0], LEDG);
• Example: sign-extender again

module SXT(IN,OUT);
    parameter IBITS;
    parameter OBITS;
    input [(IBITS-1):0] IN;
    output [(OBITS-1):0] OUT;
    assign OUT={{(OBITS-IBITS){IN[IBITS-1]}},IN};;
endmodule

SXT #( .IBITS(4), .OBITS(8) ) sxt1(SW[3:0], LEDG);