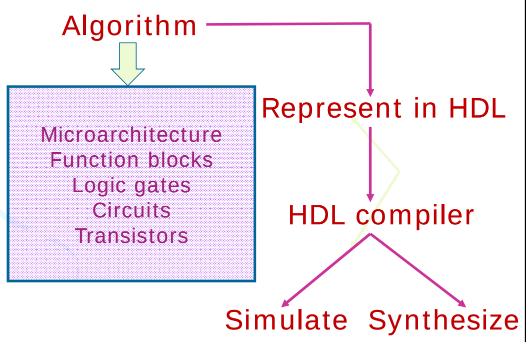
Introduction to Verilog HDL

Mainak Chaudhuri
Indian Institute of Technology Kanpur

Hardware design



HDLs

- Hardware description languages
  - Offers an abstract language for representing computation that needs to be realized in hardware
  - Verilog
    - C-like syntax
    - We will use this
  - VHDL
    - VHSIC HDL
    - Very high speed integrated circuit HDL
    - Ada-like syntax
  - BSV
    - Bluespec System Verilog

3

What is Verilog?

- A Hardware description language
  - Describes digital systems in a convenient way
  - Faster than gate-level simulation
  - Still models the hardware concurrency in detail
  - Possible to carry out both behavioral/functional and structural simulations
  - We will do only behavioral simulation
  - C-like syntax
  - Remember: you are designing hardware; a new variable means you just added a new signal to your circuit
  - Two major types of variables: wire and reg
  - Wire used for direct communication of signals
  - Reg used to store value

4

Verilog program

- A Verilog program represents the target hardware
  - Usually a collection of modules
  - A module is similar to a C function except that it has no return type
  - All inputs and outputs are module arguments
  - The modules are arranged in a hierarchy to represent the target hardware
  - Example: consider designing a four-bit adder
    - A full-adder module
    - A four-bit adder module that instantiates four full-adder modules and connects them properly
    - A top-level module representing the environment of the four-bit adder
    - The top-level module instantiates the target hardware (e.g., four-bit adder module), generates the inputs, and collects the outputs

5

Example: Full adder

```
/* this module receives as input three bits and outputs
   the sum bit and the carry bit */

module fulladder(a,b,c,sum,carry);
input a,b,c;
output sum,carry;
wire sum,carry;

assign sum=a^b^c;    // sum bit
assign carry=((a&b) | (b&c) | (a&c));   //carry bit

endmodule
```

6

Example: Full adder

```
module fulladder_top;
reg  a, b, c;
wire sum, carry;

fulladder uut(a,b,c,sum,carry);
always @(sum or carry)
 begin
   $display("time=%d: %b + %b +
     %b = %b, carry =
     %b\n",$time,a,b,c,sum,carry);
 end

initial
 begin
   #40
  $finish;
 end
```

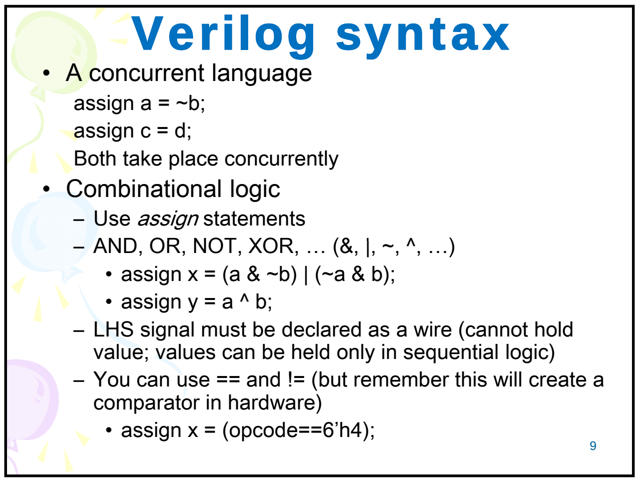
```
initial
 begin
   a = 0; b = 0; c = 0;
   #5
   a = 0; b = 1; c = 0;
   #5
   a = 1; b = 0; c = 1;
   #5
   a = 1; b = 1; c = 1;
 end

endmodule
```

7

Full adder output

```
time=0: 0 + 0 + 0 = 0, carry = 0
time=5: 0 + 1 + 0 = 1, carry = 0
time=10: 1 + 0 + 1 = 0, carry = 1
time=15: 1 + 1 + 1 = 1, carry = 1
```

8



Verilog syntax

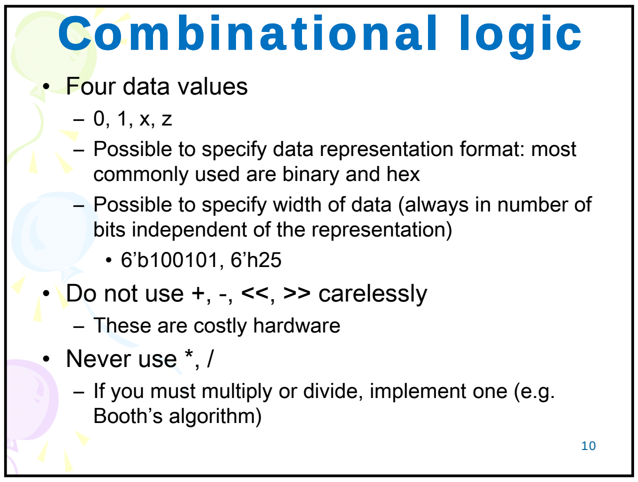
- A concurrent language

```
assign a = ~b;
assign c = d;
```

Both take place concurrently

- Combinational logic
  - Use *assign* statements
  - AND, OR, NOT, XOR, … (&, |, ~, ^, …)
    - assign x = (a & ~b) | (~a & b);
    - assign y = a ^ b;
  - LHS signal must be declared as a wire (cannot hold value; values can be held only in sequential logic)
  - You can use == and != (but remember this will create a comparator in hardware)
    - assign x = (opcode==6'h4);

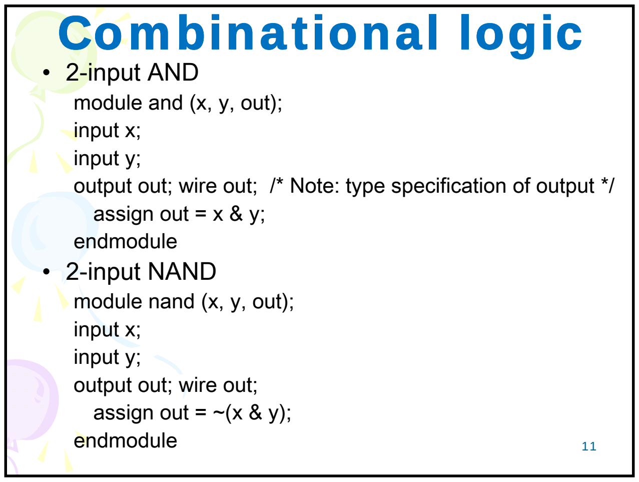
9



Combinational logic

- Four data values
  - 0, 1, x, z
  - Possible to specify data representation format: most commonly used are binary and hex
  - Possible to specify width of data (always in number of bits independent of the representation)
    - 6'b100101, 6'h25
- Do not use +, -, <<, >> carelessly
  - These are costly hardware
- Never use *, /
  - If you must multiply or divide, implement one (e.g. Booth's algorithm)

10



Combinational logic

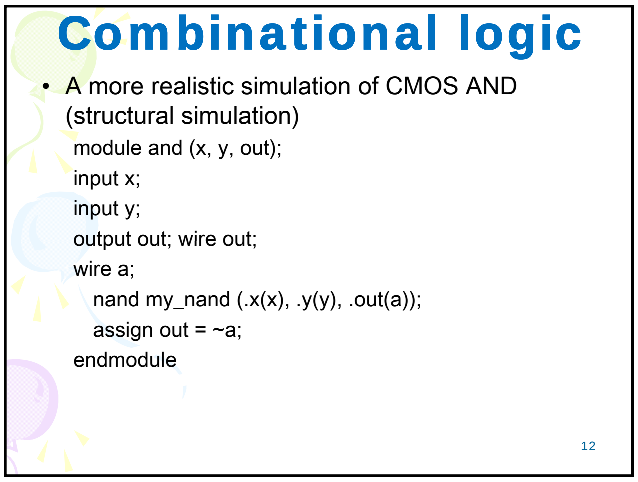
- 2-input AND

```
module and (x, y, out);
input x;
input y;
output out; wire out;  /* Note: type specification of output */
   assign out = x & y;
endmodule
```

- 2-input NAND

```
module nand (x, y, out);
input x;
input y;
output out; wire out;
   assign out = ~(x & y);
endmodule
```

11

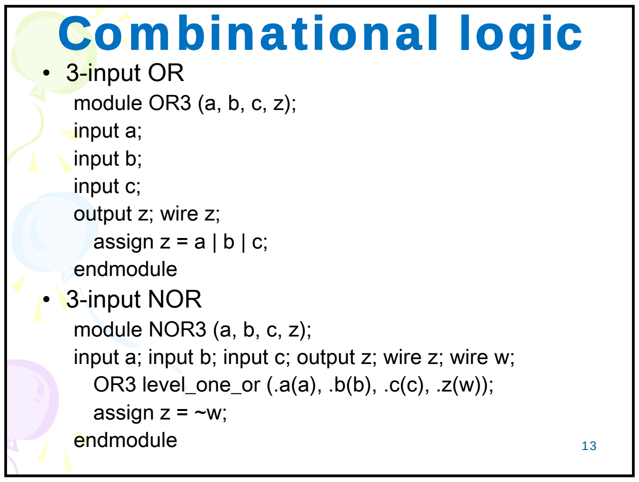


Combinational logic

- A more realistic simulation of CMOS AND (structural simulation)

```
module and (x, y, out);
input x;
input y;
output out; wire out;
wire a;
   nand my_nand (.x(x), .y(y), .out(a));
   assign out = ~a;
endmodule
```

12



Combinational logic

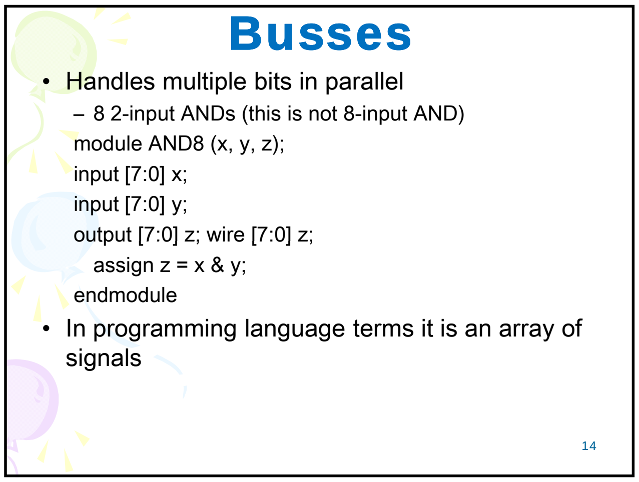
- 3-input OR

```
module OR3 (a, b, c, z);
input a;
input b;
input c;
output z; wire z;
   assign z = a | b | c;
endmodule
```

- 3-input NOR

```
module NOR3 (a, b, c, z);
input a; input b; input c; output z; wire z; wire w;
   OR3 level_one_or (.a(a), .b(b), .c(c), .z(w));
   assign z = ~w;
endmodule
```

13



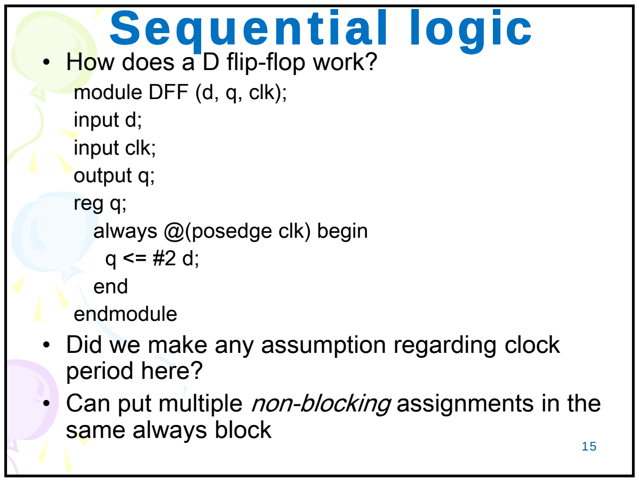
Busses

- Handles multiple bits in parallel
  - 8 2-input ANDs (this is not 8-input AND)

```
module AND8 (x, y, z);
input [7:0] x;
input [7:0] y;
output [7:0] z; wire [7:0] z;
   assign z = x & y;
endmodule
```

- In programming language terms it is an array of signals

14



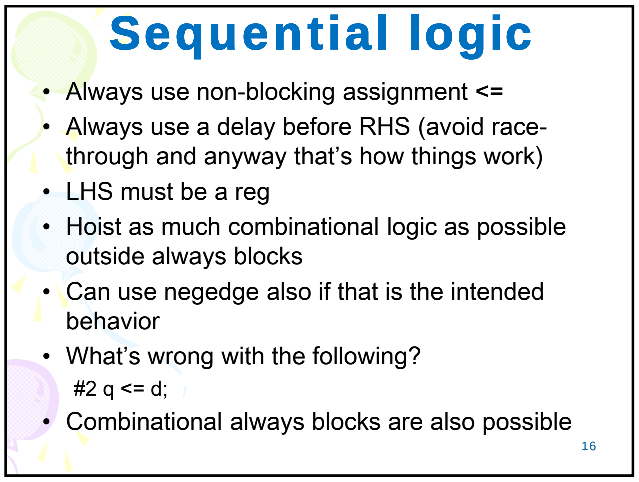
Sequential logic

- How does a D flip-flop work?

```
module DFF (d, q, clk);
input d;
input clk;
output q;
reg q;
   always @(posedge clk) begin
     q <= #2 d;
   end
endmodule
```

- Did we make any assumption regarding clock period here?
- Can put multiple *non-blocking* assignments in the same always block

15



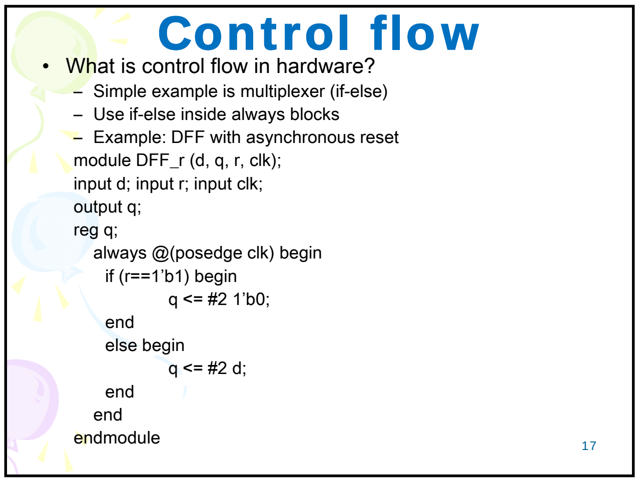
Sequential logic

- Always use non-blocking assignment <=
- Always use a delay before RHS (avoid race-through and anyway that's how things work)
- LHS must be a reg
- Hoist as much combinational logic as possible outside always blocks
- Can use negedge also if that is the intended behavior
- What's wrong with the following?

```
#2 q <= d;
```

- Combinational always blocks are also possible

16

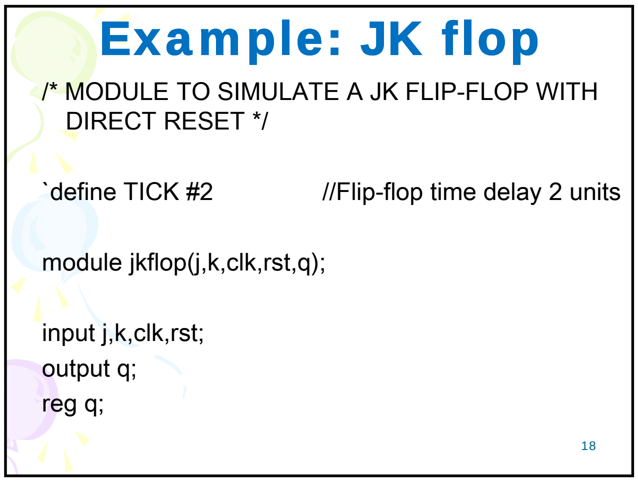


Control flow

- What is control flow in hardware?
  - Simple example is multiplexer (if-else)
  - Use if-else inside always blocks
  - Example: DFF with asynchronous reset

```
module DFF_r (d, q, r, clk);
input d; input r; input clk;
output q;
reg q;
   always @(posedge clk) begin
     if (r==1’b1) begin
              q <= #2 1’b0;
     end
     else begin
              q <= #2 d;
     end
   end
endmodule
```

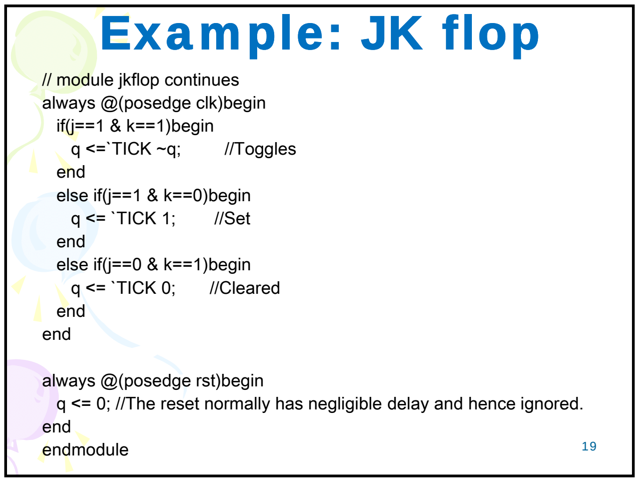
17



Example: JK flop

```
/* MODULE TO SIMULATE A JK FLIP-FLOP WITH
   DIRECT RESET */

`define TICK #2              //Flip-flop time delay 2 units

module jkflop(j,k,clk,rst,q);

input j,k,clk,rst;
output q;
reg q;
```

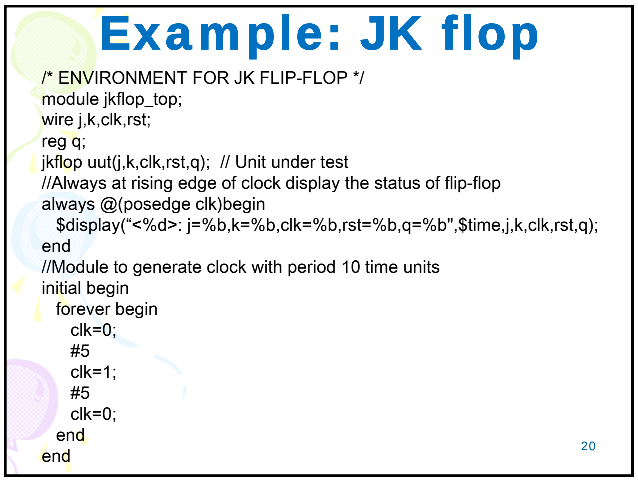
18



Example: JK flop

```
// module jkflop continues
always @(posedge clk)begin
   if(j==1 & k==1)begin
      q <=`TICK ~q;         //Toggles
   end
   else if(j==1 & k==0)begin
      q <= `TICK 1;        //Set
   end
   else if(j==0 & k==1)begin
      q <= `TICK 0;       //Cleared
   end
end

always @(posedge rst)begin
   q <= 0; //The reset normally has negligible delay and hence ignored.
end
endmodule
```

19



Example: JK flop

```
/* ENVIRONMENT FOR JK FLIP-FLOP */
module jkflop_top;
wire j,k,clk,rst;
reg q;
jkflop uut(j,k,clk,rst,q);  // Unit under test
//Always at rising edge of clock display the status of flip-flop
always @(posedge clk)begin
   $display(“<%d>: j=%b,k=%b,clk=%b,rst=%b,q=%b",$time,j,k,clk,rst,q);
end
//Module to generate clock with period 10 time units
initial begin
   forever begin
      clk=0;
      #5
      clk=1;
      #5
      clk=0;
   end
end
```

20

Example: JK flop

```
//Sample test values to run simulation (module jkflop_top continues)
initial begin
  j=0; k=0; rst=1;
  #4
  j=1; k=1; rst=0;
  #50
  j=0; k=1; rst=0;
  #20
  j=1; k=1; rst=0;
end
//Carry out simulation for 100 units of time
initial begin
  #100
  $finish;
end
endmodule
```

JK flop output

```
< 5>: j=1,k=1,clk=1,rst=0,q=0
< 15>: j=1,k=1,clk=1,rst=0,q=1
< 25>: j=1,k=1,clk=1,rst=0,q=0
< 35>: j=1,k=1,clk=1,rst=0,q=1
< 45>: j=1,k=1,clk=1,rst=0,q=0
< 55>: j=0,k=1,clk=1,rst=0,q=1
< 65>: j=0,k=1,clk=1,rst=0,q=0
< 75>: j=1,k=1,clk=1,rst=0,q=0
< 85>: j=1,k=1,clk=1,rst=0,q=1
< 95>: j=1,k=1,clk=1,rst=0,q=0
```

Combinational always

- All RHS signals must appear in sensitivity list
- Same set of LHS on all paths
- All cases must be covered (otherwise state-holding)

```
always @(sel or a) begin
   if (sel==2'b0) begin
     z = 1'b0;
   end
   else if (sel==2'b1) begin
     z = a;
   end
end
```

Blocking assignments

- Use of blocking assignment "=" inside always block
  - Executes before the next assignment can begin
  - Unlike non-blocking assignment "<=" where all RHS signals are evaluated simultaneously
  - Example:

```
always begin
     C = 1;
     A <= C;
     B = C;
end
```

Case statement

- Helps in simulating wide multiplexors

```
always @(sel or a or b or c) begin
   case (sel)
     2'b00: z = a;
     2'b01: z = b;
     2'b10: z = c;
     default: z = 1'bx;      // Note that x is a constant here
                             // (don't care)
   endcase
end
```

25

Concatenation

- Syntax: R{E1, E2,…, En} means R repetitions of concatenation of E1, …, En.

```
reg [15:0] a;
reg [31:0] b;
wire [31:0] out;
   assign out = {16{a[15]}, a} + b;
```

26

Register file / memory

- Essentially a 2D array of "signals"

```
reg [31:0] register_file [0:7];
wire [31:0] rf_bus;
wire r2b4;
   assign rf_bus = register_file [2];
   assign r2b4 = rf_bus[4];
```

- Cannot do register_file[2][4]; (this is illegal because this is not how memory works)

27

Tri-state

- Use of the z state

```
reg [31:0] mem [0:7];
wire [2:0] a;
wire [31:0] d;
wire read_enable;
   assign d = read_enable ? mem[a] : 32'bz;
```

28

Extra hardware

- Think while writing your program
  - You are building a hardware, not software

```
always @(posedge clk) begin
   if (x) begin
     z <= #2 a + b;
   end
   else if (y) begin
     w <= #2 a + b;
   end
   else begin
     r <= #2 a + b;
   end
end
```

Verilog is concurrent

- The big difference with C (get used to it)

```
initial begin
   a = 1'b0;
   b = 1'b0;
end

always @(posedge clk) begin
   a <= #1 1'b1;
   b <= #1 a;
end
```

What are the waveforms of a and b? (assume clk is 10 units and 50% duty cycle)

Built-in functions

- Very useful for debugging
  - Can use $display to print signals (allowed inside always and initial blocks)
  - Can use $monitor to catch changes on signals (allowed inside initial blocks only)
  - $time returns verilog's in-built clock time
  - $finish terminates simulation

Verilog preprocessor

- Just like C, minor difference in syntax

```
`define store_opcode 6'h4
`define TICK #2
`include "opcodes.h"

always @(posedge clk) begin
   q <= `TICK d;
end
```

Register or wire?

- General confusion: how do I decide if a signal should be declared as a reg or a wire?
  - Forget about the programming language and think about the hardware
  - If in the hardware something turns out to be a wire, it should be declared as wire and driven every cycle (otherwise it becomes Z or X)
  - Otherwise make it reg (state holding)

33

Verilog tips

- Suppose module M has an output Q of type reg
- Module M’ instantiates M and passes a reg R for Q
- This is illegal
  - A very common mistake observed in beginners
  - Illegal because this instantiation implicitly connects a register to another register without specifying a clock
- Can connect only a wire to a register through instantiation
  - This will make sense if you think about the hardware
- Module M’ can only pass a wire for Q

34

Verilog tips

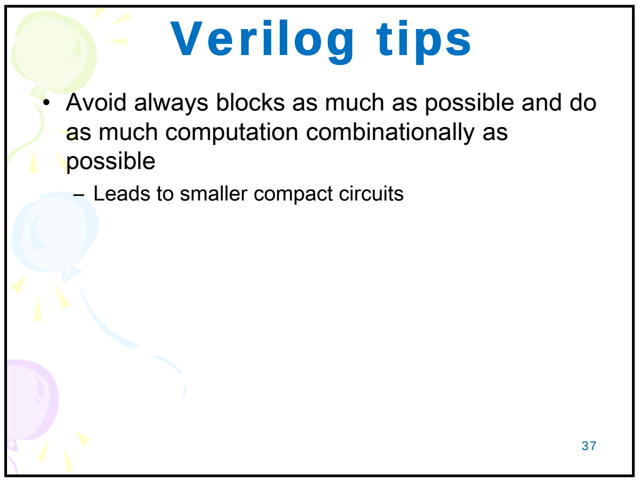
- If the LHS of an assignment is a reg, use non-blocking assignment i.e., <=
  - This will generate flip-flops, which is what we want
  - Blocking assignment inside always block is typically used to model combinational logic only
  - Careless use of blocking assignment inside always blocks can lead to generation of latches
  - Latches are difficult to handle in terms of timing

35

Verilog tips

- If there are non-blocking assignments inside an always block, make the logic edge-triggered
  - This will generate flip-flops as opposed to latches
  - Use posedge or negedge clock as the triggering signal as opposed to posedge or negedge of some other signals
    - Leads to cleaner and simpler circuits because all flip-flops can have a common clock

36



Verilog tips

- Avoid always blocks as much as possible and do as much computation combinationally as possible
  - Leads to smaller compact circuits

37