Section 7

Behavioral Modeling

Behavioral Modeling

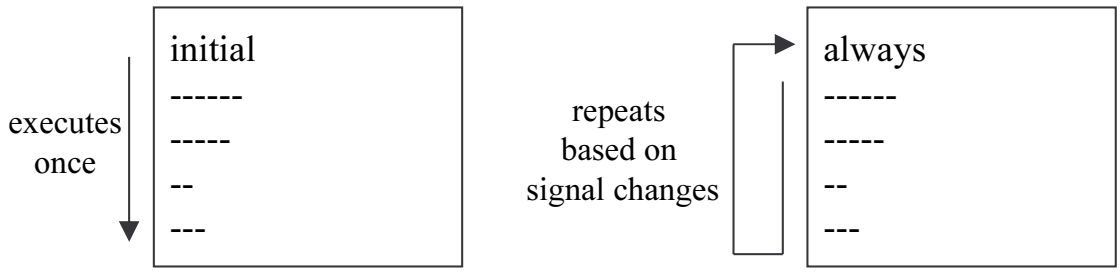
- Behavioral Modeling enables you to describe a system at a high level of abstraction
- Use Behavioral Modeling to describe the functionality of the system.
- Verilog uses high-level language constructs much like a traditional programming language.
- Behavioral modeling in Verilog is described by specifying a set of concurrently active procedural blocks.
- RTL code is a subset of Behavioral modeling. Some behavioral constructs are synthesizable.

Structured Procedures

- There are two procedure statements in Verilog
  - ***always***
  - ***initial***
- All behavioral modeling statements appear within an ***always*** or ***initial*** block.
- Each always and initial block represents separate activity within the module (concurrency).

Procedural Blocks

- Procedural blocks are constructed from the following components:
  - Procedural assignment statements
  - High-level constructs
  - Timing Controls



Procedural Assignments

- Assignments made in procedural blocks are called procedural assignments

Procedural block executes whenever a, b, or c changes.

All signals on the left hand side must be defined as register data type (such as reg). If undeclared, Verilog defaults to a 1-bit wire.

```
module adder (out, a, b, cin);
input a, b, cin;
output [1:0] out;

wire a, b, cin;
reg half_sum,  half_carry;
reg [1:0] out;

always @(a or b or cin)
begin
  half_carry = a ^ b ^ cin;
  half_sum  = ((a & b) | (a & !b & cin) | (!a & b & cin));
  out = {half_carry, half_sum};
end
endmodule
```

Procedural Execution Control

- Execution of procedural blocks can be specified in three different ways
  - Simple delays: #<delay>
    - Specify delay before or after execution for a number of time steps.
  - Edge-sensitive controls: ***always*** @(<edge> <signal>)
    - Execution occurs only on a signal edge. Optional keyword '***negedge***' or '***posedge***' can be used to specify signal edge for execution.
  - Level sensitive controls: ***wait***(<expr>)
    - Execution waits until the expression is true. If the expression is already true, block executes immediately.

Procedural Execution Control (cont.)

Delay

```
module …
…
 initial begin
  #10 a = b;
  #10 c = d;
 end
endmodule
```

Always

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

always @(y or z)
 begin
  x = y & z;
 end
endmodule
```

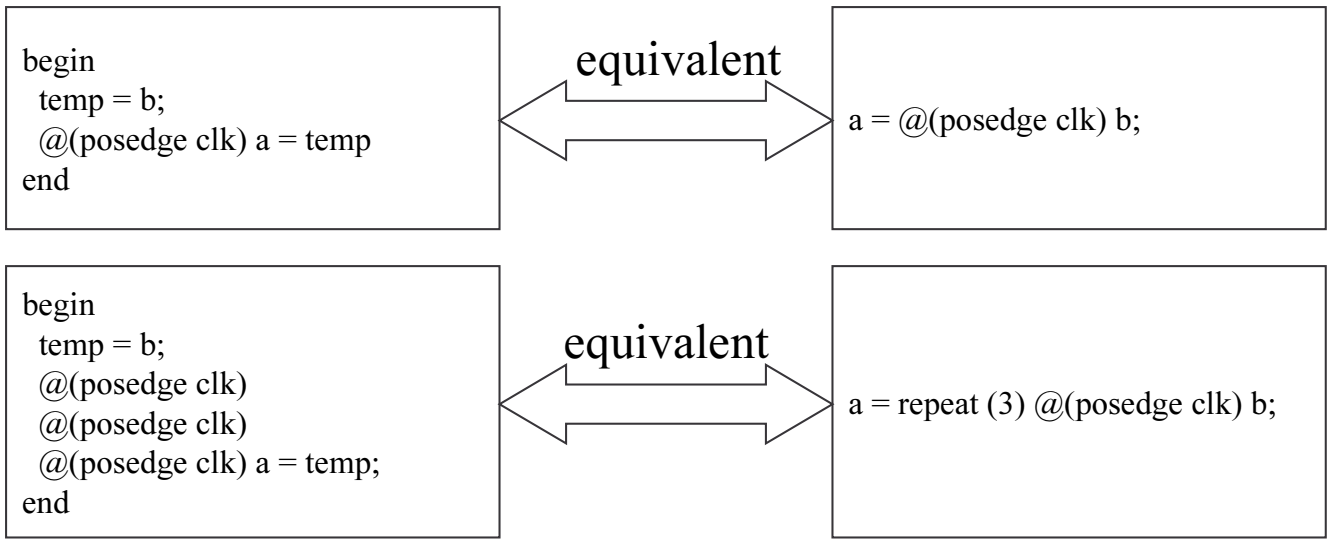
concurrent execution

Wait

```
module …
…
always @(posedge clk)
begin
   wait (t = = s)
      begin
         y <= z;
      end
end
endmodule
```

Intra-Assignments Execution

- Intra-Assignments removes the need for temporary storage.
  - Syntax: LHS = <execution control> RHS



Note: intra-assignments are not synthesizable

Block Statements

- Block statements are used to group two or more statements together.
  - Two types of blocks (can be used with '***initial***' and '***always***' blocks)
    - Sequential: enclosed with keywords '***begin***' and '***end***'.
    - Parallel: enclosed with keywords '***fork***' and '***join***'.

```
always @(stuff)
  begin
        a = 1;
  #5    b = 2;
  #10 c = 3;
  end
```

```
always @(stuff)
  fork
        a = 1;
  #5    b = 2;
  #10 c = 3;
  join
```

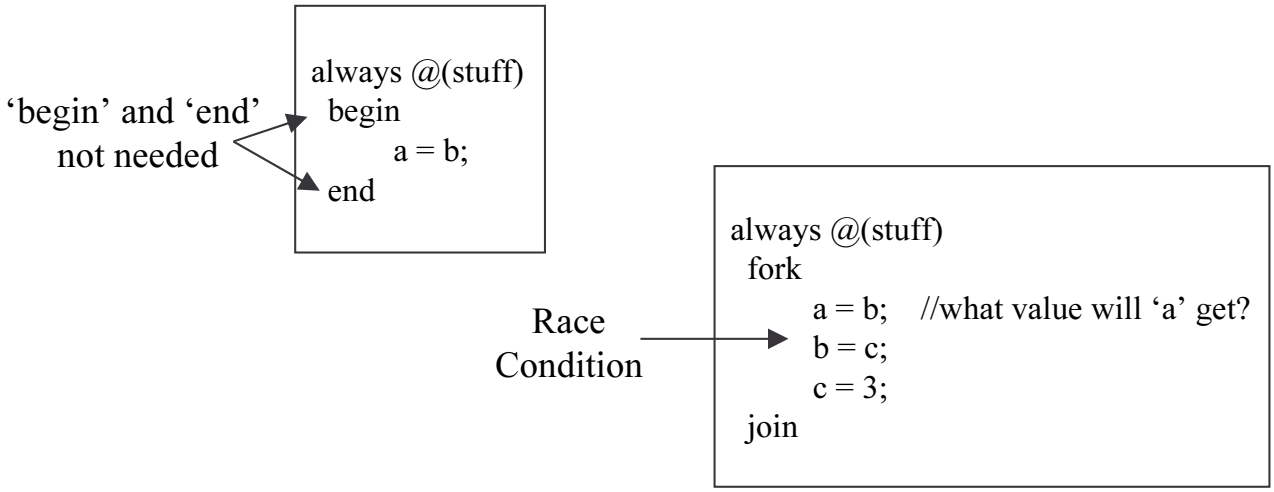
```
initial
  begin
        a = 1;
  #5    b = 2;
  #10 c = 3;
  end
```

```
initial
  fork
        a = 1;
  #5    b = 2;
  #10 c = 3;
  join
```

Note: fork/join constructs are not synthesizable

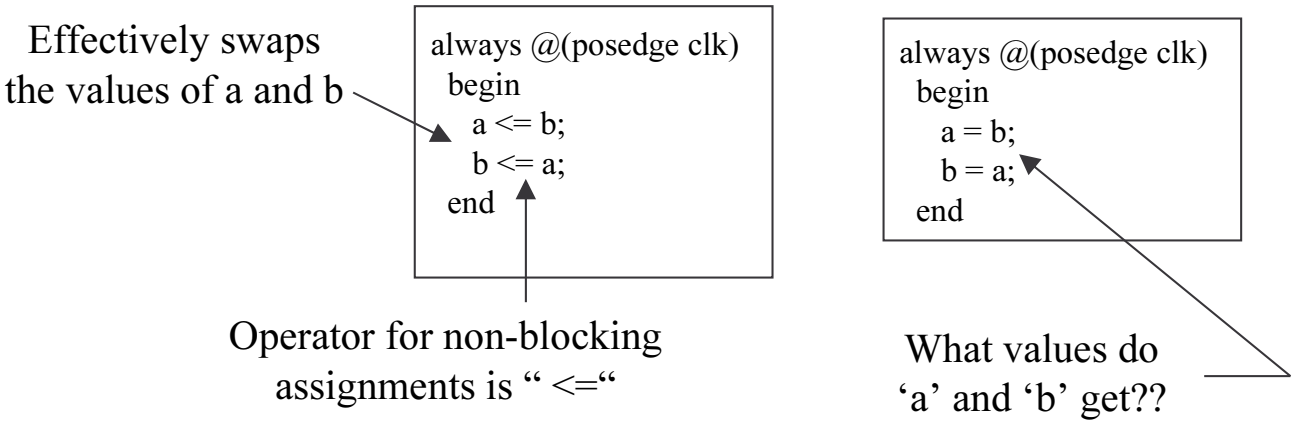
Some Things to Remember!

- ***begin*** and ***end***, ***fork*** and ***join***, are only needed when there are two or more statements in a procedural block.
- Beware of race conditions in procedural blocks.



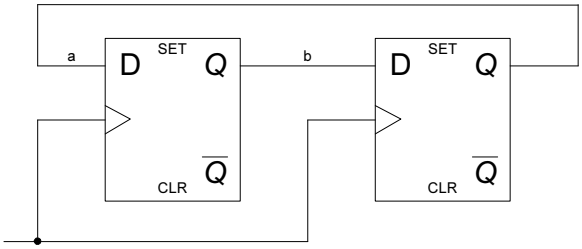
Non-blocking vs. Blocking Assignments

- A blocking assignment is executed immediately
- A non-blocking assignment is executed in two steps:
  - All RHS values are stored by the simulator.
  - At the next timing control event, the assignment is executed and the LHS is assigned the stored values.



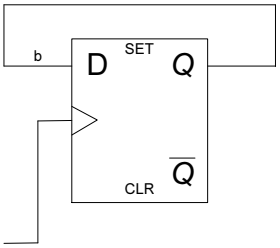
Non-blocking vs. Blocking Assignment Synthesis

```
always @(posedge clk)
  begin
    a <= b;
    b <= a;
  end
```



The non-blocking assignment infers two flops with b as the intermediate variable

```
always @(posedge clk)
  begin
    a = b;
    b = a;
  end
```



The blocking assignment implies ‘a’ is a temporary variable that is optimized away in synthesis

Conditional Statements: IF

- You can use ***if*** and ***else if*** statements in Verilog.
  - '***if***' and '***else if***' statements can be nested within other '***if***', and '***else if***' statements.

```
always @(posedge clk)
begin
  if (stuff)
     bird <= bat;
  else if (small)
      if (frog)
         cow <= pig;
      else
         dog <= cat;
  else
      sheep <= goat;
end
```

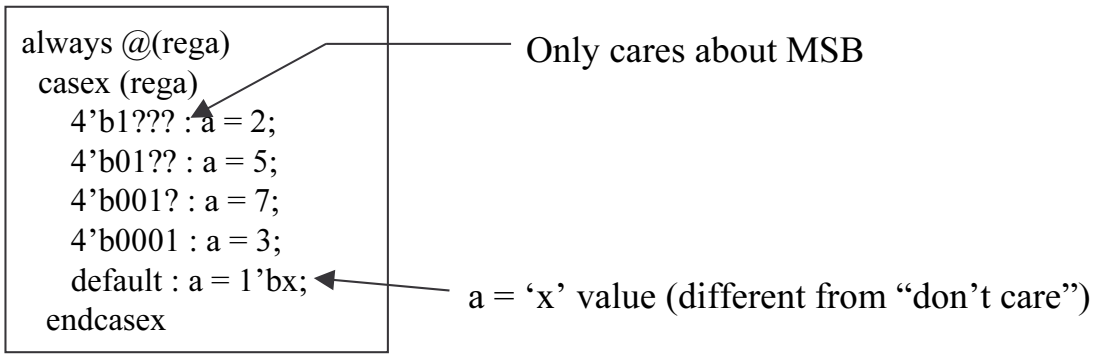
Conditional Statements: case

- '***case***' is a conditional statement that executes based on whether the expression matches a specified condition.
  - '***case***' does a bit-by-bit comparison searching for an exact match ('x' and 'z' included).
  - '***default***' statement is optional, but its usage is part of good programming.

```
always @(rega)
  case (rega)
     2'b00 : a = 2;
     2'b01 : a = 5;
     2'b10 : a = 7;
     2'b11 : begin a = 3;
                    #5 a = 1;
             end
     default : a = 0;
   endcase
```

Conditional Statements: casex and casez

- ***casex*** or **casez** allows the user to specify "don't care" conditions in the case statement.
  - '***casex***' uses 'x', 'z', and '?' as "don't care".
  - '***casez***' uses 'z' and '?' as "don't care".



Hint: When using casex or casez, only use '?' as don't care so as not to confuse with the 'x' (unknown), and 'z' (high impedance) states.

Looping Statements: Repeat

- A '***repeat***' loop executes a block of statements for a fixed number of times.
  - Syntax: ***repeat*** (<integer>)

Counter

```
initial begin
   a = 0;
  repeat (40) begin
      a = a + 1;
  end
end
```

Clock Pulse

```
initial begin
   clk = 0;
   repeat (100) begin
      #5 clk = ~clk;
   end
end
```

Note: repeat is not synthesizable

Looping Statements: while

- A '***while***' loop executes a block of statements when the condition of the '***while***' loop is true.
  - Syntax: ***while*** (<condition>)

```
initial fork
   a = 0;
  repeat (40) begin
      a = a + 1;
  end
  while (a < 40) begin
      z = z + 2;
  end
join
```

```
initial fork
   clk = 0;
   repeat (100) begin
      #5 clk = ~clk;
   end
   while (clk) begin
       a = a + 1;
   end
join
```

Note: while is generally not synthesizable

Looping Statements: forever

- A '***forever***' loop executes a block of statements until the simulation is stopped by a control external to the '***forever***' loop (such as ***$finish*** in separate procedural block).
  - Syntax: ***forever***
  - Should be used as the last statement of a procedural block (all statements past '***forever***' will never be reached to execute).
  - Generally used in testbenches. Effective modeling for continuous clock signals.

```
initial begin
   clk = 0;
   forever #10 clk = ~ clk;
end
```

Note: forever is not synthesizable

Looping Statements: for

- A '***for***' loop executes a block of statements until the loop condition is met.
  - Syntax: ***for*** (<initialization> ; <condition> ; <operation>)

```
module (…)
…
reg [15:0] mem [1023:0];
integer i;
always @(init) begin
  for ( i = 0 ; i < 1024; i = i + 1)
    begin
      mem[i] = 16'b0;
      $display("Initializing memory location %d to 0", i);
    end
end
```

Note: well structured for loops are synthesizable

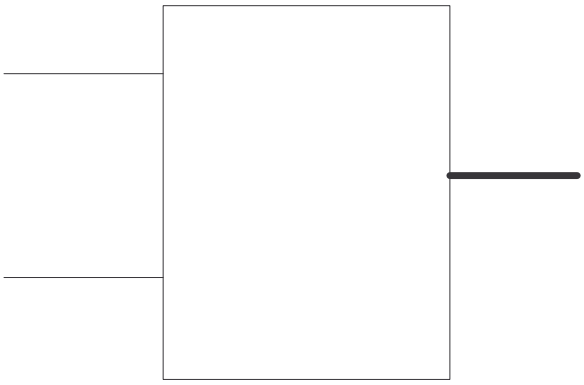
Continuous Assignments

- Combinational logic can be modeled with continuous assignments.
  - Continuous assignments take place outside of procedural blocks.
  - Continuous assignments can only contain simple LHS assignment statements.
  - The LHS of a continuous assignment must be of 'net' data type.
  - The nature of a continuous assignment makes the '@' control unnecessary.

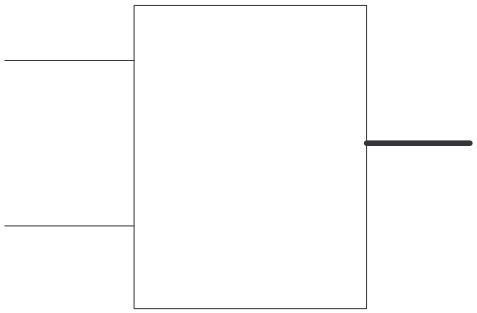
```
wire, out, stuff, q;

assign out = ~in;
assign stuff = (input1 & !input2) | input3;
assign q = sel ? b : a;
```

RTL Example: 4-bit Ripple Counter

- 4-bit ripple counter:
  - Can be modeled behaviorally as a simple adder.



Ripple Counter



```
module ripple (Q, CLOCK, CLEAR);

  input CLOCK, CLEAR;
  output [3:0] Q;

  reg [3:0] Q;

  always @(negedge CLOCK or posedge CLEAR) begin
     if (CLEAR)
       Q <= 0;
     else
       Q <= Q + 1;
  end

endmodule
```


CLOCK

Ripple Counter

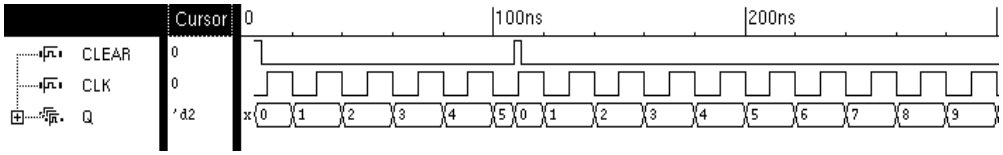
CLEAR

Ripple Counter Test

- The counter is tested the same way the dataflow model was tested.

```
module ripple_tb;

  reg CLK, CLEAR;
  wire [3:0] Q;

  ripple u1 (.Q(Q), .CLOCK(CLK), .CLEAR(CLEAR));

  always #10 CLK = ~CLK;

  initial begin
    $monitor("CLEAR = %b  Q = %d",CLEAR,Q);
    #5 {CLK,CLEAR} = 2'b01;
    #3 CLEAR = 0;
    #100 CLEAR = 1;
    #3 CLEAR = 0;
    #350 $stop;

  end // initial begin

endmodule // tb
```

Ripple Counter Test Results




```
# CLEAR = x  Q =  x
# CLEAR = 1  Q =  0
# CLEAR = 0  Q =  0
# CLEAR = 0  Q =  1
# CLEAR = 0  Q =  2
# CLEAR = 0  Q =  3
# CLEAR = 0  Q =  4
# CLEAR = 0  Q =  5
# CLEAR = 1  Q =  0
# CLEAR = 0  Q =  0
# CLEAR = 0  Q =  1
# CLEAR = 0  Q =  2
# CLEAR = 0  Q =  3
# CLEAR = 0  Q =  4
# CLEAR = 0  Q =  5
# CLEAR = 0  Q =  6
# CLEAR = 0  Q =  7
# CLEAR = 0  Q =  8
# CLEAR = 0  Q =  9
# CLEAR = 0  Q = 10
# CLEAR = 0  Q = 11
# CLEAR = 0  Q = 12
# CLEAR = 0  Q = 13
# CLEAR = 0  Q = 14
# CLEAR = 0  Q = 15
# CLEAR = 0  Q =  0
# CLEAR = 0  Q =  1
# CLEAR = 0  Q =  2
```

Review

- What type of logic do you typically model with a continuous assignment?
- What is the difference between blocking and non-blocking assignments?
- What is the main difference between a procedural block with begin - end and a procedural block with fork - join?
- What does the keyword ‘posedge’ mean?
- What are the two procedural conditional constructs in Verilog?