Carnegie Mellon

Out: 1/15/04 Homework 1 Due: 1/29/04

We'll discuss the solutions to these problems at the beginning of class on the due date. Bring your solutions then. Solutions will be handed out, so we can't accept late assignments.

If a problem doesn't specifically state "simulate this," you do not need to turn in a simulation. However you may want to run one to be sure of your answer. If a simulation is requested, you should turn in your source description and a printout of your results showing that your description was successfully executed (i.e., time/date, "no errors", results…). It's best to use the Cadence simulator. Instructions for setting up your environment for using the Cadence simulator are on the course web site.

1. Verilog Hacking (20%)

Write Verilog descriptions of a combinational 2-bit comparator (whose module header is shown below) in each of the following modeling styles. Each of the 3 modules you describe must be *functionally* identical at their ports when you simulate them.

```
module compareTheseTwo2bitUnsignedValues(A, B, AgtB, AltB, AeqB);
      (input   [1:0]    A, B,
      output            AgtB, AltB, AeqB);
      …
```

- **gate level:** Use only Verilog primitive gates — try to reduce the number of gates needed. Each gate should have a rising delay of 4 and a falling delay of 7.
- **behaviorally using procedural statements** (only use always blocks). Write this with one always block. The input-to-output delay should be 4.
- **behaviorally using only continuous assignments.** Write this as succinctly as you can. Assign a rising delay of 4 and falling delay of 7 to each assign statement

a. Write a testbench module for your designs to demonstrate the *functional* equivalence among the above three modules. (Remember, with functional equivalence, *timing* can be different — the question is whether the same function is implemented after all the delays settle out.) On initialization, this module will read an input file of test vectors to apply to these three modules. Use $readmemh (look it up in the index of the book). Apply each successive test at regularly spaced intervals (use a big-enough time). Instantiate all three of your designs so that the same wires feed the inputs of all three comparators simultaneously. Display the outputs of all three comparators using $monitor(). Make sure it is clear which comparator is producing which set of results.

b. Plot, on a timeline, the internal physical events of the gate-level module for one test vector (i.e., all of the gate outputs should be plotted). Start with all values as unknown and then change all of the primary input values (A and B) at some start time (you pick the input values). Plot from this start time until all gates settle to their final value. What is the min, max, and average number of times a gate's value changes? (Count this up over all the gates in your design, but only for one test vector.) You might want to use a $monitor to print these out.

c. If I added another comparator to the system, would any of the above physical event times change? Why/why not?

2. Waveform City (15%)

Using the following input wave form for input b, plot the different wave forms generated from the following statements. Simulate this with a "real" simulator (use the Cadence one available in the ECE cluster). Plot from time 0 through 23. Explain the differences. Why are there so many ways to describe a circuit?

The statements:

not #2 (a, b);

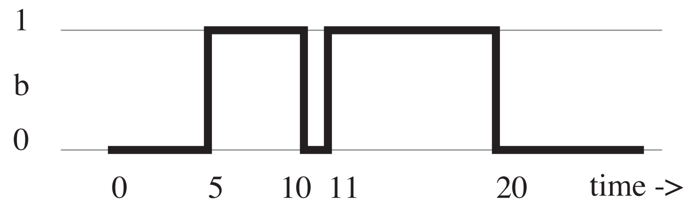
```
assign #2 a = ~b;

always @(b)
      #2 a = ~b;

always @(b)
      a = #2 ~b;

always @(b)
      #2 a <= ~b;

always @(b)
      a <= #2 ~b;
```



3. There's a price to being an author of a book (5%)

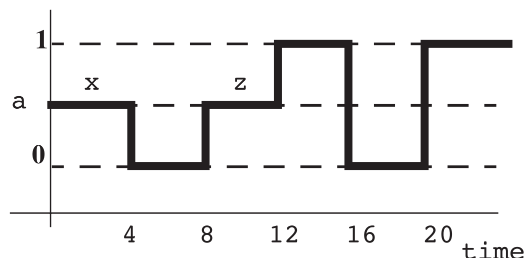
I get lots of e-mail questions asking about Verilog. Here's a revised one. They're trying to make a module that clocks the sum of its inputs. The delay of this adder is being modeled as #30. They think it's missing clock edges. How could that be? Fix the module so that the sum appears #30 after the clock edge but based on values in the system at the time of the clock edge. Behavioral Verilog only. Simulation results not needed.

```
module ThreeInputAdder (sum, in1, in2, in3, clock)
      input [3:0]    in1, in2, in3;
      input          clock;
      output [4:0]   sum;

      always @(posedge clock)
               sum = #30 in1 + in2 + in3;
endmodule
```

4. Waveform Needed (5%)

Here is a waveform that we would like our test-bench to produce. Write two versions of an initial block to produce it. One will use non-blocking statements of the form a <= #n 1'bm, and the other will use blocking statements of the form #n a = 1'bm.



5. Assign with a function (5%)

Rewrite the description from problem 1 so that it uses continuous assign statements that call Verilog functions — that is, all the functionality is contained in functions. Your answer should have the form similar to that shown below. Use the same timing as in problem 1. (You might also be able to use some of the Verilog code from problem 1.) Compile the description but you don't have to turn in a simulation.

```
module DoingItWithFunctionCalls(A, B, AgtB, AltB, AeqB);
      declarations...
      assign AgtB = isAgtB (A, B);
      assign AltB = isAltB (A, B);
      …

      define the functions here...

endmodule
```

6. Don't be so edgy (5%)

I have the darndest time explaining this to 18-240 students! Maybe you could help me out!! Input r is listed in the sensitivity list as being edge sensitive. But the D flip flop that will be synthesized doesn't have an edge-sensitive reset, only an edge-sensitive clock. Please explain if input r is edge sensitive behavior or not.

```
module dff
      (input         d, c, r,
      output reg     q);

      always @(posedge c, negedge r)
               if ~r
                     q <= 0;
               else
                     q <= d;
endmodule
```

7. What are the values (10%)

a. Fill in the table showing the values for the three registers at the given times. Legal answers are 0, 1, x, z, and indeterminate.

name	Value at the end of time 0	Value at the end of time 1
q		
r		
s		
t		

```
reg     q, r, s, t;

initial begin
        q = 1'b0;
        t <= #1 1'b0;
        q <= #1 1'b1;
        s = 1'b1;
        end

always begin
        r = 1'b1;
```

```
s = 1'b0;
wait (t === 1'b0) q = 1'b0;
r = 1'b0;
q <= #10 1'b1;
end
```

b. This doesn't seem to stop executing! Why? If you interrupted the simulator, what would the virtual time be? List all pending events at this time — i.e., ones that have been scheduled but not yet executed.

8. Alex, I'll take #8 for 15% (15%)

There are a number of different methods that a simulator could use to evaluate gate level primitives. One of the ones we mentioned kept track of the number of controlling inputs, and the number of unknown (x) inputs.

a. Show the 4-value truth table for a primitive XOR gate.

XOR	0	1	x	z
0				
1				
x				
z				

Devise a counting method for evaluation and update of an XOR gate. Assume 0, 1, and x are allowable on the inputs. Show pseudo code (with a few comments) for:

b. The update function. What happens when an input changes?

c. The evaluation function. How is the output evaluated?

9. Who needs that #delay anyway (20%)

We have a processor that communicates with a digital-to-analog (DAC) and analog-to-digital (ADC) converter, reading and writing values. The unit has several DACs and ADCs which are addressed. A write cycle takes one clock period, and a read cycle takes two. The processor and DA/ADC are each modeled with an always block. These two communicate via global registers: rwLine, addressLines, and dataLines. The ADC and DAC are modeled as memory arrays. A separate memory array (previousADC) is used to calculate a weighted average; this is the value put on the buslines.

Answer the first two questions by drawing a waveform. Show rwLine and clock as waveforms; write a value on the dataLines and addressLines. Indicate when values should change on these lines. The clock period is 25.

To reason about these two always blocks, assume they each are scheduled to execute starting at the "…" at the same time.

```
always begin // this models the processor which will write/read the DA/ADC.
      // Registers data,
      // addr, and readWrite are internal to the processor and are the
      // address to be read or written depending on readWrite. data is the
      // value to write or the value read back.
      // There are no clock edges up here in the unshown part of this always
      …       //start here
```

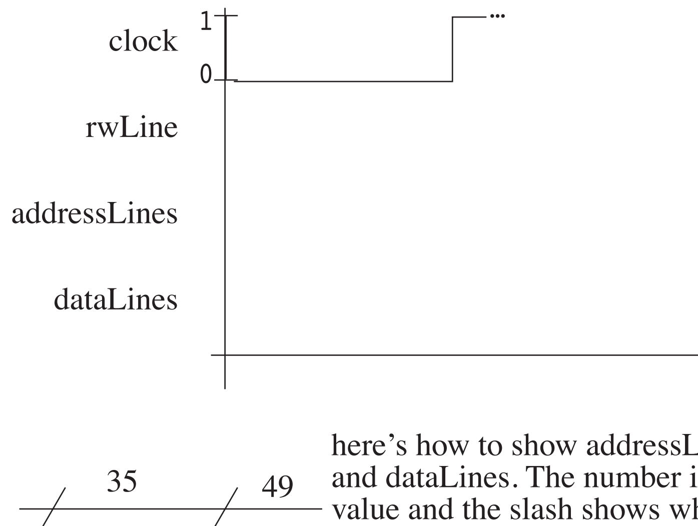
```
        #5 rwLine = readWrite;
        if (readWrite) begin    // write to a DAC
                addressLines = addr;
                dataLines = data;
        end
        else begin        // read— ADC value comes back in next clock cycle
                addressLines = addr;
                @(negedge clock);
        end
        @ (negedge clock);
        if (~readWrite)
                data = dataLines; // value returned by ADC
end

always begin
        //This is the DA/ADC that looks at the buslines (rwLine, addressLines,
        // dataLines) and either loads the DAC (on write) or drives (writes)
        // the dataLines with the averaged value read from the ADC
        // There are no clock edges up here in the unshown part of this always
        …       //start here
        @ (negedge clock);
        if (~rwLine) begin      // read ADC and average with previous value
                #2 previousADC[addressLines] =
                      (ADC[addressLines] + previousADC[addressLines])>>1;
                dataLines = previousADC[addressLines];
                @(negedge clock);
        end
        else                          // write DAC
                DAC[addressLines] = dataLines;
end
```

a. Draw the waveform for a DAC write. Write “17” into DAC 4.

b. Draw the waveform for an ADC read. Read ADC address 4.

c. The #2 and #5 are hacks that make it work. Remove them but alter the rest of the description to make it work again (use some nonblocking assignments). Use the extra copy of the description if necessary. Show the new **read** waveform.



(a copy)

```
always begin // this models the processor which will write/read the DA/ADC.
      // Registers data,
      // addr, and readWrite are internal to the processor and are the
      // address to be read or written depending on readWrite. data is the
      // value to write or the value read back.
      // There are no clock edges up here in the unshown part of this always
      …         //start here
      #5 rwLine = readWrite;
      if (readWrite) begin    // write to a DAC
               addressLines = addr;
               dataLines = data;
      end
      else begin        // read— ADC value comes back in next clock cycle
               addressLines = addr;
               @(negedge clock);
      end
      @ (negedge clock);
      if (~readWrite)
               data = dataLines; // value returned by ADC
end

always begin
      //This is the DA/ADC that looks at the buslines (rwLine, addressLines,
      // dataLines) and either loads the DAC (on write) or drives (writes)
      // the dataLines with the averaged value read from the ADC
      // There are no clock edges up here in the unshown part of this always
      …         //start here
      @ (negedge clock);
      if (~rwLine) begin      // read ADC and average with previous value
               #2 previousADC[addressLines] =
                     (ADC[addressLines] + previousADC[addressLines])>>1;
               dataLines = previousADC[addressLines];
               @(negedge clock);
      end
      else                         // write DAC
               DAC[addressLines] = dataLines;
end
```