18-360
Intro to Computer-Aided Digital Design
ECE Department

Carnegie Mellon

Out: 1/29/04 Solutions: Homework 1

1. Verilog Hacking

a. Here is one solution. Yours may differ

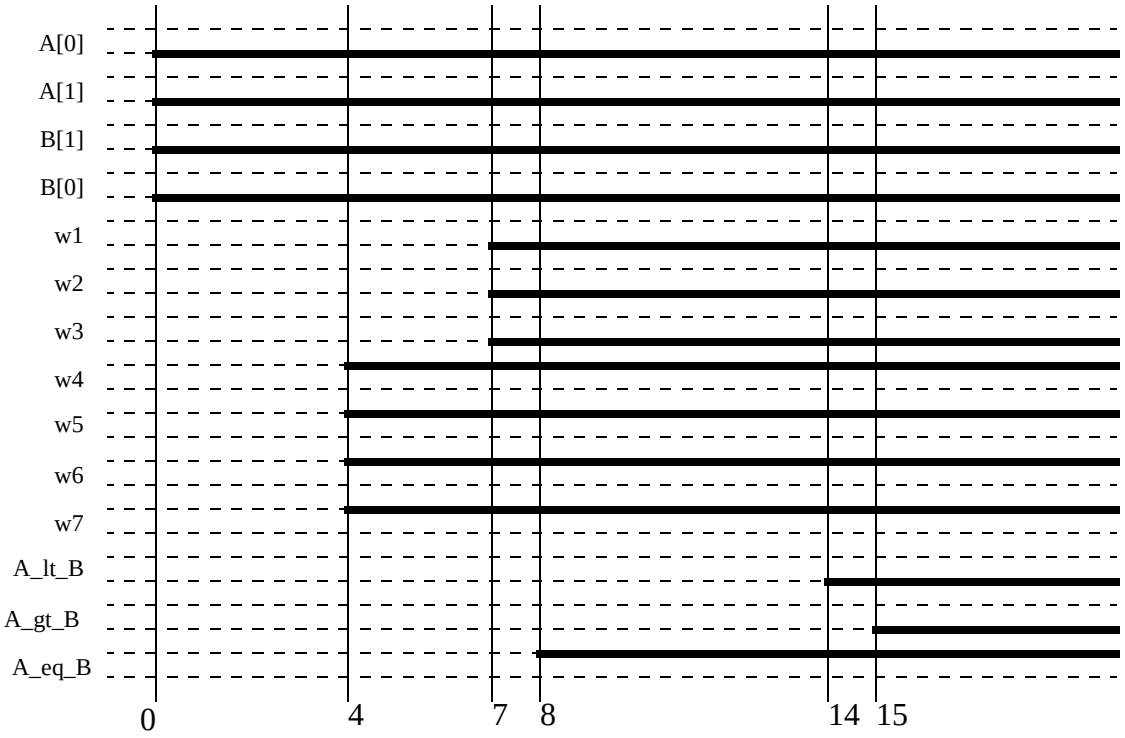
Each of the three implementations are in their own module. They are instantiated in the test module, and their output is compared for each test input. If the outputs are different, the tester prints an error.

```
module compareTheseTwo2bitUnsignedValues_gates(A,B,AgtB,AltB,AeqB);

   input [1:0] A,B;
   output      AgtB, AltB, AeqB;

   not  #(4,7) ( w6, A[1]);
   not  #(4,7) ( w7, A[0]);
   and  #(4,7) ( w1, w6, B[1] );
   and  #(4,7) ( w2, w6, w7, B[0] );
   and  #(4,7) ( w3, w7, B[1], B[0] );
   xnor #(4,7) ( w5, A[1], B[1] );
   xnor #(4,7) ( w4, A[0], B[0] );
   and  #(4,7) ( AeqB, w4, w5 );
   or   #(4,7) ( AltB, w1, w2, w3 );
   nor  #(4,7) ( AgtB, AeqB, AltB );

   // This monitor was used to generate waveforms for the gate wires.
   // initial $monitor($time,, // "w1..7= %b %b %b %b %b %b %b, lt, \
   // eq,gt= %b %b %b", w1, w2, w3, w4, w5, w6, w7,   AltB, AgtB, AeqB );

endmodule // compareTheseTwo2bitUnsignedValues_gates

module compareTheseTwo2bitUnsignedValues_always(A,B,AgtB,AltB,AeqB);

   input [1:0] A,B;
   output      AgtB, AltB, AeqB;
   reg        AgtB,AltB,AeqB;

   always@(A or B)
     begin

       #4;
       AgtB = 0;
       AltB = 0;
       AeqB = 0;

       if ( A==B )
         AeqB = 1;
       else if ( A>B )
         AgtB = 1;
       else
```

```
          AltB = 1;
      end

endmodule // compareTheseTwo2bitUnsignedValues_always


module compareTheseTwo2bitUnsignedValues_assign(A,B,AgtB,AltB,AeqB);

   input [1:0] A,B;
   output      AgtB, AltB, AeqB;

   assign #(4,7) AgtB= (A>B);
   assign #(4,7) AltB= (A<B);
   assign #(4,7) AeqB= (A==B);

endmodule // compareTheseTwo2bitUnsignedValues_assign

/*
 Compare the three input bits. Output 1 iff
 all input bits are the same.
 */
module compare_outputs( out, in );
   input [2:0] in;
   output      out;

   assign      out = ( in === 3'b111 || in === 3'b000 );

endmodule // compare_outputs

module tester;

   reg [1:0] A,B;
   wire     AgtB_gate,   AltB_gate,   AeqB_gate;
   wire     AgtB_assign, AltB_assign, AeqB_assign;
   wire     AgtB_always, AltB_always, AeqB_always;
   wire     AgtB_agree, AeqB_agree, AltB_agree;

   reg [3:0] test_in[15:0];
   integer   i;

   compareTheseTwo2bitUnsignedValues_gates
     secr  (A,B,AgtB_gate,  AltB_gate,  AeqB_gate  );

   compareTheseTwo2bitUnsignedValues_always
     temes (A,B,AgtB_always,AltB_always,AeqB_always);

   compareTheseTwo2bitUnsignedValues_assign
     sage  (A,B,AgtB_assign,AltB_assign,AeqB_assign);

   compare_outputs gt( AgtB_agree, {AgtB_gate, AgtB_assign, AgtB_always} );
   compare_outputs eq( AeqB_agree, {AeqB_gate, AeqB_assign, AeqB_always} );
  compare_outputs lt( AltB_agree, {AltB_gate, AltB_assign, AltB_always} );

   initial
```

```
        begin
          $readmemh("tests.mem", test_in );
          $monitor($time,,"Current Test: A:%b B:%b\n
                           gate:   AgtB:%b AltB:%b AeqB:%b\n
                           assign: AgtB:%b AltB:%b AeqB:%b\n
                           always: AgtB:%b AltB:%b AeqB:%b\n\n",
                           A,B,
                           AgtB_gate,AltB_gate,AeqB_gate,
                           AgtB_assign,AltB_assign,AeqB_assign,
                           AgtB_always,AltB_always,AeqB_always );

          for(i=0; i<16; i=i+1)
           begin
            {A, B} =  test_in[i];
            #1000;

            if ( AgtB_agree !== 1 )
              $display ("ERROR: The modules disagree on the gt output.");

            if ( AeqB_agree !== 1 )
              $display ("ERROR: The modules disagree on the eq output.");

            if ( AltB_agree !== 1 )
              $display ("ERROR: The modules disagree on the lt output.");

            if ( {AgtB_agree, AeqB_agree, AltB_agree} === 3'b111 )
              $display ("TEST PASSED: The modules agree on all outputs.");

           end
        end

   endmodule // tester
```

A=0 and B=0 are used as one test vector. Here are the transitions for A=x, B=x to A=0,B=0. The signal names are those used in the verilog module shown above.

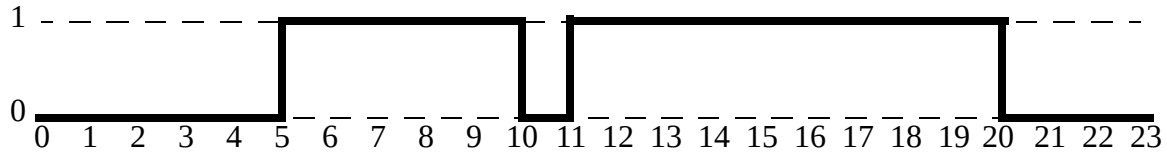


For the above transition, each of the 10 gates changes once, from x to its final value. So in this case, the min, max, and mean number of transitions of the gates are all 1. It is possible to have different values with a different circuit and/or a different test vector.

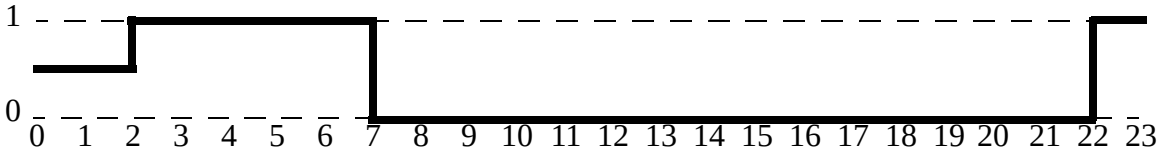
b. If a new comparator is added, the timing of the gate events will not be affected. The timing of the gate events is a function only of the inputs to those gates. In this system, those inputs are A and B. Adding a new module that reads from A and B will not change the values of A and B. Since the inputs to the gates do not change, the physical events associated with the gates will not change.

2. Waveform City

The input signal, b, looks like this:

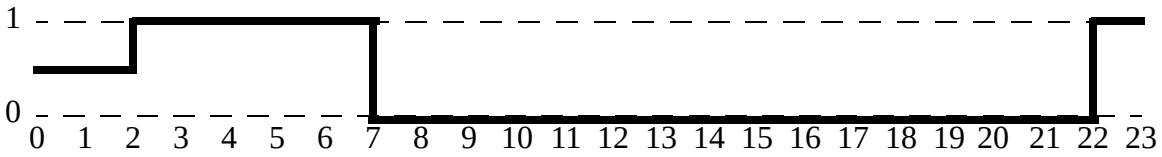


not #2(a,b):



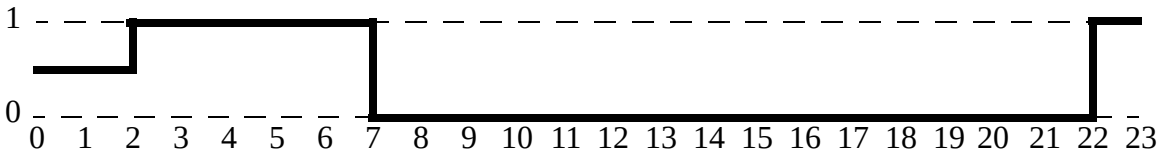
Note that the “spike” did not go through the primitive gate. It was less than the propagation delay.

assign #2 a = ~b



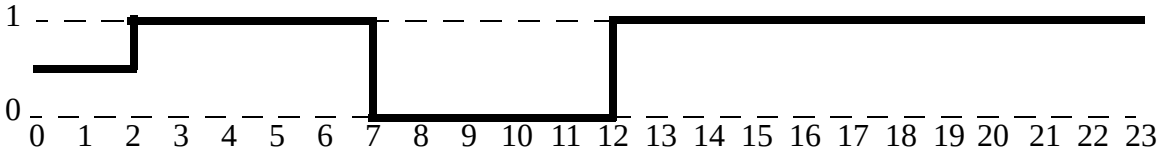
always @(b) #2 a = ~b

The delay of two prevents the change in b at time 10 from propagating through. Note that the delayed blocking assignment from the change at 10 waits till time 12 to read b, so the value of b at time 12 is used to compute the value of a at time 12. The change in b at time 11 does not cause the always block to be run, because the is suspended waiting for the #delay.



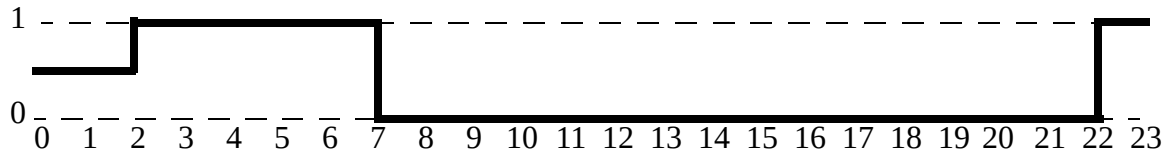
always @(b) a = #2 ~b

Here, when b changes at time 10, its value (1) is stored, causing ~b (0) to be written to a at time 12. Because the assignment is blocking, the change in b at time 11 does not activate the always block, since it is still suspended (blocked) from the change in b at time 10.



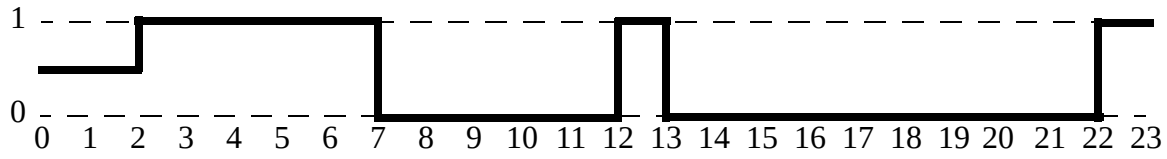
always @(b) #2 a <= ~b

In this case, the non-blocking assignment has no effect, because there are no other assignments in the always block. The behavior is exactly the same as the blocking assignment version.



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

On a change in b, this version reads the current value of b, schedules ~b to be written to a in two time units, and goes back to wait for the next change in b in zero time. So, every change of b causes a change in a, two time units later.



3. There's a price to being an author of a book

Here is a possible fix. Note that non-blocking assignment should be used in order to catch clock edges.

```
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

```
// non-blocking assignment
initial
   begin
     a <=  #0 1'bx;
     a <=  #4 1'b0;
     a <=  #8 1'bz;
     a <= #12 1'b1;
     a <= #16 1'b0;
     a <= #20 1'b1;
     #24 $finish(0);
   end
```

```
// blocking assignment
initial
   begin
     #0 a = 1'bx;
     #4 a = 1'b0;
     #4 a = 1'bz;
     #4 a = 1'b1;
     #4 a = 1'b0;
     #4 a = 1'b1;
     #4 $finish(0);
   end
```

Notice that since the non-blocking assignments take zero time, the simulation time is still 0 until the #24 is run. Both initial blocks call finish at time 24.

The non-blocking version enqueues all of the events that will change a, then dequeues them as the simulation progresses. The blocking version suspends the initial block each time a new value is generated. It's hard to say which will run faster. The tradeoff appears to be whether enqueueing and dequeueing events (non-blocking version) takes more or less time than enqueueing and dequeueing an initial block (blocking version). Both versions must also propagate their output.

5. Assign with a function

Three simple functions are used to do the work of the assign statements.

```
module compareTheseTwo2bitUnsignedValues_assign(A,B,AgtB,AltB,AeqB);

   input [1:0] A,B;
   output      AgtB, AltB, AeqB;

function isAgtB;
       input [1:0] a;
       input [1:0] b;

       begin
          isAgtB = (a>b);
       end

endfunction

function isAeqB;
       input [1:0] a;
       input [1:0] b;

       begin
          isAeqB = (a==b);
       end

endfunction

function isAltB;
       input [1:0] a;
       input [1:0] b;

       begin
          isAltB = (a<b);
       end

endfunction

   assign #(4,7) AgtB= isAgtB( A, B );
   assign #(4,7) AltB= isAltB( A, B );
   assign #(4,7) AeqB= isAeqB( A, B );

endmodule // compareTheseTwo2bitUnsignedValues_assign
```

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

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

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

On the initial negative edge of r the description is indeed edge triggered. However, if r remains low for an extended period over the course of several positive edge c events, the reset does not act edge triggered. Instead the reset is level sensitive since all further activations due to posedge c will encounter the reset conditions (because of its level).

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.

```
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
```

name	Value at the end of time 0	Value at the end of time 1
q	0, as set by the initial block	Set to 0 by the always block
r	1, as set by the always block	Indeterminate, or oscillating, as set by the always block
s	Indeterminate — depends on whether the always or initial block runs first	Set to 0 by the always block
t	x, wont be set until time 1	Set to 0

b. The always will never stop after it first stops for the wait, while t is set to 0 at time 1 and does not change any more. An event (the update of q) will be in the event list for time 11.

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

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

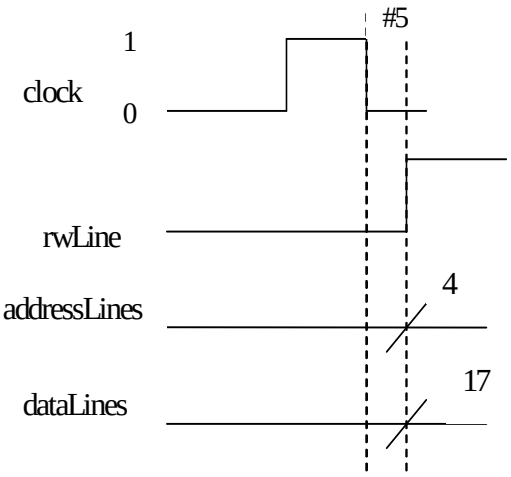
XOR	0	1	x	z
0	0	1	x	x
1	1	0	x	x
x	x	x	x	x
z	x	x	x	x

b. and c. The update and evaluation functions are as follows.

```
init_XOR (gate)
{
    // start of simulation
    gate.counter_one = 0;
    gate.counter_x = gate.num_inputs;
    eval_XOR (gate);
}

// when any single input gate.input[id]
// changes to a different value ''new_value'',
// the update_XOR function is called
update_XOR (gate, id, new_value)
{
    if (new_value == 1)
        gate.counter_one++;
    if (gate.input[id].value == 1)
        gate.counter_one--;
    if (new_value == x)
        gate.counter_x++;
    if (gate.input[id].value == x)
        gate.counter_x--;

    gate.input.value=new_value;

    eval_XOR (gate);
    }
}

eval_XOR (gate)
{
    // if any input is x, the output will be x.
    if (gate.counter_x != 0)
        gate.output = x;
    else {
        // if an odd number of inputs are 1's, the output will be 1;
        // otherwise, the output will be 0.
        gate.output.value = gate.counter_one % 2;
    }
}
```

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

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

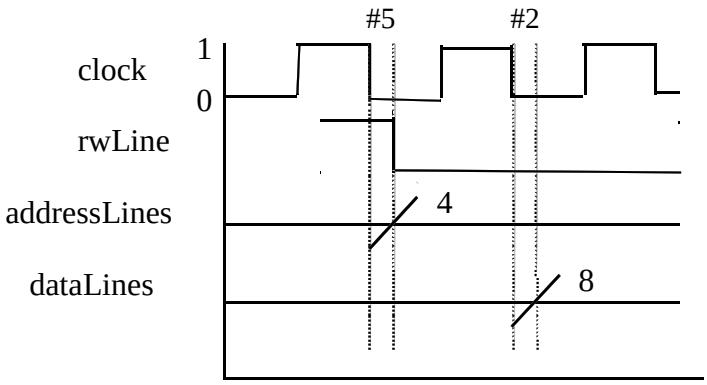


b. Draw the waveform for an ADC read. Read ADC address 4.

The value written will be (17+0) >> 1 = 8.

Processor waits 5 and asserts read and addressLines, then waits for last neg clock edge (far right) for value read.

ADC sees read, waits 2, puts value on dataLines, then waits until far right clock edge.



c. The solution to part c requires you to use non-blocking assignments so that everything happens concurrently on the clock edge. The following is an answer.

```
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
      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
                tempReg = (ADC[addressLines] +
                      previousADC[addressLines])>>1;
                previousADC[addressLines] = tempReg;
                dataLines <= tempReg;
                @(negedge clock);
      end
      else // write DAC
                DAC[addressLines] <= dataLines;
end
```