Name Solution

Digital Design Using Verilog
EE 4702-1
Midterm Examination

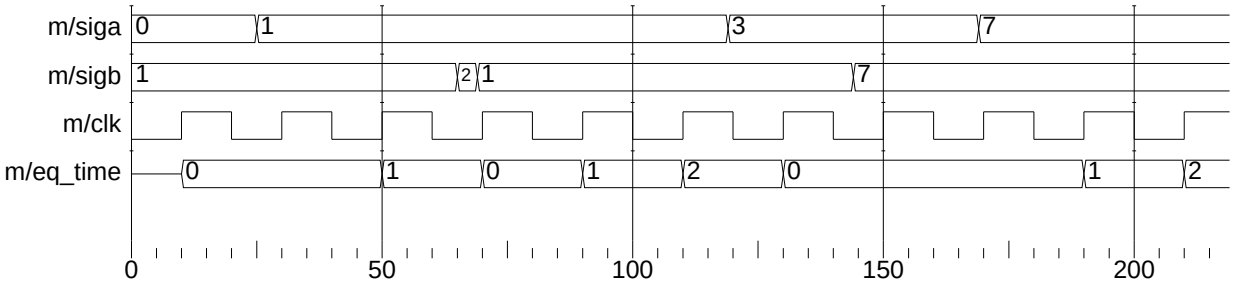
16 March 2001 8:40-9:30 CST

Problem 1		(30 pts)
Problem 2		(25 pts)
Problem 3		(35 pts)
Problem 4		(10 pts)
Exam Total		(100 pts)

Alias always @(posedge)

Good Luck!

Problem 1: Complete the Verilog behavioral description below so that it operates as follows. Compute 32-bit output `eq_time` so that it is the number of consecutive positive edges of input `clk` for which 32-bit inputs `siga` and `sigb` remain equal. The counting should start on the first positive edge of `clk` after `siga` becomes equal to `sigb`; the count starts at zero at the moment they become equal, and while they remain equal the count is incremented at each positive edge. The count should go back to zero at the first positive edge of `clk` after `siga` becomes unequal to `sigb`. The count goes to zero even if `siga` and `sigb` become equal again before the positive edge. Sample output appears in the timing diagram below. (30 pts)



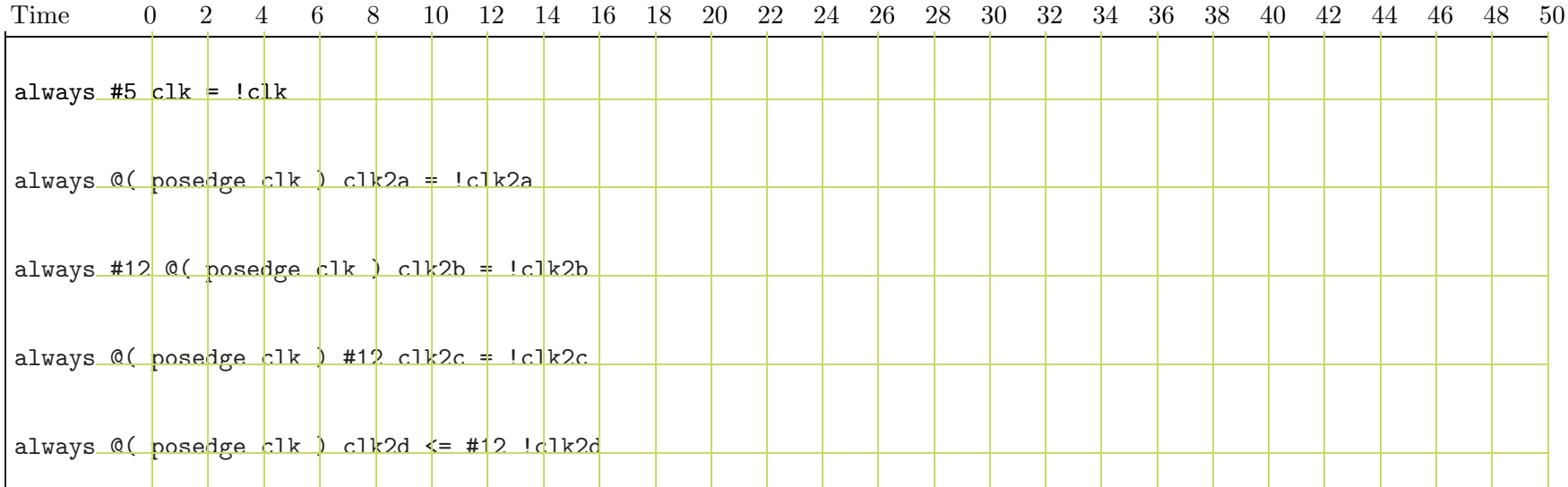

```
module monitor(eq_time, siga, sigb, clk);
   input siga, sigb, clk;
   output eq_time;
   // Don't forget to declare port types.

   // Solution:
   wire [31:0] siga, sigb;
   wire        clk;
   reg [31:0]  eq_time;

   reg [10:0]  next_count;

   always @( siga or sigb ) if ( siga != sigb ) next_count = 0;

   always @( posedge clk )
     begin
        eq_time = next_count;
        if ( siga == sigb ) next_count = next_count + 1;
     end

endmodule
```

Don't get bogged down: There are eight more problems, some can be answered quickly.

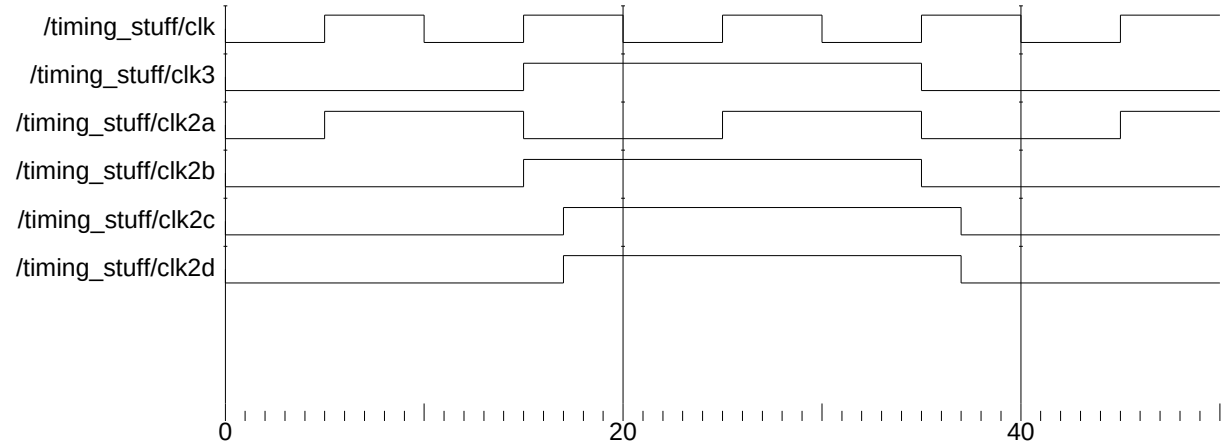
Problem 2: Complete the following timing diagram problems.

(*a*) Complete the timing diagram below. (15 pts)

```
module timing_stuff();
   reg clk, clk3, clk2a, clk2b, clk2c, clk2d,
   initial begin
      clk = 0; clk2a = 0; clk2b = 0; clk2c = 0; clk2d = 0; clk3 = 0;
   end
```

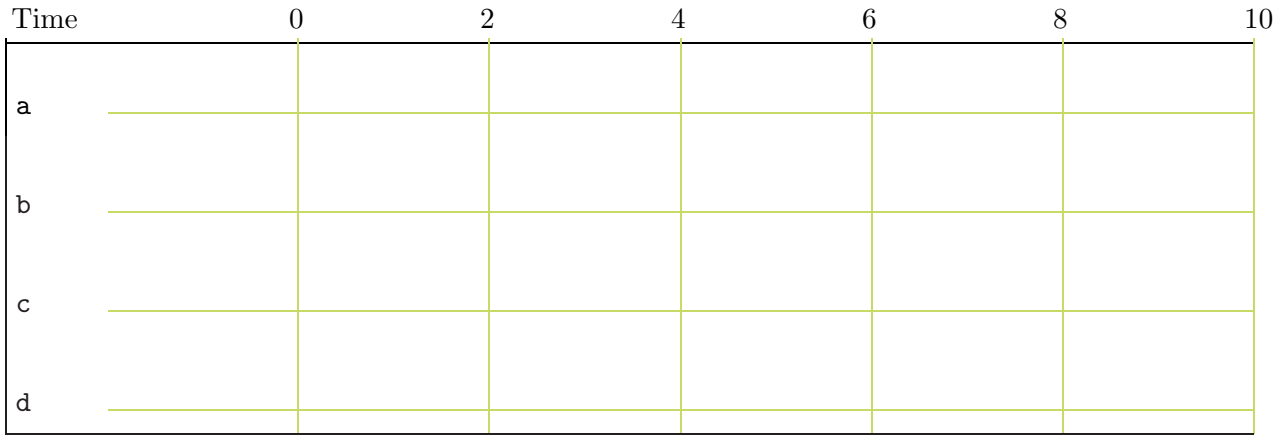


Solution:

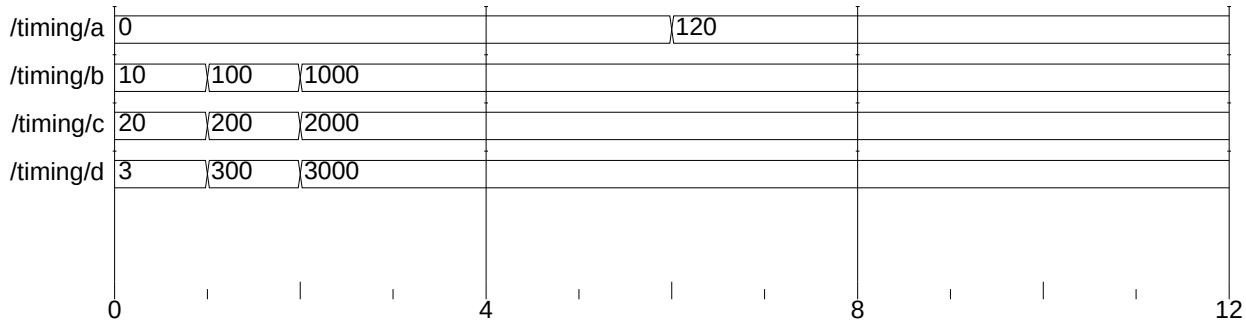

/timing_stuff/clk
/timing_stuff/clk3
/timing_stuff/clk2a
/timing_stuff/clk2b
/timing_stuff/clk2c
/timing_stuff/clk2d
0
20
40

(*b*) Complete the timing diagram below. Be sure to clearly indicate when a signal value changes. (10 pts)

```
module timing();
   integer a, b, c, d;
   initial begin
      a = 0;
      b = 10;
      c = 20;
      d <= #0 3;
      d = 30;
      d <= #1 300;
      d <= #2 3000;
      #1;
      b = 100;
      c <= 200;
      a <= #5 b + c;
      #1;
      b = 1000;
      c <= 2000;
      #10;
   end
endmodule
```



Solution:



Problem 3: Answer each question below. Some can be answered quickly, try answering those questions first.

(*a*) The `match_count_x` modules below are *supposed to* count the number of times input `symbol` is the same as input `targ`. Output `count` should be incremented if `symbol` is the same as `targ` after a change in `symbol`. Most or all of the modules below don't work properly. For each non-working module describe the problem and how it is simulated. *It is important to describe how the incorrect Verilog is simulated and why it is wrong.*

Port declarations and initializations are not shown, but assume they are present and correct. Behavior for unknown and high-impedance values is undefined. In other words, the problems are **not** related to declarations, initialization, or unknown values. (10 pts)

```
module count_match_1(count,symbol,targ); // Declarations and init. not shown.

   always wait ( symbol == targ ) count = count + 1;

endmodule
```

(4 pts) Because an iteration of `always` is done without any delay the simulator "freezes" when `symbol` is equal to `targ` as `count` is continually updated, there is no chance for `targ` or `symbol` to change.

```
module count_match_3(count,symbol,targ); // Declarations and init. not shown.

   always #10 if ( symbol == targ ) count = count + 1;

endmodule
```

(3 pts) Rather than incrementing `count` on each change in `symbol`, the code above increments `count` on ten-cycle intervals when `symbol` is equal to `targ`. It does not increment `count` when `symbol` changes, it might miss times that `symbol` is equal to `targ` (when `symbol` changes several times in the ten-cycle interval) and it will increment `count` multiple times if `symbol` remains equal to `targ` at least 20 cycles.

```
module count_match_4(count,symbol,targ); // Declarations and init. not shown.

   always @( symbol == targ ) count = count + 1;

endmodule
```

(3 pts) Variable `count` is incremented when `symbol` becomes equal to `targ` and when `symbol` becomes unequal to `targ`.

(*b*) Show how each of the three adders below can be used in the module `use_adders` to add seven to input `a`. **Do not** modify the adders themselves. (10 pts)

```
module adder1(x,a,b);
   input a, b;
   output x;
   wire [31:0] a, b;
   wire [31:0] x = a + b;
endmodule

module adder2(x,a);
   input a;
   output x;
   parameter b = 0;
   wire [31:0] a;
   wire [31:0] x = a + b;
endmodule

‘define b 7  // Part of solution.
module adder3(x,a);
   input a;
   output x;
   wire [31:0] a;
   wire [31:0] x = a + ‘b;
endmodule


module use_adders(x_1,x_2,x_3,a);
   input a;
   output x_1, x_2, x_3;  // Each output should be a + 7
   // Use adder1, adder2, and adder3 to generate respective x_ outputs.

   // Solution
   wire [31:0] x_1, x_2, x_3, a;

   adder1 a1(x_1,a,32’d7);

   adder2 #(7) a2(x_1,a);

   adder3 a3(x_1,a);



endmodule
```

(*c*) Show the values that will be assigned in each assignment to `r`. Variables `a`, `c`, and `r` are six-bit registers. (5 pts)

```
a = 6'b101010;
c = 6'bx1x0x1;

r = & a;  // Solution:  r set to 0

r = | a; // Solution:  r set to 1

r = ^ a;  // Solution:  r set to 1

r = & c;  // Solution:  r set to 0

r = | c; // Solution:  r set to 1

r = ^ c;  // Solution:  r set to  x
```

(*d*) Do the two code fragments below do the same thing? If not, how do they differ? (5 pts)

```
// Fragment A.

if ( foo > bar ) x = x + 1; else y = y + 1;


// Fragment B.

case ( foo > bar )
  1: x = x + 1;
  default: y = y + 1;
endcase
```

They do not differ.

(*e*) Why can't the following increment macro be re-written as a function or task in Verilog 95? (5 pts)

```
‘define incr(a) a=a+1
// ...
  // Sample uses of macro.
  for (i=0; i<10; ‘incr(i)) x = x + y;
  for (j=0; j<10; ‘incr(j)) begin foo(j); k = k + x; end
```

In Verilog 95 the third item in the for must be an assignment statement, so a task or function wouldn't work. A function could be used in SystemVerilog.

Problem 4: The module below counts the number of five's and nine's appearing at input `c`. Explain exactly when five's and nine's are counted (start cycle and end cycle), and describe any restrictions on the counts. (10 pts)

```
module yet_another_symbol_counter(fives, nines, c);
   input c;
   output fives, nines;
   wire [7:0] c;
   reg [31:0] fives, nines;

   initial fork

      begin
         fives = 0;
         nines = 0;
      end

      #50 fork:A
         repeat ( 42 ) @( c ) if ( c == 5 ) fives = fives + 1;
         #100 disable A;
      join

      #70 fork:B
         forever @( c ) if ( c == 9 ) nines = nines + 1;
         #200 disable B;
      join

   join

endmodule
```

The module counts fives that appear between 50 and 150 cycles into the simulation. No more than 42 new symbols appearing after cycle 50 are examined for fives. (The maximum number of fives that can be counted is 21.)

The module counts nines that appear between 70 and 270 cycles into the simulation. The number of nines that can be counted is limited only by the size of **nines**, 32 bits.