Solutions to Quiz 2

There were two versions of each question. The values and the answers for both versions are given below.

Question 1

Fill the table below with the value of each expression as a Verilog numeric literal including the correct width and the correct value in hexadecimal base. Assume the following declarations:

```
logic [7:0] y ;
logic [3:0] x ;
```

and that `y` has the value `8'h9c` (or `8'h63`) and that `x` has the value `4'b0110` (or `4'b1001`). The first row has been filled in as an example. You need not show your work or draw another box around the answer

expression	value
`y[3:0]`	
`x + y`	
`x[2:1] ? y : !y`	
`~ y[3:2] & 15`	
`y >> 4 > ~ x`	

Answers

For `y=8'h9c` and `x=4'b0110`:

expression	value
`y[3:0]`	`4'hc`
`x + y`	`8'ha2`
`x[2:1] ? y : !y`	`8'h9c`
`~ y[3:2] & 15`	`32'hc (or 32'h0)`
`y >> 4 > ~ x`	`1'h0`

and for `y=8'h63` and `x=4'b1001`:

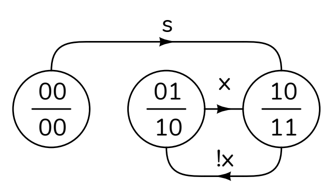
expression	value
`y[3:0]`	`4'h3`
`x + y`	`8'h6c`
`x[2:1] ? y : !y`	`8'h0`
`~ y[3:2] & 15`	`32'hf (or 32'h3)`
`y >> 4 > ~ x`	`1'h0`

Unfortunately, the answer for the expression `~ y[3:2] & 15` gives a surprising result because intermediate "context-dependent" values[1] are extended to the width of the result *before* operators are applied.

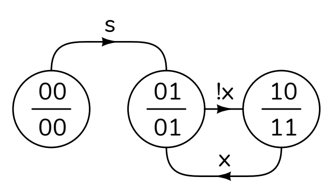
In the expression `~ y[3:2] & 15` the result has 32 bits (the larger of 2 and 32). When the expression `y[3:2]` is evaluated with a width of 32 the result is `32'h3` (or `32'h0`) not `2'h3` (or `2'h0`). The bitwise complement is `32'hffff_fffc` (or `32'hffff_ffff`). When AND'ed with `15` (`32'hf`) the result is `32'hc` (or `32'hf`). This detail was not covered in Lecture 1 and so both answers were marked correct.

Question 2

Write one `always_ff` statement and one `assign` statement in the incomplete System Verilog module below so that the module implements the following state transition diagram:



or:



The diagram follows the course conventions: each state bubble shows the value of **st***(ate)* above the value of **o** in that state; both in binary. Transitions are labelled with expressions involving the inputs. Follow the course coding guidelines but omit comments.

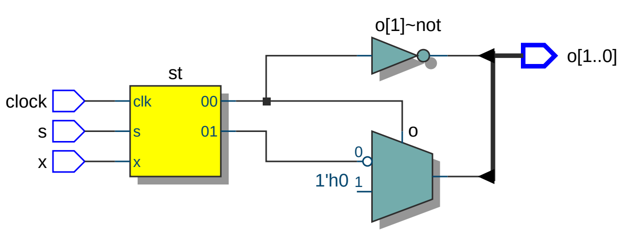
[1] Which operands are "context-dependent" and which are "self-determined" depends on the operators.

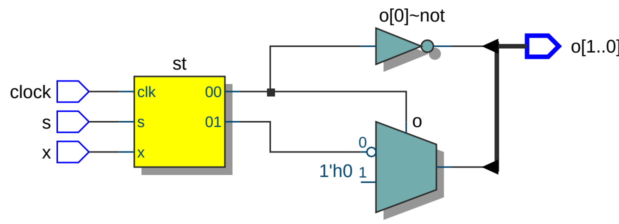
```
module q2
  ( input logic s, x, clock,
    output logic [1:0] o ) ;

   logic [1:0] st ;      // write your code below

endmodule
```

Answers

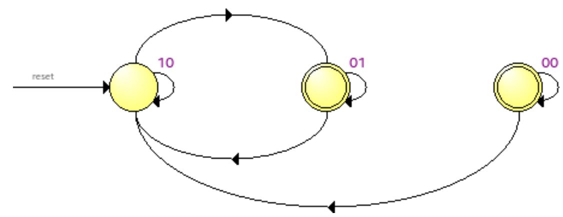
```
module q2_
  ( input logic s, x, clock,
    output logic [1:0] o ) ;

   logic [1:0] st ;              // write your code below

   always_ff @(posedge clock)
     st <= st == 2'b00 &&  s ? 2'b10 :
           st == 2'b01 &&  x ? 2'b10 :
           st == 2'b10 && !x ? 2'b01 :
           st ;

   assign o = st == 2'b00 ? 2'b00 :
              st == 2'b01 ? 2'b10 :
              2'b11 ;

endmodule

// or:

module q2
  ( input logic s, x, clock,
    output logic [1:0] o ) ;

   logic [1:0] st ;              // write your code below

   // three state transitions

   always_ff @(posedge clock)
     st <= st == 2'b00 &&  s ? 2'b01 :
           st == 2'b01 && !x ? 2'b10 :
           st == 2'b10 &&  x ? 2'b01 :
           st ;

   // outputs for each state

   assign o = st == 2'b00 ? 2'b00 :
              st == 2'b01 ? 2'b01 :
              2'b11 ;

endmodule
```

Quartus recognizes that this is a state machine and generates the following schematics:

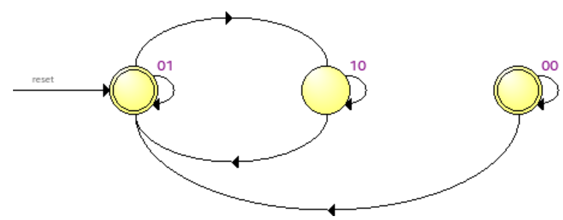




and the following state transition diagrams for the `st` state machines:

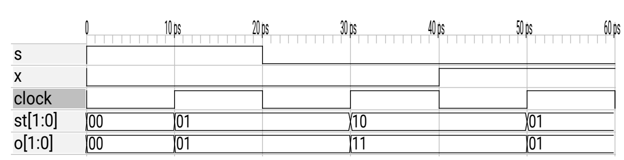


	Source State	Destination State	Condition
1	00	10	(s)
2	00	00	(!s)
3	01	10	(x)
4	01	01	(!x)
5	10	10	(x)
6	10	01	(!x)



	Source State	Destination State	Condition
1	00	01	(s)
2	00	00	(!s)
3	01	10	(!x)
4	01	01	(x)
5	10	10	(!x)
6	10	01	(x)

Simulation waveforms for the first:



and second versions: