2 Verilog [60 points]

2.1 Complete the Verilog code [30 points]

For each numbered blank (1)-(5) in the following Verilog code, **mark the choice below** (i.e., one of options A, B, C, D) that makes the Verilog module operate as described in the comments. The resulting code must have correct syntax.

```
module my_module (input clk, input rst,
  input[15:0] idata, input[1:0] op, (1)[31:0] odata);

  (2) nval = 32'd0; // defining a 32-bit signal with an initial value of 0

  always@* begin
    case (op)
      2'b00:
        nval = odata + idata; // when 'op' is decimal 0, add 'idata' to
                              // 'odata' and assign the result to 'nval'
      2'b01:
        nval = odata - idata; // when 'op' is decimal 1, subtract 'idata'
                              // from 'odata' and assign the result to 'nval'
      2'b10:
        nval = idata; // when 'op' is decimal 2, assign 'idata' to 'nval'
      (3):
        nval = 0; // when 'op' is decimal 3, assign 0 to 'nval'
    endcase
  end

  // executing the following always block on the rising edge of 'clk'
  always@ (posedge clk) begin
    if (rst)
      (4) // resetting 'odata' to 0 for the next cycle
    else
      (5) // assigning 'nval' to 'odata' for the next cycle
  end
endmodule
```

Provide your choice for each blank (1)-(5) below:

(1):	A. output	B. output reg	C. output wire	D. input reg
(2):	A. reg[31:0]	B. input[31:0]	C. wire[31:0]	D. int[31:0]
(3):	A. 2'b3	B. 3'b3	C. 2'h11	D. default
(4):	A. assign odata <= 0;	B. assign odata = 0;	C. odata == 0;	D. odata <= 0;
(5):	A. assign odata <= nval;	B. assign odata = nval;	C. odata == nval;	D. odata <= nval;

Explanation.
(1): `odata` must be declared as an `output` signal since values are assigned to it in the second `always` block. It cannot be an `input` signal since inputs are read-only signals and no assignments are allowed to them. `odata` must be also declared as `reg` since the assignments are made inside an always block.
(2): `nval` must be declared as `reg[31:0]` since values are assigned to it inside the first `always` block.
(3): `default` is a correct choice since all other cases for a 2-bit values (i.e., `2'b00`, `2'b01`, and `2'b10`) are defined in the `case` statement. The other choices are not correct since they do not properly specify the value of 3. For example, in `2'b3`, the problem is that `3` is not a valid binary digit but `2'b` must be followed by a 2-bit binary value.
(4): Choices with `assign` are not valid since the `assign` keyword cannot be used in an `always` block. Choice C does not specify an assignment operator but an equality comparison, hence it is not a valid choice either. The correct choice is D, which assigns 0 to `odata` using non-blocking assignment operator.
(5): The correct choice is D due to the same reasons as in (4).

2.2 What Does This Code Do? [30 points]

You are given a Verilog code that you are asked to analyze and find out what it does.

```
module my_module2 (input clk, output[1:0] out);

  reg state = 1'b0;
  reg[1:0] my_reg = 0;

  always@(posedge clk) begin
    state <= &out ? ~state : state;
  end

  always@(posedge clk) begin
    case(state)
      1'b0: begin
        my_reg <= my_reg + 1;
      end
      1'b1: begin
      my_reg <= my_reg - 1;
      end
    endcase
  end

  assign out = my_reg;
endmodule
```

Show the values (as unsigned decimal numbers) that the `out` signal takes, starting from the initial state of the module, for 16 consecutive clock (i.e., `clk`) cycles. Explain your answer briefly.

`out` is equal to $0, 1, 2, 3, 0, 3, 2, 3, 0, 3, 2, 3, 0, 3, 2, 3$ in the first 16 clock cycles.

Explanation.
The module either increments or decrements `my_reg` depending on the `state`. When `state` is equal to 0, `my_reg` is incremented by 1 and otherwise decremented by 1. The value of `my_reg` is directly assigned to the `out` signal, and both signals are 2-bit wide.
`my_reg` and `state` are both initially 0. Therefore, in subsequent cycles, `my_reg` gets incremented until it reaches 3. During the next cycle, a new value for `state` is being computed (i.e., the inverse of `state` as $\sim$`state`). However, since the new value of the `state` is not updated until the next positive edge of the `clk`, the second always block reads `state` as 0, and thus `my_reg` gets incremented again to become 0 (the maximum value a 2-bit register can represent is 3 and incrementing `my_reg` one more time makes it 0).
During the next cycle, `state` is 1 and `my_reg` is decremented back to 3. Since `my_reg` (and thus `out`) being 3 inverts `state`, `state` becomes 0 in the subsequent cycle and `my_reg` becomes 2 during the positive edge of `clk` when `state` is inverted. Then, `my_reg` gets incremented to 3 and 0 in the next consecutive cycles. Because `state` remains as 0 or 1 for two consecutive cycles and then gets inverted, the values of `my_reg` forever repeat the sequence of $(0, 3, 2, 3, 0, 3, 2, 3, \ldots)$.