

4 Verilog [60 points]

4.1 What Does This Code Do? [30 points]

Analyze the following Verilog module and answer the question.

```

1  module mystery_module (clk, en, in1, in2, out);
2
3      input clk, en;
4      input [63:0] in1;
5      input [7:0] in2;
6      output reg [10:0] out = 0;
7
8      reg [2:0] var1 = 0;
9
10     always @(posedge clk) begin
11         out <= out;
12         if (en & (var1 == 0)) begin
13             var1 <= var1 + 1'b1;
14
15             if (in2[var1])
16                 out <= 11'd0 + in1[var1*8 +: 8];
17             else
18                 out <= 11'd0 - in1[var1*8 +: 8];
19         end
20
21         if (var1 != 0) begin
22             var1 <= var1 + 1'b1;
23
24             if (in2[var1])
25                 out <= out + in1[var1*8 +: 8];
26             else
27                 out <= out - in1[var1*8 +: 8];
28         end
29     end
30
31 endmodule

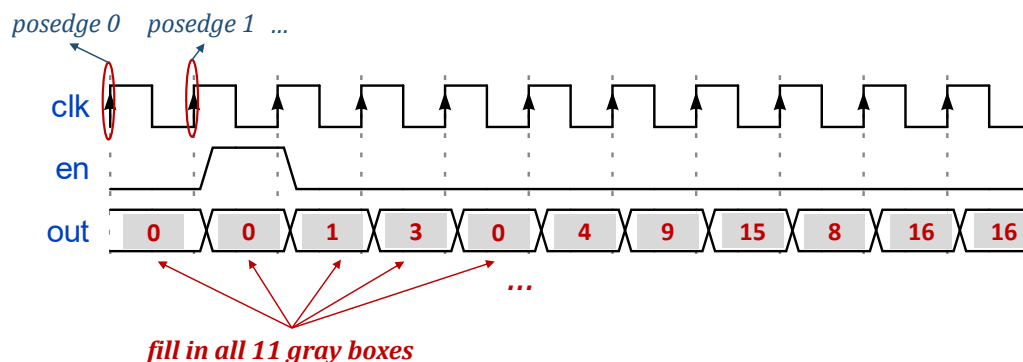
```

Assume that the inputs *in1* and *in2* *always* have the following values:

in1 = 64'h0807060504030201

in2 = 8'b10111011

What **unsigned decimal** values does the *out* signal get in the following waveform diagram? Fill in the gray boxes with an *out* value for each *clk* cycle. Briefly explain your answer.



Brief explanation (to help us award you partial credit):

Explanation. Once `en` becomes 1, the `mystery_module` begins processing the inputs `in1` and `in2`.

The output signal `out` is initially 0 (line 6).

`var1` is used to index both `in1` and `in2`. `in1` is indexed in 8-bit data chunks and every bit in `in2` is indexed separately. `var1` is initially 0 and indexes both inputs starting from their least-significant bits.

When `var1` is 0, the `mystery_module` adds (if `in2[0] = 1`) or subtracts (if `in2[0] = 0`) the least significant 8 bits of `in1` (i.e., `in1[7:0]`) to/from the 11-bit `out` register. In consecutive cycles, `var1` gets incremented by 1 and the module adds or subtracts the other 8-bit data chunks in `in1` to/from `out`.

For the given values of `in1` and `in2`, `out` gets the following values:

cycle 0: `out` = 0 (`en` = 0 so `out` remains 0)

cycle 1: `out` = 0 (`en` = 1 but `out` will be updated for the next cycle (after `posedge 2`))

cycle 2: `out` = 0 + 1 = 1

cycle 3: `out` = 1 + 2 = 3

cycle 4: `out` = 3 - 3 = 0

cycle 5: `out` = 0 + 4 = 4

cycle 6: `out` = 4 + 5 = 9

cycle 7: `out` = 9 + 6 = 15

cycle 8: `out` = 15 - 7 = 8

cycle 9: `out` = 8 + 8 = 16

cycle 10: `out` = 16 (all inputs have been processed. `var1` becomes 0 and `out` remains as is for future cycles unless `en` becomes 1)

4.2 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.

```

1  module my_module (input clk, input rst,
2      input[1:0] data, ① result);
3
4      ② state = 2'b00; // defining a 2-bit signal with an initial value of 0
5
6      always @(posedge clk) begin
7          case (state)
8              2'b00:
9                  state <= state + ③; // set the next 'state' to 2'b11
10             2'b01:
11                 state <= 2'b00;
12             2'b10: begin
13                 state <= 2'b11;
14
15                 if (④ data) // set the next 'state' to 2'b01 if
16                     state <= 2'b01; // all bits of 'data' are 1
17             end
18             2'b11:
19                 state <= 2'b10;
20         endcase
21     end
22
23
24     assign result = ⑤ state; // assign 1'b1 to 'result' if 'state' has any bit set to 1
25                             // otherwise assign 1'b0
26 endmodule

```

Provide your choice for each blank ①-⑤ below. Circle only one of A, B, C, D for each blank.

- ①: A. output B. output reg C. output reg[0:0] D. input reg
- ②: A. reg[1:0] B. reg C. wire D. wire[1:0]
- ③: A. 1'b3 B. 3'b2 C. 2'd11 D. 3
- ④: A. || B. & C. ! D. 1
- ⑤: A. | B. & C. && D. ^

Explanation.

①: `result` must be specified as a single bit 'output' signal because it gets assigned either `1'b0` or `1'b1` via an 'assign' operator. It cannot be specified as a 'output reg' because the 'assign' operator can be used only with 'wire' signals. Note: 'output' is the same as 'output wire'.

②: `state` is a two-bit signal (as we can tell from lines 8, 10, 12) and must be a 'reg' because it gets assigned a value in an 'always' block.

③: In order to transition to `state = 3` from `state = 0`, we need to add 3. Note that A. `1'b3` is not a valid syntax as 3 is not a binary number. Not in the choices, but `1'd3` would also be incorrect since 3 cannot be encoded with a single bit. C. `2'd11` has a similar problem as decimal 11 cannot be encoded with 2 bits.

④: In the given choices, only the AND-reduction (`&`) operator provides the expected functionality of resulting in 1 when all bits of `data` are 1.

⑤: In the given choices, only the OR-reduction (`|`) operator provides the expected functionality of resulting in 1 when `state` contains at least one 1.