Problem 1 (10 points)

Write a Verilog **structural** model using **primitive gates** for a five-input majority circuit. The output majority should equal 1 if a majority of the five components of vote equal 1. The output should equal 0 otherwise. Use the module declaration below as a starting point for your model. You may add wires as needed.

// Filename: problem1.v

// Author:

// Date:

// Version:

// Description:

module problem1\_YOURPID(vote, majority);

input [4:0] vote;

output majority;

// INSERT YOUR CODE HERE

endmodule

*This is important!*

Use **only Verilog’s built-in gate primitives** to represent the operators that make up your logic circuit. Imagine that gates come at a premium in this problem; your model should use no more than a total of ten logic gates. You may use logic gates of any type, and you may use logic gates having any number of inputs. *Perform a multi-level transformation to reduce the gate count of the circuit*. *Comment on how your implementation compares to a two-level circuit that implements the same function*.

|  |  |
| --- | --- |
| 00000 | 0 |
| 00001 | 0 |
| 00010 | 0 |
| 00011 | 0 |
| 00100 | 0 |
| 00101 | 0 |
| 00110 | 0 |
| 00111 | 1 |
| 01000 | 0 |
| 01001 | 0 |
| 01010 | 0 |
| 01011 | 1 |
| 01100 | 0 |
| 01101 | 1 |
| 01110 | 1 |
| 01111 | 1 |
| 10000 | 0 |
| 10001 | 0 |
| 10010 | 0 |
| 10011 | 1 |
| 10100 | 0 |
| 10101 | 1 |
| 10110 | 1 |
| 10111 | 1 |
| 11000 | 0 |
| 11001 | 1 |
| 11010 | 1 |
| 11011 | 1 |
| 11100 | 1 |
| 11101 | 1 |
| 11110 | 1 |
| 11111 | 1 |

CDE + BDE + BCE + ADE + ACE + ACD + ABE + ABD + ABC

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| AB/CDE | 000 | 001 | 011 | 010 | 100 | 101 | 111 | 110 |
| 00 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 01 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 |
| 11 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 10 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 |

F = AB + ABC + CDE + BCE + BCD + ACE + ACD + BC’DE + AC’DE

and(Y, A, B);

or(Y, A, B);

**Helpful Hints for Writing Test Benches as Procedures**

The Verilog code shown below represents one way to write a **test bench**. *Here are some very important teaching points:*

* The circuit represented by problem1 is the one that we want to test.
* In a test bench, an input port of the design under test is generally typed as a reg, since the inputs are the values targeted by the procedure used to generate the input cases. If a value is the target of a procedure, *it must be typed as a reg.*
* In a test bench, an output port of the design under test is generally typed as a wire.
* Note that the WHILE-loop is contained in a procedural block – specifically, an initial block. Structures such as IF-statements, CASE statements, FOR-loops and WHILE-loops *must be contained in procedures* such as initial blocks and always blocks.
* Procedures represented by initial blocks are not generally synthesizable, but since a test bench is not meant to be synthesizable, the initial block provides us with the means to sequence an input test set starting at simulation time t = 0.
* #20 represents a time delay of 20 time steps (usually nanoseconds). The presence of delays prevents the procedure from executing in zero-time. Delays are not synthesizable, but are appropriate for use in test benches for simulation purposes.

// Time Unit = 1 ns (#1 means 1 ns)

// Simulation Precision = 1 ns

`timescale 1ns/1ns

// Filename: tb\_problem1.v

// Author: Jason Thweatt

// Date: 09 September 2015

// Version: 1

// Description: This file contains a test bench for a 5 input circuit.

// It uses a WHILE loop in an initial block to apply all

// combinations of the inputs for a period of 20 ns each.

module tb\_problem1\_while();

reg [4:0] ins;

wire outs;

reg [4:0] count;

problem1 dut(ins, outs);

initial begin

count = 0;

while(count < 32) begin

ins = count;

#20;

count = count + 1;

end

end

endmodule

Problem 2 (5 points)

Explain why the test bench described above will execute forever. Write a working alternative that uses a WHILE-loop.

The largest number counter can hold is 31 as it is a 5-bit binary number with range 0-31. There for counter will always be less than 32, thus the while loop will loop forever.

(If you aren’t wondering why count was declared separately from ins and then assigned to ins in the procedure, then maybe you should be. The solution is not as simple as that, but it is a factor that should enter into your thinking.)

// Simulation Precision = 1 ns

`timescale 1ns/1ns

// Filename: tb\_problem1.v

// Author: Jason Thweatt

// Date: 09 September 2015

// Version: 1

// Description: This file contains a test bench for a 5 input circuit.

// It uses a WHILE loop in an initial block to apply all

// combinations of the inputs for a period of 20 ns each.

module tb\_problem1\_while();

reg [4:0] ins;

wire outs;

reg [5:0] count;

problem1 dut(ins, outs);

initial begin

count = 0;

while(count < 32) begin

ins = count;

#20;

count = count + 1;

end

end

endmodule

Problem 3 (5 points)

The WHILE-loop structure of the test bench in Problem 2 is essentially a dressed-up version of a FOR-loop. Bearing in mind the deficiency contained in the existing code, rewrite the test bench of Problem 2 as a working FOR-loop.

// Insert your header here, using Problem 1 as an example.

module tb\_problem1\_for();

// Simulation Precision = 1 ns

`timescale 1ns/1ns

// Filename: tb\_problem1.v

// Author: Jason Thweatt

// Date: 09 September 2015

// Version: 1

// Description: This file contains a test bench for a 5 input circuit.

// It uses a WHILE loop in an initial block to apply all

// combinations of the inputs for a period of 20 ns each.

module tb\_problem1\_while();

reg [4:0] ins;

wire outs;

reg [5:0] count;

problem1 dut(ins, outs);

initial begin

count = 0;

for(count = 0; count < 32; count = count + 1) begin

ins = count;

#20;

end

end

endmodule

Problem 4 (10 points)

Write a Verilog **behavioral** model for the system of Problem 1. Your model may use continuous assignment and dataflow operators, or a procedural model using an always block. However, you should make your model as simple as possible. *You don’t need to follow the gate restriction of Problem 1, but you should also take advantage of the functionality of dataflow operators to implement your model differently.* Write your model for efficiency and synthesizability.

Use the module declaration below as a starting point for your model. You may add wires or regs as needed. You may add a reg declaration to an entity to make it into a variable separately from declaring the entity as an output. Remember that any value targeted by a procedure must be typed as a reg. Remember that any value targeted by a procedure must be typed as a reg.

// Insert your header here, using Problem 1 as an example.

module problem4\_YOURPID(vote, majority);

input [4:0] vote;

output majority;

// INSERT YOUR CODE HERE

Endmodule

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| AB/CDE | 000 | 001 | 011 | 010 | 100 | 101 | 111 | 110 |
| 00 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 01 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 |
| 11 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 10 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 |

majority =

(vote[2]&vote[3]&vote[4]) |

(vote[1]&vote[3]&vote[4]) |

(vote[1]&vote[2]&vote[4]) |

(vote[0]&vote[3]&vote[4]) |

(vote[0]&vote[2]&vote[4]) |

(vote[0]&vote[2]&vote[3]) |

(vote[0]&vote[1]&vote[4]) |

(vote[0]&vote[1]&vote[3]) |

(vote[0]&vote[1]&vote[2]);

F = CDE + BDE + BCE + ADE + ACE + ACD + ABE + ABD + ABC

Problem 5 (10 points)

Use ModelSim to simulate the operation of the circuits you modeled in Problems 1 and 4. You may use your corrected test bench from Problem 2, or the test-bench you wrote in Problem 3.

Follow the instructions provided for creating a project in ModelSim and performing the simulation.

*This is important!*

Creating an efficient strategy for simulation is important. For this problem, you should be able to simulate both circuits in the same test bench, as long as you do the following:

* Make sure to place an instance of **each** module you want to simulate into the test bench.
* The two module instances use the same “input probes” but each one has to have its own “output probe.” You will need to change the wire declaration of your test bench accordingly.

Problem 1

A screenshot of a computer

Description automatically generated with medium confidence



Problem 4

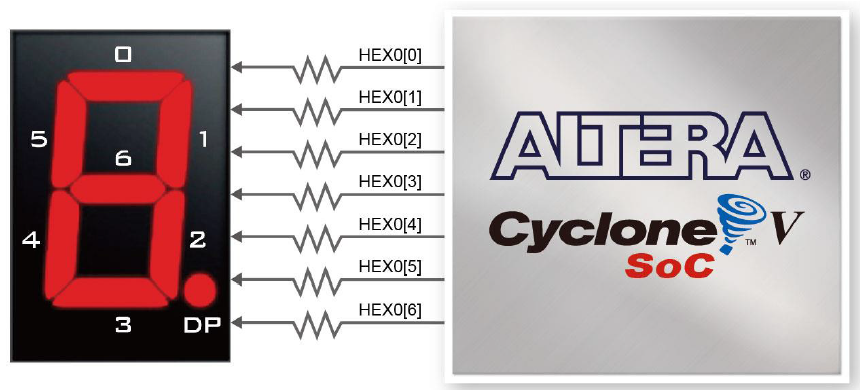
A screenshot of a computer

Description automatically generated with medium confidence

**Helpful Hints for Writing Seven-Segment Display Drivers**

A seven-segment display is often used to display characters. For the seven-segment displays on your DE1-SoC board, *each of the segment outputs is ON when logic-0 is applied to it and is OFF when logic-1 is applied to it.*

The diagram below shows the interface between our FPGA and one of the seven segment displays that the board is capable of driving. The segments are labeled 0 to 6. (Ignore the decimal point DP.) The labels indicate the position that each driver value has in a seven-bit vector.



If we wanted the display to show characters that correspond to the hexadecimal digits, they might appear as follows:

Problem 6 (15 points)

Write a Verilog model for a circuit that accepts a four-bit input representing an unsigned binary value and produces an output that drives the seven-segment display to show the corresponding hexadecimal digit. In this problem, use **continuous assignments with dataflow operators**. Make sure that the order of the signals in the vector hex\_display corresponds to the order of the driver values shown in the diagram above.

Use the following module declaration. You may add wires as needed.

// Insert your header here, using Problem 1 as an example.

module sevensegdecoder\_cont\_YOURPID(hex\_digit, hex\_display);

input [3:0] hex\_digit;

output [6:0] hex\_display;

// INSERT YOUR CODE HERE

endmodule

0,1,2,3,4,5,6,7,8,9, A,B,C,D,E,F

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| A | B | C | D |  | HEX0[0] | HEX0[1] | HEX0[2] | HEX0[3] | HEX0[4] | HEX0[5] | HEX0[6] |
| 0 | 0 | 0 | 0 |  | 1 | 1 | 1 | 1 | 1 | 1 | 0 |
| 0 | 0 | 0 | 1 |  | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |  | 1 | 1 | 0 | 1 | 1 | 0 | 1 |
| 0 | 0 | 1 | 1 |  | 1 | 1 | 1 | 1 | 0 | 0 | 1 |
| 0 | 1 | 0 | 0 |  | 1 | 1 | 1 | 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 |  | 1 | 0 | 1 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 |  | 1 | 0 | 1 | 1 | 1 | 1 | 1 |
| 0 | 1 | 1 | 1 |  | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 |  | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 |  | 1 | 1 | 1 | 1 | 0 | 1 | 1 |
| 1 | 0 | 1 | 0 |  | 1 | 1 | 1 | 0 | 1 | 1 | 1 |
| 1 | 0 | 1 | 1 |  | 0 | 0 | 1 | 1 | 1 | 1 | 1 |
| 1 | 1 | 0 | 0 |  | 1 | 0 | 0 | 1 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 |  | 0 | 1 | 1 | 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 0 |  | 1 | 0 | 0 | 1 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 |  | 1 | 0 | 0 | 0 | 1 | 1 | 1 |

HEX0[0] = C’D’ + CD’ + A’B + A’C + BC + AB’C’

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| AB/CD | 00 | 01 | 11 | 10 |
| 00 | 1 | 0 | 1 | 1 |
| 01 | 1 | 1 | 1 | 1 |
| 11 | 1 | 0 | 1 | 1 |
| 10 | 1 | 1 | 0 | 1 |

HEX0[1] = A’B’ + AC’D + B’C’D + B’C’D’ + A’C’D’ + A’CD + B’CD’

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| AB/CD | 00 | 01 | 11 | 10 |
| 00 | 1 | 1 | 1 | 1 |
| 01 | 1 | 0 | 1 | 0 |
| 11 | 0 | 1 | 0 | 0 |
| 10 | 1 | 1 | 0 | 1 |

HEX0[2] = A’B + AB’ + C’D + A’C’ + A’D

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| AB/CD | 00 | 01 | 11 | 10 |
| 00 | 1 | 1 | 1 | 0 |
| 01 | 1 | 1 | 1 | 1 |
| 11 | 0 | 1 | 0 | 0 |
| 10 | 1 | 1 | 1 | 1 |

HEX0[3] = AC’ + BC’D + A’B’D’ + A’B’C + A’CD’ + BCD’ + ABD’

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| AB/CD | 00 | 01 | 11 | 10 |
| 00 | 1 | 0 | 1 | 1 |
| 01 | 0 | 1 | 0 | 1 |
| 11 | 1 | 1 | 0 | 1 |
| 10 | 1 | 1 | 1 | 0 |

HEX0[4] = AB + CD’ + AC + B’C’D’ + AC’D’

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| AB/CD | 00 | 01 | 11 | 10 |
| 00 | 1 | 0 | 0 | 1 |
| 01 | 0 | 0 | 0 | 1 |
| 11 | 1 | 1 | 1 | 1 |
| 10 | 1 | 0 | 1 | 1 |

HEX0[5] = C’D’ + AB’ + AC + A’BC’ + BCD’

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| AB/CD | 00 | 01 | 11 | 10 |
| 00 | 1 | 0 | 0 | 0 |
| 01 | 1 | 1 | 0 | 1 |
| 11 | 1 | 0 | 1 | 1 |
| 10 | 1 | 1 | 1 | 1 |

HEX0[6] = CD’ + AC + AD + BC’ + B’C

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| AB/CD | 00 | 01 | 11 | 10 |
| 00 | 0 | 0 | 1 | 1 |
| 01 | 1 | 1 | 0 | 1 |
| 11 | 1 | 1 | 1 | 1 |
| 10 | 0 | 1 | 1 | 1 |

Problem 7 (15 points)

Repeat Problem 6 using a **procedural model**. Employ an **always** block containing **if-**statements or a **case** statement. Make sure that the order of the signals in the vector hex\_display corresponds to the order of the driver values shown in the diagram.

Use the following module declaration. You may add wires as needed. You may assign the reg type to a net to make it into a variable separately from declaring the net as an output. Remember that any value targeted by a procedure must be typed as a reg.

// Insert your header here, using Problem 1 as an example.

module sevensegdecoder\_proc\_YOURPID(hex\_digit, hex\_display);

input [3:0] hex\_digit;

output [6:0] hex\_display;

// INSERT YOUR CODE HERE

endmodule

Problem 8 (10 points)

Use ModelSim to simulate the operation of the circuits you modeled Problems 6 and 7. Remember the guidance provided in Problem 5.