CPE 322 **EXAM II** **Name _____**

Spring Semester 2021

Work should be performed systematically and neatly with the final answer being underlined. This exam is open book, open notes, closed neighbor/device/browser. Allowable items on desk include: exam, pencils, and pens. All other items must be removed from student's desk. Students have Approximately 90 minutes (1 1/2 hours) to complete this exam. Best wishes!

1. [20 points] Use Shannon's expansion theorem around a and b for the following function

   Y(a,b,c,d,e,f) = ((a ^ b) & c & ~d & f) + (~a & ~b & ~c & d & e & f) +
   (a & b & (c | e | ~f))

   so that it can be implemented using only four-variable function generators (4-input LUTs). Draw a block diagram to indicate how Y can be implemented using only four-variable function generators. Indicate the function realized by each four-variable function generator.

   Note the operators in the above are written in Verilog syntax; ~ inversion, & bitwise-AND, | bitwise-OR, ^ bitwise-XOR

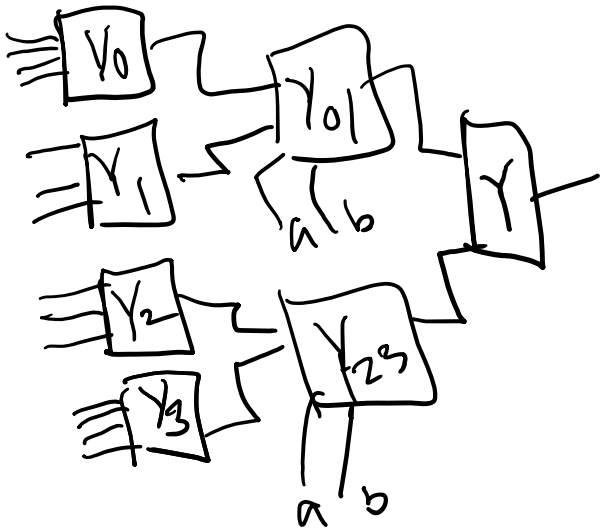
Let Y = Y0 a' b' + Y1 a' b + Y2 a b' + Y3 a b, where Y0 = Y(a=0, b=0), Y1 = Y(a=0, b=1), Y2 = Y(a=1, b=0), and Y3 = Y(a=1, b=1)

Y0(c,d,e,f) = ~c & d & e & f Y1(c,d,e,f) = c & ~d & f

Y2(c,d,e,f) = c & ~d & f Y3(c,d,e,f) = c | e | ~f

+5 an answer
+5 a drawing
+1 per partial product Y_x
+1 per correct function mapped to diagram

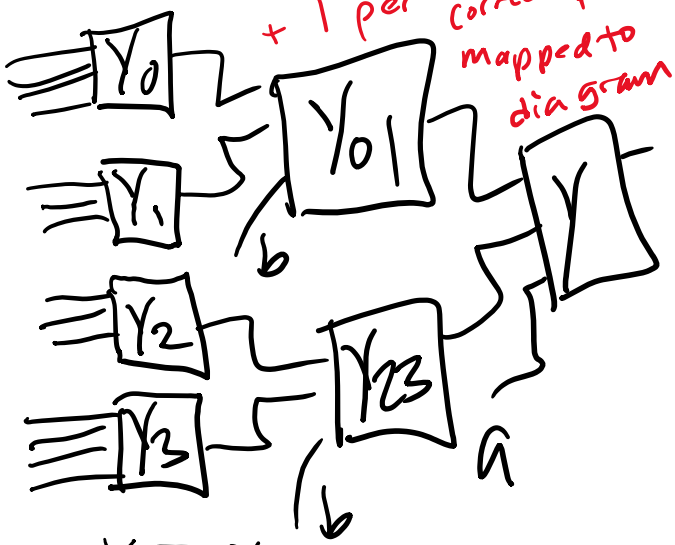
Three or more possibilities exist, since Y1 = Y2. Here are those options:



$Y_{01} = Y_0 \bar{a}\bar{b} + Y_1 \bar{a} b$

$Y_{23} = Y_2 a\bar{b} + Y_3 a b$

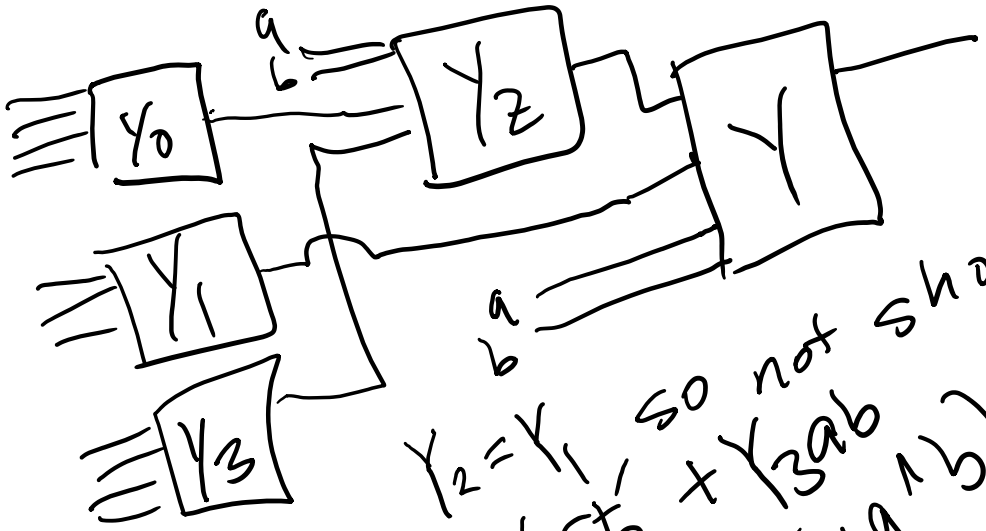
$Y = Y_{01} + Y_{23}$



$Y_{01} = Y_0 \bar{b} + Y_1 b$

$Y_{23} = Y_2 \bar{b} + Y_3 b$

$Y = Y_{01} \bar{a} + Y_{23} a$



$Y_2 = Y_1$, so not shown

$Y_2 = Y_0 \bar{a} \bar{b} + Y_3 a b$

$Y = Y_2 (a \wedge b) + Y_1 (a \wedge b)$

2. [20 points] Number Conversions
   a. Convert the following signed decimal number to a 16-bit 2's-complement signed binary format:

-2021

0xF81B

or

16'b 1111 1000 0001 1011

   b. Convert the following decimal number to a 16-bit binary coded decimal value:

322

0000 0011 0010 0010

or

0x0322

   c. Convert the following decimal number into a 32-bit unsigned hexadecimal value:

12,648,430

0x00C0FFEE

5 pts each:
+2 pts: an answer
+1 reasonably close
+2 perfect match

d. Convert the following decimal number into 32-bit short precision floating point, using 1 bit for sign, 8 bits for the exponent with a bias of 127, and with 23 bits for the mantissa:

$$\text{value} = (-1)^{\text{sign}} \times 2^{(E-127)} \times \left(1 + \sum_{i=1}^{23} b_{23-i} 2^{-i}\right)$$ expressed as the contatenated value {sign, E, b}

-29.609375

= -1 * 2^(131-127) * (1 + 7,135,232/2^23) = 0xC1ECE000

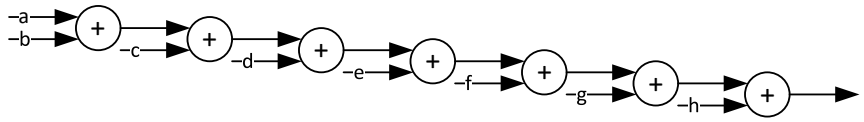
+2: an answer
+1: reasonably close
+2: perfect

3. [20 points] As discussed in class, the combination of linear logical functions can be performed either sequentially or half-at-a-time. Either method can get to the same solution, but often times the method that performs half of the operations at a time is the best choice because it reduces the number of levels of logic and the number of routing paths.

   Take the sum operation:

   sum[6:0] = a[3:0] + b[3:0] + c[3:0] + d[3:0] + e[3:0] + f[3:0] + g[3:0] + h[3:0]

   Write Verilog code that will generate a sequential set of adders, noting that when 2 numbers are added together, the range expands by a bit. 3-4 numbers grow by 2 bits, and 5-8 numbers make it grow by 3 bits, etc. Be sure to indicate the width of intermediate signals in your code when declaring wires/regs.



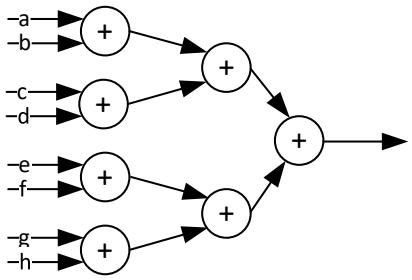

```
module sum_seq (input [3:0] a,b,c,d,e,f,g,h, output [6:0] sum);
wire [4:0] ab_sum = a + b;
wire [5:0] abc_sum = ab_sum + c;
wire [5:0] abcd_sum = abc_sum + d;
wire [6:0] abcde_sum = abcd_sum + e;
wire [6:0] abcdef_sum = abcde_sum + f;
wire [6:0] abcdefg_sum = abcdef_sum + g;
assign sum[6:0] = abcdefg + h;
endmodule
```

+1 point perfect syntax

+2 points for any answer

+1 point for each assignment/operation

Write Verilog code that twill generate a half-at-a-time set of adders, noting that every stage the intermediate sums will increase by one bit. Be sure to indicate the width of intermediate signals in your code when declaring wires/regs.



(Problem 3, Continued)

```
module sum_log (input [3:0] a,b,c,d,e,f,g,h, output [6:0] sum);
  wire [4:0] ab_sum = a + b;
  wire [4:0] cd_sum = c + d;
  wire [4:0] ef_sum = e + f;
  wire [4:0] gh_sum = g + h;
  wire [5:0] abcd_sum = ab_sum + cd_sum;
  wire [5:0] efgh_sum = ef_sum + gh_sum;
  sum = abcd_sum + efgh_sum;
endmodule
```

+1 point for perfect syntax
+2 points for an answer
+1 point each for assignment/operation

other variations are possible
watch for students who accidentally swapped a & b parts - give full credit

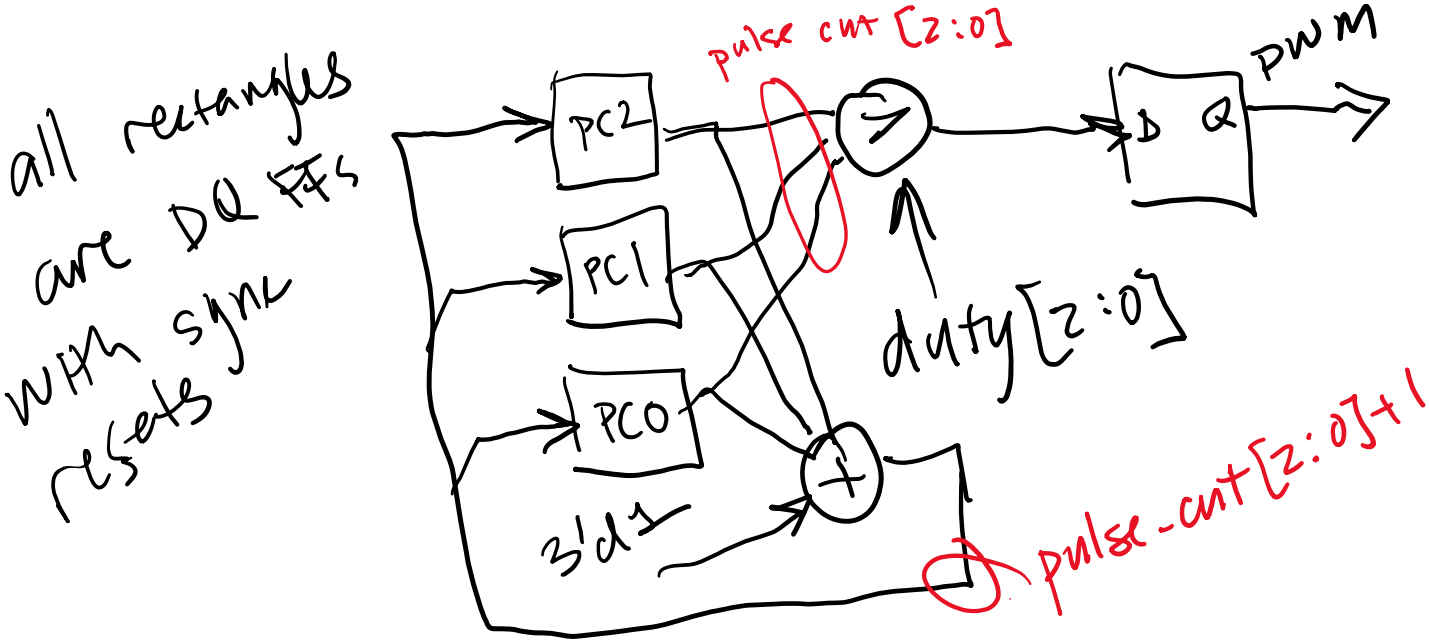
note - if students give same answer for both, give ~15 points

4. [20 points] Read the following Verilog code module, and answer the following questions from the perspective of an FPGA synthesis tool:

```
module pulse_width_mod (input clk, rst, [2:0] duty, output reg pwm);
  reg [2:0] pulse_cnt;
  always @(clk) begin
    if (rst) begin
      pwm <= 1'b0;
      pulse_cnt <= 3'b000;
    end else begin
      pulse_cnt <= pulse_cnt + 3'd1;
      if (pulse_cnt[2:0] > duty[2:0])
        pwm <= 1'b0;
      else
        pwm <= 1'b1;
    end
  end
endmodule
```

5 pts:
+2 for a diagram
+1 for 4 FFs
+1 for adder
+1 for comparator

a. Assume that all flip-flops are DQ flip-flops with active-high synchronous reset inputs R. Also assume that the (+) adder symbol shown in Problem 3 can be used in your block diagram. Draw a small block diagram of the circuit, including registers, input and output signals.



(Problem 4, continued)

b. Based on your circuit diagram from a, how many DQ Flip-Flops are required to create this circuit?

+2 for a number
+2 for ≥3
+1 for 4

4

(pulse_cnt[2:0] and pwm)

c. Based on your circuit diagram from a, and based on your own intuition and reasoning, determine how many 4-input look-up-tables (LUTs) are required to realize this circuit. DO NOT SIMPLY PUT A NUMBER, but rather please explain how many LUTs are required to realize all of the combinatorial paths in the block diagram from part a.

We will need LUTs for the comparator and for the adder. Here the adder is 3 variable bits on one branch and 3 constant bits on the other, so really we just need 3 LUTs, one for each output bit. If the student assumed that the full adder needed to use a carry chain, it could be up to 2 LUTs per stage for a total of 6 LUTs, perhaps dropping one LUT on either the LSB or MSB, so within reason it should be between 3-6 LUTs for the adder.

For the comparator to determine if two 3-bit variables are greater than each other, but then to invert the output (so alternatively a less-than or equal-to operator), one would need to first look at the MSbits, then the middle bits, then the lowest bits. If we had the top 2 bits from each number (pulse_cnt and duty) coming into a pair of LUTs, and let the output of one LUT be high if the pair of 2-bit numbers are equal, and let the output of the other be high if the upper 2-bits of pulse_cnt are greater than duty. Then add a third LUT to this function to look at the LSBs from each input number, and also bring in the equal/greater status. If the first comparison was not equal and not greater (it was less), the output is 0. If the first comparison was greater, then the output is 1. If the first comparison was equal, then the LSBs break the tie to tell whether pulse_cnt was greater than duty. So this required 3 LUTs total. Again, other combinations of LUTs that generate the same logical output are acceptable – students do not need to be optimal, but they need to provide the logical output.

+2 for a number
+2 for
≥4
+1 for
≤12

So a total of 3 LUTs for adder + 3 LUTs for comparator = 6 LUTs

d. Based on your answer for part c, briefly describe the function handled by each 4-input LUT. This can be described either by a logical equation/formula, or by a brief description of the operation(s) each LUT handles. NOTE: THIS MUST BE COMPLETED ON A LUT-BY-LUT BASIS FOR EACH LUT IDENTIFIED IN PART C.

```
Adder LUT0: F = ~pulse_cnt[0]

Adder LUT1: F = pulse_cnt[0] ^ pulse_cnt[1]
```

+1 for an answer
+1 for mention of adder

```
Adder LUT2: F = pulse_cnt[2] ^ (pulse_cnt[0] & pulse_cnt[1])
Comparator – based on "greater-than" logic
Comparator LUT0: X = pulse_cnt[2:1] > duty[2:1];
Comparator LUT1: Y = pulse_cnt[2:1] == duty[2:1];
Comparator LUT2: Z = ~X & ~(Y & pulse_cnt[0] & ~duty[0]); // inverted
Alternative Comparator – based on less-than/equal-to logic
Comparator LUT0: X = pulse_cnt[2:1] < duty[2:1];
Comparator LUT1: Y = pulse_cnt[2:1] == duty[2:1];
Comparator LUT2: Z = X | (Y & (~pulse_cnt[0] | duty[0]));
```

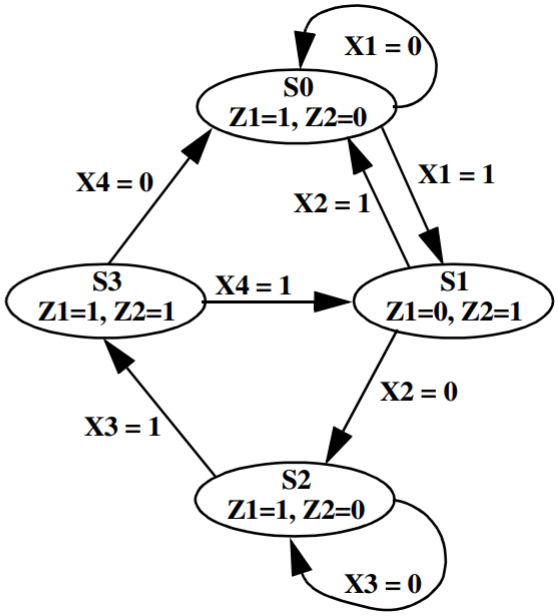
+1 for mentioning of comparator

+1 each for the absolute correct answer

5. [20 points] For the given state graph:

Inputs:
X1, X2, X3, X4

Outputs:
Z1, Z2



a. Derive the simplified next-state and output equations by inspection. Use the following one-hot state assignments for the flip-flops $Q_0Q_1Q_2Q_3$:
S0, 1000; S1, 0100; S2, 0010; S3, 0001;

Q0 = (Q3 & ~X4) | (Q1 & X2) | (Q0 & ~X1)
Q1 = (Q0 & X1) | (Q3 & X4)
Q2 = (Q1 & ~X2) | (Q2 & ~X3)
Q3 = (Q2 & X3)

Z1 = ~Q1 or Z1 = Q0 | Q2 | Q3
Z2 = Q1 | Q3

8 pts:
+1 per bit assignment attempt
+1 per correct assignment

Students will not be penalized if they swapped the order of one-hot bits, or express the above assignments/equations in some other equivalent fashion.

NOTE FOR THE FOLLOWING SECTION: ANSWER KEY USES State[3:0] instead of Q3, Q2, Q1, Q0…

b. Provide Verilog code to implement the above state graph. Note that the Z1 and Z2 outputs, which depend only on the 4-bit State register, must NOT be delayed by a clock cycle with respect to the current value of State. An additional sheet of paper is provided for your convenience:

```
module prob5_moore (input clk, rst, X1, X2, X3, X4, output Z1, Z2);
reg [3:0] State;

      always @(posedge clk) begin
            if (rst)
                  State <= 4'b0001;
                  // flipping notation here, students can write 1000
            else begin
                  // here I copied the above - any other logical
                  // function to achieve a one-hot state transition
                  // is perfectly acceptable for full credit
                  State[0] <= (State[3] & ~X4) | (State[1] & X2) |
                              (State[0] & ~X1);
                  State[1] <= (State[0] & X1) | (State[3] & X4);
                  State[2] <= (State[1] & ~X2) | (State[2] & ~X3);
                  State[3] <= (State[2] & X3);
            end
      end

      // note: if these are in an always @() process, it must either be
      // always @(*) or always @(State), otherwise a 2-point deduction
      // is taken
      assign Z1 = State[0] | State[2] | State[3];
      assign Z2 = State[1] | State[3];
endmodule
```

6 pts total

+2 for sync process
+1 point for each correct state bit assignment

+1 point for assigning each
+1 point for correct values
+1 point for correct timing