University of California at Berkeley
College of Engineering
Department of Electrical Engineering and Computer Sciences

EECS151/251A J. Wawrzynek
Spring 2021 3/11/21

Exam 1

Name: _____

Student ID number: _____

Class (EECS151 or EECS251A): _____

You have three hours to take the exam. This exam comprises a set of questions with 1 point per expected minute of completion with a total of approximately 135 points. As with homework problems, submit your solutions using Gradescope. At the end of the exam time, you will have extra time to scan and submit your answers.

You are allowed to refer to your notes, the class lecture notes, and any other reference materials that you have available. You are not allowed to speak or communicate with anyone on any topic related to the exam during the exam period. After completing the exam, sign the following statement attesting that you did not discuss the exam problems with anyone else. You may either scan this page or copy the statement word-for-word.

I hereby declare that I have not spoken with nor otherwise communicated with anybody regarding the content of this exam while taking the exam (except for course staff):

(sign here): _____

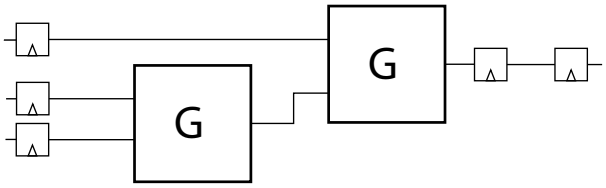
For each problem if you find yourself taking excessive time to work out a solution consider skipping the problem or a fresh approach. Also, start by answering the easier questions and then move on to the more difficult ones.

Neatness counts. We will deduct points if we need to work hard to understand your answer.

Before you turn in your exam, write your student ID number on all pages.

1. Tradeoffs [12 pts]

   You're working for an ASIC design company and responsible for a part of the company's latest chip design. You come up with a design shown in block diagram below.



   The blocks labeled G are combinational logic blocks and both implement the same Boolean function. For your process, flip-flops have the following area and delay:

$$area_{FF} = 5\,\mu m^2$$
$$\tau_{clk-q} = \tau_{setup} = 10\,ps$$

   The area and delay of your G block is:

$$area_G = 40\,\mu m^2$$
$$\tau_G = 30\,ps$$

   Your boss tells you, thanks for the design, and asks you to make sure your implementation meets the following specification for total area and critical path delay:

$$T_{max} = 80\,ps$$
$$area_{max} = 80\,\mu m^2$$

   Your implementation of G comes from a circuit generator. You had tried a range of parameters with the generator and discover that it can produce a family of designs trading-off area for delay with the following values:

area (μm^2)	10	20	30	40	50
delay (ps)	120	60	40	30	24

   (a) Does your original design meet the specifications?

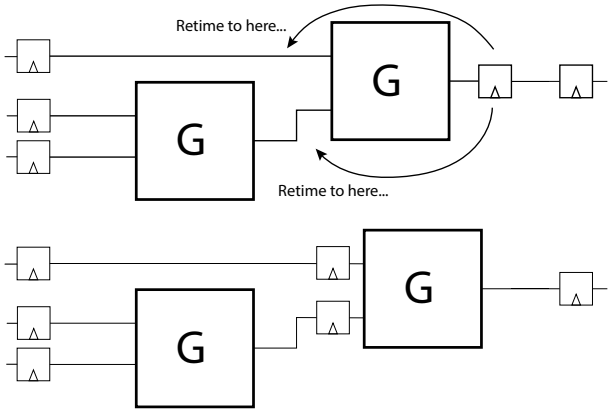
   Solution:

   No. The circuit as it is now will meet timing, but violates the area constraint. The two G blocks already contribute $80\,\mu m^2$, so there is no room left for the flip-flops.

   (b) If not, can you come up with a new design that does? *You are allowed to pick different G generator parameters and rearrange the circuit, as long as you maintain the same input/output functionality.*

Solution:

None of the other members in the generated design family will satisfy the contraints with this circuit as-is, so we will need to do some retiming. To relax the delay constraint per G block (and thereby allow us to use less area per block), we can retime the second G block by pushing a register to its inputs, as shown below.



Now that there is only 1 G block in the critical path, we can design them to be just fast enough to meet the critical path requirement, which will simultaneously minimize the area of the design. The slowest design that meets timing has 60 ps of delay and takes up 20 µm² of area, bringing the total area to

$$6(5\,\mu m^2) + 2(20\,\mu m^2) = 70\,\mu m^2$$

which meets the area constraint. Note that because of the retiming, there is now 1 additional flip-flop that contributes to the overall area.

2. FPGAs [15 pts]

(a) John is a theoretician at heart but works on FPGA architecture research. He comes up with the following two formulas for expressing LUT area and LUT delay as a function of N, the number of LUT inputs, for N ≥ 2. In the formulas k_a and k_t are layout and process related constants.

$$Area = k_a \cdot 2^N$$

$$Delay = k_t \cdot N$$

Are John's formulas accurate? Explain. What are the constants, k_a and k_t, meant to represent?

Solution:

The formulas are correct.

An N-LUT consists of 2^N configuration bits (latches) and a 2^N-to-1 MUX. A 2^N-to-1 MUX can be constructed from a binary tree of (2^N - 1) 2-to-1 MUXes with a depth of of log2(2^N) = N.

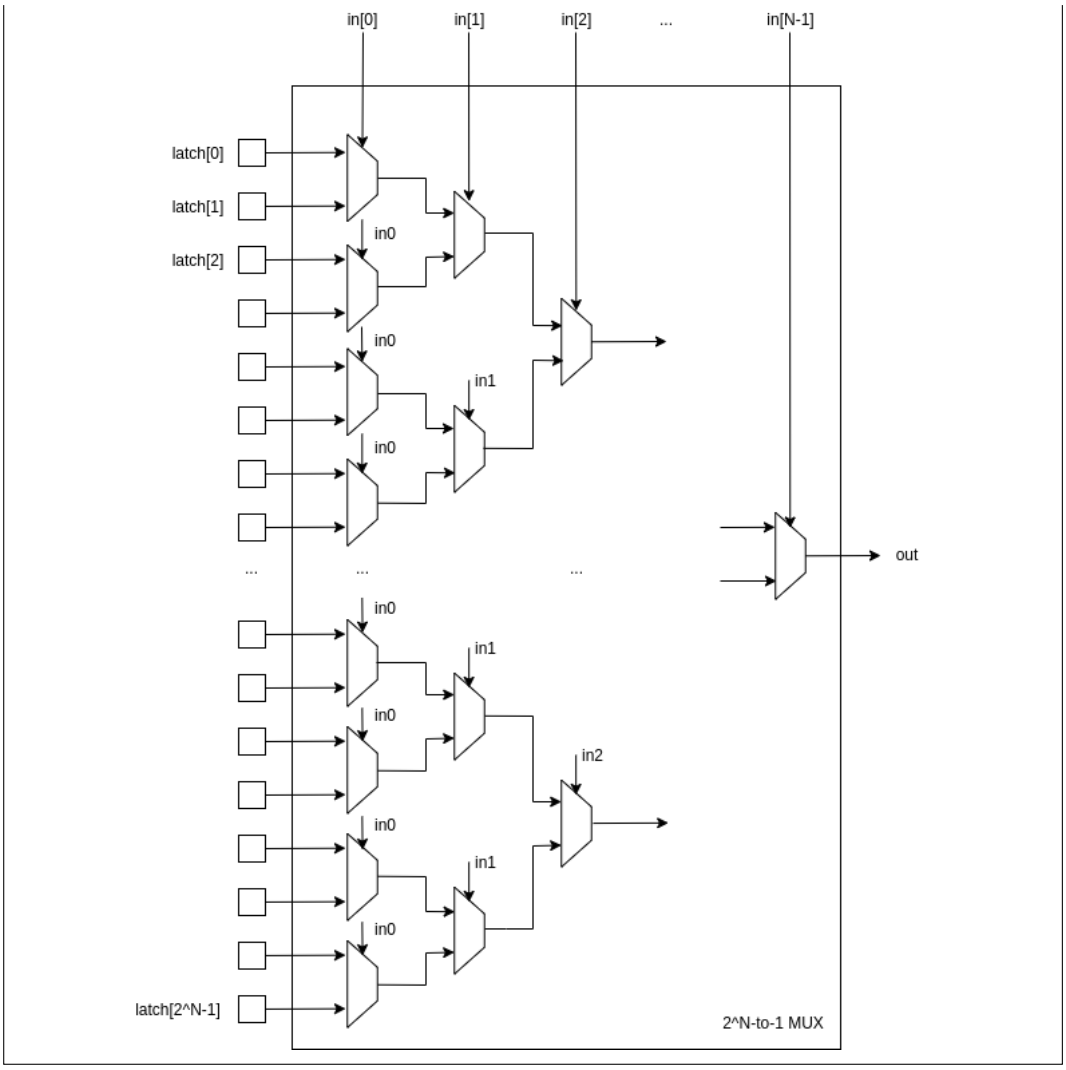
Assume the area of a config latch is a_l, and the area of a 2-to-1 MUX is a_m, and the gate delay of a 2-to-1 MUX is d.

The area of an N-LUT is $Area = a_l \cdot 2^N + a_m \cdot (2^N - 1) \approx (a_l + a_m) \cdot 2^N$ (if N is large enough)

The delay of an N-LUT is $Delay = d \cdot N$ (the longest path delay – which is the propagation delay from the latches to N levels of MUXes to the LUT output)

Therefore, the area scales proportionally to 2^N, and k_a represents the area of a config latch plus a 2-to-1 MUX. The delay scales proportionally to N, and k_t represents the gate delay of a 2-to-1 MUX.

The following figure shows how an N-LUT is typically built



(b) Consider the problem of mapping an arbitrary function of P inputs using only 3-LUTs. Without knowing details of the function, write a formula that ex-

presses the least number of 3-LUTS needed to implemement any arbitrary function. Assume $P \geq 3$. Explain your approach and show your work.

Solution:

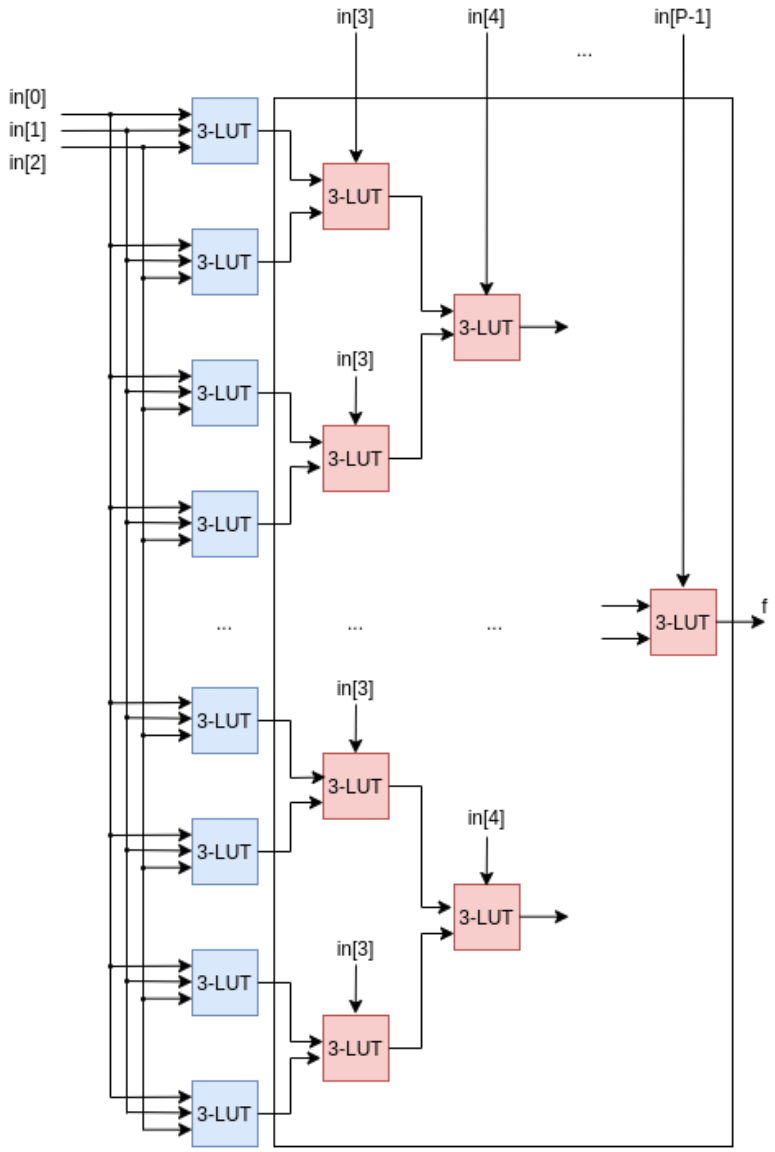
i. Divide the function truth table (2^N rows) into group of 8 rows each (3 input bits)

ii. Assign one 3-LUT per group $\Rightarrow 2^P/8 = 2^{P-3}$ 3-LUTs needed

iii. Use a tree of 2-to-1 MUXes implemented using 1 3-LUT for each 2-to-1 MUX. The tree has 2^{P-3} inputs $\Rightarrow$ a total of $2^{P-3} - 1$ 3-LUTs needed

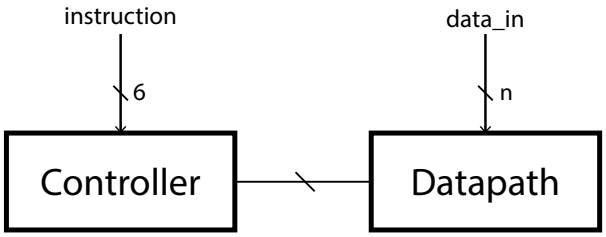
iv. Total number of LUTs for both parts $= 2^{P-3} + 2^{P-3} - 1 = 2^{P-2} - 1$

The following figure shows how we can build a P-input function (or P-LUT) from 3-LUTs. The blue 3-LUTs are used to implement logic, and the red 3-LUTs are used to implement 2-to-1 MUXes.

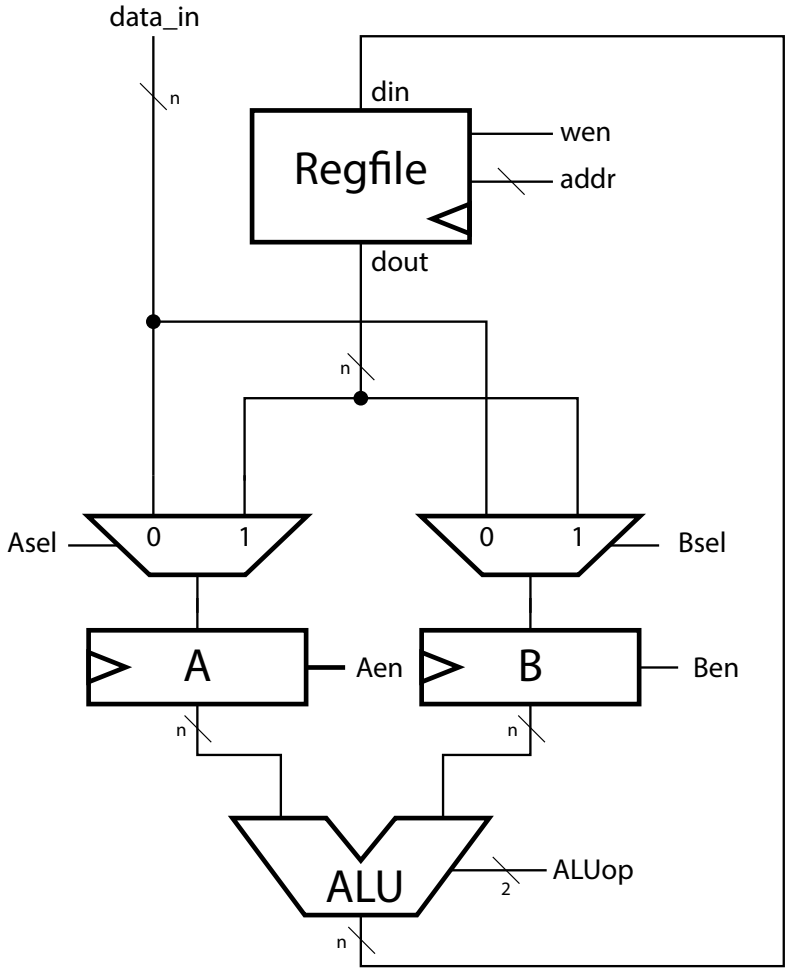


3. Verilog [25 pts]

Consider the design of a simple n-bit wide computation engine that accepts and executes exactly one instruction per clock cycle. At the top level the engine comprises a controller and a datapath, as shown below.



Details of the datapath are shown below. The register file (regfile) has asynchronous read and synchronous write.



The engine has an instruction set with only 6 instructions, described below. All instructions use the same format shown here.

```
Instruction Format:

| opcode |  reg   |
 5      3 2      0
    <- bits ->

Instruction Definitions:

opcode | action
0        regfile[reg] <- A + B
1        regfile[reg] <- A - B
2        regfile[reg] <- A XOR B
3        regfile[reg] <- A NAND B
4        B <- regfile[reg],  A <- datain
5        A <- regfile[reg],  B <- datain
```

Assume that an external circuit presents one instruction per cycle to the engine and holds it constant throughout the cycle.

Your job is to write the Verilog specification for four modules, the ALU, the datapath, the controller, and the engine. Remember, *no register inference*—use register instantiation. Also, you have available to you a predefined register file generator with the following definition:

```
module regfile (
 clk,    // system clock
 din,    // data input for writes
 dout,   // data output for read
 wen,    // write enable for synchronous write
 a       // address in register for write
 );
 parameter N;    // register width
 parameter M;    // number of registers
 parameter AW;   // width of address in bits
```

Solution:

```
// ALU
module ALU #(parameter N = 32) (
  input [N-1:0] A,
  input [N-1:0] B,
  input [1:0] ALUop,
  output reg [N-1:0] ALUout
);
```

```
  always @(*) begin
    case (ALUop)
      2'b00: ALUout = A + B;
      2'b01: ALUout = A - B;
      2'b10: ALUout = A ^ B;
      2'b11: ALUout = ~(A & B);
    endcase
  end

endmodule

// datapath
module datapath #(parameter N = 32, parameter M = 8,
 ↪  parameter AW = 3) (
  input clk,
  input [N-1:0] data_in,
  input [AW-1:0] addr,
  input wen, Asel, Bsel, Aen, Ben,
  input [1:0] ALUop
);

  wire [N-1:0] din;
  regfile #(.N(N), .M(M), .AW(AW)) RF (
    .clk(clk),
    .din(din),
    .dout(dout),
    .wen(wen),
    .a(addr),
  );

  wire [N-1:0] A_next, A_value;
  wire A_ce;
  REGISTER_CE #(.N(N)) A_reg (
    .clk(clk),
    .d(A_next),
    .q(A_value),
    .ce(A_ce)
  );

  wire [N-1:0] B_next, B_value;
  wire B_ce;
  REGISTER_CE #(.N(N)) B_reg (
    .clk(clk),
    .d(B_next),
    .q(B_value),
```

```
    .ce(B_ce)
  );

  wire [N-1:0] ALUout;
  ALU #(.N(N)) alu (
    .A(A_value),
    .B(B_value),
    .ALUop(ALUop),
    .ALUout(ALUout)
  );

  assign A_next = (Asel) ? dout : data_in;
  assign A_ce   = Aen;
  assign B_next = (Bsel) ? dout : data_in;
  assign B_ce   = Ben;
  assign din = ALUout;

endmodule

// controller
module controller #(parameter AW = 3) (
  input [5:0] instruction,
  output reg wen, Asel, Bsel, Aen, Ben, ALUop,
  output [AW-1:0] addr
);

  assign addr = instruction[2:0];
  wire [2:0] opcode = instruction[5:3];

  always @(*) begin
    Aen = 1'b0;
    Ben = 1'b0;
    Asel = 1'b0;
    Bsel = 1'b0;
    ALUop = 2'b00;
    wen = 1'b1;
    case (opcode)
      3'd0: begin
        ALUop = 2'b00;
      end
      3'd1: begin
        ALUop = 2'b01;
      end
      3'd2: begin
        ALUop = 2'b10;
```

```
        end
        3'd3: begin
          ALUop = 2'b11;
        end
        3'd4: begin
          wen = 1'b0;
          Aen = 1'b1;
          Ben = 1'b1;
          Asel = 1'b0;
          Bsel = 1'b1;
        end
        3'd5: begin
          wen = 1'b0;
          Aen = 1'b1;
          Ben = 1'b1;
          Asel = 1'b1;
          Bsel = 1'b0;
        end

        // can also have a default case statement here
        // if we do not specify the default assignments
        // before the case block
      endcase
    end

endmodule


// engine
module engine #(paramter N = 32, parameter M = 8, parameter
 ↪  Aw = 3) (
  input clk,
  input [N-1:0] data_in,
  input [5:0] instruction
);

  wire [AW-1:0] addr;
  wire Asel, Bsel, Aen, Ben, wen;
  wire [1:0] ALUop;

  datapath #(.N(N), .M(M), .AW(AW)) dp (
    .clk(clk),
    .data_in(data_in),
    .addr(addr),
    .wen(wen),
```

```
        .Asel(Asel),
        .Bsel(Bsel),
        .Aen(Aen),
        .Ben(Ben),
        .ALUop(ALUop)
      );

      controller #(.AW(AW)) ctrl (
        .instruction(instruction),
        .addr(addr),
        .wen(wen),
        .Asel(Asel),
        .Bsel(Bsel),
        .Aen(Aen),
        .Ben(Ben),
        .ALUop(ALUop)
      );

    endmodule
```

4. CMOS Gates [20 pts]

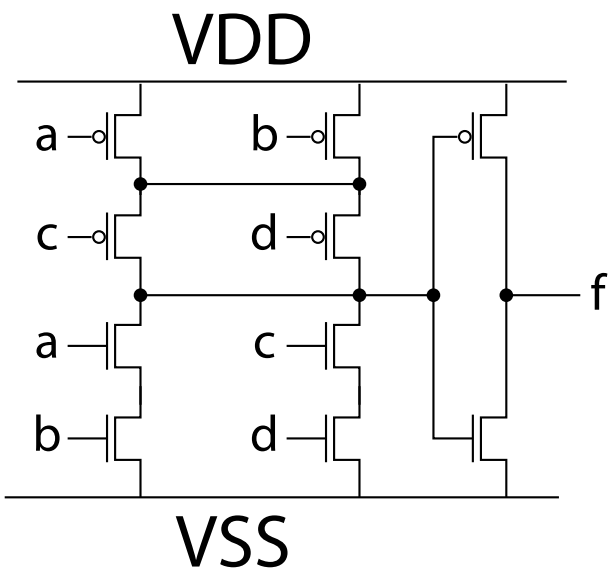
In this problem you are asked to design and analyze a 4-input static CMOS gate that implements:

$$f = ab + cd$$

Your approach will be to use a composition of a single gate that implements $\overline{f}$ followed by an inverter.

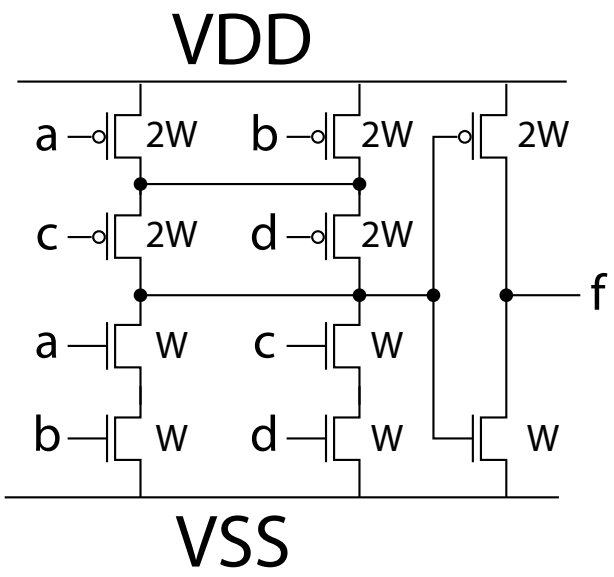
(a) Show your circuit diagram for f.

Solution:



(b) Size the transistors in your $\overline{f}$ gate so that each input has the same capacitance as a unit sized inverter, and derive the equation for worst case delay. Assume that the resistance per unit width for pFETs is twice that of nFETs. Also size the transistors so that the rise and fall times are equivalent.

Solution:



The worst-case delay is caused by having two devices in series from either rail to the output. This path is therefore the path we will balance. Since both the worst-case pull-up and worst-case pull-down have the same number of devices in series, the ratio of PMOS to NMOS device sizes will be the same as the ratio of their channel resistances. The PMOS devices have twice the resistance of the NMOS devices, so they must be double the width. By making all the PMOS devices 2W and the NMOS devices W, we automatically have the same input capacitance as a unit inverter, so we are done.

(c) Now, assuming that your output inverter is unit sized, derive a delay equation for the composite gate.

Solution:

The inverter delay can be found directly from the lecture slides,

$$t_{p,inv} = t_{p0}\left(1 + \frac{f}{\gamma}\right)$$

For the $\overline{f}$ gate,

$$C_{int} = 6W\gamma C_G$$

$$
\begin{aligned}
t_{p,\overline{f}} &= 0.69\left(\frac{2R_N}{W}\right)(C_{int} + C_L) \\
&= 0.69\left(\frac{2R_N}{W}\right)(6W\gamma C_G + C_L) \\
&= 0.69\left(\frac{2R_N}{W}\right)(3W\gamma C_G)\left(2 + \frac{C_L}{3W\gamma C_G}\right) \\
&= t_{p0}2\left(2 + \frac{C_L}{3W\gamma C_G}\right) \\
&= t_{p0}\left(4 + \frac{2C_L}{3W\gamma C_G}\right) \\
&= t_{p0}\left(4 + \frac{2f}{\gamma}\right)
\end{aligned}
$$

Since the inverter is unit-sized, it presents a fanout of 1 to the complement gate, so the final delay of the complement gate is

$$t_{p,\overline{f}} = t_{p0}\left(4 + \frac{2}{\gamma}\right)$$

Summing this with the inverter delay,

$$t_p = t_{p0}\left(5 + \frac{2}{\gamma} + \frac{f}{\gamma}\right)$$

The fanout-dependent delay is therefore entirely from the last stage (inverter), and the internal load of the inverter input turns the fanout-dependent delay of the complement gate into a constant delay.

(d) You realize that you might be able to scale up the size of the output inverter for better performance—a 4X inverter seems like a good choice. Derive a delay equation for this new composite gate.

Solution:

Sizing up the inverter 4x does not affect the delay of the gate, so the inverter delay remains the same

$$t_{p,inv} = t_{p0}\left(1 + \frac{f}{\gamma}\right)$$

Since the inverter is 4x sized, it presents a fanout of 4 to the complement gate, so the final delay of the complement gate is

$$t_{p,\overline{f}} = t_{p0}\left(4 + \frac{8}{\gamma}\right)$$

Summing this with the inverter delay,

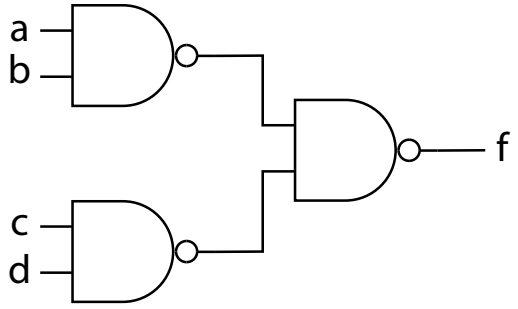
$$t_p = t_{p0}\left(5 + \frac{8}{\gamma} + \frac{f}{\gamma}\right)$$

251A only — *Optional* Challenge Question for 151

(e) An alternative way to implement this function is with NAND gates. Draw the circuit diagram for this approach and derive the delay equation with input capacitance the same as a unit inverter. Now consider scaling up the output stage. What would be its optimal size, assuming our composite gate has fanout of 4?

Solution:

This gate can be built as shown below,



The delay of a 2-NAND is, from the lecture,

$$t_p = t_{p0}\left(2 + \frac{4f}{3\gamma}\right)$$

Since each NAND is loaded by another NAND of unit size, the total delay is

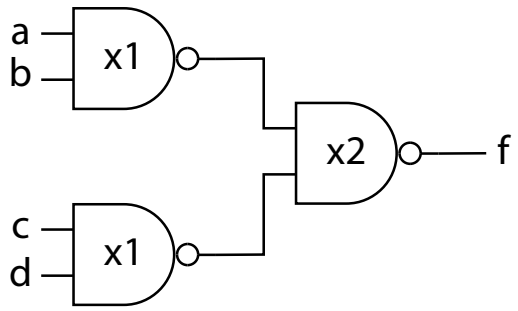
$$t_p = t_{p0}\left(4 + \frac{4}{3\gamma} + \frac{4f}{3\gamma}\right)$$

If this gate is now driving a 4x larger load (fanout of 4), then it would be optimal to distribute the fanout equally on all stages,

$$f = \sqrt{4} = 2$$

So the second NAND is 2x larger, and the overall delay is

$$t_p = t_{p0}\left(6 + \frac{8}{3\gamma} + \frac{4f}{3\gamma}\right)$$



Since the last stage has fanout of 2,

$$t_p = t_{p0}\left(6 + \frac{16}{3\gamma}\right)$$

5. Sequential Circuit Design [15 pts]

Some particular sequential circuit has a 3-bit output labeled $[x_2, x_1, x_0]$. It outputs a new value on each clock cycle in the following repeating sequence:

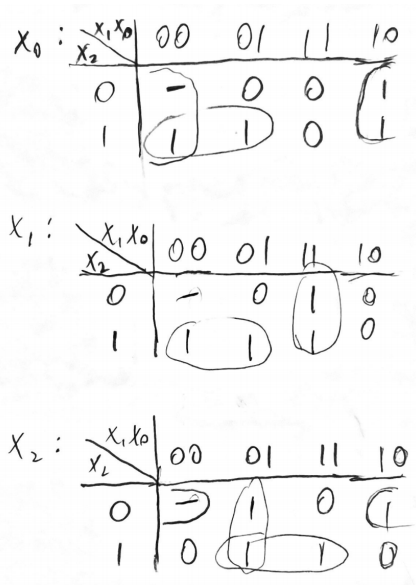
`3, 2, 5, 7, 6, 1, 4, 3, 2, ...`

Using flip-flops and 2-input ANDs and ORs, and, if needed, inverters. Derive a circuit with this behavior. Optimize for cost by trying to minimize the number of logic gates. Show your work. *Hint: think about this as a FSM.*

Solution:

In binary, the sequence, encoded as $\{x_2, x_1, x_0\}$, is `011, 010, 101, 111, 110, 001, 100, 011, 010`.

If we approach the problem as a binary-encoded FSM, we should try to come up with as simple of a representation as possible for each of x_0, x_1, and x_2. To do so, we can use a K-map to write the SOP form of each bit. For each bit, we fill each K-map entry with 1 or 0 depending on the value of the bit the cycle following the 3-bit value encoded by the K-map entry. For example, in the second cycle, bit x_0 is 0, so the entry for `011`, the previous value, in the K-map for x_0 would be 0. Here are the K-maps:



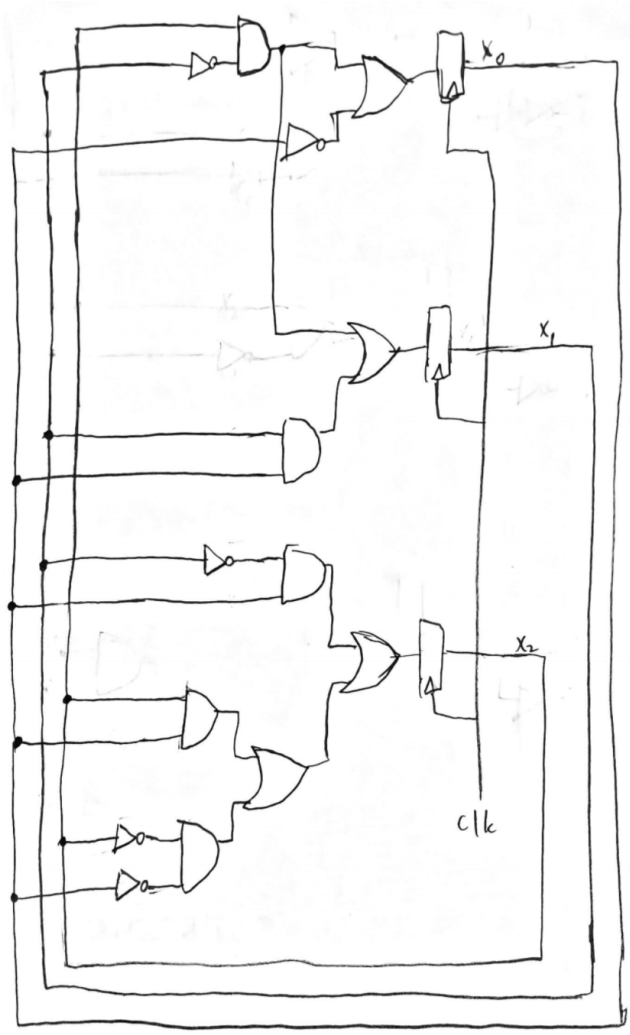
From each K-map, we can write an SOP expression:

$$x_0 = x_2\bar{x_1} + \bar{x_0}$$

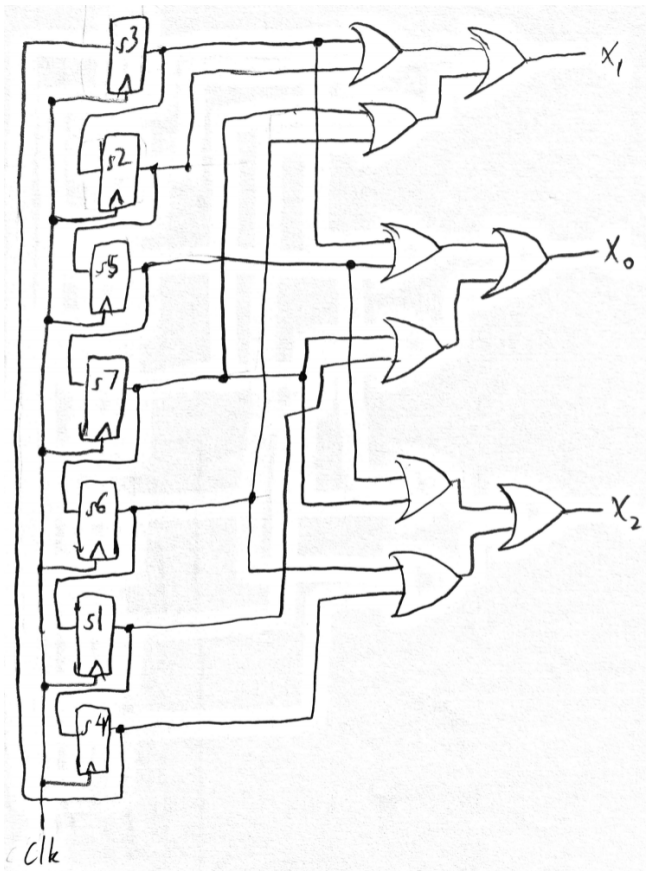
$$x_1 = x_2\bar{x_1} + x_1x_0$$

$$x_2 = \bar{x_1}x_0 + x_2x_0 + \bar{x_2}\bar{x_0}$$

Based on these expressions, we can represent each bit with a flip-flop, and assign the input of each flip-flop with some combinational logic. Note that for neatness, we did not use the minimal number of inverters in this diagram.

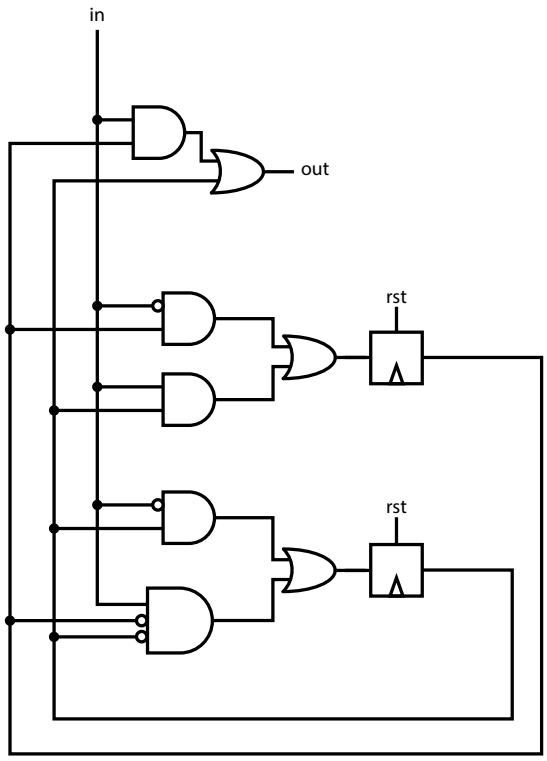


Alternatively, we can represent the 7 different states in the "FSM" with a one-hot encoding, where each state, represented by a flip-flop, simply feeds into the next, and the output is determined by which flip-flop holds a 1. This increases the number of flip-flops necessary but decreases the amount of combinational logic.


S3
S2
S5
S7
S6
S1
S4
Clk
X1
X0
X2

6. Finite State Machine [12 pts]

   Draw the state transition diagram representing the behavior of the circuit shown below. The flip-flops reset to 0.



Solution:

To understand the function of the FSM and find its state encoding, we first write a boolean expression for each state bit, based on the circuit. Call the output of the top flip-flop `s[0]`, and the output of the bottom flip-flop `s[1]`.

```
s[0] = !in & s[0] | in & s[1]
```

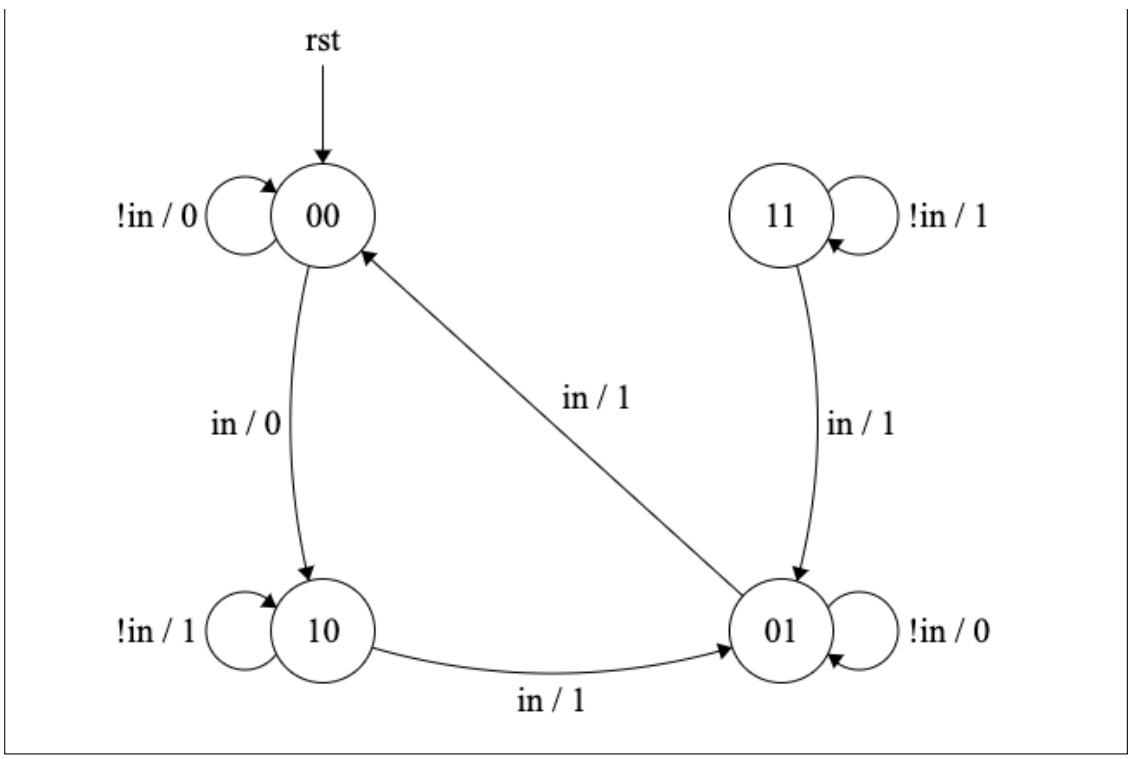
```
s[1] = !in & s[1] | in & !s[0] & !s[1]
```

From these expressions, we know that the FSM is binary-encoded, since it is possible for for `s[0]` and `s[1]` to both be 0 or both be 1 at the same time. Using these expressions, we can draw 4 states and their transitions. For example, if `{s[1], s[0]} = 2'b00` and `in = 1`, we know that the next value for `{s[1], s[0]}` is `2'b10`.

We also see that the output is a function of the current input, so we know the FSM is a Mealy machine. From the circuit, we can construct an expression for the output of the FSM at each transition:

```
out = in & s[0] | s[1]
```

Finally, we should always move to state `00` if `rst` is held high. Given all the information above, we draw the following diagram. Note that state `11` can be omitted.


rst
00
11
10
01
!in / 0
!in / 1
!in / 1
!in / 0
in / 0
in / 1
in / 1
in / 1
in / 1

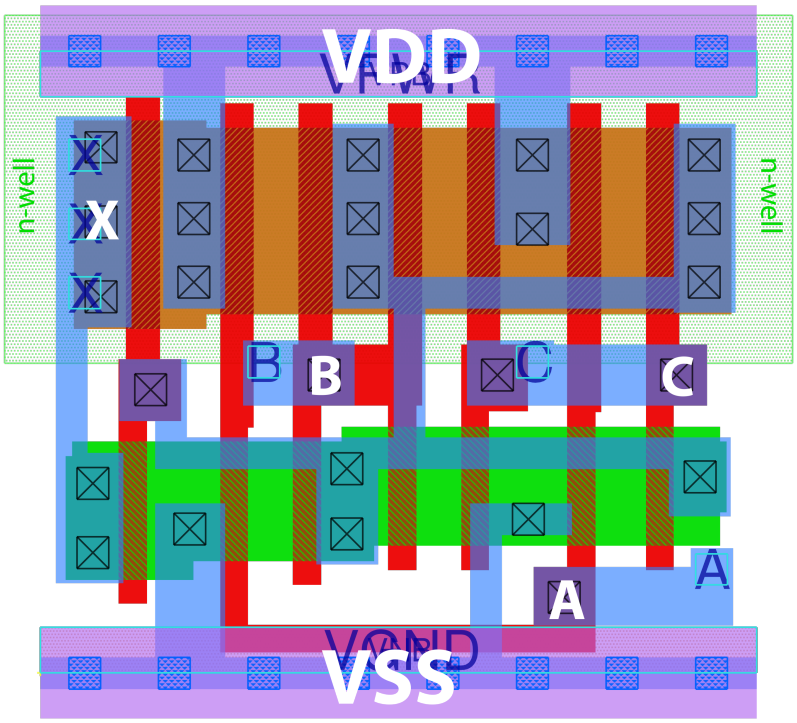
7. Layout [15 pts]

Consider the layout shown below.

(a) Extract the transistor level circuit diagram and sketch it.

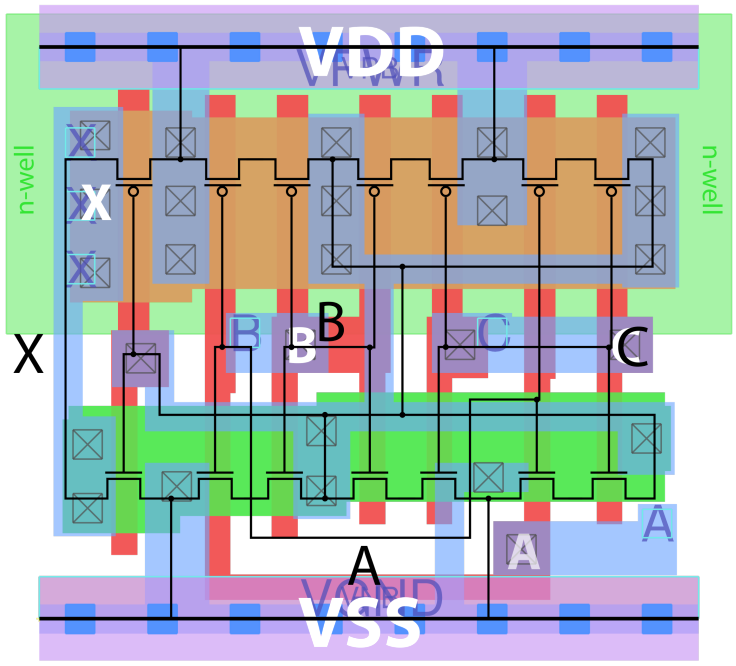
(b) Write an Boolean expression for its function.

(c) Is there a common name for this function?

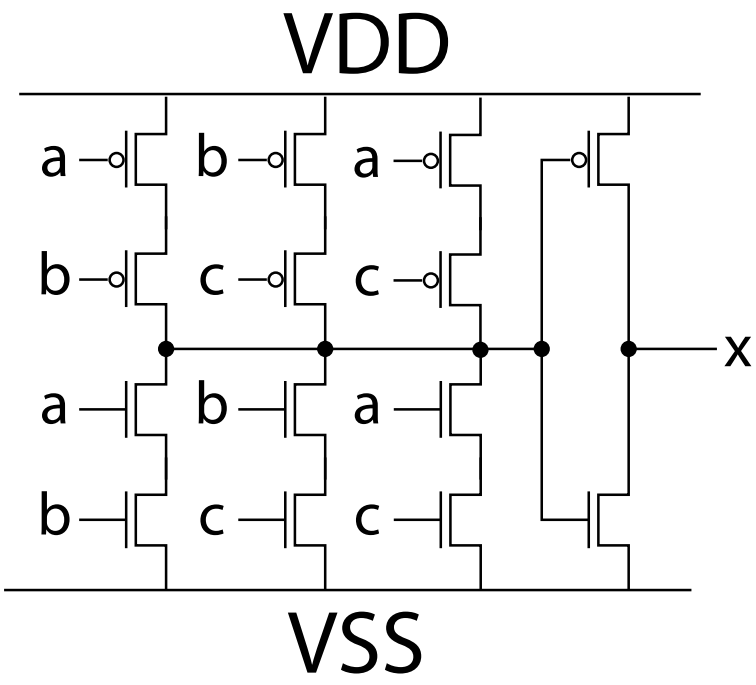


Solution:

(a) The devices are extracted as shown below,



The schematic is as follows,



(b) This circuit implements the Boolean function,

$$x = ab + ac + bc$$

Alternatively, this function can also be implemented as

$$x = (a + b)(b + c)(a + c)$$

These functions are logically equivalent.

(c) This circuit implements a **3-input majority vote** function.

8. Boolean Algebra [12 pts]

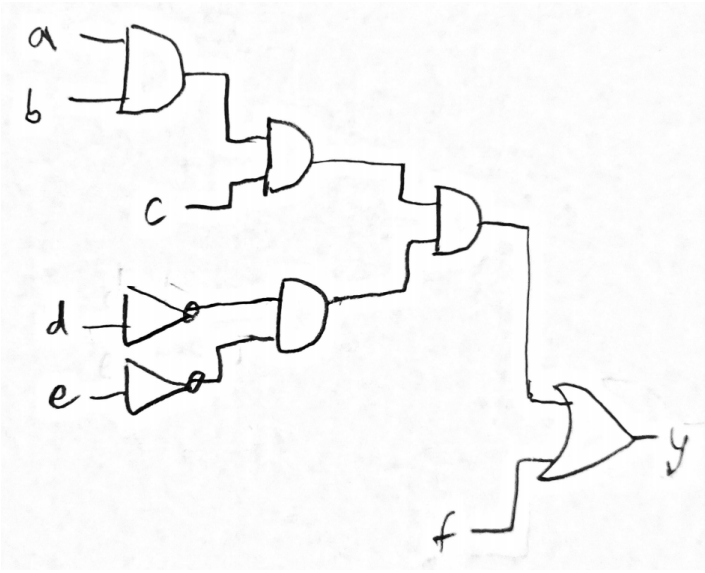
Consider the Boolean expression for some function:

$$y = abcd'e' + f$$

(a) The inputs to a circuit for this function are the signals, $a, b, c, d, e, \& f$, none of them are available in inverted form. Draw a circuit for y using only 2-input AND and OR gates and inverters that minimizes the worst case path delay. For this problem, assume that all gates and inverters have the same delay.

Solution:

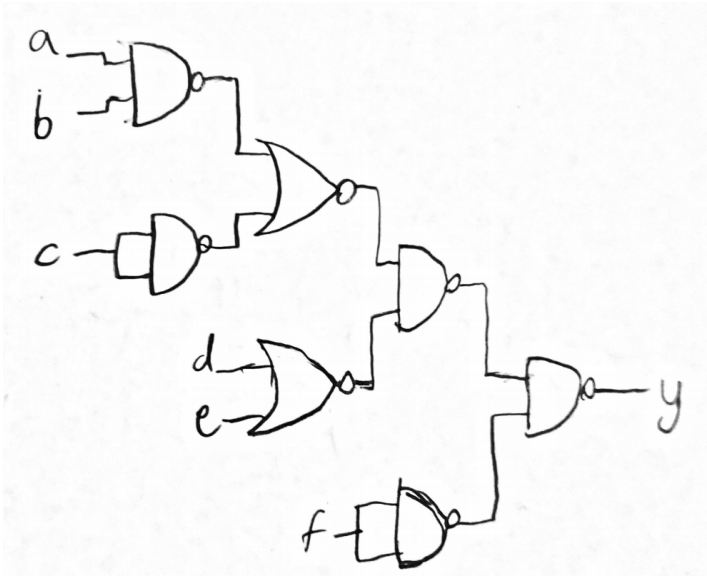
To minimize path delay, we use a binary tree-like structure.



(b) Convert the circuit to one with the same function, made up of only 2-input NAND and NOR gates.

Solution:

One method we can use to convert the circuit from part (a) is to convert the output gate from an OR to a NAND with inverted inputs. From there, we "push" the inverted inputs back. We continue this process, progressively moving back towards the inputs, until no AND/OR gates remain. Below is the circuit we arrive at following this solution.



(c) Write a Boolean expression for the NAND/NOR solution.

Solution:

Expression derived from NAND/NOR gates:

$$((((ab)' + c')'(d + e)')'f')'$$

(d) Using Boolean algebra show that your NAND/NOR algebraic expression is equivalent to y above.

Solution:

Simplify the above using De Morgan's Law:

$$\begin{aligned} & ((((ab)' + c')'(d + e)')'f')' \\ &= (((ab)' + c')'(d + e)') + f \\ &= (((ab)c)(d'e')) + f \\ &= abcd'e' + f \end{aligned}$$

Student ID number:

Thursday 18th March, 2021 18:40