

CS 199: Homework 2

Due by 10:00 a.m. on Friday, Feb 16

You are encouraged to work together with your classmates on this assignment. However, you must write up your own solutions, and indicate at the top of your front sheet who, if anyone, you worked with.

1. **Universality of Gates:** We saw that any Boolean expression corresponds to a circuit, which corresponds to a truth-table, which has a Boolean expression... etc. The demonstration that we can build an expression, hence a circuit, from any truth-table, used the “sum-of-products” technique. In essence, we made a big “OR” of ANDs. Each AND corresponded to a single row of the truth table where the output was 1. This technique made use of ORs, ANDs, and NOTs. If you think about this, what we have demonstrated was that ANY truth table can be represented using an expression that contains only AND, OR, NOT, and thus any truth table has a circuit using just these three types of gates.

We say that a set of gates is UNIVERSAL if we can build any truth table using only gates of those types. So, the set {AND, OR, NOT} is universal. What you’ll show here is that, in fact, we don’t even need all of those gate types. First, we’ll show that {AND, NOT} is universal, by showing how to eliminate OR gates.

Part 1 - Express OR by AND, NOT:

Create a truth table for $A+B$ (A OR B). Now create a truth table for $(A'B)'$. (Recall that the apostrophe indicates negation, so this is the NOT of $((\text{NOT } A) \text{ AND } (\text{NOT } B))$).

What do you conclude?

DeMorgan’s law has two parts:

- (a) $(AB)' = A' + B'$ (The negation of an AND is the OR of the negation of the parts.)
- (b) $(A + B)' = A'B'$ (The negation of an OR is the AND of the negation of the parts.)

Can you see how DeMorgan’s law applies above? Which form of DeMorgan’s law can we use to explain your observation about the truth tables?

Part 2 - Express XOR using AND, NOT:

Now let’s take a random Boolean function, say, XOR, where $A \text{ XOR } B$ is 1 exactly when one but not both of the inputs A and B are set to 1.

Show how to write XOR using only AND and NOT. *Hint: Create the truth table for XOR. Then get the sum of products representation. Finally, rewrite the OR by using the observation in Part 1.*

Part 3 - NAND is universal:

Part 1 showed that we don’t really need the OR gate, since we can always use NOT and AND to replace OR. Hence {NOT, AND} are sufficient to represent any Boolean function, so they are universal. What is surprising is that alone, the NAND gate is universal. To show this, we only need to show how to replace any “NOT” with one or more NANDs, and any AND with one or more NANDs.

First, let’s remember that NAND means “not and”, and $A \text{ NAND } B = 1$ if and only if at least one of A and B is 0.

- (a) What well-known function is $A \text{ NAND } A$ equivalent to?

- (b) Using this observation, see if you can use NAND only to replace AND. *Ask Nitish or Professor Pitt for hints if you need them.*

Now, explain briefly why NAND is sufficient to compute any Boolean function. Sit back and think about how startling that is. If you have 0s, 1s, and a (very large) bucket of NAND gates, you can put together any function you'd like.

2. **Encoders and Decoders:** In this unit, we learned how to construct several simple circuits, and some powerful ones, such as adders. In fact, we learned the sum-of-products technique that allowed us to construct *any* combinational circuit, even if we specified when the output should be 0 and when it should be 1 in a completely arbitrary way.

There are some special circuits that are very useful in a wide variety of contexts; some of these are used by almost every part of a computer. Circuits called encoders, decoders, multiplexers and demultiplexers are used by the computer processor, memory, network hardware, input and output devices, etc. In this problem, we will learn how to build encoders and decoders; in the next unit, we will learn how they are used by computers.

Part 1 - Decoders:

In the early 20th century, *player pianos* - pianos that played music automatically - became popular. The notes of a piece of music were usually represented using holes on a roll of paper. When you wanted to listen to a piece, you inserted the roll into the piano. The roll would unwind automatically, and the holes would indicate which note was to be played, and for how long. These pianos often had an intricate system of hammers and levers that would strike the appropriate notes based on the pattern of holes.¹

More modern player pianos represent the music digitally, and when music is being played back, they send electronic signals that indicate which notes are to be 'played', for how long, etc. If we had to find a representation for the 8 notes in a major music scale (Do-Re-Mi-Fa-Sol-La-Ti-Do, or C-D-E-F-G-A-B-C), we could use 3 binary digits. The pattern 000 could represent C, 001 could represent D, 010 could represent E and so on. So to represent the phrase "C-D-E-C-E-C-E", we would use 000,001,010,000,010,000,010.

So once we have the representation of the piece of music we want to play, we need circuits that 'decode' it. That is, we need a circuit that takes a binary number as input, and turns 'on' the appropriate note. For instance, if the input is 011, the circuit should turn on the note for F. At the next time step, if the input changes to 010, the circuit should turn off the note for F, and play the note for E instead.

Such a circuit is called a *decoder*. The input to a decoder circuit is a binary number, and it has to turn on one of its outputs, depending on the binary number that was input. In the previous (player piano) example, the output was the note to play next.

For another example, consider a robot that can move in one of 4 directions: North, South, East or West. To control the robot, we have to tell it which direction to move after each step it takes. We could represent the directions in binary using 2 bits, with 00 representing North, 01 as South, 10 as East, and 11 as West. Thus, the string of commands 00,10,10,10,01,01,01,11,11 represents taking 1

¹Different models of player pianos had different constructions; this is a slightly simplified explanation that describes many models.

step North, then 3 steps east, 3 South, and 2 West. The robot would need a circuit that takes our command (a binary number) as input, and turns on a motor that moves it in the appropriate direction. That is, it would need a decoder: depending on the binary number that was input to the circuit, one of the outputs is turned on.

A 2-bit decoder has 2 inputs, I_0 and I_1 , and outputs O_0 through O_3 . We can interpret the input as specifying a binary number in the range 0 through 3. If both inputs are off, this represents the binary number 00, and if both are on, we are representing 11 in binary, which is the number 3. If I_1 is off and I_0 is on, this represents the binary number 01, which is 1; if I_1 is on and I_0 is off, this represents the binary number 10 = 2.

Whatever the input, exactly one of the output lines should be on; all the others should be off. In particular, the unique output line that should be on is the one that corresponds to the binary number represented by the input. Going back to our robot example, we never want the robot to turn on two motors at once, trying to go North and South simultaneously. If the input corresponds to the binary number for South, only the output that controls the South motor should turn on.

In general, for a 2-bit decoder, if I_1 is on and I_0 off (representing the number 10), output O_2 should be on; all the others should be off. If both inputs are off (representing 00), output O_0 should be on, and if both inputs are on (representing 11), output O_3 should be on. The truth table for a 2-bit decoder is provided below.

I_1	I_0	O_0	O_1	O_2	O_3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

As practice (you do not need to turn it in), try to construct the circuit for a 2-bit decoder.

Similarly, a 3-bit decoder has 3 inputs, I_0 through I_2 , and 8 outputs, O_0 through O_7 . Just as before, only one output line should be on: the one corresponding to the binary number represented by the input lines. In our player piano example, if the input represents the binary number 100, the only output that should be on is the one that makes the piano play the note G. We wouldn't want the piano to play many other notes at the same time; that would destroy the tune.² For example, in a 3-bit decoder, if inputs I_2 and I_1 are off, and I_0 is on (representing the number 001), output O_1 should be turned on, and all the others should be off. If I_2 is on, I_1 is off, and I_0 is on (representing the number 101), output O_5 should be the only one on. The smallest binary number we can represent with 3 bits is 000, and the largest is 111 (which is 7); this is why we have 8 outputs, O_0 through O_7 .

- Draw the truth table for a 3-bit decoder.
- Design a circuit for a 3-bit decoder.
- Build the circuit using SimCir, save it, and email us your saved file.

We can generalize this to larger circuits: A k -bit decoder is a circuit that has k inputs, numbered I_0 through I_{k-1} , and 2^k outputs, numbered O_0 through O_{2^k-1} . Just as before, the inputs will specify a binary number, and one output will be on, the one corresponding to that binary number.

Part 2 - Encoders:

An *encoder* is basically the opposite of a decoder. At any time, exactly one of the inputs will be on; the outputs should be such that they represent the binary number that corresponds to the input which

²If you play music, you might wonder how we produce chords; that's a story for another time.

is on. For example, a 2-bit encoder has two outputs (O_1 and O_0), and inputs I_0 through I_3 . If input I_2 is on (which means all the others are off), the outputs should represent the number 2, or 10 in binary. Therefore, output O_1 should be on, and O_0 should be off. If input I_1 were on, the outputs should represent 01, which means that O_1 should be off, and O_0 on. If input I_3 were on, both outputs should be on, representing 11; if I_0 is on, both outputs should be off, representing 00. Remember that you never have to worry about what happens if - for instance - both I_1 and I_2 are on; you are guaranteed that at all times, only one input is on.

Here is the truth table for a 2-bit encoder. Because only 1 input is on at a time, we do not have to list all $2^4 = 16$ combinations of inputs; the truth table only contains those combinations of inputs which are valid.

I_3	I_2	I_1	I_0	O_1	O_0
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

Similarly, a 3-bit encoder has 8 inputs (I_0 through I_7), and 3 outputs (O_0 through O_2). If input 5 is on, the outputs should represent the binary number 101, so O_2 should be on, O_1 should be off, and O_0 should be on. If input 3 is on, output O_2 should be off, O_1 should be on, and O_0 should be on; this represents 011.

- Draw the truth table for a 3-bit encoder.
- Design a circuit for a 3-bit encoder.
- Build the circuit using SimCir, save it, and email us your saved file.

Again, we can build larger encoders: A k -bit encoder has 2^k inputs, I_0 through I_{2^k-1} , and k outputs, O_0 through O_{k-1} . Just as before, one input is turned on at a time; the outputs should be such that they represent the binary number corresponding to the input that is on.