CS 2110 Homework 2
Combinational & Sequential Logic

Elizabeth Hong, Henry Bui, Rick Sarkar, Kepler Boyce, Rohit Rao, Joseph Seo, Biana Jayaraman, Varun Warrier, Bryce Hanna, Andy Garcha

Spring 2025

Contents

1 Overview

1.1 Purpose

You have learned about digital logic, including transistors, gates, and combinational and sequential logic circuits. Gates (AND, OR, etc.) can be built using transistors, and combinational logic circuits (multiplexers, ALUs, etc.) can be built using gates. Sequential logic circuits (RS Latch, Gated-D Latch, etc.) also consist of gates, but unlike combinational logic circuits, they do not have set truth tables, instead depending on both current and previous inputs. Finally, we can use K-Maps and state machines to represent more advanced sequential logic. Note how the concepts build up from transistors to gates to combinational logic to sequential logic. We have provided you with a tool called CircuitSim that allows you to simulate building circuits without actually using physical hardware in order to model these concepts.

The purpose of this assignment is for you to become proficient at building gates and combinational and sequential logic circuits.

1.2 Task

You must use the provided CS 2110 version of CircuitSim, which is available via the JAR on Canvas. Other versions of CircuitSim are incompatible with our autograders.

You will submit six CircuitSim files that you produce to Gradescope: `gates.sim`, `sumofproducts.sim`, `demorgan.sim`, `plexers.sim`, `alu.sim`, `latches.sim`, and `fsm.sim`, as well as one Excel sheet `kmap.xlsx`. Each file builds upon concepts formed in the last, so it is recommended to complete the files in the order specificed by this pdf. Your circuits must work properly and produce the desired results in order to receive credit. For the ALU portion, partial credit is awarded for each operation that works properly.

Be sure to avoid using components that are disallowed for each phase of this assignment.

This document also contains tutorials on using CircuitSim.

The steps to complete this assignment are:

1. Create two of the standard logic gates (NOR, AND)
2. Create a sum of products circuit from a given truth table
3. Apply DeMorgan's Law to simplify a provided circuit
4. Create an 8-input multiplexer
5. Use multiplexers to create a circuit that evaluates the sign of a number
6. Create an 8-bit full adder
7. Use your 8-bit full adder and other components to construct two 8-bit ALUs
8. Create your own register and simple memory
9. Given a state diagram, minimize the logic by using a K-Map

2 CircuitSim Tutorials

Note: These tutorials are intended to help you get acquainted with CircuitSim before you start this homework. They are not graded. If you're comfortable with this software, feel free to skip ahead. CircuitSim is a powerful interactive ciruit simulation tool designed for educational use. This gives it the advantage of being a little more forgiving than some of the more commercial simulators. However, it still requires some time and effort to be able to use the program efficiently. With this in mind, we present you with the following assignment:

2.1 Part 1 — Read Resources

Read through the following resources

- CircuitSim Wires Documentation `https://ra4king.github.io/CircuitSim/docs/wires/`
- Tutorial 1: My First Circuit `https://ra4king.github.io/CircuitSim/tutorial/tut-1-beginner`

2.2 Part 2 — Complete Tutorial 2

Complete Tutorial 2 `https://ra4king.github.io/CircuitSim/tutorial/tut-2-xor`

Instead of saving your file as **xor.sim**, save your file as **part1.sim**. As well, make sure you label your two inputs **a** and **b**, and your output as **c**, as well as rename your subcircuit to xor.

2.3 Part 3 — Complete Tutorial 3

Complete Tutorial 3 `https://ra4king.github.io/CircuitSim/tutorial/tut-3-tunnels-splitters`

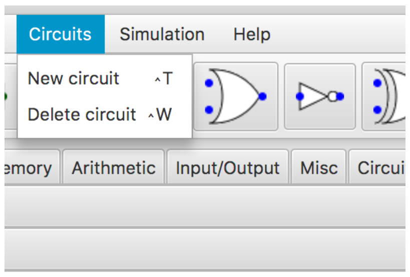
Name the subcircuit **umbrella**, the input **in**, and the output **out**. Save your file as **part2.sim**.

2.4 Part 4 — Sub-circuits Overview

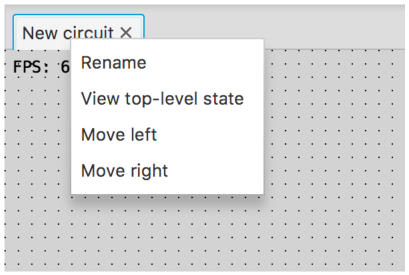
As you build circuits that are more and more sophisticated, you will want to build smaller circuits that you can use multiple times within larger circuits. Sub-circuits behave like classes in Object-Oriented languages. Any changes made in the design of a sub-circuit are automatically reflected wherever it is used. The direction of the IO pins in the sub-circuit correspond to their locations on the representation of the sub-circuit.

To create a sub-circuit:

1. Go to the "Circuits" menu and choose "New circuit"

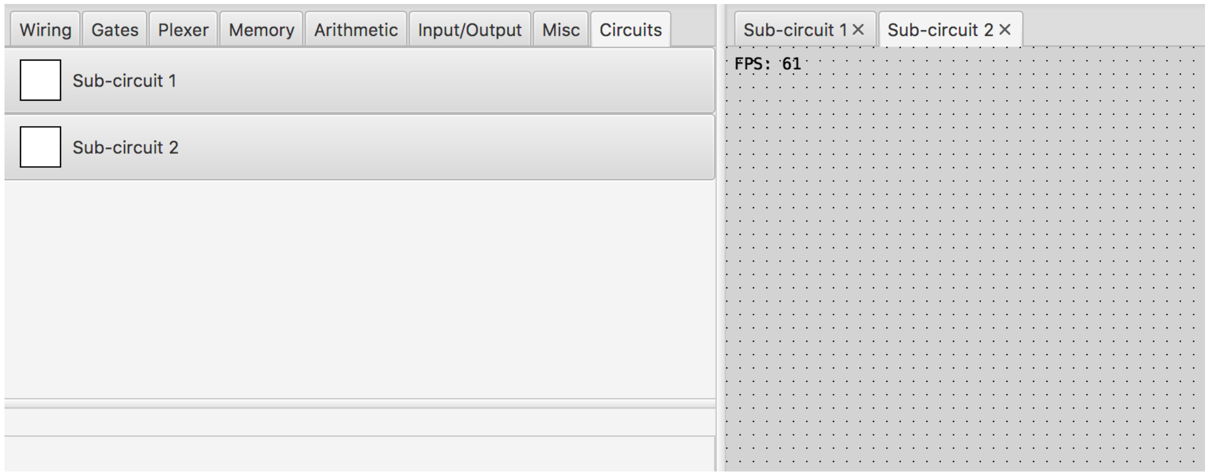


2. Name your circuit by right-clicking on the “New circuit” item and selecting “Rename”

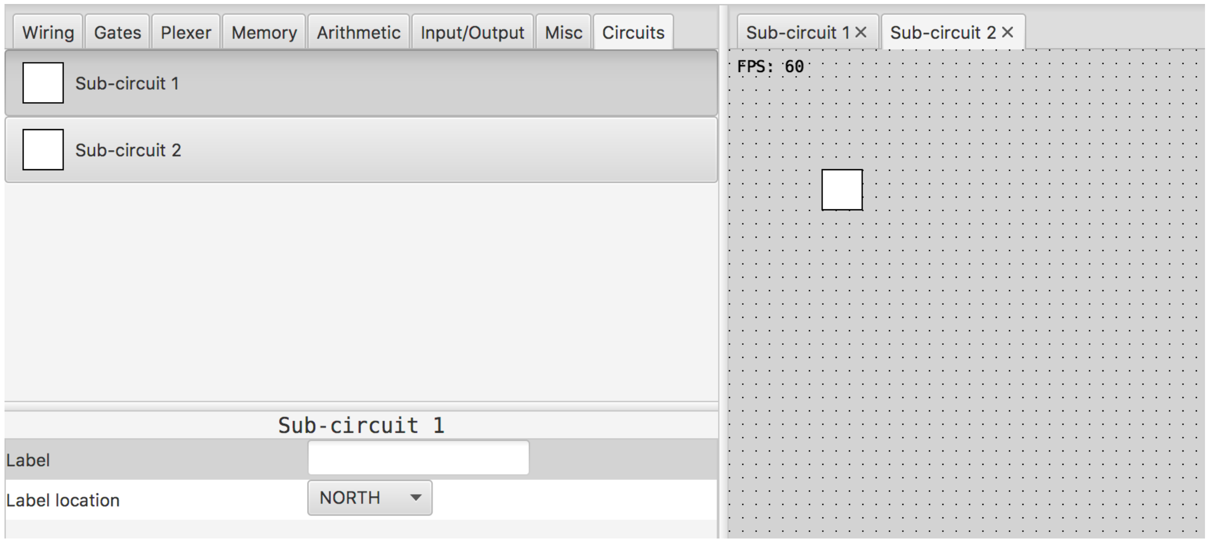


To use a sub-circuit:

1. Click the “Circuits” tab next to the “Misc” tab



2. Select the circuit you wish to use and place it in your design



Sub-circuit 1

Label	
Label location	NORTH

3 Instructions

3.1 1-Bit Logic Gates

Allowed Components: Wiring Tab

All of the circuits in this file are in the `gates.sim` file.

For this part of the assignment, you will create a transistor-level implementation of the NOR and AND logic gates.

For this section you are only allowed to use components listed in the Wiring section.

As a brief summary of the behavior of both logic gates:

NOR (Inputs: A, B - Output: OUT)

A	B	A NOR B
0	0	1
0	1	0
1	0	0
1	1	0

AND (Inputs: A, B - Output: OUT)

A	B	A AND B
0	0	0
0	1	0
1	0	0
1	1	1

Complete each of the logic gates within their named sub-circuits.

Deliverables: `gates.sim`

3.2 Sum of Products

Allowed Components: Wiring Tab, Gates Tab

You will build the circuits for this section in the Sum of Products circuit in the `sumofproducts.sim` file based on the truth table in `sumofproducts.xlsx`.

You've learned how to simplify a truth table into a sum of products form, a boolean expression composed of multiple ANDed expressions (the products) connected by OR gates (summed). Your task is to take this truth table, convert it into its sum of products equivalent, and produce a logically equivalent circuits using NOT, AND, and OR gates. **DO NOT** simplify the circuit. You may use any of the other circuits found in the GATES tab if you wish.

The truth table found in the Excel file is here for your convenience:

A	B	C	D	Output
0	0	0	0	1
0	0	0	1	0
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	1
1	0	1	0	1
1	0	1	1	0
1	1	0	0	1
1	1	0	1	0
1	1	1	0	1
1	1	1	1	1

Deliverables: `sumofproducts.sim`

3.3 DeMorgan's Law

Allowed Components: Wiring Tab, NOT Gate, 9 OR Gates, and 1 NAND Gate

The circuits in this file are in the `demorgan.sim` file.

De Morgan's laws help relate conjunctions (AND) and disjunctions (OR) of propositions with negation.

In this file, you will be creating a circuit that is an alternative to the circuit that is provided in the *Provided* circuit (first tab). Your work will be done in the *Student* circuit (second tab).

3.3.1 Provided

Nothing needs to be done in this circuit! You may find it helpful to test some inputs and build a truth table.

3.3.2 Student

This circuit needs to be directly equivalent to the circuit provided in the *Provided* circuit, meaning the **same inputs** should lead to the **same outputs**.

Here's the catch: you must to rebuild this circuit with **NOT gates, exactly 9 OR gates, 1 NAND gate, and no other gates!** You may use NOT gates / bubbles on the inputs of gates freely.

Deliverables: `demorgan.sim`

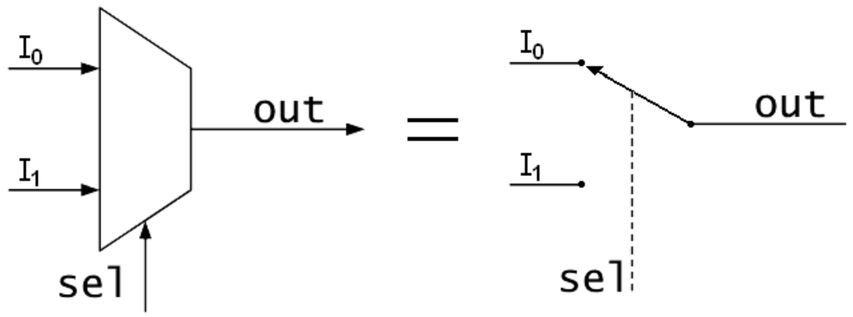
3.4 Plexers

All of the circuits in this file are in the `plexers.sim` file.

3.4.1 Multiplexer

Allowed Components: Wiring Tab, Gates Tab

The multiplexer you will be creating has 8 1-bit inputs (labeled appropriately as `A`, `B`, `C`, ..., `H`), a single 3-bit selection input (`SEL`), and one 1-bit output (`OUT`). The multiplexer uses the `SEL` input to choose a specific input line for forwarding to the output.



For example:

```
SEL = 000 ==> OUT = A
SEL = 001 ==> OUT = B
SEL = 010 ==> OUT = C
SEL = 011 ==> OUT = D
SEL = 100 ==> OUT = E
SEL = 101 ==> OUT = F
SEL = 110 ==> OUT = G
SEL = 111 ==> OUT = H
```

3.4.2 Sign Evaluation

Allowed Components: Wiring Tab, Circuits Tab, Gates Tab, and Plexer Tab

In this step, you will construct a circuit that takes an 8-bit two's complement input and evaluates whether the input is negative, zero, or positive.

Based on this, this circuit will output a 3-bit bit-vector called NZP where the most significant bit is 1 *iff* the input is negative, the middle bit is 1 *iff* the input is zero, and the least significant bit is 1 *iff* the input is positive. Only 1 bit of the output should be set at any given time. Zero is not considered a positive number.

For example:

```
INPUT = 10111000 ==> NZP = 100
INPUT = 00000000 ==> NZP = 010
INPUT = 00000100 ==> NZP = 001
```

Note that you may use the plexer tab for this circuit.

Hint: Recall that in two's complement, the most significant bit can be used to determine the sign of a number!

3.5 Adders & ALUs

All of the circuits in this file are in the `alu.sim` file.

3.5.1 1-Bit Adder

The full adder has three 1-bit inputs (`A`, `B`, and `CIN`), and two 1-bit outputs (`SUM` and `COUT`). The full adder adds `A + B + CIN` and places the sum in `SUM` and the carry-out in `COUT`.

This circuit has already been built for you. You may use it in the other circuits in this file by selecting it from the Circuits tab.

3.5.2 8-Bit Adder

Allowed Components: Wiring Tab, Circuits Tab, and Gates Tab

For this part of the assignment, create an 8-bit full adder.

This circuit should have two 8-bit inputs (`A` and `B`) for the numbers you're adding, and one 1-bit input for `CIN`. The reason for the `CIN` has to do with using the adder for purposes other than adding the two inputs.

There should be one 8-bit output for `SUM` and one 1-bit output for `COUT`.

3.5.3 Basic 8-Bit ALU

Allowed Components: Wiring Tab, Circuits Tab, Gates Tab, and Plexer Tab

You will first create a simple 8-bit ALU, using the 8-bit full adder you created previously.

For this ALU, we will be using a multiplexer. This ALU has two **8-bit** inputs for `A` and `B` and one **2-bit** input for `func`, the op-code for the operation in the list below. It has one **8-bit** output named `OUT`. The following 4 operations will be selected from:

00. Addition	`[A + B]`
01. AND	`[A & B]`
10. NOT	`[NOT A]`
11. Pass	`[PASS A]`

The **Pass** operation simply refers to outputting the value of A unchanged.

Notice that **NOT** and **Pass** only operate on the `A` input. **They should NOT rely on `B` being a particular value.**

The provided autograder will check the op-codes according to the order listed above (Addition (`00`), AND (`01`), etc.) and thus it is important that the operations are in this exact order.

3.5.4 Intermediate 8-Bit ALU

Allowed Components: Wiring Tab, Circuits Tab, Gates Tab, and Plexer Tab

You will next create a different 8-bit ALU, with identical structure as the previous, but more complex operations as outlined below. Make sure to match each `func` to the correct operation (denoted below), as you did with the basic ALU.

00. isRepeated	`[A[3:0] == A[7:4]]`

01. isLessThan63	`[A < 63]`
10. minimum(A, B / 4)	`[min(A, B / 4)]`
11. modulo8	`[A % 8]`

For the isRepeated and isLessThan63 operations, return `00000001` **if the condition is true, and** `00000000` **otherwise**.

The autograder will check the op-codes according to the order listed above and thus it is important that the operations are in this exact order.

3.6 Sequential Logic

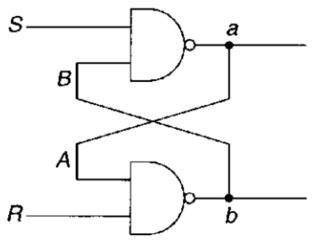
For this part of the assignment you will build your own register and simple memory from the ground up. All work for this section needs to be done in the `latches.sim` file.

3.6.1 RS Latch

The RS Latch is the basic circuit for sequential logic. It stores one bit of information, and it has 3 important states:

1. R=1 S=1 : This is called the **Quiescent State**. In this state the latch is storing a value, and nothing is trying to change that value.
2. R=1 S=0 : By changing momentarily from the Quiescent State to this state, the value of the latch is changed so that it now stores a 1.
3. R=0 S=1 : By changing momentarily from the Quiescent State to this state, the value of the latch is changed so that it now stores a 0.

This circuit is provided. Do not edit anything in this circuit. It is necessary for the Gated D Latch and D Flip Flop circuits.



3.6.2 Gated D Latch

The Gated D Latch is made up of an RS Latch as well as two additional gates that serve as a control. With that addition not only can we control what value is stored by the latch, but also when that value will be saved.

The value of the output can only be changed when Write Enable is set to 1. Notice that the Gated D Latch circuit only has one output pin, so you should disregard the inverse output of your RS Latch.

This circuit is provided. Do not edit anything in this circuit. You may use this circuit to build the D flip-flop circuit.

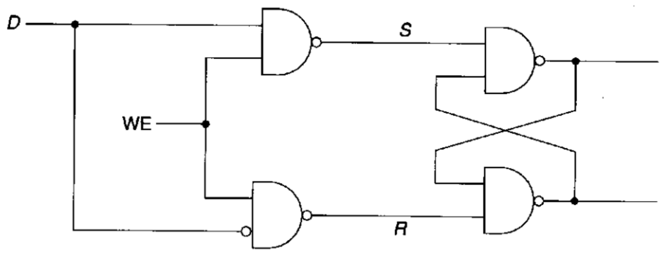


Figure 3.19 A gated D latch

3.6.3 D Flip-Flop

A leader-follower D flip-flop is composed of two Gated D latches back to back, and it implements edge triggered logic.

Implement this circuit in the "D Flip-Flop" tab of the "`latches.sim`" file. You *should* use the Gated D latch circuit.

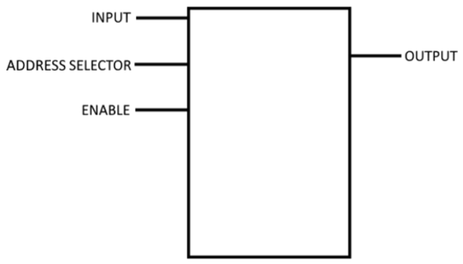
3.6.4 Register

Using the D Flip-Flop you just created, build a 3-bit register. Your register should also use edge-triggered logic. The value of the register should change on the rising edge.

Implement this circuit in the "Register" circuit in the "`latches.sim`" file. You *should* use your previous D flip-flop circuit.

3.6.5 Memory

Using the provided Gated D Latches, build a 4x1 memory where your memory has 4 addresses/rows which store 1-bit of data each. Your memory should use level-triggered logic. Take a look at the following (blackbox) memory visual....



In the provided circuit you have the following components

- 'Address Selector' or ('Selector' in circuit) - a 2-bit input and determines which row of memory we select.
  - When 'Address Selector' is 00, we select the top most row.
  - When 'Address Selector' is 01, we select the second row.
  - When 'Address Selector' is 10, we select the third row.
  - When 'Address Selector' is 11, we select the fourth row.
- 'Enable' - decides whether you want to write into memory or not.

NOTE:

- You can write/save into a particular row if the 'enable' is ON and that row is selected by the 'Address Selector'.
- Each row can contain 1-bit data.
- The 'Address Selector' also determines which row's output is displayed.

Implement this circuit in the "Memory" circuit in the "`latches.sim`" file. You *should* use your previous Gated D Latch circuit.

Deliverables: `latches.sim`

3.7 State Machines & K-Maps

Finally, you will practice your logic minimization skills by completing a K-map based on a state diagram and implementing your reduced logic in a circuit. The files for this part of the assignment are `kmap.xlsx`, `truthtable.xlsx`, and `fsm.sim`.

3.7.1 Scenario

It's a hot day (pretend it isn't January) and everyone outside is thirsty! To seize this business opportunity, you decide to build a vending machine that will sell drinks for 10¢ each. However, this will require some electronics to keep track of how much money has been inserted into the machine and dispense drinks and return change as necessary.

The vending machine will accept only two kinds of coins: nickels (5¢) and dimes (10¢). The machine thus will have four states: an "idle" state, a "5¢" state, a "drink" state, and a "drink + change" state. From any given state, there are three possible inputs to the machine: no input, inserting a nickel, and inserting a dime. There are then two outputs that the machine may have for each state: dispensing a drink and returning 5¢ of change.

The machine begins in the "idle" state. From the "idle" state:

- If a user inserts a nickel, the machine will transition to the "5¢" state.
- If a user inserts a dime, the machine will immediately have the required 10¢ and will transition directly to the "drink" state.

When the machine is in the "5¢" state:

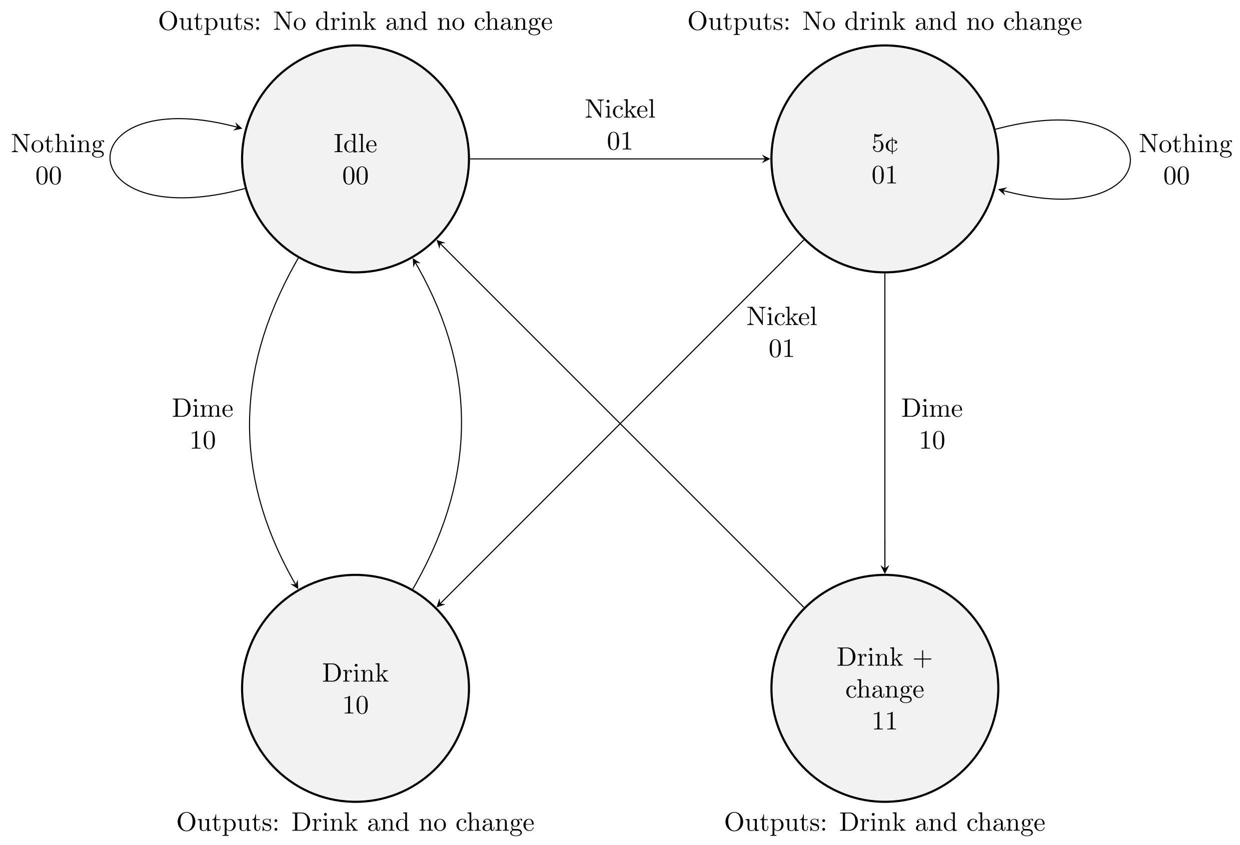
- If a user inserts another nickel, the machine will now have a total of 10¢ and it will transition to the "drink" state.
- If a user inserts a dime, then the machine will have 5¢ more than the price of the drink, so it will transition to the "drink + change" state.

If the machine is in either the "idle" state or the "5¢" state and the user does not insert any coins, the machine will remain in its current state. For both the "drink" and "drink + change" states, the machine will have appropriate outputs and always transition back to the "idle" state, regardless of the current inputs.

All of the above information is represented in a transition diagram below, as well as more information about how the states, inputs, and outputs are to be encoded with bits. Your task is to create the necessary K-maps for the state machine described here, use those K-maps to produce simplified Boolean expressions for the state machine, and then build the state machine in CircuitSim.

3.7.2 Binary Reduced State Machine Diagram

The state machine transition diagram below lays out this scenario in full:


Outputs: No drink and no change
Outputs: No drink and no change
Nothing 00
Idle 00
Nickel 01
5¢ 01
Nothing 00
Nickel 01
Dime 10
Dime 10
Drink 10
Drink + change 11
Outputs: Drink and no change
Outputs: Drink and change

3.7.3 Quick review of Binary Reduced State Machines!

We assign a different binary number to each state of our state machine so that we can use a set of bits to represent the current state. We have four states in this case, so we will need two bits to represent our states. We will assign binary numbers to our states as follows:

- State 00 refers to the "idle" state
- State 01 refers to the "5¢" state
- State 10 refers to the "drink" state
- State 11 refers to the "drink + change" state

Our state machine also has two input bits to represent whether the user is inserting a nickel, a dime, or nothing at all into the machine. These possibilities are encoded as follows:

- Input 00 refers to inserting nothing
- Input 01 refers to inserting a nickel
- Input 10 refers to inserting a dime
- Input 11 is an invalid input – assume it can never occur

Finally, our state machine has two output bits to represent whether it is dispensing a drink and/or returning change in the current state. These are:

- Output D = drink (1 = drink, 0 = no drink)
- Output C = change (1 = change, 0 = no change)

3.7.4 K-Maps

First, produce the K-maps for the state transition diagram above on the provided spreadsheet named `kmap.xlsx`. Use the K-maps to produce the reduced Boolean expressions for the state machine.

The inputs for each K-map are:

$$\begin{aligned} S0 &= \text{Current state least significant bit} \\ S1 &= \text{Current state most significant bit} \\ M0 &= \text{Input bit least significant bit} \\ M1 &= \text{Input bit most significant bit} \end{aligned}$$

The outputs to make K-maps for are:

$$\begin{aligned} N0 &= \text{Next State least significant bit} \\ N1 &= \text{Next State most significant bit} \\ D &= \text{Drink} \\ C &= \text{change} \end{aligned}$$

Please Note: This State Machine is a Moore State Machine, meaning that the output values are determined solely by the current state (you should not use the N_1 and N_0 outputs or the M_1 and M_0 inputs for determining the values for Drink (D) and Change (C)).

You will fill out one K-map per output and one per next state bit for a total of 4 K-maps (D, C, $N0$, $N1$). The respective K-maps are located in the `kmap.xlsx` file. Your K-map must give the best solution of groupings possible to receive full credit. This means you must select the optimal values for any don't cares (if applicable) in your K-maps to do this. It may be helpful to check with others on Ed Discussion to see if your circuit is optimal. In order to do this without giving away your answer you may share the number of AND and OR gates used. The final total number is enough. Try not to give away how many gates you used for each step, as it could give away how your K-maps are done.

IMPORTANT: The K-maps will be autograded. Because of this, there are a set of restrictions to how you must fill your K-maps to ensure you get full credit:

- When you fill the row and column headers for your K-maps, you may only use the following variable names: $S0$, $S1$, $M0$, and $M1$. To negate a variable, you must use an apostrophe. Two adjacent variables with nothing in between are interpreted as an AND.

  Example label: `S0'M0`

- When writing the Boolean expressions resulted from your K-map groupings, you must use the same rules as the previous bullet point, but also use "+" for OR (no spaces).

  Example grouping: For the Boolean expression `(NOT S0) OR (S1 AND M0)`, write `S0'+S1M0`

- When filling in the cells of your K-map table, you must use `0`, `1`, and `X`.

3.7.5 Binary Reduced Vending Machine

Implement this circuit in the "Binary Reduced SM" circuit of the provided `fsm.sim` file. You will lose points if your circuit does not correspond to your K-map or if your circuit is not minimal. You should use only the minimal components possible to implement the state machine.

HINT: We recommend you make a truth table for the state machine to help organize the logic, and then transfer your answers to the K-maps. We've provided `truthtable.xlsx` to help with this.

Note: You are not required to complete `truthtable.xlsx` or submit it anywhere; it is only provided for convenience when making your K-maps.

The local autograder (`java - jar hw02-tester.jar`) does not check `kmap.xlsx`. You will have to submit on Gradescope to see your grade for the K-Map.

Deliverables: `kmap.xlsx` and `fsm.sim`.

3.8 Running the Autograder

To run the autograder locally, type the following command into your terminal while in the Homework 2 directory:

```
java -jar hw02-tester.jar
```

Make sure all the tests have been passed. Keep in mind that even if you get full credit from the autograder, we reserve the right to test for more cases.

3.8.1 Important Autograder Notes:

1. CircuitSim does not autosave your work. Save your work often and use version control such as Git to avoid losing your work and having to start over.
2. The local autograder does not check `kmap.xlsx`. It checks all of the other files. In order to ensure that your answers in `kmap.xlsx` are correct, you will have to submit on Gradescope.
3. There are some errors that may inexplicably occur in the process of running the autograder (e.g., `ArrayIndexOutOfBoundsException` or `Error creating circuit for circuit '...'`). If any of these errors prevent the tests from running, try running the autograder again until the tests run.
4. The autograder may not reflect your final grade on the assignment. We reserve the right to run additional tests during grading.

4 Deliverables

Please upload the following files onto the assignment on Gradescope:

1.	`gates.sim`	`NOR, AND`
2.	`sumofproducts.sim`	`Sum of Products`
3.	`demorgan.sim`	`Provided, Student`
4.	`plexers.sim`	`MUX, Sign Evaluation`
5.	`alu.sim`	`1-Bit Adder, 8-Bit Adder, Basic ALU, Intermediate ALU`
6.	`latches.sim`	`RS Latch, Gated D-Latch, D Flip-Flop, Register, Memory`
7.	`kmap.xlsx`	`K-Map Excel Sheet`
8.	`fsm.sim`	`Binary Reduced State Machine`

5 Rules and Regulations

1. Please read the assignment in its entirety before asking questions.
2. Please start assignments early, and ask for help early. Do not email us the night the assignment is due with questions.
3. If you find any problems with the assignment, please report them to the TA team. Announcements will be posted if the assignment changes.
4. You are responsible for turning in assignments on time. This includes allowing for unforeseen circumstances. If you have an emergency please reach out to your instructor and the head TAs **IN ADVANCE** of the due date with documentation (i.e. note from the dean, doctor's note, etc).
5. You are responsible for ensuring that what you turned in is what you meant to turn in. No excuses if you submit the wrong files, what you turn in is what we grade. In addition, your assignment must be turned in via Gradescope. Email submissions will not be accepted.
6. See the syllabus for information regarding late submissions; any penalties associated with unexcused late submissions are non-negotiable.
7. We reserve the right to add more test cases to the auto grader. Full credit is awarded for wholly correct implementations.

5.1 Academic Misconduct

Academic misconduct is taken very seriously in this class. Quizzes, timed labs and the final examination are individual work. Homework assignments will be examined using cheat detection programs to find evidence of unauthorized collaboration.

You are expressly forbidden to supply a copy of your homework to another student. If you supply a copy of your homework to another student and they are charged with copying, you will also be charged. This includes storing your code on any platform which would allow other parties to it (public repositories, pastebin, etc). If you would like to use version control, use a private repository on github.gatech.edu

Homework collaboration is limited to high-level collaboration. Each individual programming assignment should be coded by you. You may work with others, but each student should be turning in their own version of the assignment.

High-level collaboration means that you may discuss design points and concepts relevant to the homework with your peers, share algorithms and pseudo-code, as well as help each other debug code. What you shouldn't be doing, however, is pair programming where you collaborate with each other on a single instance of the code, or providing other students any part of your code.

Submissions that are essentially identical will receive a zero and will be sent to the Dean of Students' Office of Academic Integrity. Submissions that are copies that have been superficially modified to conceal that they are copies are also considered unauthorized collaboration.

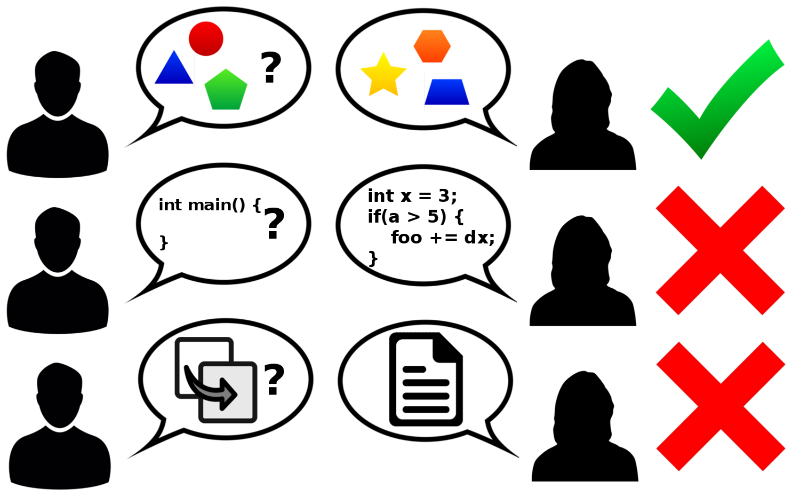


Figure 1: Collaboration rules, explained colorfully