CS 2110 Homework 3
Sequential Logic & State Machines

Prabhav Gupta, Isaac Kim, Saloni Bedi, Richard So
Henry Bui, Annelise Lloyd, Justin Almas

Spring 2024

Contents

1 Overview

1.1 Purpose

The purpose of this assignment is to practice the low-level concepts we have learned, from sequential logic to state machines in CircuitSim. Part 1 focuses on implementing sequential logic circuits like RS latches, Gated-D latches, D-Flip Flops which you will use to build a register from the ground up and eventually build a simple memory circuit as well. Part 2 focuses on using a state machine diagram and scenario to build and simplify KMaps and then build a Binary Reduced State Machine in CircuitSim. Finally, in Part 3, you will use your understanding of CircuitSim and these various circuitry concepts to build some of the key components of the Datapath.

Objectives:

1. To understand how a register circuit works
2. To learn how to make a state machine
3. To understand K-maps and simplification
4. To see how state machines can integrate with computer processing

1.2 Task

There are three parts to this assignment.

1. First, you will build sequential logic components in CircuitSim from the ground up.
2. Second, you will complete a K-Map to simplify your binary reduced state machine circuit, and implement the simplified circuit.
3. Third, you will implement parts of the datapath we have provided for you.

Before you start the assignment, be sure to read the rest of the PDF for more detailed instructions and hints.

1.3 Criteria

You must utilize the provided CS 2110 version of CircuitSim, which is available via the JAR on Canvas. Utilizing other versions of CircuitSim is strictly prohibited and may result in damaged or corrupted files.

You will submit four files to Gradescope:

- `latches.sim`
- `kmap.xlsx`
- `fsm.sim`
- `datapath.sim`

Your grade is based on: 1) the correct outputs from your circuits; and 2) not using any banned components. For part 2 (binary reduced state machine), you will lose points if your circuit does not correspond to your K-Map or if your circuit is not minimal. The grade you see on Gradescope may not be the final grade you receive, as we may run additional tests on your submission.

You may submit your code to Gradescope as many times as you like until the deadline. We will grade your last active submission. We have also provided a local checker that you can test your code with. Please submit your code to Gradescope at least once prior to the deadline, to ensure you are not encountering any issues submitting at the last minute.

2 Instructions

Part 1: For this part, you will be building your own register and simple memory from the ground up.

- Implement your circuits in the "`latches.sim`" file

Part 2: Given the a state diagram, you will be minimizing the logic by using K-Maps.

- Fill out the K-Maps located in the spreadsheet named "`kmap.xlsx`"
- The binary reduced circuit will be implemented in the "Binary Reduced SM" subcircuit of the "`fsm.sim`" file

Part 3: Finish connecting the datapath we have partially provided you

- Connect the datapath.
- The datapath will be implemented in the "Datapath" subcircuit of the "`datapath.sim`" file

Do not change/delete any of the input/output pins.
(Though you may change the actual value of the input pins)
There is more to the assignment in the next pages.

2.1 Part 1

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.

2.1.1 RS Latch

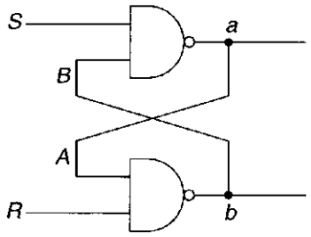
You will start by building a RS latch using NAND gates, as described in your textbook. The RS Latch is the basic circuit for sequential logic. It stores one bit of information, and it has 3 important states:

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.

Once you set the bit you wish to store, change back to the quiescent state to keep that value stored.

Notice that the circuit has two output pins; one is the bit the latch is currently storing, and the other is the opposite of that bit.

Note: In order for the RS Latch to work properly, you must not set both R and S to 0 at the same time.



- Build your circuit in the "RS Latch" subcircuit in the "`latches.sim`" file.

2.1.2 Gated D Latch

Using your RS latch subcircuit, implement a Gated D Latch as described in the textbook.

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 subcircuit only has one output pin, so you should disregard the inverse output of your RS Latch.

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

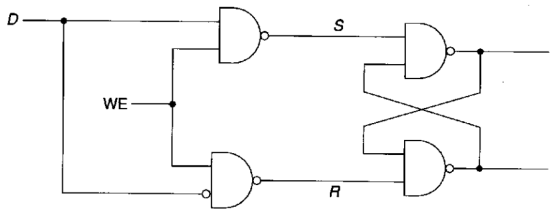


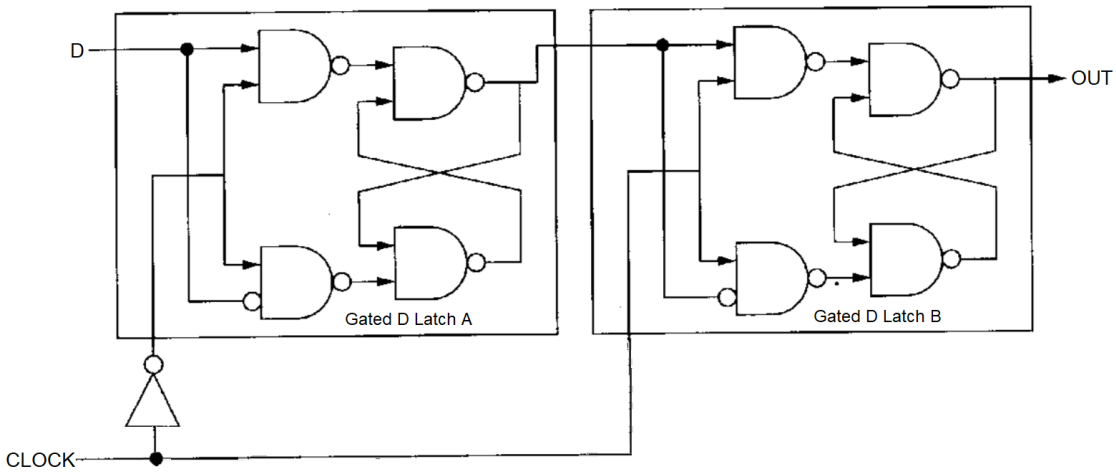
Figure 3.19 A gated D latch

2.1.3 D Flip-Flop

Using the Gated D Latch circuit you built, create a 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.

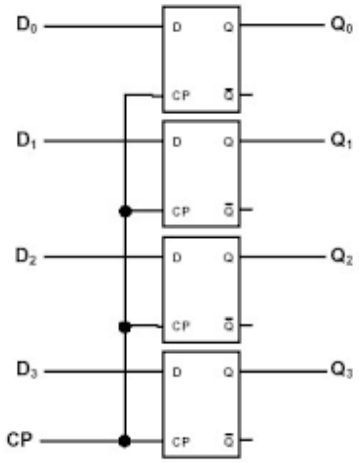
Your D flip-flop output should be able to change on the **rising edge**, which means that the state of the D Flip-Flop should only be able to change at the exact instant the clock goes from 0 to 1.



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

2.1.4 Register

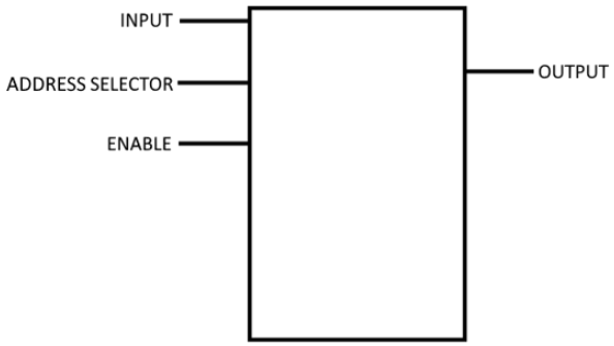
Using the D Flip-Flop you just created, build a 4-bit Register. Your register should also use edge-triggered logic. The value of the register should change on the rising edge.



- This circuit will be implemented in the "Register" subcircuit in the "`latches.sim`" file.
- You *should* use your previous D flip-flop subcircuit.

2.1.5 Memory

Using the Gated D Latches you just created, 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.

2.2 Part 2

2.2.1 Scenario

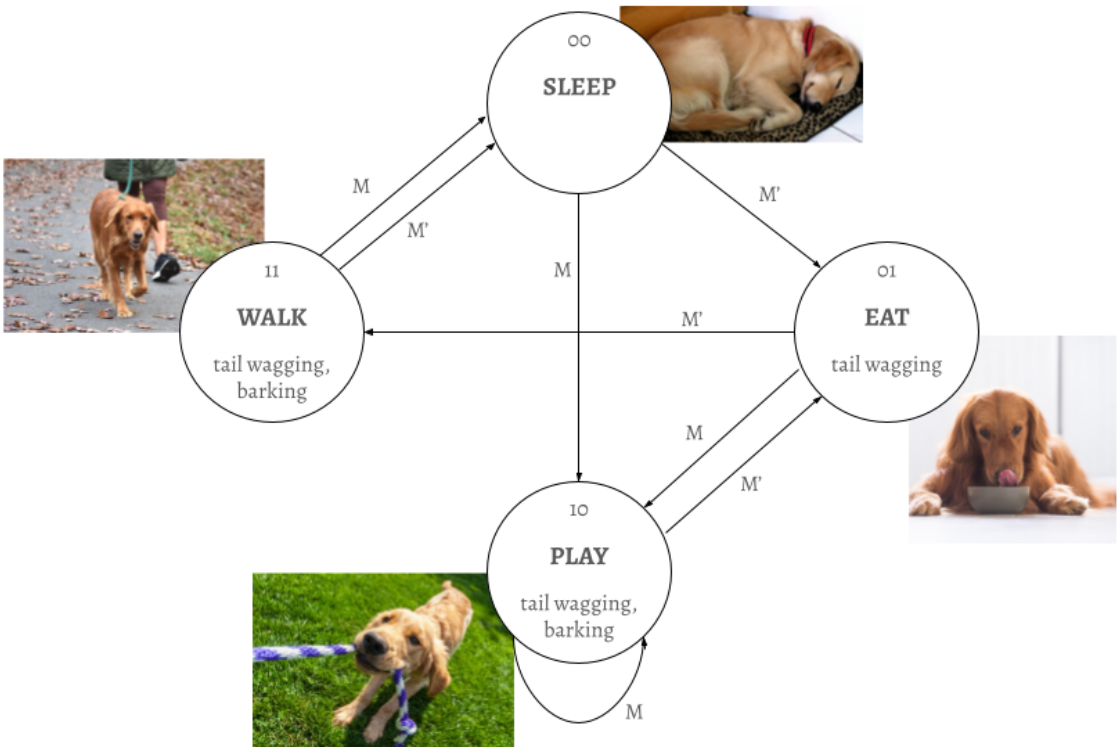
Meet Pablo, your soon to be favorite canine companion. With a fur coat softer than snow and a tail that can't stop wagging, he's a dog that can cheer you up at any time. Whether he's playing fetch or barking away at a squirrel, Pablo is proof that life is better with a furry friend by your side.

A quick observation of Pablo reveals that he primarily exists in four possible states: a Sleep state, an Eat state, a Play state, and a Walk state! At each of these states, he can be seen exhibiting some combination of two behaviors: tail wagging and barking – this is the state's output.

Pablo begins in the Sleep state. After getting enough rest, he can choose to proceed to the Eat state if he's hungry, or the Play state. From the Play state, Pablo can either stay at the Play state or, if he's had his fill of playing, can continue to the Eat state. After eating, he can go for a walk or cycle back to the Play state. Following his time in the Walk state, Pablo will be too tired for any more activity and unconditionally return to the Sleep state.

2.2.2 Binary Reduced State Machine Diagram

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



2.2.3 Quick review of Binary Reduced State Machines!

Each state corresponds with a binary number in a binary reduced state machine. The binary number represents the state number as a decimal translation. What does this mean? In a binary reduced state machine, we convert the binary number into a decimal number to find the state number. So 00 = state 0 and 11 = state 3. Each state contains a transition to the next state based on an input M. M represents M=1, while M' represents M=0. A transition on M means you move from a specified state to the state the transition points to when the M input is set to 1. The outputs are dependent on the current state.

- State 00 refers to the Sleep State
- State 01 refers to the Eat State
- State 10 refers to the Play State
- State 11 refers to the Walk State

2.2.4 KMAPS

- 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:

  - $S0$ = Current state least significant bit
  - $S1$ = Current state most significant bit
  - M = Input bit

  The outputs to make K-maps for are:

  - $N0$ = Next State least significant bit
  - $N1$ = Next State most significant bit
  - T = Tail Wagging
  - B = Barking

  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 input for determining the values for Tail Wagging (T), Barking (B)).

  - You will fill out one K-map per output and one per next state bit for a total of 4 K-maps (T, B, $N0$, $N1$). 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: $S0$, $S1$, and M. To negate a variable, you must use an apostrophe. Two adjacent variables with nothing in between are interpreted as an AND.
      Example label: `S0'M`

* 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 M)`,
  write `S0'+S1M`
* When filling in the cells of your K-map table, you must use `0`, `1`, and `X`.

2.2.5 Binary Reduced Dog

- Implement this circuit in the "Binary Reduced SM" subcircuit 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.

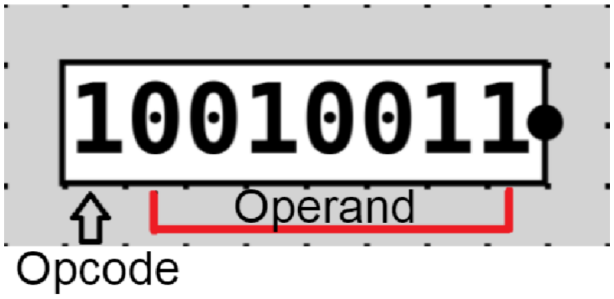
  Note: You are not required to complete `truthtable.xlsx` or submit it anywhere; it is only provided for convenience when making your K-maps.

2.3 Part 3

For this part of the assignment you will be completing parts of a processor datapath. A datapath is essentially a collection of components that can perform operations. By turning on and off certain inputs of the datapath, your datapath (upon completion) will be able to process instructions. You can use the provided microcontroller that uses the datapath to handle the instructions (apart from changing the clock).

Please Note: Much of the terminology in this section of the PDF has been heavily simplified to provide a basic overview that will allow you to create the datapath without an intricate understanding of computer processing.

2.3.1 What is an instruction?



Above is an example of an instruction.

- The arrow points to the "Opcode" (operation code). This defines what the operation is. For this datapath, we have simplified the Opcode to the left most bit (bit 7) for a possibility 2 operations. These operations will be either an AND or ADD operation. An AND operation will occur when the Opcode is a 0. An ADD operation will occur when the Opcode is a 1.
- The rest of the bits represent the "Operand", which represents the parameters of our instruction. This has also been simplified to a single value represented by 7 bits (bits 0-6). This will be the value that will be ADDed or ANDed with the value currently in the Datapath (This will be held in the general purpose register, but more on this later). The value can be determined simply by reading the 6 bits as an **unsigned** binary number.

- With this information, we can now decipher the above example. The 1 for the Opcode (the leftmost bit) tells us that this is an ADD instruction. Reading the Operand (bits 0-6) as an unsigned binary number tells us the value is 19. Therefore, putting the two together completes our instruction as ADD 19. When executing this instruction, it will take the value of 19 and adds it to our current value.

2.3.2 Components of the Datapath

- The register labeled 'PC' is a simplified version of a "Program Counter". The Program Counter stores a value (referred to as the "address") corresponding to the **next** instruction. For this datapath, the PC register will store a possible 4 addresses represented in two digit binary: 00, 01, 10, 11. Furthermore, these values correspond to the 4 possible instructions in the INPUTS & OUTPUTS section: 00 to INSTR0, 01 to INSTR1, 10 to INSTR2, 11 to INSTR3. Therefore, say the value of the PC is 10 in binary, then the current instruction being processed by the datapath would be INSTR1, since INSTR1 corresponds to the address 01, and the PC holds the address of the next instruction.

  Note: When no instructions are being processed by the datapath (before beginning any clock cycles) the value of the PC will be 00. This should makes sense because the next instruction to be processed would be instruction INSTR0.

- The register labeled 'IR' refers to the "Instruction Register". This register will store the value of the **current** instruction that we are executing. So when the INSTR1 instruction is being processed, the IR will display the value of that whole instruction. Because we are considering the whole instruction, the IR should not differentiate between the Opcode or Operand, rather it should treat the bits as one 8-bit value.

- The register labeled 'REG' is our general-purpose register. This register will hold the result of the operations we run. This register will allow you to observe whether your instructions are being processed by the datapath as intended. Remember that instructions will be applied to the value currently in the register. This means that if the current value in REG is a 5, and our next instruction is ADD 10, then REG should display a value of 15 when the instruction is finished executing.

- The component labeled ALU is similar to the one you implemented in HW02. This will handle the actual calculation of the instruction. The input "ALUK" (short for ALU Control) will determine which calculation will occur. As stated before, when the Opcode of an instruction is a 0, the operation should be an AND operation. If the Opcode is 1, the operation should be an ADD operation. Remember that instructions will be applied to the value currently in REG. This should help in deciding which inputs should be sent into the ALU when operations should occur.

- The section labeled as Microcontroller is what actually will be controlling this datapath. The display on the datapath circuit may seem complex, but it is quite simply a Finite State Machine (You implemented a binary-encoded state machine for part 2). We have added more to the Microcontroller component to allow students to manually test their datapath. Upon completion, the Microcontroller will have the datapath cycle through different states, turning inputs on and off, allowing instructions to be interpreted and executed autonomously (apart from controlling the clock).

Please Note: Values displayed on registers are in **hex** not binary.

2.3.3 What do I need to do?

The datapath provided is not complete. Your job is to determine how to connect the datapath so that instructions can be properly processed. This will include setting up the datapath to be able to "Fetch", "Decode", and "Execute" instructions, as well as setting up the control signals (the outputs of the microcontroller) to align with each step.

Fetch

In order to actually execute an instruction, we must first "Fetch" the instruction and store it in our IR. To be able to fetch an instruction, our datapath must be able to do the following:

1. When `LD.IR = 1` on a rising clock edge, use the value in the PC to load the correct instruction to the IR. For example, if `PC = 01`, we should load `inst1` into the IR. This is the actual "fetching" of the instruction.

2. When `PCINC = 1` on a rising clock edge, we should increment the value of the PC. This is so that we can get the next instruction the next time we fetch (instead of the same one over and over again).

Quick hints to Fetch:

- A tunnel labeled PC is currently attached to a MUX with tunnels to our four instructions. What value should we assign to this tunnel to retrieve the correct instruction? (No, it's not a trick question.)
- What component can you use to increment the value of a number? Hint: incrementing is just addition.

Decode

Once we have "Fetched" our instruction and stored it in our IR, we must "Decode" our instruction. This essentially means to determine what the instruction will be doing, so we may prepare our datapath accordingly. To do this we must:

1. Take the value in the IR, and set the values of the Opcode and Operand accordingly.
2. That is all! No control signals need to be set as decode simply sets up the Datapath for the following states.

Quick hints to Decode:

- Which bits of the instruction tells us the operation occurring? This is the opcode.
- Which bits of the instruction tells us what is the value being used in the operation? This is the operand.

Execute

Once we have "Decoded" our instruction, we need to actually "Execute" this instruction for anything to actually happen. In this simplified datapath, this will mean either executing an AND operation or ADD operation and appropriately update the value of "REG". For this to work:

1. Send the inputs of the operation to the ALU, remember that operations will be applied to the value currently in the REG.
2. Connect `ALUK` to the appropriate location. Remember that `ALUK` stands for "ALU control" and is used to determine the operation performed by the ALU. Remember, when the Opcode is a 0, the instruction should perform an AND operation. When the Opcode is a 1, the instruction should perform an ADD operation.
3. When `LD.REG = 1` on a rising clock edge, we should set `REG` to be equal to the resulting value from the ALU.

Quick hints to Execute:

- What values will be used as the inputs of our operation?
- What component of our datapath can perform these operations?
- What can we use to determine the operation being performed?

2.3.4 How do we actually execute an instruction?

NOTE: You should only use the state machine for your datapath once you have a completed datapath to test.

When you believe you have set up Fetch, Decode, and Execute properly. We can now test our datapath by actually executing some instructions.

1. Set up some instructions. Change the values of the instructions, so that you can see how the datapath processes them. Remember how instructions are set up. Keep in mind: It is probably a good idea to decide what values should be displayed throughout the processing, so you can verify your datapath is working correctly.

2. You can test your inputs by seting the CONTROL MODE input to 0. This will make the datapath rely on the provided state machine circuit which will set the control signals for you. You can then click the clock and the state machine will process your instructions.

3. You can use "Manual" inputs to control your datapath.

   To enter manual mode, set the CONTROL MODE input to 1. The datapath will now rely on inputs (control signals) you select to control instruction processing. You can now manually Fetch, Decode, and Execute (in this order) to process the instruction:

   (a) Fetch:

   On a rising clock edge, set `LD.IR = 1` and `PCINC = 1`. Remember, setting `LD.IR = 1` should load one of the four instructions into the IR depending on the current PC value. Setting `PCINC = 1` should increment the value of the PC.

   (b) Decode:

   Nothing needs to be done here. Again, this step is here to set up the actions of the future states.

   (c) Execute:

   On a rising clock edge, set `LD.REG = 1`, and if an ADD operation should occur, set `ALUK = 1`. Remember, this should insert the result of the ALU operation into `REG`.

4. If Fetch, Decode, and Execute are working properly (and your state machine if you are using one), you should be able to see the appropriate values across the registers and outputs.

2.4 Moving and Reorienting Components

Please make sure to follow the text that says to not delete anything that's already provided, change the general orientation of components, and change the general direction of components. Doing any of the above would result in the provided "Microcontroller (One Hot)", "Microcontroller" subcircuit (that are already built) to be connected improperly and cause short circuits. Make sure to not change the orientation or layout of the input or output pins. **Basically, please don't mess with the provided circuits (i.e. "Microcontroller (One Hot)", "Microcontroller", "ALU") or you might get errors.**

3 Testing

To test your **circuits** locally, navigate to the directory with the files for this homework and run the tester JAR file with

```
java -jar hw03-tester.jar
```

Note 1: 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, please submit on Gradescope.

Note 2: There are some errors that may inexplicably occur in the process of running the local autograder (e.g., `Error creating subcircuit for circuit '...'`). If any of these errors prevent the tests from running, try running the autograder again until the tests run.

4 Deliverables

Submit the following files to the "Homework 3: State Machines" assignment on Gradescope:

- `latches.sim`
- `fsm.sim`
- `kmap.xlsx`
- `datapath.sim`

Note: The autograder may not reflect your final grade on the assignment. We reserve the right to run additional tests during grading.

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.

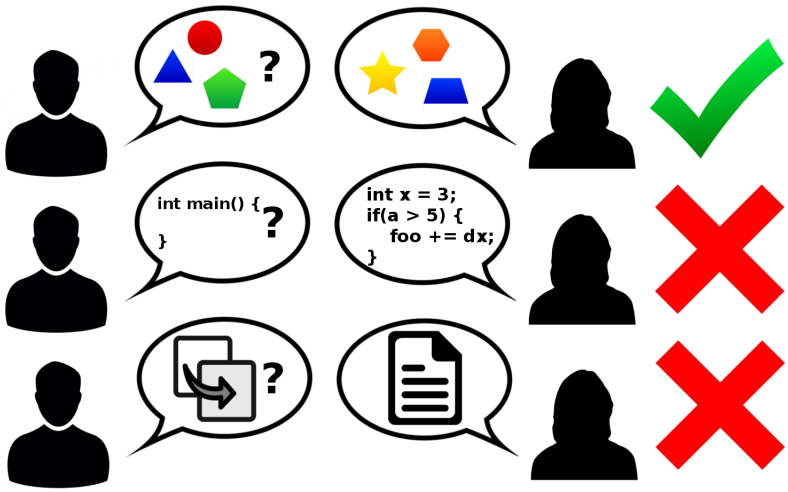


Figure 1: Collaboration rules, explained colorfully