CS 2110 Homework 3
Sequential Logic & State Machines

Jason Ng, Udit Subramanya, John Ever, Jessi Chen

Summer 2022

Contents

1 Overview

1.1 Purpose

The purpose of this assignment is to practice implementing sequential logic circuits – from an RS Latch up to a Finite State Machine, using CircuitSim. You will build a register from the ground up. Then you will build a One Hot State Machine circuit, based on a provided state machine diagram (see Part 2 below). Then you will build a reduced state machine, using a K-Map to simplify your logic.

You will also build parts of a processor datapath to see how the finite state machine and other components integrate into a full system.

Objectives:

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

1.2 Task

There are four parts to this assignment.

1. First, you will build a register in CircuitSim from the ground up.
2. Second, you will connect the datapath we have partially provided for you.
3. Third, you will implement a one hot state machine Circuit in CircuitSim, based on the state transition diagram found below.
4. Fourth, you will complete a K-Map to simplify your state machine circuit, and implement the reduced state machine.

Please read the rest of this document for more detailed instructions and hints.

1.3 Criteria

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

Your grade is based on: 1) the correct outputs from your circuits; and 2) not using any banned components. For part 4 (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 will submit three files to Gradescope: `latches.sim`; `fsm.sim`; `kmap.xlsx`.

You may submit your code to Gradescope as many times as you like until the deadline. We will grade your last 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 of the assignment you will build your own register from the ground up.

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

Part 2: Finish connecting the datapath we have partially provided you

- Connect the datapath (You should only need the Wiring & Arithmetic tabs of CircuitSim).
- The datapath will be implemented in the "`fsm.sim`" file

Part 3: Given a simple state diagram, you will build a state machine in CircuitSim using the "one-hot" style of building state machines.

- The circuit will be implemented in the "One Hot FSM" subcircuit of the "`fsm.sim`" file

Part 4: Given the same state diagram from part 3, you will be minimizing the logic by using K-Maps.

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

Do not change/delete any of the input/output pins.
(Though you may change the actual value of the input pins)

2.1 Part 1

For this part of the assignment you will build your own register from the ground up. For more information about each subcircuit refer to your textbook.

2.1.1 RS Latch

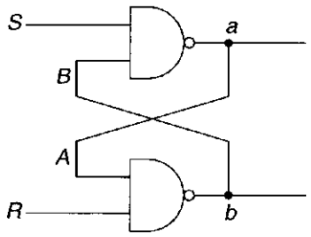
You will start by building a RS latch using NAND gates, as described in your textbook. The RS Latch is the basic circuit for sequential logic. It stores one bit of information, and it has 3 important states:

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 on 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 are not allowed to use the built-in SR Flip-Flop in CircuitSim to build this circuit

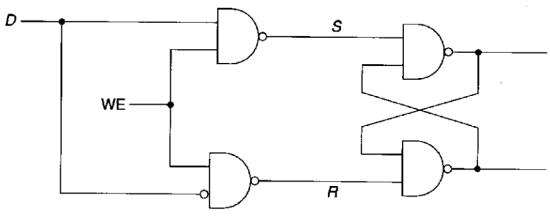


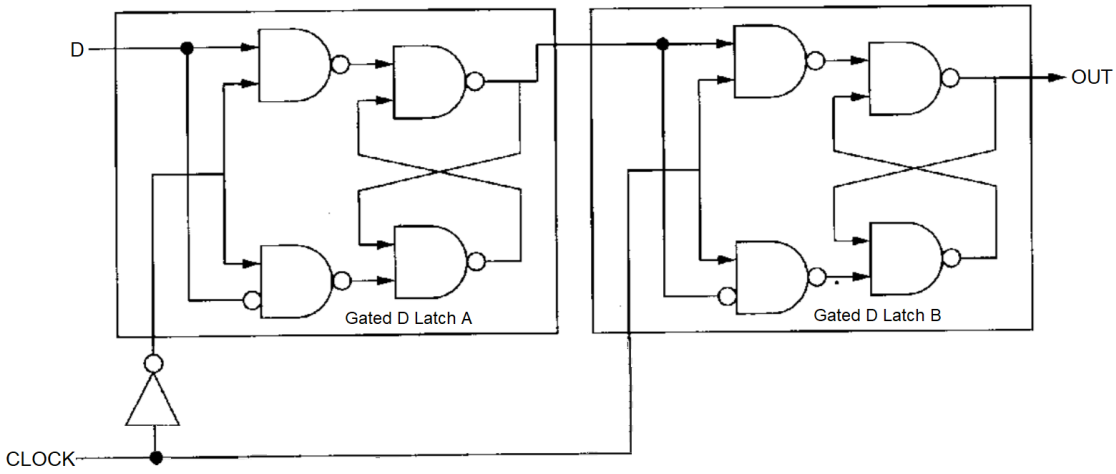
Figure 3.19 A gated D latch

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.

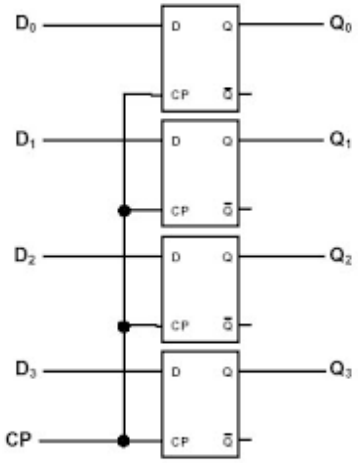
Your D flip-flop output should be able to change on the **rising edge**, which means that the state of the D Flip-Flop should only be able to change at the exact instant the clock goes from 0 to 1.



- Implement this circuit in the "D Flip-Flop" subcircuit in the "`latches.sim`" file.

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.



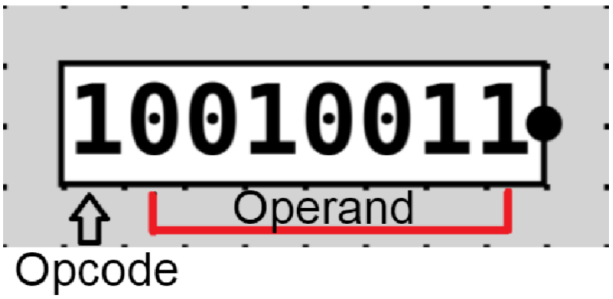
- This circuit will be implemented in the "Register" subcircuit in the "`latches.sim`" file

2.2 Part 2

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 will then build a microcontroller that will be able to control this datapath to handle said instructions by itself (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.

2.2.1 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 add it to our current value.

2.2.2 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, which will be implemented in a future section. We have added more to the Microcontroller component to allow students to manually test their datapath without building the other circuits in the FSM file, or to switch between using the One Hot or Binary Reduced State Machines, but more on this later. 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.

2.2.3 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?

2.2.4 How do we actually execute an instruction?

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. For now, you will need to use "Manual" inputs to control your datapath. Later, you will implement the Microcontroller: a Finite State Machine that serves as the brain of the datapath and can take of this for you.

   To enter manual mode, set the CONTROL MODE input to 10 or 11 (does not matter which). 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`.

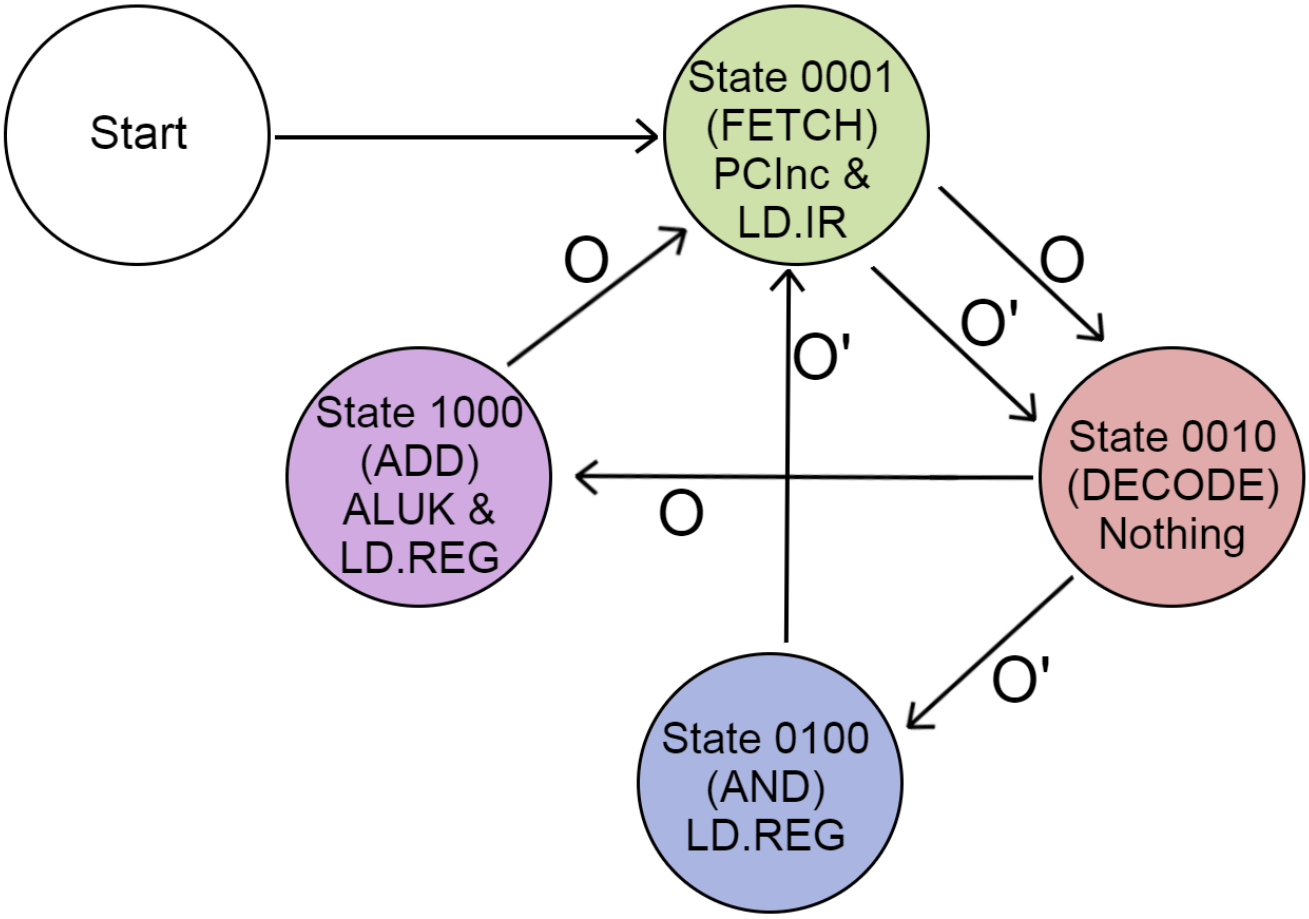
3. If Fetch, Decode, and Execute are working properly (and your state machine if you are using one), you should be able to see the appropriate values across the registers and outputs.

2.2.5 Comprehensive Test

If you are getting this error and you only just finished part 2, don't worry! This is a comprehensive test that checks if your datapath works assuming you've already implemented the One Hot and Binary Reduced Microcontrollers (Part 3 and 4). If this is your last test after Part 2, you're good to move on to part 3!

2.3 Part 3

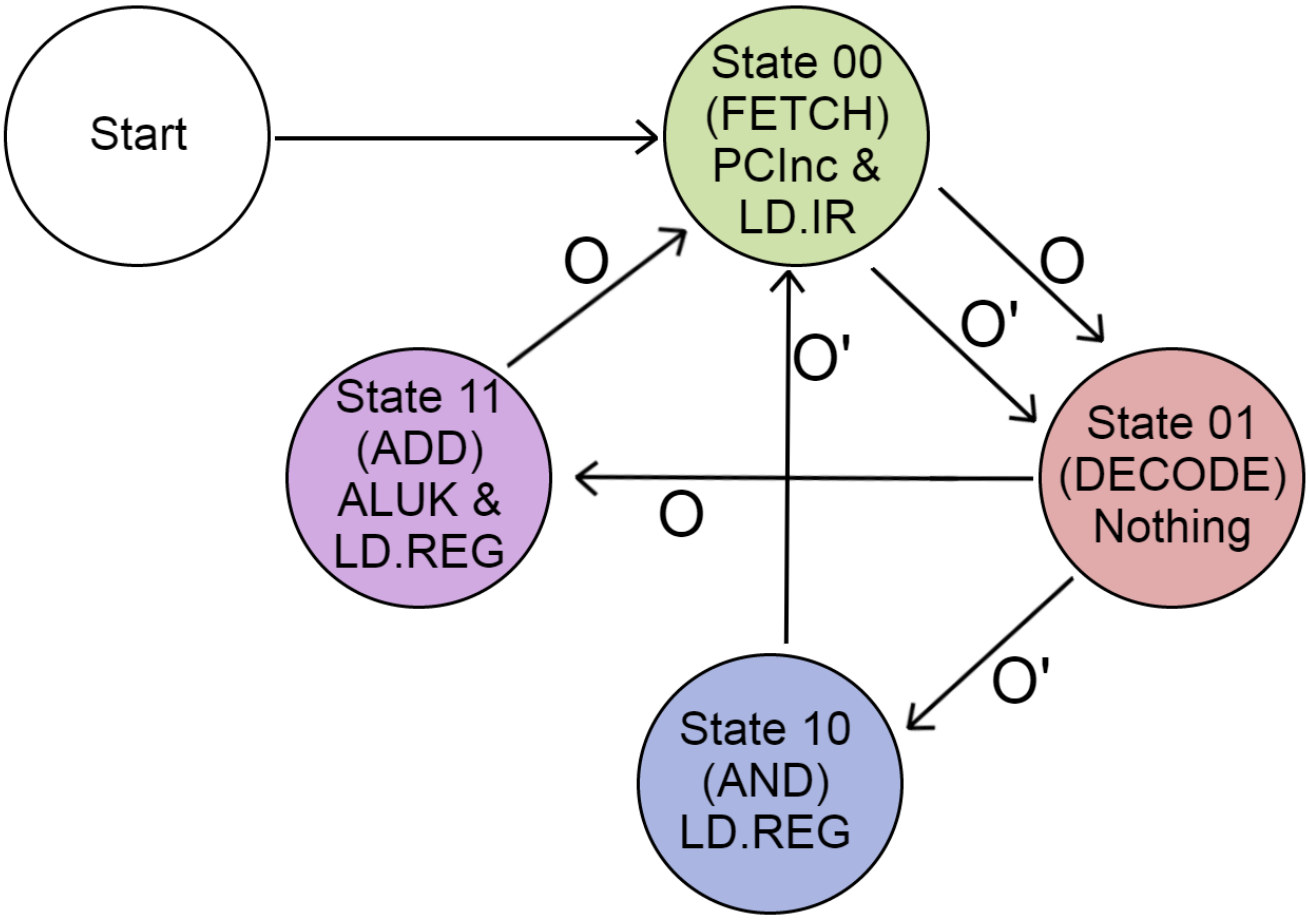
For this part of the assignment you are given the state machine transition diagram of a microcontroller that will handle fetching, decoding, and executing instructions in your datapath. The microcontroller has a few states it must go through in order to execute an instruction. Additionally, it has one input called O which can be equal to a value of 0 or 1, which corresponds to the opcode of the current instruction. Note that in CircuitSim, this input is called OP so as to not confuse the letter O with the number 0. This microcontroller has four outputs (`PCINC`, `LD.IR`, `ALUK`, and `LD.REG`) that affect the various components on the datapath you have made.



- You will be implementing this state transition diagram as a circuit using the one-hot style. Remember for one-hot you will have a register with the number of bits being the number of states of the FSM. Each bit corresponds to a state, and you are in that state if the corresponding bit is a 1. Since you can only be in one state at a time, exactly one of the bits can be 1 at each time (except when the machine starts up, when all the bits will be 0).
- You must implement this using one-hot. A template file `fsm.sim` has been given to you. Implement the state machine in the provided "one-hot state machine" subcircuit.
- Note that there are 3 inputs to the "Microcontroller (One Hot)" subcircuit: CLOCK, OP, and RESET. The CLOCK input turning off and on repeatedly will be used to represent clock ticks in your circuit. The OP input corresponds to whether or not O is on, as in the diagram above (we would call the input O but it looks way too similar to the number 0 so we added the P in the input name, representing the Opcode). RESET resets the circuit (clears all the bits of your state register).

2.4 Part 4

Take a look at this state machine transition diagram:



- 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
  - O = Input button

  The outputs to make K-maps for are:

  - $N0$ = Next State least significant bit
  - $N1$ = Next State most significant bit
  - $PCINC, LD.IR, ALUK, LD.REG$

  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 OP input for determining the values for $PCINC, LD.IR, ALUK, LD.REG$).

  - You will fill out one K-map per output and one per next state bit for a total of 6 K-maps (PCINC, LD.IR, ALUK, and LD.REG, $N0$, $N1$). 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 O. Just a reminder that O is the same as OP in the actual circuit. When building the actual circuit, it becomes quite confusing whether the letter O is actually a number 0. However, for diagram and K-Map purposes, it is just the letter O. To negate a variable, you must use an apostrophe. Two adjacent variables with nothing in between are interpreted as an AND.
    Example label: $S0'O$
  * 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.
    Example grouping: For the Boolean expression `(NOT S0) OR (S1 AND O)`, write $S0'+S1O$
  * When filling in the cells of your K-map table, you must use `0`, `1`, and `X`.

- Implement this circuit in the "Microcontroller (Binary Reduced)" 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.
- In order to test your different state machines, there is a microcontroller on the right side of you datapath circuit that you can use to change which state machine is controlling the datapath. This microcontroller has 3 different modes:

  1. Microcontroller (One Hot), which will use your implementation of the One Hot state machine to set the control signals (PCINC, LD.IR, ALUK, and LD.REG). The microcontroller selector for this mode is 00.
  2. Microcontroller (Binary Reduced), which will use your implementation of the Binary Reduced state machine to set the control signals. The microcontroller selector for this mode is 01.
  3. Manual, which is a mode that we have provided so that you can manually set the control signals for the datapath. The microcontroller selector for this mode is 10 OR 11.

- **ALSO NOTE:** If you have implemented both your datapath **AND** your state machine correctly, then you should be able to go back to the datapath circuit, and see that it processes instructions as intended. (Check what control signals should be on throughout the cycle of an instruction, and see if your state machine sets those signals correctly.)

2.5 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" subcircuit (that is already built) to be connected improperly and cause short circuits. If you take a look at this subcircuit, you'll see that at the bottom side of the

"Microcontroller (One Hot)" and "Microcontroller (Binary Reduced)" subcircuits, that it is assumed the input OP will be to the *left* of the input RESET. If you move the input OP around in the respective subcircuits and OP ends up to the *right* of the input RESET, it *will* be reflected in this current subcircuit, causing OPCODE to be connected to RESET, and RESET to be connected to OPCODE. You can imagine a similar scenario in the case for when the general layout of PCINC, LD.IR, ALUK, and LD.REG is shuffled and not in that *exact* order.

3 Testing

To test your **circuits** locally, navigate to the directory with the latches.sim and fsm.sim files and run the tester JAR file with

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

Make sure to run the local autograder in a terminal in Docker.

4 Deliverables

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

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

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

5.1 General Rules

1. Starting with the assembly homeworks, any code you write must be meaningfully commented. You should comment your code in terms of the algorithm you are implementing; we all know what each line of code does.
2. Although you may ask TAs for clarification, you are ultimately responsible for what you submit. This means that (in the case of demos) you should come prepared to explain to the TA how any piece of code you submitted works, even if you copied it from the book or read about it on the internet.
3. Please read the assignment in its entirety before asking questions.
4. Please start assignments early, and ask for help early. Do not email us the night the assignment is due with questions.
5. If you find any problems with the assignment it would be greatly appreciated if you reported them to the author (which can be found at the top of the assignment). Announcements will be posted if the assignment changes.

5.2 Submission Conventions

1. All files you submit for assignments in this course should have your name at the top of the file as a comment for any source code file, and somewhere in the file, near the top, for other files unless otherwise noted.

2. When preparing your submission you may either submit the files individually to Canvas/Gradescope or you may submit an archive (zip or tar.gz only please) of the files. You can create an archive by right clicking on files and selecting the appropriate compress option on your system. Both ways (uploading raw files or an archive) are exactly equivalent, so choose whichever is most convenient for you.

3. Do not submit compiled files, that is .class files for Java code and .o files for C code. Only submit the files we ask for in the assignment.

4. Do not submit links to files. The autograder does not understand it, and we will not manually grade assignments submitted this way as it is easy to change the files after the submission period ends.

5.3 Submission Guidelines

1. You are responsible for turning in assignments on time. This includes allowing for unforeseen circumstances. If you have an emergency let us know **IN ADVANCE** of the due time supplying documentation (i.e. note from the dean, doctor's note, etc). Extensions will only be granted to those who contact us in advance of the deadline and no extensions will be made after the due date.

2. You are also responsible for ensuring that what you turned in is what you meant to turn in. After submitting you should be sure to download your submission into a brand new folder and test if it works. 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 Canvas/Gradescope. Under no circumstances whatsoever we will accept any email submission of an assignment. Note: if you were granted an extension you will still turn in the assignment over Canvas/Gradescope.

5.4 Syllabus Excerpt on Academic Misconduct

Academic misconduct is taken very seriously in this class. Quizzes, timed-labs and the final examination are individual work.

Homework assignments are collaborative, In addition many if not all homework assignments will be evaluated via demo or code review. During this evaluation, you will be expected to be able to explain every aspect of your submission. Homework assignments will also be examined using computer programs to find evidence of unauthorized collaboration.

What is unauthorized 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. 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.

You are expressly forbidden to supply a copy of your homework to another student via electronic means. This includes simply e-mailing it to them so they can look at it. If you supply an electronic 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 site which would allow other parties to obtain your code such as but not limited to public repositories (Github), pastebin, etc. If you would like to use version control, use github.gatech.edu

5.5 Is collaboration allowed?

Collaboration is allowed on a high level, meaning 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. Furthermore, sending an electronic copy of your homework to another student for them to look at and figure out what is wrong with their code is not an acceptable way to help them, because it is frequently the case that the recipient will simply modify the code and submit it as their own.

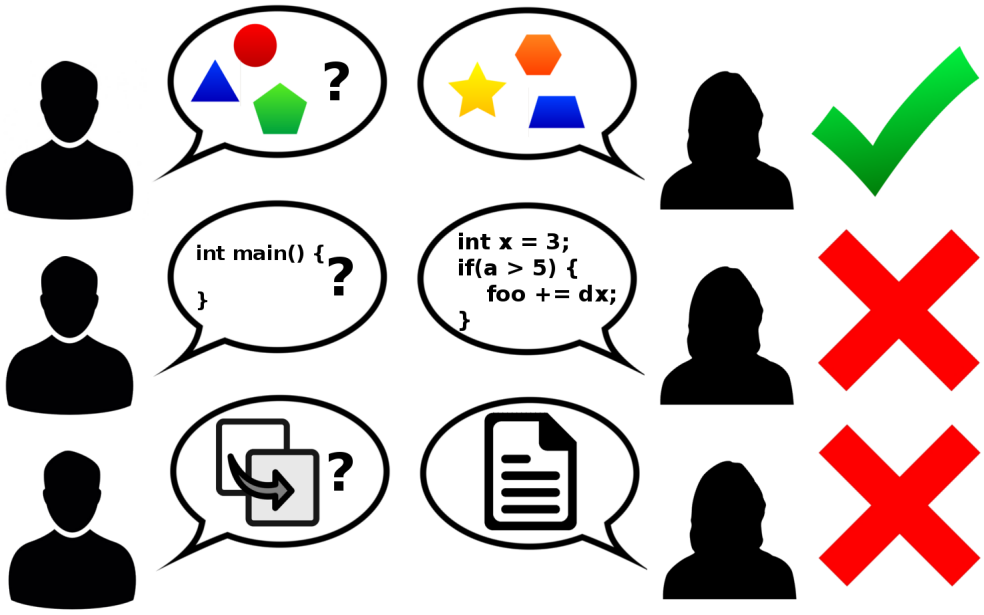


Figure 1: Collaboration rules, explained colorfully

6 Disclaimers

The following are trivial things that may lead to errors on the autograder, even though your circuit is *technically* correct.

1. Renaming or removing input pins may cause issues with the autograder because it's looking for input pins with a specific label.
2. You must use constant pins when necessary. Using any input pins for a scenario where a fixed constant value is required will lead to the autograder setting the input pin to 0 and thus cause issues for scenarios where it needed to be a constant 1.