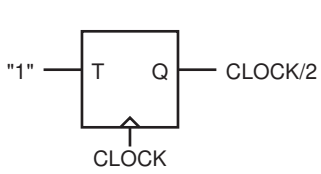
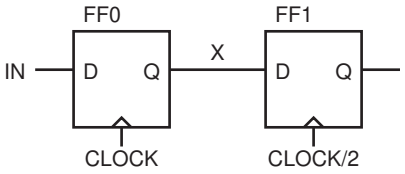
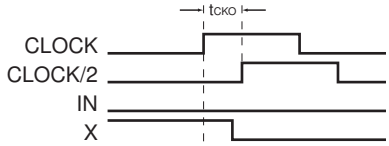
1. Synchronous Design Rules (35 pts)

1. Using separate clocks can produce unexpected behavior.

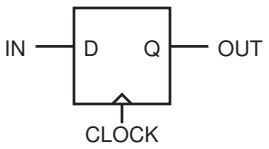


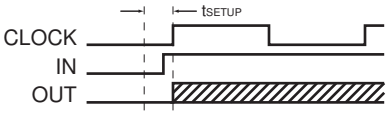




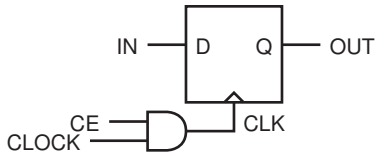
Here we expect FF1 to clock the high value of X (at the rising edge of CLOCK), but, since CLOCK/2 is delayed by t_{CKO}, X has already changed values.

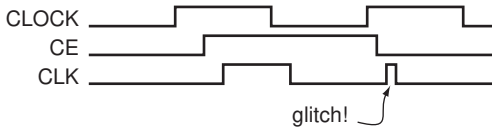
2. Unsynchronized inputs can violate setup and hold times.



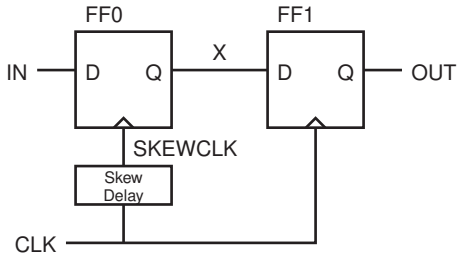


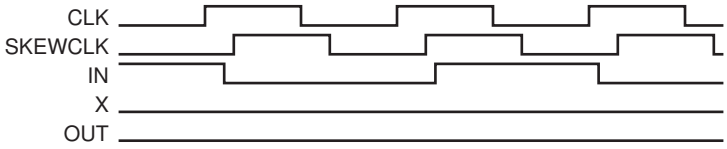
3. By gating the clock, you can introduce glitches. Any small error in the input signal could cause a clock pulse to occur and if the resulting pulse is too short, the circuit may become indeterminate.





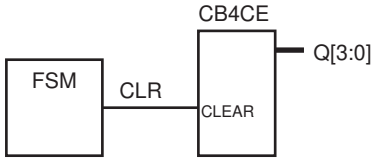
4. Clock skew is bad.

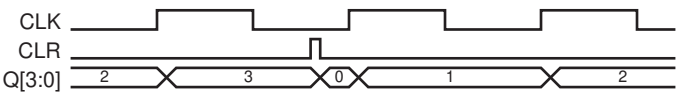




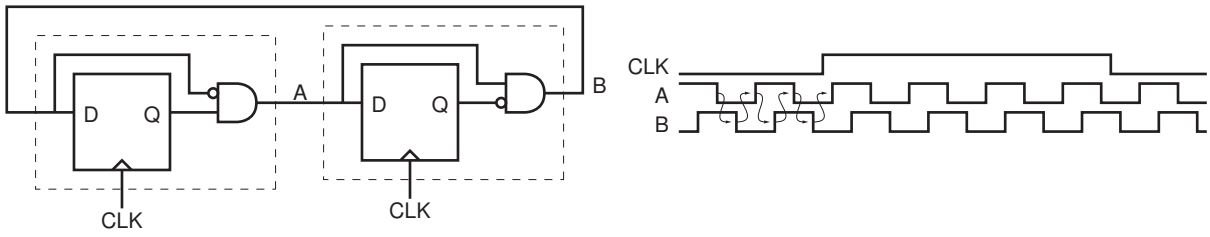
Because CLk was skewed, FF0 missed all the signal changes on IN.

5. Asynchronous control signals, such as those provided by an FSM, can be disastrous.
(Note: All FSM outputs can be glitchy unless they are synchronized.)

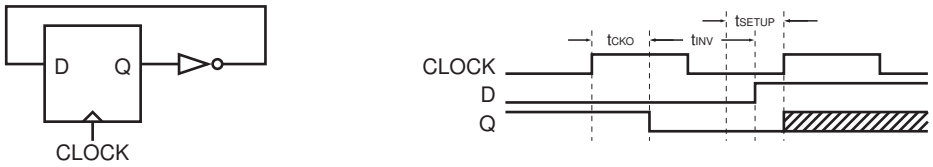




6. Since Mealey machines produce outputs as a function of its state and inputs, do not couple Mealey machines. The changing output of one machine will cause the output of the other FSM to change causing a loop of signals that perpetuates changes.



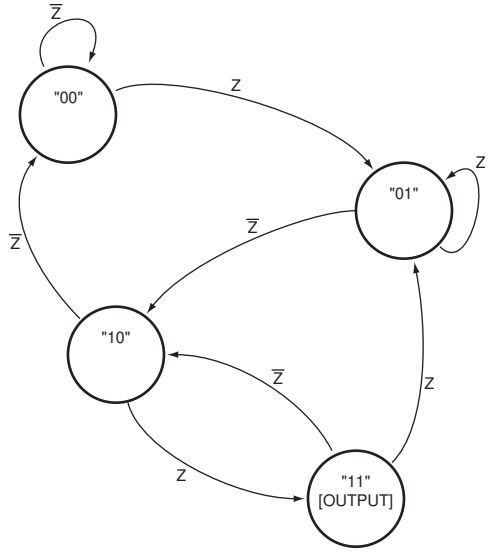
7. If the period is less than the sum of the worst case delay, t_{SETUP}, and t_{CKO}, then t_{SETUP} can be violated.



2. Microcode for FSM emulation (30 pts)

From the schematic, we can determine the following about the MCU:

1. O7 controls whether either the mux enable or the output enable. This means if O7 is high, then the microcode is a jump instruction, and if O7 is low, then the microcode controls an output.
2. O6 and O5 are used to select the appropriate mux input. These are only used when O7 is high (a jump microcode).
3. O4 through O0 control the jump destination when O7 is high. When O7 is low, O4 through O1 do nothing, but O0 is clocked into the OUTPUT.H signal.



This MCU can be programmed to jump on certain conditions (those provided by the mux: jump on Z, jump on $\overline{Z}$, jump always) or output a 0 or 1. With these "commands" we should be able to emulate a Moore type FSM for a sequence recognizer. First, we need to come up with a state trasition diagram with what we want the Moore FSM to do. One possible state diagram is shown to the right.

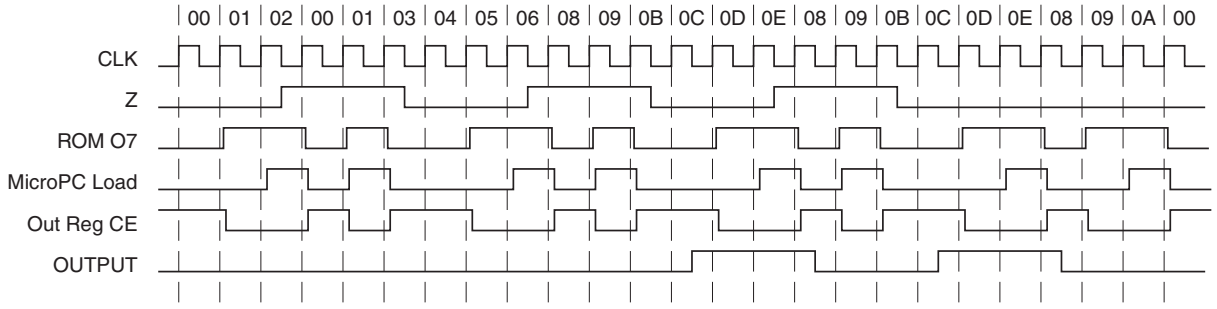
With this state diagram in mind, we should be able to figure out the microcode for the MCU. Something that should be specially noted is that each state should last for three clock cycles since the input bit rate is one third the clock cycle. In other words, we should enter each state exactly three clock cycles after we entered the pre-

vious state. We'll need to plan for this when writing the microcode instructions for the MCU. Below is a table with one possible solution.

Label	ROM Address	Mnemonic Form	Numeric Form	Hex Data
State 00	00000	DO Output 0 (Wait)	0 00 00000	00
	00001	Jump(Z) Pre 01	1 00 00011	83
	00010	Jump State 00	1 11 00000	E0
Pre 01	00011	DO Output 0 (Wait)	0 00 00000	00
State 01	00100	DO Output 0 (Wait)	0 00 00000	00
	00101	Jump(Z) Pre 01	1 00 00011	83
	00110	Jump State 10	1 11 01000	E8
Pre 10	00111	DO Output 0 (Wait)	0 00 00000	00
State 10	01000	DO Output 0 (Wait)	0 00 00000	00
	01001	Jump(Z) Pre 11	1 00 01011	8B
	01010	Jump State 00	1 11 00000	E0
Pre 11	01011	DO Output 0 (Wait)	0 00 00000	00
State 11	01100	DO Output 1	0 00 00001	01
	01101	Jump(Z) Pre 01	1 00 00011	83
	01110	Jump State 10	1 11 01000	E8

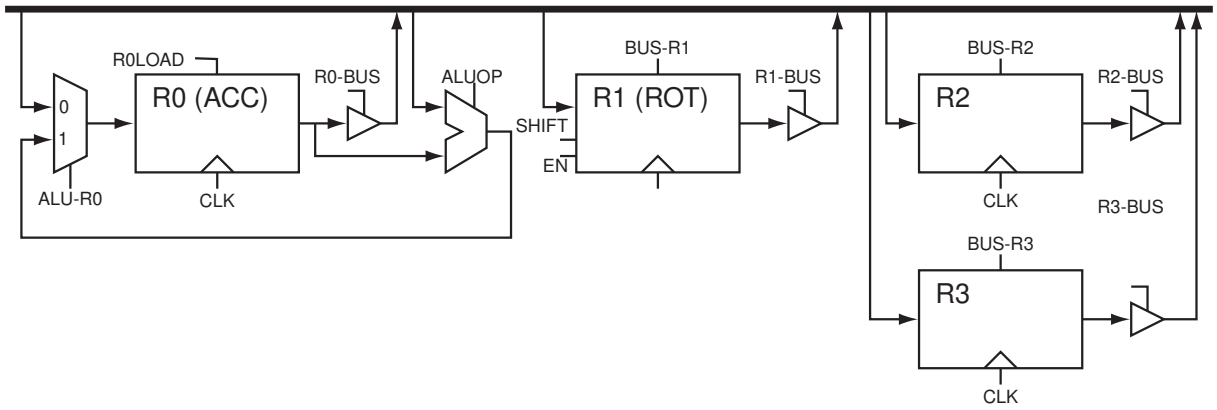
ROM Addresses 01111 to 11111 are don't cares.
Notice that sometimes the jump addresses jumped directly into a state and other times the jump was to the address before a state. This was done so that the control unit would enter a new state every third clock cycle.

The timing of the circuit (if presented with 010101000...) is shown below.



3. Microcode for CPU control (35 points)

We want to create an MCU to control the processor from Exercise 11.7. The datapath shown below is a possible solution to Exercise 11.7.



There are two types of instructions the microcode control unit has: jump instructions and register transfer instructions. Looking at this datapath, for each register transfer instruction we need to designate a source signal and a destination signal. We'll also need to specify the function of the ALU and whether the bus or the ALU is passed to R0. There are also control signals for the rotating shifter.

We'll need 2 bits to designate source register, 2 for destination register, 2 for ALU code, 2 for rotating shifter, and 1 for the R0Mux. For each register transfer instruction we'll need 9 bits. We'll also need to use one more bit to tell the MCU that we'll be performing a register transfer instruction instead of a jump. This gives us a total of 10 bits required for each microcode instruction. Let's encode a register transfer instruction as follows:

Type	Source	Destination	RolFunc	RolEn	ALUR0	ALUCode
1 bit	2 bit	2 bit	1 bit	1 bit	1 bit	2 bit

When the Type field is set to 1, the instruction is a register transfer instruction. The remaining fields are encoded according to the tables below.

Source

00: R0 → Bus 01: R1 → Bus 10: R2 → Bus 11: R3 → Bus

Destination

00: R0LOAD 01: Bus → R1 10: Bus → R2 11: Bus → R3

RolFunc

0: RSR 1: ASR

RolEn

0: Rol Disabled 1: Rol Enabled

ALUR0

0: Bus → R0 1: ALU → R0

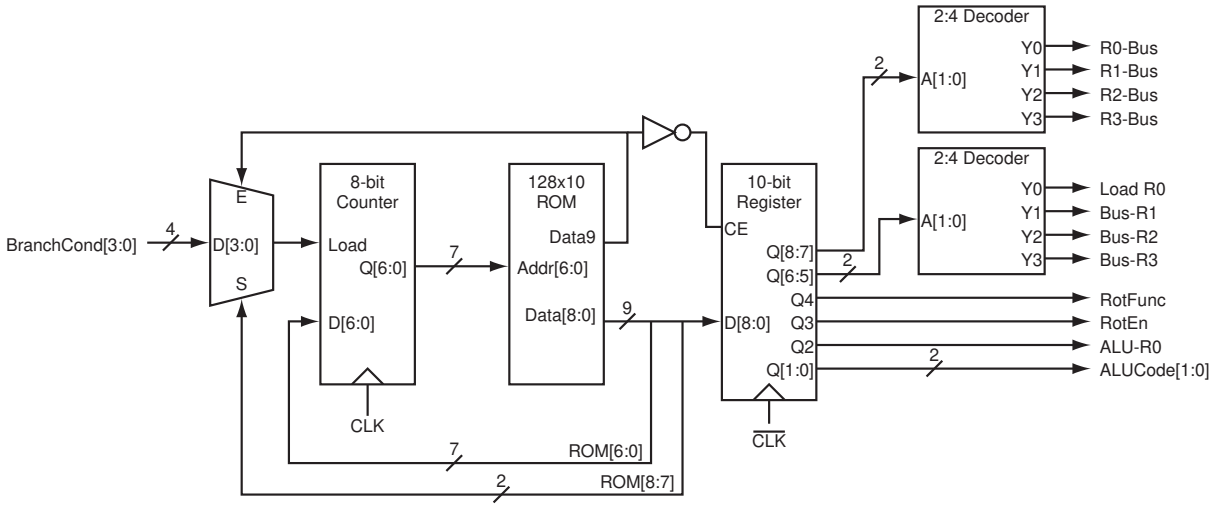
ALUCode

00: COMP 01: INC 10: ADD 11: AND

Now we're done with the register transfer instructions. For jump instructions, the Type field is set to 0. Assuming we only need 128 microinstructions, we'll need 7 bits to specify the jump address. This leaves 2 bits for branch conditions. The branch instruction format looks like:

Type	Branch	Destination Address
1 bit	2 bits	7 bits

The final MCU is:



Microcode needed to implement ADD ($R0 + R2 \rightarrow R0$):
DO R2-Bus, Load R0, ALUADD
DO R3-Bus, Bus-R3 (NOP)

Microcode needed to implement RSR:
DO RotFunc, RotEn
DO R3-Bus, Bus-R3 (NOP)