

# CSE 140L

## Lab Assignment 5

### Sequential Circuit Design

Due December 7, 2014, 4:30 pm

#### I. Introduction

This lab assignment consists of four parts. In the first part, you are asked to design your very own D, T, SR, and JK flip-flops. In the second part, you will use these newly constructed flip-flops to implement both the Moore and the Mealy versions of a Modified Booth's operation generator and controller. In the third part, you will design a variety of sequence detectors and implement, in Mealy form, the third sequence detector you designed. Finally, in the fourth part you will be given a partially specified FSM and asked to derive the missing state encodings.

It is highly recommended that you review Chapters 11 to 15 on "Sequential Logic" in your textbook before starting this lab assignment. We provide you a brief overview below of some of the relevant concepts outlined in your book before proceeding with an explanation of the actual lab assignment.

#### II. Theory

##### Flip-Flop Construction

Before we can even start with designing FSM circuits, we need to figure out how to construct storage elements. Interestingly, this can be accomplished with combinational gates. So far, all logic involving combinational gates that you have implemented in the previous labs have possessed unidirectional datapaths. For example, in the logic arising from your logic minimizer, data propagates directly from the primary inputs towards the primary output in a straight shot. This model of combinational logic, however, is rather constrained, as it does not directly allow for feedback to be taken into account when computing the primary output. This notion of feedback is quite powerful, and it is ultimately what allows us to construct storage elements.

The canonical storage element, termed an SR-latch, is shown below:

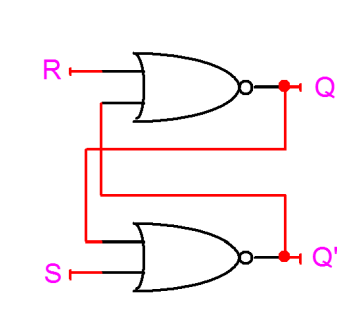


Figure 1: SR-latch

As you can see, the SR-latch is comprised of two NOR-gates and utilizes the output of each gate as feedback into the other. The behavior of the SR-latch is summarized by the following table:

| S | R | Q | Q' |
|---|---|---|----|
| 0 | 0 | Q | Q' |
| 0 | 1 | 0 | 1  |
| 1 | 0 | 1 | 0  |
| 1 | 1 | 0 | 0  |

Table 1: SR-latch Behavior

As you can see from the table, setting  $S=0$  and  $R=1$  will set  $Q$  to 0 and  $Q'$  to 1, whereas setting  $S=1$  and  $R=0$  will set  $Q$  to 1 and  $Q'$  to 0. If both  $S$  and  $R$  are set to 0, the previous values of  $Q$  and  $Q'$  are effectively stored. Conversely, setting  $S=1$  and  $R=1$  will cause  $Q$  and  $Q'$  to both be 0. The consequent violation of the property of  $Q$  and  $Q'$  being complements of each other is one of the reasons why this case is deemed to be an invalid input combination.

As we can see, we are able to store values using only two combinational gates. In the previous labs, however, storage elements like the D flip-flop were synchronized with respect to the rising edge of a clock. How can we construct a D flip-flop with this sort of synchronization? The first step is to transform our SR-latch into a gated SR-latch, which is shown below:

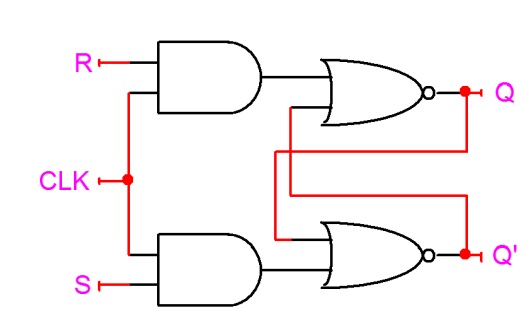


Figure 2: Gated SR-latch

Here, the values from  $S$  and  $R$  will propagate only if the  $CLK$  input is high. One can then convert this SR-latch into a D-latch, as shown in the following figure:

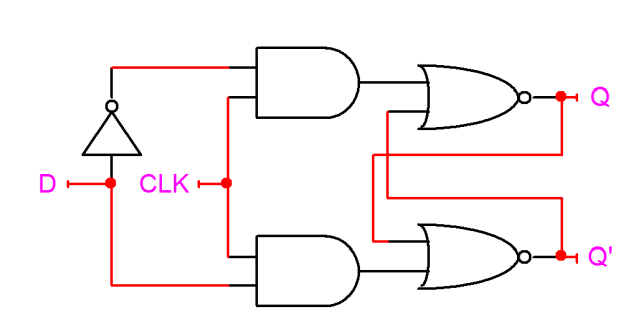


Figure 3: Gated D-latch

By hooking the S and R inputs in this way, we ensure that we are only setting or resetting the latch. While this is a step in the right direction, the D-latch will propagate data whenever the clock is high, which is not the same as the ability to propagate data on the rising edge of a clock. To effect a propagation on the rising edge of a clock, one can utilize two latches and arrange them in a master-slave configuration, which is shown below:

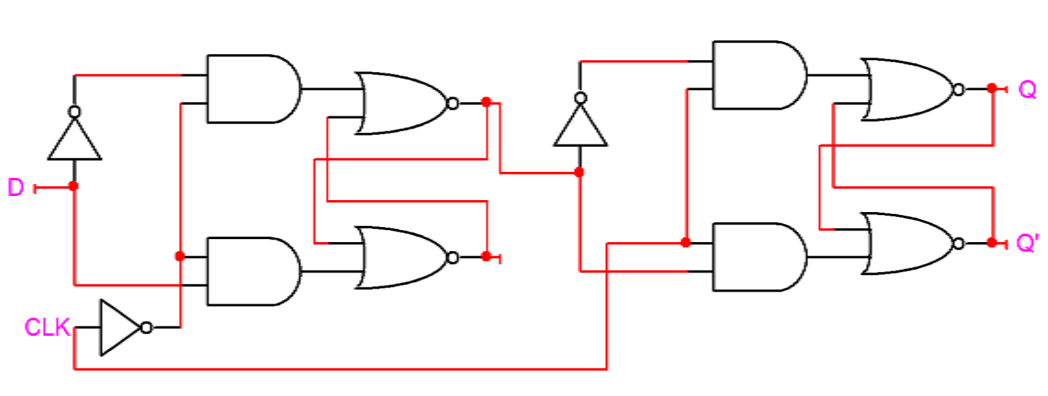


Figure 4: Master-Slave D Flip-flop

Here, what we see are two gated D-latches, where the Q output of the first (master) latch is supplied as the D input into the second (slave) latch, and the gating parameter is inverted between the two latches. The inversion of the clock is what ultimately allows for the D input to be propagated towards the Q output on the rising edge of the clock. A more detailed timing analysis of the master-slave D flip-flop can be found on page 329 of your textbook.

How does one construct other flip-flops utilizing a similar master-slave configuration? As you have probably figured out, a T flip-flop can be trivially implemented with a D flip-flop. However, the construction of an SR flip-flop requires more care. Your textbook shows the following SR flip-flop implementation, which also utilizes a master-slave architecture:

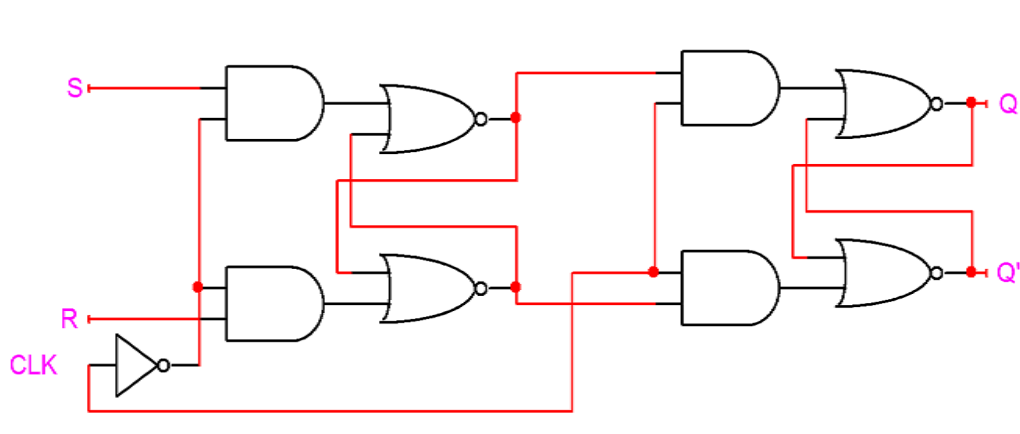


Figure 5: SR Flip-Flop

However, it should be noted that the implementation the SR flip-flop displayed above will fail in certain corner cases. As you recall, flip-flops, by definition, propagate outputs to Q and Q' on the rising edge of the clock. However, due to the ability to set S and R separately, it is possible to induce an unwanted state change in the master latch that can propagate to the slave latch. For

example, let's say we reset the flip-flop by supplying  $S = 0$  and  $R = 1$  to the inputs of the flip-flop. When we toggle CLK so that it is 1, we will induce a reset, making  $Q = 0$  and  $Q' = 1$ . Now, let's do the following:

1. Set CLK to 0
2. Set  $S = 1$  and  $R = 0$
3. Set  $S = 0$  and  $R = 0$
4. Set CLK to 1

According to the specification of an SR flip-flop, the resulting outputs should retain the value we discussed in the previous paragraph of  $Q = 0$  and  $Q' = 1$  because, on the rising edge of CLK (which is when the flip flop should be sensitive to the input combinations), S and R should both register 0. Unfortunately, because of the master-slave architecture, when CLK is 0, setting S to 1 and R to 0 will change the state of the master latch, which propagates on the rising edge of CLK to the slave latch to yield an incorrect output of  $Q = 1$  and  $Q' = 0$ .

How does one fix this corner case? As you have probably realized, the problem arises due to the ability to change the state of the master latch before the rising edge of the clock. While changing the state of the master latch is desirable when setting or resetting the flip-flop, one must be sure to restore the state of the master latch to the current state when S and R are both set to 0. Ensuring this functionality will ultimately enable you to successfully implement an SR flip-flop. Once you have successfully constructed your SR flip-flop, converting it into a JK flip-flop is trivial.

### **Sequential Circuit Design Flow**

Now that we have established the ability to construct flip flops out of simple combinational gates, one can develop some theory pertaining to the design of sequential circuits. Sequential circuits require state information, in addition to inputs, to generate their outputs. Accordingly, the key elements of sequential circuits are flip-flops, which are used to hold state information. Furthermore, by inserting combinational logic, flip-flop states and circuit inputs are integrated to determine the output signals, as well as the "next states" of flip-flops.

The conventional way to compute the combinational logic functions for next-states and outputs is through a state machine representation. The verbal description of the problem is first interpreted into state diagrams that demonstrate the transitions between states of the circuit and the input/output conditions. State diagrams, or state machines, not only resemble flow charts in principle, but also have to include timing and state characteristics for the sequence of events. State diagrams establish the starting point for the design of sequential circuits. The entire design procedure of sequential circuits can be summarized as follows:

1. Obtaining the state diagram for the sequential circuit to be designed.
2. Minimizing states if possible.
3. Assigning binary codes to the states according to the encoding algorithm chosen.
4. Generating next state and output tables based on the state diagram and the excitation table of the selected flip-flop.
5. Deriving the logic functions for next-states and outputs
6. Implementing the logic with flip-flops and combinational gates, as specified by the equations.

### **Mealy and Moore Machines**

One implementation detail concerning finite state machine implementations involves the parameters that should be utilized when evaluating the output. A *Mealy machine* is an FSM in which outputs are generated based on the current state as well as the current input. A *Moore machine*, on the other hand, only generates outputs based on the current state. Examples of these two types of FSMs can be found in Chapter 13 of the textbook.

### **State Minimization**

The minimization of states requires the observation that equivalent states can be merged. How does one identify equivalent states? One constraint is that the states have to produce the same output on every outgoing transition, which, while not sufficient, is definitely necessary to assert state equivalence. In addition to the aforementioned constraint, if two states transition to the same states on the same inputs (while of course delivering the same output on these transitions), the two states will also be equivalent. More subtle, however, is that this can be generalized into the following recursive formulation: if two states transition to *equivalent* states on the same inputs (while also delivering the same output on these transitions), then the two states will be equivalent. For example, let's say we have an example of the first case, where two states, S1 and S2, transition to identical states on the same inputs. This leads us to conclude that S1 and S2 are equivalent. Now, let's say we have two states S3 and S4. If S3 and S4 transition to identical states *except* for the case where one input generates the transition from S3 to S1 while that same input generates a transition from S4 to S2, the states S3 and S4 will still be equivalent because S1 and S2 are equivalent. This formulation can be rewritten as an implication, where S3 and S4 being equivalent implies that S1 and S2 are equivalent. If, at any point, we find that S1 and S2 are not equivalent, the implication will break, and S3 and S4 will no longer be equivalent. This gives rise to an algorithm for state minimization that utilizes an **implication table**, an example of which can be found in Chapter 15 of your textbook.

### **Encoding Heuristics**

In order to assign binary values to the states, state encoding strategies must be employed. In this lab, you will be concerned with the **minimum-bit-change heuristic** and the **prioritized adjacency heuristic**. The prioritized adjacency encoding strategy is discussed in more detail in Section 15.8 of your textbook as well.

#### ***Minimum-Bit-Change Heuristic***

The **minimum-bit-change heuristic** suggests the assignment of binary encodings to states so as to minimize the total number of bit transitions between all states. The method can be applied by assigning weights to edges in the diagram such that each weight represents the number of bit-flips between two neighboring states multiplied by the number of input combinations causing the transition. The algorithm aims to find the minimum sum of weights, that is, the state encoding satisfying the minimum number of bit transitions while traversing the diagram.

The Minimum-Bit-Change encoding heuristic is particularly suitable for SR flip-flops, since for them multiple ways hold for retaining but only a single way for changing the current state. For the case of retaining the current value of an SR flip-flop, one can see that two paths can be exploited, namely, the (re)setting of it and the use of a “don’t-change” input (SR = 00) instead, resulting in a multiplicity of possibilities that can be observed in the “don’t-care” (X) conditions

in the excitation table of SR flip-flops. As a result, the next-state logic typically becomes simpler, since we can benefit from these X bits while deriving the logic functions out of Karnaugh maps. In contrast, this encoding scheme is not quite useful while designing with D (or T) flip-flops that do not have “don’t care” conditions in their excitation tables. Neither is this heuristic of particular help for the JK flip-flops, whose every transition, retaining or flipping the current state, can exploit either of two paths, thus always generating exactly one “don’t care” in the associated Karnaugh map, making the minimization of the number of bit flips in the FSM diagram immaterial. The detailed discussion on the various types of flip-flops and their excitation tables in Section 6.5 of your textbook can help you perhaps appreciate the particular applicability of this encoding heuristic to SR flip-flops.

### ***Prioritized Adjacency Heuristic***

The **prioritized adjacency heuristic** assigns binary encodings that are Hamming distance 1 away (i.e. adjacent encodings) to state pairs in a prioritized order. As there exists a limited number of single bit distance codes, the prioritized adjacency heuristic starts by allocating adjacent encodings to the states with the highest priority, namely, the pairs of states that have the same next state under the same input condition. Once the highest priority state pairs have been handled, the heuristic suggests single Hamming distance assignments to state pairs that are next states of the same state as long as they are transitioned into on Hamming adjacent input combinations. Finally, state pairs that have identical output behaviors for the same inputs (trivial for Moore implementation, as output does not depend on inputs) are accorded single Hamming distance assignments, presuming of course that any such are still available. Each of the heuristics outlined in turn can be prioritized by a weight that reflects the number of times that it holds for that particular pair of states. These heuristics aim at placing the same values in neighboring rows and columns in the next-state and output Karnaugh maps. In this way, larger groupings of minterms (or maxterms) can be attained and more compact logic functions can be derived. As this heuristic aims at attaining topological affinity, the main principle underlying this algorithm is independent of the particular flip-flop type, and should deliver improvements to any flip-flop choice on the average case.

As an example of applying this heuristic, we will use the “101” sequence detector FSM found in Chapter 14 of your textbook (and reproduced in Figure 8 of Part 3). The first rule in assigning adjacent states is that states which have the same next state for a given input should be given adjacent assignments. Using the example of the “101” sequence detector, it should be evident that states  $(S_0, S_1)$  should be adjacent, because both states transition to  $S_1$  on an input of 1. Similarly, states  $(S_0, S_2)$  should be adjacent, because these states both transition to  $S_0$  on an input of 0. The second rule that should govern the decision for your state encoding is that states which are next states of the same state, should be given adjacent states. In the example, the sets of states that satisfy this heuristic are the next states of  $S_0$ ,  $S_1$ , and  $S_2$  or  $(S_0, S_1)$ ,  $(S_1, S_2)$ , and  $(S_0, S_2)$ , respectively. In summary, we have the following sets that are specified by the first and second adjacency heuristics:

1.  $\{ (S_0, S_1), (S_0, S_2) \}$
2.  $\{ (S_0, S_1), (S_1, S_2), (S_0, S_2) \}$

We should attempt to fulfill as many of these adjacency conditions as possible, but it is worth noting that in most cases you will be unable to satisfy every set in the heuristic. If this is the case you should first attempt to satisfy every set in your first adjacency heuristic and then move on to satisfying your second heuristic. By letting,  $S_0$  equal “11”,  $S_1$  equal “01”, and  $S_2$  equal “10” we have satisfied the entire set of the first adjacency heuristic. In this case, we have also satisfied two of the

sets in the second group. In your implementation of this sequence detector, make sure to adhere to the same procedure illustrated for this simple example. Following this heuristic will result in minimizing the necessary logic gates for your next state logic.

### III. Implementation

#### **Part 1: Flip-flop Design**

In this part, we ask that you implement your own D, T, SR, and JK flip-flops, the behaviors of which are detailed extensively in your textbook and in the preamble. We ask that your SR and JK flip-flops function correctly when confronted with corner cases like the one shown in the preamble.

Additionally, we ask that these flip flops possess *asynchronous set and reset inputs*. The concept of asynchronous inputs should not be unfamiliar, as you have used them in your previous labs. However, as a reminder, an asynchronous reset is a signal that, when asserted to 1, will set Q to 0 and Q' to 1 irrespective of the clock. Likewise, an asynchronous set is a signal that, when asserted to 1, will set Q to 1 and Q' to 0 irrespective of the clock. It should be noted that having both the set and reset be 1 is undefined and can be implemented however you see fit.

How does one construct an asynchronous set and reset? As you recall, the fundamental building block of a flip-flop was the SR-latch, and converting the SR-latch into a flip-flop required numerous gates to be added on top of the latch to ensure synchronous behavior. An asynchronous set or reset should thus possess the ability to override all of the synchronous aspects of the flip-flop so that they set or reset the latches directly.

For each flip flop design, we ask that you **create a *Logicworks* component**, as this will be useful in the later parts of the lab.

#### **Part 2: Modified Booth's FSM Design**

As you know, FSMs can be implemented in a Mealy or in a Moore fashion. In this part of the lab, you will be implementing both the Mealy and the Moore model of the state machine that generates the Modified Booth's multiplication operations.

##### ***2A. Mealy-based Booth's FSM Design***

When performing Modified Booth's multiplication, we scan the multiplier from right to left by examining three bits in every step. Therefore, there are a total of 8 possible patterns in the bits we are examining, each of which corresponds to an arithmetic operation (addition, subtraction, or no-op) specified by the Modified Booth's table. From a sequential circuit design perspective, we can also view these eight patterns as eight different states of an FSM. The transition from one state to another state will be triggered by the right shifting of the multiplier as the new bit that is shifted into the three-bit window changes the bit pattern that is under examination. Of course, as you remember the two special cases of Modified Booth's result in state transitions that are a bit more complicated than the simple shifting in of the new bit as the leftmost member of the triplet designating a particular state. Once a particular current state is thus determined, the arithmetic operation that needs to be performed at that step in a Mealy machine is a function of the current state and the new bit subsequently being shifted in.

Based on these concepts an eight state Mealy FSM for the Modified Booth's algorithm can be constructed as in Figure 6, where each node name represents the 3-bit window you examine in Modified Booth's algorithm. An interesting subtle point may be the fact that as Mealy FSMs

generate outputs on state transitions, the rightmost bit of the initial state(s) of a Mealy machine may be viewed essentially as a placeholder.

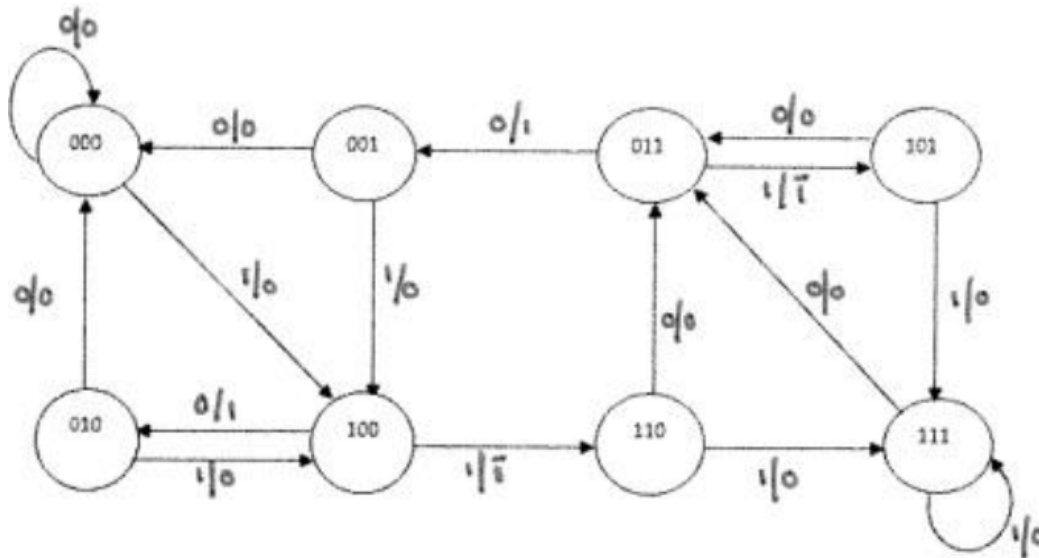


Figure 6: The eight-state Mealy FSM for Modified Booth's

As you can see in Figure 6, each edge is annotated with a pair of values, separated by a slash("/"). The left symbol on an edge denotes the input value of the FSM, which is actually the new value to be shifted into the three-bit window under examination. The value after the slash is the output value of the FSM circuit that is determined by the input value and the current state. Then the multiplication process will traverse the entire FSM according to the bit that is shifted into the three-bit window during the scanning of the multiplier.

As you might have already noticed, the eight-state FSM of Figure 6 can be optimized as some states within this diagram are actually equivalent. You are asked to generate the optimized FSM for the Modified Booth's algorithm using the state minimization algorithm, and to implement the optimized sequential circuit using **SR flip-flops**. Since this circuit issues instructions for three different arithmetic operations (addition, subtraction, and no-op, denoted by "1", "1 bar", and "0"), the output will need to consist of two bits (add, sub) to cover all three possibilities. The value of "1" in the FSM means an addition operation needs to be performed, whereas a value of "1 bar" in the FSM indicates that a subtraction operation should take place. These can be easily turned into a 2 bit representation by setting either the "add" or "sub" bit to 1. Of course these two signals cannot be 1 simultaneously, but it is possible for both of them to be 0, thus indicating a no-op.

One essential issue that needs to be considered is the initial state at the beginning of a Modified Booth's multiplication process. As discussed earlier, the initial state(s) are composed of a placeholder 0 bit on the right and a middle bit in the state designation of 0, corresponding to the initial value of FF of your Lab 2. The leftmost bit of the state designation corresponds to the rightmost bit of the multiplier register that could be either 0 or 1, leading to two possible initial states of "000" and "100", respectively.

Putting all these together, your implementation for the 8-state FSM for Modified Booth's should have a block-level view as specified in Figure 7. Although it has three inputs, the inputs "INIT" and "INIT\_VAL" are just used for initializing the state at the beginning of the multiplication process and are no longer used thereafter. "Q0" is the input value that needs to be monitored for the

determination of the next state and the output values. The definition of the output bits (add, sub) is the set of arithmetic operations as explained before.

Figure 7: Block-level view of the Modified Booth's operation generator FSM

As you might have observed in your implementations, the circuit needs 3 flip-flops to represent the complete set of its states. The hardware cost may be significantly reduced if we can minimize the number of states in the diagram. You are asked to optimize the Modified Booth's FSM by performing state minimization, and implement an optimized version of the circuit using the **minimum-bit-change encoding heuristic** and **SR flip-flops**. A correct state minimization process will reduce the number of states to 3, thus cutting down the number of flip-flops needed in the implementation from 3 to 2.

## 2C. Design of the controller

The Modified Booth's operation generator you have designed in Part 2 is not the only sequential circuit needed for a Modified Booth's multiplier. As you remember, the multiplier circuit in Lab 2 also uses a controller to synchronize the operations of various components. You will be implementing this controller in this part of your lab assignment.

In the library we built for you in Lab 2, the controller was implemented using a PROM. The PROM contains output values that are addressed by a cycle counter within the controller. This implementation style means that it has to store, in its memory, output values for every cycle of execution from 0 to 39. Perhaps, in an FSM-based design for the controller, it is possible to take advantage of similarities in many of its cycles. Let's first review the behavior of the controller using the output table below:

| Cycle                | End | Init | LoadA | Shift |
|----------------------|-----|------|-------|-------|
| 0                    | 0   | 1    | 0     | 0     |
| 1,3,5,7,9,...,35,37  | 0   | 0    | 1     | 0     |
| 2,4,6,8,10,...,36,38 | 0   | 0    | 0     | 1     |
| 39                   | 1   | 0    | 0     | 0     |

Table 2: Controller Output Table

The controller behavior shown above contains a great deal of repetition, which seems to imply the possibility of aggressive state minimization. However, unfortunately, the formal method of state minimization is unable to lower the number of states below 40. Why?

If we were to implement the preceding controller as an FSM, we would need to proceed from the first cycle to the last as given. In other words, the preceding controller is a Moore FSM whose behavior goes through a fixed number of steps regardless of input. This requires us to keep track of the number of steps so that we can finish at the correct point. If we were to put two "repeated" states into an equivalence group, for instance, we would lose track of how many more iterations we need to perform!

One way around this problem is to excise the information of the number of steps from the FSM and make it an input to it. We can do this by introducing a counter and a comparator, for example, and inputting a single line into the FSM indicating whether we have completed the iterative process. Using this extra input, we can have a smaller number of states that have the same behavior for the same input combination. For example, we can "collapse" cycles 1 through 38 into two states (by incrementing up until 19 on each Shift cycle), as they alternately generate two identical output combinations. To make the counter behave correctly, we also need additional outputs from the FSM to initialize and increment the counter as needed.

Notice that this technique does not actually lower the amount of "information" that we need; although the number of states has been reduced, an extra counter, which is also a memory element, is needed to keep track of how far the FSM has proceeded on its way to the final state. Only real state minimization could have lowered this amount by taking advantage of truly equivalent states, which do not exist in our Lab 2 controller. The use of this ad hoc technique does allow us to

simplify our next-state and output logic, though, by associating a group of states with a single encoding.

In addition to simplifying our next-state and output logic, we can also easily add an optimization that will skip over unneeded Load states. By adding an input to signal whether or not an operation needs to be performed, we can make the FSM simply stay in the Shift state if no operation needs to be performed during the next Load state.

We ask you to implement the FSM discussed using **JK flip-flops**. Use whatever state encoding technique you wish, but explain your choice in your report. Also turn in a diagram of the state machine with transitions, next-state tables and output tables.

The circuit should be put into a component called Controller that is capable of interfacing with your Lab 2 multiplier. This will be turned in within a library named "lab5\_library.clf". You also need to turn in a Lab 2 implementation with this component attached. You may use the Lab 2 implementation provided on the course web site, though you are encouraged to use your own.

### Implementation Guidelines

Now let us summarize the preceding information and try to provide you a somewhat sharper picture regarding how to design FSMs for the Lab 2 controller.

The controller will be awakened when its **Begin** input makes a transition from 1 to 0. This can be implemented by forcing the FSM to enter an 'Init' state when the **Begin** signal is a '1', and then forcing the FSM to leave the 'Init' state when the **Begin** signal is a '0'. Since the datapath-controller relies on a counter to track the number of the iterative steps, the FSM needs to include additional outputs to initialize and increment the counter. When the counter indicates that 19 pairs of the load and shift operations have been performed, the FSM will enter an 'End' state and stay at that state until the **Begin** input is set to '1' again. In sum, the data-controller can be implemented within a total number of **4** states using **2** flip-flops and a 5-bit counter which will count up to 19, incrementing on each shift. You are asked to implement this FSM using **JK flip-flops**. Again, you need to simplify your next state logic and output logic by using as few gates as possible.

### 2D. Integration into Lab 2 Multiplier

Let's finally put the optimized Modified Booth's operation generator controller and the controller you have designed in subparts B and C into some real thing and see if they work ☺. The best candidate that we can use for this purpose is of course the Lab 2 multiplier.

The optimized Modified Booth's operation generator will be used to replace the part of logic associated with the flip-flop *FF* in the original design of Lab 2. Since the *FF* flip-flop value no longer exists in the new version of the design, some glue logic may need to be slightly modified and perhaps even completely removed. The new glue logic should use the outputs of the Modified Booth's operation generator to trigger the appropriate operations for the adder/subtractor. Note that you may need to provide an inverted clock signal to the operation generator so that the controller will get the next operation in time to decide whether or not to skip the Load state. This is because there must be a phase shift in the clocks for the operation generator FSM and the controller FSM, so that the operation generator will be able to determine what operation will occur as a result of the Shift and perform it thereupon before the controller decides if the Load state is needed.

The integration of the new controller should be rather straightforward. Since the input/output interface of the new controller you have designed is almost exactly the same as the one we

provided you in the Lab 2 library, you can just do a direct replacement and provide the additional input to signal whether or not an operation is occurring by using some simple glue logic.

### **Part 3: Sequence Detectors Galore**

Now that you are acquainted with the process of state minimization and state encoding, it is time for you to design your own behavioral FSM. As you know from Chapter 14 of your textbook, many FSMs can be seen as the description of a circuit that outputs a 1 upon recognizing certain designated input sequences and outputs a 0 on the rest. For example, if one wanted to construct a circuit that outputs a 1 upon detecting the sequence “101” and outputs a 0 otherwise, one could do so by first constructing an FSM with this functionality and then converting it into a circuit using flip-flops. As you know from your textbook, a Mealy FSM that outputs a 1 upon detecting a “101” sequence looks as follows:

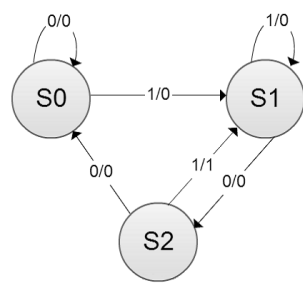


Figure 8: “101” Sequence Detector

S0 functions as the starting state, and any sequence of inputs will output a 1 if a “101” sequence is encountered. While a detailed account of the construction of the FSM shown above can be found in Chapter 14 of your textbook, it is important to note a couple of important aspects about this sequence detector.

Firstly, the FSM contains three states. Three also happens to be exactly the number of bits in the sequence the FSM detects, “101”. While this may seem coincidental at first, a moment's reflection will reveal that the minimum number of states required in a Mealy implementation of an  $n$ -bit sequence detector will be  $n$ .

The other important aspect of which you should take note is how overlapping sequences are handled. For example, the sequence “10101” possesses two overlapping “101” sequences, despite only having 5 bits. In order to account for overlapping sequences, the outlined FSM ensures that any input of 1 will induce a transition to S1, which means that S1 essentially functions as the detection of the first 1 of a sequence. This 1 can coincidentally be a part of the ending of a “101” sequence, but the transition to S1 is blind to that, which effectively is what allows for the overlap.

### **3A. Sequence Detector for “0101 → 0011”**

In light of the aforementioned observations, we ask that you create a state diagram for a sequence detector that will detect a sequence of 0011 after a sequence of 0101 has been seen. Keep in mind that after a sequence of 0101 has been identified you do not need to re-detect this sequence. Thus a bit-string of “010100110011”, should cause two consecutive sequences to be detected. You will need to signal that a sequence was detected by outputting a 1 upon detection of the bit-string. Please construct a Mealy-based FSM, and produce a state diagram that should be included in your write-up.

### **3B. Sequence Detector for “0100 → 1001”**

You may still be a bit confused about how to design state diagrams. In order to make sure that you are truly a master of the concept, we request that you produce another Mealy-based sequence detector state diagram. This FSM should output a 1 upon detection of the bit-string “1001”, only after a bit-string of “0100” has been seen. Similarly to your detector of Part 3A, after a sequence of “0100” has been seen you do not need to re-detect this sequence. When given a bit-string of “01001001”, your FSM should cause two consecutive sequences to be detected. Please construct this FSM, and produce a state diagram that should be included in your write-up.

### **3C. Sequence Detector for both “0101 → 0011 or 1001” and “0100 → 1001 or 0011”**

By now you should be a master at designing FSMs, and we now ask that you utilize your skills by actually implementing a sequence detector in LogicWorks. You will still be asked to provide a state diagram for this part, and we recommend you put significant care into this diagram because any error in your diagram will be reflected in your implementation. Also, you will find that this new detector will be a little bit more interesting than Parts 3A and 3B. This implementation must output a 1 upon detection of the sequence “0101 → 0011, 1001” or “0100 → 0011, 1001”. For clarification, this means that the sequences “0011” and “1001” are both able to be detected after recognizing a bit-string of “0101” or “0100”. Of course, you could combine your “0101 → 0011” detector and your “0100 → 1001” detector into one unoptimized sequence detector (making appropriate modifications in order to accurately transition between the two sequence detectors of course), but this begs the question as to whether the combination is actually minimal. A simple combination of your two state machines would produce something like 15 states, which would require 4 flip-flops to implement. As you know from Part 1, flip-flops incur a nontrivial hardware overhead, and the elimination of even a single flip-flop can yield significant savings in terms of area and power. One obvious observation is the fact that naively combining the two FSMs will yield two starting states, which should obviously be merged. However, this trivial simplification will still yield a greater than minimal result.

We thus ask in this part that you construct a minimized FSM that detects both the sequences “0101 → 0011, 1001” or “0100 → 0011, 1001”. While you may find the application of state minimization to be helpful, you are also permitted to build the FSM from scratch. As a hint, the minimized FSM for this sequence detector should possess exactly 8 states. Make sure to find this minimal state machine and include a diagram. Please implement your minimized FSM in LogicWorks using **T flip-flops**. Also make sure to encode the states of your FSM using **the prioritized adjacency heuristic**.

#### **Part 4: Partially Specified FSM**

In this part, you will attempt to reverse engineer an FSM and its resulting logic based on an incomplete automaton and the corresponding output logic. The automaton is shown below:

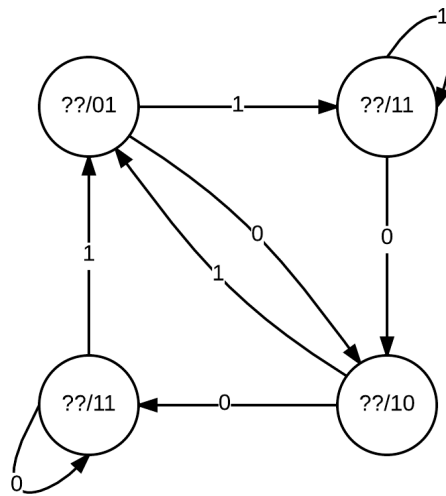


Figure 9: Partially specified FSM

In Figure 9 you see 2 flip flops, Q1 and Q0, and 2 outputs, O1 and O0. For both, the leftmost bit is the one denoted with a 1, while the rightmost is the one denoted with a 0. The output equations are shown below:

$$O1 = Q0' + Q1$$

$$O0 = Q0 + Q1'$$

We ask you to use the output equations as well as the FSM in order to derive what the state encodings must be, and upon derivation of the state encodings, we ask that you implement the FSM using **JK flip flops**. It is possible that one may obtain different solutions for the state encodings, so we ask that you simply choose one and implement it. If you have any guesses of the functionality of the resulting circuit feel free to let us know together with an explanation of how the outputs correspond to that functionality.

## IV. Deliverables

Please submit your relevant *Logicworks* files. Please also submit a report. The specific deliverables for each part are described below.

### **Part 1:**

Please submit a **Logicworks library** containing your D, T, SR, and JK flip-flop implementations. These flip-flops should possess correctly functioning asynchronous set and reset inputs, and if both asynchronous inputs are off, the flip-flops should function synchronously, where the inputs are propagated on the rising edge of the clock.

In your report, please include a description of how you implemented each flip-flop, including how you transformed the D flip-flop into a T flip-flop and the SR flip-flop into a JK flip-flop. Additionally, you should explain how you implemented the asynchronous set and reset functionality.

### **Part 2:**

Include a **.clf** and a **.cct** file. The .clf file should contain your Mealy-based Modified Booth's FSM implementation, and your Moore-based Modified Booth's FSM implementation, and your Modified Booth's Controller implementation. The .cct file should contain a working Lab 2 implementation using the components you created in the previous subparts of Part 2.

In your report, please include a state diagram of both your minimized Mealy FSM and your minimized Moore FSM. Please provide explanations as to how you were able to minimize the states for each FSM. Please also show your reasoning as to how you assigned state name encodings.

### **Part 3:**

Include **one .cct file**, which should contain your implementation of the combined sequence detector.

In your report, explain how you were able to minimize the states in the final sequence detector. Please also show your reasoning as to how the prioritized adjacency encoding was done. Include **three state diagrams**: one for the detection of sequence (0101  $\rightarrow$  0011), another for detection of sequence (0100  $\rightarrow$  1001), and a final diagram that mirrors your implementation for the combined sequence detector (0101  $\rightarrow$  0011, 1001) or (0100  $\rightarrow$  0011, 1001).

### **Part 4:**

Include **one .cct file** that contains your implementation of the FSM.

In your report, please include a **diagram** of the FSM as well as any guesses regarding its functionality. Please describe how you found the state encodings.