EECS-140/141 Introduction to Digital Logic Design
Lecture 8: Synchronous Sequential Circuits (aka Finite State Machines)

I. OVERVIEW

I.A Introduction

We now begin learning to design very *general* and *powerful* digital logic circuits by using:

We will assume all FFs have a common clock signal and all trigger off the same clock edge (positive or negative), which we will call:

Hence these are:

General form for such ckts:

w: any number of binary *inputs*

Q: any number of FF *state variables* (FF outputs)

z: any number of binary *outputs*.

Such a circuit is also called a *Finite State Machine (FSM).*

If outputs *z* depend (directly) *only* on state *Q*, the FSM is called a *Moore* FSM.

If outputs *z* depend (directly) on both state *Q* and inputs *w*, the FSM is called a *Mealy* FSM.

I.B Motivating Example: Serial Packet Data Communication

I.B.1 Background

In packet-oriented data communication systems (e.g., the Internet), information (data) is sent in *binary* form and in chunks called *packets*. Packets are sent only when there is data to be sent.

But many *serial* data comm links (like DSL and optical fiber systems) *must* send *continuous* binary values (1's and 0's).

So, there are *two* related problems to be solved:

— What does the *transmitter* send on the data link when there is no packet to send?

— How does the *receiver* determine where each packet starts and stops?

I.B.2 Solution: High-level Data Link Control (HDLC) Flags

A *part* of this *protocol* (set of rules) is the use of *Flags*.

A Flag is a particular 8-bit sequence:

When there is no packet to send, Flags are sent back-to-back. When a packet is ready, it is sent instead of Flags. So, Flags also serve as:

In *receiver*, we need a ckt to *detect* the Flag *sequence*. Receiver simply discards the Flag bits.

But now there is a *new* problem: What if there are *data* bits in a packet that look like a Flag (01111110)???

Solution: Next section.

I.B.3 Bit Stuffing

This is done by the *transmitter*. A ckt monitors the serial *packet* bits "looking" for the 6-bit sequence: 011111 (first portion of a Flag). If it finds this pattern, it inserts ("stuffs") a 0 bit just *after* this sequence. After stuffing, *packet* bits can *never* contain the 01111110 Flag sequence.

Note: The stuffed 0 is *included* when looking for subsequent 011111 sequence, since the stuffed bit may *create* a Flag sequence (example later).

I.B.4 Bit De-Stuffing

At receiver, *after* Flags have been detected and removed, the stuff bits in the packet data must be removed ("de-stuffed"). This is done by *again* looking for the *same* 6-bit 011111 sequence. At receiver, this sequence is *always* followed by a 0, which was "stuffed" in at the transmitter. So, the receiver removes this stuff bit (but again, that removed stuff bit must be included when looking for the next 011111 sequence). Example:

Overview diagram:

I.B.5 Conclusions

We need two different circuits:

a) For Flag removal at Rcv, need a ckt that detects 8-bit sequence 01111110 (and discards these bits).

b) For Bit Stuffing (Xmt) and De-Stuff (Rcv), need a ckt that *detects* 6-bit sequence 011111 in *packet* data.

II. FSM Design Examples

II.A Flag Detector (Moore FSM)

II.A.1 Description

Rather than detecting a "real" Flag (01111110), we will simplify and detect shorter "Flag": 0110. If found:

- Output $z = 1$ in *next* clock cycle after the last 0 -- this allows a Moore design.
- Start over looking for 0110 in *subsequent* bits (trailing 0 of one Flag *cannot* be leading 0 of next).

Example:

II.A.2 State Specification

Must first express circuit behavior in terms of *state* and *transitions* between states (caused by inputs).

Here, we can define the following states:

S_0:

S_1:

S_2:

S_3:

S_4:

II.A.2.a State Diagram

This is a convenient, visual way to express the state specification.

Circles:

Arrows:

Note: Each state must have arrow *leaving* for each input condition/combination.

Note: To detect "real" Flag 01111110, just need to insert 4 more states like S_2 in the "chain".

II.A.2.b State Table

Another way to specify states and transitions -- less visual, but better for proceeding with the design.

Present State	Next State $w = 0$	Next State $w = 1$	Output z
S_0			
S_1			
S_2			
S_3			
S_4			

II.A.3 Implementation

II.A.3.a State Assignment

First assign a binary string to each state.

Example:

Use y_i to represent each *present* state binary variable and Y_i to represent each *next* state binary variable. Then re-do State Table as:

Present State			Next State $w = 0$			Next State $w = 1$			Output
y_2	y_1	y_0	Y_2	Y_1	Y_0	Y_2	Y_1	Y_0	z
0	0	0							
0	0	1							
0	1	0							
0	1	1							
1	0	0							

II.A.3.b Choose FF Type

Can use any type of FF in any combination.

— Often the most straightforward choice is D FFs because:

— However, we can choose D, T, or JK in any combination.

— If anything besides D FFs, we need *another* table:

II.A.3.c K-Maps

We will need K-maps to simplify the combinational logic blocks in the FSM block diagram.

— Draw a separate K-map for *each* output z_i (this is the combinational logic *after* the FFs) and for *each* FF input (this is the combinational logic *before* the FFs).

— Current example with D FFs below.

— Note that there will often be *don't – cares* in the K-maps!

— Note that we will often have *multiple – output* circuits, so we need to consider sharing of circuitry!

More K-maps:

II.A.3.d Circuit

Handout.

II.A.4 Variations: Other FF Types

Always start with State-Assigned Table. Repeat part of it here, focusing on Y_0.

Present State			Next State $w = 0$			Next State $w = 1$			Output
y_2	y_1	y_0	Y_2	Y_1	Y_0	Y_2	Y_1	Y_0	z
0	0	0							
0	0	1							
0	1	0							
0	1	1							
1	0	0							

Now we need to construct an *Excitation Table* to specify what FF inputs are needed to force the proper next state.

II.A.4.a T FFs

To form Excitation Table for T FFs, *compare* present state variable (y_0) with next state variable (Y_0):

Excitation Table (T FFs):

Present State			FF Inputs						Output
			$w = 0$			$w = 1$			
y_2	y_1	y_0	T_2	T_1	T_0	T_2	T_1	T_0	z
0	0	0							
0	0	1							
0	1	0							
0	1	1							
1	0	0							

Now do K-maps (only T_0 here):

II.A.4.b JK FFs

For JK FFs, we have *two* inputs per FF (state variable). *Again* look at present state (y_0) and next state (Y_0) and use this table:

Practice this in HW.

II.B Stuff Sequence Detector (Mealy FSM)

II.B.1 Description

Again, we will design a simplified version to detect 011 (instead of 011111).

Input: *w*:

Output: *z*: $z = 1$ *during* clock time of second 1 in 011 sequence. This is needed so that it can trigger an action (e.g., stuff a 0 bit at transmit side) during the *next* bit time. Result: *Mealy* FSM.

Note:

II.B.2 State Specification

This time, need only 4 states:

S_0:

S_1:

S_2:

S_3:

II.B.2.a State Diagram

In *Mealy* state diagrams, transitions (arrows) are labeled with:

Convention: input_values/output_values

II.B.2.b State Table

Present State	Next State $w=0$	Next State $w=1$	Output z $w=0$	Output z $w=1$
S_0				
S_1				
S_2				
S_3				

II.B.3 Implementation

4 states implies:

II.B.3.a State Assignment

Again, use y_i for present state and Y_i for next state.

State-Assigned Table

Present State		Next State $w=0$		Next State $w=1$		Output $w=0$	Output $w=1$
y_1	y_0	Y_1	Y_0	Y_1	Y_0	z	z
0	0						
0	1						
1	0						
1	1						

II.B.3.b Choice of FF

Again choose D FF, so that $D_i = Y_i$.

II.B.3.c K-Maps

We have 3 functions (D_1, D_0, z).

With Mealy FSM, all have the same inputs (w, y_1, y_0).

II.B.3.d Circuit

II.B.4 Variation: JK FFs

Recall: we can use *any* type of FFs in *any* combination. Try JK FFs here. Need to form Excitation Table using:

Excitation Table:

Present State		Next State				Output	
		$w = 0$		$w = 1$		$w = 0$	$w = 1$
y_1	y_0	$J_1\ K_1$	$J_0\ K_0$	$J_1\ K_1$	$J_0\ K_0$	z	z
0	0						
0	1						
1	0						
1	1						

K-Maps:

II.C Highway Move-Over Sign Revisited

II.C.1 Description

3 "panels"

Now allow 2 modes:

A. All 3 arrows (panels) blink on and off together

B. First arrow (panel) 2 only, then 2 and 1, then all 3, then off, then arrow 2 only...

II.C.2 State Specification

Several possibilities: states *can* correspond to combination of arrows on. Since outputs will control arrow, outputs can be derived from states, and so this can be a Moore FSM.

S_0:

S_1:

S_2:

S_3:

II.C.2.a State Diagram

II.C.2.b State Table

Skip it this time -- will go directly to State-Assigned Table below.

II.C.3 Implementation

II.C.3.a State, Input, Output Assignments

Since we specified these symbolically, we need to assign binary values.

States: 4 states implies 2 bits, so use S_0: 00 through S_3: 11

Input: Mode A: $w = 0$ Mode B: $w = 1$

Outputs: Would want a binary control signal for each arrow with 1=on, 0=off

z_2: arrow 2 z_1: arrow 1 z_0: arrow 0

State-Assigned Table:

Present State		Next State $w = 0$ (A)		Next State $w = 1$ (B)		Output		
y_1	y_0	Y_1	Y_0	Y_1	Y_0	z_2	z_1	z_0
0	0							
0	1							
1	0							
1	1							

II.C.3.b Choice of FFs

Do this with T FFs for practice. Excitation Table:

Present State		Next State $w = 0$ (A)		Next State $w = 1$ (B)		Output		
y_1	y_0	T_1	T_0	T_1	T_0	z_2	z_1	z_0
0	0							
0	1							
1	0							
1	1							

II.C.3.c K-Maps

Need *five* K-maps (T_1, T_0, z_2, z_1, z_0).

T_1 and T_0 inputs: z_i inputs:

II.C.3.d Circuit

II.D Modified Highway Move-Over Sign

II.D.1 Description

Now *change* Mode A description:

A. On for 2 clock cycles, off for 2

B. As before.

II.D.2 State Specification (Moore)

Can't use previous state meaning -- must modify.

Split S_0 into S_{01} and S_{02}: first and second clock cycle. Same for S_3.

II.D.2.a State Diagram

6 States imply _____ FFs

Could complete the design from here.

II.D.3 Alternative State Specification

Note that each mode could be considered as a repeating sequence of 4 clock cycles (0 through 3).

So these *could* be our state if:

So, let assigned states be 00 through 11 (4 clock cycles) and draw *Mealy* state-assigned diagram.

Again: Mode A: $w = 0$ Mode B: $w = 1$

Now transition labels will be:

II.D.3.a State-Assigned Diagram

II.D.3.b State-Assigned Table

Present State		Next State				Output					
		$w = 0$ (A)		$w = 1$ (B)		$w = 0$ (A)			$w = 1$ (B)		
y_1	y_0	Y_1	Y_0	Y_1	Y_0	z_2	z_1	z_0	z_2	z_1	z_0
0	0										
0	1										
1	0										
1	1										

II.D.4 Implementation (of Mealy Design)

Use T FFs (will see why shortly).

II.D.4.a Excitation Table for States

Present State		Next State			
		$w = 0$ (A)		$w = 1$ (B)	
y_1	y_0	T_1	T_0	T_1	T_0
0	0				
0	1				
1	0				
1	1				

II.D.4.b Equations for Outputs

You draw K-maps and verify the following equations for the outputs:

And then *note* that we can use *sharing*:

II.E Book Examples

II.E.1 Serial Adder

Section 8.5

II.E.2 Counter as FSM

Section 8.7

III. FSM ANALYSIS

III.A Why Analysis?

Various reasons why we might need to analyze the behavior of an existing circuit.

1. Describe circuit behavior
2. Get different implementation(s)
3. Identify circuit problems (debugging)

III.B Analysis Procedure and Example

Essentially work backwards from circuit to description.

III.B.1 Start with Circuit Diagram

We will organize the circuit diagram in a different way than previous diagrams:

a. "Flip" the FFs so that inputs come in on right and outputs leave from left.
b. Gather all combinational logic together at top.

Diagram on next page:

Circuit diagram:

III.B.2 Identify All Variables

Including states and FF inputs.

Inputs: States:

Outputs: FF Inputs:

III.B.3 Get Logic Expressions for FF Inputs and Outputs

"Read" from the circuit...

$J_1 =$ $J_0 =$

$K_1 =$ $K_0 =$

III.B.4 Fill in State-Assigned Table

Again, we will use another commonly-used form for this:

Input	Present State		FF Inputs				Next State		Output
w	y_1	y_0	J_1	K_1	J_0	K_0	Y_1	Y_0	z
0	0	0							
0	0	1							
0	1	0							
0	1	1							
1	0	0							
1	0	1							
1	1	0							
1	1	1							

III.B.5 Draw State-Assigned Diagram

III.B.6 Try To Figure Out What It Does

Some notes/observations first.

Note 1: Never get *to* State 1 (this would be called a *transient* state). This *may be* OK. Perhaps there are really only 3 states, and this state 01 resulted from don't cares? Or, maybe there is an error.

Note 2: Never get *from* State 00 (this would be called an *absorbing* state). This *might* indicate a problem in the design, *unless* the circuit is supposed to detect the very *first* occurrence of some event.

So, as it is, what does it do??

If we assume that State 01 is not a real state and that State 10 is the Reset State, this circuit: