



Digital Design

Chapter 3: Sequential Logic Design -- Controllers

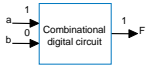
Slides to accompany the textbook *Digital Design*, First Edition,
by Frank Vahid, John Wiley and Sons Publishers, 2007.
<http://www.dfvahid.com>

Copyright © 2007 Frank Vahid


Instructors of courses requiring Vahid's *Digital Design* textbook (published by John Wiley and Sons) have permission to modify and use these slides for customary course-related activities, subject to keeping this copyright notice in place and unmodified. These slides may be posted as unmodified self versions on publicly-accessible course websites. PowerPoint sources (or pdf with animations) may not be posted to publicly-accessible websites, but may be posted for students on internal protected sites or distributed directly to students by other electronic means. Instructors may make minor revisions of the slides available to students for a reasonable photocopying charge, without incurring royalties. Any other use requires explicit permission. Instructors may obtain PowerPoint source or obtain special use permissions from Wiley - see <http://www.wiley.com> for information.

Introduction 3.1

- Sequential circuit
 - Output depends not just on present inputs (as in combinational circuit), but on past sequence of inputs
 - Stores bits, also known as having "state"
 - Simple example: a circuit that counts up in binary
- In this chapter, we will:
 - Design a new building block, a **flip-flop**, that stores one bit
 - Combine that block to build multi-bit storage – a **register**
 - Describe the sequential behavior using a **finite state machine**
 - Convert a finite state machine to a **controller** – a sequential circuit having a register and combinational logic



Combinational digital circuit



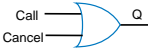
Sequential digital circuit

Must know sequence of past inputs to know output

Note: Slides with animation are denoted with a small red "A" near the animated items

Example Needing Bit Storage 3.2

- Flight attendant call button
 - Press call: light turns on
 - Stays on** after button released
 - Press cancel: light turns off
 - Logic gate circuit to implement this?

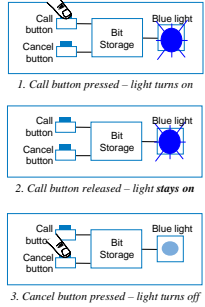


Call
Cancel

Q

Doesn't work. Q=1 when Call=1, but doesn't stay 1 when Call returns to 0

Need some form of "feedback" in the circuit



1. Call button pressed – light turns on

2. Call button released – light stays on

3. Cancel button pressed – light turns off

First attempt at Bit Storage

- We need some sort of feedback
 - Does circuit on the right do what we want?
 - No: Once Q becomes 1 (when S=1), Q stays 1 forever – no value of S can bring Q back to 0

Digital Design
Copyright © 2006
Frank Vahid

Bit Storage Using an SR Latch

- Does the circuit to the right, with cross-coupled NOR gates, do what we want?
 - Yes! How did someone come up with that circuit? Maybe just trial and error, a bit of insight...

Digital Design
Copyright © 2006
Frank Vahid

Example Using SR Latch for Bit Storage

- SR latch can serve as bit storage in previous example of flight-attendant call button
 - Call=1 : sets Q to 1
 - Q stays 1 even after Call=0
 - Cancel=1 : resets Q to 0
- But, there's a problem...

Digital Design
Copyright © 2006
Frank Vahid

Problem with SR Latch

- Problem**
 - If $S=1$ and $R=1$ simultaneously, we don't know what value Q will take

Q may oscillate. Then, because one path will be slightly longer than the other, Q will eventually settle to 1 or 0 – but we don't know which.

Digital Design
Copyright © 2006
Frank Vahid

Problem with SR Latch

- Problem not just one of a user pressing two buttons at same time
- Can also occur even if SR inputs come from a circuit that supposedly never sets $S=1$ and $R=1$ at same time
 - But does, due to different delays of different paths

The longer path from X to R than to S causes $SR=11$ for short time – could be long enough to cause oscillation

Digital Design
Copyright © 2006
Frank Vahid

Solution: Level-Sensitive SR Latch

- Add enable input "C" as shown
 - Only let S and R change when $C=0$
 - Ensure circuit in front of SR never sets $SR=11$, except briefly due to path delays
 - Change C to 1 only after sufficient time for S and R to be stable
 - When C becomes 1, the stable S and R value passes through the two AND gates to the SR latch's S1 R1 inputs.

Though $SR=11$ briefly...

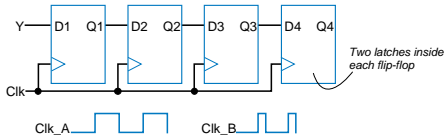
...S1R1 never = 11

Level-sensitive SR latch symbol

Digital Design
Copyright © 2006
Frank Vahid

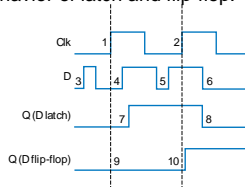
D Flip-Flop

- Solves problem of not knowing through how many latches a signal travels when $C=1$
 - In figure below, signal travels through exactly one flip-flop, for Clk_A or Clk_B
 - Why? Because on rising edge of Clk, all four flip-flops are loaded simultaneously -- then all four no longer pay attention to their input, until the next rising edge. Doesn't matter how long Clk is 1.



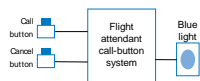
D Latch vs. D Flip-Flop

- Latch is level-sensitive: Stores D when $C=1$
- Flip-flop is edge triggered: Stores D when C changes from 0 to 1
 - Saying "level-sensitive latch," or "edge-triggered flip-flop," is redundant
 - Two types of flip-flops -- rising or falling edge triggered.
- Comparing behavior of latch and flip-flop:



Flight-Attendant Call Button Using D Flip-Flop

- D flip-flop will store bit
- Inputs are Call, Cancel, and present value of D flip-flop, Q
- Truth table shown below



Call	Cancel	Q	D
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

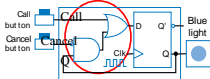
Preserve value: if $Q=0$, make $D=0$; if $Q=1$, make $D=1$

Cancel -- make $D=0$

Call -- make $D=1$

Let's give priority to Call -- make $D=1$

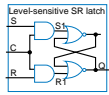
Circuit derived from truth table, using Chapter 2 combinational logic design process



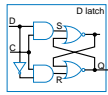
Bit Storage Summary



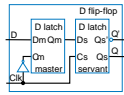
Feature: S=1 sets Q to 1, R=1 resets Q to 0. Problem: SR=11 yield undefined Q.



Feature: S and R only have effect when C=1. We can design outside circuit so SR=11 never happens when C=1. Problem: avoiding SR=11 can be a burden.



Feature: SR can't be 11 if D is stable before and while C=1, and will be 11 for only a brief glitch even if D changes while C=1. Problem: C=1 too long propagates new values through too many latches: too short may not enable a store.

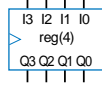
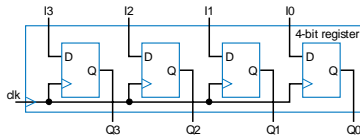


Feature: Only loads D value present at rising clock edge, so values can't propagate to other flip-flops during same clock cycle. Tradeoff: uses more gates internally than D latch, and requires more external gates than SR – but gate count is less of an issue today.

- We considered increasingly better bit storage until we arrived at the robust D flip-flop bit storage

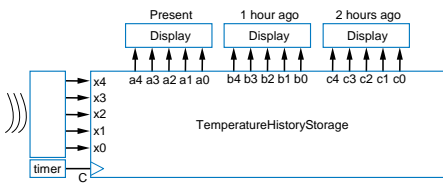
Basic Register

- Typically, we store multi-bit items
 - e.g., storing a 4-bit binary number
- **Register:** multiple flip-flops sharing clock signal
 - From this point, we'll use registers for bit storage
 - No need to think of latches or flip-flops
 - But now you know what's inside a register



Example Using Registers: Temperature Display

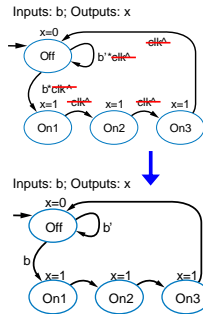
- Temperature history display
 - Sensor outputs temperature as 5-bit binary number
 - Timer pulses C every hour
 - Record temperature on each pulse, display last three recorded values



(In practice, we would actually avoid connecting the timer output C to a clock input, instead only connecting an oscillator output to a clock input.)

FSM Simplification: Rising Clock Edges Implicit

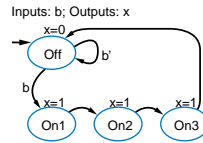
- Showing rising clock on every transition: cluttered
 - Make implicit -- assume every edge has rising clock, even if not shown
 - What if we wanted a transition *without* a rising edge
 - We don't consider such asynchronous FSMs -- less common, and advanced topic
 - Only consider **synchronous** FSMs -- rising edge on every transition



Note: Transition with no associated condition thus transitions to next state on next clock cycle

FSM Definition

- FSM consists of
 - Set of states
 - Ex: {Off, On1, On2, On3}
 - Set of inputs, set of outputs
 - Ex: Inputs: {x}, Outputs: {b}
 - Initial state
 - Ex: "Off"
 - Set of transitions
 - Describes next states
 - Ex: Has 5 transitions
 - Set of actions
 - Sets outputs while in states
 - Ex: $x=0$, $x=1$, $x=1$, and $x=1$

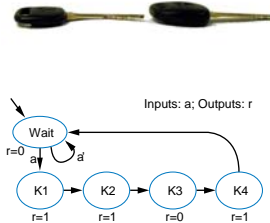


We often draw FSM graphically, known as **state diagram**

Can also use table (state table), or textual languages

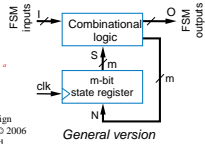
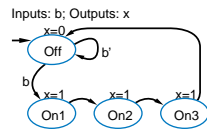
FSM Example: Secure Car Key

- Many new car keys include tiny computer chip
 - When car starts, car's computer (under engine hood) requests identifier from key
 - Key transmits identifier
 - If not, computer shuts off car
- FSM
 - Wait until computer requests ID ($a=1$)
 - Transmit ID (in this case, 1101)



Standard Controller Architecture

- How implement FSM as sequential circuit?
 - Use standard architecture
 - State register -- to store the present state
 - Combinational logic -- to compute outputs, and next state
 - For laser timer FSM
 - 2-bit state register, can represent four states
 - Input b, output x
- Known as **controller**



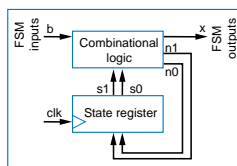
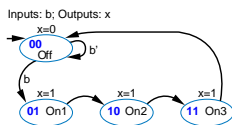
Controller Design

- Five step controller design process

Step	Description
Step 1 <i>Capture the FSM</i>	Create an FSM that describes the desired behavior of the controller.
Step 2 <i>Create the architecture</i>	Create the standard architecture by using a state register of appropriate width, and combinational logic with inputs being the state register bits and the FSM inputs and outputs being the next state bits and the FSM outputs.
Step 3 <i>Encode the states</i>	Assign a unique binary number to each state. Each binary number representing a state is known as an <i>encoding</i> . Any encoding will do as long as each state has a unique encoding.
Step 4 <i>Create the state table</i>	Create a truth table for the combinational logic such that the logic will generate the correct FSM outputs and next state signals. Ordering the inputs with state bits first makes this truth table describe the state behavior, so the table is a state table.
Step 5 <i>Implement the combinational logic</i>	Implement the combinational logic using any method.

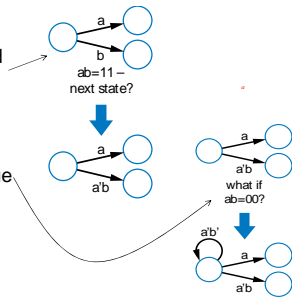
Controller Design: Laser Timer Example

- Step 1: Capture the FSM**
 - Already done
- Step 2: Create architecture**
 - 2-bit state register (for 4 states)
 - Input b, output x
 - Next state signals n1, n0
- Step 3: Encode the states**
 - Any encoding with each state unique will work



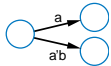
Common Pitfalls Regarding Transition Properties

- Only one condition should be true
 - For all transitions leaving a state
 - Else, which one?
- One condition must be true
 - For all transitions leaving a state
 - Else, where go?



Verifying Correct Transition Properties

- Can verify using Boolean algebra
 - Only one condition true: AND of each condition pair (for transitions leaving a state) should equal 0 → proves pair can never simultaneously be true
 - One condition true: OR of all conditions of transitions leaving a state should equal 1 → proves at least one condition must be true
- Example



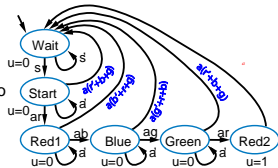
Q: For shown transitions, prove whether:

- * Only one condition true (AND of each pair is always 0)
- * One condition true (OR of all transitions is always 1)

Answer:
 $a \cdot a'b = (a \cdot a') \cdot b = 0 \cdot b = 0$
 OK!
 $a + a'b = a'(1+b) + a'b = a + ab + a'b = a + (a+a')b = a + b$
 Fails! Might not be 1 (i.e., $a=0, b=0$)

Evidence that Pitfall is Common

- Recall code detector FSM
 - We "fixed" a problem with the transition conditions
 - Do the transitions obey the two required transition properties?
 - Consider transitions of state Start, and the "only one true" property



$$\begin{aligned} ar \cdot a' &= a' \cdot a(r+b+g) \\ = (a \cdot a')r &= 0 \cdot r \\ = 0 &= 0 \end{aligned}$$

$$\begin{aligned} ar \cdot a &= a \cdot a(r+b+g) \\ = (a \cdot a)r &= r(r+b+g) = 0 \cdot r(r+b+g) \\ = (a \cdot a) \cdot r(r+b+g) &= a \cdot r(r+b+g) \\ = ar + arb + arg &= 0 + arb + arg \\ = arb + arg &= ar(b+g) \end{aligned}$$

Intuitively, press red and blue buttons at same time: conditions ar, and a(r+b+g) will both be true. Which one should be taken?

Q: How to solve?

A: ar should be arb'g' (likewise for ab, ag, ar)

Fails! Means that two of Start's transitions could be true

Note: As evidence the pitfall is common, we admit the mistake was not intentional. A reviewer of the book caught it.

Simplifying Notations

- FSMs
 - Assume unassigned output implicitly assigned 0
- Sequential circuits
 - Assume unconnected clock inputs connected to same external clock

The diagram shows two states of an FSM. The top state has $a=0, b=1, c=0$ and the bottom state has $a=0, b=0, c=1$. An arrow indicates a transition between them. To the right, a logic circuit diagram shows a clock signal 'clk' connected to multiple flip-flop clock inputs, and an output 'a'.

Digital Design
Copyright © 2006
Frank Vahid

49

More on Flip-Flops and Controllers

3.5

- Other flip-flop types
 - SR flip-flop: like SR latch, but edge triggered
 - JK flip-flop: like SR ($S \rightarrow J, R \rightarrow K$)
 - But when $JK=11$, toggles
 - $1 \rightarrow 0, 0 \rightarrow 1$
 - T flip-flop: JK with inputs tied together
 - Toggles on every rising clock edge
 - Previously utilized to minimize logic outside flip-flop
 - Today, minimizing logic to such extent is not as important
 - D flip-flops are thus by far the most common

Digital Design
Copyright © 2006
Frank Vahid

50

Non-Ideal Flip-Flop Behavior

- Can't change flip-flop input too close to clock edge
 - Setup time: time that D must be stable *before* edge
 - Else, stable value not present at internal latch
 - Hold time: time that D must be held stable *after* edge
 - Else, new value doesn't have time to loop around and stabilize in internal latch

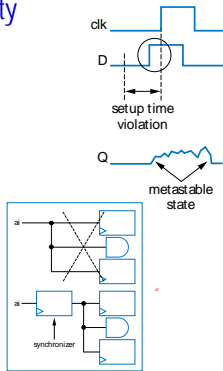
The diagram shows a clock signal 'clk' and a data signal 'D'. A 'setup time' interval is marked before the clock edge, and a 'hold time' interval is marked after. A 'Setup time violation' is shown where the data signal changes too close to the clock edge, leading to 'Leads to oscillation!' in the output signal.

Digital Design
Copyright © 2006
Frank Vahid

51

Metastability

- Violating setup/hold time can lead to bad situation known as **metastable** state
 - Metastable state: Any flip-flop state other than stable 1 or 0
 - Eventually settles to one or other, but we don't know which
 - For internal circuits, we can make sure observe setup time
 - But what if input comes from external (asynchronous) source, e.g., button press?
- Partial solution
 - Insert synchronizer flip-flop for asynchronous input
 - Special flip-flop with very small setup/hold time
 - Doesn't completely prevent metastability



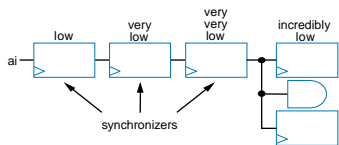
Digital Design
Copyright © 2006
Frank Vahid

52

Metastability

- One flip-flop doesn't completely solve problem
- How about adding more synchronizer flip-flops?
 - Helps, but just decreases probability of metastability
- So how solve completely?
 - Can't! May be unsettling to new designers. But we just can't guarantee a design that won't ever be metastable. We can just minimize the mean time between failure (MTBF) -- a number often given along with a circuit

Probability of flip-flop being metastable is...

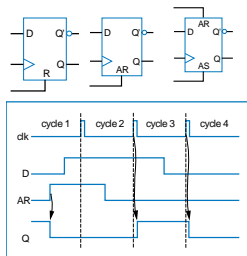


Digital Design
Copyright © 2006
Frank Vahid

53

Flip-Flop Set and Reset Inputs

- Some flip-flops have additional inputs
 - Synchronous reset: clears Q to 0 on next clock edge
 - Synchronous set: sets Q to 1 on next clock edge
 - Asynchronous reset: clear Q to 0 immediately (not dependent on clock edge)
 - Example timing diagram shown
 - Asynchronous set: set Q to 1 immediately

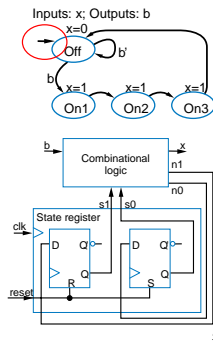


Digital Design
Copyright © 2006
Frank Vahid

54

Initial State of a Controller

- All our FSMs had initial state
 - But our sequential circuit designs did not
 - Can accomplish using flip-flops with reset/set inputs
 - Shown circuit initializes flip-flops to 01
 - Designer must ensure reset input is 1 during power up of circuit
 - By electronic circuit design

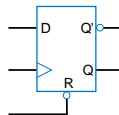


Glitching

- Glitch: Temporary values on outputs that appear soon after input changes, before stable new output values
- Designer must determine whether glitching outputs may pose a problem
 - If so, may consider adding flip-flops to outputs
 - Delays output by one clock cycle, but may be OK

Active Low Inputs

- We've assumed input action occur when input is 1
 - Some inputs are instead active when input is 0 -- "active low"
 - Shown with inversion bubble
 - So to reset the shown flip-flop, set R=0. Else, keep R=1.



Chapter Summary

- Sequential circuits
 - Have state
- Created robust bit-storage device: D flip-flop
 - Put several together to build register, which we used to hold state
- Defined FSM formal model to describe sequential behavior
 - Using solid mathematical models -- Boolean equations for combinational circuit, and FSMs for sequential circuits -- is very important.
- Defined 5-step process to convert FSM to sequential circuit
 - Controller
- So now we know how to build the class of sequential circuits known as controllers



Digital Design
Copyright © 2006
Frank Vahid

58
