University of California at Berkeley
College of Engineering
Department of Electrical Engineering and Computer Science

EECS 150 R. H. Katz
Fall 2005

Problem Set # 6
(Assigned 12 October, Due 21 October)
SOLUTIONS
(Note: there may be many different solutions to the questions below)

1. Consider the design of a simple digital alarm clock. The clock displays the current time in 24-hour HH:MM notation (HH goes from 00 to 23 and MM from 00 to 59). There are several control inputs for setting the current time and the alarm time. The TIME SET HH button allows you to rapidly advance through the hours 0 through 23. The time advances one hour for each time the button is pressed. The TIME SET MM button does the same thing for the minutes 00 through 59. In addition, the clock has a toggle input that indicates CLOCK in one position and ALARM in the other. This refers to which time is being set by the TIME SET buttons: either the current time (CLOCK) or the alarm time (ALARM). If the ALARM button is set, the clock displays the last set alarm time. Finally, there is another input controlled by a push button: ALARM ON when pushed in and ALARM OFF when popped out. Your job is to define the state diagram for the clock controller. It should support normal clock display (when CLOCK is set), alarm display (when ALARM is set), set current time (CLOCK + TIME SET buttons), set alarm time (ALARM + TIME SET buttons), and of course, the alarm should sound when the current time equals alarm time.

   (a) Identify your inputs, outputs, and name and describe your states. State any assumptions you are making.

   INPUTS:

TIME SET HH	Increment hour for clock/alarm
TIME SET MM	Increment minute for clock/alarm
TOGGLE	Toggles between clock and alarm set

   OUTPUTS:

TMI	Increases clock minute by 1
THI	Increases clock hour by 1
AMI	Increases alarm minute by 1
AHI	Increases alarm hour by 1
DISP	Indicates clock or alarm display

   STATES:

TIME MM	Increase minute counter of time
TIME HH	Increase hour counter of time
TIME IDLE	Display time without increase minute or hour
ALARM MM	Increase minute counter of alarm
ALARM HH	Increase hour counter of alarm
ALARM IDLE	Display alarm without increasing minute or hour

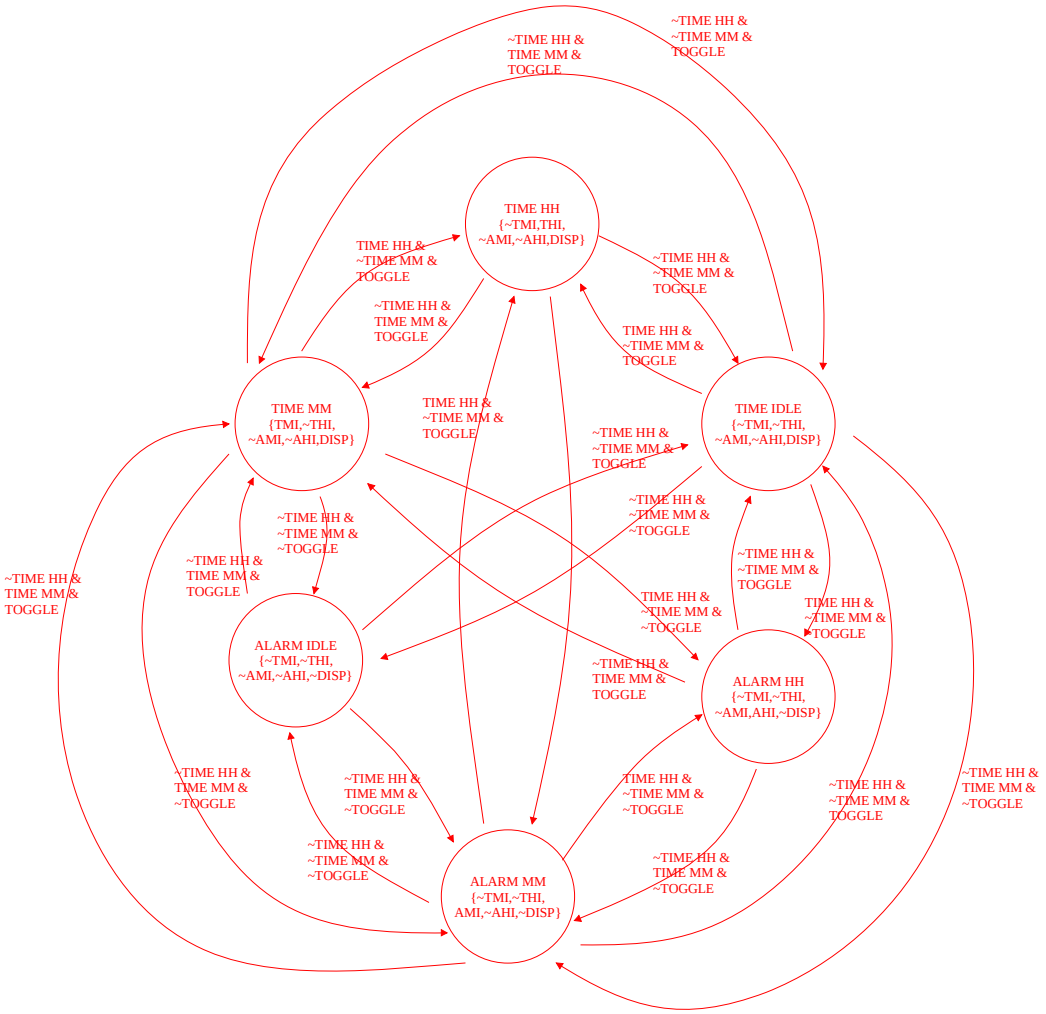
   Note: for the sake of simplicity, ALARM is passed through the controller without being manipulated, and the alarm sounding is controlled outside of the controller. Therefore, the controller does not need to know the specifics about what the current time and alarm time is. The

clock controller only needs to send signals to the datapath to manipulate the clock and alarm times.

ASSUMPTIONS: You can move from one state to any of the other states. For example, if you are pressing the Minute Advance button and you toggle the time set off, the alarm will start incrementing the alarm minute.

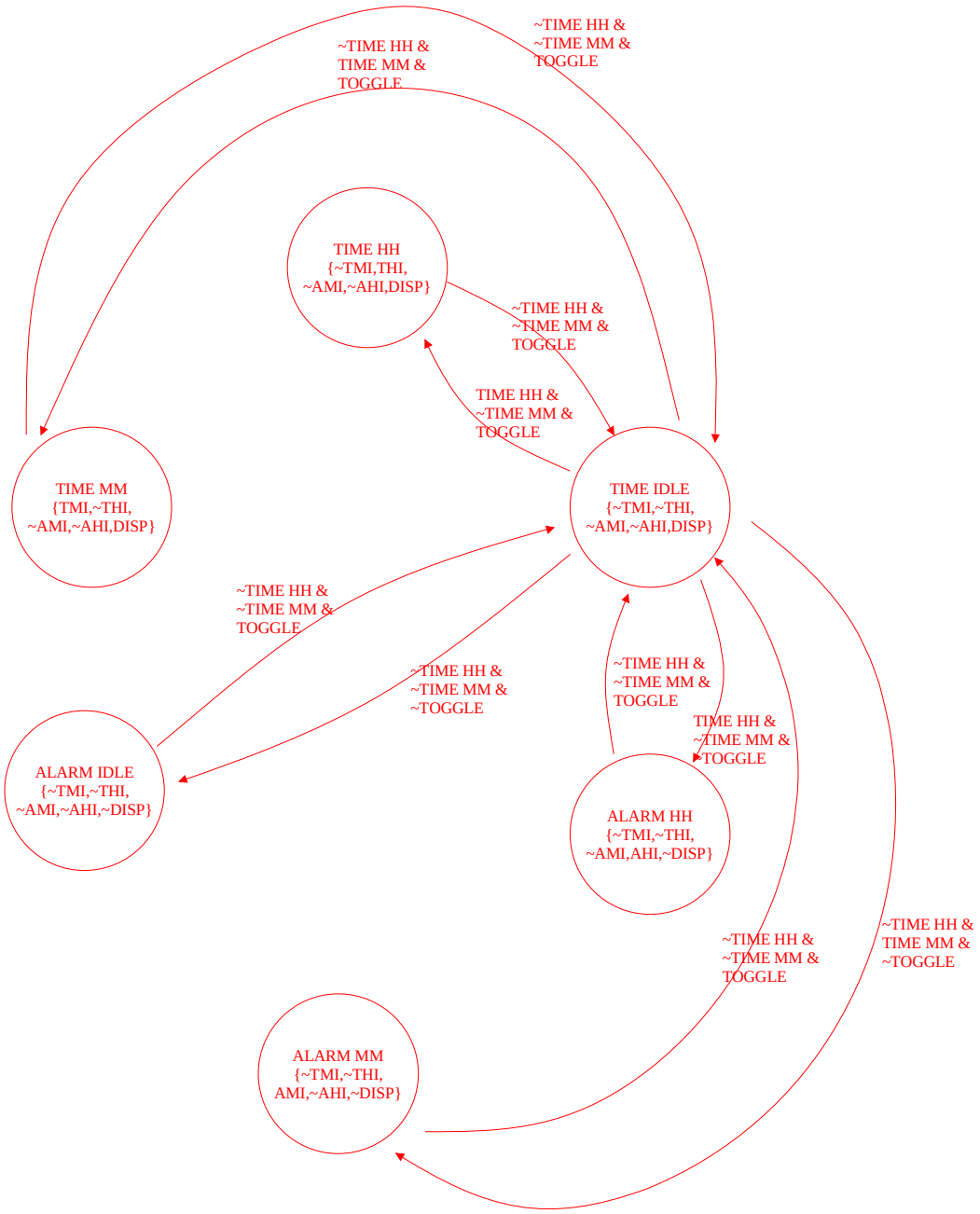
(b) Draw a symbolic state diagram for your design, labeling all state transitions.

SOLUTION 1:
ASSUMPTIONS: You can move from one state to any of the other states. For example, if you are pressing the Minute Advance button and you toggle the time set off, the alarm will start incrementing the alarm minute.



Note: The else arcs are omitted for clarity.

SOLUTION 2 (This is probably more realistic):
ASSUMPTIONS: State machine is either in “alarm” mode or “clock” mode, and it can only go from “alarm” mode to “time” mode from the ALARM_IDLE and TIME_IDLE states. That is, the state machine waits till the user finishes pressing the TIME_MM or TIME_HH buttons before changing states.



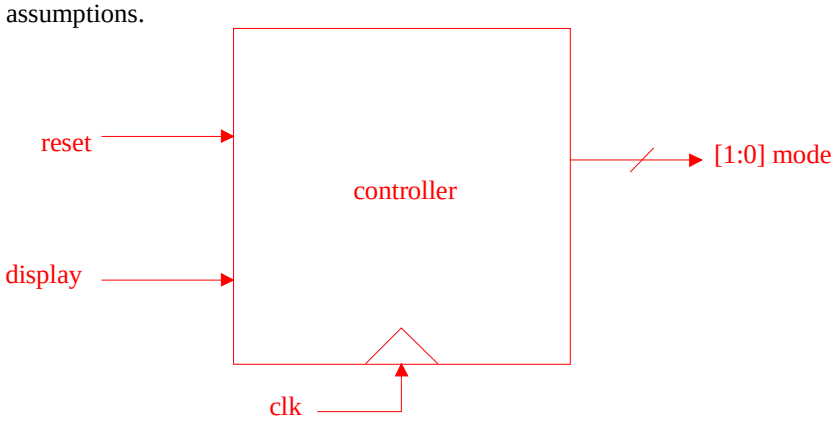
(c) Write sketch Verilog for describing this state machine, assuming a MOORE Machine implementation.

```
Input clk, rst, timehh, timemm, toggle;
Output TMI, THI, AMI, AHI, DISP;
Reg [1:0] PS, NS;

Always @ (*) begin
Case (PS)
      TIME_IDLE:  {TMI,THI,AMI,AHI,DISP}  <= {0,0,0,0,0};
      TIME_HH:    {TMI,THI,AMI,AHI,DISP}  <= {0,1,0,0,0};
      TIME_MM:    {TMI,THI,AMI,AHI,DISP}  <= {0,0,1,0,0};
      ALARM_IDLE: {TMI,THI,AMI,AHI,DISP}  <= {0,0,0,0,1};
      ALARM_HH:   {TMI,THI,AMI,AHI,DISP}  <= {0,1,0,0,0};
```

```
      ALARM_MM:   {TMI,THI,AMI,AHI,DISP}  <= {1,0,0,0,0};
end
Always @ (*) begin
Case (PS)
      TIME_IDLE:
            if(~timehh & ~timemm & toggle) NS <= TIME_IDLE;
            else if (timehh & ~timemm & toggle) NS <= TIME_HH;
            else if (~timehh & timemm & toggle) NS <= TIME_MM;
            else if (~timehh & ~timemm & ~toggle) NS <= ALARM_IDLE;
            else if (timehh & ~timemm & ~toggle) NS <= ALARM_HH;
            else if (~timehh & timemm & ~toggle) NS <= ALARM_MM;
            else NS <= PS;
      TIME_HH:
            if(~timehh & ~timemm & toggle) NS <= TIME_IDLE;
            else if (timehh & ~timemm & toggle) NS <= TIME_HH;
            else if (~timehh & timemm & toggle) NS <= TIME_MM;
            else if (~timehh & ~timemm & ~toggle) NS <= ALARM_IDLE;
            else if (timehh & ~timemm & ~toggle) NS <= ALARM_HH;
            else if (~timehh & timemm & ~toggle) NS <= ALARM_MM;
            else NS <= PS;
      TIME_MM:
            if(~timehh & ~timemm & toggle) NS <= TIME_IDLE;
            else if (timehh & ~timemm & toggle) NS <= TIME_HH;
            else if (~timehh & timemm & toggle) NS <= TIME_MM;
            else if (~timehh & ~timemm & ~toggle) NS <= ALARM_IDLE;
            else if (timehh & ~timemm & ~toggle) NS <= ALARM_HH;
            else if (~timehh & timemm & ~toggle) NS <= ALARM_MM;
            else NS <= PS;
      ALARM_IDLE:
            if(~timehh & ~timemm & toggle) NS <= TIME_IDLE;
            else if (timehh & ~timemm & toggle) NS <= TIME_HH;
            else if (~timehh & timemm & toggle) NS <= TIME_MM;
            else if (~timehh & ~timemm & ~toggle) NS <= ALARM_IDLE;
            else if (timehh & ~timemm & ~toggle) NS <= ALARM_HH;
            else if (~timehh & timemm & ~toggle) NS <= ALARM_MM;
            else NS <= PS;
      ALARM_HH:
            if(~timehh & ~timemm & toggle) NS <= TIME_IDLE;
            else if (timehh & ~timemm & toggle) NS <= TIME_HH;
            else if (~timehh & timemm & toggle) NS <= TIME_MM;
            else if (~timehh & ~timemm & ~toggle) NS <= ALARM_IDLE;
            else if (timehh & ~timemm & ~toggle) NS <= ALARM_HH;
            else if (~timehh & timemm & ~toggle) NS <= ALARM_MM;
            else NS <= PS;
      ALARM_MM:
            if(~timehh & ~timemm & toggle) NS <= TIME_IDLE;
            else if (timehh & ~timemm & toggle) NS <= TIME_HH;
            else if (~timehh & timemm & toggle) NS <= TIME_MM;
            else if (~timehh & ~timemm & ~toggle) NS <= ALARM_IDLE;
            else if (timehh & ~timemm & ~toggle) NS <= ALARM_HH;
            else if (~timehh & timemm & ~toggle) NS <= ALARM_MM;
            else NS <= PS;
end

Always @ (posedge clk)
NS <= PS
```

2. Design an in-car "trip computer" to the following specification. Under driver control, the trip computer displays miles traveled since last reset, average speed (miles traveled/elapsed time since last reset), and miles per gallon (miles traveled/gallons consumed since last reset). The inputs to the state machine are: (1) signal to increment miles traveled (once per mile traveled), (2) increment time (once per second), and (3) the integer number of gallons of gas consumed since the last reset. The user interface has two buttons: a *reset* and a *display* push button. By default, miles traveled since last reset is displayed. If *display* is pressed, the average speed in *miles per hour* is shown. If pressed again, the *miles per gallon* is shown. Pressed one more time, the display returns to miles traveled since last reset, and the display sequence can be repeated. You may assume that *display* is a synchronized, debounced signal that is asserted for exactly one clock period when pressed.

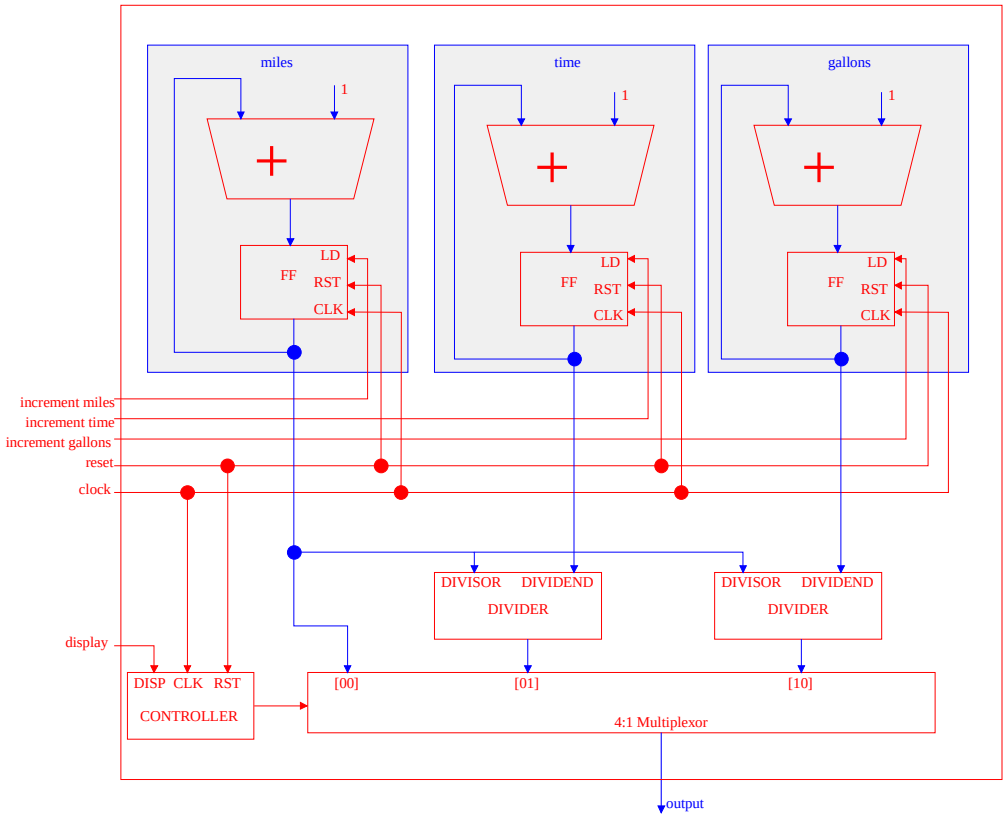
(a) Draw a simple block diagram that shows the controller's inputs and outputs. Document your assumptions.



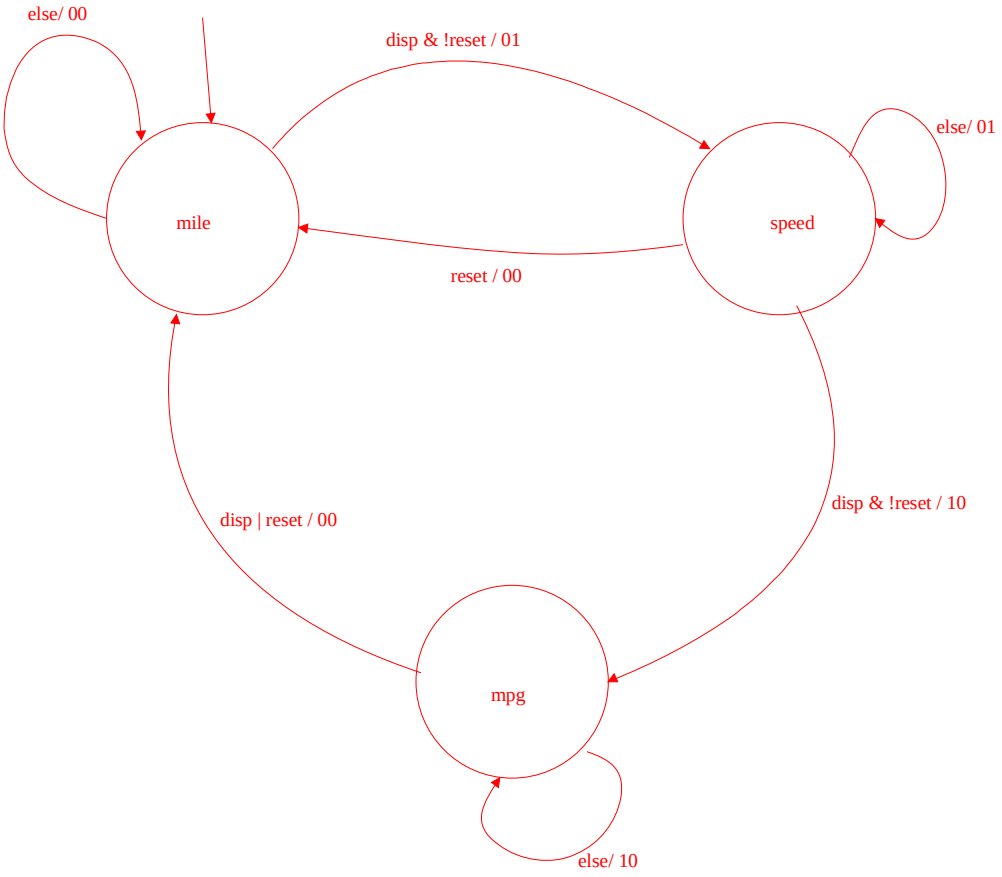
Assumptions:
- increment time, increment mile, and increment gallons are passed directly to the arithmetic operators (There is no need for these signals to be manipulated in the controller)
- mode is a 2-bit wire that drives a 4:1 mux

Mux value	Mode
00	Miles since last reset
01	Miles per hour
10	Miles per gallon
11	Undefined

(b) Draw a simple datapath block diagram for the trip computer, including any multiplexers, counters, registers, arithmetic functional units (e.g., adders, subtractors, multipliers, dividers), and display decoders and drivers you may need for its implementation. Don't worry about how wide these functional units are – simply assume that they are designed to be wide enough for this application. Show how they are interconnected and indicate the control signals that you need to control this datapath.



(c) Draw a complete MEALY MACHINE State Diagram for your controller.



(d) Write sketch Verilog for describing the state machine.

```
Input clk, rst, disp;
Output [1:0] mode;
Reg [1:0] PS, NS;
Always @ (*) begin
Case (PS)
      2'b00:
            if(disp & ~reset) begin
                  mode <= 2'b01;
                  NS <= 2'b01;
            End
            Else begin
                  Mode <= 2'b00;
                  NS <= 2'b00;
            end
      2'b01:
            if(disp & ~reset) begin
                  mode <= 2'b10;
                  NS <= 2'b10;
            End
            Else if(reset) begin
                  mode <= 2'b00;
                  NS <= 2'b00;
            End
            Else begin
                  Mode <= 2'b01;
                  NS <= 2'b01;
            end
      2'b10:
            if(disp | reset) begin
                  mode <= 2'b00;
                  NS <= 2'b00;
            End
            Else begin
                  Mode <= 2'b10;
                  NS <= 2'b10;
            end


Always @ (posedge clk)
NS <= PS
```